Logic-Based Program Transformation
- From assert/retract to CTs -

20.07.2017

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

universitätbonn

Copyright Notice

This chapter contains unpublished work.

The section about the "Really Deferred Update Semantics" and about the "Conditional Transformation Language (CTL)" 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 chapter or parts of it is strictly prohibited except 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).
...
```

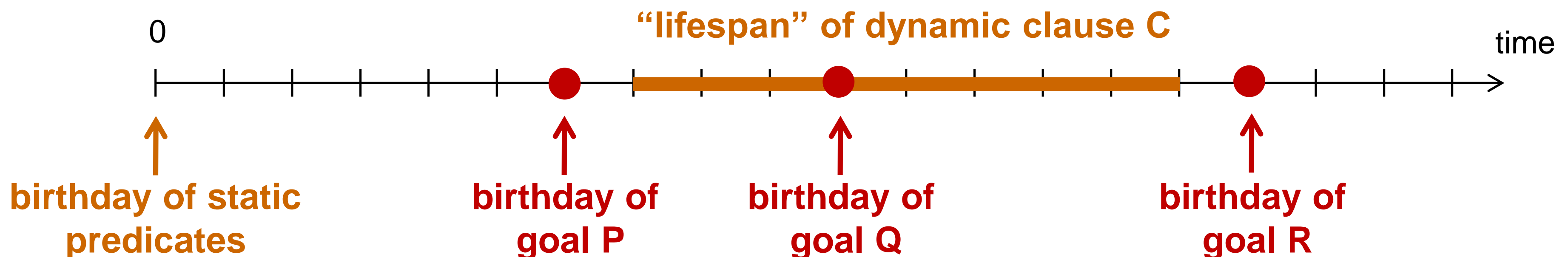
- Problem
  - Non-termination if the update affects a predicate used in the derivation

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!

```
t=0 →  :- dynamic b/1.
       a(1).       b(1).

       modify_myself(X) :-
t=0 →      b(X),
           Y is X+1,
t=1 →      assert( b(Y) ).
```

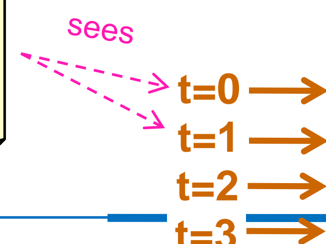
```
?- modify_myself(X).
X = 1.

?- listing(b).
:- dynamic user:b/1.
t=0 → user:b(1). % static
t=1 → user:b(2). % added
```

- 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! ☹

```
       :- dynamic b/1.
t=0 →  a(1).       b(1).

       inconsistent_bs(X) :-
t=0 →      (b(X);b(X)), ← t=1
           Y is X+1,
t=1 →      assert( b(Y) ).
```




```
?- inconsistent_bs(X).
X = 1 ;
X = 1 ; X = 2 .
?- listing(b).
:- dynamic user:b/1.
t=0 → user:b(1). % static
t=1 → user:b(2). % 1st branch
t=2 → user:b(2). % 2nd branch
t=3 → user:b(3). % 2nd branch
```

Deferred Update Semantics ▸ Observations

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

```
:- 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

Can we do better?

Really Deferred Update Semantics (RDUS)

Separation of Deduction and Change

Change Sequences

Change in Multiple Clauses

“Bubbles” principle

Really Deferred Update Semantics (RDUS) ▸ Principle and Benefits

- Idea: Since interference of deduction and change during backtracking is the cause of the problem, separate deduction and change!
  - Deduction first (Find all successful substitutions!)
  - Then change (Do all changes implied by the set of successful substitutions!)
- Examples:
  - Note that the following programs only differ in the highlighted line and that this line yields the same substitutions for X in all four cases:

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

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

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

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

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

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

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

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

  - We shall use these examples to demonstrate that the above principle yields identical operational behavior (that is, identical program updates) for programs that have the same declarative semantics (that is, identical substitution sets).

Really Deferred Update Semantics (RDUS) ▸ Principle and Benefits

- Principle: “Separate deduction and change!”
  - Deduction first (Find all successful substitutions!)
  - Then change (Do all changes implied by the set of successful substitutions!)

