From IMP to Java

Andreas Lochbihler

ETH Zurich

parts based on work by Gerwin Klein and Tobias Nipkow

2015-07-14
1. Subtyping

2. Objects and Inheritance

3. Multithreading
1 Subtyping

2 Objects and Inheritance

3 Multithreading
1 Subtyping
Refresher on Typed IMP
IMP with Subtyping
Type safety

A programming language is *type safe* if the execution of a well-typed program cannot lead to certain errors.
Type safety

A programming language is *type safe* if the execution of a *well-typed* program cannot lead to certain errors.

Type system:

**Judgements** \( \Gamma \vdash a : \tau, \Gamma \vdash b, \Gamma \vdash c \)

**Conformance** \( \Gamma \vdash s \)
A programming language is *type safe* if the execution of a well-typed program cannot lead to *certain errors*.

**Type system:**

- Judgements $\Gamma \vdash a : \tau, \Gamma \vdash b, \Gamma \vdash c$

**Conformance** $\Gamma \vdash s$

**Type soundness**

- **Progress** Well-typed programs do not get stuck.
- **Preservation** Reductions preserve well-typedness and conformance.
IMP with Integers and Reals

Values:  **datatype** val = Iv int | Rv real

Arithmetic expressions:  **datatype** aexp =
  Ic int | Rc real | V vname | Plus aexp aexp
IMP with Integers and Reals

Values: \textbf{datatype} \( \text{val} = \text{Iv int} \mid \text{Rv real} \)

Arithmetic expressions: \textbf{datatype} \( \text{aexp} = \text{Ic int} \mid \text{Rc real} \mid \text{V vname} \mid \text{Plus aexp aexp} \)

\textcolor{red}{![Cannot add integers and reals.]}

**IMP with Integers and Reals**

Values:  
\[
\text{datatype } \text{val} = \text{Iv int} \mid \text{Rv real}
\]

Arithmetic expressions:  
\[
\text{datatype } \text{aexp} = \\
\text{Ic int} \mid \text{Rc real} \mid \text{V vname} \mid \text{Plus aexp aexp}
\]

\[\text{Cannot add integers and reals.}\]

Evaluation: inductive predicate \( \text{taval} \) (big-step)

\[
\text{taval (Ic } i \text{) s (Iv } i\text{)} \quad \text{taval (Rc } r \text{) s (Rv } r\text{)}
\]
IMP with Integers and Reals

Values: **datatype** \(\text{val} = \text{Iv int} \mid \text{Rv real}\)

Arithmetic expressions: **datatype** \(\text{aexp} = \text{Ic int} \mid \text{Rc real} \mid \text{V vname} \mid \text{Plus aexp aexp}\)

⚠ Cannot add integers and reals.

Evaluation: inductive predicate \(\text{taval}\) (big-step)

\[
\begin{align*}
\text{taval } (\text{Ic } i) & \quad \text{s } (\text{Iv } i) \\
\text{taval } (\text{Rc } r) & \quad \text{s } (\text{Rv } r) \\
\text{taval } a_1 & \quad \text{s } (\text{Iv } i_1) \\
\text{taval } a_2 & \quad \text{s } (\text{Iv } i_2) \\
\frac{\text{taval } (\text{Plus } a_1 \ a_2) & \quad \text{s } (\text{Iv } (i_1 + i_2))}
\end{align*}
\]
IMP with Integers and Reals

Values: \textbf{datatype} \textit{val} = \textit{Iv} \text{ int} \mid \textit{Rv} \text{ real}

Arithmetic expressions: \textbf{datatype} \textit{aexp} =
\hspace{1cm} \textit{Ic} \text{ int} \mid \textit{Rc} \text{ real} \mid \textit{V} \text{ vname} \mid \textit{Plus} \textit{aexp} \textit{aexp}

\textcolor{red}{!} Cannot add integers and reals.

Evaluation: inductive predicate \textit{taval} (big-step)

\[
\begin{align*}
\text{taval} \ (\textit{Ic} \ i) \ s & \ (\textit{Iv} \ i) & \ \text{taval} \ (\textit{Rc} \ r) \ s & \ (\textit{Rv} \ r) \\
\text{taval} \ a_1 \ s \ (\textit{Iv} \ i_1) & \ \text{taval} \ a_2 \ s \ (\textit{Iv} \ i_2) & \ \text{taval} \ (\textit{Plus} \ a_1 \ a_2) \ s \ (\textit{Iv} \ (i_1 + i_2)) \\
\text{taval} \ a_1 \ s \ (\textit{Rv} \ r_1) & \ \text{taval} \ a_2 \ s \ (\textit{Rv} \ r_2) & \ \text{taval} \ (\textit{Plus} \ a_1 \ a_2) \ s \ (\textit{Rv} \ (r_1 + r_2))
\end{align*}
\]
IMP with Integers and Reals

Values: \textbf{datatype} \textit{val} = \textit{Iv} \text{ int} \mid \textit{Rv} \text{ real}

Arithmetic expressions: \textbf{datatype} \textit{aexp} =
- \textit{Ic} \text{ int}
- \textit{Rc} \text{ real}
- \textit{V} \text{ vname}
- \text{ Plus} \textit{aexp} \textit{aexp}

\begin{itemize}
  \item Cannot add integers and reals.
\end{itemize}

Evaluation: inductive predicate \textit{taval} (big-step)

\[\begin{align*}
  \text{taval} \ (\text{Ic} \ i) \ s \ (\text{Iv} \ i) & \quad \text{taval} \ (\text{Rc} \ r) \ s \ (\text{Rv} \ r) \\
  \text{taval} \ a_1 \ s \ (\text{Iv} \ i_1) & \quad \text{taval} \ a_2 \ s \ (\text{Iv} \ i_2) \\
  & \quad \frac{\text{taval} \ (\text{Plus} \ a_1 \ a_2) \ s \ (\text{Iv} \ (i_1 + i_2))}{\text{taval} \ a_1 \ s \ (\text{Rv} \ r_1) & \quad \text{taval} \ a_2 \ s \ (\text{Rv} \ r_2) \\
  & \quad \frac{\text{taval} \ (\text{Plus} \ a_1 \ a_2) \ s \ (\text{Rv} \ (r_1 + r_2))}{\ldots}
\end{align*}\]
Typing Arithmetic Expressions

Ensure that addition operates either on $Ic$ or on $Rc$.!
Typing Arithmetic Expressions

Ensure that addition operates either on $Ic$ or on $Rc$!

Types $Ity$ and $Rty$ for integers and reals.
Type environment $\Gamma$ maps variable names to types.
Type judgement $\Gamma \vdash a : \tau$
Typing Arithmetic Expressions

Ensure that addition operates either on $Ic$ or on $Rc$!

