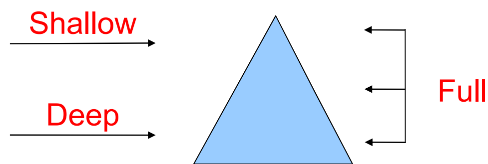
Term Replacement

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



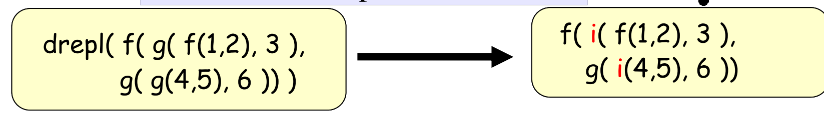
Deep replacement

```
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

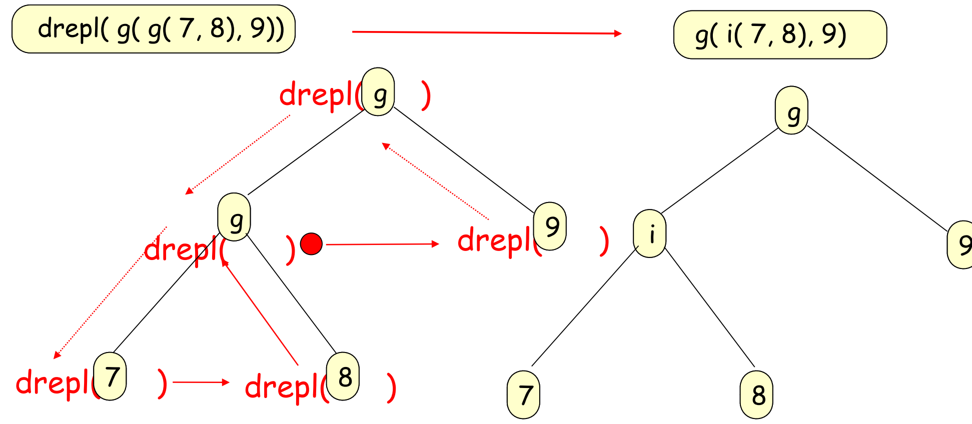
A bottom-up transformer that stops after first matching node

Only the deepest occurrences of g are replaced



Example

trafo,bottom-up,break



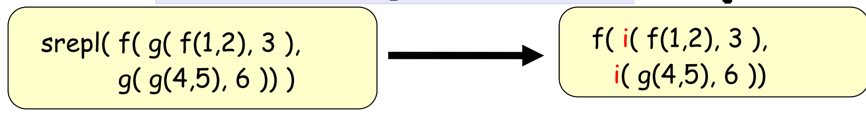
[1] drepl(g(T1, T2)) = i(T1, T2)

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)
```

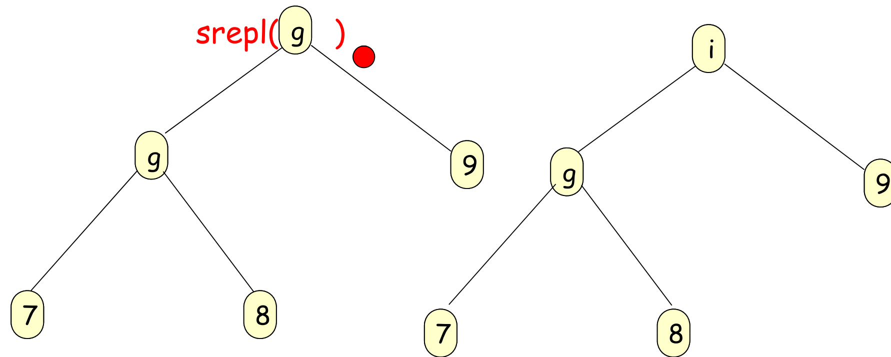
A top-down transformer that stops after first matching node