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

modify_myself(X) :-
 1  b(X),
    Y is X+1,
    { {X←1, Y←2} }
 2  assert( b(Y) ).
                     +b(2).
```

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

modify_b_first(X) :-
 1  (b(X);a(X)),
    Y is X+1,
    { {X←1, Y←2} }
 2  assert( b(Y) ).
                     +b(2).
```

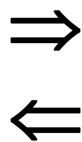
```
:- dynamic b/1.
a(1).      b(1).

modify_a_first(X) :-
 1  (a(X);b(X)),
    Y is X+1,
    { {X←1, Y←2} }
 2  assert( b(Y) ).
                     +b(2).
```

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

inconsistent_bs(X) :-
 1  (b(X);b(X)),
    Y is X+1,
    { {X←1, Y←2} }
 2  assert( b(Y) ).
                     +b(2).
```

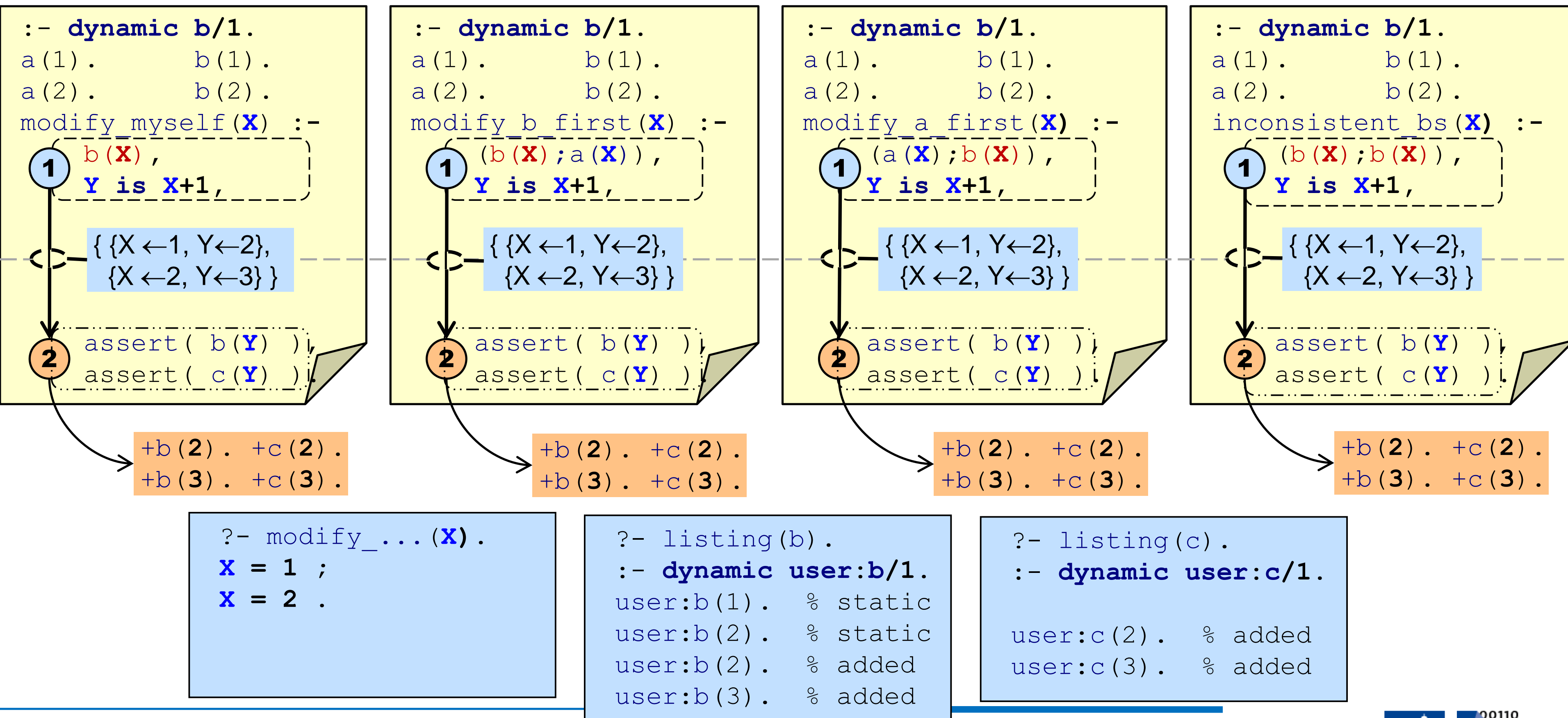
Same declarative semantics of deductive part (same substitution set)



Same declarative semantics of transformation part (same transformation set)

Really Deferred Update Semantics (RDUS) ▸ Principle Generalization

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

