Back to the future *Pointcuts as Predicates over Traces*

Karl Klose, Klaus Ostermann
Darmstadt University of Technology, Germany

Introducing GAMMA

- Object-oriented core language
  - Similar to Featherweight Java
  - Supports storage, assignments, etc.
- Aspects
  - Prolog-based pointcut language
  - Use unification to perform pointcut matching and variable binding

Aspects in GAMMA

```
class main extds Object {
 bool var;
 before set(Now,_,Address,_,_) {
  print(Address)
 }
 bool main(bool x){
   this.var := true
 }
}
```

- Pointcuts are Prolog queries
  - First argument of predicates is always a timestamp
  - **Now** denotes the time of activation
  - Variables can be used in advice
  - _ is an anonymous variable

GAMMA's pointcut language

- The whole trace of a program execution is represented as a set of Prolog facts
  - Facts represent atomic interpreter steps
    - Reading/writing fields
    - Calling a method
    - Creating objects, etc.
  - Each fact has a unique timestamp
- Pointcuts are predicates over the execution trace
  - Can refer to any point in the complete execution

Representing traces

`newObject(6, file)`
a new instance of class **`file`** has been created

`set(7, main, iota1, input, iota3)`
field **`input`** of **`main`** instance at ***`iota1`*** is set to value ***iota3***

`get(8, main, iota1, memory, iota2)`
field **`memory`** of **`main`** instance at ***iota1*** is read, value was ***`iota2`***

`calls(9, mem, iota2, alloc, true)`
method **`alloc`** of **`mem`** instance at ***iota2*** called with parameter ***true***

`endCall(10, 9, true)`
method-call at timestamp ***9*** has ended with result ***true***

Expressing temporal relations

```
before set(Now,_,_,varx,_),
       set(T,_,_,vary,_),
       isbefore(T,Now)
{...}
```

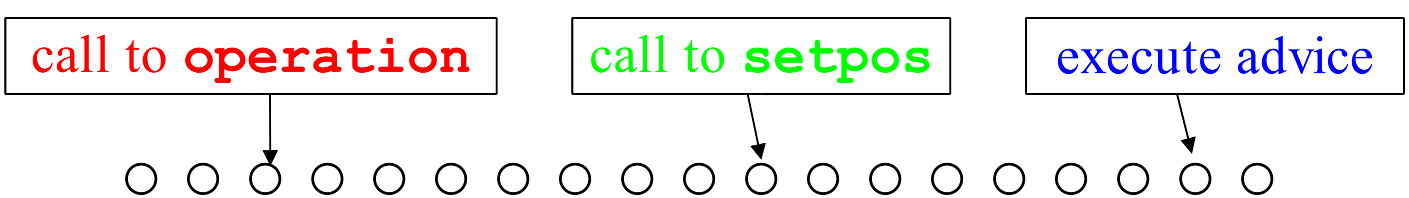
```
% T2 is in the control flow of
   the call at T1
cflow(T1, T2) :-
  calls(T1,_,_,_,_),
  endcall(T3,T1,_),
  isbefore(T1,T2),
  isbefore(T2,T3).
```

- Timestamps can be related by the predicate **`isbefore`**
- Predicates like **`cflow`** can be fomulated as rules
- Can describe sequences
  - e.g. to implement protocols

Example: Display update

```
before
calls(T1,main,_,operation,_),
cflow(T1,T2),
calls(T2,point,_,setpos),
endCall(Now, T1, _)
{
 this.display.update(true)
}
```

