A semantics for context-oriented programming languages with multiple layer activation mechanisms

Tomoyuki Aotani[†]

joint work with

Tetsuo Kamina[‡] and Hidehiko Masuhara[†]

† Tokyo Institute of Technology
‡ Ritsumeikan University

Context-oriented programming

An approach to software modularity

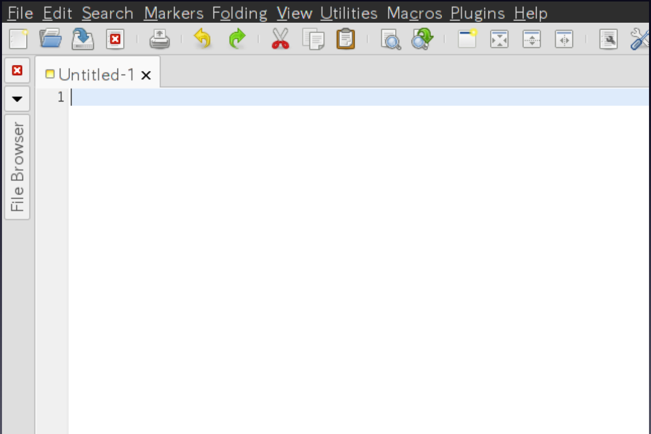
- Modularizing context-dependent behavioral variations

Contexts and behavior: saving the buffer content

```
> jedit &
```

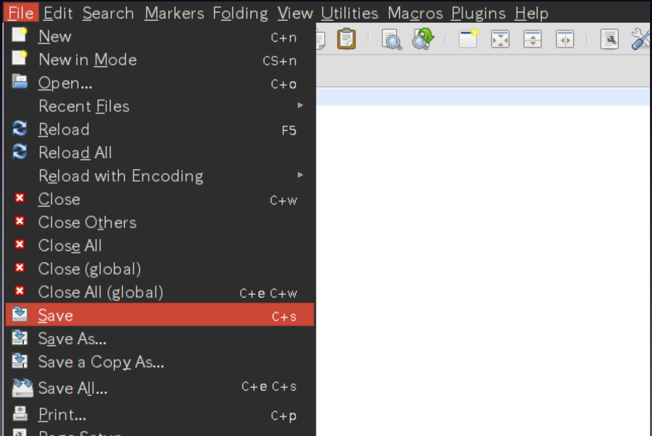
Contexts and behavior: saving the buffer content

One empty buffer with no associated file



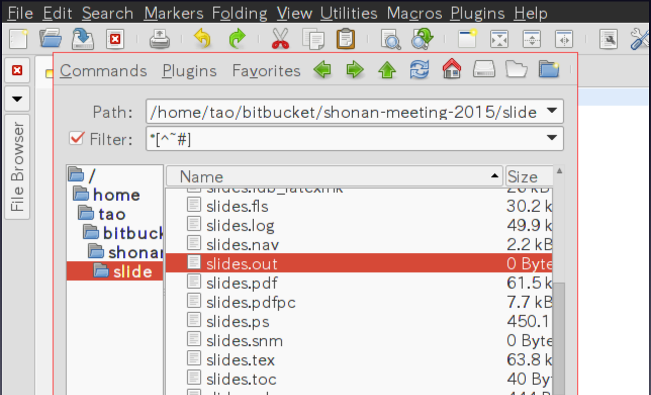
Contexts and behavior: saving the buffer content

When we save the buffer,



Contexts and behavior: saving the buffer content

When we save the buffer,
a dialog pops up and asks to specify the file



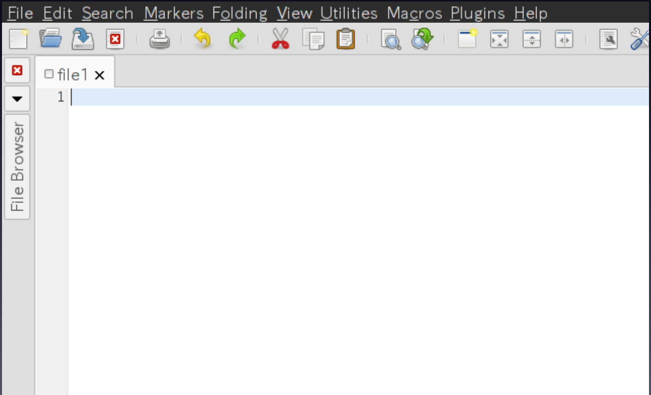
Path: /home/tao/bitbucket/shonan-meeting-2015/slide

