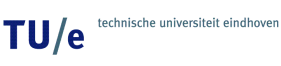
Introduction to ASF+SDF

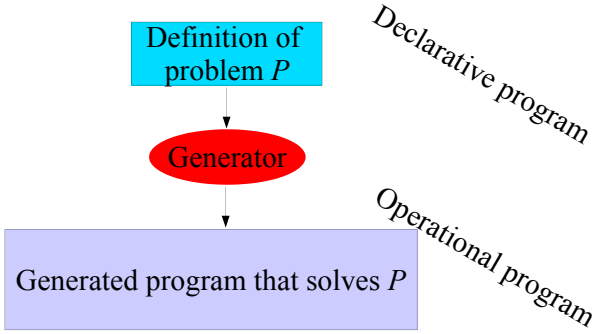
Mark van den Brand
Paul Klint
Jurgen Vinju



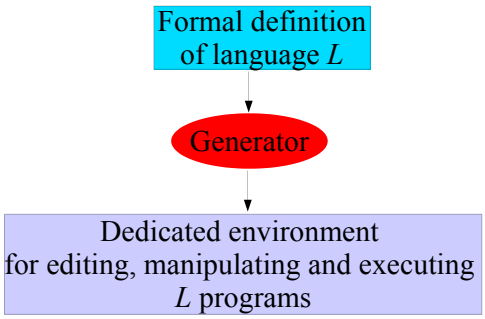
ASF+SDF

- Goal: defining languages & manipulating programs
- SDF: Syntax definition Formalism
  - lexical & context-free syntax
- ASF: Algebraic Specification Formalism
  - static & dynamic semantics; fact extraction
- ASF+SDF Meta-Environment: IDE for ASF+SDF
- Manuals/documentation: www.meta-environment.org

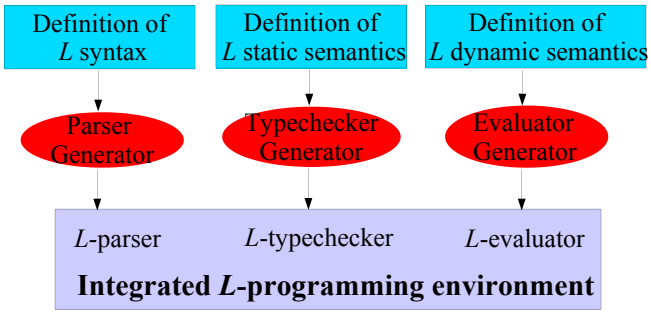
What is a Program Generator?



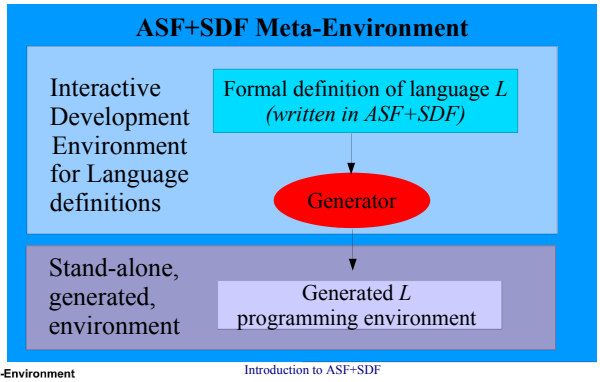
Programming Environment Generator



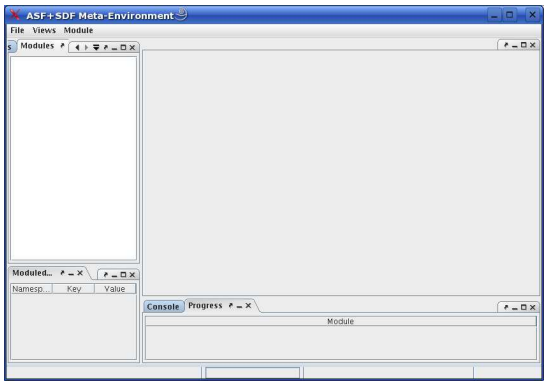
Programming Environment Generator = collection of program generators



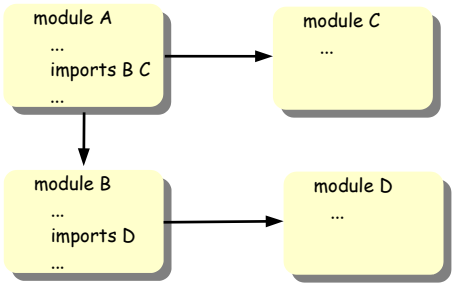
ASF+SDF Meta-Environment



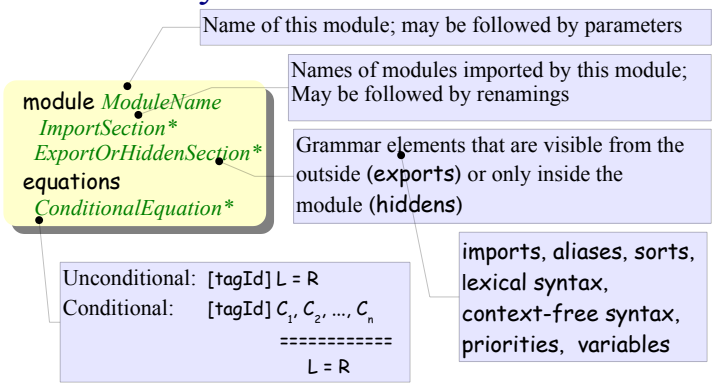
Typing asfsdf-meta ...



Anatomy of an ASF+SDF Specification



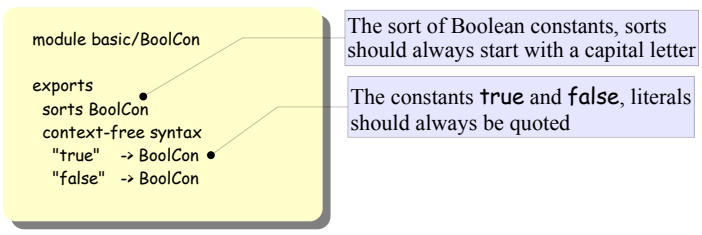
Anatomy of an ASF+SDF Module



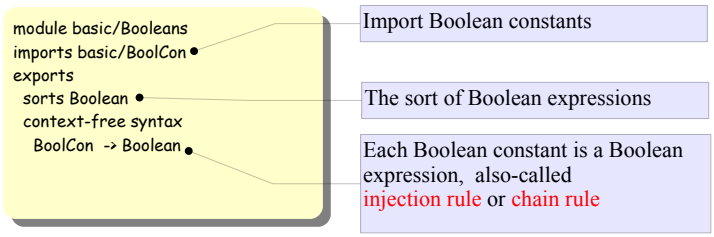
Plan

- **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**
  - Step5: define a fact extractor