Types $Ity$ and $Rty$ for integers and reals.
Type environment $\Gamma$ maps variable names to types.
Type judgement $\Gamma \vdash a : \tau$

$\Gamma \vdash Ic \, i : Ity \quad \Gamma \vdash Rc \, r : Rty$
Typing Arithmetic Expressions

Ensure that addition operates either on $Ic$ or on $Rc$.

Types $Ity$ and $Rty$ for integers and reals.

Type environment $\Gamma$ maps variable names to types.

Type judgement $\Gamma \vdash a : \tau$

\[ \Gamma \vdash Ic \, i : Ity \quad \Gamma \vdash Rc \, r : Rty \]

\[ \Gamma \vdash V \, x : \Gamma \, x \]
Typing Arithmetic Expressions

Ensure that addition operates either on \( Ic \) or on \( Rc \)!

Types \( Ity \) and \( Rty \) for integers and reals.
Type environment \( \Gamma \) maps variable names to types.
Type judgement \( \Gamma \vdash a : \tau \)

\[
\Gamma \vdash Ic \; i : Ity \quad \Gamma \vdash Rc \; r : Rty
\]

\[
\Gamma \vdash V \; x : \Gamma \; x
\]

\[
\Gamma \vdash a_1 : \tau \quad \Gamma \vdash a_2 : \tau
\]

\[
\Gamma \vdash Plus \; a_1 \; a_2 : \tau
\]
Type safety for arithmetics

Type of a value  \( \text{type } (Iv \ i) = Ity \quad \text{type } (Rv \ r) = Rty \)

Conformance  \( (\Gamma \vdash s) = (\forall x. \text{type } (s \ x) = \Gamma \ x) \)
Type safety for arithmetics

Type of a value  \( \text{type} (Iv \ i) = Ity \quad \text{type} (Rv \ r) = Rty \)

Conformance  \( (\Gamma \vdash s) = (\forall x. \text{type} (s \ x) = \Gamma \ x) \)

Progress  \( [\Gamma \vdash a : \tau; \ \Gamma \vdash s] \implies \exists v. \ \text{taval} \ a \ s \ v \)

We can use big-step style “there exists a result \( v \)” as evaluation always terminates.
Type safety for arithmetics

Type of a value \[ \text{type} (Iv \ i) = Ity \quad \text{type} (Rv \ r) = Rty \]

Conformance \[ (\Gamma \vdash s) = (\forall x. \text{type} (s \ x) = \Gamma \ x) \]

Progress \[ \frac{}{[\Gamma \vdash a : \tau; \Gamma \vdash s] \Rightarrow \exists v. \text{taval} a \ s \ v} \]
We can use big-step style “there exists a result \( v \)” as evaluation always terminates.

Preservation \[ \frac{}{[\Gamma \vdash a : \tau; \text{taval} a \ s \ v; \Gamma \vdash s] \Rightarrow \text{type} v = \tau} \]
Preservation of conformance is trivial as state does not change.
From $\text{aexp}$ to $\text{com}$

$bexp$  Judgement $\Gamma \vdash b$

Progress: $\left[\Gamma \vdash b; \Gamma \vdash s\right] \implies \exists v. \, \text{tbval} \ b \ s \ v$
From $aexp$ to $com$

$bexp$ Judgement $\Gamma \vdash b$

Progress: $[\Gamma \vdash b; \Gamma \vdash s] \implies \exists v. \ \text{tbval} \ b \ s \ v$

$com$ Judgement $\Gamma \vdash c$
From \textit{aexp} to \textit{com}

\textbf{bexp} Judgement $\Gamma \vdash b$

Progress: $[\Gamma \vdash b; \Gamma \vdash s] \Longrightarrow \exists v. \; \text{tbval} \; b \; s \; v$

\textbf{com} Judgement $\Gamma \vdash c$

Progress: (small-step style)

$[\Gamma \vdash c; \Gamma \vdash s; \; c \neq \text{SKIP}] \Longrightarrow \exists cs'. \; (c, s) \rightarrow cs'$
From $aexp$ to $com$

\text{bexp} \quad \text{Judgement} \quad \Gamma \vdash b

\text{Progress:} \quad \llbracket \Gamma \vdash b; \Gamma \vdash s \rrbracket \Rightarrow \exists \, v. \, \text{tbval} \, b \, s \, v

\text{com} \quad \text{Judgement} \quad \Gamma \vdash c

\text{Progress: (small-step style)}
\llbracket \Gamma \vdash c; \Gamma \vdash s; \, c \neq \text{SKIP} \rrbracket \Rightarrow \exists \, cs'. \, (c, \, s) \rightarrow cs'

\text{Preservation:}
\llbracket (c, \, s) \rightarrow (c', \, s')\;; \Gamma \vdash c \rrbracket \Rightarrow \Gamma \vdash c'
\llbracket (c, \, s) \rightarrow (c', \, s')\;; \Gamma \vdash c; \Gamma \vdash s \rrbracket \Rightarrow \Gamma \vdash s'
From \textit{aexp} to \textit{com}

\textbf{bexp} Judgement $\Gamma \vdash b$

Progress: $[[\Gamma \vdash b; \Gamma \vdash s]] \Rightarrow \exists v. \text{tbval} b s v$

\textbf{com} Judgement $\Gamma \vdash c$

Progress: (small-step style)

$[[\Gamma \vdash c; \Gamma \vdash s; c \neq \text{SKIP}]] \Rightarrow \exists cs'. (c, s) \rightarrow cs'$

Preservation:

$[[ (c, s) \rightarrow (c', s'); \Gamma \vdash c ]] \Rightarrow \Gamma \vdash c'$

$[[ (c, s) \rightarrow (c', s'); \Gamma \vdash c; \Gamma \vdash s ]] \Rightarrow \Gamma \vdash s'$

Type soundness

$[[ (c, s) \rightarrow^* (c', s'); \Gamma \vdash c; \Gamma \vdash s; c' \neq \text{SKIP}]]$

$\Rightarrow \exists cs''. (c', s') \rightarrow cs''$
Important points

1. Type systems are *syntax-directed*, i.e., decidable.
2. Well-typed programs cannot *go wrong* (R. Milner).
3. Errors are modeled as the semantics getting stuck. Big-step semantics usable if termination is guaranteed.
4. Sound type systems are necessarily incomplete.
Subtyping

Refresher on Typed IMP

IMP with Subtyping
Crossing type boundaries

Types partition values into disjoint sets with no interaction.

Example: We cannot use an integer as a real. We could write a program to convert an integer to a real, of course, but ...

Subtyping arranges types in a hierarchy. Key principle: Subtypes may be used whenever a supertype is required (with respect to whatever the type system checks). Java: Interface is supertype of the implementing class.
Crossing type boundaries

Types partition values into disjoint sets with no interaction.