☑ Filter: *[^~#]

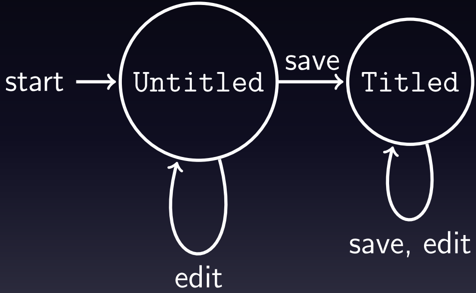
Name	Size
slides.fls	30.2 k
slides.log	49.9 k
slides.nav	2.2 kB
slides.out	0 Byte
slides.pdf	61.5 k
slides.pdfpc	7.7 kB
slides.ps	450.1
slides.snm	0 Byte
slides.tex	63.8 k
slides.toc	40 By

Contexts and behavior: saving the buffer content

Once we specified the file,
we are not asked to specify the file to save the buffer any more



Contexts and behavior: saving the buffer content



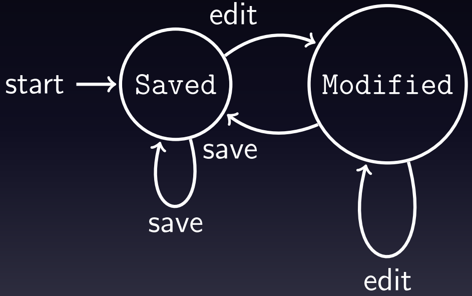
Untitled

```
void save(){
   /*ask the file name and*/
   /*save the content to the file*/
}
```

Titled

```
void save(){
   /*save the content to the file*/
}
```

Contexts and behavior: closing the application



Saved

```
void close(){
  /*close the app.*/
}
```

Modified

```
void close(){
  /*save the content to the file*/
  /*and close the app.*/
}
```

Context-oriented programming language

Context-oriented programming languages provide

- *Layers*
- *Partial method*
- *Layer activation mechanism*

Context-oriented programming language

Context-oriented programming languages provide

- *Layers* group partial methods and each has one binary state that is either active or inactive
- *Partial method*
- *Layer activation mechanism*

Context-oriented programming language

Context-oriented programming languages provide

- *Layers* group partial methods and each has one binary state that is either active or inactive
- *Partial method* in layer `L` for method `m`, namely `L.m`, defines the variation of the behaivor of `m` in the context represented by `L`
- *Layer activation mechanism*

Context-oriented programming language

Context-oriented programming languages provide

- *Layers* group partial methods and each has one binary state that is either active or inactive
- *Partial method* in layer `L` for method `m`, namely `L.m`, defines the variation of the behavior of `m` in the context represented by `L`
- *Layer activation mechanism* (de)activates layers at runtime

Layers and partial methods

```
class Buffer{
  File f;
  void save(){
    /*save to f*/
  }
}
layer Untitled{
  void Buffer.save(){
    /*ask the file name and*/
    /*save the content to the file*/
  }
}
```

Layers and partial methods

```
class Buffer{
  File f;
  void save(){
    /*save to f*/
  }
}
```

Layer

```
layer Untitled{
  void Buffer.save(){
    /*ask the file name and*/
    /*save the content to the file*/
  }
}
```

Layers and partial methods

```
class Buffer{
  File f;
  void save(){
    /*save to f*/
  }
}
layer Untitled{
  void Buffer.save(){
    /*ask the file name and*/
    /*save the content to the file*/
  }
}
```

