CS389R Recursion and induction Fall2021

Modeling a subset of Featherweight Java in ACL2

Mansei Kano

December 1, 2021

1 Introduction

Nowadays, many programming languages utilize a type system. A type system is one of the lightweight formal methods which guarantees at compile time that a well-typed program will never go wrong. There have been many attempts to mechanize type soundness proofs in the programming language field by Coq, Isabelle, etc. Although several studies have mechanized proofs using ACL2[6, 5], it is still a few attempts for machine-checked type soundness in ACL2, especially for object languages. Therefore, we tried to prove the type soundness of such a language in this project, and chose Fatherweight Java [2] as a target language.

Featherweight Java (hereinafter referred as FJ) is a minimal core calculus for modeling Java. Its functionalities are limited to the core of object oriented programming, such as variables, field access, method invocation, object creation, and casts. Formal model for a programming language is very useful for describing a specific aspect of real language, stating and proving its properties. Modeled language is also useful to argue an extension of the actual language. In fact, featherweight GJ, which adds generic type to FJ, was started from FJ and proved its type soundness. In another words, Java with generic type is type-safe as long as focusing on the core features of object-oriented programming.

All this time, like FJ, many subsets have been proposed such as Middleweight Java[1], Welterweight Java[4] and $Java_{light}$[3]. Some of the subsets are modeling a large part of Java to state that Java is type-safe. On the other hand, FJ modeled the core of Java, which enables us to skip tedious proofs, make it concise, and focus on proving the new feature, as in the case of FGJ. However, it is still large to model the whole FJ in a month as a laboratory project. Therefore, we reduce the scope and define it as a subset of FJ (or SFJ, for short) and prove it in ACL2.

2 A subset of Featherweight Java

2.1 Syntax

In this section, we introduce SFJ that models FJ without a subtyping feature. That is, there is no class inheritance, polymorphism. Because cast feature is meaningless under the nonexistence of subtyping, we also omit it. The syntax of SFJ is in Figure 1. The metavariables C and D represent class name; f stands for field name; m denotes for method name; x ranges over variables. We also use shorthand notation for specification. We write $\overline{e}$ and $\overline{C}\ \overline{x}$ as shorthand for $e_1, e_2, ...e_n$ and $C_1\ x_1, C_2\ x_2, ..., C_n\ x_n$, respectively.

We define a classtable θ and *mbody* as an auxiliary function. θ consists of two pairs, θ_f and θ_m. θ_f is mapping from classname to fields of the class. $\theta_m(C)(m)$ returns the type of the method m. *mbody* returns the variables of the parameter and the body of the method. The definitions of these functions are given in Figure 2.

$$
\begin{array}{rll}
P & ::= \overline{CL};\ e & \text{Program} \\
CL & ::= class\ C\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} & \text{Class} \\
K & ::= C(\overline{C}\ \ \overline{f})\{this.\overline{f} = \overline{f};\} & \text{Construcor} \\
M & ::= C\ m\ (\overline{C}\ \overline{x})\ \{return\ e;\} & \text{Method} \\
e & ::= x \mid e.f \mid e.m(\overline{e}) \mid new\ C(\overline{e}) & \text{Expression} \\
v & ::= new\ C(\overline{v}) & \text{Value}
\end{array}
$$

Figure 1: Syntax of SFJ

$$
\begin{array}{rl}
\theta = (\theta_f, \theta_m) & \text{classtable} \\
\theta_f(C) = \overline{D}\ \overline{f},\ where\ class\ C\ \{\overline{D}\ \overline{f};\ K\ \overline{M}\} & \\
\theta_m(C)(m) = \overline{D} \longrightarrow C,\ where\ C\ m\ (\overline{D}\ \overline{x})\ \{return\ e;\} \in \overline{M} & \\
mbody(m, C) = \overline{x}, e, where\ C\ m\ (\overline{D}\ \overline{x})\ \{return\ e;\} \in \overline{M} &
\end{array}
$$

Figure 2: Auxiliary function

2.2 Typeing

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 $\Gamma \vdash_\theta e : C$. The typing context Γ is a mapping from variables to types. The typing rules of SFJ are as follows.

Expression Typing

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

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

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

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

Method Typing

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

Class Typing

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

2.3 Operational Semantics

The evaluation for expression has the form of $e \longrightarrow e'$. They are almost the same as FJ but eliminate casts evaluation. We use the notation $[d/x]e$ as a substitution that means every x, which appears in e, replace with d. The evaluation rules are given below.

