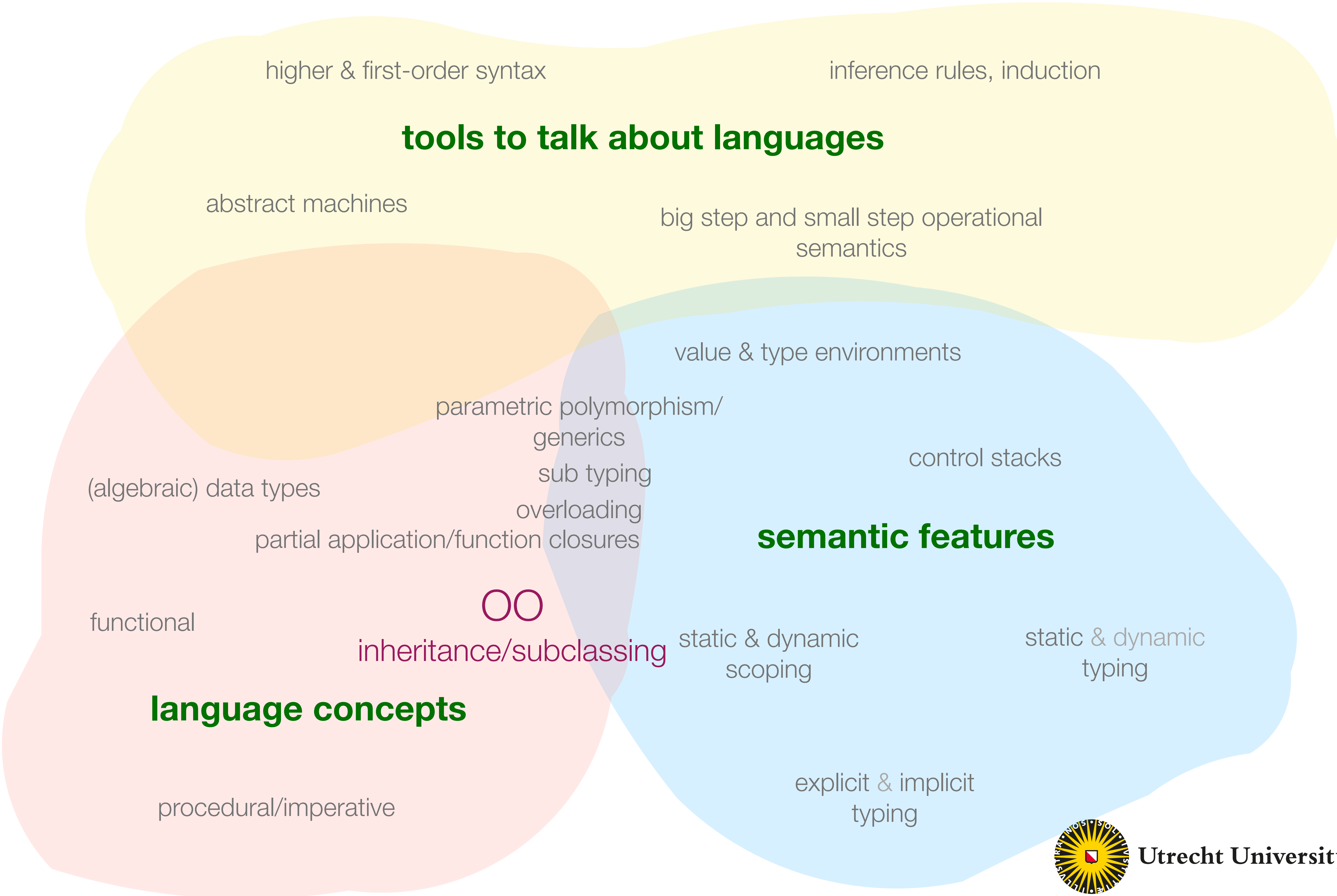
Utrecht University

Concepts of Programming Language Design

Featherweight Java

Overview



Utrecht University

Object oriented languages

- What are the characteristics of an object oriented language?
  - objects bundle data and behaviour
    - the object knows what to do
  - inheritance
    - subclass relationship in the type system
  - method overloading as a form of polymorphism
  - abstraction (or encapsulation):
    - hiding implementation details from the user, interaction through restricted interface

Object oriented languages

- We will look a these language features today (apart from abstraction) using Featherweight Java, a simple object oriented language
- we only look at the features which didn't occur in other languages
  - e.g., no destructive updates, arithmetic expressions, loops,…