BoolCon: Boolean Constants



Booleans (1)



Booleans (2)

```
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 (3)

```
context-free syntax
  "not" (Boolean)       -> Boolean
  "(" Boolean ")"       -> Boolean {bracket}

context-free priorities
  Boolean "&" Boolean -> Boolean {left} >
  Boolean "|" Boolean -> Boolean {left}
```

Shorthand for defining the infix operators and & and or |.
Both are left-associative (left).
These rules are promoted to context-free syntax rules

Booleans (4)

```
hiddens
  context-free start-symbols
    Boolean

  imports
    basic/Comments


  variables
    "Bool"[0-9\']*  -> Boolean
```

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

Import the standard comment conventions for equations

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

Booleans (5)

```
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
```

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.

Fixed versus user-defined syntax

```
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
```

Skeleton syntax for equations

Terms that use user-defined syntax

Booleans (6)

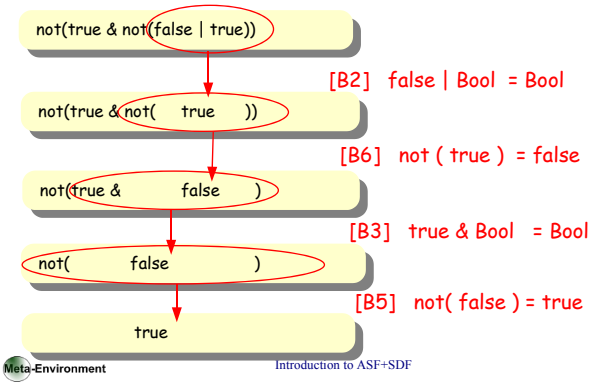
The term

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

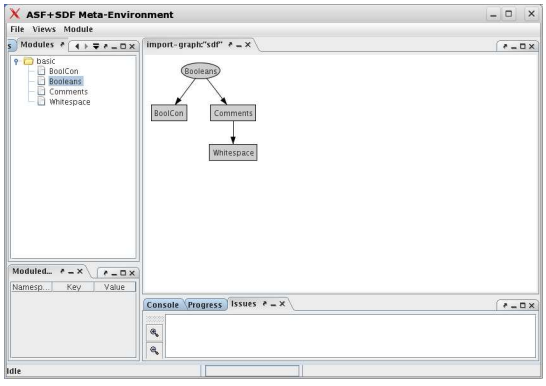
Rewrites to

```
true
```

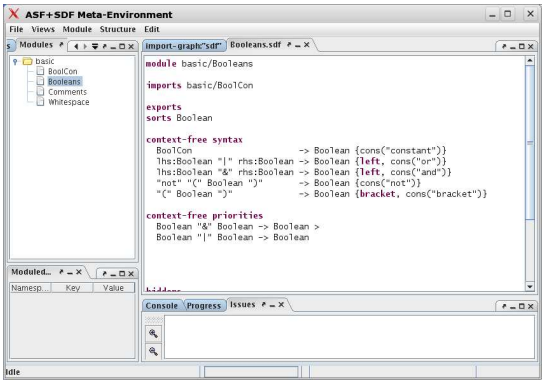

Booleans (7)



Opening Booleans



Editing Booleans.sdf



Booleans: summary

- 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**
  - Step5: define a fact extractor

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; 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

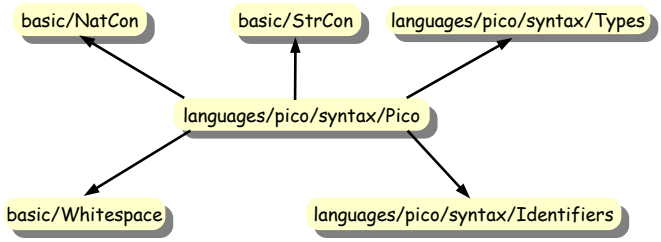
Why is it written in this clumsy style?

(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**
  - Step5: define a fact extractor

Step 1: Define syntax for Pico



Pico-syntax, 1

```
module languages/pico/syntax/Pico
 imports
    languages/pico/syntax/Identifiers
    languages/pico/syntax/Types
    basic/NatCon
    basic/StrCon
 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
```

Sorts and syntax rules for program and declarations

List of zero or more statements separated by ;

* zero or more

\+ one or more

Pico-syntax, 2

Syntax rules for statements

```
PICO-ID ":=" EXP                                  -> STATEMENT

"if" EXP "then" {STATEMENT ";"}*
    "else" {STATEMENT ";"}* "fi"                  -> STATEMENT

"while" EXP "do" {STATEMENT ";"}* "od"            -> STATEMENT
```


Pico-syntax, 3

Syntax rules for expressions

```
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
```

The sort NatCon is imported from basic/NatCon

The sort StrCon is imported from basic/StrCon

The three operators are left-associative

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

Identifiers

```
module languages/pico/syntax/Identifiers
 exports
  sorts PICO-ID

  lexical syntax
    [a-z] [a-z0-9]* -> PICO-ID

  lexical restrictions
    PICO-ID -/- [a-z0-9]
```

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

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

A lexical restriction: is aaa three, two or one identifiers? -/- can be used to define longest match

Pico-Types

```
module languages/pico/syntax/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 languages/pico/syntax/Pico defines (together with the imported modules) 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 times S
  - {S1 S2}* zero or more times S1 separated by S2
  - {S1 S2}+ one or more times S1 separated by S2
- 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;

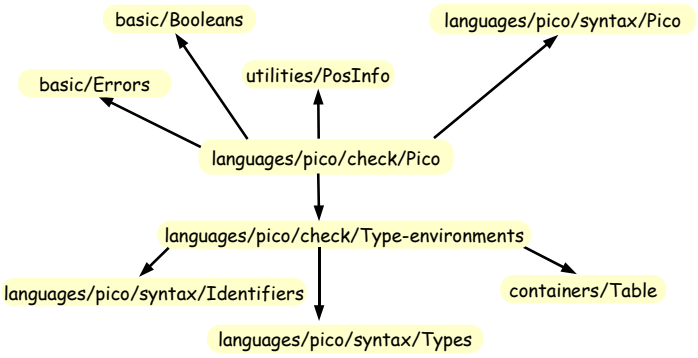
Plan

- **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**
  - Step5: define a fact extractor

