The formal semantics of a Java-like Programming Language

Tobias Nipkow and Gerwin Klein

What is Jinja?

Jinja is not Java!

Core calculus $\subset$ Jinja $\subset$ Java

Overview source language

- Syntax
- Big step semantics
- Small step semantics
- Equivalence
- Type system
- Type safety: *Well-typed programs do not get stuck*

Prelude: Formalities

Governement health warning

This is formal.

This is very formal.

This is machine checked!
(In the theorem prover Isabelle/HOL)

Basic types

- Truth values: *bool* (*True*, *False*, $\wedge$, ...)
- Natural numbers: *nat*
- Integers: *int*
- Total functions: $\Rightarrow$
- Sets over type τ: τ *set*
- Type variables: *'a*, *'b*, ...

Notation

$t::\tau$

"term t has type τ"

Pairs

- First component: $fst\ (x, y) = x$
- Second component: $snd\ (x, y) = y$
- Tuples are pairs nested to the right:
  - (a, b, c) means $(a, (b, c))$
  - $\tau_1 \times \tau_2 \times \tau_3$ means $\tau_1 \times (\tau_2 \times \tau_3)$

Lists

- Type: τ *list*
- Empty list: $[]$
- Cons: $x \cdot xs$
- Append: $xs \mathbin{@} ys$
- Length: $|xs|$
- Nth element: $xs_{[n]}$
- *map* :: $('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b\ list$
- *set* :: $'a\ list \Rightarrow 'a\ set$
- *distinct* :: $'a\ list \Rightarrow bool$

Datatype option

datatype $'a\ option\ =\ None \mid Some\ 'a$

An abbreviation: $\lfloor a \rfloor\ \equiv\ Some\ a$

Partial functions or maps

$$'a \rightharpoonup 'b \quad \equiv \quad 'a \Rightarrow 'b\ option$$

Intuitively: if $f :: \tau_1 \rightharpoonup \tau_2$ and $a :: \tau_1$ then

- $f\ a = None$ means undefined
- $f\ a = \lfloor b \rfloor$ means result is b

$$dom\ m \equiv \{a \mid m\ a \neq None\}$$

Map notation

- Empty map: $empty \equiv \lambda x.\ None$
- Update: $m(a \mapsto b) \equiv \lambda x.\ if\ x{=}a\ then\ \lfloor b \rfloor\ else\ m\ x$
- Updates: $m(a_1 \mapsto b_1, a_2 \mapsto b_2) \equiv m(a_1 \mapsto b_1)(a_2 \mapsto b_2)$
- Finite maps: $[a_1 \mapsto b_1, \ldots] \equiv empty(a_1 \mapsto b_1, \ldots)$
- List update: $m([a_1,\ldots,a_i]\ [\mapsto]\ [b_1,\ldots,b_j]) \equiv$
  $m(a_1 \mapsto b_1,\ldots,a_{min\ i\ j} \mapsto b_{min\ i\ j})$
- $map\text{-}of :: ('a \times 'b)list \Rightarrow 'a \rightharpoonup 'b$
  $map\text{-}of\ [(a_1,b_1),\ldots,(a_n,b_n)] = [a_n \mapsto b_n,\ldots,a_1 \mapsto b_1]$
- Overwrite:
  $m_1 +\!\!+ m_2 \equiv \lambda x.\ if\ x \in dom\ m_2\ then\ m_2\ x\ else\ m_1\ x$

Inference rule notations

$$[\![A_1; \ldots; A_n]\!] \Longrightarrow A$$

$$A_1 \Longrightarrow \ldots\ A_n \Longrightarrow A$$

$$\frac{A_1 \quad \ldots \quad A_n}{A}$$

Different syntax, same meaning

Jinja

Abstract Syntax

Names

Three types of names:

- Class names: *cname*
- Variable and field names: *vname*
- Method names: *mname*

Variable conventions

- *C*, *D*: class name
- *e*: expressions
- *F*: field name
- *M*: method name
- *P*: program
- *T*: type
- *v*: value
- *V*: variable name

Values

A value (type *val*) is one of the following:

- a boolean *Bool b*, where *b* :: *bool*
- an integer *Intg i*, where *i* :: *int*
- a reference *Addr a*, where *a* :: *addr*
  (*addr*: some arbitrary type of addresses)
- the null reference *Null*
- the dummy value *Unit*

Jinja is a hybrid object oriented language:
it has primitive values and classes

The formal model

$$\begin{array}{lll} \textbf{datatype}\ \mathit{val} = & \mathit{Unit} \\ & \mid\ \mathit{Null} \\ & \mid\ \mathit{Bool}\ \mathit{bool} \\ & \mid\ \mathit{Intg}\ \mathit{int} \\ & \mid\ \mathit{Addr}\ \mathit{addr} \end{array}$$

Jinja Types

A Jinja type (type *ty*) is one of the following:

- *Boolean*
- *Integer*
- *Class C*
- . . .

More later.

Expressions and statements

In Jinja, everything is an *expression*.

Statements are expressions evaluating to *Unit*.

Expressions (1)

Type: *expr*

- `new` C
- `Cast` C e
- `Val` v
- e_1 «*bop*» e_2
- `Var` V
- V `:=` e
- $e.F\{D\}$
- $e.F\{D\}$ `:=` e'
- $e.M(es)$

Interlude: annotation {D}

$e.F\{D\}$ and $e.F\{D\} := e'$:

- D is class where F is declared
- D is needed during runtime:
  field access is static, not dynamic!
- D is added by the type checker (see later)

Expressions (2)

- $\{V{:}T\,;\,e\}$
- $e_1\,;\,e_2$
- $\texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2$
- $\texttt{while}\ (e)\ e'$
- Exceptions: later

Exercise: which Java expressions/statements are missing?

Abstract and concrete syntax

Formal definition:

datatype $expr = new\ cname \mid Cast\ cname\ expr \mid \ldots$

This is *abstract* syntax.

For readability: additional *concrete* syntax, e.g.

$V := e$ is just syntactic sugar for $LAss\ V\ e$

Jinja is a mix of concrete and abstract syntax

Example

Translate `i = i + 1` into Jinja syntax.

Abbreviations

$$
\begin{array}{lcllcl}
\mathit{true} & \equiv & \texttt{Val}(\mathit{Bool\ True}) & \mathit{false} & \equiv & \texttt{Val}(\mathit{Bool\ False}) \\
\mathit{addr\ a} & \equiv & \texttt{Val}(\mathit{Addr\ a}) & \mathit{null} & \equiv & \texttt{Val}\ \mathit{Null} \\
\mathit{unit} & \equiv & \texttt{Val}\ \mathit{Unit} & & &
\end{array}
$$

Why no return statement?

Because everything has a value.

How to eliminate `return`:

- `return e` $\leadsto$ `?`
- `while (a) {if (b) return c else s};`
  `return d`
  $\leadsto$ `?`

Eliminating return in general

- Introduce new variables `res`(ult) and `cont`(inue).
- Eliminate `return`:
  `return e` $\leadsto$ `res := e; cont := false`
- Restructure control flow:
  `e1; e2` $\leadsto$ `e1; if (cont) e2`
  `while (b) e` $\leadsto$ `while (b & cont) e`
- Transform procedure body:
  `e` $\leadsto$ `cont := true; e; res`

The need for Var

Without `Var`, $V := e$ is ambiguous: is e

- a program variable or
- a meta-language variable ranging over expressions?

Binary operators

For simplicity: only = and +
Semantics: $binop :: bop \times val \times val \Rightarrow val\ option$

$$
\begin{array}{lcl}
binop\ (=, v_1, v_2) & = & \lfloor Bool\ (v_1 = v_2) \rfloor \\
binop\ (+, Intg\ i_1, Intg\ i_2) & = & \lfloor Intg\ (i_1 + i_2) \rfloor \\
binop\ (bop, v_1, v_2) & = & None
\end{array}
$$

Can only add integers!

Design principle:

In ill-typed situations, semantics does nothing

Defensive semantics

Jinja types

- They exist
- *ty* is the type of all Jinja types
- Semantics does not need them
- See later

Abstract Syntax: Programs

Concrete syntax

Many possibilities:

- Subset of Java
- Imaginary syntax, e.g.

```
class B extends A
 {field F : TF; field G : TG;
  method M1 : (p1:T1,...,pn:Tn) -> T = {...}
  ...
 }
class C extends B {...}
```

- ...

We don’t care!

Abstract syntax

- A program is a list of class declarations:
  $\mathit{J\text{-}prog} = \mathit{J\text{-}cdecl\ list}$
- A class declaration consists of the name and the class:
  $\mathit{J\text{-}cdecl} = \mathit{cname} \times \mathit{J\text{-}class}$
- A class consists of a superclass name, a list of field declarations and a list of method declarations:
  $\mathit{J\text{-}class} = \mathit{cname} \times \mathit{fdecl\ list} \times \mathit{J\text{-}mdecl\ list}$
- A field declaration is a field name and a type:
  $\mathit{fdecl} = \mathit{vname} \times \mathit{ty}$
- A method declaration is a method name, the argument types, the result type, and the method body:
  $\mathit{J\text{-}mdecl} = \mathit{mname} \times \mathit{ty\ list} \times \mathit{ty} \times \mathit{J\text{-}mb}$
- A method body is a parameter list and an expression:
  $\mathit{J\text{-}mb} = \mathit{vname\ list} \times \mathit{expr}$

Example

```
class B extends A
 {field F : TF; field G : TG;
  method M1 : (p1:T1,p2:T2) -> T =
    {i:Ti; i := i}
 }
```

$\leadsto$?

Note

Parameter names and types have been separated.

- Advantage: can re-use same declaration structure for machine programs (where method parameters do not have names) by redefining *J-mb*
- Disadvantage: need to make sure that there the same number of parameter names as types.

Exercise

Which Java features are missing?

Accessing declaration information

- $\mathit{class}\ P\ C$ is the class associated with C in P.
- $\mathit{is\text{-}class}\ P\ C$ means class C is defined in P.
- $P \vdash C \preceq^* D$ means C is a **subclass** of D. The relation is transitive and reflexive.
- $P \vdash C$ **sees** $M\colon Ts \to T = mb$ **in** D means that in P from class C a method M with argument types Ts (a type list), result type T, and body mb is visible in class D (taking overriding into account).
- $P \vdash C$ **sees** $F\!:\!T$ **in** D means that in P from class C a field F of type T is visible in class D.

Example

```
class B extends A {field F:TB
                   method M:TBs->T1 = mB}
class C extends B {field F:TC
                   method M:TCs->T2 = mC}
```

Example

$P \vdash C$ **sees** $F{:}TC$ **in** C	YES
$P \vdash C$ **sees** $F{:}TB$ **in** B	NO
$P \vdash C$ **sees** $M{:}\ TCs \to T2 = mC$ **in** C	YES
$P \vdash C$ **sees** $M{:}\ TBs \to T1 = mB$ **in** B	NO

No overloading of method names as in Java!

Formal definitions

- $\mathit{class}\ P\ C \equiv \mathit{map\text{-}of}\ P\ C$
- $\mathit{is\text{-}class}\ P\ C \equiv \mathit{class}\ P\ C \neq \mathit{None}$
- $P \vdash C \preceq^* D$ is the reflexive transitive closure of the direct subclass relation $P \vdash C \prec^1 D$ defined by

$$\frac{\mathit{class}\ P\ C = \lfloor (D, \_, \_) \rfloor \qquad C \neq \mathit{Object}}{P \vdash C \prec^1 D}$$

Collecting fields

Auxiliary relation $P \vdash C$ **has-fields** $FDTs$
with $FDTs :: ((vname \times cname) \times ty)\ list$:

- $((F, D), T)$ in $FDTs$ represents a field declaration $F{:}T$ in a superclass D of C
- $FDTs$ starts with C and climbs up the class hierarchy

$$\frac{class\ P\ C = \lfloor (D, fs, ms) \rfloor \qquad C \neq Object \qquad P \vdash D\ \textbf{has-fields}\ FDTs}{P \vdash C\ \textbf{has-fields}\ map\ (\lambda(F, T).\ ((F, C), T))\ fs\ @\ FDTs}$$

$$P \vdash Object\ \textbf{has-fields}\ []$$

Example (with `A` = `Object`):

$P \vdash C$ **has-fields** $[((F, C), TC), ((F, B), TB)]$

sees fields

$P \vdash C$ **sees** $F : T$ **in** $D \equiv$

$\exists FDTs.$

$P \vdash C$ **has-fields** $FDTs \wedge$

$map\text{-}of\ (map\ (\lambda((F, D), T).\ (F, D, T))\ FDTs)\ F = \lfloor (D, T) \rfloor$

Exercise

Define **sees** for methods.