Layer

Partial method

Layers and partial methods

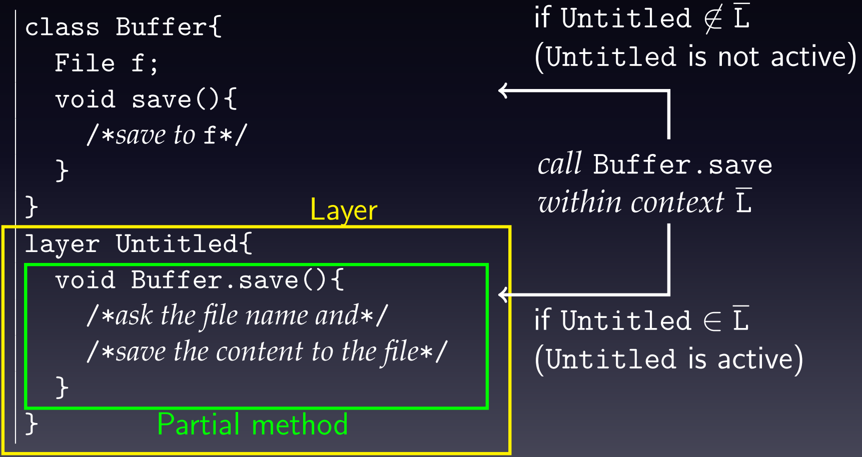
```
class Buffer{
  File f;
  void save(){
    /*save to f*/
  }
}
layer Untitled{
  void Buffer.save(){
    /*ask the file name and*/
    /*save the content to the file*/
  }
}
```

Layer

Partial method

call `Buffer.save`
within context $\overline{L}$

Layers and partial methods



```
class Buffer{
  File f;
  void save(){
    /*save to f*/
  }
}
layer Untitled{
  void Buffer.save(){
    /*ask the file name and*/
    /*save the content to the file*/
  }
}
```

if `Untitled` $\notin \overline{\texttt{L}}$
(`Untitled` is not active)

call `Buffer.save`
within context $\overline{\texttt{L}}$

if `Untitled` $\in \overline{\texttt{L}}$
(`Untitled` is active)

Layer activation mechanisms

- *Block*: (ContextJ [Appeltauer09], JCop [Appeltauer10])
- *Imperative*: (Subjective-C [González10])
- *Per-object*: (EventCJ [Kamina11], ContextErlang [Salvaneschi12])
- *Implicit/reactive*: (PyContext [von Löwis07], Flute [Bainomugisha12])

Example: block activation

`with(L)`$\{\overline{\texttt{S}};\}$ ensures that layer `L` is active during execution of $\overline{\texttt{S}}$

```
void main(){
  Buffer b=new Buffer();
  with(Untitled){
    b.save();   //→Untitled.Buffer.save
  }             // because L̄ = [Untitled]
  b.save();     //→Buffer.save because L̄ = ∅
}
```

Example: imperative activation

`activate(L)` activates layer `L` until `deactivate(L)` is executed

```
void main(){
  Buffer b=new Buffer();
  activate(Untitled);
  b.save();                   //→Untitled.Buffer.save
}                             //because L̄ = [Untitled]
```

Dynamic layer precedence

If layer $\mathtt{L}_1$ is activated more recently than layer $\mathtt{L}_2$, then $\mathtt{L}_1$ is more effective than $\mathtt{L}_2$

Example

Assume that class `Buffer` has two layers `Untitled` and `Modified` that define partial method `save`

```
void main(...){
  Buffer b=new Buffer();
  with(Untitled){        //[] ↦ [Untitled]
    with(Modified){      //[Untitled] ↦ [Modified, Untitled]
      b.save();          //→Modified.Buffer.save
    }                    //[Modified, Untitled] ↦ [Untitled]
  }                      //[Untitled] ↦ []
}
```

Dynamic layer precedence

If layer `L1` is activated more recently than layer `L2`, then `L1` is more effective than `L2`

Example

Assume that class `Buffer` has two layers `Untitled` and `Modified` that define partial method `save`

```
void main(){
  Buffer b=new Buffer();
  activate(Untitled); //[] ↦ [Untitled]
  activate(Modified); //[Untitled] ↦ [Modified, Untitled]
  b.save();           //→Modified.Buffer.save
```