Featherweight Java

- Minimal OO language

$$
\begin{array}{lcl}
\mathit{prgm} & ::= & \mathit{cdecs}\ \mathit{expr} \\
\mathit{cdecls} & ::= & \mathit{cdecl}\ \ \mathit{cdecls} \\
\mathit{cdecl} & ::= & \texttt{class}\ \mathit{classN}\ \texttt{extends}\ \mathit{classN}\ \{\underline{\mathit{classN}\ \ \mathit{fieldN}}\ ;\ \mathit{cons}\ \underline{\mathit{method}}\} \\
\mathit{cons} & ::= & \mathit{classN}\ (\underline{\mathit{classN}\ \ \mathit{varN}}\ )\{\texttt{super}\ (\underline{\mathit{varN}});\ \ \underline{\texttt{this}.\mathit{fieldN}=\mathit{varN}};\} \\
\mathit{method} & ::= & \mathit{classN}\ \mathit{methodN}\ (\ \underline{\mathit{classN}\ \ \mathit{varN}})\ \{\texttt{return}\ e\ \}; \\
\tau & ::= & \mathit{classN} \\
e & ::= & \mathit{varN}\ \mid\ e.\mathit{fieldN}\ \mid\ e.\mathit{methodN}(e)\ \mid \texttt{new}\ \mathit{classN}\ (\underline{e})\ \mid\ (\mathit{classN})\ e
\end{array}
$$

- TinyC like expressions and statements could be added to make it a proper language
  - most would not add anything of interest for our purposes
  - destructive updates would be interesting (but discussed in the context of ref types):
    - does an assignment copy the object or just the reference?

Featherweight Java

- Class expressions

$$\texttt{class}\ \mathit{classN}\ \texttt{extends}\ \mathit{classN}\ \{\underline{\mathit{classN}\ \mathit{fieldN}}\ ;\ \mathit{cons}\ \underline{\mathit{method}}\}$$

```
class C extends C'
  {C1 f1;
   C2 f2;
   …
   k
   d1
   d2
   …
  }
```

declares a class

- C to be a subclass of C'
- with additional fields $C_i\ f_i$
- a single constructor k
- methods d_i

Featherweight Java

$$\textit{classN}\ (\underline{\textit{classN}\ \ \textit{varN}})\{\texttt{super}\ (\underline{\textit{varN}});\ \underline{\texttt{this.}\textit{fieldN}\texttt{=}\textit{varN}};\}$$

- Constructor expressions

```
C (C1' x1';…;C1 x1;…){
   super (x1',...)
   this.f1 = x1;
   this.f2= x2;
   …
  }
```

declares a constructor for a class

- with arguments $C'_1\ x'_1$ corresponding to a superclass
- with arguments $C_1\ x_1$ corresponding to the new fields of the subclass
- x'_1 are initialised via the superclass
- $\texttt{this.}f_i = x_i$ fields initialised in the subclass

Featherweight Java

- Method expressions

$$\textit{classN methodN}\ (\ \underline{\textit{classN varN}})\ \{\texttt{return}\ e\};$$

```
C m (C1 x1 , C2 x2 , …) {return e;}
```

declares a method m

- which returns a value of class C
- with arguments x_i of class C_i
- and a body returning the value of expression e
- methods of the parent class can be overwritten (type has to stay the same)

Featherweight Java

- Field selection

```
e.f
```

select a field *f* from instance *e*

- Method invocation

```
e.m (e1, e2, ...)
```

invoke a method *m* of instance *e* with arguments e_1, e_2, ...

Featherweight Java

- Instance creation

```
new C (e1, e2, ...)
```

creates new instance of class *C* with arguments e_1, e_2, ...

- Casting

```
(C)e
```

casts a value *e* to class *C*

What should the semantics of a cast be?