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



Type-environments

```
module Type-environments

imports
   languages/pico/syntax/Identifiers
   languages/pico/syntax/Types
   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 ⇒ PICO-ID
Value ⇒ 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-typechecking (1)

```
module languages/pico/check/Pico
imports basic/Booleans
        basic/Errors
        languages/pico/syntax/Pico
        languages/pico/check/Type-environments

        utilities/PosInfo[EXP]
        utilities/PosInfo[PICO-ID]
        utilities/PosInfo[PROGRAM]

exports
  context-free syntax
    tcp(PROGRAM)      -> {Error ",")*
```

Standard error messages

Position information (source code coordinates) for three sorts

Check complete program

Pico-typechecking (2)

```
hiddens
  context-free syntax
    tcd(DECLS)                      -> TENV
    tcits({ID-TYPE ","}*, TENV)     -> TENV
    tcit(ID-TYPE, TENV)             -> TENV

    tcs({STATEMENT ";"}*, TENV)  -> {Error ",")*
    tcst(STATEMENT, TENV)        -> {Error ",")*
    tce(EXP, TYPE, TENV)         -> {Error ",")*
```

Auxiliary (hidden) functions

Check declarations by building a TENV representing the declarations

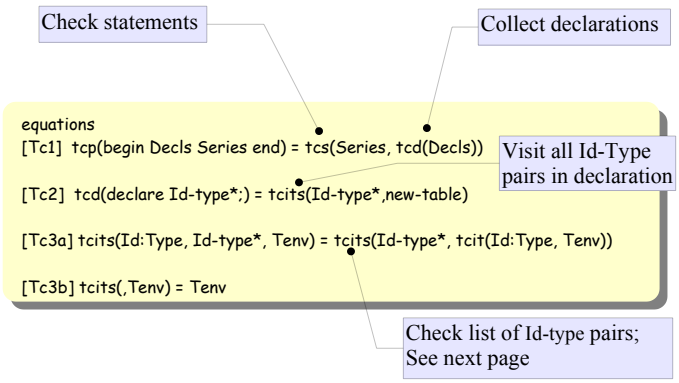
Check statements and expressions:
- using that TENV
- return list of errors

Pico-typechecking (3)

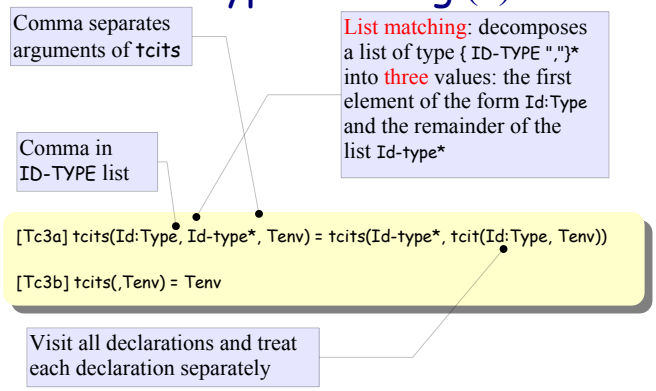
```
hiddens
  variables
    "Message"          -> StrCon
    "Error*" [0-9\']*  -> {Error ",")*
    "Decls"[0-9\']*    -> DECLS
    "Exp"[0-9\']*      -> EXP
    "Id"[0-9\']*       -> PICO-ID
    "Id-type*"[0-9\']* -> { ID-TYPE ","}*
    "Nat-con"[0-9\']*  -> NatCon
    "Series"[0-9\']*   -> {STATEMENT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

Declare a bunch of variables

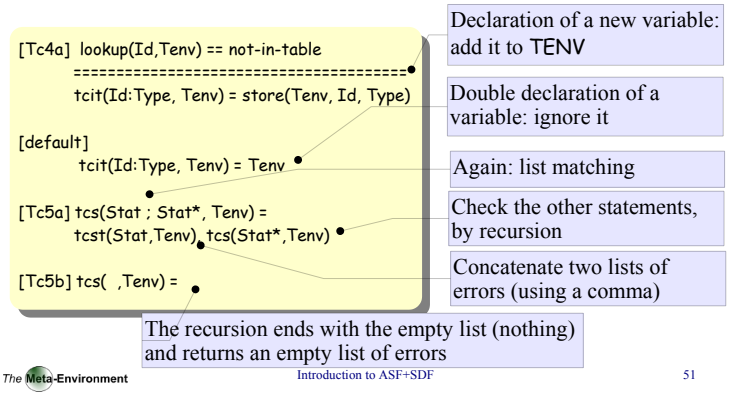
Pico-typechecking (4)



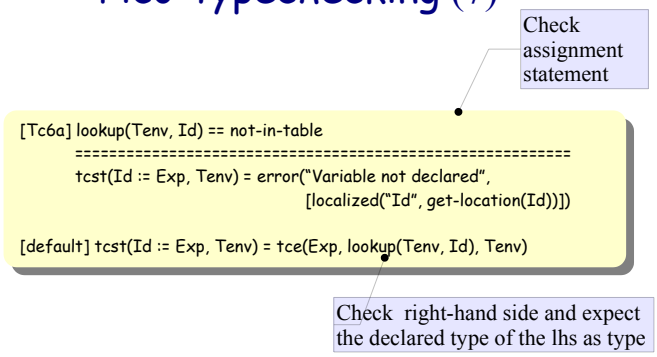
Pico-typechecking (5)



Pico-typechecking (6)



Pico-typechecking (7)



Pico-typechecking (8)

Check if statement

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

Expression should have type natural

Both branches should be type correct, we (re)use the concatenation on lists of errors

Pico-typechecking (9)

Check while statement

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


Pico-typechecking (10)

The expected type of an identifier should be its declared type

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

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

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

Empty list of errors

The elementary types of constants

Pico-typechecking (11)

Check an addition

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

Both arguments should be of type natural

Pico-typechecking (12)

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

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

[default]
       tce(Exp,natural, Tenv) =error("Expression should be of type natural",
                                     [localized{"Expression", get-location(Exp)}])
[default]
       tce(Exp,string, Tenv) =error("Expression should be of type string",
                                     [localized{"Expression", get-location(Exp)}])
```

Check - and ||

In all other cases: return an error

Pico-typechecking (13)

The function **start** connects the Pico-typechecker to the Meta-Environment (compare with main in C or Java)

- **PROGRAM**: the actual sort of the input program
- **Program**: the actual Pico program (a variable)

