Virtualization of HOL4 in Isabelle

Fabian Immler
School of Computer Science, Carnegie Mellon University, USA
fimmler@cs.cmu.edu

Jonas Rädle
Fakultät für Informatik, Technische Universität München, Germany
raedle@in.tum.de

Makarius Wenzel
Augsburg, Germany
https://sketis.net

Abstract

We present a novel approach to combine the HOL4 and Isabelle theorem provers: both are implemented in SML and based on distinctive variants of HOL. The design of HOL4 allows to replace its inference kernel modules, and the system infrastructure of Isabelle allows to embed other applications of SML. That is the starting point to provide a virtual instance of HOL4 in the same run-time environment as Isabelle. Moreover, with an implementation of a virtual HOL4 kernel that operates on Isabelle/HOL terms and theorems, we can load substantial HOL4 libraries to make them Isabelle theories, but still disconnected from existing Isabelle content. Finally, we introduce a methodology based on the transfer package of Isabelle to connect the imported HOL4 material to that of Isabelle/HOL.

2012 ACM Subject Classification Software and its engineering → Interoperability; Theory of computation → Higher order logic

Keywords and phrases Virtualization, HOL4, Isabelle, Isabelle/HOL, Isabelle/ML

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.21

Supplement Material Our implementation is available online, it works with Isabelle2019 and the following development version of the official HOL4 repository.
https://github.com/immler/hol4isabelle
https://isabelle.in.tum.de/website-Isabelle2019
https://github.com/HOL-Theorem-Prover/HOL/commit/7e03303e51f

Funding *Fabian Immler*: This material is based upon work supported by the Air Force Office of Scientific Research under grant number FA9550-18-1-0120. Any opinions, finding, and conclusion or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force.

Acknowledgements Dagstuhl Seminar 18341 *Formalization of Mathematics in Type Theory* [2] hosted the working group "Interoperability of systems" (by Mario Carneiro), which provided an excellent environment to discuss early experiments and helped to structure this approach. We thank the HOL4 team (especially Michael Norrish) for support and accepting changes to HOL4.

1 Introduction: Interoperability of Theorem Provers

Suppose you chose Isabelle/HOL as your favorite theorem prover, like many other people did, e.g., in the Isabelle Archive of Formal Proofs (AFP) [3]. Unfortunately, by committing to one prover, you miss out on all of the great developments in others. For example, you cannot re-use the substantial work on the CakeML compiler [12], which is done in HOL4.

Interoperability between theorem provers, particularly HOL-based systems, has a long history (see also Section 7). But the problem has not been solved satisfactorily so far: none of the previous approaches has managed to import a huge project like CakeML in a scalable way and such that the result can be reused in a truly idiomatic manner in the target system.

Here we propose a novel and unorthodox approach to combine Isabelle/HOL and HOL4.

Main Ideas

We observe that HOL4 is designed with a modular and replaceable kernel, and that both provers are implemented in SML; Isabelle turns the underlying platform into a sophisticated environment of managed Isabelle/ML. The main ideas are:

- Run HOL4 inside the run-time environment of Isabelle/ML.
- Replace the kernel of HOL4 by a kernel that acts as a proxy to the kernel of Isabelle/HOL.
- Keep imported HOL4 libraries unchanged, without connections to existing Isabelle/HOL libraries at first.
- Connect theory content via Isabelle's `transfer` package.

Challenges: HOL4 versus Isabelle/HOL

We briefly review aspects of HOL4 and Isabelle that are relevant for combining them in a single run-time environment. Isabelle/Pure [17] is a generic logical framework (minimal higher-order logic), and Isabelle/HOL [15] a big library of Isabelle (classical higher-order logic with many add-on tools). HOL4 [19] is a proof assistant specifically for classical higher order logic. Both Isabelle and HOL4 are implemented in SML and run atop Poly/ML [14].

HOL4 provides a small abstract interface to its logical kernel (the modules for types, terms, and theorems); this opens the possibility to choose between different kernels implementations. Isabelle's inference kernel provides abstract types and constructors similar to the ones required by the kernel interface of HOL4.

The general idea of our approach seems straightforward: We replace the kernel of one HOL-based system with the kernel of another HOL-based system. But the implementation is not trivial, there are both conceptual and engineering difficulties to master.

The *conceptual difficulty* concerns differences in how HOL4 and Isabelle maintain logical declarations (i.e. update signatures of theories): HOL4 keeps a table of declared types and constants in a global mutable state variable that changes linearly over time. In contrast, Isabelle operates on a universal context as a purely functional value, following the DAG-structure of theories and the block-structure of proofs. We address this by providing an ML environment in which global state is virtualized inside Isabelle's universal context.

The *engineering difficulty* concerns details about the mapping of HOL4 inferences to Isabelle/HOL: they must conform precisely to the behavior expected from the HOL4 kernel interface. There are some further side-conditions, like implicit state in HOL4 terms, and different policies for names of variables and constants.

Contributions and Findings

We provide a working implementation of a virtual ML environment for HOL4 (Section 2), that is well integrated in Isabelle's Prover IDE (Isabelle/jEdit) and manages global state implicitly (Section 3). We present the implementation of a kernel for the virtual HOL4 that acts as a proxy to Isabelle/HOL theories, theorems, terms, and types (Section 4). We further propose a methodology (Section 5) to connect the imported HOL4 formalization to existing libraries in Isabelle/HOL and illustrate this idea on a small example.

Our measurements (Section 4.3) indicate that this approach of combining the two provers is worth continuing towards big applications: The performance losses in virtualized HOL4 and the proxy to Isabelle inferences are quite small, with a constant slowdown on most of the basis library of HOL4.

2 Virtualization in Isabelle/ML and ML Environments

Isabelle/ML is based on Poly/ML, but it isolates programs from low-level access to the compiler and run-time system: instead of direct mutation of the toplevel environment, there are functional updates on a universal context [21, §1.1]: it contains all logical declarations, add-on data for proof tools, and the ML environment (ML types, values, signatures, structures, functors, infixes). This managed environment of Isabelle/ML imposes restrictions on user-space programs, but allows pervasive parallelism with live editing of a running program in the Prover IDE, including implicit "undo" of ML toplevel declarations.

Such *virtualization* of ML is possible thanks to special operations provided by Poly/ML, most notably `PolyML.compiler`: it augments the running program with new declarations in the static ML environment, and new evaluations on the run-time heap (using native machine-code). This has been integrated with the universal context management of Isabelle for theories and proofs. Isabelle commands like `ML` or `ML_file` augment the environment according to the structure of theory documents in a thread-safe manner, i.e. ML code within independent theories (according to their DAG structure) and proofs (according to their block structure) can run in parallel without conflicts.

