A Verified Compiler from Isabelle/HOL to CakeML

Lars Hupel and Tobias Nipkow

Technische Universität München
lars.hupel@tum.de, nipkow@in.tum.de

Abstract. Many theorem provers can generate functional programs from definitions or proofs. However, this code generation needs to be trusted. Except for the HOL4 system, which has a proof producing code generator for a subset of ML. We go one step further and provide a verified compiler from Isabelle/HOL to CakeML. More precisely we combine a simple proof producing translation of recursion equations in Isabelle/HOL into a deeply embedded term language with a fully verified compilation chain to the target language CakeML.

Keywords: Isabelle, CakeML, compiler, higher-order term rewriting

1 Introduction

Many theorem provers have the ability to generate executable code in some (typically functional) programming language from definitions, lemmas and proofs (e.g. [6,8,9,12,15,26,36]). This makes code generation part of the trusted kernel of the system. Myreen and Owens [29] closed this gap for the HOL4 system: they have implemented a tool that translates from HOL4 into *CakeML*, a subset of SML, and proves a theorem stating that a result produced by the CakeML code is correct w.r.t. the HOL functions. They also have a verified implementation of CakeML [23,39]. We go one step further and provide a once-and-for-all verified compiler from (deeply embedded) function definitions in Isabelle/HOL [31,32] into CakeML proving partial correctness of the generated CakeML code w.r.t. the original functions. This is like the step from dynamic to static type checking. It also means that preconditions on the input to the compiler are explicitly given in the correctness theorem rather than implicitly by a failing translation. To the best of our knowledge this is the first verified (as opposed to certifying) compiler from function definitions in a logic into a programming language.

Our compiler is composed of multiple phases and in principle applicable to other languages than Isabelle/HOL or even HOL:

- We erase types right away. Hence the type system of the source language is irrelevant.
- We merely assume that the source language has a semantics based on equational logic.

The compiler operates in three stages:

1. The preprocessing phase eliminates features that are not supported by our compiler. Most importantly, *dictionary construction* eliminates occurrences of type classes in HOL terms. It introduces dictionary datatypes and new constants and proves the equivalence of old and new constants (§7).
2. The *deep embedding* lifts HOL terms into terms of type term, a HOL model of HOL terms. For each constant c (of arbitrary type) it defines a constant c' of type term and proves a theorem that expresses equivalence (§3).
3. There are multiple *compiler phases* that eliminate certain constructs from the term type, until we arrive at the CakeML expression type. Most phases target a different intermediate term type (§5).

The first two stages are preprocessing, are implemented in ML and produce certificate theorems. Only these stages are specific to Isabelle. The third (and main) stage is implemented completely in the logic HOL, without recourse to ML. Its correctness is verified once and for all. All Isabelle definitions and proofs can be found in the supplementary material.

2 Related work

There is existing work in the Coq [2, 14] and HOL [29] communities for proof producing or verified extraction of functions defined in the logic. Anand *et al.* [2] present work in progress on a verified compiler from Gallina (Coq's specification language) via untyped intermediate languages to CompCert C light. They plan to connect their extraction routine to the CompCert compiler [25].

Translation of type classes into dictionaries is an important feature of Haskell compilers. In the setting of Isabelle/HOL, this has been described by Wenzel [43] and Krauss *et al.* [22]. Haftmann and Nipkow [16] use this construction to compile HOL definitions into target languages that do not support type classes, e.g. Standard ML and OCaml. In this work, we provide a certifying translation that eliminates type classes inside the logic.

Compilation of pattern matching is well understood in literature [3, 35, 37]. In this work, we contribute a transformation of sets of equations with pattern matching on the left-hand side into a single equation with nested pattern matching on the right-hand side. This is implemented and verified inside Isabelle.

Besides CakeML, there are many projects for verified compilers for functional programming languages of various degrees of sophistication and realism (e.g. [4, 11, 13]). Particularly modular is the work by Neis *et al.* [30] on a verified compiler for an ML-like imperative source language. The main distinguishing feature of our work is that we start from a set of higher-order recursion equations with pattern matching on the left-hand side rather than a lambda calculus with pattern matching on the right-hand side. On the other hand we stand on the shoulders of CakeML which allows us to bypass all complications of machine code generation. Note that much of our compiler is not specific to CakeML and that it would be possible to retarget it to, for example, Pilsner abstract syntax with moderate effort.

```
datatype term =
  Const string |
  Free string |
  Abs term |
  Bound nat |
  App term term
```

(a) Abstract syntax of de Bruijn terms

$$\textsc{Step}\ \frac{(\mathit{lhs}, \mathit{rhs}) \in R \qquad \mathsf{match}\ \mathit{lhs}\ t = \mathsf{Some}\ \sigma}{R \vdash t \longrightarrow \mathsf{subst}\ \sigma\ \mathit{rhs}}$$

$$\textsc{Beta}\ \frac{\mathsf{closed}\ t'}{R \vdash (\Lambda t)\ \$\ t' \longrightarrow t[t']} \qquad \textsc{Fun}\ \frac{R \vdash t \longrightarrow t'}{R \vdash t\ \$\ u \longrightarrow t'\ \$\ u}$$

$$\textsc{Arg}\ \frac{R \vdash u \longrightarrow u'}{R \vdash t\ \$\ u \longrightarrow t\ \$\ u'}$$

(b) Small-step semantics

Fig. 1: Basic syntax and semantics of the term type

3 Deep embedding

Starting with a HOL definition, we derive a new, *reified* definition in a deeply embedded term language depicted in Figure 1a. This term language corresponds closely to the `term` datatype of Isabelle's implementation (using de Bruijn indices [10]), but without types and schematic variables.

To establish a formal connection between the original and the reified definitions, we use a *logical relation*, a concept that is well-understood in literature [19] and can be nicely implemented in Isabelle using type classes. Note that the use of type classes here is restricted to correctness proofs; it is not required for the execution of the compiler itself. That way, there is no contradiction to the elimination of type classes occurring in a previous stage.

Notation We abbreviate $\mathsf{App}\ t\ u$ to $t \$ u$ and $\mathsf{Abs}\ t$ to $\Lambda\ t$. Other term types introduced later in this paper use the same conventions. We reserve λ for abstractions in HOL itself. Typing judgments are written with a double colon: $t :: \tau$.

Embedding operation Embedding is implemented in ML. We denote this operation using angle brackets: $\langle t \rangle$, where t is an arbitrary HOL expression and the result $\langle t \rangle$ is a HOL value of type term. It is a purely syntactic transformation, without preliminary evaluation or reduction, and it discards type information. The following examples illustrate this operation and typographical conventions concerning variables and constants:

$$\langle x \rangle = \mathsf{Free}\ \texttt{"x"} \qquad \langle \mathsf{f} \rangle = \mathsf{Const}\ \texttt{"f"} \qquad \langle \lambda x.\ \mathsf{f}\ x \rangle = \Lambda\ (\langle \mathsf{f} \rangle\ \$\ \mathsf{Bound}\ 0)$$

Small-step semantics Figure 1b specifies the small-step semantics for term. It is reminiscent of *higher-order term rewriting*, and modelled closely after equality in HOL. The basic idea is that if the proposition $t = u$ can be proved equationally in HOL (without symmetry), then $R \vdash \langle t \rangle \longrightarrow^* \langle u \rangle$ holds (where $R :: (\mathsf{term} \times \mathsf{term})\ \mathsf{set}$). We call R the *rule set*. It is the result of translating a set of defining equations $\mathit{lhs} = \mathit{rhs}$ into pairs $(\langle \mathit{lhs} \rangle, \langle \mathit{rhs} \rangle) \in R$.

Rule Step performs a rewrite step by picking a rewrite rule from R and rewriting the term at the root. For that purpose, match and subst are (mostly) standard first-order matching and substitution (see §4 for details).

Rule Beta performs β-reduction. Type term represents bound variables by de Bruijn indices. The notation $t[t']$ represents the substitution of the outermost bound variable in t with t'.

Our semantics does not constitute a fully-general higher-order term rewriting system, because we do not allow substitution under binders. For de Bruijn terms, this would pose no problem, but as soon as we introduce named bound variables, substitution under binders requires dealing with capture. To avoid this altogether, all our semantics expect terms that are substituted into abstractions to be closed. However, this does not mean that we restrict ourselves to any particular evaluation order. Both call-by-value and call-by-name can be used in the small-step semantics. But later on, the target semantics will only use call-by-value.

Embedding relation We denote the concept that an embedded term t corresponds to a HOL term a of type τ w.r.t. rule set R with the syntax $R \vdash t \approx a$. If we want to be explicit about the type, we index the relation: $\approx_\tau$.

For ground types, this can be defined easily. For example, the following two rules define $\approx_{\mathsf{nat}}$:

$$\frac{}{R \vdash \langle 0 \rangle \approx_{\mathsf{nat}} 0} \qquad \frac{R \vdash \langle t \rangle \approx_{\mathsf{nat}} n}{R \vdash \langle \mathsf{Suc}\ t \rangle \approx_{\mathsf{nat}} \mathsf{Suc}\ n}$$

Definitions of $\approx$ for arbitrary datatypes without nested recursion can be derived mechanically in the same fashion as for nat, where they constitute one-to-one relations. Note that for ground types, $\approx$ ignores R. The reason why $\approx$ is parametrized on R will become clear in a moment.

For function types, we follow Myreen and Owen's approach [29]. The statement $R \vdash t \approx f$ can be interpreted as "$t \mathbin{\$} \langle a \rangle$ can be rewritten to $\langle f\ a \rangle$ for all a". Because this might involve applying a function definition from R, the $\approx$ relation must be indexed by the rule set. As a notational convenience, we define another relation $R \vdash t \downarrow x$ to mean that there is a t' such that $R \vdash t \longrightarrow^* t'$ and $R \vdash t' \approx x$. Using this notation, we formally define $\approx$ for functions as follows:

$$R \vdash t \approx f \leftrightarrow (\forall x\ t_x.\ R \vdash t_x \downarrow x \to R \vdash t \mathbin{\$} t_x \downarrow f\ x)$$

Example As a running example, we will use the map function on lists:

$$\begin{aligned} \mathsf{map}\ f\ []\ &= [] \\ \mathsf{map}\ f\ (x \mathbin{\#} xs) &= f\ x \mathbin{\#} \mathsf{map}\ f\ xs \end{aligned}$$

The result of embedding this function is a set of rules map':

```
map' =
  {(Const "List.list.map" $ Free "f" $ (Const "List.list.Cons" $ Free "x21" $ Free "x22"),
    Const "List.list.Cons" $ (Free "f" $ Free "x21") $ ...),
  (Const "List.list.map" $ Free "f" $ Const "List.list.Nil",
    Const "List.list.Nil")}
```

together with the theorem map$'$ $\vdash$ Const "List.list.map" $\downarrow$ map, which is proven by simple induction over map. Constant names like "List.list.map" come from the fully-qualified internal names in HOL.