```
[Main] start(PROGRAM, Program) =
       start(Summary, summary("pico-check",
                              get-file-name(get-location(Program)),
                              [tcp(Program)]))
```

The result is of the sort **Summary** and is obtained by applying the typechecking function **tcp** to the input program and including it in a summary.

Typechecking an erroneous program

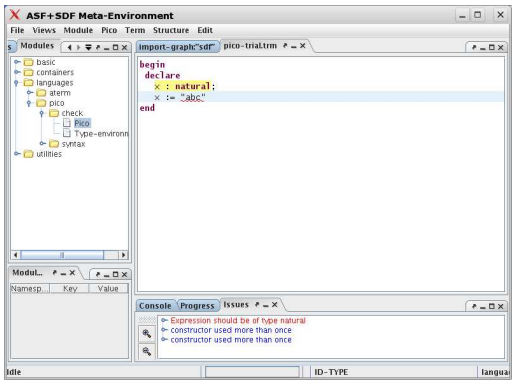
The term

```
tcp(
    begin
     declare
      x : natural;
      x := "abc"
    end
)
```

reduces to

```
summary("pico-check",
        "/home/paulk/pico.trm",
        [error("Expression should be of type natural",
               [localized("Expression",
                          area-in-file("/home/paulk/pico.trm",
                                       area(4,  4, 12, 38, 5)))])])
```


In the Meta-Environment



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 on either side
  - 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 (1)

- The module languages/pico/check/Pico defines (together with the imported modules) 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**
  - Step5: define a fact extractor

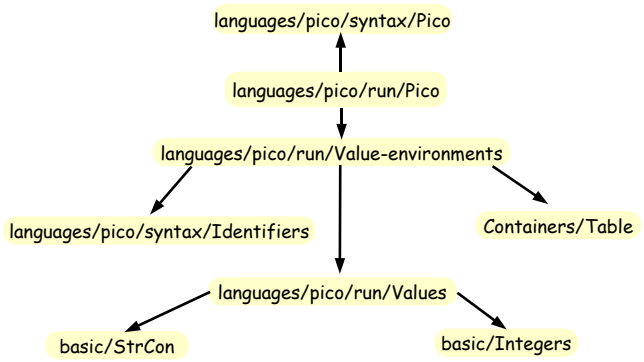
Step 3: Define evaluator for PICO

- 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 evaluator



Pico-Values

```
module languages/pico/run/Values

imports basic/Integers
        basic/StrCon
exports
 sorts VALUE
 context-free syntax
  Integer     -> VALUE
  StrCon      -> VALUE
  "nil-value" -> VALUE
```

Integers and Strings can occur as values during execution

nil-value denotes error values

Value-environments (1)

```
module languages/pico/run/Value-environments

imports languages/pico/syntax/Identifiers
        languages/pico/run/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-evaluator (1)

```
module languages/pico/run/Pico

imports languages/pico/syntax/Pico
        languages/pico/run/Value-environments
        basic/Strings
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 expression

Pico-evaluator (2)

```
hiddens
  imports basic/Comments

context-free start-symbols
    VALUE-ENV PROGRAM
```

We need layout and comments to write equations (but hide them!)

Programs and value environments need to be parsed so declare a start-symbol for them.

Pico-evaluator (3)

```
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
    "Program" [0-9\']* -> PROGRAM
```


Pico-evaluator (4)

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-evaluator (5)

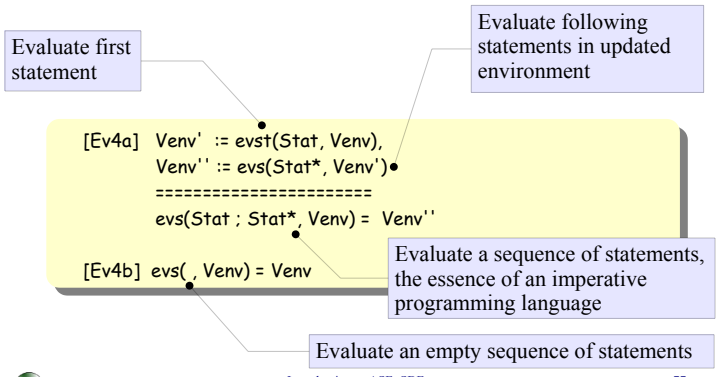
```
[Ev2]  evd(declare Id-type*;) = evits(Id-type*)

[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() = []
```

Initialize a natural variable

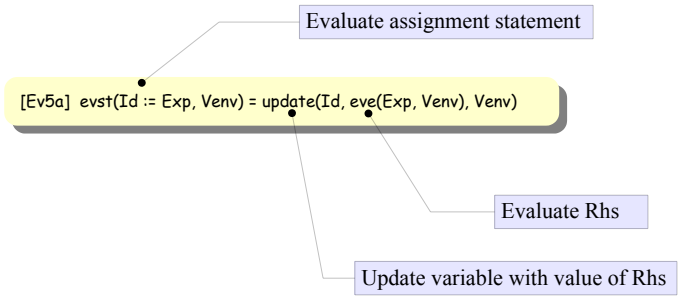
Initialize a string variable

Create a new table for the empty list of declarations

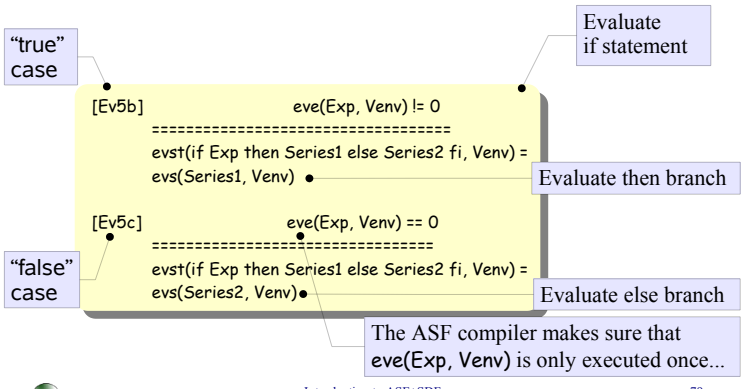
Pico-evaluator (6)



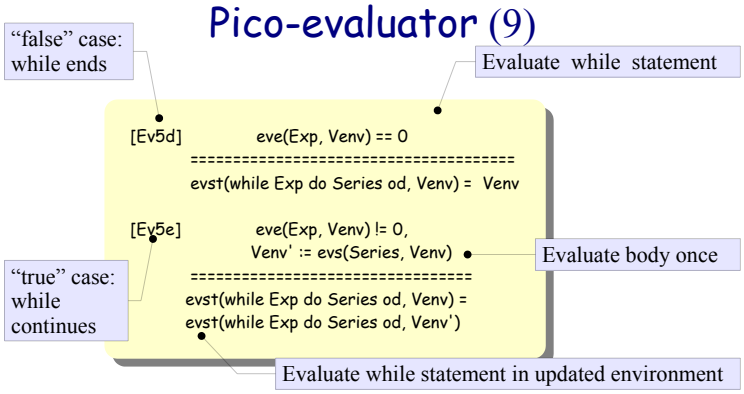
Pico-evaluator (7)



Pico-evaluator (8)



Pico-evaluator (9)



Pico-evaluator (10)

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

Pico-evaluator (11)

Evaluate addition

Evaluate left operand

Evaluate right operand

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

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

Funny: two different "+" signs, that look the same! The left one is on EXP, the right one on Integer

Pico-evaluator (12)

Evaluate - and ||

