A Verified Type System for CakeML

Yong Kiam Tan
IHPC, A*STAR
tanyk@ihpc.a-star.edu.sg

Scott Owens
University of Kent
S.A.Owens@kent.ac.uk

Ramana Kumar
NICTA and UNSW
Ramana.Kumar@nicta.com.au

Abstract

CakeML is a dialect of the (strongly typed) ML family of programming languages, designed to play a central role in high-assurance software systems. To date, the main artefact supporting this is a verified compiler from CakeML source code to x86-64 machine code. The verification effort addresses each phase of compilation from parsing through to code generation and garbage collection.

In this paper, we focus on the type system: its declarative specification, type soundness theorem, and the soundness and completeness of an implementation of type inference – all formally verified in the HOL4 proof assistant. Each of these aspects of a type system is important in any design and implementation of a typed functional programming language. They allow the programmer to soundly employ (informal) type-based reasoning, and the compiler to apply optimisations that assume type-correctness. So naturally, their verification is a critical part of a verified compiler.

1. Context

A formally verified compiler comes with a proven theorem that any observable behaviour of the object code is a permissible behaviour of the source code. Put more colloquially, we know that a verified compiler introduces no bugs. When we are concerned with the proper functioning of a safety- or security-critical software system, using a verified compiler means that we do not have to inspect what the compiler is doing. This can make building an assurance case for the system easier. If we have gone to the extent of formally verifying the system, then we know – without any lower confidence – that the system as actually executed has the properties we verified about it.

Since 2012, the CakeML project (`https://cakeml.org`) has been building a verified compiler for an ML-like programming language. The overall goal is to create an optimising compiler with mechanically checked proofs of an end-to-end correctness theorem. An important additional goal is to verify (again with mechanically checked proofs) that the correctness theorem applies to the code of the compiler that is actually executed, and not just to the compilation algorithm in the abstract. Here 'end-to-end' means that the correctness theorem relates source code, represented as a string, with machine code, represented as a list of bytes. Thus, the verification must address a lexer, a parser, a type checker, a sequence of optimisations and translations between various intermediate languages, a code generator, and a run-time system. A previous paper [4] outlined how all of these phases fit together, and detailed the interactive theorem proving techniques used to verify a non-optimising version of the compiler.

In this paper, we focus on the type checking phase of the compiler. The external interface to the type checker is simple, given (an AST for) a program, it returns a boolean: whether the program obeys CakeML's typing discipline. Its importance stems from the type soundness theorem which guarantees that well-typed programs have well-defined semantics.[1] This is important for any typed programming language: most compilers assume that the input program has well-defined semantics, and make optimisation decisions based on that. It is doubly important in the context of verified compilers: their correctness theorem has a well-definedness pre-condition, and so we cannot use the theorem until we first prove that the source program is well-defined. For some languages, well-definedness is an undecidable property (e.g., C), but for CakeML (and type-safe languages in general), a type inference algorithm can prove that the program obeys the typing discipline. The type soundness theorem then allows us to dispense with the pre-condition and give a compiler correctness theorem that applies to all programs, and characterises which ones the compiler will accept/reject during type checking.

Contributions Our previous work [4] briefly mentioned CakeML's type checker, which at that point had a type soundness theorem, and an inferencer soundness theorem, but not an inferencer completeness theorem. Here, we give a thorough account of these theorems, and additionally present

- a completeness theorem for the inferencer,
- an improved type system that dispenses with the elaboration phase,
- an improved type soundness proof for an operational semantics with more uniform handling of data constructors and modules, and
- support for a few extra language features, especially type abbreviations.

On the proofs All of the theorems in this paper have been mechanically verified in the HOL4 theorem prover and are available from CakeML's code repository (`https://code.cakeml.org`). The type system's definition is in the `semantics` directory, along with the operational semantics; the type soundness proof is in the `semantics/proofs` directory; and everything to do with the inferencer is in the `compiler/inference` directory. Overall the big technical challenges were in properly formulating the theorem statements and invariants, rather than in applying theorem proving technology, and so the former is our focus here.

[1] Equivalently, well-typed programs do not crash, get stuck, go wrong, have undefined behaviour, etc.

$$
\begin{array}{lcl}
op & := & \texttt{div} \mid \texttt{mod} \mid \texttt{+} \mid \texttt{-} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{<>} \mid \texttt{=} \mid \texttt{:=} \mid \texttt{::} \\
 & \mid & \texttt{andalso} \mid \texttt{orelse} \\
id & := & x \mid mn\texttt{.}x \\
cid & := & cn \mid mn\texttt{.}cn \\
t & := & \alpha \mid id \mid t\ id \mid \texttt{(}t\texttt{,}t\texttt{(,)*)}\ id \mid t\ \texttt{*}\ t \mid t\ \texttt{->}\ t \mid \texttt{(}t\texttt{)} \\
l & := & Const \mid \texttt{()} \mid \texttt{[]} \\
p & := & x \mid l \mid cid \mid cid\ p \mid \texttt{\_} \mid \texttt{(}p\texttt{(,}p\texttt{)*)} \mid \texttt{[}p\ \texttt{(,}p\texttt{)*]} \mid p\ \texttt{::}\ p \\
e & := & l \mid id \mid cid \mid cid\ e \mid \texttt{(}e\texttt{,}e\ \texttt{(,}e\texttt{)*)} \mid \texttt{[}e\ \texttt{(,}e\texttt{)*]} \\
 & \mid & \texttt{raise}\ e \mid e\ \texttt{handle}\ p\ \texttt{=>}\ e\ \texttt{(|}\ p\ \texttt{=>}\ e\texttt{)*} \\
 & \mid & \texttt{fn}\ p\ \texttt{=>}\ e \mid e\ e \mid e\ op\ e \mid \texttt{((}e\ \texttt{;)*}\ e\texttt{)} \\
 & \mid & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{case}\ e\ \texttt{of}\ p\ \texttt{=>}\ e\ \texttt{(|}\ p\ \texttt{=>}\ e\texttt{)*} \\
 & \mid & \texttt{let}\ (ld \mid \texttt{;})^*\ \texttt{in}\ (e\ \texttt{;})^*\ e\ \texttt{end} \\
ld & := & \texttt{val}\ x\ \texttt{=}\ e \mid \texttt{fun}\ x\ p^+\ \texttt{=}\ e\ (\texttt{and}\ x\ p^+\ \texttt{=}\ e)^* \\
c & := & cn \mid cn\ \texttt{of}\ t \\
tyd & := & tyn\ \texttt{=}\ c\ (\texttt{|}\ c)^* \\
tyn & := & \texttt{(}\alpha\texttt{(,}\alpha\texttt{)*)}\ x \mid \alpha\ x \mid x \\
d & := & \texttt{val}\ p\ \texttt{=}\ e \mid \texttt{fun}\ x\ p^+\ \texttt{=}\ e\ (\texttt{and}\ x\ p^+\ \texttt{=}\ e)^* \\
 & \mid & \texttt{datatype}\ tyd\ (\texttt{and}\ tyd)^* \mid \texttt{type}\ tyn\ \texttt{=}\ t \\
 & \mid & \texttt{exception}\ c \\
sl & := & \texttt{val}\ x\ \texttt{:}\ t \mid \texttt{type}\ tyn\ (\texttt{=}\ t)^? \mid \texttt{datatype}\ tyd\ (\texttt{and}\ tyd)^* \\
sig & := & \texttt{:>}\ \texttt{sig}\ (sl \mid \texttt{;})^*\ \texttt{end} \\
top & := & \texttt{structure}\ mn\ sig^?\ \texttt{=}\ \texttt{struct}\ (d \mid \texttt{;})^*\ \texttt{end;} \mid d\texttt{;}
\end{array}
$$

where x ranges over identifiers (must not start with a capital letter and must not be an infix operator from op), α over SML-style type variables (*e.g.*, `'a`), cn over constructor names (must start with a capital letter, or be `true`, `false`, `ref`, or `nil`), mn over module names, and $Const$ over integer, string and character constants.

Figure 1. CakeML syntax

Contents We first describe the CakeML language itself (§2), then we give a declarative type system (§3) with a corresponding type inference algorithm (§4). Then we move on to the correctness proofs: soundness and completeness for the inferencer with respect to the type system (§5), and type soundness with respect to an operational semantics via preservation and progress lemmas (§6).

