Grundlagen von OO

Harald Fecher Martin Steffen

Christian-Albrechts University Kiel

Summer 2005

Structure

Class-based languages

Advanced class-based features

Object protocols

Variance

- goal: type disciplines to avoid *typecase* etc,

$\Rightarrow$ flexibility, expressiveness, but still statically checkable

- for illustration: 3 “type constructors”/“type operators:
  1. product types $S \times T$: covariant
  2. function types $S \rightarrow T$: contra/covariant
  3. updatable products $S \# T$: invariant

Method specialization

- so far: overriding: identical type
- relax: method specialization (method spec. on override)
- note: we don't do this for fields (updatable!)

```
class c is
  method m(x:A) : B is ... end;
  method m1(x:A₁) : B₁ is ... end;
end;

subclass c′ of c is
  override m(x: A′) is ... end;
end;
```

Again: what are we allowed (safely) to do for A', B' in connection with A/B?

Method specialization and self

- implicit form of method specialization: hidden parameter self
- self may occur in the methods of c or c'
- typed: $\mathit{InstanceTypeOf}(c)$ resp. c'.

$\Rightarrow$ type of self gets specialized covariantly! (on inheritance?)

Inheritance/premethods

- different perspective: methods as functions

⇒ pre-method

- objects as record of pre-methods (+ perhaps fields)

$$pm_1 : InstanceTypeOf(c) \times A_1 \rightarrow B_1 \triangleq T$$
$$pm_2 : InstanceTypeOf(c') \times A_1 \rightarrow B_1 \triangleq T'$$

- $T \leq T'$

⇒ pm_1 is of the type of a legal premethod for c'

⇒ inheritance possible

$$\frac{c' \leq c \qquad A_1' \leq A_1 \qquad B_1 \leq B_1'}{InstanceTypeOf(c) \times A_1 \rightarrow B_1 \leq InstanceTypeOf(c') \times A_1' \rightarrow B_1'}$$

- take care: difference to override!

Self-type specialization

- so far: method specialization: nice/flexible but not enough
- ignored: method types independent of the (type of the) class[1]
- often: “recursive” class definition (type-wise): c contains InstanceTypeOf(c)

[1] S and T independent of c. Exception: self parameter

Self-type specialization

```
class c is
  var       x: int := 0;
  method m(): InstanceTypeOf(c) is ... self .. end;
end;

subclass c of c is
  var y: int := 0
end;
```

Self-type specialization

- so far: method specialization: nice/flexible but not enough
- ignored: method types independent of the (type of the) class[1]
- often: "recursive" class definition (type-wise): *c* contains InstanceTypeOf(c)
- m is being inherited.
- Question: what's "now"the return type of *m*? in general?

  InstanceTypeOf(c) or InstanceTypeOf(c′)?

- InstanceTypeOf(c'): certainly wrong[2]
- in practice: often "return self" ⇒ then *InstanceTypeOf*(*c*′) sound
- loss of information
- typecase/casts can be avoided

[1] *S* and *T* independent of *c*. Exception: self parameter

Self types

```
class c in
  var     x: int := 0;
  method m(): Self is ... self ... end;
end;
```

- assumption: *Self* is (will be) subtype of c
- note: Self in co-variant position! (for inheriting)
  - more expressive
  - avoids loss of info
  - *Self*: harder to type check than InstanceTypeOf(c)
- *Self*: even as type of fields possible, if assured that one only updates via self
- unsound: *Self* in contra-variant (arg.) position
  - *Eiffel* did this
  - later may comes info how to defuse that partially nonetheless.

Class-based languages

Advanced class-based features

Object protocols

Object types

- often: correlation of inheritance = subclassing = subtyping: simple, but unclear/not too flexible . . .

$\Rightarrow$ separation (in particular) of $\leq$ and $\leq$

- separation of spec. and implementation[3]
  - type:
    - usage, no code
    - implementation independent
    - other names: type signature, object protocol, interface.

[3] always a good idea

Object types

- often: correlation of inheritance = subclassing = subtyping: simple, but unclear/not too flexible . . .

$\Rightarrow$ separation (in particular) of $\underline{\leq}$ and $\underline{\leq}$

- separation of spec. and implementation[3]
  - type:
    - usage, no code
    - implementation independent
    - other names: type signature, object protocol, interface.

```
ObjectType Cell is
  var     contents: int;
  method  get():    int;
  method  set(n:int);      // n not needed, void as
end;                      // return type
```

```
ObjectType Cell is
```

Subclassing implies subtyping

- so far $\leq$ is $\leq$, now: something's new needed
- subtyping
  - structural problem: accidental matching
  - by names (nominal): conceptually unclear
- simplest form: width: rule
- especially: *invariant* "components" types[4]
- cf. *multiple subtyping*

$$\frac{c' \leq c}{ObjectTypeOf(c') \leq ObjectTypeOf(c)}$$

- note: implication only, partial decoupling

  subclassing-implies-subtyping

[4] no (method)-specilization yet.