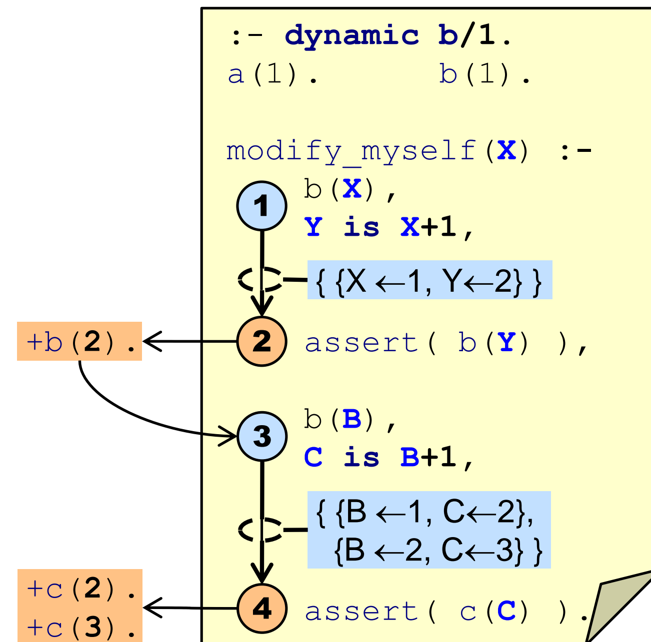


Really Deferred Update Semantics ▸ Sequences

- 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

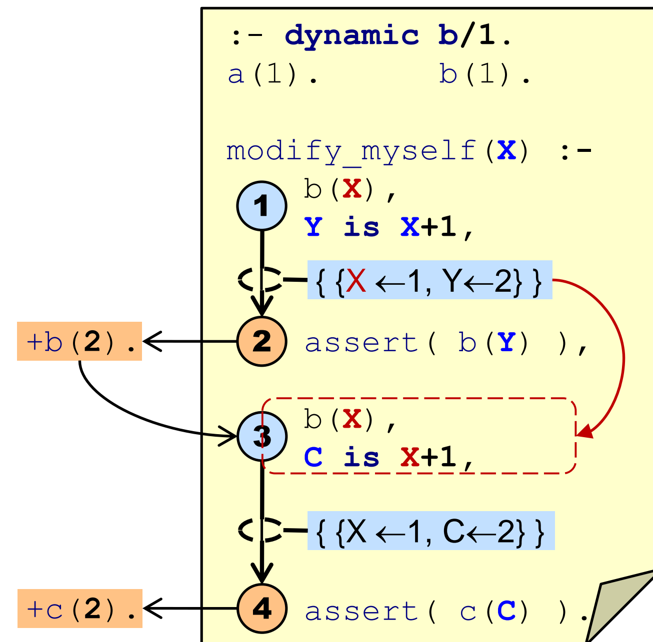


Really Deferred Update Semantics ▸ Sequences

- 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

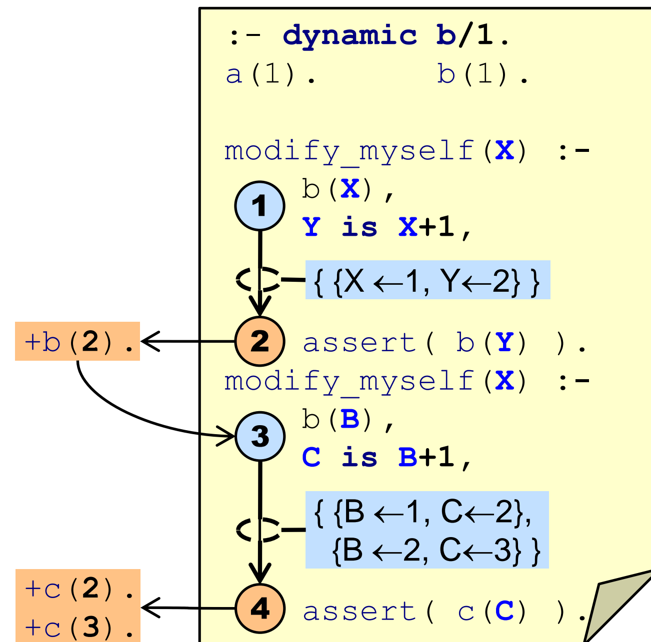


Really Deferred Update Semantics ▸ Sequences

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

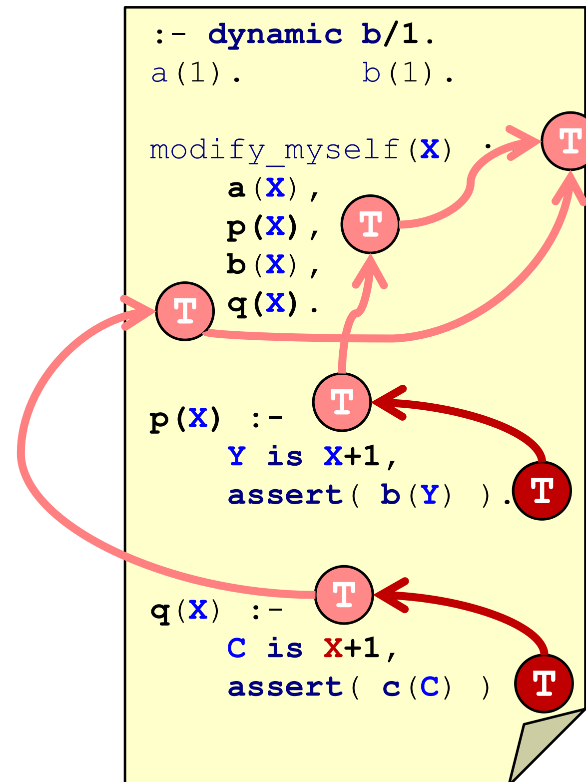
Clauses

- Do not share variables
- Only share the clause database



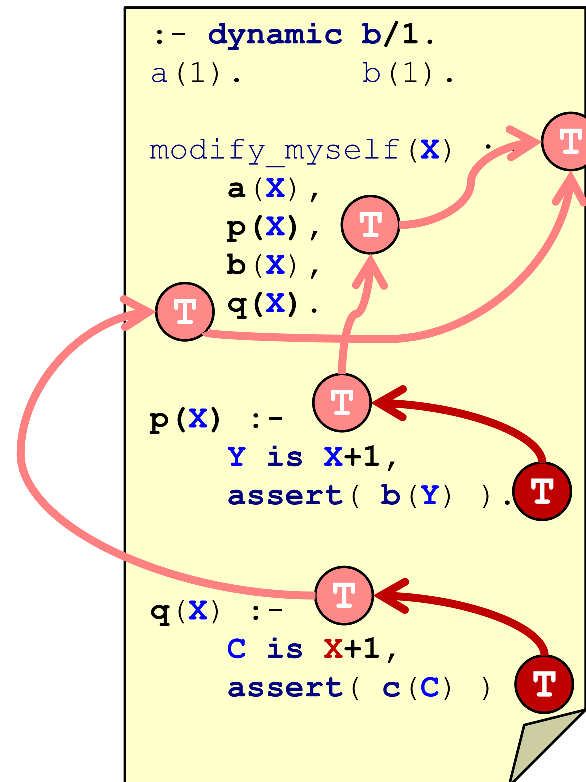
Really Deferred Update Semantics ▸ „Bubbles Principle“

- Bubbles in a glas of liquid rise to the surface
- Transformations nested deeply in a predicate rise to the surface too – if the predicate contains a nested transformation, it is itself a transformation.
  - Since p/1 and q/1 contain each an assert/1 call they are also transformations
  - Since p/1 and q/1 are transformations, modify_myself/1 is also a transformation.



