CHALMERS
UNIVERSITY OF TECHNOLOGY

Fast, Verified Computation for HOL ITPs

Journal of Automated Reasoning (2025) 69:7
https://doi.org/10.1007/s10817-025-09719-8

Fast, Verified Computation for HOL ITPs

Oskar Abrahamsson[1] · Magnus O. Myreen[1,3,4] · Michael Norrish[2] · Hrutvik Kanabar[3] · Johannes Åman Pohjola[1,4]

Received: 29 April 2024 / Accepted: 10 January 2025

Abstract
We add an efficient function for computation to the kernels of higher-order logic interactive theorem provers. First, we develop and prove sound our approach for Candle. Candle is a port of HOL Light which has been proved sound with respect to the inference rules of its higher-order logic; we extend its implementation and soundness proof. Second, we replicate our now-verified implementation for HOL4 with only minor changes, and build additional automation for ease of use. The automation exists outside of the HOL4 kernel, and requires no additional trust. We exercise our new computation function and associated automation on the evaluation of the CakeML compiler backend within HOL4's logic, demonstrating an order of magnitude speedup. This is an extended version of our previous conference paper [2], which described implementation and soundness proofs for Candle. Our HOL4 implementation and automation are new, as are the CakeML benchmarks.

Keywords Prover soundness · Higher-order logic · Interactive theorem proving

1 Introduction

Interactive theorem provers (ITPs) include facilities for computing within the hosted logic. To illustrate what we mean by such a feature, consider the following function, `sum`, which sums a list of natural numbers:

✉ Johannes Åman Pohjola
pohjola@chalmers.se

Oskar Abrahamsson
oskar.abr@icloud.com

Magnus O. Myreen
myreen@chalmers.se

Michael Norrish
michael.norrish@anu.edu.au

Hrutvik Kanabar
hrutvik.kanabar@arm.com

1 Chalmers University of Technology, Gothenburg, Sweden

2 Australian National University, Canberra, Australia

3 Arm Ltd, Cambridge, UK

4 University of Gothenburg, Gothenburg, Sweden

Published online: 18 February 2025

$$\mathsf{sum}\ xs \stackrel{\mathsf{def}}{=} \mathsf{if}\ xs = [\,]\ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{hd}\ xs + \mathsf{sum}\ (\mathsf{tl}\ xs)$$

A facility for computing within the logic can be used to automatically produce theorems such as the following, where sum [5; 9; 1] was given as input, and the following equation is the output, showing that the input reduces to 15:

$$\vdash \mathsf{sum}\ [5;\ 9;\ 1] = 15 \tag{1}$$

The ability to compute such equations in ITPs is essential for use of verified decision procedures, for proving ground cases in proofs, and for running a parser, pretty printer or even compiler inside the logic for a smaller trusted computing base (TCB).

Higher-order logic (HOL) does not have a primitive rule for (or notion of) computation. Instead, HOL ITPs such as HOL Light [13], HOL4 [20], and Isabelle/HOL [18] implement computation as a derived rule using rewriting, which in turn is a derived rule implemented outside their trusted kernels. As a result, computation is slow in these systems.

To understand why computation is so sluggish in HOL ITPs, it is worth noting that the primitive steps taken for the computation of Example (1) are numerous:

- At each step, rewriting has to match the subterm that is to be reduced next (according to a call-by-value order) against each pattern it knows (the left-hand side of the definitions of sum, hd, tl, if-then-else and more); when a match is found, it needs to instantiate the equation whose left-hand-side matched, and then reconstruct the surrounding term.
- Computation over natural numbers is far from constant-time, since 5, 9 and 1 are syntactic sugar for numerals built using the constructor-like functions and constants: Bit0, Bit1 and 0. For example, $5 = \mathsf{Bit1}\ (\mathsf{Bit0}\ (\mathsf{Bit1}\ 0))$. Deriving equations describing the evaluation of simple operations such as + requires rewriting with lemmas such as these:

$$\begin{aligned}
&\mathsf{Bit1}\ \mathsf{m} + \mathsf{Bit0}\ \mathsf{n} = \mathsf{Bit1}\ (m + n)\\
&\mathsf{Bit1}\ \mathsf{m} + \mathsf{Bit1}\ \mathsf{n} = \mathsf{Bit0}\ (\mathsf{Suc}\ (m + n))\\
&\mathsf{Suc}\ (\mathsf{Bit0}\ \mathsf{n}) = \mathsf{Bit1}\ \mathsf{n}\\
&\mathsf{Suc}\ (\mathsf{Bit1}\ \mathsf{n}) = \mathsf{Bit0}\ (\mathsf{Suc}\ n)\\
&\ldots
\end{aligned}$$

HOL ITPs employ such laborious methods for computation in order to keep their soundness-critical kernel as small as possible: the small size and simplicity of the kernel is key to the soundness argument.

1.1 Fast, Verified Computation

We want fast computation which does not compromise the soundness of slower approaches, and remains user-friendly.

First, we develop our fast computation feature for Candle. Candle is a port of HOL Light which has been proved sound (in HOL4) with respect to a formal semantics of higher-order logic [1]. By extending this proof, we can increase the size and complexity of Candle's kernel without compromising soundness.

Second, we copy this verified fast computation feature to HOL4's kernel, and implement automation outside of HOL4's kernel to ease usage of our new feature. The fast computation feature modifies HOL4's kernel, but is trustworthy because we have proved it sound for Candle's kernel and copied it over with only minor changes; the automation does not modify HOL4's kernel, so it requires no additional trust.

1.1.1 Candle: Implementation and Verification of Fast Computation

This first part of this paper (Sects. 2–8) is about how we have added a fast function for computation to the Candle HOL ITP[1] and updated Candle's soundness proof.

With this new function for computation, proving equations via computation is cheap. For the sum example:

- The input term is traversed once, and is converted to a datatype better suited for fast computation. In this representation, each occurrence of sum, hd, tl, etc. can be expanded directly without pattern-matching.
- The representation makes use of host-language integers, so $5 + (9 + (1 + 0))$ can be computed using three native addition operations.
- Once the computation is complete, the result is converted back to a HOL term and an equation similar to (1) is returned to the user.

Our function for computation works on a first-order, untyped, monomorphic subset of higher-order logic. Our implementation interprets terms of this subset using a call-by-value strategy and host-language (CakeML [15]) features such as arbitrary precision integer arithmetic.

In our experiments, we observe speed gains of several orders of magnitude when comparing Candle's new compute function against established in-logic computation implementations used by other HOL ITPs (Sect. 8).

1.1.2 HOL4: Automated Translation into Code Equations for Fast Computation

The second part of this paper (Sects. 9–12) is about how we have added the same fast computation to the kernel of the HOL4 ITP, and implemented automated tooling outside the kernel to improve its usability. This tooling allows users to invoke fast computation without writing their functions in the first-order subset. In particular, our automation:

1. Automatically translates ordinary input to the first-order subset outside the kernel.
2. Applies the fast kernel function for computation.
3. Automatically translates the result back to ordinary HOL4 outside the kernel.

The addition of the fast computation feature modifies HOL4's kernel, but all of our automation is implemented outside the kernel, requiring no additional trust. In fact, the automation is *proof-producing*: on each invocation, it constructs a theorem like (1), equating the input HOL4 term to the result of its fast computation. We evaluate our automation by applying it to previously written, ordinary HOL4 functions: the implementation of the CakeML compiler backend [22].

1.2 Contributions

We make the following contributions:

- We implement a fast interpreter for terms as a user-accessible primitive in the Candle *and HOL4* kernels. The implementation allows users to supply code equations dictating how user-defined (recursive) functions are to be interpreted.

[1] Kernel functions are analogous to inference rules in HOL implementations.

- We prove the new primitive correct with respect to the inference rules of Candle's higher-order logic, and fully integrate it into the existing end-to-end soundness proof of the Candle ITP.
- *We implement an automatic translator from ordinary HOL4 user definitions to functions operating over the datatype expected by the new kernel function.*
- We show that our compute function is significantly faster than in-logic compute facilities provided by other HOL ITPs. *Within HOL4, it speeds up significant benchmarks by an order of magnitude: in-logic execution of the CakeML type inferencer, and in-logic compilation using the CakeML compiler backend.*

In this extended version of our original conference paper, the *italicised* contributions in the list above are new.

1.3 Notation: $=$ and $=_c$, $\vdash$ and $\vdash_c$, etc.

This paper contains syntax at multiple, potentially confusing levels. The Candle logic is formalised inside the HOL4 logic. Symbols that exist in both logics are suffixed by a subscript $_c$ in its Candle version; as an example, $=$ denotes equality in the HOL4 logic, and $=_c$ denotes equality in the embedded Candle logic. Likewise, a theorem in HOL4 is prefixed by $\vdash$, while a Candle theorem is prefixed by $\vdash_c$.

1.4 Source Code and Proofs

Our Candle sources are on GitHub,[2] and the Candle project is hosted on the CakeML website.[3] Our HOL4 sources are also on GitHub.[4]

2 Approach: Candle

This section explains, at a high level, the approach we have taken to add and verify a new function for computation to Candle. We begin with some background on Candle itself; readers familiar with Candle can skip ahead to Sect. 2.2.

2.1 Background: Candle

Candle is a port of HOL Light. Whereas HOL Light is implemented using OCaml, Candle is formalised in HOL4. In HOL4, it has further been proved sound: Candle can only output theorems constructed by the primitive inference rules of its higher-order logic. As these inference rules are known to be sound, Candle can only output valid theorems. Like all LCF-style theorem provers, Candle's primitive inference rules are implemented in its kernel; unlike other provers, Candle's kernel is not *trusted* to be sound, but *verified*. Candle therefore provides a valuable testing environment for making changes to higher-order logic prover kernels: as long as the changes can be incorporated into its soundness proof, they do not undermine trust.

[2] https://github.com/CakeML/cakeml/tree/v2523/candle/prover/compute.

[3] https://cakeml.org/candle.

[4] https://github.com/HOL-Theorem-Prover/HOL/tree/718d989/src/num/theories/cv_compute.

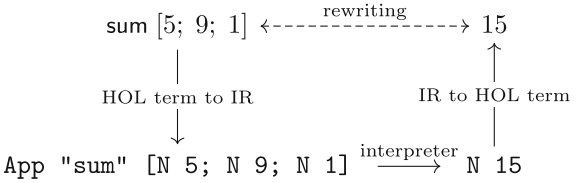

Fig. 1 Diagram illustrating the approach we take to embedding logical terms into compute expressions and evaluating them using an interpreter

Candle also has a verified implementation in CakeML. This implementation is automatically synthesised from the HOL4 functions of Candle's formalisation using a tool described in prior work [4]. This tool is *proof-producing*: when invoked, it outputs not only CakeML code, but also a HOL4 theorem that relates the behaviour of the output CakeML code with the input HOL4 functions. This can be considered a form of *verified code extraction* from a theorem prover.

2.2 Overview

First, we introduce a new computation-friendly internal representation (IR) for expressions that we want to do computation on. On entry to the new compute primitive, the given input term is translated into this new IR. This step corresponds to the downwards arrow in Fig. 1. We use an IR that is separate from the syntax of HOL (theorems, terms and types), since the datatypes used by HOL ITPs are badly suited for efficient computation.

We perform computation on the terms of our IR via interpretation. This step is the solid right arrow in Fig. 1. On termination, this interpretation arrives at a return value, which is translated to a HOL term r. This step is the up arrow in Fig. 1. The new compute primitive returns, to the user, a theorem stating that the input term is equal to the result r. The theorem states an equality between the points connected with a dashed arrow in Fig. 1.

The new compute primitive is a user-accessible function in the Candle kernel and must therefore be proved to be sound, i.e., every theorem it returns must follow by the primitive inference rules of higher-order logic (HOL).

We prove the soundness of our computation function by showing that there is some way of using the inference rules of HOL to mimic the operations of the interpreter. Our use of the inference rules amounts to showing that there is some proof by rewriting that establishes the desired equation. Since Candle performs no proof recording of any kind, it suffices, for the soundness proof, to prove (in HOL4) that there exists some derivation in the Candle logic.

The connection established by the existentially quantified proof is illustrated by the dashed arrow in Fig. 1. All reasoning about the interpreter (the lower horizontal arrow) must be w.r.t. the view of the interpreter provided by the translations to and from the IR (the vertical arrows). Nearly all of our theorems are stated in terms of the arrow upwards, i.e. from IR to HOL.

2.3 Staging

The development of our new compute primitive for Candle was staged into increasingly complex versions.

1. Version 1 (Sect. 3) was a proof-of-concept Candle function for computing the result of additions of concrete natural numbers. This function was implemented as a conversion[5]

[5] A *conversion* is a proof procedure that takes a term t as input and proves a theorem $\vdash t = t'$ for some interesting t'.

in the Candle kernel that given a term $m +_c n$ computes the result of the addition r, and returns a theorem $\vdash_c m +_c n =_c r$ to the user. Internally, the implementation makes use of the arbitrary precision integer arithmetic of the host language, i.e. CakeML. The purpose of Version 1 was to establish the concepts needed for this work rather than producing something that is actually useful from a user's point of view.
2. Version 2 (Sect. 4) improved on Version 1 by replacing the type of natural numbers by a datatype for binary trees with natural numbers at the leaves, and by supporting structured control-flow (if-then-else), projections (fst, snd) and the usual arithmetic operations. This version supports nesting of expressions.
3. Version 3 (Sect. 5) extended Version 2 with support for user-supplied code equations for user-defined constants. The code equations are allowed to be recursive and thus the interpreter had to support recursion. This extension also brought with it variables: from Version 3 and on, all interpreters are able to interpret input terms containing variables.
4. Version 4 (Sect. 6) replaced the naive interpreter with one that is designed to evaluate with less overhead. This version uses $O(1)$ operations to look up code equations and uses environments rather than substitutions for variable bindings.
5. The final Version 5 (Sect. 7) is, at the time of writing, left as future work for Candle. In Version 5, our intention is to speed up the interpreter by using partial evaluation at compile time so that less work is done at during actual interpreter execution.

