universitätbonn ROOTS

Logic-Based Program Transformation
- From Prolog to CTs -

Dr. Günter Kniesel
gk@cs.uni-bonn.de
Institut für Informatik III
Universität Bonn

Copyright Notice

This slide set partly contains unpublished results of my habilitation work.

It is provided exclusively for my students preparing for the examination of my “Advanced Logic Programming” course.

Any other use is prohibited and requires my explicit written permission.

In particular, any form of copying (either as hard copy or in electronic form) of the entire slide set or parts of it is strictly prohibited unless done for the purpose mentioned above.

Dr. Günter Kniesel

Why Transform Logic Programs?

- Expressing state change -- Example: Tracking visited elements
  - State as parameters: Inefficiency and stack overflow
  - State as facts: Exploit first argument indexing and unlimited heap
- Program optimization
  - Caching
  - Partial Evaluation
  - ...
- Transforming models of other programs
  - JTransformer: Fact transformation represents Java transformation
- Many more...

Program Transformation with Prolog

- Basic operations
  - `assert` and `retract` clauses
- Problem: Semantics of self-modifying programs
  - What if predicates used in the derivation are modified?
  - Should backtracking “see” the change?
- Two known answers
  - Immediate Update Semantics (First approach)
  - Deferred Update Semantics (Second approach – Current ISO standard)
- Schedule for today
  - Learn about both semantics and their limitations
  - Learn about a new approach: The Really Deferred Update Semantics
  - Learn about its implementation: The “Conditional Transformation” language

Immediate Update Semantics

- Changes become visible immediately and globally!
  - New clauses are used upon backtracking

```
:- dynamic b/1.
a(1).       b(1).

modify_myself(X) :-
    b(X),
    Y is X+1,
    assert( b(Y) ).
```

Modified predicate used in the derivation

```
?- modify_myself(X).
X = 1 ; % asserts b(2).
X = 2 ; % asserts b(3).
X = 3 ; % asserts b(4).
X = 4 ; % asserts b(5).
X = 5 ; % asserts b(6).
X = 6 ; % asserts b(7).
...
```

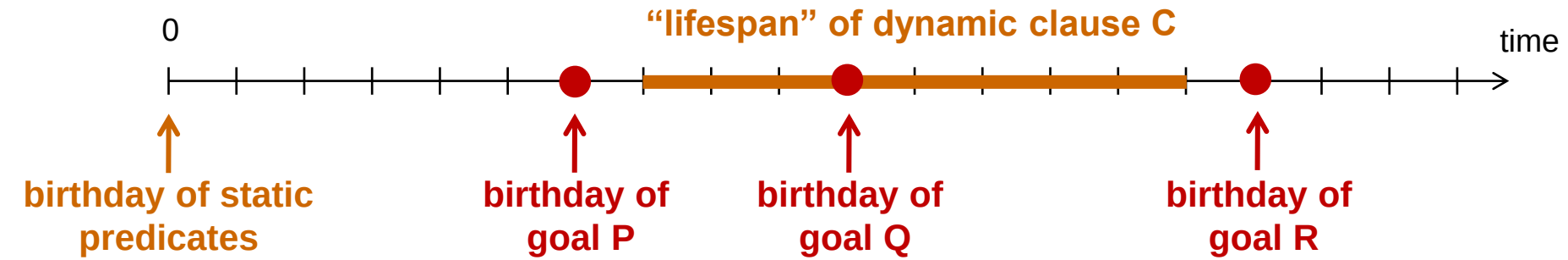
- Problem
  - Non-termination if the update affects a predicate used in the derivation

Deferred Update Semantics

- Change not visible to already executing goals!

