Journal of Automated Reasoning (2023) 67:1
https://doi.org/10.1007/s10817-022-09648-w

A Formalization and Proof Checker for Isabelle's Metalogic

Simon Roßkopf[1] · **Tobias Nipkow**[1]

Received: 19 October 2021 / Accepted: 20 July 2022 / Published online: 12 December 2022

Abstract
Isabelle is a generic theorem prover with a fragment of higher-order logic as a metalogic for defining object logics. Isabelle also provides proof terms. We formalize this metalogic and the language of proof terms in Isabelle/HOL, define an executable (but inefficient) proof term checker and prove its correctness w.r.t. the metalogic. We integrate the proof checker with Isabelle and run it on a range of logics and theories to check the correctness of all the proofs in those theories.

Keywords Theorem proving · Higher-order logic · Isabelle · Proofchecker · Metalogic

1 Introduction

One of the selling points of proof assistants is their trustworthiness. Yet in practice, soundness problems do come up in most proof assistants. Harrison [15] distinguishes errors in the logic and errors in the implementation (and cites examples). Our work contributes to the solution of both problems for the proof assistant Isabelle [35]. Isabelle is a generic theorem prover: it implements $\mathcal{M}$, a fragment of intuitionistic higher-order logic, as a metalogic for defining object logics. Its most developed object logic is HOL, and the resulting proof assistant is called Isabelle/HOL [27, 28]. The latter is the basis for our formalizations.

Our first contribution is the first complete formalization of Isabelle's metalogic. Thus our work applies to all Isabelle object logics, e.g., not only HOL but also ZF. Of course Paulson [36] describes $\mathcal{M}$ precisely, but only on paper. More importantly, his description does not yet cover polymorphism and type classes, which were introduced later [29]. The published account of Isabelle's proof terms [7] is also silent about type classes, yet type classes are a significant complication. We do, however, not formalize the theory extension mechanisms (e.g., for constant definitions) on top of the logic.

Our second contribution is a verified (against $\mathcal{M}$) and executable checker for Isabelle's proof terms. We have integrated the proof checker with Isabelle. Thus, we can guarantee that

✉ Simon Roßkopf
rosskops@in.tum.de
https://www21.in.tum.de/~rosskops/

Tobias Nipkow
https://www21.in.tum.de/~nipkow/

[1] Department of Informatics, Technical University of Munich, Munich, Germany

every theorem whose proof our proof checker accepts is provable in our definition of $\mathcal{M}$. So far we are able to check the correctness of moderately sized theories across the full range of logics implemented in Isabelle.

Although Isabelle follows the LCF-architecture (theorems that can only be manufactured by inference rules) it is based on an infrastructure optimized for performance. In particular, this includes multithreading, which is used in the kernel and has once led to a soundness issue.[1] Therefore we opt for the "certificate checking" approach (via proof terms) instead of verifying the implementation.

This is the first work that deals directly with what is implemented in Isabelle as opposed to a study of the metalogic that Isabelle is meant to implement. Instead of reading the implementation, you can now read and build on the more abstract formalization in this paper. The correspondence of the two can be established for each proof by running the proof checker.

Our formalization reflects the ML implementation of Isabelle's terms and types and some other data structures. Thus, a few implementation choices shine through, e.g., De Bruijn indices. This is necessary because we want to integrate our proof checker as directly as possible with Isabelle, with as little unverified glue code as possible, for example, no translation between De Bruijn indices and named variables. We refer to this as our *intentional implementation bias*. In principle, however, one could extend our formalization with different representations (e.g., named terms) and prove suitable isomorphisms.

Our work is purely proof theoretic; semantics is out of scope.

This paper is an extended version of a conference paper [30] presented at CADE 28. In addition to the material covered in the conference paper, it includes

- A section describing some useful derived rules in our inference system (Sect. 7)
- A more detailed description of the executable proofchecker and its verification (Sect. 8)
- An updated formalization, including an updated, more natural formalization of the order-sorted signatures (Sect. 4), generic variable types, explicitly finite data structures, and updated proof terms.

1.1 Related Work

Harrison [15] was the first to verify some of HOL's metatheory and an implementation of a HOL kernel in HOL itself. Kumar et al. [20] formalized HOL including definition principles, proved its soundness and synthesized a verified kernel of a HOL prover down to the machine language level. Abrahamsson [1] verified a proof checker for the OpenTheory [17] proof exchange format for HOL.

Wenzel [43] showed how to interpret type classes as predicates on types. We follow his approach of reflecting type classes in the logic but cannot remove them completely because of our intentional implementation bias (see above). Kunčar and Popescu [21–24] focus on the subtleties of definition principles for HOL with overloading and prove that under certain conditions, type and constant definitions preserve consistency. Åman Pohjola et al. [3] formalize some of this work by Kunčar and Popescu [21, 24].

Adams [2] presents HOL Zero, a basic theorem prover for HOL that addresses the problem of how to ensure that parser and pretty-printer do not misrepresent formulas.

Let us now move away from Isabelle and HOL. Barras verified fragments of Coq in Coq [4, 5]. Sozeau et al. [41] present the first implementation of a type checker for the kernel of Coq that is proved correct in Coq with respect to a formal specification. Carneiro [8] has

[1] https://mailmanbroy.in.tum.de/pipermail/isabelle-dev/2016-December/007251.html.

implemented a highly performant proof checker for a multi-sorted first-order logic and is in the process of verifying it in its own logic. Davis developed the bootstrapping theorem prover Milawa [9] and, together with Myreen, showed its soundness down to machine code [10].

We formalize a logic with bound variables, and there is a large body of related work that deals with this issue (e.g., [11, 18, 42]) and a range of logics and systems with special support for handling bound variables (e.g., [38–40]). We found that De Bruijn indices worked reasonably well for us.

2 Preliminaries

Isabelle types are built from type variables, e.g., $'a$ and (postfix) type constructors, e.g., $'a\ list$; the function type arrow is $\Rightarrow$. Isabelle also has a type class system explained later. The notation $t{::}\tau$ means that term t has type τ. Isabelle/HOL provides types $'a\ set$ and $'a\ list$ of sets and lists of elements of type $'a$. They come with the following vocabulary: function *set* (conversion from lists to sets), (#) (list constructor), (@) (append), $|xs|$ (length of list xs), $xs\ !\ i$ (the ith element of xs starting at 0), *list-all2* $p\ [x_1, \ldots, x_m]\ [y_1, \ldots, y_n] = (m = n \wedge p\ x_1\ y_1 \wedge \ldots \wedge p\ x_n\ y_n)$, ($\gg=$) (monadic bind) and other self-explanatory notation.

There is also the predefined data type

datatype $'a\ option = None \mid Some\ 'a$

The type $\tau_1 \rightharpoonup \tau_2$ abbreviates $\tau_1 \Rightarrow \tau_2\ option$, i.e., partial functions, which we call *maps*. Maps have a domain and a range:

$$dom\ m = \{a \mid m\ a \neq None\} \qquad ran\ m = \{b \mid \exists a.\ m\ a = Some\ b\}$$

It must be noted that in our formalization, we are not using sets/maps directly, but subtypes for finite sets/maps. This simplifies some proofs and code generation; however, there is less material about them readily available. Luckily, we can easily make use of material for general sets/maps using Isabelle's Lifting and Transfer packages[16].

Logical equivalence is written $=$ instead of $\longleftrightarrow$.

3 Types and Terms

A *name* is simply a string. Variables have (Isabelle/HOL level) type $'v$; their inner structure is immaterial for the presentation of the logic. We only require $'v$ to be infinite, to always guarantee a supply of fresh variables. We encode this using a type class for infinite types.

The logic has three layers: terms are classified by types as usual, but in addition, types are classified by *sorts*. A *sort* is simply a set of classes and classes are just strings. We discuss sorts in detail later.

Types (typically denoted by $T, U, \ldots$) are defined like this:

datatype $'v\ typ = Ty\ name\ ('v\ typ\ list) \mid Tv\ 'v\ sort$

where $Ty\ \kappa\ [T_1,\ldots,T_n]$ represents the Isabelle type $(T_1,\ldots,T_n)\ \kappa$ and $Tv\ a\ S$ represents a type variable a of sort S—sorts are directly attached to type variables and contribute to their identity. The notation $T \to U$ is short for Ty "*fun*" $[T, U]$, where "*fun*" is the name of the function type constructor.

Isabelle's terms are simply typed lambda terms in De Bruijn notation:

datatype $'v\ term = Ct\ name\ ('v\ typ) \mid Fv\ 'v\ ('v\ typ) \mid Bv\ nat$
$\mid Abs\ ('v\ typ)\ ('v\ term) \mid (\cdot)\ ('v\ term)\ ('v\ term)$

A term (typically $r, s, t, u \ldots$) can be a typed constant *Ct c T* or free variable *Fv v T*, a bound variable *Bv n* (a De Bruijn index), a typed abstraction *Abs T t* or an application $t \cdot u$. We call an occurrence of a bound variable *Bv i* in some term t *loose* if the occurrence is not enclosed in at least $i + 1$ abstractions.