The induction principle for the proof arises from the use of the **fun** command that is used to define recursive functions in HOL [21]. But the user is also allowed to specify custom equations for functions, in which case we will use heuristics to generate and prove the appropriate induction theorem. For simplicity, we will use the term *(defining) equation* uniformly to refer to any set of equations, either default ones or ones specified by the user. Embedding partially-specified functions – in particular, proving the certificate theorem about them – is currently not supported. In the future, we plan to leverage the domain predicate as produced by **fun** to generate conditional theorems.

4 Terms, matching and substitution

The compiler transforms the initial term type (Figure 1a) through various intermediate stages. This section gives an overview and introduces necessary terminology.

Preliminaries The function arrow in HOL is $\Rightarrow$. The cons operator on lists is the infix #.

Throughout the paper, the concept of *mappings* is pervasive: We use the type notation $\alpha \rightharpoonup \beta$ to denote a function $\alpha \Rightarrow \beta$ option. In certain contexts, a mapping may also be called an *environment*. We write mapping literals using brackets: $[a \Rightarrow x, b \Rightarrow y, \ldots]$. If it is clear from the context that σ is defined on a, we often treat the lookup $\sigma\ a$ as returning an $x :: \beta$.

The functions dom $:: (\alpha \rightharpoonup \beta) \Rightarrow \alpha$ set and range $:: (\alpha \rightharpoonup \beta) \Rightarrow \beta$ set return the *domain* and *range* of a mapping, respectively.

Dropping entries from a mapping is denoted by $\sigma - k$, where σ is a mapping and k is either a single key or a set of keys. We use $\sigma' \subseteq \sigma$ to denote that σ' is a sub-mapping of σ, that is, dom $\sigma' \subseteq$ dom σ and $\forall a \in$ dom $\sigma'.\ \sigma'\ a = \sigma\ a$.

Merging two mappings σ and ρ is denoted with $\sigma +\!\!+ \rho$. It constructs a new mapping with the union domain of σ and ρ. Entries from ρ override entries from σ. That is, $\rho \subseteq \sigma +\!\!+ \rho$ holds, but not necessarily $\sigma \subseteq \sigma +\!\!+ \rho$.

All mappings and sets are assumed to be finite. In the formalization, this is enforced by using subtypes of $\rightharpoonup$ and set. Note that one cannot define datatypes by recursion through sets for cardinality reasons. However, for finite sets, it is possible. This is required to construct the various term types. We leverage facilities of Blanchette *et al.*'s **datatype** command to define these subtypes [7].

Standard functions All type constructors that we use ($\rightharpoonup$, set, list, option, ...) support the standard operations map and rel. For lists, map is the regular covariant map. For mappings, the function has the type $(\beta \Rightarrow \gamma) \Rightarrow (\alpha \rightharpoonup \beta) \Rightarrow (\alpha \rightharpoonup \gamma)$. It leaves the domain unchanged, but applies a function to the range of the mapping.

Function rel_τ lifts a binary predicate $P :: \alpha \Rightarrow \alpha \Rightarrow \mathsf{bool}$ to the type constructor τ. We call this lifted relation the *relator* for a particular type.

For datatypes, its definition is structural, for example:

$$\frac{}{\mathsf{rel}_{\mathsf{list}}\ P\ []\ []} \qquad \frac{\mathsf{rel}_{\mathsf{list}}\ P\ xs\ ys \qquad P\ x\ y}{\mathsf{rel}_{\mathsf{list}}\ P\ (x\ \#\ xs)\ (y\ \#\ ys)}$$

For sets and mappings, the definition is a little bit more subtle.

Definition 1 (Set relator). *For each element $a \in A$, there must be a corresponding element $b \in B$ such that $P\ a\ b$, and vice versa. Formally:*

$$\mathsf{rel}_{\mathsf{set}}\ P\ A\ B \leftrightarrow (\forall x \in A.\ \exists y \in B.\ P\ x\ y) \wedge (\forall y \in B.\ \exists x \in A.\ P\ x\ y)$$

Definition 2 (Mapping relator). *For each a, $m\ a$ and $n\ a$ must be related according to* $\mathsf{rel}_{\mathsf{option}}\ P$. *Formally:*

$$\mathsf{rel}_{\mathsf{mapping}}\ P\ m\ n \leftrightarrow (\forall a.\ \mathsf{rel}_{\mathsf{option}}\ P\ (m\ a)\ (n\ a))$$

Term types There are four distinct term types: term, nterm, pterm, and sterm. All of them support the notions of free variables, matching and substitution. Free variables are always a finite set of strings. Matching a term against a *pattern* yields an optional mapping of type $\mathsf{string} \rightharpoonup \alpha$ from free variable names to terms.

Note that the type of patterns is itself term instead of a dedicated pattern type. The reason is that we have to subject patterns to a linearity constraint anyway and may use this constraint to carve out the relevant subset of terms:

Definition 3. *A term is* linear *if there is at most one occurrence of any variable, it contains no abstractions, and in an application $f\ \$\ x$, f must not be a free variable. The HOL predicate is called* $\mathsf{linear} :: \mathsf{term} \Rightarrow \mathsf{bool}$.

Because of the similarity of operations across the term types, they are all instances of the term type class. Note that in Isabelle, classes and types live in different namespaces. The term type and the term type class are separate entities.

Definition 4. *A* term type τ *supports the operations* $\mathsf{match} :: \mathsf{term} \Rightarrow \tau \Rightarrow (\mathsf{string} \rightharpoonup \tau)$, $\mathsf{subst} :: (\mathsf{string} \rightharpoonup \tau) \Rightarrow \tau \Rightarrow \tau$ *and* $\mathsf{frees} :: \tau \Rightarrow \mathsf{string\ set}$. *We also define the following derived functions:*

- matchs *matches a list of patterns and terms sequentially, producing a single mapping*
- closed t *is an abbreviation for* frees $t = \emptyset$
- closed σ *is an overloading of* closed, *denoting that all values in a mapping are closed*

Additionally, some (obvious) axioms have to be satisfied. We do not strive to fully specify an abstract term algebra. Instead, the axioms are chosen according to the needs of this formalization.

A notable deviation from matching as discussed in term rewriting literature is that the result of matching is only well-defined if the pattern is linear.

Definition 5. *An* equation *is a pair of a pattern* (left-hand side) *and a term* (right-hand side)*. The pattern is of the form $f \,\$\, p_1 \,\$\, \ldots \,\$\, p_n$, where f is a constant (i.e. of the form* Const *name). We refer to both f or name interchangeably as the* function symbol *of the equation.*

Following term rewriting terminology, we sometimes refer to an equation as *rule*.

4.1 De Bruijn terms (term)

The definition of term is almost an exact copy of Isabelle's internal term type, with the notable omissions of type information and schematic variables (Figure 1a). The implementation of β-reduction is straightforward via index shifting of bound variables.

4.2 Named bound variables (nterm)

datatype nterm = Nconst string | Nvar string | Nabs string nterm | Napp nterm nterm

The nterm type is similar to term, but removes the distinction between *bound* and *free* variables. Instead, there are only named variables. As mentioned in the previous section, we forbid substitution of terms that are not closed in order to avoid capture. This is also reflected in the syntactic side conditions of the correctness proofs (§5.1).

4.3 Explicit pattern matching (pterm)

datatype pterm =
  Pconst string | Pvar string | Pabs ((term × pterm) set) | Papp pterm pterm

Functions in HOL are usually defined using *implicit* pattern matching, that is, the terms p_i occurring on the left-hand side $\langle \mathsf{f}\ p_1\ \ldots\ p_n \rangle$ of an equation must be constructor patterns. This is also common among functional programming languages like Haskell or OCaml. CakeML only supports *explicit* pattern matching using case expressions. A function definition consisting of multiple defining equations must hence be translated to the form $f = \lambda x.\ \textbf{case}\ x\ \textbf{of}\ \ldots$. The elimination proceeds by iteratively removing the last parameter in the block of equations until none are left.

In our formalization, we opted to combine the notion of abstraction and case expression, yielding *case abstractions*, represented as the Pabs constructor. This is similar to the `fn` construct in Standard ML, which denotes an anonymous function that immediately matches on its argument [27]. The same construct also exists in Haskell with the `LambdaCase` language extension. We chose this representation mainly for two reasons: First, it allows for a simpler language grammar because there is only one (shared) constructor for abstraction and case expression. Second, the elimination procedure outlined above does not have to introduce fresh names in the process. Later, when translating to CakeML syntax, fresh names are introduced and proved correct in a separate step.

The set of pairs of pattern and right-hand side inside a case abstraction is referred to as *clauses*. As a short-hand notation, we use $\mathit{\Lambda}\{p_1 \Rightarrow t_1, p_2 \Rightarrow t_2, \ldots\}$.

4.4 Sequential clauses (sterm)

```
datatype sterm =
  Sconst string | Svar string | Sabs ((term × sterm) list) | Sapp sterm sterm
```

In the term rewriting fragment of HOL, the order of rules is not significant. If a rule matches, it can be applied, regardless when it was defined or proven. This is reflected by the use of sets in the rule and term types. For CakeML, the rules need to be applied in a deterministic order, i.e. sequentially. The sterm type only differs from pterm by using list instead of set. Hence, case abstractions use list brackets: $\mathit{\Lambda}[p_1 \Rightarrow t_1, p_2 \Rightarrow t_2, \ldots]$.

4.5 Irreducible terms (value)

CakeML distinguishes between *expressions* and *values*. Whereas expressions may contain free variables or β-redexes, values are closed and fully evaluated. Both have a notion of abstraction, but values differ from expressions in that they contain an environment binding free variables.

