Forward Chaining in HALO
An Implementation Strategy for History-based Logic Pointcuts

@International Conference On Dynamic Languages 2007,
Lugano, August 27th, 2007

Charlotte Herzeel, Kris Gybels, Pascal Costanza, Coen De Roover and Theo D’hondt
Programming Technology Laboratory
Vrije Universiteit Brussel
Overview

• Introduction
  • Logic Meta Programming & Aspect-Oriented Programming
  • Running example - going shopping

• History-based Aspects using LOGic:
  • Aspect-Oriented Programming for Lisp
  • meta model = the program execution
  • hybrid language (temporal logic programming/CLOS)

• Implementing HALO
  • HALO weaver Architecture
  • Query Engine based on RETE
  • Reducing Memory Overhead
Scattering & tangling
Scattering & tangling

Tyranny of the dominant decomposition
Scattering & tangling

Tyranny of the dominant decomposition

- tangling & scattering
- maintainability
- reusability
- readability
Scattering & tangling

Tyranny of the dominant decomposition

- tangling & scattering
- maintainability
- reusability
- readability

Bank

| register(Client) |
| createAccount(Client) |
| shareAccount(Client, Account) |
| login(Client) |
| logout(Client) |

1

Client

| withdraw(AccountNr, Integer) |
| credit(AccountNr, Integer) |
| view(AccountNr) |

* * *

Account

| withdraw(Client, Integer) |
| credit(Client, Integer) |
| transfer(Client, Account) |
| view(Client) |
Scattering & tangling

Tyranny of the dominant decomposition

- tangling & scattering
- maintainability
- reusability
- readability

Bank

- register(Client)
- createAccount(Client)
- shareAccount(Client, Account)
- login(Client)
- logout(Client)

Client

- withdraw(AccountNr, Integer)
- credit(AccountNr, Integer)
- view(AccountNr)

Account

- withdraw(Client, Integer)
- credit(Client, Integer)
- transfer(Client, Account)
- view(Client)
Scattering & tangling

Tyranny of the dominant decomposition
- tangling & scattering
- maintainability
- reusability
- readability

Bank
- register(Client)
- createAccount(Client)
- shareAccount(Client, Account)
- login(Client)
- logout(Client)

Client
- withdraw(AccountNr, Integer)
- credit(AccountNr, Integer)
- view(AccountNr)

Account
- withdraw(Client, Integer)
- credit(Client, Integer)
- transfer(Client, Account)
- view(Client)
Scattering & tangling

Tyranny of the dominant decomposition
- tangling & scattering
- maintainability
- reusability
- readability

CLOS: before, after, around

log sensitive operations = crosscutting concern

Bank
- register(Client)
- createAccount(Client)
- shareAccount(Client, Account)
- login(Client)
- logout(Client)

Client
- withdraw(AccountNr, Integer)
- credit(AccountNr, Integer)
- view(AccountNr)

Account
- withdraw(Client, Integer)
- credit(Client, Integer)
- transfer(Client, Account)
- view(Client)
Scattering & tangling

Tyranny of the dominant decomposition
  - tangling & scattering
  - maintainability
  - reusability
  - readability

log sensitive operations = crosscutting concern

CLOS: before, after, around

(defmethod shareAccount :around ((c client), (a account))
  (log "sharing account ~s ~s" c a)
  (call-next-method))

(defmethod withdraw :around ((c client), (i integer))
  ...
)

(defmethod credit :around ((c client), (i integer))
  ...
)
Scattering & tangling

log sensitive operations = crosscutting concern

Tyranny of the dominant decomposition

tangling & scattering
- maintainability
- reusability
- readability

CLOS: before, after, around

Bank

register(Client)
createAccount(Client)
shareAccount(Client, Account)
login(Client)
logout(Client)

1

Client

withdraw(AccountNr, Integer)
credit(AccountNr, Integer)
view(AccountNr)

* * *

Account

withdraw(Client, Integer)
credit(Client, Integer)
transfer(Client, Account)
view(Client)
Scattering & tangling

Tyranny of the dominant decomposition
- tangling & scattering
- maintainability
- reusability
- readability

CLOS: before, after, around

Aspect-Oriented Programming

log sensitive operations = crosscutting concern

Bank
- register(Client)
- createAccount(Client)
- shareAccount(Client, Account)
- login(Client)
- logout(Client)

Client
- withdraw(AccountNr, Integer)
- credit(AccountNr, Integer)
- view(AccountNr)

Account
- withdraw(Client, Integer)
- credit(Client, Integer)
- transfer(Client, Account)
- view(Client)
Tyranny of the dominant decomposition

tangling & scattering
- maintainability
- reusability
- readability

Aspect-Oriented Programming

log sensitive operations = crosscutting concern

CLOS: before, after, around

Bank

register(Client)
createAccount(Client)
shareAccount(Client, Account)
login(Client)
logout(Client)

Client

withdraw(AccountNr, Integer)
credit(AccountNr, Integer)
view(AccountNr)

Account

withdraw(Client, Integer)
credit(Client, Integer)
transfer(Client, Account)
view(Client)
Scattering & tangling

Tyranny of the dominant decomposition
- tangling & scattering
- maintainability
- reusability
- readability

CLOS: before, after, around

log sensitive operations = crosscutting concern

Aspect-Oriented Programming

Bank
- register(Client)
- createAccount(Client)
- shareAccount(Client, Account)
- login(Client)
- logout(Client)

Client
- withdraw(AccountNr, Integer)
- credit(AccountNr, Integer)
- view(AccountNr)

Account
- withdraw(Client, Integer)
- credit(Client, Integer)
- transfer(Client, Account)
- view(Client)

"at each sensitive method call,"
Scattering & tangling

Tyranny of the dominant decomposition

- tangling & scattering
  - maintainability
  - reusability
  - readability

CLOS: before, after, around

Aspect-Oriented Programming

Bank

- register(Client)
- createAccount(Client)
- shareAccount(Client, Account)
- login(Client)
- logout(Client)

Client

- withdraw(AccountNr, Integer)
- credit(AccountNr, Integer)
- view(AccountNr)

Account

- withdraw(Client, Integer)
- credit(Client, Integer)
- transfer(Client, Account)
- view(Client)

Aspect

"at each sensitive method call, make a log entry"

log sensitive operations = crosscutting concern
Scattering & tangling

Tyranny of the dominant decomposition
- tangling & scattering
- maintainability
- reusability
- readability

log sensitive operations = crosscutting concern

Aspect-Oriented Programming

CLOS: before, after, around

Aspect

Bank

- register(Client)
- createAccount(Client)
- shareAccount(Client, Account)
- login(Client)
- logout(Client)

Client

- withdraw(AccountNr, Integer)
- credit(AccountNr, Integer)
- view(AccountNr)

Account

- withdraw(Client, Integer)
- credit(Client, Integer)
- transfer(Client, Account)
- view(Client)

pointcut

"at each sensitive method call, make a log entry"
Tyranny of the dominant decomposition

- tangling & scattering
- maintainability
- reusability
- readability

CLOS: before, after, around

Aspect-Oriented Programming

log sensitive operations = crosscutting concern

tangling & scattering -- maintainability -- reusability -- readability

Bank
- register(Client)
- createAccount(Client)
- shareAccount(Client, Account)
- login(Client)
- logout(Client)

Client
- withdraw(AccountNr, Integer)
- credit(AccountNr, Integer)
- view(AccountNr)

Account
- withdraw(Client, Integer)
- credit(Client, Integer)
- transfer(Client, Account)
- view(Client)

Aspect
- pointcut
  "at each sensitive method call, make a log entry"
- advice

pointcut
"at each sensitive method call, make a log entry"
Tyranny of the dominant decomposition
  tangling & scattering
  - maintainability
  - reusability
  - readability

CLOS: before, after, around

Aspect-Oriented Programming

log sensitive operations = crosscutting concern

Aspect
  pointcut
    “at each sensitive method call, make a log entry”
  advice

Account
  withdraw(Client, Integer)
  credit(Client, Integer)
  transfer(Client, Account)
  view(Client)
Scattering & tangling

Tyranny of the dominant decomposition
- tangling & scattering
  - maintainability
  - reusability
  - readability

CLOS: before, after, around

Aspect-Oriented Programming

log sensitive operations = crosscutting concern

<table>
<thead>
<tr>
<th>Bank</th>
</tr>
</thead>
<tbody>
<tr>
<td>register(Client)</td>
</tr>
<tr>
<td>createAccount(Client)</td>
</tr>
<tr>
<td>shareAccount(Client, Account)</td>
</tr>
<tr>
<td>login(Client)</td>
</tr>
<tr>
<td>logout(Client)</td>
</tr>
</tbody>
</table>

Client

withdraw(AccountNr, Integer)
credit(AccountNr, Integer)
view(AccountNr)

Account

withdraw(Client, Integer)
credit(Client, Integer)
transfer(Client, Account)
view(Client)

Aspect

pointcut
"at each sensitive method call, make a log entry"

advice
Tyranny of the dominant decomposition

- tangling & scattering
- maintainability
- reusability
- readability

CLOS: before, after, around

Aspect-Oriented Programming

log sensitive operations = crosscutting concern

Aspect

Bank

register(Client)
createAccount(Client)
shareAccount(Client, Account)
login(Client)
logout(Client)

Client

withdraw(AccountNr, Integer)
credit(AccountNr, Integer)
view(AccountNr)

Account

withdraw(Client, Integer)
credit(Client, Integer)
transfer(Client, Account)
view(Client)

pointcut

"at each sensitive method call, make a log entry"

advice
Scattering & tangling

Tyranny of the dominant decomposition
- tangling & scattering
  - maintainability
  - reusability
  - readability

CLOS: before, after, around
- current join point only

Aspect-Oriented Programming

Bank
- register(Client)
- createAccount(Client)
- shareAccount(Client, Account)
- login(Client)
- logout(Client)

Client
- withdraw(AccountNr, Integer)
- credit(AccountNr, Integer)
- view(AccountNr)

Account
- withdraw(Client, Integer)
- credit(Client, Integer)
- transfer(Client, Account)
- view(Client)

pointcut
- "at each sensitive method call, make a log entry"

Aspect
- advice

log sensitive operations = crosscutting concern

Aspect-Oriented Programming

macros

- current join point only
Scattering & tangling

Tyranny of the dominant decomposition
tangling & scattering
- maintainability
- reusability
- readability

CLOS: before, after, around
?macros
- current join point only
! history of join points

Aspect-Oriented Programming

log sensitive operations = crosscutting concern

Aspect

Bank

register(Client)
createAccount(Client)
shareAccount(Client, Account)
login(Client)
logout(Client)

Client

withdraw(AccountNr, Integer)
credit(AccountNr, Integer)
view(AccountNr)

Account

withdraw(Client, Integer)
credit(Client, Integer)
transfer(Client, Account)
view(Client)

pointcut
"at each sensitive method call,
make a log entry"

advice

make a log entry"
Declarative Meta Programming & Aspect-Oriented Programming
Declarative Meta Programming & Aspect-Oriented Programming

(base base)
Declarative Meta Programming & Aspect-Oriented Programming

(meta (base base))
Declarative Meta Programming & Aspect-Oriented Programming

MP

(meta [base base])

meta language object language meta model
Declarative Meta Programming & Aspect-Oriented Programming

(meta [base base])

MP
- meta language
- object language
- meta model

AOP
- pointcut language
- base language
- jp model
Declarative Meta Programming & Aspect-Oriented Programming

SOUL for DMP:
Base code to facts -> Image querying
Symbiosis (quasiquoting)
Rule abstraction (Libraries)
Generating code
Detecting/Verifying coding patterns
SOUL for DMP:
Base code to facts -> Image querying
Symbiosis [quasiquoting]
Rule abstraction [Libraries]
Generating code

**Detecting/Verifying coding patterns**

```lisp
(defclass account () ((balance)))
(defclass client () ((id)))
(defmethod withdraw ((a account) nr)
  (setf (balance a) (- (balance a) nr)))
(defmethod credit ((a account) nr)
  ...)
```
Declarative Meta Programming & Aspect-Oriented Programming

SOUL for DMP:
Base code to facts -> Image querying
Symbiosis [quasiquoting]
Rule abstraction [Libraries]
Generating code
**Detecting/Verifying coding patterns**

```lisp
(defclass account () ((balance}))
(defclass client () ((id)))
(defmethod withdraw ((a account) nr)
  (setf (balance a) (- (balance a) nr)))
(defmethod credit ((a account) nr)
  ...)
```
SOUL for DMP:
Base code to facts -> Image querying
Symbiosis (quasiquoting)
Rule abstraction (Libraries)
Generating code
Detecting/Verifying coding patterns

(defclass account () ((balance))
(defclass client () ((id)))
(defmethod withdraw ((a account) nr)
 (setf (balance a) (- (balance a) nr)))
(defmethod credit ((a account) nr)
 ...)

class(‘account , <account>)
classImplementsMethodNamed
(<account> , ‘withdraw, <withdraw>)
method(<account>, <withdraw>)
methodName(<withdraw>, ‘withdraw)
methodArguments(<withdraw>, ()
methodStatements(<withdraw>, ((setf
(balance a) (- (balance a) nr))))
...