The term-has-type proposition has the syntax $Ts \vdash_\tau t : T$ where Ts is a list of types, the context for the type of the bound variables.

$$\_ \vdash_\tau Ct\ \_\ T : T \qquad \_ \vdash_\tau Fv\ \_\ T : T \qquad \frac{i < |Ts|}{Ts \vdash_\tau Bv\ i : Ts\ !\ i}$$

$$\frac{T\ \#\ Ts \vdash_\tau t : T'}{Ts \vdash_\tau Abs\ T\ t : T \to T'}$$

$$\frac{Ts \vdash_\tau u : U \qquad Ts \vdash_\tau t : U \to T}{Ts \vdash_\tau t \cdot u : T}$$

We define $\vdash_\tau t : T = [] \vdash_\tau t : T$.

Function *fv* $:: 'v\ term \Rightarrow ('v \times 'v\ typ)\ set$ collects the free variables in a term. Because bound variables are indices, *fv t* is simply the set of all (v, T) such that *Fv v T* occurs in t. The type is an integral part of a variable.

A *type substitution* is a function ϱ of type $'v \Rightarrow sort \Rightarrow 'v\ typ$. It assigns a type to each type variable and sort pair. We write ϱ \$\$ T or ϱ \$\$ t for the overloaded function which applies a type substitution to all type variables (and their sort) occurring in a type or term. The *type instance* relation is defined like this:

$$T_1 \lesssim T_2 = (\exists \varrho.\ \varrho\ \$\$\ T_2 = T_1)$$

We also need to β-contract a term *Abs T t* $\cdot\ u$ to something like "t with *Bv* 0 replaced by u." We define a function *subst-bv* such that *subst-bv u t* is that β-contractum. The definition of *subst-bv* is shown in the Appendix and can also be found in the literature (e.g., [33]).

In order to abstract over a free (term) variable, there is a function *bind-fv* $(v, T)\ t$ that (roughly speaking) replaces all occurrences of *Fv v T* in t by *Bv* 0. Again, see the Appendix for the definition. This produces (if *Fv v T* occurs in t) a term with a loose *Bv* 0. Function *Abs-fv* binds it with an abstraction:

$$Abs\text{-}fv\ v\ T\ t = Abs\ T\ (bind\text{-}fv\ (v, T)\ t)$$

While this section described the syntax of types and terms, they are not necessarily wellformed and should be considered pretypes/preterms. The wellformedness checks are described later.

4 Classes and Sorts

Isabelle has a built-in system of type classes [32] as in Haskell 98 except that class constraints are directly attached to variable names: our $Tv\ a\ [C, D, \ldots]$ corresponds to Haskell's `(Ca, Da, ...) => ... a ...`. A *sort* is Isabelle's terminology for a set of (class) names, e.g., $\{C, D, \ldots\}$, which represent a conjunction of class constraints. In our work, variables S, S' etc. stand for sorts.

Apart from the usual application in object logics, type classes also serve an important metalogical purpose: they allow us to restrict, for example, quantification in object logics to object-level types and rule out meta-level propositions.

Isabelle's type class system was first presented in a programming language context [31, 34]. We give the first machine-checked formalization. The central data structure is a so-called *order-sorted signature*. Intuitively, it is composed of a set of classes, a partial subclass ordering on them and a set of *type constructor signatures*. A type constructor signature $\kappa :: (S_1, \ldots, S_k)\ c$ for a type constructor κ states that applying κ to types $T_1, \ldots, T_k$ such that T_i has sort S_i (defined below) produces a type of class c. Formally:

type_synonym *osig* =
$(name\ set \times (name \times name)\ set \times (name \times sort\ list \times class)\ set)$

The projection functions are called *classes*, *subclass* and *tcsigs*.

The subclass ordering *sub* can be extended to a subsort ordering as follows:

$$S_1 \leq_{sub} S_2 = (\forall c_2 \in S_2.\ \exists c_1 \in S_1.\ c_1 \leq_{sub} c_2)$$

The smaller sort needs to subsume all the classes in the larger sort. In particular $\{c_1\} \leq_{sub} \{c_2\}$ iff $(c_1, c_2) \in sub$.

Now we can define a predicate *has-sort* that checks whether, in the context of some order-sorted signature (cl,sub,tcs), a type fulfills a given sort constraint:

$$\frac{S \leq_{sub} S'}{\textit{has-sort}\ (cl, sub, tcs)\ (\textit{Tv}\ a\ S)\ S'}$$

$$\frac{\forall c \in S.\ \exists Ss.\ (\kappa, Ss, c) \in tcs \wedge \textit{list-all2}\ (\textit{has-sort}\ (cl, sub, tcs))\ Ts\ Ss}{\textit{has-sort}\ (cl, sub, tcs)\ (\textit{Ty}\ \kappa\ Ts)\ S}$$

The rule for type variables uses the subsort relation and is obvious. A type $(T_1, \ldots, T_n)\ \kappa$ has sort $\{c_1, \ldots\}$ if for every c_i there is a signature $\kappa :: (S_1, \ldots, S_n)\ c_i$ and *has-sort*$(cl, sub,tcs)\ T_j\ S_j$ for $j = 1, \ldots, n$.

We *normalize* a sort by removing "superfluous" class constraints, i.e., retaining only those classes that are not subsumed by other classes. This gives us unique representatives for sorts which we call *normalized*:

normalize-sort $sub\ S = \{c \in S \mid \neg\ (\exists c' \in S.\ c' \neq c \wedge (c', c) \in sub)\}$
normalized-sort $sub\ S = (\textit{normalize-sort}\ sub\ S = S)$

We work with normalized sorts because it simplifies the derivation of efficient executable code later on.

Now we can define wellformedness of an $osig$:

wf-osig $(cl, sub, tcs) = (\textit{wf-subclass}\ cl\ sub \wedge \textit{wf-tcsigs}\ cl\ sub\ tcs)$

A subclass relation is wellformed if it is a partial order where reflexivity is restricted to the set of classes cl.Wellformedness of type constructor signatures (*wf-tcsigs*) is more complex. The conditions are the following:

- The following property requires a) that for any $\kappa :: (...)\ c_1$ there must be a $\kappa :: (...)\ c_2$ for every superclass c_2 of c_1 and b) *coregularity* which guarantees the existence of principal types [14, 31]:

$$\forall (\kappa, Ss_1, c_1) \in tcs.\ \forall c_2.\ (c_1, c_2) \in sub \longrightarrow (\exists Ss_2.\ (\kappa, Ss_2, c_2) \in tcs \wedge \textit{list-all2}\ (\textit{sort-leq}\ sub)\ Ss_1\ Ss_2)$$

- A type constructor must always take the same number of argument types:

$$\forall \kappa\ Ss_1\ c_1\ Ss_2\ c_2.\ (\kappa, Ss_1, c_1) \in tcs \wedge (\kappa, Ss_2, c_2) \in tcs \longrightarrow |Ss_1| = |Ss_2|$$

- Sorts must be normalized and must exist in cl:
  $\forall(\kappa, Ss, c){\in}tcs.\ \forall S{\in}Ss.$ *wf-sort* $cl\ sub\ S$
  where *wf-sort* $cl\ sub\ S = (\textit{normalized-sort}\ sub\ S \wedge S \subseteq cl)$
- The argument sorts uniquely determine the class of the constructed type:
  $\forall(\kappa, Ss_1, c){\in}tcs.\ \forall Ss_2.\ (\kappa, Ss_2, c) \in tcs \longrightarrow Ss_2 = Ss_1$

These conditions are used in a number of places to show that the type system is well behaved. For example, *has-sort* is upward closed:
wf-osig $(cl, sub, tcs) \wedge$ *has-sort* $(cl, sub, tcs)\ T\ S \wedge S \leq_{sub} S'$
$\longrightarrow$ *has-sort* $(cl, sub, tcs)\ T\ S'$

5 Signatures

A *signature* consists of a map from constant names to their (most general) types, a map from type constructor names to their arities, and an order-sorted signature:

type_synonym $'vsignature = (name \rightharpoonup 'v\ typ) \times (name \rightharpoonup nat) \times osig$

The three projection functions are called *const-type*, *type-arity* and *osig*. We now define a number of wellformedness checks w.r.t. a signature Σ. We start with wellformedness of types, which essentially requires that all type constructors have correct arity and all type variables have wellformed sort constraints:

$$\frac{\textit{type-arity}\ \Sigma\ \kappa = \textit{Some}\ |Ts| \qquad \forall T{\in}Ts.\ \textit{wf-type}\ \Sigma\ T}{\textit{wf-type}\ \Sigma\ (\textit{Ty}\ \kappa\ Ts)}$$

$$\frac{\textit{wf-sort}\ (\textit{classes}\ (\textit{osig}\ \Sigma))\ (\textit{subclass}\ (\textit{osig}\ \Sigma))\ S}{\textit{wf-type}\ \Sigma\ (\textit{Tv}\ a\ S)}$$

Wellformedness of a term essentially just says that all types in the term are wellformed and that the type T' of a constant in the term must be an instance of the type T of that constant in the signature: $T' \lesssim T$.

$$\frac{\dfrac{\textit{wf-type}\ \Sigma\ T}{\textit{wf-term}\ \Sigma\ (\textit{Fv}\ v\ T)} \qquad \textit{wf-term}\ \Sigma\ (\textit{Bv}\ n)}{\textit{const-type}\ \Sigma\ s = \textit{Some}\ T \qquad \textit{wf-type}\ \Sigma\ T' \qquad T' \lesssim T}{\textit{wf-term}\ \Sigma\ (\textit{Ct}\ s\ T')}$$

$$\frac{\textit{wf-term}\ \Sigma\ t \qquad \textit{wf-term}\ \Sigma\ u}{\textit{wf-term}\ \Sigma\ (t \cdot u)}$$

$$\frac{\textit{wf-type}\ \Sigma\ T \qquad \textit{wf-term}\ \Sigma\ t}{\textit{wf-term}\ \Sigma\ (\textit{Abs}\ T\ t)}$$

These rules only check whether a term conforms to a signature, not that the contained types are consistent. Combining wellformedness and $\vdash_\tau$ yields welltypedness of a term:

$$\textit{wt-term}\ \Sigma\ t = (\textit{wf-term}\ \Sigma\ t \wedge (\exists T.\ \vdash_\tau t : T))$$

Wellformedness of a signature $\Sigma = (ctf, arf, oss)$ where $oss = (cl, sub, tcs)$ is defined as follows:

$$\begin{aligned}&\textit{wf-sig}\ \Sigma = ((\forall T{\in}\textit{ran}\ ctf.\ \textit{wf-type}\ \Sigma\ T) \wedge\\ &\qquad (\textit{wf-osig}\ oss \wedge (\forall(\kappa, Ss, c){\in}tcs.\ arf\ \kappa = \textit{Some}\ |Ss|))\end{aligned}$$

In words: all types in ctf are wellformed, oss is wellformed, and the type constructors in tcs are exactly those that have a matching arity in arf.

