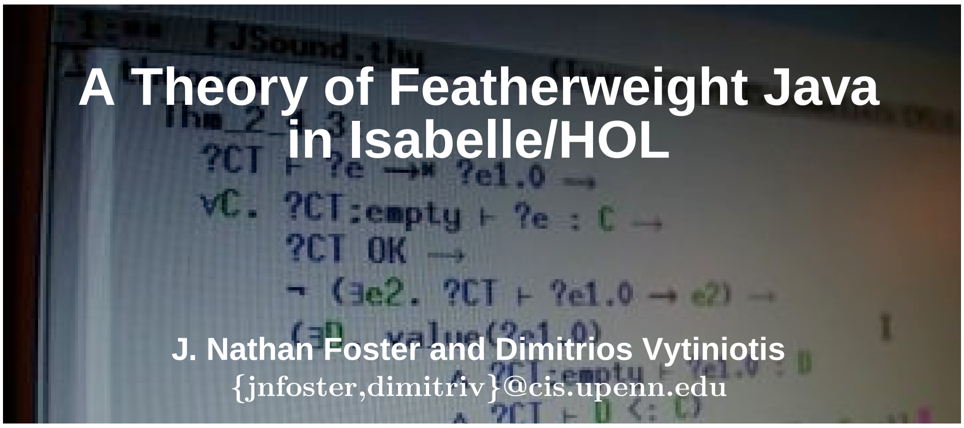

# A Theory of Featherweight Java in Isabelle/HOL

J. Nathan Foster and Dimitrios Vytiniotis
{jnfoster,dimitriv}@cis.upenn.edu

**Abstract**

We formalize the type system, small-step operational semantics, and type soundness proof for Featherweight Java [1], a simple object calculus, in Isabelle/HOL [2].

## Contents

# 1 `FJDefs`: Basic Definitions

**theory** *FJDefs* **imports** *Main*

**begin**

**lemmas** *in-set-code*[*code unfold*] = *mem-iff* [*symmetric*, *THEN eq-reflection*]

## 1.1 Syntax

We use a named representation for terms: variables, method names, and class names, are all represented as `nat`s. We use the finite maps defined in `Map.thy` to represent typing contexts and the static class table. This section defines the representations of each syntactic category (expressions, methods, constructors, classes, class tables) and defines several constants (`Object` and `this`).

### 1.1.1 Type definitions

**types** *varName* = *nat*
**types** *methodName* = *nat*
**types** *className* = *nat*
**record** *varDef* =

*vdName* :: *varName*
*vdType* :: *className*
**types** *varCtx* = *varName* $\rightharpoonup$ *className*

### 1.1.2 Constants

**consts**
*Object* :: *className*
*this* :: *varName*
**defs**
*Object* : *Object* == *0*
*this* : *this* == *0*

### 1.1.3 Expressions

**datatype** *exp* =
*Var varName*
| *FieldProj exp varName*
| *MethodInvk exp methodName exp list*
| *New className exp list*
| *Cast className exp*

### 1.1.4 Methods

**record** *methodDef* =
*mReturn* :: *className*
*mName* :: *methodName*
*mParams* :: *varDef list*
*mBody* :: *exp*

### 1.1.5 Constructors

**record** *constructorDef* =
*kName* :: *className*
*kParams* :: *varDef list*
*kSuper* :: *varName list*
*kInits* :: *varName list*

### 1.1.6 Classes

**record** *classDef* =
*cName* :: *className*
*cSuper* :: *className*
*cFields* :: *varDef list*
*cConstructor* :: *constructorDef*
*cMethods* :: *methodDef list*

### 1.1.7 Class Tables

**types** *classTable* = *className* $\rightharpoonup$ *classDef*

## 1.2 Sub-expression Relation

The sub-expression relation, written $t \in \mathit{subexprs}(s)$, is defined as the reflexive and transitive closure of the immediate subexpression relation.

**consts**
  $\mathit{isubexprs} :: (\mathit{exp} * \mathit{exp})\ \mathit{set}$
**syntax**
  $\text{-}\mathit{isubexprs} :: [\mathit{exp},\mathit{exp}] \Rightarrow \mathit{bool}\ \ (\text{-} \in \mathit{isubexprs}'(\text{-}')\ [80,80]\ 80)$
**translations**
  $e' \in \mathit{isubexprs}(e) \rightleftharpoons (e',e) \in \mathit{isubexprs}$
**inductive** *isubexprs*
**intros**
*se-field* : $e \in \mathit{isubexprs}(\mathit{FieldProj}\ e\ \mathit{fi})$
*se-invkrecv* : $e \in \mathit{isubexprs}(\mathit{MethodInvk}\ e\ m\ \mathit{es})$
*se-invkarg* : $[\![\ \mathit{ei} \in \mathit{set}\ \mathit{es}\ ]\!] \Longrightarrow \mathit{ei} \in \mathit{isubexprs}(\mathit{MethodInvk}\ e\ m\ \mathit{es})$
*se-newarg* : $[\![\ \mathit{ei} \in \mathit{set}\ \mathit{es}\ ]\!] \Longrightarrow \mathit{ei} \in \mathit{isubexprs}(\mathit{New}\ C\ \mathit{es})$
*se-cast* : $e \in \mathit{isubexprs}(\mathit{Cast}\ C\ e)$

**consts**
 $\mathit{subexprs} :: (\mathit{exp} * \mathit{exp})\ \mathit{set}$
**syntax**
  $\text{-}\mathit{subexprs} :: [\mathit{exp},\mathit{exp}] \Rightarrow \mathit{bool}\ \ (\text{-} \in \mathit{subexprs}'(\text{-}')\ [80,80]\ 80)$
**translations**
  $e' \in \mathit{subexprs}(e) \rightleftharpoons (e',e) \in \mathit{isubexprs}\hat{}*$

## 1.3 Values

A *value* is an expression of the form `new C`$(\mathit{overline} vs)$, where $\overline{vs}$ is a list of values.

**consts**
  $\mathit{vals} :: (\mathit{exp}\ \mathit{list})\ \mathit{set}$
  $\mathit{val} :: \mathit{exp}\ \mathit{set}$
**syntax**
  $\text{-}\mathit{vals} :: [\mathit{exp}\ \mathit{list}] \Rightarrow \mathit{bool}\ (\mathit{vals}'(\text{-}')\ [80]\ 80)$
  $\text{-}\mathit{val} :: [\mathit{exp}] \Rightarrow \mathit{bool}\ (\mathit{val}'(\text{-}')\ [80]\ 80)$
**translations**
  $\mathit{val}(v) \rightleftharpoons v \in \mathit{val}$
  $\mathit{vals}(\mathit{vl}) \rightleftharpoons \mathit{vl} \in \mathit{vals}$
**inductive** *vals val*
**intros**
  *vals-nil* : $\mathit{vals}([])$
  *vals-cons* : $[\![\ \mathit{val}(\mathit{vh});\ \mathit{vals}(\mathit{vt})\ ]\!] \Longrightarrow \mathit{vals}((\mathit{vh}\ \#\ \mathit{vt}))$
  *val* : $[\![\ \mathit{vals}(\mathit{vs})\ ]\!] \Longrightarrow \mathit{val}(\mathit{New}\ C\ \mathit{vs})$

## 1.4 Substitution

The substitutions of a list of expressions *ds* for a list of variables *xs* in another expression *e* or a list of expressions *es* are defined in the obvious way, and written $(ds/xs)e$ and $[ds/xs]es$ respecitvely.

**consts**
*substs* :: $(varName \rightharpoonup exp) \Rightarrow exp \Rightarrow exp$
*subst-list1* :: $(varName \rightharpoonup exp) \Rightarrow exp\ list \Rightarrow exp\ list$
*subst-list2* :: $(varName \rightharpoonup exp) \Rightarrow exp\ list \Rightarrow exp\ list$
**syntax**
*-substs* :: $[varName\ list] \Rightarrow [exp\ list] \Rightarrow [exp] \Rightarrow exp$ ('(-/-')- [80,80,80] 80)
*-subst-list* :: $[varName\ list] \Rightarrow [exp\ list] \Rightarrow [exp\ list] \Rightarrow exp\ list$ ('[-/-']- [80,80,80] 80)
**translations**
$[ds/xs]es \rightleftharpoons map\ (substs\ (map\text{-}upds\ empty\ xs\ ds))\ es$
$(ds/xs)e \rightleftharpoons substs\ (map\text{-}upds\ empty\ xs\ ds)\ e$

**primrec**
$substs\ \sigma\ (Var\ x) = (case\ (\sigma(x))\ of\ None \Rightarrow (Var\ x) \mid Some\ p \Rightarrow p)$
$substs\ \sigma\ (FieldProj\ e\ f) = FieldProj\ (substs\ \sigma\ e)\ f$
$substs\ \sigma\ (MethodInvk\ e\ m\ es) = MethodInvk\ (substs\ \sigma\ e)\ m\ (subst\text{-}list1\ \sigma\ es)$
$substs\ \sigma\ (New\ C\ es) = New\ C\ (subst\text{-}list2\ \sigma\ es)$
$substs\ \sigma\ (Cast\ C\ e) = Cast\ C\ (substs\ \sigma\ e)$
$subst\text{-}list1\ \sigma\ [] = []$
$subst\text{-}list1\ \sigma\ (h\ \#\ t) = (substs\ \sigma\ h)\ \#\ (subst\text{-}list1\ \sigma\ t)$
$subst\text{-}list2\ \sigma\ [] = []$
$subst\text{-}list2\ \sigma\ (h\ \#\ t) = (substs\ \sigma\ h)\ \#\ (subst\text{-}list2\ \sigma\ t)$

## 1.5 Lookup

The fuction *lookup f l* function returns an option containing the first element of $l$ satisfying $f$, or `None` if no such element exists

**consts** *lookup* :: $'a\ list \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a\ option$
**primrec**
$lookup\ []\ P = None$
$lookup\ (h\#t)\ P = (if\ P\ h\ then\ Some\ h\ else\ lookup\ t\ P)$

**consts** *lookup2* :: $'a\ list \Rightarrow 'b\ list \Rightarrow ('a \Rightarrow bool) \Rightarrow 'b\ option$
**primrec**
$lookup2\ []\ l2\ P = None$
$lookup2\ (h1\#t1)\ l2\ P = (if\ P\ h1\ then\ Some(hd\ l2)\ else\ lookup2\ t1\ (tl\ l2)\ P)$

## 1.6 Variable Definition Accessors

This section contains several helper functions for reading off the names and types of variable definitions (e.g., in field and method parameter declarations).

**constdefs** *varDefs-names* :: $varDef\ list \Rightarrow varName\ list$
$varDefs\text{-}names == map\ vdName$

**constdefs** *varDefs-types* :: $varDef\ list \Rightarrow className\ list$
$varDefs\text{-}types == map\ vdType$

## 1.7 Subtyping Relation

The subtyping relation, written $CT \vdash C <: D$ is just the reflexive and transitive closure of the immediate subclass relation. (For the sake of simplicity, we define subtyping directly instead of using the reflexive and transitive closure operator.) The subtyping relation is extended to lists of classes, written $CT \vdash +Cs <: Ds$.

**consts** *subtyping* :: (*classTable* ∗ *className* ∗ *className*) *set*
  *subtypings* :: (*classTable* ∗ *className list* ∗ *className list*) *set*
**syntax**
 *-subtyping* :: [*classTable*, *className*, *className*] ⇒ *bool* (- ⊢ - <: - [*80*,*80*,*80*] *80*)
 *-subtypings* :: [*classTable*, *className list*, *className list*] ⇒ *bool* (- ⊢+ - <: - [*80*,*80*,*80*] *80*)
 *-neg-subtyping* :: [*classTable*, *className*, *className*] ⇒ *bool* (- ⊢ - ¬<: - [*80*,*80*,*80*] *80*)
**translations**
 $CT \vdash S <: T \rightleftharpoons (CT,S,T) \in subtyping$
 $CT \vdash\!+\ Ss <: Ts \rightleftharpoons (CT,Ss,Ts) \in subtypings$
 $CT \vdash S \neg\!<: T \rightleftharpoons (CT,S,T) \notin subtyping$
**inductive** *subtyping*
**intros**
*s-refl* : $CT \vdash C <: C$
*s-trans* : $[\![\ CT \vdash C <: D;\ CT \vdash D <: E\ ]\!] \Longrightarrow CT \vdash C <: E$
*s-super* : $[\![\ CT(C) = Some(CDef);\ cSuper\ CDef = D\ ]\!] \Longrightarrow CT \vdash C <: D$

**inductive** *subtypings*
**intros**
*ss-nil* : $CT \vdash\!+\ [] <: []$
*ss-cons* : $[\![\ CT \vdash C0 <: D0;\ CT \vdash\!+\ Cs <: Ds\ ]\!] \Longrightarrow CT \vdash\!+\ (C0\ \#\ Cs) <: (D0\ \#\ Ds)$

## 1.8 `fields` Relation

The `fields` relation, written `fields`$(CT, C) = Cf$, relates $Cf$ to $C$ when $Cf$ is the list of fields declared directly or indirectly (i.e., by a superclass) in $C$.

**consts** *fields* :: (*classTable* ∗ *className* ∗ *varDef list*) *set*
**syntax**
 *-fields* :: [*classTable*, *className*, *varDef list*] ⇒ *bool* (*fields*′(-,-′) = - [*80*,*80*,*80*] *80*)
**translations**
 $fields(CT,C) = Cf \rightleftharpoons (CT,C,Cf) \in fields$
**inductive** *fields*
**intros**
 *f-obj*:
 $fields(CT,Object) = []$
 *f-class*:

$[\![$ $CT(C) = Some(CDef)$; $cSuper\ CDef = D$; $cFields\ CDef = Cf$; $fields(CT,D) = Dg$; $DgCf = Dg\ @\ Cf$ $]\!]$
$\Longrightarrow fields(CT,C) = DgCf$

## 1.9 mtype Relation

The `mtype` relation, written `mtype`$(CT, m, C) = Cs \to C_0$ relates a class $C$, method name $m$, and the arrow type $Cs \to C_0$. It either returns the type of the declaration of $m$ in $C$, if any such declaration exists, and otherwise returning the type of $m$ from $C$'s superclass.

**consts** *mtype* :: (*classTable* ∗ *methodName* ∗ *className* ∗ ((*className list*) ∗ *className*)) *set*
**syntax**
*-mtype* :: [*classTable*, *methodName*, *className*, *className list*, *className*] ⇒ *bool* (*mtype'*(-,-,-') = - → - [*80*,*80*,*80*,*80*] *80*)
**translations**
*mtype*(*CT*,*m*,*C*) = *Cs* → *C0* ⇌ (*CT*,*m*,*C*,(*Cs*,*C0*)) ∈ *mtype*
**inductive** *mtype*
**intros**
*mt-class*:
⟦ *CT*(*C*) = *Some*(*CDef*);
*lookup* (*cMethods CDef*) (λ*md*.(*mName md* = *m*)) = *Some*(*mDef*);
*varDefs-types* (*mParams mDef*) = *Bs*;
*mReturn mDef* = *B* ⟧
⟹ *mtype*(*CT*,*m*,*C*) = *Bs* → *B*

*mt-super*:
⟦ *CT*(*C*) = *Some* (*CDef*);
*lookup* (*cMethods CDef*) (λ*md*.(*mName md* = *m*)) = *None*;
*cSuper CDef* = *D*;
*mtype*(*CT*,*m*,*D*) = *Bs* → *B* ⟧
⟹ *mtype*(*CT*,*m*,*C*) = *Bs* → *B*

## 1.10 mbody Relation

The `mtype` relation, written `mbody`$(CT, m, C) = xs.e_0$ relates a class $C$, method name $m$, and the names of the parameters $xs$ and the body of the method $e_0$. It either returns the parameter names and body of the declaration of $m$ in $C$, if any such declaration exists, and otherwise the parameter names and body of $m$ from $C$'s superclass.