- change the static type of the object to be *C*, but don't change the object
- this is only ok if the actual type of *e* is a subclass of *C*

Featherweight Java

- Types
  - the set of types is limited to the set of class names
  - in examples, we assume the presence of types like `int` and `bool`, but we will not discuss the semantics of these types

  there is a class `Object`

  - all other classes are subclasses of `Object`

  a special variable `this` referring to the instance itself

Subclasses and Subtypes

- Subclass/superclass relationship:
  - similar to the subtype/supertype relationship, but more restrictive
  - if c is a subclass of c', then c has at least as many fields and methods as c'
  - objects of class c can be coerced to c' by ignoring the additional fields
  - we write c `<:` c' to denote the subclass relationship

```
class Point extends Object {
  int x;
  int y;

  Point (int x, int y) {
    super ();
    this.x = x;
    this.y = y;
  }
}

class ColourPoint extends Point {
  Colour c;

  ColourPoint (int x, int y, Colour c) {
   super (x, y);
   this.c = c;
  }

  Colour getc () {return this.c;}
}
```

```
class Colour extends Object {
  int red;
  int green;
  int blue;

  Colour (int r, int g, int b) {
    super ();
    this.red   = r % 256;
    this.green = g % 256;
    this.blue  = b % 256;
  }
}
```

Static Semantics

- The static semantics is defined by the following judgements:

$c <: c$	subclass relationship
$\Gamma \vdash e : c$	expression typing
$m \ ok \ in \ c$	well formed method
$c \ ok$	well formed class
$T \ ok$	well formed class table
$fields\ (c) = \{c_1\ f_1,\ c_2\ f_2,\ \ldots\ \}$	field lookup
$type\ (m,\ c) = \underline{c} \rightarrow c$	method type

- A program consists of a class table T (sequence of class declarations) and an expression e.
- We use the class table T as implicit parameter in the following.
- We only look at some interesting aspects of the static semantics.

Static Semantics

- Similar to other languages, we need to check:
  - are variable names in scope?
  - are methods called with the correct arguments?
  - if fields are accessed, are the fields attributes of the class?
  - we need to keep track of the type (ie, class) of an expression
- What is new?
  - the static type and the dynamic type are not necessarily the same!

Static Semantics

- Fields
  - we define a judgement to determine the field names and the types of a given class:

$$\mathit{fields}\ (\mathit{className}) = \{\mathit{className}_1\ \mathit{fieldName}_1,\ \ldots\ \}$$

$$\frac{}{\mathit{fields}(\texttt{Object}) = \bullet}$$

$$\frac{T(c) = \texttt{class}\ c\ \texttt{extends}\ c'\ \{\underline{c\ f};\ldots\} \qquad \mathit{fields}(c') = \underline{\underline{c'\ f'}}}{\mathit{fields}(c) = \underline{c'\ f'},\ \underline{c\ f}}$$

Static semantics

- Typing judgment for expressions of the language

$$\Gamma \vdash e : c$$

  - Every variable has to be declared

$$\frac{x : c \in \Gamma}{\Gamma \vdash x : c}$$

$$\frac{\Gamma \vdash e_0 : c_0 \quad \mathit{fields}(c_0) = c_1\ f_1 \ldots}{\Gamma \vdash e_0 . f_i : c_i}$$

Static Semantics

- Method invocation
  - argument and result types of methods are stored in the class table
  - we need to define a function ***type***, which, given a method and class, returns the type of that method (we'll leave that for later)
  - methods of super types can be applied without cast

$e_1 : c_1,\ e_2 : c_2,\ \ldots$

$c_1 <: c'_1,\ c_2 <: c'_2,\ \ldots$

$$\frac{\Gamma \vdash e_0 : c_0 \quad \Gamma \vdash \underline{e} : \underline{c} \quad \mathit{type}(m, c_0) = \underline{c'} \to c_1 \quad \underline{c} <: \underline{c'}}{\Gamma \vdash e_0.m(\underline{e}) : c_1}$$

- Instantiation

$$\frac{\Gamma \vdash \underline{e} : \underline{c} \quad \underline{c} <: \underline{c'} \quad \mathit{fields}(c) = \underline{c'\ f}}{\Gamma \vdash \mathtt{new}\ c(\underline{e}) : c}$$

Static Semantics

- All casts are *statically* valid

$$\frac{\Gamma \vdash e_0 : c'}{\Gamma \vdash (c)\, e_0 : c}$$

Utrecht University

Static Semantics

- Subclass relationship

$$\frac{}{c <: c}$$

$$\frac{c <: c' \qquad c' <: c''}{c <: c''}$$

$$\frac{T(c) \;=\; \texttt{class}\; c \;\texttt{extends}\; c' \{\ldots\}}{c <: c'}$$