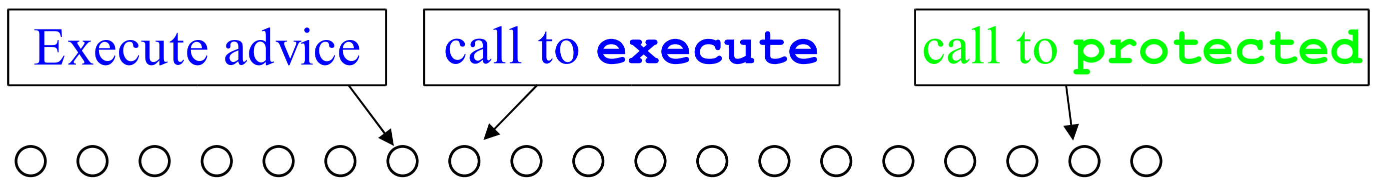
- Update display if points have been moved in **`operation`**
  - Update after completing operation
  - And do it only once



Example: Authentication

```
before
calls(Now,server,_,execute,_),
cflow(Now,T),
calls (T,database,_,protected,_)
{
 this.db.authenticate(true)
}
```

- Method **`protected`** needs authentication
- Authenticate
  - only if **`execute`** calls **`protected`**
  - But before calling **`execute`**



Paradox aspects

```
class main extds Object{
    bool create;
    before calls(Now,_,_,foo,_),
           newObject(T,a),
           isbefore(Now,T) {
        this.create := false
    }
    bool foo(bool x){
        if this.create
        then (new a; true)
        else false
    }
    bool main(bool x){
        this.create := true;
        this.foo()
    }
}
```

- Analogy to grand-mother paradoxon
  - Base program creates an object of class a
    - Enables aspect
  - The advice prevents this creation
    - Disables aspect

A model of advice application

- Look at the trace of a program as entity
  - Activation points of a trace are positions (timestamps) where pointcuts match
- Which advice should be executed first?
  - First idea: Take the earliest one
  - But: it makes difference which one is taken!
- How to handle aspect interaction?
  - Execution of advice may „inactivate“ the pointcuts of already executed advice

Properties of advice application

- T_P: Set of possible traces for a program P
- $t_1 \rightarrow_P t_2$ means that t_2 can be obtained from t_1 by
  - Inserting advice where pointcut matches
  - Removing advice whose pointcut does not match
- Observation
  - $\rightarrow_P$ may be indeterministic
  - $\rightarrow_P$ is not well-founded and not confluent
  - There is no canonical normal form

Using domain theory

- Define operator F_P from $\rightarrow_P$ by chosing a selection strategy
- Kleene: If $(T_P,\subseteq)$ is a cpo and F_P is scott-continuous then $\sup_{n\in\mathbb{N}}\langle F_P^{\,n}(\perp)\rangle$ is the least fixed point of F_P
- Problems
  - Find an partial order $\subseteq$ making $(T_P,\subseteq)$ a cpo
  - Find restrictions for programs such that F_P is scott-continuous

A sample cpo

- Let n be the length of trace s, a (b) the earliest activation point in s (t)...
- ...then define partial order $\sqsubseteq$ as the transitive and reflexive closure of

$$s \sqsubset_P t \;\Leftrightarrow\; t = (s_0, \ldots, s_{a-1}, \overbrace{u_0, \ldots, u_{k-1}}^{\text{trace of advice}}, v_0, \ldots, v_{l-1})$$
$$\wedge\; b > a + k + 1$$
$$\wedge\; n < a + l$$

Consequences

- Hard to check if F_P is scott-continuous
  - Need to look at advice interaction
  - Need sophisticated static analysis techniques
- Model has very limiting restrictions
  - Base program must terminate
  - Infinite computations can not be handled

A prototype implementation

- F_P is defined by always picking out the first activation point
- After each run, all pointcuts are passed to the Prolog database to determine the activation points
- The interpreter is reset to the timestamp of the first activation point and advice is executed

Conclusions

- GAMMAs allows to easily describe temporal relations between joinpoints
  - e.g. in protocols
  - Can emulate known temporal constructs, like cflow, as rules
  - Pointcuts can refer to past and future of the computation
- Implementation is difficult
  - Maybe interesting subsets can be implemented efficiently