At the time of writing, Version 4 (Sect. 6) is integrated into the existing Candle implementation and end-to-end soundness proof. This is the version we perform Candle benchmarks on (Sect. 8). Should refer to Sect. 6 has been integrated into the HOL4 implementation (Sect. 10). This is the version we perform HOL4 benchmarks on, in particular execution of the CakeML compiler backend (Sect. 12).

3 Addition of Natural Numbers (Version 1)

In this section, we describe how we implemented and verified a function for computing addition on natural numbers in the Candle kernel. This is the first step towards a proven-correct function for computation. The approach can be reused to produce computation functions for other kinds of binary operations (multiplication, subtraction, division, etc.) on natural numbers, and it can be used to build evaluators for arithmetic inside more general expressions (Sect. 4).

3.1 Input and Output

In Version 1, the user can input terms such as $3 +_c 5$ or $100 +_c 0$, i.e., terms consisting of one addition applied to two concrete numbers. The numbers are shown here as 3, 5, 100, 0, even though they are actually terms in a binary representation based on the constant 0_c, and the functions $\mathsf{Bit0}_c$ and $\mathsf{Bit1}_c$ in the Candle logic.

The output is a theorem equating the input with a concrete natural number. For the examples above, the function returns the following equations. The subscript $_c$ is used below to highlight that these are theorems in the Candle logic.

$$\vdash_c 3 +_c 5 =_c 8 \quad \text{or} \quad \vdash_c 100 +_c 0 =_c 100$$

The results 8 and 100 are computed using addition outside the logic. The challenge is to show that the same computation can be derived from the equations defining $+_c$ (in Candle) using the primitive inference rules of the Candle logic.

3.2 Key Soundness Lemma

In order to prove the soundness of Version 1 (required for its inclusion in the Candle kernel), we need to prove the following theorem, which states: if the arithmetic operations are defined as expected (num_thy_ok) in the current Candle theory Γ, then the addition ($+_c$) of the binary representations (mk_num) of two natural numbers m and n is equal ($=_c$) to the binary representation of $(m + n)$, where + is HOL4 addition.

$$\begin{array}{l} \vdash \mathsf{num\_thy\_ok}\ \Gamma\ \Rightarrow \\ \quad \Gamma \vdash_c \mathsf{mk\_num}\ m\ +_c\ \mathsf{mk\_num}\ n\ =_c\ \mathsf{mk\_num}\ (m\ +\ n) \end{array} \tag{2}$$

To understand the theorem statement above, let us look at the definitions of mk_num and num_thy_ok. The function mk_num converts a HOL4 natural number into the corresponding Candle natural number in binary representation:

$$\begin{array}{l} \mathsf{mk\_num}\ n \stackrel{\text{def}}{=} \\ \quad \mathsf{if}\ n\ =\ 0\ \mathsf{then}\ 0_c \\ \quad \mathsf{else\ if}\ \mathsf{even}\ n\ \mathsf{then}\ \mathsf{Bit0}_c\ (\mathsf{mk\_num}\ (n\ \mathsf{div}\ 2)) \\ \quad \mathsf{else}\ \mathsf{Bit1}_c\ (\mathsf{mk\_num}\ (n\ \mathsf{div}\ 2)) \end{array}$$

The definition of num_thy_ok asserts that various characterising equations hold for the Candle constants $+_c$, $\mathsf{Bit0}_c$ and $\mathsf{Bit1}_c$ (the complete definition is not shown below). Here m and n are natural number typed variables in Candle's logic:

$$\begin{array}{l} \mathsf{num\_thy\_ok}\ \Gamma \stackrel{\text{def}}{=} \\ \quad \Gamma \vdash_c 0_c\ +_c\ \mathsf{n}\ =_c\ \mathsf{n} \wedge \\ \quad \Gamma \vdash_c \mathsf{Suc}_c\ \mathsf{m}\ +_c\ \mathsf{n}\ =_c\ \mathsf{Suc}_c\ (\mathsf{m}\ +_c\ \mathsf{n}) \wedge \\ \quad \Gamma \vdash_c \mathsf{Bit0}_c\ \mathsf{n}\ =_c\ \mathsf{n}\ +_c\ \mathsf{n} \wedge \\ \quad \Gamma \vdash_c \mathsf{Bit1}_c\ \mathsf{n}\ =_c\ \mathsf{Suc}_c\ (\mathsf{n}\ +_c\ \mathsf{n}) \wedge \ldots \end{array}$$

We use num_thy_ok as an assumption in Theorem (2), since the computation function is part of the Candle kernel, which does not include these definitions when the prover starts from its initial state (and thus the user might define them differently).

A closer look at num_thy_ok reveals that $+_c$ is characterised by its simple Suc-based equations and $\mathsf{Bit1}_c$ is characterised in terms of Suc and $+_c$. As a result, a direct proof of Theorem (2) would be awkward at best.

To keep the proof of Theorem (2) as neat as possible, we defined the expansion of a HOL number into a tower of Suc_c applications to 0_c:

$$\mathsf{mk\_suc}\ n \stackrel{\text{def}}{=} \mathsf{if}\ n\ =\ 0\ \mathsf{then}\ 0_c\ \mathsf{else}\ \mathsf{Suc}_c\ (\mathsf{mk\_suc}\ (n\ -\ 1))$$

and split the proof of Theorem (2) into two lemmas. The first lemma is a mk_suc variant of Theorem (2):

$$\begin{array}{l} \vdash\ \mathsf{num\_thy\_ok}\ \Gamma\ \Rightarrow \\ \quad \Gamma \vdash_c \mathsf{mk\_suc}\ m\ +_c\ \mathsf{mk\_suc}\ n\ =_c\ \mathsf{mk\_suc}\ (m\ +\ n) \end{array} \tag{3}$$

and the second lemma $=_c$-equates mk_num with mk_suc:

$$\begin{array}{l} \vdash \text{num\_thy\_ok}\ \Gamma\ \Rightarrow \\ \quad \Gamma \vdash_c \text{mk\_num}\ n\ =_c\ \text{mk\_suc}\ n \end{array} \tag{4}$$

The proof of Theorem (3) was done by induction on m, and involved manually constructing the $\vdash_c$-derivation that connects the two sides of $=_c$ in Theorem (3). The proof of Theorem (4) is a complete induction on n and uses Theorem (3) when $+_c$ is encountered. Finally, the proof of Theorem (2) is a manually constructed $\vdash_c$-derivation that uses Theorems (4) and (3), and symmetry of $=_c$.

3.3 From Candle Terms to Natural Numbers

The development described above is in terms of functions (mk_num, mk_suc) that map HOL4 natural numbers into Candle terms, but the implementation also converts in the opposite direction: on initialisation, the computation function converts the given input term into its internal representation (see the leftmost arrow in Fig. 1).

We use the following function, dest_num, to extract a natural number from a Candle term. This function traverses terms, and recognises the function symbols used in Candle's binary representation of natural numbers:

$$\begin{array}{l} \text{dest\_num}\ tm\ \stackrel{\text{def}}{=} \\ \ \text{case}\ tm\ \text{of} \\ \ |\ 0_c\ \Rightarrow\ \text{Some}\ 0 \\ \ |\ \text{Bit0}_c\ r\ \Rightarrow\ \text{option\_map}\ (\lambda n.\ 2 \times n)\ (\text{dest\_num}\ r) \\ \ |\ \text{Bit1}_c\ r\ \Rightarrow\ \text{option\_map}\ (\lambda n.\ 2 \times n + 1)\ (\text{dest\_num}\ r) \\ \ |\ \_\ \Rightarrow\ \text{None} \end{array}$$

One should read the application $\text{Bit}b_c\ bs$ as a natural number in binary with least significant bit b and other bits bs.

The correctness of dest_num is captured by the following theorem, which states that $=_c$ is preserved when moving from Candle terms to natural numbers in HOL4, and back:

$$\begin{array}{l} \vdash \text{num\_thy\_ok}\ \Gamma\ \wedge\ \text{dest\_num}\ t\ =\ \text{Some}\ t'\ \Rightarrow \\ \quad \Gamma \vdash_c \text{mk\_num}\ t'\ =_c\ t \end{array} \tag{5}$$

Version 1 of the computation function also has a function for taking apart a Candle term with a top-level addition $+_c$:

$$\begin{array}{l} \text{dest\_add}\ tm\ \stackrel{\text{def}}{=} \\ \ \text{case}\ tm\ \text{of} \\ \ |\ (x +_c y)\ \Rightarrow\ \text{Some}\ (x, y) \\ \ |\ \_\ \Rightarrow\ \text{None} \end{array}$$

Equipped with the functions dest_num and dest_add, and Theorems (2) and (5), it is easy to prove the following soundness result. This theorem states: if a term t can be taken apart using dest_add and dest_num, then the term constructed by mk_num and the HOL4

addition, $+$, can be used as the right-hand side of an equation that is $\vdash_{\mathsf{c}}$-derivable.

$$\begin{array}{l} \vdash \mathsf{num\_thy\_ok}\ \Gamma\ \Rightarrow \\ \quad \mathsf{dest\_add}\ t\ =\ \mathsf{Some}\ (x,y)\ \wedge \\ \quad \mathsf{dest\_num}\ x\ =\ \mathsf{Some}\ m\ \wedge \\ \quad \mathsf{dest\_num}\ y\ =\ \mathsf{Some}\ n\ \Rightarrow \\ \quad\ \Gamma \vdash_{\mathsf{c}} t\ =_{\mathsf{c}}\ \mathsf{mk\_num}\ (m\ +\ n) \end{array} \tag{6}$$

This theorem can be used as the blueprint for an implementation that uses dest_add, dest_num and mk_num.

3.4 Checking num_thy_ok

Note that Theorem (6) assumes num_thy_ok, which requires certain equations to be true in the current theory Γ. To be sound, an implementation of our computation function must check that this assumption holds.

We deal with this issue in a pragmatic manner, by requiring that the user provides a list of theorems corresponding to the equations of num_thy_ok on each invocation of our computation function. This approach makes num_thy_ok easy to establish, but causes extra overhead on each call to the computation function.

3.5 Soundness of CakeML Implementation

Throughout this section, we have treated functions in the logic of HOL4 as if they were the implementation of the Candle kernel. We do this because the actual CakeML implementation of the Candle kernel is automatically synthesised from these functions in the HOL4 logic (Sect. 2.1).

Updating the entire Candle soundness proof for the addition of Version 1 of the compute function was straightforward, once Theorem (6) was proved and the code for checking num_thy_ok was verified.

4 Compute Expressions (Version 2)

This section describes Version 2, which generalises the very limited Version 1. While Version 1 only computed addition of natural numbers, Version 2 can compute the value of any term that fits in a subset of Candle terms that we call *compute expressions*. Compute expressions operate over a Lisp-inspired datatype which we call *compute values*; in Candle, this type is called cval.

Even though this second version might at first seem significantly more complicated than the first, it is merely a further development of Version 1. The approach is the same: the soundness theorems we prove are very similar looking. Technically, the most significant change is the introduction of a datatype, cexp, that is the internal representation of all valid input terms, i.e., compute expressions.

4.1 Compute Values

To the Candle user, the following cval datatype is important, since all terms supplied to the new compute function must be of this type. The cval datatype is a Lisp-inspired binary tree with natural numbers (num) at the leaves:

$$\mathsf{cval} \;=\; \mathsf{Pair_c}\ \mathsf{cval}\ \mathsf{cval} \mid \mathsf{Num_c}\ \mathsf{num}$$

4.2 Compute Expressions

The other important datatype is cexp, which is the internal representation that user input is translated into:

$$\begin{aligned}\mathsf{cexp} \;=\;& \mathsf{Pair}\ \mathsf{cexp}\ \mathsf{cexp}\\ \mid\;& \mathsf{Num}\ \mathsf{num}\\ \mid\;& \mathsf{If}\ \mathsf{cexp}\ \mathsf{cexp}\ \mathsf{cexp}\\ \mid\;& \mathsf{Uop}\ \mathsf{uop}\ \mathsf{cexp}\\ \mid\;& \mathsf{Binop}\ \mathsf{binop}\ \mathsf{cexp}\ \mathsf{cexp}\end{aligned}$$

$$\mathsf{uop} \;=\; \mathsf{Fst} \mid \mathsf{Snd} \mid \mathsf{IsPair}$$

$$\mathsf{binop} \;=\; \mathsf{Add} \mid \mathsf{Sub} \mid \mathsf{Mul} \mid \mathsf{Div} \mid \mathsf{Mod} \mid \mathsf{Less} \mid \mathsf{Eq}$$

The cexp datatype is extended with more constructors in Version 3, described in Sect. 5.

