Introduction to ASF+SDF

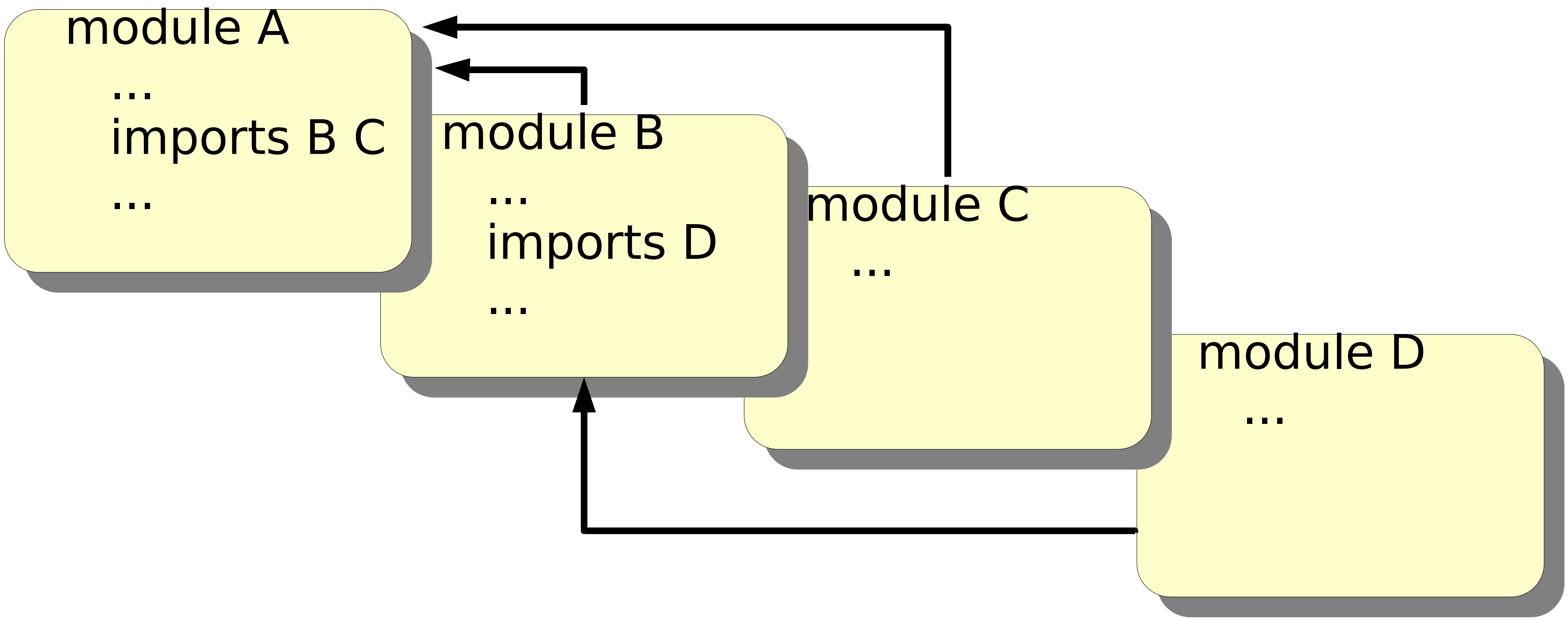
Mark van den Brand,
Paul Klint,
Jurgen Vinju

Universiteit van Amsterdam

ASF+SDF

- Goal: defining languages & manipulating programs
- SDF: Syntax definition Formalism
  - lexical syntax: keywords, comments, constants
  - context-free syntax: declarations, statements
- ASF: Algebraic Specification Formalism
  - static semantics: type checks
  - dynamic semantics: running a program
- ASF+SDF Meta-Environment User Manual:
  www.meta-environment.org

Anatomy of an ASF+SDF Specification



Anatomy of an ASF+SDF Module

```
module ModuleName
  ImportSection*
  ExportOrHiddenSection*
equations
  ConditionalEquation*
```

Name of this module; may be followed by parameters

Names of modules imported by this module; May be followed by renamings

Grammar elements that are visible from the outside (exports) or only inside the module (hiddens)

imports, aliases, sorts, lexical syntax, context-free syntax, priorities, variables

Unconditional: [tagId] L = R

Conditional: [tagId] C_1, C_2, ..., C_n

```
=============
     L = R
```

Plan

- Booleans
- Steps towards a Pico environment
  - Step 1: define syntax
  - Step 2: define a typechecker
  - Step 3: define an evaluator
  - Step 4: define a compiler
- Traversal functions

Plan

- *Booleans*
- Steps towards a Pico environment
  - Step 1: define syntax
  - Step 2: define a typechecker
  - Step 3: define an evaluator
  - Step 4: define a compiler
- Traversal functions

Booleans (1)

```
module basic/Booleans

exports
  sorts BoolCon
  context-free syntax
    "true"    -> BoolCon
    "false"   -> BoolCon
```

The sort of Booleans constants, sorts should always start with a capital letter

The constants `true` and `false`, literals should always be quoted

Booleans (2)

```
sorts Boolean

context-free start-symbols
   Boolean

context-free syntax
  BoolCon  -> Boolean
```

The sort of Boolean expressions

The start symbol of a grammar. Without start symbol the parser does not know how to start parsing an input sentence

Each Boolean constant is a Boolean expression.
Also-called injection rule or chain rule

Booleans (3)

```
Boolean "|" Boolean   -> Boolean {left}
Boolean "&" Boolean  -> Boolean {left}
"not"(Boolean)         -> Boolean
"(" Boolean ")"         -> Boolean {bracket}

context-free priorities
 Boolean "&" Boolean -> Boolean >
 Boolean "|" Boolean -> Boolean
```

The infix operators and & and or |. Both are left-associative (left)

The prefix function not

(and) may be used as brackets in Boolean expressions; they are ignored after parsing

& has higher priority than |
Example: Bool & Bool | Bool
is interpreted as:
(Bool & Bool) | Bool

Booleans (4)

```
hiddens
  imports
    basic/Comments
    basic/Whitespace
  variables
    "Bool"[0-9\']*  -> Boolean

 equations

     [B1]   true | Bool   = true
     [B2]   false | Bool  = Bool
     [B3]   true & Bool    = Bool
     [B4]   false & Bool  = false
     [B5]   not ( false ) = true
     [B6]   not ( true )  = false
```

Import the standard comment and whitespace conventions for equations

Declares the variables Bool, Bool1, Bool2, Bool', Bool'', Bool1', etc.

The meaning of &, | and not operators.

Point to ponder: the syntax of equations is not fixed but depends on the syntax definition of the functions.

Booleans (5)

The term

```
not(true & not(false | true))
```

Rewrites to

```
true
```

Booleans (6)

not(true & not(false | true))

[B2] false | Bool = Bool

not(true & not(true))

[B6] not (true) = false

not(true & false)

[B3] true & Bool = Bool

not(false)

[B5] not(false) = true

true

Booleans (7)

- Each module defines a language; in this case the language of Booleans (synonym: datatype)
- We can use this language definition to
  - Create a syntax-directed editor for the Boolean language and create Boolean terms
  - Apply the equations to this term and reduce it to normal form
  - Import it in another module; this makes the Boolean language available for the importing module

Plan

- Booleans
- *Steps towards a Pico environment*
  - Step 1: define syntax
  - Step 2: define a typechecker
  - Step 3: define an evaluator
  - Step 4: define a compiler
- Traversal functions

The Toy Language Pico

- Pico has two types: natural number and string
- Variables have to be declared
- Statements: assign, if-then-else, while-do
- Expressions: natural, string, +, - and ||
- + and - have natural operands and the result is natural
- || has string operands and the result is string
- Tests (if, while) should be of type natural

A Pico Program

```
begin declare input : natural,
                      output  :  natural,
                      repnr: natural,
                     rep: natural;
          input := 14;
          output := 1;
          while input - 1 do
              rep := output;
              repnr := input;
              while repnr - 1 do
                 output := output +
rep;
                 repnr := repnr - 1
              od;
              input := input - 1
         od
end
```

input value

output value

What does this program compute?

14! = 14 * 13 * ... * 1

Why is it written in this clumsy style?

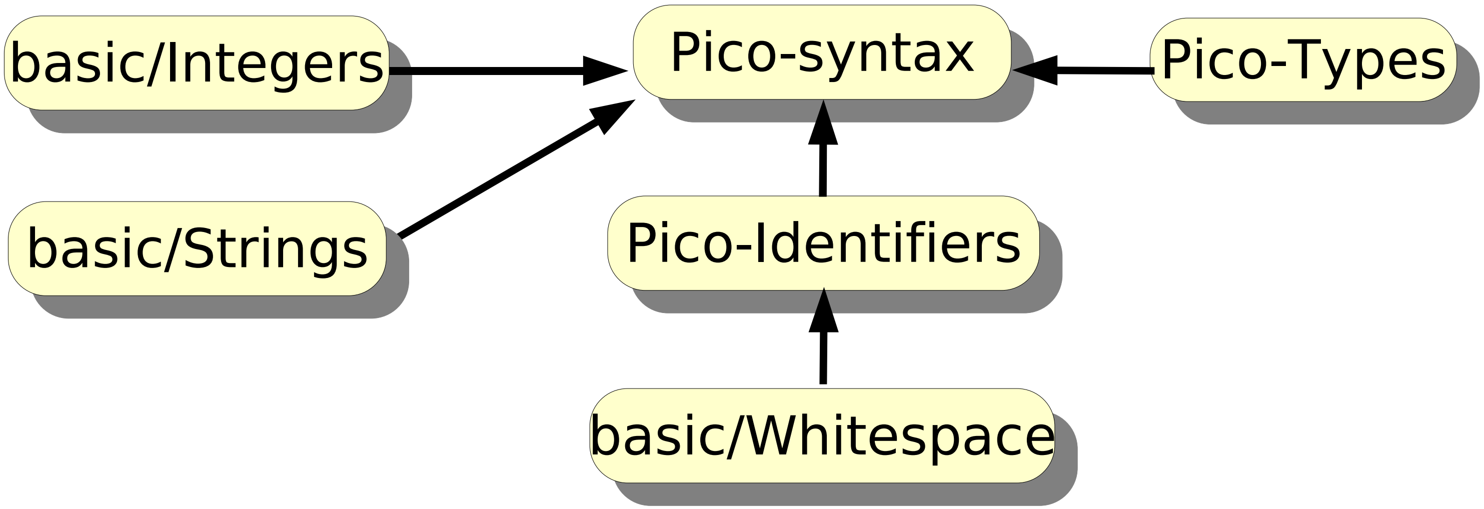
(a) Pico has no input/output statements

(b) Pico has no multiplication operator

Plan

- Booleans
- Steps towards a Pico environment
  - *Step 1: define syntax*
  - Step 2: define a typechecker
  - Step 3: define an evaluator
  - Step 4: define a compiler
- Traversal functions

Step 1: Define syntax for Pico



Pico-syntax, 1

```
module Pico-syntax
  imports
     Pico-Identifiers basic/Integers basic/Strings Pico-Types
  exports
    sorts
      PROGRAM DECLS ID-TYPE STATEMENT EXP
    context-free start-symbols
       PROGRAM
    context-free syntax
      "begin" DECLS {STATEMENT ";"}* "end"      -> PROGRAM
      "declare" {ID-TYPE  ","}* ";"              -> DECLS
      PICO-ID ":" TYPE                          -> ID-TYPE
      PICO-ID ":=" EXP                           -> STATEMENT
      "if" EXP "then" {STATEMENT ";"}*
           "else" {STATEMENT ";"}* "fi"          -> STATEMENT
      "while" EXP "do" {STATEMENT ";"}* "od"     -> STATEMENT
```

Imported modules

Declared sorts

Start symbols

Syntax rules for program, declarations and statements

List of zero or more statements separated by ;
* zero or more
\+ one or more

Pico-syntax, 2

```
PICO-ID            -> EXP
NatCon             -> EXP
StrCon             -> EXP
EXP "+" EXP        -> EXP {left}
EXP "-" EXP        -> EXP {left}
EXP "||" EXP       -> EXP {left}
"(" EXP ")"        -> EXP {bracket}

context-free priorities
   EXP "||" EXP -> EXP >
   EXP "-" EXP -> EXP >
   EXP "+" EXP -> EXP
```

Syntax rules for expressions

The sort NatCon is imported from basic/Integers

The sort StrCon is imported from basic/Strings

The three operators are left-associative

The priorities of the three operators, a disambiguation construct: 1 – (2 + 3), or (1 – 2) + 3 ??