We have added a further dimension of *named ML environments*, to support other ML applications within Isabelle (notably HOL4): the ML function `ML_Env.setup` takes a fresh name and some operations to turn source text into ML tokens and (optional) antiquotations. There are two predefined ML environments: `Isabelle` refers to regular Isabelle/ML in the context of Isabelle/Pure (with some token syntax extensions and antiquotations), and `SML` refers to official Standard ML starting from the initial basis (without syntax add-ons).

The meaning of Isabelle commands like `ML_file` has been modified to depend on the context option `ML_environment`: it specifies the names of input and output environments in the form "env_1`>`env_2" (where "*env*`>`*env*" may be abbreviated by "*env*" alone). This allows to build up ML modules in independent name spaces, and to move material between them on demand. For example, the Isabelle/ML operation `writeln` for managed output of messages (with optional Prover IDE markup) can be made accessible in plain SML like this:[1]

```
declare [[ML_environment = "Isabelle>SML"]]
ML "val println = writeln"
```

That covers the static phase of ML declarations, but there is also the dynamic phase for program evaluation. To fit this into the purely functional context model of Isabelle, each ML execution context has a private (thread-local) variable to access its Isabelle context. The system provides the initial value, which may be changed by the running program via `Context.>>` of type `(context -> context) -> unit`. Afterwards, the system takes back the result. Thus old-style ML toplevel scripts with implicit mutation become plain functions on the context. Here is an example for Isabelle/ML:

[1] `TextIO.print` is available in the `SML` environment, too, but its output shows up on `stdout` and is not easily accessible to end-users in the Prover IDE. It is also not possible to "undo" such physical output.

```
fun declare_const name ty =
  Context.>> (Context.map_theory (Sign.add_consts [(Binding.name name, ty, NoSyn)]));
declare_const "c" propT;
declare_const "d" (propT --> propT);
```

As the effect of updating the context by `Context.>>` is managed by Isabelle, going back to an earlier situation in the theory means that the updates are still absent. So we get a form of implicit "undo", simply by returning to an old version of the (immutable) context.

3 ML Environment for HOL4

We use `ML_Env.setup` from Section 2 to define a fresh ML environment with the name "`HOL4`", hereafter referred to as the HOL4-environment.

▶ **Definition 1** (HOL4-environment). *The named ML environment "HOL4" in Isabelle.*

Subsequently we describe our setup of the HOL4-environment. Its initial basis is augmented to turn `ref` cells into variables managed in the Isabelle context (Section 3.1 and 3.2). Moreover, the lexical syntax is changed to support HOL4 quotations of terms and types (Section 3.3).

3.1 Global State

SML provides a special type `'a ref` for mutable reference cells pointing to values of type `'a`. (There is also the `Array` structure, for vectors of `ref` cells, but that is unused in HOL4.) The main operations on `'a ref` are `ref: 'a -> 'a ref` to initialize a new cell, `! : 'a ref -> 'a` to get the cell's content, and infix `:= : 'a ref * 'a -> unit` to change the cell's content. We imitate that in our structure `Context_Var`: it provides type `'a var` and operations `new`, `get`, `put` with analogous signatures. It is implemented via a data slot in the Isabelle context [21, §1.1.4], holding a map from integers to the universal sum type in SML. The operation `new: 'a -> 'a var` allocates a new index in the table and provides type-safe injections and projections for stored values of the particular type `'a`.

Now the meaning of HOL4 programs shall be changed to refer to this managed variable space, whenever `'a ref` operations are seen, but this type cannot be redefined in SML user-space. Since we manage the ML environment anyway, we simply map the name "`ref`" for types and values to our counterparts `Context_Var.var` and `Context_Var.new`, respectively. The other operations can be redefined by conventional declarations in SML.

A remaining problem is the use of `ref` as a datatype constructor in pattern matching, instead of `!` as selector. To keep our language manipulation simple, we eliminated the (rare) uses of that feature of SML in the HOL4 repository: there was no problem with rewriting, e.g., `fun lookup (ref v) = v` manually to `fun lookup r = !r`.

In summary, the module `Context_Var` manages the overall state of *all* context variables of the running ML program: henceforth it can represent the global "mutable" application state within the HOL4-environment.

▶ **Definition 2** (HOL4-state). *The value of the table in* `Context_Var` *in a given universal context is called the* HOL4-state*.*

3.2 Local State

The above approach, which maps all uses of `ref` to `Context_Var`, is functionally correct, but there is a catch regarding performance and memory consumption. Conventional `ref` cells are subject to garbage collection in ML, and do not need an explicit operation to free allocated memory. In contrast, variables that were allocated once via `Context_Var.new` remain accessible in the context and will not be garbage-collected automatically.

This is particularly problematic for strictly local program variables, i.e., reference cells that are private to particular function invocations, and typically used to simplify or speed up the implementation via imperative features. Such ad-hoc variables do not survive termination of the function, and are better not made persistent in the Isabelle context.

Following this observation, we distinguish *local state* variables from global state variables that are managed in the context. This works by marking the variables in the original HOL4 sources, using the type `Uref.t` that is merely a clone of `ref` in HOL4. Now we can distinguish the two kinds of references in the virtualized Isabelle environment: unmarked `ref` becomes our `Context_Var.var`, and `Uref.t` becomes `Unsynchronized.ref` of Isabelle/ML [21, §0.7.9].

How to distinguish the two kinds of variables in the vast body of HOL4 sources? Our pragmatic approach is based on run-time profiling. For each static occurrence of `Context_Var` its number of dynamic allocations is reported. Then we inspected the worst "memory leaks" to judge their role in the program: the result is recorded in the official HOL4 sources.

3.3 Quotation Filter

HOL4 extends SML syntax with *quotations* to allow embedding of logical entities (types and terms) with their own syntax into ML.[2] HOL4 quotations come in different forms, we will illustrate the concept only for terms. A *term quotation* consists of a string delimited by two single back-quotes, e.g., ``` ``string`` ```. The HOL4 `QuoteFilter` expands this to ML source `Parse.Term [QUOTE "string"]`. It means that the string is parsed at run-time at the spot where this is inlined into the ML source, using the syntax of the implicit theory.

Our HOL4-environment in Isabelle should support quotations, too, so we include the `QuoteFilter` directly into it. We also want to use the rich capabilities of the Isabelle Prover IDE: this requires original source positions passed on to the generated ML text. The Isabelle setup of `PolyML.compiler` (Section 2) turns results from static analysis by the ML compiler into markup that Isabelle/jEdit presents to the user as colors, popups, hyperlinks etc.