4.3 Input Terms

On start up, the compute function maps the given term into the cexp type. For example, given this term as input:

$$\mathsf{cif_c}\ (\mathsf{Num_c}\ 1)\ (\mathsf{Num_c}\ 2)\ (\mathsf{fst_c}\ (\mathsf{Pair_c}\ (\mathsf{Num_c}\ 3)\ (\mathsf{Num_c}\ 4)))$$

the function will create this cexp expression:

$$\mathsf{If}\ (\mathsf{Num}\ 1)\ (\mathsf{Num}\ 2)\ (\mathsf{Uop}\ \mathsf{Fst}\ (\mathsf{Pair}\ (\mathsf{Num}\ 3)\ (\mathsf{Num}\ 4)))$$

This mapping assumes that certain functions in the Candle logic (e.g. $\mathsf{fst_c}$) correspond to certain constructs in the cexp datatype (e.g. Uop Fst). Note that there is nothing strange about this: in Version 1, we assumed that $+_c$ corresponds to addition. We formalise the assumptions about $\mathsf{fst_c}$, etc., next.

4.4 Context Assumption: cexp_ thy_ ok

Just as in Version 1, Version 2 also has an assumption on the current theory context. In Version 1, the assumption num_ thy_ ok ensured that the Candle definition of $+_c$ satisfied the relevant characterising equations. For Version 2, this assumption was extended to cover characterising equations for all names that the conversion from user input to cexp recognises: $\mathsf{cif_c}$, $\mathsf{fst_c}$, etc. These characterising equations fix a semantics for the Candle functions that

correspond to constructs of the cexp type. For simplicity, all of the Candle functions take inputs of type cval and produce outputs of type cval.

Our implementation makes no attempt at ensuring that functions are applied to sensible inputs. Consequently, it is perfectly possible to write strange terms in this syntax, such as $\mathsf{fst_c}\ (\mathsf{Num_c}\ 3)$, or $\mathsf{add_c}\ (\mathsf{Num_c}\ 3)\ (\mathsf{Pair_c}\ p\ q)$. We resolve such cases in a systematic way:

- Operations that expect numbers as input treat $\mathsf{Pair_c}$ values as $\mathsf{Num_c}\ 0$.
- Operations that expect a pair as input return $\mathsf{Num_c}\ 0$ when applied to $\mathsf{Num_c}$ values.

This treatment of the primitives can be seen in the assumption, called cexp_thy_ok, that we make about the context for Version 2. Below, x and y are variables in the Candle logic with type cval. The lines specifying $\mathsf{add_c}$ are:

$$
\begin{aligned}
&\mathsf{cexp\_thy\_ok}\ \Gamma \stackrel{\text{def}}{=} \\
&\ldots \wedge \\
&\Gamma \vdash_c \mathsf{add_c}\ (\mathsf{Num_c}\ m)\ (\mathsf{Num_c}\ n) =_c \mathsf{Num_c}\ (m +_c n) \wedge \\
&\Gamma \vdash_c \mathsf{add_c}\ (\mathsf{Pair_c}\ x\ y)\ (\mathsf{Num_c}\ n) =_c \mathsf{Num_c}\ n \wedge \\
&\Gamma \vdash_c \mathsf{add_c}\ (\mathsf{Num_c}\ m)\ (\mathsf{Pair_c}\ x\ y) =_c \mathsf{Num_c}\ m \wedge \ldots
\end{aligned}
$$

The lines specifying $\mathsf{fst_c}$ are:

$$
\begin{aligned}
&\Gamma \vdash_c \mathsf{fst_c}\ (\mathsf{Pair_c}\ x\ y) =_c x \wedge \\
&\Gamma \vdash_c \mathsf{fst_c}\ (\mathsf{Num_c}\ n) =_c \mathsf{Num_c}\ 0_c \wedge \ldots
\end{aligned}
$$

The following characteristic equations for $\mathsf{cif_c}$ illustrate that we treat $\mathsf{Num_c}\ 0_c$ as false and all other values as true:

$$
\begin{aligned}
&\Gamma \vdash_c \mathsf{cif_c}\ (\mathsf{Num_c}\ 0_c)\ x\ y =_c y \wedge \\
&\Gamma \vdash_c \mathsf{cif_c}\ (\mathsf{Num_c}\ (\mathsf{Suc}\ n))\ x\ y =_c x \wedge \\
&\Gamma \vdash_c \mathsf{cif_c}\ (\mathsf{Pair_c}\ x'\ y')\ x\ y =_c x \wedge \ldots
\end{aligned}
$$

Comparison primitives return $\mathsf{Num_c}\ 1$ for true.

4.5 Soundness

The following theorem summarises the operations and soundness of Version 2. If a term t can be successfully converted (using dest_term) into a compute expression $cexp$, then t is equal to a Candle term created (using mk_term) from the result of evaluating $cexp$ using a straightforward evaluation function (cexp_eval):

$$
\begin{aligned}
&\vdash \mathsf{cexp\_thy\_ok}\ \Gamma \Rightarrow \\
&\quad \mathsf{dest\_term}\ t = \mathsf{Some}\ cexp \Rightarrow \\
&\quad \Gamma \vdash_c t =_c \mathsf{mk\_term}\ (\mathsf{cexp\_eval}\ cexp)
\end{aligned}
\tag{7}
$$

Note the similarity between Theorems (6) and (7). Where Theorem (6) uses $+$, Theorem (7) calls cexp_eval. The evaluation function cexp_eval is defined to traverse the cexp bottom-up in the most obvious manner, respecting the evaluation rules set by the characterising equations of cexp_thy_ok.

4.6 CakeML Code and Integration

The functions dest_term, cexp_eval and mk_term are the main workhorses of the implementation of Version 2. Corresponding CakeML implementations are synthesised from these functions (Sect. 2.1). The definition of the evaluator function cexp_eval uses arithmetic operations ($+$, $-$, $\times$, div, mod, $<$, $=$) over the natural numbers. Such arithmetic operations translate into arbitrary precision arithmetic operations in CakeML.

Updating the Candle proofs for Version 2 was a straightforward exercise, given the prior integration of Version 1.

5 Recursion and User-Supplied Code Equations (VERSION 3)

Version 3 of our compute function for Candle adds support for (mutually) recursive user-defined functions. The user supplies function definitions in the form of *code equations*.

5.1 Code Equations

In our setting, a code equation for a user-defined constant c is a Candle theorem of the form:

$$\vdash_{\mathsf{c}} \ \mathsf{c}\ v_1\ \ldots\ v_n\ =_{\mathsf{c}} e$$

where each variable v_i has type cval and the expression e has type cval. Furthermore, the free variables of e must be a subset of $\{v_1, \ldots, v_n\}$. Note that any user-defined constants, including c, are allowed to appear in e in fully applied form. Every user-defined constant appearing in some right-hand side e must have a code equation describing that constant.

5.2 Updated Compute Expressions

We updated the cexp datatype to allow variables (Var), applications of user-supplied constants (App), and, at the same time, we added let-expressions (Let):

```
cexp = Pair cexp cexp
     | Num num
     | Var string
     | App string (cexp list)
     | Let string cexp cexp
     | If cexp cexp cexp
     | Uop uop cexp
     | Binop binop cexp cexp
```

Variables are present to capture the values bound by the left-hand sides of code equations and by let-expressions.

The interpreter for Version 3 of our compute function uses a substitution-based semantics, and keeps track of code equations as a simple list. This style of semantics maps well to the Candle logic's substitution primitive, thus simplifying verification, but at a price:

- At each let-expression or function application, the entire body of the let-expression or the code equation corresponding to the function is traversed an additional time, to substitute out variables.
- At each function application, the code equation corresponding to the function name is found using linear search, making the interpreter's performance degrade as more code equations are added.

We address these shortcomings in Version 4 of our compute function, in Sect. 6.

5.3 Soundness

The following theorem is the essential part of the soundness argument for Version 3. The user supplies the Version 3 compute function with: a list of theorems that allows it to establish cexp_thy_ok, a list *eqs* of code equations, and a term t to evaluate. Every theorem in *eqs* must be a Candle theorem ($\vdash_c$). Definitions *defs* are extracted from the given code equations *eqs*. A compute expression *cexp* is extracted from the given input term w.r.t. the available definitions *defs*. An interpreter, interpret, is run on the *cexp*, and its execution is parameterised by *defs* and a clock which is initialised to a large number init_ck. If the interpreter returns a result *res*, i.e. Some *res*, then an equation between the input term t and mk_term *res* can be returned to the user.

$$\begin{array}{l} \vdash \mathsf{cexp\_thy\_ok}\ \Gamma\ \Rightarrow \\ \quad (\forall eq.\ \ \mathsf{mem}\ eq\ eqs\ \Rightarrow\ \Gamma \vdash_c eq) \wedge \\ \quad \mathsf{dest\_eqs}\ eqs = \mathsf{Some}\ defs \wedge \\ \quad \mathsf{dest\_tm}\ defs\ t = \mathsf{Some}\ cexp \wedge \\ \quad \mathsf{interpret}\ \mathsf{init\_ck}\ defs\ cexp = \mathsf{Some}\ res\ \Rightarrow \\ \quad \Gamma \vdash_c t\ =_c\ \mathsf{mk\_term}\ res \end{array} \tag{8}$$

There are a few subtleties hidden in this theorem that we will comment on next.

First, the statement of Theorem 8 includes an assumption that the user-provided code equations *eqs* are theorems in the context Γ. This holds trivially for Candle's implementation: the user can only pass in the code equations as theorems, and can only construct these theorems if they are valid in the current context. Therefore, Candle's soundness result allows us to discharge this assumption where Theorem 8 is used.

Second, the functions dest_eqs and dest_term perform sanity checks on their inputs. For example, dest_eqs checks that all right-hand sides in the equations *eqs* mention only constants for which there are code equations in *eqs*.

Third, the interpret function, which is used for the actual computation, takes a clock (sometimes called fuel parameter) in order to guarantee termination. The clock is decremented by interpret on each function application (i.e. App), and, due to the substitution semantics, also on each Let. If the clock is exhausted, interpret returns None. This clock is not strictly necessary to reason about soundness: we could have implemented interpret directly in CakeML and verified its soundness by appealing to either CakeML's semantics or its program logic (which supports reasoning about diverging programs [19]). In particular, interpret can diverge without introducing unsoundness, because then no theorem is ever constructed. For simplicity, we chose to keep in line with the rest of Candle, by specifying and verifying interpret in HOL4 before synthesising a verified CakeML implementation (Sect. 5.4). This approach requires a clock argument to guarantee termination.

5.4 CakeML Code

As with previous versions, the CakeML implementation of the computation function is synthesised from the HOL4 functions (Sect. 2.1). For efficiency purposes, the generated CakeML code for interpret avoids returning an option and instead signals running out of clock using an ML exception. We note that it is very unlikely that a user has the patience to wait for a timeout since the value of init_ck is very large (maximum smallnum).

5.5 Integration

Updating the Candle proofs for Version 3 required more work than Versions 1 and 2, since we had to verify the correctness of the sanity checks performed on the user-provided list of code equations.

6 Efficient Interpreter (VERSION 4)

For Version 4, we replaced the interpreter function, interpret, with compilation to a different datatype for which we have a faster interpreter.

The new datatype for representing programs is called ce, shown below. It uses de Bruijn indexing for local variables, and represents function names as indices into a vector of function bodies, which means lookups happen in constant time during interpretation. Rather than representing primitive functions by names, the ce datatype represents primitive functions as (shallowly embedded) function values that can immediately be applied to the result of evaluating the argument expressions.

```
ce = Const num
   | Var num
   | Let ce ce
   | If ce ce ce
   | Monop (cval → cval) ce
   | Binop (cval → cval → cval) ce ce
   | App num (ce list)
```

The new faster interpreter exec, shown in Fig. 2, for the ce datatype addresses the two main shortcomings of Version 3. First, it drops the substitution semantics in favour of de Bruijn variables and an explicit environment, so that variable substitution can be deferred until (and if) the value bound to a variable is needed. Second, all function names are replaced by an index into a vector which stores all user-provided code equations.

When updating Version 3 to Version 4, we simply replaced the following line in the implementation:

interpret init_ck *defs cexp*

with the line below, which calls the compilers compile_all and compile (these translate cexp into ce, turning variables and function names into indices) and then runs exec, which interprets the program represented in terms of ce:

exec init_ck [] (compile_all *defs*) (compile *defs* [] *cexp*)

Fig. 2 Definition of the fast interpreter as functions in HOL

```
exec funs env ck (Const n) =def
  return (Num n)
exec funs env ck (Var n) =def
  return (env_lookup n env)
exec funs env ck (Monop m x) =def
  ...
exec funs env ck (Binop b x y) =def
  do
    v ← exec funs env ck x;
    w ← exec funs env ck y;
    return (b v w)
  od
exec funs env ck (App f xs) =def
  do
    check_clock ck;
    vs ← execl funs env ck xs [];
    c ← get_code f funs;
    exec funs vs (ck − 1) c
  od
exec funs env ck (Let x y) =def
  do
    check_clock ck;
    v ← exec funs env ck x;
    exec funs (v::env) (ck − 1) y
  od
exec funs env ck (If x y z) =def
  do
    v ← exec funs env ck x;
    exec funs env ck
      (if v = Num 0 then z else y)
  od
execl funs env ck [] acc =def
  return acc
execl funs env ck (x::xs) acc =def
  do
    v ← exec funs env ck x;
    execl funs env ck xs (v::acc)
  od
```

Updating the proofs for Version 4 was a routine exercise in proving the correctness of the compilers compile_all and compile. In this proof, compiler correctness is an equality: the new line computes exactly the same result as the line that it replaced (under some assumptions that are easily established in the surrounding proof). The adjustments required in the existing proofs were minimal.

```
fun exec funs env ck e =
  case e of
    Const n => Num n
  | Var n => List.nth n env
  | Monop m x =>
      (* omitted *)
  | Binop b x y =>
      let
        val v = exec funs env ck x
        val w = exec funs env ck y
      in
        b v w
      end
  | App f xs =>
      let
        val _ = check_clock ck
        val vs = execl funs env ck xs []
        val c = Vector.nth f funs
      in
        exec funs vs (ck - 1) c
      end
  | Let x y =>
      let
        val _ = check_clock ck
        val v = exec funs env ck x
      in
        exec funs (v::env) (ck - 1) y
      end
  | If x y z =>
      let
        val v = exec funs env ck x
      in
        exec funs env ck
          (if v = Num 0 then z else y)
      end
and execl funs env ck l acc =
  case l of
    [] => acc
  | (x::xs) =>
      let
        val v = exec funs ck x
      in
        execl funs env ck xs (v::acc)
      end
```

Fig. 3 CakeML code synthesised from definition of exec, as described in Sect. 2.1. It is verified to implement the HOL functions in Fig. 2

7 Partial Evaluation (Version 5)

At the time of writing, Version 5 is not yet implemented in Candle. However, the plan is replace the `exec` function of Fig. 3 with code along the lines of the code shown in Fig. 4. While the old version walks the AST as it interprets it, the new version performs all case splitting on the AST before the actual exeuction starts. When the new version exeuctes, it always has at hand a closure value that will perform the relevant next step of the computation.