Only the outermost occurrences of g are replaced



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

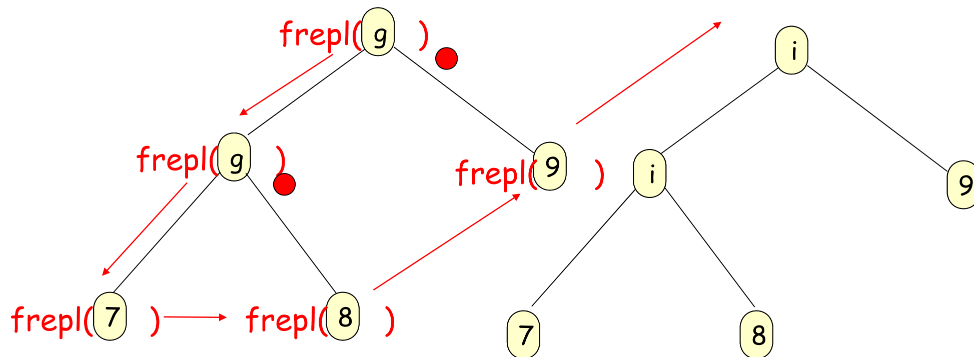
top-down and bottom-up have here the same effect

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

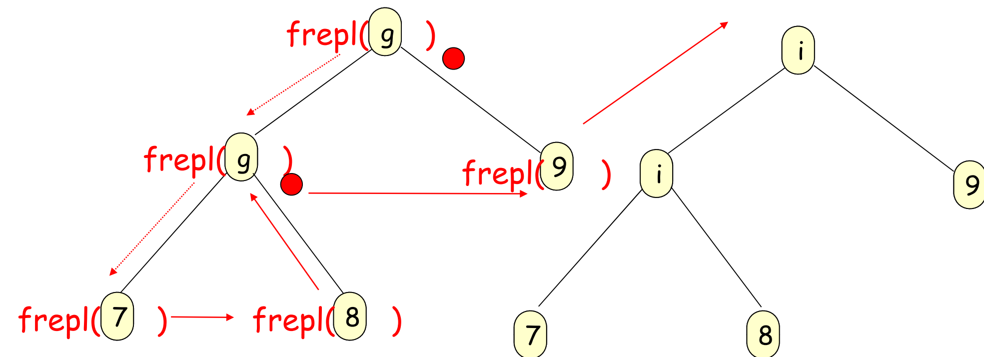
trafo, top-down, continue



[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

Disambiguation via traversals (1)

- Semantic directed disambiguation
  - Based on the concept of rewriting parse forests.
  - Applications
    - C typedefs
    - Nested constructs in COBOL

Disambiguation via traversals (2)

- Application C typedefs:
  - In C certain identifiers can be parsed as either type identifiers or variable identifiers due to operator overloading:

    `{ Bool *b1; }`
  - The above statement is either
    - a statement expression multiplying the `Bool` and `b1` variables,
    - a declaration of a pointer variable `b1` to a `Bool`.
  - The latter derivation is chosen
    - if `Bool` was declared to be a type using a typedef statement somewhere earlier in the program, otherwise
    - the former derivation is chosen.

Disambiguation via traversals (3)

- Applications C typedefs
  - Compact specification that filters one of the ambiguities in the C language.
  - Depending on the existence of a typedef declaration an identifier is either interpreted as
    - a type name or
    - a variable name.
- We construct an environment containing all declared types.

Disambiguation via traversals (4)

- Applications C typedefs
  - If the first Statement after a list of Declarations is a multiplication of an identifier that is declared to be a type, the corresponding sub-tree is removed.

```
context-free syntax

  "types" "[[" Identifier* "]]" -> Env

  filter(CompoundStatement, Env) -> CompoundStatement
   {traversal(accu,trafo,top-down,break)}

equations

  [] Env = types[[Ids1 Id Ids2]]

     ====>

     filter(amb(CSs1,{Decls Id * Expr;Stats},CSs2),Env) = amb(CSs1,CSs2)
```


Disambiguation via traversals (5)

- Note the use of concrete C syntax in this example.
- The filter function searches and removes ambiguous block-statements where the first statement uses an identifier as a variable which was declared earlier as a type.
- Similar rules are added for every part of the C syntax where an ambiguity is caused by the overlap between type identifiers and variable identifiers.
- This amounts to about a dozen rules:
  - they solve the ambiguities and
  - document exactly where our C grammar is ambiguous.

