Towards a Type System for Detecting Never-Matching Pointcut Compositions

Tomoyuki Aotani Hidehiko Masuhara

Programming Principles and Practices Group

University of Tokyo

Never-matching pointcut

Don't match any join point in any program

- get(* *)&&set(* *)
  - No get join point is a set join point
- get(* *)&&args(int)
  - No get join point has an argument

realize

```
abstract aspect A{
  abstract pointcut p();
  after(int i):
      p()&&args(i){…}
}
```

```
aspect B extends A{
  pointcut p(): get(* *);
}
```

Our approach: detect by using a type system

- Type of pointcuts
  - Represents attributes of matching join points
  - Is encoded by using record, union and the bottom types
- Guaranteed properties
  - Well-formedness of pointcuts

The property our type system assures: well-formedness

- Well-formed pointcuts:
  A pointcut p is well-formed if there exists a well-typed program that has a join point matching p

Target language

- Subset of AspectJ's pointcut language
  - mget(*T C.f*): selects a reference to an instance field (not declared as static).
  - mset(*T C.f*): selects an assignment to an instance field (not declared as static).
  - args(*T1,…,Tn*): specifies the number of arguments and their types.
  - *p1* && *p2*: makes an intersection of two pointcuts
  - *p1* || *p2*: makes an union of two pointcuts

Typing rules for mget, mset and args pointcuts

- Assign record types that represent the properties of matching join points
  - mget(*T C.f*):
    {target: *C*, args: •, kind: mget, name: *f*,
    ret: *T*}
  - mset(*T C.f*):
    {target: *C*, args: [*T*], kind: mset, name: *f*,
    ret: •}
  - args(*T1,…,Tn*): {args: [*T1,…,Tn*]}

T, *C* and *f* are identifiers or *
• represents absence

Typing rules for pointcut compositions

P: type of pointcut
pc: pointcut

- ||-composition

$$\frac{pc_1 : P_1 \quad pc_2 : P_2}{pc_1 || pc_2 : P_1 + P_2}$$

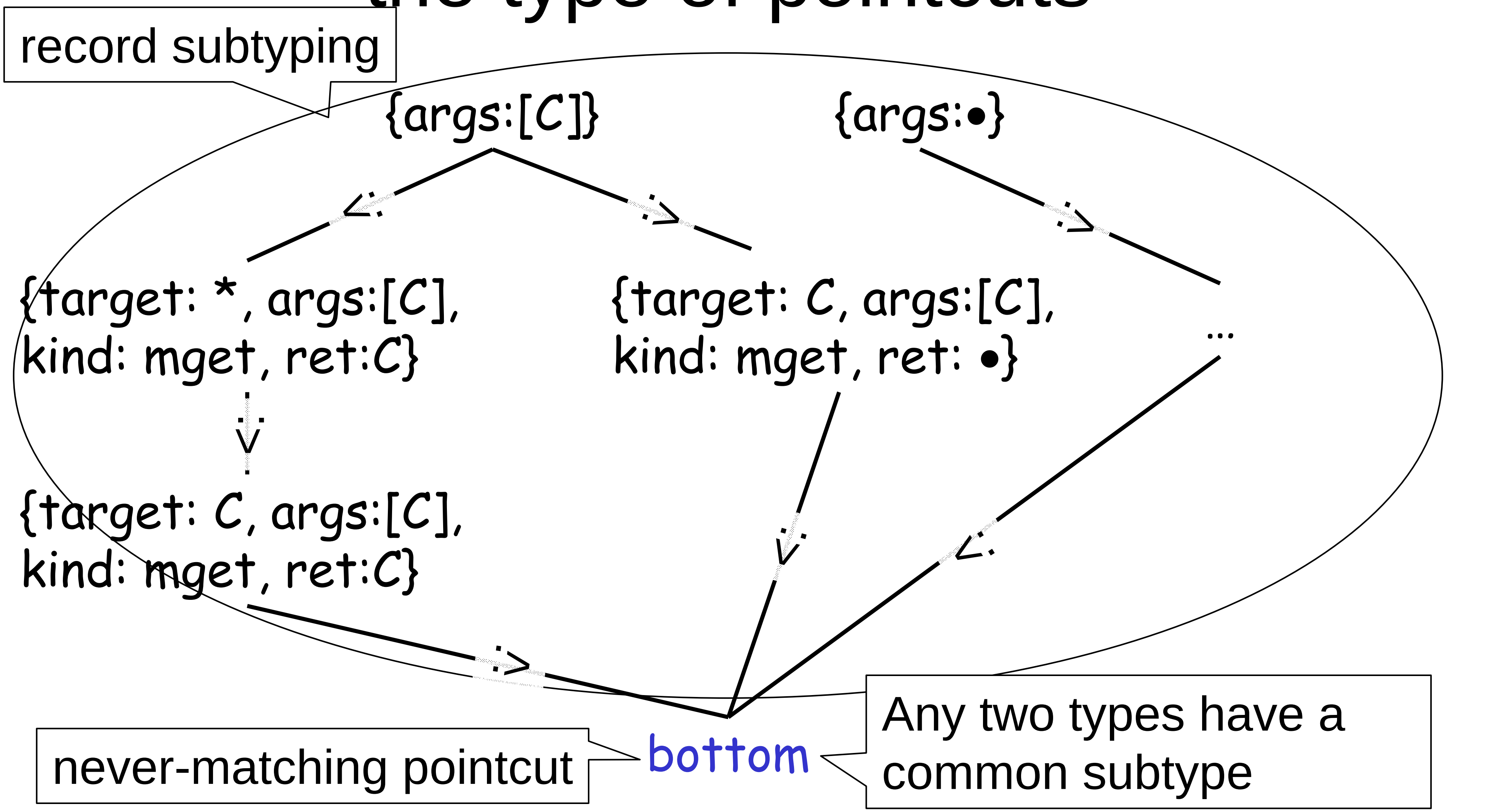
union type

- &&-composition

P: a common subtype of P_1 and P_2

$$\frac{pc_1 : P_1 \quad pc_2 : P_2 \quad \boxed{P <: P_1 \quad P <: P_2}}{pc_1 \&\& pc_2 : P}$$