(MP meta language object language meta model)
(AOP pointcut language base language jp model)
Declarative Meta Programming & Aspect-Oriented Programming

SOUL for DMP:
Base code to facts -> Image querying
Symbiosis [quasiquoting]
Rule abstraction [Libraries]
Generating code
Detecting/Verifying coding patterns

abstractClassHeuristic() if
forall(abstractClass(?C), baseClass(?C)),
forall(baseClass(?C), abstractClass(?C)).

(defclass account () ((balance)))
(defclass client () ((id)))
(defmethod withdraw ((a account) nr)
  (setf (balance a) (- (balance a) nr)))
(defmethod credit ((a account) nr)
  ...)

class('account', <account>)
classImplementsMethodNamed
  (<account>, 'withdraw', <withdraw>)
method(<account>, <withdraw>)
methodName(<withdraw>, 'withdraw')
methodArguments(<withdraw>, (),
methodStatements(<withdraw>, ((setf (balance a) (- (balance a) nr))))
  ...
SOUL for DMP:
Base code to facts -> Image querying
Symbiosis (quasiquoting)
Rule abstraction (Libraries)
Generating code
Detecting/Verifying coding patterns

(defclass account () ((balance)))
(defclass client () ((id)))
(defmethod withdraw ((a account) nr)
  (setf (balance a) (- (balance a) nr)))
(defmethod credit ((a account) nr)
  ...)

class(‘account, <account>)
classImplementsMethodNamed
(<account>, ‘withdraw, <withdraw>)
method(<account>, <withdraw>)
methodName(<withdraw>, ‘withdraw)
methodArguments(<withdraw>, (())
methodStatements(<withdraw>, ((setf
  (balance a) (- (balance a) nr)))))
...
Declarative Meta Programming & Aspect-Oriented Programming

MP: meta language, object language, meta model
AOP: pointcut language, base language, pj model
Declarative Meta Programming & Aspect-Oriented Programming

CARMA: AOP with a SOUL
modularizing scattered & tangled code
Join point type predicates
Link to shadows
Symbiosis
Loose coupling aspect/base

Pattern-based pointcuts
Declarative Meta Programming & Aspect-Oriented Programming

**Diagram:**
- **MP (Meta Programming):**
  - Meta language
  - Object language
  - Meta model

- **AOP (Aspect-Oriented Programming):**
  - Pointcut language
  - Base language
  - Jp model

**CARMA: AOP with a SOUL**
- Modularizing scattered & tangled code
- Join point type predicates
- Link to shadows
- Symbiosis
- Loose coupling aspect/base

**Pattern-based pointcuts**

- (defclass account () ...)
- (defmethod withdraw ((a account) nr)
  (setf (balance a) (- (balance a) nr)))
- (defmethod credit ((a account) nr) ...)
- (withdraw *account* 3)
- (print (get-value *account* ‘balance))
CARMA: AOP with a SOUL
modularizing scattered & tangled code
Join point type predicates
Link to shadows
Symbiosis
Loose coupling aspect/base

Pattern-based pointcuts

(defclass account () ...)  
(defmethod withdraw ((a account) nr)  
  (setf (balance a) (- (balance a) nr)))  
(defmethod credit ((a account) nr) ...)  
(withdraw *account* 3)  
(print (get-value *account* 'balance))
CARMA: AOP with a SOUL
modularizing scattered & tangled code
Join point type predicates
Link to shadows
Symbiosis
Loose coupling aspect/base
Pattern-based pointcuts

(defclass account () ...)
(defmethod withdraw ((a account) nr)
  (setf (balance a) (- (balance a) nr)))
(defmethod credit ((a account) nr) ...)
(withdraw *account* 3)
(print (get-value *account* ‘balance))

class(‘account, <account>)
classImplementsMethodNamed(<<account>, ‘withdraw, <withdraw>)
1. send(<jp>, withdraw, [<account>])
   assignment(<jp>, balance, 10, 7)
2. reference(<jp2>, balance, 7)
Declarative Meta Programming & Aspect-Oriented Programming

CARMA: AOP with a SOUL
modularizing scattered & tangled code
Join point type predicates
Link to shadows
Symbiosis
Loose coupling aspect/base
**Pattern-based pointcuts**

carma pointcut

```
(defclass account () ...)  
(defmethod withdraw ((a account) nr)  
  (setf (balance a) (- (balance a) nr)))  
(defmethod credit ((a account) nr) ...)  
(withdraw *account* 3)  
(print (get-value *account* ‘balance))
```

```
class( ‘account , <account>)
classImplementsMethodNamed  
  (<account> , ‘withdraw, <withdraw>)  
1. send(<jp>, withdraw, [<account>])  
  assignment(<jp>, balance, 10, 7)  
2. reference(<jp2>, balance, 7)
```
Declarative Meta Programming & Aspect-Oriented Programming

MP
- meta language
- object language
- meta model

AOP
- pointcut language
- base language
- jp model

CARMA: AOP with a SOUL
- modularizing scattered & tangled code
- Join point type predicates
- Link to shadows
- Symbiosis
- Loose coupling aspect/base

Pattern-based pointcuts

carma pointcut

rule abstraction, unification, recursion

(defclass account () ...)  
(defmethod withdraw ((a account) nr)  
  (setf (balance a) (- (balance a) nr)))
(defmethod credit ((a account) nr) ...)  
(withdraw *account* 3)
(print (get-value *account* ‘balance))

class( ‘account , <account>)
classImplementsMethodNamed  
(<<account> , ‘withdraw, <withdraw>>)  
1. send(<jp>, withdraw, [<account>])
   assignment(<jp>, balance, 10, 7)
2. reference(<jp2>, balance, 7)
Declarative Meta Programming & Aspect-Oriented Programming

**Meta Model**
- **Meta Language**
- **Object Language**
- **Base Language**
- **JP Model**

**CARMA: AOP with a SOUL**
- Modularizing scattered & tangled code
- Join point type predicates
- Link to shadows
- Symbiosis
- Loose coupling aspect/base

**Pattern-based pointcuts**
- CARMA pointcut

**Rule abstraction, unification, recursion**
- Reusability, reasoning

**Example Code**
```
(defclass account () ...)
(defmethod withdraw ((a account) nr)
  (setf (balance a) (- (balance a) nr)))
(defmethod credit ((a account) nr) ...)
(withdraw *account* 3)
(print (get-value *account* ‘balance))
```

**Example 1**
1. send(<jp>, withdraw, [<account>])
2. reference(<jp2>, balance, 7)
Declarative Meta Programming & Aspect-Oriented Programming

MP: meta language → object language → meta model
AOP: pointcut language → base language → jp model

(defclass account () ...)
(defmethod withdraw ((a account) nr)
  (setf (balance a) (- (balance a) nr)))
(defmethod credit ((a account) nr) ...)
(withdraw *account* 3)
(print (get-value *account* ‘balance))

class( ‘account , <account>)
classImplementsMethodNamed
  (<account> , ‘withdraw, <withdraw>)
1. send(<jp>, withdraw, [<account>])
   assignment(<jp>, balance, 10, 7)
2. reference(<jp2>, balance, 7)

rule abstraction, unification, recursion
reusability, reasoning

[Model Diagram]
HALO: AOP with a memory
Temporal/Stateful pointcuts

rule abstraction, unification, recursion
reusability, reasoning

(defclass account () ...)
(defmethod withdraw ((a account) nr)
  (setf (balance a) (- (balance a) nr)))
(defmethod credit ((a account) nr) ...)
(withdraw *account* 3)
(print (get-value *account* ‘balance))

class( ‘account , <account>)
classImplementsMethodNamed
  (<account> , ‘withdraw, <withdraw>)
  1. send(<jp>, withdraw, [<account>])
  2. reference(<jp2>, balance, 7)
Declarative Meta Programming & Aspect-Oriented Programming

HALO: AOP with a memory
Temporal/Stateful pointcuts

rule abstraction, unification, recursion
reusability, reasoning

(defclass account () ...)
(defmethod withdraw ((a account) nr)
  (setf (balance a) (- (balance a) nr)))
(defmethod credit ((a account) nr) ...)
(withdraw *account* 3)
(print (get-value *account* ‘balance))

class(‘account, <account>) record
classImplementsMethodNamed
  (<account>, ‘withdraw, <withdraw>)
1. send(<jp>, withdraw, [<account>])
assignment(<jp>, balance, 10, 7)
2. reference(<jp2>, balance, 7)
Running example
Running example
Running example
Running example
Running example
Running example
Running example
Running example
Running example
Running example
Running example
Running example
Running example
Running example
Running example

- Shop
- Article
- User
  - login(shop, name, pwd)
  - buy(shop, name, pwd)
  - checkout()
- Basket
- Promotions
  - current-rate(article)
  - set-rate(article, rate)
Running example

```
Shop
1
User
login(shop,name,pwd)
buy(shop,name,pwd)
checkout()

Article
*

Basket
1
1

Promotions
1
current-rate(article)
set-rate(article, rate)
```
Running example

- **Shop**
  - 1
  - *
  - 1
  - *

- **User**
  - login(shop,name,pwd)
  - buy(shop,name,pwd)
  - checkout()

- **Article**
  - *
  - *
  - *

- **Basket**
  - 1
  - 1

- **Promotions**
  - current-rate(article)
  - set-rate(article, rate)

⚠️ Stateful!
Mental Model

(defclass user () ((name)))
(defmethod checkout ((u user)) ...) 
(defmethod buy ((u user) (a article)) ...) 

fact generator

base

(jp ctr)

(jp facts)

TN: (gf-call checkout <user>)
TN-1: (gf-call buy <user> <cd>)

(temporal relations)

most-recent, all-past, since, cflow

rules

(defrule (buy ?u)
  (gf-call buy ?u ?a))

(aspect)

(at
  ((gf-call checkout ?u)
    (most-recent (buy ?u ?a)))
  (log “user ~s bought ~s” ?u ?a))

(query engine)
Join Point Model

CLOS

(make-instance 'user)
(get-value <kris> 'name)
(setf (get-value <kris> 'name) "kris")
(checkout <kris>)

Join Point Facts

(create 'user <kris>)
(slot-get <kris> 'name "kris")
(slot-set <kris> 'name "" "kris")
(gf-call checkout <kris>)
(end-gf-call checkout <kris> nil)
Join Point Model

CLOS

(make-instance 'user)
(get-value <kris> 'name)
(setf (get-value <kris> 'name) "kris")
(checkout <kris>)

Join Point Type Predicates

(create 'user ?obj)
(slot-get ?obj ?name ?val)
(slot-set ?obj ?name 'oval ?val)
(gf-call ?gf-name ?obj)
(end-gf-call ?gf-name ?obj ?res)
Join Point Model

CLOS

(make-instance 'user)

(get-value <kris> 'name)

(setf (get-value <kris> 'name) "kris")

(checkout <kris>)

Join Point Type Predicates

(create 'user ?obj)

(slot-get ?obj ?name ?val)

(slot-set ?obj ?name 'oval?val)

(gf-call ?gf-name ?obj)

(end-gf-call ?gf-name ?obj ?res)

Extensible ...

