The Common HOL Platform

Mark Adams
Proof Technologies Ltd, UK
Radboud University, Nijmegen, The Netherlands

The Common HOL project aims to facilitate porting source code and proofs between members of the HOL family of theorem provers. At the heart of the project is the Common HOL Platform, which defines a standard HOL theory and API that aims to be compatible with all HOL systems. So far, HOL Light and hol90 have been adapted for conformance, and HOL Zero was originally developed to conform. In this paper we provide motivation for a platform, give an overview of the Common HOL Platform's theory and API components, and show how to adapt legacy systems. We also report on the platform's successful application in the hand-translation of a few thousand lines of source code from HOL Light to HOL Zero.

1 Introduction

The HOL family of theorem provers started in the 1980s with HOL88 [5], and has since grown to include many systems, most prominently HOL4 [16], HOL Light [8], ProofPower HOL [3] and Isabelle/HOL [12]. These four main systems have developed their own advanced proof facilities and extensive theory libraries, and have been successfully employed in major projects in the verification of critical hardware and software [1, 11] and the formalisation of mathematics [7].

It would clearly be of benefit if these systems could "talk" to each other, specifically if theory, proofs and source code could be exchanged in a relatively seamless manner. This would reduce the considerable duplication of effort otherwise required for one system to benefit from the major projects and advanced capabilities developed on another. Work to date has concentrated on exchange of proofs via proof objects, with some degree of success, but little has been done to facilitate porting of source code.

The Common HOL Platform is part of the Common HOL project for facilitating the porting of source code and proofs between HOL systems. It defines a standard HOL theory compatible with the core theory of each HOL system, and an application programming interface (API) of programming components that is more-or-less common to all HOL systems. It has so far been supported in HOL Light, hol90 [15] and HOL Zero [19].

In this paper we give an overview of the platform. In Section 2, we further discuss motivation. In Section 3, we cover the platform's choice of components. In Section 4, we explain how to adapt legacy systems to conform to the platform. In Section 5, we report on its successful usage in assisting the manual porting of both new and legacy source code. In Section 6, we present our conclusions.

2 Motivation

By definition, all systems in the HOL family implement the HOL logic or a close variant. However, in practice their commonality stretches far beyond this. They have broadly similar axiomatisations of the logic, similar mechanisms for logical extension, similar formal language concrete syntax and build up similar foundational theory. Furthermore, in most basic usage at least, they each support similar paradigms of user interaction, namely simple forwards-style application of inference rules and backwards-style tactic proofs via the subgoal goal package [14], performed in an interactive functional programming session. Also, their implementations are all written in variants of the ML functional programming language, all employ an LCF-style architecture [6] and are all built up from similar libraries of programming utilities, syntax utilities, inference rules and tactics.

Other than in these basic aspects, the systems branch off in their own respects. Each builds up considerable theory beyond the basic foundations in its own way. For example, real numbers in HOL Light are constructed quite differently from real numbers in ProofPower HOL. There is also much variation in their provision of user proof commands, especially for those relating to proof automation, with each system having its own strengths and idiosyncrasies. Most different is Isabelle/HOL, which is implemented as an instantiation of the Isabelle generic theorem prover [17] rather than by having its deductive system "hardwired" as source code, and supports a variant of the HOL logic that has axiomatic type classes. Also, the predominant mode of interaction with Isabelle has become the declarative proof language Isar in conjunction with a bespoke IDE, rather than the subgoal package in an interactive ML session.

Porting proofs between HOL systems by hand involves translating proofs scripts. These proof scripts typically involve heavy use of high-level proof commands that differ between systems. In cases where such commands are used to finish off subgoals, it is often possible to find a suitably powerful command to do the same in the target system, but in other cases proof scripts have to be recreated from scratch. Automatic proof porting, via recording of low-level proof steps and export to proof object files, is vastly preferable if it can be made sufficiently reliable. Such a capability requires a platform of common foundational theory, inference rules and logical extension mechanisms in both systems.

There have been notable successes in the large scale porting of legacy proofs between HOL systems via proof objects. Obua and Skalberg [13] developed a capability for porting proofs from HOL4 to Isabelle/HOL, using a theory platform based on the HOL4 inference kernel, and then adapted this for porting from HOL Light to Isabelle/HOL. Kaliszyk and Krauss [10] developed a capability for porting from HOL Light to Isabelle/HOL, based on the HOL Light inference kernel. The OpenTheory project [9] is based around the HOL Light axiomatisation, and establishes a common proof object format for porting proofs between various HOL systems, including HOL4, ProofPower/HOL and HOL Light, with ongoing work to support Isabelle/HOL. However, these capabilities would all struggle to port something as large as the entire Flyspeck project [7]. We believe that significant advances in capability can be achieved by exploiting a broader commonality that exists between HOL systems, using a platform at a somewhat higher level than the inference kernel of one system.

Porting source code from one system to another currently requires deep knowledge of both systems' implementations and can entail weeks of effort to replicate behaviour sufficiently closely. Naive porting of high-level routines will typically result in unreliable code due to the compounding of small and subtle differences in the theory or in ML function behaviour. We know of no pre-existing capability for supporting the systematic porting of source code between HOL systems.

We believe that if the existing HOL systems can be adapted to support a well-designed API that reflects the commonality of "primary functionality" (by which we mean functionality directly concerned with theorem proving) between the systems, then much of the pain of porting source code can be avoided. There is then a platform of precisely corresponding programming components, and source code built on this platform in one system can be trivially but accurately ported to another system conforming to the same platform. As is also the case for a proof porting capability, both ML components and foundational theory have to be taken into account when designing an effective platform.

