Fast machine words in Isabelle/HOL

Andreas Lochbihler

Institute of Information Security, Department of Computer Science, ETH Zurich, Switzerland
andreas.lochbihler@inf.ethz.ch

Abstract. Code generated from a verified formalisation typically runs faster when it uses machine words instead of a syntactic representation of integers. This paper presents a library for Isabelle/HOL that links the existing formalisation of words to the machine words that the four target languages of Isabelle/HOL's code generator provide. Our design ensures that (i) Isabelle/HOL machine words can be mapped soundly and efficiently to all target languages despite the differences in the APIs; (ii) they can be used uniformly with the three evaluation engines in Isabelle/HOL, namely code generation, normalisation by evaluation, and term rewriting; and (iii) they blend in with the existing formalisations of machine words. Several large-scale formalisation projects use our library to speed up their generated code. To validate the unverified link between machine words in the logic and those in the target languages, we extended Isabelle/HOL with a general-purpose testing facility that compiles test cases expressed within Isabelle/HOL to the four target languages and runs them with the most common implementations of each language. When we applied this to our library of machine words, we discovered miscomputations in the 64-bit word library of one of the target-language implementations.

1 Introduction

Nowadays, algorithms are routinely verified formally using proof assistants and many proof assistants support the generation of executable code from the formal specification. The generated code is used for animating the formal specification [10, 38, 41, 45], validating the formal models [16, 18, 39], proving properties by evaluation [1, 21, 23, 48], and to obtain actual tools with formal guarantees such as CompCERT [37], CakeML [30], CeTA [49], CAVA [15], Cocon [28], DRAT-trim [25], and GRAT [34].

Usability of the generated code requires that it be efficient. This is mainly achieved by using (i) optimised data structures, which have been verified in the proof assistant, and (ii) hardware support for computing with data, in particular integers and arrays. To that end, the code generators of many proof assistants can be configured to map types and their operations to those provided by the target language rather than to implement them according to their construction in the logic. For example, integers can use optimised libraries like GMP instead of being implemented as lists of binary digits and arrays are translated to read-only arrays with constant-time access instead of lists with linear-time access. In today's practice, such mappings are often unverified (an exception is CakeML's verified bignum library [30]) and are therefore part of the trusted code base (TCB). As we discuss below, verified code generation could shrink the TCB, but it has not yet reached maturity.

Apart from efficiency, such mappings bridge the gap between formal logic and the real world. The mapped data types are used to exchange data with non-verified code, e.g., drivers, application interfaces, test harnesses, and foreign function interfaces (FFI) in

general [42, 44]. The proof assistant Isabelle/HOL in version Isabelle2017, e.g., provides the necessary mappings for arbitrary-precision integers, booleans, lists, and strings.[1]

In this paper, we extend this list for Isabelle/HOL with machine words of 8, 16, 32, and 64 bits (§3), and with machine words of unspecified size (§4). By reusing Isabelle/HOL's formalisation of fixed-size words [11, 12], our library inherits the infrastructure for reasoning about machine words and integrates smoothly with existing formalisations. The key challenge was to simultaneously support all target languages of Isabelle/HOL's code generator (Standard ML, OCaml, Haskell, and Scala) with their varying APIs and all evaluation mechanisms (code generation, normalisation by evaluation, and term rewriting). Supporting all target languages and all evaluators is crucial to obtain a usable and versatile library that works together with many other Isabelle/HOL libraries.

We have validated our unverified mappings by running many test cases. To that end, we have developed a general-purpose framework for Isabelle/HOL to run and test the generated code (§5.1). After we had fixed the initial mistakes in our mappings, our test cases even found a bug in the implementation of 64-bit words in PolyML 5.6.1 in 64-bit mode, which is the Standard ML implementation that runs Isabelle2017 (§5.2).

Our library is available on the Archive of Formal Proofs [40], which includes a user guide as documentation. Several projects and tools use it already. Users report significant performance improvements over using arbitrary-precision integers (§6). The testing framework is distributed with Isabelle2017 (file `HOL-Library.Code_Test`).

Contributions. The main contributions of this paper are the following:

1. We describe the design of an Isabelle/HOL library for fixed-size words that are mapped to machine words in different target languages. By using our library, users can generate faster code from their formalisations.
2. We analyse the pitfalls and subtleties of code adaptations and show how to ensure that code adaptations work for all target languages and all evaluators. Our library demonstrates the feasibility of our approach. This analysis is of interest even to other proof assistants that provide code extraction: libraries similar to ours suffer from such subtle soundness bugs although they target just one language, not four (§7).
3. To justify the soundness of our mapping, we generalise the correctness notion for code generation such that logical underspecification can be refined during code generation. We argue that the new notion is meaningful and identify conditions under which it coincides with the existing correctness notion. Such refinements can also be used in other contexts where abstract types are implemented by concrete data structures.
4. We develop a general-purpose framework for running and testing the generated code, which can be used independently of our machine word library. For example, it can compute with infinite codatatype values using Haskell's built-in laziness. The existing ML-based evaluation mechanism does not terminate for such computations.

Design choices. Our goal is to develop a practical and efficient library suitable for large-scale projects, not a fragile research prototype. Thus, it must work with the technology

[1] Immutable arrays are supported for Standard ML and Haskell, but not the other target languages of Isabelle/HOL's code generator. In the version for Isabelle2017, the Collections framework by Lammich [33] provides mutable arrays for Standard ML, Haskell, and Scala, but not OCaml.

that is already mature. In our case, this is Isabelle's existing code generator with the four target languages and its unverified mappings, which inflate the trusted code base. Although we cannot obtain formal guarantees on the generated code itself, it *is* generated from a verified formalisation in a systematic way supported by a sound theory. So our library merely adds the correctness of the (validated) mappings to the TCB, which already includes the compiler and library of the target language anyway.

The alternative would be to target the ongoing work on verified code generation such as CertiCoq [2], Œuf [43], and CakeML [30] with its Isabelle link [27]. Our mappings could then be verified down to assembly language or machine code and would thus not enlarge the TCB. Given the present state of these projects, such a library would be less versatile than ours. For example, the Isabelle link to CakeML lacks abstract datatypes, which many Isabelle/HOL projects use for code generation. Moreover, even CakeML, the most mature of the three, produces machine code that is often slower than the output of unverified compilers, although the run times' orders of magnitude are about equal [47].

More importantly, our approach will still be relevant when such mappings will be verified in the future, as the key challenge of fitting different APIs under one hood will persist. The reason is that there will be several verified compilation chains, e.g., CertiCoq and Œuf for Coq, and a versatile library should support code generation with all of them. Clearly, careful API design can avoid some of the differences, e.g., signed vs. unsigned words. But others like the varying word sizes will remain as they reflect crucial design choices in the compilation chain. In Standard ML, e.g., the word size varies by compiler precisely because each compiler organises the heap in its own way, stealing some bits of every word for memory management. A performant library must deal with such compiler-specific issues, as library users should not have to care about these details.

2 Background on Isabelle/HOL

This section introduces aspects of the proof assistant Isabelle/HOL that are relevant for this paper. Isabelle/HOL implements an extension of classical higher-order logic (HOL) [46] with Haskell-style type classes for overloading [22]. Its standard library formalises machine words [11, 12] as a HOL type α word where the type parameter α determines the number of bits via a type-class operation $\mathsf{len\text{-}of}_\alpha :: \mathsf{nat}$. More precisely, α word is defined as a copy of the integers from 0 to $2^{\mathsf{len\text{-}of}_\alpha} - 1$. For example, 32 word and 64 word denote the type of 32-bit and 64-bit words, respectively, using an encoding of numbers as types. So, the arithmetic and bit-wise operations on words are derived from those on the integers, i.e., the results are truncated by taking the remainder w.r.t. $2^{\mathsf{len\text{-}of}_\alpha}$. Technically, these operations are overloaded for integers and words using type classes.

