Generic Approach to Certified Static Checking of Module-like Language Constructs

Julia Belyakova
julbinb@gmail.com

Southern Federal University

June 20th 2017

FTfJP 2017: Formal Techniques for Java-like Programs
ECOOP 2017 Series

Contents

Interactive Theorem Provers for PL

Interactive Theorem Provers have been used for both:

1. mechanizing **formal models** of programming languages;
   - Featherweight Java with mutability [Mackay et al. 2012];
   - Dependent Object Types [Rompf and Amin 2016];
   - JSCert [Bodin et al. 2014].
2. building **certified**[1] compilers and interpreters.
   - C Compiler CompCert [Blazy and Leroy 2009];
   - SML Compiler CakeML [Tan et al. 2016];
   - JavaScript Interpreter JSRef [Bodin et al. 2014].

The Problem

Good for Proofs (formal model) $\neq$ Efficient (compiler)

[1] "Certified" means "corresponds to the formal model". For example, $\texttt{type\_check}(\Gamma, t) = \mathsf{Some}\ \tau \Rightarrow \Gamma \vdash t : \tau$.

Module-like Constructs in Programming Languages

Many programming languages have some notion of **module**:

- package;
- ML module;
- class;
- trait;
- Haskell type class.

"Modules" are often used as a means of abstraction and come in pairs of an interface and an implementation:

- interfaces/traits/protocols and classes (OO);
- signatures and modules (ML);
- type classes and instances (Haskell).

Case Study

Let's build an efficient certified static checker for a simple language with modules.

STLC with Concept Parameters: Concepts and Models

- **Concept** describes an interface — list of name-type pairs.
- **Model** defines an implementation of the concept — list of name-term pairs.

Example

```
// Concepts (interfaces)
concept CMonoid { ident : Nat;  binop : Nat -> Nat -> Nat }
concept CFoo    { boo   : Nat;  bar   : Nat -> Bool }

// Models (implementations)
model MSum of CMonoid {
  ident = 0
  binop = \x:Nat.\y:Nat. x + y
}
model MFoo of CFoo {
  boo = 5;
  bar = \x:Nat. if x > boo then true else false
}
```

```
model MProd of CMonoid {
  ident = 1
  binop = \x:Nat.\y:Nat. x * y
}
```

Static Checking of Concepts and Models

1. Concept is well-defined in CT (concepts table)
   $\Leftrightarrow$ all names are distinct $\wedge$ all types are well-defined in CT.
2. Concept Section is well-defined
   $\Leftrightarrow$ all names are distinct $\wedge$ all concepts are well-defined[2].
3. Model M is a well-defined model of C in (CT, MT)
   $\Leftrightarrow$ all names are distinct $\wedge$ all concept members are defined[2]
   $\wedge$ all terms have expected types in (CT, MT).
4. Model Section is well-defined
   $\Leftrightarrow$ all names are distinct $\wedge$ all models are well-defined[2].

Example: Model M of C $\{\, f_1 = e_{f_1};\ f_2 = e_{f_2} \,\}$

$$\vdash e_{f_1} : \tau_{f_1} \quad \wedge \quad f_1 : \tau_{f_1} \vdash e_{f_2} : \tau_{f_2}$$

[2] Later defined elements can refer to the previous ones.

Structure of Generic Library for Checking Modules

1. Definition of well-definedness of a module [formal model]: $DOk : M \to \texttt{Prop}$.

```
Definition f_mem (okCl : Prop * ctxloc) (dt : data) : Prop * ctxloc
:= match okAndCl with (ok, cl) =>
     let ok' := update_prop ok cl dt in
     let cl' := update_ctxloc  cl dt in   (ok', cl').

Definition module_ok (dts : list data) : Prop * ctxloc
:= let (ok, m) := List.fold_left f_mem dts (True, ctxloc_init) in
   (List.NoDup (get_names dts) /\ ok, m).
```

2. Algorithm checking that a module is well-defined [compiler/interpreter]: $AOk : M \to \texttt{bool}$.
3. Proof of correctness of the algorithm with respect to the formal definition (soundness and maybe completeness).

$$AOk(M) = \texttt{true} \iff DOk(M)$$

4. Efficient representation of a well-defined module (ctxloc).

STLC with Concept Parameters: Syntax

Types

$\tau := \mathtt{Nat} \mid \mathtt{Bool} \mid \tau \to \tau \mid \mathtt{C} \,\#\, \tau$	*types*
$\phi := \{f_i : \tau_i\}$	*concept types*
$\psi := (\mathtt{C}, \{f_i = e_i\})$	*model types*

Terms

$e ::= x \mid \lambda x : \tau.\, e \mid e\ e$	*STLC terms*
$\mid n \mid e + e \mid \ldots$	*nat/bool exprs*
$\mid \lambda c \,\#\, \mathtt{C}.\, e$	*concept abstraction*
$\mid e \,\#\, \mathtt{M}$	*model application*
$\mid c{::}f$	*member invocation*
$p ::= \mathtt{CSec}\ \mathtt{MSec}\ e$	*cpSTLC program*

STLC with Concept Parameters: Typing

Typing Judgement

$$\mathrm{CT} * \mathrm{MT}\,;\Gamma \vdash e : \tau,$$

where CT and MT are finite maps built from CSec and MSec.