3 Components

In this section, we give an overview of the components that make up version 0.5 of the Common HOL Platform. This is the latest version, and has been implemented for HOL Light and HOL Zero. An earlier version was implemented for hol90, but this has not yet been upgraded. Even though the platform has not yet been implemented for ProofPower HOL or HOL4, it has been carefully designed with knowledge of how these systems work. However, little consideration has so far been given to Isabelle/HOL, which presents greater challenges due to its greater differences. A significant redesign of the standard would probably be required to properly cater for Isabelle/HOL.

There is no space in this paper to list all the platform components, let alone to describe each one. Instead we provide various tables comparing some corresponding components from hol90, HOL4, ProofPower HOL, HOL Light and HOL Zero. For a given system, each platform component is either exactly represented in the system, or it is approximately represented, or it is not represented in the system. In our listings, those components only approximately corresponding are written in curly brackets.

There is not yet a single stand-alone document specifically for the purpose of precisely defining each platform component. However, part of the original motivation for the HOL Zero system was to act as a clear demonstration of the platform, and it has been designed to exactly conform to platform behaviour without adaption. Readers can download the HOL Zero source distribution [19], where source code file `commonhol.mli` gives a complete list of the API components, and the user manual appendices give a precise description of each API and theory component.

3.1 Considerations

Here we discuss some factors that should be taken into consideration when choosing the components.

Commonality Platform components should broadly reflect the commonality that exists between the systems. Including components that are only relevant in one system would entail extra effort to make the other systems conformant, and would be of little use to them. Not including components that are common to all systems would mean that basic components from one system would have to be needlessly considered when porting to a target system.

Usage Amount of usage in post-platform code should be taken into consideration when deciding the platform components. Heavily used components should almost qualify by default.

Level The components should be sufficiently high-level to be of likely use in post-platform source code. For example, including low-level subcomponents used to make a HOL term parser would be of little use, even if these components were common to all HOL systems.

Precision A platform without precisely defined components of course loses much of its purpose. In HOL systems, there are many small differences in the details of the behaviour of various corresponding basic functions. For each component, the platform should explicitly specify its exact behaviour or otherwise be clear about what is not specified. Non-conformant components must have platform-conformant variants defined as part of platform qualification.

Underspecification The API should allow some degree of flexibility in certain kinds of details about it components. For example, the ML names of the components, or the order in which function components take arguments and whether tuples or curried form is used. The API should seek to minimise the effort required to make legacy systems conformant by underspecifying these details, which are not the kinds of differences that make porting source code difficult.

Completeness The components should be complete in the sense that all primary functionality can be built from platform components alone. This becomes essential for the constructors and destructors of abstract datatypes (such as for HOL types, terms and theorems) because there is otherwise no way of manipulating such values.

Coherence The components should be chosen as a coherent set that categorise in a complete and consistent way and that composes robustly. This makes it easier to write new code based on the API, as well as helping portability.

Performance The API should not exclude components that are important to the performance of a system if this means they would otherwise need to be reimplemented in the outer platform in terms of API components to result in a significant degradation in performance.

Ease of Implementation The implementation effort required to conform to a platform is a significant consideration. Otherwise, in practice the platform will not get implemented for the full range of HOL systems, which defeats its purpose.

3.2 Theory Components

The theory components are the axioms, declarations and definitions that must exist in a conformant system's theory. They must form a sufficient basis for building up each HOL system's theory.

There is some variation in the systems' axiomatisations, especially between HOL Light and the other systems. Because each system implements the same formal logic, for our purposes of completeness it is sufficient to choose the core theory (i.e. the theory of the logical core) of one system as the theory platform, and to derive this in the other systems from their respective core theories. The outer platform (see Section 4.1) in these other systems can then "re-derive" the system's core theory using the theory platform. A platform theorem may be an axiom or definition theorem in one system and a derived theorem in another, but as far as the platform is concerned they are all just theorems.

Our theory platform features the axioms and definitions of ProofPower HOL, which we view as the most intuitive, and which are close to those of hol90 and HOL4. It also includes the HOL Light definition of the implication operator, which does not feature in the other systems because the behaviour of implication drops out from their primitive inference rules and the implication antisymmetry axiom. Including this definition means that any of the systems' primitive inference rule set suffices to complete the deductive system. A handful of fundamental theorems that are common to but derived in each system are included in the platform, such as the truth theorem and the Law of the Excluded Middle, because they are inevitably needed in implementing the platform and so may as well feature as components.

The type constants and constants declared in the theory platform include those from the basic theory about predicate logic and lambda calculus that is common to each HOL system, established in the logical core and initial derived theory of each system. This includes the function space type operator and the boolean base type, plus the equality, conjunction, disjunction, implication and logical negation operators, the universal, existential and unique existential quantifiers and the Hilbert choice operator.

Beyond this, each system builds up essentially equivalent theory of pairs, lists and natural numbers. To take advantage of this commonality, the platform also includes theory for pairs and natural numbers, including natural number numerals and 13 classic arithmetic operators including plus, multiply and exponentiation. Theory for lists does not currently feature, but is planned for inclusion in a future version.