(defrule [promotion "25-12-2008")]
Join Point Model

CLOS

(make-instance 'user)
(get-value <kris> 'name)
(setf (get-value <kris> 'name) "kris")
(checkout <kris>)

Join Point Type Predicates

(create 'user ?obj )
(slot-get ?obj ?name ?val )
(slot-set ?obj ?name 'oval?val )
(gf-call ?gf-name ?obj )
(end-gf-call ?gf-name ?obj ?res)

Extensible ...
(defrule [promotion "25-12-2008"])

Hybrid Pointcuts

(escape ?name (password ?user))
[defrule [date ?date] [escape ?date [get-current-time]]]
Join Point Model

CLOS

(make-instance ‘user)
(get-value <kris> ‘name)
(setf (get-value <kris> ‘name) “kris”)
(checkout <kris>)

Join Point Type Predicates

(create ‘user ?obj)
(slot-get ?obj ?name ?val)
(slot-set ?obj ?name ’oval ?val)
(gf-call ?gf-name ?obj)
(end-gf-call ?gf-name ?obj ?res)

Extensible...

[defrule (promotion “25-12-2008”)]

Hybrid Pointcuts

(escape ?name (password ?user))
[defrule (date ?date) [escape ?date [get-current-time]]]
Join Point Model

CLOS

(make-instance 'user)
(get-value <kris> 'name)
(setf (get-value <kris> 'name) "kris")
(checkout <kris>)

Join Point Type Predicates

(create 'user ?obj)
(slot-get ?obj ?name ?val)
(slot-set ?obj ?name 'oval?val)
(gf-call ?gf-name ?obj)
(end-gf-call ?gf-name ?obj ?res)

Extensible ...
(defrule (promotion "25-12-2008")

Hybrid Pointcuts

(escape ?name (password ?user))
(defrule (date ?date) [escape ?date [get-current-time]]])
(login <shop> <kris> “kris” “kros” )
(buy <kris> <jacket>)
(logout <kris>)
(login <shop> <kris> “kris” “kros” )
(buy <kris> <t-shirt>)
(buy <kris> <socks>)
(checkout <kris>)
Temporal Matching

```prolog
(login <shop> <kris> "kris" "kros")
(buy <kris> <jacket>)
(logout <kris>)
(login <shop> <kris> "kris" "kros")
(buy <kris> <t-shirt>)
(buy <kris> <socks>)
(checkout <kris>)
```
Temporal Matching

1. (login <shop> <kris> “kris” “kros”) (buy <kris> <jacket>) (logout <kris>) (login <shop> <kris> “kris” “kros”) (buy <kris> <t-shirt>) (buy <kris> <socks>) (checkout <kris>)
Temporal Matching

1. (login <shop> <kris> “kris” “kros” )
   (buy <kris> <jacket>)
   (logout <kris>)
   (login <shop> <kris> “kris” “kros” )
   (buy <kris> <t-shirt>)
   (buy <kris> <socks>)
   (checkout <kris>)

login

T_{customer}
Temporal Matching

1. (login <shop> <kris> “kris” “kros” )
2. (buy <kris> <jacket>)
   (logout <kris>)
   (login <shop> <kris> “kris” “kros” )
   (buy <kris> <t-shirt>)
   (buy <kris> <socks>)
   (checkout <kris>)
Temporal Matching

1. (login <shop> <kris> “kris” “kros”)  
2. (buy <kris> <jacket>)  
   (logout <kris>)  
   (login <shop> <kris> “kris” “kros”)  
   (buy <kris> <t-shirt>)  
   (buy <kris> <socks>)  
   (checkout <kris>)

login  buy

T_{customer}
Temporal Matching

1. (login <shop> <kris> “kris” “kros” )
2. (buy <kris> <jacket> )
3. (logout <kris> )
   (login <shop> <kris> “kris” “kros” )
   (buy <kris> <t-shirt> )
   (buy <kris> <socks> )
   (checkout <kris> )

login  buy

T_{customer}
Temporal Matching

1. (login <shop> <kris> “kris” “kros” )
2. (buy <kris> <jacket> )
3. (logout <kris> )
   (login <shop> <kris> “kris” “kros” )
   (buy <kris> <t-shirt> )
   (buy <kris> <socks> )
   (checkout <kris> )

login   buy   logout

T_{customer}
Temporal Matching

1. (login <shop> <kris> “kris” “kros” )
2. (buy <kris> <jacket> )
3. (logout <kris> )
4. (login <shop> <kris> “kris” “kros” )
   (buy <kris> <t-shirt> )
   (buy <kris> <socks> )
   (checkout <kris> )

login buy logout
Temporal Matching

1. \((\text{login <shop> <kris> “kris” “kros” })\)
2. \((\text{buy <kris> <jacket>})\)
3. \((\text{logout <kris>})\)
4. \((\text{login <shop> <kris> “kris” “kros” })\)
   \((\text{buy <kris> <t-shirt>})\)
   \((\text{buy <kris> <socks>})\)
   \((\text{checkout <kris>})\)

\text{login} \quad \text{buy} \quad \text{logout} \quad \text{login}
Temporal Matching

1. (login <shop> <kris> “kris” “kros” )
2. (buy <kris> <jacket> )
3. (logout <kris>)
4. (login <shop> <kris> “kris” “kros” )
5. (buy <kris> <t-shirt> )
   (buy <kris> <socks>)
   (checkout <kris>)
Temporal Matching

1. \((\text{login } <\text{shop}> \ <\text{kris}> \ \text{“kris” “kros” })\)
2. \((\text{buy } <\text{kris}> \ <\text{jacket}>))\)
3. \((\text{logout } <\text{kris}>))\)
4. \((\text{login } <\text{shop}> \ <\text{kris}> \ \text{“kris” “kros” })\)
5. \((\text{buy } <\text{kris}> \ <\text{t-shirt}>))\)
   \((\text{buy } <\text{kris}> \ <\text{socks}>))\)
   \((\text{checkout } <\text{kris}>))\)

\(\text{login \ buy \ logout \ login \ buy}\)

\(T_{\text{customer}}\)
Temporal Matching

1. \((\text{login } \langle \text{shop} \rangle \ <\text{kris} \rangle \ "\text{kris}" \ "\text{kros}"
   \))
2. \((\text{buy } \langle \text{kris} \rangle \ <\text{jacket}\rangle)\)
3. \((\text{logout } \langle \text{kris} \rangle)\)
4. \((\text{login } \langle \text{shop} \rangle \ <\text{kris} \rangle \ "\text{kris}" \ "\text{kros}"
   \))
5. \((\text{buy } \langle \text{kris} \rangle \ <\text{t-shirt}\rangle)\)
6. \((\text{buy } \langle \text{kris} \rangle \ <\text{socks}\rangle)\)
   \(\text{ (checkout } \langle \text{kris} \rangle)\)

\text{login} \quad \text{buy} \quad \text{logout} \quad \text{login} \quad \text{buy}
Temporal Matching

1. (login <shop> <kris> “kris” “kros” )
2. (buy <kris> <jacket> )
3. (logout <kris> )
4. (login <shop> <kris> “kris” “kros” )
5. (buy <kris> <t-shirt> )
6. (buy <kris> <socks> )
   (checkout <kris> )
Temporal Matching

1. (login <shop> <kris> “kris” “kros” )
2. (buy <kris> <jacket> )
3. (logout <kris> )
4. (login <shop> <kris> “kris” “kros” )
5. (buy <kris> <t-shirt> )
6. (buy <kris> <socks> )
7. (checkout <kris> )
Temporal Matching

1. (login <shop> <kris> “kris” “kros” )
2. (buy <kris> <jacket> )
3. (logout <kris> )
4. (login <shop> <kris> “kris” “kros” )
5. (buy <kris> <t-shirt> )
6. (buy <kris> <socks> )
7. (checkout <kris> )

login  buy  logout  login  buy  buy  checkout

$T_{\text{customer}}$
Temporal Matching

1. (login <shop> <kris> “kris” “kros” )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> “kris” “kros” )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)

login  buy  logout  login  buy  buy  checkout

((gf-call checkout ?args)
 (most-recent (gf-call buy ?args2)))
Temporal Matching

1. (login <shop> <kris> “kris” “kros” )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> “kris” “kros” )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)

login buy logout login buy buy checkout

T \text{customer}

\begin{align*}
\text{(gf-call checkout ?args)} \\
\quad \text{(most-recent (gf-call buy ?args2))}
\end{align*}
Temporal Matching

1. (login <shop> <kris> “kris” “kros” )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> “kris” “kros” )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)

T_{customer}

((gf-call checkout ?args)
(most-recent (gf-call buy ?args2)))
Temporal Matching

1. (login <shop> <kris> “kris” “kros”)  
2. (buy <kris> <jacket>)  
3. (logout <kris>)  
4. (login <shop> <kris> “kris” “kros”)  
5. (buy <kris> <t-shirt>)  
6. (buy <kris> <socks>)  
7. (checkout <kris>)

login buy logout login buy buy checkout

T_{customer}

(((gf-call checkout ?args) 
  (most-recent (gf-call buy ?args2))))

login buy logout login buy buy checkout

T_{customer}
Temporal Matching

1. (login <shop> <kris> “kris” “kros”)
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> “kris” “kros”)
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)

login  buy  logout  login  buy  buy  checkout

((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2)))

login  buy  logout  login  buy  buy  checkout

((gf-call checkout ?args)
  (all-past (gf-call buy ?args2)))
Temporal Matching

1. (login <shop> <kris> “kris” “kros”)  
2. (buy <kris> <jacket>)  
3. (logout <kris>)  
4. (login <shop> <kris> “kris” “kros”)  
5. (buy <kris> <t-shirt>)  
6. (buy <kris> <socks>)  
7. (checkout <kris>)

```
login  buy  logout  login  buy  buy  checkout
```

```
T_{customer}
```

```
((gf-call checkout ?args)  
(most-recent (gf-call buy ?args2)))
```

```
login  buy  logout  login  buy  buy  checkout
```

```
T_{customer}
```

```
((gf-call checkout ?args)  
(all-past (gf-call buy ?args2)))
```
Temporal Matching

1. (login <shop> <kris> “kris” “kros” )
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> “kris” “kros” )
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)

&

customer

login buy logout login buy buy checkout

((gf-call checkout ?args)
 (most-recent (gf-call buy ?args2)))