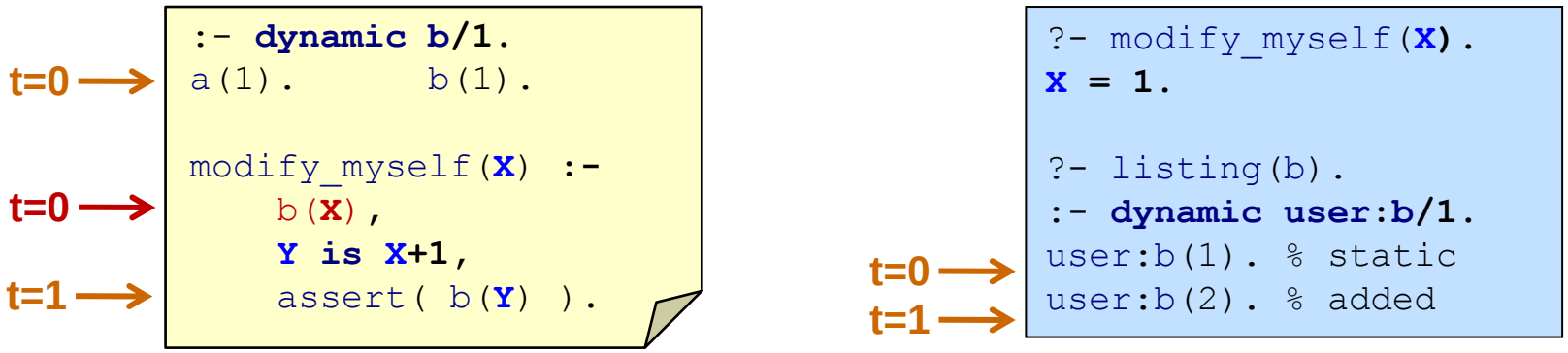
Lifespan model (Part of ISO standard for Prolog)

- Discrete time
  - 1 change of the clause database = 1 tick of the clock
- Clause
  - birthday = time when it is asserted; death = time when it is retracted
- Goal
  - birthday = time when it starts executing
- Visibility
  - Goal with birthday t only sees clauses whose lifespan contains t.

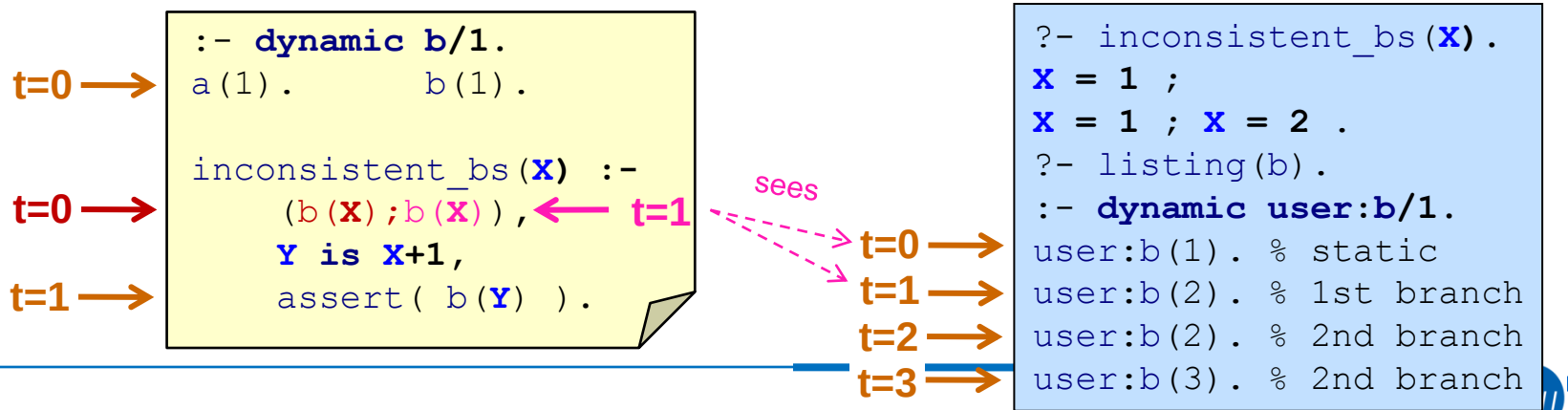


Deferred Update Semantics ▸ Examples

- Change not visible to already executing goals!