2. The CakeML language

CakeML is a strongly typed, strict, impure functional language that mostly follows the design of Standard ML (SML) [7]. It supports a substantial subset of the core language features including (possibly mutually) recursive datatypes, higher-order functions, pattern matching, references, and exceptions. Notable omissions are records, and local (non-top-level) definitions of datatypes and exceptions. The module system supports non-nested structures and signatures, but not functors. See Fig. 1 for the syntax.

CakeML has a small-step operational semantics based on a CEK machine [3], and it also has a big-step semantics that is proven equivalent. There are a few minor differences in behaviour to SML which are not relevant to us here. The type system does have three relevant simplifications: it does not support equality types, operator overloading, or polymorphic generalisation of let-bound definitions. Top-level and module-top-level definitions can be polymorphic, and we plan to add support for polymorphic let bindings in the future.

3. Type system

CakeML has a declarative type system defined by a syntax-directed, inductively defined relation. Fig. 2 gives the definition of the various environments and the shape of the main judgements.

Typing environments Γ_t are records containing four different sub-environments M, T, C and V in fields called m, t, c, and v respectively. The definition typing judgements build environment fragments that describe the defined things, and we refer to those using m, τ, c and v.

Let $\hat{t}$ be a de Bruijn indexed version of t. We need the following environments, and auxillaries:

$$
\begin{array}{lcll}
tid\_exn & := & id \mid cid & \text{stamps} \\
\hat{V} & := & \epsilon & \text{empty type env.} \\
 & \mid & \mathbb{N}, \hat{V} & \text{bind de Bruijn type variables} \\
 & \mid & x : \langle \mathbb{N}, \hat{t} \rangle, \hat{V} & \text{bind a type scheme}
\end{array}
$$

$$
\begin{array}{ll}
T \equiv id \mapsto (\alpha^* \times t) & \text{type def. env.} \\
M \equiv mn \mapsto (x \times \mathbb{N} \times \hat{t}) & \text{module env.} \\
C \equiv cid \mapsto (\alpha^* \times t^* \times tid\_exn) & \text{constructor env.}
\end{array}
$$

We will use V to refer to the subset of $\hat{V}$ that has no de Bruijn type variable bindings (the second line above). Similarly, $\hat{\Gamma}_t$ are typing environments whose variable environments are of the form $\hat{V}$.
Typing judgements:

$$
\begin{array}{ll}
\mathbb{N},\ C \vdash_p p : \hat{t},\ v & \text{pattern typing} \\
\hat{\Gamma}_t \vdash_e e : \hat{t} & \text{expression typing} \\
u,\ mn^?,\ \delta,\ \Gamma_t \vdash_d d : \delta',\ (\tau, c, v) & \text{definition typing} \\
u,\ \delta,\ \Gamma_t \vdash_t top : \delta',\ (m, \tau, c, v) & \text{top-level typing}
\end{array}
$$

Figure 2. Typing judgements

Type definition environments T support type abbreviations, mapping a type constructor name to the type that it abbreviates, along with a list of type parameters. Since a type definition can appear inside of a module, or at the top-level, the module name mn is optional. Since CakeML does not have nested structures, all identifiers either refer to the current module or a different top-level module, and so use optional module names rather than a list of module names. Constructor environments C similarly record information for all constructors introduced by a `datatype` or `exception` definition. They record the type's parameters, the types of the constructors arguments, which may refer to the parameters, and a stamp tid_exn that records which type the constructor creates, for datatype constructors, or which exception it is, for exception constructors.

Module environments M map modules to the list of value definitions in the module, along with their type schemes. Since these types are represented using de Bruijn indices, a number, rather than a list of type variables, represents how many quantified type variables they have. Finally, variable environments $\hat{V}$, map variables to type schemes, but retain enough structure to record where new type variables are bound. This structural information is only needed at the expression level so we distinguish between $\hat{\Gamma}_t$ used in expressions typing and Γ_t used for everything above that. Because of this separation, M and V are kept as distinct environments rather than using id or cid as the domain of a finite map as in T and C.

Definition environments δ are records containing the set of names of defined modules, the set of identifiers for defined types and the set of identifiers for defined exceptions. The fields are named defined_mods, defined_types and defined_exns respectively. We will see more detail on how stamps and δ work in the discussion of the typing rules in §3.2. Their purpose is to separate the notion of globally unique identity for each module, type, and constructor from the scoping mechanisms of the language. The Definition of Standard ML [7] uses a stamp generation mechanism for this purpose. Instead CakeML restricts top-level modules to having unique names, and the types defined inside a module to have different names from each other. Thus, fully qualified names of types are unique, and can be used for stamps without a gensym mechanism.

$$\text{(PVAR)}\ \frac{\texttt{check\_freevars}\ tvs\ []\ t}{tvs,\ C \vdash_p x : t,\ [(x,t)]}$$

$$\text{(VAR)}\ \frac{\begin{array}{c} tvs = \texttt{LENGTH}\ ts \\ \texttt{EVERY (check\_freevars (num\_tvs}\ \hat{\Gamma}_t\texttt{.v) [])}\ ts \\ \texttt{lookup}\ x\ \hat{\Gamma}_t = \texttt{SOME}\ (tvs,t) \end{array}}{\hat{\Gamma}_t \vdash_e x : \texttt{deBruijn\_subst}\ 0\ ts\ t}$$

$$\text{(FUN)}\ \frac{\begin{array}{c} \texttt{check\_freevars (num\_tvs}\ \hat{\Gamma}_t\texttt{.v) []}\ t_1 \\ \hat{\Gamma}_t, x : (0,t_1) \vdash_e e : t_2 \end{array}}{\hat{\Gamma}_t \vdash_e \texttt{fn}\ x\ \texttt{=>}\ e : t_1\ \texttt{->}\ t_2}$$

$$\text{(LET)}\ \frac{\begin{array}{c} \hat{\Gamma}_t \vdash_e e_1 : t_1 \\ \hat{\Gamma}_t, x : (0,t_1) \vdash_e e_2 : t_2 \end{array}}{\hat{\Gamma}_t \vdash_e \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : t_2}$$

Figure 3. Selected pattern and expression typing rules

3.1 Expressions

The pattern and expression level typing judgements are mostly typical, with the structure of $\hat{V}$ being perhaps the only surprising thing. Fig. 3 gives the rules for pattern and expression variables, functions, and `let` expressions, to show how $\hat{V}$ is used. The PVAR rule allows a pattern variable any type whose free variables are within the given bound tvs (recall that the type here uses a de Bruijn representation). The VAR rule finds the type scheme bound to the identifier in the M or $\hat{V}$ environment, depending on whether the identifier has a module name or not. It can then make an arbitrary instantiation of the tvs quantified type variables as long as their free variables are all bound in $\hat{V}$. The FUN rule binds a type scheme with no quantified type variables[2], and a type whose free variables are also bound in $\hat{V}$. Because the LET rule is monomorphic, it also uses no quantified type variables in the type scheme of x.

3.2 Definitions

Fig. 4 presents the rules for top-level (and module-top-level) definitions. It omits the rule for `fun` definitions, which is similar to the DLET_POLY rule, but specialised to making recursive functions. It also specialises the DTYPE rule to a single recursive datatype, whereas the actual rule in CakeML handles a list of mutually recursive datatypes.

There are two rules for value definitions, since CakeML, with its imperative features, has a value restriction. Both rules ensure that the pattern does not try to bind the same value twice, ensure that the pattern and expression have the same type, and then return the pattern's variable bindings. Neither, defines any new modules, types, or constructors so there are no new identifiers in their conclusions. The DLET_MONO rule does not add any type variable bindings to V when checking e, so that the resulting type will not contain any type variables. The DLET_POLY rule does bind some type variables tvs, and then does a polymorphic generalisation, quantifying the types in the resulting environment v with tvs. The last premise of each rule is used to ensure determinism with the value restriction, which we explain further in §3.3. The u parameter is used to control whether these checks should be carried out; more detail on that in §6.