Consider the expression $(\lambda x.\lambda y.x)\ (\lambda z.z)$, which is rewritten (by β-reduction) to $\lambda y.\lambda z.z$. Note how the bound variable x disappears, since it is replaced. This is contrary to how programming languages are usually implemented: evaluation does not happen by substituting the argument term t for the bound variable x, but by recording the binding $x \mapsto t$ in an environment [23]. A pair of an abstraction and an environment is usually called a *closure* [24, 40].

In CakeML, this means that evaluation of the above expression results in the closure

$$(\lambda y.x, [\texttt{"x"} \mapsto (\lambda z.z, [])])$$

Note the nested structure of the closure, whose environment itself contains a closure.

To reflect this in our formalization, we introduce a type value of values (explanation inline):

```
datatype value =
  (∗ constructor value: a data constructor applied to multiple values ∗)
  Vconstr string (value list) |
  (∗ closure: clauses combined with an environment mapping variables to values ∗)
```

$\mathsf{Vabs}\ ((\mathsf{term} \times \mathsf{sterm})\ \mathsf{list})\ (\mathsf{string} \rightharpoonup \mathsf{value})\ |$
(∗ recursive closures: a group of mutually recursive function bodies with an environment ∗)
$\mathsf{Vrecabs}\ (\mathsf{string} \rightharpoonup ((\mathsf{term} \times \mathsf{sterm})\ \mathsf{list}))\ \mathsf{string}\ (\mathsf{string} \rightharpoonup \mathsf{value})$

The above example evaluates to the closure:

$$\mathsf{Vabs}\ \left[\,\langle y\rangle \Rightarrow \langle x\rangle\,\right]\ \left[\texttt{"x"} \mapsto \mathsf{Vabs}\ [\langle z\rangle \Rightarrow \langle z\rangle]\ []\right]$$

The third case for recursive closures only becomes relevant when we conflate variables and constants. As long as the rule set *rs* is kept separate, recursive calls are straightforward: the appropriate definition for the constant can be looked up there. CakeML knows no such distinction between constants and variables, hence everything has to reside in a single environment σ.

Consider this example of odd and even:

$$\begin{aligned} \mathsf{odd}\ 0 &= \mathsf{False} & \qquad \mathsf{even}\ 0 &= \mathsf{True} \\ \mathsf{odd}\ (\mathsf{Suc}\ n) &= \mathsf{even}\ n & \qquad \mathsf{even}\ (\mathsf{Suc}\ n) &= \mathsf{odd}\ n \end{aligned}$$

When evaluating the term odd k, the definitions of even and odd themselves must be available in the environment captured in the definition of odd. However, it would be cumbersome in HOL to construct such a Vabs that refers to itself. Instead, we capture the expressions used to define odd and even in a recursive closure. Other encodings might be possible, but since we are targeting CakeML, we are opting to model it in a similar way as its authors do.

For the above example, this would result in the following global environment:

$$[\texttt{"odd"} \mapsto \mathsf{Vrecabs}\ \mathit{css}\ \texttt{"odd"}\ [], \texttt{"even"} \mapsto \mathsf{Vrecabs}\ \mathit{css}\ \texttt{"even"}\ []]$$
$$\begin{aligned} \text{where } \mathit{css} = [&\texttt{"odd"} \mapsto [\langle 0\rangle \Rightarrow \langle \mathsf{False}\rangle\,, \langle \mathsf{Suc}\ n\rangle \Rightarrow \langle \mathsf{even}\ n\rangle], \\ &\texttt{"even"} \mapsto [\langle 0\rangle \Rightarrow \langle \mathsf{True}\rangle\,, \langle \mathsf{Suc}\ n\rangle \Rightarrow \langle \mathsf{odd}\ n\rangle]] \end{aligned}$$

Note that in the first line, the right-hand sides are values, but in *css*, they are expressions. The additional string argument of Vrecabs denotes the selected function. When evaluating an application of a recursive closure to an argument (β-reduction), the semantics adds all constituent functions of the closure to the environment used for recursive evaluation.

5 Intermediate semantics and compiler phases

In this section, we will discuss the progression from de Bruijn based term language with its small-step semantics given in Figure 1a to the final CakeML semantics. The compiler starts out with terms of type term and applies multiple phases to eliminate features that are not present in the CakeML source language. Types term, nterm and pterm each have a small-step semantics only. Type sterm has a small-step and several intermediate big-step semantics that bridge the gap to CakeML. An overview of the intermediate semantics and compiler phases is depicted in Figure 2. The left-hand column gives an overview of the different phases. The right-hand column gives the types of the rule set and the semantics for each phase; you may want to skip it upon first reading.

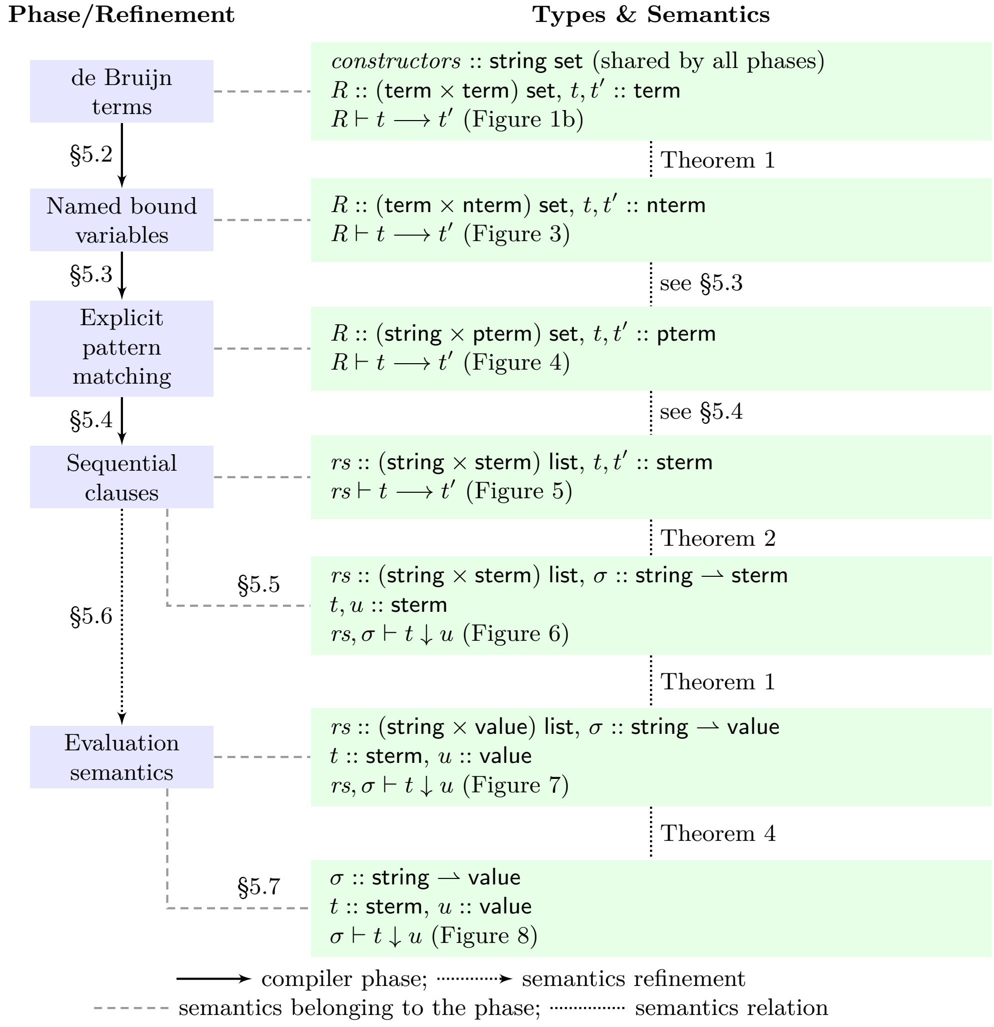
Phase/Refinement
Types & Semantics
de Bruijn terms
$constructors$ :: string set (shared by all phases)
R :: (term × term) set, t, t' :: term
$R \vdash t \longrightarrow t'$ (Figure 1b)
§5.2
Theorem 1
Named bound variables
R :: (term × nterm) set, t, t' :: nterm
$R \vdash t \longrightarrow t'$ (Figure 3)
§5.3
see §5.3
Explicit pattern matching
R :: (string × pterm) set, t, t' :: pterm
$R \vdash t \longrightarrow t'$ (Figure 4)
§5.4
see §5.4
Sequential clauses
rs :: (string × sterm) list, t, t' :: sterm
$rs \vdash t \longrightarrow t'$ (Figure 5)
Theorem 2
§5.5
rs :: (string × sterm) list, σ :: string $\rightharpoonup$ sterm
t, u :: sterm
$rs, \sigma \vdash t \downarrow u$ (Figure 6)
§5.6
Theorem 1
Evaluation semantics
rs :: (string × value) list, σ :: string $\rightharpoonup$ value
t :: sterm, u :: value
$rs, \sigma \vdash t \downarrow u$ (Figure 7)
Theorem 4
§5.7
σ :: string $\rightharpoonup$ value
t :: sterm, u :: value
$\sigma \vdash t \downarrow u$ (Figure 8)
compiler phase;
semantics refinement
semantics belonging to the phase;
semantics relation

Fig. 2: Intermediate semantics and compiler phases

5.1 Side conditions

All of the following semantics require some side conditions on the rule set. These conditions are purely syntactic. As an example we list the conditions for the correctness of the first compiler phase:

- Patterns must be linear, and constructors in patterns must be fully applied.
- Definitions must have at least one parameter on the left-hand side (§5.6).
- The right-hand side of an equation refers only to free variables occurring in patterns on the left-hand side and contain no dangling de Bruijn indices.

$$\textsc{Step}\ \frac{(\mathit{lhs}, \mathit{rhs}) \in R \qquad \mathsf{match}\ \mathit{lhs}\ t = \mathsf{Some}\ \sigma}{R \vdash t \longrightarrow \mathsf{subst}\ \sigma\ \mathit{rhs}} \qquad \textsc{Beta}\ \frac{\mathsf{closed}\ t'}{R \vdash (\Lambda x.\ t)\ \$\ t' \longrightarrow \mathsf{subst}\ \left[x \mapsto t'\right]\ t}$$

Fig. 3: Small-step semantics for nterm with named bound variables

- There are no two defining equations $\mathit{lhs} = \mathit{rhs}_1$ and $\mathit{lhs} = \mathit{rhs}_2$ such that $\mathit{rhs}_1 \neq \mathit{rhs}_2$.
- For each pair of equations that define the same constant, their arity must be equal and their patterns must be compatible (§5.3).
- There is at least one equation.
- Variable names occurring in patterns must not overlap with constant names (§5.7).
- Any occurring constants must either be defined by an equation or be a constructor.

