Modeling a subset of Featherweight Java in ACL2

Mansei Kano

Featherweight Java

- Featherweight Java[1], or FJ is a minimal core calculus for modeling Java.
  - Variables, field access, method invocation object creation, and casts

Syntax:

$$
\begin{array}{lcl}
\texttt{L} & ::= & \texttt{class C extends C \{}\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \\
\texttt{K} & ::= & \texttt{C(}\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{)\{super(}\overline{\texttt{f}}\texttt{); this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;\}} \\
\texttt{M} & ::= & \texttt{C m(}\overline{\texttt{C}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}} \\
\texttt{e} & ::= & \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m(}\overline{\texttt{e}}\texttt{)} \mid \texttt{new C(}\overline{\texttt{e}}\texttt{)} \mid \texttt{(C)e}
\end{array}
$$

- Benefits of modeling
  - Describing some aspect of language formally
  - Stating and proving its properties

Mechanized Proofs for Type System

- There have been many attempts to mechanize type soundness proofs.
  - Coq, Isabelle, Agda, etc.

- Few attempts for object languages in ACL2.
  - Chose FJ as a target language
  - From the feedback of project proposal, we reduced the scope

$$
\begin{aligned}
CL &::= class\ C\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} \\
K &::= C(\overline{C}\ \ \overline{f})\{this.\overline{f} = \overline{f};\} \\
M &::= C\ m\ (\overline{C}\ \overline{x})\ \{return\ e;\} \\
e &::= x \mid e.f \mid e.m(\overline{e}) \mid new\ C(\overline{e}) \\
v &::= new\ C(\overline{v})
\end{aligned}
$$

Expression

- We modeled expression as a mutually recursive datatype
- expr
  - A union type of var, field, new, invk
- exprlist
  - List of expr types

$$e \quad ::= x \mid e.f \mid e.m(\bar{e}) \mid new\ C(\bar{e})$$

```
(fty::deftypes expression
  (fty::deftagsum expr
                 (:VAR ((v stringp)))
                 (:FIELD ((obj expr-p)
                          (field stringp)))
                 (:NEW ((class stringp)
                        (args exprlist)))
                 (:INVK ((obj expr-p)
                         (method stringp)
                         (args exprlist))))
  (fty::deflist exprlist :elt-type expr))
```

Class,Constructor,Method

- We modeled class declarations by defaggregate

$$
\begin{array}{lll}
CL & ::= & class\ C\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} \\
K & ::= & C(\overline{C}\ \ \overline{f})\{this.\overline{f} = \overline{f};\} \\
M & ::= & C\ m\ (\overline{C}\ \overline{x})\ \{return\ e;\}
\end{array}
$$

```
(std::defaggregate fj_class
  ((name          stringp           :rule-classes :type-prescription)
   (fields        string-alistp     :rule-classes :type-prescription)
   (constructor   fj_constructor-p :rule-classes :type-prescription)
   (methods       method-listp      :rule-classes :type-prescription))
  :tag :class
  )

(std::defaggregate fj_method
  ((returnType stringp        :rule-classes :type-prescription)
   (name       stringp        :rule-classes :type-prescription)
   (params     string-alistp :rule-classes :type-prescription)
   (body       expr-p        :rule-classes :type-prescription))
  :tag :method
  )
```

Typing

- The typing judgement for expression has the form of $\theta; \Gamma \vdash e : C$
  - Because the classtable θ is stable in any context, we abbreviate the judgement as $\Gamma \vdash_\theta e : C$
  - (ex) new Pair(new A(), new B()).fst : A

$$\Gamma \vdash_\theta x : \Gamma(x) \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash_\theta e_0 : C_0 \qquad \theta_f(C_0) = \overline{C}\ \overline{f}}{\Gamma \vdash_\theta e.f_i : C_i} \qquad \text{(T-FIELD)}$$

$$\frac{\Gamma \vdash_\theta e_0 : C_0 \qquad \theta_m(C_0)(m) = \overline{D} \longrightarrow C \qquad \Gamma \vdash_\theta \overline{e} : \overline{D}}{\Gamma \vdash_\theta e_0.m(\overline{e}) : C} \qquad \text{(T-INVK)}$$

$$\frac{\theta_f(C) = \overline{D}\ \overline{f} \qquad \Gamma \vdash_\theta \overline{e} : \overline{D}}{\Gamma \vdash_\theta new\ C(\overline{e}) : C} \qquad \text{(T-NEW)}$$

$$\frac{\overline{x} : \overline{D}, this : C \vdash_{\theta,C} e : C_0}{\vdash_{\theta,C} C_0\ m\ (\overline{D}\ \overline{x})\ \{return\ e;\}} \qquad \text{(T-METHOD)}$$

g

$$\frac{K = C(\overline{C}\ \overline{f})\{this.\overline{f} = \overline{f};\} \qquad \vdash_{\theta,C} C_0\ m\ (\overline{D}\ \overline{x})\ \{return\ e;\}}{\vdash_\theta class\ C\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}} \qquad \text{(T-CLASS)}$$