Pico-Identifiers

```
module Pico-Identifiers
imports
   basic/Whitespace
exports
  sorts PICO-ID
  context-free start-symbols PICO-ID
  lexical syntax
    [a-z] [a-z0-9]* -> PICO-ID
  lexical restrictions
    PICO-ID -/- [a-z0-9]
```

Library module, explained next

Repeat zero (*) or one (+) or more times

A character class:
PICO-ID starts with a lowercase letter

Lexical restrictions, another disambiguation construct:
“aaa”, three, two or one identifier?
-/- implements a `longest match'

basic/Whitespace

```
module basic/Whitespace
exports
  lexical syntax
    [\ \t\n] -> LAYOUT

  context-free restrictions
    LAYOUT? -/- [\ \t\n]
```

Special characters are escaped: space (\ ), tabulation (\t) and newline (\n)

The sort LAYOUT has a special meaning: everything recognized as LAYOUT can be recognized between everything else

Longest match for LAYOUT

Pico-Types

```
module Pico-Types

exports
  sorts TYPE
  context-free syntax
    "natural"     -> TYPE
    "string"      -> TYPE
    "nil-type"    -> TYPE
```

The sort of possible types in a Pico program

The constants natural and string represent types as can be declared in a Pico program

The constant nil-type is used for handling error cases

Pico: factorial program

```
begin declare input : natural,
                     output  :  natural,
                     repnr: natural,
                    rep: natural;
          input := 14;
          output := 1;
          while input - 1 do
              rep := output;
              repnr := input;
              while repnr - 1 do
                 output := output +
rep;
                 repnr := repnr - 1
              od;
              input := input - 1
          od
end
```

Syntax for Pico: summary

- The modules Pico-syntax, Pico-identifiers and Pico-Types define (together with the modules they import) the syntax for the Pico language
- This syntax can be used to
  - Generate a parser that can parse Pico programs
  - Generate a syntax-directed editor for Pico programs
  - Generate a parser that can parse equations containing fragments of Pico programs

Intermezzo: Symbols (1)

An elementary symbol is:

- Literal: "abc"
- Sort (non-terminal) names: INT
- Character classes: [a-z]: one of a, b, ..., z
  - ~: complement of character class.
  - /: difference of two character classes.
  - /\: intersection of two character classes.
  - \/: union of two character classes.

Intermezzo: Symbols (2)

A complex symbol is:

- Repetition:
  - S* zero or more times S; S+ one or more
  - {S1 S2}* zero or more times S1 separated by S1
  - {S1 S2}+ one or more
- Optional: S? zero or one occurrences of S
- Alternative: S | T an S or a T
- Tuple: <S,T> shorthand for “<” S “,” T “>”
- Parameterized sorts: S[[P1, P2]]

Intermezzo: productions (functions)

- General form of a production (function):
  - S1 S2 ... Sn -> S0 Attributes
- Lexical syntax and context-free syntax are similar, but
  - Between the symbols in a production optional layout symbols may occur in the input text.
  - A context-free production is equivalent with:
    S1 LAYOUT? S2 LAYOUT? ... LAYOUT? Sn -> S0

Example: floating point numbers

```
sorts UnsignedInt SignedInt UnsignedReal Number
  lexical syntax
    [0] | ([1-9][0-9]*)                          -> UnsignedInt

    [\+\-]? UnsignedInt                          -> SignedInt

    UnsignedInt "." [0-9]+ ([eE] SignedInt)?     -> UnsignedReal
    UnsignedInt [eE] SignedInt                   -> UnsignedReal

    UnsignedInt | UnsignedReal                   -> Number
```

0 1 14 0.1 3e4 3.014e-7

00 01 04.1 3e04 3.14e-07

Intermezzo: lists, lists, lists, ...

Assume: “a” -> A

A+	a a a a			
{A “;”}+	a	a ; a	a ; a; a	a ; a; a;
(A “;”)+	a ;	a ; a;	a ; a; a;	a ; a; a
(A “;”?)+	a	a a	a ; a	a ; a;

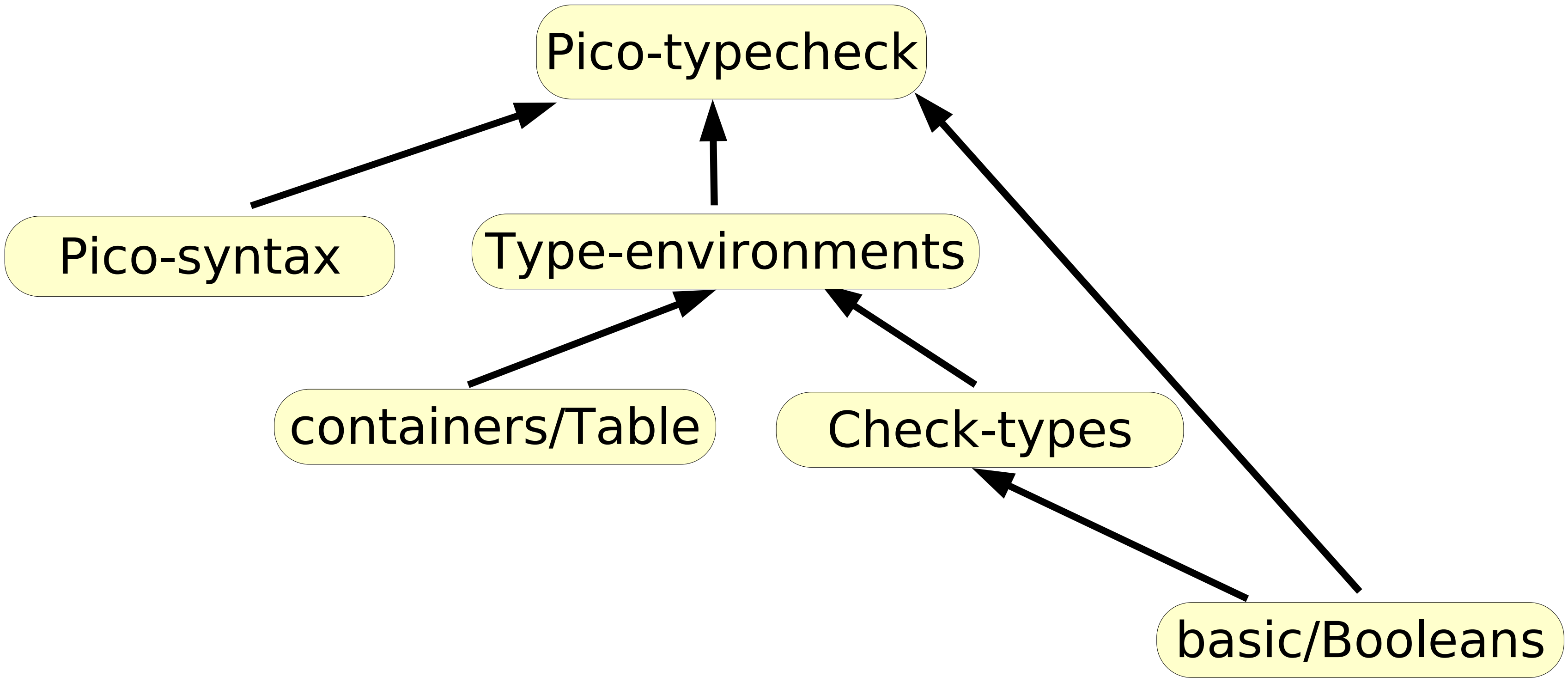
Plan

- Booleans
- Steps towards a Pico environment
  - Step 1: define syntax
  - Step 2: define a typechecker
  - Step 3: define an evaluator
  - Step 4: define a compiler
- Traversal functions

Step 2: Define typechecker for PICO

- The types are `natural` and `string`
- All variables should be declared before use
- Lhs and Rhs of assignment should have equal type
- The test in while and if-then should be natural
- Operands of + and – should be natural; result is natural
- Operands of || should be string; result string

Pico typechecker: modules



Check-types

```
module Check-types
imports Types
imports basic/Booleans basic/Comments
exports
  context-free syntax
    compatible(TYPE, TYPE) -> Boolean
hiddens
  variables
    "Type"[0-9]*  -> TYPE
equations
  [Typ1]  compatible(natural, natural) = true
  [Typ2]  compatible(string, string)   = true
  [default-Typ]
            compatible(Type1,Type2) = false
```

Check the compatibility of two types

Define the two cases of interest

Use a default equation to describe all *other cases.*

Equations are not ordered!
But: default equations are applied last.

Type-environments

```
module Type-environments

imports Check-types
    Pico-Identifiers
    containers/Table[PICO-ID TYPE]

exports
  sorts TENV
  aliases
   Table[[PICO-ID,TYPE]] -> TENV
```

Table is a parameterized library module that provides functions for managing tables of (Key, Value) pairs. Its formal parameters are Key and Value

The binding to actual parameters is:

Key $\Rightarrow$ PICO-ID
Value $\Rightarrow$ TYPE

An alias is an abbreviation.
From now on, TENV can be used instead of Table[[PICO-ID,TYPE]]

Table[Key Value]

Formal parameter Key

Formal parameter Value

```
module containers/Table[Key Value]
  imports basic/Booleans
  imports containers/List[Key]
  imports containers/List[Value]
  imports containers/List[<Key, Value>]
```

Import lists of Keys

Import lists of Values

Import lists of <Key, Value> pairs

Table[Key Value]

```
exports
  context-free syntax
    List[[<Key, Value>]]                    -> Table[[Key,Value]]

    "not-in-table"                          -> Value {constructor}
    "new-table"                             -> Table[[Key,Value]]
    lookup(Table[[Key,Value]],Key)          -> Value
    store(Table[[Key,Value]],Key,Value)     -> Table[[Key,Value]]
    delete(Table[[Key,Value]],Key)          -> Table[[Key,Value]]
    element(Table[[Key,Value]],Key)         -> Boolean
    keys(Table[[Key,Value]])                -> List[[Key]]
    values(Table[[Key,Value]])              -> List[[Value]]
```

Pico-typecheck (1)

```
module Pico-typecheck
imports basic/Booleans Pico-syntax Type-
environments
exports
  context-free syntax
    tcp(PROGRAM)                            -> Boolean
    tcd(DECLS)                              -> TENV
    tcits({ID-TYPE  ","}*, TENV)      -> TENV
    tcit(ID-TYPE, TENV)                  -> TENV
    tcs({STATEMENT ";"}*, TENV)  -> Boolean
    tcst(STATEMENT, TENV)          -> Boolean
    tce(EXP, TENV)                        -> TYPE
```

Check complete program

Check declarations by building a TENV representing the declarations

Check statements: using that TENV

Typecheck an expression using that TENV

Pico-typecheck (2)

```
hiddens
  variables
    "Decls"[0-9\']*      -> DECLS
    "Exp"[0-9\']*        -> EXP
    "Id"[0-9\']*         -> PICO-ID
    "Id-type*"[0-9\']*   -> { ID-TYPE ","}*
    "Nat-con"[0-9\']*    -> NatCon
    "Series"[0-9\']*     -> {STATEMENT ";"}+
    "Stat"[0-9\']*       -> STATEMENT
    "Stat*"[0-9\']*      -> {STATEMENT ";"}*
    "Str-con"[0-9\']*    -> StrCon
    "Tenv"[0-9\']*       -> TENV
    "Type"[0-9\']*       -> TYPE
```

Declare a bunch of variables

Pico-typecheck (3)

Check statements

Collect declarations

Visit all Id-Type pairs in declaration

```
equations
[Tc1]  tcp(begin Decls Series end) = tcs(Series, tcd(Decls))

[Tc2]  tcd(declare Id-type*;) = tcits(Id-type*,new-table)

[Tc3a] tcits(Id:Type, Id-type*, Tenv) = tcits(Id-type*, tcit(Id:Type, Tenv))

[Tc3b] tcits(,Tenv) = Tenv
```

Check list of Id-type pairs;
See next page

Pico-typecheck (4)

Comma separates arguments of tcits

List matching: decomposes a list of type { ID-TYPE ","}* into three values: the first element of the form Id:Type and the remainder of the list Id-type*