Static Semantics

- Well-formed classes:

fields of super

fields of new class

$$\frac{k = c\ (\underline{c'\ x'},\ \underline{c\ x})\{\texttt{super}(\underline{x'})\texttt{;this.}f_1\texttt{=}x_1\texttt{;}\ \ldots\}\ \textit{ok} \qquad \textit{fields}(c') = \underline{c'\ f'} \qquad m_i\ \textit{ok in}\ c}{\texttt{class}\ c\ \texttt{extends}\ c'\ \{\underline{c\ f}\texttt{;}k\ \underline{m}\}\ \textit{ok}}$$

constructor

methods

Static Semantics

$$\frac{k = c\ (\underline{c'\ x'},\ \underline{c\ x})\{\texttt{super}(\underline{x'});\texttt{this}.f_1\texttt{=}x_1;\ \ldots\}\ \mathbf{ok} \qquad \mathit{fields}(c') = \underline{c'\ f'} \qquad m_i\ \mathbf{ok}\ \mathbf{in}\ c}{\texttt{class}\ c\ \texttt{extends}\ c'\ \{\underline{c\ f};k\ \underline{m}\}\ \mathbf{ok}}$$

```
class ColourPoint extends Point {
  Colour c;

  ColourPoint (int x, int y, Colour c) {
   super (x, y);
   this.c = c;
  }

  Colour getc () {return this.c;}
}
```

Utrecht University

Static Semantics

To find the type of a method m of a class c, we have to search upwards in the class hierarchy for the definition of m.

methods defined in this class

the method we're looking for

$$\frac{T(c) = \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ldots;\ldots\underline{m''}\} \quad m''_i = c_i\ m(\underline{c_i\ x_i})\{\texttt{return}\ e\}}{\mathit{type}(m,c) = \underline{c_i} \rightarrow c_i}$$

the method we're looking for is not defined in this class

so check parent class

$$\frac{T(c) = \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ldots;\ldots\underline{m''}\} \quad m \notin m'' \quad \mathit{type}(m,c') = \underline{c_i} \rightarrow c_i}{\mathit{type}(m,c) = \underline{c_i} \rightarrow c_i}$$

$$\frac{}{\mathit{type}(m,\texttt{Object}) = \textbf{noType}}$$

Static Semantics

- Method overriding: new subclass methods must have
  - the same argument type
  - the same result type

  as the superclass method

then just make sure the body of the method is well typed

the method not present in any of the parent classes

$$\frac{T(c) = \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ldots\} \quad \mathit{type}(m,c') = \textbf{noType} \quad \underline{x}:\underline{c}; \texttt{this} : c \vdash e_0 : c_0}{c_0\ m(\underline{c\ x})\{\texttt{return}\ e_0\};\ \textbf{\textit{ok in}}\ c}$$

the method present in one of the parent classes

$$\frac{T(c) = \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ldots\} \quad \mathit{type}(m,c') = \underline{c} \rightarrow c_0 \quad \underline{x}:\underline{c}; \texttt{this} : c \vdash e_0 : c_0}{c_0\ m(\underline{c\ x})\{\texttt{return}\ e_0\};\ \textbf{\textit{ok in}}\ c}$$

make sure method has the same type as in parent class

Static Semantics

- Properties
  - casts may fail at run time: checks required
  - method invocation is statically checked
  - field selection is statically checked
  - objects have a statically determined type, but their actual type at runtime might be different
    - static vs actual dynamic type
    - dynamic semantics has to take it into account to preserve type safety!

Dynamic Semantics

- We discuss parts of the single step or structural operational semantics of Featherweight Java

- Values: an instance is a value if all of it's arguments are values

$$\frac{\underline{v}\ \textit{value}}{\texttt{new}\ c\ (\underline{v})\ \textit{value}}$$

in essence, an instance is just a collection of named fields, labelled with class names

- If not all the arguments of a constructor (or method invocation) are evaluated yet, we evaluate from the left to the right (no side effects, so order doesn't actually matter)

$$\frac{e \mapsto e'}{\texttt{new}\ c\ (\underline{v}, e, \underline{e}) \mapsto \texttt{new}\ c\ (\underline{v}, e', \underline{e})}$$

Dynamic Semantics

- Field Selection
  - c_i' f_i': *fields of a superclass*
  - c_i f_i : *fields of a class itself*
  - retrieve values of field from either superclass or class itself

$$\frac{\mathit{fields}(c) = \underline{c'}\ \underline{f'}\,;\underline{c}\ \underline{f}\,;}{\texttt{new}\ c(\underline{v'}, \underline{v}).f'_i \mapsto v'_i}$$