Type subsumption on the type of pointcuts



Well-formed pointcut:
args(int) && mset(int Point.x)

mset(int Point.x): {target: Point, args: [int] , kind: mset, name: x, ret: •} (T_2)

args(int): {args: [int]} (T_3)

$$\frac{T_1 <: T_2 \qquad T_1 <: T_3}{\text{args(int) \&\& mset(int Point.x): \{target: Point, args: [int], kind: mset, name: x, ret: } \bullet\text{\}}}$$

args(int) && mset(int Point.x): {target: Point, args: [int], kind: mset, name: x, ret: •} (T_1)

Never-matching pointcut: args(int) && mget(int Point.x)

mget(int Point.x):

{target: Point, args: •
kind:mget, name:x, ret:int}

T_1

args(int): {args: [int]}

T_2

have no common subtype

bottom <: T_1

bottom <: T_2

---

args(int) && mset(int Point.x): bottom

Conclusion

- We defined the well-formedness of pointcuts
- We demonstrated our type system for pointcuts
  - The type of pointcuts is represented as record, union and the bottom types
  - Never-matching pointcuts have the bottom type

Future work

- Complete formalization
  - We use Featherweight Java [Igrashi01]
  - How can we define the typing rule for the **not** (!) pointcut?
- Prove type-soundness
  - Well-typed programs are well-typed after well-typed aspects are woven, and don't go wrong
- Verify correctness of the design and implementation of pointcuts in AspectJ5

Related work (1/2): Types and AspectJ-like AOPL

- Typed parametric polymorphism for aspects [Jagadeesan06]
  - provides AFGJ (FGJ[Igarashi01] + pointcut + advice + proceed)
    - join point: execution
    - pointcut: exec, &&, ||
  - provides checking rules for pointcuts, which can successfully reject exec(R C.m()) && exec(* C.m'())
- MiniMAO$_1$:An imperative core language for studying Aspect-Oriented resoning [Clifton06]
  - Classic Java[Flatt99] + aspect + pointcut + advice + proceed
    - join point: call, execution
    - pointcut: call, exec, this, args, target, &&, ||, !
  - provides typing rules for pointcuts but it cannot reject exec(R C.m()) && call(R C.m())

Related work (2/2): Types and Pointcuts

- A Static Aspect Language for Checking Design Rules [Morgan07]
  - develops a DSL that can be seen that enriches declare error/warning in AspectJ.
  - provides a type system that assures a pointcut matches at least one join point.
    - The typing rule for `not` pointcut is also defined.
- A pointcut language for control-flow [Douence04]
  - provides a richer pointcut language for control-flow than AspectJ's.
  - discusses erroneous pointcut compositions and aspect interactions based on sets of join point shadows.

Typing rule for not pointcut in DSL[Morgan07]

- $!pc$ has the same type to pc i.e. $\dfrac{pc : P}{!pc : P}$

Joinpoint Type	Description
namespace	Namespace
type	Type
method	Method (including constructors)
argument	Method argument
field	Field
property	Property
event	Event
attribute	Attribute of a program element
genericArgument	Type argument (to a generic type)
bytecode	Instruction in the program

Well-formed pointcut: (mget(* *.*) || mset(* *.*)) && args(int)

- mget(* *.*) && args(int): bottom
- mset(* *.*) && args(int): {target:*, args: [int], kind:mset, name:*, ret:.}

Using the typing rule for ||-compositions,

- (mget(* *.*) || mset(* *.*)) && args(int): {target:*, args: [int], kind:mset, name:*, ret:.} + bottom

Limitation of current type system

- ArrayList <: Object cannot be accepted
  - Our type system does not know the relation of ArrayList and Object
- Possible solution: specifying a reliable class hierarchy H

$$\frac{H \vdash \text{ArrayList} <: \text{Object}}{H \vdash \{\text{this:ArrayList}\} <: \{\text{this:Object}\}}$$

Unsafe join point reflection

caching return values of method calls

```
aspect Memoize{
  Hashtable store;
  after(): call(* *()){
    Object key=tjp.getTarget();
    if(!store.containsKey(key))
      store.add(key.clone(),proceed());
    return store.get(key);
  }
}
```

returns null when tjp matches calls to class methods

throws NullPointerException

Rejecting unsafe join point reflection

```
call(* *()):
{target:Maybe<*>, args:[], kind:call, ret:*}
```

```
tjp.getTarget(): Maybe<*>
```

```
aspect Memoize{
  Hashtable store;
  after(): call(* *()){
    Maybe<Object> key=tjp.getTarget();
    if(!store.containsKey(key))
      store.add(key.clone(),proceed());
    return store.get(key);
  }
}
```