- Predicates used in the derivation do not see changes ☺
- ... but only if they are in a disjunctive branch that starts before the change
  - Predicates in different disjunctive branches behave inconsistently! ☹



Deferred Update Semantics ▸ Observations

- The non-modifying parts succeed for the substitutions { {X ←1, Y←2} }

same !

```
:- dynamic b/1.
b(1).

modify_myself(X) :-
    b(X),
    Y is X+1,
    assert( b(Y) ).
```

```
:- dynamic b/1.
b(1).

inconsistent_bs(X) :-
    (b(X);b(X)),
    Y is X+1,
    assert( b(Y) ).
```

- The different overall behaviour results from interference of deduction and change during backtracking

```
?- modify_myself(X).
X = 1.

?- listing(b).
:- dynamic user:b/1.
user:b(1). % static
user:b(2). % added
```

```
?- inconsistent_bs(X).
X = 1 ;
X = 1 ; X = 2 .
?- listing(b).
:- dynamic user:b/1.
user:b(1). % static
user:b(2). % 1st branch
user:b(2). % 2nd branch
user:b(3). % 2nd branch
```

Summary: Prolog Update Dilemma

- Immediate Update Semantics
  - Non-terminating
- Deferred Update Semantics
  - Trades termination for semantic inconsistencies
  - Does not defer update, just defers making it visible
  - Some goals see the update some don't
  - ➢ Multiple occurrences of the same goal can have different semantics

