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?

Definition of problem \( P \)

Generator

Generated program that solves \( P \)

Programming Environment Generator

Formal definition of language \( L \)

Generator

Dedicated environment for editing, manipulating and executing \( L \) programs
Programming Environment Generator
= collection of program generators

ASF+SDF Meta-Environment

Typing asf sdf-meta ...

Anatomy of an ASF+SDF Specification
Anatomy of an ASF+SDF Module

- **ModuleName**
- **ImportSection**
- **ExportOrHiddenSection**
- **equations**
- **ConditionalEquation**

**Unconditional:** \[ \text{unconditional} \]

**Conditional:** \[ \text{conditional} \] \( \begin{align*}
  \text{[tags]} & \quad \text{L} = R \\
  \text{[tags]} & \quad C_1, C_2, \ldots, C_n
\end{align*} \]

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

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 5: define a fact extractor

BoolCon: Boolean Constants

- The sort of Boolean constants, sorts should always start with a capital letter
- The constants `true` and `false`, literals should always be quoted

Booleans (1)

- Import Boolean constants
- The sort of Boolean expressions
- Each Boolean constant is a Boolean expression, also called injection rule or chain rule
Booleans (2)

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

Booleans (3)

The prefix function not

context-free syntax

\texttt{not} (\texttt{Boolean}) \rightarrow \texttt{Boolean}

context-free priorities

Boolean \& Boolean \rightarrow \texttt{Boolean (left)}

Boolean | Boolean \rightarrow \texttt{Boolean (left)}

\& has higher priority than |

Example:

Boolean & Boolean | Boolean

is interpreted as:

(Boolean & Boolean) | Boolean

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)

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

import basic/Comments

variables "Bool[0-9\"]" \rightarrow Boolean

Booleans (5)

equations

$[B1] \text{true} \text{ if } \text{Bool} = \text{true}$

$[B2] \text{false} \text{ if } \text{Bool} = \text{false}$

$[B3] \text{true} \& \text{Bool} = \text{true}$

$[B4] \text{false} \& \text{Bool} = \text{false}$

$[B5] \text{not} (\text{false}) = \text{true}$

$[B6] \text{not} (\text{true}) = \text{false}$

The meaning of \&, | and \texttt{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

Skeleton syntax for equations

1. true | Bool = true
2. false | Bool = Bool
3. true & Bool = Bool
4. false & Bool = false
5. not(false) = true
6. not(true) = false

Terms that use user-defined syntax

Booleans (6)

The term:

\[ \text{not(true \& not(false \& true))} \]

Rewrites to:

true

Booleans (7)

Opening Booleans
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 5: define a fact extractor

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

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,
  rep : natural;
input := 14;
output := 1;
while input - 1 do
  rep := input;
  while rep - 1 do
    output := output + rep;
    rep := rep - 1;
  end;
input := input - 1;
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/NetCon
  basic/StrCon
exports
  sorts
    PROGRAM
    ID
    TYPE
    STATEMENT
    EXP
    context-free start-symbols
    PROGRAM
    context-free syntax
  "begin" DECLARES (STATEMENT * ) * end!
  → PROGRAM
  ID → TYPES
  DECLARE "ID : TYPE *
  " → DECLS
  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 "]" | "." -> STATEMENT
"while" EXP "do" (STATEMENT "]" | "." -> STATEMENT
```

**Pico-syntax, 3**

Syntax rules for expressions

```
PICO-ID
NatCon
StrCon
EXP:"" EXP
EXP:" EXP
EXP:" EXP
EXP:" EXP
EXP:" EXP
context-free priorities
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 a 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,
input := 14;
output := 1;
while input > 0 do
rep := output;
repnr := input;
while repnr > 0 do
output := output + rep;
repnr := repnr - 1
od;
input := input - 1
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"
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
{\-}\[0-9]+ UnsignedInt       → SignedInt
UnsignedInt "\.[0-9]+ ([eE] SignedInt)? → UnsignedReal
UnsignedInt [eE] SignedInt    → SignedReal
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

**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 5: 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
**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
- 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 ",")