Activating active layers

When an active layer is activated, it just moves to the head of the context $\overline{\mathtt{L}}$. In other words, every layer can appear at most once in the context.

```
void main(){
  Buffer c=new Buffer();
  activate(Untitled); //[] ↦ [Untitled]
  activate(Modified); //[Untitled] ↦ [Mod., Untitled]
  activate(Untitled); //[Mod., Untitled] ↦ [Untitled, Mod.]
  b.save();           //→Untitled.Buffer.save
```

Activating active layers

When an active layer is activated, it just moves to the head of the context $\overline{\text{L}}$. In other words, every layer can appear at most once in the context.

```
void main(){
  Buffer b=new Buffer();
  with(Untitled){         //[] ↦ [Untitled]
    with(Modified){       //[Untitled] ↦ [Mod., Untitled]
      with(Untitled){     //[Mod., Untitled] ↦ [Untitled, Mod.]
        b.save();         //→Untitled.Buffer.save
      }                   //[Untitled, Mod.] ↦ [Mod., Untitled]
      b.save();           //→Modified.Buffer.save
    }
  }
```

Qestion: mixed activation

```
void main(){
  Buffer b=new Buffer();
  with(Untitled){
    activate(Untitled);
  }
  b.save();      //→Buffer.save? Or Untitled.Buffer.save?
}
```

Qestion: mixed activation

```
void main(){
  Buffer b=new Buffer();
  activate(Untitled);
  activate(Modified);
  with(Untitled){
    b.save();//→Untitled.Buffer.save? Modified.Buffer.save?
    with(Modified){
      activate(Modified);
    }
  }
  b.save();  //→Untitled.Buffer.save? Modified.Buffer.save?
}
```

Our choice

```
void main(){
  Buffer b=new Buffer();
  with(Untitled){
    activate(Untitled);
  }
  b.save();       //→Untitled.Buffer.save because
}                 //there is no deactivate after activate
```

Our choice

```
void main(){
  Buffer b=new Buffer();
  activate(Untitled);
  activate(Modified);
  with(Untitled){
    b.save();//→Untitled.Buffer.save (most recently activated)
    with(Modified){
      activate(Modified);
    }
  }
  b.save();  //→Modified.Buffer.save (most recently activated)
}
```

Our approach: distributivity-based semantics[Uustalu'05]

- Computations depending on active layers are structured with a comonad
- Computations that (de)activate layers imperatively are structured with a monad
- Mixing block and imperative activation can be easily achieved via a distributive law of the comonad over the monad

Terms and values

```
data Tm = V Var                      (Variable)
        | S Sym                      (Function symbol)
        | L Var [(Layer, Tm)]        (Abstraction)
        | Tm :@ Tm                   (Application)
        | A Layer                    (Imperative activation)
        | D Layer                    (Imperative deactivation)
        | With Layer Tm              (Block activation)
        | Without Layer Tm           (Block deactivation)
        | Tm :@@ Tm                  (Proceeding application)

type Val d t = d (Val d t) → t (Val d t)
```

The type parameters `d` and `t` are functors

- `d a`: the type of values with two lists of active layers
- `t a`: the type of computations performing imperative layer (de)activation

Terms and values

```
data Tm = V Var                    (Variable)
        | S Sym                    (Function symbol)
        | L Var [(Layer, Tm)]      (Abstraction)
        | Tm :@ Tm                 (Application)
        | A Layer                  (Imperative activation)
        | D Layer                  (Imperative deactivation)
        | With Layer Tm            (Block activation)
        | Without Layer Tm         (Block deactivation)
        | Tm :@@ Tm                (Proceeding application)

type Val d t = d (Val d t) → t (Val d t)
```

Moreover,

- (`d`, `extract`, $\Rightarrow\!\!\!\gg$) is a comonad
- (`t`, `return`, $\gg\!=$) is a monad
- there is a distributive law of the comonad over the monad