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

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

$$\frac{e \longrightarrow e'}{e.f \longrightarrow e'.f} \quad \text{(EC-Field)}$$

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

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

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

3 ACL2 representation

3.1 Expression

We use fty::deftagsum in the fty library to model expresisons. The macro enables us to define a tagged sum type like Haskell or ML. Although modeling variable and field access are straightforward, object creation and method invocation are a bit complicated because SFJ may receive a list of expressions as an argument. Therefore, we define exprlist with each type expr and define the expression as a mutually recursive data type. The expression of SFJ can be modeled shown below.

```
(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))
```

3.2 Class, Construcotor, Method

We modeled class, method, and constructor by using defaggregte. The macro introduces a record structure like C language. Those models are given below.

```
(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
  )

(std::defaggregate fj_constructor
  ((returnType       stringp       :rule-classes :type-prescription)
   (params           string-alistp :rule-classes :type-prescription)
   (inits            string-listp  :rule-classes :type-prescription))
```

```
  :tag :constructor
  )

(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
  )
```

string-alistp is a guard we define that assures it is pair of string lists like '(("a" . "A") ("b" . "B") ... ("z" . "Z")). We use string for variables, classnames, and Types. In SFJ, the body of the constructor is in the form of $this.\overline{f} = \overline{f}$, but we only need the name of variables, so we use string-listp for the body. method-listp is the guard that guarantees it is the nil-terminated list of methods.

3.3 Typing

Although typing rules are defined as relation with θ, Γ, e, C in FJ and SFJ, we defined them as a function in our ACL2 representation. The function typeof receives θ, Γ, e and returns e's type, that is, $typeof(\theta, \Gamma, e) = C$. If e cannot be typed, C is nil. However, we defined expression as a mutual recursive datatype in ACL2, any functions that receive e have to be mutual recursion in our encoding. Thus, we also define typeof-list for list of expressions. It receives $\theta, \Gamma, \overline{e}$ and return list of e's types. That is, $typeof\text{-}list(\theta, \Gamma, \overline{e}) = \overline{C}$. The type checking function is as follows.

```
(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 mlist
                                    ↪  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))))))
```

The first step of typeof is pattern matching for e. If e is a variable, then typeof returns $e's$ type from the type context. IF it is field access, then call typeof recursively for $e.obj$ which is a sub expression of e, and returns the field's type. If it is an object creation, then typeof-list is called for the arguments. If they are well-typed, then return the classname as a type. Otherwise, return nil because it doesn't satisfy T-New. The typeof does almost the same thing if e is a method invocation.

We also implemented functions correspoding to T-Class and T-Method. Unlike the typeof function, those functions return t or nil because their roles are simply checked whether the class, constructor, methods are given correctly, and the expression in the method body is typed well.

3.4 Evaluation

As well as typing, evaluation is defined as relation with e and e'. In our ACL2 representation, it is a mutually recursive function with eval-expr and eval-exprlist. The eval-expr receives θ, e and returns e', where e' is the expression evaluated by one step from e. That is, $\mathit{eval\text{-}expr}(\theta, e) = e'$. On the other hand, The eval-exprlist receives a list of expressions and returns them by evaluating each expression by one step. That is, $\mathit{eval\text{-}exprlist}(\theta, \overline{e}) = \overline{e'}$. Both functions return nil if the evaluation doesn't happen. Our implementation of them is given below.

```
(defines eval-expr
  (define eval-expr (classDefs e)
      :measure (expr-count e)
      :guard (and (expr-p e)
                  (class-listp classDefs))
      :verify-guards nil
      (expr-case e
                 :FIELD (if (isvalue e.obj)
                            (let ((val (eval-expr classDefs e.obj)))
                              (expr-case val
                                         :NEW (getithField val.args
                                                           (fields classDefs val.class)
                                                           e.field)
                                         :otherwise nil))
                            (expr-field (eval-expr classDefs e.obj) e.field))
                 :NEW (expr-new e.class (eval-exprlist classDefs e.args))
                 :INVK (if (isvalue e.obj)
                           (if (isvaluelist e.args)
                             (expr-case e.obj
                                        :NEW (let ((method (getMethod (getMethodList classDefs
                                        ↪  e.obj.class)
                                                                      e.method)))
                                               (substitution (fj_method->body method)
                                                             (getVarArgs (fj_method->params method))
                                                             e.args
                                                             e.obj))
                                        :otherwise nil)
                             (expr-invk e.obj e.method (eval-exprlist classDefs e.args)))
                           (expr-invk (eval-expr classDefs e.obj) e.method e.args))
                 :otherwise nil))
  (define eval-exprlist (classDefs e-list)
    :measure (exprlist-count e-list)
    (if (atom e-list)
      nil
      (cons (eval-expr classDefs (car e-list))
            (eval-exprlist classDefs (cdr e-list))))))
```

The eval-expr first does pattern matching of e. If e is field access, then either E-Field or EC-Field may be able to apply, so we call (isvalue e.obj) to check whether $e.obj$ is a value. The isvalue is a mutually recursive function that returns t if given expression is a value. Otherwise, it returns nil. If e is an object creation, then we call eval-exprelist to evaluate the arguments. If e is method invocation, then we call the isvalue to determine between E-Invk, EC-Inck and EC-Invk2. We call the substitution function if it is E-Invk to replace all parameters and string "this" with arguments and current object, respectively. If e is a variable, then return nil because there is no evaluation rule related to variables.