Big Step Semantics

State

$$
\begin{array}{lcl}
\mathit{state} & = & \mathit{heap} \times \mathit{locals} \\
\mathit{locals} & = & \mathit{vname} \rightharpoonup \mathit{val} \\
\mathit{heap} & = & \mathit{addr} \rightharpoonup \mathit{obj} \\
\mathit{obj} & = & \mathit{cname} \times \mathit{fields} \\
\mathit{fields} & = & \mathit{vname} \times \mathit{cname} \rightharpoonup \mathit{val}
\end{array}
$$

Example state:

$$
\begin{array}{l}
([7 \mapsto (B, [(F,B) \mapsto \mathit{Addr}\ 1]), \\
\quad 5 \mapsto (C, [(F,C) \mapsto \mathit{Bool}\ \mathit{True}, (F,B) \mapsto \mathit{Intg}\ 5])], \\
\quad [V \mapsto \mathit{Null}])
\end{array}
$$

Is this state “welformed”?

Question

What is the key difference between our abstract state and some concrete representation in memory?

Variable names

s :: *state*

l :: *locals*

h :: *heap*

Expression evaluation

Transition format:

$$P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle$$

where e' is fully evaluated: a value (or an exception)

Evaluation rules

Val and Var

$$P \vdash \langle \mathtt{Val}\ v, s \rangle \Rightarrow \langle \mathtt{Val}\ v, s \rangle$$

$$\frac{l\ V = \lfloor v \rfloor}{P \vdash \langle \mathtt{Var}\ V, (h, l) \rangle \Rightarrow \langle \mathtt{Val}\ v, (h, l) \rangle}$$

Note

Semantics is defensive

Example: $\langle \mathit{Var}\ V, (h, l) \rangle$ can only evaluate if $V \in \mathit{dom}\ l$

Field access

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathit{addr}\ a, (h, l) \rangle \quad h\ a = \lfloor (C, fs) \rfloor \quad fs\ (F, D) = \lfloor v \rfloor}{P \vdash \langle e.F\{D\}, s_0 \rangle \Rightarrow \langle \mathtt{Val}\ v, (h, l) \rangle}$$

Binary operations

$$\frac{P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \mathtt{Val}\ v_1, s_1 \rangle \quad P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \mathtt{Val}\ v_2, s_2 \rangle \quad \mathit{binop}\ (\mathit{bop}, v_1, v_2) = \lfloor v \rfloor}{P \vdash \langle e_1\ \text{«}\mathit{bop}\text{»}\ e_2, s_0 \rangle \Rightarrow \langle \mathtt{Val}\ v, s_2 \rangle}$$

new

$$\mathit{new\text{-}Addr}\ h \equiv$$

$$\mathit{if}\ \exists\, a.\ h\ a = \mathit{None}\ \mathit{then}\ \lfloor \mathit{SOME}\ a.\ h\ a = \mathit{None} \rfloor\ \mathit{else}\ \mathit{None}$$

$$\mathit{init\text{-}fields} :: \big((\mathit{vname} \times \mathit{cname}) \times \mathit{ty}\big)\ \mathit{list} \Rightarrow \mathit{fields}$$

$$\mathit{init\text{-}fields} \equiv \mathit{map\text{-}of} \circ \mathit{map}\ \big(\lambda(FC, T).\ (FC, \mathit{default\text{-}val}\ T)\big)$$

$$\frac{\begin{array}{c}\mathit{new\text{-}Addr}\ h = \lfloor a \rfloor \\ P \vdash C\ \textbf{has-fields}\ \mathit{FDTs} \\ h' = h(a \mapsto (C, \mathit{init\text{-}fields}\ \mathit{FDTs}))\end{array}}{P \vdash \langle \mathsf{new}\ C, (h, l)\rangle \Rightarrow \langle \mathit{addr}\ a, (h', l)\rangle}$$

Cast

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathit{addr}\ a, (h, l) \rangle \quad h\ a = \lfloor (D, \mathit{fs}) \rfloor \quad P \vdash D \preceq^* C}{P \vdash \langle \texttt{Cast}\ C\ e, s_0 \rangle \Rightarrow \langle \mathit{addr}\ a, (h, l) \rangle}$$

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathit{null}, s_1 \rangle}{P \vdash \langle \texttt{Cast}\ C\ e, s_0 \rangle \Rightarrow \langle \mathit{null}, s_1 \rangle}$$

Variable assignment

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathtt{Val}\ v, (h, l) \rangle}{P \vdash \langle V := e, s_0 \rangle \Rightarrow \langle \mathtt{Val}\ v, (h, l(V \mapsto v)) \rangle}$$

Field assignment

$$\frac{\begin{array}{c} P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \mathit{addr}\ a, s_1 \rangle \\ P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \mathtt{Val}\ v, (h_2, l_2) \rangle \\ h_2\ a = \lfloor (C, fs) \rfloor \\ fs' = fs((F, D) \mapsto v) \\ h_2' = h_2(a \mapsto (C, fs')) \end{array}}{P \vdash \langle e_1.F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \mathtt{Val}\ v, (h_2', l_2) \rangle}$$

Method call

$$\frac{\begin{array}{c} P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathit{addr}\ a, s_1 \rangle \\ P \vdash \langle \mathit{ps}, s_1 \rangle\ [\Rightarrow]\ \langle \mathit{map}\ \mathtt{Val}\ \mathit{vs}, (h_2, l_2) \rangle \\ h_2\ a = \lfloor (C, \mathit{fs}) \rfloor \\ P \vdash C\ \textbf{sees}\ M\colon Ts \to T = (\mathit{pns}, \mathit{body})\ \textbf{in}\ D \\ |\mathit{vs}| = |\mathit{pns}| \\ l_2' = \mathit{empty}(\mathit{this} \mapsto \mathit{Addr}\ a, \mathit{pns}\ [\mapsto]\ \mathit{vs}) \\ P \vdash \langle \mathit{body}, (h_2, l_2') \rangle \Rightarrow \langle e', (h_3, l_3) \rangle \end{array}}{P \vdash \langle e.M(\mathit{ps}), s_0 \rangle \Rightarrow \langle e', (h_3, l_2) \rangle}$$

$P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle$

Evaluate *es* from left to right:

$$P \vdash \langle [],s\rangle\ [\Rightarrow]\ \langle [],s\rangle$$

$$\frac{P \vdash \langle e,s_0\rangle \Rightarrow \langle \texttt{Val}\ v,s_1\rangle \qquad P \vdash \langle es,s_1\rangle\ [\Rightarrow]\ \langle es',s_2\rangle}{P \vdash \langle e\cdot es,s_0\rangle\ [\Rightarrow]\ \langle \texttt{Val}\ v\cdot es',s_2\rangle}$$

Why `Val` v and not just e'?

Because of exceptions (later).

Example

```
class C extends Object
{ field F:Integer
  method m(i:Integer):Integer =
    this.F{C} + Var i
}
```

State:

$h = empty(a \mapsto (C, fs))$

$fs = empty((F, C) \mapsto Intg\ 2)$

$l = empty(V \mapsto Addr\ a)$

Evaluation:

$P \vdash \langle \texttt{Var}\ V.m([\texttt{Val}\ (Intg\ 1)]),(h, l)\rangle \Rightarrow \langle \texttt{Val}\ (Intg\ 3),(h, l)\rangle$

Fields versus methods

- Field lookup $e.F\{D\}$ is *static*:
  annotation D is based on the static type of e.
- Method lookup $e.M(es)$ is *dynamic*:
  it is based on the dynamic type of the object e.

Local variable

$$\frac{P \vdash \langle e_0,(h_0, l_0(V := None))\rangle \Rightarrow \langle e_1,(h_1, l_1)\rangle}{P \vdash \langle \{V{:}T;\ e_0\},(h_0, l_0)\rangle \Rightarrow \langle e_1,(h_1, l_1(V := l_0\ V))\rangle}$$

Sequential composition

$$\frac{P \vdash \langle e_0, s_0 \rangle \Rightarrow \langle \texttt{Val}\ v, s_1 \rangle \qquad P \vdash \langle e_1, s_1 \rangle \Rightarrow \langle e_2, s_2 \rangle}{P \vdash \langle e_0\texttt{;}\ e_1, s_0 \rangle \Rightarrow \langle e_2, s_2 \rangle}$$

Conditional

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathit{true}, s_1 \rangle \qquad P \vdash \langle e_1, s_1 \rangle \Rightarrow \langle e', s_2 \rangle}{P \vdash \langle \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2, s_0 \rangle \Rightarrow \langle e', s_2 \rangle}$$

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathit{false}, s_1 \rangle \qquad P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle e', s_2 \rangle}{P \vdash \langle \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2, s_0 \rangle \Rightarrow \langle e', s_2 \rangle}$$

While loop

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathit{false}, s_1 \rangle}{P \vdash \langle \texttt{while}\ (e)\ c, s_0 \rangle \Rightarrow \langle \mathit{unit}, s_1 \rangle}$$

$$\frac{\begin{array}{c} P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathit{true}, s_1 \rangle \\ P \vdash \langle c, s_1 \rangle \Rightarrow \langle \texttt{Val}\ v_1, s_2 \rangle \\ P \vdash \langle \texttt{while}\ (e)\ c, s_2 \rangle \Rightarrow \langle e_3, s_3 \rangle \end{array}}{P \vdash \langle \texttt{while}\ (e)\ c, s_0 \rangle \Rightarrow \langle e_3, s_3 \rangle}$$

Digression: simultaneous inductive definitions

$P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$ and $P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle$ are defined simultaneously:

> $P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$ uses $P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle$ and
> $P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle$ uses $P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$.

Consequence: *simultaneous rule induction*:

> $P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle \Longrightarrow R_1\ P\ e\ s\ e'\ s'$
> $P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow R_2\ P\ es\ s\ es'\ s'$

must be proved simultaneously.
Slight complication: property R_2 is auxiliary and needs to be determined. Usually R_2 is R_1 lifted to lists.

Example: Final expressions

$$\mathit{final}\ e \equiv \exists v.\ e = \texttt{Val}\ v$$
$$\mathit{finals}\ es \equiv \exists vs.\ es = \mathit{map}\ \texttt{Val}\ vs$$

Lemma If $P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$ then $\mathit{final}\ e'$.
If $P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle$ then $\mathit{finals}\ es'$.

Proof by simultaneous rule induction.

Small Step Semantics

Transition format

$$P \vdash \langle e,s \rangle \rightarrow \langle e',s' \rangle$$

$$P \vdash \langle es,s \rangle \; [\rightarrow] \; \langle es',s' \rangle$$

Reduction strategy:

Reduce subsexpressions as long as necessary, then reduce whole expression.