Terms and values

We first show the comonadic semantics of the following subset of our language (imperative activation is omitted)

```
data Tm = V Var                    (Variable)
        | S Sym                    (Function symbol)
        | L Var [(Layer, Tm)]      (Abstraction)
        | Tm :@ Tm                 (Application)
        | A Layer                  (Imperative activation)
        | D Layer                  (Imperative deactivation)
        | With Layer Tm            (Block activation)
        | Without Layer Tm         (Block deactivation)
        | Tm :@@ Tm                (Proceeding application)

type Val d = d (Val d) → Val d
```

Comonad for COP

Comonads are represented as instances of the following type class:

```
class Comonad d where
  extract :: d a → a
  (=>>) :: d a → (d a → b) → d b
```

LS is the functor of our comonad:

```
data LS a = LS a [Layer] [Layer]
```

and we can define the comonad:

```
instance Comonad LS where
  extract (LS x _ _) = x
  x@(LS _ lsP lsF) =>> f = LS (f x) lsP lsF
```

Specifically, we have to check the three coherence conditions, but it is easy and trivial (our comonad is an environment comonad)

Comonadic interpretation

Comonadic semantics is given by the following function

```
type Val d = d (Val d) → Val d
type Env d = [(Var, Val d)] -- list of variable-value pairs
class Comonad d ⇒ ComonadEv d where
  ev :: Tm → d (Env d) → Val d
```

But for simplicity, we define `ev` as follows

```
ev :: Tm → LS (Env LS) → Val LS
```

Comonadic interpretation

Comonadic semantics is given by the following function

```
type Val d = d (Val d) → Val d
type Env d = [(Var, Val d)] -- list of variable-value pairs
ev :: Tm → LS (Env LS) → Val LS
```

`With` and `Without` just change the full list of active layers

```
ev (With ly t) (LS env lsP lsF) =
  ev t (LS env lsP (ly : removeL ly lsF))

ev (Without ly t) (LS env lsP lsF) =
  ev t (LS env lsP (removeL ly lsF))
```

where `removeL ly lst` removes `ly` from `lst` if exists

Comonadic interpretation

Comonadic semantics is given by the following function

```
type Val d = d (Val d) → Val d
type Env d = [(Var, Val d)] -- list of variable-value pairs
ev :: Tm → LS (Env LS) → Val LS
```

To interpret applications, we need to wrap the argument value with the context. We use ⇛ for this purpose.

```
ev (f :@ a) denv = let f’ = ev f denv
                       a’ = denv ⇛ ev a
                   in calling f’ a’
```

`calling` copies the full list of active layers to the partial list of active layers

```
calling :: (LS a → a) → LS a → a
calling f (LS x lsP lsF) = f (LS x lsF lsF)
```

Comonadic interpretation

Comonadic semantics is given by the following function

```
type Val d = d (Val d) → Val d
type Env d = [(Var, Val d)] -- list of variable-value pairs
ev :: Tm → LS (Env LS) → Val LS
```

Proceeding applications are interpreted in the same way to applications, but the partial list of active layers are reduced

```
ev (f :@@ a) denv = let f' = ev f denv
                        a' = denv =>> ev a
                    in proceeding f' a'

proceeding :: (LS a → a) → LS a → a
proceeding f (LS x lsP lsF) = f (LS x (tail lsP) lsF)
```

Comonadic interpretation

Comonadic semantics is given by the following function

```
type Val d = d (Val d) → Val d
type Env d = [(Var, Val d)] -- list of variable-value pairs
ev :: Tm → LS (Env LS) → Val LS
```

Abstractions are interpreted as comonadic functions, which select the body term w.r.t. the context of the argument cv

```
ev (L x body) denv = f where
   f :: LS (Val LS) → Val LS
   f cv = ev (dispatch cv body) (cmap repair (czip cv denv))
   repair (a, env) = update x a env
}
```

Distributivity-based interpretation

Now lets think an interpretation of the full language

```
data Tm = V Var                          (Variable)
        | S Sym                          (Function symbol)
        | L Var [(Layer, Tm)]            (Abstraction)
        | Tm :@ Tm                       (Application)
        | A Layer                        (Imperative activation)
        | D Layer                        (Imperative deactivation)
        | With Layer Tm                  (Block activation)
        | Without Layer Tm               (Block deactivation)
        | Tm :@@ Tm                      (Proceeding application)

type Val d t = d (Val d t) → t (Val d t)
```