6 Logic

Isabelle's metalogic $\mathcal{M}$ is an extension of the logic described by Paulson [36]. It is a fragment of intuitionistic higher-order logic. The basic types and connectives of $\mathcal{M}$ are the following:

Concept	Representation	Abbreviation
Type of propositions	*Ty* "*prop*" []	*prop*
Implication	*Ct* "*imp*" (*prop* $\to$ *prop* $\to$ *prop*)	$\Longrightarrow$
Universal quantifier	*Ct* "*all*" ((*T* $\to$ *prop*) $\to$ *prop*)	$\bigwedge_T$
Equality	*Ct* "*eq*" (*T* $\to$ *T* $\to$ *prop*)	$=_T$

The type subscripts of $\bigwedge$ and $\equiv$ are dropped in the text if they can be inferred.

Readers familiar with Isabelle syntax must keep in mind that for readability we use the symbols $\bigwedge$, $\Longrightarrow$ and $\equiv$ for the *encodings* of the respective symbols in Isabelle's metalogic. We avoid the corresponding metalogical constants completely in favor of HOL's $\forall$, $\longrightarrow$, $=$, and inference rule notation.

The provability judgment of $\mathcal{M}$ is of the form $\Theta,\Gamma \vdash t$ where Θ is a theory, Γ (the hypotheses) is a set of terms of type *prop*, and t a term of type *prop*.

A *theory* is a pair of a signature and a set of axioms:

type_synonym $'v\ theory = 'v\ signature \times 'v\ term\ set$

The projection functions are *sig* and *axioms*. We extend the notion of wellformedness from signatures to theories:

wf-theory $(\Sigma, axs) =$
$(wf\text{-}sig\ \Sigma \wedge (\forall p{\in}axs.\ wt\text{-}term\ \Sigma\ p \wedge \vdash_\tau p : prop) \wedge is\text{-}std\text{-}sig\ \Sigma \wedge eq\text{-}axs \subseteq axs)$

The first two conjuncts need no explanation. Predicate *is-std-sig* (not shown) requires the signature to have certain minimal content: the basic types ($\to$, *prop*) and constants ($\equiv$, $\bigwedge$, $\Longrightarrow$) of $\mathcal{M}$ and the additional types and constants for type class reasoning from Sect. 6.3. Our theories also need to contain a minimal set of axioms. The set *eq-axs* is an axiomatic basis for equality reasoning and will be explained in Sect. 6.2.

We will now discuss the inference system in three steps: the basic inference rules, equality, and type class reasoning.

6.1 Basic Inference Rules

The *axiom rule* states that wellformed type-instances of axioms are provable:

$$\frac{wf\text{-}theory\ \Theta \qquad t \in axioms\ \Theta \qquad wf\text{-}inst\ (sig\ \Theta)\ \varrho}{\Theta,\Gamma \vdash \varrho\ \$\$\ t}$$

where $\varrho :: 'v \Rightarrow sort \Rightarrow 'v\ typ$ is a type substitution and \$\$ denotes its application (see Sect. 3). The types substituted into the type variables need to be wellformed and conform to the sort constraint of the type variable:

$$\begin{aligned}&\mathit{wf\text{-}inst}\ \Sigma\ \varrho = \\ &\quad(\forall v\ S.\ \varrho\ v\ S \neq \mathit{Tv}\ v\ S \longrightarrow \mathit{has\text{-}sort}\ (\mathit{osig}\ \Sigma)\ (\varrho\ v\ S)\ S \wedge \mathit{wf\text{-}type}\ \Sigma\ (\varrho\ v\ S))\end{aligned}$$

The conjunction only needs to hold if ϱ actually changes something, i.e., if $\varrho\ v\ S \neq \mathit{Tv}\ v\ S$. This condition is not superfluous because otherwise $\mathit{has\text{-}sort}\ oss\ (\mathit{Tv}\ v\ S)\ S$ and $\mathit{wf\text{-}type}\ \Sigma\ (\mathit{Tv}\ v\ S)$ only hold if S is wellformed w.r.t Σ.

Note that there are no extra rules for general instantiation of type or term variables. Type variables can only be instantiated in the axioms. Term instantiation can be performed using the $\bigwedge$ introduction and elimination rules.

The *assumption rule* allows us to prove terms already in the hypotheses:

$$\frac{\mathit{wf\text{-}term}\ (\mathit{sig}\ \Theta)\ t \qquad \vdash_\tau t : \mathit{prop} \qquad t \in \Gamma}{\Theta, \Gamma \vdash t}$$

Both $\bigwedge$ and $\Longrightarrow$ are characterized by introduction and elimination rules:

$$\frac{\mathit{wf\text{-}theory}\ \Theta \qquad \Theta, \Gamma \vdash t \qquad (x, T) \notin \mathit{FV}\ \Gamma \qquad \mathit{wf\text{-}type}\ (\mathit{sig}\ \Theta)\ T}{\Theta, \Gamma \vdash \bigwedge_T\ (\mathit{Abs\text{-}fv}\ x\ T\ t)}$$

$$\frac{\Theta, \Gamma \vdash \bigwedge_T\ (\mathit{Abs}\ T\ t) \qquad \vdash_\tau u : T \qquad \mathit{wf\text{-}term}\ (\mathit{sig}\ \Theta)\ u}{\Theta, \Gamma \vdash \mathit{subst\text{-}bv}\ u\ t}$$

$$\frac{\mathit{wf\text{-}theory}\ \Theta \qquad \Theta, \Gamma \vdash u \qquad \mathit{wf\text{-}term}\ (\mathit{sig}\ \Theta)\ t \qquad \vdash_\tau t : \mathit{prop}}{\Theta, \Gamma - \{t\} \vdash t \Longrightarrow u}$$

$$\frac{\Theta, \Gamma_1 \vdash t \Longrightarrow u \qquad \Theta, \Gamma_2 \vdash t}{\Theta, \Gamma_1 \cup \Gamma_2 \vdash u}$$

where $\mathit{FV}\ \Gamma = (\bigcup_{t \in \Gamma}\ \mathit{fv}\ t)$.

6.2 Equality

Most rules about equality are not part of the inference system but are axioms (the set *eq-axs* mentioned above). Consequences are obtained via the axiom rule.

The first three axioms express that $=$ is reflexive, symmetric, and transitive:

$$x \equiv x \qquad x \equiv y \Longrightarrow y \equiv x \qquad x \equiv y \Longrightarrow y \equiv z \Longrightarrow x \equiv z$$

The next two axioms express that terms A and B of type *prop* are equal iff they imply each other:

$$A \equiv B \Longrightarrow A \Longrightarrow B \qquad (A \Longrightarrow B) \Longrightarrow (B \Longrightarrow A) \Longrightarrow A \equiv B$$

The last equality axioms are congruence rules for application and abstraction:

$$\begin{aligned}&f \equiv g \Longrightarrow x \equiv y \Longrightarrow (f \cdot x) \equiv (g \cdot y) \\ &\bigwedge (\mathit{Abs}\ T\ ((f \cdot \mathit{Bv}\ 0) \equiv (g \cdot \mathit{Bv}\ 0))) \\ &\Longrightarrow \mathit{Abs}\ T\ (f \cdot \mathit{Bv}\ 0) \equiv \mathit{Abs}\ T\ (g \cdot \mathit{Bv}\ 0)\end{aligned}$$

Paulson [36] gives a slightly different congruence rule for abstraction, which allows to abstract over an arbitrary, free x in f, g. We are able to derive this rule in our inference system.

