Fast, Verified Computation for Candle

Oskar Abrahamsson ✉ 🆔
Chalmers University of Technology, Gothenburg, Sweden

Magnus O. Myreen ✉ 🆔
Chalmers University of Technology, Gothenburg, Sweden

───── **Abstract** ─────

This paper describes how we have added an efficient function for computation to the kernel of the Candle interactive theorem prover. Candle is a CakeML port of HOL Light which we have, in prior work, proved sound w.r.t. the inference rules of the higher-order logic. This paper extends the original implementation and soundness proof with a new kernel function for fast computation. Experiments show that the new computation function is able to speed up certain evaluation proofs by several orders of magnitude.

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:

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

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 [11], HOL4 [13], and Isabelle/HOL [12] 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:

  $\mathsf{Bit1}\ \mathsf{m} + \mathsf{Bit0}\ \mathsf{n} = \mathsf{Bit1}\ (m + n)$
  $\mathsf{Bit1}\ \mathsf{m} + \mathsf{Bit1}\ \mathsf{n} = \mathsf{Bit0}\ (\mathsf{Suc}\ (m + n))$
  $\mathsf{Suc}\ (\mathsf{Bit0}\ \mathsf{n}) = \mathsf{Bit1}\ \mathsf{n}$
  $\mathsf{Suc}\ (\mathsf{Bit1}\ \mathsf{n}) = \mathsf{Bit0}\ (\mathsf{Suc}\ n)$
  $\cdots$

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.

This paper is about how we have added a fast function for computation to the Candle HOL ITP[1]. Candle has a different soundness argument that allows it to move away from being simple in order to be trustworthy: Candle has been proved (in HOL4) to be sound w.r.t. a formal semantics of higher-order logic [3].

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) 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 (Sec. 8).

Contributions

We make the following contributions:

- We implement a fast interpreter for terms as a user-accessible primitive in the Candle kernel. The implementation allows users to supply code equations dictating how user-defined (recursive) functions are to be interpreted.
- The new primitive has been proven correct with respect to the inference rules of higher-order logic, and has been fully integrated into the existing end-to-end soundness proof of the Candle ITP.
- Our compute function is, in our experiments, significantly faster than the equivalent runs of in-logic compute facilities provided by other HOL ITPs.

---

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

$$\begin{array}{ccc}
\textsf{sum } [5;\ 9;\ 1] & \xleftarrow{\quad\text{rewriting}\quad} & 15 \\
\Big\downarrow & & \Big\uparrow \\
\text{HOL term to IR} & & \text{IR to HOL term} \\
\textsf{App "sum" [N 5; N 9; N 1]} & \xrightarrow{\text{interpreter}} & \textsf{N 15}
\end{array}$$
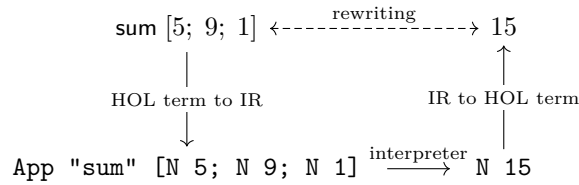
■ **Figure 1** Diagram illustrating the approach we take to embedding logical terms into compute expressions and evaluating them using an interpreter.

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

This paper contains syntax at multiple, potentially confusing levels. The Candle logic is formalized 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 $\vdash_c$.

Source code and proofs

Our sources are at github.com/CakeML/cakeml/tree/master/candle/prover/compute, and the Candle project is hosted at cakeml.org/candle.

2 Approach

This section explains, at a high level, the approach we have taken to add a new function for computation to Candle.

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 Figure 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 Figure 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 Figure 1. The new compute primitive returns, to the user, a theorem stating that the input term is equal to the result of computation r. The theorem states that an equality between the points connected with a dashed arrow in Figure 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 Figure 1. All reasoning about the interpreter (the lower horizontal arrow) must be wrt. 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.

2.1   Overview

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

1. Version 1 (Sec. 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[2] 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 (Sec. 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 (Sec. 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 (Sec. 6) replaced the naive interpreter with one that is designed to evaluate with less overhead. This version uses $O(1)$ operations to look up to code equations and uses environments rather than substitutions for variable bindings. This is the version we perform benchmarks on (Sec. 8).