Really Deferred Update Semantics

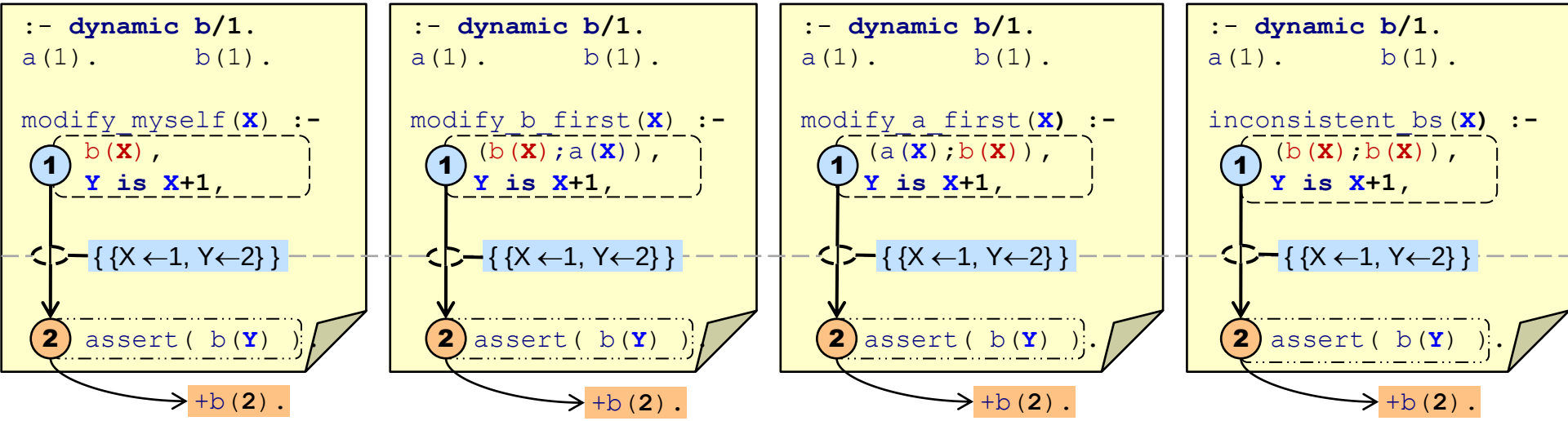
Separation of Deduction and Change

Change Sequences

Change in Multiple Clauses

Really Deferred Update Semantics ▸ Principle and Benefits

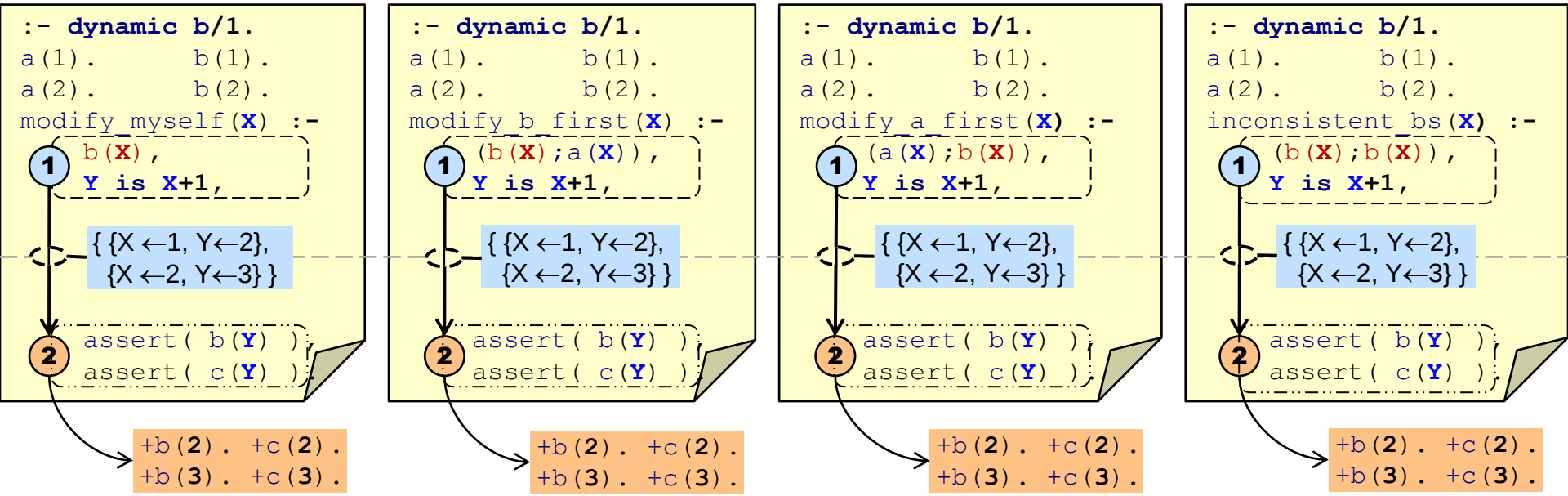
- Principle
  - Deduction first (all successful substitutions!)
  - Then change (all changes implied by the set of successful substitutions!)



Same declarative semantics of deductive part (same substitution set) ⇒ ⇐ Same declarative semantics of transformation part (same transformation set)

Really Deferred Update Semantics ▸ Principle Generalization

- Principle holds even for
  - multiple results of deduction part (non-singleton substitution set)
  - multiple transformations in change part




```
?- modify_...(X).
X = 1 ;
X = 2 .
```

```
?- listing(b).
:- dynamic user:b/1.
user:b(1).   % static
user:b(2).   % static
user:b(2).   % added
user:b(3).   % added
```

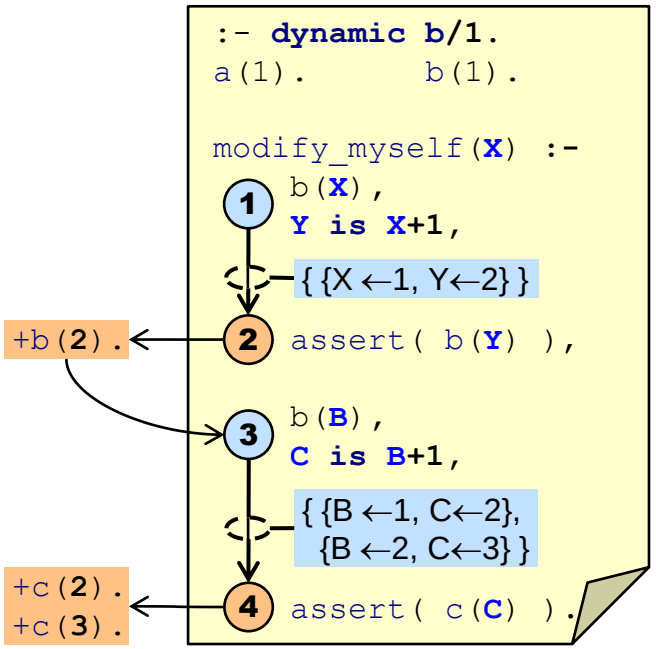
```
?- listing(c).
:- dynamic user:c/1.

user:c(2).   % added
user:c(3).   % added
```

Really Deferred Update Semantics ▸ Sequences

- Question
  - What if the update is not the last element of a clause?
- Answer
  - Sequence of Deduction/Change blocks
  - The changes of each block are visible as soon as they are performed
  - In particular, they are visible to the deduction part of the next one!

