Formalizing the Metatheory of Logical Calculi and Automatic Provers in Isabelle/HOL

(Invited Paper)

Jasmin Christian Blanchette
Department of Computer Science
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
j.c.blanchette@vu.nl

Abstract
IsaFoL (Isabelle Formalization of Logic) is an undertaking that aims at developing formal theories about logics, proof systems, and automatic provers, using Isabelle/HOL. At the heart of the project is the conviction that proof assistants have become mature enough to actually help researchers in automated reasoning when they develop new calculi and tools. In this paper, I describe and reflect on three verification subprojects to which I contributed: a first-order resolution prover, an imperative SAT solver, and generalized term orders for λ-free higher-order logic.

1 Introduction
At programming language conferences such as POPL and ICFP, submissions are often accompanied by formalizations. Proof assistants are even used in the classroom to teach language semantics and type systems [68, 78]. Paradoxically, the automated reasoning community has largely stood on the sidelines of these developments.

Like the shoemaker’s children who go barefoot, we reflexively turn to “pen and paper”—by which we usually mean TEX—to define our logics, specify our proof systems, and establish their soundness and completeness. The automatic/interactive divide of our community is part of the reason. Few automatic prover developers have first-hand experience with a proof assistant. Nevertheless, it stands to reason that the members of the automated reasoning community should be early adopters of proof assistants. If we cannot convince these close colleagues of the value of tools called “theorem provers,” how are we going to seduce mainstream mathematicians, philosophers, and engineers?

The IsaFoL (Isabelle Formalization of Logic) effort1 aims at changing this situation. This “coalition of the willing” was inaugurated in 2015, initially as a Bitbucket repository that would enable Mathias Fleury in Saarbrücken and Anders Schlichtkrull in Copenhagen to carry out their respective Ph.D. projects while avoiding duplicated work. Their foresighted and well-funded bosses, Christoph Weidenbach and Jørgen Villadsen, have made this project possible.

My motto for the project is “Coq at POPL, why not Isabelle at CADE?” But in fact, Isabelle has been represented at CADE and IJCAR for several years, thanks to Tobias Nipkow, Lawrence Paulson, Christian Urban, and others. Moreover, René Thiemann, Christian Sternagel, and their colleagues have been using Isabelle to formalize term rewriting for over a decade. Their IsaFoR library directly inspired IsaFoL.2

Our main objective with IsaFoL is to develop libraries and a methodology to support modern research in automated reasoning, especially about propositional and first-order logic. Proof assistants can help when developing proofs, but also when modifying them, to study generalizations and variants. Reviewing becomes much easier when a formalization exists; reviewers can then focus on checking the definitions and theorem statements. Although my primary interest is in metatheory per se, once we have formal libraries, we can build verified provers and proof checkers on top of them.

The project has grown to include contributions from researchers and students in Amsterdam, Copenhagen, Gothenburg, Grenoble, Munich, Saarbrücken, and Zurich. Contributors include Heiko Becker, Alexander Bentkamp, Andreas Halkjær From, Alexander Birch Jensen, Peter Lammich, John Brunte Larsen, Julius Michaelis, Tobias Nipkow, Nicolas Peltier, Andrei Popescu, Simon Robillard, Sophie Tourret, Dmitriy Traytel, and Petar Vukmirović. The IsaFoL repository on Bitbucket is where much of the development takes place. Once individual entries have become mature enough, they tend to migrate to Isabelle’s Archive of Formal Proofs, which is continuously tested and updated to work with the most recent version of Isabelle. The maintenance burden largely falls onto the Isabelle developers, following the “you break it, you fix it” principle.

The use of Isabelle/HOL was initially motivated by personal preference, but it has turned out to be a fortunate choice. Isar proofs [99], locales [4], (co)datatypes [9], the code

1https://bitbucket.org/isafol/isafol
2 Font aficionados will notice the divergent preferences concerning serifs and kernings. Pronunciation is also crucial: to avoid confusion, Japanophiles should clearly articulate izaforu (IsaFoL) and izaf0 (IsaFoR).
At the time, I had little experience using Isabelle, so it was fortunate that Traytel agreed to act as my mentor.

The chapter had been haunting me as a tsundoku for a couple of years. Formalizing it meant I would finally have to take it from my reading pile and read it thoroughly. But there were also sound scientific reasons to choose this target for formalization, as we remarked later [87, Section 1]:

The text is a standard introduction to superposition-like calculi (together with Handbook Chapters 7 and 27). It offers perhaps the most detailed treatment of the lifting of a resolution-style calculus’s static completeness to a saturation prover’s dynamic completeness. It introduces a considerable amount of general infrastructure, including different types of inference systems (sound, reduc-tive, counterexample-reducing, etc.), theorem proving processes, and an abstract notion of redundancy. The resolution calculus, extended with a term order and literal selection, captures most of the insights underlying ordered paramodulation and superposition, but with a simple notion of model.

Traytel and I made considerable progress in two intense weeks, reaching the crucial Section 4.3. This is where the resolution calculus is lifted from ground (propositional) to nonground (first-order) clausal logic and where the RP prover is introduced and shown to be refutationally complete.

