Isabelle's Metalogic: Formalization and Proof Checker*

Tobias Nipkow[0000−0003−0730−515X] and Simon Roßkopf[0000−0002−7955−8749]

Technical University of Munich, Germany

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.

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 [11] 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 [30]. 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 [25,24]. 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 just HOL but also ZF. Of course Paulson [29] describes $\mathcal{M}$ precisely, but only on paper. More importantly, his description does not yet cover polymorphism and type classes, which were introduced later [26]. The published account of Isabelle's proof terms [4] is also silent about type classes. Yet type classes are a significant complication (as, for example, Kunčar and Popescu [18] found out).

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 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 moderatly 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

* Supported by Wirtschaftsministerium Bayern under DIK-2002-0027//DIK0185/03 and DFG GRK 2428 ConVeY

performance. In particular, this includes multithreading, which is used in the kernel and has once lead to a soundness issue. 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.

The formalization (and more) has been submitted as supplementary material.

1.1 Related Work

Harrison [11] was the first to verify some of HOL's metatheory and an implementation of a HOL kernel in HOL itself. Kumar *et al.* [13] formalized HOL including definition principles, proved its soundness and synthesized a verified kernel of a HOL prover down to the machine language level. Abrahamsson [2] verified a proof checker for the OpenTheory [12] proof exchange format for HOL.

Wenzel [37] 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 [15,16,17,18] 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.* [1] formalize [15,18].

Adams [3] 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. Sozeau *et al.* [35] 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 [6] has 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.

We formalize a logic with bound variables, and there is a large body of related work that deals with this issue (e.g. [36,21,7]) and a range of logics and systems with special support for handling bound variables (e.g. [32,33,34]). 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* $p\ [x_1, \ldots, x_m]\ [y_1, \ldots, y_n] = (m = n \wedge p\ x_1\ y_1 \wedge \ldots \wedge p\ x_n\ y_n)$ and other self-explanatory notation.

The *Field* of a relation r is the set of all x such that $(x,\_)$ or $(\_,x)$ is in r.

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:

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

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

3 Types and Terms

A *name* is simply a string. Variables have type *var*; their inner structure is immaterial for the presentation of the logic.

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 class names. We discuss sorts in detail later.

Types (typically denoted by T, U, ...) are defined like this:

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

where $Ty\ \kappa\ [T_1,...,T_n]$ 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. 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 $term = Ct\ name\ typ \mid Fv\ var\ typ \mid Bv\ nat \mid Abs\ typ\ term \mid (\cdot)\ term\ term$

A term (typically r, s, t, u ...) can be a typed constant $Ct\ c\ T$ or free variable $Fv\ v\ T$, a bound variable $Bv\ n$ (a De Brujin index), a typed abstraction $Abs\ T\ t$ or an application $t \cdot u$.

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.

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

$$\frac{T \# Ts \vdash_\tau t : T'}{Ts \vdash_\tau \mathit{Abs}\ T\ t : T \to T'} \qquad \frac{Ts \vdash_\tau u : U \qquad Ts \vdash_\tau t : U \to T}{Ts \vdash_\tau t \cdot u : T}$$

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

Function $\mathit{fv} :: \mathit{term} \Rightarrow (\mathit{var} \times \mathit{typ})\ \mathit{set}$ collects the free variables in a term. Because bound variables are indices, $\mathit{fv}\ t$ is simply the set of all (v, T) such that $\mathit{Fv}\ v\ T$ occurs in t. The type is an integral part of a variable.

A *type substitution* is a function ϱ of type $\mathit{var} \Rightarrow \mathit{sort} \Rightarrow \mathit{typ}$. It assigns a type to each type variable and sort pair. We write $\varrho\ \$\$\ T$ or $\varrho\ \$\$\ t$ for the overloaded function which applies such 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 $\mathit{Abs}\ T\ t \cdot u$ to something like "t with $\mathit{Bv}\ 0$ replaced by u". We define a function $\mathit{subst\text{-}bv}$ such that $\mathit{subst\text{-}bv}\ u\ t$ is that β-contractum. The definition of $\mathit{subst\text{-}bv}$ is shown in the Appendix and can also be found in the literature (e.g. [23]).