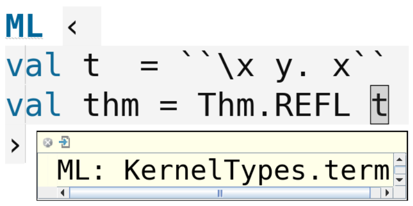
`QuoteFilter` is implemented with the lexical analyzer generator ML-Lex [1]. We made minor modifications to that in the HOL4 repository, to have it return precise position information (together with the expanded text). Consequently, the Isabelle/HOL4 source text provides ML IDE annotations both for SML and the result of inlined quotations (but not inside the HOL4 term language, unlike Isabelle). The example in Figure 1 illustrates this Prover IDE experience: The embedded HOL4 term `t` has ML type `KernelTypes.term`; this can be inspected by hovering with the mouse cursor over `t`.

3.4 Implicit "Undo" of Changes in HOL4-state

We can now illustrate the advantage of implicit "undo" with the HOL4-state. Assume you declare a constant `foo`, realize that you spelled it wrong and want to redeclare it as `fop`. When interacting with the HOL4 toplevel, you need to keep the global state of constants in mind, delete `foo` (make it inaccessible in the term language), and introduce `fop` like this:

```
> Theory.new_constant("foo", Type.bool);
> Theory.delete_const "foo";
> Theory.new_constant("fop", Type.bool);
```

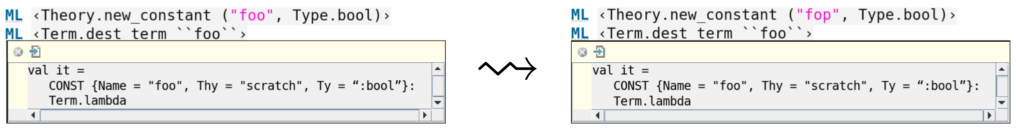
[2] This is similar to *antiquotations* in Isabelle, but the terminology is reversed, because the outer source is the Isabelle theory and proof language, which may *quote* ML, which may *antiquote* the term language.



```
ML ‹
val t  = ``\x y. x``
val thm = Thm.REFL t
›
ML: KernelTypes.term
```

Figure 1 A Quotation in the HOL4-environment and IDE markup in Isabelle/jEdit.

With virtual state management by Isabelle, however, it suffices to edit the document, changing `foo` to `fop` and the previous declaration of `foo` simply disappears (see Figure 2).



```
ML ‹Theory.new_constant ("foo", Type.bool)›
ML ‹Term.dest_term ``foo``›
val it =
   CONST {Name = "foo", Thy = "scratch", Ty = “:bool”}:
   Term.lambda
```

$\rightsquigarrow$

```
ML ‹Theory.new_constant ("fop", Type.bool)›
ML ‹Term.dest_term ``foo``›
val it =
   CONST {Name = "foo", Thy = "scratch", Ty = “:bool”}:
   Term.lambda
```

Figure 2 Implicit “undo” of operations on virtual state in HOL4 in Isabelle/jEdit. After editing `foo` to `fop`, `fop` is no longer registered as a constant `CONST`, but is parsed as a variable `VAR`.

3.5 The Standard Kernel in the Virtualized Environment

To test that everything is properly set up, we load the original HOL4 sources (with the standard kernel) in the previously described HOL4-environment. So in this case, HOL4 can be seen as an isolated application running in the HOL4-environment, without any connection to the logic of Isabelle/HOL whatsoever. We did this for the following build-sequences:

- `core`: e.g., natural numbers, datatypes, lists, and bossLib
- `more`: e.g., integers, topology, n-bit vectors
- `large`: e.g., real numbers, probability theory, temporal logic, floating point numbers

Figure 4 shows performance figures of virtualized HOL4 vs. original HOL4. Broadly summarized, virtualized HOL4 is about 1.5 times as slow.

4 An Isabelle/HOL Kernel for HOL4

Now that we have demonstrated that the HOL4-environment provides a suitable environment to run HOL4 in Isabelle, let us take a look at what is required to have HOL4 produce actual Isabelle/HOL theorems. This requires an implementation of the actual logical kernel, i.e., modules for types, terms, and theorems, which we describe in Section 4.1. But it also requires modifications to the theory management of HOL4 to properly integrate in the virtualized HOL4-environment with Isabelle's theory management, which we describe in Section 4.2. We report on performance measurements in Section 4.3.

4.1 Logical Kernel

HOL4 and Isabelle/HOL both follow the LCF-approach, which means they define an abstract datatype of theorems with inference rules as (type-safe) operations [4]. Note that Isabelle/HOL's types, terms, and theorems are actually implemented in Isabelle/Pure.

The *HOL4-Kernel* is the collection of ML modules for types, terms, and theorems in HOL4. HOL4 prescribes an interface (ML signatures) to the HOL4-Kernel, which makes it possible to select different implementations of the HOL4-Kernel at compile time. HOL4 comes with a *standard* HOL4-Kernel, where terms are represented with de-Bruijn indices and explicit substitutions, as well as an *experimental* HOL4-Kernel with named bound variables.

In the subsequent implementation of our *Isabelle* HOL4-Kernel, types, terms, and theorems of the HOL4-Kernel interface are implemented by their counterparts in Isabelle/Pure.

4.1.1 Types

Besides constructor names, Isabelle's type language only differs from the standard HOL4-Kernel in that it is many-sorted and includes schematic type variables. Therefore, all types produced by our kernel have the base sort `HOL.type` and do not include schematic type variables. So the type structure is mainly a copy of the standard HOL4-Kernel with constructors replaced and some small adaptations.

The Isabelle/Pure theorem module does not use these (unchecked) types directly, but rather operates on values of the abstract type `ctyp`, which represents types that are well-formed wrt. some theory. Certifying types is an expensive operation, and since the Isabelle HOL4-Kernel should never need to produce malformed types, it would be beneficial to use certified types as our underlying type representation.

But unfortunately this is impossible, because the HOL4-Kernel (and subsequently all of HOL4) requires that `hol_type` is an ML equality type. Isabelle/Pure's abstract type `ctyp` does not satisfy this requirement and it is unrealistic to remove this requirement from the HOL4-Kernel.

There are two occurrences in the Isabelle HOL4-Kernel where certification of types is necessary: The first occurrence is instantiation of type variables, `INST_TYPE` in HOL4 that maps to `Thm.instantiate_frees`). In this case it is reasonable to expect that the size of these types is small, so that we can ignore this fine point. The second occurrence is construction of variables `Term.mk_var : (string * hol_type) -> term`. Variables are constructed so frequent that re-certification can be prohibitively expensive. We work around this by introducing a cache where already certified types can be looked up.