5. The final Version 5 (Sec. 7) is, at the time of writing, left as future work. In Version 5, our intention is to split the compute function into stages so that users can initialize and feed in code equations separately from calls to the main compute function. This should make repeated calls to the compute facility faster.

At the time of writing, Version 4 (Sec. 6) is integrated into the existing Candle implementation and end-to-end soundness proof.

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 (Sec. 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.

---

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

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.

$$
\begin{aligned}
&\vdash \text{num\_thy\_ok } \Gamma \implies \\
&\quad \Gamma \vdash_c \text{mk\_num } m +_c \text{mk\_num } n =_c \\
&\qquad \text{mk\_num } (m + n)
\end{aligned}
\tag{2}
$$

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:

$$
\begin{aligned}
&\text{mk\_num } n \overset{\text{def}}{=} \\
&\quad \text{if } n = 0 \text{ then } 0_c \\
&\quad \text{else if even } n \text{ then Bit0}_c \ (\text{mk\_num } (n \text{ div } 2)) \\
&\quad \text{else Bit1}_c \ (\text{mk\_num } (n \text{ div } 2))
\end{aligned}
$$

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

$$
\begin{aligned}
&\text{num\_thy\_ok } \Gamma \overset{\text{def}}{=} \\
&\quad \Gamma \vdash_c 0_c +_c n =_c n \wedge \\
&\quad \Gamma \vdash_c \text{Suc}_c \ m +_c n =_c \text{Suc}_c \ (m +_c n) \wedge \\
&\quad \Gamma \vdash_c \text{Bit0}_c \ n =_c n +_c n \wedge \\
&\quad \Gamma \vdash_c \text{Bit1}_c \ n =_c \text{Suc}_c \ (n +_c n) \wedge \ldots
\end{aligned}
$$

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 characterized by its simple Suc-based equations and Bit1$_c$ is characterized 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:

$$
\begin{aligned}
&\text{mk\_suc } n \overset{\text{def}}{=} \\
&\quad \text{if } n = 0 \text{ then } 0_c \\
&\quad \text{else Suc}_c \ (\text{mk\_suc } (n - 1))
\end{aligned}
$$

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

$$\vdash \textsf{num\_thy\_ok } \Gamma \Rightarrow$$
$$\Gamma \vdash_{\overline{\textsf{c}}} \textsf{mk\_suc } m +_{\textsf{c}} \textsf{mk\_suc } n =_{\textsf{c}} \qquad\qquad (3)$$
$$\textsf{mk\_suc } (m + n)$$

and the second lemma $=_{\textsf{c}}$-equates mk_num with mk_suc:

$$\vdash \textsf{num\_thy\_ok } \Gamma \Rightarrow$$
$$\Gamma \vdash_{\overline{\textsf{c}}} \textsf{mk\_num } n =_{\textsf{c}} \textsf{mk\_suc } n \qquad\qquad (4)$$

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

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 initialization, the computation function converts the given input term into its internal representation (see the leftmost arrow in Figure 1).

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

$$\textsf{dest\_num } tm \overset{\text{def}}{=}$$
$$\textsf{case } tm \textsf{ of}$$
$$\mid 0_{\textsf{c}} \Rightarrow \textsf{ Some } 0$$
$$\mid \textsf{Bit0}_{\textsf{c}} \ r \Rightarrow \textsf{ option\_map } (\lambda \, n.\ 2 \times n) \ (\textsf{dest\_num } r)$$
$$\mid \textsf{Bit1}_{\textsf{c}} \ r \Rightarrow \textsf{ option\_map } (\lambda \, n.\ 2 \times n + 1) \ (\textsf{dest\_num } r)$$
$$\mid \_ \Rightarrow \textsf{ None}$$

One should read the application $\textsf{Bit} b \ bs$ 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 $=_{\textsf{c}}$ is preserved when moving from Candle terms to natural numbers in HOL4, and back:

$$\vdash \textsf{num\_thy\_ok } \Gamma \wedge$$
$$\textsf{dest\_num } t = \textsf{Some } t' \Rightarrow \qquad\qquad (5)$$
$$\Gamma \vdash_{\overline{\textsf{c}}} \textsf{mk\_num } t' =_{\textsf{c}} t$$

Version 1 of the computation function also has a function for taking apart a Candle term with a top-level addition $+_{\textsf{c}}$:

$$\textsf{dest\_add } tm \overset{\text{def}}{=}$$
$$\textsf{case } tm \textsf{ of}$$
$$\mid (x +_{\textsf{c}} y) \Rightarrow \textsf{ Some } (x,y)$$
$$\mid \_ \Rightarrow \textsf{ None}$$

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.