The Lifting and Transfer tools [26] can lift definitions and transfer theorems across quotients. We use them for the special case of subtypes (typedefs in HOL), e.g., from integers to words. In this case, a lifted definition is executable if the original term is.

The code generator [20, 21] generates code from a fragment of HOL to functional programming languages, mapping HOL types and functions to datatypes and functions in the target language. Four languages are supported: Standard ML, OCaml, Haskell, and Scala. The code generator ensures partial correctness of the generated code. That is, if the code terminates successfully, then the result satisfies the properties that have been proven about the HOL functions. This guarantee relies on the code generator's assumption that

the generated functional program behaves according to a higher-order rewrite system (HORS). In this view, datatype constructors are uninterpreted function symbols and the equations of a function yield a set of rewrite equations. Executing the generated program in the target language corresponds to performing rewrite steps with the corresponding equations on the term representation. Since the equations have counterparts in HOL, all these steps could also have been taken in the logic, so the result is derivable in HOL. Conversely, nothing can be said if the execution raises an exception or does not terminate.[2]

Moreover, this approach decouples the logical definitions from the extracted code, as the HORS does not attach logical meaning to the function symbols themselves. Any HOL function of the right type can thus serve as a datatype constructor and any HOL equation can be used to implement a function if the constraints of the target language are met. They are therefore called code constructors and code equations. For example, one can change the implementation of nat to a binary representation without changing the definition in the logic or the proofs. This corresponds to data refinement [20].

Isabelle/HOL's code generator also provides a minimalistic foreign function interface (FFI) via **code-printing** declarations [19, §7]. These declarations instruct the code generator to output a specified string instead of what it would normally generate for a HOL type or HOL function in the specified target language. As they act on the concrete syntax, such declarations are called code adaptations. They are used to map integers,[3] booleans, lists, unit, and strings to their counterparts in the target language. Code adaptations lack a formal semantics and are therefore part of the TCB.

The code equations can also be used to evaluate HOL terms and to prove theorems by execution. Isabelle/HOL has three different mechanisms to do so: (i) generating and running Standard ML code for ground terms and propositions; (ii) symbolic normalisation by evaluation (NBE) [1]; and (iii) term rewriting within Isabelle. The first mechanism uses the full power of the code generator, mapping HOL types to Standard ML data types and functions to SML functions using the code equations and the code adaptations. NBE represents HOL values as a term data type in Standard ML and HOL functions as Standard ML functions that manipulate terms according to code equations; no code adaptations are used. Term rewriting uses only the code equations in a call-by-value strategy. Note that the same set of code equations is used for all target languages and for all mechanisms.

Only term rewriting is checked by Isabelle's kernel and can thus be trusted. When the other two evaluation mechanisms are used, code generation and possibly the code adaptations become part of the TCB. In return, they are much faster than term rewriting. When proving theorems, Isabelle tags all theorems whose derivation involved some step outside of the kernel, such as NBE or code generation. So everyone can easily check whether Isabelle's kernel has completely checked all steps of a theorem's derivation.

With the existing Isabelle/HOL setup for α word, the generated code represents words as arbitrary-precision integers and all operations take the remainder modulo $2^{\mathsf{len\text{-}of}_\alpha}$. This sets the efficiency baseline for evaluating our library.

[2] Non-termination does not affect logical soundness as the function definitions' consistency in HOL must have been established independently of the code generator.

[3] Isabelle/HOL provides two types of integers: int and integer. The latter is always mapped to target-language integers and the former can be implemented using the latter. Here, we ignore this distinction and always assume that integers are implemented by target-language integers.

3 Fixed-Size Machine Words

We now introduce HOL types for words of 8, 16, 32, and 64 bits (§3.1), present the code adaptations for all target languages (§§3.2–3.3), and argue why they are sound (§3.4).

3.1 Types of Unsigned Words

Recall that the type α word of Isabelle/HOL words is polymorphic in the number of bits. Yet, code adaptations can only be given for type constructors such as word, not compound types like 32 word. As target languages provide only monomorphic word types, we must map 32 word to a different target language type than say 64 word. We therefore first introduce new HOL type constructors for (unsigned) machine words of 8, 16, 32, and 64 bits. In detail, we define types uint8, uint16, uint32, and uint64 as type copies of the existing unsigned word formalisation. As the construction is identical for all bit lengths, we only show the one for uint32 and use uint$*$ to refer to all four types.

All arithmetic and bit-wise operations are lifted from 32 word to uint32 using the Lifting tool [26]. Cast operations between all uint$*$ types are also available. Here, we give just two examples: the overloaded addition operation $(+)$ and the conversion function word-of-int from integers.

```
lift-definition (+) :: uint32 ⇒ uint32 ⇒ uint32
  is (+) :: 32 word ⇒ 32 word ⇒ 32 word .

lift-definition uint32-of-int :: int ⇒ uint32 is word-of-int .
```

In principle, we could easily transfer all the existing theorems about these operations, too. But our library does not do so as we consider uint$*$ primarily as types for code generation, not for proving theorems. Instead, whenever we must prove a theorem about uint$*$, we first transfer the statement to α word (for the appropriate choice of α) using Transfer and then use the existing, well-engineered proof automation. This approach avoids duplicating theorems and tactics and thus saves the subsequent maintenance efforts.

3.2 Setting up Code Generation

With the uint$*$ types and their operations in place, we can now design the code constructors, code equations, and code adaptations. Our design should achieve three goals:

1. It should work simultaneously for all four target languages, all three evaluation mechanisms, and all strategies of Quickcheck [5]. Recall that the code constructors and equations are shared by all target languages and evaluation mechanisms. So we must find constructors and equations that are suitable for all of them.
2. The code adaptations for the uint$*$ operations should yield very efficient code.
3. The adaptations should be as small as possible to reduce the chance of errors.

In case of conflicting goals, we will value the efficiency of the target language mapping higher than the efficiency of the other evaluators (normalisation and term rewriting). The evaluators are typically used only for small HOL programs, where efficiency is not as crucial as in generated applications.

To support evaluation by normalisation and term rewriting, we design our code equations such that they implement uint$*$ in terms of α word, which in turn is implemented

using arbitrary-precision integers. In detail, we declare uint$*$ as abstract datatypes to the code generator [20], such that code equations cannot pattern match on the code constructor for uint$*$. This ensures that code equations respect the abstraction barrier of uint$*$, so that we can later change the generated code using adaptations without worrying that users of our library might have declared code equations that look into the internal construction of uint$*$. In fact, all this setup is already in place by the way we have defined uint$*$ and their operations using the Lifting tool. Moreover, the conversion function uint32-of-int from integers to uint32 acts as the "smart constructor" to create values of type uint32.