&

login buy logout login buy buy checkout

((gf-call checkout ?args)
 (all-past (gf-call buy ?args2)))
Temporal Matching

1. (login <shop> <kris> “kris” “kros” )  
2. (buy <kris> <jacket>)  
3. (logout <kris>)  
4. (login <shop> <kris> “kris” “kros”)  
5. (buy <kris> <t-shirt>)  
6. (buy <kris> <socks>)  
7. (checkout <kris>)

login buy logout login buy buy checkout

T_{customer}

((gf-call checkout ?args)  
 (most-recent (gf-call buy ?args2)))

login buy logout login buy buy checkout

T_{customer}

((gf-call checkout ?args)  
 (all-past (gf-call buy ?args2))))
Temporal Matching

1. (login <shop> <kris> “kris” “kros” )
2. (buy <kris> <jacket> )
3. (logout <kris> )
4. (login <shop> <kris> “kris” “kros” )
5. (buy <kris> <t-shirt> )
6. (buy <kris> <socks> )
7. (checkout <kris> )

\[
\text{login buy logout login buy buy checkout}
\]

\[
((\text{gf-call checkout ?args})
\text{(most-recent (gf-call buy ?args2))})
\]

\[
\text{login buy logout login buy buy checkout}
\]

\[
((\text{gf-call checkout ?args})
\text{(all-past (gf-call buy ?args2))})
\]
Temporal Matching

1. (login <shop> <kris> “kris” “kros” )
2. (buy <kris> <jacket> )
3. (logout <kris> )
4. (login <shop> <kris> “kris” “kros” )
5. (buy <kris> <t-shirt> )
6. (buy <kris> <socks> )
7. (checkout <kris> )

login buy logout login buy buy checkout

T_{customer}

((gf-call checkout ?args) (most-recent (gf-call buy ?args2)))

login buy logout login buy buy checkout

T_{customer}

((gf-call checkout ?args) (all-past (gf-call buy ?args2)))

login buy logout login buy buy checkout

T_{customer}
Temporal Matching

1. (login <shop> <kris> “kris” “kros” )
2. (buy <kris> <jacket> )
3. (logout <kris> )
4. (login <shop> <kris> “kris” “kros” )
5. (buy <kris> <t-shirt> )
6. (buy <kris> <socks> )
7. (checkout <kris> )

login buy logout login buy buy checkout

((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2)))

login buy logout login buy buy checkout

[((gf-call checkout ?args)
  (all-past (gf-call buy ?args2)))

login buy logout login buy buy checkout

((gf-call checkout ?args)
  (since (most-recent (gf-call login ?args2))
   (all-past (gf-call buy ?args3))

T_{\text{customer}}
Temporal Matching

1. (login <shop> <kris> "kris" "kros")
2. (buy <kris> <jacket>)
3. (logout <kris>)
4. (login <shop> <kris> "kris" "kros")
5. (buy <kris> <t-shirt>)
6. (buy <kris> <socks>)
7. (checkout <kris>)

login buy logout login buy buy checkout

((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2)))

login buy logout login buy buy checkout

((gf-call checkout ?args)
  (all-past (gf-call buy ?args2)))

login buy logout login buy buy checkout

((gf-call checkout ?args)
  (since (most-recent (gf-call login ?args2))
     (all-past (gf-call buy ?args3)))

$T_{\text{customer}}$
Temporal Matching

1. (login <shop> <kris> “kris” “kros” )
2. (buy <kris> <jacket> )
3. (logout <kris> )
4. (login <shop> <kris> “kris” “kros” )
5. (buy <kris> <t-shirt> )
6. (buy <kris> <socks> )
7. (checkout <kris> )

login  buy  logout  login  buy  buy  checkout

T_{customer}

((gf-call checkout ?args)
 (most-recent (gf-call buy ?args2)))

login  buy  logout  login  buy  buy  checkout

T_{customer}

((gf-call checkout ?args)
 (all-past (gf-call buy ?args2)))

login  buy  logout  login  buy  buy  checkout

T_{customer}

((gf-call checkout ?args)
 (since (most-recent (gf-call login ?args2))
 (all-past (gf-call buy ?args3)))

T_{customer}
Temporal Matching

1. (login <shop> <kris> “kris” “kros”)  
2. (buy <kris> <jacket>)  
3. (logout <kris>)  
4. (login <shop> <kris> “kris” “kros”)  
5. (buy <kris> <t-shirt>)  
6. (buy <kris> <socks>)  
7. (checkout <kris>)

```
login  buy  logout  login  buy  buy  checkout
```

```
((gf-call checkout ?args)  
 (most-recent (gf-call buy ?args2)))
```

```
login  buy  logout  login  buy  buy  checkout
```

```
((gf-call checkout ?args)  
 (all-past (gf-call buy ?args2)))
```

```
login  buy  logout  login  buy  buy  checkout
```

```
((gf-call checkout ?args)  
 (since (most-recent (gf-call login ?args2))  
 (all-past (gf-call buy ?args3)))
```
Temporal Matching

1. (login <shop> <kris> “kris” “kros” )
2. (buy <kris> <jacket> )
3. (logout <kris> )
4. (login <shop> <kris> “kris” “kros” )
5. (buy <kris> <t-shirt> )
6. (buy <kris> <socks> )
7. (checkout <kris> )

(login buy logout login buy buy checkout)

(((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2))))

(login buy logout login buy buy checkout)

(((gf-call checkout ?args)
  (all-past (gf-call buy ?args2))))

(login buy logout login buy buy checkout)

(((gf-call checkout ?args)
  (since (most-recent (gf-call login ?args2))
           (all-past (gf-call buy ?args3))))
Temporal Matching

1. (login <shop> <kris> “kris” “kros” )
2. (buy <kris> <jacket> )
3. (logout <kris> )
4. (login <shop> <kris> “kris” “kros” )
5. (buy <kris> <t-shirt> )
6. (buy <kris> <socks> )
7. (checkout <kris> )

login  buy   logout  login  buy   buy  buy   checkout

(((gf-call checkout ?args)
  (most-recent (gf-call buy ?args2)))

login  buy   logout  login  buy   buy   buy   checkout

(((gf-call checkout ?args)
  (all-past (gf-call buy ?args2)))

login  buy   logout  login  buy   buy   buy   checkout

(((gf-call checkout ?args)
  (since (most-recent (gf-call login ?args2))
  (all-past (gf-call buy ?args3))

Stateful context-exposure
Stateful context-exposure

(let ((promo (singleton-instance 'promotions)))
  (setf (rate-for promo) <cd> 0.05)
  (login <kris> <shop> "kris" "kros")
  (buy <kris> <cd>)
  (setf (rate-for promo) <cd> 0.00)
  (buy <kris> <cd>)))
Stateful context-exposure

(let ((promo (singleton-instance ‘promotions)))
  (setf (rate-for promo) <cd> 0.05)
  (login <kris> <shop> “kris” ”kros”)
  (buy <kris> <cd>)
  (setf (rate-for promo) <cd> 0.00)
  (buy <kris> <cd>)))
Stateful context-exposure

(let ((promo (singleton-instance 'promotions)))
  (setf (rate-for promo) <cd> 0.05)
  (login <kris> <shop> "kris" "kros")
  (buy <kris> <cd>)
  (setf (rate-for promo) <cd> 0.00)
  (buy <kris> <cd>)))

(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
     (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))

T <kris>
Stateful context-exposure

1. (let ((promo (singleton-instance 'promotions)))
   (setf (rate-for promo) <cd> 0.05)
   (login <kris> <shop> "kris" "kros")
   (buy <kris> <cd>)
   (setf (rate-for promo) <cd> 0.00)
   (buy <kris> <cd>))

(at
   ((gf-call buy ?user ?article)
    (most-recent (gf-call login ?user _ _ _)
      (rate-for ?rate ?article)))
   (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
Stateful context-exposure

1. (let ((promo (singleton-instance 'promotions)))
2.  (setf (rate-for promo) <cd> 0.05)
   (login <kris> <shop> "kris" "kros")
   (buy <kris> <cd>)
   (setf (rate-for promo) <cd> 0.00)
   (buy <kris> <cd>)))

(at
   ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
   (rate-for ?rate ?article)))
   (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
Stateful context-exposure

1. (let ((promo (singleton-instance 'promotions)))
2.  (setf (rate-for promo) <cd> 0.05)
3.  (login <kris> <shop> "kris" "kros")
    (buy <kris> <cd>)
    (setf (rate-for promo) <cd> 0.00)
    (buy <kris> <cd>)))

(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
                (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
Stateful context-exposure

1. (let ((promo (singleton-instance 'promotions)))
2.  (setf (rate-for promo) <cd> 0.05)
3.  (login <kris> <shop> "kris" "kros")
    (buy <kris> <cd>)
    (setf (rate-for promo) <cd> 0.00)
    (buy <kris> <cd>)))

(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
     (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
Stateful context-exposure

1. (let ((promo (singleton-instance 'promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> "kris" "kros")
4.   (buy <kris> <cd>)
    (setf (rate-for promo) <cd> 0.00)
    (buy <kris> <cd>)))

(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
     (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
Stateful context-exposure

1. (let ((promo (singleton-instance ‘promotions)))
2.  (setf (rate-for promo) <cd> 0.05)
3.  (login <kris> <shop> “kris” ” kros”)
4.  (buy <kris> <cd>)
    (setf (rate-for promo) <cd> 0.00)
    (buy <kris> <cd>))

(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
    (rate-for ?rate ?article)))
  (format t “~s gets a ~s % discount on ~s” ?user ?rate ?article))
Stateful context-exposure

1. (let ((promo (singleton-instance 'promotions)))
2.  (setf (rate-for promo) <cd> 0.05)
3.  (login <kris> <shop> "kris" "kros")
4.  (buy <kris> <cd>)
5.  (setf (rate-for promo) <cd> 0.00)
   (buy <kris> <cd>)))

(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
                (rate-for ?rate ?article)))
   (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))

login       buy

T <kris>
Stateful context-exposure

1. (let ((promo (singleton-instance 'promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> "kris" "kros")
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>)))

(login  buy

(at
  ((gf-call buy ?user ?article)
     (most-recent (gf-call login ?user _ _ _)
                  (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))

T <kris>
Stateful context-exposure

1. (let ((promo (singleton-instance ‘promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> “kris” ”kros”)
4.   (buy <kris> <cd>))
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>)))

(at
   ((gf-call buy ?user ?article)
    (most-recent (gf-call login ?user _ _ _)
                 (rate-for ?rate ?article)))
   (format t “~s gets a ~s % discount on ~s” ?user ?rate ?article))
Stateful context-exposure

1. (let ((promo (singleton-instance 'promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> "kris" "kros")
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>)))

(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
                (rate-for ?rate ?article))
   (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))

login  buy  buy

T <kris>
Stateful context-exposure

1. (let ((promo (singleton-instance 'promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> “kris” ”kros”)
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>))

(at
   ((gf-call buy ?user ?article)
    (most-recent (gf-call login ?user _ _ _)
                (rate-for ?rate ?article)))
   (format t “~s gets a ~s % discount on ~s” ?user ?rate ?article))
Stateful context-exposure

1. (let ((promo (singleton-instance ‘promotions)))
2.  (setf (rate-for promo) <cd> 0.05)
3.  (login <kris> <shop> “kris” ”kros”)
4.  (buy <kris> <cd>)
5.  (setf (rate-for promo) <cd> 0.00)
6.  (buy <kris> <cd>))

(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
     (rate-for ?rate ?article)))
  (format t “~s gets a ~s % discount on ~s” ?user ?rate ?article))
Stateful context-exposure

1. (let ((promo (singleton-instance ‘promotions)))
2.  (setf (rate-for promo) <cd> 0.05)
3.  (login <kris> <shop> “kris” ”kros” )
4.  (buy <kris> <cd>)
5.  (setf (rate-for promo) <cd> 0.00)
6.  (buy <kris> <cd>))