Monad for COP

Monads are represented as instances of the following type class:

```
class Monad t where
  return :: a → t a
  (>>=) :: t a → (a → t b) → t b
```

`AE` is the functor of our monad:

```
data Ev = Acti | Deacti
type Eff = [(Layer, Ev)]
data AE a = AE a Eff
```

and the monad for COP is defined as:

```
instance Monad AE where
  return x = AE x []
  (AE x eff) >>= f = let (AE y eff') = f x
                     in AE y (merge eff' eff)
```

Distributive law of the comonad over the monad

The Distributive laws of comonads over monads[Brookes92, Power02] allow us to put the effects represented by monads to the contexts represented by comonads

In Haskell, the distributive combination is implemented as follows:

```
class (Comonad d, Monad t) ⇒ Dist d t where
  dist :: d (t a) → t (d a)
```

For example, for our case:

```
instance Dist LS AE where
  dist (LS (AE v eff) lsP lsF) =
    AE (LS v lsP (eApp eff lsF)) eff
```

where `eApp` applies the effect to the list of active layers

Distributivity-based interpretation

Distributivity-based interpretation is given by

```
type Val d t = d (Val d t) → t (Val d t)
type Env d t = [(Var, Val d t)] -- list of variable-value pairs
ev :: Tm → LS (Env LS AE) → AE (Val LS AE)
```

Imperative activation and deactivation just generate layer activation and deactivation events respectively

```
ev (A ly) denv = AE u [(ly, Acti)]
ev (D ly) denv = AE u [(ly, Deacti)]
```

where `u` is some function, e.g.,

```
u x = return (extract x)
```

Distributivity-based interpretation

Distributivity-based interpretation is given by

```
type Val d t = d (Val d t) → t (Val d t)
type Env d t = [(Var, Val d t)] -- list of variable-value pairs
ev :: Tm → LS (Env LS AE) → AE (Val LS AE)
```

Another interesting case is applications

```
ev (f :@ a) denv =
  do f' ← ev f denv
     a' ← dist (denv ⋙ ev a)
     calling f' a'
```

This does not work because the effects of imperative activation during the evaluation of `f` is ignored by the evaluation of `a`

Distributivity-based interpretation

Distributivity-based interpretation is given by

```
type Val d t = d (Val d t) → t (Val d t)
type Env d t = [(Var, Val d t)] -- list of variable-value pairs
ev :: Tm → LS (Env LS AE) → AE (Val LS AE)
```

One way to propagate the effect of imperative activation in `f` is to use the distributive law not only at the evaluation of `a` but also `f`

```
ev (f :@ a) denv =
  do f’ ← dist (denv =>> ev f)
     x’ ← dist (czip f’ denv =>> snd ∘ extract =>> ev a)
     calling (extract f’) x’
```

`f’` has the full list of active layers that reflect the imperative activation during the evaluation of `f`

Related work

Formal studies on COP languages [Clarke09, Igarashi11, Igarashi12, Kamina14, Inoue15]

- Only one activation mechanism is considered, or the two activation are managed separately
- Most of the studies are focused on "type safety"

Distributivity-based semantics for dataflow programming [Uustalu05]

- Our work is largely inspired by the work

Conclusions and future work

We showed

- Comonadic interpretation for the COP languages that support only block activation
- Distributivity-based interprentation for the languages that support both block and imperative activation

Future work includes

- Concurrent context changes
- Type system, verification
- Re-implementing ServalCJ[Kamina15] compiler based on the semantics

Thank you

```
void main(){
  Buffer b=new Buffer();
  activate(Untitled);
  activate(Modified);
  with(Untitled){
    b.save();//→Untitled.Buffer.save? Modified.Buffer.save?
    with(Modified){
      activate(Modified);
    }
  }
  b.save();  //→Untitled.Buffer.save? Modified.Buffer.save?
}
```