The conditions for the subsequent phases are sufficiently similar that we do not list them again.

In the formalization, we use named contexts to fix the rules and assumptions on them (*locales* in Isabelle terminology). Each phase has its own locale, together with a proof that after compilation, the preconditions of the next phase are satisfied. Correctness proofs assume the above conditions on R and similar conditions on the term that is reduced. For brevity, this is usually omitted in our presentation.

5.2 Naming bound variables: From term to nterm

Isabelle uses de Bruijn indices in the term language for the following two reasons: For substitution, there is no need to rename bound variables. Additionally, α-equivalent terms are equal. In implementations of programming languages, these advantages are not required: Typically, substitutions do not happen inside abstractions, and there is no notion of equality of functions. Therefore CakeML uses named variables and in this compilation step, we get rid of de Bruijn indices.

The "named" semantics is based on the nterm type. The rules that are changed from the original semantics (Figure 1b) are given in Figure 3 (Fun and Arg remain unchanged). Notably, β-reduction reuses the substitution function.

For the correctness proof, we need to establish a correspondence between terms and nterms. Translation from nterm to term is trivial: Replace bound variables by the number of abstractions between occurrence and where they were bound in, and keep free variables as they are. This function is called nterm_to_term.

The other direction is not unique and requires introduction of *fresh* names for bound variables. In our formalization, we have chosen to use a *monad* to produce these names. This function is called term_to_nterm. We can also prove

$$\textsc{Beta}\ \frac{(pat, rhs) \in C \qquad \mathsf{match}\ pat\ t = \mathsf{Some}\ \sigma \qquad \mathsf{closed}\ t}{R \vdash (\Lambda\ C)\ \$\ t \longrightarrow \mathsf{subst}\ \sigma\ rhs}$$

$$\textsc{Step'}\ \frac{(name, rhs) \in R}{R \vdash \mathsf{Pconst}\ name \longrightarrow rhs}$$

Fig. 4: Small-step semantics for pterm with pattern matching

the obvious property nterm_to_term (term_to_nterm t) = t, where t is a term without dangling de Bruijn indices.

Generation of fresh names in general can be thought of as picking a string that is not an element of a (finite) set of already existing names. For Isabelle, the *Nominal* framework [41,42] provides support for reasoning over fresh names, but unfortunately, its definitions are not executable.

Instead, we chose to model generation of fresh names as a monad α fresh with the following primitive operations in addition to the monad operations:

$$\begin{aligned} \mathsf{run} &:: \alpha\ \mathsf{fresh} \Rightarrow \mathsf{string\ set} \Rightarrow \alpha \\ \mathsf{fresh\_name} &:: \mathsf{string\ fresh} \end{aligned}$$

In our implementation, we have chosen to represent α fresh as roughly isomorphic to the state monad.

Compilation of a rule set proceeds by translation of the right-hand side of all rules:

$$\mathsf{compile}\ R = \{(p, \mathsf{term\_to\_nterm}\ t) \mid (p, t) \in R\}$$

The left-hand side is left unchanged for two reasons: function match expects an argument of type term (see §4), and patterns do not contain abstractions or bound variables.

Theorem 1 (Correctness of compilation). *Assuming a step can be taken with the compiled rule set, it can be reproduced with the original rule set.*

$$\frac{\mathsf{compile}\ R \vdash t \longrightarrow u \qquad \mathsf{closed}\ t}{R \vdash \mathsf{nterm\_to\_term}\ t \longrightarrow \mathsf{nterm\_to\_term}\ u}$$

We prove this by induction over the semantics (Figure 3).

5.3 Explicit pattern matching: From nterm to pterm

Usually, functions in HOL are defined using *implicit* pattern matching, that is, the left-hand side of an equation is of the form $\langle \mathsf{f}\ p_1\ \ldots\ p_n \rangle$. For any given function f, there may be multiple such equations. In this compilation step, we transform sets of equations for f defined using implicit pattern matching into a single equation for f of the form $\langle \mathsf{f} \rangle = \Lambda\ C$, where C is a set of clauses.

The strategy we employ currently requires successive elimination of a single parameter from right to left, in a similar fashion as Slind's pattern matching compiler [37, §3.3.1]. Recall our running example (map). It has arity 2. We omit the brackets $\langle\rangle$ for brevity. First, the list parameter gets eliminated:

$$\begin{aligned} \mathsf{map}\ f = \lambda\ [] &\Rightarrow [] \\ \mid x \# xs &\Rightarrow f\ x \# \mathsf{map}\ f\ xs \end{aligned}$$

Finally, the function parameter gets eliminated:

$$\begin{aligned} \mathsf{map} = \lambda\ f \Rightarrow \big(\lambda\ [] &\Rightarrow [] \\ \mid x \# xs &\Rightarrow f\ x \# \mathsf{map}\ f\ xs\big) \end{aligned}$$

This has now arity 0 and is defined by a twice-nested abstraction.

Semantics The target semantics is given in Figure 4 (the Fun and Arg rules from previous semantics remain unchanged). We start out with a rule set R that allows only implicit pattern matching. After elimination, only explicit pattern matching remains. The modified Step rule merely replaces a constant by its definition, without taking arguments into account.

Restrictions For the transformation to work, we need a strong assumption about the structure of the patterns p_i to avoid the following situation:

$$\begin{aligned} &\mathsf{map}\ f\ [] &&= [] \\ &\mathsf{map}\ g\ (x \# xs) &&= g\ x \# \mathsf{map}\ g\ xs \end{aligned}$$

Through elimination, this would turn into:

$$\begin{aligned} \mathsf{map} = \lambda\ f &\Rightarrow \big(\lambda\ [] \Rightarrow []\big) \\ \mid g &\Rightarrow \big(\lambda\ x \# xs \Rightarrow f\ x \# \mathsf{map}\ f\ xs\big) \end{aligned}$$

Even though the original equations were non-overlapping, we suddenly obtained an abstraction with two overlapping patterns. Slind observed a similar problem [37, §3.3.2] in his algorithm. Therefore, he only permits *uniform* equations, as defined by Wadler [35, §5.5]. Here, we can give a formal characterization of our requirements as a computable function on pairs of patterns:

```
fun pat_compat :: term ⇒ term ⇒ bool where
pat_compat (t₁ $ t₂) (u₁ $ u₂) ↔ pat_compat t₁ u₁ ∧ (t₁ = u₁ → pat_compat t₂ u₂)
pat_compat t u ↔ (overlapping t u → t = u)
```

This compatibility constraint ensures that any two overlapping patterns (of the same column) $p_{i,k}$ and $p_{j,k}$ are equal and are thus appropriately grouped together in the elimination procedure. We require all defining equations of a constant to be mutually compatible. Equations violating this constraint will be flagged during embedding (§3), whereas the pattern elimination algorithm always succeeds.

While this rules out some theoretically possible pattern combinations (e.g. the *diagonal* function [35, §5.5]), in practice, we have not found this to be a problem:

$$\textsc{Step}\ \frac{(\mathit{name}, \mathit{rhs}) \in R}{R \vdash \mathsf{Sconst}\ \mathit{name} \longrightarrow \mathit{rhs}} \qquad \textsc{Beta}\ \frac{\mathsf{first\_match}\ \mathit{cs}\ t = \mathsf{Some}\ (\sigma, \mathit{rhs}) \qquad \mathsf{closed}\ t}{R \vdash (\Lambda\ \mathit{cs})\ \$\ t \longrightarrow \mathsf{subst}\ \sigma\ \mathit{rhs}}$$

Fig. 5: Small-step semantics for sterm

All of the function definitions we have tried (§8) satisfied pattern compatibility (after automatic renaming of pattern variables). As a last resort, the user can manually instantiate function equations. Although this will always lead to a pattern compatible definition, it is not done automatically, due to the potential blow-up.

Discussion Because this compilation phase is both non-trivial and has some minor restrictions on the set of function definitions that can be processed, we may provide an alternative implementation in the future. Instead of eliminating patterns from right to left, patterns may be grouped in tuples. The above example would be translated into:

$$\begin{aligned} \mathsf{map} = \lambda\ &(f, []) \Rightarrow [] \\ \mid\ &(f, x \mathbin{\#} xs) \Rightarrow f\ x \mathbin{\#} \mathsf{map}\ f\ xs \end{aligned}$$

We would then leave the compilation of patterns for the CakeML compiler, which has no pattern compatibility restriction.

The obvious disadvantage however is that this would require the knowledge of a tuple type in the term language which is otherwise unaware of concrete datatypes.

5.4 Sequentialization: From pterm to sterm

The semantics of pterm and sterm differ only in rule Step and Beta. Figure 5 shows the modified rules, where instead of any matching clause the first matching clause in a case abstraction is picked. For the correctness proof, the order of clauses does not matter: we only need to prove that a step taken in the sequential semantics can be reproduced in the unordered semantics. As long as no rules are dropped, this is trivially true. For that reason, the compiler orders the clauses lexicographically. At the same time the rules are also converted from type (string × pterm) set to (string × sterm) list. Below, *rs* will always denote a list of the latter type.

5.5 Big-step semantics for sterm

This big-step semantics for sterm is not a compiler phase but moves towards the desired evaluation semantics. In this first step, we reuse the sterm type for

$$\textsc{Const}\ \frac{(name, rhs) \in rs}{rs, \sigma \vdash \mathsf{Sconst}\ name \downarrow rhs} \qquad \textsc{Var}\ \frac{\sigma\ name = \mathsf{Some}\ v}{rs, \sigma \vdash \mathsf{Svar}\ name \downarrow v}$$

$$\textsc{Abs}\ \frac{}{rs, \sigma \vdash \Lambda\ cs \downarrow \Lambda\,[(pat, \mathsf{subst}\ (\sigma - \mathsf{frees}\ pat)\ t \mid (pat, t) \leftarrow cs]}$$

$$\textsc{Comb}\ \frac{\begin{array}{c} rs, \sigma \vdash t \downarrow \Lambda\ cs \\ rs, \sigma \vdash u \downarrow u' \qquad \mathsf{first\_match}\ cs\ u' = \mathsf{Some}\ (\sigma', rhs) \qquad rs, \sigma +\!\!+\, \sigma' \vdash rhs \downarrow v \end{array}}{rs, \sigma \vdash t\ \$\ u \downarrow v}$$

$$\textsc{Constr}\ \frac{name \in constructors \qquad rs, \sigma \vdash t_1 \downarrow u_1 \qquad \cdots \qquad rs, \sigma \vdash t_n \downarrow u_n}{rs, \sigma \vdash \mathsf{Sconst}\ name\ \$\ t_1\ \$ \ldots \$\ t_n \downarrow \mathsf{Sconst}\ name\ \$\ u_1\ \$ \ldots \$\ u_n}$$

Fig. 6: Big-step semantics for sterm

evaluation results, instead of evaluating to the separate type value. This allows us to ignore environment capture in closures for now.