Two kinds of rules

- Subexpression reduction (for construct c):

  $P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow$
  $P \vdash \langle c\ \ldots\ e\ \ldots, s\rangle \rightarrow \langle c\ \ldots\ e'\ \ldots, s'\rangle$

  Example:

  $P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow$
  $P \vdash \langle \texttt{Cast}\ C\ e,s\rangle \rightarrow \langle \texttt{Cast}\ C\ e',s'\rangle$

- Expression reduction

  Example: $P \vdash \langle \texttt{Cast}\ C\ null,s\rangle \rightarrow \langle null,s\rangle$

Notation: state components

$$hp \equiv fst$$
$$lcl \equiv snd$$

Val and Var

$$\frac{\mathit{lcl}\ s\ V = \lfloor v \rfloor}{P \vdash \langle \mathtt{Var}\ V, s \rangle \rightarrow \langle \mathtt{Val}\ v, s \rangle}$$

Field access

$$\frac{P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle}{P \vdash \langle e.F\{D\},s\rangle \rightarrow \langle e'.F\{D\},s'\rangle}$$

$$\frac{hp\ s\ a = \lfloor (C, fs)\rfloor \qquad fs\ (F, D) = \lfloor v\rfloor}{P \vdash \langle addr\ a.F\{D\},s\rangle \rightarrow \langle \texttt{Val}\ v,s\rangle}$$

Binary operations

$$\frac{P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle}{P \vdash \langle e \;\text{«}bop\text{»}\; e_2,s\rangle \rightarrow \langle e' \;\text{«}bop\text{»}\; e_2,s'\rangle}$$

$$\frac{P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle}{P \vdash \langle \mathtt{Val}\; v_1 \;\text{«}bop\text{»}\; e,s\rangle \rightarrow \langle \mathtt{Val}\; v_1 \;\text{«}bop\text{»}\; e',s'\rangle}$$

$$\frac{bop\ (bop, v_1, v_2) = \lfloor v \rfloor}{P \vdash \langle \mathtt{Val}\; v_1 \;\text{«}bop\text{»}\; \mathtt{Val}\; v_2,s\rangle \rightarrow \langle \mathtt{Val}\; v,s\rangle}$$

new

$$\frac{\begin{array}{c}\mathit{new\text{-}Addr}\ h = \lfloor a \rfloor \\ P \vdash C\ \textbf{has-fields}\ \mathit{FDTs} \\ h' = h(a \mapsto (C, \mathit{init\text{-}fields}\ \mathit{FDTs}))\end{array}}{P \vdash \langle \mathsf{new}\ C, (h, l)\rangle \rightarrow \langle \mathit{addr}\ a, (h', l)\rangle}$$

Cast

$$\frac{P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle}{P \vdash \langle \texttt{Cast}\ C\ e,s\rangle \rightarrow \langle \texttt{Cast}\ C\ e',s'\rangle}$$

$$\frac{hp\ s\ a = \lfloor (D, fs) \rfloor \qquad P \vdash D \preceq^* C}{P \vdash \langle \texttt{Cast}\ C\ (addr\ a),s\rangle \rightarrow \langle addr\ a,s\rangle}$$

$$P \vdash \langle \texttt{Cast}\ C\ null,s\rangle \rightarrow \langle null,s\rangle$$

Variable assignment

$$\frac{P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle}{P \vdash \langle V := e, s \rangle \rightarrow \langle V := e', s' \rangle}$$

$$P \vdash \langle V := \mathtt{Val}\ v, (h, l) \rangle \rightarrow \langle \mathtt{Val}\ v, (h, l(V \mapsto v)) \rangle$$

Field assignment

$$\frac{P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle}{P \vdash \langle e.F\{D\} := e_2,s\rangle \rightarrow \langle e'.F\{D\} := e_2,s'\rangle}$$

$$\frac{P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle}{P \vdash \langle \mathtt{Val}\ v.F\{D\} := e,s\rangle \rightarrow \langle \mathtt{Val}\ v.F\{D\} := e',s'\rangle}$$

$$\frac{h\ a = \lfloor (C, fs)\rfloor \qquad fs' = fs((F, D) \mapsto v) \qquad h' = h(a \mapsto (C, fs'))}{P \vdash \langle addr\ a.F\{D\} := \mathtt{Val}\ v,(h, l)\rangle \rightarrow \langle \mathtt{Val}\ v,(h', l)\rangle}$$

Method call

$$\frac{P \vdash \langle e,s\rangle \to \langle e',s'\rangle}{P \vdash \langle e.M(es),s\rangle \to \langle e'.M(es),s'\rangle}$$

$$\frac{P \vdash \langle es,s\rangle\ [\to]\ \langle es',s'\rangle}{P \vdash \langle \texttt{Val}\ v.M(es),s\rangle \to \langle \texttt{Val}\ v.M(es'),s'\rangle}$$

$$[\![hp\ s\ a = \lfloor (C, fs) \rfloor;\ P \vdash C\ \textbf{sees}\ M\colon Ts \to T = (pns, body)\ \textbf{in}\ D;$$
$$|vs| = |pns|;\ |Ts| = |pns|]\!]$$
$$\Longrightarrow P \vdash \langle addr\ a.M(map\ \texttt{Val}\ vs), s\rangle \to$$
$$\langle blocks\ (this{\cdot}pns, Class\ D{\cdot}Ts, Addr\ a{\cdot}vs, body), s\rangle$$

$$blocks([V,\ldots],[T,\ldots],[v,\ldots],e) = \{V{:}T;\ V := Val\ v;\ \ldots\ e\}$$

Evaluate *es* from left to right:

$$\frac{P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle}{P \vdash \langle e \cdot es, s \rangle \; [\rightarrow] \; \langle e' \cdot es, s' \rangle}$$

$$\frac{P \vdash \langle es, s \rangle \; [\rightarrow] \; \langle es', s' \rangle}{P \vdash \langle \mathtt{Val}\; v \cdot es, s \rangle \; [\rightarrow] \; \langle \mathtt{Val}\; v \cdot es', s' \rangle}$$

Local variable

Trick: store value of local variable as $\{V{:}T;\ V := \mathtt{Val}\ v;\ e'\}$

$assigned\ V\ e \equiv \exists v\ e'.\ e = V := \mathtt{Val}\ v;\ e'$

$$[\![P \vdash \langle e,(h, l(V := None))\rangle \to \langle e',(h', l')\rangle; l'\ V = None;$$
$$\neg\ assigned\ V\ e]\!]$$
$$\Longrightarrow P \vdash \langle \{V{:}T; e\},(h, l)\rangle \to \langle \{V{:}T; e'\},(h', l'(V := l\ V))\rangle$$

$$[\![P \vdash \langle e,(h, l(V := None))\rangle \to \langle e',(h', l')\rangle; l'\ V = \lfloor v \rfloor;$$
$$\neg\ assigned\ V\ e]\!]$$
$$\Longrightarrow P \vdash \langle \{V{:}T; e\},(h, l)\rangle \to$$
$$\langle \{V{:}T; V := \mathtt{Val}\ v;\ e'\},(h', l'(V := l\ V))\rangle$$

Local variable (2)

$$[\![P \vdash \langle e,(h,l(V \mapsto v))\rangle \rightarrow \langle e',(h',l')\rangle; l'\,V = \lfloor v' \rfloor]\!]$$
$$\Longrightarrow P \vdash \langle \{V{:}T;\ V\ \texttt{:=}\ \texttt{Val}\ v\texttt{;}\ e\},(h,l)\rangle \rightarrow$$
$$\langle \{V{:}T;\ V\ \texttt{:=}\ \texttt{Val}\ v'\texttt{;}\ e'\},(h',l'(V := l\,V))\rangle$$

$$P \vdash \langle \{V{:}T;\ V\ \texttt{:=}\ \texttt{Val}\ v\texttt{;}\ \texttt{Val}\ u\},s\rangle \rightarrow \langle \texttt{Val}\ u,s\rangle$$

$$P \vdash \langle \{V{:}T;\ \texttt{Val}\ u\},s\rangle \rightarrow \langle \texttt{Val}\ u,s\rangle$$

Sequential composition

$$\frac{P \vdash \langle e, s\rangle \rightarrow \langle e', s'\rangle}{P \vdash \langle e\texttt{;}\ e_2, s\rangle \rightarrow \langle e'\texttt{;}\ e_2, s'\rangle}$$

$$P \vdash \langle \texttt{Val}\ v\texttt{;}\ e_2, s\rangle \rightarrow \langle e_2, s\rangle$$

Conditional

$$\frac{P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle}{P \vdash \langle \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2,s\rangle \rightarrow \langle \texttt{if}\ (e')\ e_1\ \texttt{else}\ e_2,s'\rangle}$$

$$P \vdash \langle \texttt{if}\ (\mathit{true})\ e_1\ \texttt{else}\ e_2,s\rangle \rightarrow \langle e_1,s\rangle$$
$$P \vdash \langle \texttt{if}\ (\mathit{false})\ e_1\ \texttt{else}\ e_2,s\rangle \rightarrow \langle e_2,s\rangle$$

While loop

$$P \vdash \langle \texttt{while}\ (b)\ c, s \rangle \longrightarrow \langle \texttt{if}\ (b)\ (c\texttt{;}\ \texttt{while}\ (b)\ c)\ \texttt{else}\ \mathit{unit}, s \rangle$$

Example

$$P \vdash \langle \{ V{:}\mathit{Boolean};\ U := \mathit{true};\ V := \mathtt{Var}\ U;\ \mathtt{Var}\ V \},$$
$$(h, \mathit{empty})\rangle \longrightarrow^{*}$$
$$\langle \mathit{true}, (h, \mathit{empty}(U \mapsto \mathit{Bool\ True}))\rangle$$

Example

```
class C extends Object
{ field F:Integer
  method m(i:Integer):Integer =
    this.F{C} + Var i
}
```

State:

$h = empty\,(a \mapsto (C, fs))$

$fs = empty\,((F, C) \mapsto Intg\ 2)$

$l = empty\,(V \mapsto Addr\ a)$

Reduction:

$P \vdash \langle \texttt{Var}\ V.m([\texttt{Val}\ (Intg\ 1)]),(h, l)\rangle \rightarrow^* \langle \texttt{Val}\ (Intg\ 3),(h, l)\rangle$

Exceptions

Syntax

- `throw` e

  Jinja: e can be of any class

  Java: e must be of a subclass of `Throwable`

- `try` e_1 `catch` $(C\ V)$ e_2

Example:

`try throw (new` C`) catch (`$C\ V$`) Var` V

behaves like `new` C

System exceptions

System exception classes:

$$\mathit{NullPointer}, \mathit{ClassCast}, \mathit{OutOfMemory} :: \mathit{cname}$$

Simplification: System exceptions are allocated *statically* not created *dynamically*:

$$\mathit{addr\text{-}of\text{-}sys\text{-}xcpt} :: \mathit{cname} \Rightarrow \mathit{addr}$$

Abbreviations:

$$\begin{array}{lcl} \texttt{Throw}\ a & \equiv & \texttt{throw}(\mathit{addr}\ a) \\ \texttt{THROW}\ C & \equiv & \texttt{Throw}(\mathit{addr\text{-}of\text{-}sys\text{-}xcpt}\ C) \end{array}$$

Extension of big step semantics

Variable access

$$\frac{l\,V = \lfloor v \rfloor}{P \vdash \langle \mathtt{Var}\ V, (h, l) \rangle \Rightarrow \langle \mathtt{Val}\ v, (h, l) \rangle}$$

Uninitialized variables cannot be evaluated

Field access

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle addr\ a, (h, l) \rangle \quad h\ a = \lfloor (C, fs) \rfloor \quad fs\ (F, D) = \lfloor v \rfloor}{P \vdash \langle e.F\{D\}, s_0 \rangle \Rightarrow \langle \mathtt{Val}\ v, (h, l) \rangle}$$

Exception creation:

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle null, s_1 \rangle}{P \vdash \langle e.F\{D\}, s_0 \rangle \Rightarrow \langle \mathtt{THROW}\ NullPointer, s_1 \rangle}$$

Exception propagation:

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathtt{throw}\ e', s_1 \rangle}{P \vdash \langle e.F\{D\}, s_0 \rangle \Rightarrow \langle \mathtt{throw}\ e', s_1 \rangle}$$

Binary operations

$$\frac{P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \texttt{Val}\ v_1, s_1 \rangle \quad P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \texttt{Val}\ v_2, s_2 \rangle \quad \mathit{binop}\ (\mathit{bop}, v_1, v_2) = \lfloor v \rfloor}{P \vdash \langle e_1\ \text{«}\mathit{bop}\text{»}\ e_2, s_0 \rangle \Rightarrow \langle \texttt{Val}\ v, s_2 \rangle}$$

Exception propagation:

$$\frac{P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \texttt{throw}\ e, s_1 \rangle}{P \vdash \langle e_1\ \text{«}\mathit{bop}\text{»}\ e_2, s_0 \rangle \Rightarrow \langle \texttt{throw}\ e, s_1 \rangle}$$

$$\frac{P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \texttt{Val}\ v_1, s_1 \rangle \qquad P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \texttt{throw}\ e, s_2 \rangle}{P \vdash \langle e_1\ \text{«}\mathit{bop}\text{»}\ e_2, s_0 \rangle \Rightarrow \langle \texttt{throw}\ e, s_2 \rangle}$$

new

$$\frac{\begin{array}{c}\mathit{new\text{-}Addr}\ h = \lfloor a \rfloor \\ P \vdash C\ \textbf{has-fields}\ \mathit{FDTs} \\ h' = h(a \mapsto (C, \mathit{init\text{-}fields}\ \mathit{FDTs}))\end{array}}{P \vdash \langle \mathsf{new}\ C, (h, l)\rangle \Rightarrow \langle \mathit{addr}\ a, (h', l)\rangle}$$

Exception creation:

$$\frac{\mathit{new\text{-}Addr}\ h = \mathit{None}}{P \vdash \langle \mathsf{new}\ C, (h, l)\rangle \Rightarrow \langle \mathsf{THROW}\ \mathit{OutOfMemory}, (h, l)\rangle}$$