Dynamic Semantics

- Field Selection
  - c_i' f_i': *fields of a superclass*
  - c_i f_i : *fields of a class itself*
  - retrieve values of field from either superclass or class itself

$$\frac{\mathit{fields}(c) = \underline{c'}\ \underline{f'};\underline{c}\ \underline{f};}{\texttt{new}\ c(\underline{v'},\underline{v}).f'_i \mapsto v'_i}$$

$$\frac{\mathit{fields}(c) = \underline{c'\ f'};\underline{c\ f};}{\texttt{new}\ c(\underline{v'},\underline{v})\,.\,f_j \mapsto v_j}$$

Note: contents of fields cannot be changed after initialisation.

We could extend the language using the techniques described previously

Utrecht University

Dynamic Semantics

- Type Cast $(c')\texttt{new}\ c(\underline{v})$
  - if c' is a super type of c, the cast is just ignored
  - otherwise, cause a checked run-time error

$$\frac{c <: c'}{(c')\texttt{new}\ c(\underline{v}) \mapsto \texttt{new}\ c(\underline{v})} \qquad \frac{c' \not<: c}{(c')\texttt{new}\ c(\underline{v}) \mapsto \texttt{error}}$$

➡ during evaluation, we discard the information about the static type!

Dynamic Semantics

- Method Invocation
  - how can we find the correct methods for method invocation?

$$\texttt{new}\ c\ (\underline{v}).m(\underline{v'}) \mapsto ?$$

Dynamic Semantics

- Dynamic Dispatch
  - find the body of a method m of a class c, we search upwards in the class hierarchy for the first definition of m.

$$\frac{T(c) = \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ldots\underline{m}\} \quad m_i = c_i\ m(\underline{c_i}\ \underline{x})\{\texttt{return}(e)\}}{\boldsymbol{body}(m,\, c) = \underline{x}{\rightarrow}e}$$

Dynamic Semantics

- Dynamic Dispatch
  - find the body of a method $\boldsymbol{m}$ of a class $\boldsymbol{c}$, we search upwards in the class hierarchy for the first definition of $\boldsymbol{m}$.

$$\frac{T(c) = \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ldots \underline{m}\} \quad m_i = c_i\ m(\underline{c_i}\ \underline{x})\{\texttt{return}(e)\}}{\boldsymbol{body}(m,\, c) = \underline{x} \rightarrow e}$$

$$\frac{T(c) = \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ldots \underline{m}\} \quad m \notin \underline{m} \quad \boldsymbol{body}(m,\, c') = \underline{x} \rightarrow e}{\boldsymbol{body}(m,\, c) = \underline{x} \rightarrow e}$$