Comma in ID-TYPE list

```
[Tc3a] tcits(Id:Type, Id-type*, Tenv) = tcits(Id-type*, tcit(Id:Type, Tenv))

[Tc3b] tcits(,Tenv) = Tenv
```

Visit all declarations and treat each declaration separately

Pico-typecheck (5)

```
[Tc4a]  lookup(Id,Tenv) == nil-type
        ==========================
        tcit(Id:Type, Tenv) = store(Tenv, Id, Type)

[Tc4b]     lookup(Id,Tenv) != nil-type
           ===================
           tcit(Id:Type, Tenv) = Tenv

[Tc5a] tcs(Stat ; Stat*, Tenv) =
       tcst(Stat,Tenv) & tcs(Stat*,Tenv)

[Tc5b] tcs( ,Tenv) = true
```

Declaration of a new variable: add it to TENV

Double declaration of a variable: ignore it

Again: list matching

Check the other statements, by recursion

The recursion ends with the empty list (nothing)

Pico-typecheck (6)

Check assignment statement

```
[Tc6a] tcst(Id := Exp, Tenv) = compatible(tce(Id,Tenv), tce(Exp,Tenv))
```

Type of lhs

Type of rhs

Both types must be compatible

Pico-typecheck (7)

Check if statement

Expression should have type natural

```
[Tc6b]  tce(Exp,Tenv) == natural
        ============================
        tcst(if Exp then Series1 else Series2 fi, Tenv) =
        tcs(Series1, Tenv) & tcs(Series2, Tenv)
```

Both branches should be type correct, we (re)use the Boolean & function

Pico-typecheck (8)

Check while statement

```
[Tc6c]                tce(Exp, Tenv) == natural
                   ===========================
        tcst(while Exp do Series od, Tenv) = tcs(Series, Tenv)


[default-Tc6] tcst(Stat, Tenv) = false
```

In all other cases the typecheck of a statement fails

Pico-typecheck (9)

The type of an identifier is its declared type

```
[Tc7a]  tce(Id, Tenv) = lookup(Id, Tenv)

[Tc7b]  tce(Nat-con, Tenv) = natural

[Tc7c]  tce(Str-con, Tenv) = string
```

The elementary types of constants

Pico-typecheck (10)

Both arguments should be of type natural, this equation has two conditions

```
[Tc7d]  tce(Exp1, Tenv) == natural,
        tce(Exp2, Tenv) == natural
         ================================
             tce(Exp1 + Exp2, Tenv) = natural
```

Check an addition

Result type is natural

Pico-typecheck (11)

```
[Tc7e]  tce(Exp1, Tenv) == natural, tce(Exp2, Tenv) == natural
          ===========================
                tce(Exp1 - Exp2, Tenv) = natural

[Tc7f]  tce(Exp1, Tenv) == string, tce(Exp2, Tenv) == string
         ===============================
                tce(Exp1 || Exp2, Tenv) = string

[default-Tc7]
         tce(Exp,Tenv) = nil-type
```

Check - and ||

In all other cases the expression gets type nil-type

Typechecking the factorial program

The term

```
tcp(
      begin declare input : natural,
                    output  :  natural,
                    repnr: natural,
                   rep: natural;
               input := 14;
               output := 1;
               while input - 1 do
                   rep := output;
                   repnr := input;
                   while repnr - 1 do
                      output := output + rep;
                      repnr := repnr - 1
                   od;
                   input := input - 1
                od
      end
)
```

reduces to

```
true
```

Intermezzo: equations (1)

Left-hand side may never consist of a single variable:

```
[B1] Bool = true & Bool
```

Right-hand side may not contain uninstantiated variables:

```
[B1] true & Bool1 = Bool2
```

Intermezzo: equations (2)

Rules are not ordered, so this program either executes B1, or B2, but you don't know which!

```
[B1] true & Bool = Bool
[B2] true & false = false
```

Solution: default rule is tried when all other rules fail:

```
[B1] true & Bool = Bool
[default-B1] Bool1 & Bool2 = Bool1
```

Or.. add conditions to make them mutually exclusive

Intermezzo: equations (3)

- A conditional equation succeeds when left-hand side matches and all conditions are successfully evaluated
- An equation may have zero or more conditions:
  - equality: “==”; no uninstantiated variables may be used
  - inequality: “!=”; no uninstantiated variables
  - match: “:=”; rhs may not contain uninstantiated variables, lhs may contain new variables,
  - and not-match: “!:=”; guess what it does...

Typechecking Pico: summary

- The modules `Pico-typecheck`, `Check-types` and `Type-environments` define (together with the modules they import) the typechecking rules for the Pico language
- They can be used to
  - Generate a stand-alone Pico typechecker
  - Add a typecheck button to a syntax-directed editor for Pico programs

Typechecking Pico: summary (2)

- ASF+SDF: provides syntax and data-structures for analyzing and manipulating programs
- Does not *assume anything* about the language you manipulate (no heuristics)
- You can, *and have to*, "define" the static semantics of Pico
- An implementation is generated from the definition

Plan

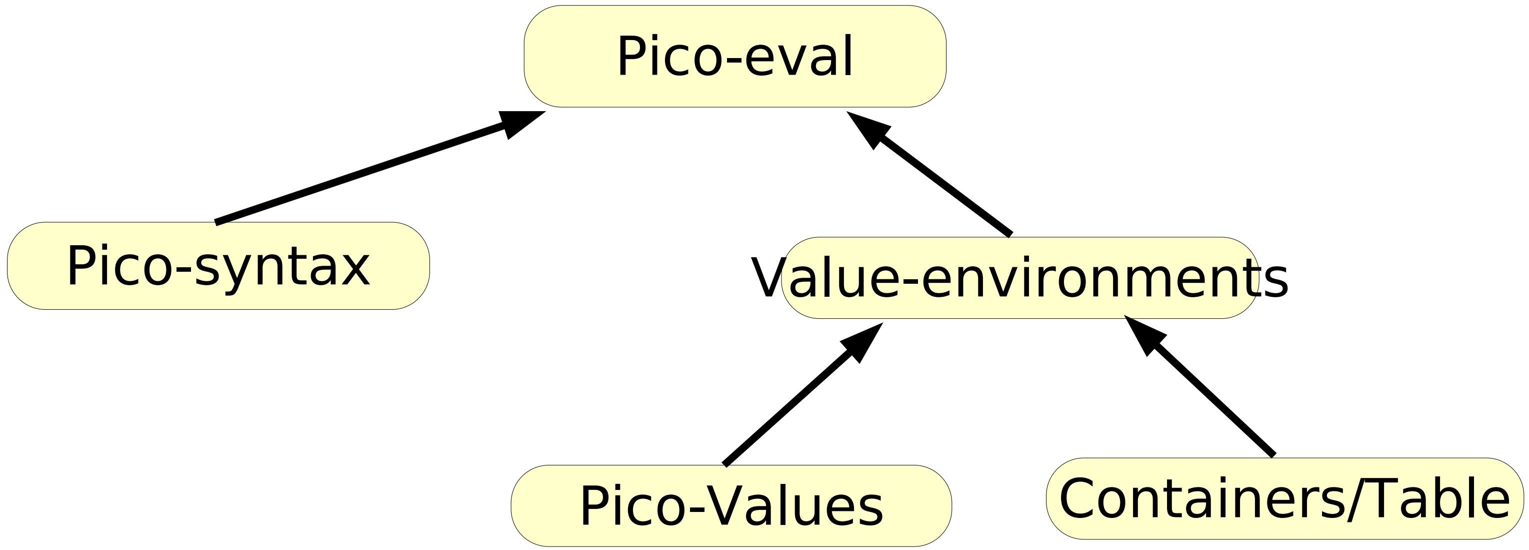
- Booleans
- Steps towards a Pico environment
  - Step 1: define syntax
  - Step 2: define a typechecker
  - *Step 3: define an evaluator*
  - Step 4: define a compiler
- Traversal functions

Step 3: Define evaluator for PICO

- Natural variables are initialized to 0
- String variables are initialized to “”
- Variable on lhs of assignment gets value of Rhs
- Variable evaluates to its current value
- Test in while and if-then equal to 0 => false
- Test in while and if-then not equal to 0 => true

Pico evaluator

The Pico evaluator/runner/interpreter simply "transforms" a Pico program to the output it generates, by stepwise reduction. This is called an "operational" semantics.



A transformation like this is similar to any other transformation, like for example a transformation from a Java class to a report of identified "code smells".

Pico-Values

```
module Pico-Values

imports basic/Integers basic/Strings

exports
  sorts VALUE
  context-free syntax
    Integer     -> VALUE
    String      -> VALUE
    "nil-value" -> VALUE
```

Integers and Strings can occur as values during execution

nil-value denotes error values

Value-environments (1)

```
module Value-environments

imports Pico-Identifiers Pico-Values
     containers/Table[PICO-ID VALUE]

exports
  sorts VENV
  aliases
    Table[[PICO-ID, VALUE]] -> VENV
```

Use Table again, to get a mapping from PICO-ID to VALUE

Call it VENV: this will represent the run-time values of variables, (the Pico "heap"!)

Pico-eval (1)

```
module Pico-eval

imports Pico-syntax Value-environments

exports
  context-free syntax
    evp(PROGRAM)                          -> VENV
    evd(DECLS)                            -> VENV
    evits({ID-TYPE  ","}*)                -> VENV
    evs({STATEMENT ";"}*, VENV)  -> VENV
    evst(STATEMENT, VENV)        -> VENV
    eve(EXP, VENV)                        -> VALUE
```

Evaluate a program

Evaluate declarations

Evaluate statements

Evaluate an expression

Pico-eval (2)

```
hiddens
  variables
    "Decls"[0-9\']*       -> DECLS
    "Exp"[0-9\']*         -> EXP
    "Id"[0-9]*            -> PICO-ID
    "Id-type*"[0-9\']*  -> {ID-TYPE ","}*
    "Nat"[0-9\']*         -> Natural
    "Nat-con"[0-9\']*   -> NatCon
    "Series"[0-9\']*     -> {STATEMENT ";"}+
    "Stat"[0-9\']*        -> STATEMENT
    "Stat*"[0-9\']*      -> {STATEMENT ";"}*
    "Str-con"[0-9\']*   -> StrCon
    "Str"[0-9\']*         -> String
    "Value"[0-9\']*      -> VALUE
    "Venv"[0-9\']*       -> VENV
```

Pico-eval (3)

Evaluate a program

```
equations
[Ev1]  evp(begin Decls Series end) = evs(Series,
                                         evd(Decls))
```

Evaluate the statements

Evaluate the declarations; Result a VENV with all variables set to default values

Pico-eval (4)

```
[Ev2]  evd(declare Id-type*;) = evits(Id-type*)

[Ev3a] evits(Id:natural, Id-type*) = store(evits(Id-type*),Id,0)

[Ev3b]  evits(Id:string, Id-type*) = store(evits(Id-type*), Id,””)

[Ev3c]  evits() = new-table
```

Initialize a natural variable

Initialize a string variable

Create a new table for the empty list of declarations

Pico-eval (5)

Evaluate first statement

Evaluate following statements in updated environment

```
[Ev4a]   Venv'  := evst(Stat, Venv),
         Venv'' := evs(Stat*, Venv')
         ======================
         evs(Stat ; Stat*, Venv) =  Venv''

[Ev4b]  evs( , Venv) = Venv
```

Evaluate a sequence of statements, the essence of an imperative programming language

Evaluate an empty sequence of statements

Pico-eval (6)

Evaluate assignment statement

```
[Ev5a]  evst(Id := Exp, Venv) = update(Id, eve(Exp, Venv),
Venv)
```

Evaluate Rhs

Update variable with value of Rhs

Pico-eval (7)

Evaluate
if statement

"true" case

evaluate
then branch

```
[Ev5b]                      eve(Exp, Venv) != 0
        ========================================
        evst(if Exp then Series1 else Series2 fi, Venv) =
        evs(Series1, Venv)

[Ev5c]                      eve(Exp, Venv) == 0
        ====================================
        evst(if Exp then Series1 else Series2 fi, Venv) =
        evs(Series2, Venv)
```

"false" case

evaluate
else branch

The ASF compiler makes sure that
"eve(Exp, Venv) is only executed once...