Next, we describe the code adaptations that map the uint$*$ types and functions to the target language primitives. Yet, the provided word types vary across the target languages and even across different implementations of the same language. Table 1 lists the available word sizes for the most common implementations of the four target languages (marked with $\surd$). As can be seen, the support varies widely: only 32-bit words are provided by all implementations. PolyML provides 64-bit words only when run in 64-bit mode. For OCaml and Scala, most word types provide only signed operations, which interpret the most significant bit as a sign (marked as grey cells). Following α word, our library provides unsigned words, so extra effort will be needed in these cases. The last row shows the bit widths of the languages' standard word type. We will look at this row in more detail in §4.

bits	PolyML 32	PolyML 64	SMLNJ	mlton	OCaml 32	OCaml 64	GHC	Scala
8	$\surd$	$\surd$	$\surd$	$\surd$			$\surd$	$\surd$
16				$\surd$			$\surd$	$\surd$
32	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
64		$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
?	31	63	31	32	31	63	≥ 30	32

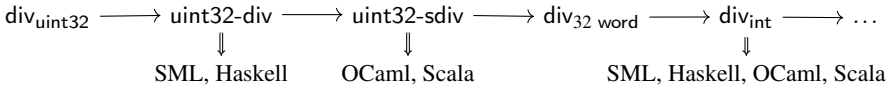
Table 1. Bounded integers in the standard library by target language. Supported fixed sizes are marked with $\surd$. The last row lists the bit size of the default type. Grey cells indicate that only signed operations are available.

The code adaptations for the types and most operations are straightforward as the libraries provide suitable functions. The type uint32, e.g., is mapped as follows. In the remainder of this section, we discuss the non-trivial cases.

```
code_printing type_constructor uint32 →
  (SML) Word32.word  (OCaml) int32  (Haskell) Data.Word.Word32  (Scala) Int
```

If a target language does not provide a particular bit width (e.g., 8 and 16 bits in OCaml), we omit the code adaptations. The generated code will thus follow the code equations that the evaluators use. So, 8- and 16-bit words are implemented in OCaml using arbitrary-precision Big_ints, taking the remainder w.r.t. 2^8 or 2^{16} after every operation. With some more effort, they could also be implemented using 32-bit words.

Division and remainder require a more elaborate design of the code equations, which the drawing below illustrates. We define a cascade of constants div, uint32-div, uint32-sdiv, ... that model the division operators of the different target languages. Code equations ($\longrightarrow$) implement each constant using the next one. Code adaptations ($\Longrightarrow$) map the constants to right target languages; they thereby terminate the cascade early.



We now look at the different constants implementing division. As is customary in HOL, division by 0 yields 0 and taking the remainder w.r.t. 0 is the identity function [24], but the target languages typically raise exceptions. To avoid the exceptions and thus make the generated code fail less often, we define a new constant uint32-div that is unspecified for 0 and add 0 as a special case to div's code equation (and the remainder's):

definition uint32-div $x\ y = (\mathsf{if}\ y = 0\ \mathsf{then}\ \mathsf{undefined}\ (\mathsf{div})\ x\ 0\ \mathsf{else}\ x\,\mathsf{div}\,y)$

lemma [code]: $(x\,\mathsf{div}\,y) = (\mathsf{if}\ y = 0\ \mathsf{then}\ 0\ \mathsf{else}\ \text{uint32-div}\ x\ y)$

Here, undefined is an unspecified, polymorphic HOL constant. By applying it to the div function and the arguments x and 0, we get a fresh, unspecified formal value $x/0$ for every x. This way, mapping uint32-div to target language operations remains sound even if these return different results for dividing different x by 0—provided that the same value is consistently returned for the same x, if any. [4] For example,

code-printing constant uint32-div $\rightarrow$
(SML) `Word32.div (_, _)` (Haskell) `Prelude.div`

where `(_, _)` expresses that Standard ML's `Word32.div` takes both arguments as a tuple.

Unfortunately, mapping uint32-div to OCaml's and Scala's division operations directly would be unsound, as OCaml's `int32` and Scala's `Int` are signed. Therefore, we define another division operation uint32-sdiv on uint32 that interprets uint32 as signed words and coincides with uint32-div when a division by zero occurs. Next, we prove a code equation that implements uint32-div using uint32-sdiv. The following equation expresses the algorithm adapted from Hacker's Delight [50, §9.3] on α word, where $\ll$ and $\gg$ denote unsigned bit shifts to the left and right, and sdiv denotes signed division. We prove the equation for all x and y of type α word with $y \neq 0$, and then lift it to all the uint$*$ types. This is possible thanks to the polymorphic α word.

$$\begin{aligned}(x\,\mathsf{div}\,y, x\,\mathsf{mod}\,y) = (&\mathsf{if}\ 1 \ll (\mathsf{len\text{-}of}_\alpha - 1) \leq y\ \mathsf{then}\ \mathsf{if}\ x < y\ \mathsf{then}\ (0,x)\ \mathsf{else}\ (1, x-y)\\ &\mathsf{else}\ \mathsf{let}\ q = ((x \gg 1)\ \mathsf{sdiv}\ y) \ll 1;\ r = x - q * y\ \mathsf{in}\\ &\quad \mathsf{if}\ r \geq y\ \mathsf{then}\ (q+1, r-y)\ \mathsf{else}\ (q,r))\end{aligned}$$

Thus, we get the following OCaml and Scala code adaptations for division. Note that there are no code adaptations for uint32-div for OCaml and Scala.

code-printing constant uint32-sdiv $\rightarrow$ (OCaml) `Int32.div` (Scala) `_ / _`

The cascade of constants also applies to evaluation by normalisation and term rewriting, as they use the same code equations. Since there are no code adaptations, they follow the cascade until the end, i.e., arithmetic on integers. That is, they perform division and remainder on uint32 by testing for the zero divisor and only then performing

[4] Alternatively, we could have (under-)specified uint32-div with a conditional definition like

definition uint32-div where $y \neq 0 \longrightarrow \text{uint32-div}\ x\ y = x\,\mathsf{div}\,y$

lemma [*code*]: $\text{uint32-div}\ x\ y = (\mathsf{if}\ y = 0\ \mathsf{then}\ \mathsf{Code.abort}\ \text{"Div0"}\ (\lambda\_.\ \text{uint32-div}\ x\ y)\ \mathsf{else}\ x\,\mathsf{div}\,y)$

As the precondition makes the defining equation unsuitable for code generation, we would have to manually state and derive an unconditional code equation like the one shown, with which division by zero would make the normalisation evaluator fail to terminate. The definition with undefined requires no further setup for code generation and does not cause non-termination.

a *signed* division according to the given algorithm, which is implemented via 32 word and the arbitrary-precision integers. This is an example of where we accept inefficiencies in the evaluators in favour of better generated code. Accordingly, the same roundabout way of implementing division also applies for uint$*$ types that are not supported natively by the target language. In OCaml, e.g., 8- and 16-bit words follow the cascade until the code adaptations for arbitrary-precision integers branch off to OCaml's `Big_int` library.

The other operations affected by the signed interpretation are dealt with in a similar way. The smart constructor uint32-of-int :: int $\Rightarrow$ uint32, in particular, requires adjusting the integer range from HOL's 0 to $2^{32}-1$ to OCaml's and Scala's -2^{31} to $2^{31}-1$. Like for division and remainder, we state and verify a conversion algorithm for arbitrary bit lengths as a lemma on α word and lift it to uint$*$ using the Transfer package.