Disambiguation via traversals (6)

- The example resembles the dangling else construction
- It is more complex due to the fact that more constructs are optional.

```
0001 ADD A TO B
0002   SIZE ERROR
0003     ADD C TO D
0004       NOT SIZE ERROR
0005         CONTINUE
0006 .
```


Disambiguation via traversals (7)

- The SIZE ERROR and NOT SIZE ERROR constructs are optional post-fixes of the ADD statement. They can be considered as a kind of exception handling.
- In order to understand what is going on we show a part of a COBOL grammar:

```
Add-stat ::= Add-stat-simple Size-error-phrases
Size-error-phrases ::= Size-error-stats? Not-size-error-stats?
Size-error-stats ::= "SIZE" "ERROR" Statement-list
Not-size-error-stats ::= "NOT" "SIZE" "ERROR" Statement-list
Statement-list ::= Statement*
```


Disambiguation via traversals (8)

- The grammar shows that the COBOL language design does not provide explicit scope delimiters for some deeply nested Statement-lists.
- The NOT SIZE ERROR can be either part of the ADD-statement on line 0001 or 0003.
- The period on line 0006 closes both statements.
- The COBOL definition states that the “dangling” phrase should always be taken with the innermost construct, which is the ADD-statement on line 0003.
- There are 16 of such ambiguities in the COBOL definition.

Disambiguation via traversals (9)

- The nested dangling constructs in COBOL can be filtered using a simple specification.
- There is no context information involved, just a simple structural analysis.
- The rewrite rule filters the derivations where the dangling block of code was not assigned to the correct branch:

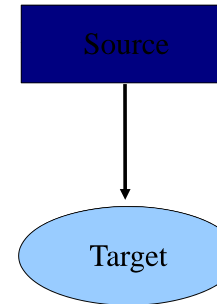
```
equations
[] amb(ASs1,
       AddStatSimple1
       SIZE ERROR Stats1 AddStatSimple2
       NOT SIZE ERROR Stats2, ASs2) = amb(ASs1, ASs2)
```

Compilation is ...

- A well-defined process with well-defined input, output and constraints
- Input: source program in a fixed language with well-defined syntax and semantics
- Output: a fixed target language with well-defined syntax and semantics
- Constraints are known (correctness, performance)
- A batch-like process

Compilation is ...

Single, well defined, source



A batch-like process with clear constraints

Single, well defined, target

Understanding is ...

- An exploration process with as input
  - system artifacts (source, documentation, tests, ...)
  - implicit knowledge of its designers or maintainers
- There is no clear target language
- An interactive process:
  - Extract elementary facts
  - Abstract to get derived facts needed for analysis
  - View derived facts through visualization or browsing

Examples of understanding problems

- Which programs call each others?
- Which programs use which databases?
- If we change this database record, which programs are affected?
- Which programs are more complex than others?
- How much code clones exist in the code?

Examples of the results of understanding

- Textual reports indicating properties of system parts (complexity, use of certain utilities, ...)
- Same, but in hyperlinked format
- Graphs (call graphs, use def graphs for databases)
- More sophisticated visualizations

Other aspects of Understanding

- Systems consist of several source languages
- Analysis techniques over multiple language =>

  a language-independent analysis framework is needed
- A very close link to the source text is needed

Relation-based analysis

- What happens if we use relations for fine grain software analysis (ex: find uninitialized variables)
- What happens if we use a relational calculus?
- What happens if we use term rewriting as basic computational mechanism?
  - relations can represent graphs in the rewriting world
- Could yield a unifying framework for analysis and transformation