Finally, there are the lambda calculus rules. There is no need for α conversion because α-equivalent terms are already identical thanks to the De Bruijn indices for bound variables. For β and η conversion the following rules are added. In contrast to the rest of this subsection, these are not expressed as axioms.

$$\frac{\begin{array}{c}\mathit{wf\text{-}theory}\ \Theta \\ \mathit{wt\text{-}term}\ (\mathit{sig}\ \Theta)\ (\mathit{Abs}\ T\ t) \qquad \mathit{wf\text{-}term}\ (\mathit{sig}\ \Theta)\ u \qquad \vdash_\tau u : T\end{array}}{\Theta,\Gamma \vdash (\mathit{Abs}\ T\ t \cdot u) \equiv \mathit{subst\text{-}bv}\ u\ t}\ (\beta)$$

$$\frac{\mathit{wf\text{-}theory}\ \Theta \qquad \mathit{wf\text{-}term}\ (\mathit{sig}\ \Theta)\ t \qquad \vdash_\tau t : T \to T'}{\Theta,\Gamma \vdash \mathit{Abs}\ T\ (t \cdot \mathit{Bv}\ 0) \equiv t}\ (\eta)$$

Rule (β) uses the substitution function *subst-bv* as explained in Sect. 3 (and defined in the Appendix).

Rule (η) requires a few words of explanation. We do not explicitly require that t does not contain *Bv 0*. This is already a consequence of the precondition that $\vdash_\tau t : T \to T'$: it implies that t is closed. For that reason, it is perfectly unproblematic to remove the abstraction above t.

6.3 Type Class Reasoning

Wenzel [43] encoded class constraints of the form "type T has class C" in the term language as follows. There is a unary type constructor named "$itself$" and T *itself* abbreviates *Ty* "$itself$" [T]. The notation $TYPE_{T\ itself}$ is short for *Ct* "$type$" (T *itself*) where "$type$" is the name of a constant. You should view $TYPE_{T\ itself}$ as the term-level representation of type T.

Next we represent the predicate "is of class C" on the term level. For this, we define some fixed injective mapping *const-of-class* from class to constant names. For each new class C, a new constant *const-of-class* C of type T *itself* $\to$ *prop* is added. The term *Ct* (*const-of-class* C) (T *itself* $\to$ *prop*) $\cdot\ TYPE_{T\ itself}$ represents the statement "type T has class C". This is the inference rule deriving such propositions:

$$\frac{\begin{array}{c}\mathit{wf\text{-}theory}\ \Theta \\ \mathit{const\text{-}type}\ (\mathit{sig}\ \Theta)\ (\mathit{const\text{-}of\text{-}class}\ C) = \mathit{Some}\ ('a\ \mathit{itself} \to \mathit{prop}) \\ \mathit{wf\text{-}type}\ (\mathit{sig}\ \Theta)\ T \qquad \mathit{has\text{-}sort}\ (\mathit{osig}\ (\mathit{sig}\ \Theta))\ T\ \{C\}\end{array}}{\Theta,\Gamma \vdash \mathit{Ct}\ (\mathit{const\text{-}of\text{-}class}\ C)\ (T\ \mathit{itself} \to \mathit{prop}) \cdot TYPE_{T\ itself}}$$

This is how the *has-sort* inference system is integrated into the logic.

This concludes the presentation of $\mathcal{M}$. We have shown some minimal sanity properties, incl. that all provable terms are of type *prop* and wellformed:

$$\Theta,\Gamma \vdash t \longrightarrow \vdash_\tau t : \mathit{prop} \wedge \mathit{wf\text{-}term}\ (\mathit{sig}\ \Theta)\ t$$

The attentive reader will have noticed that we do not require unused hypotheses in Γ to be wellformed and of type *prop*. Similarly, we only require *wf-theory* Θ in rules that need it to preserve wellformedness of the terms and types involved. To restrict to wellformed theories and hypotheses, we define a top-level provability judgment that requires wellformedness:

$$\Theta,\Gamma \Vdash t = (\mathit{wf\text{-}theory}\ \Theta \wedge (\forall h{\in}\Gamma.\ \mathit{wf\text{-}term}\ (\mathit{sig}\ \Theta)\ h \wedge \vdash_\tau h : \mathit{prop}) \wedge \Theta,\Gamma \vdash t)$$

7 Admissible Rules

Reasoning directly with these basic rules can be very tedious. In the following, we discuss some useful admissible rules that we frequently encountered during our formalization work and sketch their formal proofs.

As already mentioned, our inference system has no inbuilt way of performing term substitutions and one has to simulate them using $\bigwedge$-introductions and eliminations. This can be particularly annoying when performing simultaneous substitutions, as one needs to ensure that no interferences occur. We define a function *subst-term* which takes a list of (variable,term) pairs, an association list, and substitutes them simultaneously into a term and prove the following corresponding rule:

$$\frac{\textit{wf-theory}\ \Theta \qquad \Theta,\Gamma \vdash B \qquad \textit{tinst-ok}\ (\textit{sig}\ \Theta)\ insts \qquad \textit{fst}\ `\ insts \cap \textit{FV}\ \Gamma = \emptyset}{\Theta,\Gamma \vdash \textit{subst-term}\ insts\ B}$$

where *tinst-ok* requires there to be only one instantiation per variable and for the terms to be wellformed and of the same type as their corresponding variable. The last condition ensures that we do not substitute into variables occurring in an assumption. To prove this rule, we start with the special case of a single instantiation, which is performed by a single $\bigwedge$-introduction followed by a $\bigwedge$-elimination. To use this result to prove our desired rule, we need to perform the simultaneous instantiations sequentially. This is not possible in general, as they might interfere with one another. To remedy this, we decompose the substitution process into two phases: First we replace all variables we want to substitute into with distinct, fresh variables. Then we modify the original instantiations, so they substitute into their corresponding new variables instead. As these are fresh, they do not occur in the substituted terms. Therefore, no interference occurs and we can perform both phases sequentially.

Another useful derived result is the weakening rule, which tells us that correct inferences remain correct when adding additional (superfluous) assumptions:

$$\frac{\Theta,\Gamma \vdash B \qquad \textit{wf-term}\ (\textit{sig}\ \Theta)\ A \qquad \vdash_\tau A : \textit{prop}}{\Theta,\{A\} \cup \Gamma \vdash B}$$

We prove this rule by rule induction on $\Theta,\Gamma \vdash B$. Most cases do not interact with Γ at all and are, therefore, trivial. The only interesting case is the one for $\bigwedge$-introduction, where, once again, variable capture causes trouble. To prove this case, we would like to use the corresponding $\bigwedge$-introduction rule, but the added assumption A might contain the variable we want to bind. To fix this problem, we use the substitution rule proved above to rename the problematic variable in A. As we cannot use this rule to substitute into the hypotheses, we use implication rules to move A into the proposition on the right side of the turnstile.

Another major source of complications is equality reasoning. We start by providing corresponding proof rules for each equality axiom, making them easier to use. For example, the resulting rule for reflexivity looks like this:

$$\frac{\textit{wf-theory}\ \Theta \qquad \forall A \in \Gamma.\ \textit{wf-term}\ (\textit{sig}\ \Theta)\ A \wedge \vdash_\tau A : \textit{prop} \qquad \textit{wt-term}\ (\textit{sig}\ \Theta)\ t}{\Theta,\Gamma \vdash t \equiv t}$$

The proofs for all these rules are very similar. We first prove them for an empty set of assumptions, to, once again, avoid accidental variable capturing. For this, we use the axiom rule and the derived simultaneous term instantiation rule to substitute the correct types and terms into the axiom. Then we allow for arbitrary (well typed) assumptions using the weakening rule.

By just combining these equality rules and rules of the inference system one can derive new rules, like for example, Paulson's original congruence for abstraction. Such derivations are, however, complicated by having to propagate the wellformedness conditions of all involved

objects, which makes them lengthy. As they also clutter the presentation, we will not show them for the rest of this section and assume that all involved objects in the rules are wellformed.

A last derived rule, which will be useful later, is the fact that β-reduction preserves provability. Our inference system already contains a rule concerning β-reduction but it can only be applied at the top of a term.

We first define an inductive notion of a β-step, based on a formalization by Nipkow [33].

$$\mathit{Abs}\ T\ s \cdot t \to_\beta \mathit{subst\text{-}bv}\ t\ s$$

$$\frac{s \to_\beta t}{s \cdot u \to_\beta t \cdot u} \qquad \frac{s \to_\beta t}{u \cdot s \to_\beta u \cdot t} \qquad \frac{s \to_\beta t}{\mathit{Abs}\ T\ s \to_\beta \mathit{Abs}\ T\ t}$$

This allows us to state a more useful β rule, which proves equality of two terms if they just differ by a single beta step, regardless of where it occurs.

$$\frac{t \to_\beta u}{\Theta, \Gamma \vdash t \equiv u}$$

We naturally want to prove this statement by induction over $(\to_\beta)$. Because $(\to_\beta)$ is defined by four rules, there are four cases. The first case allows application of the β rule of the inference system, the next two use the congruence rule for applications and the respective induction hypotheses. However, the last case is a problem, as descending under an abstraction can expose previously bound variables, which means we cannot apply our proof rules. To remedy this, we replace the now loose variable with a fresh free variable and perform our reasoning with the again wellformed term. The following rule justifies this transformation and is readily proved by combining basic and derived inference rules:

$$\frac{(x, \tau') \notin \mathit{FV}\ \Gamma \cup \mathit{fv}\ s \cup \mathit{fv}\ t \qquad \Theta, \Gamma \vdash \mathit{subst\text{-}bv}\ (\mathit{Fv}\ x\ \tau')\ s \equiv \mathit{subst\text{-}bv}\ (\mathit{Fv}\ x\ \tau')\ t}{\Theta, \Gamma \vdash \mathit{Abs}\ \tau'\ s \equiv \mathit{Abs}\ \tau'\ t}$$

However, applying this rule makes it impossible to use the induction hypothesis as adding the substitution changes the shape of the goal. The solution is to generalize the β rule to reflect that one might have passed abstractions and substituted the respective loose variables by new fresh variables:

$$\frac{\mathit{set}\ vs \cap (\mathit{fv}\ t \cup \mathit{FV}\ \Gamma) = \emptyset \qquad t \to_\beta u}{\begin{array}{l}\Theta, \Gamma \vdash \mathit{subst\text{-}bvs}\ (\mathit{map}\ (\lambda(v, \tau).\ \mathit{Fv}\ v\ \tau)\ vs)\ t \equiv \\ \qquad \mathit{subst\text{-}bvs}\ (\mathit{map}\ (\lambda(v, \tau).\ \mathit{Fv}\ v\ \tau)\ vs)\ u\end{array}}$$

This rule uses the *subst-bvs* function, which behaves like the previously seen *subst-bv* function, only instantiating multiple loose variables simultaneously. We can prove that it is possible to merge the call to *subst-bv*, arising in the problematic case with the other substitutions. Therefore, we are now able to apply the induction hypothesis. The original rule is the special case for an empty list of variables.

