Compiler Construction

Types and Type Soundness

UNI FREIBURG

University of Freiburg

Matthias Keil, Peter Thiemann

University of Freiburg

21. November 2016

Outline

Excursion to a World Without Types: Scripting Languages

UNI FREIBURG

Lightweight programming languages evolved from command languages

Lightweight data structures hashmap (object), strings

Lightweight syntax familiar, no semicolon, (often not well specified), ...

Lightweight typing dynamic, weak, duck typing

Lightweight metaprogramming

Lightweight implementation interpreted, few tools

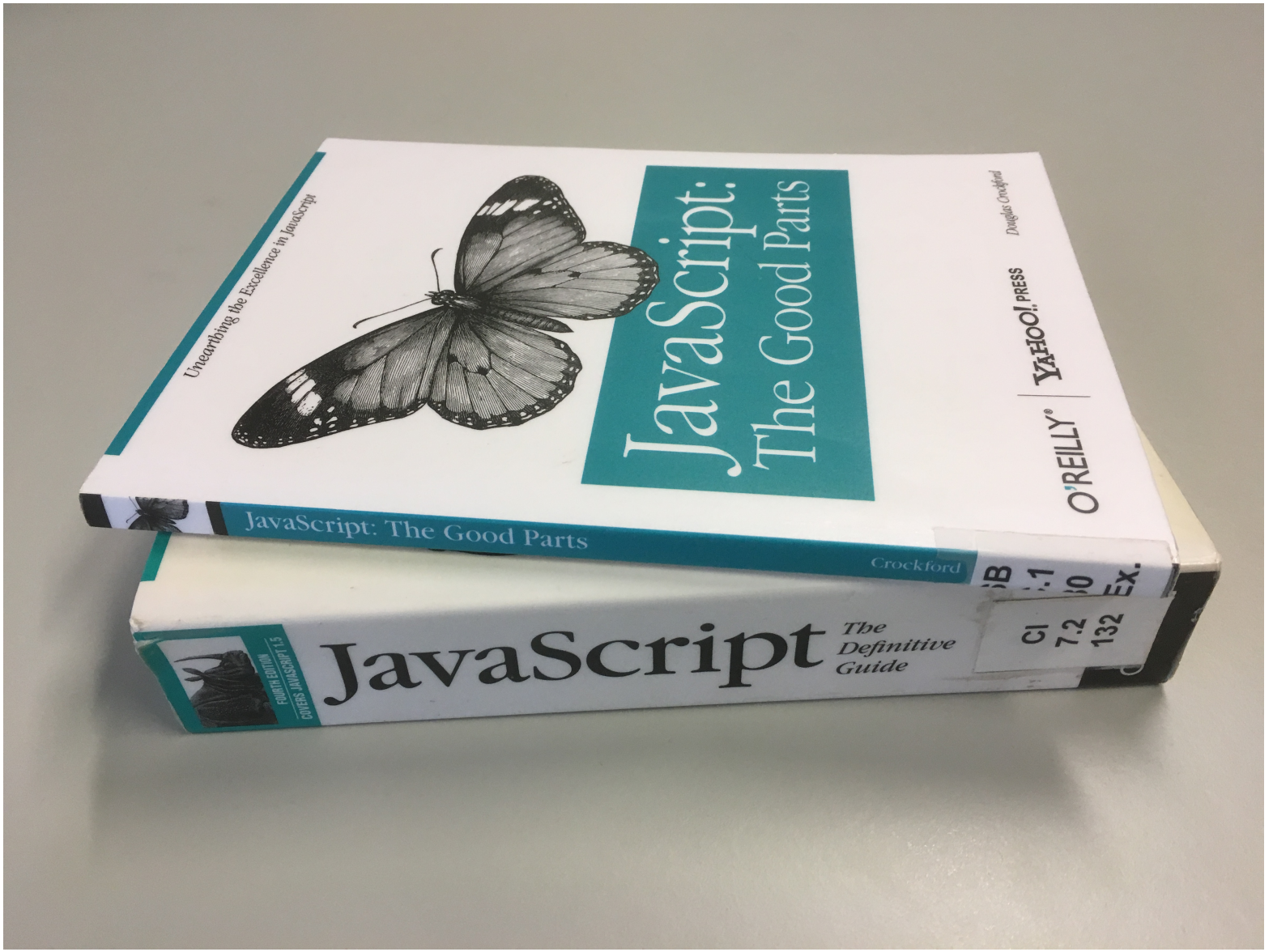
JavaScript, a Typical Scripting Language