This simple idea of a cascade of constants with selective code adaptations yields more efficient code than what Isabelle code generation experts had come up with previously. Traditionally, code adaptations identified a domain on which the implementations in all target languages behave the same. The division and remainder operations on arbitrary-precision integers in Isabelle/HOL's standard library illustrate this approach. They are not directly mapped to the target language operations because they differ on negative numbers: dividing -5 by 3, e.g., yields -1 in Scala and OCaml whereas it results in -2 in Haskell and Standard ML (and Isabelle/HOL). Isabelle's standard library instead defines a special division-modulo operation divmod-abs that first takes absolute values and serialises it to target-language expressions that do the same.

definition divmod-abs $m\ n = (|m| \operatorname{div} |n|, |m| \bmod |n|)$

code-printing divmod-abs $\rightarrow$
(SML) `IntInf.divMod (IntInf.abs _, IntInf.abs _)`
(Haskell) `divMod (abs _) (abs _)`

and similarly for OCaml and Scala. The original division and remainder operations are implemented using divmod-abs where signs and values for negative numbers are adjusted as necessary. This approach clearly is not optimal with respect to efficiency, as some computations such as taking the absolute value are performed twice, once in the code equation for div (and mod) and once again in the code adaptation. In particular, those operations are computed even if the target language's operations exactly fit Isabelle's (like in the case of Haskell and Standard ML). For PolyML 5.6.1, we measured that the overhead of these checks and additional operations is about 100%, i.e., a division operation takes twice as long as it would have to. So users have to pay the performance penalty even if they are not interested in generating code in languages with mismatching operations.

In contrast, our cascading approach has no overhead for languages with perfectly matching operations (Standard ML and Haskell) and much less overhead for the others, where we have precisely modelled the target language operations in the logic and verified the implementation. The same could be done for division and remainder on integers.

3.3 Dealing with Underspecification

The bit shift operations $<<$, $>>$, and $>>>$ (right shift with sign extension) are not affected by the signed interpretation, but they behave differently in different target languages. In Scala, they only take the lower bits of the shift into account. For example, shifting 1 by

65 bits to the left as a uint32 yields 2, as the lower $5 = \log_2 32$ bits of 65 denote the value 1. In Haskell and OCaml, the result of these operations is unspecified when the shift is negative or exceeds the word size. In Standard ML, the bit shift operations correctly honour all bits of the shift, but the shift must be given as a `Word`, whose size varies with the implementation (as shown in the last row in Table 1). In Isabelle, however, the shifts are specified as (unbounded) natural numbers, so we must take overflows into account.

Given the underspecification in Haskell and OCaml, we cannot model the target language's bit shifts exactly in HOL, as we do for sdiv. Instead, we resort to underspecification in HOL, too. For each shift operation, we define a version which is specified only for the bit shifts that do not exceed the word size. For $\ll$ on uint32, e.g., we define

$$\text{uint32-shiftl}\ x\ i = (\text{if}\ i < 0 \vee i \geq 32\ \text{then undefined}\ (\ll)\ x\ i\ \text{else}\ x \ll \text{nat}\ i)$$

where we model the underspecification using undefined as we did for uint32-div in §3.2. We prove a code equation for $\ll$ (and one for uint32-shiftl for the evaluators)

$$x \ll n = (\text{if}\ n < 32\ \text{then uint32-shiftl}\ x\ (\text{int}\ n)\ \text{else}\ 0),$$

where nat and int convert between integers and natural numbers, and map uint32-shiftl directly to the target languages.

3.4 Soundness of Code Adaptations for Underspecified HOL Functions

Recall from §2 that the HORS view on code generation assumes that the successful execution steps of the generated program corresponds to rewrite steps in HOL. This guarantees partial correctness of the generated code. Clearly, code adaptations violate this invariant. Fortunately, we can generalise the reasoning to code adaptations for fully specified HOL functions, by assuming that there is a HOL proof tactic that can justify the result of a successful execution of the mapped code. This assumption can either be validated using tests (§5) or by giving a formal semantics to the generated code and verifying the translation [30]. For our library, this approach works for all the arithmetic operations, as the only such underspecified operations are division and remainder, where in the underspecified cases the generated code fails with an exception.

Unfortunately, this argument does not carry over to the bit shifts described in §3.3. Clearly, evaluating $1 \ll 65$ in, say, Scala does return a specific value—namely 2—and there is no way to prove that the unspecified HOL value undefined $(\ll)$ 1 65 equals 2. The code adaptations thus tighten the specification, i.e., they correspond to a kind of refinement. We now describe the correctness guarantees obtained by such an implicit refinement and identify the necessary assumptions on the target language operations.

The set-theoretic semantics of HOL assigns arbitrary values of the right type to unspecified constants, i.e., constants that have been declared, but not (yet) defined [32]. We can therefore consider the underspecification of a function as picking sufficiently many freshly declared constants and returning one of them for each argument where the underspecification occurs. Skolemizing over all the arguments and even the intended HOL function, we end up with an equivalent specification, e.g., the family $\lambda x\ i.$ undefined $(\ll)$ x i of unspecified uint32 values. We can view this underspecification as model-theoretic non-determinism, which code adaptations can refine. Like deferred Isabelle/HOL definitions of constants that have been declared earlier, a code adaptation conceptually defines the family of unspecified values as the values that the

target language implementation will compute. Clearly, these definitions are only conceptual, because they never manifest as a definitional theorem that Isabelle's kernel could check. Moreover, the chosen values depend on the particular target language implementation that will run the generated code. In this view, code adaptations constitute a deferred definition mechanism that executes when code is generated and whose effect is revoked at the end of code generation (as these definitions are not recorded in the logic).

This interpretation shows that any result computed by the generated code must be a possible value in *some* HOL model. Assuming that the formalisation is consistent, we obtain a (weaker) version of partial correctness, namely every theorem provable in HOL applies to the result. This is because the theorems hold in all HOL models and the result lives in one of them. Yet, we can no longer argue that the result is derivable from the HOL definitions, i.e., that *all* HOL models enforce this result. In other words, the generated code can only produce results which are *consistent* with the formalisation, but not necessarily *enforced* by it. In summary, we obtain the guarantee that it is impossible to prove in HOL that the result violates any provable property of the formalisation.

Our correctness argument hinges on three requirements, which our library meets:

1. The unspecified values are indeed logically unspecified. Otherwise, the refinement can lead to inconsistencies.
2. The function computed by the code adaptation in the target language implementation must be definable in HOL. In particular, the function must be pure, i.e., consistently return the same result for the same arguments, independent of the calling context, and its HOL definition must not introduce cyclic dependencies [31]. Obviously, it must also coincide with the mapped HOL function on the domain where it is specified.[5]
3. The code must not be used to prove theorems in the logic. Theorems proven by the refined code could silently introduce the implicit refinements as axioms into the logic. That is, some theorems might actually not be derivable from the stated axioms.

The last requirement means that implicit refinement via code adaptations must not be used when we prove theorems by code generation. The proofs of the code equations for the bit shift operations show that their results do not depend on the unspecified behaviour of the auxiliary functions like uint32-shiftl, i.e., we can use these operations in proofs by evaluation. However, users might directly call these auxiliary functions with unintended arguments (e.g., uint32-shiftl 1 2^{32}). To be safe, we ensure that in the code target Eval, which is used for proving theorems, code adaptations never cause implicit refinements. We achieve this by explicitly checking whether the arguments lie in the specified domain and otherwise raise an exception. For example,

```
code-printing constant uint32-shiftl → (Eval)
  (fn x => fn i => if i < 0 orelse i >= 32 then raise (Fail "<<")
                   else Word32.<<(x, Word.fromLargeInt (IntInf.toLarge i)))
```

[5] The bit shifts are underspecified only in Haskell and OCaml. In Haskell, this assumption is satisfied as the bit shift operations belong to the Safe Haskell subset where pure functions cannot have side effects, i.e., referential transparency holds. As OCaml maps bit shifts directly to C, the interpretation of undefined behaviour would allow to the compiler to violate this assumption. However, to our knowledge, none of the state-of-the-art compilers exploits such technically undefined bit shifts badly. They all map it consistently to some bit shift instructions on the hardware, which does meet our requirements. The compilation strategy can change in the future though.