```
[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
```

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 (1)

- The module languages/pico/run/Pico (together with the imported modules) define 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!

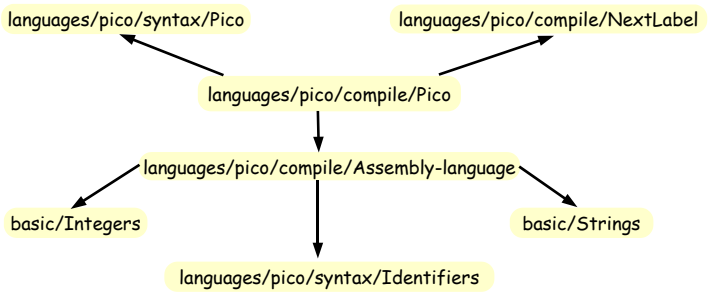
Plan

- **Booleans**
- **Steps towards a Pico environment**
  - Step 1: define syntax
  - Step 2: define a typechecker
  - Step 3: define an evaluator
  - *Step 4: define a compiler*
- **Traversal functions**
  - Step5: define a fact extractor

Pico compiler

- A simple compiler:
  - input a Pico program
  - output: Assembly for a stack based instruction set (in the same spirit as Java bytecode)
- This is a classic example of a program transformation

Pico compiler



AssemblyLanguage (1)

```
module languages/pico/compile/AssemblyLanguage

imports basic/Integers
        basic/Strings
        languages/pico/syntax/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 languages/pico/compile/NextLabel
imports AssemblyLanguage
exports
  context-free syntax
    "nextlabel" "(" Label ")" -> Label

hiddens
  variables
    "Char*"[0-9]* -> [a-z0-9]+
equations

[1] nextlabel(label(Char+)) = label(Char+  x)
```

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

/ "(" 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-compiler (1)

```
module languages/pico/compile/Pico
imports languages/pico/syntax/Pico
        languages/pico/compile/AssemblyLanguage
        languages/pico/compile/NextLabel

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

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

The main compiler function: translate program (trp)

Translation of statements generates instructions and new labels: <{Instr ";"}+, Label>

Pico-compiler (2)

```
hiddens
  context-free start-symbols
    PROGRAM Instrs
```

Define relevant start symbols

Pico-compiler (3)

```
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
    "Str"[0-9\']*       -> String

    "Instr*"[0-9\']*    -> {Instr ";"}+
    "Label" [0-9\']*    -> Label
    "Program"           -> PROGRAM
```


Pico-compiler (4)

```
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
```

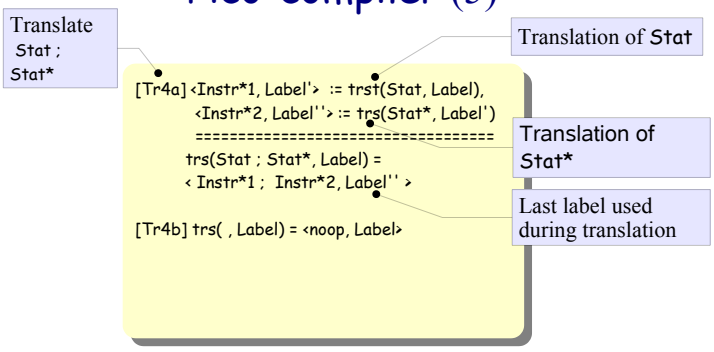
Translate a program

Translate a declaration section

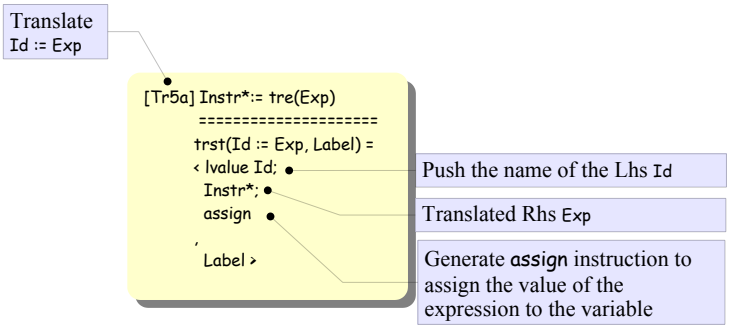
Translate a variable declaration

Translate an empty list

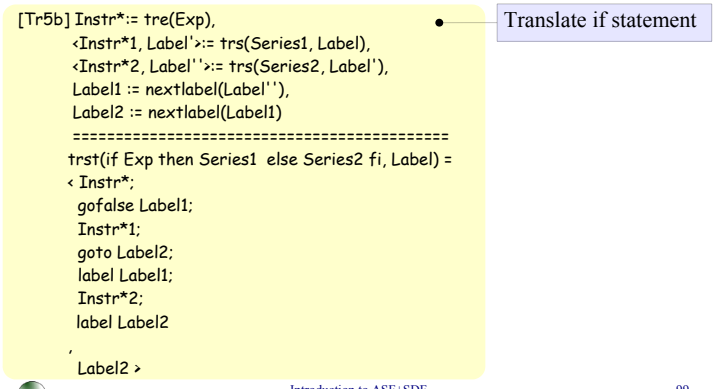
Pico-compiler (5)