The rule for type abbreviations (DTABBREV) checks that the abbreviated type is well formed i.e. that the raw type variables it mentions are bound in $targs$. It introduces no new modules, data types, or constructors. It binds the name of the abbreviation in the type definition environment, after first expanding out all of the other abbreviations mentioned in the type. Abbreviations cannot be recursive, because the well-formedness check uses the previous scope that does not contain a binding for this abbreviation yet. It could contain a reference to a *previous* abbreviation with the same name, but that would be expanded to the previous definition. The effect of all of this is that type names for abbreviations are lexically scoped (i.e., uses of a type name refer to the most recent enclosing type abbreviation), and that the type system internally keeps all abbreviations maximally expanded.

The rule for defining a new exception constructor (DEXN) checks that the exception has not already been declared (although an exception with the same name in a different module is allowed), and that its argument types are well-formed. It records that an exception has been declared with its full module path, and binds the exception name in the constructor environment with no type parameters, fully abbreviation-expanded argument types, and the stamp of the exception.

The rule for defining a new datatype (DTYPE) checks that the datatype is not already defined (although a datatype with the same name in a different module is allowed, as are datatypes that have the same name as a type abbreviation in the same module). The constructors must be distinct from each other, but could have the same name as constructors in other datatypes. The argument types to the constructors must be well formed. Since the datatype is allowed to be recursive, the type definition environment is first extended with a binding for the type being defined. This treats the datatype's name as an abbreviation for the stamp that represents the true identity of the type. Lastly, the constructor environment is extended with each constructor mapping to the type parameters, fully abbreviation-expanded argument types, and the type that is being constructed.

3.3 The value restriction and principal types

CakeML has an SML-style value restriction that prohibits the polymorphic generalisation of bindings whose definitions are not syntactic values. Although application of the value restriction is straightforward, subtleties arise when considering the type of a top-level definition at an intermediate program point. To illustrate, consider the following two identity definitions, `id_poly` and `id_mono`.

```
val id_poly = fn x => x;
val id_mono =
  if true then (fn x => x) else (raise Bind);
```

The expression on the right of both definitions can have a variety of types according to the typing relation, `int -> int`, `string -> string`, etc. But only `id_poly` can be given a principal type[3] `'a -> 'a`. The key question is what type should `id_mono` be given when there is no usage of the function to disambiguate. For example, the definition might have been entered at the REPL, or it might be defined, but not used, in a separately compiled module with no explicit signature. For CakeML, such definitions do not type check, so that the CakeML type system maintains the principal type property: every expression with a type has a unique principal type. This is the critical property that supports a complete

[2] We use $\hat{\Gamma}_t, x : (tvs, t)$ to mean $\hat{\Gamma}_t$ with its variable environment field extended with the mapping of x to the type scheme (tvs, t).

[3] Recall that a principal type is one that can be instantiated to obtain any other type that the binding could have.

We need auxiliary functions where

- `is_value` e holds iff e is a literal constant, variable, function, or constructor applied to values,
- `distinct` holds iff its argument list does not contain the same element twice,
- `pat_bindings` p `[]` returns the variables bound by pattern p,
- `bind_tvar` tvs Γ_t records the position where tvs type variables are bound in the variable environment of Γ_t,
- `add_tvs` tvs v quantifies an additional tvs type variables in each type scheme of v,
- `most_gen_env` and `type_pe_determ` are defined in §3.3, and
- `expand_abbrev` T ts expands all of the type abbreviations in ts according to the environment T.

$$\text{(DLET\_POLY)}\ \frac{\begin{array}{c}\texttt{is\_value}\ e \\ \texttt{distinct (pat\_bindings}\ p\ \texttt{[])} \\ tvs,\ \Gamma_t.\ \texttt{c} \vdash_p p : t,\ v \\ \texttt{bind\_tvar}\ tvs\ \Gamma_t \vdash_e e : t \\ u \Rightarrow \texttt{most\_gen\_env}\ \Gamma_t\ p\ e\ tvs\ v\end{array}}{u,\ mn^?,\ \delta,\ \Gamma_t \vdash_d \texttt{val}\ p = e : \delta_\emptyset,\ (\emptyset, \texttt{[]},\ \texttt{add\_tvs}\ tvs\ v)}$$

$$\text{(DLET\_MONO)}\ \frac{\begin{array}{c}\texttt{distinct (pat\_bindings}\ p\ \texttt{[])} \\ 0,\ \Gamma_t.\ \texttt{c} \vdash_p p : t,\ v \\ \Gamma_t \vdash_e e : t \\ u \Rightarrow \neg\texttt{is\_value}\ e \wedge \texttt{type\_pe\_determ}\ \Gamma_t\ p\ e\end{array}}{u,\ mn^?,\ \delta,\ \Gamma_t \vdash_d \texttt{val}\ p = e : \delta_\emptyset,\ (\emptyset, \texttt{[]},\ \texttt{add\_tvs}\ 0\ v)}$$

$$\text{(DTABBREV)}\ \frac{\begin{array}{c}\texttt{check\_freevars}\ 0\ targs\ t \\ \texttt{check\_type\_names}\ \Gamma_t\texttt{.t}\ t \\ \texttt{distinct}\ targs\end{array}}{u,\ mn^?,\ \delta,\ \Gamma_t \vdash_d \texttt{type}\ (targs)\ tn = t : \delta_\emptyset,\ (tn \mapsto (targs, \texttt{expand\_abbrev}\ \Gamma_t\texttt{.t}\ t), \texttt{[]}, \texttt{[]})}$$

$$\text{(DEXN)}\ \frac{\begin{array}{c}\texttt{check\_exn\_tenv}\ mn^?\ cn\ ts \\ \texttt{mk\_id}\ mn^?\ cn \notin \delta.\ \texttt{defined\_exns} \\ \texttt{EVERY (check\_type\_names}\ \Gamma_t\texttt{.t)}\ ts \\ \delta' = \delta_\emptyset\ \texttt{with defined\_exns :=}\ \{\texttt{mk\_id}\ mn^?\ cn\}\end{array}}{u,\ mn^?,\ \delta,\ \Gamma_t \vdash_d \texttt{exception}\ cn\ \textbf{of}\ ts : \delta',\ (\emptyset,\ \texttt{[(}cn\texttt{,[],MAP (expand\_abbrev}\ \Gamma_t\texttt{.t)}\ ts\texttt{, TypeExn (mk\_id}\ mn^?\ cn\texttt{))], []})}$$

$$\text{(DTYPE)}\ \frac{\begin{array}{c}t' = tn \mapsto (tvs,\ \texttt{Tapp (MAP Tvar}\ tvs\texttt{) (TC\_name (mk\_id}\ mn^?\ tn\texttt{)))} \\ merged\_t = \texttt{merge\_mod\_env}\ (\emptyset, t')\ \Gamma_t\texttt{.\ t} \\ \texttt{check\_ctor\_tenv}\ mn^?\ merged\_t\ \texttt{[(}tvs\texttt{,}tn\texttt{,}\ ctors\texttt{)]} \\ new\_tdecls = \texttt{set [mk\_id}\ mn^?\ tn\texttt{]} \\ new\_tdecls \cap \delta.\ \texttt{defined\_types} = \emptyset \\ \delta' = \delta_\emptyset\ \texttt{with defined\_types :=}\ new\_tdecls\end{array}}{u,\ mn^?,\ \delta,\ \Gamma_t \vdash_d \texttt{datatype}\ tvs\ tn = ctors : \delta',\ (t',\ \texttt{build\_ctor\_tenv}\ mn^?\ merged\_t\ \texttt{[(}tvs\texttt{,}tn\texttt{,}ctors\texttt{)]},\ \texttt{[]})}$$

Figure 4. Definition typing rules (`fun` rule omitted, `datatype` rule simplified)

inference algorithm. See Appendix A for a comparison with the decisions made by other ML implementations.