**consts** *mbody* :: (*classTable* ∗ *methodName* ∗ *className* ∗ (*varName list* ∗ *exp*)) *set*
**syntax**
*-mbody* :: [*classTable*, *methodName*, *className*, *varName list*, *exp*] ⇒ *bool* (*mbody'*(-,-,-') = - . - [*80*,*80*,*80*,*80*] *80*)
**translations**
*mbody*(*CT*,*m*,*C*) = *xs* . *e* ⇌ (*CT*,*m*,*C*,(*xs*,*e*)) ∈ *mbody*

**inductive** *mbody*
**intros**
*mb-class*:
$[\![$ $CT(C) = Some(CDef)$;
$lookup\ (cMethods\ CDef)\ (\lambda md.(mName\ md = m)) = Some(mDef)$;
$varDefs\text{-}names\ (mParams\ mDef) = xs$;
$mBody\ mDef = e$ $]\!]$
$\Longrightarrow mbody(CT,m,C) = xs\ .\ e$

*mb-super*:
$[\![$ $CT(C) = Some(CDef)$;
$lookup\ (cMethods\ CDef)\ (\lambda md.(mName\ md = m)) = None$;
$cSuper\ CDef = D$;
$mbody(CT,m,D) = xs\ .\ e$ $]\!]$
$\Longrightarrow mbody(CT,m,C) = xs\ .\ e$

### 1.11 Typing Relation

The typing relation, written $CT;\Gamma \vdash e : C$ relates an expression $e$ to its type $C$, under the typing context $\Gamma$. The multi-typing relation, written $CT;\Gamma \vdash +es : Cs$ relates lists of expressions to lists of types.

**consts**
*typing* :: $(classTable * varCtx * exp * className)$ *set*
*typings* :: $(classTable * varCtx * exp\ list * className\ list)$ *set*
**syntax**
*-typing* :: [*classTable*, *varCtx*, *exp list*, *className*] ⇒ *bool* (-;- ⊢ - : - [*80*,*80*,*80*,*80*] *80*)
*-typings* :: [*classTable*, *varCtx*, *exp list*, *className*] ⇒ *bool* (-;- ⊢+ - : - [*80*,*80*,*80*,*80*] *80*)
**translations**
$CT;\Gamma \vdash e : C \rightleftharpoons (CT,\Gamma,e,C) \in typing$
$CT;\Gamma \vdash+\ es : Cs \rightleftharpoons (CT,\Gamma,es,Cs) \in typings$

**inductive** *typings typing*
**intros**
*ts-nil* : $CT;\Gamma \vdash+ [] : []$

*ts-cons* :
$[\![$ $CT;\Gamma \vdash e0 : C0$; $CT;\Gamma \vdash+ es : Cs$ $]\!]$
$\Longrightarrow CT;\Gamma \vdash+ (e0\ \#\ es) : (C0\ \#\ Cs)$

*t-var* :
$[\![$ $\Gamma(x) = Some\ C$ $]\!] \Longrightarrow CT;\Gamma \vdash (Var\ x) : C$

*t-field* :
$[\![$ $CT;\Gamma \vdash e0 : C0$;
$fields(CT,C0) = Cf$;
$lookup\ Cf\ (\lambda fd.(vdName\ fd = fi)) = Some(fDef)$;

$\quad vdType\ fDef = Ci\ \rrbracket$
$\Longrightarrow CT;\Gamma \vdash FieldProj\ e0\ fi : Ci$

*t-invk* :
$\llbracket\ CT;\Gamma \vdash e0 : C0;$
$\quad mtype(CT,m,C0) = Ds \rightarrow C;$
$\quad CT;\Gamma \vdash+ es : Cs;$
$\quad CT \vdash+ Cs <: Ds;$
$\quad length\ es = length\ Ds\ \rrbracket$
$\Longrightarrow CT;\Gamma \vdash MethodInvk\ e0\ m\ es : C$

*t-new* :
$\llbracket\ fields(CT,C) = Df;$
$\quad length\ es = length\ Df;$
$\quad varDefs\text{-}types\ Df = Ds;$
$\quad CT;\Gamma \vdash+ es : Cs;$
$\quad CT \vdash+ Cs <: Ds\ \rrbracket$
$\Longrightarrow CT;\Gamma \vdash New\ C\ es : C$

*t-ucast* :
$\llbracket\ CT;\Gamma \vdash e0 : D;$
$\quad CT \vdash D <: C\ \rrbracket$
$\Longrightarrow CT;\Gamma \vdash Cast\ C\ e0 : C$

*t-dcast* :
$\llbracket\ CT;\Gamma \vdash e0 : D;$
$\quad CT \vdash C <: D;\ C \neq D\ \rrbracket$
$\Longrightarrow CT;\Gamma \vdash Cast\ C\ e0 : C$

*t-scast* :
$\llbracket\ CT;\Gamma \vdash e0 : D;$
$\quad CT \vdash C \neg\!<: D;$
$\quad CT \vdash D \neg\!<: C\ \rrbracket$
$\Longrightarrow CT;\Gamma \vdash Cast\ C\ e0 : C$

We occasionally find the following induction principle, which only mentions the typing of a single expression, more useful than the mutual induction principle generated by Isabelle, which mentions the typings of single expressions and of lists of expressions.

**lemma** *typing-induct*:
**assumes** $CT;\Gamma \vdash e : C$ (**is** $?T$)
**and** $\bigwedge C\ CT\ \Gamma\ x.\ \Gamma\ x = Some\ C \Longrightarrow P\ CT\ \Gamma\ (Var\ x)\ C$
**and** $\bigwedge C0\ CT\ Cf\ Ci\ \Gamma\ e0\ fDef\ fi.\ \llbracket CT;\Gamma \vdash e0 : C0;\ P\ CT\ \Gamma\ e0\ C0;\ (CT,\ C0,\ Cf) \in FJDefs.fields;\ lookup\ Cf\ (\lambda fd.\ vdName\ fd = fi) = Some\ fDef;\ vdType\ fDef = Ci\rrbracket \Longrightarrow P\ CT\ \Gamma\ (FieldProj\ e0\ fi)\ Ci$
**and** $\bigwedge C\ C0\ CT\ Cs\ Ds\ \Gamma\ e0\ es\ m.\ \llbracket CT;\Gamma \vdash e0 : C0;\ P\ CT\ \Gamma\ e0\ C0;\ (CT,\ m,\ C0,\ Ds,\ C) \in mtype;\ CT;\Gamma \vdash+ es : Cs;\ \bigwedge i\ .\ \llbracket\ i < length\ es\ \rrbracket \Longrightarrow\ P\ CT\ \Gamma\ (es!i)\ (Cs!i);\ CT \vdash+ Cs <: Ds;\ length\ es = length\ Ds\rrbracket \Longrightarrow P\ CT\ \Gamma\ (MethodInvk\ e0\ m\ es)\ C$

**and** $\bigwedge C\ CT\ Cs\ Df\ Ds\ \Gamma\ es.\ [\![(CT,\ C,\ Df) \in FJDefs.fields;\ length\ es = length\ Df;\ varDefs\text{-}types\ Df = Ds;\ CT;\Gamma \vdash+ es : Cs;\ \bigwedge i.\ [\![\ i < length\ es\ ]\!] \Longrightarrow P\ CT\ \Gamma\ (es!i)\ (Cs!i);\ CT \vdash+ Cs <: Ds]\!] \Longrightarrow P\ CT\ \Gamma\ (New\ C\ es)\ C$
**and** $\bigwedge C\ CT\ D\ \Gamma\ e0.\ [\![CT;\Gamma \vdash e0 : D;\ P\ CT\ \Gamma\ e0\ D;\ CT \vdash D <: C]\!] \Longrightarrow P\ CT\ \Gamma\ (Cast\ C\ e0)\ C$
**and** $\bigwedge C\ CT\ D\ \Gamma\ e0.\ [\![CT;\Gamma \vdash e0 : D;\ P\ CT\ \Gamma\ e0\ D;\ CT \vdash C <: D;\ C \neq D]\!] \Longrightarrow P\ CT\ \Gamma\ (Cast\ C\ e0)\ C$
**and** $\bigwedge C\ CT\ D\ \Gamma\ e0.\ [\![CT;\Gamma \vdash e0 : D;\ P\ CT\ \Gamma\ e0\ D;\ CT \vdash C \neg<: D;\ CT \vdash D \neg<: C]\!] \Longrightarrow P\ CT\ \Gamma\ (Cast\ C\ e0)\ C$
**shows** $P\ CT\ \Gamma\ e\ C$ (**is** *?P*)
**proof** −
**let** $?IH = CT;\Gamma \vdash+ es : Cs \longrightarrow (\forall i < length\ es.\ \ P\ CT\ \Gamma\ (es!i)\ (Cs!i))$
**have** $?IH \wedge (?T \longrightarrow ?P)$
**proof**(*induct rule*:*typings-typing.induct*)
**case** (*ts-nil* $CT\ \Gamma$) **show** *?case* **by** *auto*
**next**
**case** (*ts-cons* $C0\ CT\ Cs\ \Gamma\ e0\ es$)
**show** *?case* **proof**
**fix** $i$
**show** $i < length\ (e0\#es) \longrightarrow P\ CT\ \Gamma\ ((e0\#es)!i)\ ((C0\#Cs)!i)$ **using** *ts-cons* **by**(*cases* $i$, *auto*)
**qed**
**next**
**case**(*t-field* $C0\ CT\ Cf\ e0\ fDef\ fi$) **show** *?case* **using** *prems* **by** *auto*
**next**
**case**(*t-invk* $C\ C0\ CT\ Cs\ Ds\ \Gamma\ e0\ es\ m$) **show** *?case* **using** *prems* **by** *auto*
**next**
**case**(*t-new* $C\ CT\ D\ \Gamma\ e0$) **show** *?case* **using** *prems* **by** *auto*
**next**
**case**(*t-ucast* $C\ CT\ \Gamma\ e0$) **show** *?case* **using** *prems* **by** *auto*
**next**
**case**(*t-dcast* $C\ CT\ \Gamma\ e0$) **show** *?case* **using** *prems* **by** *auto*
**next**
**case**(*t-scast* $C\ CT\ \Gamma\ e0$) **show** *?case* **using** *prems* **by** *auto*
**qed**
**thus** *?thesis* **using** *prems* **by** *auto*
**qed**

## 1.12 Method Typing Relation

A method definition $md$, declared in a class $C$, is well-typed, written $CT \vdash md$`OK IN` $C$ if its body is well-typed and it has the same type (i.e., overrides) any method with the same name declared in the superclass of $C$.

**consts** *method-typing* :: (*classTable* ∗ *methodDef* ∗ *className*) *set*
*method-typings* :: (*classTable* ∗ *methodDef list* ∗ *className*) *set*
**syntax**
*-method-typing* :: [*classTable*, *methodDef*, *className*] ⇒ *bool* (*-* ⊢ *- OK IN -* [*80*,*80*,*80*] *80*)

$-method\text{-}typings :: [classTable, methodDef\ list, className] \Rightarrow bool\ (\text{-} \vdash\!+ \text{-}\ OK\ IN$
$\text{-}\ [80,80,80]\ 80)$

**translations**

$CT \vdash md\ OK\ IN\ C \rightleftharpoons (CT,md,C) \in method\text{-}typing$

$CT \vdash\!+ mds\ OK\ IN\ C \rightleftharpoons (CT,mds,C) \in method\text{-}typings$

**inductive** *method-typing*
**intros**
*m-typing*:

$$
\begin{array}{l}
[\![\ CT(C) = Some(CDef); \\
\quad cName\ CDef = C; \\
\quad cSuper\ CDef = D; \\
\quad mName\ mDef = m; \\
\quad lookup\ (cMethods\ CDef)\ (\lambda md.(mName\ md = m)) = Some(mDef); \\
\quad mReturn\ mDef = C0;\ mParams\ mDef = Cxs;\ mBody\ mDef = e0; \\
\quad varDefs\text{-}types\ Cxs = Cs; \\
\quad varDefs\text{-}names\ Cxs = xs; \\
\quad \Gamma = (map\text{-}upds\ empty\ xs\ Cs)(this \mapsto C); \\
\quad CT;\Gamma \vdash e0 : E0; \\
\quad CT \vdash E0 <: C0; \\
\quad \forall\, Ds\ D0.\ (mtype(CT,m,D) = Ds \rightarrow D0) \longrightarrow (Cs{=}Ds \wedge C0{=}D0)\ ]\!] \\
\Longrightarrow CT \vdash mDef\ OK\ IN\ C
\end{array}
$$

**inductive** *method-typings*
**intros**
*ms-nil* :

$CT \vdash\!+ []\ OK\ IN\ C$

*ms-cons* :

$$
\begin{array}{l}
[\![\ CT \vdash m\ OK\ IN\ C; \\
\quad CT \vdash\!+ ms\ OK\ IN\ C\ ]\!] \\
\Longrightarrow CT \vdash\!+ (m\ \#\ ms)\ OK\ IN\ C
\end{array}
$$

## 1.13 Class Typing Relation

A class definition $cd$ is well-typed, written $CT \vdash cd\texttt{OK}$ if its constructor initializes each field, and all of its methods are well-typed.

**consts** $class\text{-}typing :: (classTable * classDef)\ set$
**syntax**

$-class\text{-}typing :: [classTable, classDef] \Rightarrow bool\ (\text{-} \vdash \text{-}\ OK\ [80,80]\ 80)$

**translations**

$CT \vdash cd\ OK \rightleftharpoons (CT,cd) \in class\text{-}typing$

**inductive** *class-typing*
**intros**

$$
\begin{array}{l}
t\text{-}class\text{: } [\![\ cName\ CDef = C; \\
\qquad cSuper\ CDef = D; \\
\qquad cConstructor\ CDef = KDef; \\
\qquad cMethods\ CDef = M;
\end{array}
$$

$kName\ KDef = C$;
$kParams\ KDef = (Dg@Cf)$;
$kSuper\ KDef = varDefs\text{-}names\ Dg$;
$kInits\ KDef = varDefs\text{-}names\ Cf$;
$fields(CT,D) = Dg$;
$CT \vdash\!+ M\ OK\ IN\ C$ ⟧
$\Longrightarrow CT \vdash CDef\ OK$

## 1.14 Class Table Typing Relation

A class table is well-typed, written $CT$ `OK` if for every class name $C$, the class definition mapped to by $CT$ is is well-typed and has name $C$.

**consts** *ct-typing* :: *classTable set*
**syntax**
*-ct-typing* :: $classTable \Rightarrow bool$ (- $OK$ $80$)
**translations**
$CT\ OK \rightleftharpoons CT \in ct\text{-}typing$
**inductive** *ct-typing*
**intros**
*ct-all-ok*:
⟦ $Object \notin dom(CT)$;
$\forall\, C\ CDef.\ CT(C) = Some(CDef) \longrightarrow (CT \vdash CDef\ OK) \wedge (cName\ CDef = C)$ ⟧
$\Longrightarrow CT\ OK$

## 1.15 Evaluation Relation

The single-step and multi-step evaluation relations are written $CT \vdash e \rightarrow e'$ and $CT \vdash e \rightarrow^* e'$ respectively.