A similar approach was taken to prove that η reduction preserves provability. Because of symmetry, we have also proved that β/η expansion preserves provability, or combined that β/η convertibility does not affect provability.

8 Proof Terms and Checker

Berghofer and Nipkow [7] added proof terms to Isabelle. We present an executable checker for these proof terms that is proved sound w.r.t. the above formalization of the metalogic.

Berghofer and Nipkow also developed a proof checker but it is unverified and checks the generated proof terms by feeding them back through Isabelle's unverified inference kernel.

It is crucial to realize that all we need to know about the proof term checker is the soundness theorem below. The internals are, from a soundness perspective, irrelevant, which is why we can get away with sketching them informally. For this reason, we will not give definitions for all involved functions in the following presentation, preferring informal descriptions (all definitions are of course part of the formalization). This is in contrast to the logic itself, which acts like a specification, which is why we presented it in detail.

This is our data type of proof terms:

$$\begin{aligned}&\textbf{datatype}\ 'v\ proofterm = \mathit{PAxm}\ name\ ((('v \times sort) \times 'v\ typ)\ list)\\ &\quad |\ \mathit{PThm}\ name\ ((('v \times sort) \times 'v\ typ)\ list)\ |\ \mathit{PBound}\ nat\\ &\quad |\ \mathit{Abst}\ ('v\ typ)\ ('v\ proofterm)\ |\ \mathit{AbsP}\ ('v\ term)\ ('v\ proofterm)\\ &\quad |\ \mathit{Appt}\ ('v\ proofterm)\ ('v\ term)\ |\ \mathit{AppP}\ ('v\ proofterm)\ ('v\ proofterm)\\ &\quad |\ \mathit{OfClass}\ ('v\ typ)\ name\ |\ \mathit{Hyp}\ ('v\ term)\end{aligned}$$

These proof terms are not designed to record proofs in our inference system, but to mirror the proof terms generated by Isabelle. Nevertheless, the constructors of our proof terms correspond roughly to the rules of the inference system. As the axiom rule of our inference system allows for type instantiations, *PAxm* contains an axiom and a type substitution. This substitution is encoded as an association list instead of a function. The axiom is referenced by $name$. During proof checking, a mapping from names to terms is provided. *PThm* represents a (previously proved) theorem, containing the same information as a *PAxm* constructor. While we treat theorems as axioms, they use a different constructor, as axioms and theorems make use of different namespaces in the implementation. *AbsP* and *Abst* correspond to introduction of $\Longrightarrow$ and $\bigwedge$, *AppP* and *Appt* correspond to the respective eliminations. *Hyp* and *PBound* relate to the assumption rule, where *Hyp* refers to a free assumption while *PBound* contains a De Bruijn index referring to an assumption added during the proof by an *AbsP* constructor. *OfClass* denotes a proof that a type belongs to a given type class.

Isabelle looks at terms modulo $\alpha\beta\eta$-equivalence and, therefore, does not save β or η steps, while they are explicit steps in our inference system. Therefore we have no constructors corresponding to the (β) and (η) rules. The remaining equality axioms are naturally handled by the *PAxm* constructor.

In the rest of this section, we discuss how to derive an executable proof checker. Executability means that the checker is defined as a set of recursive functions that Isabelle's code generator can translate into one of a number of target languages, in particular its implementation language SML [6, 12, 13].

Because of the approximate correspondence between proof term constructors and inference rules, implementing the proof checker largely amounts to providing executable versions of each inference rule, as in LCF: each rule becomes a function that checks the side conditions, and if they are true, computes the conclusion from the premises given as arguments. However, as the checked proof terms are not for our exact inference system but the implementation, there is some additional work to perform. The heart of our checker is a recursive function, where var is a concrete type of variables discussed further down:

$$\begin{aligned}&\mathit{replay}::var\ theory\\ &\qquad \Rightarrow var\ typ\ list \Rightarrow var\ term\ list \Rightarrow var\ proofterm \Rightarrow var\ term\ option\end{aligned}$$

It takes a theory, a context (see below), a list of current assumptions and a proof term and returns the certified proposition for a valid proof term or *None* for an invalid one. We now

discuss some of the more involved implementation steps and illustrate them with some cases of the *replay* function.

First, as we save only names of axioms and theorems in the proof terms, not the propositions themselves, we need a way to look these up. Therefore, our proof checker actually uses a slightly different theory type than the one show before:

type_synonym $'v\ theory' =$
$'v\ signature \times ((name \rightharpoonup 'v\ term) \times (name \rightharpoonup 'v\ term))$

It replaces the set of axioms with two (finite) maps. These allow efficient, name-based lookup of the actual axiom/theorem. We use two maps, as the Isabelle implementation uses distinct namespaces for axioms and theorems. The set of axioms in the original type can be recovered as the union of the ranges of the two maps. In the following, we will hide this implementation detail and use the original $theory$ type.

The type var of variables is defined as follows:

type_synonym $indexname = (name \times int)$
datatype $var =$ *Free* $name \mid$ *Var* $indexname \mid$ *Internal* nat

The constructors *Free* and *Var* are "inherited" from the Isabelle implementation of type $term$. Both represent free variables but the ones represented by *Var* can be instantiated by unification. We added a third constructor, not present in Isabelle, which we use for easy generation of fresh variables by simply counting up.

Such internal variables are generated only when visiting an *Abst* constructor ($\bigwedge$-introduction) during the traversal. *Abst* constructors introduce $\bigwedge$-quantifiers, which eventually bind variables present in the contained proof term. The proof terms generated by Isabelle already use De Bruijn notation for these variables, so descending under *Abst* constructor can produce loose variables. For each of them, we add an internal variable to the context, which contains exactly the types of *Abst*-bound variables passed while descending into the proof term. To obtain a fresh variable, we use the size of the current context. Conceptually, when passing an *Abst* constructor, our proof checker substitutes the newly generated variable for this index everywhere in the proof term, then replays the proof, and binds the variable again in the result, therefore, always working with closed terms. However, performing this substitution first would require an extra traversal of the proof term at each *Abst* constructor. To avoid this, we remember all passed *Abst* constructors and substitute the corresponding variables simultaneously.

replay $\Theta\ vs\ Hs$ (*Abst* $T\ p) =$
(*if wf-type* (*sig* Θ) T
then map-option $(\lambda t.\ \bigwedge_T$ (*Abs-fv* (*Internal* $|vs|$) $T\ t$)) (*replay* Θ $(T \#\ vs)\ Hs\ p$)
else None)

Such substitutions happen at the *AbsP* ($\Longrightarrow$-introduction) and the *Appt* ($\bigwedge$-elimination) case, by means of the *subst-bvs* function. Note that, somewhat counterintuitively, the innermost abstraction/lowest De Bruijn index corresponds to the highest internal name. We only show the *AbsP* case here:

replay $\Theta\ vs\ Hs$ (*AbsP* $t\ p) =$
(*let* $t' =$ *subst-bvs* (*map-index* $(\lambda i\ T.$ *Fv* (*Internal* $(|vs| - Suc\ i))\ T)\ vs)\ t$;
$rep =$ *replay* $\Theta\ vs\ (t' \#\ Hs)\ p$
in if $\vdash_\tau t'$: *prop* $\wedge$ *wf-term* (*sig* Θ) t' *then map-option* $((\Longrightarrow)\ t')\ rep$ *else None*)

We have shown that the result of running *replay* does not contain any internal variables (as long as the inputs do not contain any).