Really Deferred Update Semantics ‣ „Bubbles Principle“

- Bubbles in a glas of liquid rise to the surface
- All calls of transformations split the containing predicate into a preceding declarative part and a transformation part:
  - Modify_myself/1 contains four parts
    - ⇨ a and b are purely deductive
    - ⇨ p and q are transformations
  - p/1 contains two parts
    - ⇨ is/2 is purely deductive
    - ⇨ assert/1 is the transformation
  - q/1 has the same structure as p/1



RDUS Summary

- Uniform principle: „Separate deduction and change!“

- Generalisation for
  - Multiple substitutions
  - Multiple changes
  - Multiple deduction / change sequences within a clause
  - Multiple clauses
  - Nested transformations („Bubbles principle“)

- But Prolog interpreters / compilers do not work that way!

- How to get the desired behaviour?

→ Own logic-based transformation language: Conditional Transformations

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 Principle to the CT Language

Idea

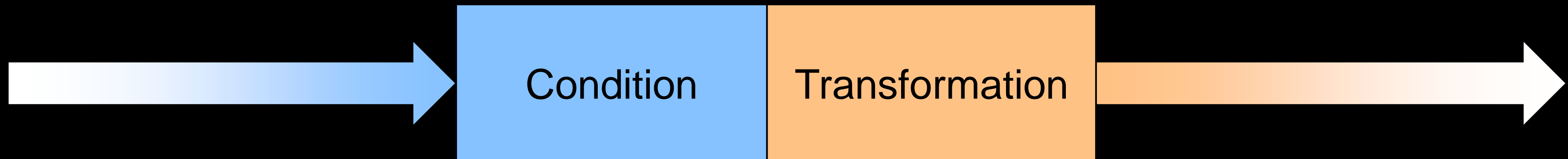
- Separate deduction and change
  - Deduction first
  - Then change

Language

- Conditional Transformation (CT)
  - Condition
  - Transformation

Principle

Condition is true for a set of substitutions $\Rightarrow$ Run Transformation for those substitutions



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, ...)

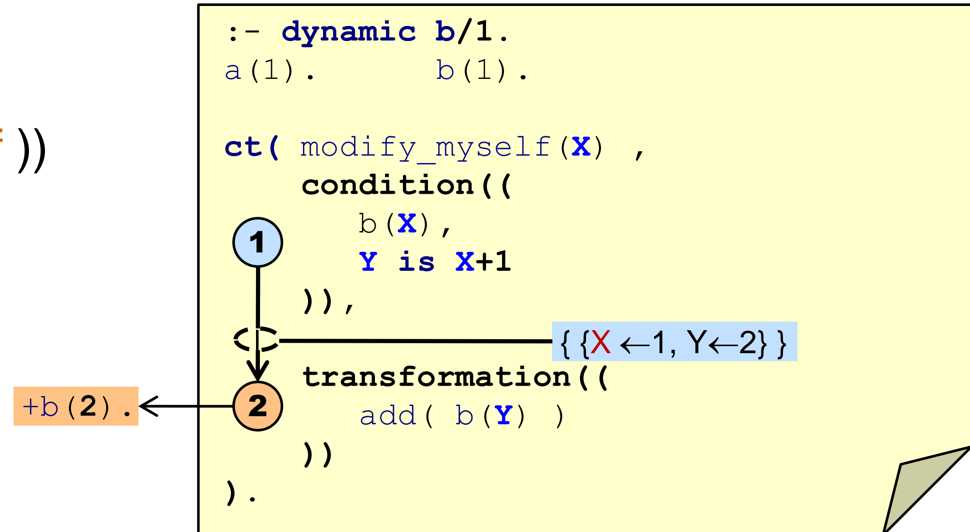
Implement metainterpreter!

- 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

Conditional Transformations ‣ Syntax

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

Declare modify_myself(X) as CT:



Execute it via metainterpreter:

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

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

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
- PROP Seq = A term propseq/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_prop(X),
       propseq(ct(create_bs(X)),
               ct(create_cs(X)))
).
```

```
?- 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)
```