Non-Propagating Sequence

- Blocks do not share variables
- Only share the clause database

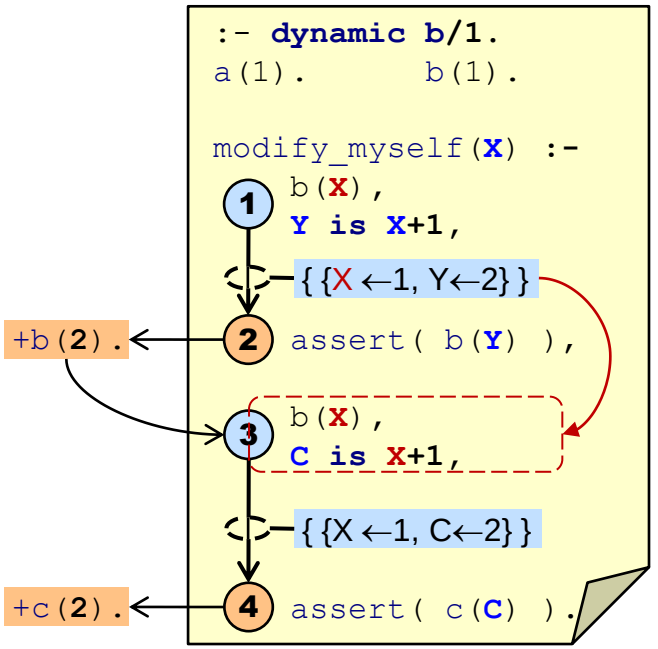


Really Deferred Update Semantics ▸ Sequences

- Question
  - What if the update is not the last element of a clause?
- Answer
  - Sequence of Deduction/Change blocks
  - The changes of each block are visible as soon as they are performed
  - In particular, they are visible to the deduction part of the next one!

Propagating Sequence

- Blocks do share variables
- ... and the clause database

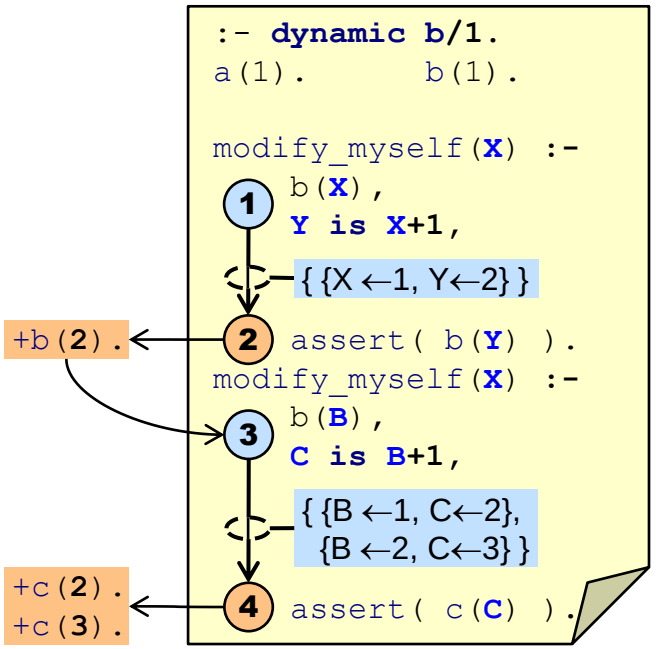


Really Deferred Update Semantics ▸ Sequences

- Question
  - What if updates are contained in different clauses of a predicate?
- Answer
  - Semantically the same as a non-propagating sequence!

Clauses

- Do not share variables
- Only share the clause database



universitätbonn ROOTS

Conditional Transformations (CTs)

– A Language with Really Deferred Update Semantics –

From the Really Deferred Update Semantics concepts to the CT language

How to implement Really Deferred Update Semantics?