**Example:** We cannot use a integer $Iv$ as a real $Rv$. We could write a program to convert an integer to a real, of course, but ...
Crossing type boundaries

Types partition values into disjoint sets with no interaction.

Example: We cannot use a integer $Iv$ as a real $Rv$. We could write a program to convert an integer to a real, of course, but ...

Subtyping arranges types in a hierarchy. Key principle *substitutability*: Subtypes may be used whenever a supertype is required (with respect to whatever the type system checks).
Crossing type boundaries

Types partition values into disjoint sets with no interaction.

Example: We cannot use an integer $Iv$ as a real $Rv$. We could write a program to convert an integer to a real, of course, but ...

Subtyping arranges types in a hierarchy.
Key principle *substitutability*: Subtypes may be used whenever a supertype is required (with respect to whatever the type system checks).

Java: Interface is supertype of the implementing class.
Subtyping for Ints and Reals

How can we arrange \textit{Ity} and \textit{Rty} in a subtyping relation?

- \textit{Ity} should be a subtype of \textit{Rty}
- \textit{Rty} should be a subtype of \textit{Ity}
Subtyping for Ints and Reals

How can we arrange \( Ity \) and \( Rty \) in a subtyping relation?

• \( Ity \) should be a subtype of \( Rty \)
• \( Rty \) should be a subtype of \( Ity \)

Single-step subtype relation \( \sqsubseteq_1 \): \( Ity \sqsubseteq_1 Rty \)

Subtype relation \( \sqsubseteq \) is the reflexive, transitive closure of \( \sqsubseteq_1 \).

Examples: \( Ity \sqsubseteq Rty \) and \( Ity \sqsubseteq Ity \) but \( Rty \not\sqsubseteq Ity \)
Changes to Type System

1. Add a subsumption rule

\[
\frac{\Gamma \vdash a : \tau \quad \tau \sqsubseteq \tau'}{\Gamma \vdash a : \tau'}
\]
Changes to Type System

1. Add a subsumption rule

   \[
   \Gamma \vdash a : \tau \\
   \tau \sqsubseteq \tau' \\
   \hline
   \Gamma \vdash a : \tau'
   \]

   or

2. Allow different types for operands of \textit{Plus} and \textit{Less}.

   \[
   \Gamma \vdash a_1 : \tau_1 \\
   \Gamma \vdash a_2 : \tau_2 \\
   \hline
   \Gamma \vdash \text{Plus} \ a_1 \ a_2 : \text{max} \sqsubseteq \tau_1 \ \tau_2
   \]

   \[
   \Gamma \vdash a_1 : \tau_1 \\
   \Gamma \vdash a_2 : \tau_2 \\
   \tau_1 \sqsubseteq \text{Rty} \quad \tau_2 \sqsubseteq \text{Rty} \\
   \hline
   \Gamma \vdash \text{Less} \ a_1 \ a_2
   \]

Trade-offs like in the security type setting.
Changes to Type System

1. Add a subsumption rule

   \[
   \Gamma \vdash a : \tau \quad \tau \sqsubseteq \tau' \\
   \hline
   \Gamma \vdash a : \tau'
   \]

   or

2. Allow different types for operands of \textit{Plus} and \textit{Less}.

   \[
   \begin{align*}
   \Gamma \vdash a_1 : \tau_1 & \quad \Gamma \vdash a_2 : \tau_2 \\
   \hline
   \Gamma \vdash Plus a_1 a_2 : \max \sqsubseteq \tau_1 \tau_2
   \end{align*}
   \]

   \[
   \begin{align*}
   \Gamma \vdash a_1 : \tau_1 & \quad \Gamma \vdash a_2 : \tau_2 \quad \tau_1 \sqsubseteq \text{Rty} \quad \tau_2 \sqsubseteq \text{Rty} \\
   \hline
   \Gamma \vdash Less a_1 a_2
   \end{align*}
   \]

Trade-offs like in the security type setting.
Assume that $\Gamma \ "x" = Rty$. Then,

\[
\Gamma \vdash "x" ::= Plus (Rc 1.0) (Ic 2)
\]
Example

Assume that $\Gamma "x" = Rty$. Then,

\[
\Gamma \vdash Plus (Rc 1.0) (Ic 2) : \Gamma "x" = Rty
\]

\[
\Gamma \vdash "x" ::= Plus (Rc 1.0) (Ic 2)
\]
Assume that $\Gamma \ "x" = Rty$. Then,

$$
\Gamma \vdash Rc\ 1.0 : Rty \quad \Gamma \vdash Ic\ 2 : Rty
$$

\[
\frac{
\Gamma \vdash Plus\ (Rc\ 1.0)\ (Ic\ 2) : \Gamma \ "x" = Rty
}{
\Gamma \vdash \ "x" ::= Plus\ (Rc\ 1.0)\ (Ic\ 2)
}\]
Example

Assume that $\Gamma "x" = Rty$. Then,

\[
\begin{array}{c}
\Gamma \vdash Rc\ 1.0 : Rty \\
\Gamma \vdash Ic\ 2 : Ity \\
\text{Ity} \sqsubseteq Rty \\
\end{array}
\]

\[
\frac{
\Gamma \vdash Ic\ 2 : Rty \\
\Gamma \vdash "x" = Rty \\
\Gamma \vdash Plus\ (Rc\ 1.0)\ (Ic\ 2) : \Gamma "x" = Rty \\
}{
\Gamma \vdash "x" ::= Plus\ (Rc\ 1.0)\ (Ic\ 2)
} \]
Changes to the Semantics

Addition and comparison must handle mixed operands.

New rules: (real converts int to real)

\[
\begin{align*}
\text{taval } a_1 & \quad s \quad (Rv \ r_1) & \text{taval } a_2 & \quad s \quad (Iv \ i_2) \\
\text{taval } (\text{Plus} \ a_1 \ a_2) & \quad s \quad (Rv \ (r_1 + \text{real} \ i_2)) \\
\text{taval } a_1 & \quad s \quad (Iv \ i_1) & \text{taval } a_2 & \quad s \quad (Rv \ r_2) \\
\text{taval } (\text{Plus} \ a_1 \ a_2) & \quad s \quad (Rv \ (\text{real} \ i_1 + r_2))
\end{align*}
\]
Changes to the Semantics

Addition and comparison must handle mixed operands.

