Object Reuse and Behavior Adaptation in Java-like languages *

(Sketched Proofs)

Lorenzo Bettini
Dipartimento di Informatica, Università di Torino

Betti Venneri
Dipartimento di Sistemi e Informatica, Università di Firenze

1. Properties

In this document we sketch the proof of the type safety property for CompObj, which means that no message-not-understood error can occur at run time during method invocations on composed objects.

First of all, we have to define the typing rules for annotated run-time expressions, that is for lists and pairs of lists which are the run-time representation of objects. Moreover, the rule for method invocation needs to take into account method call annotations. These new rules are defined in Figure 1, while all the other typing rules are the same as the ones presented in Figure 2, where e is intended to denote a run-time expression. We note that the type of a composed object is the one of the head of the list, consistently with the typing rule for object composition. Moreover, rule (T-LIST) implicitly requires that $1 \neq \varepsilon$ since $\varepsilon$ is not typable.

By the definition of annotation, it is easy to verify the following basic property.

**Property 1.1 (Types are preserved under annotation).** If $\Gamma \vdash e : T$ for some $\Gamma$ and $T$, then $\Gamma \vdash e[\Gamma] : T$.

Thus, in the following, to make the presentation of the properties simpler, we will not write the annotations explicitly when they are not significant, and we will use $e$ to refer to annotated run-time expressions.

The main property is the Substitution Lemma, which shows that by replacing variables with object values having subtypes, in an expression $e$, the resulting expression is still well-typed, namely it has a subtype of the original expression. The crucial cases are when the expression is either a method invocation or an object composition. For simplicity, in this context we are not considering occurrences of $\text{next}$ in the expression $e$. With some additional technical steps we can extend the Substitution Lemma to expressions containing $\text{next}$.

**Lemma 1.2 (Substitution Lemma).** If $\Gamma, x : \mathcal{C}, \text{this} : (\mathcal{C}/\emptyset) \vdash e : T$, and

1. $\Gamma \vdash e : T' where T' <: \mathcal{C}$,

2. $\Gamma \vdash v : T_1 where T_1 <: (\mathcal{C}/\emptyset)$,

then $\Gamma \vdash [x \leftarrow e, \text{this} : v]e : T'$ for some $T'$ such that $T' <: T$.

**Proof.** By induction on the derivation of $\Gamma, x : T, \text{this} : (\mathcal{C}/\emptyset) \vdash e : T$, with a case analysis on the last applied rule.

(TA-INVK) The thesis follows by the induction hypothesis in all cases but in the most crucial one when

- $e = \text{this.m(\mathcal{V})}^{\mathcal{B} \rightarrow \mathcal{B}}$,
- $v = (1, l')$,
- $\text{finddef}(m, 1, \mathcal{B} \rightarrow \mathcal{B}) = 1_1$, where $1_1 \neq 0 \neq 1$

that is, when there is an object in the list where $m$ is redefining (with the same signature). Since, by hypothesis $\Gamma \vdash (1, l') : T_1 <: (\mathcal{C}/\emptyset)$, then $\text{concrete}(T_1)$ holds; so $1 \subseteq l_1 \subseteq l'$ implies $\Gamma \vdash (1_1, l') : T_1$ and $\text{concrete}(T)$. By (TA-INVK) we have

$$\Gamma \vdash \text{this} : (\mathcal{C}/\emptyset) \quad \Gamma \vdash \text{m(\mathcal{V})}^{\mathcal{B} \rightarrow \mathcal{B}} : \mathcal{B}$$

By definition of $\text{finddef}$, we have that $\text{mtype}(m, T_1) = \mathcal{B} \rightarrow \mathcal{B}$; by induction hypothesis, $\mathcal{V} = [x \leftarrow e, \text{this} : v] \emptyset_0$ is such that $\Gamma \vdash \mathcal{V} : T'$ for some $T'$ such that $T' <: \emptyset_0$. Thus, we can apply (TA-INVK), (using transitivity $<$):

$$\Gamma \vdash (\mathcal{V}) \quad mtype(m, T_1) = \mathcal{B} \rightarrow \mathcal{B}$$

$$\Gamma \vdash \text{concrete}(T_1)$$

(T-COMP) For $e = e_1 \leftrightarrow e_2$, the last applied rule is (T-COMP):