The representation of natural number numerals varies between HOL systems: in HOL Light, HOL4 and HOL Zero, each numeral is constructed using compounding of two unary operators on the zero constant (one for multiplying by two and adding one, and one for multiplying by two and adding zero or

hol90	HOL4	ProofPower	HOL Light	HOL Zero
"bool"	"bool"	"BOOL"	"bool"	"bool"
"fun"	"fun"	"$\rightarrow$"	"fun"	"->"
"prod"	"prod"	"$\times$"	"prod"	"#"
"ind"	"ind"	"IND"	"ind"	"ind"
"num"	"num"	"$\mathbb{N}$"	"num"	"nat"
"T"	"T"	"T"	"T"	"true"
"F"	"F"	"F"	"F"	"false"
"="	"="	"="	"="	"="
"/\"	"/\"	"$\wedge$"	"/\"	"/\"
"\/"	"\/"	"$\vee$"	"\/"	"\/"
"~"	"~"	"$\neg$"	"~"	"~"
"!"	"!"	"$\forall$"	"!"	"!"
"?"	"?"	"$\exists$"	"?"	"?"
"?!"	"?!"	"$\exists_1$"	"?!"	"?!"
"@"	"@"	"ε"	"@"	"@"
IMP_ANTISYM_AX	IMP_ANTISYM_AX$^+$	$\Rightarrow$_antisym_axiom	-	imp_antisym_ax
ETA_AX	ETA_AX	η_axiom	ETA_AX	eta_ax
SELECT_AX	SELECT_AX	ε_axiom	SELECT_AX	select_ax
BOOL_CASES_AX	BOOL_CASES_AX	bool_cases_axiom	BOOL_CASES_AX$^+$	bool_cases_thm$^+$
INFINITY_AX	INFINITY_AX	infinity_axiom	INFINITY_AX	infinity_ax
T_DEF	T_DEF	t_def	T_DEF	true_def
F_DEF	F_DEF	f_def	F_DEF	false_def
AND_DEF	AND_DEF	$\wedge$_def	{AND_DEF}	conj_def
-	-	-	IMP_DEF	-
OR_DEF	OR_DEF	$\vee$_def	OR_DEF	disj_def
NOT_DEF	NOT_DEF	$\neg$_def	NOT_DEF	not_def
FORALL_DEF	FORALL_DEF	$\forall$_def	FORALL_DEF	forall_def
EXISTS_DEF	EXISTS_DEF	$\exists$_def	EXISTS_THM$^+$	exists_def
{UEXISTS_DEF}	{UEXISTS_DEF}	$\exists_1$_def	{UEXISTS_DEF}	uexists_def

Table 1: The type constants, some of the constants and some of the theorems (including all the axioms) of the theory platform. Derived theorems in a given system are marked with $^+$.

two depending on the system), whereas numerals in hol90 and ProofPower HOL form an infinite family of constants. However, beyond the definition of a set of basic numeral arithmetic evaluation inference rules, these differences do not surface in practice in the implementations of the systems. Thus we have abstracted away from the theory platform the detail of how numerals are defined.

3.3 API Components

The API components form the ML interface for programming primary functionality. There are approximately 475 components, mainly consisting of ML function and constant values, but also seven datatypes and three exceptions. Three configuration values are also provided, that hold the HOL system name and version and the Common HOL Platform version. In each conformant system, the API is provided as an ML module interface file, with components given the same ordering to aid comparison between systems.

Note that table components that have ML infix fixity in a given system are written in parentheses.

3.3.1 Functional Programming Library

There are around 100 functional programming library components (see Table 2 for a selection).

hol90	HOL4	ProofPower	HOL Light	HOL Zero
`curry`	`curry`	`curry`	`curry`	`curry`
`uncurry`	`uncurry`	`uncurry`	`uncurry`	`uncurry`
`C`	`C`	`switch`	`C`	`swap_arg`
`I`	`I`	`I`	`I`	`id_fn`
`K`	`K`	`K`	`K`	`con_fn`
`W`	`W`	-	`W`	`dbl_arg`
`(o)`	`(o)`	`(o)`	`(o)`	`(<*)`
`(##)`	`(##)`	`(**)`	`(F_F)`	`pair_apply`
`map`	`map`	`map`	`map`	`map`
`map2`	`map2`	-	`map2`	`bimap`
`{funpow}`	`{funpow}`	`fun_pow`	`{funpow}`	`funpow`
`itlist`	`itlist`	`fold`	`itlist`	`foldr`
`rev_itlist`	`rev_itlist`	`revfold`	`rev_itlist`	`foldl`
`end_itlist`	`end_itlist`	-	`end_itlist`	`foldr1`
-	-	-	-	`foldl1`

Table 2: Some of the functional programming library API components.

Included are many basic operations on ML pairs, lists and strings, such as selecting the first element of a pair, reversing the order of elements in a list, or turning an integer into a string. Association lists are also supported. Also included are various classic functional programming meta operations, e.g. for applying a function to each element in a set, or folding up a list into a single element by repeated application of a binary operator. There is also a collection of set operations on lists, such as set membership and set union, under either equality comparison or a supplied equivalence relation.

For coherence, we fill out the gaps that exist in the various legacy systems' libraries. For example, all kinds of folding operators and their inverses, unfolding operators, are provided, and all set operations are provided for both under equality and a supplied equivalence relation.

Three kinds of standard exception are catered for: normal failure, catastrophic failure and "local failure" (used for control flow within a function). The API underspecifies the form of the exception arguments and the textual content of error messages

Note that there is some variation in the behaviour of some library functions between systems. For example, `funpow`, which iterates a function application for the number of times specified by a supplied integer, does not fail in hol90, HOL4 or HOL Light if the integer is negative. Generally, platform functions are specified to fail if supplied with invalid arguments, and the platform version of `funpow` fails if its supplied integer is negative, as is done in ProofPower HOL and HOL Zero.

3.3.2 Type, Term and Theorem Utilities

Around 150 HOL type, term and theorem manipulation utilities are provided (see Table 3 for a selection).