In some preliminary stress tests, Version 5 is almost twice as fast as Version 4. We intend to verify Version 5 and upgrade the Candle implementation to use it.

8 Evaluation: Candle

In this section, we report on experiments comparing our new verified compute function (Version 4) to the existing in-logic interpreters of HOL4, HOL Light, and Isabelle/HOL.[6] These are implemented as derived rules, as alluded to in Sect. 1. We tested the performance of each on the following four example programs written as functions in the logic of HOL.

- the factorial function,
- enumeration of primes,
- generating and reversing a list of numbers,
- simulation of a 100-by-100 grid of cells in Conway's Game of Life.

The tests were run on an Intel i7-7700K 4.2GHz with 64 GiB RAM running Ubuntu 20.04. The code used for these experiments is available on the CakeML website.[7]

The results, in Fig. 1, show that Candle's new compute function runs orders of magnitude faster than the existing in-logic interpreters of HOL4, HOL Light, and Isabelle/HOL, on all four examples. In fact, it was difficult to choose input sizes large enough for us to gather meaningful measurements from our computation function, while keeping the runtimes of its derived counterparts within minutes. For this reason, we added one large data point to the end of each experiment. In Fig. 1, a dash, —, indicates that we did not test this.

The first two examples, factorial and primes, demonstrate the speed of computing basic arithmetic, while the latter two examples, list reversal and Conway's Game of Life, highlight that Candle's compute primitive is also well suited for structural computations, such as tree traversals, that do not involve much arithmetic.

8.1 Factorial

The first example is a standard, non-tail-recursive factorial function, tested on inputs of various sizes. The results of the tests are shown in the upper left corner of Table 1. This is the only test where HOL Light outperforms HOL4. We suspect HOL Light benefits from the effort that has gone into making its basic arithmetic evaluate fast.

[6] Because the most widely used tools for fast computation in Isabelle/HOL bypass the kernel (c.f. subsection 13.3), we compare instead with `Code_Simp.dynamic_conv`, a tool written by Florian Haftmann which performs in-logic computation by using the simplifier with the code equations as rewrite rules.

[7] https://cakeml.org/candle_benchmarks.html.

Table 1 Running times for Candle's compute primitive, HOL4's EVAL, HOL Light's EVAL, and Isabelle/HOL's in-logic `Code_Simp.dynamic_conv`

fact n for different values of n					primes_upto n for different values of n				
n	Candle	HOL4	H.Light	Isabelle	n	Candle	HOL4	H.Light	Isabelle
256	<1 ms	2.3 s	0.6 s	14 s	256	<1 ms	0.5 s	1.3 s	2.6 s
512	<1 ms	4.1 s	3.5 s	202 s	512	<1 ms	1.6 s	5.2 s	9.8 s
1024	<1 ms	127 s	17.6 s	2451 s	1024	2 ms	6.3 s	20.7 s	35.6 s
2048	11 ms	684 s	86.1 s	–	2048	9 ms	24.2 s	83.4 s	132 s
32,768	0.9 s	–	–	–	32,768	1.7 s	–	–	–
rev_enum n for different values of n					n steps of Conway's Game of Life				
n	Candle	HOL4	H.Light	Isabelle	n	Candle	HOL4	H.Light	Isabelle
256	0.02	1.1	66.2	10.2	1	0.03	0.6	14.9	1.5
512	0.03	2.3	251	37.1	10	0.08	5.3	147	15.0
1024	0.07	4.7	1005	172	100	0.8	54	1474s	148
2048	0.1	9.5	4203	791	1000	8.0	568	14,623	1466s
32,768	2.5	–	–	–	10,000	79	–	–	–

Below dash, – indicates *not measured*

```
fun exec funs env ce =
  let
    fun impossible () = raise Bind
    val default = (fn env => impossible ())
    val code = Array.tabulate (Vector.length funs, fn k => default)
    fun run ce =
      case ce of
        Const n => (fn env => Num n)
      | Var n => List.nth n
      | Monop (m, x) => (* omitted *)
      | Binop (b, x, y) =>
          let
            val run_x = run x
            val run_y = run y
          in
            fn env => b (run_x env) (run_y env)
          end
      | Let (x, y) =>
          let
            val run_x = run x
            val run_y = run y
          in
            fn env => run_y (run_x env :: env)
          end
      | If (x, y, z) =>
          let
            val run_x = run x
            val run_y = run y
            val run_z = run z
          in
            fn env => if run_x env = cv_zero then run_z env else run_y env
          end
      | App (f, xs) =>
          let
            val run_xs = map run xs
          in
            fn env => (Array.sub (code, f)) (map (fn r => r env) run_xs)
          end
    (* populate code array with run functions *)
    val _ = Vector.appi (fn (i,x) => Array.update(code, i, run x)) funs
    val run_ce = run ce
    (* perform the actual execution *)
    val result = run_ce env
  in
    result
  end;
```

Fig. 4 Interpreter `exec` sped up using partial evaluation before exeuction

8.2 Prime Eumeration

The second example, primes_upto, enumerates all primes up to n and returns them as a list. We chose to implement the checks for primality using trial division, since it is challenging to compute division and remainder efficiently inside the logic. The results of the tests are shown in the upper right corner of Table 1.

8.3 List Reversal

The third example performs repeated list reversals. The function rev_enum creates a list of the natural numbers $[1, 2, \ldots, n]$ and then calls a tail-recursive list reverse function rev on this list 1000 times. The results of the tests are shown in the lower left corner of Table 1. On this and the next benchmark HOL Light performs much worse than HOL4 and Isabelle/HOL.

8.4 Conway's Game of Life

The fourth example simulates a 100-by-100 grid of cells in Conway's Game of Life. The surface of this 100-by-100 square is set up with five Gosper glider generators that will interact. The set up is self contained, i.e., it never touches the boundaries of the 100-by-100 grid. The simulation runs for n steps of Conway's Game of Life. The results of the tests are shown in the lower right corner of Table 1.

9 Approach: HOL4

This section describes, at a high level, how we have added our fast computation feature to the HOL4 theorem prover and built automation to make it as usable as possible there.

In particular, we implement Version 4 (Sect. 6) of the fast computation feature in HOL4's kernel. Evaluation proofs in HOL4 can then benefit from fast computation without compromising trust due to our verification in Candle. In HOL4, the function is implemented in Standard ML, in contrast to Candle's CakeML implementation (which is itself automatically synthesised from HOL4 definitions—Sect. 2.1). It also requires minor modifications due to incompatibilities between HOL4 and HOL Light, on which Candle is based (Sect. 10).

However, the new computation feature is not very user-friendly when used directly: it only accepts terms and code equations which are stated in terms of the cval type. We want to perform fast computation using ordinary user-defined functions which do not conform to this subset of the logic. Therefore, we build an automatic tool which translates common (mostly first-order) HOL functions into the cval datatype (Sect. 11). The tool provides a user-friendly flow for invoking the new compute function: it hides almost all details of cval from the user. Critically, this flow is *proof-producing*: it returns a top-level theorem which equates the input term with the result of its fast computation, with no mention of cval. We evaluate our flow by applying it to previously written, ordinary HOL4 functions: the implementations of CakeML's type inferencer and compiler backend (Sect. 12). We then demonstrate significant speedups when executing both the inferencer and compiler backend within HOL4's logic. In future work, we hope to port the tool back to Candle too.

9.1 Notation

All logical syntax from here on is within the HOL4 logic. In HOL4, the cval datatype is renamed to cv. Its constructors are Num and Pair (c.f. $\mathsf{Num_c}$ and $\mathsf{Pair_c}$), and its "primitive" functions are prefixed by "cv_" (e.g. cv_fst and cv_if instead of $\mathsf{fst_c}$ and $\mathsf{cif_c}$).

10 Fast Computation for HOL4

The implementation of Version 4 (Sect. 6) in HOL4 had to contend with two key differences between HOL4 and Candle. Both are inherited from HOL Light, on which Candle is based. The first concerns HOL4's theory structure: each HOL4 constant belongs to a theory, so HOL4 namespaces are structured; however, HOL Light and Candle have flat namespaces.

The second concerns the representation of natural numbers: recall that Candle (and HOL Light) uses the constants $\mathsf{0_c}$, $\mathsf{Bit0_c}$ and $\mathsf{Bit1_c}$. HOL4 instead uses 0, Bit1, and Bit2. The difference here is illustrated by the following theorems:

$$\vdash_c \mathsf{Bit0_c}\ n =_c 2 \times_c n \qquad \vdash\ \mathsf{Bit1}\ n = 2 \times n + 1$$
$$\vdash_c \mathsf{Bit1_c}\ n =_c 2 \times_c n +_c 1 \qquad \vdash\ \mathsf{Bit2}\ n = 2 \times n + 2$$

This discrepancy slightly alters the translation between logical and native numbers performed by our new compute function.

To bring behaviour in line with Candle (and HOL Light), we also had to extend HOL4's natural number division and modulo operators to specify dividing by zero and using a modulus of zero:

$$n\ \mathsf{div}\ 0 \stackrel{\mathsf{def}}{=} 0 \qquad n\ \mathsf{mod}\ 0 \stackrel{\mathsf{def}}{=} n$$

This is also in line with several other systems (e.g. Isabelle/HOL, Lean, and Coq). Previously, there were no equations defining the behaviour of these two constants when applied to zero.

We also removed the clock argument to the interpreter in HOL4: unlike in Candle's HOL4 formalisation, we do not need to consider termination.

11 Automated Translation into Code Equations

This section describes automation that allows users to apply fast computation without first writing definitions in the cv subset.

This automation centres around canonical in-logic translations for each supported HOL4 type: a translation *from* HOL4 terms of the type to untyped cv terms; and a translation *to* HOL4 terms of the type from cv terms. To perform fast computation of an ordinary HOL4 term, the automation does the following:

1. Automatically translates *from* the ordinary HOL4 term to its cv version.
2. Invokes the new kernel function to perform fast computations over the cv version.
3. Translates the resulting cv term back *to* an ordinary HOL4 term.

In this way, almost all details of cv are hidden from the user.

Critically, the automation is written outside of HOL4's kernel and demands no additional trust. In particular, the automation is proof-producing: at a high level, each of the three steps above produces a theorem as follows, where term is the input term.

$$\vdash\ \mathsf{from}\ \mathsf{term} = \mathsf{term}_{\mathsf{cv}} \qquad \text{(step 1: } \textit{from} \text{ translation)}$$

$$\vdash \ \mathsf{term}_{\mathsf{cv}} = \mathsf{value}_{\mathsf{cv}} \qquad \text{(step 2: new kernel function)}$$
$$\vdash \ \mathit{to}\ \mathsf{value}_{\mathsf{cv}} = \mathsf{term}' \qquad \text{(step 3: } \mathit{to} \text{ translation)}$$

We can compose these theorems as follows. Here we rely on the key property of *from* and *to* translations, namely that *to* is a left-inverse of *from*: $\vdash \mathit{to} \circ \mathit{from} = \mathsf{id}$.

$$\begin{aligned} & \mathsf{term} \\ = \ & \mathit{to}\ (\mathit{from}\ \mathsf{term}) && \text{(key property of } \mathit{from}/\mathit{to}) \\ = \ & \mathit{to}\ (\mathsf{term}_{\mathsf{cv}}) && \text{(step 1)} \\ = \ & \mathit{to}\ (\mathsf{value}_{\mathsf{cv}}) && \text{(step 2)} \\ = \ & \mathsf{term}' && \text{(step 3)} \end{aligned}$$

The top-level theorem presented to the user is therefore $\vdash \mathsf{term} = \mathsf{term}'$. Overall, our automation has invoked the new kernel function without exposing the cv datatype to the user.

Therefore, our automation must generate the *from* and *to* translations automatically for each supported type. It must further translate ordinary HOL4 functions to corresponding cv code equations using *from* translations, so that it can handle user-defined functions. The remainder of this section describes how we have designed our automation in more detail.