**consts** *reduction* :: $(classTable * exp * exp)$ *set*
**syntax**
*-reduction* :: $[classTable,\ exp,\ exp] \Rightarrow bool$ (- $\vdash$ - $\rightarrow$ - $[80,80,80]$ $80$)

**translations**
$CT \vdash e \rightarrow e' \rightleftharpoons (CT,e,e') \in reduction$
**inductive** *reduction*
**intros**

*r-field*:
⟦ $fields(CT,C) = Cf$;
$lookup2\ Cf\ es\ (\lambda fd.(vdName\ fd = fi)) = Some(ei)$ ⟧
$\Longrightarrow CT \vdash FieldProj\ (New\ C\ es)\ fi \rightarrow ei$

*r-invk*:
⟦ $mbody(CT,m,C) = xs\ .\ e0$;
$substs\ ((map\text{-}upds\ empty\ xs\ ds)(this \mapsto (New\ C\ es)))\ e0 = e0'$ ⟧
$\Longrightarrow CT \vdash MethodInvk\ (New\ C\ es)\ m\ ds \rightarrow e0'$

*r-cast*:
$[\![\ CT \vdash C <: D\ ]\!]$
$\Longrightarrow CT \vdash Cast\ D\ (New\ C\ es) \rightarrow New\ C\ es$

*rc-field*:
$[\![\ CT \vdash e0 \rightarrow e0'\ ]\!]$
$\Longrightarrow CT \vdash FieldProj\ e0\ f \rightarrow FieldProj\ e0'\ f$

*rc-invk-recv*:
$[\![\ CT \vdash e0 \rightarrow e0'\ ]\!]$
$\Longrightarrow CT \vdash MethodInvk\ e0\ m\ es \rightarrow MethodInvk\ e0'\ m\ es$

*rc-invk-arg*:
$[\![\ CT \vdash ei \rightarrow ei'\ ]\!]$
$\Longrightarrow CT \vdash MethodInvk\ e0\ m\ (el@ei\#er) \rightarrow MethodInvk\ e0\ m\ (el@ei'\#er)$

*rc-new-arg*:
$[\![\ CT \vdash ei \rightarrow ei'\ ]\!]$
$\Longrightarrow CT \vdash New\ C\ (el@ei\#er) \rightarrow New\ C\ (el@ei'\#er)$

*rc-cast*:
$[\![\ CT \vdash e0 \rightarrow e0'\ ]\!]$
$\Longrightarrow CT \vdash Cast\ C\ e0 \rightarrow Cast\ C\ e0'$

**consts** *reductions* :: $(classTable * exp * exp)\ set$
**syntax**
*-reductions* :: $[classTable, exp, exp] \Rightarrow bool$ (- $\vdash$ - $\rightarrow*$ - [80,80,80] 80)
**translations**
$CT \vdash e \rightarrow* e' \rightleftharpoons (CT,e,e') \in reductions$
**inductive** *reductions*
**intros**
*rs-refl*: $CT \vdash e \rightarrow* e$
*rs-trans*: $[\![\ CT \vdash e \rightarrow e';\ CT \vdash e' \rightarrow* e''\ ]\!] \Longrightarrow\ CT \vdash e \rightarrow* e''$

**end**

## 2 FJAux: Auxiliary Lemmas

**theory** *FJAux* **imports** *FJDefs*
**begin**

### 2.1 Non-FJ Lemmas

#### 2.1.1 Lists

**lemma** *mem-ith*:
**assumes** $ei \in set\ es$
**shows** $\exists\ el\ er.\ es = el@ei\#er$

**using** *prems*
**proof**(*induct es*)
**case** *Nil* **thus** *?case* **by** *auto*
**next**
**case** (*Cons esh est*)
**{ assume** $esh = ei$
**with** *Cons* **have** *?case* **by** *blast*
**} moreover {**
**assume** $esh \neq ei$
**with** *Cons* **have** $ei \in set\ est$ **by** *auto*
**with** *Cons* **obtain** *el er* **where** $esh \# est = (esh\#el) @ (ei\#er)$ **by** *auto*
**hence** *?case* **by** *blast* **}**
**ultimately show** *?case* **by** *blast*
**qed**

**lemma** *ith-mem*: $\bigwedge i.\ [\![\ i < length\ es\ ]\!] \Longrightarrow es!i \in set\ es$
**proof**(*induct es*)
**case** *Nil* **thus** *?case* **by** *auto*
**next**
**case** (*Cons h t*) **thus** *?case* **by**(*cases i*, *auto*)
**qed**

### 2.1.2 Maps

**lemma** *map-shuffle*:
**assumes** $length\ xs = length\ ys$
**shows** $[xs[\mapsto]ys,x\mapsto y] = [(xs@[x])[\mapsto](ys@[y])]$
**using** *prems*
**proof**(*induct xs ys rule*:*list-induct2*,
*auto simp add*:*map-upds-append1*)
**qed**

**lemma** *map-upds-index*:
**assumes** $length\ xs = length\ As$
**and** $[xs[\mapsto]As]x = Some\ Ai$
**shows** $\exists\, i.(As!i = Ai)$
$\wedge\ (i < length\ As)$
$\wedge\ (\forall\,(Bs::'c\ list).((length\ Bs = length\ As)$
$\longrightarrow ([xs[\mapsto]Bs]\ x = Some\ (Bs\ !i))))$
(**is** $\exists\, i.\ ?P\ i\ xs\ As$
**is** $\exists\, i.(?P1\ i\ As) \wedge (?P2\ i\ As) \wedge (\forall\, Bs::('c\ list).(?P3\ i\ xs\ As\ Bs)))$
**using** *prems* **proof**(*induct xs As rule*:*list-induct2*)
**assume** $[[][\mapsto][]]\ x = Some\ Ai$
**moreover have** $\neg[[][\mapsto][]]\ x = Some\ Ai$ **by** *auto*
**ultimately show** $\exists\, i.\ ?P\ i\ []\ []$ **by** *contradiction*
**next**
**fix** *xa xs y ys*
**assume** *length-xs-ys*: $length\ xs = length\ ys$
**and** *IH*: $[xs\ [\mapsto]\ ys]\ x = Some\ Ai \Longrightarrow \exists\, i.\ ?P\ i\ xs\ ys$

**and** *map-eq-Some*: $[xa \ \# \ xs \ [\mapsto] \ y \ \# \ ys] \ x = Some \ Ai$
**from** *prems* **have** *map-decomp*: $[xa\#xs \ [\mapsto] \ y\#ys] = [xa\mapsto y] \ {+}{+} \ [xs[\mapsto]ys]$ **by** *fastsimp*
**from** *length-xs-ys IH map-eq-Some* **show** $\exists\, i.\ ?P \ i \ (xa\#xs) \ (y \ \# \ ys)$
**proof**(*cases* $[xs[\mapsto]ys]x$)
  **case**(*Some Ai'*)
  **hence** $([xa \mapsto y] \ {+}{+} \ [xs[\mapsto]ys]) \ x = Some \ Ai'$ **by**(*rule map-add-find-right*)
  **hence** *P*: $[xs[\mapsto]ys] \ x = Some \ Ai$ **using** *prems* **by** *simp*
  **from** *IH*[*OF P*] **obtain** *i* **where**
    *R1*: $ys \ ! \ i = Ai$
    **and** *R2*: $i < length \ ys$
    **and** *pre-r3*: $\forall (Bs{::}'c \ list).\ ?P3 \ i \ xs \ ys \ Bs$ **by** *fastsimp*
  **{** **fix** $Bs{::}'c \ list$
    **assume** *length-Bs*: $length \ Bs = length \ (y\#ys)$
    **then obtain** *n* **where** $length \ (y\#ys) = Suc \ n$ **by** *auto*
    **with** *length-Bs* **obtain** *b bs* **where** *Bs-def*: $Bs = b\#bs$ **by** (*auto simp add*:*length-Suc-conv*)
    **with** *length-Bs* **have** $length \ ys = length \ bs$ **by** *simp*
    **with** *pre-r3* **have** $([xa\mapsto b] \ {+}{+} \ [xs[\mapsto]bs]) \ x = Some \ (bs!i)$ **by**(*auto simp only*:*map-add-find-right*)
    **with** *pre-r3 Bs-def length-Bs* **have** $?P3 \ (i{+}1) \ (xa\#xs) \ (y\#ys) \ Bs$ **by** *simp* **}**
  **with** *R1 R2* **have** $?P \ (i{+}1) \ (xa\#xs) \ (y\#ys)$ **by** *auto*
  **thus** *?thesis* **..**
**next**
  **case** *None*
  **with** *map-decomp* **have** $[xa\mapsto y] \ x = Some \ Ai$ **using** *prems* **by** (*auto simp only*:*map-add-SomeD*)
  **hence** *ai-def*: $y = Ai$ **and** *x-eq-xa*:$x{=}xa$ **by** (*auto simp only*:*map-upd-Some-unfold*)

  **{** **fix** $Bs{::}'c \ list$
    **assume** *length-Bs*: $length \ Bs = length \ (y\#ys)$
    **then obtain** *n* **where** $length \ (y\#ys) = Suc \ n$ **by** *auto*
    **with** *length-Bs* **obtain** *b bs* **where** *Bs-def*: $Bs = b\#bs$ **by** (*auto simp add*:*length-Suc-conv*)
    **with** *length-Bs* **have** $length \ ys = length \ bs$ **by** *simp*
    **hence** $dom([xs[\mapsto]ys]) = dom([xs[\mapsto]bs])$ **by** *auto*
    **with** *None* **have** $[xs[\mapsto]bs] \ x = None$ **by** (*auto simp only*:*domIff*)
    **moreover from** *x-eq-xa* **have** *sing-map*: $[xa\mapsto b] \ x = Some \ b$ **by** (*auto simp only*:*map-upd-Some-unfold*)
    **ultimately have** $([xa\mapsto b] \ {+}{+} \ [xs[\mapsto]bs]) \ x = Some \ b$ **by** (*auto simp only*:*map-add-Some-iff*)
    **with** *Bs-def* **have** $?P3 \ 0 \ (xa\#xs) \ (y\#ys) \ Bs$ **by** *simp* **}**
  **with** *ai-def* **have** $?P \ 0 \ (xa\#xs) \ (y\#ys)$ **by** *auto*
  **thus** *?thesis* **..**
**qed**
**qed**

## 2.2 FJ Lemmas

### 2.2.1 Substitution

**lemma** *subst-list1-eq-map-substs* :
  $\forall\, \sigma.\ subst\text{-}list1 \ \sigma \ l = map \ (substs \ \sigma) \ l$

**by** (*induct l*, *simp-all*)

**lemma** *subst-list2-eq-map-substs* :
$\forall \sigma$. *subst-list2* $\sigma$ *l* = *map* (*substs* $\sigma$) *l*
**by** (*induct l*, *simp-all*)

### 2.2.2 Lookup

**lemma** *lookup-functional*:
**assumes** *lookup l f* = *o1*
**and** *lookup l f* = *o2*
**shows** *o1* = *o2*
**using** *prems* **by**(*induct l*, *auto*)

**lemma** *lookup-true*:
*lookup l f* = *Some r* $\Longrightarrow$ *f r*
**proof**(*induct l*)
**case** *Nil* **thus** *?case* **by** *simp*
**next**
**case**(*Cons h t*) **thus** *?case* **by**(*cases f h*, *auto simp add*:*lookup.simps*)
**qed**

**lemma** *lookup-hd*:
$[\![$ *length l* > *0*; *f* (*l*!*0*) $]\!]$ $\Longrightarrow$ *lookup l f* = *Some* (*l*!*0*)
**proof**(*induct l*, *auto*)
**qed**

**lemma** *lookup-split*: *lookup l f* = *None* $\vee$ ($\exists$ *h*. *lookup l f* = *Some h*)
**by** (*induct l*, *simp-all*)

**lemma** *lookup-index*:
**assumes** *lookup l1 f* = *Some e*
**shows** $\bigwedge$*l2*. $\exists$ *i* < (*length l1*). *e* = *l1*!*i* $\wedge$ ((*length l1* = *length l2*) $\longrightarrow$ *lookup2 l1 l2 f* = *Some* (*l2*!*i*))
**using** *prems*
**proof**(*induct l1*)
**case** *Nil* **thus** *?case* **by** *auto*
**next**
**case** (*Cons h1 t1*)
**{ assume** *asm*:*f h1*
**hence** *0*<*length* (*h1* # *t1*) $\wedge$ *e* = (*h1* # *t1*) ! *0*
**using** *prems* **by** (*auto simp add*:*lookup.simps*)
**moreover {**
**assume** *length* (*h1* # *t1*) = *length l2*
**hence** *length l2* = *Suc* (*length t1*) **by** *auto*
**then obtain** *h2 t2* **where** *l2-def*:*l2* = *h2*#*t2* **by** (*auto simp add*: *length-Suc-conv*)
**hence** *lookup2* (*h1* # *t1*) *l2 f* = *Some* (*l2* ! *0*) **using** *asm* **by**(*auto simp*: *add lookup2.simps*)
**}**

```
    ultimately have ?case by auto
  } moreover {
    assume asm:¬ (f h1)
    hence lookup t1 f = Some e
      using prems by (auto simp add:lookup.simps)
    then obtain i where
      i<length t1
      and e = t1 ! i
      and  ih:(length t1 = length (tl l2) ⟶ lookup2 t1 (tl l2) f = Some ((tl l2) !
i))
      using prems by blast
    hence Suc i < length (h1#t1) ∧ e = (h1#t1)!(Suc i) using prems by auto
    moreover {
      assume length (h1 # t1) = length l2
      hence lens:length l2 = Suc (length t1) by auto
     then obtain h2 t2 where l2-def:l2 = h2#t2 by (auto simp add: length-Suc-conv)
      hence lookup2 t1 t2 f = Some (t2 ! i) using ih l2-def lens by auto
      hence lookup2 (h1 # t1) l2 f = Some (l2!(Suc i))
        using asm l2-def by(auto simp: add lookup2.simps)
    }
    ultimately have ?case by auto
  }
  ultimately show ?case by auto
qed

lemma lookup2-index:
  ⋀l2. ⟦ lookup2 l1 l2 f = Some e;
  length l1 = length l2 ⟧ ⟹ ∃ i < (length l2). e = (l2!i) ∧ lookup l1 f = Some
(l1!i)
proof(induct l1)
  case Nil thus ?case by auto
next
  case (Cons h1 t1)
  hence length l2 = Suc (length t1) using prems by auto
 then obtain h2 t2 where l2-def:l2 = h2#t2 by (auto simp add: length-Suc-conv)
  { assume asm:f h1
    hence e = h2 using prems by (auto simp add:lookup2.simps)
     hence 0<length (h2#t2) ∧ e = (h2#t2) ! 0 ∧ lookup (h1 # t1) f = Some
((h1 # t1) ! 0)
      using asm by (auto simp add:lookup.simps)
    hence ?case using l2-def by auto
  } moreover {
    assume asm:¬ (f h1)
     hence ∃ i<length t2. e = t2 ! i ∧ lookup t1 f = Some (t1 ! i) using prems
l2-def by auto
     then obtain i where i<length t2 ∧  e = t2 ! i ∧ lookup t1 f = Some (t1 !
i) by auto
     hence (Suc i) < length(h2#t2) ∧ e = ((h2#t2) ! (Suc i)) ∧ lookup (h1#t1)
f = Some ((h1#t1) ! (Suc i))
```

    **using** *asm* **by** (*force simp add*: *lookup.simps*)
  **hence** *?case* **using** *l2-def* **by** *auto*
 **}**
 **ultimately show** *?case* **by** *auto*
**qed**

**lemma** *lookup-append*:
 **assumes** *lookup l f* = *Some r*
 **shows** *lookup* (*l*@*l′*) *f* = *Some r*
 **using** *prems* **by**(*induct l*, *auto*)

**lemma** *method-typings-lookup*:
 **assumes** *lookup-eq-Some*: *lookup M f* = *Some mDef*
 **and** *M-ok*: *CT* ⊢+ *M OK IN C*
 **shows** *CT* ⊢ *mDef OK IN C*
 **using** *lookup-eq-Some M-ok*
**proof**(*induct M*)
 **case** *Nil* **thus** *?case* **by** *fastsimp*
**next**
 **case** (*Cons h t*) **thus** *?case* **by**(*cases f h*, *auto elim*:*method-typings.elims simp add*:*lookup.simps*)
**qed**

### 2.2.3 Functional

These lemmas prove that several relations are actually functions

**lemma** *mtype-functional*:
 **assumes** *mtype*(*CT*,*m*,*C*) = *Cs* → *C0*
 **and** *mtype*(*CT*,*m*,*C*) = *Ds* → *D0*
 **shows** *Ds*=*Cs* ∧ *D0*=*C0*
**using** *prems* **by**(*induct*, *auto elim*:*mtype.elims*)

**lemma** *mbody-functional*:
 **assumes** *mb1*: *mbody*(*CT*,*m*,*C*) = *xs* . *e0*
 **and** *mb2*: *mbody*(*CT*,*m*,*C*) = *ys* . *d0*
 **shows** *xs* = *ys* ∧ *e0* = *d0*
**using** *prems* **by**(*induct*, *auto elim*:*mbody.elims*)

**lemma** *fields-functional*:
 **assumes** *fields*(*CT*,*C*) = *Cf*
 **and** *CT OK*
 **shows** ⋀*Cf′*. ⟦ *fields*(*CT*,*C*) = *Cf′*⟧ ⟹ *Cf* = *Cf′*
**using** *prems* **proof**(*induct*)
 **case** (*f-obj CT*)
 **hence** *CT*(*Object*) = *None* **by** (*auto elim*: *ct-typing.elims*)
 **thus** *?case* **using** *f-obj* **by** (*auto elim*: *fields.elims*)
**next**
 **case** (*f-class C CDef CT Cf D Dg DgCf DgCf′*)
 **hence** *f-class-inv*:

$(CT\ C = Some\ CDef) \wedge (cSuper\ CDef = D) \wedge (cFields\ CDef = Cf)$
**and** $CT\ OK$ **by** *fastsimp*
**hence** *c-not-obj*:$C \neq Object$ **by** (*force elim*:*ct-typing.elims*)
**from** *f-class* **have** *flds*:$fields(CT,C) = DgCf'$ **by** *fastsimp*
**then obtain** $Dg'$ **where**
$fields(CT,D) = Dg'$
**and** $DgCf' = Dg'$ @ $Cf$
**using** *f-class-inv c-not-obj* **by** (*auto elim*:*fields.elims*)
**hence** $Dg' = Dg$ **using** *f-class* **by** *auto*
**thus** *?case* **using** *prems* **by** *force*
**qed**

### 2.2.4 Subtyping and Typing

**lemma** *typings-lengths*: **assumes** $CT;\Gamma \vdash+ es{:}Cs$ **shows** *length es = length Cs*
**using** *prems* **by**(*induct es Cs*, *auto elim*:*typings-typing.elims*)

**lemma** *typings-index*:
**assumes** $CT;\Gamma \vdash+ es{:}Cs$
**shows** $\bigwedge i.\ [\![\ i < length\ es\ ]\!] \Longrightarrow CT;\Gamma \vdash (es!i) : (Cs!i)$
**proof** −
**have** *length es = length Cs* **using** *prems* **by** (*auto simp*: *typings-lengths*)
**thus** $\bigwedge i.\ [\![\ i < length\ es\ ]\!] \Longrightarrow CT;\Gamma \vdash (es!i) : (Cs!i)$
**using** *prems* **proof**(*induct es Cs rule*:*list-induct2*)
**case** *1* **thus** *?case* **by** *auto*
**next**
**case** (*2 esh est hCs tCs i*)
**thus** *?case* **by**(*cases i*, *auto elim*:*typings-typing.elims*)
**qed**
**qed**

**lemma** *subtypings-index*:
**assumes** $CT \vdash+ Cs <: Ds$
**shows** $\bigwedge i.\ [\![\ i < length\ Cs\ ]\!] \Longrightarrow CT \vdash (Cs!i) <: (Ds!i)$
**using** *prems* **proof**(*induct*)
**case** *ss-nil* **thus** *?case* **by** *auto*
**next**
**case** (*ss-cons hCs CT tCs hDs tDs i*)
**thus** *?case* **by**(*cases i*, *auto*)
**qed**

**lemma** *subtyping-append*:
**assumes** $CT \vdash+ Cs <: Ds$
**and** $CT \vdash C <: D$
**shows** $CT \vdash+ (Cs@[C]) <: (Ds@[D])$
**using** *prems*
**proof**(*induct rule*:*subtypings.induct*,
*auto simp add*:*subtypings.intros elim*:*subtypings.elims*)

**qed**

**lemma** *typings-append*:
**assumes** $CT;\Gamma \vdash\!+ \ es : Cs$
**and** $CT;\Gamma \vdash e : C$
**shows** $CT;\Gamma \vdash\!+ \ (es@[e]) : (Cs@[C])$
**proof** $-$
**have** *length es* = *length Cs* **using** *prems* **by**(*simp-all add*:*typings-lengths*)
**thus** $CT;\Gamma \vdash\!+ \ (es@[e]) : (Cs@[C])$ **using** *prems*
**proof**(*induct es Cs rule*:*list-induct2*)
**have** $CT;\Gamma \vdash\!+ \ []:[]$ **by**(*simp add*:*typings-typing.ts-nil*)
**moreover from** *prems* **have** $CT;\Gamma \vdash e : C$ **by** *simp*
**ultimately show** $CT;\Gamma \vdash\!+ \ ([]@[e]) : ([]@[C])$ **by** (*auto simp add*:*typings-typing.ts-cons*)
**next**
**fix** *x xs y ys*
**assume** *length xs* = *length ys*
**and** *IH*: $[\![CT;\Gamma \vdash\!+ \ xs : ys;\ CT;\Gamma \vdash e : C]\!] \Longrightarrow CT;\Gamma \vdash\!+ \ (xs \ @ \ [e]) : (ys \ @$
$[C])$
**and** *x-xs-typs*: $CT;\Gamma \vdash\!+ \ (x \ \# \ xs) : (y \ \# \ ys)$
**and** *e-typ*: $CT;\Gamma \vdash e : C$
**from** *x-xs-typs* **have** *x-typ*: $CT;\Gamma \vdash x : y$ **and** $CT;\Gamma \vdash\!+ \ xs : ys$ **by**(*auto*
*elim*:*typings-typing.elims*)
**with** *IH e-typ* **have** $CT;\Gamma \vdash\!+ \ (xs@[e]) : (ys@[C])$ **by** *simp*
**with** *x-typ* **have** $CT;\Gamma \vdash\!+ \ ((x\#xs)@[e]) : ((y\#ys)@[C])$ **by** (*auto simp*
*add*:*typings-typing.ts-cons*)
**thus** $CT;\Gamma \vdash\!+ \ ((x \ \# \ xs) \ @ \ [e]) : ((y \ \# \ ys) \ @ \ [C])$ **by**(*auto simp add*:*typings-typing.ts-cons*)
**qed**
**qed**

**lemma** *ith-typing*: $\bigwedge Cs.\ [\![\ CT;\Gamma \vdash\!+ \ (es@(h\#t)) : Cs\ ]\!] \Longrightarrow CT;\Gamma \vdash h : (Cs!(length$
$es))$
**proof**(*induct es*, *auto elim*:*typings-typing.elims*)
**qed**

**lemma** *ith-subtyping*: $\bigwedge Ds.\ [\![\ CT \vdash\!+ \ (Cs@(h\#t)) <: Ds\ ]\!] \Longrightarrow CT \vdash h <:$
$(Ds!(length\ Cs))$
**proof**(*induct Cs*, *auto elim*:*subtypings.elims*)
**qed**

**lemma** *subtypings-refl*: $CT \vdash\!+ \ Cs <: Cs$
**by**(*induct Cs*, *auto simp add*: *subtyping.s-refl subtypings.intros*)

**lemma** *subtypings-trans*: $\bigwedge Ds\ Es.\ [\![\ CT \vdash\!+ \ Cs <: Ds;\ \ CT \vdash\!+ \ Ds <: Es\ ]\!] \Longrightarrow$
$CT \vdash\!+ \ Cs <: Es$
**proof**(*induct Cs*)
**case** *Nil* **thus** *?case*
**by** (*auto elim*:*subtypings.elims simp add*:*subtypings.ss-nil*)
**next**
**case** (*Cons hCs tCs*)

**then obtain** *hDs tDs*
**where** $h1{:}CT \vdash hCs <: hDs$ **and** $t1{:}CT \vdash{+}\ tCs <: tDs$ **and** $Ds = hDs\#tDs$
**by** (*auto elim*:*subtypings*.*elims*)
**then obtain** *hEs tEs*
**where** $h2{:}CT \vdash hDs <: hEs$ **and** $t2{:}CT \vdash{+}\ tDs <: tEs$ **and** $Es = hEs\#tEs$
**using** *Cons* **by** (*auto elim*:*subtypings*.*elims*)
**moreover from** *subtyping*.*s-trans*[*OF h1 h2*] **have** $CT \vdash hCs <: hEs$ **by** *fastsimp*
**moreover with** *t1 t2* **have** $CT \vdash{+}\ tCs <: tEs$ **using** *Cons* **by** *simp-all*
**ultimately show** *?case* **by** (*auto simp add*:*subtypings*.*intros*)
**qed**

**lemma** *ith-typing-sub*:
$\bigwedge Cs.\ \llbracket\ CT;\Gamma \vdash{+}\ (es@(h\#t)) : Cs;$
$CT;\Gamma \vdash h' : Ci';$
$CT \vdash Ci' <: (Cs!(length\ es))\ \rrbracket$
$\Longrightarrow \exists\, Cs'.\ (CT;\Gamma \vdash{+}\ (es@(h'\#t)) : Cs' \wedge CT \vdash{+}\ Cs' <: Cs)$
**proof**(*induct es*)
**case** *Nil*
**then obtain** *hCs tCs*
**where** *ts*: $CT;\Gamma \vdash{+}\ t : tCs$
**and** *Cs-def*: $Cs = hCs\ \#\ tCs$ **by**(*auto elim*:*typings-typing*.*elims*)
**from** *Cs-def Nil* **have** $CT \vdash Ci' <: hCs$ **by** *auto*
**with** *Cs-def* **have** $CT \vdash{+}\ (Ci'\#tCs) <: Cs$ **by**(*auto simp add*:*subtypings*.*ss-cons*
*subtypings-refl*)
**moreover from** *ts Nil* **have** $CT;\Gamma \vdash{+}\ (h'\#t) : (Ci'\#tCs)$ **by**(*auto simp add*:*typings-typing*.*ts-cons*)
**ultimately show** *?case* **by** *auto*
**next**
**case** (*Cons eh et*)
**then obtain** *hCs tCs*
**where** $CT;\Gamma \vdash eh : hCs$
**and** $CT;\Gamma \vdash{+}\ (et@(h\#t)) : tCs$
**and** *Cs-def*: $Cs = hCs\ \#\ tCs$
**by**(*auto elim*:*typings-typing*.*elims*)
**moreover with** *Cons* **obtain** $tCs'$
**where** $CT;\Gamma \vdash{+}\ (et@(h'\#t)) : tCs'$
**and** $CT \vdash{+}\ tCs' <: tCs$
**by** *auto*
**ultimately have**
$CT;\Gamma \vdash{+}\ (eh\#(et@(h'\#t))) : (hCs\#tCs')$
**and** $CT \vdash{+}\ (hCs\#tCs') <: Cs$
**by**(*auto simp add*:*typings-typing*.*ts-cons subtypings*.*ss-cons subtyping*.*s-refl*)
**thus** *?case* **by** *auto*
**qed**

**lemma** *mem-typings*:
$\bigwedge Cs.\ \llbracket\ CT;\Gamma \vdash{+}\ es{:}Cs;\ ei \in set\ es\rrbracket \Longrightarrow \exists\, Ci.\ CT;\Gamma \vdash ei{:}Ci$
**proof**(*induct es*)
**case** *Nil* **thus** *?case* **by** *auto*
**next**

**case** (*Cons eh et*) **thus** *?case*
**by**(*cases ei=eh*, *auto elim*:*typings-typing.elims*)
**qed**

**lemma** *typings-proj*:
**assumes** $CT;\Gamma \vdash+ ds : As$
**and** $CT \vdash+ As <: Bs$
**and** *length ds* = *length As*
**and** *length ds* = *length Bs*
**and** $i <$ *length ds*
**shows** $CT;\Gamma \vdash ds!i : As!i$ **and** $CT \vdash As!i <: Bs!i$
**proof** −
**show** $CT;\Gamma \vdash ds!i : As!i$ **and** $CT \vdash As!i <: Bs!i$
**using** *prems* **by** (*auto simp add*:*typings-index subtypings-index*)
**qed**

**lemma** *subtypings-length*:
$CT \vdash+ As <: Bs \Longrightarrow$ *length As* = *length Bs*
**by**(*induct rule*:*subtypings.induct*,*simp-all*)

**lemma** *not-subtypes-aux*:
**assumes** $CT \vdash C <: Da$
**and** $C \neq Da$
**and** *CT C* = *Some CDef*
**and** *cSuper CDef* = *D*
**shows** $CT \vdash D <: Da$
**using** *prems*
**proof**(*induct rule*:*subtyping.induct*, *auto intro*:*subtyping.intros*)
**qed**

**lemma** *not-subtypes*:
**assumes** $CT \vdash A <: C$
**shows** $\bigwedge D.\ [\![\ CT \vdash D \neg\!<: C;\ \ CT \vdash C \neg\!<: D]\!] \Longrightarrow CT \vdash A \neg\!<: D$
**using** *prems*
**proof**(*induct rule*:*subtyping.induct*)
**case** *s-refl* **thus** *?case* **by** *auto*
**next**
**case** (*s-trans C CT D E Da*)
**have** *da-nsub-d*:$CT \vdash Da \neg\!<: D$ **proof**(*rule ccontr*)
**assume** $\neg\ CT \vdash Da \neg\!<: D$
**hence** *da-sub-d*:$CT \vdash Da <: D$ **by** *auto* **pr**
**have** *d-sub-e*:$CT \vdash D <: E$ **using** *prems* **by** *fastsimp*
**thus** *False* **using** *prems* **by** (*force simp add*:*subtyping.s-trans*[*OF da-sub-d d-sub-e*])
**qed**
**have** *d-nsub-da*:$CT \vdash D \neg\!<: Da$ **using** *s-trans* **by** *auto*
**from** *da-nsub-d d-nsub-da s-trans* **show** $CT \vdash C \neg\!<: Da$ **by** *auto*
**next**
**case** (*s-super C CDef CT D Da*)

**have** $C \neq Da$ **proof**(*rule ccontr*)
**assume** $\neg\ C \neq Da$
**hence** $C = Da$ **by** *auto*
**hence** $CT \vdash Da <: D$ **using** *prems* **by**(*auto simp add*: *subtyping.s-super*)
**thus** *False* **using** *prems* **by** *auto*
**qed**
**thus** *?case* **using** *prems* **by** (*auto simp add*: *not-subtypes-aux*)
**qed**

### 2.2.5 Sub-Expressions

**lemma** *isubexpr-typing*:
**assumes** $e1 \in isubexprs(e0)$
**shows** $\bigwedge C.\ [\![\ CT;empty \vdash e0 : C\ ]\!] \Longrightarrow \exists D.\ CT;empty \vdash e1 : D$
**using** *prems*
**proof**(*induct rule*:*isubexprs.induct*, *auto elim*:*typings-typing.elims simp add*:*mem-typings*)
**qed**

**lemma** *subexpr-typing*:
**assumes** $e1 \in subexprs(e0)$
**shows** $\bigwedge C.\ [\![\ CT;empty \vdash e0 : C\ ]\!] \Longrightarrow \exists D.\ CT;empty \vdash e1 : D$
**using** *prems*
**by**(*induct rule*:*rtrancl.induct*,*auto*,*force simp add*:*isubexpr-typing*)

**lemma** *isubexpr-reduct*:
**assumes** $d1 \in isubexprs(e1)$
**shows** $\bigwedge d2.\ [\![\ CT \vdash d1 \rightarrow d2\ ]\!] \Longrightarrow \exists e2.\ CT \vdash e1 \rightarrow e2$
**using** *prems mem-ith*
**proof**(*induct*,
*auto elim*:*isubexprs.elims intro*:*reduction.intros*,
*force intro*:*reduction.intros*,
*force intro*:*reduction.intros*)
**qed**

**lemma** *subexpr-reduct*:
**assumes** $d1 \in subexprs(e1)$
**shows** $\bigwedge d2.\ [\![\ CT \vdash d1 \rightarrow d2\ ]\!] \Longrightarrow \exists e2.\ CT \vdash e1 \rightarrow e2$
**using** *prems*
**proof**(*induct rule*:*rtrancl.induct*,
*auto*, *force simp add*: *isubexpr-reduct*)
**qed**

**end**

# 3 FJSound: Type Soundness

**theory** *FJSound* **imports** *FJAux*

**begin**

Type soundness is proved using the standard technique of progress and subject reduction. The numbered lemmas and theorems in this section correspond to the same results in the ACM TOPLAS paper.