The DLET_MONO rule uses `type_pe_determ` to ensure that ambiguously typed non-values are not type-able. Any bindings v and v' that can arise from the pattern and expression of a `val` definition must be equivalent.

```
type_pe_determ Γt p e ⟺
∀ t1 v t2 v'.
  0, Γt.c ⊢p p : t1, v ∧ Γt ⊢e e : t1 ∧
  0, Γt.c ⊢p p : t2, v' ∧ Γt ⊢e e : t2 ⇒
  v = v'
```

Of course, `id_mono` does not really need the value restriction for type soundness, but the same concerns apply to definitions such as the following two:

```
let val f = ref [] in          val f = ref [];
let val z = 5 :: (!f) in       val z = 5 :: (!f);
  z
end
```

The left expression can be typed as `int list` by assigning `f` the type `int list ref`. For the definitions on the right, we cannot choose to type `f` with `int list ref` without prior knowledge of the subsequent definition. Note also that the value restriction prevents us from assigning `f` a polymorphic type in both cases.

A more exotic problem arises if the typing relation allowed defined values to have types that are not principal. Consider:

```
val f = fn x => ref x;
val z = f [];
```

If we assign `f` its principal type, `'a -> 'a ref`, then, if not for the value restriction, `f []` would have type `'a list ref`. Thus, following the above discussion, the definition of `z` should signal a type error. If we had instead chosen to type `f` with the less general type, `int list -> int list ref`, then `z` could be typed as `int list ref`. However, we cannot choose a less general type for `f` without prior knowledge of the subsequent definition. The DLET_POLY rule uses `most_gen_env` to ensure that the typing relation always gives the most general environments, and that the above definition of `z` is a type error, `weakE` v v' holds iff we can apply substitutions on de Bruijn variables bound in v to obtain v'.

```
most_gen_env Γt p e tvs v ⟺
∀ tvs' v' t'.
  tvs', Γt.c ⊢p p : t', v' ∧ bind_tvar tvs' Γt ⊢e e : t' ⇒
  weakE (add_tvs tvs v) (add_tvs tvs' v')
```

4. Inference algorithm

Our type inference algorithm is based on Milner's Algorithm $\mathcal{W}$ [6], extended to top-level definitions. Internally, the inferencer uses a state-exception monad to track its progress when it performs type inference at the expression level. The monadic state consists of a substitution that maps unification variables to types and a counter that generates fresh unification variables. As in Algorithm $\mathcal{W}$, the substitution is used to backtrack and apply unification constraints as the inferencer walks an expression recursively; using a monad allows us to represent this cleanly in pure higher order logic. Our unification algorithm is based on triangular substitutions and was verified previously [5]; we define encoding and decoding functions to convert between the inferencer types and the generic terms over which the verified unification algorithm operates. Like the type system, we also keep track of a typing environment Γ_i and the defined names δ_i. These environments are similar to their type system counterparts but we can use more efficient representations in the inferencer. To emphasize this difference, we prefix both environment's record fields with `inf_`.

```
infer_e Γi x =
do
  (tvs,t) ← lookup x Γi.inf_v;
  uvs ← n_fresh_uvar tvs;
  return (infer_deBruijn_subst uvs t)
od

infer_e Γi (fn x => e) =
do
  u ← fresh_uvar;
  t ← infer_e (Γi,x : (0,u)) e;
  return (u -> t)
od

infer_e Γi (e1 e2) =
do
  t1 ← infer_e Γi e1;
  t2 ← infer_e Γi e2;
  u ← fresh_uvar;
  add_constraint t1 (t2 -> u);
  return u
od
```

Figure 5. Selected expression inference cases.

4.1 Expressions

Type inference for expressions, `infer_e`, is where we make primary use of unification. Every call to `infer_e` either fails with a type error or succeeds and returns a type. On successful inference, we obtain the inferred type by applying the substitution in the final monad state, *subst*, to the returned type, t. We write this as t`[`*subst*`]` and refer to it as a *solution* of the inferencer.

The important cases, corresponding to variables, functions and applications respectively, are shown in Fig. 5. Various helper functions are used to interact with the monad: `infer_deBruijn_subst` replaces bound de Bruijn variables with fresh unification variables while `add_constraint` applies unification constraints to the current substitution. In the function case, we recursively call `infer_e` on the nested expression e after adding x to the variable environment with a fresh unification variable, u, for its type. This unification variable may get constrained inside the recursive call but it might also be left unconstrained[4]. Unlike in Algorithm $\mathcal{W}$, unconstrained unification variables are handled at the top-level.

4.2 Definitions

At the top-level, our inferencer essentially applies the typing rules directly to type check its input. We focus here on two illustrative cases of the type inferencer for definitions, shown in Fig. 6. The various `guard` expressions are used to check the preconditions of the type system rules. The rest of the inferencer, e.g. top-level module definitions, corresponds closely to the type system and so we do not discuss it further.

The first case in Fig. 6 corresponds to type inference for new variable definitions of the form `val` p `=` e. Starting from an empty substitution in the initial monadic state, we infer a type for e and ensure the typing constraints introduced by the bindings in pattern p are satisfied. Next, `subst_list` applies the internal substitution over the types in env'. We then perform a *generalisation* step where all the remaining, unconstrained unification variables in ts are replaced with bound type variables. If the value restriction applies, we additionally check that this step did not end up gener-

[4] For example, if the expression being typed was the identity function, `fn` x `=>` x, then we would obtain a solution consisting of the empty substitution and type u `->` u.

```
infer_d mn? δi Γi (val p = e) =
do
  init_state;
  n ← get_next_uvar;
  t1 ← infer_e Γi e;
  (t2,env′) ← infer_p Γi.inf_c p;
  names ← return (MAP FST env′);
  guard (distinct names)
    "Duplicate pattern variable";
  add_constraint t1 t2;
  ts ← subst_list (MAP SND env′);
  (tvs,s,ts′) ← return (gen_list n 0 ∅ ts);
  guard (tvs = 0 ∨ is_value e)
    "Value restriction violated";
  return (δ∅,∅,[],ZIP (names,MAP (λ t. (tvs,t)) ts′))
od

infer_d mn? δi Γi (type (targs) tn = t) =
do
  guard (distinct targs) "Duplicate type variables";
  guard
    (check_freevars 0 targs t ∧
     check_type_names Γi.inf_t t)
    "Bad type definition";
  return
    (δ∅,tn ↦ (targs,expand_abbrev Γi.inf_t t),[],[])
od
```

Figure 6. Selected definition inference cases.

alising any variables, i.e. there were no unconstrained unification variables.

The latter case corresponds to type inference for a new type abbreviation. Like the type system, new type abbreviations are checked for well-formedness before they are added to the typing environment.

On successful inference in either case, we return a 4-tuple consisting of the newly defined names, type definitions, constructor definitions and value definitions respectively. These are added to the typing environment as we move on to subsequent definitions.

5. Inferencer verification

We divide the verification effort for our type inferencer into soundness and completeness theorems. Informally, inferencer soundness shows that any program with an inferred type has a valid typing derivation in the type system while inferencer completeness shows that any type that can be derived in the type system for a program is generalised by the inferred type. In both directions, we further divide the proofs into expression-level and top-level theorems. This division is useful as it turns out that both expression-level theorems are required for each of the top-level proofs. Several conversion functions, e.g. `conv_decl` will appear in the theorems below. These convert between representations of the type system and the inferencer, e.g. from sets to lists, but are otherwise non-crucial to the proofs.

5.1 Expression-level theorems

The key difference between type system judgements and inferencer solutions at the expression level is the presence of unification variables in their respective typing judgements. Moreover, the inferencer is completely deterministic while the relational type system can have several typing judgements for a single expression[5]. Hence, the inferred type needs to generalise all possible types for an expression and unification variables allow it to do this deterministically: they should appear wherever there is a free choice of type.