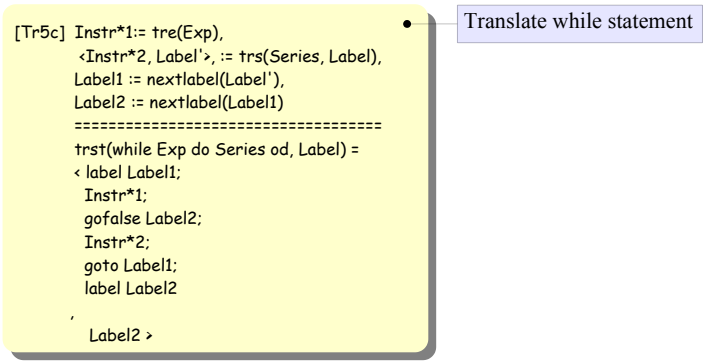
Pico-compiler (6)



Pico-compiler (7)



Pico-compiler (8)



Pico-compiler (9)

```
[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 module languages/pico/compile/Pico defines (together with the imported modules) 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
- This is just another transformation

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***
  - Step5: define a fact extractor

Traversal Functions (1)

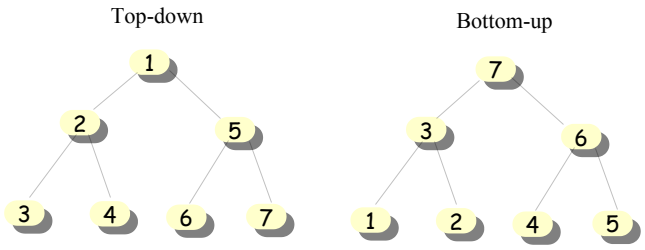
- Many functions have the characteristic that they traverse the tree *recursively* and only do something interesting at a few nodes
- Example: count the identifiers in a program
- Using a recursive (inductive) definition:
  - # of equations is equal to number of syntax rules
  - think about Cobol or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

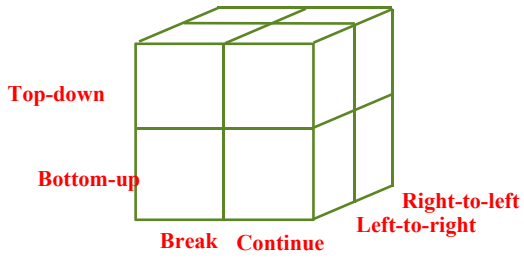
Top-down versus Bottom-up



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

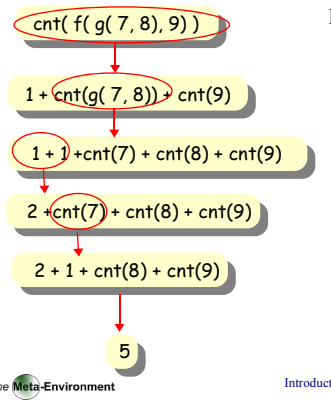
Count this node

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

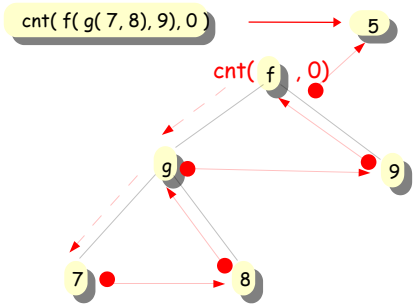
Accumulated value

Traversed tree (matches every node)

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

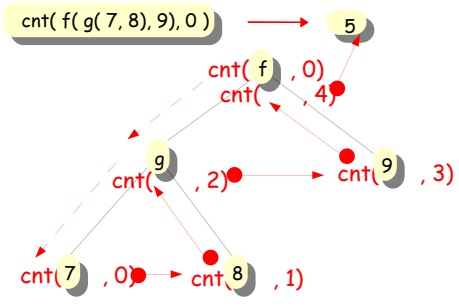
→ 11

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

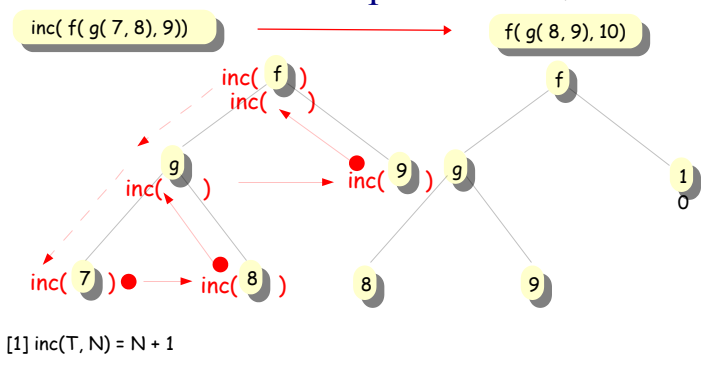
Leaf N

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

Amount N2

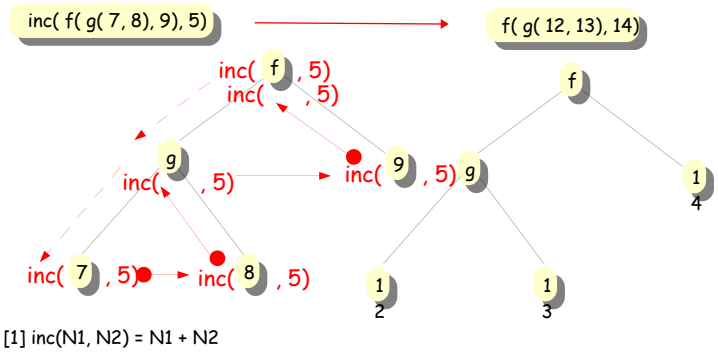
Leaf N1

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



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

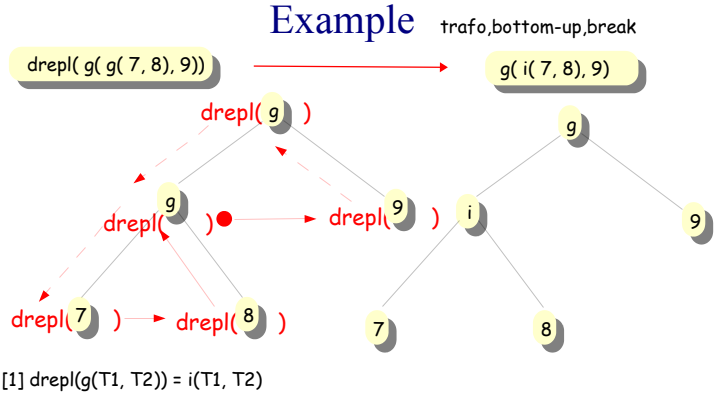
Only the deepest occurrences of g are replaced

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



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)
```