At the ground level, ordered resolution consists of the single (n + 1)-ary inference rule

\[
(C_i \lor A_1 \lor \cdots \lor A_n) \lor \neg A_1 \lor \cdots \lor \neg A_n \lor D
\]

with multiple side conditions that restrict the search space. This rule is refutationally complete, meaning that any unsatisfaction set that is closed under applications of the rule will contain the empty clause \( \bot \). A redundancy criterion identifies clauses and inferences that can safely be ignored; for example, the unit clause \( p(a) \lor q(b) \) redundant.

Next, Bachmair and Ganzinger introduce the concept of a theorem proving process: a transition system that starts with an initial clause set \( N_0 \) and where each transition \( \triangleright \) corresponds either to an inference or the removal of redundant clauses. Under a fairness assumption, the calculus’s completeness theorem characterizes the limit of a derivation \( N_0 \triangleright N_1 \triangleright N_2 \triangleright \cdots \).

Section 4.3 is where the trouble starts. For nonground clauses, the resolution rule takes the form

\[
(C_i \lor A_1 \lor \cdots \lor A_n) \lor \neg A_1 \lor \cdots \lor \neg A_n \lor D
\]

where \( \sigma \) is the most general unifier of the constraints \( A_1 \sigma \equiv \cdots \equiv A_n \sigma = A_i \). Ordering restrictions block inferences where \( A_i \sigma \) is smaller than \( C \sigma \) or \( D \sigma \). A literal selection mechanism further prunes the search space.

Traytel and I initially stopped here. By a stroke of good fortune, Schlichtkrull decided to resume the proof two years later.
later. After months of labor, and with expert help from Waldmann, he reached the final \texttt{qed}. The resulting IJCAR 2018 paper [87] was well received by the anonymous reviewers, reaping a “strong accept” and two “accepts.” One of the reviewers wrote:

The authors convinced me that their development is a great tool for exploring/developing calculus extensions. It will enable us to “extend/hack without fear.”

(The italics are mine throughout this paper.)

The \texttt{RP} prover is naturally formulated in Isabelle as an inductive predicate. Bachmair and Ganzinger’s transition rules correspond directly to introduction rules:

\begin{verbatim}
inductive \texttt{\sim} :: 'a state ⇒ 'a state ⇒ bool where
tautology_delete: Neg A ∈ C ∧ Pos A ∈ C ⇒ (\texttt{\not{\sim}} (N ∪ {C}, P, O)) ⇒ (\texttt{\sim} (N, P, O))
| forward_subsume: D ∈ P ∪ O ∧ subsumes D C ⇒ (\texttt{\not{\sim}} (N ∪ {C}, P, O)) ⇒ (\texttt{\sim} (N, P, O))
| backward_subsume_P: D ∈ N ∧ ssubsumes D C ⇒ (\texttt{\not{\sim}} N, P ∪ {C}) ⇒ (\texttt{\sim} (N, P, O))
| backward_subsume_O: D ∈ N ∧ ssubsumes D C ⇒ (\texttt{\not{\sim}} N, P, O ∪ {C}) ⇒ (\texttt{\sim} (N, P, O))
| forward_reduce: D ∈ P ∪ O ∧ reduces D C L ⇒ (\texttt{\not{\sim}} (N ∪ {C ∪ {L}}, P, O)) ⇒ (\texttt{\sim} (N ∪ {C}, P, O))
| backward_reduce_P: D ∈ N ∧ reduces D C L ⇒ (\texttt{\not{\sim}} (N, P ∪ {C ∪ {L}}, O)) ⇒ (\texttt{\sim} (N, P ∪ {C}, O))
| backward_reduce_O: D ∈ N ∧ reduces D C L ⇒ (\texttt{\not{\sim}} (N, P, O ∪ {C ∪ {L}})) ⇒ (\texttt{\sim} (N, P ∪ {C}, O))
| clause_process: (\texttt{\not{\sim}} (N ∪ {C}, P, O)) ⇒ (\texttt{\sim} (N, P ∪ {C}, O))
| inference_compute: (\emptyset, P ∪ {C}, O) ⇒ \texttt{\not{\sim}} (N, P, O ∪ {C})
\end{verbatim}

We faced various difficulties when formalizing Section 4.3. It did not help that the text contains dozens of small mistakes. Even the statement of the nonground resolution rule suffers from typos and ambiguities. While we agree with the reviewer who wrote that “most of us unconsciously autocorrect it, and read it with the intended meaning,” on several occasions we found ourselves blindly trusting the text, only to be disappointed later.

Reasoning about the \((n+1)\)-premise resolution rule was particularly tedious. Ellipsis is a convenient pen-and-paper device that lacks a counterpart in proof assistants. We ended up keeping the clauses \(C_i\) and atoms \(A_i\) = \{\(A_{i_1}, \ldots, A_{i_k}\)\} in parallel lists. Ideally, the first \(n\) premises of the rule would be regarded as a multiset; there is no need to consider \(!n\) inferences all yielding the same clause. However, there is no such things as “parallel multisets,” and our attempts at storing tuples \((C_i, A_i)\) in multisets only made matters worse.