Language and Interpreter

Examples

From the RDUS Principles to the CT Language

Idea

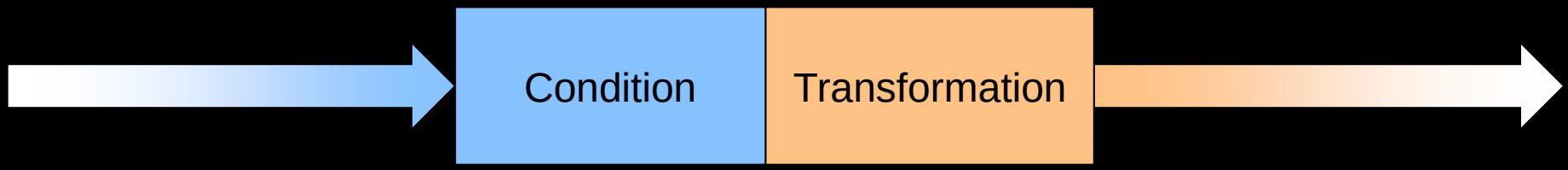
- Separate deduction and change
  - Deduction first
  - Then change

Language

- Conditional Transformation (CT)
  - Condition
  - Transformation

Principle

Condition is true for a set of substitutions ⇒ Run Transformation for those substitutions



Conditional Transformations (CTs)

- CT = A fact ct/3 or ct/4
  - Arg1 = Head
  - Arg2 = condition((Cond))
  - Arg3 = transformation((Transf))
- Cond
  - Any Prolog goal that does not perform changes
- Transf
  - Sequence of `add`(Term) and `del`(Term) operations or `skip`

Our Example as a CT

- modify_myself(X)

```
:- dynamic b/1.
a(1).      b(1).

ct( modify_myself(X) ,
    condition((
       b(X),
       Y is X+1
    )),
                          {{X ←1, Y←2}}
    transformation((
       add( b(Y) ),
    ))
).
```

1

2

+b(2).

```
?- execCTS( ct( modify_myself(X) ) ).
X = _G1100.

?- listing(b).
:- dynamic user:b/1.
user:b(1).                    % static
user:b(2).                    % added
```

„Encapsulate Field“ Example

- Problem
- Problem Detector
- Problem Fix
- Integration with Eclipse

Non-Encapsulated Fields are

BAD

Detector for Non-Encapsulated Fields

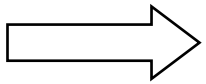
```
non_private_field(Class,Field,FieldType,FieldName,Modif) :-
    fieldT(Field,Class,FieldType,FieldName,_),
    modifierT(Field,Modif),
    ( Modif = public
    ; Modif = package
    ; Modif = protected
    ).


field_without_getter(Field,Class,Type,Name,Getter) :-
    non_private_field(Class,Field,Type,Name,_Modif),
    concat(get, Name, Getter),
    % No method with signature "Type Getter()" :
    not( methodT(_Meth,Class,Getter,[],Type,_,_) ).
```

- Suppose, we find 231 non-encapsulated fields!
- Would you bother to encapsulate them one by one?