(login buy buy)

(at
  ((gf-call buy ?user ?article)
    (most-recent (gf-call login ?user _ _ _)
      (rate-for ?rate ?article)))
  (format t “~s gets a ~s % discount on ~s” ?user ?rate ?article))
Stateful context-exposure

1. (let ((promo (singleton-instance 'promotions)))
2.  (setf (rate-for promo) <cd> 0.05)
3.  (login <kris> <shop> "kris" "kros")
4.  (buy <kris> <cd>)
5.  (setf (rate-for promo) <cd> 0.00)
6.  (buy <kris> <cd>)))

_login_ _buy_ _buy_

(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
     (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
Stateful context-exposure

1. (let ((promo (singleton-instance ‘promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> “kris” ”kros” )
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>)))

(login buy buy

T <kris>

(at
   ((gf-call buy ?user ?article)
      (most-recent (gf-call login ?user _ _ _)
                   (rate-for ?rate ?article)))
   (format t “~s get a ~s % discount on ~s” ?user ?rate ?article))

(defrule (rate-for ?rate ?article)
   (escape ?promo (singleton-instance ‘promotions)
   (escape ?rate (discount-rate-for ?promo ?article)))
Stateful context-exposure

1. (let ((promo (singleton-instance ‘promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> "kris" "kros")
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>)))

(login  buy  buy)

(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
    (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
Stateful context-exposure

1. (let ((promo (singleton-instance 'promotions)))
2.  (setf (rate-for promo) <cd> 0.05)
3.  (login <kris> <shop> "kris" "kros")
4.  (buy <kris> <cd>))
5.  (setf (rate-for promo) <cd> 0.00)
6.  (buy <kris> <cd>)))

(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
     (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))

T <kris>
Stateful context-exposure

1. (let ((promo (singleton-instance ‘promotions)))
2.  (setf (rate-for promo) <cd> 0.05)
3.  (login <kris> <shop> “kris” ”kros”)
4.  (buy <kris> <cd>)
5.  (setf (rate-for promo) <cd> 0.00)
6.  (buy <kris> <cd>))

```lisp
(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
                (rate-for ?rate ?article))
   (format t “~s gets a ~s % discount on ~s” ?user ?rate ?article))
)```
Stateful context-exposure

1. (let ((promo (singleton-instance ‘promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> “kris” ” kros” )
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>)))

"<kris> gets a 0.05% discount on <cd>"
Stateful context-exposure

1. (let ((promo (singleton-instance 'promotions)))
2. (setf (rate-for promo) <cd> 0.05)
3. (login <kris> <shop> "kris" "kros")
4. (buy <kris> <cd>)
5. (setf (rate-for promo) <cd> 0.00)
6. (buy <kris> <cd>))

(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
     (rate-for ?rate ?article))
   (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))

"<kris> gets a 0.05% discount on <cd>"
1. (let ((promo (singleton-instance 'promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> "kris" "kros")
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>)))

```
(at
  ((gf-call buy ?user ?article)
    (most-recent (gf-call login ?user _ _ _)
      (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

"<kris> gets a 0.05% discount on <cd>"
Stateful context-exposure

1. (let ((promo (singleton-instance ‘promotions)))
2.   (setf (rate-for promo) <cd> 0.05)
3.   (login <kris> <shop> “kris” ”kros”)
4.   (buy <kris> <cd>)
5.   (setf (rate-for promo) <cd> 0.00)
6.   (buy <kris> <cd>))

(at
  ((gf-call buy ?user ?article)
    (most-recent (gf-call login ?user _ _ _)
    (rate-for ?rate ?article)))
  (format t “~s gets a ~s % discount on ~s” ?user ?rate ?article))

“<kris> gets a 0.05% discount on <cd>”

“<kris> gets a 0.05% discount on <cd>”
Stateful context-exposure

1. (let ((promo (singleton-instance ‘promotions)))
2.  (setf (rate-for promo) <cd> 0.05)
3.  (login <kris> <shop> “kris” ” kros” )
4.  (buy <kris> <cd>)
5.  (setf (rate-for promo) <cd> 0.00)
6.  (buy <kris> <cd>)))

(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
     (rate-for ?rate ?article)))
  (format t “~s gets a ~s % discount on ~s” ?user ?rate ?article))

“<kris> gets a 0.05% discount on <cd>”

“<kris> gets a 0.05% discount on <cd>”

(at
  ((gf-call buy ?user ?article)
   (rate-for ?rate ?article)
   (most-recent (gf-call login ?user _ _ _)))
  (format t “~s gets a ~s % discount on ~s” ?user ?rate ?article))
Stateful context-exposure

1. (let ((promo (singleton-instance ‘promotions)))
2.  (setf (rate-for promo) <cd> 0.05)
3.  (login <kris> <shop> “kris” ”kros” )
4.  (buy <kris> <cd>)
5.  (setf (rate-for promo) <cd> 0.00)
6.  (buy <kris> <cd>)))

(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
     (rate-for ?rate ?article)))
   (format t “~s gets a ~s % discount on ~s” ?user ?rate ?article))

“<kris> gets a 0.05% discount on <cd>”
Stateful context-exposure

1. (let ((promo (singleton-instance 'promotions)))
2.  (setf (rate-for promo) <cd> 0.05)
3.  (login <kris> <shop> "kris" "kros")
4.  (buy <kris> <cd>)
5.  (setf (rate-for promo) <cd> 0.00)
6.  (buy <kris> <cd>)))

(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
     (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))

"<kris> gets a 0.05% discount on <cd>"

"<kris> gets a 0.05% discount on <cd>"
Stateful context-exposure

1. (let ((promo (singleton-instance ‘promotions)))
2.  (setf (rate-for promo) <cd> 0.05)
3.  (login <kris> <shop> “kris” ” kros”)
4.  (buy <kris> <cd>)
5.  (setf (rate-for promo) <cd> 0.00)
6.  (buy <kris> <cd>)))

(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
     (rate-for ?rate ?article)))
  (format t “~s gets a ~s % discount on ~s” ?user ?rate ?article))

“<kris> gets a 0.05% discount on <cd>”

(at
  ((gf-call buy ?user ?article)
   (rate-for ?rate ?article)
   (most-recent (gf-call login ?user _ _ _)))
  (format t “~s gets a ~s % discount on ~s” ?user ?rate ?article))

“<kris> gets a 0.05% discount on <cd>”
Stateful context-exposure

1. (let ((promo (singleton-instance 'promotions)))
2. (setf (rate-for promo) <cd> 0.05)
3. (login <kris> <shop> "kris"" kros")
4. (buy <kris> <cd>)
5. (setf (rate-for promo) <cd> 0.00)
6. (buy <kris> <cd>)))

```
(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _))
   (rate-for ?rate ?article)))
(format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))
```

"<kris> gets a 0.05% discount on <cd>"

"<kris> gets a 0.05% discount on <cd>"
Stateful context-exposure

1. (let ((promo (singleton-instance 'promotions)))
2.  (setf (rate-for promo) <cd> 0.05)
3.  (login <kris> <shop> "kris" "kros")
4.  (buy <kris> <cd>)
5.  (setf (rate-for promo) <cd> 0.00)
6.  (buy <kris> <cd>))

(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
    (rate-for ?rate ?article)))
  (format t "~s gets a ~s % discount on ~s" ?user ?rate ?article))

"<kris> gets a 0.05% discount on <cd>"

"<kris> gets a 0.00% discount on <cd>"
Stateful context-exposure

1. (let ((promo (singleton-instance ‘promotions)))
2.  (setf (rate-for promo) <cd> 0.05)
3.  (login <kris> <shop> “kris” ”kros”)
4.  (buy <kris> <cd>)
5.  (setf (rate-for promo) <cd> 0.00)
6.  (buy <kris> <cd>)))

(at
  ((gf-call buy ?user ?article)
   (most-recent (gf-call login ?user _ _ _)
     (rate-for ?rate ?article)))
  (format t “~s gets a ~s % discount on ~s” ?user ?rate ?article))

“<kris> gets a 0.05% discount on <cd>”
“<kris> gets a 0.05% discount on <cd>”

(at
  ((gf-call buy ?user ?article)
   (rate-for ?rate ?article)
   (most-recent (gf-call login ?user _ _ _)))
  (format t “~s gets a ~s % discount on ~s” ?user ?rate ?article))

“<kris> gets a 0.05% discount on <cd>”
“<kris> gets a 0.00% discount on <cd>”

Stateful context-exposure
(defclass user () ((name)))
(defmethod checkout ((u user)) ...)
(defmethod buy ((u user) (a article)) ...)
Weaver Implementation

(base)
(defclass user () ((name))
(defmethod checkout ((u user)) ...)
(defmethod buy ((u user) (a article)) ...)

(fact base)
TN: (gf-call checkout <user>)
TN-1: (gf-call buy <user> <cd>)

(jp ctr)

(fact generator)

(temporal relations)
most-recent, all-past, since, cflow

(rules)
(deerule (buy ?u)
  (gf-call buy ?u ?a))

(aspect)
(at
  ((gf-call checkout ?u)
   (most-recent (buy ?u ?a)))
  (log “user ~s bought ~s” ?u ?a))
Weaver Implementation

(defclass user () ((name)))
(defmethod checkout ((u user))
    (defmethod buy ((u user)))

* by wrapping class & generic function protocols in CLOS MOP
* explicit meta class tagging

(temporal relations)
(most-recent, all-past, since, cflow)
(deerule (buy ?u)
    (gf-call buy ?u ?a))

(query engine)

(rules)

(aspect)

(fact base)
(TN: (gf-call checkout <user>))
(TN-1: (gf-call buy <user> <cd>))

(fact generator)
Weaver Implementation

(base)
(defclass user () ((name)))
(defmethod checkout ((u user)) ...)
(defmethod buy ((u user) (a article)) ...)
Weaver Implementation

base
(defclass user () ((name)))
(defmethod checkout ((u user)) ...)
(defmethod buy ((u user) (a article)) ...)

fact generator

jp ctr

fact base
TN: (gf-call checkout <user>)
TN-1: (gf-call buy <user> <cd>)

temporal relations
most-recent, all-past, since, cflow

rules
(derule (buy ?u)
  (gf-call buy ?u ?a))

aspect
(at
  ((gf-call checkout ?u)
    (most-recent (buy ?u ?a)))
  (log “user ~s bought ~s” ?u ?a))
Weaver Implementation

Fact Generator

Base

(defclass user () ((name)))
(defmethod checkout ((u user)) ...)
(defmethod buy ((u user) (a article)) ...)

Fact Base

TN: (gf-call checkout <user>)
TN-1: (gf-call buy <user> <cd>)

Temporal Relations

Most-recent, all-past, since, cflow

Rules

(defrule (buy ?u)
  (gf-call buy ?u ?a))

Aspect

(at
  ((gf-call checkout ?u)
   (most-recent (buy ?u ?a)))
  (log "user ~s bought ~s" ?u ?a))

Query Engine

* Extension of RETE forward chaining, supporting temporal operators + hybrid pointcuts
* Copy semantics through static analysis
Weaver Implementation

(defclass user () ((name)))
(defmethod checkout ((u user)) ...)
(defmethod buy ((u user) (a article)) ...)