Another general difficulty with the chapter was that it is not always clear which hypotheses apply where. The text both presents a general framework and applies it, but dependencies are not tracked precisely, and many lemmas are never invoked explicitly. It was a challenge to understand the informal proof well enough to organize the locales and state the definitions and lemmas precisely and correctly.

Finally, some of the arguments are incomplete or misleading. The proof of Lemma 4.11 relies on the observation that \(D'\) cannot be deleted by backward reduction or backward subsumption. However, in principle, \(D'\) could be deleted due to the presence of a more general clause \(D''\), which in turn could be deleted due to \(D'''\), and so on. The key missing observation is that this process can be iterated at most finitely many times, generalization being a well-founded relation.

With the locales, definitions, and lemmas in place, carrying out the proofs was mostly straightforward. We relied heavily on Sledgehammer to discharge the proof obligations.

We did find a significant mistake in the chapter. Theorem 4.13 states that \(\text{RP}\) is refutationally complete, but this does not hold due to the improper treatment of inferences containing duplicate premises. Remarkably, we discovered the mistake several months after reaching the final \texttt{qed}, while reviewing our definitions. We had inadvertently “autocorrected” Bachmair and Ganzinger’s definition of \(\text{RP}\).

Formalization helps track assumptions and dependencies precisely. It helps us answer questions such as, “Is Lemma 3.13 actually needed, and if so, where?” Indeed, such a question recently arose in the course of Bentkamp’s research on superposition [7]. He wanted to understand why the literal selection function \(S_M\) is defined so that \(\langle ii \rangle S_M(C) = S(C)\), if \(C\) is not in \(K\). He quickly got two replies. Waldmann wrote:

\begin{verbatim}
As far as I can see, \(S_M\) is really only needed for ground instances, and then case (ii) is irrelevant. I guess they just wanted to define \(S_M\) as a total function.
\end{verbatim}

Thanks to Isabelle, Schlichtkrull could be more confident:

I tried to change the definition in the formalization to return the empty multiset if \(C\) is not in \(K\). With this definition the above properties also hold, \textit{and the proof goes through}.

This anecdote nicely illustrates how formal proofs help generate knowledge and understanding. Here, they helped Bentkamp “extend/hack without fear.”

The second half of Bachmair and Ganzinger’s chapter, starting with Section 5, focuses on variations, such as nonstandard clauses and hyperresolution. Most of these are not implemented in modern provers, and we did not attempt to formalize this material. Instead, Schlichtkrull, Traytel, and I further refined the abstract prover \(\text{RP}\) to obtain an executable functional program. This work is described in a manuscript [86] that was originally written for an audience interested in the principles of programming languages.

We started by defining the inductive predicate \(\text{RP}_n\), which resembles \(\text{RP}\) but adds a timestamp to clauses and a weight function. Bachmair and Ganzinger [3, p. 44] mention this idea in a footnote, but they require the weight function to be monotone in both the timestamp and the clause size, claiming that this is necessary to ensure fairness. Although it often...
makes sense to prefer smaller clauses, our proof reveals that
this is not necessary to ensure fairness.

Next, we defined \( \text{RP}_d \) as a deterministic functional pro-
gram. \( \text{RP}_d \) simply calls the auxiliary function \( \text{RP}_d \) repeatedly until a final state (with \( N = P = 0 \)) is reached.
\( \text{RP}_d \) is a function of about 40 lines of code that is loosely
modeled after Vampire’s main loop [96]. To introduce this
possibly nonterminating function in Isabelle, we defined \( \text{RP}_d \)
by means of an option monad, using the \textbf{partial_function}
command [48], so that it returns a value of the form Some \( R \)
if the computation terminates and None otherwise.

Finally, we made \( \text{RP}_d \) executable by connecting it to IsaFoR,
which provides first-order terms and operations on them,
such as unification and the Knuth–Bendix order. We in-
voked Isabelle’s code generator [37] to export the prover to
Standard ML. The resulting program, \( \text{RP}_e \), consists of about
1000 lines of functional ML code, including dependencies.

After working hard to obtain an executable prover, we
evaluated it on a representative subset of 1000 TPTP bench-
marks against two leading provers, E and Vampire, as well as
Metis [44], which is written in Standard ML. The table below
gives the number of problems solved (proved or disproved)
by each prover [86]:

\[
\begin{array}{cccc}
\text{Vampire} & \text{E} & \text{Metis} & \text{RP}_e \\
834 & 770 & 527 & 353 \\
\end{array}
\]

Although our prover cannot yet challenge the state of the art,
it’s performance is respectable and could be improved further
using refinement. In his presentations, Natarajan Shankar
often stresses the view that

\[
\text{algorithm} = \text{inference} + \text{strategy} + \text{indexing}
\]

Indeed, the performance of an automatic proof procedure
comes largely from three sources: the calculus, the heuris-
tics, and the data structures. \( \text{RP}_e \) implements an excellent
calculus, but mediocre heuristics and data structures. In the
next section, we will look at a verification project that takes
these two aspects seriously.