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

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

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

4 Classes and Sorts

Isabelle has a built-in system of type classes [22] as in Haskell 98 except that class constraints are directly attached to variable names: our $Tv\ a\ [C,D,\ldots]$ corresponds to Haskell's `(C a, D a, ...) => ... 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 [28,27]. We give the first machine-checked formalization. The central data structure is a so-called *order-sorted signature*. Intuitively, it is comprised of a set of class names, 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 $osig = ((name \times name)\ set \times (name \rightharpoonup (class \rightharpoonup sort\ list)))$

To explain this formalization we start from a pair $(sub,tcs) :: osig$ and recover the informal order-sorted signature described above. The set of classes is simply the *Field* of the sub relation. The tcs component represents the set of all type constructor signatures $\kappa :: (Ss)\ c$ (where Ss is a list of sorts) such that $tcs\ \kappa = \mathsf{Some}\ dm$ and $dm\ c = \mathsf{Some}\ Ss$. Representing $\kappa :: (Ss)\ c$ as a triple, we define

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

TCS is the translation of tcs, the data structure close to the implementation, to an equivalent but more intuitive version TCS that is close to the informal presentations in the literature.

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 (sub,tcs), a type fulfills a given sort constraint:

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

$$\frac{tcs\ \kappa = \mathsf{Some}\ dm \qquad \forall\, c{\in}S.\ \exists\, Ss.\ dm\ c = \mathsf{Some}\ Ss \wedge \mathsf{list\text{-}all2}\ (\mathsf{has\text{-}sort}\ (sub,\ tcs))\ Ts\ Ss}{\mathsf{has\text{-}sort}\ (sub,\ tcs)\ (\mathsf{Ty}\ \kappa\ Ts)\ S}$$

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* $(sub,\ tcs)\ T_j\ S_j$ 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 $sub\ S = \{c \in S \mid \neg\ (\exists\, c'{\in}S.\ (c',\ c) \in sub \wedge (c,\ c') \notin sub)\}$
normalized-sort $sub\ S = (\mathit{normalize\text{-}sort}\ sub\ S = S)$

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 $(sub,\ tcs) = (\mathit{wf\text{-}subclass}\ sub \wedge \mathit{wf\text{-}tcsigs}\ sub\ tcs)$

A sublass relation is wellformed if it is a partial order where reflexivity is restricted to its *Field*. Wellformedness of type constructor signatures (*wf-tcsigs*) is more complex. We describe it in terms of TCS derived from tcs (see above). 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 [28,10].
  $\forall\,(\kappa,\ Ss_1,\ c_1){\in}TCS.$
  $\quad\forall\,c_2.\ (c_1,\ c_2)\in sub \longrightarrow$
  $\qquad(\exists\,Ss_2.\ (\kappa,\ Ss_2,\ c_2)\in TCS \wedge \mathit{list\text{-}all2}\ (\lambda S_1\ S_2.\ S_1 \leq_{sub} S_2)\ Ss_1\ Ss_2)$
- A type constructor must always take the same number of argument types:
  $\forall\,\kappa\ Ss_1\ c_1\ Ss_2\ c_2.$
  $\quad(\kappa,\ Ss_1,\ c_1)\in TCS \wedge (\kappa,\ Ss_2,\ c_2)\in TCS \longrightarrow |Ss_1| = |Ss_2|$
- Sorts must be normalized and must exists in sub:
  $\forall\,(\kappa,\ Ss,\ c){\in}TCS.\ \forall\,S{\in}\mathit{set}\ Ss.\ \mathit{wf\text{-}sort}\ sub\ S$
  where $\mathit{wf\text{-}sort}\ sub\ S = (\mathit{normalized\text{-}sort}\ sub\ S \wedge S \subseteq \mathit{Field}\ sub)$

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:

$\mathit{wf\text{-}osig}\ (sub,\ tcs) \wedge \mathit{has\text{-}sort}\ (sub,\ tcs)\ T\ S \wedge S \leq_{sub} S'$
$\longrightarrow \mathit{has\text{-}sort}\ (sub,\ tcs)\ T\ S'$

5 Signatures

A *signature* consist 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 $signature = (name \rightharpoonup typ) \times (name \rightharpoonup nat) \times osig$

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 is pretty obvious:

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

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

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: $T' \lesssim T$.

$$\frac{\mathit{wf\text{-}type}\ \Sigma\ T}{\mathit{wf\text{-}term}\ \Sigma\ (\mathit{Fv}\ v\ T)} \qquad \mathit{wf\text{-}term}\ \Sigma\ (\mathit{Bv}\ n)$$

$$\frac{\mathit{const\text{-}type}\ \Sigma\ s = \mathit{Some}\ T \qquad \mathit{wf\text{-}type}\ \Sigma\ T' \qquad T' \lesssim T}{\mathit{wf\text{-}term}\ \Sigma\ (\mathit{Ct}\ s\ T')}$$