**Pico-typechecking (2)**

- main functions
- check declarations by building a TENV representing the declarations
- check statements and expressions: using that TENV; return list of errors

**Pico-typechecking (3)**

- auxiliary (hidden) functions
- declare a bunch of variables
Pico-typechecking (8)

Check if statement

\[
\text{Tc6b} \quad \text{tcs}(\text{if Exp then Series1 else Series2 \text{ fi}, Tenv}) = \\
\text{tcs(Exp, natural, Tenv)}, \quad \text{tcs(Series1, Tenv)}, \quad \text{tcs(Series2, Tenv)}
\]

Expression should have type natural

Both branches should be type correct, we reuse the concatenation on lists of errors

Pico-typechecking (9)

Check while statement

\[
\text{Tc6c} \quad \text{tcs(while Exp do Series od, Tenv)} = \\
\text{tcs(Exp, natural, Tenv)}, \quad \text{tcs(Series, Tenv)}
\]

Pico-typechecking (10)

The expected type of an identifier should be its declared type

\[
\text{Tc7a} \quad \text{Type = lookup(Id, Tenv)} \\
\]

Empty list of errors

The elementary types of constants

\[
\text{Tc7b} \quad \text{tcs(Nat-con, natural, Tenv)} = \\
\text{Tc7c} \quad \text{tcs(Str-con, string, Tenv)} = \\
\]

Pico-typechecking (11)

Check an addition

\[
\text{Tc7d} \quad \text{tcs(Exp1 + Exp2, natural, Tenv)} = \\
\text{tcs(Exp1, natural, Tenv)}, \quad \text{tcs(Exp2, natural, Tenv)}
\]

Both arguments should be of type natural
Pico-typechecking (12)

\[
\begin{align*}
[Tc7e] & \quad \text{tce}(\text{Exp} \cdot \text{Exp}^2, \text{natural}, \text{Env}) = \\
& \quad \text{tce}(\text{Exp}, \text{natural}, \text{Env}), \text{tce}(\text{Exp}^2, \text{natural}, \text{Env}) \\
[Tc7f] & \quad \text{tce}(\text{Exp} \mid \text{Exp}, \text{string}, \text{Env}) = \\
& \quad \text{tce}(\text{Exp}, \text{string}, \text{Env}), \text{tce}(\text{Exp}, \text{string}, \text{Env}) \\
& \quad [\text{default}] \\
& \quad \text{tce}(\text{Exp}, \text{natural}, \text{Env}) = \text{error}(\text{"Expression should be of type natural"}, \\
& \quad \quad [\text{localized}("\text{Expression}", \text{get-location}(\text{Exp}))]) \\
& \quad [\text{default}] \\
& \quad \text{tce}(\text{Exp}, \text{string}, \text{Env}) = \text{error}(\text{"Expression should be of type string"}, \\
& \quad \quad [\text{localized}("\text{Expression}", \text{get-location}(\text{Exp}))]) \\
& \quad \text{in all other cases: return an error}
\end{align*}
\]

Pico-typechecking (13)

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

- \texttt{PROGRAM}: the actual sort of the input program
- \texttt{Program}: the actual Pico program (a variable)

\[
\begin{align*}
\text{start}(\text{PROGRAM}, \text{Program}) = \\
& \quad \text{start}([\text{summary} \text{summary("pico-check"}, \\
& \quad \quad \text{get-file-name(get-location(\text{Program})),} \\
& \quad \quad \text{tce(\text{Program}))])}
\end{align*}
\]

The result is of the sort \texttt{Summary} and is obtained by applying the typechecking function \texttt{tce} to the input program and including it in a summary.