11.1 To and From cv

The key property of from and *to* functions mentioned above is that *to* is a left-inverse of from. We define this property as from_to:

$$\mathsf{from\_to}\ \mathit{from}\ \mathit{to} \stackrel{\mathsf{def}}{=} \forall x.\ \mathit{to}\ (\mathit{from}\ x) = x$$

Note that for simplicity, from and *to* are translation functions, not relations. This choice has a convenient side effect: every representation in the cv datatype is unique, which plays well with HOL4 equality:

$$\vdash \mathsf{from\_to}\ \mathit{from}\ \mathit{to} \Rightarrow \forall x\ y.\ \mathit{from}\ x = \mathit{from}\ y \iff x = y$$

Below, we show from and *to* functions for the following types: bool, num, and list.

$$\begin{aligned}
&\mathsf{from_bool}\ b \stackrel{\text{def}}{=} \mathsf{if}\ b\ \mathsf{then}\ \mathsf{Num}\ 1\ \mathsf{else}\ \mathsf{Num}\ 0\\
&\mathsf{to_bool}\ (\mathsf{Num}\ n) \stackrel{\text{def}}{=} \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{F}\ \mathsf{else}\ \mathsf{T}\\
&\\
&\mathsf{from_num}\ n \stackrel{\text{def}}{=} \mathsf{Num}\ n\\
&\mathsf{to_num}\ (\mathsf{Num}\ n) \stackrel{\text{def}}{=} n\\
&\\
&\mathsf{from_list}\ \mathit{from_a}\ [] \stackrel{\text{def}}{=} \mathsf{Num}\ 0\\
&\mathsf{from_list}\ \mathit{from_a}\ (x :: xs) \stackrel{\text{def}}{=}\\
&\quad \mathsf{Pair}\ (\mathit{from_a}\ x)\ (\mathsf{from_list}\ \mathit{from_a}\ xs)\\
&\\
&\mathsf{to_list}\ \mathit{to_a}\ (\mathsf{Num}\ n) \stackrel{\text{def}}{=} []\\
&\mathsf{to_list}\ \mathit{to_a}\ (\mathsf{Pair}\ x\ y) \stackrel{\text{def}}{=} \mathit{to_a}\ x :: \mathsf{to_list}\ \mathit{to_a}\ y
\end{aligned}$$

These satisfy the following from_ to theorems. Note how the type variable α in the list type α list is represented by the *from_a* (resp. *to_a*) function argument to from_ list (resp. to_list).

$$\vdash \mathsf{from\_to}\ \mathsf{from\_bool}\ \mathsf{to\_bool}$$

$$\vdash \mathsf{from\_to}\ \mathsf{from\_num}\ \mathsf{to\_num}$$

$$\begin{aligned}
&\vdash \mathsf{from_to}\ \mathit{from_a}\ \mathit{to_a} \Rightarrow\\
&\quad \mathsf{from_to}\ (\mathsf{from_list}\ \mathit{from_a})\ (\mathsf{to_list}\ \mathit{to_a})
\end{aligned}$$

We derive from_ to theorems for a variety of "primitive" HOL4 types: booleans, natural numbers, characters, integers, rationals, machine words, options, pairs, sums, and lists. We straightforwardly implement a library that automatically defines from and *to* functions for user-defined datatypes and derives corresponding from_ to theorems.

11.2 Using *from* Theorems

Using these from_ to theorems, we can demonstrate our high-level approach (see the start of this section) on a concrete example. Suppose the user wants to evaluate the term $1 < 2 + 3$. As always, the input term must be closed. The automation first derives a from theorem: its left-hand side is the input term wrapped in the appropriate from function (in this case, from_ bool); and its right-hand side consists only of cv functions.

$$\vdash \mathsf{from\_bool}\ (1 < 2 + 3) = \mathsf{cv}_{\mathsf{lt}}\ (\mathsf{Num}\ 1)\ (\mathsf{cv\_add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3)) \tag{9}$$

The tool then uses the new compute function to prove that $1 < 2 + 3$ equals T, relying on the from_ to theorem for the bool type and Theorem (9) above:

$$\begin{aligned}
&1 < 2 + 3\\
&= \mathsf{to_bool}\ (\mathsf{from_bool}\ (1 < 2 + 3)) && (\text{by from_ to theorem for bool})\\
&= \mathsf{to_bool}\ (\mathsf{cv_lt}\ (\mathsf{Num}\ 1)\ (\mathsf{cv_add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))) && (\text{by } 9)\\
&= \mathsf{to_bool}\ (\mathsf{Num}\ 1) && (\text{new compute function})\\
&= \mathsf{T} && (\text{definition of to_bool})
\end{aligned}$$

This concrete example directly mirrors the high-level approach described at the start of this section. Though it is very simple, this example showcases the overall approach which remains unchanged regardless of the complexity of the input term.

11.3 Deriving *from* Theorems

The example in Sect. 11.2 shows that our automation must be able to derive from theorems of the form:

$$\mathsf{from\_function}\ closed\_user\_input\ =\ cv\_expression$$

We define a judgement for such from equations, cv_rep, to better structure the terms our automation will handle:

$$\mathsf{cv\_rep}\ pre\ cv\_e\ from\ e\ \stackrel{\mathsf{def}}{=}\ pre \Rightarrow\ from\ e\ =\ cv\_e \tag{10}$$

This can be read as follows: "given precondition pre, the function from translates term e to the cv term cv_e".

Our automation derives cv_rep judgements bottom-up, using various helper lemmas. For example, to derive cv_rep for a natural number literal, it instantiates the following judgement:

$$\vdash\ \mathsf{cv\_rep\ T\ (Num}\ n)\ \mathsf{from\_num}\ n \tag{11}$$

To derive cv_rep for natural number addition (+), it uses the following result to establish a connection with cv_add:

$$\begin{aligned}\vdash\ &\mathsf{cv\_rep}\ p_1\ cv_1\ \ \mathsf{from\_num}\ n_1 \wedge\ \mathsf{cv\_rep}\ p_2\ cv_2\ \ \mathsf{from\_num}\ n_2 \Rightarrow \\ &\mathsf{cv\_rep}\ (p_1 \wedge p_2)\ (\mathsf{cv\_add}\ cv_1\ cv_2)\ \ \mathsf{from\_num}\ (n_1 + n_2)\end{aligned} \tag{12}$$

By way of example, consider the term $2 + 3$. Lemmas (11) and (12) are used to derive a cv_rep judgement as follows:

1. $\vdash$ cv_rep T (Num 2) from_num 2 (by 11)
2. $\vdash$ cv_rep T (Num 3) from_num 3 (by 11)
3. $\vdash$ cv_rep (T $\wedge$ T) (cv_add (Num 2) (Num 3)) from_num $(2 + 3)$ (by steps 1-2 and 12)

In this example, the preconditions are trivial. We will see the need for non-trivial preconditions when translating recursive and partially specified functions (Sects. 11.5 and 11.7).

We have derived lemmas akin to (11) and (12) for the various "primitive" literals and operations of the types mentioned in Sect. 11.1.

11.4 Translating Functions

Any interesting user input will contain functions—our automation must be able to translate them into code equations. We first describe the translation of non-recursive functions, taking

add1 below as an example.

$$\mathsf{add1}\ n \stackrel{\text{def}}{=} n + 1$$

We first derive a cv_rep judgement for the right-hand side of the function definition: $n + 1$. Unlike the example in Sect. 11.3, this term has a free variable, that is, the argument n to the function. To handle such free variables, our automation produces a trivial cv_rep judgement on-the-fly:

$$\vdash \mathsf{cv\_rep}\ \mathsf{T}\ (\mathsf{from\_num}\ n)\ \ \mathsf{from\_num}\ n$$

To see why this is trivial, simply unfold the definition of cv_rep (10).

The resulting cv_rep judgement for the right-hand side is therefore as follows, where we simplify the precondition $\mathsf{T} \wedge \mathsf{T}$ to T for brevity:

$$\vdash \mathsf{cv\_rep}\ \mathsf{T}\ (\mathsf{cv\_add}\ (\mathsf{from\_num}\ n)\ (\mathsf{Num}\ 1))\ \ \mathsf{from\_num}\ (n + 1) \tag{13}$$

Now, the second argument of the cv_rep judgement above (13) is essentially the right-hand side of the cv version of add1 we wish to define. We need only generalise from_num n to cv_n to define cv_add1:

$$\mathsf{cv\_add1}\ cv\_n \stackrel{\text{def}}{=} \mathsf{cv\_add}\ cv\_n\ (\mathsf{Num}\ 1)$$

We now rewrite (13) using the definitions of cv_add1 and add1:

$$\vdash \mathsf{cv\_rep}\ \mathsf{T}\ (\mathsf{cv\_add1}\ (\mathsf{from\_num}\ n))\ \ \mathsf{from\_num}\ (\mathsf{add1}\ n)$$

Unfolding the definition of cv_rep shows precisely the correspondence between cv_add1 and add1 (this theorem is derived by our automation and saved for the user):

$$\vdash \mathsf{from\_num}\ (\mathsf{add1}\ n) = \mathsf{cv\_add1}\ (\mathsf{from\_num}\ n)$$

A postprocessing step derives a cv_rep judgement suitable for use in future translations which refer to add1. In particular, we transform the judgement to mirror Theorem (12), so that each argument variable results in a separate assumption.

$$\vdash \mathsf{cv\_rep}\ p\ cv\ \ \mathsf{from\_num}\ n \Rightarrow\ \ \mathsf{cv\_rep}\ p\ (\mathsf{cv\_add1}\ cv)\ \ \mathsf{from\_num}\ (\mathsf{add1}\ n)$$

11.5 Translating Recursive Functions

Translating recursive functions is more involved. We describe the process by example, using the following input definition:

$$\mathsf{num\_list}\ n \stackrel{\text{def}}{=} \mathsf{if}\ 0 < n\ \mathsf{then}\ n :: \mathsf{num\_list}\ (n - 1)\ \mathsf{else}\ []$$

First, we show the cv_rep judgement for if, which is unchanged from non-recursive translations. Note how the final precondition uses the condition of the if to guard the preconditions

of the two branches (p_2 and p_3).

$$\vdash \mathsf{cv\_rep}\ p_1\ c_1\ \ \mathsf{from\_bool}\ b \wedge \mathsf{cv\_rep}\ p_2\ c_2\ \mathit{from\_a}\ t \wedge \mathsf{cv\_rep}\ p_3\ c_3\ \mathit{from\_a}\ e \Rightarrow \mathsf{cv\_rep}\ (p_1 \wedge (b \Rightarrow p_2) \wedge (\neg b \Rightarrow p_3))\ (\mathsf{cv\_if}\ c_1\ c_2\ c_3)\ \mathit{from\_a}\ (\mathsf{if}\ b\ \mathsf{then}\ t\ \mathsf{else}\ e)$$

The key challenge with recursive functions is to produce cv_rep judgements for their recursive calls. We create trivial cv_rep judgements, much like the ones we produce for variables (Sect. 11.4). For example, the recursive call to num_list produces the judgement below. Here, the variable *cv_num_list* is a placeholder for the cv_num_list function we will soon define.

$$\vdash \mathsf{cv\_rep}\ p\ \mathit{cv}\ \ \mathsf{from\_num}\ n \Rightarrow \mathsf{cv\_rep}\ (p \wedge \mathsf{from\_list}\ \ \mathsf{from\_num}\ (\mathsf{num\_list}\ n) = \mathit{cv\_num\_list}\ (\mathsf{from\_num}\ n))\ (\mathit{cv\_num\_list}\ \mathit{cv})\ (\mathsf{from\_list}\ \ \mathsf{from\_num})\ (\mathsf{num\_list}\ n)$$

Using this cv_rep judgement, we continue the bottom-up derivation as usual to produce the following judgement for the right-hand side of num_list:

$$\begin{aligned}\vdash\ & \mathsf{cv\_rep} \\ & (0 < n \Rightarrow \\ & \quad \mathsf{from\_list}\ \ \mathsf{from\_num}\ (\mathsf{num\_list}\ (n-1)) = \mathit{cv\_num\_list}\ (\mathsf{from\_num}\ (n-1))) \\ & (\mathsf{cv\_if}\ (\mathsf{cv\_lt}\ (\mathsf{Num}\ 0)\ (\mathsf{from\_num}\ n)) \\ & \quad (\mathsf{Pair}\ (\mathsf{from\_num}\ n)\ (\mathit{cv\_num\_list}\ (\mathsf{cv\_sub}\ (\mathsf{from\_num}\ n)\ (\mathsf{Num}\ 1))))\ (\mathsf{Num}\ 0)) \\ & (\mathsf{from\_list}\ \ \mathsf{from\_num})\ (\mathsf{if}\ 0 < n\ \mathsf{then}\ n :: \mathsf{num\_list}\ (n-1)\ \mathsf{else}\ [])\end{aligned} \tag{14}$$

Now, we can define the cv function cv_num_list as before, by considering the second argument to the cv_rep judgement:

$$\begin{aligned}&\mathsf{cv\_num\_list}\ \mathit{cv\_n} \stackrel{\mathsf{def}}{=} \\ &\ \mathsf{cv\_if}\ (\mathsf{cv\_lt}\ (\mathsf{Num}\ 0)\ \mathit{cv\_n}) \\ &\quad (\mathsf{Pair}\ \mathit{cv\_n}\ (\mathsf{cv\_num\_list}\ (\mathsf{cv\_sub}\ \mathit{cv\_n}\ (\mathsf{Num}\ 1))))\ (\mathsf{Num}\ 0)\end{aligned}$$

This function is recursive, so it may require a termination proof. Our automation can derive simple termination proofs (or avoid them for tail-recursive functions); for more complicated situations the user may need to supply one.