Pico-eval (8)

Evaluate while statement

"false" case: while ends

```
[Ev5d]           eve(Exp, Venv) == 0
        ========================
        evst(while Exp do Series od, Venv) =  Venv

[Ev5e]           eve(Exp, Venv) != 0,
                Venv' := evs(Series, Venv)
        ========================
       evst(while Exp do Series od, Venv) =
       evst(while Exp do Series od, Venv')
```

"true" case: while continues

Evaluate body once

Evaluate while statement in updated environment

Pico-eval (9)

A variable evaluates to its current value in the environment

```
[Ev6a]  eve(Id, Venv) = lookup(Venv, Id)
[Ev6b]  eve(Nat-con, Venv) = Nat-con
[Ev6c]  eve(Str-con, Venv) = Str-con
```

Constants evaluate to themselves

MetaEnv

Pico-eval (10)

Evaluate left operand

Evaluate right operand

```
[Ev6d]  Nat1 := eve(Exp1, Venv),
        Nat2 := eve(Exp2, Venv)
       =================
        eve(Exp1 + Exp2, Venv) = Nat1 + Nat2
```

Evaluate addition

Funny: two different "+" signs, that look the same! One is "Integer", one is "EXP"

Add the resulting values, reuses the definition of Integer arithmetic from the library module basic/Integers

Pico-eval (11)

```
[Ev6e]  Nat1 := eve(Exp1, Venv),
         Nat2 := eve(Exp2, Venv)
         ============================
        eve(Exp1 - Exp2, Venv) =  Nat1 -/ Nat2

[Ev6f]  Str1 := eve(Exp1, Venv),
         Str2 := eve(Exp2, Venv)
        ===========================
        eve(Exp1 || Exp2, Venv) = Str1 || Str2

[default-Ev6]  eve(Exp,Venv) = nil-value
```

Evaluate - and ||

Cutoff subtraction for naturals, e.g. 3 -/ 4 = 0
We stay inside naturals

All other cases evaluate to nil-value

Evaluating the factorial program

The term

```
evp(
      begin declare input : natural,
                    output  :  natural,
                    repnr: natural,
                   rep: natural;
               input := 14;
               output := 1;
               while input - 1 do
                  rep := output;
                  repnr := input;
                  while repnr - 1 do
                     output := output + rep;
                     repnr := repnr - 1
                  od;
                  input := input - 1
               od
      end
)
```

reduces to

```
[<input,1>,
<repnr,1>,
<output,87178291200
>,
<rep,43589145600>]
```

Evaluating Pico: summary

- The modules Pico-eval, Pico-values, and Value-environments define (together with the modules they import) the evaluation rules for the Pico language
- They can be used to
  - Generate a stand-alone Pico evaluator
  - Add an evaluation button to a syntax-directed editor for Pico programs

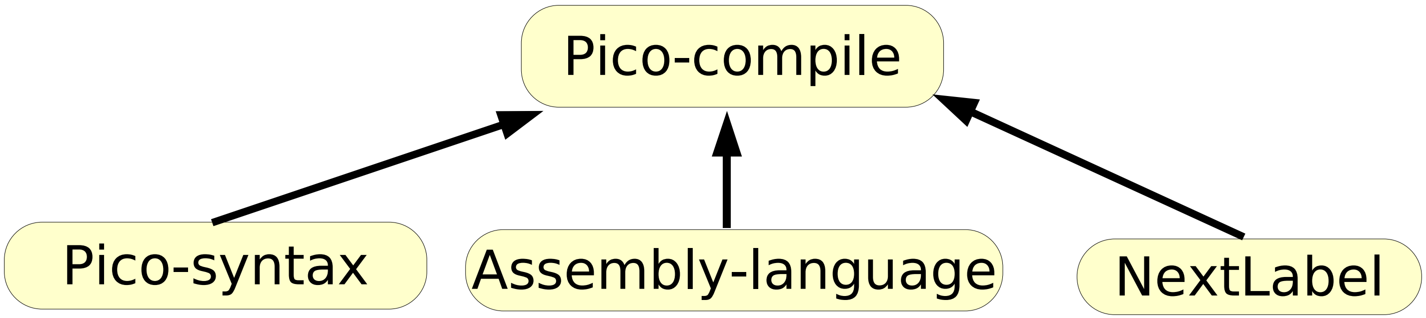
Evaluating Pico: summary (2)

- ASF+SDF is used to define a rather complex transformation
- No assumptions about the transformation, it is just a convenient language for *manipulating trees*
- But.. there is more!

Plan

- Booleans
- Steps towards a Pico environment
  - Step 1: define syntax
  - Step 2: define a typechecker
  - Step 3: define an evaluator
  - *<u>Step 4: define a compiler</u>*
- Traversal functions

Pico compiler



A more standard example of a transformation:
input Pico; output Assembly for a stack based instruction set
(similar to Java bytecode)

AssemblyLanguage (1)

```
module AssemblyLanguage

imports basic/Integers basic/Strings Pico-Identifiers
exports
  sorts Label Instr
  lexical syntax
    [a-z0-9]+                    -> Label

  context-free syntax
    "dclnat" PICO-ID    -> Instr
    "dclstr" PICO-ID      -> Instr
```

Instruction labels

Directives to allocate a variable

AssemblyLanguage (2)

```
"push" NatCon      -> Instr
"push" StrCon      -> Instr
"rvalue" PICO-ID   -> Instr
"lvalue" PICO-ID   -> Instr
"assign"           -> Instr
"add"              -> Instr
"sub"              -> Instr
"conc"             -> Instr
"label" Label      -> Instr
"goto" Label       -> Instr
"gotrue" Label     -> Instr
"gofalse" Label    -> Instr
"noop"             -> Instr
aliases
  {Instr ";"}+     -> Instrs
```

- Push a constant on the stack
- Push a variable's value on the stack
- Push a variable's name on the stack
- Assign to variable
- Operators
- Declare a label
- (Conditional) jump instructions
- Dummy instruction
- Convenient shorthand

NextLabel

```
module NextLabel
imports AssemblyLanguage
exports
  context-free syntax
    "nextlabel" "(" Label ")" -> Label

hiddens
  variables
    "Char*"[0-9]* -> CHAR*
equations

 [1] nextlabel(label(Char*)) = label(Char* "
```

For every lexical definition with result sort L, the lexical constructor function

`l "(" CHAR+ ")" -> L`

is generated:

- l is the sort name in lower case letters (here: label)
- This gives access to the text of lexical tokens

Pico-compile (1)

```
module Pico-compile
imports Pico-syntax AssemblyLanguage NextLabel

exports
  context-free syntax
    trp( PROGRAM )                        -> Instrs

hiddens
  context-free syntax
    trd(DECLS)                            -> Instrs
    trits({ID-TYPE  ","}*)                -> Instrs
    trs({STATEMENT ";"}*, Label)  -> <Instrs, Label>
    trst(STATEMENT, Label)        -> <Instrs, Label>
    tre(EXP)                              -> Instr
```

Translation of statements generates instructions and new labels (<Instrs, Label>)

Pico-compile (2)

```
hiddens
  variables
    "Decls"[0-9\']*      -> DECLS
    "Exp"[0-9\']*        -> EXP
    "Id"[0-9]*           -> PICO-ID
    "Id-type*"[0-9\']* -> {ID-TYPE ","}*
    "Nat"[0-9\']*        -> Natural
    "Nat-con"[0-9\']*  -> NatCon
    "Series"[0-9\']*     -> {STATEMENT ";"}+
    "Stat"[0-9\']*       -> STATEMENT
    "Stat*"[0-9\']*     -> {STATEMENT ";"}*
    "Str-con"[0-9\']*   -> StrCon
    "Str"[0-9\']*        -> String

    "Instr*"[0-9\']*    -> Instrs
    "Label" [0-9\']*     -> Label
```

Pico-compile (3)

```
equations

[Tr1]   Instr*1 := trd(Decls),
         <Instr*2, Label> := trs(Series, x)
         =====================
        trp(begin Decls Series end) = Instr*1; Instr*2

[Tr2]   trd(declare Id-type*;) = trits(Id-type*)

[Tr3a] trits(Id:natural, Id-type*) = dclnat Id;
                                      trits(Id-type*)

[Tr3b] trits(Id:string, Id-type*) = dclstr Id;
                                     trits(Id-type*)
[Tr3c] trits() = noop
```

Translate a program

Translate a declaration section

Translate a variable declaration

Translate an empty list

Pico-compile (4)

```
[Tr4a] <Instr*1, Label'>  := trst(Stat, Label),
         <Instr*2, Label''> := trs(Stat*, Label')
         =====================
       trs(Stat ; Stat*, Label) =
       < Instr*1 ;  Instr*2, Label'' >

[Tr4b] trs( , Label) = <noop, Label>
```

Translate Stat ; Stat*

Translation of Stat

Translation of Stat*

Last label used during translation

Pico-compile (5)

```
[Tr5a] Instr*:= tre(Exp)

=================
        trst(Id := Exp, Label) =
        < lvalue Id,
          Instr*;
          assign
        ,
          Label >
```

Translate Id := Exp

Push the name of the Lhs Id

Translated Rhs Exp

Assign the value of the expression to the variable

Pico-compile (6)

```
[Tr5b] Instr*:= tre(Exp),
         <Instr*1, Label'>:= trs(Series1, Label),
         <Instr*2, Label''>:= trs(Series2, Label'),
         Label1 := nextlabel(Label''),
         Label2 := nextlabel(Label1)
         =====================
        trst(if Exp then Series1  else Series2 fi, Label) =
        < Instr*;
          gofalse Label1;
          Instr*1;
          goto Label2;
          label Label1;
          Instr*2;
         label Label2
        ,
          Label2 >
```

Translate if statement

Pico-compile (7)

```
[Tr5c]  Instr*1:= tre(Exp),
          <Instr*2, Label'>, := trs(Series, Label),
         Label1 := nextlabel(Label'),
         Label2 := nextlabel(Label1)
         ===============
         trst(while Exp do Series od, Label) =
         < label Label1;
           Instr*1;
           gofalse Label2;
           Instr*2;
           goto Label1;
           label Label2
        ,
           Label2 >
```

Translate while statement

Pico-compile (8)

```
[Tr6a] tre(Nat-con) = push Nat-con
[Tr6b] tre(Str-con) = push Str-con
[Tr6c] tre(Id) = rvalue Id

[Trcd] Instr*1:= tre(Exp1),  Instr*2:= tre(Exp2)
       ===================
       tre(Exp1 + Exp2) = Instr*1; Instr*2; add

[Tr6e] Instr*1:= tre(Exp1),   Instr*2:= tre(Exp2)
       ========================
       tre(Exp1 - Exp2) = Instr*1; Instr*2; sub

[Tr6f] Instr*1:= tre(Exp1),   Instr*2 := tre(Exp2)
       =====================
       tre(Exp1 || Exp2) = Instr*1; Instr*2; conc
```

Translate constants

Translate variable

Translate +, - and ||

Compiling the factorial program

The term

```
trp( begin declare input : natural,
                   output  :  natural,
                   repnr: natural,
                   rep: natural;
         input := 14;
         output := 1;
         while input - 1 do
            rep := output;
            repnr := input;
            while repnr - 1 do
               output := output + rep;
               repnr := repnr - 1
            od;
            input := input - 1
         od
     end
)
```

reduces to

```
dclnat input;
dclnat output;
dclnat repnr;
dclnat rep;
noop; lvalue input;
 push 14;
 assign ;
 lvalue output;
 push 1;
 assign ;
 label xxxx;
 rvalue input; push 1;
sub;
 gofalse xxxxx;
 lvalue rep;
 rvalue output;
 assign ;
 lvalue repnr;
 rvalue input;
 assign ;
label xx;
 rvalue repnr; push 1; sub;
 gofalse xxx;
 lvalue output;
 rvalue output; rvalue rep;
add;
 assign ;
 lvalue repnr;
 rvalue repnr; push 1; sub;
 assign ;
 noop;
 goto xx;
 label xxx ;
 lvalue input;
 rvalue input; push 1; sub;
 assign ;
 noop;
 goto xxxx;
 label xxxxx ;
 noop
```

Compiling Pico: summary

- The modules `Pico-compile`, `AssemblyLanguage`, and `NextLabel` define (together with the modules they import) the compilation rules for the Pico language
- They can be used to
  - Generate a stand-alone Pico compiler
  - Add an compilation button to a syntax-directed editor for Pico programs

Compiling Pico: summary

- Just another transformation by ASF+SDF

Plan

- Booleans
- Steps towards a Pico environment
  - Step 1: define syntax
  - Step 2: define a typechecker
  - Step 3: define an evaluator
  - Step 4: define a compiler
- *Traversal functions*