The bulk of these utilities are syntax functions for HOL types or terms, for constructing, destructing and testing for a given syntactic category. Two levels of syntactic category are supported for both types and terms. Firstly, there are the primitive syntactic categories, namely the type variables and type constant applications for types, and variables, constants, function applications and lambda abstractions for terms. These are very widely used throughout the HOL implementations. Secondly, there are the basic syntactic categories associated with the type constants and constants of predicate logic and lambda calculus that feature in the theory platform. Some of these are also used heavily throughout the HOL implementations, but we include support for all such syntactic categories in the API for coherence with the theory platform and the API inference rules.

hol90	HOL4	ProofPower	HOL Light	HOL Zero
`type_of`	`type_of`	`type_of`	`type_of`	`type_of`
`type_vars_in_term`	`type_vars_in_term`	{`term_tyvars`}	`type_vars_in_term`	`term_tyvars`
`aconv`	`aconv`	(`~=$`)	`aconv`	`alpha_eq`
-	`rename_bvar`	-	{`alpha`}	`rename_bvar`
`free_vars`	`free_vars`	`frees`	`frees`	`free_vars`
`free_varsl`	`free_varsl`	-	`freesl`	`list_free_vars`
-	`var_occurs`	`is_free_in`	{`vfree_in`}	`var_free_in`
{`free_in`}	`free_in`	-	`free_in`	`term_free_in`
`all_vars`	-	-	`variables`	`all_vars`
`all_varsl`	-	-	-	`list_all_vars`
`inst`	{`inst`}	{`inst`}	{`inst`}	`tyvar_inst`
-	`rename_bvar`	-	{`alpha`}	`rename_bvar`
-	-	{`var_subst`}	`vsubst`	`var_inst`
{`subst`}	{`subst`}	`subst`	`subst`	`subst`

Table 3: Some of the term utility API components.

There are various ML bindings for HOL constants and base types featured in the theory platform, and for commonly used HOL type variables. Also included are utilities for destructing a theorem into its assumptions and conclusion parts, and for equality and alpha-equivalence comparison of theorems. There are also various type and term operations defined that are essential for defining an inference kernel. These include calculating the type of a term, listing the type variables of a type, testing for the alpha equivalence of two terms, and performing variable and type variable instantiation.

The platform utilities for HOL terms are generally specified to work modulo alpha equivalence in their arguments. This was decided because different systems generate bound variable names differently when avoiding variable capture in type variable and variable instantiation, and so this measure makes the API functions more robust when ported. An arbitrary bound variable name used in an operation in one system could otherwise cause the equivalent operation in another system to fail. Note that hol90's `free_in`, which tests for one term occurring free in another, does not work modulo alpha equivalence, and so does not conform to the platform.

Note that there are various subtle differences between different systems' utilities that can trip up casually ported code. Examples include ProofPower HOL's `mk_const` constructor, which does not test that a constructed constant is well-formed, and hol90's and HOL4's `dest_imp` and `is_imp`, which work for logical negation as well as implication (although HOL4 has `dest_imp_only` and `is_imp_only` for implication only). The API chooses more conventional behaviour.

3.3.3 Theory Extension and Listing Commands

Around 40 theory extension and querying functions are provided. This includes primitive theory extension commands for type declaration, term declaration, constant definition, constant specification and type constant definition. On top of these, there are a few basic derived theory extension commands, for example the command to define a function constant using a universal quantifier for the function arguments instead of a lambda abstraction. Most systems have more sophisticated extension commands, but these are excluded from the platform because there is much variation in their capability between systems.

Each system also provides querying commands to access information about the theory extensions that have been made, although HOL Light omits support for querying about primitive type constant definitions. Such commands are essential for the approach for proof auditing advocated in [2], and a complete set features in the API.

3.3.4 Inference Rules

Around 100 basic inference rules are provided by the API (see Table 4 for a selection).

It is sufficient for the platform inference rules to include just a kernel of primitive rules[1] that suffice, when coupled with the axiom and definition theorems in the theory platform, to implement the HOL deductive system. Given our choice of theory platform, any of the systems' primitive inference rules would be sufficient. However, efficiency is also a consideration. If a primitive rule of a given system were missing from the API, it would have to be reimplemented in that system's outer platform in terms of the API inference rules, and which would in turn need to be implemented in terms of the system's primitives. An execution of such a recreated primitive could require 10 pre-platform rule applications or more, resulting in an unacceptable performance penalty. Thus we choose to include the union of primitive rules from each system in the platform (with the exception of one HOL Light primitive explained below). This principle qualifies around 35 rules for inclusion in the platform. Note that each system except HOL Zero has primitive rules that are derive able in terms of other primitives, but are included to improve the system's performance, which explains why the union includes as many as 35.

Also included are around 15 other inference rules at roughly the same level as the union of the primitive inference rules, including the equality symmetry rule and the cut rule, for using the conclusion of one theorem to eliminate an assumption in another. A further 25 rules are included for performing equality congruence over certain operators, in addition to the two that are present as a result of being primitive inference rules. For coherence, these fill out the patchy provision in existing HOL systems with full coverage for the HOL operators supported by the API syntax functions.

In addition, for natural arithmetic expressions there are conversions provided for performing evaluation of operators applied to numeral arguments for each of the 13 natural arithmetic operators featured in the theory platform. This is sufficient to provide complete coverage of the primitive natural numeral arithmetic inference rules provided by hol90 and ProofPower HOL (which represent numerals as constants). This allows the platform to keep abstract the underlying representation of numerals.

It is vital that the API specifies precise behaviour for each of its inference rules. There is a degree of variation in the behaviour of various rules between systems. We outline here some ways in which the platform promotes robustness in the details of the behaviour it specifies for its inference rules.

As with the API's term utilities, its inference rules also work modulo alpha equivalence, for the same reasons. Note that the successful execution of HOL Light's `BETA` rule (not to be confused with its `BETA_CONV` rule) can fail depending on the name used for a bound variable in one of its arguments, and because of this it is excluded from the API, despite being a primitive of HOL Light. Fortunately, the consequences on performance in HOL Light are minimal because `BETA` can be implemented purely in terms of `BETA_CONV`, which is in the API.

