Invited talk at the Colloquium
Logic in Computer Science in Munich
on July 24, 1998

Operational Semantics of Java and Subject Reduction

David von Oheimb[1], TU München

`http://www.in.tum.de/~oheimb/`

[1]supported by DFG Schwerpunktprojekt *Deduktion*

Overview

- Part I: Operational Semantics
  - Introduction
  - Semantics of Java
  - Evaluation Semantics
  - Transition Semantics
  - Discussion
- Part II: Type-Safety
  - Motivation
  - Evaluation Semantics
  - Transition Semantics
  - Subject Reduction: Problems & Solutions
  - Non-termination
  - Conclusion

Semantics: Aims

- precise specification
- proof of properties
- compiler verification
- program verification

Semantics: Approaches

- Denotational
- Operational
- Axiomatic

Semantics: Design Goals

- faithfulness to the official specification
- ease of validation
- succinctness and simplicity
- abstraction and minimal redundancy
- maintainability and extendibility
- adequacy for theorem proving

Denotational Semantics

... is model-oriented

... gives language designer's point of view

Interpretations:

$$\mathcal{S}[\![\_]\!] :: stmt \Rightarrow (state \Rightarrow \qquad state)$$
$$\mathcal{E}[\![\_]\!] :: expr \Rightarrow (state \Rightarrow val \times state)$$

$$\mathcal{S}[\![s_1 ;\ s_2]\!] = \mathcal{S}[\![s_2]\!] \circ \mathcal{S}[\![s_1]\!]$$
$$\mathcal{E}[\![X\texttt{:=}e]\!] = \lambda\sigma.\ \text{let } (v,\sigma') = \mathcal{E}[\![e]\!]\sigma \\ \text{in } (v,\sigma'[X \mapsto v])$$

compositional, primitve recursive definitions

proofs by structural induction, unfolding of def.

$\oplus$ abstractness

$\ominus$ complex mathematical models

Operational Semantics

... is machine-oriented

... gives implementor's and programmer's view

Evaluation judgments:

$$state \quad -stmt\rightarrow \quad state$$
$$state \; -expr \triangleright val \rightarrow \; state$$

$$\frac{\sigma_0 \; -s_1 \rightarrow \; \sigma_1 \qquad \sigma_1 \; -s_2 \rightarrow \; \sigma_2}{\sigma_0 \; -s_1 ; \; s_2 \rightarrow \; \sigma_2}$$

$$\frac{\sigma \; -e \triangleright v \rightarrow \; \sigma'}{\sigma \; -X := e \triangleright v \rightarrow \; \sigma'[X \mapsto v]}$$

relational, inductive definitions

proofs by rule induction, rule application

$\oplus$ validation

$\oplus$ simplicity

$\ominus$ abstractness?

Axiomatic Semantics

... is property-oriented

... gives specifier's and verifier's point of view

Hoare triples:

$$assn = state \Rightarrow bool$$

$$\models \{assn\} \quad stmt \quad \{assn\}$$

$$\models \{assn\} \; expr \rhd val \; \{assn\}$$

$$\frac{\models \{P\}\ s_1\ \{Q\} \quad \models \{Q\}\ s_2\ \{R\}}{\models \{P\}\ s_1;\ s_2\ \{R\}}$$

$$\frac{\models \{P\}\ e \rhd v\ \{P'\}}{\models \{P[v/X]\}\ X\texttt{:=}e\ \{P'\}}$$

relational, inductive definitions

proofs by rule induction, wp-calculus

$\oplus$ abstractness

$\ominus$ validation

Semantics: Terms and Values

$$
\begin{aligned}
stmt = {} & \mathsf{Skip} \\
 & \mid stmt;\ stmt \\
 & \mid \ldots
\end{aligned}
$$

$$
\begin{aligned}
expr = {} & lit \\
 & \mid expr\ ?\ expr : expr \\
 & \mid expr[expr] \\
 & \mid expr[expr] \mathtt{:=} expr \\
 & \mid expr\mathbf{.}\,mname(expr) \\
 & \mid \ldots
\end{aligned}
$$

$$
\begin{aligned}
lit = {} & \mathsf{Bool}\ bool \\
 & \mid \mathsf{Intg}\ int \\
 & \mid \mathsf{Null} \\
 & \mid \ldots
\end{aligned}
$$

$$
\begin{aligned}
val = {} & lit \\
 & \mid \mathsf{Addr}\ loc
\end{aligned}
$$

Semantics: Type System

Types:

$$
\begin{array}{ll}
tname & = \ldots \\
ty & = prim\_ty \mid ref\_ty \\
prim\_ty & = \texttt{boolean} \mid \texttt{int} \mid \ldots \\
ref\_ty & = \mathsf{Class}\ tname \mid \mathsf{Iface}\ tname \mid ty[] \mid \mathsf{NullT}
\end{array}
$$