The work on formalizing \( \text{RP} \) and \( \text{RP}_e \) opens exciting perspec-
tives. First, it paves the way for the formalization of
superposition provers. Pelletier [76] took us by surprise when
he announced, in 2016, his Isabelle formalization of the su-
perposition calculus. Based on his work, it should be possible
to derive an \( \text{SP} \) prover analogous to \( \text{RP} \), implementing most
of the simplification rules and heuristics described in, for
example, the E paper [89]. Verifying data structures such as
discrimination trees, feature-vector indices, and fingerprint
indices would pose interesting challenges. In unpublished
work, Vukmirović has verified the metatheory of fingerprints
for \( \lambda \)-free higher-order logic [97] using Isabelle.

Another perspective is to improve Bachmair and Ganz-
inger’s framework. In unpublished work, Waldmann, Tour-
ret, Robillard, and I have conceived, with pen and paper, a
framework that captures abstractly the lifting from a ground
calculus’s completeness result to a noground RP-like prover.
An Isabelle formalization is underway, which should culmi-
nate with a streamlined proof of Bachmair and Ganzinger’s
Theorem 4.13. The framework will also support infinitary in-
ferrations; these are useful for automating higher-order logic,
where the unification procedure may yield an infinite stream
of incomparable unifiers.

Around 2012, Vampire was extended with a SAT solver,
using a novel architecture called AVATAR [96]. The empirical
results were sensational, but a fundamental question was
left unanswered: is AVATAR refutationally complete? The
literature contains contradictory, erroneous definitions of
the architecture [11, 79, 96], but following discussions with
Giles Reger and his colleagues, I believe I have reached a
precise definition, for which refutational completeness holds,
while isolating a few potential sources of incompleteness in
Vampire. Furthermore, Vukmirović and I have shown that
labeled splitting [31], as implemented in SPASS, can be seen
as an instance of a slightly generalized AVATAR. I hope
that we can establish all of this formally in Isabelle, using the
new framework by Waldmann et al. described above.

3 A CDCL SAT Solver

As part of a 2015 M.Sc. internship in Saarbrücken, Fleury
started formalizing Weidenbach’s textbook draft, tentatively
called \textit{Automated Reasoning—The Art of Generic Problem Solv-
ing}. He continued as a Ph.D. student, focusing largely on
SAT solving and the conflict-driven clause learning (CDCL)
calculus implemented in most modern SAT solvers.

Inconveniently for us, there already existed several veri-
fied CDCL-based solvers [58, 60, 61, 70, 91]. We found a niche
by emphasizing the stepwise refinement methodology and
the connection between calculi variants, and by considering
some aspects that had not been the focus of formalization
before: clause forgetting, solver restarts, and incremental
solving. We hoped this would suffice to get our submission
accepted at IJCAR 2016; little did we expect to receive the
best paper award. In the jury’s words, our paper [14]
formalizes a modern SAT solver via a chain of refine-
ments in a proof assistant, contributing to the program
of formalizing highly-technical research in the field of
automated reasoning using tools developed in this field.

We considered both the abstract CDCL calculus described
by Nieuwenhuis, Oliveras, and Tinelli [67] and a more con-
crete, implementation-oriented calculus proposed by Wei-
denbach [98]. The calculi are represented by inductive predi-
cates on state tuples, roughly along the lines of the RP prover
described in the previous section.

The proofs are largely elementary, relying on basic results
about multisets and well-founded relations. We generally
found Nieuwenhuis et al.’s arguments easy to follow, with
the notable exception of a gap in their termination proof.
Weidenbach’s concise proofs were often more challenging.
to formalize. To give a flavor of his text, I quote a passage

By contradiction. Assume CDCL learns the same clause
twice, i.e., it reaches a state \((M, N, U, D \lor L)\) where Jump
is applicable and \(D \lor L \in N \lor U\). More precisely, the
state has the form \((K_n \cdots K_2 K_1 M_2 M_1, N, U, D \lor L)\)
where the \(K_i, i \geq 1\) are propagated literals that do not
occur complemented in \(D\), as otherwise \(D\) cannot be of
level \(i\). Furthermore, one of the \(K_i\) is the complement
of \(L\). But now, because \(D \lor L\) is false in \(K_n \cdots K_2 K_1 M_2
K_1 M_1\) and \(D \lor L \in N \lor U\) instead of deciding \(K_i^+\) the
literal \(L\) should be propagated by a reasonable strategy.
A contradiction. Note that none of the \(K_i\) can be
annotated with \(D \lor L\).

Fleury needed over 700 lines of Isabelle to capture this para-

As a proof of concept, Fleury implemented a deterministic
SAT solver and extracted functional Standard ML code from
it, preserving the formal guarantees established about CDCL:
soundness, completeness, and termination. The resulting
program was very inefficient; it could solve none of the 2009
SAT Competition problems in reasonable time.