## 3.1 Method Type and Body Connection

**lemma** *mtype-mbody*:
  **assumes** $mtype(CT,m,C) = Cs \rightarrow C0$
  **shows** $\exists xs\ e.\ mbody(CT,m,C) = xs\ .\ e \wedge length\ xs = length\ Cs$
  **using** *prems*
  **proof**(*induct rule*:*mtype.induct*)
    **case**(*mt-class C0 Cs C CDef CT m mDef*)
    **thus** *?case*
      **by** (*force simp add*:*varDefs-types-def varDefs-names-def elim*:*mtype.elims intro*:*mbody.mb-class*)
  **next**
    **case**(*mt-super C0 Cs C CDef CT D m*)
    **then obtain** *xs e* **where** $mbody(CT,m,D) = xs\ .\ e$ **and** $length\ xs = length\ Cs$ **by** *auto*
    **thus** *?case* **using** *mt-super* **by** (*auto intro*:*mbody.mb-super*)
  **qed**

**lemma** *mtype-mbody-length*:
  **assumes** $mt{:}mtype(CT,m,C) = Cs \rightarrow C0$
  **and** $mb{:}mbody(CT,m,C) = xs\ .\ e$
  **shows** $length\ xs = length\ Cs$
**proof** −
  **from** *mtype-mbody*[*OF mt*] **obtain** $xs'\ e'$
    **where** $mb2{:}\ mbody(CT,m,C) = xs'\ .\ e'$
    **and** $length\ xs' = length\ Cs$
    **by** *auto*
  **with** *mbody-functional*[*OF mb mb2*] **show** *?thesis* **by** *auto*
**qed**

## 3.2 Method Types and Field Declarations of Subtypes

**lemma** *A-1-1*:
  **assumes** $CT \vdash C <: D$ **and** *CT OK*
  **shows** $(mtype(CT,m,D) = Cs \rightarrow C0) \Longrightarrow (mtype(CT,m,C) = Cs \rightarrow C0)$
  **using** *prems* **proof** (*induct rule*:*subtyping.induct*)
  **case** (*s-refl C CT*) **show** *?case* **by** *assumption*
  **next**
  **case** (*s-trans C CT D E*) **thus** *?case* **by** *auto*
  **next**
  **case** (*s-super C CDef CT D*)
  **hence** $CT \vdash CDef\ OK$ **and** $cName\ CDef = C$
    **by**(*auto elim*:*ct-typing.elims*)

```
  with s-super obtain M
    where CT ⊢+ M OK IN C and cMethods CDef = M
    by(auto elim:class-typing.elims)
  let ?lookup-m = lookup M (λmd. (mName md =m))
  show ?case using prems
  proof(cases ∃ mDef. ?lookup-m = Some mDef)
  case True
    then obtain mDef where ?lookup-m = Some mDef by(rule exE)
    hence mDef-name: mName mDef = m by (rule lookup-true)
   have CT ⊢ mDef OK IN C using prems by(auto simp add:method-typings-lookup)
    then obtain CDef' m' D' Cs' C0'
      where CT C = Some CDef'
        and cSuper CDef' = D'
        and mName mDef = m'
        and mReturn mDef = C0'
        and varDefs-types (mParams mDef) = Cs'
        and ∀ Ds D0. (mtype(CT,m',D') = Ds → D0) ⟶ Cs'=Ds ∧ C0'=D0
      by (auto elim: method-typing.elims)
   with s-super mDef-name have
        CDef=CDef'
     and D=D'
     and m=m'
     and ∀ Ds D0. (mtype(CT,m,D) = Ds → D0) ⟶ Cs'=Ds ∧ C0' = D0
     using prems by auto
   thus ?thesis using prems by (auto intro:mtype.intros)
  next
  case False
   hence ?lookup-m = None by (simp add: lookup-split)
   show ?thesis using prems by (auto simp add:mtype.intros)
  qed
qed


lemma sub-fields:
  assumes CT ⊢ C <: D
  shows ⋀Dg. fields(CT,D) = Dg ⟹ ∃ Cf. fields(CT,C) = (Dg@Cf)
using prems proof(induct)
  case (s-refl C CT)
  hence fields(CT,C) = (Dg@[]) by simp
  thus ?case ..
next
  case (s-trans C CT D E)
  then obtain Df Cf where fields(CT,C) = ((Dg@Df)@Cf) by force
  thus ?case by auto
next
  case (s-super C CDef CT D Dg)
  then obtain Cf where cFields CDef = Cf by force
  with s-super have fields(CT,C) = (Dg@Cf) by(simp add:f-class)
  thus ?case ..
```

**qed**

## 3.3 Substitution Lemma

**lemma** *A-1-2*:
  **assumes** *CT OK*
  **and** $\Gamma = \Gamma 1$ ++ $\Gamma 2$
  **and** $\Gamma 2 = [xs\ [\mapsto]\ Bs]$
  **and** *length xs* = *length ds*
  **and** *length Bs* = *length ds*
  **and** $\exists As.\ CT;\Gamma 1 \vdash\!+\ ds : As \wedge CT \vdash\!+\ As <: Bs$
  **shows** $CT;\Gamma \vdash\!+\ es{:}Ds \Longrightarrow \exists Cs.\ (CT;\Gamma 1 \vdash\!+\ ([ds/xs]es){:}Cs \wedge CT \vdash\!+\ Cs <: Ds)$ (**is** *?TYPINGS* $\Longrightarrow$ *?P1*)
  **and** $CT;\Gamma \vdash e{:}D \Longrightarrow \exists C.\ (CT;\Gamma 1 \vdash ((ds/xs)e){:}C \wedge CT \vdash C <: D)$ (**is** *?TYPING* $\Longrightarrow$ *?P2*)
**proof** −
  **let** *?COMMON-ASMS* = $(CT\ OK) \wedge (\Gamma = \Gamma 1$ ++ $\Gamma 2) \wedge (\Gamma 2 = [xs\ [\mapsto]\ Bs]) \wedge (length\ Bs = length\ ds) \wedge (\exists As.\ CT;\Gamma 1 \vdash\!+\ ds : As \wedge CT \vdash\!+\ As <: Bs)$
  **have** *RESULT*: (*?TYPINGS* $\longrightarrow$ *?COMMON-ASMS* $\longrightarrow$ *?P1*)
       $\wedge$ (*?TYPING* $\longrightarrow$ *?COMMON-ASMS* $\longrightarrow$ *?P2*)
  **proof**(*induct rule*:*typings-typing.induct*)
    **case** (*ts-nil CT* $\Gamma$)
    **show** *?case*
    **proof** (*rule impI*)
      **have** $(CT;\Gamma 1 \vdash\!+\ ([ds/xs][]){:}[]) \wedge (CT \vdash\!+\ [] <: [])$
        **by** (*auto simp add*: *typings-typing.intros subtypings.intros*)
      **from** *this* **show** $\exists Cs.(CT;\Gamma 1 \vdash\!+\ ([ds/xs][]){:}Cs) \wedge (CT \vdash\!+\ Cs <: [])$ **by** *auto*
    **qed**
  **next**
    **case**(*ts-cons C0 CT Cs′* $\Gamma$ *e0 es*)
    **show** *?case*
    **proof** (*rule impI*)
      **assume** *asms*: $(CT\ OK) \wedge (\Gamma = \Gamma 1$ ++ $\Gamma 2) \wedge (\Gamma 2 = [xs\ [\mapsto]\ Bs]) \wedge (length\ Bs = length\ ds) \wedge (\exists As.\ CT;\Gamma 1 \vdash\!+\ ds : As \wedge CT \vdash\!+\ As <: Bs)$
      **with** *ts-cons* **have** *e0-typ*: $CT;\Gamma \vdash e0 : C0$ **by** *fastsimp*
      **with** *ts-cons asms* **have**
        $\exists C.(CT;\Gamma 1 \vdash (ds/xs)\ e0 : C) \wedge (CT \vdash C <: C0)$
        **and** $\exists Cs.(CT;\Gamma 1 \vdash\!+\ [ds/xs]es : Cs) \wedge (CT \vdash\!+\ Cs <: Cs')$
        **by** *auto*
      **then obtain** *C Cs* **where**
        $(CT;\Gamma 1 \vdash (ds/xs)\ e0 : C) \wedge (CT \vdash C <: C0)$
        **and** $(CT;\Gamma 1 \vdash\!+\ [ds/xs]es : Cs) \wedge (CT \vdash\!+\ Cs <: Cs')$ **by** *auto*
      **hence** $CT;\Gamma 1 \vdash\!+\ [ds/xs](e0\#es) : (C\#Cs)$
        **and** $CT \vdash\!+\ (C\#Cs) <: (C0\#Cs')$
        **by** (*auto simp add*: *typings-typing.intros subtypings.intros*)
      **then show** $\exists Cs.\ CT;\Gamma 1 \vdash\!+\ map\ (substs\ [xs\ [\mapsto]\ ds])\ (e0\ \#\ es) : Cs \wedge CT \vdash\!+\ Cs <: (C0\ \#\ Cs')$ **by** *auto*
    **qed**
  **next**

**case** ($t\text{-}var\ C'\ CT\ \Gamma\ x$)
**show** *?case*
**proof** (*rule impI*)
**assume** *asms*: $(CT\ OK) \wedge (\Gamma = \Gamma 1 ++ \Gamma 2) \wedge (\Gamma 2 = [xs\ [\mapsto]\ Bs]) \wedge (length\ Bs = length\ ds) \wedge (\exists As.\ CT;\Gamma 1 \vdash + ds : As \wedge CT \vdash + As <: Bs)$
**hence**
*lengths*: $length\ ds = length\ Bs$
**and** *G-def*: $\Gamma = \Gamma 1 ++ \Gamma 2$
**and** *G2-def* : $\Gamma 2 = [xs[\mapsto]Bs]$ **by** *auto*
**from** *lengths G2-def* **have** *same-doms*: $dom([xs[\mapsto]ds]) = dom(\Gamma 2)$ **by** *auto*
**from** *asms* **show** $\exists C.\ CT;\Gamma 1 \vdash substs\ [xs\ [\mapsto]\ ds]\ (Var\ x) : C \wedge CT \vdash C <: C'$
**proof** (*cases* $\Gamma 2\ x$)
**case** *None*
**with** *G-def t-var* **have** *G1-x*: $\Gamma 1\ x = Some\ C'$ **by**(*simp add*:*map-add-Some-iff*)
**from** *None same-doms* **have** $x \notin dom([xs[\mapsto]ds])$ **by** (*auto simp only*:*domIff*)

**hence** $[xs[\mapsto]ds]x = None$ **by**(*auto simp only*:*map-add-Some-iff*)
**hence** $(ds/xs)(Var\ x) = (Var\ x)$ **by** *auto*
**with** *G1-x* **have**
$CT;\Gamma 1 \vdash (ds/xs)(Var\ x) : C'$ **and** $CT \vdash C' <: C'$
**by** (*auto simp add*:*typings-typing.intros subtyping.intros*)
**thus** *?thesis* **by** *auto*
**next**
**case** (*Some Bi*)
**with** *G-def t-var* **have** *c′-eq-bi*: $C' = Bi$ **by** (*auto simp add*: *map-add-SomeD*)
**from** *prems* **have** $length\ xs = length\ Bs$ **by** *simp*
**with** *Some G2-def* **have** $\exists i.(Bs!i = Bi) \wedge (i < length\ Bs) \wedge (\forall l.((length\ l = length\ Bs) \longrightarrow ([xs[\mapsto]l]\ x = Some\ (l!i))))$
**by** (*auto simp add*: *map-upds-index*)
**then obtain** $i$ **where**
*bs-i-proj*:$(Bs!i = Bi)$
**and** *i-len*:$i < length\ Bs$
**and** $P$:$(\bigwedge(l::exp\ list).((length\ l = length\ Bs) \longrightarrow ([xs[\mapsto]l]\ x = Some\ (l!i))))$
**by** *fastsimp*
**from** *lengths P* **have** *subst-x*:$([xs[\mapsto]ds]x = Some\ (ds!i))$ **by** *auto*
**from** *prems* **obtain** *As* **where** *as-ex*:$CT;\Gamma 1 \vdash + ds : As \wedge CT \vdash + As <: Bs$ **by** *fastsimp*
**hence** $length\ As = length\ Bs$ **by** (*auto simp add*: *subtypings-length*)
**hence** *proj-i*:$CT;\Gamma 1 \vdash ds!i : As!i \wedge CT \vdash As!i <: Bs!i$ **using** *i-len lengths as-ex* **by** (*auto simp add*: *typings-proj*)
**hence** $CT;\Gamma 1 \vdash (ds/xs)(Var\ x) : As!i \wedge CT \vdash As!i <: C'$ **using** *c′-eq-bi bs-i-proj subst-x* **by** *auto*
**thus** *?thesis* **..**
**qed**
**qed**
**next**
**case**(*t-field C0 CT Cf Ci* $\Gamma$ *e0 fDef fi*)

**show** *?case*
**proof**(*rule impI*)
**assume** *asms*: $(CT\ OK) \wedge (\Gamma = \Gamma 1 ++ \Gamma 2) \wedge (\Gamma 2 = [xs\ [\mapsto]\ Bs]) \wedge (length\ Bs = length\ ds) \wedge (\exists As.\ CT;\Gamma 1 \vdash+ ds : As \wedge CT \vdash+ As <: Bs)$
**from** *t-field* **have** *flds*: $fields(CT,C0) = Cf$ **by** *fastsimp*
**from** *prems* **obtain** $C$ **where** *e0-typ*: $CT;\Gamma 1 \vdash (ds/xs)e0 : C$ **and** *sub*: $CT \vdash C <: C0$ **by** *auto*
**from** *sub-fields*[*OF sub flds*] **obtain** $Dg$ **where** *flds-C*: $fields(CT,C) = (Cf@Dg)$ **..**
**from** *t-field* **have** *lookup-CfDg*: $lookup\ (Cf@Dg)\ (\lambda fd.\ vdName\ fd = fi) = Some\ fDef$ **by**(*simp add:lookup-append*)
**from** *e0-typ flds-C lookup-CfDg t-field* **have** $CT;\Gamma 1 \vdash (ds/xs)(FieldProj\ e0\ fi) : Ci$ **by**(*simp add:typings-typing.intros*)
**moreover have** $CT \vdash Ci <: Ci$ **by** (*simp add:subtyping.intros*)
**ultimately show** $\exists C.\ CT;\Gamma 1 \vdash (ds/xs)(FieldProj\ e0\ fi) : C \wedge CT \vdash C <: Ci$ **by** *auto*
**qed**
**next**
**case**(*t-invk C C0 CT Cs Ds* $\Gamma$ *e0 es m*)
**show** *?case*
**proof**(*rule impI*)
**assume** *asms*: $(CT\ OK) \wedge (\Gamma = \Gamma 1 ++ \Gamma 2) \wedge (\Gamma 2 = [xs\ [\mapsto]\ Bs]) \wedge (length\ Bs = length\ ds) \wedge (\exists As.\ CT;\Gamma 1 \vdash+ ds : As \wedge CT \vdash+ As <: Bs)$
**hence** *ct-ok*: $CT\ OK$ **..**
**from** *t-invk* **have** *mtyp*: $mtype(CT,m,C0) = Ds \rightarrow C$
**and** *subs*: $CT \vdash+ Cs <: Ds$
**and** *lens*: $length\ es = length\ Ds$
**by** *auto*
**from** *prems* **obtain** $C'$ **where** *e0-typ*: $CT;\Gamma 1 \vdash (ds/xs)e0 : C'$ **and** *sub'*: $CT \vdash C' <: C0$ **by** *auto*
**from** *prems* **obtain** $Cs'$ **where** *es-typ*: $CT;\Gamma 1 \vdash+ [ds/xs]es : Cs'$ **and** *subs'*: $CT \vdash+ Cs' <: Cs$ **by** *auto*
**have** *subst-e*: $(ds/xs)(MethodInvk\ e0\ m\ es) = MethodInvk\ ((ds/xs)e0)\ m\ ([ds/xs]es)$
**by**(*auto simp add:substs-subst-list1-subst-list2.simps subst-list1-eq-map-substs*)
**from**
*e0-typ*
*A-1-1*[*OF sub' ct-ok mtyp*]
*es-typ*
*subtypings-trans*[*OF subs' subs*]
*lens*
*subst-e*
**have** $CT;\Gamma 1 \vdash (ds/xs)(MethodInvk\ e0\ m\ es) : C$ **by**(*auto simp add:typings-typing.intros*)
**moreover have** $CT \vdash C <: C$ **by**(*simp add:subtyping.intros*)
**ultimately show** $\exists C'.\ CT;\Gamma 1 \vdash (ds/xs)(MethodInvk\ e0\ m\ es) : C' \wedge CT \vdash C' <: C$ **by** *auto*
**qed**
**next**
**case**(*t-new C CT Cs Df Ds* $\Gamma$ *es*)