Type relations (Widening etc.) :

$$\frac{\mathsf{is\_type}\ \Gamma\ \ T}{\Gamma \vdash T \preceq T} \qquad \frac{\Gamma \vdash \mathsf{RefT}\ \ S\ \preceq\ \mathsf{RefT}\ \ T}{\Gamma \vdash (\mathsf{RefT}\ \ S)[] \preceq (\mathsf{RefT}\ \ T)[]}$$

$$\frac{\Gamma \vdash C \rightsquigarrow I}{\Gamma \vdash \mathsf{Class}\ \ C \preceq \mathsf{Iface}\ \ I} \qquad \frac{\mathsf{is\_iface}\ \Gamma\ \ I}{\Gamma \vdash \mathsf{Iface}\ \ I \preceq \texttt{Object}}$$

Widening is ...

- syntactical (not behavioural) subtyping
- explicit (given by declaration)

Semantics: Well-Typedness

Typing judgments:

$env = prog \times (name,\ ty)\ table$

$env \vdash stmt :: \diamond$

$env \vdash expr :: ty$

$$\frac{E \vdash s_1 :: \diamond\ ; \quad E \vdash s_2 :: \diamond}{E \vdash s_1\, \mathtt{;}\ s_2 :: \diamond}$$

$$\frac{\text{typeof empty } x = \text{Some } T}{E \vdash \text{Lit } x :: T}$$

$$\frac{E \vdash a[i] :: T \quad E \vdash v :: T' \quad T' \preceq T}{E \vdash a[i] \,\mathtt{:=}\, v :: T'}$$

$$\frac{\begin{array}{c} E \vdash e_0 :: \mathtt{boolean} \quad E \vdash e_1 :: T_1 \quad E \vdash e_2 :: T_2 \\ T_1 \preceq T_2 \wedge T = T_2 \ \vee \\ T_2 \preceq T_1 \wedge T = T_1 \end{array}}{E \vdash e_0\ ?\ e_1\ :\ e_2 :: T}$$

Theorem: Typing is unique.

Semantics: State

$$
\begin{aligned}
state &= (xcpt)\ option \times heap \times locals \\
heap &= (loc\quad,\ obj)\ table \\
locals &= (name,\ val)\ table
\end{aligned}
$$

$$
\begin{aligned}
obj\_tag &= \mathsf{CInst}\ tname \\
&\ \mid\ \ \mathsf{Arr}\quad ty\ int \\
vn &= \ldots \\
obj &= obj\_tag \times (vn,\ val)\ table
\end{aligned}
$$

Terms may have dynamic type!

$$
\begin{aligned}
&\mathsf{typeof} :: state \Rightarrow val \Rightarrow ty\ option \\
&\mathsf{typeof}\ \sigma\ (\mathsf{Bool}\ b) \ = \mathsf{Some}\ \mathtt{boolean} \\
&\mathsf{typeof}\ \sigma\ (\mathsf{Intg}\ i) \ = \mathsf{Some}\ \mathtt{int} \\
&\mathsf{typeof}\ \sigma\ \ \mathsf{Null} \ = \mathsf{Some}\ \mathsf{NullT} \\
&\mathsf{typeof}\ \sigma\ (\mathsf{Addr}\ a) \ = \mathsf{opt\_map}\ \mathsf{obj\_ty}\ (\mathsf{heap}\ \sigma\ a)
\end{aligned}
$$

Evaluation (“big-step”) Semantics

... is result-oriented:
A term (expression) is evaluated (yielding a value).
Strict distinction of terms and values!