This was good enough for me but not for Fleury. With
Lammich’s help, he proceeded to specify two watched liter-
al1s (2WL) and other imperative data structures [64], gradu-
ally departing from Weidenbach’s draft. This research is
described in a JAR article [12] and a CPP 2018 paper [32].
The 2WL scheme makes it possible to efficiently identify
candidate clauses for unit propagation and conflict detection,
which are the core CDCL operations. In each clause, the
solver distinguishes two watched literals—literals that can
possibly influence their clauses’ truth value in the solver’s
current state. The solver maintains the “2WL invariant” for
each clause. Unfortunately, the literature is imprecise about
the nature of this invariant and about when it is required to
hold. Fleury quickly found himself studying MiniSat’s source

CPP’19, January 14-15, 2019, Cascais/Lisbon, Portugal
Restarts is a technique that enables the solver to explore another part of the search space. We can keep completeness by gradually increasing the interval between restarts. Our calculi included an optional Restart rule all along to allow such behavior, but it was not implemented in the first IsaSAT.

Forgetting removes some learned clauses—consequences of the initial problem clauses that are derived during solving. The abstract CDCL calculus included a Forget rule from the start. The main difficulty is that each clause now needs to store a Boolean flag indicating whether it is deleted. This requires adding a header to the clause data structure for storing this and other useful information that guides the heuristics, and adapting all the refinement proofs.

Blocking literals are literals stored directly in the watch list, next to a pointer (or index) to the clause, that can be checked directly without dereferencing the pointer. The information is redundant, but it often saves a pointer dereference.

Machine (64-bit) integers are large enough to store clause indices and other numbers for the vast majority of SAT problems in practice, and they are more efficient than ML’s unbounded integers. To use them without losing the formal guarantees about the program, we generate two versions of the prover’s body and connect them with code of the form:

```c
while ¬ done ∧ ¬ overflow do
  (invoke the 64-bit version of the solver’s body);
if ¬ done then
  (convert the state from 64-bit to unbounded integers);
while ¬ done do
  (invoke the unbounded version of the solver’s body)
```

In a preliminary evaluation, Fleury ran IsaSAT against four other solvers on a collection of 3313 benchmark problems, consisting of all the SAT Competition problems from the 2009, 2011, 2013, 2014, 2016, and 2017 editions of the SAT Competition and the 2015 edition of the SAT-Race. The solvers Glucose [1] and CaDiCaL [10] represent the state of the art; MiniSat [30] is a well-known reference solver; and versat [70] is the only other efficient verified solver we know of. Since IsaSAT does not implement preprocessing techniques yet, CryptoMiniSat [92] was used to simplify all problems before benchmarking. The tests were run with a time limit of 30 minutes per problem. The table below shows the number of solved problems, whether satisfiable or unsatisfiable, for each system:

<table>
<thead>
<tr>
<th>Glucose</th>
<th>CaDiCaL</th>
<th>MiniSat</th>
<th>IsaSAT</th>
<th>versat</th>
</tr>
</thead>
<tbody>
<tr>
<td>1670</td>
<td>1645</td>
<td>1361</td>
<td>704</td>
<td>357</td>
</tr>
</tbody>
</table>

Given that competitive SAT solvers are extremely optimized programs, these results are very encouraging. However, we must bear in mind that it took two years of hard labor, and tens of thousands of Isabelle lines, to get from 0 to 704. An obvious avenue for future work would be to add optimizations such as pre- and inprocessing [45].

A benefit of having a verified SAT solver is that it can be employed as a backend in other verified tools. We have started looking into extending IsaSAT with theory reasoning, as in an SMT (satisfiability modulo theories) solver, with the goal of integrating it in CeTA [23], a verified safety and termination proof checker developed as part of IsaFoR.

## 4 Lambda-Free Higher-Order Terms Orders

The last subproject has a different flavor from the other two, in that the formalization arose as a side effect of carrying out research in automated reasoning and not as an end in itself. The Matryoshka project, which started in 2017 and funds a collaboration between Amsterdam, Nancy, Saarbrücken, and Stuttgart, aims at extending superposition provers and SMT solvers with higher-order features. As a stepping stone towards full higher-order logic, we started by focusing on a λ-free fragment. Unlike in first-order logic, variables may be applied, and function symbols may be given fewer arguments than they can take. This language is sometimes called “applicative first-order” in the term-rewriting community.

Bentkamp developed, under Waldmann’s and my supervision, a refutationally complete superposition calculus for λ-free higher-order logic and implemented it in a prototype prover developed with Cunanes [7]. He wrote his proofs directly in HeLaX, which was possible only because he is extremely rigorous and could count on two experts to check his proofs—namely, Waldmann and our colleague Tourret. In principle, he could have started from Peltier’s Isabelle formalization of superposition [76], but it seemed more difficult than working in HeLaX, especially given that he was more familiar with Waldmann’s informal proof, which has a different structure from Peltier’s.