- Initially developed by Brendan Eich of Netscape Corp.
- Standardized as ECMAScript (ECMA-262 Edition 6)
- Application areas (scripting targets)
  - client-side web scripting (dynamic HTML, SVG, XUL, GWT)
  - server-side scripting (Whitebeam, Cocoon, iPlanet, nodejs)
  - animation scripting (diablo, dim3, k3d)
  - cloud scripting (Google Apps Script)

- Java-style syntax
- Object-based imperative language
  - no classes, but prototype concept
  - objects are hashtables
- First-class functions
  - a functional language
- Weak, dynamic type system
- **Slogan** *Any type can be converted to any other reasonable type*

JavaScript, The Good and the Bad Parts

University of Freiburg



Problems with JavaScript

UNI FREIBURG

Symptomatic for other scripting languages

- No module system
  - No namespace management
  - No interface descriptions
- No static type system
- No application specific datatypes
  primitive datatypes, strings, hashtables
- Type conversions are sometimes surprising
  "A scripting language should never throw an exception [the script should just continue]" (Rob Pike, Google)
- Few development tools (debugger)

$\Rightarrow$ Conceived for small applications, but ...

Specific Problems with JavaScript

- Most popular applications
  - client-side scripting
  - AJAX
- Dynamic modification of page content via DOM interface
  - DOM = document object model
  - W3C standard interface for accessing and modifying XML
  - Mainly used in web browers

Specific Problems with JavaScript

- Most popular applications
  - client-side scripting
  - AJAX
- Dynamic modification of page content via DOM interface
  - DOM = document object model
  - W3C standard interface for accessing and modifying XML
  - Mainly used in web browers
- Incompatible DOM implementations in Web browsers

$\Rightarrow$ programming recipes instead of techniques

$\Rightarrow$ platform independent libraries like jQuery

- Security holes via dynamically loaded code or XSS

$\Rightarrow$ sandboxing, analysis

Can You Write Reliable Programs in JavaScript?

- Struggle with the lack of *e.g.* a module system
  - Ad-hoc structuring of large programs
  - Naming conventions
  - Working in a team
- Work around DOM incompatibilities
  - Use existing JavaScript frameworks (widgets, networking)
  - Frameworks are also incompatible
- Wonder about unexpected results

Rule 1:

JavaScript is object-based. An object is a hash table that maps named properties to values.

Rule 1:

JavaScript is object-based. An object is a hash table that maps named properties to values.

Rule 2:

Every value has a type. For most reasonable combinations, values can be converted from one type to another type.

Rule 1:

JavaScript is object-based. An object is a hash table that maps named properties to values.

Rule 2:

Every value has a type. For most reasonable combinations, values can be converted from one type to another type.

Rule 3:

Types include `null`, `boolean`, `number`, `string`, `object`, and `function`.

UNI FREIBURG

Rule 1:

JavaScript is object-based. An object is a hash table that maps named properties to values.

Rule 2:

Every value has a type. For most reasonable combinations, values can be converted from one type to another type.

Rule 3:

Types include `null`, `boolean`, `number`, `string`, `object`, and `function`.

Rule 4:

'Undefined' is a value (and a type).

Let's define an object `obj`: `var obj = { x: 1 }`
What are the values/outputs of

- `obj.x`
- `obj.y`
- `print(obj.y)`
- `obj.y.z`

- `var obj = {x:1}`
- `obj.x`
  → `1`
- `obj.y`
  →
- `print(obj.y)`
  → `undefined`
- `obj.y.z`
  → *"<stdin>", line 12: uncaught JavaScript exception: ConversionError: The undefined value has no properties. (<stdin>; line 12)*

Rule 5:

An object is really a dynamic mapping from strings to values.

- `var x = 'x'`
- `obj[x]`
  $\rightarrow$ `1`
- `obj.undefined = 'gotcha'`
  $\rightarrow$ `gotcha`
- `obj[obj.y]`

What is the effect/result of the last expression?

Rule 5:

An object is really a dynamic mapping from strings to values.

- `var x = 'x'`
- `obj[x]`
  $\rightarrow$ `1`
- `obj.undefined = 'gotcha'`
  $\rightarrow$ `gotcha`
- `obj[obj.y]`

What is the effect/result of the last expression?

- $\rightarrow$ `obj[obj.y]== obj[undefined]==`
  `obj['undefined']==obj.undefined=='gotcha'`

Recall Rule 2:

Every value has a type. For most reasonable combinations, values can be converted from one type to another type.

- `var a = 17`
- `a.x = 42`
  $\rightarrow$ `42`
- `a.x`
  $\rightarrow$

What is the effect/result of the last expression?