Evaluation judgments:

$prog \vdash state \quad -stmt\rightarrow \quad state$

$prog \vdash state -expr \rhd val \rightarrow state$

$$\frac{\Gamma \vdash \sigma_0 \ -s_1\rightarrow \ \sigma_1 \quad \Gamma \vdash \sigma_1 \ -s_2\rightarrow \ \sigma_2}{\Gamma \vdash \sigma_0 \ -s_1;\ s_2\rightarrow \ \sigma_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash \sigma_0 \ -e_1 \rhd a\rightarrow \ \sigma_1 \quad \Gamma \vdash \sigma_1 \ -e_2 \rhd i\rightarrow \ \sigma_2 \\ \Gamma \vdash \sigma_2 \ -e_3 \rhd v \rightarrow \ \sigma_3 \quad T = \mathsf{fst}\ (\mathsf{heap}\ \sigma_3\ a) \\ {\sigma_3}' = \ \ldots\ \sigma_3\ \ldots\ a\ \ldots\ [i \mapsto v]\ \ldots \\ \ldots\ \mathsf{raise\_if}\ (\neg\ \Gamma,\sigma_3 \vdash v\ \mathsf{fits}\ T)\ \mathtt{ArrStore}\ \ldots\end{array}}{\Gamma \vdash \sigma_0 \ -e_1[e_2]\mathtt{:=}e_3 \rhd v\rightarrow \ {\sigma_3}'}$$

$\oplus$ same style as the Java specification

$\oplus$ readability through self-contained rules

$\ominus$ cannot express interleaving

Transition ("small-step") Semantics

... is computation-oriented:
A term is transformed to another term.

Transition judgments:

$$prog \vdash (stmt, state) \mapsto (stmt, state)$$
$$prog \vdash (expr, state) \rightarrow (expr, state)$$

$$\frac{\Gamma \vdash (s_1, \sigma) \mapsto ({s_1}', \sigma')}{\Gamma \vdash (s_1;\ s_2, \sigma) \mapsto ({s_1}';\ s_2, \sigma')}$$

$$\frac{}{\Gamma \vdash (\mathsf{Skip};\ s_2, \sigma) \mapsto (s_2, \sigma)}$$

Terms represent ...

- the (source) program
- intermediate stages of computation
- final values ("ground" terms, literals)

Transition Semantics: Array assignment example

$$\frac{\Gamma \vdash (e_1,\sigma) \rightarrow ({e_1}',\sigma')}{\Gamma \vdash (e_1[e_2]\texttt{:=}e_3,\sigma) \rightarrow ({e_1}'[e_2]\texttt{:=}e_3,\sigma')}$$

Scheme: $ES\ C \stackrel{\mathsf{def}}{=} \dfrac{\Gamma \vdash (\ \ t,\sigma) \rightarrow (\ \ t',\sigma')}{\Gamma \vdash (C\ t,\sigma) \rightarrow (C\ t',\sigma')}$

abbreviates rules to

$$ES\ (\lambda e_1.\ e_1[e_2]\texttt{:=}e_3)$$

$$ES\ (\lambda e_2.\ \mathsf{Addr}\ a[e_2])\texttt{:=}e_3)$$

$$ES\ (\lambda e_3.\ \mathsf{Addr}\ a[\mathsf{Intg}\ i])\texttt{:=}e_3)$$

$$\frac{\begin{array}{c} T = \mathsf{fst}\ (\mathsf{heap}\ \sigma\ a) \\ \sigma' = \ldots\ \sigma\ \ldots\ a\ \ldots\ [i{\mapsto}v]\ \ldots \\ \ldots\ \mathsf{raise\_if}\ (\neg\ \Gamma,\sigma \vdash v\ \mathsf{fits}\ T)\ \texttt{ArrStore}\ \ldots \end{array}}{\Gamma \vdash (\mathsf{Addr}\ a[\mathsf{Intg}\ i])\texttt{:=}\mathsf{Lit}\ v,\sigma) \rightarrow (\mathsf{Lit}\ v,\sigma')}$$

Transition Semantics ...