Admittedly, it might have been easier to include the range checks for the shift operations in the code adaptations of all targets, not just Eval. This would have saved us from implicit refinements and their implications on soundness, at the cost of two more integer comparisons per executed bit shift. But in the next section, we take underspecification to the level of types, where we cannot avoid it any more.

4 Machine Words of Unspecified Length

Words of 8, 16, 32, and 64 bits are not optimally efficient for all target languages. Some implementations offer words of 31 and 63 bits, which are implemented more efficiently as they need not boxing in memory. They use the missing bit to distinguish between primitive values and pointers, exploiting that the lowest bit of a pointer is always 0 due to memory alignment constraints. Accordingly, the bit length also depends on whether the runtime runs in 32-bit or 64-bit mode. The last row in Table 1 shows these bit widths by implementation. The Haskell API specifies only a lower bound of 30 bits; GHC in version 7.6.3 provides 32 bits in 32-bit mode and 64 bits in 64-bit mode. This may change in future versions, e.g., if the memory management starts to use some bits of a processor words for tagging like PolyML and OCaml do. The table therefore shows only the API constraint.

In this section, we introduce a type uint that maps to these machine words in the target languages. Generated code can thus benefit from unboxing, i.e., run faster with less memory. As the exact bit width varies across target language, implementation, and architecture, we again resort to underspecification in HOL to achieve sound code adaptations. That is, uint denotes the type of all machine words of a given non-zero length, but we do not specify the length in HOL. Formally, we introduce an uninterpreted type default-size and specify that $\mathsf{len\text{-}of}_{\mathsf{default\text{-}size}}$ be some positive number.[6] Then, uint denotes the type of all words of length $\mathsf{len\text{-}of}_{\mathsf{default\text{-}size}}$, which the code generator maps to `Word.word` in Standard ML, `Data.Word.Word` in Haskell, `int` in OCaml, and `Int` in Scala.

typedecl default-size

specification $\mathsf{len\text{-}of}_{\mathsf{default\text{-}size}} > 0$ **by** auto

typedef uint = UNIV :: default-size word set ..

code-printing `type-constructor` uint →
(SML) `Word.word` (Haskell) `Data.Word.Word` (OCaml) `int` (Scala) `Int`

The operations and code adaptations for uint are analogous to uint$*$, as described in §3. Signed and underspecified operations are handled in the same way, too. We map $\mathsf{len\text{-}of}_{\mathsf{default\text{-}size}}$ to the target language's bit width, e.g., `Word.wordSize` in Standard ML.

[6] Technically, the command **specification** defines the constant using Hilbert choice ε and derives the given property, after the specification has been shown to be satisfiable (**by** auto). So some unintended equations about $\mathsf{len\text{-}of}_{\mathsf{default\text{-}size}}$ are provable, e.g., $\mathsf{len\text{-}of}_{\mathsf{default\text{-}size}} = (\varepsilon x.\ x > 0)$. To avoid violating requirement 1 from §3.4, we hide the defining equation and only work with the specification. Arthan [4] discusses the problem of unintended identities for underspecified constants in detail.

The underspecification for uint is much more invasive than uint$*$'s. For the latter, only a few auxiliary operations like uint32-shiftl are underspecified, but all of the official operations are fully specified. On uint, in contrast, we do not even know what number $3*5$ denotes. For example, $3*5=7$ holds in HOL models where $\mathsf{len\text{-}of}_{\mathsf{default\text{-}size}}=3$. Evaluation by code generation therefore does not make sense for uint and our code adaptations ensure that all uint operations always raise exceptions in the evaluation target Eval. It might be possible to configure the other evaluators (normalisation and term rewriting) such that they evaluate uint expressions symbolically, but we have not succeeded in doing so yet. Therefore, evaluation and proving theorems by execution is currently not supported for uint.

So, what can be done with those unspecified uint? Here are three useful applications. First, Lammich [33] has implemented bit vectors as a list of uint. He formalises bit vectors on polymorphic words α word, making no assumptions about α. For example, the n-th bit of the bit vector is stored in the $(n \bmod \mathsf{len\text{-}of}_\alpha)$-th bit of the $(n \operatorname{div} \mathsf{len\text{-}of}_\alpha)$-th list element. So, $v\,!\,(n \operatorname{div} \mathsf{len\text{-}of}_\alpha)\,!!\,(n \bmod \mathsf{len\text{-}of}_\alpha)$ looks up the n-th bit in the bit vector v, where $l\,!\,i$ returns the i-th element of the list or array l. Then, he lifts his formalisation to uint using the Transfer tool. Thus, the generated code adapts the size of the list to the target language implementation.

Second, hashing does not rely on the exact size of the values. Algorithms based on hashing deal with clashes anyway, so their correctness does not depend on the exact hash values. Yet, hashing must be fast. Taking uint for hash values enables such fast hashing.

Third, finite rings $\mathbb{Z}/p\mathbb{Z}$ can be implemented via uint if $p^2 < 2^{\mathsf{len\text{-}of}_{\mathsf{default\text{-}size}}}$, which can be tested dynamically. We evaluate such an implementation in §6.

5 Validation

The code adaptations in our library are rather complicated, with many subtleties and corner cases. It is therefore imperative to validate the code adaptations. In theory, as all word types are finite, we could certify the code adaptations by running the generated code for all possible argument values and checking that the mapped HOL term evaluates to the same result (unless it is unspecified). In practice, this might be feasible for uint8 and uint16, but the argument space for 32- and 64-bit words is too large. Therefore, we content ourselves by running selected test cases.

In this section, we present a generic-purpose testing framework in Isabelle/HOL (§5.1) and the design and results of our validation (§5.2).

5.1 Automating Regression Tests for Code Generation

To automate the testing, we have developed a general-purpose testing tool for Isabelle/HOL's code generator, which is distributed with Isabelle2017 (theory `HOL-Library.Code_Test`). Our tool provides a new command **`code_test`** that takes a list of test cases and a list of target language implementations. A test case is any boolean HOL term. The supported target language implementations are PolyML, MLton, SMLNJ, GHC, OCaml, and Scala. For each target language implementation, the command performs five steps:

1. It generates code for all the test cases in the corresponding target language.
2. It produces a test harness tailored to the target language implementation.

3. If necessary, it compiles the generated code and the test harness.
4. It executes all test cases by running the (compiled) program.
5. It reports which test cases have succeeded or failed, and for the failed ones, it outputs the evaluation result for selected subterms, e.g., the two sides of (in)equalities.

If code generation or any of the test cases fails, the command raises an error in Isabelle/HOL, which makes it suitable for regression testing.