Superposition relies on a well-founded term order. Together with Becker, Waldmann, and Wand, I designed two (families of) orders that generalize the familiar Knuth–Bendix order (KBO) and recursive path order (RPO). Becker formalized most of KBO during an internship; I completed his work and proceeded with RPO. This research was presented at Fossacs 2017 [21] and CADE 2017 [5].

The term orders are comparatively simple mathematical objects that lend themselves well to mechanization. I worked directly in Isabelle to define the orders and state their desired properties, starting with the ground case. The following fragment corresponds to my first attempt at defining a lexicographic path order (LPO), a special case of RPO, on ground terms:

```isabelle
datatype `c tm = F `c (`c tm list)

context
fixes <c:: `c ⇒ `c ⇒ bool
```


Formalizing the Metatheory of Calculi and Provers

assumes irreflp \lt\, and transp \lt\, and
\[ f \lt g \lor g \lt f \lor f = g \]

definition chksubs :: \('c \tm \Rightarrow \tm \Rightarrow bool' where
chksubs \, R \, s \, t \iff
\begin{align*}
\text{(case (s, t) of (F f ss, F g ts) \Rightarrow} \\
(\forall \, s' \in \, R \, s' \, t \lor \, s' = t) \land \\
(\, s \neq [] \land t \neq [] \land \text{last ss} = \text{last ts} \Rightarrow} \\
R \, (F \, f \, (\text{butlast ss}) \, (F \, g \, (\text{butlast ts}))))
\end{align*}

inductive \lt :: \('c \tm \Rightarrow \tm \Rightarrow bool' where
\[ s < t \lor s = t \implies s < F \, f \, (ts \, @ \,[t]) \]
| \[ s = F \, f \, ss \implies t = F \, g \, ts \implies s < t \iff \]
\[ F \, f \, (ss \, @ \,[u]) < F \, g \, (ts \, @ \,[u]) \]
| \[ s = F \, f \, ss \implies t = F \, g \, ts \implies f \lt g \implies \]
chksubs \, (<) \, s \, t \iff s < t
| \[ s = F \, f \, ss \implies t = F \, g \, ts \implies \text{chksubs} \, (<) \, s \, t \iff \]
lexordp \, (<) \, ss \, ts \iff s < t

lemma irrefl: \neg \, s < s
lemma trans: \, s < t \implies t < u \implies s < u
lemma total: \, s < t \lor t < s \lor s = t
lemma compfun: \, s < t \implies F \, f \, (ss \, @ \, s \, \cdot \, ss') < F \, f \, (ss \, @ \, t \, \cdot \, ss')
lemma comparg: \, F \, f \, ss < F \, g \, ts \iff F \, f \, (ss \, @ \, [s]) < F \, g \, (ts \, @ \, [s])

end

The specification is very short and depends on no background theory beyond list operations (cons \cdot, append \@, butlast, lexordp). It is a perfect match for my counterexample generator Nitpick [15]. And indeed, the tool quickly finds a counterexample to the irrefl conjecture; given a type \( 'c \) with two distinct symbols \( f, g \) such that \( f < \, g \), we have \( f \lt g \). Sadly, Nitpick’s rival Quickcheck [24] fails with the error “No type arity tm :: full_exhaustive.”

Using Nitpick, I was able to converge to an almost correct design. A major flaw had escaped my attention, namely: without arities or typing constraints, LPO is not well founded, because it allows infinite descending chains such as

\[ f \, b \leq f \, a \, b \leq f \, a \, a \, b \leq f \, a \, a \, a \, b \leq \ldots \]

This was noticed early on by Waldmann. Nitpick is helpless here, because it is based on finite model finding. It can be used to detect cycles but not acyclic divergence.

For the proofs, we drew our inspiration mostly from Baader and Nipkow’s textbook [2]. However, they cover only LPO and not RPO. When formalizing RPO, I faced a chicken-and-egg problem that took me several days to untangle. The issue is related to the multiset order, which is used to define RPO.

There exist two main formulations of the multiset order: Dershowitz–Manna [29] and Huet–Oppen [42]. They are equivalent on partial orders. Since RPO is a partial order, at first I chose Huet–Oppen, which we had used for KBO; however, until we have proved irreflexivity and transitivity of RPO, we cannot assume it is a partial order, so the choice between the two multiset orders is crucial. Zantema [101] was well aware of this, but I came across his work too late. And regardless, I could not think clearly while under the charm of the myth “Dershowitz–Manna = Huet–Oppen.”