All previous $\longrightarrow$ relations were parametrized by a rule set. Now the big-step predicate is of the form $rs, \sigma \vdash t \downarrow t'$ where $\sigma :: \mathsf{string} \rightharpoonup \mathsf{sterm}$ is a variable environment.

This semantics also introduces the distinction between *constructors* and *defined constants.* If C is a constructor, the term $\langle \mathsf{C}\ t_1\ \ldots\ t_n \rangle$ is evaluated to $\langle \mathsf{C}\ t'_1\ \ldots\ t'_n \rangle$ where the t'_i are the results of evaluating the t_i.

The full set of rules is shown in Figure 6. They deserve a short explanation:

Const Constants are retrieved from the rule set *rs*.
Var Variables are retrieved from the environment σ.
Abs In order to achieve the intended invariant, abstractions are evaluated to their fully substituted form.
Comb Function application $t\,\$\,u$ first requires evaluation of t into an abstraction $\Lambda\ cs$ and evaluation of u into an arbitrary term u'. Afterwards, we look for a clause matching u' in *cs*, which produces a local variable environment σ', possibly overwriting existing variables in σ. Finally, we evaluate the right-hand side of the clause with the combined global and local variable environment.
Constr For a constructor application $\langle \mathsf{C}\ t_1\ \ldots \rangle$, evaluate all t_i. The set *constructors* is an implicit parameter of the semantics.

Lemma 1 (Closedness invariant). *If σ contains only closed terms,* $\mathsf{frees}\ t \subseteq \mathsf{dom}\ \sigma$ *and $rs, \sigma \vdash t \downarrow t'$, then t' is closed.*

Correctness of the big-step w.r.t. the small-step semantics is proved easily by induction on the former:

Lemma 2. *For any closed environment σ satisfying* $\mathsf{frees}\ t \subseteq \mathsf{dom}\ \sigma$,

$$rs, \sigma \vdash t \downarrow u \to rs \vdash \mathsf{subst}\ \sigma\ t \longrightarrow^* u$$

By setting $\sigma = []$, we obtain:

Theorem 2 (Correctness). $rs, [] \vdash t \downarrow u \land \mathsf{closed}\ t \to rs \vdash t \longrightarrow^* u$

$$\textsc{Const}\ \frac{(\mathit{name}, \mathit{rhs}) \in \mathit{rs}}{\mathit{rs}, \sigma \vdash \mathsf{Sconst}\ \mathit{name} \downarrow \mathit{rhs}} \qquad \textsc{Var}\ \frac{\sigma\ \mathit{name} = \mathsf{Some}\ v}{\mathit{rs}, \sigma \vdash \mathsf{Svar}\ \mathit{name} \downarrow v}$$

$$\textsc{Abs}\ \frac{}{\mathit{rs}, \sigma \vdash \Lambda\ \mathit{cs} \downarrow \mathsf{Vabs}\ \mathit{cs}\ \sigma}$$

$$\textsc{Comb}\ \frac{\begin{array}{c} \mathit{rs}, \sigma \vdash t \downarrow \mathsf{Vabs}\ \mathit{cs}\ \sigma' \\ \mathit{rs}, \sigma \vdash u \downarrow v \qquad \mathsf{first\_match}\ \mathit{cs}\ v = \mathsf{Some}\ (\sigma'', \mathit{rhs}) \qquad \mathit{rs}, \sigma' +\!\!+\ \sigma'' \vdash \mathit{rhs} \downarrow v' \end{array}}{\mathit{rs}, \sigma \vdash t\ \$\ u \downarrow v'}$$

$$\textsc{RecComb}\ \frac{\begin{array}{c} \mathit{rs}, \sigma \vdash t \downarrow \mathsf{Vrecabs}\ \mathit{css}\ \mathit{name}\ \sigma' \qquad \mathit{css}\ \mathit{name} = \mathsf{Some}\ \mathit{cs} \qquad \mathit{rs}, \sigma \vdash u \downarrow v \\ \mathsf{first\_match}\ \mathit{cs}\ v = \mathsf{Some}\ (\sigma'', \mathit{rhs}) \qquad \mathit{rs}, \sigma' +\!\!+\ \sigma'' \vdash \mathit{rhs} \downarrow v' \end{array}}{\mathit{rs}, \sigma \vdash t\ \$\ u \downarrow v'}$$

$$\textsc{Constr}\ \frac{\mathit{name} \in \mathit{constructors} \qquad \mathit{rs}, \sigma \vdash t_1 \downarrow v_1 \qquad \cdots \qquad \mathit{rs}, \sigma \vdash t_n \downarrow v_n}{\mathit{rs}, \sigma \vdash \mathsf{Sconst}\ \mathit{name}\ \$\ t_1\ \$ \ldots \$\ t_n \downarrow \mathsf{Vconstr}\ \mathit{name}\ [v_1, \ldots, v_n]}$$

Fig. 7: Evaluation semantics from sterm to value

5.6 Evaluation semantics: Refining sterm to value

At this point, we introduce the concept of values into the semantics, while still keeping the rule set (for constants) and the environment (for variables) separate. The evaluation rules are specified in Figure 7 and represent a departure from the original rewriting semantics: a term does not evaluate to another term but to an object of a different type, a value. We still use $\downarrow$ as notation, because big-step and evaluation semantics can be disambiguated by their types.

The evaluation model itself is fairly straightforward. As explained in §4.5, abstraction terms are evaluated to closures capturing the current variable environment. Note that at this point, recursive closures are not treated differently from non-recursive closures. In a later stage, when rs and σ are merged, this distinction becomes relevant.

We will now explain each rule that has changed from the previous semantics:

Abs Abstraction terms are evaluated to a closure capturing the current environment.

Comb As before, in an application $t\,\$\,u$, t must evaluate to a closure $\mathsf{Vabs}\ cs\ \sigma'$. The evaluation result of u is then matched against the clauses cs, producing an environment σ''. The right-hand side of the clause is then evaluated using $\sigma' +\!\!+\ \sigma''$; the original environment σ is effectively discarded.

RecComb Similar as above. Finding the matching clause is a two-step process: First, the appropriate clause list is selected by name of the currently active function. Then, matching is performed.

Constr As before, for an n-ary application $\langle \mathsf{C}\ t_1\ \ldots \rangle$, where C is a data constructor, we evaluate all t_i. The result is a Vconstr value.

Conversion between sterm and value To establish a correspondence between evaluating a term to an sterm and to a value, we apply the same trick as in §5.2.

Instead of specifying a complicated relation, we translate value back to sterm: simply apply the substitutions in the captured environments to the clauses.

The translation rules for Vabs and $\mathsf{Vrecabs}$ are kept similar to the Abs rule from the big-step semantics (Figure 6). Roughly speaking, the big-step semantics always keeps terms fully substituted, whereas the evaluation semantics defers substitution.

Similarly to §5.2, we can also define a function $\mathsf{sterm\_to\_value} :: \mathsf{sterm} \Rightarrow \mathsf{value}$ and prove that one function is the inverse of the other.

Matching The value type, instead of using binary function application as all other term types, uses n-ary constructor application. This introduces a conceptual mismatch between (binary) patterns and values. To make the proofs easier, we introduce an intermediate type of n-ary patterns. This intermediate type can be optimized away by fusion.

Correctness The correctness proof requires a number of interesting lemmas.

Lemma 3 (Substitution before evaluation). *Assuming that a term t can be evaluated to a value u given a closed environment σ, it can be evaluated to the same value after substitution with a sub-environment σ'. Formally: $rs, \sigma \vdash t \downarrow u \wedge \sigma' \subseteq \sigma \to rs, \sigma \vdash \mathsf{subst}\ \sigma'\ t \downarrow u$*

This justifies the "pre-substitution" exhibited by the Abs rule in the big-step semantics in contrast to the environment-capturing Abs rule in the evaluation semantics.

Theorem 3 (Correctness). *Let σ be a closed environment and t a term which only contains free variables in $\mathsf{dom}\ \sigma$. Then, an evaluation to a value $rs, \sigma \vdash t \downarrow v$ can be reproduced in the big-step semantics as $rs', \mathsf{map}\ \mathsf{value\_to\_sterm}\ \sigma \vdash t \downarrow \mathsf{value\_to\_sterm}\ v$, where $rs' = [(name, \mathsf{value\_to\_sterm}\ rhs) \mid (name, rhs) \leftarrow rs]$.*

Instantiating the correctness theorem The correctness theorem states that, for any given evaluation of a term t with a given environment rs, σ containing values, we can reproduce that evaluation in the big-step semantics using a derived list of rules rs' and an environment σ' containing sterms that are generated by the $\mathsf{value\_to\_sterm}$ function. But recall the diagram in Figure 2. In our scenario, we start with a given rule set of sterms (that has been compiled from a rule set of terms). Hence, the correctness theorem only deals with the opposite direction.

It remains to construct a suitable rs such that applying $\mathsf{value\_to\_sterm}$ to it yields the given sterm rule set. We can exploit the side condition (§5.1) that all bindings define functions, not constants:

Definition 6 (Global clause set). *The mapping $\mathsf{global\_css} :: \mathsf{string} \rightharpoonup ((\mathsf{term} \times \mathsf{sterm})\ \mathsf{list})$ is obtained by stripping the Sabs constructors from all definitions and converting the resulting list to a mapping.*

For each definition with name f we define a corresponding term $v_f = \mathsf{Vrecabs}\ \mathsf{global\_css}\ f\ []$. In other words, each function is now represented by a recursive closure bundling all functions. Applying value_to_sterm to v_f returns the original definition of f. Let rs denote the original sterm rule set and rs_v the environment mapping all f's to the v_f's.