Type parameter

- general “technique” for code reuse[5]
- more flexibility in OO

```
Object Type Person is
  method eat(food: Food);
end;

Object Type Vegetarian is
   ..
    method eat (food: Vegetables);

end
```

- Vegetarian $\leq$ Person?
- solution: abstraction!
- type operators

example

```
ObjectOperator  PersonEating[F <: Food] is
..
  method eat(food: F);
end;

ObjectOperator VegetarianEating[F <: Vegetables] is
  ..
   method eat (food: F);

end
```

Bounded type parameterization

- F: type parameter/type variable
- instantiated with a type
- explicit specification of the upper bound
  - VegetariantEating[Food]: Nope
  - VegetariantEating[Veggies] = Vegetarian
- take care:
  - PersonEating: no type, there are no members
  - VegetarianEating $\leq$ PersonEating

$$\frac{F \leq \mathit{Vegetables}}{\mathit{VegetarianEating}[F] \leq \mathit{PersonEating}[F]}$$

  - especially: Vegetarian = VegetarianEating[Veggies] $\leq$ PersonEating[Veggies]

Bounded abstract types

```
Object Type Person is
  type F ≤ Food        // bounded  by Food
  var lunch : F              // personal lunch

  method eat (food: F);
end;
```

Bounded abstract types

- also: partially abstract types
- different solution to the “vegetarians-as-persons” problem
- no parameter for “F”

- problem; F no parameter
- no food outside Person
- instantiation: choice of (internal) F + lunch
- but

$$\boxed{\mathit{Person} \leq \mathit{Vegetarian}}$$

Subclassing without subtyping

- so far: object types: partial decoupling
- now: complete decoupling

inheritance-is-not-subtyping

- goal: Self-types in contravariant position

$\Rightarrow$: “more inheriantce, less subsumption” (since subclasses do no longer lead to subtyping)

Example

```
Object Type Max is
  var     n : int;
  method max(other: Max): Max
end
```

```
Object Type MinMax is
  var     n : int;
  method max(other: MinMax): MinMax;
  method min(other: MinMax): MinMax;
end
```

```
class maxclass is
  var     n : int := 0;
  method max(other: Self): Self is
   if    self.n > other.n
  then self
  else other
end


subclass minmaxClass of maxClass is
  var     n : int := 0;

  method min(other: Self) : Self is
     if ``the other way around''  then .. else ..
end
```

- min/max: binary methods
- formal: what is “Self” in the object types?
- Recursion

$$ObjectTypeOf(maxClass) = Max \quad (1)$$

Example

```
subclass minmaxClass' of minmaxClass is // further
  override max(other: Self): Self is
    if   other.min(self) = other
    then return self
    else return other
end;
```

- inheritance of method min: specialization of *Self*
- we know minmaxClass $\leq$ maxClass, but

$$Minmax \not\leq Max$$

- problem: overriding, contravariant self
- binary methods

Class-based languages

Advanced class-based features

Object protocols

Object protocols

- as seen: subclasses $\not\Rightarrow$ subtypes
- of course *MinClass* and *MaxClass* not without relationship

$\Rightarrow$ new relationship

- first of all: op. abstraction does not help

  ObjectOperator P[M $\leq$ Max]:= ... end;
- reason: MinMax $\not\leq$ Max
- but: MinMax can “do” everything that Max “can do”

$\Rightarrow$ subprotocol relation

Example

```
ObjectOperator MaxProtocol[X] is
  var n: Int
  method max(o:X) : X;
end;

ObjectOperator MinMaxProtocol[X] is
  var n: Int
  method max(o:X) : X;
  method min(o:X) : X;
end;
```

Object protocols

- recursive types in Min/Max
- note: for each recursive type one has the "abstraction"

$$\textit{rec. type } T \Rightarrow \textit{type operator/T-protocol}$$

$$\textit{fixpoint of T-protocol} \Rightarrow T$$

- relation of Max and MinMax?
  1. MinMax $\leq$ MaxProtocol[MinMax]
  2. alternative
     - define $\leq$: $\dfrac{\forall T.P[T] \leq P'[T]}{P \leq P'} \leq$
     - then MinMaxProtocol $\leq$ MaxProtocol

Sub-protocols

2 solutions

$$\frac{S \leq T - Protocol(S)}{S \leq_{sp} T}\ \text{SP} \qquad \frac{S - Protocol \leq T - Protocol}{S \leq_{sp} T}\ \text{S}$$

- back to the failed def. of the O-operator
  1. ObjectOperator P_1[X ≤ MaxProtocol[X]] is . . . end;
  2. ObjectOperator P_2[P ≤ MaxProtocol] is . . . end;
- now it works:
  1. P_1[MinMax]
  2. P_2[MinMaxProtocol]
- 1. F-bounded parameterization
  2. higher-order bounded parameterization

Matching

- no *explicit* intro of type operators
- $\leq_{\#}$: relation over types
- intention: capture subprotocol relation

$$\frac{S \leq T - Protocol[S]}{S \leq_{\#} T}\ \textsc{Match}_1$$

$$\frac{S - Protocol \leq T - Protocol}{S \leq_{\#} T}\ \textsc{Match}_2$$

$\Rightarrow$ $MinMax \leq_{\#} Max$

- no subsumption
- but: as said: good for parameterization
- wrap up: contra-variant self = binary methods $\Rightarrow$ subprotocol/matching

[1] Martín Abadi and Luca Cardelli.
A Theory of Objects.
Monographs in Computer Science. Springer, 1996.