From the RDUS Principles to the CT Language ▶ Summary

Principles

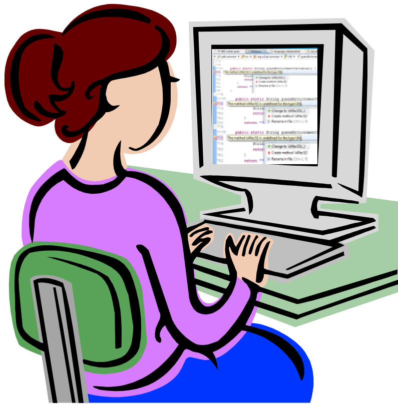
- Separate deduction and change
  - Deduction first
  - Then change
- Non-propagating sequence
  - Seq. of deduction/change blocks that share no variables
  - Seq. of clauses
- Propagation sequence
  - Seq. of deduction/change blocks that share variables
- "Bubbles" principle
  - Any predicate that directly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

Language terminology

- Conditional Transformation (CT)
  - Condition
  - Transformation
- OR sequence
  - Seq. of CTs
  - Shared variables are ignored
- PROP sequence
  - Seq. of CTs
  - Substitutions are propagated through shared variables
- CTSEQ(Head, Seq)
  - CTs or CT Sequences with name and parameters

Recall Chapter 6 ▸ Software Development Assistants

- Example: „Monitor, advise, act“ cycle in the Eclipse IDE



```
776
777    public static String guessEnvironmentV
778    The method isMacS() is undefined for the type Util
779          Strin   Change to 'isMacOS(..)'
780          retur   Create method 'isMacS()'
781        }         Rename in file (Ctrl+2, R)
782        return ""
```

- Behind the scenes: Software Analysis and Transformation (SAT)

- Research goal: Make SAT development quick and easy
  - Focus on conceptual essence of SAT tasks
  - … not on APIs, implementation, manual tuning, …

„Java String Pitfall“ Example

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

Problem ▸ Java String Pitfall

- The == operator compares object identities ☹
- Java strings must be compared with the „equals()" method.

```
public void startElement(String uri, String localName,
      String qName, Attributes attributes) throws SAXException {

   if(localName == "key"){
      …
   } else if(localName == "dict"){
      …
   }
}
```

➢ Can we create a detector for this?

universitätbonn

Java String Pitfall ▸ Detection

- Detecting identity comparison operation ('==') on string arguments:

```
identity_comparison_on_strings(Operation) :-
    operationT(Operation, Parent, Enclosing, [Lhs, Rhs], '==', 0),
    fully_qualified_name(StringType, 'java.lang.String'),     % JT
    get_type(Lhs, StringType),                                % JT
    get_type(Rhs, StringType).                                % JT
```

- Register as smell detector with JTransformer:

See
https://sewiki.iai.uni-bonn.de/research/jtransformer/tutorial/analysis_cc

Java String Pitfall ▸ Fixing it with a CT

- Find each bad "==" operation and replace it with a call to the equals() method from type String:

```
ct(
   identity_comparison_on_strings(Operation),
  ( identity_comparison_on_strings(Operation),
    operationT(Operation, Parent, Encl, [Lhs, Rhs], '==', 0),
    get_type(Lhs, StringType),
    methodT(Equals, StringType, 'equals', _, _, _, _),

  ),
  ( delete(operationT(Operation,Parent,Encl,[Lhs,Rhs], '==', 0)),
    add(callT(Operation, Parent, Enclosing, Lhs, 'equals', [Rhs], Equals))
  )
).
```

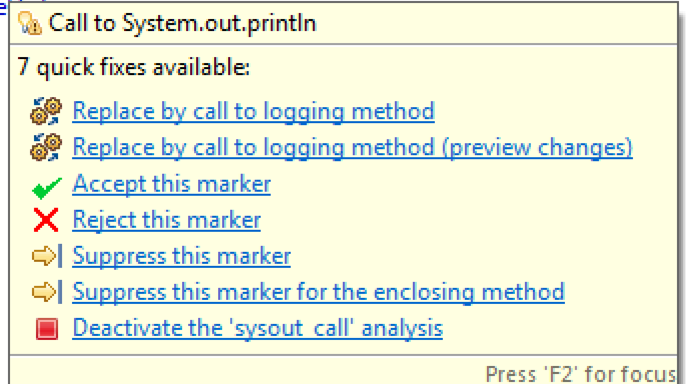

universitätbonn

Java String Pitfall ▸ Fixing it with a CT (2)

- Register the fix to JTransformer

https://sewiki.iai.uni-bonn.de/research/jtransformer/tutorial/quickfix

➢ Automatic registration with Eclipse Quick-Fix Framework



Chapter 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

© 2009-2017 Dr. G. Kniesel

Running CTs (and CT Sequences)

Call the CT metainterpreter:

```
?- execCTS(Term).
```

The execCTS metainterpreter implements the full CT language (CTL):

- ct(Head) → call CT whose head unifies with Head
- ctseq(Head) → call named CT sequence whose head unifies with Head
- orseq(T1,T2) → execute unnamed OR-Sequence. T1 and T2 can be any legal CTL term.
- propseq(T1,T2) → execute unnamed PROP-Sequence. T1 and T2 can be any legal CTL term.
- … and more …

Running CTs (and CT Sequences) in JTransformer

Call the CT metainterpreter:

```
?- execCTS(Term).
```

The execCTS metainterpreter implements the full CT language (CTL):

- ct(Head) → call CT whose head unifies with Head
- ctseq(Head) → call named CT sequence whose head unifies with Head
- orseq(T1,T2) → execute unnamed OR-Sequence. T1 and T2 can be any legal CTL term.
- propseq(T1,T2) → execute unnamed PROP-Sequence. T1 and T2 can be any legal CTL term.
- … and more …