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

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. Subsequent versions will remove this overhead (Sec. 7).

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 synthesized from these functions in the HOL4 logic, using the tool described in prior work [2].

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 generalizes 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:

$$\text{cval} = \text{Pair}_c \text{ cval cval}$$
$$\mid \text{Num}_c \text{ num}$$

4.2   Compute expressions

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

$$\text{cexp} = \text{Pair cexp cexp}$$
$$\mid \text{Num num}$$
$$\mid \text{If cexp cexp cexp}$$
$$\mid \text{Uop uop cexp}$$
$$\mid \text{Binop binop cexp cexp}$$

$$\text{uop} = \text{Fst} \mid \text{Snd} \mid \text{IsPair}$$

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

The cexp datatype is extended with more constructors in Version 3, described in Section 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:

$$\text{cif}_c \text{ (Num}_c \text{ 1) (Num}_c \text{ 2)}$$
$$\text{(fst}_c \text{ (Pair}_c \text{ (Num}_c \text{ 3) (Num}_c \text{ 4)))}$$

the function will create this cexp expression:

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

This mapping assumes that certain functions in the Candle logic (e.g. 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 formalize the assumptions about 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 characterizing equations. For Version 2, this assumption was extended to cover characterizing equations for all names that the conversion from user input to cexp recognizes: cif_c, fst_c, etc. These characterizing 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 fst_c (Num$_c$ 3), or add_c (Num$_c$ 3) (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:

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

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

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

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:

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

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

4.5 Soundness

The following theorem summarizes 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`):

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

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 characterizing 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 synthesized from these functions. 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.

<div style="border-left: 4px solid orange; padding-left: 10px;">

5   Recursion and user-supplied code equations (Version 3)

</div>

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:

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

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 may be 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 Section 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 ($\vdash_{\mathsf{c}}$). 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 parameterized by *defs* and a clock which is initialized 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.

$$
\begin{aligned}
&\vdash \mathsf{cexp\_thy\_ok}\ \Gamma\ \Rightarrow \\
&(\forall eq.\ \mathsf{mem}\ eq\ eqs\ \Rightarrow\ \Gamma \vDash_{\mathsf{c}} eq)\ \wedge \\
&\mathsf{dest\_eqs}\ eqs = \mathsf{Some}\ defs\ \wedge \\
&\mathsf{dest\_tm}\ defs\ t = \mathsf{Some}\ cexp\ \wedge \\
&\mathsf{interpret\ init\_ck}\ defs\ cexp = \mathsf{Some}\ res\ \Rightarrow \\
&\Gamma \vDash_{\mathsf{c}} t =_{\mathsf{c}} \mathsf{mk\_term}\ res
\end{aligned}
\tag{8}
$$

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. The user is not in any way obliged to prove this: the fact that they can supply the compute primitive with a list of theorems means that they are valid in Candle's context at that point. 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. This clock is not strictly necessary, but made it easier to use the existing CakeML code synthesis tools. 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.

5.4   CakeML code

As with previous versions, the CakeML implementation of the computation function is synthesized from the HOL4 functions. 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 Figure 2, for the ce datatype addresses the two main shortcomings of Version 3. First, it drops the substitution semantics in favor 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*)

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.

7   Staged set up (Version 5)

At the time of writing, Version 5 is not yet implemented. However, the plan is to reduce the overhead of calling the compute function.

In Versions 1–4, the characteristic equations need to be checked (i.e., establishing cexp_thy_ok) and the user-supplied code equations must be compiled, on each call to the compute primitive. In these versions, even a simple evaluation, such as $1 + 2$, will make the compute function check all of the characteristic equations, every time.