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

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

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

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 $oss = (sub,\ tcs)$ is defined as follows:

wf-sig $\Sigma =$
$((\forall\, T{\in}\textit{ran}\ ctf.\ \textit{wf-type}\ \Sigma\ T) \wedge \textit{wf-osig}\ oss \wedge \textit{dom}\ tcs = \textit{dom}\ arf \wedge$
$(\forall\, \kappa\ dm.\ tcs\ \kappa = \textit{Some}\ dm \longrightarrow (\forall\, Ss{\in}\textit{ran}\ dm.\ arf\ \kappa = \textit{Some}\ |Ss|)))$

In words: all types in *ctf* are wellformed, *oss* is wellformed, the type constructors in *tcs* are exactly those that have an arity in *arf*, for every type constructor signature $(\kappa,\ Ss,\ \_)$ in *tcs*, κ has arity $|Ss|$.

6 Logic

Isabelle's metalogic $\mathcal{M}$ is an extension of the logic described by Paulson [29]. It is a fragment of intuitionistic higher-order logic. The basic types and connectives of M are the following:

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

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 favour 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 *theory* = *signature* × *term set*

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

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

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 Section 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 Section 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:

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

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

$$\mathit{wf\text{-}inst}\ (\Sigma,\ \mathit{axs})\ \varrho = \\ (\forall\, v\ S.\ \varrho\ v\ S \neq \mathit{Tv}\ v\ S \longrightarrow \mathit{has\text{-}sort}\ (\mathit{osig}\ \Sigma)\ (\varrho\ v\ S)\ S \wedge \mathit{wf\text{-}type}\ \Sigma\ (\varrho\ v\ S))$$

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 $\mathit{has\text{-}sort}\ \mathit{oss}\ (\mathit{Tv}\ v\ S)\ S$ 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 forall 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 $\equiv$ 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 of type *prop* (A and B) are equal iff they are logically equivalent:

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

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

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

Paulson [29] 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 Brujin 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.

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

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

Rule (β) uses the substitution function *subst-bv* as explained in Section 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 [37] 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 $\mathit{TYPE}_{T\ \mathit{itself}}$ is short for *Ct "type"* (T *itself*) where *"type"* is the name of a new uninterpreted constant. You should view $\mathit{TYPE}_{T\ \mathit{itself}}$ 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*) $\cdot$ $TYPE_{T\ \mathit{itself}}$ represents the statement "type T has class c". This is the inference rule deriving such propositions:

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

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:

Theorem 1. $\Theta,\Gamma \vdash t \longrightarrow \vdash_\tau t : \mathit{prop} \wedge \mathit{wf\text{-}term}\ (\mathit{sig}\ \Theta)\ t$

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 Proof Terms and Checker

Berghofer and Nipkow [4] 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 was unverified and checked 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. This is in contrast to the logic itself, which acts like a specification, which is why we presented in detail.

This is our data type of proof terms:

datatype *proofterm* = *PAxm* *term* (((*var* $\times$ *sort*) $\times$ *typ*) *list*) | *PBound* *nat*
| *Abst* *typ* *proofterm* | *AbsP* *term* *proofterm* | *Appt* *proofterm* *term*
| *AppP* *proofterm* *proofterm* | *OfClass* *typ* *name* | *Hyp* *term*

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. *PAxm* contains an axiom and a type substitution. This substitution is encoded as an association list instead of a function. *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 Brujin 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 the 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 [5,9,8].

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. The overall checker is a function

replay :: *theory* $\Rightarrow$ *proofterm* $\Rightarrow$ *term option*

In particular we need to make the inductive wellformedness checks for sorts, types and terms, signatures and theories executable. Mostly, this amounts to providing recursive versions of inductive definitions and proving them equivalent.

We now discuss some of the more difficult implementation steps. To model Isabelle's view of terms modulo $\alpha\beta\eta$-equivalence, we $\beta\eta$ normalize our terms (α-equivalence is for free thanks to De Brujin notation) during the reconstruction of the proof. A lengthy proof shows that this preserves provability (we do not go into the details):

wf-theory $\Theta \wedge$ *finite* $\Gamma \wedge (\forall A{\in}\Gamma.$ *wt-term* (*sig* Θ) $A \wedge \vdash_\tau A$: *prop*$) \wedge \Theta,\Gamma \vdash t \wedge$ *beta-eta-norm* $t =$ *Some* $u \longrightarrow \Theta,\Gamma \vdash u$

Isabelle's code generator needs some help handling the maps used in the (order-sorted) signatures. We provide a refinement of maps to association lists. Another 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. In another step, we refine the type substitution to an association list as well.