New rules: (\textit{real converts int to real})

\[
\begin{align*}
\text{taval } a_1 & \text{ s } (Rv \; r_1) \quad \text{taval } a_2 & \text{ s } (Iv \; i_2) \\
\text{taval } (\text{Plus } a_1 & \; a_2) & \text{ s } (Rv \; (r_1 + \text{real } i_2)) \\
\text{taval } a_1 & \text{ s } (Iv \; i_1) \quad \text{taval } a_2 & \text{ s } (Rv \; r_2) \\
\text{taval } (\text{Plus } a_1 & \; a_2) & \text{ s } (Rv \; (\text{real } i_1 + r_2)) \\
\text{taval } a_1 & \text{ s } (Iv \; i_1) \quad \text{taval } a_2 & \text{ s } (Rv \; r_2) \\
\text{tbval } (\text{Less } a_1 & \; a_2) & \text{ s } (\text{real } i_1 < r_2) \\
\text{taval } a_1 & \text{ s } (Rv \; r_1) \quad \text{taval } a_2 & \text{ s } (Iv \; i_2) \\
\text{tbval } (\text{Less } a_1 & \; a_2) & \text{ s } (r_1 < \text{real } i_2)
\end{align*}
\]
Conformance

Before: \[ \Gamma \vdash s \iff (\forall x. \text{type} (s \ x) = \Gamma \ x) \]

Conformance is not preserved any more:
\[ \Gamma "x" = Rty, \ s "x" = Rv \ 1.0, \text{ and } \Gamma \vdash "x" ::= Ic \ 2, \text{ and } 
("x" ::= Ic \ 2, \ s) \rightarrow (\text{SKIP, } s("x" ::= Iv \ 2)) \]
but not \[ \Gamma \vdash s("x" ::= Iv \ 2) \]
Conformance

Before: \( \Gamma \vdash s \iff (\forall x. \text{type}(s \ x) = \Gamma \ x) \)

Conformance is not preserved any more:
\( \Gamma \ "x" = Rty, s \ "x" = Rv \ 1.0, \text{ and } \Gamma \vdash \ "x" ::= Ic \ 2, \text{ and } ("x" ::= Ic \ 2, s) \rightarrow (\text{SKIP}, s("x" ::= Iv \ 2)) \)
but not \( \Gamma \vdash s("x" ::= Iv \ 2) \)

Weaken conformance notion (subtype instead of =):
for values \( v : \leq \tau \iff \text{type} \ v \sqsubseteq \tau \)
for states \( \Gamma \vdash s \iff (\forall x. \ s \ x : \leq \Gamma \ x) \)
Subject reduction

Previously preservation of types:

\[
[\Gamma \vdash a : \tau; taval a s v; \Gamma \vdash s] \implies \text{type } v = \tau
\]
Subject reduction

Previously preservation of types:

\[
[\Gamma \vdash a : \tau; \text{taval} a \ s \ v; \Gamma \vdash s] \implies \text{type} \ v = \tau
\]

Now \textit{subject reduction}: Type can become more specific during evaluation.

\[
[\Gamma \vdash a : \tau; \text{taval} a \ s \ v; \Gamma \vdash s] \implies \text{type} \ v \sqsubseteq \tau
\]
Subject reduction

Previously preservation of types:

\[ \Gamma \vdash a : \tau; \ taval \ a \ s \ v; \ \Gamma \vdash s \Rightarrow \text{type} \ v = \tau \]

Now subject reduction:
Type can become more specific during evaluation.

\[ \Gamma \vdash a : \tau; \ taval \ a \ s \ v; \ \Gamma \vdash s \Rightarrow \text{type} \ v \sqsubseteq \tau \]

Example: Let \( \Gamma \ "x" = Rty \), so \( \Gamma \vdash V "x" : Rty \).
Let \( s "x" = Iv 1 \), so \( taval \ (V "x") \ s \ (Iv 1) \) and \( \Gamma \vdash s \).
But \( \text{type} \ (Iv 1) = Ity \).
Progress

Progress theorems stay the same, but the proofs have more cases as there are more rules.

\[ [\Gamma \vdash a : \tau; \Gamma \vdash s] \implies \exists v. \text{taval} a s v \]
\[ [\Gamma \vdash b; \Gamma \vdash s] \implies \exists v. \text{tbval} b s v \]
\[ [\Gamma \vdash c; \Gamma \vdash s; c \neq \text{SKIP}] \implies \exists cs'. (c, s) \rightarrow cs' \]
Tagging and compilation

Without subtyping, tags on values were used to detect when something goes wrong. Now, the semantics relies on them to insert coercion real.

Do we need tags in the compiled code, too?
Tagging and compilation

Without subtyping, tags on values were used to detect when something goes wrong. Now, the semantics relies on them to insert coercion \textit{real}.

Do we need tags in the compiled code, too?

\textbf{In general} Yes (for dynamic method lookup and casts).

\textbf{Here} No. Compiler can use type system to statically determine where to insert coercions.

\textbf{!} In practice, integers and reals are dealt by implicit coercions, not subtyping.
1. Subtyping

2. Objects and Inheritance

3. Multithreading
Objects and Inheritance

Modelling Objects and Classes

Inheritance and Subtyping

Dynamic and Static Binding
So far, only atomic values \textit{int} and \textit{real}.

Classes have several fields identified by names \textit{fname}.

```java
class C {
    int x;
    byte y;
    C z;

    { z = new C();
      z.x = this.y + 3;
    }
}
```
So far, only atomic values \textit{int} and \textit{real}.

Classes have several fields identified by names \textit{fname}.

Need new syntax for

- allocation \textit{new} \textit{cname}
- field access \textit{exp}·\textit{fname}
- field update \textit{exp}·\textit{fname} := \textit{exp}

```

class C {
    int x;
    byte y;
    C z;

    { z = new C();
      z.x = this.y + 3;
    }
}
```
So far, only atomic values \textit{int} and \textit{real}.

Classes have several fields identified by names \textit{fname}.

Need new syntax for

- allocation: \textit{new} \textit{cname}
- field access: \textit{exp} \cdot \textit{fname}
- field update: \textit{exp} \cdot \textit{fname} := \textit{exp}

New type of value \textit{obj}:

\textbf{datatype} \textit{obj} = Object \textit{cname} (\textit{fname} \Rightarrow \textit{val} \textit{option})
So far, only atomic values *int* and *real*. Classes have several fields identified by names *fname*. Need new syntax for

- allocation: `new cname`
- field access: `exp·fname`
- field update: `exp·fname := exp`

New type of value *obj*:

```plaintext
datatype obj = Object cname (fname ⇒ val option)
```

Everything depends on a *program declaration*

```plaintext
P :: program = (cname × (fname × ty) list) list
```
References and the Heap

Objects can only be stored on the heap. They are referenced by addresses, a new kind of value.

\[ \text{heap} = \text{addr} \Rightarrow \text{obj option} \]

**datatype** \( \text{val} = \text{Iv int} \mid \text{Bv bool} \mid \text{Addr addr} \mid \ldots \)
References and the Heap

Objects can only be stored on the heap. They are referenced by addresses, a new kind of value.

$$heap = addr \Rightarrow obj \; option$$

**datatype** $val = Iv \; int \mid Bv \; bool \mid Addr \; addr \mid \ldots$

State consists of the heap and a store for local variables:

$$state = heap \times locals$$

$$locals = vname \Rightarrow val \; option$$
Every class becomes a type.

\[
\text{datatype } ty = Ity \mid Bty \mid \text{Class } \text{cname} \mid \ldots
\]
Every class becomes a type.