Wrapper objects for numbers

- `m = new Number (17); n = new Number (4)`
- `m+n`
  $\rightarrow$ 21

Wrapper objects for numbers

- `m = new Number (17); n = new Number (4)`
- `m+n`
  $\rightarrow$ 21

Wrapper objects for booleans

- `flag = new Boolean(false);`
- `result = flag ? true : false;`

What is the value of `result`?

Rule 6:

Functions are first-class, but behave differently when used as methods or as constructors.

- `function f () { return this.x }`
- `f()`
  → `x`
- `obj.f = f`
  → `function f() { return this.x; }`
- `obj.f()`
  → `1`
- `new f()`
  → `[object Object]`

Distinguishing Absence and Undefinedness I

- `obju = { u : {}.xx }`
  → `[object Object]`
- `objv = { v : {}.xx }`
  → `[object Object]`
- `print(obju.u)`
  → `undefined`
- `print(objv.u)`
  → `undefined`

Rule 7:

The `with` construct puts its argument object on top of the current environment stack.

- `u = 'defined'`
  $\rightarrow$ `defined`
- `with (obju) print(u)`
  $\rightarrow$ `undefined`
- `with (objv) print(u)`
  $\rightarrow$ `defined`

Distinguishing Absence and Undefinedness III

Rule 8:

The `for` construct has an `in` operator to range over all defined indexes.

- `for (i in obju) print(i)`
  → `u`
- `for (i in objv) print(i)`
  → `v`
- `delete objv.v`
  → `true`
- `for (i in objv) print(i)`
  →
- `delete objv.v`
  → `true`

UNI FREIBURG

- Common errors such as
  - using non-objects as objects, *e.g.* using numbers as functions
  - invoking non-existing methods
  - accessing non-existing fields
  - surprising conversions

  can all be caught by a

  Static Type System

- and much more.

- Large software systems: many people involved
  - project manager, designer, programmer, tester, ...
- Essential: divide into components with clear defined interfaces and specifications
  - How to divide the problem?
  - How to divide the work?
  - How to divide the tests?
- Problems
  - Are suitable libraries available?
  - Do the components match each other?
  - Do the components fulfill their specification?

- Programming language/environment has to ensure:
  - each component implements its interfaces
  - the implementation fulfills the specification
  - each component is used correctly
- Main problem: meet the interfaces and specifications
  - Minimal interface: *management of names*
    Which operations does the component offer?
  - Minimal specification: *types*
    Which types do the arguments and the result of the operations have?
  - See interfaces in Java

- Which kind of security do types provide?
- Which kind of errors can be detected by using types?
- How do we provide type safety?
- How can we formalize type safety?

Grammar for a subset of Java expressions

x	::=	$\ldots$	variables
n	::=	$0 \mid 1 \mid \ldots$	numbers
b	::=	`true` $\mid$ `false`	truth values
e	::=	$x \mid n \mid b \mid e{+}e \mid !e$	expressions

Correct and Incorrect Expressions

Type correct expressions

```
1 boolean flag;
2 int num;
3 0
4 true
5 17+4
6 !flag
```

Expressions with type errors

```
1 !num
2 flag+1
3 17+(!false)
4 !(2+3)
```

- For each kind of expression a typing rule defines
  - if an expression is type correct and
  - how to obtain the result type of the expression from the types of the subexpressions.
- Five kinds of expressions
  - Constant numbers have type `int`.
  - Truth values have type `boolean`.
  - The expression $e_1 + e_2$ has type `int`, if e_1 and e_2 have type `int`.
  - The expression !e has type `boolean`, if e has type `boolean`.
  - A variable x has the type, with which it was declared.

The Language of Types

$$\tau \quad ::= \quad \texttt{int} \mid \texttt{boolean} \qquad \text{types}$$

Typing judgment: expression e has type τ

$$\vdash e : \tau$$

Formalization of "Typing Rules"

- A typing judgment is *valid*, if it is derivable according to the *typing rules*.
- To infer a valid typing judgment J we use a *deduction system*.
- A deduction system consists of a set of typing judgments and a set of typing rules.
- A typing rule (*inference rule*) is a pair $(J_1 \ldots J_n, J_0)$ which consists of a list of judgments (*assumptions*, $J_1 \ldots J_n$) and a judgment (*conclusion*, J_0) that is written as

$$\frac{J_1 \ldots J_n}{J_0}$$

- If $n = 0$, a rule (ε, J_0) is an *axiom*.