In the end we obtain a proof checker

check-proof Θ P p = (*wf-theory* $\Theta \wedge$ *replay* Θ P = *Some* p)

that checks theory Θ and 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. Function *check-proof* checks this claim. As one of our main results, we can prove the correctness of our checker:

Theorem 2. *check-proof* $\Theta\ P\ p \longrightarrow \Theta$,*set* (*hyps* P) $\vdash p$

The proof itself is conceptually simple and proceeds by induction over the structure of proof terms. For each proof constructor we need to show that the corresponding inference rule leads to the same conclusion as its functional version used by *replay*. Most of the proof effort goes into a large library of results about terms, types, signatures, substitutions, wellformedness etc. required for the proof, most importantly the fact that $\beta\eta$ normalization preserve provability.

8 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 10,000 LOC. The main components are:

- Almost half the formalization (4,700 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 derived 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.

9 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. For Isabelle code experts it should be mentioned that the two `term` constructors `Free` and `Var` in Isabelle (which both represent free variables but `Var` can be instantiated by unification) are combined in type *var* of the formalization which we left unspecified but which in fact looks like this: **datatype** *var* = *Free name* | *Var indexname*. This is purely to trivialize the glue code, in our formalization *var* is totally opaque.

Proof term translation is trivial except for two special cases. Previously proved lemmas become axioms in the translation (see also below) and 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). 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 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.

10 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 90% of the time. All tests were performed on a Intel Core i7-9750H CPU running at 2.60GHz and 32GB of RAM.

Logic	LOC	Time
FOL	4,500	45 secs
ZF	55,000	25 mins
HOL	10,000	26 mins

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 [31]. 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 the libraries of sets, functions, orderings, lattices and groups. The formalizations are non-trivial and makes heavy use of Isabelle's type classes.

Why can we check about five times as many lines of code in ZF compared to HOL? Profiling revealed that the proof checker spends a lot of time in functions that access the signature, especially the wellformedness checks. The primary reasons: inefficient data structures (e.g. association lists) and thus the running time depends heavily on size of signature and increases with every new constant, type and class. To make matters worse, there is no sharing of any kind in terms/types and their wellformedness checks. Because ZF is free of polymorphism and type classes, these wellformedness checks are much simpler.

11 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 el.* [13] but its code generator produces CakeML [14], not SML.
- The unverified glue code in the integration of our proof checker into Isabelle (Sect. 9).

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 [3] is concerned with.

12 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 [19,20]); 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 [5] 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 sets of theorems, explicitly checking cyclic dependencies (which are currently prevented by the glue code, see Sect. 9).

A presentation layer as discussed in Sect. 11 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 internal data structures.

It would also be nice to have a model-theoretic semantics for $\mathcal{M}$. We believe that the work by Kunčar and Popescu [15,16,17,18] could be adapted from HOL to $\mathcal{M}$. This would in particular yield semantically justified cyclicity checks for constant and type definitions which we currently treat as axioms because a purely syntactic justification is unclear.

Acknowledgements

We thank Kevin Kappelmann, Magnus Myreen, Larry Paulson, Andrei Popescu and Makarius Wenzel for their comments.

A Appendix

subst-bv $u\ t$ = *subst-bv2* $t\ 0\ u$

subst-bv2 (*Bv* i) $n\ u$ = (*if* $i < n$ *then Bv* i *else if* $i = n$ *then* u *else Bv* $(i - 1)$)
subst-bv2 (*Abs* $T\ t$) $n\ u$ = *Abs* T (*subst-bv2* $t\ (n + 1)$ (*lift* $u\ 0$))
subst-bv2 $(f \cdot t)\ n\ u$ = *subst-bv2* $f\ n\ u \cdot$ *subst-bv2* $t\ n\ u$
subst-bv2 $t\ \_\ \_$ = t

lift (*Bv* i) n = (*if* $n \leq i$ *then Bv* $(i + 1)$ *else Bv* i)
lift (*Abs* $T\ t$) n = *Abs* T (*lift* $t\ (n + 1)$)
lift $(f \cdot t)\ n$ = *lift* $f\ n \cdot$ *lift* $t\ n$
lift $t\ \_$ = t

bind-fv $T\ t$ = *bind-fv2* $T\ 0\ t$

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

References

1. 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 (2020), `https://easychair.org/publications/paper/9Hcd`