**datatype** \( ty = Ity \mid Bty \mid Class\ cname \mid \ldots \)

The type of \( Addr\ a \) is the class of the object stored at \( a \).
Typing

Every class becomes a type.

\[
\text{datatype } ty = \text{Ity} \mid \text{Bty} \mid \text{Class }\text{ cname} \mid \ldots
\]

The type of \(\text{Addr } a\) is the class of the object stored at \(a\).

Typing rule checks:

\[
\text{new } C
\]
Typing

Every class becomes a type.

\[
\text{datatype } ty = \text{Ity} \mid \text{Bty} \mid \text{Class } \text{cname} \mid \ldots
\]

The type of \(Addr \ a\) is the class of the object stored at \(a\).

Typing rule checks:

\[
\text{new } C \quad \text{Class } C \text{ is declared in } P.
\]

\[
e \cdot F
\]
Typing

Every class becomes a type.

**datatype** \( ty = Ity \mid Bty \mid Class \text{ } cname \mid \ldots \)

The type of \( Addr \ a \) is the class of the object stored at \( a \).

Typing rule checks:

- **new C** Class C is declared in \( P \).
- \( e \cdot F \): The type of \( e \) is a class which declares field \( F \).
- \( e \cdot F := e' \): The type of \( e \) is a class which declares field \( F \),
Typing

Every class becomes a type.

**datatype** \( ty = Ity | Bty | Class \text{\ lowercase\ } cname | \ldots \)

The type of \( Addr \ a \) is the class of the object stored at \( a \).

Typing rule checks:

\[ new \ C \ \text{Class } C \text{ is declared in } P. \]

\[ e \cdot F \ \text{The type of } e \text{ is a class which declares field } F. \]

\[ e \cdot F := e' \ \text{The type of } e \text{ is a class which declares field } F, \]
\[ \text{and type of } e' \text{ is a subtype of the declared type.} \]
Typing

Every class becomes a type.

\textbf{datatype} \; ty = \; Ity \; | \; Bty \; | \; Class \; cname \; | \; \ldots \\

The type of \(Addr\) \(a\) is the class of the object stored at \(a\).

Typing rule checks:

\textit{new C} \; Class \; C \; is \; declared \; in \; P.

\(e \cdot F\) \; The \; type \; of \; \(e\) \; is \; a \; class \; which \; declares \; field \; \(F\).

\(e \cdot F \; := \; e'\) \; The \; type \; of \; \(e\) \; is \; a \; class \; which \; declares \; field \; \(F\), \nand \; type \; of \; \(e'\) \; is \; a \; subtype \; of \; the \; declared \; type.

Consequences:

- Type of \(e\) \; depends \; on \; the \; heap \; if \; \(e\) \; contains \(Addr\).
- Programs \; must \; not \; contain \; literal \; addresses.
new $C$

Semantics
new $C$

1. find a fresh address in the heap,
new $C$

1. find a fresh address in the heap,
2. initialise the object’s fields with default values

e \cdot F
Semantics

\textit{new} \textit{C}

1. find a fresh address in the heap,
2. initialise the object’s fields with default values

\textit{e} \cdot \textit{F}

1. evaluate \textit{e} to value \textit{Addr a}
2. lookup field \textit{F} in object at address \textit{a}
Semantics

new \( C \)

1. find a fresh address in the heap,
2. initialise the object’s fields with default values

\( e \cdot F \)

1. evaluate \( e \) to value \( \text{Addr} \ a \)
2. lookup field \( F \) in object at address \( a \)

\( e \cdot F := e' \)

1. evaluate \( e \) to value \( \text{Addr} \ a \)
2. evaluate \( e' \) to value \( v \).
3. store \( v \) in field \( F \) of object on heap at address \( a \)
Type safety

Heap conformance $hconf\ h$: Each object on the heap $h$

1. refers to a declared class,
2. stores values for all declared fields, and
3. every stored value conforms to the field’s type.
Type safety

Heap conformance \( hconf h \): Each object on the heap \( h \)

1. refers to a declared class,
2. stores values for all declared fields, and
3. every stored value conforms to the field’s type.

Monotonicity:
Typeability is preserved under changes to the heap.
Type safety

Heap conformance $hconf \ h$: Each object on the heap $h$

1. refers to a declared class,
2. stores values for all declared fields, and
3. every stored value conforms to the field’s type.

Monotonicity:
Typeability is preserved under changes to the heap.

Progress + preservation ensure that

1. all required classes are available, and
2. only existing fields of objects can be accessed (no memory corruption).
Methods

Adaptations are similar to those for fields:

**Program** add lists of method declarations to classes

\[
mdecl = mname \times (vname \times ty) list \times ty \times com
\]

**Syntax** for method calls \( e \cdot M(es) \)
Methods

Adaptations are similar to those for fields:

**Program** add lists of method declarations to classes

\[ m\text{decl} = m\text{name} \times (v\text{name} \times ty) \text{ list} \times ty \times com \]

**Syntax** for method calls \( e \cdot M(es) \)

**Typing** check that \( M \) exists in class of callee \( e \)

and parameters \( es \) conform to the declared types.
Methods

Adaptations are similar to those for fields:

Program  add lists of method declarations to classes

\[ mdecl = \text{mname} \times (\text{vname} \times \text{ty}) \text{list} \times \text{ty} \times \text{com} \]

Syntax  for method calls  \( e \cdot M(es) \)

Typing  check that  \( M \) exists in class of callee  \( e \)  
and parameters  \( es \) conform to the declared types.

Semantics  evaluate  \( e \) and  \( es \), 
lookup method body  \( c \) and execute  \( c \).

Local scopes for variables needed!

Type safety  shows that every method called exists.
2 Objects and Inheritance
   Modelling Objects and Classes
   Inheritance and Subtyping
   Dynamic and Static Binding
Every class $D$ has a *superclass* $C$ (except `Object`).

class C {
  int f;
  int m1(int x) { ... }
}
class D extends C {
  int g;
  bool m2(int y) { ... }
}
Every class $D$ has a **superclass** $C$ (except Object).

```java
class C {
    int f;
    int m1(int x) { ... }
}
class D extends C {
    int g;
    bool m2(int y) { ... }
}
```

The subclass (transitively) **inherits** all fields and methods from its superclasses.

```java
D d = new D();
d.m1(5);
d.f = 2;
```
Every class $D$ has a **superclass** $C$ (except `Object`).

```java
class C {
    int f;
    int m1(int x) { ... }
}
class D extends C {
    int g;
    bool m2(int y) { ... }
}
```

The subclass (transitively) **inherits** all fields and methods from its superclasses.

```java
D d = new D();
d.m1(5);
d.f = 2;
```

Formalise direct subclass relationship $D \preceq_1 C$. 
Subtyping

class C {
    int f;
    int m1(int x) { ... }
}

class D extends C {
    int g;
    bool m2(int y) { ... }
}

C c = new D();     // implicit upcast

Substitutability:
Subclass can be used instead of superclass.
Subtyping

```java
class C {
    int f;
    int m1(int x) { ... }
}

class D extends C {
    int g;
    bool m2(int y) { ... }
}

C c = new D(); // implicit upcast
```

**Substitutability:**
Subclass can be used instead of superclass.

If $D \prec_1 C$, then $D \sqsubseteq_1 C$. 
Subtyping

class C {
    int f;
    int m1(int x) { ... }
}
class D extends C {
    int g;
    bool m2(int y) { ... }
}

C c = new D(); // implicit upcast

Substitutability:
Subclass can be used instead of superclass.