Typechecking an erroneous program

\[
\begin{align*}
tcp( \\
& \quad \text{begin} \\
& \quad \text{declare} \\
& \quad \quad x : \text{natural}; \\
& \quad \quad x := \text{"abc"} \\
& \quad \text{end}
\end{align*}
\]

The term

reduces to

\[
\begin{align*}
& \quad \text{summary("pico-check"}, \\
& \quad \quad \text{\"home\}/\text{paulk/pico.trm"}, \\
& \quad \quad [\text{error("Expression should be of type natural"}, \\
& \quad \quad \quad [\text{localized}("\text{Expression}", \\
& \quad \quad \quad \quad \text{area-in-file("\text{home\}/\text{paulk/pico.trm"}, \\
& \quad \quad \quad \quad \text{area}(4, 4, 12, 38, 5)])]
\end{align*}
\]

In the Meta-Environment
Intermezzo: equations (1)

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

\[ [B1] \text{Bool} = \text{true} \& \text{Bool} \]

Right-hand side may not contain uninstantiated variables:

\[ [B1] \text{true} \& \text{Bool}1 = \text{Bool}2 \]

Intermezzo: equations (2)

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

\[ [B1] \text{true} \& \text{Bool} = \text{Bool} \]
\[ [B2] \text{true} \& \text{false} = \text{false} \]

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

\[ [B1] \text{true} \& \text{Bool} = \text{Bool} \]
\[ \text{[default-B1]} \]
\[ \text{Bool}1 \& \text{Bool}2 = \text{Bool1} \]

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: “!e”; no uninstantiated variables
  - match: “=!e”; rhs may not contain uninstantiated variables, lhs may contain new variables,
  - and not-match: “!/=e”; 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**
  - Step 5: 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”.
Value-environments (1)

module languages/pico/run/Value-environments
imports languages/pico/syntax/Identifiers
languages/pico/run/Values
containers/Table

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
eval(PROGRAM) -> VENV

eval(DECLS) -> VENV

eval(IDS(ID-TYPE ".*") -> VENV

eval(STMT ".*", VENV) -> VENV

eval(STMT, VENV) -> VENV

eval(EXPR, VENV) -> VALUE

Evaluate a program
Evaluate declarations
Evaluate statements
Evaluate expression

Integers and Strings can occur as values during execution

nil-value denotes error values
Pico-evaluator (2)

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
import basic/Comments
context-free start-symbols
VALUE-ENV PROGRAM

variables
"Decl*"[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 ")
"String*[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) = eva(Series, eVd(Declsl)

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

Pico-evaluator (5)

[Ev2] eVd(declare Id-type*,) = evlts(Id-type*)

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

[Ev3b] eVlts(Id-string, Id-type*) = store(eVlts(Id-type*),Id,""

[Ev3c] eVlts() = []

Initialize a natural variable

Initialize a string variable

Create a new table for the empty list of declarations
Pico-evaluator (6)

Evaluate first statement

\[ \text{Ev4a} \quad \text{Venv'} = \text{evs(Stat, Venv)}, \]
\[ \text{Venv'}' = \text{evs(Stat', Venv')}, \]
\[ \text{evs(Stat ; Stat', Venv) = Venv'} \]

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

Evaluate an empty sequence of statements

Pico-evaluator (7)

Evaluate assignment statement

\[ \text{Ev5a} \quad \text{evs(Id = Exp, Venv) = update(Id, evs(Exp, Venv), Venv)} \]

Evaluate Rhs

Update variable with value of Rhs

Pico-evaluator (8)

"true" case

\[ \text{Ev5b} \quad \text{evs(Exp, Venv) = 0} \]
\[ \text{evs(if Exp then Series1 else Series2 fi, Venv) = evs(Series1, Venv)} \]

Evaluate if statement

Evaluate then branch

"false" case

\[ \text{Ev5c} \quad \text{evs(Exp, Venv) = 0} \]
\[ \text{evs(if Exp then Series1 else Series2 fi, Venv) = evs(Series2, Venv)} \]

Evaluate else branch

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

Pico-evaluator (9)

"false" case: while ends

\[ \text{Ev6d} \quad \text{evs(Exp, Venv) = 0} \]
\[ \text{evs(while Exp do Series od, Venv) = Venv} \]

Evaluate while statement

"true" case: while continues

\[ \text{Ev6e} \quad \text{evs(Exp, Venv) = 0}, \]
\[ \text{Venv' = evs(Series, Venv)}, \]
\[ \text{evs(while Exp do Series od, Venv') = evs(while Exp do Series od, Venv)} \]

Evaluate body once

Evaluate while statement in updated environment
Pico-evaluator (10)