For example, the following invocation tests that our code adaptations correctly use Scala's signed division on `bytes` for computing the unsigned fraction:

code-test $251\,\mathsf{div}\,3 = (83 :: \mathsf{uint8})$ `in` Scala

In case of a failure, **code-test** outputs to what the left and right-hand side have evaluated in Scala. To that end, **code-test** also generates code for reifying the result value as a HOL term in the target language. This HOL term is then serialised as a YXML string in the same format that Isabelle/PIDE uses to communicate with the prover process [51]. Term reification is shared with the counter-example generator Quickcheck [8, §3.3.4], so it automatically works for most user-defined types, in particular all (co)datatypes.

The different target language implementations are modularly supported by drivers. A driver gets as input (i) the directory for the code, (ii) the names of the generated files, and (iii) the name of the generated function that executes all test cases. The driver outputs (i) the names and contents of its test harness files, and (ii) bash commands for compiling and running the code and the test harness.

Drivers must be registered with our tool under an identifier, e.g., PolyML and MLton, and with an associated code target, e.g., SML. The tool then takes care of all the rest, such as parsing the user's input, invoking Isabelle/HOL's code generator, writing all files to a fresh temporary directory, compiling and running the program, and showing the pretty-printed result to the user. Thus, users can easily write and register their own drivers when they want to test other implementations.

5.2 Test Case Selection and Validation Results

As is common practice, we partition the argument values into equivalence classes and select only one representative from each equivalence class. For $\mathsf{uint}*$, we consider the three classes $\{0,\ldots,2^{l-1}-1\}$, $\{2^{l-1},\ldots,2^l-1\}$, and $\{2^l,\ldots\}$, where l denotes the bit length of the word type. For bit indices, we choose the classes $\{0,\ldots,l-2\}$, $\{l-1\}$, and $\{l,\ldots\}$. The most significant bit $l-1$ has its own class because of the signed operations.

We have run all these test cases with all implementations and all evaluators. In fact, the test cases are routinely run by the regression test system of the Archive of Formal Proofs. This ensures that incompatible changes in Isabelle/HOL's code generator configuration are quickly detected.

During the development of our library, the test cases revealed many errors in the code adaptations, both syntactic and semantic errors, e.g., forgetting appropriate casts in Scala to counter the automatic promotion to `Int`. Of course, we have addressed all the errors and now all test cases pass. This indicates that our test cases are reasonable.

Surprisingly, the tests did not only reveal errors in our code adaptations. For PolyML 5.6.1, which Isabelle2017 runs on, one of our tests on 64-bit words failed when PolyML

runs in 64-bit mode. The problem is that PolyML's `Word64` structure does not correctly implement division. For example, `Word64.div(0wxFFFFFFFFFFFFFFFB, 0wx3)` evaluates to `0wx55555553` instead of `0wx5555555555555553`. The error occurs only in 64-bit mode because PolyML does not provide a `Word64` structure in 32-bit mode. Meanwhile, Matthews has implemented the `Word64` structure differently in PolyML 5.7, thereby eliminating the bug. Isabelle2017 itself is not affected by the error because its implementation does not use 64-bit words. To support evaluation of uint64 terms in Isabelle2017, our library tests at load time whether the underlying 64-bit PolyML system provides the incorrect `Word64` structure and—if so—generates a replacement based on arbitrary-precision integers.

6 Evaluation

We have been developing our library of machine words since 2013. Meanwhile, it has been picked up by several other users in their projects. This shows that our library is usable. Moreover, we can evaluate the performance by looking at real-world use cases instead of unrealistic micro-benchmarks. In this section, we describe how the projects used our library and comment on the performance impact we are aware of. For one project, we also ran the benchmarks to measure the performance impact of our library ourselves.

Divason et al. [14] have verified the Berlekamp-Zassenhaus algorithm for factoring polynomials over the integers. The algorithm factors a given polynomial over the finite rings $\mathbb{Z}/p^k\mathbb{Z}$ for $k = 1,2,4,8,\ldots$ using Berlekamp's algorithm and Hensel's lifting lemma. Zassenhaus' algorithm then reconstructs the factorisation over the integers. Divason et al. have parametrised the factorisation algorithm and the Hensel lifting over the arithmetic operations. So they can choose the most efficient implementation dynamically according to the following strategy. If $p^k < 2^{16}$, all computations are done in uint32 as multiplying two 16-bit numbers stays below 2^{32}. If $2^{16} \leq p^k < 2^{32}$, their implementation uses uint64. Otherwise, arbitrary-precision integers are used.

To quantify the performance gain by using uint$*$, we ran three versions of the factorisation algorithm (generated in Haskell from AFP version 70d9faada9d0). The first version omits the range checks for p^k and always uses arbitrary-precision integers. This establishes the baseline. The second version chooses the implementation type according to the above strategy. The third version uses uint if $p^k < \sqrt{2^{\mathsf{len\text{-}of}_{\mathsf{default\text{-}size}}}}$ and arbitrary-precision integers otherwise. We used the benchmarks by Divason et al. [14]: 400 randomly generated polynomials with 100 to 500 coefficients. The measurements were performed on an Intel i7 quad core at 2.4 GHz with 16 GB RAM running Ubuntu 14.04 LTS. The generated code was compiled with GHC 7.6.3 with option `-O2`.

Factoring all 400 polynomials using arbitrary-precision integers took 33.70 min in total. Using uint32/uint64 reduces the time to 27.42 min, i.e., a reduction by 18.6 %. Per polynomial, the time reduction ranged between 10.0 % and 39.9 % with median 19.9 % and relative standard deviation 5.9 %. This shows that our library consistently provides better efficiency than computing with GMP integers, despite the additional range checks. The difference between uint32/uint64 and uint was insignificant. This is because GHC 7.6.3 always boxes machine words (`Data.Word.Word`). To measure the effect of boxing, we also ran the second and third version with PolyML 5.7.1 in 64-bit mode, where uint is only 63 bits, but unboxed: on average, uint is 4.0 % faster than uint32/uint64.

Fleury et al. [17] are developing a verified SAT solver using Isabelle/HOL. For efficiency reasons, `uint32` words are used for propositional variables and literals, where the positive and negative literals of a variable v are given by $2 \cdot v$ and $2 \cdot v + 1$, respectively. Both literals of a variable can thus be computed efficiently using bit operations. Fleury told us in personal communication that switching from GMP integers to `uint32` improved performance of the generated Standard ML code considerably. Bit shifts on GMP integers are apparently significantly slower, even if the values fit in GMP's small integers.

The Isabelle Collections framework and the Monadic Refinement framework [33, 35, 36] use our library for implementing hash functions, on which verified hash sets and hash arrays build. Using these frameworks, Esparza et al. [15] have generated an LTL model checker in Standard ML from their formalisation. They observed a speed-up of one order of magnitude when they changed hashing from arbitrary-precision integers to our library.

Lochbihler and Züst [42] obtain a Haskell implementation of the TLS protocol generated from Isabelle/HOL. Unlike in the other projects, they use `uint`$*$ not for efficiency reasons, but for exchanging data with foreign Haskell functions and for constructing the protocol messages. The socket API functions take arguments that are machine words of 8, 16, or 32 bits, and some fields in the protocol messages also have such bit lengths.

7 Related Work

Many proof assistants provide libraries for fixed-size words. Those that support code generation to machine integers are all tailored to one particular target language, usually the language the prover is implemented in. In contrast, our library shows how to fit the varying APIs of four target languages into one library while retaining efficiency.