(base)

(fact generator)

(jp ctr)

(fact base)

(TN: (gf-call checkout <user>))
(TN-1: (gf-call buy <user> <cd>))

(temporal relations)

(most-recent, all-past, since, cflow)

(rules)

(defrule (buy ?u)
  (gf-call buy ?u ?a))

(aspect)

(at
  ((gf-call checkout ?u)
   (most-recent (buy ?u ?a)))
  (log “user ~s bought ~s” ?u ?a))

(query engine)
Weaver Implementation

(defclass user () ((name))
(defmethod checkout ((u user)) ...)
(defmethod buy ((u user) (a article)) ...) )

(temporal relations
most-recent, all-past, since, cflow
(rules
(defrule (buy ?u)
  (gf-call buy ?u ?a))

(aspect
(at
  ((gf-call checkout ?u)
    (most-recent (buy ?u ?a)))
  (log "user ~s bought ~s" ?u ?a))

(query engine)

(fact base
TN: (gf-call checkout <user>)
TN-1: (gf-call buy <user>)

(fact generator)
jp ctr

(base

Weaver Implementation

(defclass user () ((name))
(defmethod checkout ((u user))  
(defmethod buy ((u user) (a article))  

(base)

(fact generator)

(query engine)

(temporal relations)

(most-recent, all-past, since, cflow)

(rules)

(defrule (buy ?u)  
(gf-call buy ?u ?a))

(aspect)

(at  
((gf-call checkout ?u)  
(most-recent (buy ?u ?a)))

(log “user ~s bought ~s” ?u ?a))

* reducing the join point history dynamically
* gbc in RETE based on semantics of temporal operators
* results presented @ RV 2007
Backward Chaining vs Rete Forward Chaining

(at ((gf-call checkout (?user))
   (most-recent (gf-call login ?user ?shop)
      (escape ?rate (current-rate ?shop))))
(discoun...
Backward Chaining vs Rete Forward Chaining

(at ((gf-call checkout (?user))
   (most-recent (gf-call login ?user ?shop)
     (escape ?rate (current-rate ?shop))))
(discoun ?user ?rate))
Backward Chaining vs Rete Forward Chaining

(at ((gf-call checkout (?user))
   (most-recent (gf-call login ?user ?shop)
       (escape ?rate (current-rate ?shop))))
   (discount ?user ?rate))
Backward Chaining vs Rete Forward Chaining

```
(at ((gf-call checkout (?user))
    (most-recent (gf-call login ?user ?shop)
      (escape ?rate (current-rate ?shop)))))
(discout ?user ?rate))
```

![Diagram showing backward chaining process](image-url)
Backward Chaining vs Rete Forward Chaining

\[
\begin{align*}
(at & ((gf\text{-}call\ checkout\ (?user)) \\
& (most\text{-}recent\ (gf\text{-}call\ login\ ?user\ ?shop) \\
& \quad (escape\ ?rate\ (current\text{-}rate\ ?shop)))) \\
& (discount\ ?user\ ?rate))
\end{align*}
\]
Backward Chaining vs Rete Forward Chaining

```
(at ((gf-call checkout (?user))
    (most-recent (gf-call login ?user ?shop)
        (escape ?rate (current-rate ?shop)))
    (discount ?user ?rate))
```

Diagram showing the flow of logic with backward chaining.
Backward Chaining vs Rete Forward Chaining

```
(at ((gf-call checkout (?user))
    (most-recent (gf-call login ?user ?shop)
        (escape ?rate (current-rate ?shop)))
    (discount ?user ?rate))
```

```
(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))
```

```
(gf-call checkout (?user))
```

```
most-recent
```

```
?(discount ?user ?rate)
```

---

Query engine
Backward Chaining vs Rete Forward Chaining

```
(at ((gf-call checkout (?user))
  (most-recent (gf-call login ?user ?shop)
    (escape ?rate (current-rate ?shop))))
(disc...
Backward Chaining vs Rete Forward Chaining

(at (gf-call checkout (?user))
  (most-recent (gf-call login ?user ?shop)
    (escape ?rate (current-rate ?shop))))
(discoutn ?user ?rate))
Backward Chaining vs Rete Forward Chaining

```prolog
(at ((gf-call checkout (?user))
    (most-recent (gf-call login ?user ?shop)
        (escape ?rate (current-rate ?shop))))
    (discount ?user ?rate))
```

![Diagram](image-url)
Backward Chaining vs Rete Forward Chaining

(at ((gf-call checkout (?user))
    (most-recent (gf-call login ?user ?shop)
        (escape ?rate (current-rate ?shop))))
(escape ?rate (current-rate ?shop)))
(discount ?user ?rate))

(query engine)

Backward Chaining

(fg-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(fg-call checkout (?user))

and

most-recent

?(discount ?user ?rate)
Backward Chaining vs Rete Forward Chaining

(at ((gf-call checkout (?user)))
  (most-recent (gf-call login ?user ?shop)
    (escape ?rate (current-rate ?shop))))
(discount ?user ?rate))
Backward Chaining vs Rete Forward Chaining

```
(at ((gf-call checkout (?user))
    (most-recent (gf-call login ?user ?shop)
        (escape ?rate (current-rate ?shop))))
  (discount ?user ?rate))
```
Backward Chaining vs Rete Forward Chaining

(at ((gf-call checkout (?user))
  (most-recent (gf-call login ?user ?shop)
    (escape ?rate (current-rate ?shop))))
  (discount ?user ?rate))

(query engine) -> (gf-call checkout (?user)) -> (gf-call login ?user ?shop) -> (escape ?rate (current-rate ?shop)) -> (most-recent) -> (discount ?user ?rate)
Backward Chaining vs Rete Forward Chaining

(at ((gf-call checkout (?user))
    (most-recent (gf-call login ?user ?shop)
        (escape ?rate (current-rate ?shop))))
  (discount ?user ?rate))
Backward Chaining vs Rete Forward Chaining

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user)) and

most-recent

?(discount ?user ?rate)

query engine
"On every checkout, give a discount based on the rate at login time"
"On every checkout, give a discount based on the rate at login time"

\[
\begin{align*}
&\text{(set-rate *shop* 0.05)} \\
&\text{(login *kris* *shop*)} \\
&\text{(buy *kris* *cd*)} \\
&\text{(set-rate *shop* 0.00)} \\
&\text{(checkout *kris*)}
\end{align*}
\]

\[
\text{(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))}
\]

\[
\text{(gf-call checkout (?user))}
\]

\[
\text{and}
\]

\[
\text{most-recent}
\]

\[
\text{?(discount ?user ?rate)}
\]

Backward Chaining
"On every checkout, give a discount based on the rate at login time”

1. (set-rate *shop* 0.05)
   (login *kris* *shop*)
   (buy *kris* *cd* )
   (set-rate *shop* 0.00)
   (checkout *kris*)

On every checkout, give a discount based on the rate at login time.

(query engine)

(?(discount *user* *rate*)

(gf-call login *user* *shop*) (escape *rate* (current-rate *shop*))

(gf-call checkout (?user*))

and

most-recent

Backward Chaining
“On every checkout, give a discount based on the rate at login time”

1. (set-rate *shop* 0.05)
   (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)  
   (login *kris* *shop*)  
   (buy *kris* *cd*)  
   (set-rate *shop* 0.00)  
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
   (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)

query engine

Backward Chaining
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
   (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05) 
   (login *kris* *shop*) 
   (buy *kris* *cd*) 
   (set-rate *shop* 0.00) 
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

Backward Chaining

query engine

?(discount ?user ?rate)
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
   (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call 'set-rate <shop> 0.05)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

?-(discount ?user ?rate)
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)  
   (login *kris* *shop*)  
   (buy *kris* *cd*)  
   (set-rate *shop* 0.00)  
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

Backward Chaining

query engine

?(discount ?user ?rate)
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

query engine

?(discount ?user ?rate)

Backward Chaining
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

??(discount ?user ?rate)

query engine

Backward Chaining
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)  
(2 gf-call ‘login <kris> <shop>)

(gf-call login ?user ?shop)  
(escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)

(query engine)

?(discount ?user ?rate)

Backward Chaining

most-recent

and

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))
"On every checkout, give a discount based on the rate at login time"
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

? (discount ?user ?rate)

query engine

Backward Chaining

On every checkout, give a discount based on the rate at login time

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

? (discount ?user ?rate)

query engine

Backward Chaining
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)

query engine

Backward Chaining

!
"On every checkout, give a discount based on the rate at login time"

1. `(set-rate *shop* 0.05)`
2. `(login *kris* *shop*)`
3. `(buy *kris* *cd*)`
   `(set-rate *shop* 0.00)`
   `(checkout *kris*)`

(1 `gf-call 'set-rate <shop> 0.05`)`
(2 `gf-call 'login <kris> <shop>`)

```
(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))
(gf-call checkout (?user))
```

(query engine)

