Showing a CakeML program is safe

Hrutvik Kanabar, Dr Scott Owens
Supported by the UK Research Institute in Verified Trustworthy Software Systems (VeTSS)

We want a way of ensuring safety which:
- Is as general as possible.
- Allows us to circumvent restrictions of syntactic typing and large proof obligations.
- Permits reasoning about (almost) any expressible CakeML code.
- Is compositional with little added effort from users.

We are using the technique of logical relations.

In our case, these are:
- Families of type-indexed relations: there is a relation for each type in CakeML.
- Step-indexed (aka “fuelled”): to cope with recursive types and impredicativity.
- Compositional: “logical” as they respect actions of language type constructors e.g. for logical relation $R_\tau (-)$:
  $$R_\tau_1 (e_1) \land R_\tau_1 (e_2) \implies R_\tau_2 (e_1 e_2).$$

Our logical relation:

$$\Gamma \models e : \tau \implies e \text{ is safe at type } \tau \text{ in context } \Gamma$$

In other words:
“$e$ has semantic type $\tau$ in semantic context $\Gamma$”

The story so far...

We have applied the technique to a version of System F, augmented by existential, general isorecursive, product, and sum types.
This models a fragment of CakeML, and is specified in HOL4 too.

We are now modelling use cases in this language – see right.

What is CakeML?

CakeML is an open-source, functional programming language, verified compiler, and proof ecosystem.
- Language: ML-like, based on a subset of Standard ML.
- Semantics: functional big-step, specified in higher-order logic (HOL).
- Compiler: verified end-to-end correct, bootstrappable.
- HOL4: the HOL interactive theorem-prover used to specify the language/compiler, and prove properties about CakeML.

Integrating safe & unsafe code for faster programs

We want to call fast, untypeable libraries in typeable, safe user code.

For each syntactic typing rule, we prove a semantic equivalent (a compatibility lemma):

$$\vdots \vdots \vdash \frac{\vdots \vdash}{\vdash \vdash}$$

Now for any program composing typeable (safe) user code and untypeable (unsafe) library code, we can re-use the typing derivation for the safe part.

We only have to verify the unsafe library code – there is no added cost to the user. We use this approach in our version of System F.

Formally guaranteeing correct data encapsulation

We want to show that uses of a module preserve its internal data invariants.

We have expressed invariants as a semantic type, and so reasoned about them using our logical relation. We plan to prove a general, Reynolds-style abstraction theorem in future work.

Candle: a HOL prover written in CakeML

- HOL theorem-provers use a small, trusted “kernel” module.
- This has a hidden theorem type, constructed by inference rules.
- Only the module can create theorems, which must therefore be valid.
- We want to prove invariants about the Candle theorem type, and so reason about soundness of Candle.

Preserving formal safety guarantees

We want to use verified Coq code alongside CakeML code safely.

Coq extracts code to OCaml by inserting unsafe casts to fit into OCaml’s weaker type system, using Obj.magic : $\alpha \rightarrow \beta$.

Using semantic types in a verified extraction pipeline, we can justify the casts and so remove them – this regains formal proof of safety.