Example: Typing Rules for JAUS

- A number n has type `int`.

$$\frac{}{\vdash n : \texttt{int}}\ (\text{INT})$$

- A truth value has type `boolean`.

$$\frac{}{\vdash b : \texttt{boolean}}\ (\text{BOOL})$$

Example: Typing Rules for JAUS (cont'd)

- An expression $e_1 + e_2$ has type `int` if e_1 and e_2 have type `int`.

$$\frac{\vdash e_1 : \texttt{int} \quad \vdash e_2 : \texttt{int}}{\vdash e_1 + e_2 : \texttt{int}} \text{(ADD)}$$

- An expression $!e$ has type `boolean`, if e has type `boolean`.

$$\frac{\vdash e : \texttt{boolean}}{\vdash !e : \texttt{boolean}} \text{(NOT)}$$

- A judgment J is *valid* if a derivation tree for J exists.
- *Definition*: A *derivation tree for the judgment* J is either
  1. J, if J is an instance of an axiom, or
  2. $\dfrac{\mathcal{J}_1 \ldots \mathcal{J}_n}{J}$, if $\dfrac{J_1 \ldots J_n}{J}$ is an instance of a rule and each $\mathcal{J}_k$ is a derivation tree for J_k.

Example: Derivation Trees

(INT)

- $\vdash 0 : \texttt{int}$ is a derivation tree for judgment $\vdash 0 : \texttt{int}$.

(BOOL)

- $\vdash \texttt{true} : \texttt{boolean}$ is a derivation tree for $\vdash \texttt{true} : \texttt{boolean}$.
- The judgment $\vdash 17 + 4 : \texttt{int}$ holds, because of the derivation tree

$$\dfrac{\begin{array}{cc}(\text{ADD}) & \\ (\text{INT}) & (\text{INT}) \\ \vdash 17 : \texttt{int} & \vdash 4 : \texttt{int}\end{array}}{\vdash 17 + 4 : \texttt{int}}$$

- Programs declare variables
- Programs use variables according to their declaration
- Declarations are collected in a *type environment*.

$$\Gamma \quad ::= \quad \emptyset \mid \Gamma, x : \tau \qquad \text{type environment}$$

- An open typing judgment contains a type environment: The expression e has the type t in the type environment Γ.

$$\Gamma \vdash e : \tau$$

- Typing rule for variables:
  A variable has the type, with which it is declared.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \text{(VAR)}$$

Extension of the Remaining Typing Rules

- The typing rules propagate the typing environment.

$$\frac{}{\Gamma \vdash n : \texttt{int}}\ (\text{INT})$$

$$\frac{}{\Gamma \vdash b : \texttt{int}}\ (\text{BOOL})$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}\ (\text{ADD})$$

$$\frac{\Gamma \vdash !e : \texttt{boolean}}{\Gamma \vdash e : \texttt{boolean}}\ (\text{NOT})$$

Example: Derivation with Variable

The declaration `boolean flag;` matches the type assumption

$$\Gamma = \emptyset, \texttt{flag} : \texttt{boolean}$$

Hence the derivation

$$\frac{\dfrac{\texttt{flag} : \texttt{boolean} \in \Gamma}{\Gamma \vdash \texttt{flag} : \texttt{boolean}}}{\Gamma \vdash \texttt{!}\ \texttt{flag} : \texttt{boolean}}$$

- Formal system for
  - syntax of expressions and types (CFG, BNF)
  - typing judgments
  - validity of typing judgments
- Open questions
  - How to evaluate expressions?
  - Connection between evaluation and typing judgments

Evaluation of Expressions

- Define a binary *reduction relation* $e \longrightarrow e'$ over expressions
- Expression e *reduces in one step to* e' (Notation: $e \longrightarrow e'$) if one computational step leads from e to e'.
- Example:
  - $5{+}2 \longrightarrow 7$
  - $(5{+}2){+}14 \longrightarrow 7{+}14$

Result of Computations

- A value v is a number or a truth value.
- An expression can reach a value after many steps:
  - 0 steps: 0
  - 1 step: $5+2 \longrightarrow 7$
  - 2 steps: $(5+2)+14 \longrightarrow 7+14 \longrightarrow 21$
- but
  - $!4711$
  - $1+\texttt{false}$
  - $(1+2)+\texttt{false} \longrightarrow 3+\texttt{false}$
- These expressions cannot perform a reduction step. They correspond to run-time errors.
- *Observation: these errors are type errors!*

- A value is a number or a truth value.

$$v ::= n \mid b \qquad \text{values}$$

- One reduction step
  - If the two operands are numbers, we can add the two numbers to obtain a number as result.

$$\frac{}{\lceil n_1 \rceil + \lceil n_2 \rceil \longrightarrow \lceil n_1 + n_2 \rceil} \text{(B-ADD)}$$

    $\lceil n \rceil$ stands for the syntactic representation of the number n.
  - If the operand of a negation is a truth value, the negation can be performed.

$$\frac{}{\texttt{!true} \longrightarrow \texttt{false}} \text{(B-TRUE)} \qquad \frac{}{\texttt{!false} \longrightarrow \texttt{true}} \text{(B-FALSE)}$$

Formalization: Nested Expressions

What happens if the operands of operations are not values?
Evaluate the subexpressions first.