As is often the case, once the main ideas were clarified, the formal proofs were straightforward to develop. According to generated logs, one some days I invoked Sledgehammer over 100 times. The first-order nature of most proof obligations (with the notable exception of well-foundedness) was a good match for automatic provers. The following Isar fragment, where each subproof (highlighted in gray) was produced by Sledgehammer, is fairly representative:

have arity_hd (head s) = 1
  by (metis One_nat_def arity.wary_AppE
dual_order.order_iff_strict eSuc_enat
enat_defs(1),2 ileI linorder_not_le not_iless0
wary_st wt_gt_\delta_if_supernary wt_s)
hence nargs_s :: num_args s = 0
  by (metis enat Ord_simps(2) less_one_nargs_lt
one_enat_def)
have s = hd (head s)
  by (simp add: Hd_head_id nargs_s)
them obtain f where
  f \in ground_heads (head s) and
  wt_sym f + the_enat (\delta \cdot \arity_sym f) = \delta
  using exists_wt_sym wt_s by fastforce

The first proof above relies on 12 named lemmas or local facts. Developing such a proof manually could easily have taken half an hour or more without the help of Sledgehammer (to find the proof) or the metis proof method (to reconstruct it), and we would have required searching for lemmas or memorizing their names.

In our two papers and the associated technical reports, we presented informal versions of the Isabelle proofs, for human consumption. This has been an opportunity to clean up and restructure the Isabelle proofs, to emphasize the important steps. We also used Nitpick to create examples to illustrate fine points in the papers [5, Example 12; 21, Example 9].

Given two orders \( \lt_1 \) and \( \lt_2 \), we could for example ask the tool to generate terms \( s, t \) such that \( s \lt_1 t \) but \( s \gt_2 t \).

Proof assistants really come into their own when we start modifying existing developments. Late in the project, we asked ourselves, “Could we generalize definition so-and-so in such-and-such a way?” After spending one hour with Isabelle, I was convinced the answer was yes, and a few hours later I was done repairing the proofs. There was very little to change in the technical report, but I would have had a hard time locating the passages that needed modifications and convincing myself that I had found them all.

The two papers received a lukewarm welcome. The term orders were not implemented in any termination tool, nor (at the time) in a superposition prover, and some related work
We used this opportunity to introduce a type of nested finite multisets and defined Dershowitz and Manna’s nested multiset order on it [29]:

\[
\text{datatype 'a nmultiset} = \\
\text{Elem 'a} | \text{MSet ('a nmultiset multiset)}
\]

This enabled us to finally give a positive answer to Paulson, who in 2014 had asked on the Isabelle mailing list:

I wonder whether anybody is aware of a formalisation (in any system) of the nested multiset ordering, as described in the classic paper “Proving Termination With Multiset Orderings”?

Using Isabelle’s Lifting and Transfer tools [43], we established a bijection between \( h\text{multiset} \) and the Elem-free fragment of \( 'a \text{nmultiset} \) and exploited it to lift definitions and properties. Notably, lifting the nested multiset order gives the familiar \(< \) operator on ordinals. The order’s well-foundedness proof can be transferred as well. Ordinal arithmetic operations such as addition and multiplication can be defined directly in terms of multiset operations.

Overall, we were able to quickly develop a versatile, practical library of syntactic ordinals, which we used not only for our TKBO variant but also in a formal proof of Goodstein’s theorem. This research was presented at FSCD 2017 [13].

5 Related Work

IsaFoL consists of many more subprojects beyond those described above. The first two listed below predate IsaFoL, but they are very much in its spirit and are mentioned on its web page. The entries are listed in rough chronological order:

- equisatisfiability of sort encodings for first-order logic, by Popescu and myself [16, 17];
- abstract soundness and completeness results for first-order logics using coinductive methods, by Popescu, Traytel, and myself [18–20];
- soundness and refutational completeness of first-order unordered resolution, by Schlichtkrull [84, 85];
- soundness and refutational completeness of a generalization of the superposition calculus, by Peltier [76];
- soundness and completeness of resolution-based prime implicate generation, by Peltier [75];
- metatheoretical results about a paraconsistent propositional logic, by Schlichtkrull and Villadsen [88, 94];
- soundness of a small-kernel first-order prover with equality described in Harrison’s textbook [40], by Jensen, Larsen, Schlichtkrull, and Villadsen [46, 47];
- correctness of an optimized tool chain for checking SAT solver certificates, by Lammich [54, 55];
- various metatheoretical results about a wide range of proof systems for classical propositional logic (sequent calculus, natural deduction, Hilbert systems, and resolution), by Michaelis and Nipkow [62, 63];
Formalizing the Metatheory of Calculi and Provers

- extensions of Berghofer’s formalization [8] of a first-order natural deduction calculus, by From [35];
- soundness of a substitutionless proof system for first-order logic, by From, Larsen, Schlichtkrull, and Villadsen [35];
- soundness and completeness of an epistemic logic with countably many agent, by From [34];
- modernization of Ridge and Margeson’s formalization [80, 81] of a sequent calculus for a term-free first-order logic, by Villadsen, Schlichtkrull, and From [95].

Formalizing metatheoretical results about logic, proof systems, and reasoning tools is an attractive proposition for many researchers in our field. Landmark achievements in the 1980s and 1990s include Shankar’s proof of Gödel’s first incompleteness theorem in Nqthm [90], Persson’s completeness proof for intuitionistic predicate logic in ALF [77], and Harrison’s formalization of basic first-order model theory in HOL Light [38].