If $D \prec_1 C$, then $D \sqsubseteq_1 C$.

New requirements on program declarations:

1. Superclass must exist.
2. Subclass hierarchy must be acyclic.
2 Objects and Inheritance
   Modelling Objects and Classes
   Inheritance and Subtyping
   Dynamic and Static Binding
Example

What does the following Java program print?

class C {
    int x = 1;
    int f() { return 3; }
}

class D extends C {
    int x = 2;
    int f() { return 5; }
}

C c = new D(); // implicit upcast
System.out.print(c.x * c.f());
What does the following Java program print? 5

class C {
    int x = 1;
    int f() { return 3; }
}

class D extends C {
    int x = 2;
    int f() { return 5; }
}

C c = new D();  // implicit upcast
System.out.print(c.x * c.f());

Field lookup is static:
Search for field name starts at class determined before execution.
Example

What does the following Java program print? 5

class C {
    int x = 1;
    int f() { return 3; }
}
class D extends C {
    int x = 2;
    int f() { return 5; }
}

C c = new D(); // implicit upcast
System.out.print(c.x * c.f());

Field lookup is static:
Search for field name starts at class determined before execution.

Method lookup is dynamic:
Search for method name starts at actual class of the callee at runtime.
Dynamic Methods Calls

Semantics of $e \cdot M(es)$

1. Evaluate $e$ to $Addr\: a$ and $es$ to values $vs$.
2. Lookup method $M$ of class $C$ of object at address $a$.
3. Bind $this$ to $Addr\: a$ and formal parameters to $vs$.
4. Execute method body.

Type safety shows:
• Lookup always finds a unique method declaration.
• Downcast of $this$ pointer is safe in step 3.

Requirements:
1. Overwriting method generalises parameter types and specialises result type.
Dynamic Methods Calls

Semantics of $e \cdot M(es)$

1. Evaluate $e$ to Addr $a$ and $es$ to values $vs$.
2. Lookup method $M$ of class $C$ of object at address $a$.
3. Bind $this$ to Addr $a$ and formal parameters to $vs$.
4. Execute method body.

Type safety shows:
- Lookup always finds a unique method declaration.
- Downcast of $this$ pointer is safe in step 3.
Dynamic Methods Calls

Semantics of $e \cdot M(es)$

1. Evaluate $e$ to $Addr\ a$ and $es$ to values $vs$.
2. Lookup method $M$ of class $C$ of object at address $a$.
3. Bind $this$ to $Addr\ a$ and formal parameters to $vs$.
4. Execute method body.

Type safety shows:
- Lookup always finds a unique method declaration.
- Downcast of $this$ pointer is safe in step 3.

Requirements:
1. Overwriting method generalises parameter types and specialises result type.
Static Fields

Fields are hidden, not overwritten. Output:

class C {
    int x = 1;
}
class D extends C {
    int x = 2;
}

D d = new D();
C c = d;  // implicit upcast
System.out.print(c.x + d.x);
Static Fields

Fields are hidden, not overwritten. Output: 3

class C {
    int x = 1;
}

class D extends C {
    int x = 2;
}

d = new D();
c = d;  // implicit upcast
System.out.println(c.x + d.x);
Static Fields

Fields are hidden, not overwritten. Output: 3

class C {
    int x = 1;
}
class D extends C {
    int x = 2;
}

D d = new D();
C c = d;  // implicit upcast
System.out.print(c.x + d.x);

Objects on the heap must distinguish
  • field x declared in C and
  • field x declared in D.

Solution: Include declaring class name in key.

datatype obj = Object cname (cname x fname ⇒ val option)
Static Fields

Fields are hidden, not overwritten. Output: 3

class C {
    int x = 1;
}
class D extends C {
    int x = 2;
}

D d = new D();
C c = d;       // implicit upcast
System.out.print(c.x + d.x);

- Semantics must know declaring class at run-time,
- but actual object may be from a subclass,
- so program code must keep track of static type information.
- **Annotate** with class name:
  \[ e \cdot F\{C\} \text{ and } e \cdot F\{C\} := e' \]
Fields are hidden, not overwritten. Output: 3

class C {
    int x = 1;
}
class D extends C {
    int x = 2;
}

D d = new D();
C c = d;                        // implicit upcast
System.out.print(c.x + d.x);

Consequences:

- **Preprocessor** annotates field access before start.
- Type system must compute most specific class.
- Subsumption rule cannot be used.
Summary about Objects

Take-away message:

- Fixed program context + big lookup machinery
- Type safety shows that called methods exist and memory is not corrupted.
- Static binding via annotations
Summary about Objects

Take-away message:

- Fixed program context + big lookup machinery
- Type safety shows that called methods exist and memory is not corrupted.
- Static binding via annotations

More OO aspects:

1. Null pointers (exceptions)
2. Multiple inheritance (casts are not for free)
3. Arrays and subtyping (ArrayStoreException)
   
   ```
   ((Object[]) new String[1])[0] := new Object();
   ```
4. Inner classes (complicated lookup rules)
1. Subtyping

2. Objects and Inheritance

3. Multithreading
A *thread* is a command with local state. It *executes in parallel* with other commands.
Threads in Java

A *thread* is a command with local state. It *executes in parallel* with other commands.

Every Java thread has an object associated to it. The thread is controlled via its methods:

- **run** command to be executed by the thread
- **start** start the execution of the thread
- **join** wait for the thread to terminate

```java
Thread t = new Thread () {
    public void run () { System.out.print("foo"); }
};
t.start();
System.out.print("bar");
t.join();
```
A **thread** is a command with local state. It **executes in parallel** with other commands.