We instantiate Theorem (14) with the newly defined cv_num_list, and rewrite it using the definitions of cv_num_list and num_list:

$$\begin{aligned}\vdash\ & \mathsf{cv\_rep} \\ & (0 < n \Rightarrow\ \mathsf{from\_list}\ \ \mathsf{from\_num}\ (\mathsf{num\_list}\ (n-1)) = \mathsf{cv\_num\_list}\ (\mathsf{from\_num}\ (n-1))) \\ & (\mathsf{cv\_num\_list}\ (\mathsf{from\_num}\ n))\ (\mathsf{from\_list}\ \ \mathsf{from\_num})\ (\mathsf{num\_list}\ n)\end{aligned}$$

If we make explicit the universal quantification over n and expand the definition of cv_rep, we can see that this theorem has a familiar structure:

$$\begin{aligned}\vdash \forall n.\ & (0 < n \Rightarrow \\ & \quad \mathsf{from\_list}\ \ \mathsf{from\_num}\ (\mathsf{num\_list}\ (n-1)) = \mathsf{cv\_num\_list}\ (\mathsf{from\_num}\ (n-1))) \Rightarrow \\ & \mathsf{from\_list}\ \ \mathsf{from\_num}\ (\mathsf{num\_list}\ n) = \mathsf{cv\_num\_list}\ (\mathsf{from\_num}\ n)\end{aligned}$$

This matches the antecedent of the induction theorem arising from the termination proof of num_list, namely:

$$\vdash (\forall n.\ (0 < n \Rightarrow P\ (n-1)) \Rightarrow P\ n) \Rightarrow \forall v.\ P\ v$$

A simple application of *modus ponens* gives the following:

$$\vdash \forall v.\ \mathsf{from\_list\ from\_num\ (num\_list}\ v) = \mathsf{cv\_num\_list\ (from\_num}\ v)$$

Again, postprocessing reformulates this for use in future translations of terms which refer to num_list:

$$\begin{aligned}\vdash\ & \mathsf{cv\_rep}\ p\ cv\ \mathsf{from\_num}\ n \Rightarrow \\ & \mathsf{cv\_rep}\ p\ (\mathsf{cv\_num\_list}\ cv)\ (\mathsf{from\_list\ from\_num})\ (\mathsf{num\_list}\ n)\end{aligned}$$

11.6 Translating Let-Bindings and Pattern Matching

Our examples so far have only bound variables at the top-level as function arguments. Interesting user input may contain local variable bindings due to let-bindings and pattern matching using case-expressions. We require slightly more involved cv_rep judgements to support these.

In HOL4, let-bindings are syntactic sugar for applications of the LET constant, which is simply defined as function application: $\mathsf{LET}\ f\ x \stackrel{\text{def}}{=} f\ x$. For example, the binding $\mathsf{let}\ y = x + 1\ \mathsf{in}\ z \times y$ desugars to $\mathsf{LET}\ (\lambda y.\ z \times y)\ (x+1)$. We use the following cv_rep judgement to translate $\mathsf{LET}\ f\ x$. Here, $from\_a$ is the from function for the type of the let-bound variable, and the second precondition, p_2, is a function.

$$\begin{aligned}\vdash\ & \mathsf{cv\_rep}\ p_1\ c_1\ from\_a\ x \wedge (\forall v.\ \mathsf{cv\_rep}\ (p_2\ v)\ (c_2\ (from\_a\ v))\ from\_b\ (f\ v)) \Rightarrow \\ & \mathsf{cv\_rep}\ (p_1 \wedge \forall v.\ v = x \Rightarrow p_2\ v)\ (\mathsf{LET}\ c_2\ c_1)\ from\_b\ (\mathsf{LET}\ f\ x)\end{aligned}$$

Note that the second assumption ($\forall v.$ cv_rep ...) is universally quantified: it must hold for *any* application of f to a variable v. Note too that we lift the application $from\_a\ v$ into the assumptions, not just v.

We use similar cv_rep judgements to translate pattern matching via case-expressions. In HOL4, case-expressions desugar to functions such as $\mathsf{list\_case}\ x\ nil\ cons$, in the case of lists. Here, x is the list being pattern matched: if it is empty, the pattern match is equal to nil, if it is non-empty, then $\exists h\ t.\ x = h :: t$ and the pattern match is equal to $cons\ h\ t$.

$$\begin{aligned}\mathsf{list\_case}\ []\ nil\ cons &\stackrel{\text{def}}{=} nil \\ \mathsf{list\_case}\ (h :: t)\ nil\ cons &\stackrel{\text{def}}{=} cons\ h\ t\end{aligned}$$

The cv_rep judgement for list_case is shown below. It handles local variable bindings in the non-empty case in much the same way as the cv_rep judgement for LET above.

$$\begin{aligned}&\vdash \mathsf{cv\_rep}\ p\ cv\ (\mathsf{from\_list}\ from\_a)\ x \wedge \mathsf{cv\_rep}\ p\_nil\ c\_nil\ from\_b\ nil \wedge \\ &\quad (\forall h\ t.\\ &\qquad \mathsf{cv\_rep}\ (p\_cons\ h\ t)\ (c\_cons\ (from\_a\ h)\ (\mathsf{from\_list}\ from\_a\ t))\ from\_b\ (cons\ h\ t)) \Rightarrow \\ &\qquad \mathsf{cv\_rep}\ (p \wedge (x = [] \Rightarrow p\_nil) \wedge \forall h\ t.\ x = h::t \Rightarrow p\_cons\ h\ t)\\ &\qquad (\mathsf{cv\_if}\ (\mathsf{cv\_ispair}\ cv)\ (c\_cons\ (\mathsf{cv\_fst}\ cv)\ (\mathsf{cv\_snd}\ cv))\ c\_nil)\ from\_b\\ &\qquad (\mathsf{list\_case}\ x\ nil\ cons)\end{aligned}$$

Our tooling automatically derives such cv_rep judgements for the *type_case* constant of each datatype it encounters.

11.7 Translating Partially Specified Functions

All functions in higher-order logic are total; however, they can be *partially specified* if they leave certain cases unspecified. For example, the function that extracts the head of a list is partially specified because it has no equation for the empty list case:

$$\mathsf{hd}\ (h::t) \stackrel{\mathsf{def}}{=} h$$

To translate such examples, we first transform the pattern match on the left-hand side of the definition to a case-expression on its right-hand side. For partially specified functions (such as hd above), this produces an assumption which ensures we are not in any of the unspecified cases. As the unspecified cases are now impossible, they can simply take the value of the HOL4 constant for an arbitrary value (arb, a fixed but unspecified value of any HOL4 type). For example, the definition of hd above becomes:

$$\vdash (\exists t\ h.\ v = h::t) \Rightarrow \mathsf{hd}\ v = \mathsf{case}\ v\ \mathsf{of}\ [] \Rightarrow \mathsf{arb} \mid h::t \Rightarrow h \tag{15}$$

There is no value in the cv type which corresponds to arb, so we use a trivial cv_rep theorem with a false precondition:

$$\vdash \mathsf{cv\_rep}\ \mathsf{F}\ (\mathsf{Num}\ 0)\ a\ \mathsf{arb}$$

The result of translating the hd function (as described in Sect. 11.4) remains a fully specified cv function, cv_hd:

$$\mathsf{cv\_hd}\ cv\_v \stackrel{\mathsf{def}}{=} \mathsf{cv\_if}\ (\mathsf{cv\_ispair}\ cv\_v)\ (\mathsf{cv\_fst}\ cv\_v)\ (\mathsf{Num}\ 0)$$

However, the partiality becomes clear in the user-presentable theorem relating hd and cv_hd. This theorem has a precondition which requires the input to satisfy a new constant defined by our automation, hd_pre:

$$\vdash \mathsf{hd\_pre}\ v \Rightarrow from\_a\ (\mathsf{hd}\ v) = \mathsf{cv\_hd}\ (\mathsf{from\_list}\ from\_a\ v)$$

$$\mathsf{hd\_pre}\ v \stackrel{\mathsf{def}}{=} (\exists t\ h.\ v = h::t) \wedge v \neq []$$

The definition of hd_pre has two equivalent conjuncts, which seems verbose until we consider their origins: the first is simply the precondition of (15); the second arises when translating the arb on the right-hand side of (15). To work with hd_pre, users should derive a readable lemma, such as $\forall v.\ \mathsf{hd\_pre}\ v \iff v \neq []$.

Partial specification and recursion interact to produce more interesting preconditions. For example, consider a function to extract the last element of a list:

$$\vdash \mathsf{last}\ (h :: t) = \mathsf{if}\ t = []\ \mathsf{then}\ h\ \mathsf{else}\ \mathsf{last}\ t$$

During translation, our tooling automatically defines the precondition for such a function as an inductive relation. For last, this relation has only one rule:

$$\frac{(\exists h.\ v = [h]) \vee \exists t\ h.\ v = h :: t \quad v \neq [] \quad \forall h\ t.\ v = h :: t \Rightarrow t \neq [] \Rightarrow \mathsf{last\_pre}\ t}{\mathsf{last\_pre}\ v}$$

In manual proof, one can easily show that $\forall v.\ \mathsf{last\_pre}\ v \iff v \neq []$ by induction.

During development of this part of our tool, *we discovered a neat trick*: the induction theorem arising from the definition of last_pre closely resembles the induction theorem required when translating a fully specified recursive function, as described at the end of Sect. 11.5, and can be automatically transformed to match exactly. This allows a mode of operation in which users can instruct our tooling to treat an input recursive function as partially specified: this has the effect of outsourcing the induction proof of Sect. 11.5 to the user. This can be useful when the induction theorem required for the proof in Sect. 11.5 cannot be found, or does not quite suffice.

11.8 Translating Higher-Order Functions

The cv type is unable to represent any function type with an infinite domain (e.g. any function which accepts a natural number as input). Therefore from_to (Sect. 11.1) cannot hold of most interesting function types, making it impossible for our tool to translate higher-order functions in general.

However, we can handle functions with finite domains, as well as uses of higher-order functions that can be rephrased as first-order characterisations.

11.8.1 Functions with Finite Domains

Consider a HOL function $f : \mathsf{bool} \to \mathsf{num}$. We can represent this in the cv type as a pair whose first element is the result of applying the function to true, and whose second value is the function applied to false:

$$\mathsf{Pair}\ (\mathsf{Num}\ (f\ \mathsf{T}))\ (\mathsf{Num}\ (f\ \mathsf{F}))$$

In other words, the cv representation is a lookup table for the function: an exhaustive enumeration of its input–output behaviour. We can then represent the application f T using cv_fst (similarly cv_snd for the false case).

This idea generalises to any function with a finite domain: we can represent it as a pair which encapsulates a lookup table, and represent its application as a projection from the pair. Our automation is sufficiently extensible that users can define the *from*/*to* functions for

such representations, and pass the corresponding from_ to theorem to the tooling for use in translation.

We have exercised this ability on a small example of addressable memory.[8] In particular, we represent a memory with 256-bit addresses which stores natural number values as the HOL4 type 256 word → num. We define lookup and update functions for this memory, manually define cv versions of these, and derive cv_rep theorems which relate the two. Again, our automation is sufficiently extensible that users can supply such manually derived cv_rep theorems for use in translation.

However, we cannot exhaustively enumerate all possible input–output pairs for a function with 2^{256} possible input values. Instead, we make a small optimisation: our lookup table consists of a default output value, and a series of input–output pairs for which the output values differs from the default. In this way, we can efficiently represent a sparsely populated memory as a cv value.

11.8.2 First-Order Characterisations

We have implemented automation that recognises common patterns (e.g. mapping over a list with a concrete function), and proves them equivalent to first-order characterisations. A preprocessing phase rewrites the original pattern to its first-order characterisation before translation. In practice, we have found that the preprocessing allows our tool to remove most common uses of higher-order functions such as map, filter, and so on.

For example, suppose an input function contains map add1 l for some list l. The preprocessing defines a copy of map with its function argument specialised to add1:

$$\mathsf{map\_add1}\ x \stackrel{\text{def}}{=} \mathsf{map}\ \mathsf{add1}\ x$$

It then uses the definition of map to derive first-order equations for map_add1:

$$\vdash \mathsf{map\_add1}\ [] = [] \wedge \forall h\ t.\ \mathsf{map\_add1}\ (h :: t) = \mathsf{add1}\ h :: \mathsf{map\_add1}\ t$$

Preprocessing then rewrites any uses of map add1 to map_add1 so that the main part of the translator never sees this higher-order function.

12 Evaluation: HOL4

In this section, we evaluate the HOL4 implementation of our fast new computation feature (Sect. 10) and its associated automation (Sect. 11). We demonstrate its performance by exercising it on significant benchmarks: in-logic evaluation of the CakeML type inferencer [23] and in-logic self-compilation of the CakeML compiler backend [22]. We demonstrate its usability by example: we showcase the user experience for the in-logic evaluation of several small examples. These include partially specified and recursive functions.

[8] https://github.com/HOL-Theorem-Prover/HOL/blob/718d989/examples/cv_compute/finite_fun_exampleScript.sml.