It was also decided that API inference rules should not depend on the presence of assumptions in their theorem arguments, also to help robustness. It is harmless for a rule to remove an assumption if it can, and this should not result in failure in rules composed with it. So, for example, the rule for discharging an assumption matching a supplied term should not fail if the assumption is not present in the theorem argument. Note that ProofPower's classical contradiction rule `c_contr_rule` breaks this principle, but other systems' equivalents do not.

There are also various other differences in behaviour between seemingly equivalent rules in different HOL systems. One particularly extreme case is the rule for instantiating type variables, called `INST` in hol90, HOL4 and HOL Light, which is a primitive of every HOL system. In hol90, only type variables in the conclusion are instantiated. In HOL Light and HOL4, non-variable types in the instantiation list

[1] In the paper, we occasionally abbreviate the term *inference rule* to *rule*.

hol90	HOL4	ProofPower	HOL Light	HOL Zero
ASSUME*	ASSUME*	asm_rule*	ASSUME*	assume_rule*
BETA_CONV*	BETA_CONV*	simple_β_conv*	BETA_CONV	beta_conv*
CCONTR*	CCONTR*	{c_contr_rule}	CCONTR	ccontr_rule
CHOOSE*	CHOOSE*	simple_$\exists$_elim	CHOOSE	choose_rule
CONJ*	CONJ*	$\wedge$_intro	CONJ	conj_rule
CONJUNCT1*	CONJUNCT1*	$\wedge$_left_elim	CONJUNCT1	conjunct1_rule
CONJUNCT2*	CONJUNCT2*	$\wedge$_right_elim	CONJUNCT2	conjunct2_rule
CONTR*	CONTR	contr_rule	CONTR	contr_rule
-	-	-	DEDUCT_ANTISYM_RULE*	deduct_anitsym_rule
DISCH*	DISCH*	$\Rightarrow$_intro*	DISCH	disch_rule*
DISJ1*	DISJ1*	$\vee$_right_intro	DISJ1	disj1_rule
DISJ2*	DISJ2*	$\vee$_left_intro	DISJ2	disj2_rule
DISJ_CASES*	DISJ_CASES*	$\vee$_elim	DISJ_CASES	disj_cases_rule

Table 4: Some of the inference rule API components. Primitive rules in a given system are marked with *.

argument do not cause failure. And in ProofPower HOL, any free variables that would otherwise become equal as a result of the instantiation are renamed. None of these idiosyncrasies exist in the API version.

3.3.5 Parsing and Pretty Printing

Around 20 functions supporting parsing and pretty printing are provided in the API. This includes functions for parsing strings into HOL types and terms, and printers for types, terms and theorems. There is also support for setting the fixity of HOL functions and type operators. The fixities supported exceed what is provided by hol90, ProofPower HOL and HOL Light, but do not extend to the full range of fixities supported by HOL4. There are plans to extend the platform to support all of HOL4's fixities.

4 Implementation

4.1 Architecture

For a legacy system to conform to an API, its source code must be adapted so that every component of the API is implemented in the system. For the Common HOL API, we use a software architecture for adapting legacy HOL systems that is designed with the three goals of minimising implementation effort, enabling API-level virtualisation, and facilitating the demonstration that the adapted system exhibits precisely the same behaviour as the legacy system.

To achieve this, we choose an appropriate point in the build of the legacy system that corresponds to the level of the API (the *platform level*), and insert an ML module for the API components (the *platform module*) at this point. All legacy source code occurs either before or after the platform level (respectively called the *pre-platform* and *post-platform* code) and stays exactly the same. Keeping the pre- and post-platform code the same makes it easier to argue that the system's behaviour has not been altered.

In the platform module, we define the API in terms of pre-platform functionality. Any API components not precisely implemented as a pre-platform component must be implemented here. This includes components missing from the legacy system, or with imprecisely corresponding equivalents in the pre-platform code or that are implemented as post-platform code. For any implemented as post-platform code, the full tree of post-platform code used to define it can be shifted into the platform module, or, if this is too big, then a more succinct version can be implemented specially for the platform. The code for post-platform API components can then be deleted from its original position in the source code (thus the post-platform code remains the same except for deleted code that occurs in the platform module).

In our architecture, all post-platform code implementing primary functionality is implemented in terms of the API. This enables the API to act as a virtualisation layer through which all primary functionality is executed. This virtualisation layer can then be used for recording proofs as they are executed, before exporting them to proof objects. In order to achieve this and keep the post-platform code the same, we must somehow have a way of referring to pre-platform code that is used by post-platform code but is not in the API. We do this by implementing a module immediately after the platform module in the build that re-implements all such pre-platform code in terms of the platform, overwriting the pre-platform code. We call this the *outer platform* module.

In arguing that the system's behaviour has not altered in the API-adjusted version of the system, we must justify why any reimplementation of post-platform code in the platform module, and any reimplementation of pre-platform code in the outer platform module, preserves functionality.

Given that the API components correspond to classic basic components of a HOL system that tend to be implemented towards the start of the build of the system, finding an appropriate insertion point for the platform level tends to be fairly straightforward. It is to be found after the definition of the HOL type and term datatypes and basic utilities for manipulating them, the inference kernel, the initial theory and the parser and pretty printer. It is typically before the derived inference rules for predicate logic and the theory for pairs and natural numbers, which would need to be moved to or recreated in the platform module.

4.2 Adapting HOL Light

We now describe how we adapted HOL Light SVN release 197 to conform to the platform. The reader may find it instructive to download the adapted system [18].