Cast

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathit{addr}\ a, (h, l) \rangle \quad h\ a = \lfloor (D, \mathit{fs}) \rfloor \quad P \vdash D \preceq^* C}{P \vdash \langle \mathtt{Cast}\ C\ e, s_0 \rangle \Rightarrow \langle \mathit{addr}\ a, (h, l) \rangle}$$

Exception creation:

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathit{addr}\ a, (h, l) \rangle \quad h\ a = \lfloor (D, \mathit{fs}) \rfloor \quad \neg\, P \vdash D \preceq^* C}{P \vdash \langle \mathtt{Cast}\ C\ e, s_0 \rangle \Rightarrow \langle \mathtt{THROW}\ \mathit{ClassCast}, (h, l) \rangle}$$

Exception propagation:

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathtt{throw}\ e', s_1 \rangle}{P \vdash \langle \mathtt{Cast}\ C\ e, s_0 \rangle \Rightarrow \langle \mathtt{throw}\ e', s_1 \rangle}$$

Variable assignment

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathtt{Val}\ v, (h, l) \rangle \qquad l' = l(V \mapsto v)}{P \vdash \langle V := e, s_0 \rangle \Rightarrow \langle \mathtt{Val}\ v, (h, l') \rangle}$$

Exception propagation:

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathtt{throw}\ e', s_1 \rangle}{P \vdash \langle V := e, s_0 \rangle \Rightarrow \langle \mathtt{throw}\ e', s_1 \rangle}$$

Field assignment (1)

$$\frac{P \vdash \langle e_1, s_0\rangle \Rightarrow \langle \mathit{addr}\ a, s_1\rangle \qquad P \vdash \langle e_2, s_1\rangle \Rightarrow \langle \mathtt{Val}\ v, (h_2, l_2)\rangle \qquad h_2\ a = \lfloor (C, fs)\rfloor \qquad fs' = fs((F, D) \mapsto v) \qquad h_2' = h_2(a \mapsto (C, fs'))}{P \vdash \langle e_1.F\{D\} := e_2, s_0\rangle \Rightarrow \langle \mathtt{Val}\ v, (h_2', l_2)\rangle}$$

Exception creation:

$$\frac{P \vdash \langle e_1, s_0\rangle \Rightarrow \langle \mathit{null}, s_1\rangle \qquad P \vdash \langle e_2, s_1\rangle \Rightarrow \langle \mathtt{Val}\ v, s_2\rangle}{P \vdash \langle e_1.F\{D\} := e_2, s_0\rangle \Rightarrow \langle \mathtt{THROW}\ \mathit{NullPointer}, s_2\rangle}$$

Principles:

- Evaluate all arguments before throwing an exception.
- Propagate exceptions immediately.

Field assignment (2)

Exception propagation:

$$\frac{P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \texttt{throw}\ e', s_1 \rangle}{P \vdash \langle e_1.F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \texttt{throw}\ e', s_1 \rangle}$$

$$\frac{P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \texttt{Val}\ v, s_1 \rangle \qquad P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \texttt{throw}\ e', s_2 \rangle}{P \vdash \langle e_1.F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \texttt{throw}\ e', s_2 \rangle}$$

Method call

Exception creation:

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathit{null}, s_1 \rangle \qquad P \vdash \langle \mathit{ps}, s_1 \rangle\ [\Rightarrow]\ \langle \mathit{map}\ \mathtt{Val}\ \mathit{vs}, s_2 \rangle}{P \vdash \langle e.M(\mathit{ps}), s_0 \rangle \Rightarrow \langle \mathtt{THROW}\ \mathit{NullPointer}, s_2 \rangle}$$

Exception propagation:

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathtt{throw}\ e', s_1 \rangle}{P \vdash \langle e.M(\mathit{ps}), s_0 \rangle \Rightarrow \langle \mathtt{throw}\ e', s_1 \rangle}$$

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathtt{Val}\ v, s_1 \rangle \qquad P \vdash \langle \mathit{es}, s_1 \rangle\ [\Rightarrow]\ \langle \mathit{map}\ \mathtt{Val}\ \mathit{vs} \mathbin{@} \mathtt{throw}\ \mathit{ex} \cdot \mathit{es}', s_2 \rangle}{P \vdash \langle e.M(\mathit{es}), s_0 \rangle \Rightarrow \langle \mathtt{throw}\ \mathit{ex}, s_2 \rangle}$$

Expression list

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathtt{Val}\ v, s_1 \rangle \qquad P \vdash \langle es, s_1 \rangle\ [\Rightarrow]\ \langle es', s_2 \rangle}{P \vdash \langle e \cdot es, s_0 \rangle\ [\Rightarrow]\ \langle \mathtt{Val}\ v \cdot es', s_2 \rangle}$$

Exception propagation:

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathtt{throw}\ e', s_1 \rangle}{P \vdash \langle e \cdot es, s_0 \rangle\ [\Rightarrow]\ \langle \mathtt{throw}\ e' \cdot es, s_1 \rangle}$$

Sequential composition

$$\frac{P \vdash \langle e_0, s_0 \rangle \Rightarrow \langle \texttt{Val}\ v, s_1 \rangle \qquad P \vdash \langle e_1, s_1 \rangle \Rightarrow \langle e_2, s_2 \rangle}{P \vdash \langle e_0\texttt{;}\ e_1, s_0 \rangle \Rightarrow \langle e_2, s_2 \rangle}$$

Exception propagation:

$$\frac{P \vdash \langle e_0, s_0 \rangle \Rightarrow \langle \texttt{throw}\ e, s_1 \rangle}{P \vdash \langle e_0\texttt{;}\ e_1, s_0 \rangle \Rightarrow \langle \texttt{throw}\ e, s_1 \rangle}$$

if

Exception propagation:

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \texttt{throw}\ e', s_1 \rangle}{P \vdash \langle \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2, s_0 \rangle \Rightarrow \langle \texttt{throw}\ e', s_1 \rangle}$$

while

Exception propagation:

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \texttt{throw}\ e', s_1 \rangle}{P \vdash \langle \texttt{while}\ (e)\ c, s_0 \rangle \Rightarrow \langle \texttt{throw}\ e', s_1 \rangle}$$

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathit{true}, s_1 \rangle \qquad P \vdash \langle c, s_1 \rangle \Rightarrow \langle \texttt{throw}\ e', s_2 \rangle}{P \vdash \langle \texttt{while}\ (e)\ c, s_0 \rangle \Rightarrow \langle \texttt{throw}\ e', s_2 \rangle}$$

throw

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathit{addr}\ a, s_1 \rangle}{P \vdash \langle \texttt{throw}\ e, s_0 \rangle \Rightarrow \langle \texttt{Throw}\ a, s_1 \rangle}$$

Exception creation:

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathit{null}, s_1 \rangle}{P \vdash \langle \texttt{throw}\ e, s_0 \rangle \Rightarrow \langle \texttt{THROW}\ \mathit{NullPointer}, s_1 \rangle}$$

Exception propagation:

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \texttt{throw}\ e', s_1 \rangle}{P \vdash \langle \texttt{throw}\ e, s_0 \rangle \Rightarrow \langle \texttt{throw}\ e', s_1 \rangle}$$

`try catch` *(1)*

Normal evaluation:

$$\frac{P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \texttt{Val}\ v_1, s_1 \rangle}{P \vdash \langle \texttt{try}\ e_1\ \texttt{catch}\ (C\ V)\ e_2, s_0 \rangle \Rightarrow \langle \texttt{Val}\ v_1, s_1 \rangle}$$

try catch *(2)*

Exception handling:

$$\frac{\begin{array}{c} P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \mathtt{Throw}\ a, (h_1, l_1) \rangle \\ h_1\ a = \lfloor (D, fs) \rfloor \qquad P \vdash D \preceq^* C \\ P \vdash \langle e_2, (h_1, l_1(V \mapsto Addr\ a)) \rangle \Rightarrow \langle e_2', (h_2, l_2) \rangle \end{array}}{P \vdash \langle \mathtt{try}\ e_1\ \mathtt{catch}\ (C\ V)\ e_2, s_0 \rangle \Rightarrow \langle e_2', (h_2, l_2(V := l_1\ V)) \rangle}$$

Exception propagation:

$$\frac{\begin{array}{c} P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \mathtt{Throw}\ a, (h_1, l_1) \rangle \\ h_1\ a = \lfloor (D, fs) \rfloor \qquad \neg\, P \vdash D \preceq^* C \end{array}}{P \vdash \langle \mathtt{try}\ e_1\ \mathtt{catch}\ (C\ V)\ e_2, s_0 \rangle \Rightarrow \langle \mathtt{Throw}\ a, (h_1, l_1) \rangle}$$

Final expressions redefined (exercise)

$final\ e \equiv (\exists v.\ e = \texttt{Val}\ v) \vee (\exists a.\ e = \texttt{Throw}\ a)$

$finals\ es \equiv ?$

Lemma If $P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$ then $final\ e'$.
If $P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle$ then $finals\ es'$.

Proof by simultaneous rule induction.

Extension of small step semantics

Variable access

$$\frac{\mathit{lcl}\ s\ V = \lfloor v \rfloor}{P \vdash \langle \mathtt{Var}\ V, s \rangle \to \langle \mathtt{Val}\ v, s \rangle}$$

$\mathit{lcl}\ s\ V = \mathit{None} \Longrightarrow P \vdash \langle \mathit{Var}\ V, s \rangle \to ?$

Uninitialized variables cannot be reduced

Ill-formed configuration — small step semantics is *stuck*

Field access

Exception creation:

$$P \vdash \langle null.F\{D\},s\rangle \rightarrow \langle \texttt{THROW}\ NullPointer,s\rangle$$

Exception propagation:

$$P \vdash \langle \texttt{throw}\ e.F\{D\},s\rangle \rightarrow \langle \texttt{throw}\ e,s\rangle$$

Exception creation:

$$\frac{\mathit{new\text{-}Addr}\ h = \mathit{None}}{P \vdash \langle \mathsf{new}\ C, (h, l) \rangle \rightarrow \langle \mathsf{THROW}\ \mathit{OutOfMemory}, (h, l) \rangle}$$

Cast

Exception creation:

$$\frac{hp\ s\ a = \lfloor (D, fs) \rfloor \qquad \neg\ P \vdash D \preceq^* C}{P \vdash \langle \texttt{Cast}\ C\ (addr\ a), s \rangle \rightarrow \langle \texttt{THROW}\ ClassCast, s \rangle}$$

Exception propagation:

$$P \vdash \langle \texttt{Cast}\ C\ (\texttt{throw}\ e), s \rangle \rightarrow \langle \texttt{throw}\ e, s \rangle$$

Variable assignment

Exception propagation:

$$P \vdash \langle V \texttt{ := throw } e, s \rangle \rightarrow \langle \texttt{throw } e, s \rangle$$

Field assignment

Exception creation:

$$P \vdash \langle null.F\{D\} := \texttt{Val}\ v, s\rangle \rightarrow \langle \texttt{THROW}\ NullPointer, s\rangle$$

Exception propagation:

$$P \vdash \langle \texttt{throw}\ e.F\{D\} := e_2, s\rangle \rightarrow \langle \texttt{throw}\ e, s\rangle$$

$$P \vdash \langle \texttt{Val}\ v.F\{D\} := \texttt{throw}\ e, s\rangle \rightarrow \langle \texttt{throw}\ e, s\rangle$$

Method call

Exception creation:

$$P \vdash \langle null.M(map\ \mathtt{Val}\ vs), s\rangle \rightarrow \langle \mathtt{THROW}\ NullPointer, s\rangle$$

Exception propagation:

$$P \vdash \langle \mathtt{throw}\ e.M(es), s\rangle \rightarrow \langle \mathtt{throw}\ e, s\rangle$$

$$P \vdash \langle \mathtt{Val}\ v.M(map\ \mathtt{Val}\ vs\ @\ \mathtt{throw}\ e{\cdot}es'), s\rangle \rightarrow \langle \mathtt{throw}\ e, s\rangle$$

Expression list

$$\frac{P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle}{P \vdash \langle e\cdot es,s\rangle\ [\rightarrow]\ \langle e'\cdot es,s'\rangle}$$

$$\frac{P \vdash \langle es,s\rangle\ [\rightarrow]\ \langle es',s'\rangle}{P \vdash \langle \mathtt{Val}\ v\cdot es,s\rangle\ [\rightarrow]\ \langle \mathtt{Val}\ v\cdot es',s'\rangle}$$

Sequential composition

$$\frac{P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle}{P \vdash \langle e;\ e_2,s\rangle \rightarrow \langle e';\ e_2,s'\rangle}$$

$$P \vdash \langle \mathtt{Val}\ v;\ e_2,s\rangle \rightarrow \langle e_2,s\rangle$$