To formalise the relation between type system judgements and inferencer solutions with unification variables, we begin by defining a *substitution completion* relation. Intuitively, s_2 is the result of applying additional unification constraints, $constraints$, to s_1. To be in the relation, the extra checks on s_2 shown below are used to guarantee that the result of applying s_2 on any inferred type, $t\,[s_2]$, has no unification variables. Our soundness and completeness theorems then relate this resulting type to the typing judgement.

```
sub_complete tvs next_uvar s1 constraints s2 ⟺
pure_add_constraints s1 constraints s2 ∧
count next_uvar ⊆ FDOM s2 ∧
∀ uv.
  uv ∈ FDOM s2 ⇒ check_t tvs ∅ (Infer_Tuvar uv)[s2]
```

Next, we need invariants on the inferencer state. The first of these invariants checks that the monadic state is consistent, e.g. we do not use unification variables that have not been generated. The second invariant similarly checks for consistency between parts of the constructor and module environments of $\hat{\Gamma}_t$ and Γ_i. These parts of the typing environments are used but not modified at the expression level. The final piece of both theorems are invariants that link the changing parts of both typing environments, namely, the variables and their types. These will be explained in the corresponding theorems.

```
check_state st v_env ⟺
t_wfs st.subst ∧
check_env (count st.next_uvar) v_env ∧
FDOM st.subst ⊆ count st.next_uvar

check_env_e Γ̂t Γi ⟺
check_menv Γi.inf_m ∧ menv_alpha Γi.inf_m Γ̂t.m ∧
tenv_ctor_ok Γ̂t.c ∧ Γi.inf_c = Γ̂t.c
```

The soundness theorem shows that (under suitable consistency assumptions) *any* completion of a solution from `infer_e` corresponds to a typing judgement in the type system. The soundness invariant[6], `tenv_inv`, carries this property up to the variable typing environment: it states that whenever we successfully lookup a variable x in Γ_i with type t, a corresponding lookup of x in $\hat{\Gamma}_t$ yields type t' such that t' `= conv_t` t`[`s`]`.

Theorem 5.1. Expression-level soundness.

```
⊢ infer_e Γi e st = (Success t,st′) ∧
  check_env_e Γ̂t Γi ∧ check_state st Γi.inf_v ∧
  sub_complete (num_tvs Γ̂t.v) st′.next_uvar
    st′.subst constraints s ∧
  tenv_inv s Γi.inf_v Γ̂t.v ⇒
  Γ̂t ⊢e e : conv_t t[s]
```

Proof. By induction using the induction theorem of `infer_e` and case analysis. Our `tenv_inv` invariant is motivated by the cases where we add variables into the typing environment i.e. `Fun`, `Let` and variable lookups `Var`. The proof is otherwise routine with the correct choice of invariant. □

The completeness theorem shows that (under suitable consistency assumptions) for any typing judgement, the inferencer succeeds and we can find *some* completion of its solution to match that typing judgement. Like Theorem 5.1, we need a completeness invariant, `tenv_invC`, that carries this property up to the variable typing environment. Namely, we assume that the inferencer is

[5] For example, the identity function `fn` x `=>` x can be typed as `int -> int`, `bool -> bool` or `'a -> 'a` in the type system whereas we will always infer u `->` u where u is a fresh, unconstrained unification variable.

[6] Our actual invariants also deal with alpha equivalence between the environments that can be introduced at the definitions level.

started in some state from which we already know a completion such that lookups in $\hat{\Gamma}_t$ corresponds to lookups in Γ_i under the completion.

Theorem 5.2. Expression-level completeness.

$\vdash \hat{\Gamma}_t \vdash_e e : t \wedge$ `check_env_e` $\hat{\Gamma}_t\ \Gamma_i \wedge$
`check_state` $st\ \Gamma_i$`.inf_v` $\wedge$
`sub_complete (num_tvs` $\hat{\Gamma}_t$`.v)` st`.next_uvar` st`.subst`
$constraints\ s \wedge$ `FDOM` s `= count` st`.next_uvar` $\wedge$
`tenv_invC` $s\ \Gamma_i$`.inf_v` $\hat{\Gamma}_t$`.v` $\Rightarrow$
$\exists\, t'\ st'\ s'\ constraints'$.
`infer_e` $\Gamma_i\ e\ st$ `= (Success` t'`,`st'`)` $\wedge$
`sub_complete (num_tvs` $\hat{\Gamma}_t$`.v)` st'`.next_uvar`
st'`.subst` $constraints'\ s' \wedge$
`FDOM` s' `= count` st'`.next_uvar` $\wedge$ `t_compat` $s\ s' \wedge$
t `= conv_t` t'`[`s'`]`

Proof. By rule induction on typing derivations. Interesting cases occur when we add variables to the typing environment and when we need to apply unification constraints in the inferencer.

To illustrate, let us consider $\vdash_e$ `fn x => x + 5: int -> int`. The type system types this by adding some (valid) type, say, `x:int` to its environment. The inferencer on the other hand, adds a fresh unification variable u to its environment. Since we generated a new unification variable, we need to constrain it in the substitution completion in a way that satisfies `tenv_invC` for our inductive hypothesis. To do this, we precisely apply the constraint corresponding to the type picked by the type system i.e. we constrain u to `int`.

More interestingly, when we type the body `x + 5`, we inductively know a list of the unification constraints, *constraints* (including the one for u) that completes the inferencer's initial internal substitution, st, to match the type system. However, the inferencer now attempts to apply its own unification constraint $constraints'$ between u and `int` on st. Our general strategy for these cases is to first show that applying $constraints'$ after applying *constraints* succeeds but has no effect since it must be implied by *constraints*. Then, we show that the order of unification constraints can be reordered without changing the resulting substitution. This implies that (1) applying $constraints'$ on st succeeds and (2) further applying *constraints* on the result gives us a completed substitution. These can then be used as suitable witnesses for the conclusion of this theorem. □

5.2 Top-level theorems

Our top-level soundness and completeness theorems apply to the type checking phase of an entire CakeML program. As before, we focus here on the handling of new definitions as the type system and inferencer behave similarly above the definitions level. In both directions, the main difficulty is in reconciling the value restriction rules of the type system with the relatively simpler implementation in the inferencer. The form of our value restrictions leads to an interesting interplay between the expression-level soundness/completeness theorems and both top-level theorems. Note that the first argument to the type system is set to `true` i.e. the additional principal type restrictions are turned on. Since the type system picks principal types and the inferencer also infers a principal type, our completeness theorem additionally shows that their results must be alpha equivalent.

To begin, we define a full invariant between Γ_t and Γ_i that checks consistency of the typing environments. For example, it encompasses `check_env_e` used above and the `tenv_bvl` check forces the variable environments to be of the form V as described above. Crucially, `tenv_alpha` forces the two variable environments to be alpha equivalent.

`env_rel` $\Gamma_t\ \Gamma_i \iff$
`tenv_bvl` Γ_t`.v` $\wedge$ `tenv_val_ok` Γ_t`.v` $\wedge$
`tenv_mod_ok` Γ_t`.m` $\wedge$ `check_menv` Γ_i`.inf_m` $\wedge$
`menv_alpha` Γ_i`.inf_m` Γ_t`.m` $\wedge$ `check_cenv` Γ_t`.c` $\wedge$
Γ_i`.inf_c =` Γ_t`.c` $\wedge$ `tenv_tabbrev_ok` Γ_t`.t` $\wedge$
Γ_i`.inf_t =` Γ_t`.t` $\wedge$ `check_env` $\emptyset$ Γ_i`.inf_v` $\wedge$
`tenv_alpha` Γ_i`.inf_v` Γ_t`.v`

Theorem 5.3. Definition-level soundness.

$\vdash$ `infer_d` $mn^?\ \delta_i\ \Gamma_i\ d\ st$ `=`
`(Success (`δ_i'`,`τ`,`c`,`v`),`st'`)` $\wedge$ `env_rel` $\Gamma_t\ \Gamma_i \Rightarrow$
`T,` $mn^?$`, conv_decl` δ_i`,` Γ_t
$\vdash_d d$: `conv_decl` δ_i'`, (`τ`,`c`,conv_v` v`)`

Proof. By case analysis on the input definition. The important cases arise in variable definitions of the form `val` p `=` e.
Case: e is not a value. By Theorem 5.1, we have that the inferred solution for e is a valid typing in the type system. The inferencer additionally checks that no unconstrained unification variables are in the inferred type, t. To use the DLET_MONO rule, we need to show additionally that t is the unique choice of type for e. Suppose for contradiction that we had some other type for e in the type system, t' such that $t \neq t'$; by Theorem 5.2, there is a completion of our inferred solution to yield t'. However, since there are no unconstrained unification variables in the solution, the additional unification constraints from this completion cannot change the inferred type. Hence, t `=` t' and the inferred type is unique.
Case: e is a value. We need to show that the inferred type for e is (1) a valid typing judgement in the type system and (2) a most general type. For (1), we first note that the generalisation process replaces unconstrained unification variables with bound type variables. We can equivalently apply a set of unification constraints between unification variables and type variables. Since every unconstrained unification variable is now constrained, this is a substitution completion. Hence, by Theorem 5.1, this is valid typing judgement for e. For (2), any other type for e in the type system, t', has, by Theorem 5.2, a corresponding completion of the inferencer's solution that yields it. The inferencer generalizes unification variables that are unconstrained but these are precisely the variables that get constrained by the completion. Hence, we use this completion to construct the required type variable substitution by matching each type variable to the substituted type of the unification variable it generalises. To illustrate further, consider the identity function `val f = fn x => x`. The right-hand expression is typed as u `->` u by the inferencer which then generalizes it to `'a -> 'a`. For any other valid type, e.g. `int -> int`, we know by Theorem 5.2 a completion which, in this case maps u to `int`. We can use this to produce a corresponding de Bruijn substitution, namely, mapping `'a` to `int`. □

Theorem 5.4. Definition-level completeness.