Example: Create Accessor Method

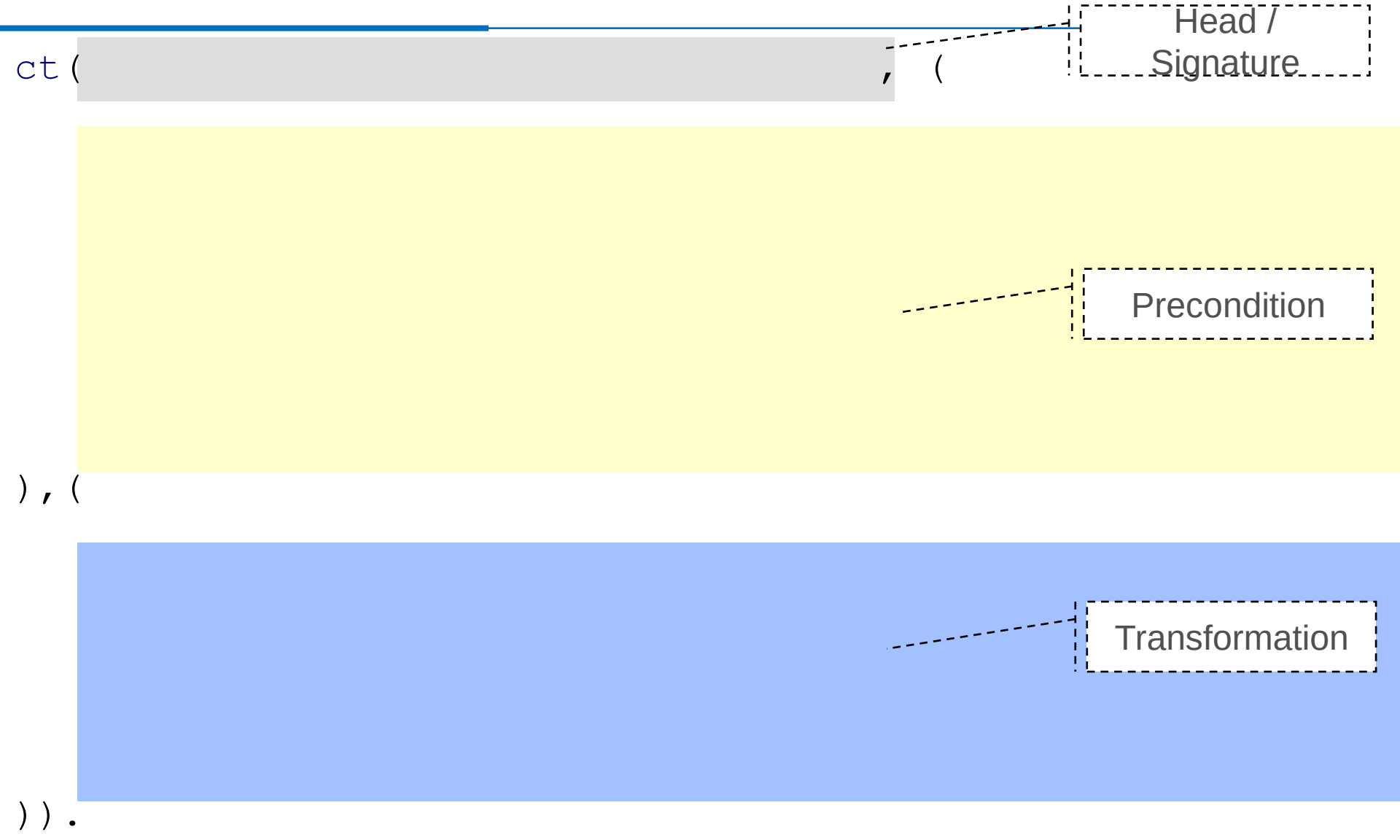
- „AddGetter“ CT
  - for all fields that have no getter method ...
  - ... add method that returns the field's value

```
public class C {

  B b = new B();




  ...

}
```

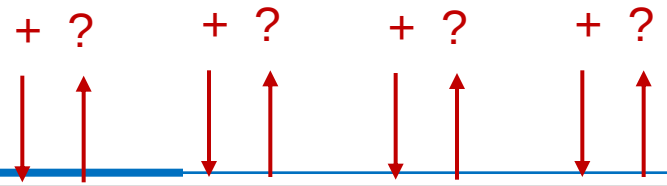


```
public class C {

  B b = new B();

  B getB() {
     return b;
  }

   ...

}
```

The Structure of a CT



The CT



No method with signature "<Type> get<Name>()" exists.

```
ct( addGetter(Class,Field,Type,GName) , (
    classT(Class,_,_,_),
    fieldT(Field,Class,Type,Name,_),

    concat(get, Name, GName),
    not( methodT(Getter,Class, GName,[],Type,_,_),
         getFieldT(_, _, Getter, _, _, Field) ),
    new_id(Method),...,new_id(Get)
),(
    add( methodT(Method,Class, GName,[],Type,[],Block) ),
    add( blockT(Block,Method,Method,[Return]) ),
    add( returnT(Return,Block,Method,Get) ),
    add( getFieldT(Get,Return,Method,null,Name,Field) ),
    add_to_class(Class,Method)
)).
```

Create new identities for new elements:

Create method "<Type> get<Name>() { return <Field>}":