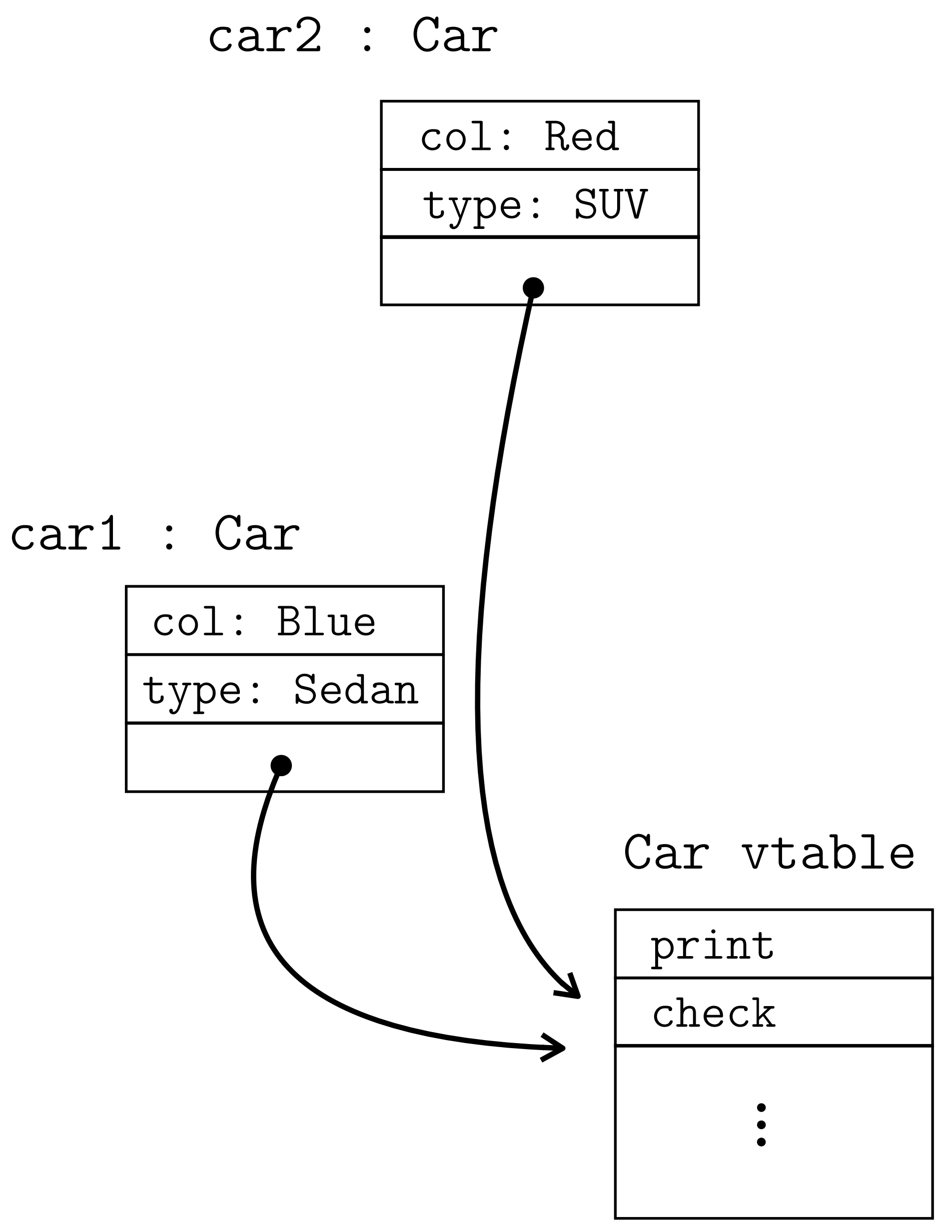
Dynamic Semantics

- Method Invocation
  - method invocation relies on an auxiliary predicate, ***body*** (defined later) which provides the list of formal arguments and the body of a method ***m*** defined in a class ***c***
  - replace all formal parameters $\underline{x}$ by actual parameters $\underline{v}'$
  - replace every occurrence of `this` by the instance itself
  - substitution semantics is possible, because we have no destructive updates in the language

$$\frac{\textbf{\textit{body}}(m,\ c) = \underline{x} \rightarrow e_0}{\texttt{new}\ c\ \texttt{(}\underline{v}\texttt{)}\texttt{.}m\texttt{(}\underline{v}'\texttt{)} \longmapsto e_0[\underline{x} := \underline{v}'][\texttt{this} := \texttt{new}\ c\ \texttt{(}\underline{v}\texttt{)}]}$$

Dynamic Semantics

- How is method invocation implemented?
  - searching for the method every time would be too slow!
  - objects have a hidden field
    - reference to a (virtual) method table, which contains references to all methods of the class
    - table shared among all the objects of a class
    - not necessary to traverse the whole class hierarchy at runtime, because table for a class can be determined at compile time
    - still, it's more expensive than a regular function call



Type Safety

- Is Featherweight Java type save?
  - dynamic semantics of casts preserves actual (ie, dynamic) type of an instance
  - the actual type of an expression may be “smaller” in the subtype ordering during execution
- Preservation:

  If $e : c$ and $e \mapsto e'$, then $e' : c'$ for some c' such that $c' <: c$

- Progress

  If $e : c$ then either

  1. e is a value `new` $c'(\underline{v})$, with $c' <: c$ or
  2. e is equal to `(`c`)new` $c'(\underline{v})$, with $c' \not<: c$, then $e \mapsto$ `error` or
  3. there exists e', such that $e \mapsto e'$

Utrecht University

Conditionals and subtyping

- What should the type of a conditional expression be?

```
if e then e1 else e2
```

Java

- In Featherweight Java, the subclass relationship (inheritance) is a special form of subtyping (we can coerce by 'deleting' fields)
- Inheritance (subclassing) and sub typing are not the same
  - inheritance is a method of code re-use through extension
  - subtyping expresses a behavioural relationship
- In Java
  - inheritance gives rise to a subtype relationship
  - not every subtype relationship in Java arises through inheritance