$\vdash$ `T,` $mn^?$`,` δ`,` $\Gamma_t \vdash_d d$: δ'`, (`τ`,`c`,`v`)` $\wedge$ `env_rel` $\Gamma_t\ \Gamma_i \wedge$
`conv_decl` δ_i `=` $\delta \Rightarrow$
$\exists\, st'\ \delta_i'\ v'$.
`conv_decl` δ_i' `=` δ' $\wedge$
`infer_d` $mn^?\ \delta_i\ \Gamma_i\ d\ st$ `=`
`(Success (`δ_i'`,`τ`,`c`,`v'`),`st'`)` $\wedge$
`tenv_alpha` v' `(bind_var_list2` v `Empty)` $\wedge$
`MAP FST` v' `= MAP FST` v $\wedge$ `check_env` $\emptyset\ v'$

Proof. By case analysis on the input definition. The important cases arise in variable definitions of the form `val` p `=` e.
Case: e is not a value. By DLET_MONO, we have a unique type, t, for e and by Theorem 5.2 the inferencer succeeds and there is a completion of its solution to yield t. Suppose for contradiction that the inferencer's solution has at least one unconstrained unification variable. We now construct two distinct completions by unifying

all such unconstrained unification variables with `int` and `bool` respectively. By Theorem 5.1, the resulting (distinct) types are both valid typing judgements in the type system. This contradicts the uniqueness of t so we have no unification variables in the solution and the value restriction check in the inferencer succeeds.

Case: e is a value. We need to show that (1) the inferencer succeeds and (2) any most general type for e is alpha equivalent to the inferred solution. For (1), we note directly that by Theorem 5.2, there exists a completion of the inferencer's solution for that type. We prove (2) by constructing type variable substitutions from (2.1) the type system's type to the inferred type and (2.2) the inferred type to the type system's type. The proof of (2.1) is similar to soundness: we construct a substitution completion from the generalisation step and by Theorem 5.1, this is a valid typing of e in the type system which must be generalised by a most general type. The proof of (2.2) uses Theorem 5.2 to construct type variable substitutions for the generalised unification variables. □

6. Type soundness

While the soundness and completeness theorems for the inferencer give us a practical algorithm for type checking CakeML programs, the type soundess theorem tells us that those programs that do have a type will not get stuck. This is important for the usual software engineering reasons, but also because the rest of the verified compiler uses the knowledge that the source program does not get stuck. For example, for a function application, the compiler can generate code that directly pulls a pointer from a closure record and jumps to it. It does not have to also generate a check that the value being called is actually a closure, because the operational semantics would get stuck if a non-closure value ends up being applied as a function.

We prove type soundness in two stages. The first, for expressions, is proved via preservation and progress lemmas with respect to a small-step operational semantics [11]. The second uses a big-step semantics for definitions, and is proved directly by induction over the list of definitions. This is relatively straightforward, since a definition cannot diverge, unless one of its constituent expressions does. Most of the interesting details occur at the expression level, and so we focus on it here.

A typical type soundness proof uses a structural operational semantics or a reduction semantics, where function application is modelled with substitution. In contrast, our semantics is based on the CEK-machine [3].[7] Thus, we have environments that give values to free variables, continuation stacks that explicitly model control flow, and closures to represent function values; we also have a store. We chose this style of semantics because it fit well with our big-step semantics for expressions, which is what the compiler verification uses – we have proved the two semantics equivalent, so we can use either according to convenience. If we had chosen a different form of small-step semantics our proofs (especially our big-step/small-step equivalence proofs) would be structured differently, but the more intricate details would be essentially similar.

6.1 Values and environments

Figure 7 gives the definition of environments and values, as well as the shape of additional typing judgements to give them types. Constructor values contain the unique stamp of their constructor (or none in the case of a tuple). Closures contain an expression, the name of the function's argument, and all three kinds of environments, since the expression can refer to free variables, constructor names, and module names. Recursive closures contain a bundle of named recursive functions, in addition to the environments.

[7] We are not aware of any type soundness proof in the literature that uses such a semantics.

$$
\begin{array}{rl}
Mv \equiv id \mapsto v & \text{module environments} \\
Cv \equiv cid \mapsto (\mathbb{N} \times tid\_exn) & \text{constructor environments} \\
\Delta \equiv (cn \times tid\_exn) \mapsto (\alpha^* \times t^*) & \text{constructor stamp environments} \\
S \equiv \ldots & \text{store typing (definition omitted)}
\end{array}
$$

$$
\begin{array}{rcll}
v & := & \texttt{Lit}\ l & \text{literals} \\
 & | & \texttt{Con}\ (cid\ tid\_exn)^?\ v^* & \text{constructors} \\
 & | & \texttt{Closure}\ Mv\ Cv\ Vv\ x\ e & \text{closure values} \\
 & | & \texttt{RecClosure}\ Mv\ Cv\ Vv\ (\langle x, x, e\rangle)^*\ x & \text{recursive closures} \\
 & | & \texttt{Loc}\ loc & \text{heap locations} \\
 & | & \texttt{Vector}\ v^* & \text{immutable vectors} \\
Vv & := & \epsilon & \text{empty environment} \\
 & | & x : v, Vv & \text{bind a value}
\end{array}
$$

where loc ranges over numbers.

$$
\begin{array}{rl}
tvs,\ \Delta,\ S \vdash_v v : \hat{t} & \text{value typing} \\
\Delta,\ S \vdash_{env} Vv : V & \text{environment typing} \\
S,\ \Delta \vdash_{mod} Mv : M & \text{module env. typing} \\
\Delta \vdash_{con} Cv : C & \text{constructor env. typing}
\end{array}
$$

Figure 7. Values and typing judgements

The Mv and Vv environments are the counterparts to the M and V type environments, mapping identifiers to values rather than types. Although Mv is straightforward, Vv has a small subtlety.

Type environments V can both bind variables to types and bind new type variables, but Vv can only bind variables to values and cannot bind type variables. This is sufficient because of the value restriction. To type a `let`-expression (or top-level definition), the type environment is extended with the type variables (implicitly) bound by the `let`, before typing the bound expression. However, the value restriction requires that that expression is a syntactic value, and so the environment does not need to be extended because the syntactic value can immediately be converted to a value (element of v) without consulting the environment. This supports a simplification for looking up a value in Vv: it can directly have the same type as it had when it was added to Vv. If instead, it could pass over a type variable binding, its type would have to have its de Bruijn indices shifted when doing so, and the preservation theorem would have to take this possibility into account.