- Negation

$$\frac{e \longrightarrow e'}{!e \longrightarrow !e'} \text{(B-NEG)}$$

- Addition, first operand

$$\frac{e_1 \longrightarrow e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2} \text{(B-ADD-L)}$$

- Addition, second operand (only evaluate the second, if the first is a value)

$$\frac{e \longrightarrow e'}{v + e \longrightarrow v + e'} \text{(B-ADD-R)}$$

- An expression that contains variables cannot be evaluated with the reduction steps.
- Eliminate variables with *substitution*, *i.e.*, replace each variable with a value. Then reduction can proceed.
- Applying a substitution $[v_1/x_1, \ldots v_n/x_n]$ to an expression e, written as

$$e[v_1/x_1, \ldots v_n/x_n]$$

  changes in e each occurrence of x_i to the corresponding value v_i.
- Example:
  - $(\texttt{!flag})[\texttt{false}/\texttt{flag}] \equiv \texttt{!false}$
  - $(\texttt{m+n})[25/\texttt{m}, 17/\texttt{n}] \equiv 25+17$

Type Correctness Informally

- Type correctness: If there exists a type for an expression e, then e evaluates to a value in a finite number of steps.
- In particular, no run-time error happens.
- For the language JAUS the converse also holds (this is not correct in general, like in full Java).
- Prove in two steps (after Wright and Felleisen)
  Assume e has a type, then it holds:

  Progress: Either e is a value or there exists a reduction step for e.

  Preservation: If $e \longrightarrow e'$, then e' and e have the same type.

Theorem: Type Soundness of JAUS

- If $\vdash e : \tau$, then there exists a value v with $\vdash v : \tau$ and reduction steps

$$e_0 \longrightarrow e_1, e_1 \longrightarrow e_2, \ldots, e_{n-1} \longrightarrow e_n$$

  with $e \equiv e_0$ and $e_n \equiv v$.
- If e contains variables, then we have to substitute them with suitable values (choose values with same types as the variables).

- 1995 public presentation of Java
- Obtained importance very quickly
- Questions
  - Type safety?
  - Semantics of Java?
- 1997/98 resolved
  - Drossopoulou/Eisenbach
  - Flatt/Krishnamurthi/Felleisen
  - Igarashi/Pierce/Wadler (Featherweight Java, FJ)

UNI FREIBURG

- Construction of a formal model:
  consideration of completeness and compactness
- FJ: minimal model (compactness)
- complete definition: one page
- ambition:
  - the most important language features
  - short proof of type soundness
  - FJ $\subseteq$ Java

- class definition
- object creation **new**
- method call (*dynamic dispatch*), recursion with **this**
- field access
- type cast
- method *override*
- subtypes

Omitted

- assignment
- interfaces
- *overloading*
- **super**-calls
- **null**-references
- primitive types
- abstract methods
- inner classes
- shadowing of fields of super classes
- access control (**private**, **public**, **protected**)
- *exceptions*
- concurrency
- reflection, generics, variable argument lists

Example Programs

```
1 class A extends Object { A() { super(); } }
2
3 class B extends Object { B() { super(); } }
4
5 class Pair extends Object {
6   Object fst;
7   Object snd;
8   // Constructor
9   Pair (Object fst, Object snd) {
10    super(); this.fst = fst; this.snd = snd;
11  }
12  // Method definition
13  Pair setfst (Object newfst) {
14    return new Pair (newfst, this.snd);
15  }
16 }
```

- Class definition: always define super class
- Constructors:
  - one per class, always defined
  - arguments correspond to fields
  - always the same form:
    super-call, then copy the arguments into the fields
- method body: always in the form **return**. . .

Guarantees of Java's Type System

UNI FREIBURG

If a Java program is type correct, then

- all field accesses refer to existing fields
- all method calls refer to existing methods,
- but failing type casts are possible.