The variable environments σ and σ' can safely be set to the empty mapping, because top-level terms are evaluated without any free variable bindings.

Corollary 1 (Correctness). $rs_v, [] \vdash t \downarrow v \to rs, [] \vdash t \downarrow \mathsf{value\_to\_sterm}\ v$

Note that this step was not part of the compiler (although rs_v is computable) but it is a refinement of the semantics to support a more modular correctness proof.

Example Recall the odd and even example from §4.5. After compilation to sterm, the rule set looks like this:

$$\begin{aligned} rs = \{&(\texttt{"odd"}, \mathsf{Sabs}\ [\langle 0\rangle \Rightarrow \langle \mathsf{False}\rangle, \langle \mathsf{Suc}\ n\rangle \Rightarrow \langle \mathsf{even}\ n\rangle]),\\ &(\texttt{"even"}, \mathsf{Sabs}\ [\langle 0\rangle \Rightarrow \langle \mathsf{True}\rangle, \langle \mathsf{Suc}\ n\rangle \Rightarrow \langle \mathsf{odd}\ n\rangle])\} \end{aligned}$$

This can be easily transformed into the following global clause set:

$$\begin{aligned} \mathsf{global\_css} = [&\texttt{"odd"} \mapsto [\langle 0\rangle \Rightarrow \langle \mathsf{False}\rangle, \langle \mathsf{Suc}\ n\rangle \Rightarrow \langle \mathsf{even}\ n\rangle],\\ &\texttt{"even"} \mapsto [\langle 0\rangle \Rightarrow \langle \mathsf{True}\rangle, \langle \mathsf{Suc}\ n\rangle \Rightarrow \langle \mathsf{odd}\ n\rangle]] \end{aligned}$$

Finally, rs_v is computed by creating a recursive closure for each function:

$$\begin{aligned} rs_\mathrm{v} = [&\texttt{"odd"} \mapsto \mathsf{Vrecabs}\ \mathsf{global\_css}\ \texttt{"odd"}\ [],\\ &\texttt{"even"} \mapsto \mathsf{Vrecabs}\ \mathsf{global\_css}\ \texttt{"even"}\ []] \end{aligned}$$

5.7 Evaluation with recursive closures

CakeML distinguishes between non-recursive and recursive closures [29]. This distinction is also present in the value type. In this step, we will conflate variables with constants which necessitates a special treatment of recursive closures. Therefore we introduce a new predicate $\sigma \vdash t \downarrow v$ in Figure 8 (in contrast to the previous $rs, \sigma \vdash t \downarrow v$). We examine the rules one by one:

Const/Var Constant definition and variable values are both retrieved from the same environment σ. We have opted to keep the distinction between constants and variables in the sterm type to avoid the introduction of another term type.

Abs Identical to the previous evaluation semantics. Note that evaluation never creates recursive closures at run-time (only at compile-time, see §5.6). Anonymous functions, e.g. in the term $\langle \mathsf{map}\ (\lambda x.\ x)\rangle$, are evaluated to non-recursive closures.

$$\textsc{Const}\ \frac{\mathit{name} \notin \mathit{constructors} \qquad \sigma\ \mathit{name} = \mathsf{Some}\ v}{\sigma \vdash \mathsf{Sconst}\ \mathit{name} \downarrow v}$$

$$\textsc{Var}\ \frac{\sigma\ \mathit{name} = \mathsf{Some}\ v}{\sigma \vdash \mathsf{Svar}\ \mathit{name} \downarrow v} \qquad \textsc{Abs}\ \frac{}{\sigma \vdash \Lambda\ \mathit{cs} \downarrow \mathsf{Vabs}\ \mathit{cs}\ \sigma}$$

$$\textsc{Comb}\ \frac{\begin{array}{c}\sigma \vdash t \downarrow \mathsf{Vabs}\ \mathit{cs}\ \sigma' \\ \sigma \vdash u \downarrow v \qquad \mathsf{first\_match}\ \mathit{cs}\ v = \mathsf{Some}\ (\sigma'', \mathit{rhs}) \qquad \sigma' +\!\!+\ \sigma'' \vdash \mathit{rhs} \downarrow v'\end{array}}{\sigma \vdash t\ \$\ u \downarrow v'}$$

$$\textsc{RecComb}\ \frac{\begin{array}{c}\sigma \vdash t \downarrow \mathsf{Vrecabs}\ \mathit{css}\ \mathit{name}\ \sigma' \\ \mathit{css}\ \mathit{name} = \mathsf{Some}\ \mathit{cs} \qquad \sigma \vdash u \downarrow v \qquad \mathsf{first\_match}\ \mathit{cs}\ v = \mathsf{Some}\ (\sigma'', \mathit{rhs}) \\ \sigma' +\!\!+\ \mathsf{mk\_rec\_env}\ \mathit{css}\ \sigma' +\!\!+\ \sigma'' \vdash \mathit{rhs} \downarrow v'\end{array}}{\sigma \vdash t\ \$\ u \downarrow v'}$$

$$\textsc{Constr}\ \frac{\mathit{name} \in \mathit{constructors} \qquad \sigma \vdash t_1 \downarrow v_1 \qquad \cdots \qquad \sigma \vdash t_n \downarrow v_n}{\sigma \vdash \mathsf{Sconst}\ \mathit{name}\ \$\ t_1\ \$ \ldots \$\ t_n \downarrow \mathsf{Vconstr}\ \mathit{name}\ [v_1, \ldots, v_n]}$$

Fig. 8: ML-style evaluation semantics

Comb Identical to the previous evaluation semantics.

RecComb Almost identical to the evaluation semantics. Additionally, for each function $(\mathit{name}, \mathit{cs}) \in \mathit{css}$, a new recursive closure $\mathsf{Vrecabs}\ \mathit{css}\ \mathit{name}\ \sigma'$ is created and inserted into the environment. This ensures that after the first call to a recursive function, the function itself is present in the environment to be called recursively, without having to introduce coinductive environments.

Constr Identical to the evaluation semantics.

Conflating constants and variables By merging the rule set *rs* with the variable environment σ, it becomes necessary to discuss possible clashes. Previously, the syntactic distinction between Svar and Sconst meant that $\langle x \rangle$ and $\langle \mathsf{x} \rangle$ are not ambiguous: all semantics up to the evaluation semantics clearly specify where to look for the substitute. This is not the case in functional languages where functions and variables are not distinguished syntactically.

Instead, we rely on the fact that the initial rule set only defines constants. All variables are introduced by matching before β-reduction (that is, in the Comb and RecComb rules). The Abs rule does not change the environment. Hence it suffices to assume that variables in patterns must not overlap with constant names (see §5.1).

Correspondence relation Both constant definitions and values of variables are recorded in a single environment σ. This also applies to the environment contained in a closure. The correspondence relation thus needs to take a different sets of bindings in closures into account.

Hence, we define a relation $\approx_{\mathrm{v}}$ that is implicitly parametrized on the rule set *rs* and compares environments. We call it *right-conflating*, because in a correspondence $v \approx_{\text{v}} u$, any bound environment in u is thought to contain both variables and constants, whereas in v, any bound environment contains only variables.

Definition 7 (Right-conflating correspondence). *We define $\approx_v$ coinductively as follows:*

$$\frac{v_1 \approx_{\text{v}} u_1 \qquad \cdots \qquad v_n \approx_{\text{v}} u_n}{\mathsf{Vconstr}\ name\ [v_1, \ldots, v_n] \approx_{\text{v}} \mathsf{Vconstr}\ name\ [u_1, \ldots, u_n]}$$

$$\frac{\forall x \in \mathsf{frees}\ cs.\ \sigma_1\ x \approx_{\text{v}} \sigma_2\ x \qquad \forall x \in \mathsf{consts}\ cs.\ rs\ x \approx_{\text{v}} \sigma_2\ x}{\mathsf{Vabs}\ cs\ \sigma_1 \approx_{\text{v}} \mathsf{Vabs}\ cs\ \sigma_2}$$

$$\frac{\begin{array}{c}\forall cs \in \mathsf{range}\ css.\ \forall x \in \mathsf{frees}\ cs.\ \sigma_1\ x \approx_{\text{v}} \sigma_2\ x \\ \forall cs \in \mathsf{range}\ css.\ \forall x \in \mathsf{consts}\ cs.\ \sigma_1\ x \approx_{\text{v}} (\sigma_2 +\!\!+\ \mathsf{mk\_rec\_env}\ css\ \sigma_2)\ x\end{array}}{\mathsf{Vrecabs}\ css\ name\ \sigma_1 \approx_{\text{v}} \mathsf{Vrecabs}\ css\ name\ \sigma_2}$$

Consequently, $\approx_{\text{v}}$ is not reflexive.

Correctness The correctness lemma is straightforward to state:

Theorem 4 (Correctness). *Let σ be an environment, t be a closed term and v a value such that $\sigma \vdash t \downarrow v$. If for all constants x occurring in t, $rs\ x \approx_{\text{v}} \sigma\ x$ holds, then there is an u such that $rs, [] \vdash t \downarrow u$ and $u \approx_{\text{v}} v$.*

As usual, the rather technical proof proceeds via induction over the semantics (Figure 8). It is important to note that the global clause set construction (§5.6) satisfies the preconditions of this theorem:

Lemma 4. *If name is the name of a constant in rs, then*

$$\mathsf{Vrecabs}\ \mathsf{global\_css}\ name\ [] \approx_{\text{v}} \mathsf{Vrecabs}\ \mathsf{global\_css}\ name\ []$$

Because $\approx_{\text{v}}$ is defined coinductively, the proof of this precondition proceeds by coinduction.

5.8 CakeML

CakeML is a verified implementation of a subset of Standard ML [23, 39]. It comprises a parser, type checker, formal semantics and backend for machine code. The semantics has been formalized in Lem [28], which allows export to Isabelle theories.

Our compiler targets CakeML's abstract syntax tree. However, we do not make use of certain CakeML features; notably mutable cells, modules, and literals. We have derived a smaller, executable version of the original CakeML semantics, called *CupCakeML*, together with an equivalence proof. The correctness proof of the last compiler phase establishes a correspondence between CupCakeML and the final semantics of our compiler pipeline.