4.1.2 Terms

The HOL4-Kernel interface fixes an abstract interface to well-typed terms. In Isabelle/Pure, well-formed and well-typed terms are an abstract subtype `cterm` (for certified `term`) of a datatype of preterms[3]. Figure 3 compares the representation of `preterms` in Isabelle/Pure and terms in the standard HOL4-Kernel. Constants `Const` and free variables `Free/Fv` are constructed from a name and a type, bound variables `Bound/Bv` are represented with de-Bruijn indices. In Isabelle/Pure, λ-abstraction `Abs` takes information on how to display the bound variable with a string, the type of the bound variable, and a body. The standard HOL4-Kernel maintains the invariant that `Abs` only occurs with a free variable `Fv` as first argument, thereby representing the same information as Isabelle/Pure. Function application of function `f` and argument `x` is written as infix operation `f $ x` or combination `Comb`. `Var` in Isabelle represents schematic (unifiable) variables, but this is not needed for HOL4.

[3] This actually is the datatype `term` in Isabelle, but to avoid confusion, we call it `preterm` here.

```
datatype preterm =
  Const of string * typ
| Free  of string * typ
| Bound of int
| Abs   of string * typ * preterm
| $     of preterm * preterm
| Var   of indexname * typ
```

(a) Preterms in Isabelle/Pure.

```
datatype term =
  Const of kernelid * hol_type
| Fv    of string * hol_type
| Bv    of int
| Abs   of term * term
| Comb  of term * term
| Clos  of term Subst.subs * term
```

(b) Internal terms in the standard HOL4-Kernel.

Figure 3 Different term representations in Isabelle and HOL4.

HOL4's standard kernel has an explicit constructor `Clos` for closures, terms with an environment attached to them. For rewriting-heavy applications (e.g., the CakeML bootstrapping), `Clos` might be performance-critical. Nevertheless, we decided to ignore this feature for the moment, because e.g., the experimental kernel does not feature closures, either. Should one wish to achieve the same asymptotic complexity for rewriting with explicit closures, one could add a special Let-construct to Isabelle/HOL. Pattern-matches on `preterms` in the Isabelle HOL4-Kernel would then need to introduce a special case that tests for the presence of this special Let constant (just like the standard HOL4-Kernel has a special case for `Clos`).

While the Isabelle/Pure interface to `cterm` is rather minimal, it exposes some primitives for building abstractions and applications without having to re-certify the result. This allows us to base our implementation of the HOL4-Kernel term structure on certified terms, avoiding the expensive operation of certification as much as possible.

One slight complication was that, at the beginning of our work, the type of terms was declared as an `eqtype` in the HOL4-Kernel signature, and thus could not be instantiated with an abstract type. The HOL4 developers had already started to work (with an independent motivation) towards removing the `eqtype` constraint, and this removal is now completed.

We encountered another problem: the HOL4-Kernel sometimes produces malformed terms, in particular terms containing loose bound variables. These terms result from `break_abs`, which destructs an abstraction without turning the variable bound under the abstraction into a free variable. We work around this by using free variables with special names to represent the loose variables. These uniquely named variables are also useful to efficiently destruct abstractions in the regular way (i.e. by turning the bound variable into a free variable), which may otherwise involve renaming, and to ensure that Isabelle/Pure does not rename variables using its own convention, which is different from that of the HOL4-Kernel.

We cannot pattern-match on the abstract type `cterm` but would like our implementation to stay close to that in the standard HOL4 kernel, which heavily uses pattern matching on terms. We therefore match on the underlying `preterm` of a `cterm` and carry out the actual operation on the result of destructing the `cterm` according to the matched pattern. To illustrate this technique, here is part of the implementation of the operation `trav`, which applies a function `f : cterm -> unit` to all constants and free variables in a term.

```
fun trav f ct =
  let fun trv (Free _) ct = f ct
        | trv (Rator $ Rand) ct =
          let val (cRator,cRand) = dest_comb ct
          in (trv Rator cRator ; trv Rand cRand)
          end
  ...
  in trv (preterm_of ct) ct end
```

Here we access the underlying preterm of `ct` using the Isabelle/Pure function `preterm_of`, which is cheap, and then give both to the internal function `trv`, which pattern matches on the term and either applies `f` at the appropriate places or destructs the `cterm` according to the matched pattern with `dest_comb`.

4.1.3 Axiomatization of HOL

The HOL4-Kernel axiomatizes higher order logic. For the Isabelle HOL4-Kernel, we obviously do not want to add new axioms, but rather map calls that axiomatize to existing constants, axioms, and theorems in Isabelle/HOL.

The HOL4-Kernel has builtin type operators for functions `->`, booleans `bool`, and an inductive type `ind`, which we map directly to the corresponding type operators from the axiomatization of Isabelle/HOL for functions ⇒, booleans *bool*, and an inductive type *ind*.

The HOL4-Kernel has builtin constants for equality `= : 'a -> 'a -> bool`, Hilbert choice `@ : ('a -> bool) -> 'a`, and implication `==> : bool -> bool -> bool`. The axiomatization of Isabelle/HOL also contains equality *(=) :: 'a ⇒ 'a ⇒ bool*, implication *(⟶) :: bool ⇒ bool ⇒ bool*, and Hilbert choice *Eps :: ('a ⇒ bool) ⇒ 'a*, so we can map to those directly, as well.

The HOL4-Kernel introduces axioms for defined constants (`T`, `F`, `ONE_ONE`, `ONTO`):

```
("BOOL_CASES_AX", ``!t. (t=T) \/ (t=F)``)
("ETA_AX",        ``!t:'a->'b. (\x. t x) = t``)
("SELECT_AX",     ``!(P:'a->bool) x. P x ==> P ($@ P)``)
("INFINITY_AX",   ``?f:ind->ind. ONE_ONE f /\ ~ONTO f``)
```

Without extending the set of axioms in Isabelle/HOL, we map those axioms to theorems that we proved explicitly in Isabelle/HOL.

4.1.4 Theorems

The HOL4-Kernel theorem module modifies the internal representation of theorems in a soundness-critical way. In contrast to that, the Isabelle/Pure-based implementation simply defers operations to (trusted) Isabelle/Pure primitives.

Usually, Isabelle/Pure inferences cannot be used directly, but require some adaptation of interfaces. This is because of the distinction between meta-logic Isabelle/Pure and object-logic Isabelle/HOL. For example, the HOL4-Kernel primitive `MK_COMB : thm -> thm -> thm` is supposed to yield a theorem `f x = g y` from theorems `f = g` and `x = y`. Isabelle/Pure provides a similar inference `Thm.combination : thm -> thm -> thm`, but for meta-equality. I.e., it produces $f\ x \equiv g\ y$ from theorems $f \equiv g$ and $x \equiv y$. The Isabelle/HOL axiomatization states that HOL-equality *(=)* reflects Pure-equality *(≡)*, so we can insert inference steps to convert theorems with Pure-equality to (and from) theorems with HOL-equality.

Overall, we do not use the builtin unification of Isabelle/Pure, but always compute explicit instantiations, hoping that this is the most efficient implementation.