Every Java thread has an object associated to it. The thread is controlled via its methods:

- **run** command to be executed by the thread
- **start** start the execution of the thread
- **join** wait for the thread to terminate

```java
Thread t = new Thread() {
    public void run() {
        System.out.print("foo");
    }
};
t.start();
System.out.print("bar");
t.join();
```
Interleaving semantics

Threads require a **small-step semantics**. Each step $\rightarrow$ is atomic. Steps of threads are interleaved.

$$
\begin{align*}
    t_1 \\
    t_2
\end{align*}
$$
Threads require a **small-step semantics**. Each step $\rightarrow$ is atomic. Steps of threads are interleaved.

\[
\begin{align*}
  t_1 & \rightarrow \\
  t_2 & \rightarrow
\end{align*}
\]
Interleaving semantics

Threads require a small-step semantics. Each step $\rightarrow$ is atomic. Steps of threads are interleaved.

\[
\begin{align*}
t_1 \rightarrow \\
t_2 \rightarrow
\end{align*}
\]
Interleaving semantics

Threads require a small-step semantics. Each step $\rightarrow$ is atomic. Steps of threads are interleaved.

\[
\begin{align*}
t_1 & \rightarrow \rightarrow \\
\rightarrow \\
t_2 & \rightarrow
\end{align*}
\]
Interleaving semantics

Threads require a **small-step semantics**. Each step $\rightarrow$ is atomic. Steps of threads are interleaved.

\[
\begin{align*}
  t_1 & \rightarrow \rightarrow \\
  t_2 & \rightarrow
\end{align*}
\]
Interleaving semantics

Threads require a **small-step semantics**. Each step $\rightarrow$ is atomic. Steps of threads are interleaved.

$$
\begin{align*}
t_1 & \rightarrow \rightarrow \rightarrow \\
t_2 & \rightarrow \rightarrow 
\end{align*}
$$
Interleaving semantics

Threads require a small-step semantics. Each step $\rightarrow$ is atomic. Steps of threads are interleaved.

\[
t_1 \quad \rightarrow \quad \rightarrow \quad \rightarrow \\
t_2 \quad \rightarrow \quad \rightarrow \
\]
Interleaving semantics

Threads require a small-step semantics. Each step $\rightarrow$ is atomic. Steps of threads are interleaved.

$t_1 \rightarrow \rightarrow \rightarrow$

$t_2 \rightarrow \rightarrow \rightarrow$
Interleaving semantics

Threads require a small-step semantics. Each step $\rightarrow$ is atomic. Steps of threads are interleaved.

\[
\begin{align*}
t_1 & \rightarrow \rightarrow \rightarrow \rightarrow \\
t_2 & \rightarrow \rightarrow \rightarrow \rightarrow \\
\end{align*}
\]
Interleaving semantics

Threads require a small-step semantics. Each step $\rightarrow$ is atomic. Steps of threads are interleaved.

$$t_1 \rightarrow \rightarrow \rightarrow \rightarrow$$

$$t_2 \rightarrow \rightarrow \rightarrow \rightarrow$$

Thread pool  \( pool = tid \Rightarrow (com \times locals) \ option \)
Threads require a **small-step semantics**. Each step $\rightarrow$ is atomic. Steps of threads are interleaved.

$$
\begin{array}{c}
  t_1 \rightarrow \rightarrow \rightarrow \\
  t_2 \rightarrow \rightarrow \rightarrow \\
\end{array}
$$

**Thread pool** $pool = tid \Rightarrow (com \times locals) \ option$

**State** $mstate = pool \times heap$
Interleaving semantics

Threads require a small-step semantics.
Each step $\rightarrow$ is atomic. Steps of threads are interleaved.

\[
\begin{array}{c}
t_1 \rightarrow \rightarrow \rightarrow \\
t_2 \rightarrow \rightarrow \rightarrow \\
\end{array}
\]

Thread pool $\text{pool} = \text{tid} \Rightarrow (\text{com} \times \text{locals}) \text{ option}$

State $\text{mstate} = \text{pool} \times \text{heap}$ ! Heap is shared.
Interleaving semantics

Threads require a **small-step semantics**. Each step $\rightarrow$ is atomic. Steps of threads are interleaved.

\[
\begin{align*}
& t_1 \rightarrow \rightarrow \rightarrow \\
& t_2 \rightarrow \rightarrow \rightarrow
\end{align*}
\]

**Thread pool** $pool = tid \Rightarrow (com \times locals) \text{ option}$

**State** $mstate = pool \times heap$  ![Heap is shared.](image)

**Semantics** $\leadsto$ interleaves $\rightarrow$ of individual threads

\[
\begin{align*}
p \ t &= Some \ (c, \ l) \quad (c, \ (h, \ l)) \rightarrow (c', \ (h', \ l')) \\
(p, \ h) \leadsto (p(t := Some \ (c', \ l')), \ h')
\end{align*}
\]
Synchronisation

Locks

At most one thread can hold a lock at a time.

\[\texttt{synchronized (l) \{ \hspace{1em} // acquire lock on object } l \]

\[\hspace{1em} o.x = o.x + 1; \]
\[o.y[1] = 0; \]
\[\hspace{1em} // release lock on object } l \]

Formalisation:

- Step \(a \rightarrow \text{now}\) carries a synchronisation action, e.g., Lock \(l\), Unlock \(l\)
- Global state also stores the locks and their state.
- \(;\) checks action against state and updates lock state accordingly.

Other forms of synchronisation in Java:
- not covered here
- fork–join
- interrupts
- wait-notify
- java.util.concurrent.*
Synchronisation

Locks

At most one thread can hold a lock at a time.

```java
synchronized (l) {
    // acquire lock on object l
    o.x = o.x + 1;
    o.y[1] = 0;
}
// release lock on object l
```

Formalisation:

- Step $\xrightarrow{a}$ now carries a **synchronisation action** $a$
  - e.g., Lock $l$, Unlock $l$
- Global state also stores the locks and their state.
- $\sim$ checks action against state and updates lock state accordingly.
Synchronisation

Locks
At most one thread can hold a lock at a time.

```java
synchronized (l) {
    // acquire lock on object l
    o.x = o.x + 1;
    o.y[1] = 0;
}
// release lock on object l
```

Formalisation:

- Step $\xrightarrow{a}$ now carries a synchronisation action $a$
  e.g., $Lock\ l$, $Unlock\ l$
- Global state also stores the locks and their state.
- $\sim$ checks action against state and updates lock state accordingly.