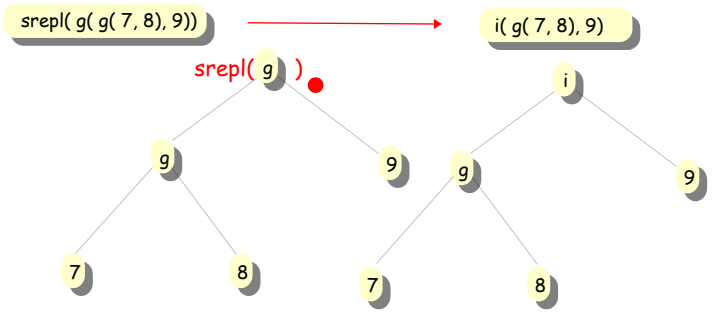
A top-down transformer that stops after first matching node

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

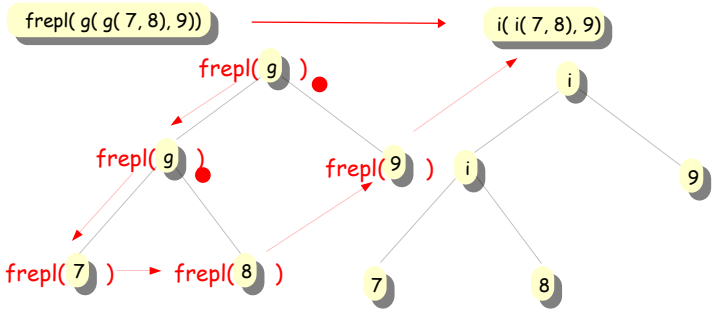
top-down and bottom-up have here the same effect

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

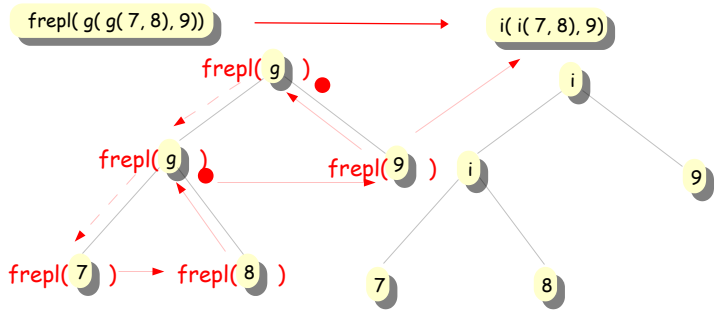
trafo, top-down, continue



[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 Stats2) =
          IF Expr OptThen Stats1 ELSE Stats2 END-IF
```

Add missing END-IF keywords

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

Equations for the two cases

A funny Pico typechecker

- Replace all variables by their declared type:
  - x +3 ⇒ type(natural) + type(natural)
- Simplify type correct expressions:
  - type(natural) + type(natural) ⇒ 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

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**
  - *Step5: define a fact extractor*

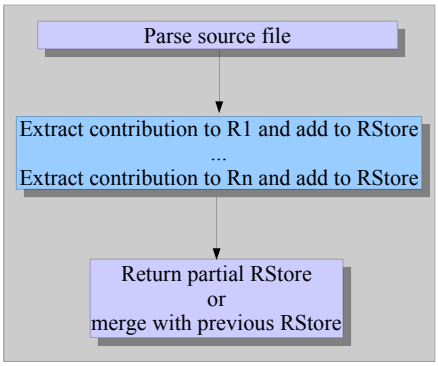
Extracting Facts using ASF+SDF

- Dump-and-Merge: Facts can be extracted per file and be merged later
- Extract-and-Update: Facts are extracted per file and merged with previously extracted RStore
- Both styles can be used, matter of taste

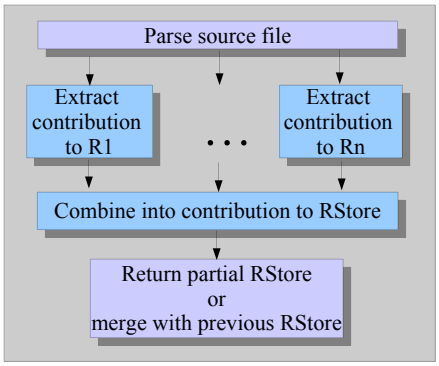
Extracting Facts using ASF+SDF

- Write traversal functions that extract facts from source file
- All-in-One: one function extracts all facts in one traversal
  - typically an accumulator that returns an Rstore
  - makes contribution to named relations in Rstore
- Separation-of-Concerns: separate function for each fact to be extracted
- SoC is more modular and preferred

All-in-One Strategy



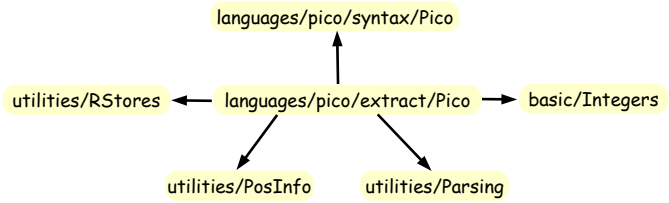
Separation of Concerns Strategy



Extracting Facts from Pico Programs

- Use `RStore` for creating and extending Rstores
- Use `utilities/PosInfo[Sort]` for getting position information for specific sorts
- Use `utilities/Parse[Sort]` for unparsing a tree to a string (unparse-to-string)
- Write (example) functions
  - `cflow` for extracting control flow
  - `countStatements` for making a statement histogram

Fact extractor



RStores

- A set of typed set/relational variables (see Rscript)
- Primitives for
  - Creating a new Rstore (`create-store`)
  - Declaring a new variable with its type (`declare`)
  - Setting/getting the value of a variable (`set`/`get`)
  - Modifying the value of a variable by inserting, deleting or replacing elements (`insert`, `replace`, `delete`, `lookup`)
  - Changing integer variables (`inc`, `dec`)

Sets-and-Relations

- Provides most of the Rscript functionality inside ASF+SDF
- Uses one common element type: `Relem`
  - less type-safe than RScript
- Provides all Rscript primitives:
  - `union`, `difference`, `intersection`
  - `size`, `subset`, `superset`, `element-of`, ...

Fact extractor

```
module languages/pico/extract/Pico

imports utilities/RStores
imports languages/pico/syntax/Pico
imports basic/Integers
imports utilities/Parsing[PICO-ID]
imports utilities/Parsing[STATEMENT]
imports utilities/Parsing[EXP]
imports utilities/PosInfo[STATEMENT]
imports utilities/PosInfo[EXP]

hiddens
context-free syntax
  controlFlow(PROGRAM, RStore)           -> RStore
  statementHistogram(PROGRAM, RStore) -> RStore
```

Declare the two extraction functions

Fact extractor

`countStatements` is defined as traversal function that will be applied to the sorts `PROGRAM` and `STATEMENT`; It accumulates an `RStore`