12.1 Performance

Using our new fast compute feature and associated automation, we have enabled fast in-logic evaluation of existing ordinary HOL4 functions: CakeML's type inferencer and CakeML's compiler backend. In particular, we first fed these functions to our automation so that it could produce cv code equations. Then, we invoked our automation on several existing evaluation proofs which use these functions. Previously, the proofs relied on HOL4's existing in-logic evaluation facilities, known as EVAL (Sect. 13). We have been able to speed them up significantly using our fast computation, without significantly exposing the cv type. The table below summarises the improvement.

	EVAL	This work
Type inferencer	$\sim$ 2 hours	$<$ 1 second
Compiler backend	$\sim$ 14.5 hours	$\sim$ 45 minutes

12.1.1 CakeML Type Inferencer

In-logic evaluation of CakeML's type inferencer is crucial to the proof of soundness of the Candle theorem prover. Previously, this took more than two hours on an Intel® desktop machine with 64 GB RAM. Now, evaluation takes less than a second on the same machine. Of course, we incur the cost of our automation translating the inferencer functions to cv code equations. However, this takes less than 2 min on the same Intel® machine, producing 90 cv equations.

One interesting aspect of this translation was the need to first re-express the (type) unification algorithm (ultimately from [14]) tail-recursively. The input algorithm's natural expression involves both nested recursion (when a type has more than one sub-type), and partiality (it requires a well-formed input substitution). Transforming to tail-recursive, CPS, form is semi-automatic (mostly by rewriting), but, because cv-values cannot include abstractions, the continuations need to be defunctionalised. We were guided by Danvy and Nielsen [8], and ultimately generated a (verified equivalent) work-list version of the unification algorithm.

This process can be illustrated by the handling of the (slightly less) complicated substitution function used in the unification algorithm (Fig. 5). This recursively applies a substitution map (the s parameter) over the three possible forms of inference terms: constants (UConst), variables (UVar), and applications of operators (identified by numbers n) to lists of terms (UApp). Constants are left alone, applications have the algorithm applied recursively to the list argument, and variables are looked up in the substitution map with the function cvwalk before the substitution can be applied again.

The first step of the transformation (to CPS) is handled by judicious application of the contify function, which has the almost trivial definition:

$$\mathsf{contify}\ k\ v \stackrel{\mathsf{def}}{=} k\ v$$

More important are two illustrative consequences, a handling of "normal" function applications, and of if-then-else:

$$\vdash \mathsf{contify}\ k\ (f\ x)\ =\ \mathsf{contify}\ (\lambda fk.\ \mathsf{contify}\ (\lambda xk.\ \mathsf{cwc}\ (fk\ xk)\ k)\ x)\ f$$

$$\vdash \mathsf{contify}\ k\ (\mathsf{if}\ g\ \mathsf{then}\ t\ \mathsf{else}\ e)\ =\ \mathsf{contify}\ (\lambda gk.\ \mathsf{if}\ gk\ \mathsf{then}\ \mathsf{contify}\ k\ t\ \mathsf{else}\ \mathsf{contify}\ k\ e)\ g$$

The first equation refers to the cwc constant, which is semantically identical to contify but takes its two arguments in the opposite order. Using a second constant means that a naïve rewriting strategy using contify behaves well. The second equation makes it clear that when evaluating conditional forms, it doesn't make sense to chain the continuations through the evaluation of the then and else branches. Instead, both sub-terms are transformed with the same continuation k. Similar forms are required for case-constants that discriminate on data type constructors (these lie behind the case $\cdots$ of $\cdots$ syntax, as mentioned in Sect. 11.6).

The transformation of csubst starts by defining an auxiliary (cwalkstarl) to handle the recursive calls to map (csubst s), and allows the definition of a continuation-passing $\mathsf{csubst_k}$, subsuming both. The definition of $\mathsf{csubst_k}\ s\ itms\ k$ is then no more than

$$\mathsf{contify}\ k\ (\mathsf{cwalkstarl}\ s\ itms)$$

Using the "contification" rewrites above, it is then straightforward to derive the characterisation in Fig. 6.

The last step is defunctionalisation, generating "work-list" versions of these functions. There are four different continuation abstractions in Fig. 6: one for each constructor form, and one (beginning $\lambda\, r_2. \ldots$) that wraps the inner UApp continuation. Each abstraction has just one continuation variable free (k in all cases), so the concrete type we use to represent continuations is naturally a list, where each element of the list bundles the other free variables of the abstraction (except the unvarying substitution map s). Thus the form for the UApp continuation is APk, which takes a list of terms and a number (corresponding to the variables r_2 and n respectively). The fourth form has the same type, but in this case the list of terms corresponds to the *rest* variable.

This leads to the definition of the kclkont type:

$$\mathsf{kclkont}\ =\ \mathsf{UCk}\ \mathsf{num}\ |\ \mathsf{APk}\ (\mathsf{infer\_t}\ \mathsf{list})\ \mathsf{num}\ |\ \mathsf{UVk}\ \mathsf{num}\ |\ \mathsf{CONSk}\ (\mathsf{infer\_t}\ \mathsf{list})\ \mathsf{num}$$

With the continuation abstractions encoded as the type kclkont list, the next step of defunctionalisation is to define what it is to apply such a continuation to a result (a list of inference terms). The combination of this application function and the original is presented in Fig. 7.

Fig. 5 csubst: the equation defining the substitution function. Its precondition, cwfs s, requires that the lookup-tree s be well-formed

$$\begin{array}{l}
\vdash \mathsf{cwfs}\ s\ \Rightarrow \\
\quad \mathsf{csubst}\ s\ itm\ = \\
\quad\ \ \mathsf{case}\ itm\ \mathsf{of} \\
\quad\ \ \ \ \mathsf{UConst}\ c\ \Rightarrow\ \mathsf{UConst}\ c \\
\quad\ \ \ \ |\ \mathsf{UApp}\ l\ n\ \Rightarrow\ \mathsf{UApp}\ (\mathsf{map}\ (\mathsf{csubst}\ s)\ l)\ n \\
\quad\ \ \ \ |\ \mathsf{UVar}\ v\ \Rightarrow \\
\quad\ \ \ \ \ \ \mathsf{case}\ \mathsf{cvwalk}\ s\ v\ \mathsf{of} \\
\quad\ \ \ \ \ \ \ \ \mathsf{UConst}\ x\ \Rightarrow\ \mathsf{UConst}\ x \\
\quad\ \ \ \ \ \ \ \ |\ \mathsf{UApp}\ l\ n\ \Rightarrow\ \mathsf{UApp}\ (\mathsf{map}\ (\mathsf{csubst}\ s)\ l)\ n \\
\quad\ \ \ \ \ \ \ \ |\ \mathsf{UVar}\ v_4\ \Rightarrow\ \mathsf{UVar}\ v_4
\end{array}$$

```
⊢ csubst_k s itms k =
    case itms of
      [] ⇒ k []
    | UConst c::rest ⇒ csubst_k s rest (λ r1. k (UConst c::r1))
    | UApp l n::rest ⇒ csubst_k s l (λ r2. csubst_k s rest (λ r1. k (UApp r2 n::r1)))
    | UVar v::rest ⇒
      case tcvwalk s v of
        UConst c ⇒ csubst_k s rest (λ r1. k (UConst c::r1))
      | UApp l n ⇒ csubst_k s l (λ r2. csubst_k s rest (λ r1. k (UApp r2 n::r1)))
      | UVar v ⇒ csubst_k s rest (λ r1. k (UVar v::r1))
```

Fig. 6 csubst$_k$: the equation defining the continuation-passing substitution function. Here and later we omit the cwfs *s* assumption. The earlier lookup function cvwalk has also been replaced by a tail-recursive reformulation tcvwalk

```
csubst_wl s v itms k ≝
  if v then
    case k of
      [] ⇒ itms
    | UCk n::k ⇒ csubst_wl s T (UConst n::itms) k
    | APk itms0 n::k ⇒ csubst_wl s T (UApp itms0 n::itms) k
    | UVk n::k ⇒ csubst_wl s T (UVar n::itms) k
    | CONSk itms0 n::k ⇒ csubst_wl s F itms0 (APk itms n::k)
  else
    case itms of
      [] ⇒ csubst_wl s T [] k
    | UConst c::itms ⇒ csubst_wl s F itms (UCk c::k)
    | UApp itms0 n::itms ⇒ csubst_wl s F itms0 (CONSk itms n::k)
    | UVar v::itms ⇒
      case tcvwalk s v of
        UConst c ⇒ csubst_wl s F itms (UCk c::k)
      | UApp itms0 n ⇒ csubst_wl s F itms0 (CONSk itms n::k)
      | UVar v ⇒ csubst_wl s F itms (UVk v::k)
```

Fig. 7 The final, tail-recursive, first-order formulation of substitution over inference terms. Rather than present this as two mutually recursive functions, the fact that they have the same type allows us to have one function with a boolean flag *v* to pick between the two "modes". When *v* is true, the reified continuation *k* is applied to argument *itms*; when false, the *itms* list is processed

Once tail-recursive and first-order, the side-conditions around partiality are cleanly handled by the methods above (Sect. 11.7) and the cv-translation proceeds without further difficulty.

12.1.2 CakeML Compiler Backend

We extended our use of fast computation for CakeML's entire compiler backend, once again using our automation to replace a previous EVAL-based approach. As with the type inferencer above, we needed to re-express some functions to fit the new setting; however, the necessary changes were much more limited. In particular, various compiler functions operated over

lists of expressions, using a nested pattern match to deconstruct the expression at the head of the list. We expressed (verified equivalent) versions as a mutual recursion between two functions: one which operates over single expressions, and one which operates over lists of expressions.

We incorporated our use of fast computation into CakeML's regression testing framework. Each regression test bootstraps the compiler by verified self-compilation [17], that is, by in-logic evaluation of the compiler backend on itself. The process is repeated for multiple targets: x64 (64- and 32-bit) and Silver, a custom ISA for a verified processor [16]. Previously it took around 2.5 days per run, now it takes less than a day. This has made it feasible to add an additional target (Arm 64-bit) without significantly bloating run times.

The table below summarises run times for these parts of the regression test.

	EVAL	This work
x64 64-bit	∼14.5 hours	∼45 minutes
x64 32-bit	∼7 hours	∼50 minutes
Silver	∼8 hours	∼40 minutes
Arm 64-bit	–	∼55 minutes

12.2 Usability

We have already demonstrated that our automation is usable in significant projects by applying it to CakeML. Here, we give a further flavour of its usability using three small examples:

- Parity checking, defined in mutual recursion;
- Incrementing each element of a list: the `add1` and `map_add1` examples from Sects. 11.4 and 11.8 respectively; and
- Functional quicksort.

For each example, we show how the user can invoke our automation in a HOL4 REPL. We display user input on the left, and HOL4 output on the right. We superficially simplify HOL4 output for ease of reading, and elide output that is not interesting.

12.2.1 Parity Checking:

even and odd

Consider the following mutually recursive definition, where the `monospaced` metalanguage identifier on the left is bound to the object language definition on the right:

`even_odd_def:`

$$\begin{array}{ll} \mathsf{even}\ 0 \stackrel{\text{def}}{=} \mathsf{T} & \mathsf{odd}\ 0 \stackrel{\text{def}}{=} \mathsf{F} \\ \mathsf{even}\ (\mathsf{Suc}\ n) \stackrel{\text{def}}{=} \mathsf{odd}\ n & \mathsf{odd}\ (\mathsf{Suc}\ n) \stackrel{\text{def}}{=} \mathsf{even}\ n \end{array}$$

The user can invoke our automation using the `cv_trans` and `cv_eval` entrypoints as follows:

```
> cv_trans even_odd_def;
                                                                    …
                    Finished translating even, odd, stored in cv_even_thm
```

```
> Theorem even_999 = cv_eval ``even 999'';
                                                    even_999 = ⊢ even 999 ⟺ F
```

Using `cv_trans`, our automation has successfully translated even and odd into code equations: it has defined cv versions cv_even and cv_odd, using their tail-recursion to establish termination. The saved theorem `cv_even_thm` is then as follows:

$$\texttt{cv\_even\_thm:} \quad \vdash \mathsf{from\_bool}\ (\mathsf{even}\ v) = \mathsf{cv\_even}\ (\mathsf{Num}\ v) \wedge \mathsf{from\_bool}\ (\mathsf{odd}\ v) = \mathsf{cv\_odd}\ (\mathsf{Num}\ v)$$

The user can then perform fast computation using even and odd using the `cv_eval` entrypoint, as demonstrated by the theorem `even_999`.

12.2.2 Incrementing Each Element of a List: *add1_list*

Consider the following definitions:

$$\texttt{add1\_def:} \quad \mathsf{add1}\ n \stackrel{\text{def}}{=} n + 1$$

$$\texttt{add1\_list\_def:} \quad \mathsf{add1\_list}\ l \stackrel{\text{def}}{=} \mathsf{map}\ \mathsf{add1}\ l$$

The user can invoke our automation as follows.

```
> cv_auto_trans add1_list_def;

                    Starting translation of add1_list from exampleTheory.
                                                                        …
                     Starting translation of map_add1 from exampleTheory.
                         Starting translation of add1 from exampleTheory.
                                                                        …
                Finished translating add1_list, stored in cv_add1_list_thm

> Theorem add1_123 = cv_eval ``add1_list [1; 2; 3]'';

                                        add1_123 = ⊢ add1_list [1; 2; 3] = [2; 3; 4]
```

Here, we have used the `cv_auto_trans` entrypoint, which is a more automatic version of `cv_trans`. In particular, it has:

- Determined that to translate `add1_list_def`, it must first translate `add1_def`.
- Followed the procedure described in Sect. 11.8 to produce and translate a first-order characterisation of map add1.