**show** *?case*
**proof**(*rule impI*)
**assume** *asms*: $(CT\ OK) \wedge (\Gamma = \Gamma 1\ {+}{+}\ \Gamma 2) \wedge (\Gamma 2 = [xs\ [\mapsto]\ Bs]) \wedge (length\ Bs = length\ ds) \wedge (\exists As.\ CT;\Gamma 1 \vdash + ds : As \wedge CT \vdash + As <: Bs)$
**hence** *ct-ok*: $CT\ OK$ **..**
**from** *t-new* **have**
*subs*: $CT \vdash + Cs <: Ds$
**and** *flds*: $fields(CT,C) = Df$
**and** *len*: $length\ es = length\ Df$
**and** *vdts*: $varDefs\text{-}types\ Df = Ds$
**by** *auto*
**from** *prems* **obtain** $Cs'$ **where** *es-typ*: $CT;\Gamma 1 \vdash + [ds/xs]es : Cs'$ **and** *subs'*: $CT \vdash + Cs' <: Cs$ **by** *auto*
**have** *subst-e*: $(ds/xs)(New\ C\ es) = New\ C\ ([ds/xs]es)$
**by**(*auto simp add:substs-subst-list1-subst-list2.simps subst-list2-eq-map-substs*)
**from** *es-typ subtypings-trans*[*OF subs' subs*] *flds subst-e len vdts*
**have** $CT;\Gamma 1 \vdash (ds/xs)(New\ C\ es) : C$ **by**(*auto simp add:typings-typing.intros*)
**moreover have** $CT \vdash C <: C$ **by**(*simp add:subtyping.intros*)
**ultimately show** $\exists C'.\ CT;\Gamma 1 \vdash (ds/xs)(New\ C\ es) : C' \wedge CT \vdash C' <: C$ **by** *auto*
**qed**
**next**
**case**(*t-ucast C CT D* $\Gamma$ *e0*)
**show** *?case*
**proof**(*rule impI*)
**assume** *asms*: $(CT\ OK) \wedge (\Gamma = \Gamma 1\ {+}{+}\ \Gamma 2) \wedge (\Gamma 2 = [xs\ [\mapsto]\ Bs]) \wedge (length\ Bs = length\ ds) \wedge (\exists As.\ CT;\Gamma 1 \vdash + ds : As \wedge CT \vdash + As <: Bs)$
**from** *prems* **obtain** $C'$ **where** *e0-typ*: $CT;\Gamma 1 \vdash (ds/xs)e0 : C'$
**and** *sub1*: $CT \vdash C' <: D$
**and** *sub2*: $CT \vdash D <: C$ **by** *auto*
**from** *sub1 sub2* **have** $CT \vdash C' <: C$ **by** (*rule s-trans*)
**with** *e0-typ* **have** $CT;\Gamma 1 \vdash (ds/xs)(Cast\ C\ e0) : C$ **by**(*auto simp add: typings-typing.intros*)
**moreover have** $CT \vdash C <: C$ **by** (*rule s-refl*)
**ultimately show** $\exists C'.\ CT;\Gamma 1 \vdash (ds/xs)(Cast\ C\ e0) : C' \wedge CT \vdash C' <: C$ **by** *auto*
**qed**
**next**
**case**(*t-dcast C CT D* $\Gamma$ *e0*)
**show** *?case*
**proof**(*rule impI*)
**assume** *asms*: $(CT\ OK) \wedge (\Gamma = \Gamma 1\ {+}{+}\ \Gamma 2) \wedge (\Gamma 2 = [xs\ [\mapsto]\ Bs]) \wedge (length\ Bs = length\ ds) \wedge (\exists As.\ CT;\Gamma 1 \vdash + ds : As \wedge CT \vdash + As <: Bs)$
**from** *prems* **obtain** $C'$ **where** *e0-typ*: $CT;\Gamma 1 \vdash (ds/xs)e0 : C'$ **by** *auto*
**have** $(CT \vdash C' <: C) \vee$
$(C \neq C' \wedge CT \vdash C <: C') \vee$
$(CT \vdash C \neg{<:}\ C' \wedge CT \vdash C' \neg{<:}\ C)$ **by** *blast*
**moreover**
**{ assume** $CT \vdash C' <: C$

**with** *e0-typ* **have** $CT;\Gamma 1 \vdash (ds/xs)\ (Cast\ C\ e0) : C$ **by** (*auto simp add*: *typings-typing.intros*)
}
**moreover**
{ **assume** $(C \neq C' \wedge CT \vdash C <: C')$
**with** *e0-typ* **have** $CT;\Gamma 1 \vdash (ds/xs)\ (Cast\ C\ e0) : C$ **by** (*auto simp add*: *typings-typing.intros*)
}
**moreover**
{ **assume** $(CT \vdash C \neg\!<: C' \wedge CT \vdash C' \neg\!<: C)$
**with** *e0-typ* **have** $CT;\Gamma 1 \vdash (ds/xs)\ (Cast\ C\ e0) : C$ **by** (*auto simp add*: *typings-typing.intros*)
}
**ultimately have** $CT;\Gamma 1 \vdash (ds/xs)\ (Cast\ C\ e0) : C$ **by** *auto*
**moreover have** $CT \vdash C <: C$ **by**(*rule s-refl*)
**ultimately show** $\exists C'.\ CT;\Gamma 1 \vdash (ds/xs)(Cast\ C\ e0) : C' \wedge CT \vdash C' <: C$ **by** *auto*
**qed**
**next**
**case**(*t-scast* $C\ CT\ D\ \Gamma\ e0$)
**show** *?case*
**proof**(*rule impI*)
**assume** *asms*: $(CT\ OK) \wedge (\Gamma = \Gamma 1 ++ \Gamma 2) \wedge (\Gamma 2 = [xs\ [\mapsto]\ Bs]) \wedge (length\ Bs = length\ ds) \wedge (\exists As.\ CT;\Gamma 1 \vdash\!+\ ds : As \wedge CT \vdash\!+\ As <: Bs)$
**from** *prems* **obtain** $C'$ **where** *e0-typ*:$CT;\Gamma 1 \vdash (ds/xs)e0 : C'$
**and** *sub1*: $CT \vdash C' <: D$
**and** *nsub1*: $CT \vdash C \neg\!<: D$
**and** *nsub2*: $CT \vdash D \neg\!<: C$ **by** *auto*
**from** *not-subtypes*[*OF sub1 nsub1 nsub2*] **have** $CT \vdash C' \neg\!<: C$ **by** *fastsimp*
**moreover have** $CT \vdash C \neg\!<: C'$ **proof**(*rule ccontr*)
**assume** $\neg\ CT \vdash C \neg\!<: C'$
**hence** $CT \vdash C <: C'$ **by** *auto*
**hence** $CT \vdash C <: D$ **using** *sub1* **by**(*rule s-trans*)
**with** *nsub1* **show** *False* **by** *auto*
**qed**
**ultimately have** $CT;\Gamma 1 \vdash (ds/xs)\ (Cast\ C\ e0) : C$ **using** *e0-typ* **by** (*auto simp add*: *typings-typing.intros*)
**thus** $\exists C'.\ CT;\Gamma 1 \vdash (ds/xs)(Cast\ C\ e0) : C' \wedge CT \vdash C' <: C$ **by** (*auto simp add*: *s-refl*)
**qed**
**qed**
**thus** *?TYPINGS* $\Longrightarrow$ *?P1* **and** *?TYPING* $\Longrightarrow$ *?P2* **using** *prems* **by** *auto*
**qed**

## 3.4 Weakening Lemma

This lemma is not in the same form as in TOPLAS, but rather as we need it in subject reduction

**lemma** *A-1-3*:

**shows** $(CT;\Gamma 2 \vdash+ es : Cs) \Longrightarrow (CT;\Gamma 1{+}{+}\Gamma 2 \vdash+ es : Cs)$ (**is** $?P1 \Longrightarrow ?P2$)
**and** $CT;\Gamma 2 \vdash e : C \Longrightarrow CT;\Gamma 1{+}{+}\Gamma 2 \vdash e : C$ (**is** $?Q1 \Longrightarrow ?Q2$)
**proof** −
**have** $(?P1 \longrightarrow ?P2) \wedge (?Q1 \longrightarrow ?Q2)$
**by**(*induct rule*:*typings-typing.induct*, *auto simp add*: *map-add-find-right typings-typing.intros*)

**thus** $?P1 \Longrightarrow ?P2$ **and** $?Q1 \Longrightarrow ?Q2$ **by** *auto*
**qed**

## 3.5 Method Body Typing Lemma

**lemma** *A-1-4*:
**assumes** *ct-ok*: *CT OK*
**and** *mb*:$mbody(CT,m,C) = xs\ .\ e$
**and** *mt*:$mtype(CT,m,C) = Ds \rightarrow D$
**shows** $\exists D0\ C0.\ (CT \vdash C <: D0) \wedge$
$(CT \vdash C0 <: D) \wedge$
$(CT;[xs[\mapsto]Ds](this \mapsto D0) \vdash e : C0)$
**using** *mb ct-ok mt* **proof**(*induct rule*: *mbody.induct*)
**case** (*mb-class C CDef CT e m mDef xs*)
**hence**
*m-param*:*varDefs-types* (*mParams mDef*) = *Ds*
**and** *m-ret*:*mReturn mDef* = *D*
**and** $CT \vdash CDef\ OK$
**and** *cName CDef* = *C*
**by** (*auto elim*:*mtype.elims ct-typing.elims*)
**hence** $CT \vdash+ (cMethods\ CDef)\ OK\ IN\ C$ **by** (*auto elim*:*class-typing.elims*)
**hence** $CT \vdash mDef\ OK\ IN\ C$ **using** *mb-class* **by**(*auto simp add*:*method-typings-lookup*)
**hence** $\exists\ E0.\ ((CT;[xs[\mapsto]Ds,this{\mapsto}C] \vdash e : E0) \wedge (CT \vdash E0 <: D))$
**using** *mb-class m-param m-ret* **by**(*auto elim*:*method-typing.elims*)
**then obtain** *E0*
**where** $CT;[xs[\mapsto]Ds,this{\mapsto}C] \vdash e : E0$
**and** $CT \vdash E0 <: D$
**and** $CT \vdash C <: C$ **by** (*auto simp add*: *s-refl*)
**thus** *?case* **by** *blast*
**next**
**case** (*mb-super C CDef CT Da e m xs*)
**hence** *ct*: *CT OK*
**and** *IH*: $[\![CT\ OK;\ (CT,\ m,\ Da,\ Ds,\ D) \in mtype]\!]$
$\Longrightarrow \exists D0\ C0.\ (CT \vdash Da <: D0) \wedge (CT \vdash C0 <: D)$
$\wedge\ (CT;[xs\ [\mapsto]\ Ds,\ this \mapsto D0] \vdash e{:}C0)$ **by** *fastsimp*
**from** *mb-super* **have** *c-sub-da*: $CT \vdash C <: Da$ **by** (*auto simp add*:*s-super*)
**from** *mb-super* **have** *mt*:$mtype(CT,m,Da) = Ds \rightarrow D$ **by** (*auto elim*: *mtype.elims*)
**from** *IH*[*OF ct mt*] **obtain** *D0 C0*
**where** *s1*: $CT \vdash Da <: D0$
**and** $CT \vdash C0 <: D$
**and** $CT;[xs\ [\mapsto]\ Ds,\ this \mapsto D0] \vdash e : C0$ **by** *auto*
**thus** *?case* **using** *s-trans*[*OF c-sub-da s1*] **by** *blast*
**qed**

### 3.6 Subject Reduction Theorem

**theorem** *Thm-2-4-1*:
  **assumes** $CT \vdash e \rightarrow e'$
  **and** $CT$ $OK$
  **shows** $\bigwedge C.$ $[\![$ $CT;\Gamma \vdash e : C$ $]\!]$
  $\Longrightarrow \exists\, C'.$ $(CT;\Gamma \vdash e' : C' \wedge CT \vdash C' <: C)$
  **using** *prems* **proof**(*induct rule*: *reduction.induct*)
  **case** (*r-field Ca CT Cf e' es fi*)
  **hence** $CT;\Gamma \vdash$ *FieldProj* (*New Ca es*) *fi* : $C$
    **and** *ct-ok*: *CT OK*
    **and** *flds*: *fields*(*CT*,*Ca*) = *Cf*
    **and** *lkup2*: *lookup2 Cf es* ($\lambda$*fd. vdName fd* = *fi*) = *Some e'* **by** *fastsimp*
  **then obtain** *Ca' Cf' fDef*
    **where** *new-typ*: $CT;\Gamma \vdash$ *New Ca es* : *Ca'*
    **and** *flds'*:*fields*(*CT*,*Ca'*) = *Cf'*
    **and** *lkup*: *lookup Cf'* ($\lambda$*fd. vdName fd* = *fi*) = *Some fDef*
    **and** *C-def*: *vdType fDef* = *C* **by** (*auto elim*: *typings-typing.elims*)
  **hence** *Ca-Ca'*: *Ca* = *Ca'* **by** (*auto elim*:*typings-typing.elims*)
  **with** *flds'* **have** *Cf-Cf'*: *Cf* = *Cf'* **by**(*simp add*:*fields-functional*[*OF flds ct-ok*])
  **from** *new-typ* **obtain** *Cs Ds Cf''*
    **where** *fields*(*CT*,*Ca'*) = *Cf''*
    **and** *es-typs*: $CT;\Gamma \vdash+$ *es*:*Cs*
    **and** *Ds-def*: *varDefs-types Cf''* = *Ds*
    **and** *length-Cf-es*: *length Cf''* = *length es*
    **and** *subs*: $CT \vdash+ Cs <: Ds$
    **by**(*auto elim*:*typings-typing.elims*)
  **with** *Ca-Ca'* **have** *Cf-Cf''*: *Cf* = *Cf''* **by**(*auto simp add*:*fields-functional*[*OF flds ct-ok*])
  **from** *length-Cf-es Cf-Cf'' lookup2-index*[*OF lkup2*] **obtain** *i* **where**
    *i-bound*: *i* < *length es*
    **and** *e'* = *es*!*i*
    **and** *lookup Cf* ($\lambda$*fd. vdName fd* = *fi*) = *Some* (*Cf*!*i*) **by** *auto*
  **moreover with** *C-def Ds-def lkup lkup2* **have** *Ds*!*i* = *C* **using** *Ca-Ca' Cf-Cf' Cf-Cf'' i-bound length-Cf-es flds'*
    **by** (*auto simp add*:*nth-map varDefs-types-def fields-functional*[*OF flds ct-ok*])
  **moreover with** *subs es-typs* **have**
    $CT;\Gamma \vdash (es!i){:}(Cs!i)$ **and** $CT \vdash (Cs!i) <: (Ds!i)$ **using** *i-bound*
    **by**(*auto simp add*:*typings-index subtypings-index typings-lengths*)
  **ultimately show** *?case* **by** *auto*
**next**
  **case**(*r-invk Ca CT ds e e' es m xs*)
  **from** *r-invk* **have** *mb*: *mbody*(*CT*,*m*,*Ca*) = *xs* . *e* **by** *fastsimp*
  **from** *r-invk* **obtain** *Ca' Ds Cs*
    **where** $CT;\Gamma \vdash$ *New Ca es* : *Ca'*
    **and** *mtype*(*CT*,*m*,*Ca'*) = $Cs \rightarrow C$
    **and** *ds-typs*: $CT;\Gamma \vdash+$ *ds* : *Ds*
    **and** *Ds-subs*: $CT \vdash+ Ds <: Cs$
    **and** *l1*: *length ds* = *length Cs* **by**(*auto elim*:*typings-typing.elims*)
  **hence** *new-typ*: $CT;\Gamma \vdash$ *New Ca es* : *Ca*