The platform level in the HOL Light build file was chosen between the source files `parser.ml` and `equal.ml`. About 1,000 lines of post-platform code implementing platform components were moved into the platform module. Much of this was derived inference rules implemented using lemmas proved using HOL Light's automated proof facilities. Instead of recreating these facilities inside the platform module, we employed Common HOL proof porting to export the proofs of these lemmas as proof objects, which were then hand-translated into a total of around 400 lines of forwards style proof script in the platform module. An alternative approach was used to recreate the 13 evaluation rules for natural numeral arithmetic, whose implementation in `calc_num.ml` involves lemmas proved in hundreds of lines of proof script. Instead of exporting proof objects for these lemmas, the inference rules were given a completely different implementation in the platform module, ported from HOL Zero in about 800 lines.

About 1,000 lines of code were required to fill out platform components missing from HOL Light. For those components with an approximate equivalent already in HOL Light, the existing component was used in the implementation of the platform variant (e.g. see Figure 1), to ensure that the platform variant had roughly the same performance as the original. Those components with no approximate HOL Light version were ported from HOL Zero. In total, the components ported from HOL Zero required about 1,350 lines of supporting source code to be ported from HOL Zero, mainly involving forwards proof to prove lemmas. The platform module interface is written in about 500 lines of code.

For the outer platform, primitive inference rules and theory commands that do not correspond to platform components must be precisely recreated in terms of the platform. In HOL Light, this involves the `INST_TYPE` and `BETA` rules and all the theory commands. Also, non-platform theorems used to define platform theory needed to be recreated. In total, the outer platform required around 800 lines of code.

```
let INST_TYPE1 theta th =
  let () = if (forall (is_vartype o snd) theta)
             then failwith "INST_TYPE: Non-type-variable in instantiation domain" in
  INST_TYPE theta th;;
```

Figure 1: Using HOL Light's original INST_TYPE in the definition of the platform variant.

Overall, the platform and outer platform modules involved around 6,000 lines of source code, including the platform module interface. This took around two weeks of effort to create. The code was mostly systematically produced, being either moved from other parts of HOL Light, ported from HOL Zero, translated from proof object files, or simply a listing of platform components. The only code requiring creative thought was in the platform module variants of components with approximate equivalents already in HOL Light, and in much of the outer platform, totalling to around 1,000 lines.

5 Use Cases

In this section, we report on two use cases for the Common HOL Platform in assisting manual ports of source code between platform-adapted HOL systems. In both cases, the port was from HOL Light to HOL Zero. This is on the easy end of the difficulty spectrum in inter-HOL-system code porting, because both systems are implemented in the same dialect of ML, i.e. OCaml, and because the target system, HOL Zero, is almost a blank canvas with very little post-platform code to consider. Other HOL systems have considerable post-platform code, and porting should attempt to reuse any pre-existing code if it is straightforward to do so, to avoid creating an almost duplicate stack of supporting functionality in the target system. However, both ports described here would still be difficult without the support of the platform, and so the use cases provide useful insight.

5.1 Legacy Code Port: HOL Light Rewriting Mechanism to HOL Zero

In our first use case, we ported HOL Light's entire rewriting apparatus to HOL Zero. This is defined relatively early on in HOL Light's post-platform code, but provides vital functionality that is used throughout the rest of the system, and goes far beyond what HOL Zero is capable of in terms of proof automation. It is implemented in 360 lines of code, in the HOL Light source file `simp.ml`, and relies on 60 lines of code defining discrimination nets, and a further 300 lines of post-platform code defining supporting functionality such as conversion combinators. Thus there was a total of 720 lines to port, but this would probably be less if porting to another HOL system because it would already support conversion combinators. See Figures 2 and 3 for a sample of 32 lines from the port.

The manual port was carried out in about 2 hours 30 minutes of effort. Note that this time does not include approximately 30 minutes of effort required to extract out the 360 lines of HOL Light supporting code prior to the port. The porting itself involved systematically looking up HOL Zero equivalents of HOL Light platform functions, and renaming accordingly. HOL Light's uppercase names, that don't conform to normal OCaml lexical syntax, also needed to be converted to lowercase names. Instantiation lists, which have old-to-new ordering in HOL Zero but new-for-old ordering in HOL Light, needed to be switched around. The datatype constructors for types and terms, which are visible outside their defining module in HOL Light but not in HOL Zero, required some pattern matches to be replaced with abstract destructors and if-expressions. The function `term_match` name-clashed with a pre-existing HOL Zero function, and so was renamed to `hl_term_match`.

```
let mk_rewrites =
  let IMP_CONJ_CONV = REWR_CONV(ITAUT `p ==> q ==> r <=> p /\ q ==> r`)
  and IMP_EXISTS_RULE =
    let cnv = REWR_CONV(ITAUT `(!x. P x ==> Q) <=> (?x. P x) ==> Q`) in
    fun v th -> CONV_RULE cnv (GEN v th) in
  let collect_condition oldhyps th =
    let conds = subtract (hyp th) oldhyps in
    if conds = [] then th else
    let jth = itlist DISCH conds th in
    let kth = CONV_RULE (REPEATC IMP_CONJ_CONV) jth in
    let cond,eqn = dest_imp(concl kth) in
    let fvs = subtract (subtract (frees cond) (frees eqn)) (freesl oldhyps) in
    itlist IMP_EXISTS_RULE fvs kth in
  let rec split_rewrites oldhyps cf th sofar =
    let tm = concl th in
    if is_forall tm then
      split_rewrites oldhyps cf (SPEC_ALL th) sofar
    else if is_conj tm then
      split_rewrites oldhyps cf (CONJUNCT1 th)
        (split_rewrites oldhyps cf (CONJUNCT2 th) sofar)
    else if is_imp tm & cf then
      split_rewrites oldhyps cf (UNDISCH th) sofar
    else if is_eq tm then
      (if cf then collect_condition oldhyps th else th)::sofar
    else if is_neg tm then
      let ths = split_rewrites oldhyps cf (EQF_INTRO th) sofar in
      if is_eq (rand tm)
      then split_rewrites oldhyps cf (EQF_INTRO (GSYM th)) ths
      else ths
    else
      split_rewrites oldhyps cf (EQT_INTRO th) sofar in
  fun cf th sofar -> split_rewrites (hyp th) cf th sofar;;
```

Figure 2: A sample of legacy source code from HOL Light's `simp.ml`.