Constructors need two additional environments, Cv which models the lexical scoping of constructors, and maps identifiers to the constructor's number of arguments and its unique identity (i.e., its stamp). By including the number of arguments, the small-step semantics can get stuck if the constructor is given the wrong number, and hence the compiler can assume that constructors are never applied to the wrong number of arguments. The Δ environment is not used by the small-step semantics, or the type system, but it is needed in the value typing judgements. It maps constructor stamps to the type information for that constructor. This separation supports our type soundness proof. Consider the following program:

```
datatype t = D of int
val x = D 4
datatype u = D of bool
val y = x
val z = D true
```

After executing the first 2 definitions, we have a state with the following environments:

$$Cv \equiv \{\texttt{D} \mapsto \langle 1, \texttt{t}\rangle\}$$
$$\Delta \equiv \{\langle \texttt{D}, \texttt{t}\rangle \mapsto \langle [], \texttt{int}\rangle\}$$
$$Vv \equiv \{\texttt{x} \mapsto \texttt{Con D t 4}\}$$

Thus, looking ahead, `y` has type `t`, as does `x`, and `z` has type `u`. If we evaluate the next definition, the environments change:

$$Cv \equiv \{\texttt{D} \mapsto \langle 1, \texttt{u}\rangle\}$$
$$\Delta \equiv \{\langle \texttt{D}, \texttt{u}\rangle \mapsto \langle [], \texttt{bool}\rangle; \langle \texttt{D}, \texttt{t}\rangle \mapsto \langle [], \texttt{int}\rangle\}$$
$$Vv \equiv \{\texttt{x} \mapsto \texttt{Con D t 4}\}$$

Because the binding of constructor names in Cv follow lexical scoping, the new binding for constructor `D` shadows the previous one. This allows `z` to retain type `u`. Because Δ uses the stamp, it keeps both bindings, and since the value for `x` already in the environment also uses the stamp, the type of `x` remains `t`. It is essential that neither type changes.

The value typing judgement uses three pieces of data: a set of bound type variables `tvs` for closures that were created with those variables; a constructor stamp environment Δ for constructor values (N.B., it does not take a lexical Cv because once something is a value, it should have no free variables of any sort); and a store for location values. The environment and module environment judgements do not need the type variable input, because their rules existentially quantify one when needed. The constructor environment judgement additionally does not need the store typing.

Figure 8 gives selected rules for typing values. The VTUPLE rule simply types all of the argument values ($\vdash_{vs}$ does $\vdash_{v}$ for all of its arguments) and returns the tuple constructor applied to those types.

The VCON rule finds the type parameters and argument types of the constructor in Δ. It makes an arbitrary (well-formed) instantiation of those parameters in the argument types, and checks that those are the actual types of the argument values. The returned type is the type constructor that corresponds to tid_exn, applied to the instantiating types.

The VCLOSURE rule types all of the environments in the closure, and uses the resulting type environments to type the expression.

The ENVBIND rule types an environment, one binding at a time. It uses an arbitrary tvs to ensure that closures originating from polymorphic bindings maintain enough polymorphism. Consider the following example:

```
fun f x = x
val a = f 1
val b = f true
```

After evaluating the first definition, the environment is

$$Vv \equiv \{\texttt{f} \mapsto \texttt{Closure } \{\} \ \{\} \ \{\} \ \texttt{x x}\}$$

If we are not allowed to use a polymorphic type for `f`, then we have to make a choice, say `int`, and we get the following, which cannot type both subsequent definitions.

$$V \equiv \texttt{f} : \langle 0, \texttt{int} \to \texttt{int}\rangle, \epsilon$$

Allowing type variables lets us instead get the following, which can type both definitions (recall that types use de Bruijn indices).

$$V \equiv \texttt{f} : \langle 1, 0 \to 0\rangle, \epsilon$$

Lastly, the CONENV rule checks for various consistency conditions on the three forms of constructor environments. For example, `check_con_env` is used to check that the same constructor names appear in all three environments and have corresponding types.

6.2 Definitions

The definition level type soundness does not introduce any new environments or intermediate types. The only challenge is in carefully managing the various invariants about which names are defined δ, to ensure that the stamp uniqueness guarantees that we rely on are maintained. Most of these are straightforward, but tedious, so we omit them here, and only explain the u parameter to the definition-level judgements in Figure 4.

Recall that we use u to control whether we check if a value-restricted definition has a principal type or not. In order for our inferencer completeness result to hold, we have to do this check since the inferencer has to reject (at least some) programs that lack principal types. However, the type soundness induction does not go through with the check present. Therefore we strengthen the theorem by proving type soundness assuming that u is false, and therefore the check is not performed. We also prove the (obvious) fact that this is a strictly more permissive type system: that any program with a type when u is true, also has the same type when it is false. This lets us move the type soundness theorem itself from the more permissive type system, to the less permissive one that corresponds to the inference algorithm.

The following program illustrates how type preservation fails when the check is left on.

```
val f x = x
val x = ref [1]
val _ = x := []
val y = f x
```

Here `x` and `y` have type `int list`, and the type checker accepts the program. However, after evaluating the first three statements, we are left with the following environment (omitting the indirection through the store to simplify):

$$Vv \equiv \{\texttt{x} \mapsto \texttt{ref []}; \texttt{f} \mapsto \ldots\}$$

From this, we can generate many different type environments, and all of them will give y a different type, and thus, the `type_pe_determ` check of DLET_MONO will fail.

$$V \equiv \{\texttt{x} \mapsto \langle 0, \texttt{int ref}\rangle; \texttt{f} \mapsto \ldots\}$$
$$V' \equiv \{\texttt{x} \mapsto \langle 0, \texttt{bool ref}\rangle; \texttt{f} \mapsto \ldots\}$$

Without the check, the type system can give `y` the now non-principal type of `int list ref` to satisfy the preservation theorem.

Modules and signatures, instead of the store, can also provide an example.

```
structure M :> sig val f : int -> int end =
struct
  fun f x = x
end;
val v = (fn y => y) M.f;
```

Here, once evaluation has put `f` into the module environment and the signature is lost, `v` no longer has a principal type.

7. Related Work

7.1 Type inference

Damas and Milner proved Algorithm $\mathcal{W}$ to be a sound, principal type inferencer for ML expressions [1]. Our work formalizes this result for a real ML implementation, namely CakeML, and further extends it to modules, constructors, references, and exceptions. This required us to develop a declarative type system with principal types for value restricted definitions.

Type system design is delicate, and adding features beyond the basic Hindley-Milner system can easily ruin principal types –

$$\text{(VTUPLE)}\ \frac{tvs,\ \Delta,\ S \vdash_{vs}\ vs\ :\ ts}{tvs,\ \Delta,\ S \vdash_{v}\ \texttt{Conv NONE}\ vs\ :\ \texttt{Tapp}\ ts\ \texttt{TC\_tup}}$$

$$\text{(VCON)}\ \frac{\begin{array}{c}\texttt{EVERY (check\_freevars}\ tvs\ \texttt{[])}\ ts' \\ \texttt{LENGTH}\ tvs' = \texttt{LENGTH}\ ts' \\ tvs,\ \Delta,\ S \vdash_{vs}\ vs\ :\ \texttt{MAP (type\_subst (}\emptyset\ \texttt{|++ REVERSE (ZIP (}tvs', ts'\texttt{))))}\ ts \\ \texttt{FLOOKUP}\ \Delta\ (cn,tn) = \texttt{SOME}\ (tvs',ts)\end{array}}{tvs,\ \Delta,\ S \vdash_{v}\ \texttt{Conv (SOME (}cn,tn\texttt{))}\ vs\ :\ \texttt{Tapp}\ ts'\ \texttt{(tid\_exn\_to\_tc}\ tn\texttt{)}}$$

$$\text{(VCLOSURE)}\ \frac{\begin{array}{c}\Delta \vdash_{con}\ env.\ \texttt{c}\ :\ \Gamma_t.\ \texttt{c} \\ \texttt{tenv\_mod\_ok}\ \Gamma_t.\ \texttt{m} \\ S,\ \Delta \vdash_{mod}\ env.\ \texttt{m}\ :\ \Gamma_t.\ \texttt{m} \\ \Delta,\ S \vdash_{env}\ env.\ \texttt{v}\ :\ \Gamma_t.\ \texttt{v} \\ \texttt{check\_freevars}\ tvs\ \texttt{[]}\ t_1 \\ \texttt{bind\_tvar}\ tvs\ \Gamma_t,n\ :\ (0,\ t_1) \vdash_{e}\ e\ :\ t_2\end{array}}{tvs,\ \Delta,\ S \vdash_{v}\ \texttt{Closure}\ env\ n\ e\ :\ t_1\ \texttt{->}\ t_2}$$

$$\text{(ENVBIND)}\ \frac{tvs,\ \Delta,\ S \vdash_{v}\ v\ :\ t \quad \Delta,\ S \vdash_{env}\ env\ :\ \Gamma_t}{\Delta,\ S \vdash_{env} (n,v)::\ env\ :\ \Gamma_t, n\ :\ (tvs,t)}\qquad \text{(CONENV)}\ \frac{\begin{array}{c}\texttt{tenv\_ctor\_ok}\ C \\ \texttt{ctMap\_ok}\ \Delta \\ \texttt{check\_con\_env}\ \Delta\ Cv\ C\end{array}}{\Delta \vdash_{con}\ Cv\ :\ C}$$

Figure 8. Selected value typing rules

hence the hope for a complete inferencer – and possibly render the system undecidable. Despite these complications, the mainstream of typed functional programming language design seems to be for increasing the features, and living without principal types, or requiring some type annotations. Sometimes the trouble comes from a subtle interaction of seemingly innocuous features (as in the "avoidance problem" from the SML module system [2]). That is the case here, with the value restriction, although we were able to restrict the type system enough to regain principal types, but the value restriction is a very minor tweak to the type system. For other popular features, including GADTs, there is no solution for now, and one must be resigned to using the type inference algorithm as the most precise specification of the type system [10].