$\ominus$ requires more rules
$\oplus$ each single rule is simpler
$\ominus$ requires distinction: ground/non-ground

$\ominus$ requires artificial terms, namely addresses
$\ominus$ $\leadsto$ extended internal language
$\ominus$ $\leadsto$ extended type system

$\oplus$ allows for interleaving:

$$\frac{\Gamma \vdash (s_1,\sigma) \mapsto ({s_1}',\sigma')}{\Gamma \vdash (s_1 \parallel s_2,\sigma) \mapsto ({s_1}' \parallel s_2,\sigma')}$$

$$\frac{\Gamma \vdash (s_2,\sigma) \mapsto ({s_2}',\sigma')}{\Gamma \vdash (s_1 \parallel s_2,\sigma) \mapsto (s_1 \parallel {s_2}',\sigma')}$$

The types of terms may get narrower
during computation
$\ominus\ominus$ $\leadsto$ requires (for Java) modifications of
intermediate typing rules or language

Static Resolution of Overloading

Method overloading problem:

```
class D extends C {
 static char ol(C x) { return 'c'; }
 static int  ol(D y) { return 42; }
 static void m (C x) { println(ol(x));}
 static void test()  { m(new D()); }
}
```

`test()` should print `'c'`.

Naive execution:

$(m(\mathtt{new}\ D()),\sigma) \rightarrow$
$(m(\mathsf{Addr}\ dd),\sigma') \rightarrow$
$(println(ol(\mathsf{Addr}\ dd)),\sigma') \rightarrow_{\beta}$
$(println(42),\sigma') \rightarrow$
$\ldots$

$\leadsto$ requires static information!

Static Overloading: Solution

Adding Type Annotations for method call:

$$\frac{E \vdash e{::}\mathsf{RefT}\ T \qquad E \vdash p{::}pT \qquad \mathsf{max\_spec}\ E\ T\ (m,pT) = \{((md,(n,rT)),pT')\}}{E \vdash e.m(\{pT'\}p){::}rT}$$

The same with field access/assignment:

$$\frac{E \vdash e{::}\mathsf{Class}\ C \qquad \mathsf{cfield}\ E\ C\ fn = \mathsf{Some}\ (fd,fT)}{E \vdash \{fd\}e.fn{::}fT}$$

$\leadsto$ new intermediate language!

Java $\supset$ Java$_s$ $\xrightarrow{elab}$ Java$_{se}$

Operational Semantics: Conclusion

design goal	Eval. Sem.	Trans. Sem.
allows faithful description	$\oplus$	$\oplus\oplus$
ease of validation	$\oplus\oplus$	$\oplus$
succinctness and simplicity	$\oplus\oplus$	$\oplus$
abstraction and minimal redundancy	$\oplus$	$\oplus/\ominus$
maintainability and extendibility	$\oplus$	$\ominus$
adequacy for theorem proving	$\oplus\oplus$	$\oplus$?

Type-Safety (Type Soundness): Aims

Type-safety . . .

- is a key property of programming languages
- relates the type system and the semantics
- guarantees the absence of run-time type errors

A state-based language is type-safe $\stackrel{def}{\Longleftrightarrow}$

$$\begin{array}{l} \forall \text{expression } e,\ \text{type } T,\ \text{value } v,\ \text{states } \sigma, \sigma',\ \text{environment } E. \\ \qquad e \text{ has static type } T \ \wedge\ \sigma \ \text{ conforms to } E \wedge \\ \qquad e \text{ evaluates to } v \text{ transforming } \sigma \text{ into } \sigma' \ \Longrightarrow \\ \qquad\quad v \text{ conforms to } T \ \wedge\ \sigma' \text{ conforms to } E \end{array}$$

"Well-typed programs do not go wrong" (cf. Robin Milner, 1978)

Type-Safety in Evaluation Semantics [ON98]

Notions Conformance

- a value conforms to its (static) type:
  $\Gamma,\sigma \vdash v{::}\preceq T \overset{\text{def}}{=} \exists T'.\ \text{typeof}\ \sigma\ \ v = \text{Some}\ \ T' \wedge \Gamma \vdash T' \preceq T$
- all variables conform to their respective types:
  $\sigma\,{::}\preceq(\Gamma,\Lambda) \overset{\text{def}}{=} \forall T{\in}\ldots\sigma\ldots\Lambda\ldots.\ \Gamma,\sigma \vdash v{::}\preceq T\ \ldots$

Theorem Type Soundness

$E = (\Gamma,\Lambda) \wedge \text{wf\_prog}\ \Gamma \wedge E \vdash e{::}\,T \wedge \sigma\,{::}\preceq E \wedge \Gamma \vdash \sigma\ \ {-e}{\triangleright}v{\rightarrow}\ \sigma' \Longrightarrow$
$(\text{fst}\ \sigma' = \text{None} \Longrightarrow \Gamma,\sigma' \vdash v{::}\preceq T) \wedge \sigma'\,{::}\preceq E$