As already mentioned, term instantiations can be performed by means of the $\bigwedge$ rules. However, the proof terms generated by Isabelle do not use these rules when instantiating term variables in axioms. Instead, all variables in an axiom are assumed to already have been universally quantified, so that only the elimination step remains. For our checker, this means we need to also $\bigwedge$-quantify all free variables when handling a *PAxm* constructor. A pitfall here is the order in which we quantify the free variables. The structure of the proof terms expects them to occur in the order given by a reverse inorder traversal of the axiom. As theorems are treated as just another kind of axioms, *PThm* is handled analogously.

$$\begin{aligned}&\mathit{replay}\ \Theta\ \_\ \_\ (\mathit{PAxm}\ n\ Tis) =\\ &\quad (\mathit{if}\ \mathit{inst\text{-}ok}\ (\mathit{sig}\ \Theta)\ Tis\\ &\quad \mathit{then}\ \mathit{map\text{-}option}\ (\lambda t.\ \mathit{all\text{-}close}\ (\mathit{subst\text{-}typ}\ Tis\ t))\ (\mathit{axioms}\ \Theta\ n)\ \mathit{else}\ \mathit{None})\end{aligned}$$

To model Isabelle's view of terms modulo $\alpha\beta\eta$-equivalence, we sometimes $\beta\eta$ normalize our terms (α-equivalence is for free thanks to De Bruijn notation) during the reconstruction of the proof. This is necessary when replaying an *AppP* constructor because checking the conditions of the corresponding implication elimination rule requires checking equality of two terms. For all other constructors, no equality checks are necessary. To avoid repeatedly traversing the terms we only normalize in the *AppP* case and work with possibly non-$\beta\eta$ normalized terms in all other cases.

$$\begin{aligned}&\mathit{replay}\ \Theta\ vs\ Hs\ (\mathit{AppP}\ p_1\ p_2) =\\ &\quad \mathit{let}\ rep1 = \mathit{replay}\ \Theta\ vs\ Hs\ p_1 \gg\!= \mathit{beta\text{-}eta\text{-}norm};\\ &\qquad rep2 = \mathit{replay}\ \Theta\ vs\ Hs\ p_2 \gg\!= \mathit{beta\text{-}eta\text{-}norm}\\ &\quad \mathit{in}\ \mathit{case}\ (rep1, rep2)\ \mathit{of}\\ &\qquad (\mathit{Some}\ (A \Longrightarrow B), \mathit{Some}\ A') \Rightarrow\\ &\qquad\ \ \mathit{if}\ A{=}A'\ \mathit{then}\ \mathit{Some}\ B\ \mathit{else}\ \mathit{None}\\ &\qquad \mid\ \_ \Rightarrow \mathit{None}\end{aligned}$$

For our soundness proof of the checker, we need to verify that these normalizations preserve provability. For this, we show that they can be expressed as a finite number of β reduction steps, followed by a finite number of η reduction steps. These steps can then be justified using the rules presented in Sect. 7, yielding us the desired result.

$$\begin{aligned}&\mathit{wf\text{-}theory}\ \Theta \wedge (\forall A{\in}\Gamma.\ \mathit{wt\text{-}term}\ (\mathit{sig}\ \Theta)\ A \wedge \vdash_\tau A : \mathit{prop}) \wedge\\ &\quad \Theta, \Gamma \vdash t \wedge \mathit{beta\text{-}eta\text{-}norm}\ t = \mathit{Some}\ u \longrightarrow\\ &\quad \Theta, \Gamma \vdash u\end{aligned}$$

For our *replay* function to be executable, all constructs used by it must be executable as well. As we are continuously using explicitly finite data structures in the definition of (order sorted) signatures, Isabelle's code generator needs no further help handling them. Still, the representation of type constructor signatures in Sect. 4 as an unstructured set tcs does not facilitate efficient access to relevant information. In particular, to compute $\mathit{has\text{-}sort}\ oss\ (\mathit{Ty}\ \kappa\ Ts)\ S$, one needs to find the signatures for κ required to fulfill all class constraints in S. This means searching all of tcs for each constraint. To speed this up, we define an alternative representation *TCS*, inspired by the Isabelle implementation, which the code generator can transparently use as a replacement. This *TCS* component has type $name \rightharpoonup (class \rightharpoonup sort\ list)$, and it first groups all signatures by type constructor and then allows finding necessary argument sort constraints by passing an expected return class. More formally, *TCS* represents the set of all type constructor signatures $\kappa :: (Ss)\ c$ such that $TCS\ \kappa = \mathit{Some}\ dm$ and $dm\ c = \mathit{Some}\ Ss$.

We can therefore recreate the equivalent but more intuitive, original version tcs the following way:

$$tcs = \{(\kappa, Ss, c) \mid \exists dm.\ TCS\ \kappa = Some\ dm \wedge dm\ c = Some\ Ss\}$$

We also need to make the inductive wellformedness checks for sorts, types, terms, signatures and theories executable. Mostly, this amounts to providing recursive versions for their inductive definitions and proving them equivalent. A problematic point is the definition of the type instance relation ($\lesssim$), which contains an (unbounded) existential quantifier. To make this executable, we provide an implementation which tries to compute a suitable type substitution by matching the types.

In the end, we obtain an executable proof checker

$$\begin{aligned}&\mathit{check\text{-}proof}\ \Theta\ P\ p = \\ &\quad (\mathit{wf\text{-}theory}\ \Theta \wedge \\ &\quad (\forall h \in \mathit{hyps}\ P.\ \mathit{wf\text{-}term}\ (\mathit{sig}\ \Theta)\ h \wedge \vdash_\tau h : \mathit{prop}) \wedge \mathit{replay\text{-}norm}\ \Theta\ P = Some\ p)\end{aligned}$$

where $\mathit{replay\text{-}norm}\ \Theta\ P = (\mathit{replay}\ \Theta\ []\ (\mathit{hyps}\ P)\ P \gg\!= \mathit{beta\text{-}eta\text{-}norm})$. This final $\beta\eta$ normalization step is once again necessary to account for possible different but $\alpha\beta\eta$-equivalent results.

check-proof checks wellformedness of theory Θ and the hypotheses and then checks if proof P proves the given proposition p. The latter check is important because the Isabelle theorems that we check contain both a proof and a proposition that the theorem claims to prove. As one of our main results, we can prove the correctness of our checker:

$$\mathit{check\text{-}proof}\ \Theta\ P\ p \longrightarrow \Theta, \mathit{hyps}\ P \Vdash p$$

The proof itself is conceptually simple and proceeds by induction over the structure of the computation of *replay*. For each proof constructor we need to show that there are corresponding inference rules in our system for each step taken by the functional version *replay*. Most of the proof effort goes into a large library of results about terms, types, signatures, substitutions, wellformedness etc. required for this proof. In particular, we need to prove derived rules characterizing all the technical operations we use, similar to Sect. 7.

9 Size and Structure of the Formalization

All material presented so far has been formalized in Isabelle/HOL. The definition of the inference system (incl. types, terms etc.) resides in a separate theory $Core$ that depends only on the basic library of Isabelle/HOL. It takes about 300 LOC and is fairly high level and readable – we presented most of it. This is at least an order or magnitude smaller than Isabelle's inference kernel (which is not clearly delineated) – of course, the latter is optimized for performance. Its abstract type of theorems alone takes about 2,500 LOC, not counting any infrastructure of terms, types, unification etc.

The whole formalization consists of 12,000 LOC. The main components are:

- Almost half the formalization (5,500 LOC) is devoted to providing a library of operations on types and terms and their properties. This includes, among others, executable functions for type checking, different types of substitutions, abstractions, the wellformedness checks, and β and η reductions.
- Proving admissible rules of our inference system takes up 3,000 LOC. A large part of this is deriving rules for equality and the β and η reductions. Weakening rules are also derived.

- Making the wellformedness checks for (order-sorted) signatures and theories as well as the type instance checks executable takes 1,800 LOC.
- Definition and correctness proof for the checker builds on the above material and take only about 500 additional LOC.
- Around 1,000 LOC are spent on preliminary material, most importantly results about finite sets and maps, transferred from existing material for general sets and maps.

10 Integration with Isabelle

As explained above, Isabelle generates SML code for the proof checker. This code has its own definitions of types, terms etc. and needs to be interfaced with the corresponding data structures in Isabelle. This step requires 150 lines of handwritten SML code (*glue code*) that translates Isabelle's data structures into the corresponding data structures in the generated proof checker such that we can feed them into *check-proof*. We cannot verify this code and therefore aim to keep it as small and simple as possible. This is the reason for the previously mentioned *intentional implementation bias* we introduced in our formalization. We describe now how the various data types are translated. We call a translation trivial if it merely replaces one constructor by another, possibly forgetting some information.

The translation of types and terms is trivial as their structure is almost identical in the two settings.

Proof term translation is trivial except for two special cases. So-called "oracles" (typically the result of unfinished proofs, i.e. ,"sorry" on the user level) are rejected (but none of the theories we checked contain oracles). Furthermore, translating previously proved lemmas requires some additional name handling work. Also remember that the translation of proofs is not safety critical because all that matters is that in the end we obtain a correct proof of the claimed proposition.

We also provide functions to translate relevant content from the background theory: axioms (including previously proved theorems) and (order-sorted) signatures. This mostly amounts to extracting association lists from efficient internal data structures. Translating the axioms also involves translating some alternative internal representation of type class constraints into their standard form presented in Sect. 6.3.

The checker is integrated into Isabelle by calling it every time a new named theorem has been proved. The set of theorems proved so far is added to the axiomatic basis for this check. Cyclic dependencies between lemmas are ruled out by this ordering because every theorem is checked before being added to the axiomatic basis. However, an explicit cyclicity check is not part of the formalization (yet), which speaks only about checking single proofs.

11 Running the Proof Checker

We run this modified Isabelle with our proof checker on multiple theories in various object logics contained in the Isabelle distribution. A rough overview of the scope of the covered material for some logics and the required running times can be found in the following table. The running times are the total times for running Isabelle, not just the proof checking, but the latter takes over 90% of the time. All tests were performed on an Intel Core i7-9750H CPU running at 2.60GHz and 32GB of RAM.