4.2 Theory Management

HOL4 uses the dedicated Holmake tool to manage dependencies of source files. Moreover, it can compile theory files from script files: this requires special attention when trying to incorporate this in the HOL4-environment and cooperate with the Isabelle HOL4-Kernel.

4.2.1 HOL4-Scripts and HOL4-Theory Files

Theories in HOL4 are structured along a concept called *theory segments*. A segment records logical declarations like types, constants, and theorems, together with pointers to parent segments. The theory represented by a segment is the union of all the logical declarations of the segment and its parents. A theory segment is constructed in different stages:

- One starts from a so-called *script*. A script contains all the ML-declarations that define types, constants, prove theorems, or e.g., augment syntax.
- When compiling a script, all changes that the script makes w.r.t. to the current (logical) theory are recorded and saved in a special file-format. Compilation of a script will also generate *theory files*.
- Theory files are ML modules that contain all the information required to load the recorded changes and apply them to the current (logical) theory.

Note that re-importing a HOL4 theory from the file-system does *not* reconstruct theorems by kernel inferences, instead it trusts the imported statement with an oracle. We do not want to reproduce this part of the workflow in Isabelle/HOL, in particular because we do not want to increase the trusted code base by theorem import (and essentially all of HOL4).

Instead we remember the theorem values that are created when running the script. HOL4 offers a hook that allows users to register custom code to be called upon exporting a theory, and we use this to store the theorems as abstract ML values in the universal Isabelle context.

We provide a small wrapper around HOL4's module that reads theories from the file-system. This wrapper does not assume theorems via an oracle, it rather looks up the theorem values that were previously stored in the Isabelle context.

In the HOL4 system, scripts are run in a separate process, and the only artifacts that they produce are the generated theory files. This means that in our case, after running a script, the HOL4-state needs to be reset: the script modified the underlying HOL4 theory, but these changes are not supposed to persist. In order to do so, we simply remember the HOL4-state before compiling the script in Isabelle/ML and update the HOL4-state afterwards to the previously remembered state.

4.2.2 Holmake

Holmake is the main tool to manage dependencies of script files, theory files and other ML files for HOL4. Upon invocation in a directory containing HOL4 source files, Holmake computes dependencies between files, and compiles and runs plain ML code, proof scripts, and generated theory files.

We can compile and run Holmake in the `HOL4` ML environment, but its overall setup is – due to the previous discussion about storing theorems when running scripts – too alien for us to use it in the HOL4-environment. Instead, we write custom code that emulates the behavior of Holmake reasonably well. Our emulation builds on a part of Holmake, the *Holdep* tool. From the output of Holdep, we recurse over the dependencies. With additional dependencies for Theory files (they depend on Script files), our emulation is sufficiently close so that it "does the right thing"[4] for a large part of HOL4's source directories.

On several occasions, HOL4 saves a "heap", i.e., the global state of the Poly/ML process after loading a number of SML modules and theories. In our virtual environment, this simply amounts to keeping the HOL4-state instead of resetting it to the previous value.

[4] Quoting a comment in the implementation of Holmake.

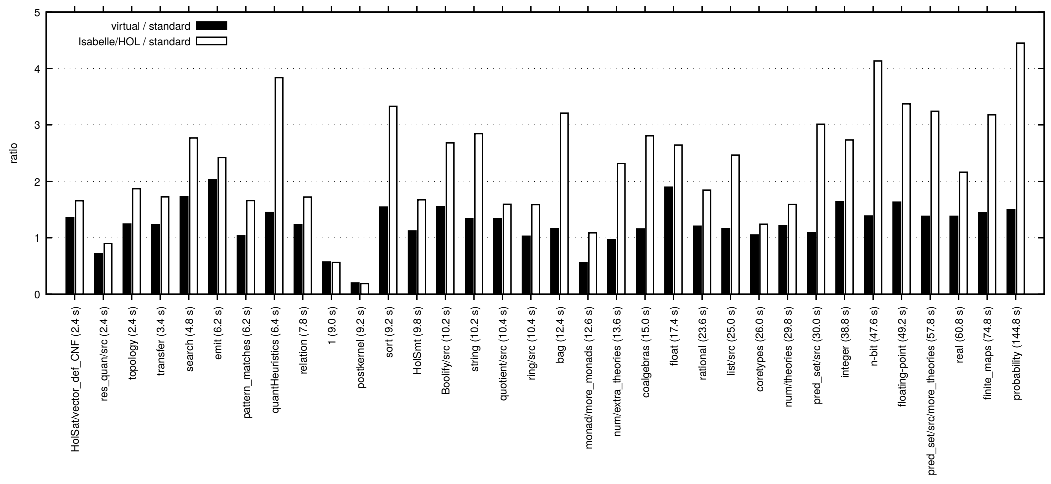


Figure 4 Performance measurements: HOL4 source directory on the x-axis, sorted by elapsed time (in parentheses) to build that directory with standard HOL4. Bars show the time (relative to standard HOL4) to build that directory with virtualized standard HOL4-Kernel (black) and the Isabelle HOL4-Kernel (white).

4.3 Performance Measurements

In this section we report on performance measurements. We investigate whether the overhead incurred by virtualization and adapting kernel interfaces is reasonably moderate and how well it scales with the size of the application.

In Figure 4, we compare the elapsed time for building theories in HOL4 source directories from the `core`, `more`, and `large` build sequences. We exclude the directories that take less than 2 seconds to build in standard HOL4. The reported times are the minimum out of 5 measurements for each directory. The experiments were run (single threaded, because Isabelle and HOL4 have different schemes for parallelization) on a laptop computer with Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz and 32 GB of RAM, running Windows 10.

We observe that the virtual HOL4-Kernel takes about 1.5 times as long as standard HOL4. We interpret this as an indication that we have a good handle on the management of global and local state (Sections 3.1 and 3.2).

Some directories build faster with the virtual HOL4-Kernel. This is likely due to different bootstrapping in the virtual HOL4-environment: more dependencies are already pre-loaded. Another reason could be fewer IO operations in the HOL4-environment, because we virtualize access to the file-system by in-memory data lookup and storage.

The final observation concerning Figure 4 is that the slowdown of the Isabelle HOL4-Kernel is never by more than a factor of 4.5 and usually lies between one and three. We believe that this is a moderate overhead for the adaptations between inferences of Isabelle/Pure and the HOL4-Kernel interface (described in Section 4). Moreover this overhead seems to be constant w.r.t the size of the development, so we expect it to scale to even bigger applications.