2. 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
3. Adams, M.: HOL Zero's solutions for Pollack-inconsistency. Lect. Notes in Comp. Sci., vol. 9807, pp. 20–35. Springer (2016), https://doi.org/10.1007/978-3-319-43144-4_2
4. 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 (2000)
5. 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 (2002)
6. 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 (2020), https://doi.org/10.1007/978-3-030-53518-6_5
7. Gheri, L., Popescu, A.: A formalized general theory of syntax with bindings: Extended version. J. Automated Reasoning **64**(4), 641–675 (2020), https://doi.org/10.1007/s10817-019-09522-2
8. 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 (2013)
9. 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 (2010)
10. 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 (2006), https://doi.org/10.1007/978-3-540-74464-1_11
11. Harrison, J.: Towards self-verification of HOL Light. In: Furbach, U., Shankar, N. (eds.) Proceedings of the third International Joint Conference, IJCAR 2006. Lect. Notes in Comp. Sci., vol. 4130, pp. 177–191. Springer, Seattle, WA (2006)
12. Hurd, J.: OpenTheory: Package management for higher order logic theories. In: Reis, G., Théry, L. (eds.) Workshop on Programming Languages for Mechanized Mathematics Systems (ACM SIGSAM PLMMS 2009). pp. 31–37 (2009)
13. Kumar, R., Arthan, R., Myreen, M.O., Owens, S.: Self-formalisation of higher-order logic — semantics, soundness, and a verified implementation. J. Automated Reasoning **56**(3), 221–259 (2016), https://doi.org/10.1007/s10817-015-9357-x
14. 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 (Jan 2014). https://doi.org/10.1145/2535838.2535841
15. 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 (2015), https://doi.org/10.1007/978-3-319-22102-1_16
16. 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 (2017), https://doi.org/10.1007/978-3-662-54434-1_27

17. Kunčar, O., Popescu, A.: Safety and conservativity of definitions in HOL and Isabelle/HOL. Proc. ACM Program. Lang. **2**(POPL), 24:1–24:26 (2018), https://doi.org/10.1145/3158112
18. Kunčar, O., Popescu, A.: A consistent foundation for Isabelle/HOL. J. Automated Reasoning **62**(4), 531–555 (2019), https://doi.org/10.1007/s10817-018-9454-8
19. 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 (2010), https://doi.org/10.1007/978-3-642-14052-5_24
20. 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 (2013), https://doi.org/10.1007/978-3-642-39634-2_11
21. Journal of Automated Reasonig: Special Issue: Theory and Applications of Abstraction, Substitution and Naming, vol. 49. Springer (Aug 2012), https://link.springer.com/journal/10817/volumes-and-issues/49-2
22. Nipkow, T.: Order-sorted polymorphism in Isabelle. In: Huet, G., Plotkin, G. (eds.) Logical Environments. pp. 164–188. Cambridge University Press (1993)
23. Nipkow, T.: More Church-Rosser proofs (in Isabelle/HOL). J. Automated Reasoning **26**, 51–66 (2001)
24. Nipkow, T., Klein, G.: Concrete Semantics with Isabelle/HOL. Springer (2014), http://concrete-semantics.org
25. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, Lect. Notes in Comp. Sci., vol. 2283. Springer (2002)
26. 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 (1992), https://doi.org/10.1007/3-540-55602-8_201
27. Nipkow, T., Prehofer, C.: Type reconstruction for type classes. J. Functional Programming **5**(2), 201–224 (1995)
28. 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 (1991)
29. Paulson, L.C.: The foundation of a generic theorem prover. J. Automated Reasoning **5**, 363–397 (1989)
30. Paulson, L.C.: Isabelle: A Generic Theorem Prover, Lect. Notes in Comp. Sci., vol. 828. Springer (1994)
31. Pelletier, F.: Seventy-five problems for testing automatic theorem provers. J. Automated Reasoning **2**, 191–216 (06 1986). https://doi.org/10.1007/BF02432151
32. 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 (1989)
33. 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 (1999), https://doi.org/10.1007/3-540-48660-7_14
34. 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 (2010), https://doi.org/10.1007/978-3-642-12251-4_1

35. 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), 8:1–8:28 (2020), https://doi.org/10.1145/3371076
36. Urban, C.: Nominal techniques in Isabelle/HOL. J. Automated Reasoning **40**, 327–356 (2008), https://doi.org/10.1007/s10817-008-9097-2
37. 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 (1997), https://doi.org/10.1007/BFb0028402