$$\frac{\mathtt{C} \in \mathit{dom}(\mathrm{CT}) \qquad \mathrm{CT} * \mathrm{MT}\,;\Gamma, c\#\mathtt{C} \vdash e : \tau}{\mathrm{CT} * \mathrm{MT}\,;\Gamma \vdash \lambda c\#\mathtt{C}.e : \mathtt{C}\,\#\,\tau} \text{ (T-CAbs)}$$

$$\frac{c\#\mathtt{C} \in \Gamma \qquad \mathtt{C} \in \mathit{dom}(\mathrm{CT}) \qquad f : \tau_f \in \mathrm{CT}(\mathtt{C})}{\mathrm{CT} * \mathrm{MT}\,;\Gamma \vdash c{::}f : \tau_f} \text{ (T-CInvc)}$$

$$\frac{\mathtt{M\ of\ C}\,\{\ldots\} \in \mathtt{MT} \qquad f : \tau_f \in \mathrm{CT}(\mathtt{C}) \qquad M\#\mathtt{C}' \notin \Gamma}{\mathrm{CT} * \mathrm{MT}\,;\Gamma \vdash M{::}f : \tau_f} \text{ (T-MInvc)}$$

Example

f =

$\lambda c\#\mathtt{CMonoid}.\lambda x{:}\mathtt{Nat}.\ c{::}\mathtt{binop}\ x\ 5 : \mathtt{CMonoid}\,\#\,\mathtt{Nat} \to \mathtt{Nat}$

$f\,\#\,\mathtt{MSum} : \mathtt{Nat} \to \mathtt{Nat}$

f # MSum *evaluates to*

$\lambda x{:}\mathtt{Nat}.\ \mathtt{MSum}{::}\mathtt{binop}\ x\ 5$

f # MSum 3 *evaluates to*

$\mathtt{MSum}{::}\mathtt{binop}\ 3\ 5 \longrightarrow (\lambda x{:}\mathtt{Nat}.\ \lambda y{:}\mathtt{Nat}.\ x + y)\ 3\ 5 \longrightarrow^{*} 8$

STLC with Concept Parameters: Semantics

Small-Step Operational Semantics

$$\mathrm{CT} * \mathrm{MT}\,; t \longrightarrow t'$$

$$\frac{\mathtt{M}\ \mathtt{of}\ \mathtt{C}\{\ldots\} \in \mathtt{MT} \qquad f = t_f \in \mathrm{MT}(\mathtt{M})}{\mathrm{CT} * \mathrm{MT}\,; M{::}f \longrightarrow QMM(\mathrm{MT}, \mathtt{M}, t_f)}\ \text{(E-CInvc)}$$

Where $QMM(\mathrm{MT}, \mathtt{M}, t)$ qualifies with model name M all free variables of t_f that appear in $\mathrm{MT}(\mathtt{M})$.

Example

$$\begin{aligned}
&\mathtt{MFoo::bar} &&= (\lambda x{:}\mathtt{Nat}.\mathbf{if}\ x > \mathsf{boo}\ \mathbf{then} \ldots) \\
&\mathtt{MFoo::bar}\ 42 &&\longrightarrow (\lambda x{:}\mathtt{Nat}.\mathbf{if}\ x > \mathsf{MFoo::boo}\ \mathbf{then} \ldots)\ 42
\end{aligned}$$

STLC with Concept Parameters: Soundness Problem

To prove type soundness, we need to prove that evaluation of the member invocation preserves typing:

$$\mathrm{CT}{*}\mathrm{MT} \vdash \mathsf{M}::f : \tau_f \wedge \mathsf{M}::f \longrightarrow QMM(t_f) \implies \mathrm{CT}{*}\mathrm{MT} \vdash QMM(t_f) : \tau_f$$

This has something to do with the definition of well-definedness for models and a model section.

:(

Have to unfold generic definitions of well-definedness;
copy-paste driven reasoning about fold-left based definitions.

We need a generic principle of reasoning about the definitions.

Current Progress

- Low-level library for certified transformation of lists into sets (`MSet`), and lists of pairs into finite maps (FMap): ~2000 LOC.
- Generic library for certified checking of simple modules, single-pass modules, and single-pass modules-implementations: ~1500 LOC.
- STLC with Concept Parameters: soundness proof up to 2 lemmas about well-definedness of models: ~6000 LOC.

Source code: concept-params at github/julbinb.

Future Work

1. Reasoning principles for generic definitions of well-definedness.
2. More strategies of checking modules (e.g. all members can refer to each other; nested modules; first-class modules).

References I

J. Mackay et al. "Encoding Featherweight Java with Assignment and Immutability Using the Coq Proof Assistant". In: *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*. FTfJP '12. Beijing, China: ACM, 2012, pp. 11–19.

T. Rompf and N. Amin. "Type Soundness for Dependent Object Types (DOT)". In: *SIGPLAN Not.* 51.10 (Oct. 2016), pp. 624–641.

M. Bodin et al. "A Trusted Mechanised JavaScript Specification". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: ACM, 2014, pp. 87–100.

S. Blazy and X. Leroy. "Mechanized semantics for the Clight subset of the C language". In: *Journal of Automated Reasoning* 43.3 (2009), pp. 263–288.

References II

Y. K. Tan et al. “A New Verified Compiler Backend for CakeML”. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP 2016. Nara, Japan: ACM, 2016, pp. 60–73.