Formal Definition

CL	$::=$		class definition
		class C **extends** $D\,\{C_1\ f_1; \ldots\ K\ M_1 \ldots\}$	
K	$::=$		constructor definition
		$C(C_1\ f_1, \ldots)\ \{$**super**$(g_1, \ldots);$ **this**$.f_1 = f_1; \ldots\}$	
M	$::=$		method definition
		$C\ m(C_1\ x_1, \ldots)\ \{$**return** $t;\}$	
t	$::=$		expressions
		x	variable
		$t.f$	field access
		$t.m(t_1, \ldots)$	method call
		new $C(t_1, \ldots)$	object creation
		$(C)\ t$	type cast
v	$::=$		values
		new $C(v_1, \ldots)$	object creation

- `this`
  - special variable, do not use it as field name or parameter
  - implicit bound in each method body
- sequences of field names, parameter names and method names include no repetition
- **class** C **extends** $D\,\{C_1\ f_1; \ldots\ K\ M_1 \ldots\}$
  - defines class C as subclass of D
  - fields $f_1 \ldots$ with types $C_1 \ldots$
  - constructor K
  - methods $M_1 \ldots$
  - fields from D will be added to C, shadowing is not supported

- $C(D_1\ g_1, \ldots, C_1\ f_1, \ldots)\ \{\textbf{super}(g_1, \ldots); \textbf{this}.f_1 = f_1; \ldots\}$
  - define the constructor of class C
  - fully specified by the fields of C and the fields of the super classes.
  - number of parameters is equal to number of fields in C and all its super classes.
  - body start with $\textbf{super}(g_1, \ldots)$, where $g_1, \ldots$ corresponds to the fields of the super classes
- $D\ m(C_1\ x_1, \ldots)\ \{\textbf{return}\ t;\}$
  - defines method m
  - result type D
  - parameter $x_1 \ldots$ with types $C_1 \ldots$
  - body is a **return** statement

Class Table

- The *class table* CT is a map from class names to class definitions
  - $\Rightarrow$ each class has exactly one definition
    - the CT is global, it corresponds to the program
    - "arbitrary but fixed"
- Each class except Object has a superclass
  - Object is not part of CT
  - Object has no fields
  - Object has no methods ($\neq$ Java)
- The class table defines a subtype relation $C <: D$ over class names: the reflexive and transitive closure of subclass definitions.

$$\text{REFL}\quad C <: C$$

$$\text{TRANS}\quad \frac{C <: D \qquad D <: E}{C <: E}$$

$$\text{EXT}\quad \frac{CT(C) = \textbf{class } C \textbf{ extends } D \ldots}{C <: D}$$

$$\text{REFL}\quad C <: C$$

$$\text{TRANS}\quad \frac{C <: D \qquad D <: E}{C <: E}$$

$$\text{EXT}\quad \frac{CT(C) = \textbf{class } C \textbf{ extends } D \ldots}{C <: D}$$

Java: *Assignment compatibility*

If $C <: D$, then

- a C-value can be assigned to a D-variable and
- a C-value can be passed as a D-parameter.

UNI FREIBURG

$$\text{IMPL}\quad \frac{CT(C) = \textbf{class } C \textbf{ implements } I \ldots}{C <: I}$$

$$\text{IEXT}\quad \frac{CT(I) = \textbf{interface } I \textbf{ extends } J \ldots}{I <: J}$$

Consistency of CT

1. $CT(C) = \textbf{class}\ C \ldots$ for all $C \in dom(CT)$
2. Object $\notin dom(CT)$
3. For each class name C mentioned in CT:
   $C \in dom(CT) \cup \{\text{Object}\}$
4. The relation <: is antisymmetric (no cycles)

Example: Classes Do Refer to Each Other


```
1 class Author extends Object {
2   String name; Book bk;
3
4   Author (String name, Book bk) {
5     super();
6     this.name = name;
7     this.bk = bk;
8   }
9 }
10 class Book extends Object {
11   String title; Author ath;
12
13   Book (String title, Author ath) {
14     super();
15     this.title = title;
16     this.ath = ath;
17   }
18 }
```

$$\mathit{fields}(\text{Object}) = \bullet$$

$$\frac{CT(C) = \textbf{class}\ C\ \textbf{extends}\ D\ \{C_1\ f_1; \ldots\ K\ M_1 \ldots\} \qquad \mathit{fields}(D) = D_1\ g_1, \ldots}{\mathit{fields}(C) = D_1\ g_1, \ldots, C_1\ f_1, \ldots}$$