Naraschewski and Nipkow have mechanised a soundness and completeness proof for Algorithm $\mathcal{W}$ [8]. Their underlying language is a MiniML, with similar features to Damas and Milner's language, lacking imperative features, or any value restriction. They also axiomatized the specification of the unification algorithm, whereas we tie into an existing verified implementation. In contrast, they verify completeness for a system that generalises nested lets, whereas we have not yet done that.

7.2 Type soundness

Our type system and type soundness proof broadly follow our previous work in OCaml$_{light}$ [9], with the main extension being support for modules and signatures. Similar to OCaml$_{light}$, we use de Bruijn indices for type variables, and concrete names for other variables. We also explicitly bind type variables in the environment. However, our operational semantics is different (CEK-style, rather than SOS-style), and we have a more flexible treatment of constructor names that follow lexical scoping, whereas OCaml$_{light}$ required them to be unique.

8. Conclusion

We have formally specified a type system and type inferencer for CakeML. We provide a type soundness theorem for the type system as well as soundness and completeness theorems linking the type inferencer's behaviour to the type system's. CakeML aims to be a practical programming language that is both easy to program in and easy to reason about. The theorems in this paper are steps toward the latter goal: any input CakeML program that is accepted by the inferencer is guaranteed to have well-defined semantics.

Without completeness, our specification [4] of the top-level CakeML read-eval-print loop (REPL) was unsatisfactory. When a type-incorrect definition is entered, the REPL should print `<type error>`, and await a new definition, and not execute any part of the type-incorrect one. Ideally, whether a definition is type correct is specified with respect to the type system, whereas the implementation of the REPL uses the inferencer. However, without a inferencer completeness theorem, we could not prove that the implementation of the REPL did not signal a type error even when the specification said it should not. Thus, we had to use the inferencer to specify which definitions had type errors, even as the inferencer soundness theorem allowed us to use the type system to specify what type good definitions had. By verifying completeness, we have significantly improved the semantics of the CakeML REPL.

Acknowledgements

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

[1] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212,

New York, NY, USA, 1982. ACM. URL: http://doi.acm.org/10.1145/582153.582176, doi:10.1145/582153.582176.

[2] D. Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon University, 2005.

[3] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[4] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–191. ACM Press, Jan. 2014. doi:10.1145/2535838.2535841.

[5] R. Kumar and M. Norrish. (Nominal) Unification by recursive descent with triangular substitutions. In *Interactive Theorem Proving, First International Conference, ITP 2010*, volume 6172 of *LNCS*, 2010.

[6] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3), 1978.

[7] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[8] W. Naraschewski and T. Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning*, 23:299–318, 1999.

[9] S. Owens. A sound semantics for OCaml light. In *Programming Languages and Systems: 17th European Symposium on Programming, ESOP 2008*, volume 4960 of *LNCS*, pages 1–15. Springer, Mar. 2008. doi:10.1007/978-3-540-78739-6_1.

[10] D. Vytiniotis, S. L. Peyton Jones, T. Schrijvers, and M. Sulzmann. OutsideIn(X) Modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5), 2011.

[11] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.

A. The value restriction in ML implementations

There are three options of what to do when an unconstrained type cannot be generalised due to the value restriction. CakeML signals a type error, as do Moscow ML and OCaml when compiling. PolyML and SML/NJ create a new abstract type variable, `_a` or `?.X1`, which then ensures the function cannot be called, since there are no values of that type. At the REPL, Moscow ML and OCaml both create special, mutable type variables that are set on the first use and actually change the type of `id_mono`. MLton is included for completeness, but since it is a whole program compiler, this particular issue cannot arise.

A.1 CakeML

```
 -- CakeML starting up --
val id_mono = if true then (fn x => x) else (raise Bind);
<type error>
id_mono 1;
<type error>
```

A.2 PolyML

```
Poly/ML 5.5.2 Release
> val id_mono = if true then (fn x => x) else (raise Bind);
Warning-The type of (id_mono) contains a free type variable. Setting it to a unique
   monotype.
val id_mono = fn: _a -> _a
> id_mono 1;
Error-Type error in function application.
   Function: id_mono : _a -> _a
   Argument: 1 : int
   Reason:
      Can't unify int (*In Basis*) with
      _a (*Constructed from a free type variable.*)
      (Different type constructors)
Found near id_mono 1
Static Errors
```

A.3 SML/NJ

```
Standard ML of New Jersey v110.78 [built: Thu Aug 20 19:23:18 2015]
- val id_mono = if true then (fn x => x) else (raise Bind);
stdIn:1.6-1.58 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val id_mono = fn : ?.X1 -> ?.X1
- id_mono 1;
stdIn:2.1-2.10 Error: operator and operand don't agree [overload conflict]
  operator domain: ?.X1
  operand:         [int ty]
  in expression:
    id_mono 1
```

A.4 Moscow ML repl

```
Moscow ML version 2.10
Enter `quit();' to quit.
- val id_mono = if true then (fn x => x) else (raise Bind);
! Warning: Value polymorphism:
! Free type variable(s) at top level in value identifier id_mono
> val id_mono = fn : 'a -> 'a
- id_mono 1;
! Warning: the free type variable 'a has been instantiated to int
> val it = 1 : int
- id_mono;
> val it = fn : int -> int
- id_mono true;
! Toplevel input:
! id_mono true;
!         ^^^^
! Type clash: expression of type
!   bool
! cannot have type
!   int
```

A.5 Moscow ML compiled

```
dhcp297B:tmp sao$ cat test.sml
structure test = struct
  val id_mono = if true then (fn x => x) else (raise Bind)
end;
dhcp297B:tmp sao$ mosmlc test.sml
! Value polymorphism: Free type variable at top level in value identifier id_mono
```

A.6 OCaml repl

```
        OCaml version 4.02.3

# let id_mono = if true then (fun x -> x) else assert false;;
val id_mono : '_a -> '_a = <fun>
# id_mono 1;;
- : int = 1
# id_mono;;
- : int -> int = <fun>
# id_mono true;;
Error: This expression has type bool but an expression was expected of type
         int
```

A.7 OCaml compiled

```
dhcp297B:tmp sao$ cat test.ml
let id_mono = if true then (fun x -> x) else assert false
dhcp297B:tmp sao$ ocamlc test.ml
File "test.ml", line 1, characters 14-57:
Error: The type of this expression, '_a -> '_a,
       contains type variables that cannot be generalized
```

A.8 MLton

```
dhcp297B:tmp sao$ cat test.sml
structure S = struct
  val id_mono = if true then (fn x => x) else (raise Bind)
end;
val _ = S.id_mono 1;
val _ = S.id_mono true;
dhcp297B:tmp sao$ mlton test.sml
Warning: test.sml 2.3.
  Unable to locally determine type of variable: id_mono.
    type: ??? -> ???
    in: val id_mono = if true then fn x => x else raise Bind
Error: test.sml 5.9.
  Function applied to incorrect argument.
    expects: [int]
    but got: [bool]
    in: S.id_mono true
compilation aborted: parseAndElaborate reported errors
```