Toy program

```
begin declare x : natural, y : natural,
              z : natural;
   x :=  3;
   if     3 then
          z := y + x
   else
          x := 4
   fi
    y := z
end
```

y is undefined

z may be undefined

Toy program

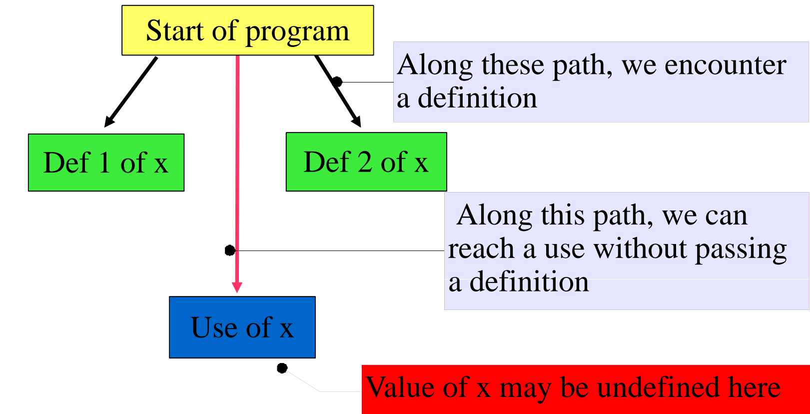
rel[int,str] DEFS = {<1,"x">, <3,"z">, <4,"x">, <5,"y">}

```
begin declare x : natural, y : natural,
              z : natural;
[1] x :=  3;
    if [2] 3 then
       [3] z := y + x
    else
       [4] x := 4
    fi
[5] y := z
end
```

rel[int,str] USES = {<3,"y">, <3,"x">, <5,"z">}

rel[int,int] PRED = {<0,1>, <1,2>, <2,3>,<2,4>, <3,5>,<4,5>}

Finding uninitialized variables



Applying the undefined query

```
begin declare x : natural, y : natural,
              z : natural;
[1] x :=  3;
    if [2] 3 then
       [3] z := y + x
    else
       [4] x := 4
    fi
[5] y := z
end
```

y is undefined

z may be undefined

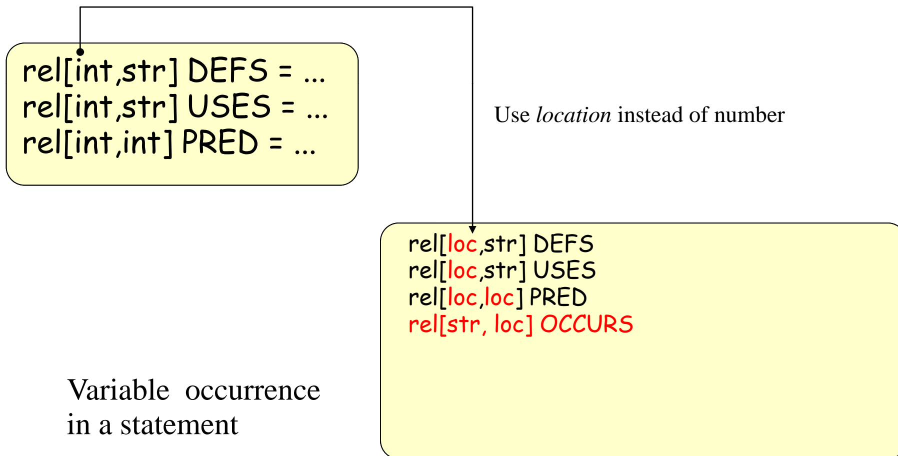
Result:

{<5,"z">, <3,"y">}

Some Questions

- There are several additional questions:
  - In the example so far we have worked with statement numbers but how do we make a connection with the source text?
  - How do we extract relations like PRED and USE from the source text?

Use locations to connect with the source text



Example Rstore

```
rstore(
  <PRED, rel[loc,loc],
           {<area-in-file("/home/.../example.pico", area(4, 2,4, 8,84, 6)),
              area-in-file("/home/.../example.pico", area(5, 2,5, 8,94, 6))>,
            <area-in-file("/home/.../example.pico", area(5, 2,5, 8, 94, 6)),
              area-in-file("/home/.../example.pico", area(6, 2,10, 4, 104, 56))>,
             ... }>,

 <DEFS, {
   <OCCURS, rel[str,loc],
        {<"y", area-in-file("/home/.../example.pico",area(11, 2,11, 3,164, 1))>,
          <"z", area-in-file("/home/.../example.pico", area(11, 7,11, 8,169, 1))>,
        ... }>
     }>
)
```


Extracting Facts

- Goal: extract facts from source code and use as input for queries
- How should fact extraction be organized?
- How to write 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