The coq-bits library [6, 29] by Blot et al. models signed 8-, 16-, and 32-bit words in Coq. Using Coq's code adaptation command **`Extract Inlined Constant`**, the library maps all word types to OCaml's `int` type. They program exhaustive test cases in Coq and prove that the test cases suffice to establish that the translation is correct. But they run the test cases only for 8- and 16-bit words, as exhaustively testing 32- or 64-bit words is impractical. Thereby, they have missed that their mapping is unsound for 32-bit integers when Ocaml runs in 32-bit mode as `int` has only 31 significant bits then.

Armand et al. [3] added OCaml's 31-bit machine integers to Coq's evaluation engine, which is comparable to Isabelle/HOL's normalisation evaluator [1]. Théry [48] relies on them to establish by evaluation that the Mini-Rubik cube can always be solved in at most 11 steps. While we have made sure that our library supports evaluation, the normalisation evaluator uses the symbolic representation for the `uint`$*$ types. Changing this representation to Standard ML machine words would require a complete re-design of the evaluator since it does not support any form of code adaptation. Our mappings therefore need not be trusted for normalisation. Regarding execution times, code extraction is much faster than normalisation in Isabelle anyway and even more so with our library.

Maude provides fixed-size words similar to Isabelle's Word library [9, §9.5]. Yet, they are not mapped to machine words, but emulated using arbitrary-precision integers.

Greve et al. [18] describe how ACL2 code can be written in such a way that the underlying LISP compiler uses unboxed machine words (`fixnum`) instead of arbitrary-precision integers. They annotate their code with many declarations that restrict the

allowed integer range to signed 32 bit words. ACL2's guard checker accordingly demands a proof that the range is respected. Divason et al. [14] had to prove similar respectfulness theorems when they implemented the $GF(p^k)$ operations on the `uint`$*$ types. Most proofs were automatic using the Transfer package and the existing theorems for α `word`. Like for Haskell, the exact range of `fixnum` in LISP is implementation-defined; at least 16 bits are required. Greve et al. ignore this issue and assume that at least 32 bits are provided.

PVS's ground evaluator generates LISP code from PVS specifications. It also supports unchecked code adaptations, which are called semantic attachments [10]. Muñoz' library PVSio [44] provides semantic attachments for, among others, floating point arithmetic, which replaces exact arithmetic on reals. Semantic attachments cannot be used to prove theorems by ground evaluation to prevent inconsistencies, e.g., due to rounding errors. Isabelle/HOL's code generator allows code adaptations for proofs. We therefore carefully craft the adaptations for the target Eval and raise exceptions in underspecified cases.

The problem of refining underspecified functions for code generation is also addressed by the Isabelle Monadic Refinement framework [36] and its Coq counterpart Fiat [13]. In both frameworks, programs must be written in a non-determinism monad. They can then be refined within the logic towards a deterministic implementation. This refinement approach could be used to model the non-determinism due to the different bit sizes in the various target language implementations. Users would however have to write all their functions in the monad and refine the non-determinism way before code generation. This would severely impair the usability of our library. We therefore opted for model-theoretic refinement and accepted that this refinement is unverified.

8 Conclusion and Future Work

We have presented a library for efficiently computing with machine words of 8, 16, 32, and 64 bits in Isabelle/HOL. It distinguishes itself from other such libraries in that it simultaneously supports all four target languages of Isabelle's code generator and all of Isabelle's evaluation mechanisms. Thus, formalisations based on our library do not have to commit to a particular language and can instead be used in any Isabelle context. We achieve this flexibility using a model-theoretic refinement semantics for code adaptations. To validate our library, we have developed a general-purpose regression test framework for Isabelle/HOL and tested the correctness of our code adaptations. Our library has successfully boosted the performance of the generated code in several projects.

We have also used the test framework to obtain HOL evaluators in Haskell, OCaml, and Scala. Haskell in particular is useful as its lazy evaluation semantics handles infinite codatatype values, on which the existing call-by-value evaluators do not terminate.

Our code adaptations are unverified—like all code adaptations for Isabelle/HOL. The adaptations and the machine word implementations in the target languages are therefore in the trusted code base whenever our library is used for code generation. This applies to (i) tools obtained by code generation and (ii) proofs by evaluation. As Isabelle tags all theorems whose proof has not been checked by the kernel, users can always check whether a theorem has gone through the kernel. If they do not want to trust the adaptations, they can always prove their theorems by term rewriting (or normalisation).

We will add more word types, e.g., signed words, on demand. While their formalisation is very easy thanks to the length-polymorphic Word library, getting the code adaptations right requires a careful study of the language specifications.

When the projects on verified code generation reach maturity, we hope to formally verify our mappings to reduce the TCB. In the meantime, it would be interesting to systematize the test case generation, e.g., by model-driven testing as implemented in HOL-Testgen [7]. We could validate the code adaptations further and check whether target language implementations correctly implement the operations. In this scenario, our library is only the starting point. Other libraries like Yu's formalisation of IEEE floating point numbers [52] could also benefit from validation. Although testing can never formally establish the correctness of code adaptations, it is a very practical approach to ensuring soundness.

Acknowledgements. Peter Lammich contributed an initial formalisation of machine words of unspecified length. Rafael Häuselmann helped to implement the **`code-test`** command. René Thiemann and Mathias Fleury encouraged us to develop the library further. The author was supported by the Swiss National Science Fund under grant 153217.

References

1. K. Aehlig, F. Haftmann, and T. Nipkow. A compiled implementation of normalisation by evaluation. *J. Funct. Program.*, 22(1):9–30, 2012.
2. A. Anand, A. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. Savary Belanger, M. Sozeau, and M. Weaver. CertiCoq: A verified compiler for Coq. In *CoqPL 2017*, 2017.
3. M. Armand, B. Grégoire, A. Spiwack, and L. Théry. Extending Coq with imperative features and its application to SAT verification. In M. Kaufmann and L. C. Paulson, editors, *ITP 2010*, volume 6172 of *LNCS*, pages 83–98. Springer, 2010.
4. R. Arthan. On definitions of constants and types in HOL. *Journal of Automated Reasoning*, 56(3):205–219, 2016.
5. J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic proof and disproof in Isabelle/HOL. In C. Tinelli and V. Sofronie-Stokkermans, editors, *FroCoS 2011*, volume 6989 of *LNCS*, pages 12–27. Springer, 2011.
6. A. Blot, P.-É. Dagand, and J. Lawall. From sets to bits in Coq. In O. Kiselyov and A. King, editors, *FLOPS 2016*, volume 9613 of *LNCS*, pages 12–28. Springer, 2016.
7. A. D. Brucker and B. Wolff. HOL-TestGen: An interactive test-case generation framework. In M. Chechik and M. Wirsing, editors, *FASE 2009*, volume 5503 of *LNCS*, pages 417–420, Berlin, Heidelberg, 2009. Springer.
8. L. Bulwahn. *Counterexample Generation for Higher-Order Logic Using Functional and Logic Programming*. PhD thesis, Fakultät für Informatik, Technische Universität München, 2013.
9. M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.
10. J. Crow, S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. Evaluating, testing, and animating PVS specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 2001.
11. J. Dawson. Isabelle theories for machine words. In M. Goldsmith and B. Roscoe, editors, *AVOCS 2007*, volume 250(1) of *ENTCS*, pages 55–70. Elsevier, 2009.
12. J. Dawson, P. Graunke, B. Huffman, G. Klein, and J. Matthews. Machine words in Isabelle/HOL. http://isabelle.in.tum.de/dist/library/HOL/HOL-Word/document.pdf, 2017.