Logic	LOC	Time
FOL	4,500	40 s
ZF	55,000	12 min
HOL	29,000	110 min

We can check the material in several smaller object logics in their entirety. One of the larger such logics is first-order logic (FOL). These logics do not develop any applications but FOL comes with proof automation and theories testing that automation, in particular Pelletier's collection of problems that were considered challenges in their day [37]. Because the proofs are found automatically, the resulting proof terms will typically be quite complex and good test material for a proof checker.

The logic ZF (Zermelo-Fraenkel set theory) builds on FOL but contains real applications and is an order of magnitude larger than FOL. We are able to check all material formalized in ZF in the Isabelle distribution.

Isabelle's most frequently used and largest object logic is HOL. We managed to check some of the initial theories of the main library. These theories contain the basic logic and among others the libraries of sets, functions, orderings, lattices, groups, rings, fields and natural numbers. The formalizations are non-trivial and make heavy use of Isabelle's type classes.

Why is checking material in ZF easier than in HOL? Profiling revealed that the proof checker spends a lot of time in functions that access the signature, especially the wellformedness checks. One reason for this is inefficient data structures (e.g., association lists) and thus the running time depends heavily on the size of the signature and increases with every new constant, type and class. This is aggravated by our current approach which exports the current state of the background theory and has to ensure its wellformedness before each check. Furthermore, there is no sharing of any kind in terms/types and their wellformedness checks. Because ZF is free of polymorphism and type classes, these checks are much simpler. Lastly, the presence of type classes also increases the size of the involved proofterms. These effects can also be seen purely in the HOL material. For example, despite having similar sizes and containing roughly the same number of theorems, checking the material on rings takes about 10 times as long as the one on natural numbers.

12 Trust Assumptions

We need to trust the following components outside of the formalization:

- The verification (and code generation) of our proof checker in Isabelle/HOL. This is inevitable, one has to trust some theorem prover to start with. We could improve the trustworthiness of this step by porting our proofs to the verified HOL prover by Kumar et al. [20] but its code generator produces CakeML [19], not SML.
- The unverified glue code in the integration of our proof checker into Isabelle (Sect. 10).

Because users currently cannot examine Isabelle's internal data structures that we start from, they have to trust Isabelle's front end that parses and transforms some textual input file into internal data structures. One could add a (possibly verified) presentation layer that outputs those internal representations into a readable format that can be inspected, while avoiding the traps Adams [2] is concerned with.

13 Future Work

Our primary focus will be on scaling up the proof checker to not just deal with all of HOL but with real applications (including itself!). There is a host of avenues for exploration. Just to name a few promising directions: more efficient data structures than association lists (e.g., via existing frameworks [25, 26]); caching of wellformedness checks for types and terms; exploiting sharing within terms and types (tricky because our intentionally simple glue code creates copies); working with the compressed proof terms [6] that Isabelle creates by default instead of uncompressing them as we do now.

We will also upgrade the formalization of our checker from individual theorems to sets of theorems, explicitly checking cyclic dependencies (which are currently prevented by the glue code, see Sect. 10).

A presentation layer as discussed in Sect. 12 would not just allow the inspection of the internal representation of the theories but could also be extended to the proofs themselves, thus, permitting checkers to be interfaced with Isabelle on a textual level instead of on internal data structures.

Another possible next step would be to formalize the theory extension mechanisms including verified cyclicity checks. It would also be nice to have a model-theoretic semantics for $\mathcal{M}$. We believe that the work by Kunčar and Popescu [21–24] could be adapted from HOL to $\mathcal{M}$ for these purposes.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s10817-022-09648-w.

Acknowledgements We thank Kevin Kappelmann, Magnus Myreen, Larry Paulson, Andrei Popescu, Makarius Wenzel, and the anonymous reviewers for their comments. Supported by Wirtschaftsministerium Bayern under DIK-2002-0027//DIK0185/03 and DFG GRK 2428 ConVeY

Funding Open Access funding enabled and organized by Projekt DEAL.

A Appendix

$$\mathit{subst\text{-}bv}\ u\ t = \mathit{subst\text{-}bv2}\ t\ 0\ u$$

$$\begin{aligned}
&\mathit{subst\text{-}bv2}\ (\mathit{Bv}\ i)\ n\ u = (\mathit{if}\ i < n\ \mathit{then}\ \mathit{Bv}\ i\ \mathit{else\ if}\ i = n\ \mathit{then}\ u\ \mathit{else}\ \mathit{Bv}\ (i-1))\\
&\mathit{subst\text{-}bv2}\ (\mathit{Abs}\ T\ t)\ n\ u = \mathit{Abs}\ T\ (\mathit{subst\text{-}bv2}\ t\ (n+1)\ (\mathit{lift}\ u\ 0))\\
&\mathit{subst\text{-}bv2}\ (f \cdot t)\ n\ u = \mathit{subst\text{-}bv2}\ f\ n\ u \cdot \mathit{subst\text{-}bv2}\ t\ n\ u\\
&\mathit{subst\text{-}bv2}\ t\ _\ _ = t
\end{aligned}$$

$$\begin{aligned}
&\mathit{lift}\ (\mathit{Bv}\ i)\ n = (\mathit{if}\ n \leq i\ \mathit{then}\ \mathit{Bv}\ (i+1)\ \mathit{else}\ \mathit{Bv}\ i)\\
&\mathit{lift}\ (\mathit{Abs}\ T\ t)\ n = \mathit{Abs}\ T\ (\mathit{lift}\ t\ (n+1))\\
&\mathit{lift}\ (f \cdot t)\ n = \mathit{lift}\ f\ n \cdot \mathit{lift}\ t\ n\\
&\mathit{lift}\ t\ _ = t
\end{aligned}$$

$$\textit{bind-fv}\ T\ t = \textit{bind-fv2}\ T\ 0\ t$$

$$\textit{bind-fv2}\ var\ n\ (\textit{Fv}\ v\ T) = (\textit{if}\ var = (v, T)\ \textit{then}\ \textit{Bv}\ n\ \textit{else}\ \textit{Fv}\ v\ T)$$
$$\textit{bind-fv2}\ var\ n\ (\textit{Abs}\ T\ t) = \textit{Abs}\ T\ (\textit{bind-fv2}\ var\ (n+1)\ t)$$
$$\textit{bind-fv2}\ var\ n\ (f \cdot u) = \textit{bind-fv2}\ var\ n\ f \cdot \textit{bind-fv2}\ var\ n\ u$$
$$\textit{bind-fv2}\ \_\ \_\ t = t$$

$$\textit{tinst-ok}\ \Sigma\ insts \equiv$$
$$\textit{distinct}\ (\textit{map}\ \textit{fst}\ insts) \wedge \textit{list-all}\ (\lambda((v, T), t).\ \textit{wf-term}\ \Sigma\ t \wedge \vdash_\tau t : T)\ insts$$

$$\textit{subst-term}\ \_\ (\textit{Ct}\ c\ T) = \textit{Ct}\ c\ T$$
$$\textit{subst-term}\ insts\ (\textit{Fv}\ idn\ T) = subst\_fv\ idn\ T\ insts$$
$$\textit{subst-term}\ \_\ (\textit{Bv}\ n) = \textit{Bv}\ n$$
$$\textit{subst-term}\ insts\ (\textit{Abs}\ T\ t) = \textit{Abs}\ T\ (\textit{subst-term}\ insts\ t)$$
$$\textit{subst-term}\ insts\ (t \cdot u) = \textit{subst-term}\ insts\ t \cdot \textit{subst-term}\ insts\ u$$

$$\textit{subst-bvs}\ s\ t = \textit{subst-bvs2}\ t\ 0\ s$$

$$\textit{subst-bvs2}\ (\textit{Bv}\ i)\ n\ us =$$
$$(\textit{if}\ i < n\ \textit{then}\ \textit{Bv}\ i$$
$$(\textit{else if}\ i - n < |us|\ \textit{then}\ us\ !\ (i - n)\ \textit{else}\ \textit{Bv}\ (i - |us|)))$$
$$\textit{subst-bvs2}\ (\textit{Abs}\ T\ t)\ n\ us = \textit{Abs}\ T\ (\textit{subst-bvs2}\ t\ (n+1)\ (\textit{map}\ (\lambda t.\ \textit{lift}\ t\ 0)\ us))$$
$$\textit{subst-bvs2}\ (f \cdot t)\ n\ us = \textit{subst-bvs2}\ f\ n\ us \cdot \textit{subst-bvs2}\ t\ n\ us$$
$$\textit{subst-bvs2}\ t\ \_\ \_ = t$$