From the RDUS Principles to the CT Language

Principles

- Separate deduction and change
  - Deduction first
  - Then change
- Non-propagating sequence
  - Seq. of deduction/change blocks that share no variables
  - Seq. of clauses
- Propagation sequence
  - Seq. of deduction/change blocks that share variables
- Bubbles principle
  - Any predicate that directly or indirectly performs a change is treated as a transformation

Language terminology

- Conditional Transformation (CT)
  - Condition
  - Transformation
- OR sequence
  - Seq. of CTs
  - They may share variables but shared variables are ignored
- PROP sequence
  - Seq. of CTs that share variables
- CTSEQ(Head, Seq)
  - CTs or CT Sequences with name and parameters

How to implement Really Deferred Update Semantics (ReDUS)?

Change SWI-Prolog?

- Complex system
  - implemented in C
  - 30 years of development
- Complex modification
  - change core concepts: backtracking!
- Would affect all users
  - possibly breaks old programs
- Would be too specific
  - does not help users of other Prolog implementations (YAP, Quintus, ...)

Implement metainterpreter!

- Relatively easy
  - implementation in Prolog
  - simple principles
- Only change what you need to
  - Reuse basic interpreter for deductive part
- Only affects those who use it
  - Users may decide themselves
- General solution
  - Portable to every Prolog system

Non-Propagating Sequence: OR-Sequence

- Named Sequence = a fact ctseq(Head,Seq)
  - Seq = Any CT sequence
- OR Seq = A term orseq/2
  - Arg1 = First Subsequence
  - Arg2 = Second Subsequence

```
:- dynamic b/1.
a(1).       b(1).

ct( create_bs(X) ,
    condition(( a(X), Y is X+1 )),
    transformation(( add(b(Y)) ))
).

ct( create_cs(X) ,
    condition(( b(X), Y is X+1 )),
    transformation(( add(c(Y)) ))
).

ctseq( do_both(X),
       orseq(ct(create_bs(X)),
             ct(create_cs(X)))
).
```

```
?- execCTS( ctseq( do_both(X) ) ).
X = _G1100.

?- listing(b).
:- dynamic user:b/1.
user:b(1).              % static
user:b(2).              % added for a(1)

?- listing(c).
:- dynamic user:c/1.
user:c(2).              % added for b(1)
user:c(3).              % added for b(2)
```

Propagating Sequence: PROP-Sequence

- Named Sequence = a fact ctseq(Head,Seq)
  - Seq = Any CT sequence

```
:- dynamic b/1.
a(1).       b(1).

ct( create_bs(X) ,
    condition(( a(X), Y is X+1 )),
    transformation(( add(b(Y)) ))
).

ct( create_cs(X) ,
    condition(( b(X), Y is X+1 )),
    transformation(( add(c(Y)) ))
).

ctseq( do_both_prop(X),
       propseq(ct(create_bs(X)),
               ct(create_cs(X)))
).
```

- PROP Seq = A term propseq/2
  - Arg1 = First Subsequence
  - Arg2 = Second Subsequence

```
?- execCTS( ctseq( do_both_prop(X) ) ).
X = _G1100.

?- listing(b).
:- dynamic user:b/1.
user:b(1).          % static
user:b(2).          % added for a(1)

?- listing(c).
:- dynamic user:c/1.
user:c(2).          % added for b(1)
```

CT Language Overview

- See https://sewiki.iai.uni-bonn.de/research/ctc/language and

- @PhDThesis{kniesel:habil:2012,

```
    Type      = {Habilitation},
    Author    = {G{\"u}nter Kniesel},
    Title     = {{Logic-Based Software Analysis and Transformation –
                  Foundations, Tools and Applications}},
    School    = {CS Dept. III, University of Bonn, Germany},
    Note      = {(unpublished manuscript)},
    Year      = {2012}
}
```

Summary

- Immediate Update Semantics
  - Updates performed and visible immediately
  - Non-terminating
- Deferred Update Semantics
  - Updates performed immediately but visible only to “older” goals
  - Inconsistent semantics of same goal in different disjunctive branches
- Really Deferred Update Semantics
  - Separate derivation and update
  - Defer updates until all backtracking of the derivation is finished
- Conditional Transformations
  - A logic language based on the Really Deferred Update Semantics
  - Generic implementation as a metainterpreter