Other forms of synchronisation in Java: not covered here
fork–join, interrupts, wait-notify, java.util.concurrent.*
Type safety: preservation

Lift well-typing and conformance pointwise from single threads to a pool:

\[ \mathcal{G} \vdash (p, h) \iff (\forall t. \mathcal{G} \vdash p_t \checkmark) \land \text{hconf } h \]

where \( \Gamma \vdash (c, l) \checkmark \iff \Gamma \vdash c \land \Gamma \vdash l \)
Type safety: preservation

Lift well-typing and conformance pointwise from single threads to a pool:

\[ \mathcal{G} \vdash (p, h) \iff (\forall t. \mathcal{G} \vdash p_t \checkmark) \land \text{hconf}\ h \]

where \( \Gamma \vdash (c, l) \checkmark \iff \Gamma \vdash c \land \Gamma \vdash l \)

**Theorem:**
If \( \rightarrow \) preserves well-typing and conformance, then \( \sim \) preserves \( \mathcal{G} \vdash s \checkmark \).
Type safety: progress

Progress:

*If everything is OK,*

*and the state is not final,*

*then you can take one more step.*

1. Progress of \(\sim\) requires more than progress of \(\rightarrow\).

\(a\) could issue only impossible actions \(a\)
Type safety: progress

Progress:

If everything is OK, and the state is not final, then you can take one more step.

Progress of $\sim$ requires more than progress of $\rightarrow$. $a \rightarrow$ could issue only impossible actions $a$.

Must show: If all possible steps of one thread carry an Unlock, then the thread holds one of the locks.
Type safety: progress

Progress:

*If everything is OK,*
*and the state is not final,*
*then you can take one more step.*

1. **Progress of** $\sim$ **requires more than progress of** $\rightarrow$.
   $\rightarrow$ could issue only impossible actions $a$.

   **Must show:** If all possible steps of one thread carry an *Unlock*, then the thread holds one of the locks.

2. When is a state *final*?
Type safety: progress

Progress:

*If everything is OK, and the state is not final, then you can take one more step.*

\[ \text{Progress of } \sim \text{ requires more than progress of } \rightarrow. \]
\[ \rightarrow^a \text{ could issue only impossible actions } a \]

**Must show:** If all possible steps of one thread carry an *Unlock*, then the thread holds one of the locks.

1. When is a state *final*?
   When all threads have reached *SKIP*?

What about deadlock?
Type safety: progress

Progress:

*If everything is OK,*
*and the state is not final,*
*then you can take one more step.*

1. Progress of $\sim$ requires more than progress of $\rightarrow$. $\rightarrow$ could issue only impossible actions $a$.

   **Must show:** If all possible steps of one thread carry an *Unlock*, then the thread holds one of the locks.

2. When is a state *final*?
   When all threads have reached *SKIP*?
   What about deadlock?
Deadlock

A thread is waiting for something that it will never get.
A thread is waiting for something that it will never get.

THREAD 1:
```
synchronized (l1) {
    synchronized (l2) {
        synchronized (l2) {
            synchronized (l1) {

        }
    }
}
```

THREAD 2:
```
synchronized (l2) {
    synchronized (l1) {

}
```

locks held

locks needed

thread 1
thread 2
Deadlock

A thread is waiting for something that it will never get.

THREAD 1:
```java
synchronized (l1) {
    synchronized (l2) {
        synchronized (l2) {}
    }
    synchronized (l1) {}
}
```

THREAD 2:
```java
synchronized (l2) {
    synchronized (l1) {}
}
```

locks held locks needed

thread 1    l1
thread 2
Deadlock

A thread is waiting for something that it will never get.

THREAD 1:
synchronized (l1) {
    synchronized (l2) {
        synchronized (l2) {}
    }
synchronized (l1) {}
}

THREAD 2:
synchronized (l2) {
synchronized (l1) {}
}

locks held  locks needed

thread 1      l1
thread 2      l2
Deadlock

A thread is waiting for something that it will never get.

THREAD 1:
synchronized (l1) {
  synchronized (l2) {
    synchronized (l2) {}  
  }
  synchronized (l1) {}  
}

THREAD 2:
synchronized (l2) {
  synchronized (l1) {}  
}

locks held  |  locks needed  
---|---|---|---
thread 1    | l1 | l2
thread 2    | l2 |
A thread is waiting for something that it will never get.

THREAD 1:
\[
\text{synchronized (l1) {}
\quad \text{synchronized (l2) {}}
\quad \text{synchronized (l2) {}}
\]

THREAD 2:
\[
\text{synchronized (l2) {}
\quad \text{synchronized (l1) {}}
\]

<table>
<thead>
<tr>
<th>locks held</th>
<th>locks needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread 1</td>
<td>l1</td>
</tr>
<tr>
<td>thread 2</td>
<td>l2</td>
</tr>
</tbody>
</table>
Deadlock

A thread is waiting for something that it will never get.

THREAD 1:
```
synchronized (l1) {
    synchronized (l2) {
        synchronized (l2) {}    
    }
    synchronized (l1) {}     
}
```

THREAD 2:
```
synchronized (l2) {
    synchronized (l1) {}     
}
```

locks held locks needed

<table>
<thead>
<tr>
<th>thread 1</th>
<th>l1</th>
<th>l2</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread 2</td>
<td>l2</td>
<td>l1</td>
</tr>
</tbody>
</table>

cyclic waiting
Deadlock

A thread is waiting for something that it will never get.

THREAD 1:
```
synchronized (l1) {
  synchronized (l2) {
    synchronized (l2) {}
  }
  synchronized (l1) {}
```

THREAD 2:
```
synchronized (l2) {
  synchronized (l1) {}
```

<table>
<thead>
<tr>
<th>locks held</th>
<th>locks needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread 1</td>
<td>l1</td>
</tr>
<tr>
<td>thread 2</td>
<td>l2</td>
</tr>
</tbody>
</table>

cyclic waiting

Final = all threads have reached SKIP or are in deadlock

A state \((p, h, \text{locks})\) is in deadlock iff for all threads \(t\) such that \(p t = \text{Some} (c, l)\), whenever \((c, (h, l)) \xrightarrow{a} \_\), then \(a = \text{Lock} \ l\) and \(\text{locks} \ l = \text{Some} \ t'\) with \(t' \neq t\).
Summary on Multithreading

- Separate layer \(\leadsto\) interleaves threads
- Synchronisation via action labels \(\xrightarrow{a}\)
- Progress up to deadlock
- Type safety proves that all synchronisation mechanisms are fully implemented.
Summary on Multithreading

- Separate layer $\leadsto$ interleaves threads
- Synchronisation via action labels $\xrightarrow{a}$
- Progress up to deadlock
- Type safety proves that all synchronisation mechanisms are fully implemented.

Modelling the other synchronisation mechanisms:
- They are implemented as methods of system classes. Need a formal system for native methods.
- Introduce new possibilities for deadlock.
Further Reading


or look at my PhD thesis.

Memory Model  Andreas Lochbihler: Making the Java memory model safe. ACM TOPLAS 35(4), Article 12 (65 pages), 2014.

Or come to my talk today at 2:15 pm in E1 5 HS 002
Further Reading


or look at my PhD thesis.

Memory Model      Andreas Lochbihler: Making the Java memory model safe. ACM TOPLAS 35(4), Article 12 (65 pages), 2014.

Or come to my talk today at 2:15 pm in E1 5 HS 002