References

1. Abrahamsson, O.: A verified proof checker for higher-order logic. J. Log. Algebraic Methods Program. **112**, 100530 (2020). https://doi.org/10.1016/j.jlamp.2020.100530
2. Adams, M.: HOL Zero's solutions for Pollack-inconsistency. In: Blanchette, J.C., Merz, S. (eds.) Interactive Theorem Proving. Lect. Notes in Comp. Sci., vol. 9807, pp. 20–35. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43144-4_2
3. Åman Pohjola, J., Gengelbach, A.: A mechanised semantics for HOL with ad-hoc overloading. In: Albert, E., Kovács, L. (eds.) LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning. EPiC Series in Computing, vol. 73, pp. 498–515. EasyChair, online (2020). https://doi.org/10.29007/413d
4. Barras, B.: Coq en coq. technical report 3026. Technical report, INRIA (1996)
5. Barras, B.: Verification of the interface of a small proof system in coq. In: Giménez, E., Paulin-Mohring, C. (eds.) Types for Proofs and Programs, pp. 28–45. Springer, Berlin (1998)
6. Berghofer, S., Nipkow, T.: Executing higher order logic. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) Types for Proofs and Programs (TYPES 2000). Lect. Notes in Comp. Sci., vol. 2277, pp. 24–40. Springer, Berlin (2002). https://doi.org/10.1007/3-540-45842-5_2
7. Berghofer, S., Nipkow, T.: Proof terms for simply typed higher order logic. In: Harrison, J., Aagaard, M. (eds.) Theorem Proving in Higher Order Logics. Lect. Notes in Comp. Sci., vol. 1869, pp. 38–52. Springer, Berlin (2000). https://doi.org/10.1007/3-540-44659-1_3
8. Carneiro, M.M.: Metamath zero: designing a theorem prover prover. In: Benzmüller, C., Miller, B.R. (eds.) Intelligent Computer Mathematics, CICM 2020. Lect. Notes in Comp. Sci., vol. 12236, pp. 71–88. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53518-6_5
9. Davis, J.: A self-verifying theorem prover. PhD thesis, The University of Texas at Austin (2009)
10. Davis, J., Myreen, M.: The reflective Milawa theorem prover is sound (down to the machine code that runs it). J. Automated Reason. **55**, 117–183 (2015). https://doi.org/10.1007/s10817-015-9324-6
11. Gheri, L., Popescu, A.: A formalized general theory of syntax with bindings: extended version. J. Automated Reason. **64**(4), 641–675 (2020). https://doi.org/10.1007/s10817-019-09522-2
12. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data refinement in Isabelle/HOL. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) Interactive Theorem Proving (ITP 2013). Lect. Notes in Comp. Sci., vol. 7998, pp. 100–115. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-39634-2_10

13. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) Functional and Logic Programming (FLOPS 2010). Lect. Notes in Comp. Sci., vol. 6009, pp. 103–117. Springer, Berlin (2010). https://doi.org/10.1007/978-3-642-12251-4_9
14. Haftmann, F., Wenzel, M.: Constructive type classes in isabelle. In: Altenkirch, T., McBride, C. (eds.) Types for Proofs and Programs, TYPES 2006. Lect. Notes in Comp. Sci., vol. 4502, pp. 160–174. Springer, Berlin (2006). https://doi.org/10.1007/978-3-540-74464-1_11
15. Harrison, J.: Towards self-verification of HOL Light. In: Furbach, U., Shankar, N. (eds.) In: Proceedings of the Third International Joint Conference, IJCAR 2006. Lect. Notes in Comp. Sci., vol. 4130, pp. 177–191. Springer, Seattle (2006). https://doi.org/10.1007/11814771_17
16. Huffman, B., Kunčar, O.: Lifting and transfer: a modular design for quotients in Isabelle/HOL. In: Gonthier, G., Norrish, M. (eds.) Certified Programs and Proofs, pp. 131–146. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03545-1_9
17. Hurd, J.: OpenTheory: package management for higher order logic theories. In: Reis, G.D., Théry, L. (eds.) Workshop on Programming Languages for Mechanized Mathematics Systems (ACM SIGSAM PLMMS 2009), pp. 31–37 (2009)
18. Journal of Automated Reasoning: Special Issue: Theory and Applications of Abstraction, Substitution and Naming. https://link.springer.com/journal/10817/volumes-and-issues/49-2
19. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Principles of Programming Languages (POPL), pp. 179–191. ACM Press, New York (2014). https://doi.org/10.1145/2535838.2535841
20. Kumar, R., Arthan, R., Myreen, M.O., Owens, S.: Self-formalisation of higher-order logic—semantics, soundness, and a verified implementation. J. Automated Reason. **56**(3), 221–259 (2016). https://doi.org/10.1007/s10817-015-9357-x
21. Kunčar, O., Popescu, A.: A consistent foundation for Isabelle/HOL. In: Urban, C., Zhang, X. (eds.) Interactive Theorem Proving, ITP 2015. Lect. Notes in Comp. Sci., vol. 9236, pp. 234–252. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22102-1_16
22. Kunčar, O., Popescu, A.: Comprehending Isabelle/HOL's consistency. In: Yang, H. (ed.) Programming Languages and Systems, ESOP 2017. Lect. Notes in Comp. Sci., vol. 10201, pp. 724–749. Springer, Berlin (2017). https://doi.org/10.1007/978-3-662-54434-1_27
23. Kunčar, O., Popescu, A.: Safety and conservativity of definitions in HOL and Isabelle/HOL. Proc. ACM Program. Lang. 2(POPL), 24-12426 (2018). https://doi.org/10.1145/3158112
24. Kunčar, O., Popescu, A.: A consistent foundation for Isabelle/HOL. J. Automated Reason. **62**(4), 531–555 (2019). https://doi.org/10.1007/s10817-018-9454-8
25. Lammich, P., Lochbihler, A.: The Isabelle collections framework. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving, ITP 2010. Lect. Notes in Comp. Sci., vol. 6172, pp. 339–354. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_24
26. Lochbihler, A.: Light-weight containers for Isabelle: efficient, extensible, nestable. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) Interactive Theorem Proving, ITP 2013. Lect. Notes in Comp. Sci., vol. 7998, pp. 116–132. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-39634-2_11
27. Nipkow, T., Klein, G.: Concrete Semantics with Isabelle/HOL. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10542-0
28. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic. Lect. Notes in Comp. Sci., vol. 2283. Springer, Berlin (2002). https://doi.org/10.1007/3-540-45949-9
29. Nipkow, T., Paulson, L.C.: Isabelle-91. In: Kapur, D. (ed.) Automated Deduction—CADE-11. Lect. Notes in Comp. Sci., vol. 607, pp. 673–676. Springer, Berlin (1992). https://doi.org/10.1007/3-540-55602-8_201
30. Nipkow, T., Roßkopf, S.: Isabelle's metalogic: formalization and proof checker. In: Platzer, A., Sutcliffe, G. (eds.) Automated Deduction—CADE 28, pp. 93–110. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_6
31. Nipkow, T., Snelting, G.: Type classes and overloading resolution via order-sorted unification. In: Hughes, J. (ed.) Proc. 5th ACM Conf. Functional Programming Languages and Computer Architecture. Lect. Notes in Comp. Sci., vol. 523, pp. 1–14. Springer, Berlin (1991). https://doi.org/10.1007/3540543961_1
32. Nipkow, T.: Order-sorted polymorphism in Isabelle. In: Huet, G., Plotkin, G. (eds.) Logical Environments, pp. 164–188. Cambridge University Press, Cambridge (1993)
33. Nipkow, T.: More Church-Rosser proofs (in Isabelle/HOL). J. Automated Reason. **26**, 51–66 (2001). https://doi.org/10.1007/3-540-61511-3_125
34. Nipkow, T., Prehofer, C.: Type reconstruction for type classes. J. Funct. Program. **5**(2), 201–224 (1995). https://doi.org/10.1017/S0956796800001325
35. Paulson, L.C.: Isabelle: A Generic Theorem Prover. Lect. Notes in Comp. Sci., vol. 828. Springer, Berlin (1994). https://doi.org/10.1007/bfb0030541

36. Paulson, L.C.: The foundation of a generic theorem prover. J. Automated Reason. **5**, 363–397 (1989). https://doi.org/10.1007/BF00248324
37. Pelletier, F.: Seventy-five problems for testing automatic theorem provers. J. Automated Reason. **2**, 191–216 (1986). https://doi.org/10.1007/BF02432151
38. Pfenning, F., Schürmann, C.: System description: Twelf—a meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) Automated Deduction, CADE-16. Lect. Notes in Comp. Sci., vol. 1632, pp. 202–206. Springer, Berlin (1999). https://doi.org/10.1007/3-540-48660-7_14
39. Pfenning, F.: Elf: a language for logic definition and verified metaprogramming. In: Logic in Computer Science (LICS 1989), pp. 313–322. IEEE Computer Society Press, Pacific Grove (1989)
40. Pientka, B.: Beluga: Programming with dependent types, contextual data, and contexts. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) Functional and Logic Programming, FLOPS 2010. Lect. Notes in Comp. Sci., vol. 6009, pp. 1–12. Springer, Berlin (2010). https://doi.org/10.1007/978-3-642-12251-4_1
41. Sozeau, M., Boulier, S., Forster, Y., Tabareau, N., Winterhalter, T.: Coq Coq correct! Verification of type checking and erasure for Coq. In: Coq. Proc. ACM Program. Lang. 4(POPL), 8-1828 (2020). https://doi.org/10.1145/3371076
42. Urban, C.: Nominal techniques in Isabelle/HOL. J. Automated Reason. **40**, 327–356 (2008). https://doi.org/10.1007/s10817-008-9097-2
43. Wenzel, M.: Type classes and overloading in higher-order logic. In: Gunter, E.L., Felty, A.P. (eds.) Theorem Proving in Higher Order Logics, TPHOLs'97. Lect. Notes in Comp. Sci., vol. 1275, pp. 307–322. Springer, Berlin (1997). https://doi.org/10.1007/BFb0028402