A variable evaluates to its current value in the environment

\[
\begin{align*}
\text{[Ev6a]} & \quad \text{eve(Id, Venv) = lookup(Venv, Id)} \\
\text{[Ev6b]} & \quad \text{eve(Nat-con, Venv) = Nat-con} \\
\text{[Ev6c]} & \quad \text{eve(Str-con, Venv) = Str-con}
\end{align*}
\]

Constants evaluate to themselves

Pico-evaluator (11)

Evaluate addition

Evaluate left operand

Evaluate right operand

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

\[
\begin{align*}
\text{[Ev6d]} & \quad \text{Nat1 = eve(Exp1, Venv).} \\
\text{Nat2 = eve(Exp2, Venv).} \\
\text{eve(Exp1 \times Exp2, Venv) = Nat1 \times Nat2}
\end{align*}
\]

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

Pico-evaluator (12)

Evaluate - and |

Cutoff subtraction for naturals, e.g. \(3 \div 4 = 0\) We stay inside naturals

\[
\begin{align*}
\text{[Ev6e]} & \quad \text{Nat1 = eve(Exp1, Venv),} \\
\text{Nat2 = eve(Exp2, Venv).} \\
\text{eve(Exp1 - Exp2, Venv) = Nat1 - Nat2}
\end{align*}
\]

\[
\begin{align*}
\text{[Ev6f]} & \quad \text{Str1 = eve(Exp1, Venv),} \\
\text{Str2 = eve(Exp2, Venv).} \\
\text{eve(Exp1 || Exp2, Venv) = Str1 || Str2}
\end{align*}
\]

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

All other cases evaluate to nil-value

Evaluating the factorial program

\[
\begin{align*}
\text{evp} & \quad \text{begin declare input : natural,} \\
& \quad \text{output : natural,} \\
& \quad \text{repr : natural;} \\
& \quad \text{input = 14;} \\
& \quad \text{output = 1;} \\
& \quad \text{while input > 1 do} \\
& \quad \text{repr = input;} \\
& \quad \text{while repr > 1 do} \\
& \quad \text{output = output \times repr;} \\
& \quad \text{repr = repr - 1 od;} \\
& \quad \text{input = input - 1 od;} \\
& \quad \text{end}
\end{align*}
\]

The term \(\text{evp(14)}\) reduces to

\[
\begin{align*}
\text{[input,1],} \\
\langle\text{repr,1},} \\
\text{output,87178291200\rangle,} \\
\text{rep,43589145600\rangle}
\end{align*}
\]
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
  - Step 5: 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 (9)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Tr6a]</td>
<td><code>tre(Nat-con) = push Nat-con</code></td>
</tr>
<tr>
<td>[Tr6b]</td>
<td><code>tre(Str-con) = push Str-con</code></td>
</tr>
<tr>
<td>[Tr6c]</td>
<td><code>tre(Id) = rvalue Id</code></td>
</tr>
</tbody>
</table>

#### Translate constants

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Tr6d]</td>
<td><code>Instr*1 = tre(Expl1), Instr*2 = tre(Exp2)</code></td>
</tr>
<tr>
<td>[Tr6e]</td>
<td><code>tre(Expl1 + Expl2) = Instr*1: Instr*2: add</code></td>
</tr>
<tr>
<td>[Tr6f]</td>
<td>`tre(Expl</td>
</tr>
</tbody>
</table>

#### Translate variable

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tre(Id)</code></td>
<td><code>rvalue Id</code></td>
</tr>
</tbody>
</table>

#### Translate +, - and |

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tre(Id)</code></td>
<td><code>rvalue Id</code></td>
</tr>
</tbody>
</table>

### 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

### Compiling the factorial program

The term `trp` reduces to

```plaintext
trp(\begin{align*}
&\text{begin declare input : natural, output : natural;} \\
&\text{  reperr : natural, rep : natural;} \\
&\text{  input : 14;} \\
&\text{  output : 1;} \\
&\text{  while input - 1 do} \\
&\text{     rep = output;} \\
&\text{     input = input - 1;} \\
&\text{     while rep = reperr - 1 do} \\
&\text{         output = output + rep;} \\
&\text{         rep = reperr - 1;} \\
&\text{     end} \\
&\text{  end}
\end{align*})
```