Proof by rule induction over $\_ \vdash \_\ \ {-\_}{\triangleright}{\_}{\rightarrow}\ \_$

Type-Safety: General lemmata

$\text{wf\_prog}\ \Gamma \Longrightarrow \text{wf}\ (\lambda(J,I).\ \Gamma \vdash I \prec_{\mathsf{i}} J\ ) \wedge \text{wf}\ (\lambda(D,C).\ \Gamma \vdash C \prec_{\mathsf{c}} D)$

$\text{wf\_prog}\ \Gamma \wedge \text{cfield}\ \Gamma\ C\ fn = \text{Some}\ (fd,fT) \wedge \text{Class}\ C' \preceq \text{Class}\ C \Longrightarrow$
$\quad \text{table\_of}\ (\text{fields}\ \Gamma\ C')\ (fn,fd) = \text{Some}\ fT$

$\text{wf\_prog}\ \Gamma \wedge \Gamma \vdash \sigma\ {-s}{\rightarrow}\ \sigma' \Longrightarrow \sigma \trianglelefteq \sigma'$

Type-Safety for Evaluation Semantics

$$\begin{array}{lccccccc}
\text{Java} \supset & \text{Java}_s & \xrightarrow{elab} & \text{Java}_{se} & \rightarrow & \text{Value}\ v \\
& \downarrow & & \downarrow & & \downarrow \\
\text{Type:} & \tau & = & \tau & \succeq & \tau'
\end{array}$$

Type-Safety in Transition Semantics

Type Soundness goal:

$E = (\Gamma,\Lambda) \wedge$ wf_prog $\Gamma \wedge E \vdash e :: T \wedge \sigma :: \preceq E \wedge \neg\ \Gamma \vdash (e,\sigma) \Uparrow \Longrightarrow \exists e'\ \sigma'.$
$\Gamma \vdash (e,\sigma) \rightarrow^* (e',\sigma') \wedge e'$ is ground $\wedge$
(let $v =$ val_of e' in $\Gamma,\sigma' \vdash v :: \preceq T) \wedge \sigma' :: \preceq E$

desired proof pattern: **Subject Reduction** (i.e. type preservation)

$$E = (\Gamma,\Lambda) \wedge \text{wf\_prog}\ \Gamma \wedge E,\sigma \models e :: T \wedge \sigma :: \preceq E \wedge \Gamma \vdash (e,\sigma) \rightarrow (e',\sigma') \Longrightarrow$$
$$\exists E'\ T'.\ E \leq E' \wedge E',\sigma' \models e' :: T' \wedge \sigma' :: \preceq E' \wedge T' \preceq T$$

Stronger version (execution does not get stuck):
wf_prog $\Gamma \wedge \ldots \wedge \neg\ e$ is ground $\Longrightarrow \exists e'\ \sigma'.\ \Gamma \vdash (e,\sigma) \rightarrow (e',\sigma') \wedge \ldots$
to be proved by rule induction over $\_,\_ \models \_ :: \_$

Subject Reduction: extensions required [DE98]

Terms:

$expr' = expr \mid \mathsf{Addr}\ loc$

Typing judgments:

$env, state \models stmt :: \diamond$
$env, state \models expr :: ty$

Typing rules: lifted previous rules +

$$\frac{\mathsf{typeof}\ \sigma\ (\mathsf{Addr}\ a) = \mathsf{Some}\ T}{E,\sigma \models \mathsf{Addr}\ a :: T} \quad (e.g.\ \mathsf{Addr}\ a[e])$$

$$\frac{\mathsf{is\_type}\ \Gamma\ (\mathsf{RefT}\ t)}{E,\sigma \models \mathsf{Null} :: \mathsf{RefT}\ t} \quad (e.g.\ \mathsf{Null}[e])$$

Consequence: Typing is not unique.

Subject Reduction: complex lemmata required

Substitution ("OO polymorphism") Property:

Substitution on types is monotonic w.r.t. widening

$E{,}\sigma \models e{::}\,T \wedge s$ subterm of e $\wedge\ E{,}\sigma \models s{::}S \wedge E{,}\sigma \models s'{::}S' \wedge S' \preceq S \Longrightarrow \exists\, T'.$
$E{,}\sigma \models e[s'/s]{::}\,T' \wedge T' \preceq T$

holds only if several problems are solved.