Utrecht University

Java

- Featherweight Java has two types of polymorphism

- operations work on a class and all of its subclasses (form of subtyping)
  - methods can be overwritten
    - all methods have the same type, object ‘knows’ what the correct method is
      - dynamic dispatch default for Java, but not in C# (need to use virtual m.)
    - different to overloading, which can be resolved via type (compile time)
      - many OO language allow overloading of methods with different types
      - Haskell overloading resolved via type

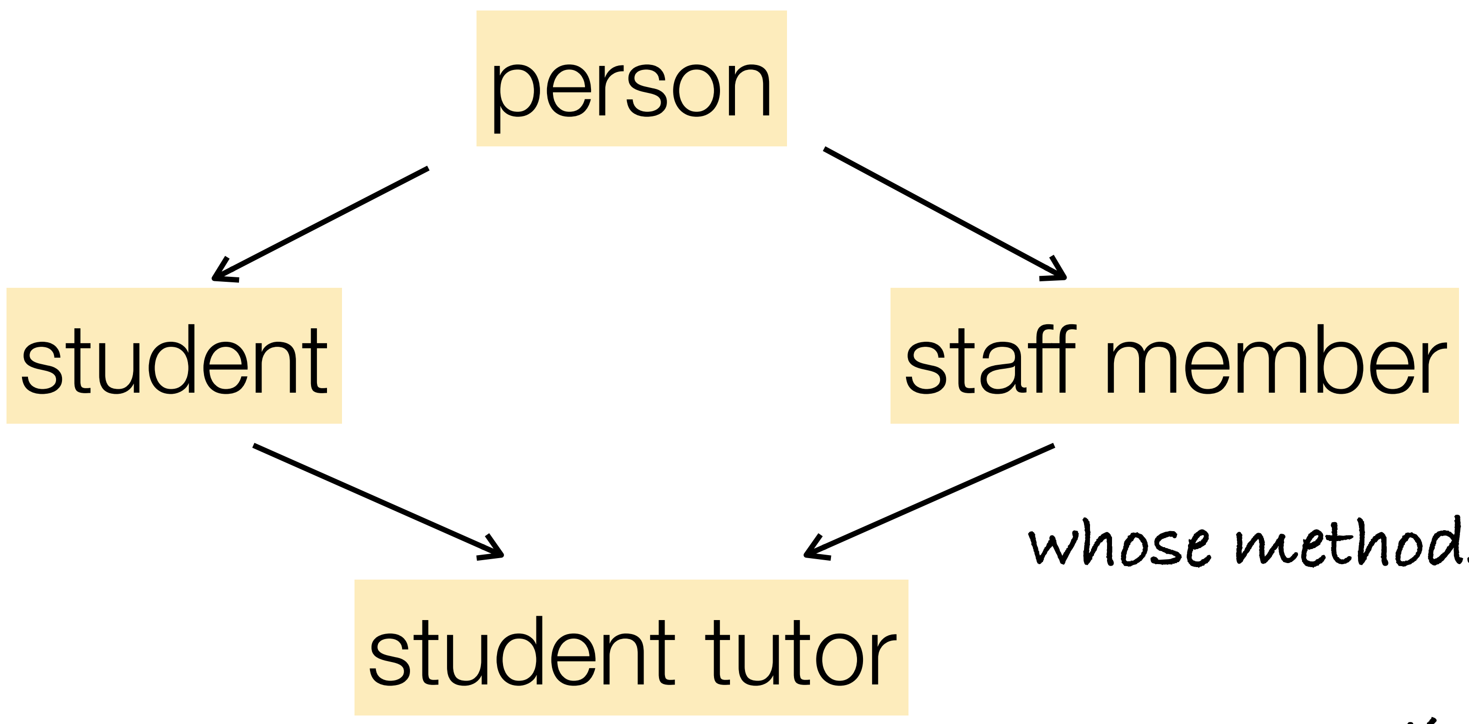
```
(==) ::   Eq a => a -> a -> Bool

‘‘123’’ == ‘‘Hi’’
```

Utrecht University

Multiple inheritance

- Some OO language allow classes to extend/inherit from multiple superclasses



whose methods should be inherited?

The diamond problem

  - supported in C++, Eiffel, Python, OCaml
  - not supported in C#, Swift, Java (though some features available through interfaces/interfaces)

Utrecht University