Traversal Functions (1)

- Many functions have the characteristic that they traverse the tree *recursively* and only do something interesting at a few nodes
- Example: count the identifiers in a program
- Using a recursive (inductive) definition:
  - # of equations is equal to number of syntax rules
  - think about Cobol or Java with hundreds of rules
- Traversal functions automate *recursion*

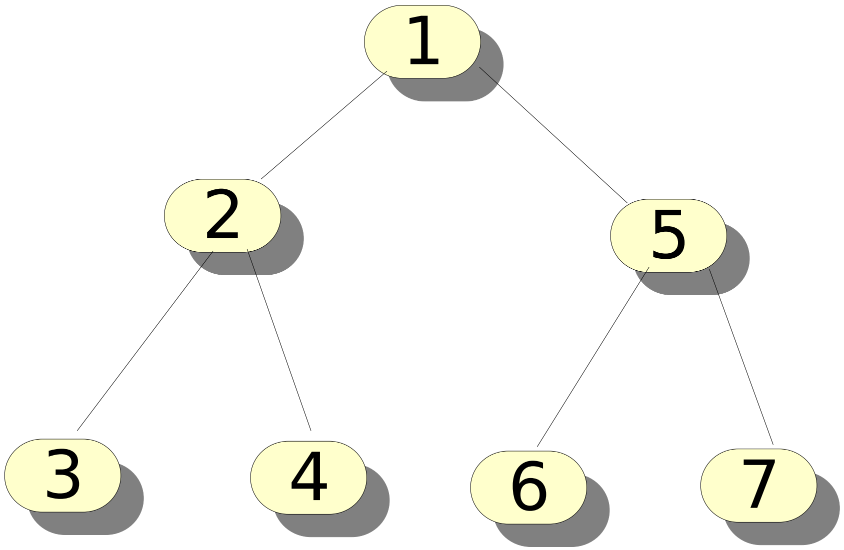
Traversal Functions (2)

There are two important aspects of traversal functions:

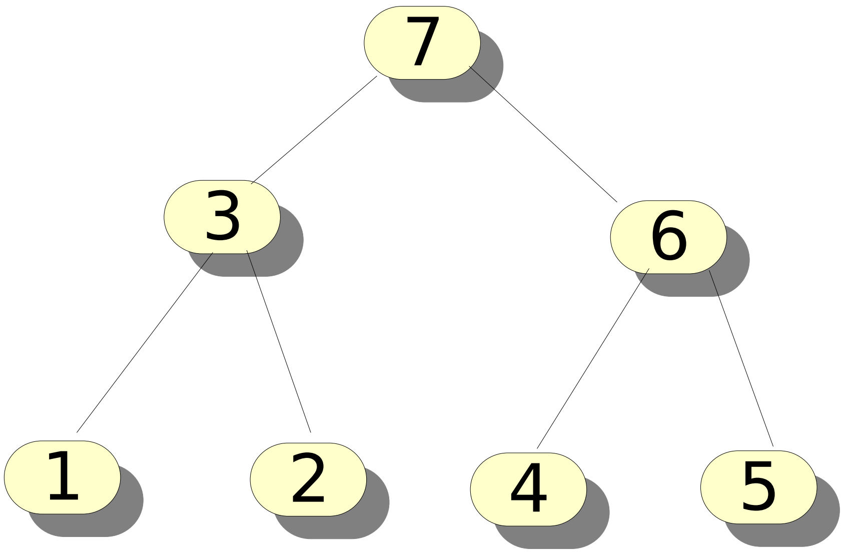
- the kind of traversal
  - accumulate a value during traversal
  - transform the tree during traversal
- the order of traversal
  - top-down versus bottom-up
  - left-to-right versus right-to-left (we only have the first)
  - break or continue after a visit

Top-down versus Bottom-up

Top-down



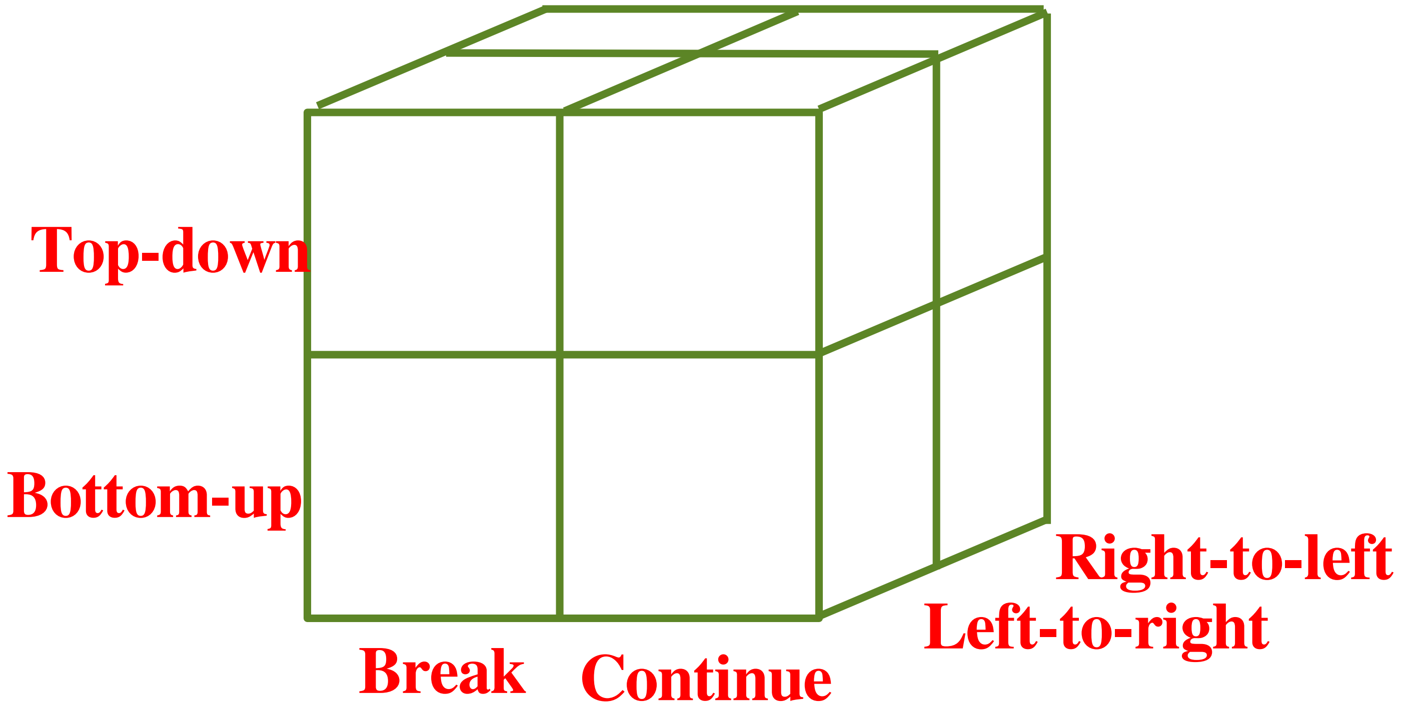
Bottom-up



Three kinds of traversals

- Accumulator: `traversal(accu)`
  - accumulate a value during traversal
- Transformer: `traversal(trafo)`
  - perform local transformations
- Accumulating transformer: `traversal(accu, trafo)`
  - accumulate *and* transform

Traversal Cube: visiting behaviour



Simple Trees

```
module Tree-syntax
imports Naturals
exports
  sorts TREE
  context-free syntax
    NAT                -> TREE
    f(TREE, TREE) -> TREE
    g(TREE, TREE) -> TREE
    h(TREE, TREE) -> TREE
  variables
    "N"[0-9]*         -> NAT
    "T"[0-9]*         -> TREE
```

Simple trees containing numbers as leaves and constructors f, g, or h

Count nodes (classical)

```
module Tree-cnt
imports Tree-syntax
exports
context-free syntax
  cnt(TREE)       -> NAT
equations
[1] cnt(N)            = 1
[2] cnt(f(T1,T2)) = 1+cnt(T1)+cnt(T2)
[3] cnt(g(T1,T2)) =
1+cnt(T1)+cnt(T2)
[4] cnt(h(T1,T2)) =
1+cnt(T1)+cnt(T2)
```

Count the nodes in a tree

These equations are needed to visit all nodes in the tree

A new equation has to be added for each new constructor

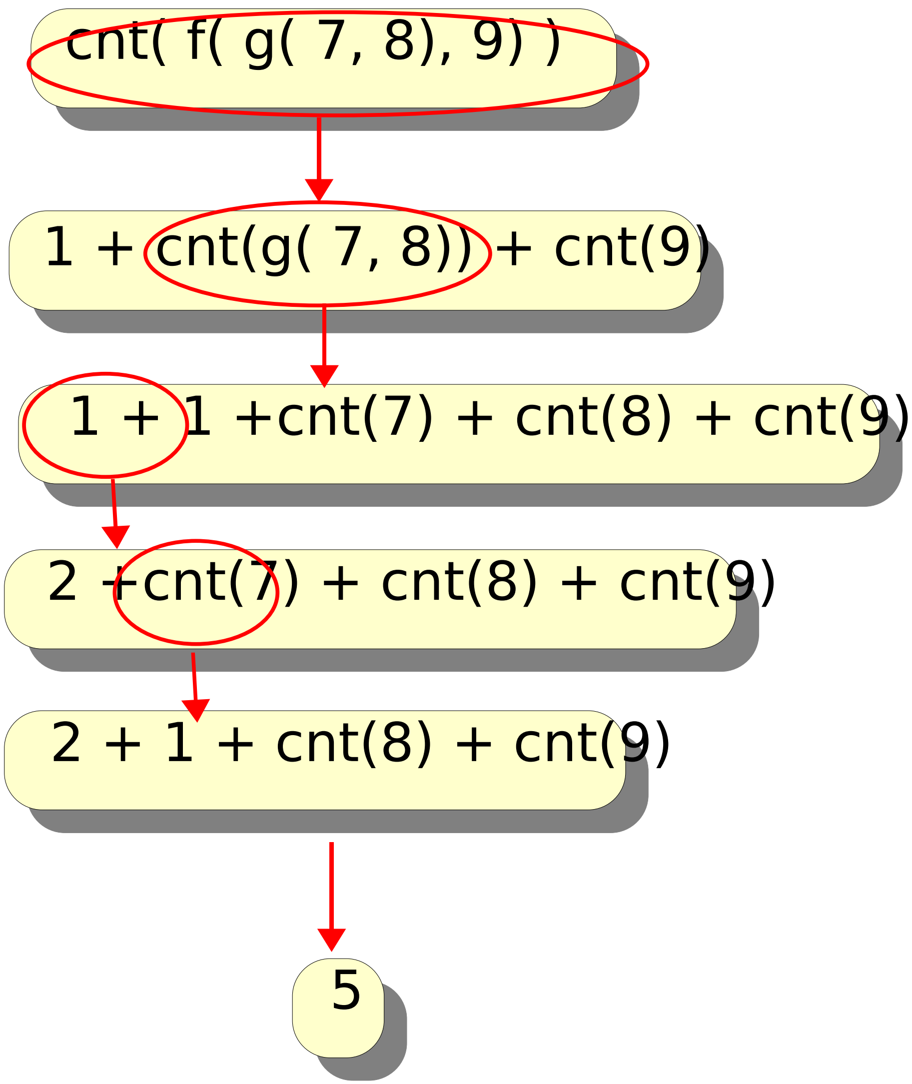
Count this node

Count nodes in both subtrees

```
cnt( f( g( f(1,2), 3 ),
          g( g(4,5), 6 )),
     )
```

→ 11

Example



Left-most innermost reduction:

[2] cnt(f(T1,T2)) =
1+cnt(T1)+cnt(T2)

[3] cnt(g(T1,T2)) =
1+cnt(T1)+cnt(T2)

Addition of integers

[1] cnt(N) = 1

... Similar reductions

Using Accumulators

- Goal: traverse term and accumulate a value
- `fun(Tree, Accu) -> Accu {traversal(accu, ...)}`
- `Tree`: term to be traversed (always the first argument)
- `Accu`: value to be accumulated (always second argument)
- Important: the sorts of second argument and result are always equal.
- Optional: extra arguments
- `fun(Tree, Accu, A1, ...) -> Accu {traversal(...)}`