Side effect Property [Sym98]:

Reduction of terms does not affect the type of other terms

$E{,}\sigma \models e{::}\,T \wedge$ wf_prog $E \wedge \sigma \mathrel{::\preceq} E \wedge E{,}\sigma \models s{::}S \wedge (s{,}\sigma) \to (s'{,}\sigma') \Longrightarrow \exists\, T'.$
$E{,}\sigma' \models e{::}\,T' \wedge T' \preceq T$

even a stronger version (with type equality) holds.

Subject Reduction: array problem

```
class D extends C {
  void silly() {
    C[] a = new D[10];
    C   v = new D();
    a[2] = v;
  }
}
```

$$\frac{E,\sigma \models e_1[e_2] :: T \quad E,\sigma \models e_3 :: T' \quad T' \preceq T}{E,\sigma \models e_1[e_2] := e_3 :: T'}$$

Development of $E,\sigma \models e_1[e_2] := e_3 :: T'$:

e_1	T	e_3	T'	well-typed?
a	C	v	C	True
$\rightarrow$Addr $d1$	D	v	C	False
Addr $d1$	D	$\rightarrow$ Addr $d2$	D	True

Subject Reduction: array solution

Alternative 1: Evaluate rhs *before* lhs

$\ominus\ominus$ would alter semantics

Alternative 2: Relax intermediate typing rule to

$$\frac{E,\sigma \models a[i] :: T \quad E,\sigma \models v :: T'}{E,\sigma \models a[i] \,\texttt{:=}\, v :: T'}$$

$\ominus$ requires Array Property (of a term t):
Failure of widening in any array assignment within t can be detected

$a[i] \,\texttt{:=}\, e$ subterm of t $\wedge$ $E,\sigma \models a[i] :: T \wedge$
$E,\sigma \models e :: T' \wedge \neg\, T' \preceq T \Longrightarrow \exists n \geq 0\ C\ C'.$
$T = C[]_1 \ldots []_n \wedge\ T' = C'[]_1 \ldots []_n \wedge \neg\, C' \preceq C$

Subject Red.: conditional problem

```
interface I {...}
interface J {...}
class C implements I,J {...}
class D implements I,J {
  static Object m(Object o, J j) {
    return (b ? o : j);
  void test() {
    Object o = m (new C(), new D()); }
}
```

$$\frac{E{,}\sigma \models e_0 :: \texttt{boolean} \quad E{,}\sigma \models e_1 :: T_1 \quad E{,}\sigma \models e_2 :: T_2 \qquad \begin{array}{c} T_1 \preceq T_2 \wedge T{=}T_2 \;\vee \\ T_2 \preceq T_1 \wedge T{=}T_1 \end{array}}{E{,}\sigma \models e_0 \;?\; e_1 \;:\; e_2 :: T}$$

$(m(\texttt{new}\ C(),\ \texttt{new}\ D()),\sigma) \rightarrow \ldots \rightarrow$
$(m(\textsf{Addr}\ cc,\ \textsf{Addr}\ dd)\ ,\sigma') \rightarrow_\beta$
$(b\ ?\ \textsf{Addr}\ cc : \textsf{Addr}\ dd,\sigma') \rightarrow$

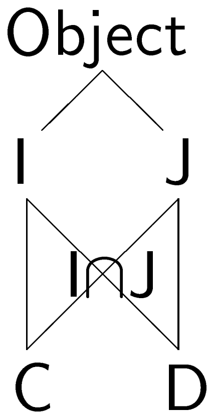
Compound Types

Alternative 1: Use most specific supertype

$\oplus\oplus$ ideal solution

$\ominus$ requires modifications of the type system

$$\frac{E,\sigma \models e_0 :: \texttt{boolean} \quad E,\sigma \models e_1 :: T_1 \quad E,\sigma \models e_2 :: T_2 \qquad T = \bigcap\{\, T.\ T_1 \preceq T \wedge T_2 \preceq T \}}{E,\sigma \models e_0\ ?\ e_1\ :\ e_2 :: T}$$



$E,\sigma \models b\ ?\ \mathsf{Addr}\ cc : \mathsf{Addr}\ dd :: I{\cap}J$

$I{\cap}J$ is called Compound Type [BW98]

Compound Types are ...