Exception propagation:

$$P \vdash \langle \mathtt{throw}\ e;\ e_2,s\rangle \rightarrow \langle \mathtt{throw}\ e,s\rangle$$

`if`

Exception propagation:

$$P \vdash \langle \texttt{if}\ (\texttt{throw}\ e)\ e_1\ \texttt{else}\ e_2, s\rangle \rightarrow \langle \texttt{throw}\ e, s\rangle$$

while

$$P \vdash \langle \texttt{while}\ (b)\ c, s \rangle \longrightarrow \langle \texttt{if}\ (b)\ (c\texttt{;}\ \texttt{while}\ (b)\ c)\ \texttt{else}\ \mathit{unit}, s \rangle$$

throw

$$\frac{P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle}{P \vdash \langle \texttt{throw}\ e,s\rangle \rightarrow \langle \texttt{throw}\ e',s'\rangle}$$

Exception creation:

$$P \vdash \langle \texttt{throw}\ \mathit{null},s\rangle \rightarrow \langle \texttt{THROW}\ \mathit{NullPointer},s\rangle$$

Exception propagation:

$$P \vdash \langle \texttt{throw}\ (\texttt{throw}\ e),s\rangle \rightarrow \langle \texttt{throw}\ e,s\rangle$$

`try catch` *(1)*

$$\frac{P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle}{P \vdash \langle \texttt{try}\ e\ \texttt{catch}\ (C\ V)\ e_2,s\rangle \rightarrow \langle \texttt{try}\ e'\ \texttt{catch}\ (C\ V)\ e_2,s'\rangle}$$

Normal evaluation:

$$P \vdash \langle \texttt{try}\ \texttt{Val}\ v\ \texttt{catch}\ (C\ V)\ e_2,s\rangle \rightarrow \langle \texttt{Val}\ v,s\rangle$$

`try catch` *(2)*

Exception handling:

$$[\![hp\ s\ a = \lfloor (D, fs) \rfloor ; P \vdash D \preceq^* C]\!]$$
$$\Longrightarrow P \vdash \langle \texttt{try Throw}\ a\ \texttt{catch}\ (C\ V)\ e_2, s \rangle \rightarrow$$
$$\langle \{ V{:}Class\ C; V\ \texttt{:=}\ addr\ a\ \texttt{;}\ e_2 \}, s \rangle$$

$$[\![hp\ s\ a = \lfloor (D, fs) \rfloor ; \neg P \vdash D \preceq^* C]\!]$$
$$\Longrightarrow P \vdash \langle \texttt{try Throw}\ a\ \texttt{catch}\ (C\ V)\ e_2, s \rangle \rightarrow \langle \texttt{Throw}\ a, s \rangle$$

Equivalence of big and small steps

Method bodies and local variables

Local variables for execution of method body in method call:

- Big step semantics:
  only *this* and the parameters
- Small step semantics:
  also access to all enclosing local variables

  $$\{V{:}T;\ addr\ a.M([Val\ v])\} \rightarrow$$
  $$\{V{:}T;\ \{P{:}T';\ P := Val\ v;\ e\}\}$$

  if M has method body $([P], e)$

  $\Longrightarrow$ dynamic binding — more bug than feature

Method bodies should only access *this* and the parameters

Free variables (= global variables)

$$fv :: expr \Rightarrow vname\ set$$

$$fv\ (\texttt{Cast}\ C\ e) = fv\ e$$
$$fv\ (\texttt{Val}\ v) = \emptyset$$
$$fv\ (e_1\ «bop»\ e_2) = fv\ e_1 \cup fv\ e_2$$
$$fv\ (\texttt{Var}\ V) = \{V\}$$
$$fv\ (V\ \texttt{:=}\ e) = \{V\} \cup fv\ e$$
$$fv\ (e.F\{D\}) = fv\ e$$
$$fv\ \{V{:}T\,;\ e\} = fv\ e - \{V\}$$
$$fv\ (\texttt{try}\ e_1\ \texttt{catch}\ (C\ V)\ e_2) = fv\ e_1 \cup (fv\ e_2 - \{V\})$$
$$\vdots$$

Weak well-formedness

Definition A program is *weakly well-formed* (*wwf-J-prog P*) iff every method body (Vs, e) uses only its own local variables, *this* and the parameters *Vs*:

$$fv\ e \subseteq \{this\} \cup set\ Vs$$

Equivalence

Theorem If *wwf-J-prog P* then
$P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$ iff $(P \vdash \langle e,s\rangle \rightarrow^* \langle e',s'\rangle \wedge \mathit{final}\ e')$

Proof of $\Longrightarrow$ and $\Longleftarrow$ separately.

Big $\Longrightarrow$ small

Theorem If *wwf-J-prog* P and $P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$ then $P \vdash \langle e,s\rangle \rightarrow^* \langle e',s'\rangle$.

Proof by rule induction on $\Rightarrow$. Only blocks, try-catch and (especially!) method call are non-trivial.

small $\Longrightarrow$ *Big*

Theorem If *wwf-J-prog* P and $P \vdash \langle e,s\rangle \rightarrow^* \langle e',s'\rangle$ and *final* e' then $P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$.

Proof by induction on $\rightarrow^*$.

- Basis: final expressions evaluate to themselves
- Step: by Lemma

Lemma If *wwf-J-prog* P and $P \vdash \langle e,s\rangle \rightarrow \langle e'',s''\rangle$ and $P \vdash \langle e'',s''\rangle \Rightarrow \langle e',s'\rangle$ then $P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$.

Proof by rule induction on $\rightarrow$.

Type system

Overview

- Jinja types
- Auxiliary notions
- The typing rules: what is the type of an expression?

Jinja Types

A Jinja type (type *ty*) is one of the following:

- *Boolean*
- *Integer*
- *Class C*
- *NT* (“Null Type”)
- *Void*

The formal model

$$
\begin{array}{lcl}
\textbf{datatype}\ \mathit{ty} & = & \mathit{Boolean} \\
& | & \mathit{Integer} \\
& | & \mathit{Class\ cname} \\
& | & \mathit{NT} \\
& | & \mathit{Void}
\end{array}
$$

Reference types

A reference type is either *NT* or *Class C*:

$$\mathit{is\text{-}refT}\ T \equiv T = \mathit{NT} \vee (\exists C.\ T = \mathit{Class}\ C)$$

Valid types

Types should only refer to existing classes:

$$\textit{is-type } P\ (\textit{Class } C) = \textit{is-class } P\ C$$
$$\textit{is-type } P\ T = \textit{True}$$

The type of a value

$$typeof :: heap \Rightarrow val \Rightarrow ty\ option$$

$$typeof_h\ Unit = \lfloor Void \rfloor$$

$$typeof_h\ Null = \lfloor NT \rfloor$$

$$typeof_h\ (Bool\ b) = \lfloor Boolean \rfloor$$

$$typeof_h\ (Intg\ i) = \lfloor Integer \rfloor$$

$$typeof_h\ (Addr\ a) =$$

$$(case\ h\ a\ of\ None \Rightarrow None \mid \lfloor (C, fs) \rfloor \Rightarrow \lfloor Class\ C \rfloor)$$

$$typeof_{empty}\ v \quad \equiv \quad typeof_{empty}\ v$$

From subclass to subtype

The subclass relation $P \vdash C \preceq^* C'$ induces a

subtype relation $P \vdash T \leq T'$

(Java: "T widens to T'")

$$\frac{P \vdash C \preceq^* D}{P \vdash Class\ C \leq Class\ D}$$

$$P \vdash NT \leq Class\ C$$

$$P \vdash T \leq T$$

The pointwise extension of $\leq$ to lists of types: $[\leq]$

Environments

An *environment* is a map from variable names to types:

$$E :: vname \rightharpoonup ty$$

Environments record the type of each local variable

Expression has type

$$P, E \vdash e :: T$$

Expression e has type T
(in the context of program P and environment E)

Typing rules

Val

$$\frac{typeof_{empty}\ v = \lfloor T \rfloor}{P,E \vdash \texttt{Val}\ v :: T}$$

No *Addr*esses!

Var

$$\frac{E\ V = \lfloor T \rfloor}{P, E \vdash \mathsf{Var}\ V :: T}$$

Field access

$$\frac{P,E \vdash e :: Class\ C \qquad P \vdash C\ \textbf{sees}\ F{:}T\ \textbf{in}\ D}{P,E \vdash e.F\{D\} :: T}$$

D must be the class declaring F

Can be viewed as test or computation of D

Binary operations

$$\frac{P,E \vdash e_1 :: T_1 \quad P,E \vdash e_2 :: T_2 \quad P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1}{P,E \vdash e_1 \text{ «=» } e_2 :: \mathit{Boolean}}$$

$$\frac{P,E \vdash e_1 :: \mathit{Integer} \qquad P,E \vdash e_2 :: \mathit{Integer}}{P,E \vdash e_1 \text{ «+» } e_2 :: \mathit{Integer}}$$

new

$$\frac{\mathit{is\text{-}class}\ P\ C}{P, E \vdash \mathsf{new}\ C :: \mathit{Class}\ C}$$

Cast

$$\frac{P,E \vdash e :: \mathit{Class}\ D \quad \mathit{is\text{-}class}\ P\ C \quad P \vdash C \preceq^* D \vee P \vdash D \preceq^* C}{P,E \vdash \texttt{Cast}\ C\ e :: \mathit{Class}\ C}$$

Variable assignment

$$\frac{E\ V = \lfloor T \rfloor \qquad P,E \vdash e :: T' \qquad P \vdash T' \leq T \qquad V \neq \mathit{this}}{P,E \vdash V := e :: T'}$$

In Java: $P,E \vdash V := e :: ?$

Field assignment

$$\frac{P,E \vdash e_1 :: \mathit{Class}\ C \quad P \vdash C\ \textbf{sees}\ F{:}T\ \textbf{in}\ D \quad P,E \vdash e_2 :: T' \quad P \vdash T' \leq T}{P,E \vdash e_1.F\{D\} := e_2 :: T'}$$

Method call

$$\frac{\begin{array}{c} P,E \vdash e :: Class\ C \\ P \vdash C\ \textbf{sees}\ M\colon Ts \to T = (pns, body)\ \textbf{in}\ D \\ P,E \vdash es\ [::]\ Ts' \\ P \vdash Ts'\ [\leq]\ Ts \end{array}}{P,E \vdash e.M(es) :: T}$$

Expression lists

$$P,E \vdash es\ [::]\ Ts$$

$$P,E \vdash []\ [::]\ []$$

$$\frac{P,E \vdash e :: T \qquad P,E \vdash es\ [::]\ Ts}{P,E \vdash e{\cdot}es\ [::]\ T{\cdot}Ts}$$

Local variable

$$\frac{\textit{is-type}\ P\ T \qquad P, E(V \mapsto T) \vdash e :: T'}{P, E \vdash \{V{:}T\,;\, e\} :: T'}$$

Sequential composition

$$\frac{P, E \vdash e_1 :: T_1 \qquad P, E \vdash e_2 :: T_2}{P, E \vdash e_1 \,;\, e_2 :: T_2}$$

Conditional

$$\frac{\begin{array}{c} P,E \vdash e :: \mathit{Boolean} \\ P,E \vdash e_1 :: T_1 \qquad P,E \vdash e_2 :: T_2 \\ P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1 \\ P \vdash T_1 \leq T_2 \longrightarrow T = T_2 \\ P \vdash T_2 \leq T_1 \longrightarrow T = T_1 \end{array}}{P,E \vdash \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2 :: T}$$

While loop

$$\frac{P,E \vdash e :: \mathit{Boolean} \qquad P,E \vdash c :: T}{P,E \vdash \texttt{while}\ (e)\ c :: \mathit{Void}}$$

throw

$$\frac{P,E \vdash e :: \mathit{Class}\ C}{P,E \vdash \texttt{throw}\ e :: \mathit{Void}}$$

Alternative (e.g. in ML):

$$\frac{P,E \vdash e :: \mathit{Class}\ C}{P,E \vdash \texttt{throw}\ e :: T}$$

Advantage:

Exceptions may be used *anywhere*

Complication:

Expressions may have multiple (unrelated) types

try-catch

$$\frac{P,E \vdash e_1 :: T \quad P,E(V \mapsto \mathit{Class}\ C) \vdash e_2 :: T \quad \mathit{is\text{-}class}\ P\ C}{P,E \vdash \texttt{try}\ e_1\ \texttt{catch}\ (C\ V)\ e_2 :: T}$$

Generalize?