```
?({discount ?user ?rate})
```

![Backward Chaining]

`most-recent`
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)
(3 gf-call ‘buy <kris> <cd>)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)

query engine

Backward Chaining
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time”

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)
(3 gf-call ‘buy <kris> <cd>)

(query engine)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)
(3 gf-call ‘buy <kris> <cd>)
(4 gf-call ‘set-rate <shop> 0.00)

query engine

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)
(3 gf-call ‘buy <kris> <cd>)
(4 gf-call ‘set-rate <shop> 0.00)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user)) and

most-recent

? (discount ?user ?rate)

query engine

Backward Chaining
“On every checkout, give a discount based on the rate at login time”

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)
(3 gf-call ‘buy <kris> <cd>)
(4 gf-call ‘set-rate <shop> 0.00)
(5 gf-call ‘checkout <kris>)

(query engine)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)

Backward Chaining
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)

query engine

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)
(3 gf-call ‘buy <kris> <cd>)
(4 gf-call ‘set-rate <shop> 0.00)
(5 gf-call ‘checkout <kris>)
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

1. (gf-call 'set-rate <shop> 0.05)
2. (gf-call 'login <kris> <shop>)
3. (gf-call 'buy <kris> <cd>)
4. (gf-call 'set-rate <shop> 0.00)
5. (gf-call 'checkout <kris>)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user)) and

most-recent

?(discount ?user ?rate)

query engine

Backward Chaining
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user)) and

most-recent

?(discount ?user ?rate)
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

1. (gf-call ‘set-rate <shop> 0.05)
2. (gf-call ‘login <kris> <shop>)
3. (gf-call ‘buy <kris> <cd>)
4. (gf-call ‘set-rate <shop> 0.00)
5. (gf-call ‘checkout <kris>)

(query engine)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)

Backward Chaining
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(query engine)
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user)) and

most-recent

?(discount ?user ?rate)

query engine
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time”

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)
(3 gf-call ‘buy <kris> <cd>)
(4 gf-call ‘set-rate <shop> 0.00)
(5 gf-call ‘checkout <kris>)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(5 gf-call ‘checkout <kris>) and

most-recent

?(discount ?user ?rate)

query engine

Backward Chaining
"On every checkout, give a discount based on the rate at login time”

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)
(3 gf-call ‘buy <kris> <cd>)
(4 gf-call ‘set-rate <shop> 0.00)
(5 gf-call ‘checkout <kris>)
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)
(3 gf-call ‘buy <kris> <cd>)
(4 gf-call ‘set-rate <shop> 0.00)
(5 gf-call ‘checkout <kris>)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(5 gf-call ‘checkout <kris>)

and

most-recent

?(discount ?user ?rate)

query engine

Backward Chaining
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)  
2. (login *kris* *shop*)  
3. (buy *kris* *cd*)  
4. (set-rate *shop* 0.00)  
5. (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)  
(2 gf-call ‘login <kris> <shop>)  
(3 gf-call ‘buy <kris> <cd>)  
(4 gf-call ‘set-rate <shop> 0.00)  
(5 gf-call ‘checkout <kris>)

(gf-call login ?user ?shop)  
(escape ?rate (current-rate ?shop))

(5 gf-call ‘checkout <kris>)  
and

most-recent  

?(discount ?user ?rate)

query engine  

Backward Chaining
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

On every checkout, give a discount based on the rate at login time.

(5 gf-call ‘checkout <kris>)

and

most-recent

?(discount ?user ?rate)

query engine

(5 gf-call ‘checkout <kris>)

(4 gf-call ‘set-rate <shop> 0.00)

(3 gf-call ‘buy <kris> <cd>)

(2 gf-call ‘login <kris> <shop>)

(1 gf-call ‘set-rate <shop> 0.05)

Backward Chaining

query engine

On every checkout, give a discount based on the rate at login time.
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)
(3 gf-call ‘buy <kris> <cd>)
(4 gf-call ‘set-rate <shop> 0.00)
(5 gf-call ‘checkout <kris>)

(2 gf-call ‘login <kris> <shop>)

(escape ?rate (current-rate ?shop))

(5 gf-call ‘checkout <kris>)

and

most-recent

? (discount ?user ?rate)

query engine

Backward Chaining
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)
(3 gf-call ‘buy <kris> <cd>)
(4 gf-call ‘set-rate <shop> 0.00)
(5 gf-call ‘checkout <kris>)

(2 gf-call ‘login <kris> <shop>)
(5 gf-call ‘checkout <kris>)

(escape ?rate (current-rate ?shop))

query engine

?(discount ?user ?rate)

Backward Chaining

and

most-recent
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)
(3 gf-call ‘buy <kris> <cd>)
(4 gf-call ‘set-rate <shop> 0.00)
(5 gf-call ‘checkout <kris>)

(2 gf-call ‘login <kris> <shop>)
(5 gf-call ‘checkout <kris>)

(escape ?rate (current-rate ?shop))

Backward Chaining

query engine

?(discount ?user ?rate)
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)

(2 gf-call 'login <kris> <shop>)
(escape 0.00 (current-rate <shop>))

(5 gf-call 'checkout <kris>)

and

most-recent

query engine

?(discount ?user ?rate)
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)

(2 gf-call 'login <kris> <shop>)
(escape 0.00 (current-rate <shop>))

(5 gf-call 'checkout <kris>)

(query engine)

?(discount ?user ?rate)

Backward Chaining

most-recent

and

On every checkout, give a discount based on the rate at login time.
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)
(3 gf-call ‘buy <kris> <cd>)
(4 gf-call ‘set-rate <shop> 0.00)
(5 gf-call ‘checkout <kris>)

(2 gf-call ‘login <kris> <shop>)
(escape 0.00 (current-rate <shop>))

(5 gf-call ‘checkout <kris>)

and

most-recent

?(discount ?user ?rate)

query engine

Backward Chaining
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)
(3 gf-call ‘buy <kris> <cd>)
(4 gf-call ‘set-rate <shop> 0.00)
(5 gf-call ‘checkout <kris>)

(2 gf-call ‘login <kris> <shop>)
(escape 0.00 (current-rate <shop>))

(5 gf-call ‘checkout <kris>)

and

most-recent

(query engine)

(discount <kris> 0.00)
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)  
2. (login *kris* *shop*)  
3. (buy *kris* *cd*)  
4. (set-rate *shop* 0.00)  
5. (checkout *kris*)

1. (gf-call 'set-rate <shop> 0.05)  
2. (gf-call 'login <kris> <shop>)  
3. (gf-call 'buy <kris> <cd>)  
4. (gf-call 'set-rate <shop> 0.00)  
5. (gf-call 'checkout <kris>)

(2 gf-call 'login <kris> <shop>)

(escape 0.00 (current-rate <shop>))

(5 gf-call 'checkout <kris>)

and

most-recent

(query engine)

(discount <kris> 0.00)

Backward Chaining

On every checkout, give a discount based on the rate at login time.
"On every checkout, give a discount based on the rate at login time"

(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

most-recent

?(discount ?user ?rate)

query engine
"On every checkout, give a discount based on the rate at login time"

(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

most-recent

?(discount ?user ?rate)
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)

(query engine)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user)) and

most-recent

?(discount ?user ?rate)

Backward Chaining
Rete Forward Chaining
"On every checkout, give a discount based on the rate at login time"

(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)

(query engine)

On every checkout, give a discount based on the rate at login time.

(query engine)

(esc query engine)

(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)
"On every checkout, give a discount based on the rate at login time"

```
(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)
```

```
(gf-call login ?user ?shop)
(escape ?rate (current-rate ?shop))
```

```
(gf-call checkout (?user))
```

```
and
```

```
most-recent
```

```
?(discount ?user ?rate)
```

**Query Engine**

**Backward Chaining**

**Rete Forward Chaining**
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)

query engine

Backward Chaining

Rete Forward Chaining
"On every checkout, give a discount based on the rate at login time"

```
(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)
```

```
(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))
```

```
(gf-call checkout (?user))
```

```
?/(discount ?user ?rate)
```

```
and
```

```
most-recent
```

```
Backward Chaining
```

```
Rete Forward Chaining
```

query engine
"On every checkout, give a discount based on the rate at login time"

```lisp
(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)
```

```
(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))
```

```
(gf-call checkout (?user))
```

```
and
```

```
most-recent
```

```
?(discount ?user ?rate)
```

Diagram illustrating Backward Chaining and Rete Forward Chaining.
"On every checkout, give a discount based on the rate at login time"

```
(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)
```
"On every checkout, give a discount based on the rate at login time"

(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)

(query engine)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)

Backward Chaining

Rete Forward Chaining
"On every checkout, give a discount based on the rate at login time"

```lisp
(set-rate *shop* 0.05)
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)
```

```
(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))
```

```
(gf-call checkout (?user))
```

```
and
```

```
most-recent
```

```
?(discount ?user ?rate)
```

Query engine

Backward Chaining

Rete Forward Chaining
"On every checkout, give a discount based on the rate at login time"

\[
\text{(set-rate *shop* 0.05)} \\
\text{(login *kris* *shop*)} \\
\text{(buy *kris* *cd*)} \\
\text{(set-rate *shop* 0.00)} \\
\text{(checkout *kris*)}
\]

\[
\text{(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))}
\]

\[
\text{(gf-call checkout (?user))}
\]

\[
\text{?*(discount ?user ?rate)}
\]

Backward Chaining vs Rete Forward Chaining
"On every checkout, give a discount based on the rate at login time"

(set-rate *shop* 0.05)  
(login *kris* *shop*)  
(buy *kris* *cd*)  
(set-rate *shop* 0.00)  
(checkout *kris*)

(query engine)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))  
and  
most-recent  
?:(discount ?user ?rate)

Backward Chaining

Rete Forward Chaining
"On every checkout, give a discount based on the rate at login time"

=set-rate *shop* 0.05
(login *kris* *shop*)
(buy *kris* *cd*)
(set-rate *shop* 0.00)
(checkout *kris*)

(query engine)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

? (discount ?user ?rate)

and

most-recent

Backward Chaining

Rete Forward Chaining
“On every checkout, give a discount based on the rate at login time”

1. (set-rate *shop* 0.05)  
   (login *kris* *shop*)  
   (buy *kris* *cd*)  
   (set-rate *shop* 0.00)  
   (checkout *kris*)

(query engine)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

most-recent

?(discount ?user ?rate)

Backward Chaining

Rete Forward Chaining
"On every checkout, give a discount based on the rate at login time"

1. \(\text{set-rate *shop* 0.05}\)
   \(\text{login *kris* *shop*}\)
   \(\text{buy *kris* *cd*}\)
   \(\text{set-rate *shop* 0.00}\)
   \(\text{checkout *kris*}\)

\(1 \text{ gf-call 'set-rate <shop> 0.05} \)

\(\text{gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))}\)

\(\text{gf-call checkout (?user)}\)

\(\text{and}\)

\(\text{most-recent}\)

\(\text{?(discount ?user ?rate)}\)

---

Backward Chaining

Rete Forward Chaining
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
   (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)

query engine

Backward Chaining

Rete Forward Chaining
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. \((\text{set-rate } \text{*shop* } 0.05)\)
   \((\text{login } \text{kris* } \text{*shop*})\)
   \((\text{buy } \text{kris* } \text{*cd*})\)
   \((\text{set-rate } \text{*shop* } 0.00)\)
   \((\text{checkout } \text{kris*})\)

(1 \(\text{gf-call } \text{`set-rate <shop> } 0.05\)\))

\((\text{gf-call login } ?\text{user } ?\text{shop})\) (\(\text{escape } ?\text{rate (current-rate } ?\text{shop)}\))

\((\text{gf-call checkout } (?\text{user}))\)

and

most-recent

?(discount ?user ?rate)

query engine

Backward Chaining

Rete Forward Chaining
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)  
   (login *kris* *shop*)  
   (buy *kris* *cd*)  
   (set-rate *shop* 0.00)  
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)

(gf-call login ?user ?shop)  
   (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)

Backward Chaining

Rete Forward Chaining

query engine
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)  
   (login *kris* *shop*)  
   (buy *kris* *cd*)  
   (set-rate *shop* 0.00)  
   (checkout *kris*)

   (1 gf-call 'set-rate <shop> 0.05)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)

query engine

Backward Chaining

Rete Forward Chaining
"On every checkout, give a discount based on the rate at login time"

1. \((\text{set-rate *shop* 0.05})\)
\((\text{login *kris* *shop*})\)
\((\text{buy *kris* *cd*})\)
\((\text{set-rate *shop* 0.00})\)
\((\text{checkout *kris*})\)

\((\text{gf-call login ?user ?shop})\) \((\text{escape ?rate (current-rate ?shop)})\)
**Backward Chaining vs Rete Forward Chaining**

"On every checkout, give a discount based on the rate at login time"

1. `(set-rate *shop* 0.05)`
   `(login *kris* *shop*)`
   `(buy *kris* *cd*)`
   `(set-rate *shop* 0.00)`
   `(checkout *kris*)`