- $\bullet$ — empty list
- *fields*(Author) = String name; Book bk;
- Usage: evaluation steps, typing rules

$$\frac{CT(C) = \textbf{class } C \textbf{ extends } D \,\{C_1\ f_1; \ldots\ K\ M_1 \ldots\} \qquad M_j = E\ m(E_1\ x_1, \ldots)\ \{\textbf{return } t;\}}{mtype(m, C) = (E_1, \ldots) \to E}$$

$$\frac{CT(C) = \textbf{class } C \textbf{ extends } D \,\{C_1\ f_1; \ldots\ K\ M_1 \ldots\} \qquad (\forall j)\ M_j \neq F\ m(F_1\ x_1, \ldots)\ \{\textbf{return } t;\} \qquad mtype(m, D) = (E_1, \ldots) \to E}{mtype(m, C) = (E_1, \ldots) \to E}$$

- Usage: typing rules

$$\frac{CT(C) = \textbf{class } C \textbf{ extends } D \, \{C_1 \; f_1; \ldots \; K \; M_1 \ldots\} \quad M_j = E \; m(E_1 \; x_1, \ldots) \, \{\textbf{return } t;\}}{mbody(m, C) = (x_1 \ldots, t)}$$

$$\frac{CT(C) = \textbf{class } C \textbf{ extends } D \, \{C_1 \; f_1; \ldots \; K \; M_1 \ldots\} \quad (\forall j) \; M_j \neq F \; m(F_1 \; x_1, \ldots) \, \{\textbf{return } t;\} \quad mbody(m, D) = (y_1 \ldots, u)}{mbody(m, C) = (y_1 \ldots, u)}$$

- Usage: evaluation steps

$$override(m, \text{Object}, (E_1 \dots) \to E)$$

$$\frac{CT(C) = \textbf{class}\ C\ \textbf{extends}\ D\ \{C_1\ f_1; \dots\ K\ M_1 \dots\}
\quad M_j = E\ m(E_1\ x_1, \dots)\ \{\textbf{return}\ t;\}}{override(m, C, (E_1 \dots) \to E)}$$

$$\frac{CT(C) = \textbf{class}\ C\ \textbf{extends}\ D\ \{C_1\ f_1; \dots\ K\ M_1 \dots\}
\quad (\forall j)\ M_j \neq F\ m(F_1\ x_1, \dots)\ \{\textbf{return}\ t;\}
\quad override(m, D, (E_1, \dots) \to E)}{override(m, C, (E_1, \dots) \to E)}$$

- Usage: typing rules

Operational Semantics (definition of the evaluation steps)

Direct Evaluation Steps

- Evaluation: relation $t \longrightarrow t'$ for one evaluation step

$$\textsc{E-ProjNew}\quad \frac{\mathit{fields}(C) = C_1\ f_1, \ldots}{(\textbf{new}\ C(v_1, \ldots)).f_i \longrightarrow v_i}$$

$$\textsc{E-InvkNew}\quad \frac{\mathit{mbody}(m, C) = (x_1 \ldots, t)}{\begin{array}{c}(\textbf{new}\ C(v_1, \ldots)).m(u_1, \ldots) \\ \longrightarrow t[\textbf{new}\ C(v_1, \ldots)/\mathsf{this}, u_1, \ldots/x_1, \ldots]\end{array}}$$

$$\textsc{E-CastNew}\quad \frac{C <: D}{(D)(\textbf{new}\ C(v_1, \ldots)) \longrightarrow \textbf{new}\ C(v_1, \ldots)}$$

$$\textsc{E-Field}\ \frac{t \longrightarrow t'}{t.f \longrightarrow t'.f}$$

$$\textsc{E-Invk-Recv}\ \frac{t \longrightarrow t'}{t.m(t_1, \dots) \longrightarrow t'.m(t_1, \dots)}$$

$$\textsc{E-Invk-Arg}\ \frac{t_i \longrightarrow t_i'}{v.m(v_1, \dots, t_i, \dots) \longrightarrow v.m(v_1, \dots, t_i', \dots)}$$

Evaluation Steps in Context (cont'd)

E-New-Arg

$$\frac{t_i \longrightarrow t_i'}{\textbf{new}\ C(v_1, \ldots, t_i, \ldots) \longrightarrow \textbf{new}\ C(v_1, \ldots, t_i', \ldots)}$$