HOL Light non-conformant versions of platform functions, such as its `variant` function, required special attention. Unlike the platform equivalent, this function does not fail if its avoidance list contains non-variables, and so the code was adapted to either filter them out or check that non-variables are not possible from program context. Other complications included two uses of HOL Light's intuitionistic tautology prover, `ITAUT`. It was decided to keep this function outside the scope of the port, despite it being used to prove two lemmas, to reduce the amount of supporting code. For the HOL Zero version, one of the lemmas already existed in HOL Zero's small library of predicate logic theorems, and the other was proved in 10 minutes in a 16-line proof using HOL Zero's forward inference rules.

After the port was completed, it was tested on various rewriting examples, and one error was found. This took 45 minutes of debugging to track down and correct, and was due to a quirk in the failure exception returned by HOL Light's `rev_assoc` function, which has error message text `"find"` (instead of `"rev_assoc"`). This particular error message was explicitly trapped in the HOL Light code, but naively porting this to HOL Zero didn't work because its equivalent function, `inv_assoc`, uses error message text `"inv_assoc"`. As explained in Section 3.3.1, this aspect of porting is not catered for by the platform, and must be done manually.

```
let mk_rewrites =
  let imp_conj_conv = rewr_conv imp_imp_thm
  and imp_exists_rule =
    let cnv = rewr_conv imp_exists_rule_thm in
    fun v th -> conv_rule cnv (gen_rule v th) in
  let collect_condition oldhyps th =
    let conds = subtract (asms th) oldhyps in
    if conds = [] then th else
    let jth = foldr disch_rule conds th in
    let kth = conv_rule (repeatc imp_conj_conv) jth in
    let cond,eqn = dest_imp(concl kth) in
    let fvs = subtract (subtract (free_vars cond) (free_vars eqn))
                       (list_free_vars oldhyps) in
    foldr imp_exists_rule fvs kth in
  let rec split_rewrites oldhyps cf th sofar =
    let tm = concl th in
    if is_forall tm then
      split_rewrites oldhyps cf (spec_all_rule th) sofar
    else if is_conj tm then
      split_rewrites oldhyps cf (conjunct1_rule th)
        (split_rewrites oldhyps cf (conjunct2_rule th) sofar)
    else if is_imp tm & cf then
      split_rewrites oldhyps cf (undisch_rule th) sofar
    else if is_eq tm then
      (if cf then collect_condition oldhyps th else th)::sofar
    else if is_not tm then
      let ths = split_rewrites oldhyps cf (eqf_intro_rule th) sofar in
      if is_eq (rand tm)
      then split_rewrites oldhyps cf (eqf_intro_rule (gsym_rule th)) ths
      else ths
    else
      split_rewrites oldhyps cf (eqt_intro_rule th) sofar in
  fun cf th sofar -> split_rewrites (asms th) cf th sofar;;
```

Figure 3: The translation into HOL Zero of the legacy code sample from `simp.ml`.

5.2 New Code Port: HOL Light Proof Importer to HOL Zero

In the second use case, we used the platform to port HOL Light's importer for Common HOL proof objects. This was a fundamentally easier exercise because the proof importer is written specifically in terms of the API, and because Common HOL proof porting works at the level of platform inference rules itself. The proof importer is implemented in 2,200 lines of code.

It took about 1 hour 15 minutes to perform the porting. Despite the source code being three times longer than in the legacy code port, it took only half the time. The easier nature of the task meant that everything went smoothly first time. The effort consisted almost entirely of systematically applying search-and-replace to replace HOL Light platform function names with HOL Zero equivalents and carrying out manual adjustments for functions that take their arguments differently in the different systems.

The resulting source code was tested by importing into HOL Zero the text formalisation part of the Flyspeck project, as part of a partial audit of the project as described in [2]. This involved the tens of millions of platform-level inference rule steps. The import into HOL Zero worked first time, suggesting the code was ported correctly.

6 Conclusions

In defining a standard for basic theory and programming components, the Common HOL Platform is attempting to lay the foundation for much better portability between HOL systems, both in terms of porting proofs and porting source code. The feasibility of large scale proof porting has already been established by others, but arguably there is scope for doing better still, given a better foundation. However, the feasibility of quick and reliable source code porting has not been explored until now.

In this paper, we have given an overview of the platform's components and explained the reasons behind some of the careful design decisions made. We have also demonstrated using the platform in two use cases of manually porting source code from HOL Light to HOL Zero, one for legacy code and one for new code written specially for the platform. In both cases, several hundred lines of code were successfully and reliably ported within a few hours. Much of the effort normally involved in a manual port is removed, because almost all that needs to be considered is functionality implemented above the platform level. Finding corresponding low-level components in the two systems, and the subtle ways in which they can differ, has already been taken care of by the platform. As far as we are aware, this represents a leap in the productivity of source code porting between HOL systems, even when accounting for it being less challenging than the general porting case due to both systems being implemented in the same dialect of ML and due to HOL Zero effectively being a blank canvas.