Remarks (1)

The type of an expression
depends only on the type of its subexpressions.

$\Longrightarrow$ The rules form a terminating (Prolog) program
for computing the type of an expression.

Remarks (2)

The rules are *syntax directed*:
exactly one rule per construct

$\Longrightarrow$ Each rule can be turned into an equivalence:

$$(P,E \vdash \texttt{Val}\ v :: T) = (\mathit{typeof}_{\mathit{empty}}\ v = \lfloor T \rfloor)$$

$$(P,E \vdash V{:=}e :: T) =$$

$$(V \neq \mathit{this} \wedge (\exists T'.\ E\ V = \lfloor T' \rfloor \wedge P,E \vdash e :: T \wedge P \vdash T \leq T'))$$

$\vdots$

Terminology

We say e is *well-typed* or *type correct* (in the context of P and E) iff there is a T such that $P,E \vdash e :: T$ follows from the typing rules.

Uniqueness

Thm Every well-typed expression has a unique type:

If $P, E \vdash e :: T$ and $P, E \vdash e :: T'$ then $T = T'$.

Equivalent: Every expression has at most one type.

Annotating expressions with {D}

Adding {D}

$$P, E \vdash e \leadsto e'$$

where

- e is an expression *without* $\{D\}$ annotations, the input
- e' is an expression *with* $\{D\}$ annotations, the output

Some typical rules for $\mathbf{P,E \vdash e \leadsto e'}$

$$P,E \vdash \texttt{Val}\ v \leadsto \texttt{Val}\ v$$

$$\frac{P,E(V \mapsto T) \vdash e \leadsto e'}{P,E \vdash \{V{:}T\,;\, e\} \leadsto \{V{:}T\,;\, e'\}}$$

$$\frac{P,E \vdash e1 \leadsto e1' \qquad P,E \vdash e2 \leadsto e2'}{P,E \vdash e1\,;\, e2 \leadsto e1'\,;\, e2'}$$

Adding the annotation

$$\frac{P,E \vdash e \rightsquigarrow e' \qquad P,E \vdash e' :: \mathit{Class}\ C \qquad P \vdash C\ \textbf{sees}\ F{:}T\ \textbf{in}\ D}{P,E \vdash e.F \rightsquigarrow e'.F\{D\}}$$

$$\frac{P,E \vdash e_1 \rightsquigarrow {e_1}' \qquad P,E \vdash e_2 \rightsquigarrow {e_2}' \qquad P,E \vdash {e_1}' :: \mathit{Class}\ C \qquad P \vdash C\ \textbf{sees}\ F{:}T\ \textbf{in}\ D}{P,E \vdash e_1.F := e_2 \rightsquigarrow {e_1}'.F\{D\} := {e_2}'}$$

Is it a local variable or a field?

$$\frac{E\ V = \mathit{None} \qquad E\ \mathit{this} = \lfloor \mathit{Class}\ C \rfloor \qquad P \vdash C\ \textbf{sees}\ V{:}T\ \textbf{in}\ D}{P,E \vdash \mathsf{Var}\ V \rightsquigarrow \mathsf{Var}\ \mathit{this}.V\{D\}}$$

$$\frac{E\ V = \lfloor T \rfloor}{P,E \vdash \mathsf{Var}\ V \rightsquigarrow \mathsf{Var}\ V}$$

Summary

$$P, E \vdash e \leadsto e'$$

- Adds annotations to field access and update
- Turns variables into fields if necessary
- Can be viewed as a preprocessor
- Is normally part of the type checker in a compiler

Well-formedness

Aim

Formalization of all context conditions (*static semantics*)

"Types and more"

When do we have enough?

If nothing bad can happen during the execution of well-formed programs

How is "nothing bad can happen" formalized?

Execution does not get stuck (*type safety*, later)

Overview

- Definite assignment
- Method overriding
- Well-formedness

Definite assignment

Definite assignment

Variables must be assigned to before use

Fields are automatically initialized with their default value upon object creation

In theory

Thm It is in general undecidable if a variable/field has been initialized before use.
Proof $e;\ \{V{:}T;\ \mathtt{var}\ V\}$

1. V is not initialized before use iff e terminates (normally).
2. Therefore a program analyzer would need to decide whether e terminates.
3. But it is in general undecidable if a given Turing machine/program/expression e terminates.

In practice: variables

Fact It is easy to check if a variable has been initialized before use: if in doubt, be conservative.

Examples:

- `if` (b) V `:= Val` v `else Val` v`; Var` V
  is rejected because b could be false.
  What if b is never false? Then `if` is pointless!
- `throw` e`;` $\{V{:}T\,;$ `Var` $V\}$
  is rejected.

Rejection is not a problem:

> Insert dummy initialization $V := \mathit{Val}\ v$
> right after declaration of V.

In practice: fields

Fact It is hard to check if a field of an object has been initialized before use.

Example: Let *p* be a formal parameter of class type. Inside the method body

- variable *p* is initialized,
- but is field *F* of *p* initialized??

Java and Jinja initialize all fields of an object with their default value.

$\mathcal{A}$

$$\mathcal{A} :: expr \Rightarrow vname\ set$$

$\mathcal{A}\ e =$ the set of all (global) variables that are definitely assigned to if e terminates normally

$$\begin{aligned}
\mathcal{A}\ (\texttt{new}\ C) &= \emptyset \\
\mathcal{A}\ (\texttt{Cast}\ C\ e) &= \mathcal{A}\ e \\
\mathcal{A}\ (\texttt{Val}\ v) &= \emptyset \\
\mathcal{A}\ (e_1\ \text{«}bop\text{»}\ e_2) &= \mathcal{A}\ e_1 \cup \mathcal{A}\ e_2 \\
\mathcal{A}\ (\texttt{Var}\ V) &= \emptyset \\
\mathcal{A}\ (V := e) &= \{V\} \cup \mathcal{A}\ e \\
\mathcal{A}\ (e.F\{D\}) &= \mathcal{A}\ e \\
\mathcal{A}\ (e_1.F\{D\} := e_2) &= \mathcal{A}\ e_1 \cup \mathcal{A}\ e_2 \\
\mathcal{A}\ (e.M(es)) &= \mathcal{A}\ e \cup (\bigcup e \in set\ es.\ \mathcal{A}\ e)
\end{aligned}$$

$\mathcal{A}$

$$
\begin{array}{ll}
\mathcal{A}\ \{V{:}T\,;\ e\} & = \mathcal{A}\ e - \{V\} \\
\mathcal{A}\ (e_1\texttt{;}\ e_2) & = \mathcal{A}\ e_1 \cup \mathcal{A}\ e_2 \\
\mathcal{A}\ (\texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2) & = \mathcal{A}\ e \cup (\mathcal{A}\ e_1 \cap \mathcal{A}\ e_2) \\
\mathcal{A}\ (\texttt{while}\ (b)\ e) & = \mathcal{A}\ b \\
\mathcal{A}\ (\texttt{throw}\ e) & = \{V \mid \mathit{True} e\} \\
\mathcal{A}\ (\texttt{try}\ e_1\ \texttt{catch}\ (C\ V)\ e_2) & = \mathcal{A}\ e_1 \cap (\mathcal{A}\ e_2 - \{V\})
\end{array}
$$

Motivation for *throw*:

$$\mathcal{A}\ (\texttt{if}\ (e)\ V\ \texttt{:=}\ \texttt{Val}\ v\ \texttt{else}\ \texttt{throw}\ e_2) = \mathcal{A}\ e \cup \{V\}$$

$\mathcal{D}$

$$\mathcal{D} :: \mathit{expr} \Rightarrow \mathit{vname\ set} \Rightarrow \mathit{bool}$$

$\mathcal{D}\ e\ A =$
if initially all variables in A are initialized
then execution of e does not access an uninitialized variable

$$\begin{aligned}
\mathcal{D}\ (\mathtt{new}\ C)\ A &= \mathit{True} \\
\mathcal{D}\ (\mathtt{Cast}\ C\ e)\ A &= \mathcal{D}\ e\ A \\
\mathcal{D}\ (\mathtt{Val}\ v)\ A &= \mathit{True} \\
\mathcal{D}\ (e_1\ \text{«}\mathit{bop}\text{»}\ e_2)\ A &= (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \cup \mathcal{A}\ e_1)) \\
\mathcal{D}\ (\mathtt{Var}\ V)\ A &= (V \in A) \\
\mathcal{D}\ (V := e)\ A &= \mathcal{D}\ e\ A \\
\mathcal{D}\ (e.F\{D\})\ A &= \mathcal{D}\ e\ A \\
\mathcal{D}\ (e_1.F\{D\} := e_2)\ A &= (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \cup \mathcal{A}\ e_1))
\end{aligned}$$

$\mathcal{D}$

$$
\begin{aligned}
&\mathcal{D}\ \{V{:}T\,;\ e\}\ A && = \mathcal{D}\ e\ (A - \{V\}) \\
&\mathcal{D}\ (e_1\texttt{;}\ e_2)\ A && = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \cup \mathcal{A}\ e_1)) \\
&\mathcal{D}\ (\texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2)\ A && = \\
&\qquad (\mathcal{D}\ e\ A \wedge \mathcal{D}\ e_1\ (A \cup \mathcal{A}\ e) \wedge \mathcal{D}\ e_2\ (A \cup \mathcal{A}\ e)) \\
&\mathcal{D}\ (\texttt{while}\ (b)\ e)\ A && = (\mathcal{D}\ b\ A \wedge \mathcal{D}\ e\ (A \cup \mathcal{A}\ b)) \\
&\mathcal{D}\ (\texttt{throw}\ e)\ A && = \mathcal{D}\ e\ A \\
&\mathcal{D}\ (\texttt{try}\ e_1\ \texttt{catch}\ (C\ V)\ e_2)\ A && = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \cup \{V\}))
\end{aligned}
$$

Well-formedness (sneak preview)

Each method body (Vs, e) must fulfill the condition

$$\mathcal{D}\ e\ (\{this\} \cup set\ Vs)$$

Correctness

Thm If $P \vdash \langle e,(h, l)\rangle \Rightarrow \langle e',(h', l')\rangle$ then $\mathcal{A}\ e \subseteq dom\ l'$
Proof by rule induction over $\Rightarrow$

Correctness of $\mathcal{D}$: part of type safety proof (later)

A small imprecision

Let $e = \texttt{if}\ (\texttt{Var}\ B)\ \{V{:}T;\ \texttt{throw}\ \_\}\ \texttt{else}\ V\ \texttt{:=}\ \mathit{true}$

$\mathcal{A}\ \{V{:}T;\ \texttt{throw}\ \_\} = \mathcal{A}\ (\texttt{throw}\ \_) - \{V\}$

$\mathcal{A}\ e = \mathcal{A}(\mathit{Var}\ B) \cup (\mathcal{A}\ \{V{:}T;\ \texttt{throw}\ \_\} \cap \mathcal{A}(V := \mathit{true}))$
$= \{\}$

$\mathcal{D}\ (e\texttt{;}\ \texttt{Var}\ V)\ \{B\} =$
$(\mathcal{D}\ e\ \{B\} \wedge \mathcal{D}(\mathit{Var}\ V)\ (\{B\} \cup \{\})) =$
$(\mathcal{D}\ e\ \{B\} \wedge V \in \{B\}) = \mathit{False}$

$\mathcal{A}\ e = \{\}$ is too pessimistic but not incorrect

The problem

$\mathcal{A}\ \{V{:}T\,;\ \texttt{throw}\ \_\}$
should be $\{V \mid True\}$ and not $\{V \mid True\} - \{V\}$.

Posible solutions:

- Redefine $\mathcal{A}$ and $\mathcal{D}$.
- Do not allow nested declaration of the same V. Java! Then it does not matter if $V \in \mathcal{A}\ \{V{:}T\,;\ \_\}$ or not.

We leave things as they are and put up with the small imprecision.

Method overriding

The situation

```
class C extends B
{...
 method M(p:T):R = ...
  ...
}

class D extends C
{...
 method M(p:T’):R’ = ...
  ...
}
```

`M` in `D` *overrides* (overwrites, redefines) `M` in `C`

The question

How should `T` and `T'` be related?
How should `R` and `R'` be related?