The printed output logs the steps the automation has taken to produce the final translation of `add1_list_def`. The final saved theorem is as follows:

$$\texttt{cv\_add1\_list\_thm:} \quad \mathsf{from\_list\ Num}\ (\mathsf{add1\_list}\ l) \stackrel{\text{def}}{=} \mathsf{cv\_add1\_list}\ (\mathsf{from\_list\ Num}\ l)$$

As before, `cv_eval` can then be used to perform fast computation using add1_list.

12.2.3 Functional Quicksort: *qsort*

Consider the following standard definition of functional quicksort:

```
qsort_def:   ⊢ qsort l =
                 if length l < 2 then l
                 else
                  let pivot = hd l;
                  (left ,right) = partition pivot (tl l)
                  in
                    qsort left @ [pivot] @ qsort right
```

We elide the definition of partition for brevity. There are two key features to note:

- This function is not obviously terminating, so HOL4 cannot automatically infer its termination. To convince HOL4, the user must prove that partition preserves list lengths, and so the recursive calls to qsort are on strictly smaller lists.
- This function is not obviously total, due to its use of hd which is not specified on empty lists. Fortunately, the guard on the length of *l* ensures hd is applied to a non-empty list.

The user cannot invoke `cv_trans` or `cv_auto_trans` to translate qsort: these will be unable to prove termination of the cv version cv_qsort. Also, both entrypoints will fail if their translations produce any preconditions, ensuring preconditions cannot be silently introduced. The use of cv_hd in cv_qsort would produce such a precondition as described in Sect. 11.7. Therefore we use a different entrypoint, `cv_trans_pre_rec`:

```
> cv_trans partition_def;
                                                                                …
               Finished translating partition, stored in cv_partition_thm

> cv_trans_pre_rec qsort_def termination_tactic;

                      Finished translating qsort, stored in cv_qsort_thm

                    WARNING: definition of cv_qsort has a precondition.
                      You can set up the precondition proof as follows:
                                                                                …
> Theorem qsort_pre[cv_pre]:
∀l. qsort_pre l
Proof
Induct using qsort_ind >> Cases_on `l` >> rw[qsort_pre_cases]
QED
                                                                                …
> Theorem qsort_321 = cv_eval ``qsort [3; 2; 1]``;

                                          qsort_321 = ⊢ qsort [3; 2; 1] = [1; 2; 3]
```

First, the user translates `partition_def` as usual. Then they invoke `cv_trans_pre_rec` with the additional argument `termination_tactic`. This is a tactic which must convince HOL4 that `cv_qsort` terminates. Proofs of termination for cv functions can be a little fiddly, but they mostly echo the proofs for the ordinary functions from which the cv versions were derived. In this case, the tactic relies on a lemma that cv_partition preserves the sizes of its cv terms (a 3-line proof), and shows that cv_qsort is always called on strictly smaller arguments (a 4-line proof).

Note that during interactive development, the user would first invoke `cv_trans_pre qsort_def` to discover the necessary termination goal. This entrypoint does not fail if the

translation produces a precondition, and will return the precondition and any termination goal to the user. After establishing an appropriate tactic to solve the termination goal, the user can then pass it to `cv_trans_pre_rec`.

This leaves the precondition, qsort_pre. The user must define a theorem which proves the precondition before they can use qsort for fast computation. Here, we show the necessary theorem using standard HOL4 syntax with a `cv_pre` annotation. Theorems with this annotation are passed to our automation, which will ensure the precondition is appropriately discharged. The short proof above is all that is required, because the precondition boils down to showing that a list with length not less than two is non-empty. The proof uses induction because it must discharge the preconditions arising from the recursive calls to qsort. Here, we rewrite (`rw`) using `qsort_pre_cases`: the automatically derived inversion lemma for the inductive relation qsort_pre, which has been defined according to Sect. 11.7.

13 Related Work

This section discusses related work in the area of computation in ITPs.

13.1 HOL4

Barras implemented a fast interpreter for terms in HOL4 [6], usually referred to as EVAL. This is effectively a derived rule as described in Sect. 1. EVAL implements an extended version of Crégut's abstract machine KN [7], and performs strong reduction of open terms. It supports user-defined datatypes and pattern-matching, as well as rewriting using user-supplied conversions. It is this EVAL function that was used when benchmarking HOL4 in Sect. 8 and evaluating CakeML's type inferencer and compiler backend in Sect. 12.

Unlike our work in Candle, EVAL operates directly on HOL terms, though the automation described in Sect. 11 reduces this gap in HOL4. The HOL4 kernel was modified by Barras to make this as efficient as possible: the HOL4 kernel uses de Bruijn terms and explicit substitutions to ensure that EVAL runs fast. However, true to LCF tradition, all interpreter steps are implemented using basic kernel inferences.

13.2 HOL Light

A HOL Light port of EVAL exists [21] and was used in the performance comparisons of Sect. 8. However, unlike HOL4, the HOL Light kernel has not been optimised for running EVAL; HOL Light uses name-carrying terms without explicit substitutions, making this port comparably slow.

13.3 Isabelle/HOL

Isabelle/HOL supports two mechanisms for efficient evaluation, both due to Haftmann and Nipkow. A code generation feature [11, 12] can be used to synthesise ML, Haskell and Scala programs from closed terms, which can then be compiled and executed efficiently. We borrow the concept of code equations (Sect. 5) from their work, but note that Isabelle's code equations are more general than ours.

The second option is based on a normalisation-by-evaluation (NBE) mechanism [5] and synthesises ad-hoc ML interpreters over an untyped lambda calculus datatype from (possibly open) HOL terms. The ML code is compiled and executed by an ML compiler, and the resulting values are reinterpreted as HOL terms.

Both methods support a rich, higher-order, computable fragment of HOL. However, both also escape the logic, make use of unverified functions for synthesising functional programs, and rely on unverified compilers and language runtimes for execution.

13.4 Dependent Type Theories

Computation is an integral part of ITPs based on higher-order type theories, such as Coq [24], and Lean [9]. Their logics identify terms up to normal form and must reduce terms as part of their proof checking (i.e., type checking). Consequently, their trusted kernels must implement an interpreter or compiler of some sort.

Coq supports proof by computation using its interpreter (accessible via vm_compute), as well as native code generation to OCaml (accessible via native_compute). Internally, Coq's interpreter implements an extended version of the ZAM machine used in the interactive mode of the OCaml compiler [10], but with added support for open terms.

A formalisation of the abstract machine used in the interpreter exists [10], but the actual Coq implementation is completely unverified.

13.5 First-Order Logic

ACL2 is an ITP for a quantifier-free first-order logic with recursive, untyped functions. It axiomatises a purely functional fragment of Common Lisp, which doubles as term syntax and host language for the system. As a consequence, some terms can be compiled and executed at native speed. However, this execution speed comes at a cost: no verified Lisp compiler exists that can host ACL2 and its soundness-critical code encompasses essentially the entire theorem prover.

14 Conclusion

We have added efficient functions for computation to two HOL ITPs: Candle and HOL4. Using Candle, we developed our computation function in stages and verified its soundness: it can only produce theorems that follow from the inference rules of higher-order logic. For HOL4, we further built automation to provide a simple user interface to the function.

Our new compute function requires all input to be first-order computations over a Lisp-inspired datatype for compute values (cval/ cv). Our automation provides a user-friendly flow to invoking the new compute function: users can write ordinary definitions and evaluate them without interacting with the first-order format.

In our experiments, this new computation functionality performed several orders of magnitude faster than in-logic evaluation mechanisms provided by mainstream HOL ITPs. Using our automation, we have successfully exercised it on a significant existing benchmark, the CakeML compiler backend, with an order of magnitude speedup. At present, the performance numbers suggest that we do not need to go to the trouble of replacing our interpreter-based solution with a solution that compiles the given input to native machine code for extra performance. However, future case studies might lead us to explore such options too.

We envision that future case studies might explore how facilities for fast in-logic computation might open the door to for verified decision procedures (for linear arithmetic, linear algebra, or word problems) in HOL provers. Such proof procedures have typically been programmed in the meta language (SML and OCaml) of HOL provers.

Acknowledgements We want to thank Jeremy Avigad, John Harrison, Tobias Nipkow and Freek Wiedijk for feedback we received when the first author prepared this as a chapter for his PhD thesis [3]. We thank Thomas Sewell for showing us how to benchmark in-logic evaluation in Isabelle/HOL; and Paulo Torrens for pointing at us some relevant literature for CPS translation and defunctionalisation. The first two authors were supported by the Swedish Foundation for Strategic Research.

Author Contributions O.A. and M.M. wrote most of the manuscript. H.K., M.N. and J.Å.P. contributed to the sections that are new in this extended version of the original conference paper.

Funding Open access funding provided by University of Gothenburg.

Data Availability No datasets were generated or analysed during the current study.

References

1. Abrahamsson, O., Myreen, M.O., Kumar, R., Sewell, T.: Candle: a verified implementation of HOL Light. In: Andronick, J., de Moura, L. (eds.) Interactive Theorem Proving (ITP). LIPIcs, vol. 237, pp. 3:1–3:17 Springer, Cham (2022)
2. Abrahamsson, O., Myreen, M.O.: Fast, verified computation for Candle. In: Naumowicz, A., Thiemann, R. (eds.) Interactive Theorem Proving (ITP). LIPIcs, vol. 268, pp. 4:1–4:17. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPIcs.ITP.2023.4
3. Abrahamsson, O.: A verified theorem prover for higher-order logic. PhD thesis, Chalmers University of Technology (2022)
4. Abrahamsson, O., et al.: Proof-producing synthesis of CakeML from monadic HOL functions. J. Autom. Reason. **64**, 1287–1306 (2020)
5. Aehlig, K., Haftmann, F., Nipkow, T.: A compiled implementation of normalisation by evaluation. J. Funct. Program. **22**, 9–30 (2012)
6. Barras, B.: Programming and computing in HOL. In: Aagaard, M., Harrison, J. (eds.) Theorem Proving in Higher Order Logics (TPHOLs). Lecture Notes in Computer Science, vol. 1869, pp. 17–37. Springer, Berlin (2000)
7. Crégut, P.: An abstract machine for lambda-terms normalization. In: Kahn, G. (ed.) Conference on LISP and Functional Programming (LFP), pp. 333–340. ACM, New York (1990)
8. Danvy, O., Nielsen, L.R.: Defunctionalization at work. Tech. Rep. BRICS RS-01-23, BRICS, Department of Computer Science, University of Aarhus (2001)
9. de Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: Platzer, A., Sutcliffe, G. (eds.) International Conference on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 12699, pp. 625–635. Springer, Berlin (2021)
10. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: Wand, M., Jones, S.L.P. (eds.) International Conference on Functional Programming (ICFP), pp. 235–246. ACM, New York (2002)
11. Haftmann, F.: Code generation from specifications in higher-order logic. PhD thesis, Technical University Munich (2009)

12. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) Functional and Logic Programming (FLOPS). Lecture Notes in Computer Science, vol. 6009, pp. 103–117. Springer, Berlin (2010)
13. Harrison, J.: HOL light: an overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics (TPHOLs). Lecture Notes in Computer Science, vol. 5674, pp. 60–66. Springer, Berlin (2009)
14. Kumar, R., Norrish, M.: (nominal) unification by recursive descent with triangular substitutions. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving (ITP), 2010. Proceedings. Lecture Notes in Computer Science, vol. 6172, pp. 51–66. Springer, Berlin (2010). https://doi.org/10.1007/978-3-642-14052-5_6
15. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Jagannathan, S., Sewell, P. (eds.) Principles of Programming Languages (POPL), pp. 179–192. ACM, New York (2014). https://doi.org/10.1145/2535838.2535841
16. Lööw, A., et al.: Verified compilation on a verified processor. In: McKinley, K.S., Fisher, K. (eds.) Programming Language Design and Implementation (PLDI), pp. 1041–1053. ACM, New York (2019). https://doi.org/10.1145/3314221.3314622
17. Myreen, M.O.: A minimalistic verified bootstrapped compiler (proof pearl). In: Hritcu, C., Popescu, A. (eds.) International Conference on Certified Programs and Proofs (CPP), pp. 32–45. ACM, New York (2021). https://doi.org/10.1145/3437992.3439915
18. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic. Lecture Notes in Computer Science, vol. 2283. Springer, Berlin (2002)
19. Pohjola, J. Å., Rostedt, H., Myreen, M.O.: Characteristic formulae for liveness properties of non-terminating CakeML programs. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) Interactive Theorem Proving, ITP. LIPIcs, vol. 141, pp. 32:1–32:19. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.32
20. Slind, K., Norrish, M.: A brief overview of HOL4. In: Mohamed, O.A., Muñoz, C.A., Tahar, S. (eds.) Theorem Proving in Higher Order Logics (TPHOLs). LNCS, vol. 5170. Springer, Berlin (2008)
21. Solovyev, A.: HOL Light's computelib. https://github.com/jrh13/hol-light/blob/master/compute.ml. Accessed 11 June 2022
22. Tan, Y.K., et al.: The verified CakeML compiler backend. J. Funct. Program. (2019). https://doi.org/10.1017/S0956796818000229
23. Tan, Y.K., Owens, S., Kumar, R.: A verified type system for CakeML. In: Lämmel, R. (ed.) Implementation and Application of Functional Programming Languages (IFL). ACM, New York (2015)
24. The Coq Development Team: The Coq reference manual. https://coq.inria.fr/distrib/current/refman/. Accessed 11 June 2022