Typing in ACL2 representation

- We defined typing as mutually recursive functions.
- $typeof(\theta, \Gamma, e) = C$
  - C is nil if e cannot be typed
- $typeof\text{-}list(\theta, \Gamma, \overline{e}) = \overline{C}$
  - $\overline{C}$ is nil-terminated list

```
(defines typeof
  (define typeof (classDefs ctxt e)
    :measure (expr-count e)
    :guard (and (class-listp classDefs)
                (string-alistp ctxt)
                (expr-p e))
    :verify-guards nil
    (expr-case e
               :VAR (getTypeFromCtxt ctxt e.v)
               :FIELD (let ((e-type (typeof classDefs ctxt e.obj)))
                          (getFieldType (fields classDefs e-type) e.field))
               :NEW (let ((type-list (getTypesFromParams (fields classDefs e.class))))
                      (if (equal (typeof-list classDefs ctxt e.args)
                                 type-list)
                        e.class
                        nil))
               :INVK (let ((objtype (typeof classDefs ctxt e.obj)))
                       (let ((mlist (getMethodList classDefs objtype)))
                         (if (equal (typeof-list classDefs ctxt e.args)
                                    (getTypesFromParams (fj_method->params (getMethod r
                                    ↪  e.method))))
                           (fj_method->returnType (getMethod mlist e.method))
                           nil))))))
  (define typeof-list (classDefs ctxt e-list)
    :measure (exprlist-count e-list)
    :guard (and (exprlist-p e-list)
                (class-listp classDefs)
                (string-alistp ctxt))
    (if (atom e-list)
      nil
      (cons (typeof classDefs ctxt (car e-list))
            (typeof-list classDefs ctxt (cdr e-list))))))
```

Evaluation

- The evaluation for expression has the form of $e \longrightarrow e'$.
  - (ex) new Pair(new A(), new B()).fst $\longrightarrow$ new A()

$$\frac{\theta_f(C) = \overline{C}\ \overline{f}}{(new\ C(\overline{e})).f_i \longrightarrow e_i} \tag{E-Field}$$

$$\frac{mbody(m, C) = \overline{x}, e_0}{(new\ C(\overline{e})).m(\overline{d}) \longrightarrow [\overline{d}/\overline{x}, new\ C(\overline{e})/this]e_0} \tag{E-Invk}$$

$$\frac{e \longrightarrow e'}{e.f \longrightarrow e'.f} \tag{EC-Field}$$

$$\frac{e_0 \longrightarrow e_0'}{e_0.m(\overline{e}) \longrightarrow e_0'.m(\overline{e})} \tag{EC-Invk}$$

$$\frac{e_i \longrightarrow e_i'}{e_0.m(..., e_i, ...) \longrightarrow e_0'.m(..., e_i', ...)} \tag{EC-Invk2}$$

$$\frac{e_i \longrightarrow e_i'}{e_0.m(..., e_i, ...) \longrightarrow e_0'.m(..., e_i', ...)} \tag{EC-New}$$

Evaluation in ACL2 representation

- We defined evaluation as mutually recursive functions.

- $\mathit{eval}\text{-}\mathit{expr}(\theta, e) = e'$
  - e' is nil if no evaluation rule can apply

- $\mathit{eval}\text{-}\mathit{exprlist}(\theta, \overline{e}) = \overline{e'}$
  - $\overline{e'}$ is nil-terminated expression list

Type soundness

- Type soundness is a property of the system that assures a well-typed program will not get stuck.
    - Getting stuck is a state that the expression is not a value, and no evaluation can apply to it.
- Proved by the following theorems

Theorem 1 (Progress) *if* $\cdot \vdash_\theta e : T$, *and* $\vdash_\theta \overline{CL}$, *then e is either a value or there exists e' such that* $e \longrightarrow e'$.

Theorem 2 (Preservation) *if* $\cdot \vdash_\theta e : T$, $\vdash_\theta \overline{CL}$, *and* $e \longrightarrow e'$ *then* $\cdot \vdash_\theta e' : T$.

Type soundness in ACL2 (failed attempt)

- These theorems can be modeled by using functions as follows

Theorem 3 (Progress') *if* $typeof(\theta, \Gamma, e) \neq nil$ *and* $t\text{-}class\text{-}list(\theta, \overline{CL}) = t$, *then* $isvalue(e) = t$ *or* $eval\text{-}expr(\theta, e) \neq nil$.

Theorem 4 (Preservation') *if* $typeof(\theta, \Gamma, e) \neq nil$, $t\text{-}class\text{-}list(\theta, \overline{CL}) = t$, *and* $eval\text{-}expr(\theta, e) \neq nil$ *then* $typeof(\theta, \Gamma, e) = typeof(\theta, \Gamma, eval\text{-}expr(\theta, e))$.

- We may be tempted to formalize them in ACL2 given below.

```
(defthm progress
    (implies (and (class-listp classDefs)
                  (expr-p e)
                  (typeof classDefs nil e)
                  (t-class-list classDefs))
             (or (isvalue e)
                 (eval-expr classDefs e))))
```

Type soundness in ACL2

- Previous approach was failed
  - No induction scheme due to mutual recursion.
- We can handle this by stating conjunction of functions for e and $\bar{e}$ with make-flag
- This approach seems to work, but it still requires lots of subtle lemmas
  - Proof is not completed yet

```
(make-flag typeof-flag typeof)

(defthm-typeof-flag
  (defthm progress-term
    (implies (and (class-listp classDefs)
                  (expr-p e)
                  (typeof classDefs nil e)
                  (t-class-list classDefs))
             (or (isvalue e)
                 (eval-expr classDefs e)))
    :rule-classes :type-prescription
    :flag typeof)
  (defthm progress-list
    (implies (and (class-listp classDefs)
                  (exprlist-p e-list)
                  (typeof-list classDefs nil e-list)
                  (t-class-list classDefs))
             (or (isvaluelist e-list)
                 (eval-exprlist classDefs e-elist)))
    :rule-classes :type-prescription
    :flag typeof-list))
```

References

1. Igarashi, A., Pierce, B. C., & Wadler, P. (2001). Featherweight Java: a minimal core calculus for Java and GJ. ACM Transactions on Programming Languages and Systems (TOPLAS), 23(3), 396-450.