**and** *mt*: $mtype(CT,m,Ca) = Cs \rightarrow C$ **by** (*auto elim*:*typings-typing.elims*)
**from** *ds-typs new-typ* **have** $CT;\Gamma \vdash+ (ds\ @[New\ Ca\ es]) : (Ds\ @[Ca])$ **by** (*simp add*:*typings-append*)
**moreover from** *A-1-4*[*OF - mb mt*] *r-invk* **obtain** *Da E*
**where** $CT \vdash Ca <: Da$
**and** *E-sub-C*: $CT \vdash E <: C$
**and** *e0-typ1*: $CT;[xs[\mapsto]Cs,this\mapsto Da] \vdash e : E$ **by** *auto*
**moreover with** *Ds-subs* **have** $CT \vdash+ (Ds@[Ca]) <: (Cs@[Da])$ **by**(*auto simp add*:*subtyping-append*)
**ultimately have** *ex*: $\exists\, As.\ CT;\Gamma \vdash+ (ds\ @[New\ Ca\ es]) : As \wedge CT \vdash+ As <: (Cs@[Da])$ **by** *auto*
**from** *e0-typ1* **have** *e0-typ2*: $CT;(\Gamma\ ++\ [xs[\mapsto]Cs,this\mapsto Da]) \vdash e : E$ **by**(*simp only*:*A-1-3*)
**from** *e0-typ2 mtype-mbody-length*[*OF mt mb*] **have** *e0-typ3*: $CT;(\Gamma\ ++\ [(xs@[this])[\mapsto](Cs@[Da])]) \vdash e : E$ **by**(*force simp only*:*map-shuffle*)
**let** $?\Gamma 1 = \Gamma$ **and** $?\Gamma 2 = [(xs@[this])[\mapsto](Cs@[Da])]$
**have** *g-def*: $(?\Gamma 1\ ++\ ?\Gamma 2) = (?\Gamma 1\ ++\ ?\Gamma 2)$ **and** *g2-def*: $?\Gamma 2 = ?\Gamma 2$ **by** *auto*
**from** *A-1-2*[*OF - g-def g2-def - - ex*] *e0-typ3 r-invk l1 mtype-mbody-length*[*OF mt mb*] **obtain** $E'$
**where** *e′-typ*: $CT;\Gamma \vdash substs\ [(xs@[this])[\mapsto](ds@[New\ Ca\ es])]\ e : E'$
**and** *E′-sub-E*: $CT \vdash E' <: E$ **by** *force*
**moreover from** *e′-typ l1 mtype-mbody-length*[*OF mt mb*] **have** $CT;\Gamma \vdash substs\ [xs[\mapsto]ds,this\mapsto(New\ Ca\ es)]\ e : E'$ **by**(*auto simp only*:*map-shuffle*)
**moreover from** *E′-sub-E E-sub-C* **have** $CT \vdash E' <: C$ **by** (*rule subtyping.s-trans*)
**ultimately show** *?case* **using** *r-invk* **by** *auto*
**next**
**case** (*r-cast Ca CT D es*)
**then obtain** $Ca'$
**where** $C = D$
**and** $CT;\Gamma \vdash New\ Ca\ es : Ca'$ **by** (*auto elim*: *typings-typing.elims*)
**thus** *?case* **using** *r-cast* **by** (*auto elim*: *typings-typing.elims*)
**next**
**case** (*rc-field CT e0 e0′ f*)
**then obtain** *C0 Cf fd*
**where** $CT;\Gamma \vdash e0 : C0$
**and** *Cf-def*: $fields(CT,C0) = Cf$
**and** *fd-def*:$lookup\ Cf\ (\lambda fd.\ (vdName\ fd = f)) = Some\ fd$
**and** $vdType\ fd = C$
**by** (*auto elim*:*typings-typing.elims*)
**moreover with** *rc-field* **obtain** $C'$
**where** $CT;\Gamma \vdash e0' : C'$
**and** $CT \vdash C' <: C0$ **by** *auto*
**moreover from** *sub-fields*[*OF - Cf-def*] **obtain** $Cf'$
**where** $fields(CT,C') = (Cf@Cf')$ **..**
**moreover with** *fd-def* **have** $lookup\ (Cf@Cf')\ (\lambda fd.\ (vdName\ fd = f)) = Some\ fd$
**by**(*simp add*:*lookup-append*)
**ultimately have** $CT;\Gamma \vdash FieldProj\ e0'\ f : C$ **by**(*auto simp add*:*typings-typing.t-field*)

```
  thus ?case by (auto simp add:subtyping.s-refl)
next
  case (rc-invk-recv CT e0 e0' es m C)
  then obtain C0 Ds Cs
    where ct-ok:CT OK
    and CT;Γ ⊢ e0 : C0
    and mt:mtype(CT,m,C0) = Ds → C
    and CT;Γ ⊢+ es : Cs
    and length es = length Ds
    and CT ⊢+ Cs <: Ds
    by (auto elim:typings-typing.elims)
  moreover with rc-invk-recv obtain C0'
    where CT;Γ ⊢ e0' : C0'
    and CT ⊢ C0' <: C0 by auto
  moreover with A-1-1[OF - ct-ok mt] have mtype(CT,m,C0') = Ds → C by
simp
  ultimately have CT;Γ ⊢ MethodInvk e0' m es : C by(auto simp add:typings-typing.t-invk)
  thus ?case by (auto simp add:subtyping.s-refl)
next
  case (rc-invk-arg CT e0 ei ei' el er m C)
  then obtain Cs Ds C0
    where typs: CT;Γ ⊢+ (el@(ei#er)) : Cs
    and e0-typ: CT;Γ ⊢ e0 : C0
    and mt: mtype(CT,m,C0) = Ds → C
    and Cs-sub-Ds: CT ⊢+ Cs <: Ds
    and len: length (el@(ei#er)) = length Ds
    by(auto elim:typings-typing.elims)
  hence CT;Γ ⊢ ei:(Cs!(length el)) by (simp add:ith-typing)
  with rc-invk-arg obtain Ci'
    where ei-typ: CT;Γ ⊢ ei':Ci'
    and Ci-sub: CT ⊢ Ci' <: (Cs!(length el))
    by auto
  from ith-typing-sub[OF typs ei-typ Ci-sub] obtain Cs'
    where es'-typs: CT;Γ ⊢+ (el@(ei'#er)) : Cs'
    and Cs'-sub-Cs: CT ⊢+ Cs' <: Cs by auto
  from len have length (el@(ei'#er)) = length Ds by simp
  with es'-typs subtypings-trans[OF Cs'-sub-Cs Cs-sub-Ds]  e0-typ mt have
    CT;Γ ⊢ MethodInvk e0 m (el@(ei'#er)) : C
    by(auto simp add:typings-typing.t-invk)
  thus ?case by (auto simp add:subtyping.s-refl)
next
  case (rc-new-arg Ca CT ei ei' el er C)
  then obtain Cs Df Ds
    where typs: CT;Γ ⊢+ (el@(ei#er)) : Cs
    and flds: fields(CT,C) = Df
    and len: length (el@(ei#er)) = length Df
    and Ds-def: varDefs-types Df = Ds
    and Cs-sub-Ds: CT ⊢+ Cs <: Ds
    and C-def: Ca = C
```

  **by**(*auto elim*:*typings-typing.elims*)
 **hence** $CT;\Gamma \vdash ei{:}(Cs!(length\ el))$ **by** (*simp add*:*ith-typing*)
 **with** *rc-new-arg* **obtain** $Ci'$
  **where** *ei-typ*: $CT;\Gamma \vdash ei'{:}Ci'$
  **and** *Ci-sub*: $CT \vdash Ci' <: (Cs!(length\ el))$
  **by** *auto*
 **from** *ith-typing-sub*[*OF typs ei-typ Ci-sub*] **obtain** $Cs'$
  **where** *es′-typs*: $CT;\Gamma \vdash+ (el@(ei'\#er)) : Cs'$
  **and** *Cs′-sub-Cs*: $CT \vdash+ Cs' <: Cs$ **by** *auto*
 **from** *len* **have** $length\ (el@(ei'\#er)) = length\ Df$ **by** *simp*
 **with** *es′-typs subtypings-trans*[*OF Cs′-sub-Cs Cs-sub-Ds*] *flds Ds-def C-def* **have**
  $CT;\Gamma \vdash New\ Ca\ (el@(ei'\#er)) : C$
  **by**(*auto simp add*:*typings-typing.t-new*)
 **thus** *?case* **by** (*auto simp add*:*subtyping.s-refl*)
**next**
 **case** (*rc-cast C CT e0 e0′ Ca*)
 **then obtain** $D$
  **where** $CT;\Gamma \vdash e0 : D$
  **and** *Ca-def*: $Ca = C$
  **by**(*auto elim*:*typings-typing.elims*)
 **with** *rc-cast* **obtain** $D'$
  **where** *e0′-typ*: $CT;\Gamma \vdash e0'{:}D'$ **and** $CT \vdash D' <: D$
  **by** *auto*
 **have** $(CT \vdash D' <: C) \lor$
  $(C \neq D' \land CT \vdash C <: D') \lor$
  $(CT \vdash C \neg{<:}\ D' \land CT \vdash D' \neg{<:}\ C)$ **by** *blast*
 **moreover {**
  **assume** $CT \vdash D' <: C$
  **with** *e0′-typ* **have** $CT;\Gamma \vdash Cast\ C\ e0' : C$ **by** (*auto simp add*: *typings-typing.t-ucast*)
 **} moreover {**
  **assume** $(C \neq D' \land CT \vdash C <: D')$
  **with** *e0′-typ* **have** $CT;\Gamma \vdash Cast\ C\ e0' : C$ **by** (*auto simp add*: *typings-typing.t-dcast*)
 **} moreover {**
  **assume** $(CT \vdash C \neg{<:}\ D' \land CT \vdash D' \neg{<:}\ C)$
  **with** *e0′-typ* **have** $CT;\Gamma \vdash Cast\ C\ e0' : C$ **by** (*auto simp add*: *typings-typing.t-scast*)
 **} ultimately have** $CT;\Gamma \vdash Cast\ C\ e0' : C$ **by** *auto*
 **thus** *?case* **using** *Ca-def* **by** (*auto simp add*:*subtyping.s-refl*)
**qed**

## 3.7 Multi-Step Subject Reduction Theorem

**corollary** *Cor-2-4-1-multi*:
 **assumes** $CT \vdash e \rightarrow\!*\ e'$
 **and** $CT\ OK$
 **shows** $\bigwedge C.\ [\![\ CT;\Gamma \vdash e : C\ ]\!] \Longrightarrow \exists\, C'.\ (CT;\Gamma \vdash e' : C' \land CT \vdash C' <: C)$
 **using** *prems* **proof** *induct*
 **case** (*rs-refl CT e C*) **thus** *?case* **by** (*auto simp add*:*subtyping.s-refl*)
**next**
 **case**(*rs-trans CT e e′ e′′ C*)

**hence** *e-typ*: $CT;\Gamma \vdash e : C$
**and** *e-step*: $CT \vdash e \rightarrow e'$
**and** *ct-ok*: $CT\ OK$
**and** *IH*: $\bigwedge D.\ [\![CT;\Gamma \vdash e' : D;\ CT\ OK]\!] \Longrightarrow \exists E.\ CT;\Gamma \vdash e'' : E \wedge CT \vdash E <: D$
**by** *auto*
**from** *Thm-2-4-1*[*OF e-step ct-ok e-typ*] **obtain** $D$ **where** *e′-typ*: $CT;\Gamma \vdash e' : D$ **and** *D-sub-C*: $CT \vdash D <: C$ **by** *auto*
**with** *IH*[*OF e′-typ ct-ok*] **obtain** $E$ **where** $CT;\Gamma \vdash e''$: $E$ **and** *E-sub-D*: $CT \vdash E <: D$ **by** *auto*
**moreover from** *s-trans*[*OF E-sub-D D-sub-C*] **have** $CT \vdash E <: C$ **by** *auto*
**ultimately show** *?case* **by** *auto*
**qed**

## 3.8 Progress

The two "progress lemmas" proved in the TOPLAS paper alone are not quite enough to prove type soundness. We prove an additional lemma showing that every well-typed expression is either a value or contains a potential redex as a sub-expression.

**theorem** *Thm-2-4-2-1*:
**assumes** $CT;empty \vdash e : C$
**and** *FieldProj* (*New C0 es*) *fi* $\in$ *subexprs*(*e*)
**shows** $\exists$ *Cf fDef*. *fields*(*CT*, *C0*) = *Cf* $\wedge$ *lookup Cf* ($\lambda$*fd*. (*vdName fd* = *fi*)) = *Some fDef*
**proof** –
**obtain** *Ci* **where** $CT;empty \vdash$ (*FieldProj* (*New C0 es*) *fi*) : *Ci*
**using** *prems* **by** (*force simp add*:*subexpr-typing*)
**then obtain** *Cf fDef C0′*
**where** $CT;empty \vdash$ (*New C0 es*) : *C0′*
**and** *fields*(*CT*,*C0′*) = *Cf*
**and** *lookup Cf* ($\lambda$*fd*. (*vdName fd* = *fi*)) = *Some fDef*
**by** (*auto elim*:*typings-typing.elims*)
**thus** *?thesis* **by** (*auto elim*:*typings-typing.elims*)
**qed**

**lemma** *Thm-2-4-2-2*:
**assumes** $CT;empty \vdash e : C$
**and** *MethodInvk* (*New C0 es*) *m ds* $\in$ *subexprs*(*e*)
**shows** $\exists$ *xs e0*. *mbody*(*CT*,*m*,*C0*) = *xs* . *e0* $\wedge$ *length xs* = *length ds*
**proof** –
**obtain** $D$ **where** $CT;empty \vdash$ *MethodInvk* (*New C0 es*) *m ds* : $D$
**using** *prems* **by** (*force simp add*:*subexpr-typing*)
**then obtain** *C0′ Cs*
**where** $CT;empty \vdash$ (*New C0 es*) : *C0′*
**and** *mt*:*mtype*(*CT*,*m*,*C0′*) = *Cs* $\rightarrow$ $D$
**and** *length ds* = *length Cs*
**by** (*auto elim*:*typings-typing.elims*)

**with** *mtype-mbody*[*OF mt*] **show** *?thesis* **by** (*force elim*:*typings-typing.elims*)
**qed**

**lemma** *closed-subterm-split*:
**assumes** *CT*;$\Gamma \vdash$ *e* : *C* **and** $\Gamma$ = *empty*
**shows**
((∃ *C0 es fi*. (*FieldProj* (*New C0 es*) *fi*) ∈ *subexprs*(*e*))
∨ (∃ *C0 es m ds*. (*MethodInvk* (*New C0 es*) *m ds*) ∈ *subexprs*(*e*))
∨ (∃ *C0 D es*. (*Cast D* (*New C0 es*)) ∈ *subexprs*(*e*))
∨ *val*(*e*)) (**is** *?F e* ∨ *?M e* ∨ *?C e* ∨ *?V e* **is** *?IH e*)
**using** *prems* **proof**(*induct CT* Γ *e C rule*:*typing-induct*)
**case** *1* **thus** *?case* **using** *prems* **by** *auto*
**next**
**case** (*2 C CT* Γ *x*) **thus** *?case* **by** *auto*
**next**
**case** (*3 C0 Ct Cf Ci* Γ *e0 fDef fi*)
**have** *s1*: *e0* ∈ *subexprs*(*FieldProj e0 fi*) **by**(*auto simp add*:*isubexprs.intros*)
**from** *3* **have** *?IH e0* **by** *auto*
**moreover**
**{ assume** *?F e0*
**then obtain** *C0 es fi′* **where** *s2*: *FieldProj* (*New C0 es*) *fi′* ∈ *subexprs*(*e0*) **by** *auto*
**from** *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
**} moreover {**
**assume** *?M e0*
**then obtain** *C0 es m ds* **where** *s2*: *MethodInvk* (*New C0 es*) *m ds* ∈ *subexprs*(*e0*) **by** *auto*
**from** *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
**} moreover {**
**assume** *?C e0*
**then obtain** *C0 D es* **where** *s2*: *Cast D* (*New C0 es*) ∈ *subexprs*(*e0*) **by** *auto*
**from** *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
**} moreover {**
**assume** *?V e0*
**then obtain** *C0 es* **where** *e0* = (*New C0 es*) **and** *vals*(*es*) **by** (*force elim*:*vals-val.elims*)
**hence** *?case* **by**(*force intro*:*isubexprs.intros*)
**}**
**ultimately show** *?case* **by** *blast*
**next**
**case** (*4 C C0 CT Cs Ds* Γ *e0 es m*)
**have** *s1*: *e0* ∈ *subexprs*(*MethodInvk e0 m es*) **by**(*auto simp add*:*isubexprs.intros*)
**from** *4* **have** *?IH e0* **by** *auto*
**moreover**
**{ assume** *?F e0*
**then obtain** *C0 es fi* **where** *s2*: *FieldProj* (*New C0 es*) *fi* ∈ *subexprs*(*e0*) **by** *auto*
**from** *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
**} moreover {**
**assume** *?M e0*