#### 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 5: define a fact extractor
Traversals Functions (1)

- Many functions have the characteristic that they traverse the tree \textit{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 \textit{recursion}

Traversals 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(\textit{accu})
  - accumulate a value during traversal
- Transformer: traversal(\textit{trafo})
  - perform local transformations
- Accumulating transformer: traversal(\textit{accu, trafo})
  - accumulate \textit{and} transform
Traversal Cube: visiting behaviour

Simple Trees

module Tree-synt
imports Naturals
exports
sorts TREE
context-free syntax
NAT  → TREE
f(TREE, TREE) → TREE
g(TREE, TREE) → TREE
h(TREE, TREE) → TREE
doctors simple trees containing numbers as leaves and constructors f, g, or h

Count nodes (classical)

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

Example

Left-most innermost reduction:
[2] cnt(f(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**

Example: **accu, bottom-up, continue**

```
cnt( f( g(7, 8), 9), 0 )
  cnt( f( 1, 0) )
    cnt( 1 )
      7
    cnt( 9 )
      8
  cnt( 9 )
    9
```

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

Example: **accu, bottom-up, continue**

```
cnt( f( g(7, 8), 9), 0 )
  cnt( f( 1, 0) )
    cnt( 1 )
      7
    cnt( 9 )
      8
  cnt( 9 )
    9
```

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

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

Increment leaves

A bottom-up transformer that continues after each matching node

Leaf \( N \)

\[
\text{inc(} f( g(1,2), 3), \text{g}(g(4,5), 6)) \rightarrow f( g(2,3), 4), \text{g}(g(0,6), 7))
\]

\( \text{is replaced by N+1} \)

Example

\( \text{trafo.bottom-up.continue} \)

\[
\text{inc(} f( g(7, 8), 9)) \rightarrow f( g(8, 9), 10)
\]

\( \text{[1]} \text{ inc(T, N) = N + 1} \)

Increment leaves with explicit amount

A bottom-up transformer that continues after each matching node

Leaf \( N1 \)

\[
\text{inc(} f( g(1,2), 3), \text{g}(g(4,5), 6)), 7) \rightarrow f( g(8, 9), 10), \text{g}(g(11,12), 13))
\]

\( \text{Amount N2} \)

\( \text{Replace N1 by N1+N2} \)

Amount
**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
import Tree-syntax
context-free syntax
i(TREE, TREE) -> TREE
drepl(TREE) -> TREE (traversal(trafo,bottom-up,break))
equations
1) drepl(g(T1, T2)) = i(T1, T2)

Only the deepest occurrences of g are replaced
```

A bottom-up transformer that stops after first matching node

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

---

**Example**

```
drepl( g( g(7,8), 9))
g( i(7,8), 9)
```

A bottom-up transformer that stops after first matching node

```
drepl( g(7,8), 9))
```

Only the deepest occurrences of g are replaced

```
drepl(7) → drepl(8)
```

```
drepl(9)
```

```
g( i(7,8), 9)
```

---

**Example**

```
inc( f( g(7,8), 9), 5) → inc( f( g(12,13), 14)
```

[1] inc(N1, N2) = N1 + N2
### Shallow replacement

**Module**

```haskell
define module Tree-srepl
   imports Tree-syntax
   exports
      context-free syntax
         {Tree, Tree} -> Tree
         srepl(Tree) -> Tree (traversal(traf, top-down, break))

   equations
      srepl(g(T1, T2)) = i(T1, T2)
```

**Example**

A top-down transformer that stops after first matching node

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

Only the outermost occurrences of `g` are replaced

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

### Full replacement

**Module**