13. B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *POPL 2015*, pages 689–700, New York, NY, USA, 2015. ACM.
14. J. Divasón, S. Joosten, R. Thiemann, and A. Yamada. A formalization of the Berlekamp-Zassenhaus factorization algorithm. In *CPP 2017*, pages 17–29, New York, NY, USA, 2017. ACM.
15. J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J.-G. Smaus. A fully verified executable LTL model checker. In N. Sharygina and H. Veith, editors, *CAV 2013*, volume 8044 of *LNCS*, pages 463–478. Springer, 2013.
16. A. Farzan, J. Meseguer, and G. Roşu. Formal JVM code analysis in JavaFAN. In C. Rattray, S. Maharaj, and C. Shankland, editors, *AMAST 2004*, volume 3116 of *LNCS*, pages 132–147. Springer, 2004.
17. M. Fleury, J. C. Blanchette, and P. Lammich. A verified SAT solver with watched literals using imperative HOL. In *CPP 2018*, pages 158–171. ACM, 2018.
18. D. Greve, M. Wilding, and D. Hardin. High-speed, analyzable simulators. In M. Kaufmann, P. Manolios, and J. Strother Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, volume 4 of *Advances in Formal Methods*, pages 113–135. Springer, 2000.
19. F. Haftmann. Code generation from Isabelle/HOL theories. http://isabelle.in.tum.de/dist/Isabelle2017/doc/codegen.pdf, 2017.
20. F. Haftmann, A. Krauss, O. Kunčar, and T. Nipkow. Data refinement in Isabelle/HOL. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP 2013*, volume 7998 of *LNCS*, pages 100–115. Springer, 2013.
21. F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *FLOPS 2010*, volume 6009 of *LNCS*, pages 103–117. Springer, 2010.
22. F. Haftmann and M. Wenzel. Constructive type classes in Isabelle. In T. Altenkirch and C. McBride, editors, *TYPES 2006*, volume 4502 of *LNCS*, pages 160–174. Springer, 2007.
23. T. C. Hales, J. Harrison, S. McLaughlin, T. Nipkow, S. Obua, and R. Zumkeller. A revision of the proof of the Kepler conjecture. *Discrete & Computational Geometry*, 44(1):1–34, 2010.
24. J. Harrison. *Theorem proving with the real numbers*. Springer, London, 1998.
25. M. Heule, W. Hunt, M. Kaufmann, and W. Nathan. Efficient, verified checking of propositional proofs. In M. Ayala-Rincón and C. A. Muñoz, editors, *ITP 2017*, pages 269–284. Springer, 2017.
26. B. Huffman and O. Kunčar. Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In *CPP 2013*, volume 8307 of *LNCS*, pages 131–146. Springer, 2013.
27. L. Hupel and T. Nipkow. A verified compiler from Isabelle/HOL to CakeML. In *ESOP 2018*, LNCS. Springer, 2018. To appear.
28. S. Kanav, P. Lammich, and A. Popescu. A conference management system with verified document confidentiality. In A. Biere and R. Bloem, editors, *CAV 2014*, volume 8559 of *LNCS*, pages 167–183. Springer, 2014.
29. A. Kennedy, N. Benton, J. B. Jensen, and P.-E. Dagand. Coq: The world's best macro assembler? In *PPDP 2013*, pages 13–24, New York, NY, USA, 2013. ACM.
30. R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *POPL 2014*, pages 179–191, New York, NY, USA, 2014. ACM.
31. O. Kunčar. Correctness of Isabelle's cyclicity checker: Implementability of overloading in proof assistants. In *CPP 2015*, pages 85–94, New York, NY, USA, 2015. ACM.
32. O. Kunčar and A. Popescu. A consistent foundation for Isabelle/HOL. In C. Urban and X. Zhang, editors, *ITP 2015*, volume 9236 of *LNCS*, pages 234–252, Cham, 2015. Springer.
33. P. Lammich. Collections framework. *Archive of Formal Proofs*, 2009. http://isa-afp.org/entries/Collections.html, Formal proof development.

34. P. Lammich. The GRAT tool chain. In S. Gaspers and T. Walsh, editors, *SAT 2017*, volume 10491 of *LNCS*, pages 457–463. Springer, 2017.
35. P. Lammich and A. Lochbihler. The Isabelle collections framework. In M. Kaufmann and L. C. Paulson, editors, *ITP 2010*, volume 6172 of *LNCS*, pages 339–354. Springer, 2010.
36. P. Lammich and T. Tuerk. Applying data refinement for monadic programs to Hopcroft's algorithm. In L. Beringer and A. Felty, editors, *ITP 2012*, volume 7406 of *LNCS*, pages 166–182. Springer, 2012.
37. X. Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.
38. H. Liu and J. S. Moore. Executable JVM model for analytical reasoning: A study. In *IVME 2003*, pages 15–23. ACM, 2003.
39. A. Lochbihler. *A Machine-Checked, Type-Safe Model of Java Concurrency : Language, Virtual Machine, Memory Model, and Verified Compiler*. PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, July 2012.
40. A. Lochbihler. Native word. *Archive of Formal Proofs*, 2017. http://devel.isa-afp.org/entries/Native_Word.html, Formal proof development.
41. A. Lochbihler and L. Bulwahn. Animating the formalised semantics of a Java-like language. In M. van Eekelen, H. Geuvers, J. Schmalz, and F. Wiedijk, editors, *ITP 2011*, volume 6898 of *LNCS*, pages 216–232. Springer, 2011.
42. A. Lochbihler and M. Züst. Programming TLS in Isabelle/HOL. Isabelle Workshop 2014, http://www.andreas-lochbihler.de/pub/lochbihler14iw.pdf, 2014.
43. E. Mullen, S. Pernsteiner, J. R. Wilcox, Z. Tatlock, and D. Grossman. Œuf: Minimizing the Coq extraction TCB. In *CPP 2018*, pages 172–185. ACM, 2018.
44. C. Muñoz. Rapid prototyping in PVS. Contractor Report NASA/CR-2003-212418, NASA, Langley Research Center, Hampton VA 23681-2199, USA, 2003.
45. T. Nipkow. Teaching semantics with a proof assistant: No more LSD trip proofs. In V. Kuncak and A. Rybalchenko, editors, *VMCAI 2012*, volume 7148 of *LNCS*, pages 24–38. Springer, 2012.
46. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
47. S. Owens, M. Norrish, R. Kumar, M. O. Myreen, and Y. K. Tan. Verifying efficient function calls in CakeML. In *ICFP 2017*, volume 1 of *Proc. ACM Program. Lang.*, pages 18:1–18:27. ACM, 2017.
48. L. Théry. Proof pearl: Revisiting the Mini-Rubik in Coq. In O. Ait Mohamed, C. Muñoz, and S. Tahar, editors, *TPHOLs 2008*, volume 5170 of *LNCS*, pages 310–319. Springer, 2008.
49. R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs 2009*, volume 5674 of *LNCS*, pages 452–468. Springer, 2009.
50. H. S. Warren. *Hacker's Delight*. Addison-Wesley, 2 edition, 2012.
51. M. Wenzel. Isabelle as document-oriented proof assistant. In J. H. Davenport, W. M. Farmer, J. Urban, and F. Rabe, editors, *CICM/MKM 2011*, LNAI, pages 244–259, Berlin, Heidelberg, 2011. Springer.
52. L. Yu. A formal model of IEEE floating point arithmetic. *Archive of Formal Proofs*, 2013. http://isa-afp.org/entries/IEEE_Floating_Point.html, Formal proof development.