Let us now compare our approach of transporting theorems from HOL4 to Isabelle/HOL with OpenTheory (see also Section 7). With the OpenTheory approach, HOL4 scripts are run with a HOL4-Kernel that produces OpenTheory files (`.art`). These files are post-processed with the OpenTheory tool (`opentheory info --article`), which, e.g., deletes unwanted constants. The resulting file can then be imported into Isabelle/HOL, once the importer is set up with information on how to map HOL4 type operators and constants

Table 1 Performance in comparison with OpenTheory. Absolute time and time relative to standard HOL4 (row 1) are reported for transporting HOL4 theories `relation` and `real_topology` to Isabelle/HOL via OpenTheory (rows 2.1-2.3) and our Isabelle HOL4-Kernel (row 3).

		`relation`		`real_topology`	
		absolute [s]	relative	absolute [s]	relative
1	standard HOL4	1.8	1.0	68.4	1.0
2.1	HOL4 OpenTheory kernel (`.art`)	10.2	5.7	546	8.0
2.2	`opentheory info --article` (`.ot.art`)	31.2	17.3	2416	35.3
2.3	Isabelle OpenTheory import	0.9	0.5		
3	Isabelle HOL4-Kernel	6.1	3.4	310	4.5

to their Isabelle/HOL counterparts. In Table 1, we report our performance measurements for exporting and importing the HOL4 theory `relation` and `real_topology`. We chose `relation` because it has few dependencies and therefore allowed us to set up the OpenTheory importer with moderate effort. We chose `real_topology` in order to investigate performance on a large theory: regarding build time, it is the largest theory in the basis library.

The actual import is faster than the original build of the theory `relation`. But producing OpenTheory files is slower than our Isabelle HOL4-Kernel. Post-processing of OpenTheory files is very expensive, it takes more than 17 times as long for the small `relation` theory and even 35 times as long for the large `real_topology` theory.

4.4 Debugging

When attempting to build HOL4 using our Isabelle HOL4-Kernel, we came across many failures that were related either to implementation errors or to unexpected behavior of the Isabelle/Pure primitives we use. Debugging these failures was often a challenge: Errors frequently occurred far from their root cause, especially when program flow in HOL4 is controlled by exceptions. In order to help with the debugging process, we implemented yet another kernel that performs every operation using both the standard HOL4-Kernel and our Isabelle HOL4-Kernel simultaneously, comparing their results. This yields an error at the earliest point where the behavior of the standard kernel and the Isabelle kernel diverge and therefore points directly to discrepancies in the implementation (together with concrete arguments that caused the bad behavior).

5 Transfer

In order to keep the Isabelle HOL4-Kernel as simple and maintainable as possible, we do not make any attempts at transforming the imported definitions or theorems to somewhat more idiomatic concepts in Isabelle/HOL. For example, apart from the axiomatization in Section 4.1.3, we do not map types/constants from HOL4 to existing types/constants in Isabelle/HOL. We also do not use the Isabelle/HOL datatype package, but simply use the constructions performed by HOL4.

Overall, we get a completely separate formalization of closely related concepts. E.g., both Isabelle and HOL4 define natural numbers and lists. We realign those in a post-hoc fashion, and Isabelle's `transfer` package [5, 13] is a powerful, flexible, and efficient tool perfectly suited for these needs.

Subsequently, we propose an approach that allows the user to obtain an idiomatic Isabelle/HOL formalization from the imported HOL4 libraries. This requires some user interaction, but arguably there has to be some human interaction to judge what an "idiomatic" definition looks like. We provide infrastructure to make this process as comfortable as possible. In particular, we enable the user to mix HOL4 syntax and Isabelle/HOL syntax in order to state and prove theorems that relate Isabelle/HOL concepts to HOL4 concepts.

The running example to illustrate our approach will be lists. The HOL4 formalization defines the type of lists as a datatype:

```
Datatype.Hol_datatype ‘list = NIL | CONS of ’a => list‘
```

In the Isabelle HOL4-Kernel (Section 4), we adopt the naming scheme that identifiers `id` from a HOL4 theory segment `seg` are mapped to Isabelle identifiers *seg___id*. This means that in the Isabelle/HOL foundation, the above HOL4-datatype definition results in the definition of a type *'a list___list* and constants *list___NIL*, *list___CONS* (the datatype definition is in the segment `list`).

typedecl *'a list___list*
consts *list___NIL::'a list___list*
consts *list___CONS::'a ⇒ 'a ⇒ 'a list___list*

In the rest of this section, we show how to relate these constants to the "idiomatic", existing datatype constructors of lists in Isabelle/HOL:

datatype *'a list = Nil | Cons 'a "'a list"*

5.1 Mixing HOL4 and Isabelle Entitites

We first describe how a user can mix HOL4 and Isabelle-syntax. This takes inspiration from an experiment by Hupel [6] that allows the embedding of ML values into a formal context. We provide special syntax for Isabelle, that allows the user to write *HOL4<expr>* in inner syntax, and *expr* will be parsed by the HOL4 parser and return the corresponding Isabelle/HOL term. For example, a property *P* of two-element HOL4-lists can be expressed as *P (HOL4<length [x, y]>)*, which will be parsed as *P (list___length (list___CONS x (list___CONS y list___NIL)))*.

Theorems produced by the Isabelle HOL4-Kernel do not even show up in the Isabelle/Isar namespace, they are only stored as ML-values in the universal context in the HOL4-environment. In order to comfortably refer to HOL4 theorems in Isabelle/Isar, we provide an attribute *[[hol4_thm segment.THEOREM]]* that refers to the theorem `THEOREM` from the HOL4-segment `segment`.

5.2 Transfer Rules

The main tool for proving theorems along isomorphisms is the `transfer` package [5, 13]. The central element for setting up the `transfer` package are *transfer rules*. For the purposes of this presentation, transfer rules are of the form *R c d* for relations *R :: 'a => 'b => bool* and constants *c :: 'a*, *d :: 'b*. For simplicity, we assume that *R* is bi-total and bi-unique, i.e., the graph of a bijection between *'a* and *'b*, but the `transfer` package is sufficiently flexible to deal with partial and quotient relations as well. We say that *R c d* is a transfer rule for constant *d*.

Relators are used to construct relations for compound types, in particular the function relator with infix syntax `===>` relates functions *f* and *g* that yield *S*-related results *f x, g x* for *R*-related arguments *x, y*. Written as a transfer rule: *(R ===> S) f g*.

Given a set of transfer rules and a theorem, the `transfer` package looks up transfer rules for each of the constants that occur in the theorem and composes them (recall that the transfer relations encode bijections) to obtain a theorem for an isomorphic (along the transfer rules) theorem. E.g., given a transfer rule `(R ===> (=)) P Q` for predicates `P :: 'a => bool` and `Q :: 'b => bool`, a transfer rule `R c d` to transfer a constant `d::'b` to `c::'a`, and a theorem `Q d`, the transfer package will produce the theorem `P c`. To clarify the intuition, `Q` and `d` will be constants imported from HOL4, and `P` and `c` related idiomatic constants in Isabelle/HOL.