Count nodes (traversals)

```
module Tree-cnt
imports Tree-syntax
exports
 context-free syntax
   cnt(TREE, NAT) -> NAT {traversal(accu,bottom-up,continue)}
equations
[1] cnt(T, N) = N + 1
```

A bottom-up accumulator that continues after each matching node

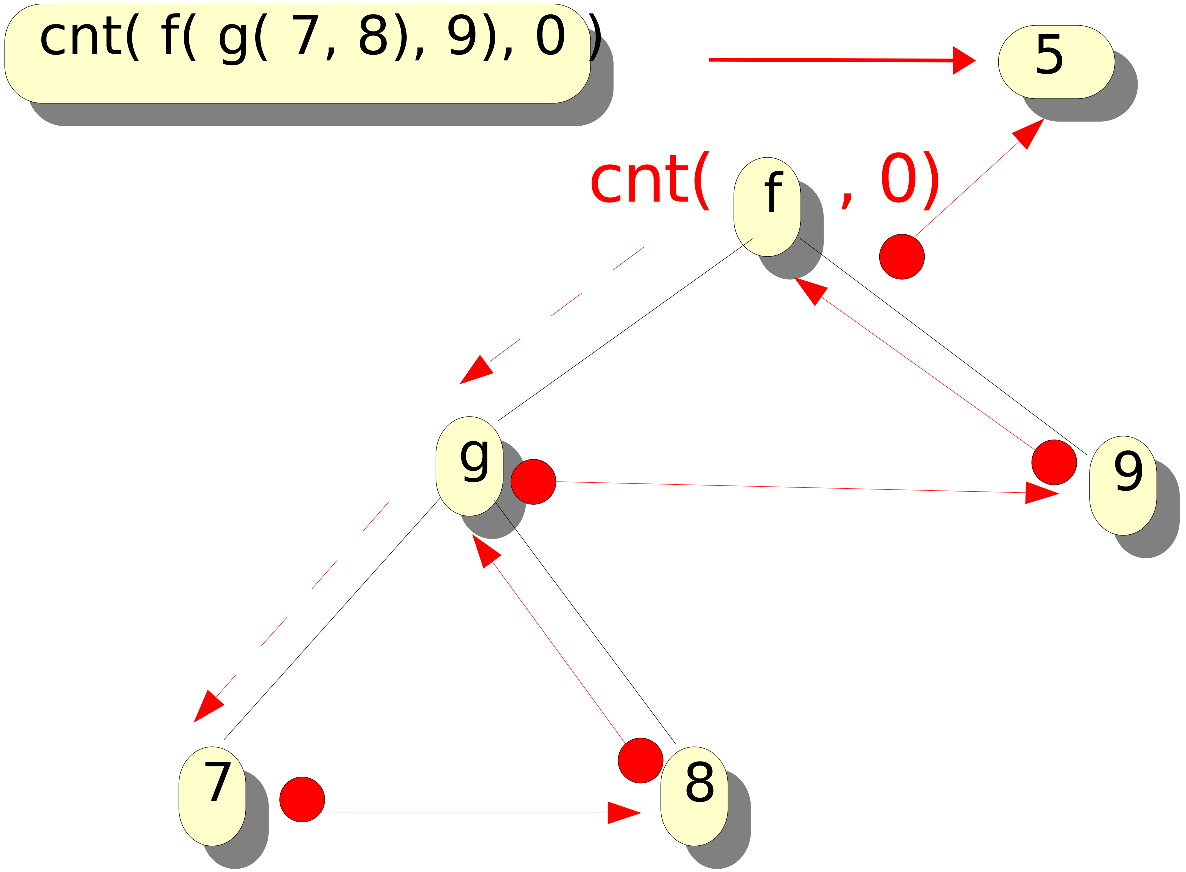
Accumulated value

Traversed tree (matches every node)

```
cnt( f( g( f(1,2), 3 ),
          g( g(4,5), 6 )),
      0)
```

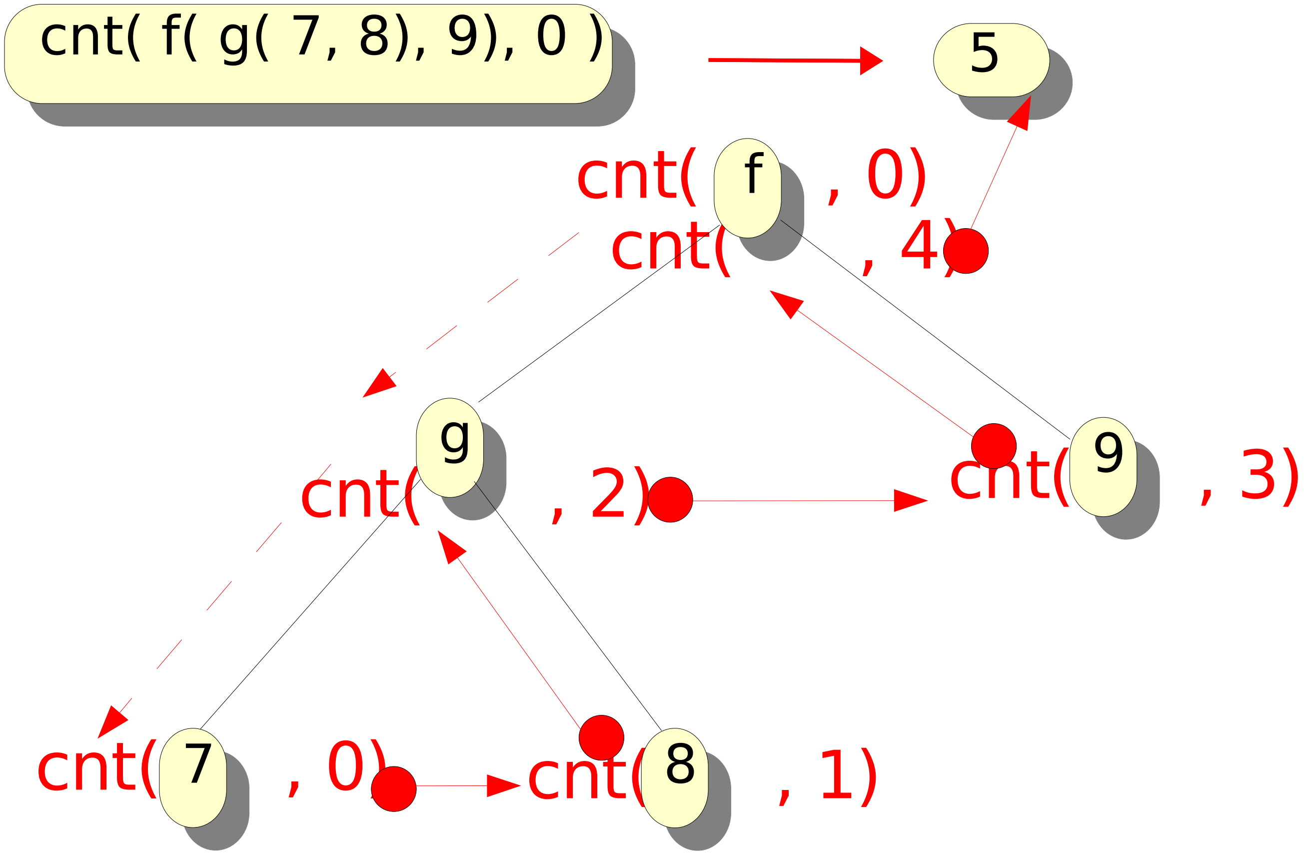
→ 11

Example: accu,bottom-up,continue



[1] cnt(T,N) = N +
1

Example: accu,bottom-up,continue



[1] cnt(T, N) = N + 1

Using Transformers

- fun(Tree) -> Tree {traversal(trafo, ...)}
- Tree: term to be traversed (always the first argument)
- Important: the sorts of the first argument and result are always equal.
- Optional: extra arguments
- fun(Tree, A1, A2, ...) -> Tree {traversal(...)}

Increment leaves

```
module Tree-inc
imports Tree-syntax
exports
context-free syntax
  inc(TREE) -> TREE {traversal(trafo,bottom-up,continue)}
equations
[1] inc(N) = N + 1
```

A bottom-up transformer that continues after each matching node

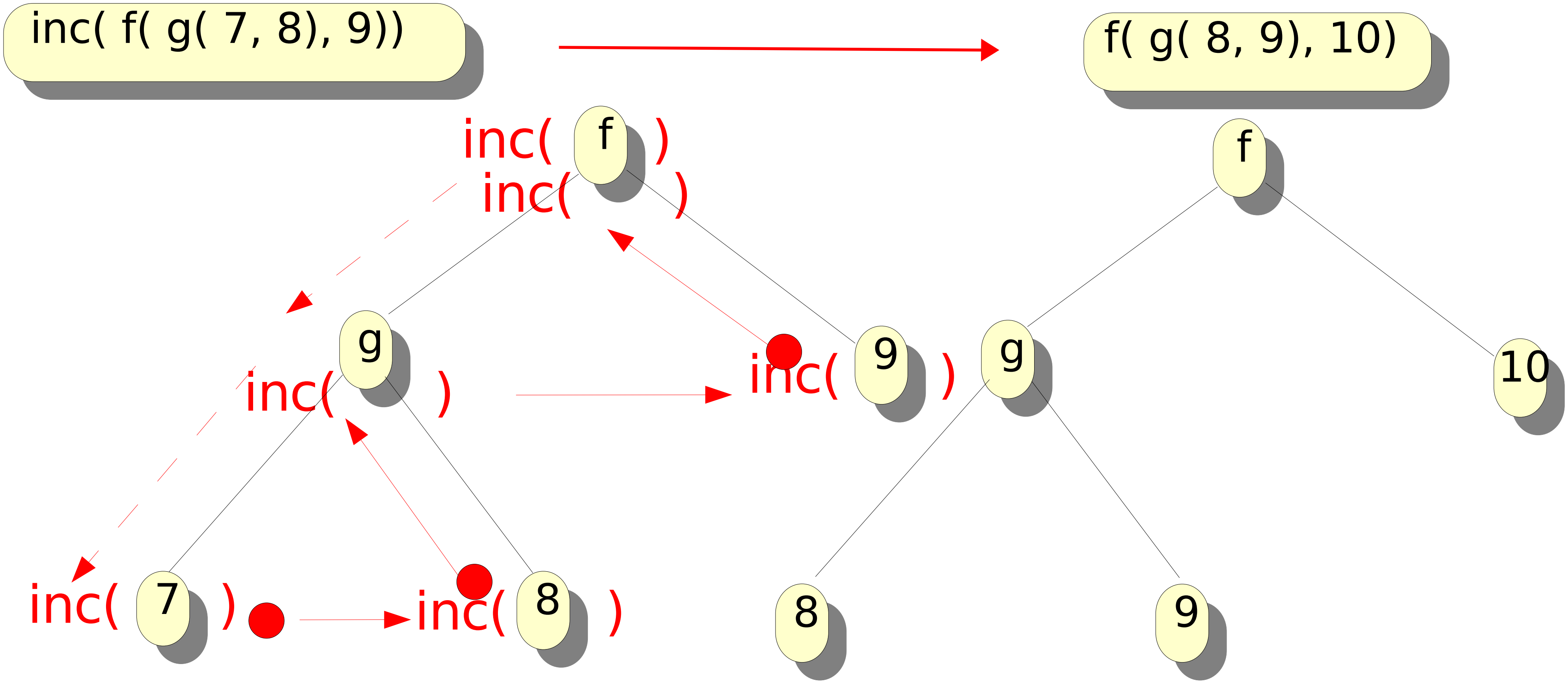
Leaf N

is replaced by N+1

```
inc( f( g( f(1,2),  3 ),
         g( g(4,5), 6 )) )
```

```
f( g( f(2,3), 4 ),
   g( g(5,6), 7 ))
```

Example trafo,bottom-up,continue



[1] inc(T, N) = N + 1

Increment leaves with explicit amount

```
module Tree-incp
imports Tree-syntax
exports
context-free syntax
  inc(TREE, NAT) -> TREE {traversal(trafo,bottom-up,continue)}
equations
[1] inc(N1, N2) = N1 + N2
```