For the correctness proof of the CakeML compiler, its authors have extracted the Lem specification into HOL4 theories [1]. In our work, we directly target CakeML abstract syntax trees (thereby bypassing the parser) and use its big-step semantics, which we have extracted into Isabelle.[1]

Conversion from sterm to exp After the series of translations described in the earlier sections, our terms are syntactically close to CakeML's terms (Cake.exp). The only remaining differences are outlined below:

- CakeML does not combine abstraction and pattern matching. For that reason, we have to translate $\Lambda\ [p_1 \Rightarrow t_1, \ldots]$ into $\Lambda x.$ **case** x **of** $p_1 \Rightarrow t_1 \mid \ldots$, where x is a fresh variable name. We reuse the fresh monad to obtain a bound variable name. Note that it is not necessary to thread through already created variable names, only existing names. The reason is simple: a generated variable is bound and then immediately used in the body. Shadowing it somewhere in the body is not problematic.
- CakeML has two distinct syntactic categories for identifiers (that can represent variables or functions) and data constructors. Our term types however have two distinct syntactic categories for constants (that can represent functions or data constructors) and variables. The necessary prerequisites to deal with this are already present in the ML-style evaluation semantics (§5.7) which conflates constants and variables, but has a dedicated Constr rule for data constructors.

Types During embedding (§3), all type information is erased. Yet, CakeML performs some limited form of type checking at run-time: constructing and matching data must always be fully applied. That is, data constructors must always occur with all arguments supplied on right-hand and left-hand sides.

Fully applied constructors in terms can be easily guaranteed by simple preprocessing. For patterns however, this must be ensured throughout the compilation pipeline; it is (like other syntactic constraints) another side condition imposed on the rule set (§5.1).

The shape of datatypes and constructors is managed in CakeML's environment. This particular piece of information is allowed to vary in closures, since ML supports local type definitions. Tracking this would greatly complicate our proofs. Hence, we fix a global set of constructors and enforce that all values use exactly that one.

Correspondence relation We define two different correspondence relations: One for values and one for expressions.

Definition 8 (Expression correspondence).

$$\textsc{Var}\ \frac{}{\mathsf{rel\_e}\ (\mathsf{Svar}\ n)\ (\mathsf{Cake.Var}\ n)} \qquad \textsc{Const}\ \frac{n \notin \mathit{constructors}}{\mathsf{rel\_e}\ (\mathsf{Sconst}\ n)\ (\mathsf{Cake.Var}\ n)}$$

[1] based on a repository snapshot from March 27, 2017 (0c48672)

$$\textsc{Constr}\ \frac{n \in \mathit{constructors} \qquad \mathsf{rel\_e}\ t_1\ u_1 \qquad \cdots}{\mathsf{rel\_e}\ (\mathsf{Sconst}\ \mathit{name}\ \$\ t_1\ \$\ \ldots\ \$\ t_n)\ (\mathsf{Cake.Con}\ (\mathsf{Some}\ (\mathsf{Cake.Short}\ \mathit{name})\ [u_1, \ldots, u_n]))}$$

$$\textsc{App}\ \frac{\mathsf{rel\_e}\ t_1\ u_1 \qquad \mathsf{rel\_e}\ t_2\ u_2}{\mathsf{rel\_e}\ t_1\ \$\ t_2\ \mathsf{Cake.App}\ \mathsf{Cake.Opapp}\ [u_1, u_2]}$$

$$\textsc{Fun}\ \frac{\begin{array}{c} n \notin \mathsf{ids}\ (\Lambda\ [p_1 \Rightarrow t_1, \ldots]) \cup \mathit{constructors} \\ q_1 = \mathsf{mk\_ml\_pat}\ p_1 \qquad \mathsf{rel\_e}\ t_1\ u_1 \qquad \cdots \end{array}}{\mathsf{rel\_e}\ (\Lambda\ [p_1 \Rightarrow t_1, \ldots])\ (\mathsf{Cake.Fun}\ n\ (\mathsf{Cake.Mat}\ (\mathsf{Cake.Var}\ n))\ [q_1 \Rightarrow u_1, \ldots])}$$

$$\textsc{Mat}\ \frac{\mathsf{rel\_e}\ t\ u \qquad q_1 = \mathsf{mk\_ml\_pat}\ p_1 \qquad \mathsf{rel\_e}\ t_1\ u_1 \qquad \cdots}{\mathsf{rel\_e}\ (\Lambda\ [p_1 \Rightarrow t_1, \ldots]\ \$\ t)\ (\mathsf{Cake.Mat}\ u\ [q_1 \Rightarrow u_1, \ldots])}$$

We will explain each of the rules briefly here.

Var Variables are directly related by identical name.

Const As described earlier, constructors are treated specially in CakeML. In order to not confuse functions or variables with data constructors themselves, we require that the constant name is not a constructor.

Constr Constructors are directly related by identical name, and recursively related arguments.

App CakeML does not just support general function application but also unary and binary operators. In fact, function application is the binary operator Opapp. We never generate other operators. Hence the correspondence is restricted to Opapp.

Fun/Mat Observe the symmetry between these two cases: In our term language, matching and abstraction are combined, which is not the case in CakeML. This means we relate a case abstraction to a CakeML function containing a match, and a case abstraction applied to a value to just a CakeML match.

There is no separate relation for patterns, because their translation is simple.

The value correspondence (rel_v) is structurally simpler. In the case of constructor values (Vconstr and Cake.Conv), arguments are compared recursively. Closures and recursive closures are compared extensionally, i.e. only bindings that occur in the body are checked recursively for correspondence.

Correctness We use the same trick as in §5.6 to obtain a suitable environment for CakeML evaluation based on the rule set *rs*.

Theorem 5 (Correctness). *If the compiled expression* sterm_to_cake t *terminates with a value* u *in the CakeML semantics, there is a value* v *such that* rel_v $v\ u$ *and* $rs \vdash t \downarrow v$.

6 Composition

The complete compiler pipeline consists of multiple phases. Correctness is justified for each phase between intermediate semantics and correspondence relations, most of which are rather technical. Whereas the compiler may be complex and impenetrable, the trustworthiness of the constructions hinges on the obviousness of those correspondence relations.

Fortunately, under the assumption that terms to be evaluated and the resulting values do not contain abstractions – or closures, respectively – all of the correspondence relations collapse to simple structural equality: two terms are related if and only if one can be converted to the other by consistent renaming of term constructors.

The actual compiler can be characterized with two functions. Firstly, the translation of term to Cake.exp is a simple composition of each term translation function:

```
definition term_to_cake :: term ⇒ Cake.exp where
term_to_cake = sterm_to_cake ∘ pterm_to_sterm ∘ nterm_to_pterm ∘ term_to_nterm
```

Secondly, the function that translates function definitions by composing the phases as outlined in Figure 2, including iterated application of pattern elimination:

```
definition compile :: (term × term) fset ⇒ Cake.dec where
compile = Cake.Dletrec ∘ compile_srules_to_cake ∘ compile_prules_to_srules ∘
  compile_irules_to_srules ∘ compile_irules_iter ∘ compile_crules_to_irules ∘
  consts_of ∘ compile_rules_to_nrules
```

Each function compile_* corresponds to one compiler phase; the remaining functions are trivial. This produces a CakeML top-level declaration. We prove that evaluating this declaration in the top-level semantics (evaluate_prog) results in an environment cake_sem_env. But cake_sem_env can also be computed via another instance of the global clause set trick (§5.6).

Equipped with these functions, we can state the final correctness theorem:

```
theorem compiled_correct:
  (∗ If CakeML evaluation of a term succeeds ... ∗)
  assumes evaluate False cake_sem_env s (term_to_cake t) (s', Rval ml_v)
  (∗ ... producing a constructor term without closures ... ∗)
  assumes cake_abstraction_free ml_v
  (∗ ... and some syntactic properties of the involved terms hold ... ∗)
  assumes closed t and ¬ shadows_consts (heads rs ∪ constructors) t and
    welldefined (heads rs ∪ constructors) t and wellformed t
  (∗ ... then this evaluation can be reproduced in the term−rewriting semantics ∗)
  shows rs ⊢ t →* cake_to_term ml_v
```

This theorem directly relates the evaluation of a term t in the full CakeML (including mutability and exceptions) to the evaluation in the initial higher-order term rewriting semantics. The evaluation of t happens using the environment produced from the initial rule set. Hence, the theorem can be interpreted as the correctness of the pseudo-ML expression **let rec** *rs* **in** t.

```
class add =
  fixes plus :: 'a ⇒ 'a ⇒ 'a

definition f :: ('a::add) ⇒ 'a where
f x = plus x x
```

(a) Source program

```
datatype 'a dict_add = Dict_add ('a ⇒ 'a ⇒ 'a)

fun cert_add :: ('a::add) dict_add ⇒ bool where
cert_add (Dict_add pls) = (pls = plus)

fun f' :: 'a dict_add ⇒ 'a ⇒ 'a where
f' (Dict_add pls) x = pls x x

lemma f'_eq: cert_add dict → f' dict = f
<proof>
```

(b) Result of translation

Fig. 9: Dictionary construction in Isabelle

Observe that in the assumption, the conversion goes from our terms to CakeML expressions, whereas in the conclusion, the conversion goes the opposite direction.

7 Dictionary construction

Isabelle's type system supports *type classes* (or simply *classes*) [17, 43] whereas CakeML does not. In order to not complicate the correctness proofs, type classes are not supported by our embedded term language either. Instead, we eliminate classes and instances by a dictionary construction [18] before embedding into the term language. Haftmann and Nipkow give a pen-and-paper correctness proof of this construction [16, §4.1]. We augmented the dictionary construction with the generation of a certificate theorem that shows the equivalence of the two versions of a function, with type classes and with dictionaries. This section briefly explains our dictionary construction.

Figure 9 shows a simple example of a dictionary construction. Type variables may carry *class constraints* (e.g. α :: add). The basic idea is that classes become *dictionaries* containing the functions of that class; class instances become dictionary definitions. Dictionaries are realized as datatypes. Class constraints become additional dictionary parameters for that class. In the example, class add becomes dict_add; function f is translated into f' which takes an additional parameter of type dict_add. In reality our tool does not produce the Isabelle source code shown in Figure 9b but performs the constructions internally. The correctness lemma f′_eq is proved automatically. Its precondition expresses that the dictionary must contain exactly the function(s) of class add. For any monomorphic instance, the precondition can be proved outright based on the certificate theorems proved for each class instance as explained next.