Guiding principle: Type Safety

If *e* is statically (at compile time) of class *C*
then the evaluation of *e* should yield a subclass of *C*

Then $e.M(\ldots)$ can never fail at runtime as in SmallTalk:

`Method not understood`

Covariance in the result type

$$\texttt{R'} \leq \texttt{R}$$

New result type must be *subtype* of old result type

Otherwise:

```
class R' { }
class R extends R' { method M2() = ...}

  {V:C; V := new D; V.M(...).M2()}
```

is type correct but semantics gets stuck.

Contravariance in the argument type

$$\texttt{T} \leq \texttt{T}'$$

New argument type must be *supertype* of old argument type

Why?

1. Assume `e.M(e')` is statically well-typed.
2. Assume `e` has static type `C`.
3. Then the static type of `e'` is $\leq$ `T`.
4. Now assume the dynamic type of `e` is `D`
5. Thus the dynamic type of `e'` must be $\leq$ `T'`.
6. But we can only guarantee `e'` has type `T`.
7. $\Longrightarrow$ $\texttt{T} \leq \texttt{T}'$

Example

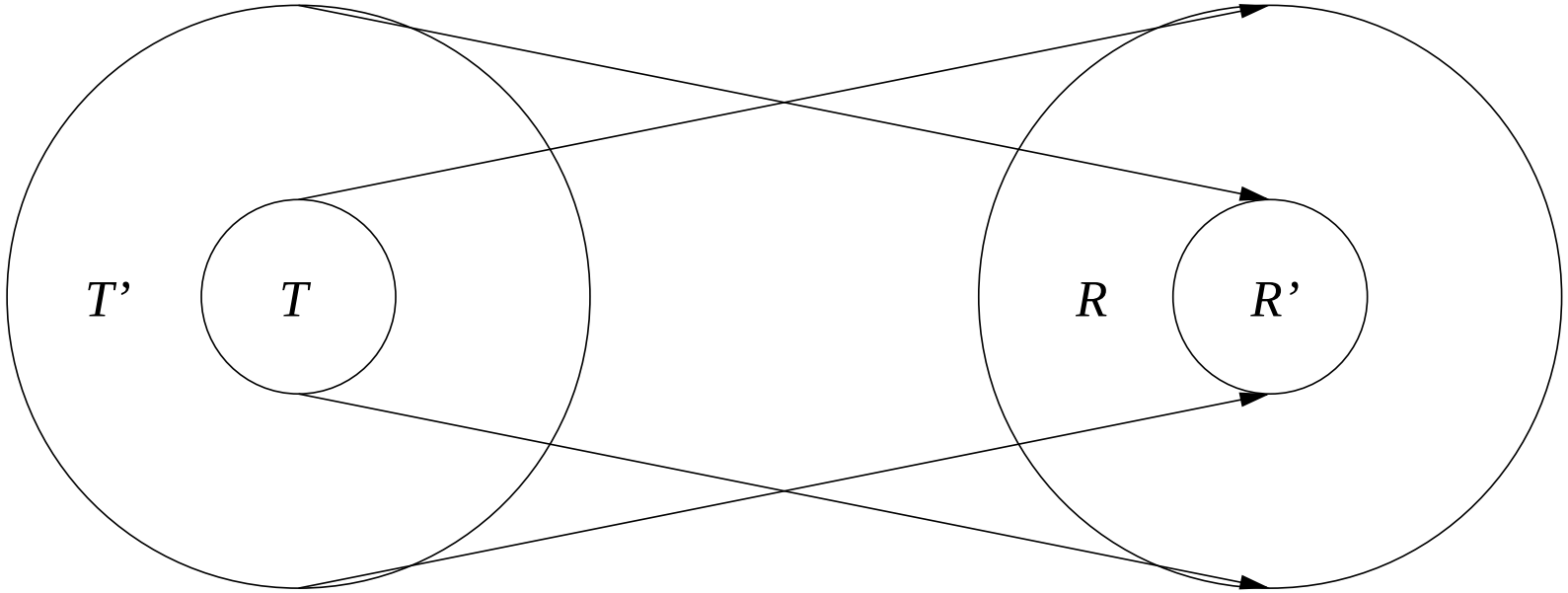
```
class C extends B
{ method M(p:T):R = ... }

class D extends C
{ method M(p:T'):R = p.M2() }

class T { }

class T' extends T { method M2():R = ... }
```

Problem?

```
{V:C; V := new D; V.M(new T)}
```

is type correct but semantics gets stuck.

The set theoretic perspective



For total functions:

$$\frac{T \subseteq T' \quad R' \subseteq R}{T' \to R' \subseteq T \to R}$$

Terminology

$\_ \to \_$ is	*contravariant*	in the	first	argument
$\_ \to \_$ is	*covariant*	in the	second	argument

Alternatively: *antimonotone*, *monotone*

Method overriding in Jinja, Java and Eiffel

Jinja Contravariant in parameters, covariant in result

Classic Java Invariant:

If $T=T'$ then $R=R'$
else overloading, not overriding.

Overloading: multiple methods with the same name but different parameter types are visible at the same time. Use static type of actual parameters in method selection (tricky).

Java 1.5 Invariant in parameters, covariant in result

Eiffel Covariant in parameters and result (not type safe)

An example

```
class Point
{field x: Integer
 method eq(p:Point):Boolean = (this.x = p.x)
}

class ColPoint extends Point
{field col: Integer
 method eq(cp:ColPoint):Boolean =
  (if (this.x = cp.x) (this.col = cp.col)
   else false)
}
```

Java versus Eiffel

```
{p1:ColPoint; p1 := new ColPoint;
 p1.x := 5; p1.col := 0;
 {p2:Point; p2 := new ColPoint;
  p2.x := 5; p2.col := 1;
  p1.eq(p2)}}
```

evaluates to

- in Java: `true`
- in Eiffel: `false`
- in Jinja: `eq` in `ColPoint` not wellformed!

How can we program `ColPoint` in Jinja?

`ColPoint` *in Jinja*

```
class ColPoint extends Point
{field col:  Integer
 method eqCP(cp:ColPoint):Boolean =
  (if (this.x = cp.x) (this.col = cp.col)
   else false)
}
```

Similar to Java: make argument type part of method name.
In Java this happens implicitly via overloading.

Well-formedness

Well-formed program P

wf-J-prog P

- The system classes are declared:
  $\{Object, NullPointer, ClassCast, OutOfMemory\} \subseteq set\,(map\ fst\ P)$
- No class is declared twice:
  $distinct\ (map\ fst\ P)$
- Every class declaration $(C, D, fs, ms) \in set\ P$ is well-formed.

Well-formed class declaration (C,D,fs,ms)

- All field declarations in *fs* are well-formed
- No field in *fs* is declared twice
- All method declarations in *ms* are well-formed
- No method in *ms* is declared twice
- If $C \neq Object$ then
  - D is a class in P
  - D is not a subclass of C: $\neg\, P \vdash D \preceq^* C$
  - *Overriding:* if $(M, Ts, T, mb) \in set\ ms$ and $P \vdash D$ **sees** $M: Ts' \to T' = mb'$ **in** D' then $P \vdash T \leq T'$ and $P \vdash Ts'\ [\leq]\ Ts$
    Method overriding is *covariant* in the result type and *contravariant* in the argument types

Well-formed method declaration (**M**, **Ts**, **T**, **Vs**, **e**)

- All parameter types are valid: $\forall\, T \in \mathit{set}\ \mathit{Ts}.\ \mathit{is\text{-}type}\ P\ T$
- The result type is valid: *is-type P T*
- There are as many parameter types as names: $|\mathit{Ts}| = |\mathit{Vs}|$
- All parameter names are distinct: *distinct Vs*
- *this* is not a parameter name: $\mathit{this} \notin \mathit{set}\ \mathit{Vs}$
- The method body is well-typed: $\exists\, T'.\ P, \mathit{empty}(\mathit{this} \mapsto \mathit{Class}\ C, \mathit{Vs}\ [\mapsto]\ \mathit{Ts}) \vdash e :: T' \wedge P \vdash T' \leq T$
- All local variables are assigned to before use: $\mathcal{D}\ e\ (\{\mathit{this}\} \cup \mathit{set}\ \mathit{Vs})$

Well-formed field declaration $(\mathbf{F}, \mathbf{T})$

The type is valid: *is-type P T*

Type Safety

What is type safety?

The type system guarantees safety of the execution.

means

Execution of well-typed terms does not get stuck.

means

If *e* is well-typed and not final then *e* can be reduced.

Well-typed expressions do not go wrong.

(Robin Milner, *A Theory of Type Polymorphism in Programming*, 1978)

Big step versus small step semantics

When is $\langle e,s \rangle$ stuck?

- Small step semantics: $\nexists e' s'.\ P \vdash \langle e,s \rangle \rightarrow \langle e',s' \rangle$
- Big step semantics: $\nexists e' s'.\ P \vdash \langle e,s \rangle \Rightarrow \langle e',s' \rangle$ No!

Big step semantics cannot distinguish between being stuck and nontermination!

Example:

- $\langle \mathit{while}\ (\mathit{true})\ \mathit{unit}, s \rangle$ does not terminate
- $\langle V := \mathit{Var}\ V, (h, \mathit{empty}) \rangle$ is stuck

The advantages of type safety

- For the programer:

  No uncontrollable runtime errors as in C.

- For the language implementor:

  Many explicit runtime checks unnecessary.

Example: Well-typedness of $\langle addr\ a.F\{D\},(h,l)\rangle$ will guarantee $a \in dom\ h$.

Decomposing type safety

Wanted:

If e is well-typed, its reduction can *never* get stuck.

Sufficient?

If e is well-typed and not final then e can be reduced (via $\rightarrow$) to some e':

No! What if e' is not well-typed any more?

Type safety = Progress $\wedge$ Preservation

Progress: Well-typed non-final expressions can be reduced.

Preservation: Reduction preserves well-typedness.
If e has type T and reduces to e' then e' has type T' where $T' \leq T$.

Progress

Formalization of progress (1)

First attempt:

If $P,E \vdash e :: T$ and $\neg$ *final* e then $\exists e' s'.\ P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle$

Not true for arbitrary s:

- Reduction of $\langle \mathit{Var}\ V,(h,l)\rangle$ needs $V \in \mathit{dom}\ l$
  In general: $\mathcal{D}\ e\ (\mathit{dom}\ l)$
- Reduction of $\langle \mathit{addr}\ a.F\{D\},(h,l)\rangle$ needs
  if $h\ a = \lfloor(C, fs)\rfloor$ then $(F, D) \in \mathit{dom}\ fs$
  In general: each object on the heap must have all the fields required by its class.

Heap conformance

Def Heap h conforms to program P iff each object in h conforms to P and all system exceptions are preallocated.

Def Object (C, fs) conforms to program P iff for every field $F{:}T$ declared in D that C has, there is a value v such that $fs\ (F, D) = \lfloor v \rfloor$ and the type of v is a subtype of T.

Heap conformance (formally)

$$P \vdash h\ \surd \equiv$$

$$(\forall\, a\ obj.\ h\ a = \lfloor obj \rfloor \longrightarrow P,h \vdash obj\ \surd) \wedge preallocated\ h$$

$$preallocated\ h \equiv$$

$$\forall\, C \in sys\text{-}xcpts.\ \exists\, fs.\ h\ (addr\text{-}of\text{-}sys\text{-}xcpt\ C) = \lfloor (C, fs) \rfloor$$

Object conformance (formally)

$$P,h \vdash (C, fs) \surd \equiv$$

$$\forall F\ D\ T.$$

$$P \vdash C\ \textbf{has}\ F{:}T\ \textbf{in}\ D \longrightarrow$$

$$(\exists v.\ fs\ (F, D) = \lfloor v \rfloor \wedge P,h \vdash v :\leq T)$$

$$P,h \vdash v :\leq T \equiv \exists T'.\ typeof_h\ v = \lfloor T' \rfloor \wedge P \vdash T' \leq T$$

Formalization of progress (2)

If $P,E \vdash e :: T$ and $P \vdash h \surd$ and $\mathcal{D}\ e\ (dom\ l)$ and $\neg\ final\ e$ then $\exists e' s'.\ P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle$.

Problem: e' need not be well-typed anymore.
Why?

- e' may contain *Addr*esses.
- The type of a subexpression of e' may have decreased.
  Example: $P \vdash \langle \texttt{Cast}\ C\ e,s\rangle \rightarrow \langle \texttt{Cast}\ C\ e',s'\rangle$
  - $P,E \vdash e :: Class\ D$, $\quad P \vdash C \preceq^* D$ (down cast)
  - $P,E \vdash e' :: Class\ D'$, $\quad C$ and D' are incomparable.