```
context-free syntax
  countStatements(PROGRAM, RStore)      -> RStore
                                           {traversal(accu, bottom-up, continue)}
  countStatements(STATEMENT, RStore) -> RStore
                                           {traversal(accu, bottom-up, continue)}
  cflow({STATEMENT ";"}*)                -> <RElem,RElem,RElem>

context-free start-symbols
  PROGRAM RStore RElem
```

`cflow` is an ordinary function that returns triples for each Pico language construct of the form:
<entry points, internal connections, exit points>

Fact extractor

```
variables
  "Program" [0-9]* -> PROGRAM
  "Decls" [0-9]*   -> DECLS
  "Stat" [0-9]*    -> STATEMENT
  "Stat*" [0-9]*   -> {STATEMENT ";"}*
  "Stat+" [0-9]*   -> {STATEMENT ";"}+
  "Exp" [0-9]*     -> EXP
  "Id" [0-9]*      -> PICO-ID
  "Entry" [0-9]*   -> RElem
  "Exit" [0-9]*    -> RElem
  "Rel" [0-9]*     -> RElem
  "Control" [0-9]* -> RElem
variables
  "Store"[0-9]*    -> RStore {strict}
  "Int" [0-9]*     -> Integer {strict}
```


Histogram

```
equations

[hist] statementHistogram(Program, Store) =
       countStatements(Program,
                       declare(Store,
                               StatementHistogram,
                               rel[str,int]))
equations

[] countStatements(Id := Exp, Store)   =
   inc(Store, StatementHistogram, "Assignment")
[] countStatements(if Exp then Stat*1 else Stat*2 fi, Store) =
   inc(Store, StatementHistogram, "Conditional")
[] countStatements(while Exp do Stat* od, Store)  =
   inc(Store, StatementHistogram, "Loop")
```

Declare the variable StatementHistogram and apply countStatements

Only declare the cases of interest and increment relevant counter

Cflow: series

```
equations

[cfg] Store1 := declare(Store, ControlFlow, rel[<str,loc>,<str,loc>]),
     <Entry, Rel, Exit> := cflow(Stat*)
     ========================================================
     controlFlow(begin Decls Stat* end, Store) =
     set(Store1, ControlFlow, Rel)

[cfg-1]
  <Entry1, Rel1, Exit1> := cflow(Stat),
  <Entry2, Rel2, Exit2> := cflow(Stat+)
  ================================
  cflow(Stat ; Stat+) =
  < Entry1, union(Rel1, union(Rel2, product(Exit1, Entry2))), Exit2 >

[cfg-2] cflow() = <{}, {}, {}>
```

Declare the variable ControlFlow and apply cflow

Compute controlflow for a series of statements: <Entry1, Rel1∪Rel2∪(Exit1×Entry2), Exit2>

Cflow: while

```
equations

[cfg-3]
  <Entry, Rel, Exit> := cflow(Stat*),
  Control := <unparse-to-string(Exp), get-location(Exp)>
  ================================================
  cflow(while Exp do Stat* od) =
  <
    {Control},
    union(product({Control}, Entry), union(Rel, product(Exit, {Control}))),
    {Control}
  >
```

Compute controlflow for a while statement: <{Control}, ({Control}×Entry)∪Rel∪(Exit×{Control}), {Control}>

Cflow: if

```
[cfg-4]
 <Entry1, Rel1, Exit1> := cflow(Stat*1),
 <Entry2, Rel2, Exit2> := cflow(Stat*2),
 Control := <unparse-to-string(Exp), get-location(Exp)>
 ================================================
 cflow(if Exp then Stat*1 else Stat*2 fi) =
 < {Control},
   union(product({Control}, Entry1),
         union(product({Control}, Entry2),
               union(Rel1, Rel2))),
  union(Exit1, Exit2) >

[default-cfg]
 Control := <unparse-to-string(Stat), get-location(Stat)>
 ================================================
 cflow(Stat) = <{Control}, {}, {Control}>
```

Compute controlflow for an if statement:
<{Control},
({Control}×Entry1)∪
({Control}×Entry2)∪
Rel1∪Rel2,
Exit1∪Exit2>

Connecting the pieces ...

```
[main] Store1 := create-store(),
       Store2 := statementHistogram(Program, Store1),
       Store3 := controlFlow(Program, Store2)
       ================================================
      start(PROGRAM, Program) = start(RStore, Store3)
```

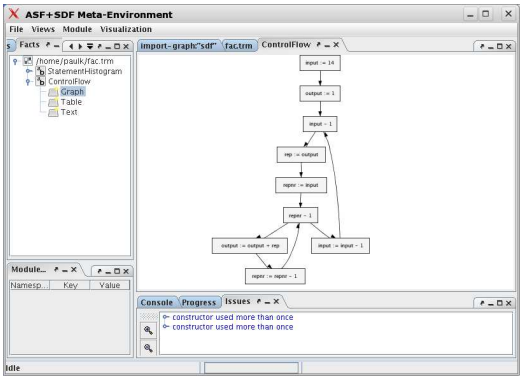
Create a new RStore

Add histogram facts

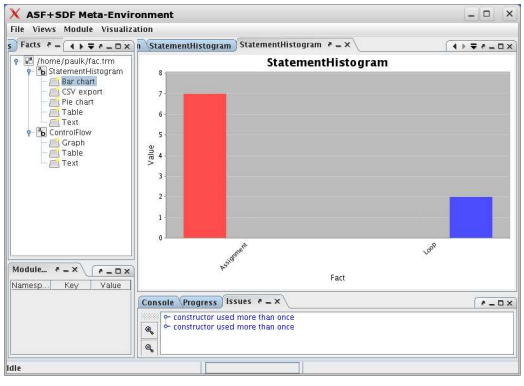
Add controlflow facts

Extraction of a given PROGRAM will result in an RStore

Graph view (factorial)



Barchart view (factorial)



Further reading

- www.meta-environment.org (select Documentation):
  - Guided Tour: Playing with Booleans (flash)
  - ASF+SDF by Example
  - Writing Language Definitions in ASF+SDF
  - The Language Specification Formalism ASF+SDF
  - The Syntax Definition Formalism SDF
  - An Explanation of Error Messages of SDF (draft)
  - An Explanation of Error Messages of ASF (draft)
  - The Architecture of The Meta-Environment