Not shown in the example is the translation of class instances. The basic form of a class instance in Isabelle is $\tau :: (c_1, \ldots, c_n)\ c$ where τ is an n-ary type constructor. It corresponds to Haskell's $(c_1\ \alpha_1, \ldots, c_n\ \alpha_n) \Rightarrow c\ (\tau\ \alpha_1 \ldots \alpha_n)$ and is translated into a function $\mathsf{inst\_}c\mathsf{\_}\tau :: \alpha_1\ \mathsf{dict\_}c_1 \Rightarrow \cdots \Rightarrow \alpha_n\ \mathsf{dict\_}c_n \Rightarrow (\alpha_1, \ldots, \alpha_n)\ \tau\ \mathsf{dict\_}c$ and the following certificate theorem is proved:

$$\mathsf{cert\_}c_1\ \mathit{dict}_1 \rightarrow \cdots \rightarrow \mathsf{cert\_}c_n\ \mathit{dict}_n \rightarrow \mathsf{cert\_}c\ (\mathsf{inst\_}c\mathsf{\_}\tau\ \mathit{dict}_1\ \ldots\ \mathit{dict}_n)$$

For a more detailed explanation of how the dictionary construction works, we refer to the corresponding entry in the Archive of Formal Proofs [20].

8 Evaluation

We have tried out our compiler on examples from existing Isabelle formalizations. This includes an implementation of Huffman encoding, lists and sorting, string functions [38], and various data structures from Okasaki's book [33], including binary search trees, pairing heaps, and leftist heaps. These definitions can be processed with slight modifications: functions need to be totalized (see the end of §3). However, parts of the tactics required for deep embedding proofs (§3) are too slow on some functions and hence still need to be optimized.

9 Conclusion

For this paper we have concentrated on the compiler from Isabelle/HOL to CakeML abstract syntax trees. Partial correctness is proved w.r.t. the big-step semantics of CakeML. In the next step we will link our work with the compiler from CakeML to machine code. Tan *et al.* [39, §10] prove a correctness theorem that relates their semantics with the execution of the compiled machine code. In that paper, they use a newer iteration of the CakeML semantics (functional big-step [34]) than we do here. Both semantics are still present in the CakeML source repository, together with an equivalence proof.

Evaluation of our compiled programs is already possible via Isabelle's predicate compiler [5], which allows us to turn CakeML's big-step semantics into an executable function. We have used this execution mechanism to establish for sample programs that they terminate successfully. We also plan to prove that our compiled programs terminate, i.e. total correctness.

The total size of this formalization, excluding theories extracted from Lem, is currently approximately 20000 lines of proof text (90 %) and ML code (10 %). The ML code itself produces relatively simple theorems, which means that there are less opportunities for it to go wrong. This constitutes an improvement over certifying approaches that prove complicated properties in ML.

References

1. The HOL System Description (2014), `https://hol-theorem-prover.org/`
2. Anand, A., Appel, A.W., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Bélanger, O.S., Sozeau, M., Weaver, M.: CertiCoq: A verified compiler for Coq. In: CoqPL'17: The Third International Workshop on Coq for Programming Languages (2017)

3. Augustsson, L.: Compiling pattern matching. In: Functional Programming Languages and Computer Architecture. pp. 368–381. Springer Berlin Heidelberg (1985)
4. Benton, N., Hur, C.: Biorthogonality, step-indexing and compiler correctness. In: Hutton, G., Tolmach, A.P. (eds.) Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009. pp. 97–108. ACM (2009), `http://doi.acm.org/10.1145/1596550.1596567`
5. Berghofer, S., Bulwahn, L., Haftmann, F.: Turning inductive into equational specifications. In: LNCS. pp. 131–146. Springer (2009)
6. Berghofer, S., Nipkow, T.: Executing higher order logic. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) Types for Proofs and Programs (TYPES 2000). vol. 2277, pp. 24–40 (2002)
7. Blanchette, J.C., Hölzl, J., Lochbihler, A., Panny, L., Popescu, A., Traytel, D.: Truly modular (co)datatypes for Isabelle/HOL. In: Interactive Theorem Proving. pp. 93–110. Springer International Publishing (2014)
8. Boespflug, M., Dénès, M., Grégoire, B.: Full reduction at full throttle. In: Jouannaud, J., Shao, Z. (eds.) Certified Programs and Proofs. Lecture Notes in Computer Science, vol. 7086, pp. 362–377. Springer (2011)
9. Boyer, R.S., Moore, J.S.: Single-threaded objects in ACL2. In: Krishnamurthi, S., Ramakrishnan, C.R. (eds.) Practical Aspects of Declarative Languages (PADL 2002). LNCS, vol. 2257, pp. 9–27. Springer (2002)
10. de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. Indagationes Mathematicae (Proceedings) 75(5), 381 – 392 (1972)
11. Chlipala, A.: A verified compiler for an impure functional language. In: Hermenegildo, M.V., Palsberg, J. (eds.) POPL 2010. pp. 93–106. ACM (2010)
12. Crow, J., Owre, S., Rushby, J., Shankar, N., Stringer-Calvert, D.: Evaluating, testing, and animating PVS specifications. Tech. rep., Computer Science Laboratory, SRI International, Menlo Park, CA (Mar 2001)
13. Flatau, A.D.: A Verified Implementation of an Applicative Language with Dynamic Storage Allocation. Ph.D. thesis, University of Texas at Austin (1992)
14. Forster, Y., Kunze, F.: Verified extraction from coq to a lambda-calculus. In: The 8th Coq Workshop (2016)
15. Greve, D.A., Kaufmann, M., Manolios, P., Moore, J.S., Ray, S., Ruiz-Reina, J., Sumners, R., Vroon, D., Wilding, M.: Efficient execution in an automated reasoning environment. J. Funct. Program. 18(1), 15–46 (2008)
16. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) Functional and Logic Programming: 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings. pp. 103–117. Springer, Berlin, Heidelberg (2010)
17. Haftmann, F., Wenzel, M.: Constructive type classes in Isabelle. In: Altenkirch, T., McBride, C. (eds.) Types for Proofs and Programs: International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers. pp. 160–174. Springer, Berlin, Heidelberg (2007)
18. Hall, C.V., Hammond, K., Jones, S.L.P., Wadler, P.L.: Type classes in haskell. ACM Transactions on Programming Languages and Systems 18(2), 109–138 (mar 1996)
19. Hermida, C., Reddy, U.S., Robinson, E.P.: Logical relations and parametricity – a reynolds programme for category theory and programming languages. Electronic Notes in Theoretical Computer Science 303, 149 – 180 (2014)
20. Hupel, L.: Dictionary construction. Archive of Formal Proofs (May 2017), `http://isa-afp.org/entries/Dict_Construction.html`, Formal proof development

21. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. Journal of Automated Reasoning 44(4), 303–336 (2010)
22. Krauss, A., Schropp, A.: A mechanized translation from higher-order logic to set theory. In: Interactive Theorem Proving. pp. 323–338. Springer (2010)
23. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: A verified implementation of ML. In: POPL. pp. 179–191. POPL '14, ACM (2014)
24. Landin, P.J.: The mechanical evaluation of expressions. The Computer Journal 6(4), 308–320 (jan 1964)
25. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (Jul 2009), `http://doi.acm.org/10.1145/1538788.1538814`
26. Letouzey, P.: A new extraction for coq. In: Geuvers, H., Wiedijk, F. (eds.) Types for Proofs and Programs. LNCS, vol. 2646, pp. 200–219. Springer (2003)
27. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The definition of Standard ML (revised). MIT Press (1997)
28. Mulligan, D.P., Owens, S., Gray, K.E., Ridge, T., Sewell, P.: Lem: Reusable engineering of real-world semantics. In: ICFP. pp. 175–188. ICFP '14, ACM (2014)
29. Myreen, M.O., Owens, S.: Proof-producing translation of higher-order logic into pure and stateful ML. JFP 24(2-3), 284–315 (005 2014)
30. Neis, G., Hur, C.K., Kaiser, J.O., McLaughlin, C., Dreyer, D., Vafeiadis, V.: Pilsner: a compositionally verified compiler for a higher-order imperative language. pp. 166–178. ICFP 2015, ACM, New York, NY, USA (2015)
31. Nipkow, T., Klein, G.: Concrete Semantics. Springer (2014)
32. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283 (2002), 218 pp.
33. Okasaki, C.: Purely functional data structures. Cambridge University Press (1999)
34. Owens, S., Myreen, M.O., Kumar, R., Tan, Y.K.: Functional Big-Step Semantics, pp. 589–615. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
35. Peyton Jones, S.L.: The Implementation of Functional Programming Languages. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1987)
36. Shankar, N.: Static analysis for safe destructive updates in a functional language. In: Pettorossi, A. (ed.) Logic Based Program Synthesis and Transformation (LOPSTR 2001). LNCS, vol. 2372, pp. 1–24. Springer (2001)
37. Slind, K.: Reasoning about Terminating Functional Programs. Ph.D. thesis, Technische Universität München (1999)
38. Sternagel, C., Thiemann, R.: Haskell's show class in isabelle/hol. Archive of Formal Proofs (Jul 2014), `http://isa-afp.org/entries/Show.html`, Formal proof development
39. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A., Owens, S., Norrish, M.: A new verified compiler backend for CakeML. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming - ICFP 2016. Association for Computing Machinery (ACM) (2016)
40. Turner, D.A.: Some history of functional programming languages. In: Lecture Notes in Computer Science, pp. 1–20. Springer Berlin Heidelberg (2013)
41. Urban, C.: Nominal techniques in Isabelle/HOL. Journal of Automated Reasoning 40(4), 327–356 (2008), `http://dx.doi.org/10.1007/s10817-008-9097-2`
42. Urban, C., Berghofer, S., Kaliszyk, C.: Nominal 2. Archive of Formal Proofs (Feb 2013), `http://isa-afp.org/entries/Nominal2.shtml`, Formal proof development
43. Wenzel, M.: Type classes and overloading in higher-order logic. In: Gunter, E.L., Felty, A. (eds.) TPHOLs '97. pp. 307–322. Springer, Berlin, Heidelberg (1997)