**then obtain** *C0 es′ m′ ds* **where** *s2*: *MethodInvk* (*New C0 es′*) *m′ ds* ∈ *subexprs*(*e0*) **by** *auto*
**from** *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
**}** **moreover** **{**
**assume** *?C e0*
**then obtain** *C0 D es* **where** *s2*: *Cast D* (*New C0 es*) ∈ *subexprs*(*e0*) **by** *auto*
**from** *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
**}** **moreover** **{**
**assume** *?V e0*
**then obtain** *C0 es′* **where** *e0* = (*New C0 es′*) **and** *vals*(*es′*) **by** (*force elim*:*vals-val.elims*)
**hence** *?case* **by**(*force intro*:*isubexprs.intros*)
**}**
**ultimately show** *?case* **by** *blast*
**next**
**case** (*5 C CT Cs Df Ds* Γ *es*)
**hence**
*length es* = *length Cs*
$\bigwedge$ *i*. ⟦*i* < *length es*; *CT*;Γ ⊢ (*es*!*i*) : (*Cs*!*i*); Γ = *empty*⟧ ⟹ *?IH* (*es*!*i*)
**and** *CT*;Γ ⊢+ *es* : *Cs*
**by** (*auto simp add*:*typings-lengths*)
**hence** (∃ *i* < *length es*. (*?F* (*es*!*i*) ∨ *?M* (*es*!*i*) ∨ *?C* (*es*!*i*))) ∨ (*vals*(*es*)) (**is** *?Q es*)
**proof**(*induct es Cs rule*:*list-induct2*)
**case** *1* **thus** *?Q* [] **by**(*auto intro*:*vals-val.intros*)
**next**
**case** (*2 h t Ch Ct*)
**hence** *h-t-typs*: *CT*;Γ ⊢+ (*h*#*t*) : (*Ch*#*Ct*)
**and** *OIH*: $\bigwedge$ *i*. ⟦*i* < *length* (*h*#*t*); *CT*;Γ ⊢ ((*h*#*t*)!*i*) : ((*Ch*#*Ct*)!*i*); Γ = *empty*⟧ ⟹ *?IH* ((*h*#*t*)!*i*)
**and** *G-def*: Γ = *empty*
**by** *auto*
**from** *h-t-typs* **have**
*h-typ*: *CT*;Γ ⊢ (*h*#*t*)!*0* : (*Ch*#*Ct*)!*0*
**and** *t-typs*: *CT*;Γ ⊢+ *t* : *Ct*
**by**(*auto elim*:*typings-typing.elims*)
**{** **fix** *i* **assume** *i* < *length t*
**hence** *s-i*: *Suc i* < *length* (*h*#*t*) **by** *auto*
**from** *OIH*[*OF s-i*] **have** ⟦*i* < *length t*; *CT*;Γ ⊢ (*t*!*i*) : (*Ct*!*i*); Γ = *empty*⟧ ⟹ *?IH* (*t*!*i*) **by** *auto* **}**
**with** *t-typs* **have** *?Q t* **using** *2* **by** *auto*
**moreover** **{**
**assume** ∃ *i* < *length t*. (*?F* (*t*!*i*) ∨ *?M* (*t*!*i*) ∨ *?C* (*t*!*i*))
**then obtain** *i*
**where** *i* < *length t*
**and** *?F* (*t*!*i*) ∨ *?M* (*t*!*i*) ∨ *?C* (*t*!*i*) **by** *force*
**hence** (*Suc i* < *length* (*h*#*t*)) ∧ (*?F* ((*h*#*t*)!(*Suc i*)) ∨ *?M* ((*h*#*t*)!(*Suc i*)) ∨ *?C* ((*h*#*t*)!(*Suc i*))) **by** *auto*
**hence** ∃ *i* < *length* (*h*#*t*). (*?F* ((*h*#*t*)!*i*) ∨ *?M* ((*h*#*t*)!*i*) ∨ *?C* ((*h*#*t*)!*i*))

```
..
        hence ?Q (h#t) by auto
      } moreover {
        assume v-t: vals(t)
        from OIH[OF - h-typ G-def] have ?IH h by auto
        moreover
        { assume ?F h ∨ ?M h ∨ ?C h
          hence ?F ((h#t)!0) ∨ ?M ((h#t)!0) ∨ ?C ((h#t)!0) by auto
          hence ?Q (h#t) by force
        } moreover {
          assume ?V h
          with v-t have vals((h#t)) by (force intro:vals-val.intros)
          hence ?Q(h#t) by auto
        } ultimately have ?Q(h#t) by blast
      } ultimately show ?Q(h#t) by blast
    qed
    moreover {
      assume ∃ i<length es. ?F (es!i) ∨ ?M (es!i) ∨ ?C(es!i)
      then obtain i where i-len: i < length es and r: ?F (es!i) ∨ ?M (es!i) ∨
?C(es!i) by force
       from ith-mem[OF i-len] have s1:es!i ∈ subexprs(New C es) by(auto in-
tro:isubexprs.se-newarg)
      { assume ?F (es!i)
       then obtain C0 es' fi where s2: FieldProj (New C0 es') fi ∈ subexprs(es!i)
by auto
         from rtrancl-trans[OF s2 s1] have ?F(New C es) ∨ ?M(New C es) ∨
?C(New C es) by auto
      } moreover {
        assume ?M (es!i)
        then obtain C0 es' m' ds where s2: MethodInvk (New C0 es') m' ds ∈
subexprs(es!i) by force
         from rtrancl-trans[OF s2 s1] have ?F(New C es) ∨ ?M(New C es) ∨
?C(New C es) by auto
      } moreover {
        assume ?C (es!i)
        then obtain C0 D es' where s2: Cast D (New C0 es') ∈ subexprs(es!i)
by auto
         from rtrancl-trans[OF s2 s1] have ?F(New C es) ∨ ?M(New C es) ∨
?C(New C es) by auto
      } ultimately have ?F(New C es) ∨ ?M(New C es) ∨ ?C(New C es) using
r by blast
      hence ?case by auto
    } moreover {
      assume vals(es)
      hence ?case by(auto intro:vals-val.intros)
    } ultimately show ?case by blast
  next
    case (6 C CT D Γ e0)
    have s1: e0 ∈ subexprs(Cast C e0) by(auto simp add:isubexprs.intros)
```

**from** *6* **have** *?IH e0* **by** *auto*
**moreover**
**{ assume** *?F e0*
**then obtain** *C0 es fi* **where** *s2*: *FieldProj* (*New C0 es*) *fi* $\in$ *subexprs*(*e0*) **by** *auto*
**from** *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
**} moreover {**
**assume** *?M e0*
**then obtain** *C0 es m ds* **where** *s2*: *MethodInvk* (*New C0 es*) *m ds* $\in$ *subexprs*(*e0*) **by** *auto*
**from** *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
**} moreover {**
**assume** *?C e0*
**then obtain** *C0 D′ es* **where** *s2*: *Cast D′* (*New C0 es*) $\in$ *subexprs*(*e0*) **by** *auto*
**from** *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
**} moreover {**
**assume** *?V e0*
**then obtain** *C0 es′* **where** *e0* = (*New C0 es′*) **and** *vals*(*es′*) **by** (*force elim*:*vals-val.elims*)
**hence** *?case* **by**(*force intro*:*isubexprs.intros*)
**}**
**ultimately show** *?case* **by** *blast*
**next**
**case** (*7 C CT D* $\Gamma$ *e0*)
**have** *s1*: *e0* $\in$ *subexprs*(*Cast C e0*) **by**(*auto simp add*:*isubexprs.intros*)
**from** *7* **have** *?IH e0* **by** *auto*
**moreover**
**{ assume** *?F e0*
**then obtain** *C0 es fi* **where** *s2*: *FieldProj* (*New C0 es*) *fi* $\in$ *subexprs*(*e0*) **by** *auto*
**from** *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
**} moreover {**
**assume** *?M e0*
**then obtain** *C0 es m ds* **where** *s2*: *MethodInvk* (*New C0 es*) *m ds* $\in$ *subexprs*(*e0*) **by** *auto*
**from** *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
**} moreover {**
**assume** *?C e0*
**then obtain** *C0 D′ es* **where** *s2*: *Cast D′* (*New C0 es*) $\in$ *subexprs*(*e0*) **by** *auto*
**from** *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
**} moreover {**
**assume** *?V e0*
**then obtain** *C0 es′* **where** *e0* = (*New C0 es′*) **and** *vals*(*es′*) **by** (*force elim*:*vals-val.elims*)
**hence** *?case* **by**(*force intro*:*isubexprs.intros*)
**}**
**ultimately show** *?case* **by** *blast*

**next**
  **case** (*8 C CT D* Γ *e0*)
  **have** *s1*: *e0* ∈ *subexprs*(*Cast C e0*) **by**(*auto simp add*:*isubexprs.intros*)
  **from** *8* **have** *?IH e0* **by** *auto*
  **moreover**
  **{ assume** *?F e0*
    **then obtain** *C0 es fi* **where** *s2*: *FieldProj* (*New C0 es*) *fi* ∈ *subexprs*(*e0*) **by** *auto*
    **from** *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
  **} moreover {**
    **assume** *?M e0*
    **then obtain** *C0 es m ds* **where** *s2*: *MethodInvk* (*New C0 es*) *m ds* ∈ *subexprs*(*e0*) **by** *auto*
    **from** *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
  **} moreover {**
    **assume** *?C e0*
    **then obtain** *C0 D′ es* **where** *s2*: *Cast D′* (*New C0 es*) ∈ *subexprs*(*e0*) **by** *auto*
    **from** *rtrancl-trans*[*OF s2 s1*] **have** *?case* **by** *auto*
  **} moreover {**
    **assume** *?V e0*
    **then obtain** *C0 es′* **where** *e0* = (*New C0 es′*) **and** *vals*(*es′*) **by** (*force elim*:*vals-val.elims*)
    **hence** *?case* **by**(*force intro*:*isubexprs.intros*)
  **}**
  **ultimately show** *?case* **by** *blast*
**qed**

## 3.9 Type Soundness Theorem

**theorem** *Thm-2-4-3*:
  **assumes** *e-typ*: *CT*;*empty* ⊢ *e* : *C*
  **and** *ct-ok*: *CT OK*
  **and** *multisteps*: *CT* ⊢ *e* →∗ *e1*
  **and** *no-step*: ¬(∃ *e2*. *CT* ⊢ *e1* → *e2*)
  **shows** (*val*(*e1*) ∧ (∃ *D*. *CT*;*empty* ⊢ *e1* : *D* ∧ *CT* ⊢ *D* <: *C*))
    ∨ (∃ *D C es*. (*Cast D* (*New C es*) ∈ *subexprs*(*e1*) ∧ *CT* ⊢ *C* ¬<: *D*))
**proof** −
**from** *prems Cor-2-4-1-multi*[*OF multisteps ct-ok e-typ*] **obtain** *C1*
  **where** *e1-typ*: *CT*;*empty* ⊢ *e1* : *C1*
  **and** *C1-sub-C*: *CT* ⊢ *C1* <: *C* **by** *auto*
**from** *e1-typ* **have** ((∃ *C0 es fi*. (*FieldProj* (*New C0 es*) *fi*) ∈ *subexprs*(*e1*))
    ∨ (∃ *C0 es m ds*. (*MethodInvk* (*New C0 es*) *m ds*) ∈ *subexprs*(*e1*))
    ∨ (∃ *C0 D es*. (*Cast D* (*New C0 es*)) ∈ *subexprs*(*e1*))
    ∨ *val*(*e1*)) (**is** *?F e1* ∨ *?M e1* ∨ *?C e1* ∨ *?V e1*) **by** (*simp add*: *closed-subterm-split*)
**moreover**
**{ assume** *?F e1*
  **then obtain** *C0 es fi* **where** *fp*: *FieldProj* (*New C0 es*) *fi* ∈ *subexprs*(*e1*) **by**

*auto*
**then obtain** *Ci* **where** *CT*;*empty* ⊢ *FieldProj* (*New C0 es*) *fi* : *Ci* **using** *e1-typ* **by**(*force simp add*:*subexpr-typing*)
**then obtain** *C0′* **where** *new-typ*: *CT*;*empty* ⊢ *New C0 es* : *C0′* **by** (*force elim*: *typings-typing.elims*)
**hence** *C0* = *C0′* **by** (*auto elim*:*typings-typing.elims*)
**with** *new-typ* **obtain** *Df* **where** *f1*: *fields*(*CT*,*C0*) = *Df* **and** *lens*: *length es* = *length Df* **by**(*auto elim*:*typings-typing.elims*)
**from** *Thm-2-4-2-1*[*OF e1-typ fp*] **obtain** *Cf fDef*
**where** *f2*: *fields*(*CT*,*C0*) = *Cf*
**and** *lkup*: *lookup Cf* (λ*fd*. *vdName fd* = *fi*) = *Some*(*fDef*) **by** *force*
**moreover from** *fields-functional*[*OF f1 ct-ok f2*] *lens* **have** *length es* = *length Cf* **by** *auto*
**moreover from** *lookup-index*[*OF lkup*] **obtain** *i* **where**
*i*<*length Cf*
**and** *fDef* = *Cf* ! *i*
**and** (*length Cf* = *length es*) ⟶ *lookup2 Cf es* (λ*fd*. *vdName fd* = *fi*) = *Some* (*es* ! *i*) **by** *auto*
**ultimately have** *lookup2 Cf es* (λ*fd*. *vdName fd* = *fi*) = *Some* (*es*!*i*) **by** *auto*
**with** *f2* **have** *CT* ⊢ *FieldProj*(*New C0 es*) *fi* → (*es*!*i*) **by**(*auto intro*:*reduction.intros*)
**with** *fp* **have** ∃ *e2*. *CT* ⊢ *e1* → *e2* **by**(*simp add*:*subexpr-reduct*)
**with** *no-step* **have** *?thesis* **by** *auto*
**} moreover {**
**assume** *?M e1*
**then obtain** *C0 es m ds* **where** *mi*:*MethodInvk* (*New C0 es*) *m ds* ∈ *subexprs*(*e1*) **by** *auto*
**then obtain** *D* **where** *CT*;*empty* ⊢ *MethodInvk* (*New C0 es*) *m ds* : *D* **using** *e1-typ* **by**(*force simp add*:*subexpr-typing*)
**then obtain** *C0′ Es E*
**where** *m-typ*: *CT*;*empty* ⊢ *New C0 es* : *C0′*
**and** *mtype*(*CT*,*m*,*C0′*) = *Es* → *E*
**and** *length ds* = *length Es*
**by** (*auto elim*:*typings-typing.elims*)
**from** *Thm-2-4-2-2*[*OF e1-typ mi*] **obtain** *xs e0* **where** *mb*: *mbody*(*CT*, *m*, *C0*) = *xs* . *e0* **and** *length xs* = *length ds* **by** *auto*
**hence** *CT* ⊢ (*MethodInvk* (*New C0 es*) *m ds*) → (*substs*[*xs*[↦]*ds*,*this*↦(*New C0 es*)]*e0*) **by**(*auto simp add*:*reduction.intros*)
**with** *mi* **have** ∃ *e2*. *CT* ⊢ *e1* → *e2* **by**(*simp add*:*subexpr-reduct*)
**with** *no-step* **have** *?thesis* **by** *auto*
**} moreover {**
**assume** *?C e1*
**then obtain** *C0 D es* **where** *c-def*: *Cast D* (*New C0 es*) ∈ *subexprs*(*e1*) **by** *auto*
**then obtain** *D′* **where** *CT*;*empty* ⊢ *Cast D* (*New C0 es*) : *D′* **using** *e1-typ* **by** (*force simp add*:*subexpr-typing*)
**then obtain** *C0′* **where** *new-typ*: *CT*;*empty* ⊢ *New C0 es* : *C0′* **and** *D-eq-D′*: *D* = *D′* **by** (*auto elim*:*typings-typing.elims*)
**hence** *C0-eq-C0′*: *C0* = *C0′* **by**(*auto elim*:*typings-typing.elims*)
**hence** *?thesis* **proof**(*cases CT* ⊢ *C0* <: *D*)

**case** *True*
**hence** $CT \vdash Cast\ D\ (New\ C0\ es) \rightarrow (New\ C0\ es)$ **by**(*auto simp add:reduction.intros*)
**with** *c-def* **have** $\exists\, e2.\ CT \vdash e1 \rightarrow e2$ **by** (*simp add:subexpr-reduct*)
**with** *no-step* **show** *?thesis* **by** *auto*
**next**
**case** *False*
**with** *c-def* **show** *?thesis* **by** *auto*
**qed**
**} moreover {**
**assume** *?V e1*
**hence** *?thesis* **using** *prems* **by**(*auto simp add:Cor-2-4-1-multi*)
**} ultimately show** *?thesis* **by** *blast*
**qed**

**end**

## References


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

[2] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283. 2002. http://www.in.tum.de/~nipkow/LNCS2283/.