- anonymous reference types
- determined by their constituent structure
- useful also in component software development

Extended Beta Reduction

Alternative 2: Use beta-reduction with casts.

```
static Object m(Object o, J j) {...}
```

$(m(\texttt{new}\ C(), \texttt{new}\ D()),\sigma) \rightarrow \ldots \rightarrow$
$(m(\text{Addr}\ cc, \text{Addr}\ dd)\ ,\sigma') \rightarrow_{\beta *}$
$(b\ ?\ (\texttt{Object})\text{Addr}\ cc : (J)\text{Addr}\ dd,\sigma') \rightarrow$
$(\text{if}\ c\ \text{then}\ (\texttt{Object})\text{Addr}\ cc\ \text{else}\ (J)\text{Addr}\ dd,\sigma') \rightarrow$
$\ldots$

no run-time costs!
works accidentally: $\_\ ?\ \_\ :\ \_$ is non-strict

$\oplus$ simple solution: no changes to type system
$\ominus$ does not solve method overloading problem

Conditional problem: solutions

Alternative 3: Relax intermediate typing rule to

$$\frac{E,\sigma \models e_0 :: \texttt{boolean} \quad E,\sigma \models e_1 :: T_1 \quad E,\sigma \models e_2 :: T_2 \quad T_1 \preceq T \quad T_2 \preceq T}{E,\sigma \models e_0 \ ? \ e_1 \ : \ e_2 :: T}$$

$\ominus$ ambiguous intermediate typing

Alternative 4: Introduce type annotation

$$\frac{E,\sigma \models e_0 :: \texttt{boolean} \quad E,\sigma \models e_1 :: T_1 \quad E,\sigma \models e_2 :: T_2 \quad T_1 \preceq T \quad T_2 \preceq T}{E,\sigma \models e_0 \ ? \ \{T\} \ e_1 \ : \ e_2 :: T}$$

$\oplus$ preserves unique typing

Felleisen & Hieb Trick

Brute-force solution:
Annotate all terms with static type

Refined solution:
Annotate all variables with static type,
add assignments where needed

e.g. before method call:

```
void m (C x) { println(ol(x));}
```

becomes

```
void m (C x) { C Cx = x;
               println(ol(Cx)); }
```

⊕ general solution
⊖ artificial model

Subject Reduction: Preferred solution

Intermediate language with crucial annotations only

Relaxed internal typing rules reflect computational behavior

$$
\begin{array}{lccccccccc}
\text{Java} \supset & \text{Java}_s & \xrightarrow{elab} & \text{Java}_{se} & \subset & \text{Java}_r & \to & \text{Java}_r \\
 & \downarrow & & \downarrow & & \downarrow & & \downarrow \\
\text{Type:} & \tau & = & \tau & = & \{\tau \ldots\} & \succeq & \{\tau' \ldots\}
\end{array}
$$

Non-termination and Type-Safety

The Type Soundness theorems

- $\ldots \wedge \Gamma \vdash \sigma \; -e \triangleright v \rightarrow \; \sigma' \Longrightarrow \ldots$
- $\ldots \wedge \quad \neg \; \Gamma \vdash (e,\sigma) \Uparrow \quad \Longrightarrow \ldots$

do not address non-termination.

How about reactive programs?

— Establish Type-Safety for any finite prefix!

- Transition Semantics: (Subject Reduction)*
- Evaluation Semantics: Use counter & exception

Type-Safety: Conclusion

Transition Semantics involves intermediate steps.

This requires ...

- extending the term language
- extending all typing rules
- modifying some typing rules
- proving additional (heavy) lemmata

Thus:

Transition Semantics is intricate. Life with Evaluation Semantics is much easier!

References

[BW98] Martin Büchi and Wolfgang Weck. Java needs compound types. Technical Report 182, Turku Center for Computer Science, May 1998. `http://www.abo.fi/~mbuechi/publications/`

[DE98] Sophia Drossopoulou and Susan Eisenbach. Java is type safe — probably. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1??? of *LNCS*. Springer, 1998.

[ON98] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1??? of *LNCS*. Springer, 1998.

[Sym98] Donald Syme. Proving Java type soundness. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1??? of *LNCS*. Springer, 1998.