The basis of our setup consists of transfer rules for all of the constants defined in the HOL4 `bool` theory. To give an example, the transfer rule for universal quantification relates the HOL4 all-quantifier `!` with the Isabelle/HOL all-quantifier `All` for `(A ===> (=))`-related predicates, given a bijection `A`.

lemma *[transfer_rule]: "((A ===> (=)) ===> (=)) All HOL4<(!)>"* **if** *"bi_total A"*

We prove similar rules for e.g., conjunction, implication, negation, True, and False. The proofs are straightforward, because the definitions in HOL4 and Isabelle/HOL both follow the axiomatization from Section 4.1.3.

5.3 Setting up Transfer for Lists

We now proceed with setting up transfer rules for HOL4-lists. The HOL4 datatype package constructs (among others) an induction theorem:

```
list_INDUCT = `!P. P [] /\ (!t. P t ==> !h. P (h::t)) ==> !l. P l`
```

In the Isabelle/HOL foundation, we can look up this theorem with the attribute *[[hol4_thm <list.list_INDUCT>]]*. The resulting theorem looks like this:

```
bool___! (λP. bool___/\ (P list___NIL)
  (bool___! (λt.  P t ⟶ bool___! (λh.  P (list___CONS h t)))) ⟶ bool___! P)
```

It can be transferred (automatically, using the *untransferred* attribute) along the transfer rules for the constants from the HOL4 `bool` theory (`bool_!` and `bool___/\`) in order to prove a proper Isabelle/HOL induction rule, mixing in HOL4-syntax for the HOL4-constants `NIL` and `CONS`:

lemma *hol4_list_induction[case_names NIL CONS, induct type]:*
  "P x" **if** *"P HOL4<NIL>"* **and** *"(⋀x xs. P xs ⟹ P (HOL4<CONS> x xs))"*
using *[[hol4_thm listTheory.list_INDUCT>, untransferred, of P x]]* **by** *simp*

In order to establish a connection between lists in Isabelle/HOL and lists in HOL4, we define a function (in Isabelle/HOL) from Isabelle/HOL lists to their HOL4-counterparts.

fun *convert_list :: "'a list ⇒ 'a list___list"*
where *"convert_list [] = HOL4<NIL>"*
    | "convert_list (x#xs) = HOL4<CONS x> (convert_list xs)"

With the induction rule for HOL4-lists, as well as injectivity of `CONS` and disjointness of `NIL` and `CONS` – those rules are also provided by HOL4 – we can easily prove injectivity and surjectivity of *convert_list*. Therefore the relation *(λx y. convert_list x = y)* is bi-total and bi-unique and it can be used as a transfer relation for lists of elements of the same type.

The `transfer` package provides some more support, in particular automation to set up parametric transfer rules:

```
setup_lifting ‹Quotient (=) convert_list convert_list' (λx y. convert_list x = y)›
  parametric ‹(list_all2 A ===> list_all2 A ===> (=)) (=) (=)›
```

This defines a relator *rh4_list::('a ⇒ 'b ⇒ bool) ⇒ 'a list ⇒ 'b list ⇒ bool*, where *rh4_list A is hs* expresses that *is* and *hs* are of the same shape and their elements are (pointwise) in relation *A*. With this setup, we can prove transfer rules for `NIL` and `CONS`

```
lemma [transfer_rule]: rh4_list A Nil (HOL4‹NIL›)
lemma [transfer_rule]: (A ===> rh4_list A ===> rh4_list A) Cons HOL4‹CONS›
```

With these rules proved, the `transfer` package can be used to automatically transfer every theorem that involves `NIL`, `CONS` and constants from HOL4 theory `bool`.

5.4 Primitive Recursive Definitions

HOL4 does not expose generic recursors for primitive recursive functions. A convenient way to transfer constants that are defined by primitive recursion in HOL4 is to define (in Isabelle/HOL) a recursor for HOL4 lists in terms of the recursor *rec_list* for Isabelle/HOL lists. The command **lift_definition** provides infrastructure to define constants in terms of isomorphic constants (here the isomorphism is between between *list* and *list___list* and has been set up by the previous **setup_lifting** command).

```
lift_definition rec_hol4_list::
  "'a ⇒ ('b ⇒ 'b list___list ⇒ 'a ⇒ 'a) ⇒ 'b list___list ⇒ 'a"
  is rec_list::"'a ⇒ ('b ⇒ 'b list ⇒ 'a ⇒ 'a) ⇒ 'b list ⇒ 'a"
  parametric list.rec_transfer .
```

Then one only needs to express the respective constants in terms of this recursor, e.g., for appending lists:

```
lemma "append xs ys = rec_list ys (λ x _. Cons x) xs"
lemma "HOL4‹APPEND› xs ys = rec_hol4_list ys (λ x _. HOL4‹CONS› x) xs"
```

These lemmas follow directly from the definition of the recursors and the involved functions. Then the transfer rule for `APPEND` (*"(rh4_list A ===> rh4_list A ===> rh4_list A) append (HOL4‹APPEND›)"*) can be proved automatically by the *transfer_prover* (after unfolding the above equivalences), because there are transfer rules for each of the involved constants.

5.5 Example: Transfer Theorems

We demonstrate the possibility of transferring theorems from HOL4 to Isabelle/HOL and back on a derived example. Assume that someone proved `FERMAT` in HOL4, which we simulate by an axiomatization in the virtual HOL4:

```
val FERMAT = Theory.new_axiom ("FERMAT",
``!a b c n. SUC (SUC 0) < n ==> ~(SUM (MAP (\x. SUC x ** n) [a; b]) = SUC c ** n)``)
```

A significant result! Because we proved transfer rules for all of the constants occurring in the theorem statement, we can import this result with a single invocation of *untransferred*.

```
lemma fermat: "Suc (Suc 0) < n ⟹ (∑x←[a, b]. (Suc x) ^ n) ≠ Suc c ^ n"
  using [[hol4_thm fermatTheory.FERMAT, untransferred]] by simp
```

We can even communicate this result back to the virtual HOL4 system: First of all, we need to prove the lemma (in Isabelle/HOL) in a HOL4-friendly format. Again, we have transfer rules for all of the constants, so a single invocation of *transfer* allows us to prove the lemma.

```
lemma fermat_hol4: "HOL4‹! a b c n.  SUC (SUC 0) < n ==>
  ~(SUM (MAP (\x.  SUC x ** n) [a; b]) = SUC c ** n)›"
by transfer (use fermat_isabelle in simp)
```

Then `val fermat_hol4 = @{thm fermat_hol4}` can be used as a regular theorem value in the virtual HOL4 environment and prove the round-tripped theorem `FERMAT2`.

```
val FERMAT2 = store_thm("FERMAT2", ‘‘! a b c n.
    SUC (SUC 0) < n ==> ~(SUM (MAP (\x. SUC x ** n) [a; b]) = SUC c ** n)‘‘,
  METIS_TAC [fermat_hol4]);
```

5.6 Discussion: Manual Interaction

Our proposed approach clearly involves some amount of manual interaction. First, suitable definitions in Isabelle/HOL need to be identified or defined. Then, suitable transfer rules need to be proved for each of these definitions. But (and that is an important aspect), the amount of manual interaction is proportional to the number of constant definitions, and not to the size or number of results that one wants to transfer.