```haskell
define module Tree-frepl
   imports Tree-syntax
   exports
      context-free syntax
         {Tree, Tree} -> Tree
         frepl(Tree) -> Tree (traversal(traf, top-down, continue))

   equations
      frepl(g(T1, T2)) = i(T1, T2)
```

**Example**

A top-down transformer that continues after each matching node

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

All occurrences of `g` are replaced

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

---

*Note: Diagrams showing transformations of tree structures.*
Example

\[ \text{frepl}(g(g(7, 8), 9)) \]

\[ \text{frepl}(7) \rightarrow \text{frepl}(9) \]

\[ \text{frepl}(g(T1, T2)) = i(T1, T2) \]

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):

\[
\begin{align*}
\text{IF expr THEN} \\
\text{IF expr THEN} \\
\text{stats} \\
\text{ELSE} \\
\text{stats}
\end{align*}
\]

A funny Pico typechecker

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

---

\[ \text{frepl}(g(7, 8), 9)) \]

\[ \text{frepl}(9) \rightarrow \text{frepl}(9) \]

\[ \text{frepl}(g(T1, T2)) = i(T1, T2) \]

\[
\begin{align*}
\text{IF expr THEN} \\
\text{IF expr THEN} \\
\text{stats} \\
\text{ELSE} \\
\text{stats}
\end{align*}
\]

\[
\begin{align*}
\text{module End-If-Trafo} \\
\text{imports Cobol} \\
\text{exports} \\
\text{context-free syntax} \\
\text{addEndIf(Program)} = \text{Program} (\text{traversal}(\text{trafo}, \text{continue, top-down})) \\
\text{variables} \\
\text{"Stats"}'(0-9)' \rightarrow \text{StatsOptIfNotClosed} \\
\text{"Expr"}'(0-9)' \rightarrow \text{L-exp} \\
\text{"OptThen"}'(0-9)' \rightarrow \text{OptThen} \\
\text{equations} \\
\text{[1] addEndIf(IF Expr OptThen Stats)} = \\
\text{IF Expr OptThen Stats END-IF} \\
\text{[2] addEndIf(IF Expr OptThen Stats1 ELSE Stats2)} = \\
\text{IF Expr OptThen Stats1 ELSE Stats2 END-IF}
\end{align*}
\]
Example

Yields after typechecking:

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

Erroneous statement leaves a residue

Pico-typecheck (1)

module Pico-typecheck
imports Pico-syntax
exports
context-free syntax
type(TYPE)
  replace(STATS, ID-TYPE) ⇒ STATS (traversal(traverse(bottom-up, break)));
  replace(EXPR, ID-TYPE) ⇒ EXP (traversal(traverse(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)

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

equations
[0] begin declare Id-type, Dec*; Stat* end ⇒
  begin declare Dec*; 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)

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
Traversals 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**
  - *Step 5: 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

```python
module languages/pico/extraction/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/PostInfo[STATEMENT]
imports utilities/PostInfo[EXP]

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

declare the two extraction functions

Fact extractor

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

```python
countStatements(PROGRAM, RStore)  -> RStore
(traversal(accu, bottom-up, continue))
countStatements(STATEMENT, RStore)  -> RStore
(traversal(accu, bottom-up, continue))
cflow(STATEMENT "$\star\star\star\star$")  -> 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

Fact extractor

variables
"Program" [0-9]* → PROGRAM
"Decals" [0-9]* → DECLS
"Stat" [0-9]* → STATEMENT
"Stat*" [0-9]* → (STATEMENT,"")
"Exp" [0-9]* → EXP
"Id" [0-9]* → PILO-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)

equations
[expr] Store1 := declare (Store, ControlFlow, rel[str,loc, str.loc]).
<Entry, Rel, Exit> = cflow(Stat)

expressions
[cfg-1] <Entry1, Rel1, Exit1> = cflow(Stat1).
<Entry2, Rel2, Exit2> = cflow(Stat2).

<cfg-2> cflow() = (1, 1, 1)

Histogram

Declare the variable StatementHistogram and apply countStatements

declarations
(hist) StatementHistogram (Program, Store) =

countStatements (Program, Store, StatementHistogram).

equations
[] countStatements Stat =

only declare the cases of interest and increment relevant counter

Cflow: series

Declare the variable ControlFlow and apply cflow

Cflow: while

Cflow: while
Cflow: if

```
<cfg 4>
<Entry1, Ral, Exit1> = cflow(Stat*1),
<Entry2, Ral2, Exit2> = cflow(Stat*2),
Control := <unparse-to-string(Exp), get-location(Exp)>

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

```

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