$$\Gamma \vdash \text{new } \mathcal{C}(\mathcal{V}) : \mathcal{C}$$

$$\Gamma \vdash \text{new } \mathcal{C}(\mathcal{V}) : \mathcal{C}$$

(T-LISTH)

$$\Gamma \vdash 1 : T$$

$$\Gamma \vdash 1' : l'$$

$$\Gamma \vdash (1', l') : T$$

(T-ORUNTIME)

$$\Gamma \vdash e : T$$

$$\Gamma \vdash e : T$$

(TA-INVK)

$$\Gamma \vdash \text{m(\mathcal{V})}^{\mathcal{B} \rightarrow \mathcal{B}} : \mathcal{B}$$

Figure 1. Run-time expression typing

[Copyright notice will appear here once 'preprint' option is removed.]
Lemma 1.3. If $\text{mtype}(m, C) = \mathbb{E} \rightarrow \mathbb{B}$ and $\text{mbody}(m, C) = (x, e)$, then, for some $D$, where $C = D$ or $D$ is a superclass of $C$, there exists $T < B$ such that:

1. If $m \notin \text{reddef}(C)$, then $x : \mathbb{E} \text{this} : (D/\emptyset) \vdash e : T$.

2. Otherwise, $x : \mathbb{E} \text{this} : (D/\emptyset), \text{next} : S \vdash e : T$, where $S = \text{required}(D)$.

Proof. Straightforward induction on a derivation of $\text{mbody}(m, C)$, using (T-METHOD) or (T-METHO) for point 1, and (T-RMETHOD) for point 2.

Theorem 1.4 (Type Preservation). If $\Gamma \vdash e : T$ and $e \rightarrow e'$ then $\Gamma \vdash e' : T'$ for some $T' < T$.

Proof. By induction on a derivation of $e \rightarrow e'$. Concerning reduction rules, the most interesting case is when the last applied rule is (R-INVK) or (R-RINVK): then the proof follows from Lemma 1.3 and Substitution Lemma 1.2. The property on congruence rules follows straightforwardly by the induction hypothesis: in the particular case when $e$ is an object composition, the proof proceeds as the case (T-COMP) of Lemma 1.2.

Concerning the progress property, we actually prove that values which can result from applying an evaluation rule to a well-typed expression are of the shape $(1,1)$, where $1 \neq e$, which we call final values, while pairs of different lists are used only as receivers of method invocations. A final value $(1,1)$ denotes the fully evaluated object which is represented by the list 1. Recall that pairs of lists do not occur in the source code of our language, namely an evaluation rule always applies to a well-typed (annotated) source code expression.

Property 1.5 (Source Code Reduction). Suppose $e$ is a closed, well-typed expression, which is the annotated version of a source code expression. Then $e \rightarrow e'$ for some $e'$.

Proof. By induction on typing rules of Figure 2, using (TA-INVK) in place of (T-INVK).

Theorem 1.6 (Progress). Let $e$ be a closed run-time expression. If $\vdash e : T$, for some $T$, and $e \rightarrow e'$ for some $e'$, then either $e'$ is a final value, or $e'$ can be reduced.

Proof. By induction on the derivation $e \rightarrow e'$, with a case analysis on the final rule, taking into account that $e'$ is well typed by Theorem 1.4. We only consider reduction rules: if $e \rightarrow e'$ is obtained by using a congruence rule, then the thesis follows from the induction hypothesis.

(R-New), (R-Comp) Immediate: $e'$ is a final value.

(R-Field) $\forall$ are obtained by reducing well-typed expressions, then $\forall$ are final values by the induction hypothesis.

(R-Invk), (R-RInvk) As in the previous case, all the values in $e$ are final values, then $e'$ reduces.

(R-DInvk) We can apply either (R-INVK) or (R-RINVK).
Theorem 1.7 (Type Safety). Let $e$ be a closed source code expression, such that $\vdash e : T$. Let $e'$ be its annotated version. If $e' \rightarrow^{*} e''$, and $e''$ is such that no evaluation rule applies, then $e''$ is a final value.

Proof. By Theorem 1.4 and Theorem 1.6.