4 Type Soundness

Type systems of programming language prevent some errors during execution by detecting type errors at compile time. Type soundness is an important property of the system that 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. For example, $a, a.m(b)$, and $new\ C(a, b, c)$ are stuck expression in our language. Type soundness is usually proved in two steps known as the progress and preservation theorem. The progress theorem assures that if an expression e is typed, it is a value or there exists another expression e' such that $e \longrightarrow e'$. While the preservation theorem states if an expression e is typed by T and evaluated to e', then the expression e' is also typed by T. In SFJ, preservation and progress theorems are given below.

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* $\cdot \vdash_\theta e' : T$.

We use the notation $\cdot$ to indicate empty type context. In the previous section, we define typing and evaluation as a function. Therefore, we translated the theorems into function-based ones 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* $typeof(\theta, \Gamma, e) = typeof(\theta, \Gamma, eval\text{-}expr(\theta, e))$.

The t-class-list is the function for checking each class is well-typed under the T-Class rule. If so, it returns t; otherwise, it returns nil. A straightforward representation of the above progress theorem in ACL2 would be as follows.

```
(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))))
```

The class-listp and expr-p are recognizers for list of fj_class and an expression, respectively. Those are also included in the guard for the typeof and eval-expr function. Therefore, we state (class-listp classDefs) and (expr-p) as hypotheses. However, the above formalization does not work because those functions, typeof, isvalue, eval-expr are defined by mutual recursion, so ACL2 cannot find any induction scheme. The simple way to handle mutually recursive function is stating the conjunction of both functions with make-flag, which is a feature of ACL2 that provides an induction scheme for mutual recursion. Our formalization of the progress theorem is shown below.

```
(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))
```

We made a flag function and attempted to prove the conjunction of the progress theorem by induction of the typeof function. Usually, in hand proof, we can prove by both induction of typing and evaluation, but typing is simpler than evalution in SFJ, we chose the induction by typing. The modeling of preservation theorem is given below.

```
(defthm-typeof-flag
  (defthm preservation-term
    (implies (and (class-listp classDefs)
                  (expr-p e)
                  (typeof classDefs nil e)                ; type(e) = T
                  (eval-expr classDefs e)                 ; eval(e) = e'
```

```
              (t-class-list classDefs))
         (equal (typeof classDefs nil e)              ; type(e) = type((eval(e)))
                (typeof classDefs nil (eval-expr classDefs e))))
  :rule-classes :type-prescription
  :flag typeof)
(defthm preservation-list
  (implies (and (class-listp classDefs)
                (exprlist-p e-list)
                (typeof-list classDefs nil e-list)
                (eval-exprlist classDefs e-list)
                (t-class-list classDefs))
           (equal (typeof-list classDefs nil e-list)
                  (typeof-list classDefs nil (eval-exprlist classDefs e-list))))
  :rule-classes :type-prescription
  :flag typeof-list))
```

Unfortunately, we could not prove both theorems because it required a bunch of subtle lemmas, and we could not finish them. Even some of the required lemmas include mutual recursion, so it would not be an easy task.

5 Conclusion

This report provides an idea of how to model object-oriented language such as FJ and formalize type soundness. We use fty::deftagsum and std::defaggregate to model expressions and class declarations. Typing and evaluation are defined as mutually recursive functions, and type soundness is formalized by taking conjunction of functions for single expression and a list of expressions with make-flag. Although we reduced the scope of modeling FJ, we could not complete the proof of type soundness. The most significant factor that makes proof complicated is mutual recursion. In object language such as FJ, a method receives a list of expressions. In contrast, in a functional language such as lambda calculus, we can simulate it by a function with single arity that receives a higher-order function. That is, our formalization would be more concise if the target language is a functional language. Our approach may work for modeling FJ, but it requires lots of work to prove the type soundness.

Acknowledgement

I would like to thank Professor Warren Hunt, Teaching Assistant Scott Staley and Carl Kwan for helping me to work on this project.

References

[1] Gavin M Bierman, MJ Parkinson, and AM Pitts. Mj: An imperative core calculus for java and java with effects. Technical report, University of Cambridge, Computer Laboratory, 2003.

[2] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[3] Tobias Nipkow and David Von Oheimb. Java light is type-safe—definitely. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 161–170, 1998.

[4] Johan Östlund and Tobias Wrigstad. Welterweight java. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 97–116. Springer, 2010.

[5] Daniel J Singer. Proving type soundness of simple languages in acl2. *Ann Arbor*, 1001:48109.

[6] Sol Swords and William R Cook. Soundness of the simply typed lambda calculus in acl2. In *Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 35–39, 2006.