A bottom-up transformer that continues after each matching node

Amount

Replace N1 by N1 +N2

Leaf N1

Amount N2

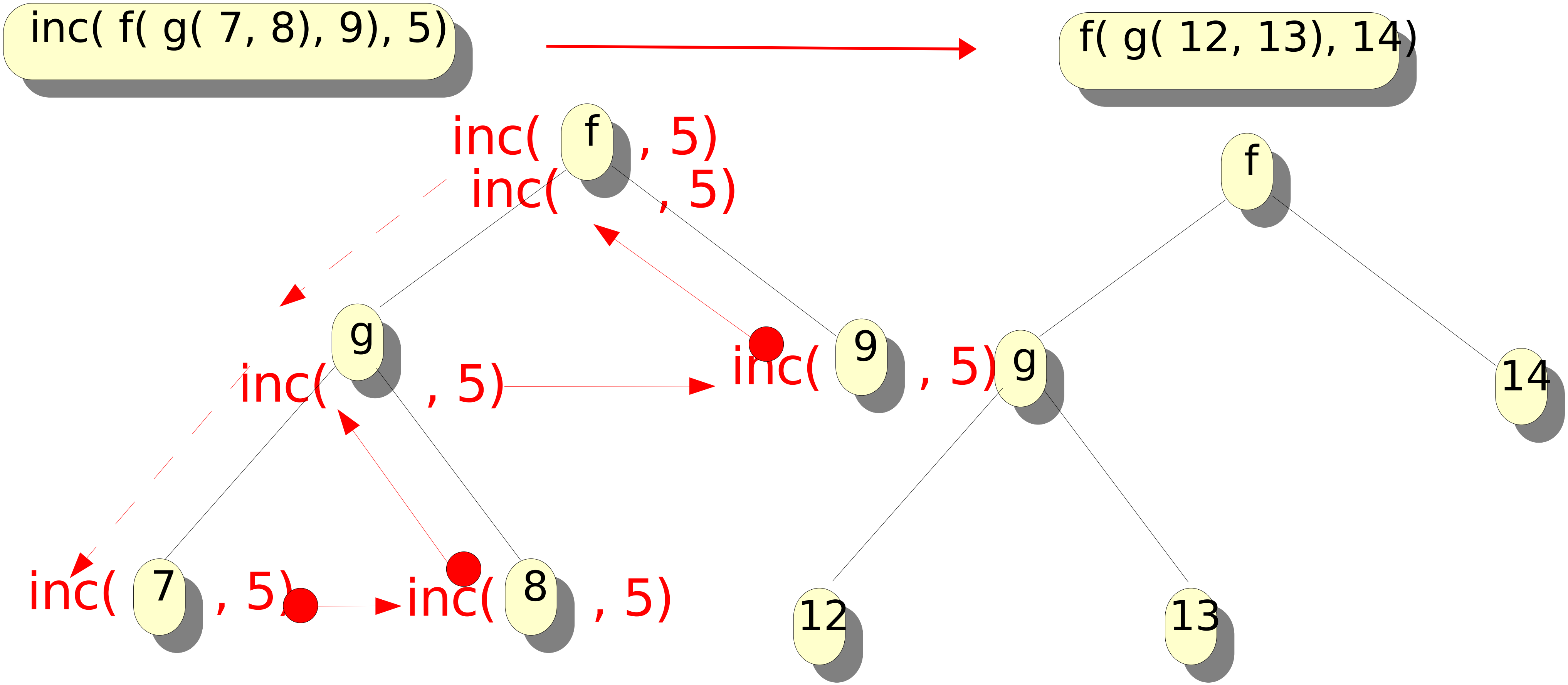
```
inc( f( g( f(1,2),  3 ),
        g( g(4,5), 6 )),
     7 )
```

→

```
f( g( f( 8,  9), 10),
   g( g(11,12),  13))
```

Example

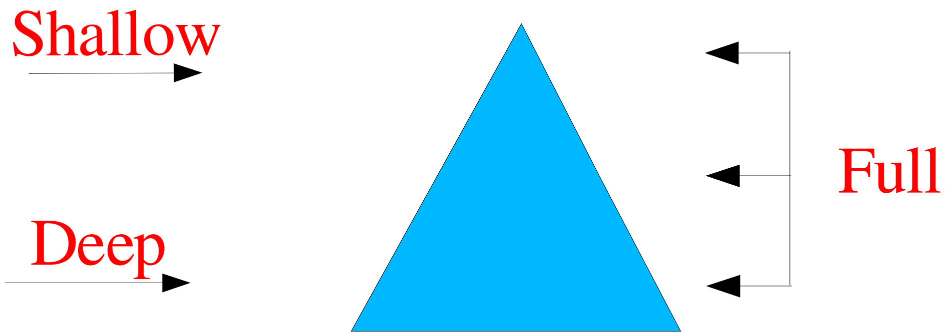
trafo,bottom-up,continue



[1] inc(N1, N2) = N1 + N2

Term Replacement

- Deep replacement: replace only occurrences close to the leaves
- Shallow replacement: replace only occurrences close to the root
- Full replacement: replace all occurrences



Deep replacement

```
module Tree-drepl
imports Tree-syntax
exports
context-free syntax
   i(TREE, TREE)   -> TREE
   drepl(TREE)     -> TREE {traversal(trafo,bottom-up,break)}
equations
[1] drepl(g(T1, T2)) = i(T1, T2)
```

Auxiliary constructor i

A bottom-up transformer that stops after first matching node

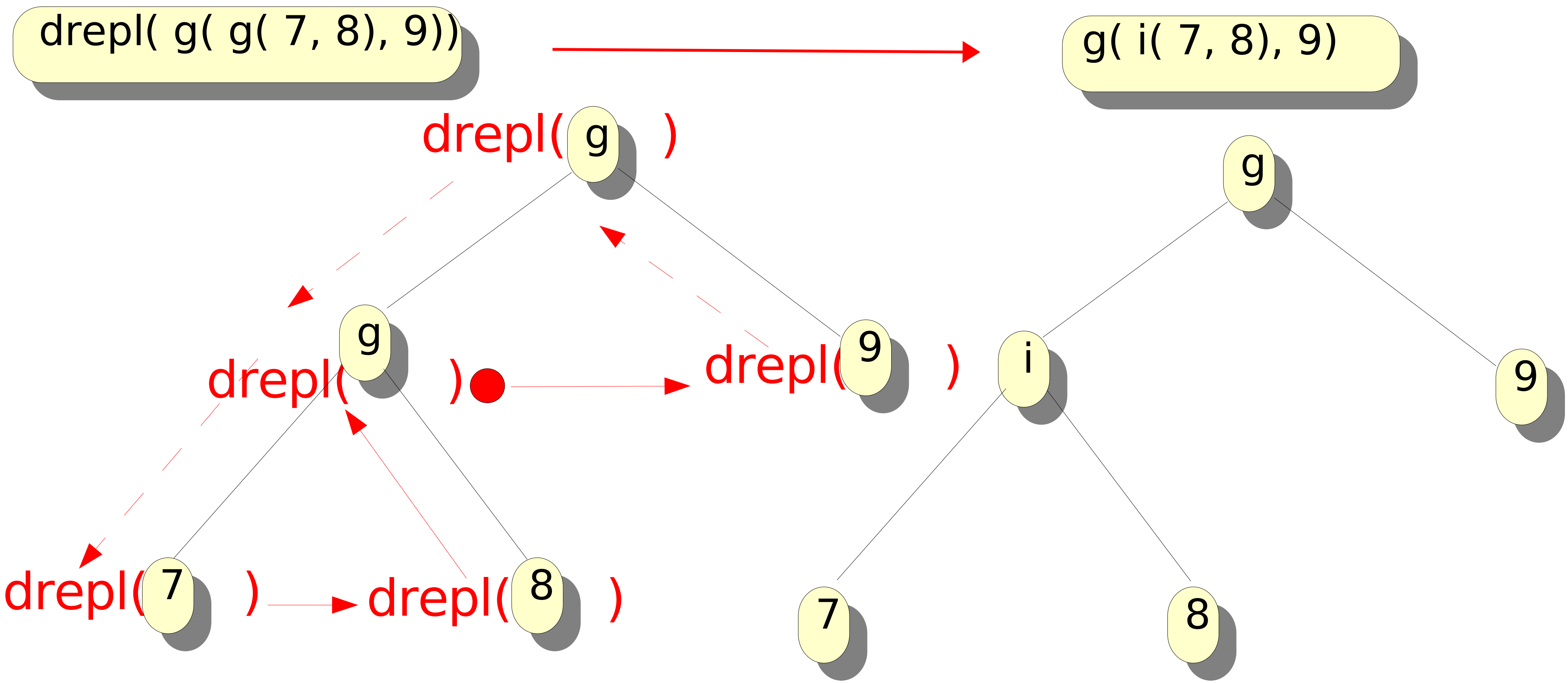
Only the deepest occurrences of g are replaced

```
drepl( f( g( f(1,2), 3 ),
          g( g(4,5), 6 )) )
```

→

```
f( i( f(1,2), 3 ),
   g( i(4,5), 6 ))
```

Example trafo,bottom-up,break



[1] drepl(g(T1, T2)) = i(T1, T2)

Shallow replacement

```
module Tree-srepl
imports Tree-syntax
exports
context-free syntax
   i(TREE, TREE)   -> TREE
   srepl(TREE)      -> TREE {traversal(trafo, top-down, break)}
equations
[1] srepl(g(T1, T2)) = i(T1, T2)
```

A top-down transformer that stops after first matching node

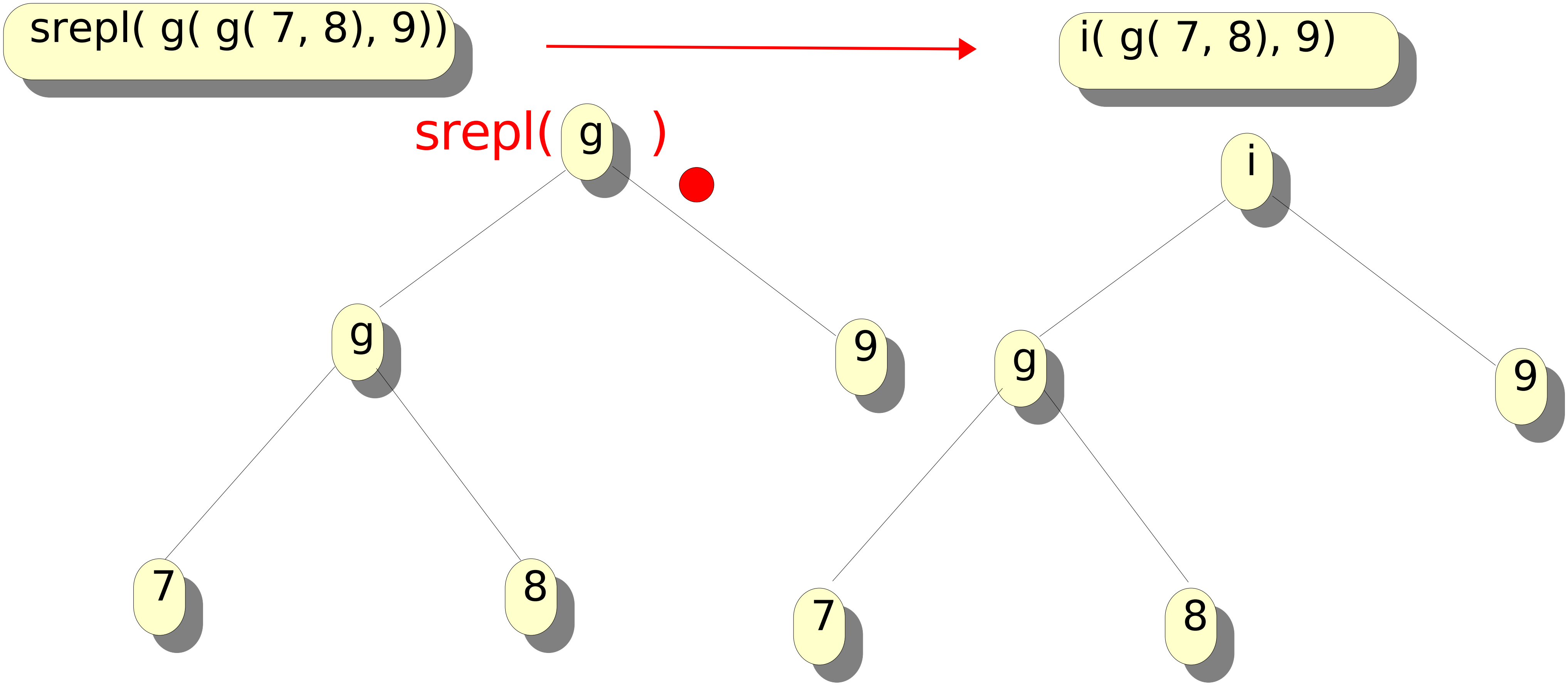
Only the outermost occurrences of g are replaced

```
srepl( f( g( f(1,2), 3 ),
          g( g(4,5), 6 )) )
```