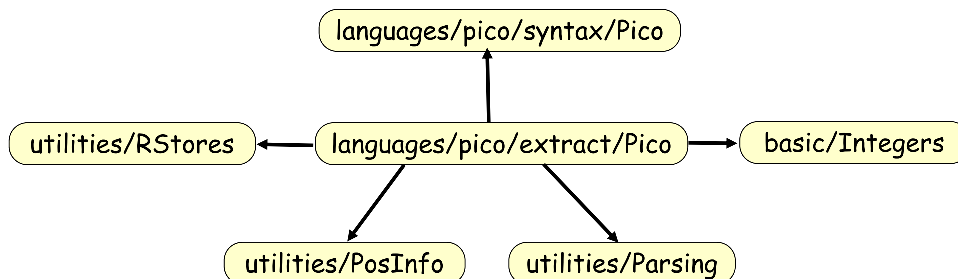
Extracting Facts using ASF+SDF

- 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

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

Declare the variable `ControlFlow` and apply `cflow`

```
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() = <{}, {}, {}>
```

Compute control flow 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 control flow for a while statement:

```
<{Control},
  ({Control}×Entry)∪Rel∪(Exit×{Control}),
 {Control}>
```


Cflow: if

an if statement:

```
[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 control flow for an if statement:

```
<{Control},
 ({Control}×Entry1)∪
 ({Control}×Entry2)∪
 Rel1∪Rel2,
 Exit1∪Exit2>
```

Connecting the pieces ...

Create a new RStore

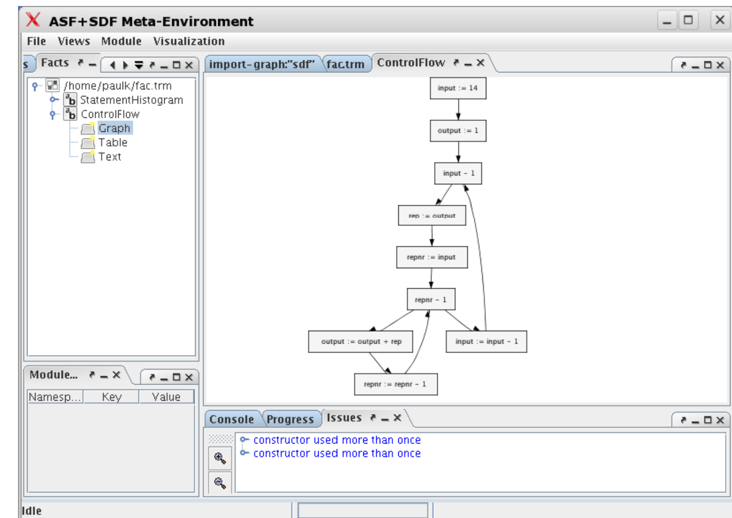
Add histogram facts

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

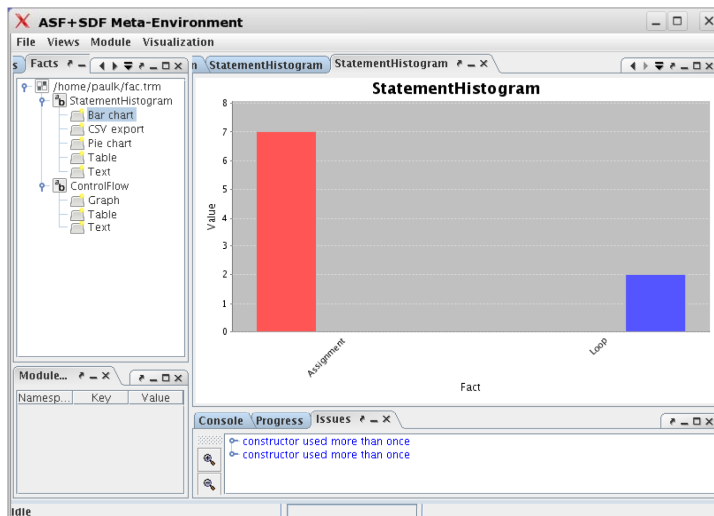
Add control flow facts

Extraction of a given PROGRAM will result in an RStore

Graph view (factorial)



Barchart view (factorial)



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