For Version 5, the plan is to curry the arguments of the compute primitive and arrange the implementation to: perform the characteristic equations checks after the first argument is given and then return a function that performs the rest of the computation given the remaining arguments. Note that the returned function can only exist if all of the characteristic equation checks have passed. The verification of the Candle prover has not yet dealt with any such conditionally existing function values. We expect these values will need special treatment in the Candle prover's soundness theorem.

exec *funs env ck* (Const *n*) $\overset{\text{def}}{=}$
  return (Num *n*)
exec *funs env ck* (Var *n*) $\overset{\text{def}}{=}$
  return (env_lookup *n env*)
exec *funs env ck* (Monop *m x*) $\overset{\text{def}}{=}$
  do
    v ← exec *funs env ck x*;
    return (*m v*)
  od
exec *funs env ck* (Binop *b x y*) $\overset{\text{def}}{=}$
  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*) $\overset{\text{def}}{=}$
  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*) $\overset{\text{def}}{=}$
  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*) $\overset{\text{def}}{=}$
  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* $\overset{\text{def}}{=}$
  return *acc*
execl *funs env ck* (*x*::*xs*) *acc* $\overset{\text{def}}{=}$
  do
    v ← exec *funs env ck x*;
    execl *funs env ck xs* (*v*::*acc*)
  od

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

```
fun exec funs env ck e =
  case e of
    Const n => Num n
  | Var n => List.nth n env
  | Monop m x =>
      let
        val v = exec funs env ck x
      in
        m v
      end
  | 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
```

Figure 3 CakeML code generated from definition of exec.

■ **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`. Below dash, —, indicates *not measured*.

fact n for different values of n

n	Candle	HOL4	H.Light	Isabelle
256	<1 ms	2.3 s	0.6 s	14 s
512	<1 ms	4.1 s	3.5 s	202 s
1024	<1 ms	127 s	17.6 s	2451 s
2048	11 ms	684 s	86.1 s	—
32768	0.9 s	—	—	—

primes_upto n for different values of n

n	Candle	HOL4	H.Light	Isabelle
256	<1 ms	0.5 s	1.3 s	2.6 s
512	<1 ms	1.6 s	5.2 s	9.8 s
1024	2 ms	6.3 s	20.7 s	35.6 s
2048	9 ms	24.2 s	83.4 s	132 s
32768	1.7 s	—	—	—

rev_enum n for different values of n

n	Candle	HOL4	H.Light	Isabelle
256	0.02 s	1.1 s	66.2 s	10.2 s
512	0.03 s	2.3 s	251 s	37.1 s
1024	0.07 s	4.7 s	1005 s	172 s
2048	0.1 s	9.5 s	4203 s	791 s
32768	2.5 s	—	—	—

n steps of Conway's Game of Life

n	Candle	HOL4	H.Light	Isabelle
1	0.03 s	0.6 s	14.9 s	1.5 s
10	0.08 s	5.3 s	147 s	15.0 s
100	0.8 s	54 s	1474 s	148 s
1000	8.0 s	568 s	14623 s	1466 s
10000	79 s	—	—	—

8 Evaluation

In this section, we report on experiments comparing our new compute function to the in-logic interpreters of HOL4, HOL Light, and Isabelle/HOL. We tested the performance of each on the following four example programs written as function 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 at cakeml.org/candle_benchmarks.html.

The results, in Figure 1, show that Candle's new compute function runs orders of magnitude faster than the derived rules 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 Figure 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.

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 out performs HOL4. We suspect HOL Light benefits from the effort that has gone into making basic arithmetic evaluate fast in HOL Light.

Prime enumeration

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.

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.

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 to have a set up that consists of five Gosper glider generators that 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 Related Work

This section discusses related work in the area of computation in interactive theorem provers.

9.1 HOL4

Barras implemented a fast interpreter for terms in HOL4 [5], usually referred to as EVAL. EVAL implements an extended version of Crégut's abstract machine KN [6], and performs strong reduction of open terms, and supports user-defined datatypes and pattern-matching, and rewriting using user-supplied conversions. It is this EVAL function that was used when benchmarking HOL4 in Section 8.

Unlike our work, EVAL operates directly on HOL terms. 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.

9.2 HOL Light

A HOL Light port of EVAL exists [14] and was used in Section 8. However, unlike HOL4, the HOL Light kernel has not been optimized for running EVAL; HOL Light uses name-carrying terms without explicit substitutions, making this port comparably slow.