```
(1 gf-call 'set-rate <shop> 0.05)
```

```
(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))
```

```
(gf-call checkout (?user))
```

```
and
```

```
most-recent
```

```
?(discount ?user ?rate)
```

**Query Engine**

- **Backward Chaining**
- **Rete Forward Chaining**
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call `set-rate <shop> 0.05)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)

query engine

Backward Chaining

Rete Forward Chaining
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)

query engine

Backward Chaining

Rete Forward Chaining
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. \((\text{set-rate} \ \text{*shop*} \ 0.05)\)
2. \((\text{login} \ \text{*kris*} \ \text{*shop*})\)
   \((\text{buy} \ \text{*kris*} \ \text{*cd*})\)
   \((\text{set-rate} \ \text{*shop*} \ 0.00)\)
   \((\text{checkout} \ \text{*kris*})\)

\((\text{gf-call} \ \text{login} \ ?\text{user} \ ?\text{shop}) \ (\text{escape} \ ?\text{rate} \ (\text{current-rate} \ ?\text{shop}))\)

\((\text{gf-call} \ \text{checkout} \ (?\text{user}))\)

\((\text{most-recent})\)

\(?\text{(discount} \ ?\text{user} \ ?\text{rate})\)

query engine

Backward Chaining

Rete Forward Chaining
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time."

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))

(query engine)

?(discount ?user ?rate)

Backward Chaining

Rete Forward Chaining
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))
(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)

query engine

Backward Chaining

Rete Forward Chaining
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)

(gf-call login ?user ?shop) (escape ?rate (current-rate ?shop))

(gf-call checkout (?user))
and
most-recent
?(discount ?user ?rate)

<table>
<thead>
<tr>
<th>Backward Chaining</th>
<th>Re却 Forward Chaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>!</td>
</tr>
</tbody>
</table>

query engine
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)

(2 gf-call ‘login <kris> <shop>)
(escape ?rate (current-rate ?shop))

(query engine)

(?(discount ?user ?rate)

?sf:most-recent

and

?sf:most-recent

Backward
Chaining

Rete
Forward
Chaining
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
   (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)

(2 gf-call ‘login <kris> <shop>)
(escape 0.05 (current-rate <shop>))

(query engine)

?(discount ?user ?rate)
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. \((\text{set-rate } \text{*shop* } 0.05)\)
2. \((\text{login } \text{*kris* } \text{*shop*})\)
   \((\text{buy } \text{*kris* } \text{*cd*})\)
   \((\text{set-rate } \text{*shop* } 0.00)\)
   \((\text{checkout } \text{*kris*})\)

\((\text{1 gf-call} \text{ 'set-rate } \text{<shop>} \text{ 0.05})\)
\((\text{2 gf-call} \text{ 'login } \text{<kris>} \text{ <shop>})\)

\((\text{2 gf-call } \text{ 'login } \text{<kris>} \text{ <shop>})\)
\((\text{escape } 0.05 \text{ (current-rate } \text{<shop>})\))

\((\text{gf-call checkout } (\text{?user}))\)
\(\text{and}\)
\((\text{most-recent})\)

\(\text{query engine}\)

?\((\text{discount } \text{?user } \text{?rate})\)

\(\text{Backward Chaining}\)
\(\text{Rete Forward Chaining}\)
"On every checkout, give a discount based on the rate at login time”

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)

(2 gf-call ‘login <kris> <shop>)
(escape 0.05 (current-rate <shop>))

(gf-call checkout (?user))

(query engine)

?(discount ?user ?rate)

Backward Chaining

Rete Forward Chaining
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
   (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)

(2 gf-call 'login <kris> <shop>)
(escape 0.05 (current-rate <shop>))

(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)

query engine

Backward Chaining

Rete Forward Chaining
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
   (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)
(3 gf-call ‘buy <kris> <cd>)

(2 gf-call ‘login <kris> <shop>)
(escape 0.05 (current-rate <shop>))

(gf-call checkout (?user))

and

most-recent

?((discount ?user ?rate)

query engine

Backward Chaining

Rete Forward Chaining
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
   (checkout *kris*)

1. (gf-call 'set-rate <shop> 0.05)
2. (gf-call 'login <kris> <shop>)
3. (gf-call 'buy <kris> <cd>)
4. (gf-call 'set-rate <shop> 0.00)

(2 gf-call 'login <kris> <shop>)
(escape 0.05 (current-rate <shop>))

(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)

query engine

Backward Chaining

Rete Forward Chaining
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

1. (gf-call 'set-rate <shop> 0.05)
2. (gf-call 'login <kris> <shop>)
3. (gf-call 'buy <kris> <cd>)
4. (gf-call 'set-rate <shop> 0.00)

(2 gf-call 'login <kris> <shop>) (escape 0.05 (current-rate <shop>))

(gf-call checkout (?user)) and

most-recent

?(discount ?user ?rate)

query engine

Backward Chaining
Rete Forward Chaining
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. `(set-rate *shop* 0.05)`
2. `(login *kris* *shop*)`
3. `(buy *kris* *cd*)`
4. `(set-rate *shop* 0.00)`
5. `(checkout *kris*)`

(1 `gf-call 'set-rate <shop> 0.05`)
(2 `gf-call 'login <kris> <shop>`)
(3 `gf-call 'buy <kris> <cd>`)
(4 `gf-call 'set-rate <shop> 0.00`)
(5 `gf-call 'checkout <kris>`)

(2 `gf-call 'login <kris> <shop>`)
(escape 0.05 (current-rate <shop>))

(gf-call checkout (?user))

and

most-recent

?(discount ?user ?rate)

<table>
<thead>
<tr>
<th>Backward Chaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Warning]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rete Forward Chaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Superman]</td>
</tr>
</tbody>
</table>
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)  (1 gf-call ‘set-rate <shop> 0.05)
2. (login *kris* *shop*)    (2 gf-call ‘login <kris> <shop>)
3. (buy *kris* *cd*)        (3 gf-call ‘buy <kris> <cd>)
4. (set-rate *shop* 0.00)   (4 gf-call ‘set-rate <shop> 0.00)
5. (checkout *kris*)         (5 gf-call ‘checkout <kris>)

(2 gf-call ‘login <kris> <shop>) (escape 0.05 (current-rate <shop>))

(5 gf-call ‘checkout <kris>) and

most-recent

?(discount ?user ?rate)

query engine

Backward Chaining

Rete Forward Chaining
Backward Chaining vs Rete Forward Chaining

"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

(1 gf-call 'set-rate <shop> 0.05)
(2 gf-call 'login <kris> <shop>)
(3 gf-call 'buy <kris> <cd>)
(4 gf-call 'set-rate <shop> 0.00)
(5 gf-call 'checkout <kris>)

(2 gf-call 'login <kris> <shop>)
(escape 0.05 (current-rate <shop>))

(5 gf-call 'checkout <kris>)

?(discount ?user ?rate)

query engine

Backward Chaining

Rete Forward Chaining
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)
(3 gf-call ‘buy <kris> <cd>)
(4 gf-call ‘set-rate <shop> 0.00)
(5 gf-call ‘checkout <kris>)

(2 gf-call ‘login <kris> <shop>)
(escape 0.05 (current-rate <shop>))

(5 gf-call ‘checkout <kris>)

and

most-recent

(discount <kris> 0.05)

query engine

Backward Chaining

Rete Forward Chaining
"On every checkout, give a discount based on the rate at login time"

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

(1 gf-call ‘set-rate <shop> 0.05)
(2 gf-call ‘login <kris> <shop>)
(3 gf-call ‘buy <kris> <cd>)
(4 gf-call ‘set-rate <shop> 0.00)
(5 gf-call ‘checkout <kris>)

(2 gf-call ‘login <kris> <shop>)
(escape 0.05 (current-rate <shop>))

(5 gf-call ‘checkout <kris>)

and

most-recent

(discount <kris> 0.05)

Backward Chaining

Rete Forward Chaining

query engine
"On every checkout, give a discount based on the rate at login time”

1. (set-rate *shop* 0.05)
2. (login *kris* *shop*)
3. (buy *kris* *cd*)
4. (set-rate *shop* 0.00)
5. (checkout *kris*)

1. (gf-call ‘set-rate <shop> 0.05)
2. (gf-call ‘login <kris> <shop>)
3. (gf-call ‘buy <kris> <cd>)
4. (gf-call ‘set-rate <shop> 0.00)
5. (gf-call ‘checkout <kris>)

(2 gf-call ‘login <kris> <shop>)

(escape 0.05 (current-rate <shop>))

(5 gf-call ‘checkout <kris>)

(query engine)

(escape 0.05 (current-rate <shop>))

(1 gf-call ‘set-rate <shop> 0.05)

(escape 0.05 (current-rate <shop>))

(Temporal Extensions)

(Escape Extension)
Reducing Memory Overhead

(at ((gf-call checkout (?user)))
  (most-recent (gf-call login ?user ?shop)
    (escape ?rate (current-rate ?shop))))
(discount ?user ?rate))

(login *kris* *shop*)
(buy *kris* *cd*)
(checkout *kris*)
(login *kris* *shop*)
(buy *kris* *dvd*)
(checkout *kris*)
Reducing Memory Overhead

(at ((gf-call checkout (?user)))
  (most-recent (gf-call login ?user ?shop)
    (escape ?rate (current-rate ?shop))))
(discounf ?user ?rate))

(login *kris* *shop*)
(buy *kris* *cd*)
(checkout *kris*)
(login *kris* *shop*)
(buy *kris* *dvd*)
(checkout *kris*)

Code  (gf-call checkout (?user))

<table>
<thead>
<tr>
<th>T1</th>
<th>?user</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>&lt;kris&gt;</td>
</tr>
<tr>
<td>6</td>
<td>&lt;kris&gt;</td>
</tr>
</tbody>
</table>

(escape ?rate (current-rate ?shop))

<table>
<thead>
<tr>
<th>T2</th>
<th>?rate</th>
<th>?user</th>
<th>?shop</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.05</td>
<td>&lt;kris&gt;</td>
<td>&lt;shop&gt;</td>
</tr>
<tr>
<td>4</td>
<td>0.05</td>
<td>&lt;kris&gt;</td>
<td>&lt;shop&gt;</td>
</tr>
</tbody>
</table>
Reducing Memory Overhead

(at ((gf-call checkout (?user)))
  (most-recent (gf-call login ?user ?shop)
    (escape ?rate (current-rate ?shop))))
)(discount ?user ?rate)
Reducing Memory Overhead (2)

<table>
<thead>
<tr>
<th></th>
<th>Nr of Memory Table Entries Still Allocated</th>
<th>Total Nr of Memory Table Entries Ever Made</th>
<th>Nr of Generated Join Point Facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1 non copy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1 copy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2 non copy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2 copy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3 non copy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3 copy</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

It helps!

Modularizing crosscuts in an e-commerce application in Lisp using HALO
Charlotte Herzeel, Kris Gybels, Pascal Costanza, Theo D'Hondt
In "Proc. of the International Lisp Conference", 2007

Escaping with future variables in HALO
Charlotte Herzeel, Kris Gybels, Pascal Costanza
In "Proc. of the Runtime Verification Workshop", 2007
Summary & Future Work

• Summary:
  • history-based AOP for CLOS; stateful pointcuts
  • temporal logic-based: abstraction & logic variables
  • Rete forward chaining-based implementation
  • memory reduction

• Future Work:
  • language design:
    • exploit logic-programming (i.e. unification & recursion)
    • exploit temporal logic-programming (mixing temporal logic/advice code)
    • before/after ... what about around? (roll-back mechanism)
  • reducing memory overhead:
    • shadow weaving
Summary & Future Work

• Summary:
  • history-based AOP for CLOS; stateful pointcuts
  • temporal logic-based: abstraction & logic variables
  • Rete forward chaining-based implementation
  • memory reduction

• Future Work:
  • **language design:**
    • exploit logic-programming (i.e. unification & recursion)
    • exploit temporal logic-programming (mixing temporal logic/advice code)
    • before/after... what about around? (roll-back mechanism)
  • **reducing memory overhead:**
    • shadow weaving

Thanks for listening!
Questions?
Summary & Future Work

• Summary:
  • history-based AOP for CLOS; stateful pointcuts
  • temporal logic-based: abstraction & logic variables
  • Rete forward chaining-based implementation
  • memory reduction

• Future Work:
  • **language design:**
    • exploit logic-programming (i.e. unification & recursion)
    • exploit temporal logic-programming (mixing temporal logic/advice code)
    • before/after ... what about around? [roll-back mechanism]
  • **reducing memory overhead:**
    • shadow weaving

Thanks for listening!
Questions?
http://prog.vub.ac.be/HALO
References


• Arranging language features for more robust pattern-based crosscuts

• Seamless Integration of Rule-Based Knowledge and Object-Oriented Functionality with Linguistic Symbiosis, Maja D'Hondt, Kris Gybels, Viviane Jonckers, In "Proceedings of the 19th Annual ACM Symposium on Applied Computing (SAC 2004), Special Track on Object-Oriented Programming, Languages and Systems. Nicosia, Cyprus. ACM. March 2004.", 2004


• Expressive Pointcuts for Increased Modularity, Klaus Ostermann and Mira Mezini and Christoph Bockisch, Proceedings of the European Conference on Object-Oriented Programming (ECOOP)


• Escaping with future variables in HALO, Charlotte Herzeel, Kris Gybels, Pascal Costanza, Proceedings of the Runtime Verification Workshop 2007