Solution: modified (more liberal) type system $P,E,h \vdash e : T$

Two kinds of expressions

Input expressions:

- do not contain addresses
- must satisfy various pragmatic type conditions (eg only up or down casts)
- are checked statically by $P,E \vdash e :: T$

Runtime expressions: any expression reached via reduction ($\rightarrow^*$) from an input expression

Aim: show that runtime expressions satisfy an *invariant* expressed as a weaker type system $P,E,h \vdash e : T$

There is no runtime type checking in Jinja!

Just a technical device for the proof of type safety.

Requirements for $\mathbf{P,E,h \vdash e : T}$

- Weaker than input type system:
  $P,E \vdash e :: T \Longrightarrow P,E,h \vdash e : T$
- Must ensure progress
- Must ensure (type) preservation

The rules for P,E,h ⊢ e : T

Many rules are similar to their $P,E \vdash e :: T$ counterpart, but with h added.

Examples

$$\frac{typeof_h\ v = \lfloor T \rfloor}{P,E,h \vdash \texttt{Val}\ v : T}$$

$$\frac{P,E,h \vdash e_1 : T_1 \qquad P,E,h \vdash e_2 : T_2}{P,E,h \vdash e_1\texttt{;}\ e_2 : T_2}$$

We concentrate on the rules that have really changed.

Reasons for change

Reduction of subterm reduces its type.

Extreme case: subterm becomes *null*.

Binary operations

$$\frac{P,E,h \vdash e_1 : T_1 \qquad P,E,h \vdash e_2 : T_2}{P,E,h \vdash e_1 \text{ «=» } e_2 : \mathit{Boolean}}$$

$$\frac{P,E,h \vdash e_1 : \mathit{Integer} \qquad P,E,h \vdash e_2 : \mathit{Integer}}{P,E,h \vdash e_1 \text{ «+» } e_2 : \mathit{Integer}}$$

Cast

$$\frac{P,E,h \vdash e : T \qquad \textit{is-refT } T \qquad \textit{is-class } P\ C}{P,E,h \vdash \texttt{Cast}\ C\ e : \textit{Class } C}$$

Variable assignment

$$\frac{E\ V = \lfloor T \rfloor \qquad P,E,h \vdash e : T' \qquad P \vdash T' \leq T}{P,E,h \vdash V := e : T'}$$

Just to show that $V \neq this$ is not necessary for type safety.

Field access

$$\frac{P,E,h \vdash e : \mathit{Class}\ C \qquad P \vdash C\ \textbf{has}\ F{:}T\ \textbf{in}\ D}{P,E,h \vdash e.F\{D\} : T}$$

- Statically: *sees*
- Dynamically: *has*

Is that all?

$$\frac{P,E,h \vdash e : NT}{P,E,h \vdash e.F\{D\} : T}$$

Field assignment

$$\frac{P,E,h \vdash e_1 : \mathit{Class}\ C \qquad P \vdash C\ \textbf{has}\ F{:}T\ \textbf{in}\ D \qquad P,E,h \vdash e_2 : T_2 \qquad P \vdash T_2 \leq T}{P,E,h \vdash e_1.F\{D\} := e_2 : T_2}$$

$$\frac{P,E,h \vdash e_1 : \mathit{NT} \qquad P,E,h \vdash e_2 : T_2}{P,E,h \vdash e_1.F\{D\} := e_2 : T_2}$$

has field

$P \vdash C$ **has** $F{:}T$ **in** D means that in P a (not necessarily proper) superclass D of C has a field F of type T.

Formal definition:

$P \vdash C$ **has** $F{:}T$ **in** $D \equiv$

$\exists\, \mathit{FDTs}.\ P \vdash C$ **has-fields** $\mathit{FDTs} \wedge \mathit{map\text{-}of\ FDTs}\ (F, D) = \lfloor T \rfloor$

Example

```
class B extends A {field F:TB}
class C extends B {field F:TC}
```

$P \vdash C$ **sees** $F:TC$ **in** C	YES
$P \vdash C$ **sees** $F:TB$ **in** B	NO
$P \vdash C$ **has** $F:TC$ **in** C	YES
$P \vdash C$ **has** $F:TB$ **in** B	YES

Method call

As before:

$$\frac{\begin{array}{c} P,E,h \vdash e : Class\ C \\ P \vdash C\ \textbf{sees}\ M\colon Ts \to T = (pns, body)\ \textbf{in}\ D \\ P,E,h \vdash es\ [:]\ Ts' \\ P \vdash Ts'\ [\leq]\ Ts \end{array}}{P,E,h \vdash e.M(es) : T}$$

In addition:

$$\frac{P,E,h \vdash e : NT \qquad P,E,h \vdash es\ [:]\ Ts}{P,E,h \vdash e.M(es) : T}$$

Local variable

$$\frac{P,E(V \mapsto T),h \vdash e : T'}{P,E,h \vdash \{V{:}T; e\} : T'}$$

is-type P T is not necessary for type safety.

throw

$$\frac{P,E,h \vdash e : T_r \qquad \textit{is-ref}T\ T_r}{P,E,h \vdash \texttt{throw}\ e : T}$$

try-catch

$$\frac{P,E,h \vdash e_1 : T_1 \quad P,E(V \mapsto \mathit{Class}\ C),h \vdash e_2 : T_2 \quad P \vdash T_1 \leq T_2}{P,E,h \vdash \texttt{try}\ e_1\ \texttt{catch}\ (C\ V)\ e_2 : T_2}$$

Uniqueness of types

Expressions do not have unique types w.r.t. $P, E, h \vdash e : T$

So what?

$P, E, h \vdash e : T$ is not used to compute T
but to show that e has a type.

Easy lemma

Lemma If $P,E \vdash e :: T$ then $P,E,h \vdash e : T$.

Proof Easy rule induction.

Progress

If everything is ok and e is not final then e reduces:

Thm (Progress) If $\mathit{wf\text{-}J\text{-}prog}\ P$ and $P,E,h \vdash e : T$ and $P \vdash h\ \surd$ and $\mathcal{D}\ e\ (\mathit{dom}\ l)$ and $\neg\ \mathit{final}\ e$ then $\exists e'\ s'.\ P \vdash \langle e,(h, l)\rangle \rightarrow \langle e',s'\rangle$.

Why

$\mathit{wf\text{-}J\text{-}prog}\ P$:	method call
$P \vdash h\ \surd$:	field access
$\mathcal{D}\ e\ (\mathit{dom}\ l)$:	variable access

Preservation

Formalization of preservation

First attempt:

If $P,E,h \vdash e : T$ and $P \vdash \langle e,(h, l)\rangle \rightarrow \langle e',(h', l')\rangle$ then
$\exists\, T'.\ P,E,h' \vdash e' : T' \wedge P \vdash T' \leq T$

No true for arbitrary (h,l):

- Reduction of field access needs:
  field must have value of the right type,
  i.e. $P \vdash h\ \surd$
- Reduction of variable access needs:
  variable must have value of the right type,
  i.e. l (values) must conform to E (types).

Local variable and state conformance

Def Local variables *l* conform to environment *E* iff each variable $V \in dom\ l$ has a value conforming to type $E\ V$.

$$P,h \vdash l\ (:\leq)\ E \equiv$$

$$\forall V\ v.\ l\ V = \lfloor v \rfloor \longrightarrow (\exists T.\ E\ V = \lfloor T \rfloor \wedge P,h \vdash v :\leq T)$$

State conformance:

$$P,E \vdash (h, l)\ \surd \equiv P \vdash h\ \surd \wedge P,h \vdash l\ (:\leq)\ E$$

Single step type preservation

If everything is ok, reduction preserves well-typedness and may reduce type:

Thm If *wf-J-prog* P and $P \vdash \langle e,(h, l)\rangle \rightarrow \langle e',(h', l')\rangle$ and $P,E \vdash (h, l)\ \surd$ and $P,E,h \vdash e : T$ then $\exists T'.\ P,E,h' \vdash e' : T' \wedge P \vdash T' \leq T$.

Proof by rule induction on $\rightarrow$.

A complication

Example case:

$$\frac{P \vdash \langle e,(h, l)\rangle \rightarrow \langle e',(h', l')\rangle}{P \vdash \langle e;\ e_2,(h, l)\rangle \rightarrow \langle e';\ e_2,(h', l')\rangle}$$

Complication:

does $P,E,h \vdash e_2 : T_2$ imply $P,E,h' \vdash e_2 : T_2$?

Yes, because h changes only in a safe fashion:

if $h\ a = \lfloor(C, fs)\rfloor$ then $h'\ a = \lfloor(C, fs')\rfloor$

The class of an object on the heap stays fixed

The heap extension relation $\trianglelefteq$

Definition

$$h \trianglelefteq h' \equiv \forall\, a\, C\, fs.\, h\, a = \lfloor (C, fs) \rfloor \longrightarrow (\exists\, fs'.\, h'\, a = \lfloor (C, fs') \rfloor)$$

Lemma If $P \vdash \langle e,(h, l)\rangle \rightarrow \langle e',(h', l')\rangle$ then $h \trianglelefteq h'$.
Proof by rule induction on $\rightarrow$.

Lemma If $P,E,h \vdash e : T$ and $h \trianglelefteq h'$ then $P,E,h' \vdash e : T$.
Proof by rule induction on $P,E,h \vdash e : T$.

Back to type preservation

Now single step type preservation can be proved:

Thm If *wf-J-prog* P and $P \vdash \langle e,(h, l)\rangle \rightarrow \langle e',(h', l')\rangle$ and $P,E \vdash (h, l)\ \surd$ and $P,E,h \vdash e : T$ then $\exists T'.\ P,E,h' \vdash e' : T' \wedge P \vdash T' \leq T$.

Extension to $\rightarrow^*$ needs preservation of conformance.

Preservation of state conformance

Lemma If $P \vdash \langle e,(h, l)\rangle \rightarrow \langle e',(h', l')\rangle$ and $P,E,h \vdash e : T$ and $P \vdash h\ \surd$ then $P \vdash h'\ \surd$.

Proof by rule induction on $\rightarrow$.

Lemma If $P \vdash \langle e,(h, l)\rangle \rightarrow \langle e',(h', l')\rangle$ and $P,E,h \vdash e : T$ and $P,h \vdash l\ (:\leq)\ E$ then $P,h' \vdash l'\ (:\leq)\ E$.

Proof by rule induction on $\rightarrow$.

Many step type preservation

Thm If $\mathit{wf\text{-}J\text{-}prog}\ P$ and $P \vdash \langle e,s\rangle \rightarrow^* \langle e',s'\rangle$ and $P,E \vdash s\ \surd$ and $P,E,\mathit{hp}\ s \vdash e : T$ then $\exists T'.\ P,E,\mathit{hp}\ s' \vdash e' : T' \wedge P \vdash T' \leq T$.

Proof by induction on the length of the reduction sequence, i.e. by rule induction on $\rightarrow^*$.

Preservation of $\mathcal{D}$

Lemma If *wf-J-prog P* and $P \vdash \langle e,(h, l)\rangle \rightarrow \langle e',(h', l')\rangle$ and $\mathcal{D}\ e\ (dom\ l)$ then $\mathcal{D}\ e'\ (dom\ l')$.

Proof by rule induction on $\rightarrow$.

Finally: type safety

Irreducible expressions are values or exceptions:

Corollary If *wf-J-prog* P and $P,E \vdash s\ \surd$ and $P,E \vdash e :: T$ and $\mathcal{D}\ e\ (dom(lcl\ s))$ and $P \vdash \langle e,s\rangle \rightarrow^* \langle e',s'\rangle$ and $\nexists e''\ s''.\ P \vdash \langle e',s'\rangle \rightarrow \langle e'',s''\rangle$ then either $\exists v.\ e' = \mathtt{Val}\ v \wedge P,hp\ s' \vdash v :\leq T$ or $\exists a.\ e' = \mathtt{Throw}\ a \wedge a \in dom\ (hp\ s')$.

Proof by many step subject reduction, preservation of $\mathcal{D}$, and progress.

What does type safety really tell us?

- If you trust the semantics:
  the type system is correct w.r.t. the semantics
- If you trust the type system:
  the semantics is complete w.r.t. the type system, no reduction rules are missing.
- If you don’t trust either:
  at least type system and semantics fit together

Completeness of the type system

Is the type system complete?

Is *e* well-typed if reduction of *e* does not get stuck?

No!

It is undecidable if reduction gets stuck.
Same argument as for undecidability of definite assignment.

Example: `if` (*true*) *true* `else` `Val` (*Intg 42*)

Decidable type systems
for interesting (= undecidable) properties
are necessarily incomplete.