Following Shankar’s 1984 proof, Gödel’s first incompleteness theorem has been formalized in Coq by O’Connor [69], in HOL Light by Harrison (in unpublished work), and in Isabelle/HOL by Paulson [72, 73]. Paulson also succeeded at verifying the second incompleteness theorem.

The completeness theorem for first-order logic has been mechanized many times in proof assistants. In Isabelle/HOL, Berghofer [8] proved the completeness of a natural deduction calculus, and Margeson and Ridge [80, 81] proved soundness, completeness, and cut-elimination of a sequent calculus for a term-free first-order logic. I refer to a recent article I wrote with Popescu and Traytel [20] for a discussion of such work.

Term rewriting is another popular target of formalization. The CoLoR library by Blanqui and Koprowski [22] and the CiME3 toolkit by Contejean et al. [26], both in Coq, as well as IsaFoR [93] in Isabelle, have explored this territory. They include not only formalized metatheory but also verified (non)termination and (non)confluence checkers built on it.

SAT solving also lends itself particularly well to formalization. Marić [59, 60] verified a CDCL-based SAT solver in Isabelle/HOL, including watched literals, as a purely functional program. He also formalized the abstract CDCL calculus by Nieuwenhuis et al. and, together with Janičić [61], the more implementation-oriented calculus by Krstić and Goel [49]. Lescuyer [58] formalized the CDCL calculus and the core of an SMT solver in Coq. Another verification of a CDCL-based SAT solver, including termination but excluding two watched literals, is by Shankar and Vaucher [91] in PVS. Impressively, most of this work was carried out by Vaucher during a brief internship. Finally, Oe et al. [70] used Guru to specify and verify versat. The generated C program consists of 15 000 lines of code. Optimized data structures are implemented, notably for watched literals. However, termination is not guaranteed, and model soundness is established through a run-time check.

A pragmatic approach to obtain trustworthy unsatisfiability judgments from a SAT solver is to have it produce a certificate, which can be given to an independent checker. An efficient format for this is DRAT [41]. The standard DRAT checker [100] is an unverified C program, but verified checkers have now been developed by Cruz-Filipe et al. [27] and Lammich [54] using ACL2, Coq, and Isabelle/HOL.

I alluded, in the introduction, to the self-referential thrill of formalizing theorem provers. Harrison [39] took this idea to its logical extreme when he verified, in HOL Light, HOL Light’s inference kernel. To circumvent the impossibility of defining higher-order logic’s semantics in itself, he carried out two distinct proofs: one where the formalized logic has no infinity axiom, and one where HOL Light is extended with an axiom to increase its strength. Harrison’s formalization was ported to HOL4 and extended by Kumar et al. [50] to include definitional mechanisms and to exploit CakeML [51], a verified ML environment. In another line of work, Davis built an ACL2-style prover called Milawa [28]. The development consists of a stack of provers, each used to verify the one above it. Together with Davis, Myreen [66] connected Milawa to a verified Lisp implementation [65] that was developed for hosting Milawa. A noteworthy feature of the prover is its switch command, which lets the user replace the inference kernel by an arbitrary kernel that has been proved sound, enabling powerful, highly optimized extensions that would be impossible using a traditional LCF architecture [36].

6 Conclusion

In this paper, I reported on some of the steps my colleagues and I have taken to help drive the adoption of proof assistants in the automated reasoning community. Far from following a definite plan, at every turn we focused on topics that appealed to us and for which we could perceive clear value in formalization. We have barely scratched the surface; a lot of exciting work still awaits us.

Automated reasoning is near ideal territory for proof assistants. Compared with other application areas, the proof obligations are manageable, and little background theory needs to be formalized before we can get started. Conveniently, researchers in the area are not afraid of logic, although they often lack familiarity with proof assistants and their higher-order formalisms.

Isabelle/HOL has been a very suitable vehicle for this kind of work, and we will continue using it. It is comparatively easy to use, has a simple but expressive logic, is based on a trustworthy LCF-style foundation, and includes rich libraries developed by a large, and growing, user base.

Acknowledgments

I am grateful to all contributors to the IsaFoR effort, listed in the introduction, and to our colleagues Maximilian Kirchmeier, Christian Sternagel, Uwe Waldmann, Daniel Wand,
The research has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). My work was also partially supported by a Deutsche Forschungsgemeinschaft (DFG) project (grant Ni 491/14-1, Hardening the Hammer).

The people who have contributed to this research and the Archive of Formal Proofs development team have made the work possible. I thank Assia Mahboubi and Magnus Myreen for the invitation to write in these pages and for their useful comments. Alexander Bentkamp, Mathias Fleury, Robert Lewis, Anders Schlichtkrull, Mark Summerfield, and Dmitriy Traytel have suggested many textual improvements. I should also acknowledge that some of the text above has appeared in a similar form in my earlier papers [12, 21, 32, 86, 87].

The research has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). My work was also partially supported by a Deutsche Forschungsgemeinschaft (DFG) project (grant Ni 491/14-1, Hardening the Hammer).

References

Formalizing the Metatheory of Calculi and Provers


CWP’19, January 14-15, 2019, Cascais/Lisbon, Portugal