E-Cast

$$\frac{t \longrightarrow t'}{(C)t \longrightarrow (C)t'}$$

Typing Rules

Overview of typing judgments

- $C <: D$
  C is subtype of D
- $A \vdash t : C$
  Under type assumption A, the expression t has type C.
- $F\ m(C_1\ x_1, \ldots)\ \{\textbf{return}\ t;\}$ OK in C
  Method declaration is accepted in class C.
- **class** C **extends** $D\ \{C_1\ f_1; \ldots\ K\ M_1 \ldots\}$ OK
  Class declaration is accepted
- Type assumptions defined by

$$A ::= \emptyset \mid A, x : C$$

Accepted Class Declaration

$$\frac{K = C(D_1\ g_1, \ldots, C_1\ f_1, \ldots)\ \{\textbf{super}(g_1, \ldots); \textbf{this}.f_1 = f_1; \ldots\} \quad \mathit{fields}(D) = D_1\ g_1 \ldots \quad (\forall j)\ M_j \text{ OK in } C}{\textbf{class}\ C\ \textbf{extends}\ D\ \{C_1\ f_1; \ldots\ K\ M_1 \ldots\}}$$

Accepted Method Declaration

$$\frac{\begin{array}{c} x_1 : C_1, \ldots, \mathsf{this} : C \vdash t : E \\ E <: F \qquad CT(C) = \textbf{class } C \textbf{ extends } D \ldots \\ \mathit{override}(m, D, (C_1, \ldots) \to F) \end{array}}{F\ m(C_1\ x_1, \ldots)\ \{\textbf{return } t;\}\ \mathsf{OK\ in}\ C}$$

Expression Has Type

$$\frac{x : C \in A}{A \vdash x : C}\ \textsc{T-Var}$$

$$\frac{A \vdash t : C \qquad \mathit{fields}(C) = C_1\ f_1, \ldots}{A \vdash t.f_i : C_i}\ \textsc{T-Field}$$

Expression Has Type (cont'd)

F-Invk

$$\frac{A \vdash t : C \qquad (\forall i)\ A \vdash t_i : C_i \qquad (\forall i)\ C_i <: D_i \qquad mtype(m, C) = (D_1, \dots) \to D}{A \vdash t.m(t_1, \dots) : D}$$

F-New

$$\frac{(\forall i)\ A \vdash t_i : C_i \qquad (\forall i)\ C_i <: D_i \qquad fields(C) = D_1\ f_1, \dots}{A \vdash \textbf{new}\ C(t_1, \dots) : C}$$

Type Rules for Type Casts

$$\textsc{T-UCast}\ \frac{A \vdash t : D \qquad D <: C}{A \vdash (C)t : C}$$

$$\textsc{T-DCast}\ \frac{A \vdash t : D \qquad C <: D \qquad C \neq D}{A \vdash (C)t : C}$$

- "Preservation" and "Progress" yields type safety
- "Preservation":
  If $A \vdash t : C$ and $t \longrightarrow t'$, then $A \vdash t' : C'$ with $C' <: C$.
- "Progress": (short version)
  If $A \vdash t : C$, then $t \longrightarrow t'$, for some t', or $t \equiv v$ is a value, or t contains a subexpression e'

$$e' \equiv (C)(\textbf{new } D(v_1, \ldots))$$

with $D \not<: C$.

$\Rightarrow$
- All method calls and field accesses evaluate without errors.
- Type casts can fail.

- Consider the expression `(A) ((Object)new B())`
- It holds that $\emptyset \vdash$ `(A) ((Object)new B())`: A
- It holds that `(A) ((Object)new B())` $\longrightarrow$ `(A) (new B())`
- But `(A) (new B())` has no type!

- Consider the expression `(A) ((Object)new B())`
- It holds that $\emptyset \vdash$ `(A) ((Object)new B())`: A
- It holds that `(A) ((Object)new B())` $\longrightarrow$ `(A) (new B())`
- But `(A) (new B())` has no type!
- Workaround: add additional rule for this case "*stupid cast*" —subsequent evaluation step fails

$$\text{T-SCast}\ \frac{A \vdash t : D \qquad C \not<: D \qquad D \not<: C}{A \vdash (C)t : C}$$

- We can prove preservation with this rule.

If $A \vdash t : C$, then one of the following cases applies:

1. t does not terminate
   i.e., there exists an infinite sequence of evaluation steps
   $$t = t_0 \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \ldots$$
2. t evaluates to a value v after a finite number of evaluation steps
   i.e., there exists a finite sequence of evaluation steps
   $$t = t_0 \longrightarrow t_1 \longrightarrow \ldots \longrightarrow t_n = v$$
3. t gets stuck at a failing cast
   i.e., there exists a finite sequence of evaluation steps
   $$t = t_0 \longrightarrow t_1 \longrightarrow \ldots \longrightarrow t_n$$
   where t_n contains a subterm $(C)(\textbf{new}\ D(v_1, \ldots))$ such that $D \not<: C$.