9.3 Isabelle/HOL

Isabelle/HOL supports two mechanisms for efficient evaluation, both due to Haftmann and Nipkow. A code generation feature [9, 10] can be used to synthesize ML, Haskell and Scala programs from closed terms, which can then be compiled and executed efficiently. We borrow the concept of code equations (Sec. 5) from their work, but note that Isabelle's code equations are more general than ours.

The second option is based on normalization-by-evaluation (NBE) mechanism [4] and synthesizes 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 synthesizing functional programs, and rely on unverified compilers and language runtimes for execution.

9.4   Dependent type theories

Computation is an integral part of ITPs based on higher-order type theories, such as Coq [15], and Lean [7]. 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 [8], but with added support for open terms.

A formalization of the abstract machine used in the interpreter exists [8], but the actual Coq implementation is completely unverified.

9.5   First-order logic

ACL2 is an ITP for a quantifier-free first-order logic with recursive, untyped functions. It axiomatizes 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, its soundness critical code encompasses essentially the entire theorem prover.

10   Conclusion

We have added a new verified function for computation to the Candle ITP. The new computation function was developed in stages through different versions. For each version, we proved that the new function only produces theorems that follow by the inference rules of HOL. In our experiments, Candle's new computation functionality produced performance numbers that are several orders of magnitude faster than in-logic evaluation mechanisms provided by mainstream HOL ITPs.

Our new compute function requires all functions that it uses to be first-order functions that perform all computations using a Lisp-inspired datatype for compute values (`cval`). We leave it to future work to relax this requirement.

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.

──── **References** ────

1 Oskar Abrahamsson. *A Verified Theorem Prover for Higher-Order Logic*. PhD thesis, Chalmers University of Technology, 2022.

2 Oskar Abrahamsson, Son Ho, Hrutvik Kanabar, Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Yong Kiam Tan. Proof-producing synthesis of CakeML from monadic HOL functions. *Journal of Automated Reasoning (JAR)*, 2020. URL: `https://rdcu.be/b4FrU`.

3 Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, and Thomas Sewell. Candle: A verified implementation of HOL Light. In June Andronick and Leonardo de Moura, editors, *Interactive Theorem Proving (ITP)*, volume 237 of *LIPIcs*, pages 3:1–3:17, 2022. `doi:10.4230/LIPIcs.ITP.2022.3`.

4 Klaus Aehlig, Florian Haftmann, and Tobias Nipkow. A compiled implementation of normalisation by evaluation. *J. Funct. Program.*, 22(1):9–30, 2012. `doi:10.1017/S0956796812000019`.

5 Bruno Barras. Programming and computing in HOL. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1869 of *Lecture Notes in Computer Science*, pages 17–37. Springer, 2000. `doi:10.1007/3-540-44659-1_2`.

6 Pierre Crégut. An abstract machine for lambda-terms normalization. In Gilles Kahn, editor, *Conference on LISP and Functional Programming (LFP)*, pages 333–340. ACM, 1990. `doi:10.1145/91556.91681`.

7 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *International Conference on Automated Deduction (CADE)*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. `doi:10.1007/978-3-030-79876-5_37`.

8 Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Mitchell Wand and Simon L. Peyton Jones, editors, *International Conference on Functional Programming (ICFP)*, pages 235–246. ACM, 2002. `doi:10.1145/581478.581501`.

9 Florian Haftmann. *Code generation from specifications in higher-order logic*. PhD thesis, Technical University Munich, 2009.

10 Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming (FLOPS)*, volume 6009 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 2010. `doi:10.1007/978-3-642-12251-4_9`.

11 John Harrison. HOL Light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66. Springer, 2009. `doi:10.1007/978-3-642-03359-9_4`.

12 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. `doi:10.1007/3-540-45949-9`.

13 Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of *LNCS*. Springer, 2008. `doi:10.1007/978-3-540-71067-7_6`.

14 Alexey Solovyev. HOL Light's computelib. Accessed 2022-06-11. `https://github.com/jrh13/hol-light/blob/master/compute.ml`.

15 The Coq Development Team. The Coq reference manual. Accessed 2022-06-11. `https://coq.inria.fr/distrib/current/refman/`.