It would be interesting to see how far HOL source code porting could be pushed. Certainly it is feasible to port more challenging parts of HOL Light to HOL Zero. Obvious candidates are the subgoal package, the intuitionistic tautology checker and the powerful `MESON_TAC`. Implementing the latest version of the platform for hol90, HOL4 and ProofPower HOL, and porting to these systems is another challenge worth pursuing. The platform has already been designed with these systems in mind, and it would at least enable Common HOL proof exporters and importers to be quickly ported to these systems.

One insight that comes from looking at code from the various HOL systems is how much the subgoal package is used in the implementation of other parts of HOL systems, suggesting that it should be part of the API. This should be a fairly easy extension to make, since beyond the implementation of an initial few tactics, code using it appears to operate at the abstract level using tacticals, rather than use the inner workings that differ between HOL systems. Another change worth making is to update the platform for the reform to primitive theory extension currently underway in various HOL systems [4]. And finally, catering for Isabelle/HOL must be a long term priority. This would probably require a significant overhaul of the platform to fit with such a different system, but if done well it would pay dividends to have good portability between the widest used HOL system and the rest of the family.

The systematic manner in which the porting can be carried out lends itself to automation, or at least to partial automation. The most difficult to automate is probably the intelligent use of the target system's legacy supporting code to avoid the ugly situation of creating two parallel stacks of code implementing effectively the same thing. Thus partial automation looks a more realistic prospect. We believe there are no fundamental difficulties in automatically porting between ML dialects, because the subsets of ML that tend to be used in the implementation of HOL systems are trivially corresponding between OCaml and SML. So we see there being good prospects for reducing further the time taken to reliably port source code, even in more challenging cases.

References

[1] M. Adams & P. Clayton (2005): *ClawZ: Cost-Effective Formal Verification for Control Systems*. In: *Proceedings of the 7th International Conference on Formal Methods and Software Engineering*, *Lecture Notes*

in Computer Science 3785, Springer, pp. 465–479, doi:10.1007/11576280_32.

[2] M. Adams (2015): *Proof Auditing Formalised Mathematics*. Available at `http://www.proof-technologies.com/flyspeck/qed_paper.pdf`. Accepted for publication in the Journal of Formalized Reasoning.

[3] R. Arthan & R. Jones (2005): *Z in HOL in ProofPower*. In Issue 2005-1 of the British Computer Society Specialist Group Newsletter on Formal Aspects of Computing Science.

[4] R. Arthan (2014): *HOL Constant Definition Done Right*. In: *Proceedings of the 5th International Conference on Interactive Theorem Proving*, *Lecture Notes in Computer Science* 8558, Springer, pp. 531–536, doi:10.1007/978-3-319-08970-6_34.

[5] M. Gordon & T. Melham (1993): *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press.

[6] M. Gordon, R. Milner & C. Wadsworth (1979): *Edinburgh LCF: A Mechanised Logic of Computation*. *Lecture Notes in Computer Science* 78, Springer, doi:10.1007/3-540-09724-4.

[7] T. Hales et al. (2015): *A Formal Proof of the Kepler Conjecture*. Preprint available at `arxiv.org`. ArXiv:1501.02155v1 [math.MG].

[8] J. Harrison (2009): *HOL Light: An Overview*. In: *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, *Lecture Notes in Computer Science* 5674, Springer, pp. 60–66, doi:10.1007/978-3-642-03359-9_4.

[9] J. Hurd (2011): *The OpenTheory Standard Theory Library*. In: *Proceedings of the Third International Symposium on NASA Formal Methods*, *Lecture Notes in Computer Science* 6617, Springer, pp. 177–191, doi:10.1007/978-3-642-20398-5_14.

[10] C. Kaliszyk & A. Krauss (2013): *Scalable LCF-Style Proof Translation*. In: *Proceedings of the 4th International Conference on Interactive Theorem Proving*, *Lecture Notes in Computer Science* 7998, Springer, pp. 51–66, doi:10.1007/978-3-642-39634-2_7.

[11] G. Klein et al. (2009): *seL4: Formal Verification of an OS Kernel*. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ACM, pp. 207–220, doi:10.1145/1629575.1629596.

[12] T. Nipkow, L. Paulson & M. Wenzel (2002): *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. *Lecture Notes in Computer Science* 2283, Springer, doi:10.1007/3-540-45949-9.

[13] S. Obua & S. Skalberg (2006): *Importing HOL into Isabelle/HOL*. In: *Proceedings of the Third International Joint Conference on Automated Reasoning*, *Lecture Notes in Computer Science* 4130, Springer, pp. 298–302, doi:10.1007/11814771_27.

[14] L. Paulson (1987): *Logic and Computation: Interactive proof with Cambridge LCF*. Cambridge University Press, doi:10.1017/CBO9780511526602.

[15] K. Slind (1991): *An Implementation of Higher Order Logic*. Technical Report 91-419-03, Computer Science Department, University of Calgary.

[16] K. Slind & M. Norrish (2008): *A Brief Overview of HOL4*. In: *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, *Lecture Notes in Computer Science* 5170, Springer, pp. 28–32, doi:10.1007/978-3-540-71067-7_6.

[17] M. Wenzel, L. Paulson & T. Nipkow (2008): *The Isabelle Framework*. In: *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, *Lecture Notes in Computer Science* 5170, Springer, pp. 33–38, doi:10.1007/978-3-540-71067-7_7.

[18] *HOL Light adaptation for Common HOL*. Available at `http://www.proof-technologies.com/commonhol/commonhol-0.5-hl-svn197.tgz`.

[19] *HOL Zero homepage*. Available at `http://www.proof-technologies.com/holzero/`.