This manual effort is certainly well invested for larger applications, in particular because intermediate constructions need not be set up in order to transfer final results. For example, there is no need to set up transfer for all intermediate languages in the CakeML stack to transfer a final correctness theorem that involves only machine-code and CakeML-syntax.

6 Conclusion

Thanks to the proper setup of a virtual ML environment, we have managed the daunting task of taming mutable state in a large application program: HOL4. Many big and small problems had to be overcome; it is great to see such a collaboration between different systems already work so well. This effort has induced changes to the internals of both HOL4 and Isabelle, which were overseen by the respective experts and incorporated into their repositories.

All involved systems profit from this work: HOL4 has yet another kernel, and remaining issues of the emulation could point to HOL4 code that needs further improvement. Isabelle now has a systematic treatment of alternative ML environments, with user-defined static basis and token language. Since virtual HOL4 runs inside Isabelle/jEdit [20], we could even see that as a viable IDE for HOL4 in the near future, although its user community is still very content with more traditional vi and Emacs interfaces.

We imagine fruitful interoperability: For example, HOL4 tactics could be used for Isabelle proofs, or HOL4 users could work with Isabelle/HOL formalizations (e.g., from the AFP) in the HOL4-environment.

A full import of the CakeML project in Isabelle/HOL is still future work, but it could yield a much larger user-base for the CakeML formalization, when all the tools of HOL4 and Isabelle/HOL can be combined in a single environment. The material on CakeML by Hupel in the AFP [7] is already awaiting to be formally connected.

7 Related Work

In 1995, Slind implemented the TFL package [18] generically, such that the ML sources worked both for Isabelle/HOL and HOL4. A few years later, both sides were maintained independently and diverged: today Isabelle has still a legacy `recdef` command and HOL4 a substantially extended `Definition` function, both based on TFL. Despite its limited success in bridging the gap between Isabelle/HOL and HOL4, the TFL package shares the key idea of our approach to load original ML sources into the other proof assistant.

More conventional export and import facilities write internal data structures to the file-system (essentially a trace of the inference kernel and theory content) and load them into the other system. A notable example is the HOL(Light) to Isabelle converter: the first version by Skalberg and Obua [16] had scalability problems due to massive amounts of XML data written to a Unix file-system. This has been greatly improved by Kaliszyk and Krauss [10]: the HOL-Light standard library is loaded into Isabelle/HOL in a few minutes.

OpenTheory by Hurd [8] is a similar approach based on kernel traces, but its theory and proof representations follow a published standard format. This has been designed to cover all members of the HOL family, but this excludes Isabelle/HOL with its distinctive deviations in the primitive logic (e.g. support for type-classes with overloaded definitions). Consequently, the OpenTheory importer for Isabelle did not get beyond experimental state so far, and an exporter never worked out. We also see fundamental problems in scalability to really large libraries: the OpenTheory standard library [9] is rather small compared to applications seen today, e.g. in Isabelle AFP [3], or CakeML [12].

More ambitious export-import projects even attempt to bridge the gap between HOL-Light and Coq [11]. This introduces new questions on the logic, but the fundamental problems of scalability and systems engineering remain the same. It should be noted that our approach is closely related to the original idea behind LCF [4]: instead of handing around proof terms, we merely run a program in a controlled manner to get to the intended theory content.

References

1 Andrew W Appel, James S Mattson, and David Tarditi. A lexical analyzer generator for Standard ML. *Distributed with Standard ML of New Jersey*, 1989.

2 Andrej Bauer, Martín Escardó, Peter L. Lumsdaine, and Assia Mahboubi. Formalization of Mathematics in Type Theory (Dagstuhl Seminar 18341). *Dagstuhl Reports*, 8(8):130–155, 2019. `doi:10.4230/DagRep.8.8.130`.

3 Manuel Eberl, Gerwin Klein, Tobias Nipkow, Larry Paulson, and René Thiemann (editors). The Archive of Formal Proofs. Online Journal. URL: `https://www.isa-afp.org`.

4 M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

5 Brian Huffman and Ondřej Kunčar. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs*, pages 131–146, Cham, 2013. Springer International Publishing.

6 Lars Hupel. Splicing runtime ML values into Isar. isabelle-users mailing list. 09 Jun 2015, `https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2015-June/msg00076.html`.

7 Lars Hupel. CakeML. *Archive of Formal Proofs*, 2018, 2018. URL: `https://www.isa-afp.org/entries/CakeML.html`.

8 J. Hurd. OpenTheory: Package Management for Higher Order Logic Theories. In G. Dos Reis and L. Théry, editors, *Programming Languages for Mechanized Mathematics Systems*, pages 31–37. ACM, 2009.

9 Joe Hurd. The OpenTheory Standard Theory Library. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods – Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2011. `doi:10.1007/978-3-642-20398-5`.

10 Cezary Kaliszyk and Alexander Krauss. Scalable LCF-Style Proof Translation. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 51–66, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

11 Chantal Keller and Benjamin Werner. Importing HOL Light into Coq. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, pages 307–322, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

12 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192, 2014. `doi:10.1145/2535838.2535841`.

13 Ondřej Kunčar. *Types, Abstraction and Parametric Polymorphism in Higher-Order Logic*. Dissertation, Technische Universität München, München, 2016. URL: `http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20160408-1285267-1-5`.

14 David Matthews. The Poly/ML implementation of Standard ML. Website. URL: `https://www.polyml.org`.

15 Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

16 Steven Obua and Sebastian Skalberg. Importing HOL into Isabelle/HOL. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, pages 298–302, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

17 Lawrence C Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

18 Konrad Slind. Function definition in higher-order logic. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics*, pages 381–397, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

19 Konrad Slind and Michael Norrish. A brief overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32. Springer, 2008.

20 Makarius Wenzel. Asynchronous User Interaction and Tool Integration in Isabelle/PIDE. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Vienna, Austria*, volume 8558 of *Lecture Notes in Computer Science*. Springer, 2014. URL: `https://www21.in.tum.de/~wenzelm/papers/itp-pide.pdf`.

21 Makarius Wenzel. *The Isabelle/Isar Implementation*, 2019. URL: `http://isabelle.in.tum.de/website-Isabelle2019/dist/Isabelle2019/doc/implementation.pdf`.