→

```
f( i( f(1,2), 3 ),
   i( g(4,5), 6 ))
```

Example trafo, top-down, break



[1] srepl(g(T1, T2)) = i(T1, T2)

Full replacement

```
module Tree-frepl
imports Tree-syntax
exports
context-free syntax
   i(TREE, TREE)   -> TREE
   frepl(TREE)     -> TREE {traversal(trafo,top-down,continue)}
equations
[1] frepl(g(T1, T2)) = i(T1, T2)
```

A top-down transformer that continues after each matching node

top-down and bottom-up have here the same effect

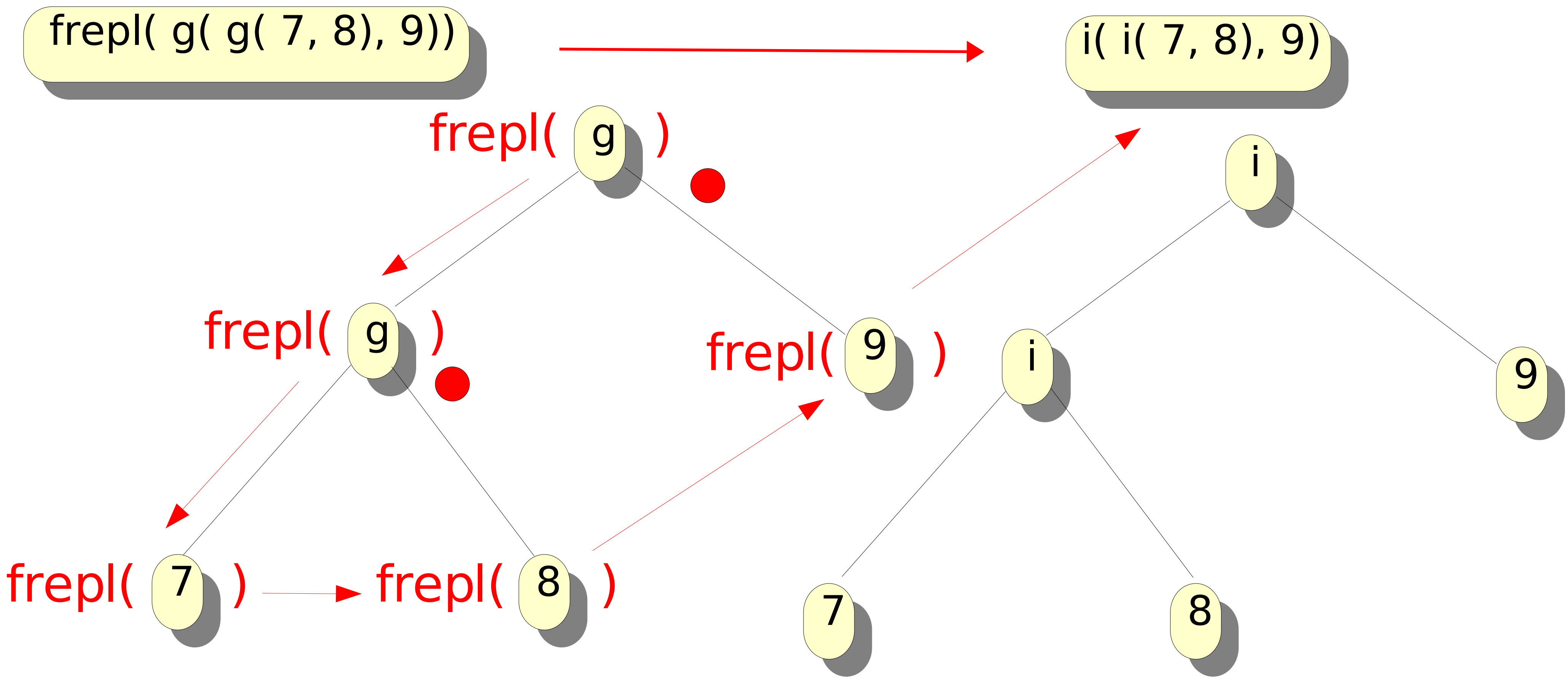
All occurrences of g are replaced

```
frepl( f( g( f(1,2), 3 ),
          g( g(4,5), 6 )) )
```

→

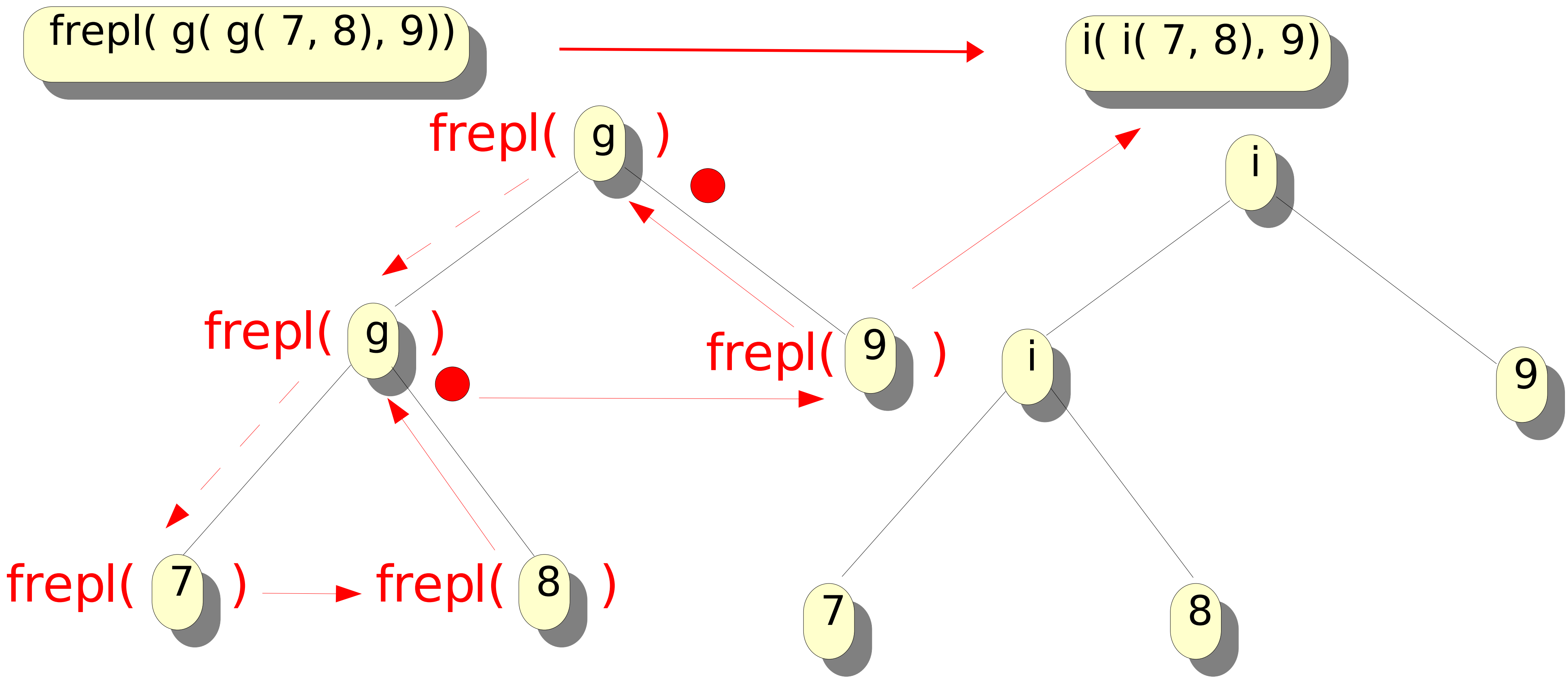
```
f( i( f(1,2), 3 ),
   i( i(4,5), 6 ))
```

Example trafo, top-down, continue



[1] frepl(g(T1, T2)) = i(T1, T2)

Example trafo, bottom-up, continue



[1] frepl(g(T1, T2)) = i(T1, T2)

A real example: Cobol transformation

- Cobol 75 has two forms of conditional:
  - “IF” Expr “THEN” Stats “END-IF”?
  - “IF” Expr “THEN” stats “ELSE” Stats “END-IF”?
- These are identical (*dangling else* problem):

```
IF expr THEN
    IF expr THEN
      stats
    ELSE
      stats
```

```
IF expr THEN
    IF expr THEN
      stats
ELSE
   stats
```

A real example: Cobol transformation

```
module End-If-Trafo
imports Cobol
exports
context-free syntax
  addEndIf(Program)-> Program {traversal(trafo,continue,top-down)}
variables
 "Stats"[0-9]*      -> StatsOptIfNotClosed
 "Expr"[0-9]*        -> L-exp
 "OptThen"[0-9]*   -> OptThen
equations
[1] addEndIf(IF Expr OptThen Stats) =
            IF Expr OptThen Stats END-IF

[2] addEndIf(IF Expr OptThen Stats1 ELSE
            IF Expr OptThen Stats1 ELSE Stats2 END-IF
```

Add missing END-IF keywords

Equations for the two cases

Impossible to do with regular expression tools like `grep` since conditionals can be nested

A funny Pico typechecker

- Replace all variables by their declared type:
  - x +3 $\Rightarrow$ type(natural) + type(natural)
- Simplify type correct expressions:
  - type(natural) + type(natural) $\Rightarrow$ type(natural)
- Remove all type correct statements:
  - type(natural) := type(natural)
- A type correct program reduces to empty
- Otherwise, only incorrect statements remain

Example

```
begin
   declare x : natural,
              y : natural,
              s : string;
     x := 10; s := "abc";
     if x then
          x := x + 1
     else
          s := x + 2
     fi;
     y := x + 2;
end
```

Yields after typechecking:

```
begin
  declare;
  type(string) := type(natural);
end
```

Erroneous statement leaves a residue

Pico-typecheck (1)

```
module Pico-typecheck
imports Pico-syntax
exports
context-free syntax
 type(TYPE)                    -> ID
 replace(STATS, ID-TYPE) -> STATS {traversal(trafo,bottom-up,break)}
 replace(EXP    , ID-TYPE)  -> EXP {traversal(trafo,bottom-up,break)}
```

Extend identifiers so that we can replace them with type information

The traversal function replace.
In the equations, the first argument may be of various sorts.
Each variant that is used in the equations has to be declared here.

Pico-typecheck (2)

```
equations
[0] begin declare Id-type, Decl*; Stat* end =
      begin declare Decl*; replace(Stat*, Id-type)
end

[1] replace(Id     , Id : Type) = type(Type)
[2] replace(Nat-con, Id : Type) = type(natural)
[3] replace(Str-con, Id : Type) = type(string)

[4] type(string) || type(string)  = type(string)
[5] type(natural) + type(natural) = type(natural)
[6] type(natural) - type(natural) = type(natural)
```

Visit each variable declaration and use replace to replace the variable by its type

Replace variables and constants by their type

Replace type-correct expressions by their type

Pico-typecheck (3)

```
[7] Stat*1; if type(natural) then Stat*2 else Stat*3 fi ; Stat*4
     = Stat*1; Stat*2; Stat*3; Stat*4

[8] Stat*1; while type(natural) do Stat*2 od; Stat*3
     = Stat*1; Stat*2; Stat*3

[9] Stat*1; type(Type) := type(Type); Stat*2
     = Stat*1; Stat*2
```

Remove type-correct expressions and statements

Traversal functions ...

- ... automate common kinds of tree traversals
- ... reduce number of required equations significantly
- ... lead to easier to understand specifications
- ... can be implemented efficiently
- ... have been applied in a lot of applications

Further reading

- M.G.J. van den Brand and P. Klint, ASF+SDF Meta-Environment User Manual
  www.cwi.nl/projects/MetaEnv/meta/doc/manual/user-manual.html
- M.G.J. van den Brand, P. Klint and J. Vinju, Term rewriting with traversal functions, ACM Transactions on Software Engineering and Methodology, **12**(2):152-190, 2003
- www.cwi.nl/projects/MetaEnv