21. Technical Space TreeWare
Simplification and Attribute Analysis on Trees

Prof. Dr. rer. nat. Uwe Aßmann

Institut für Software- und Multimediatechnik

Lehrstuhl Softwaretechnologie

Fakultät für Informatik

Technische Universität Dresden

http://st.inf.tu-dresden.de/teaching/most

Version 17-0.4, 28.10.17

1) Tree Simplification and Tree Rewriting
2) Analysis on Trees
    1) Metric Interpretation
    2) Attribute Analysis
3) Attribute Grammars for Interpreters on Syntax Trees

DRESDEN concept
Exzellenz aus Wissenschaft und Kultur

Obligatory Literature

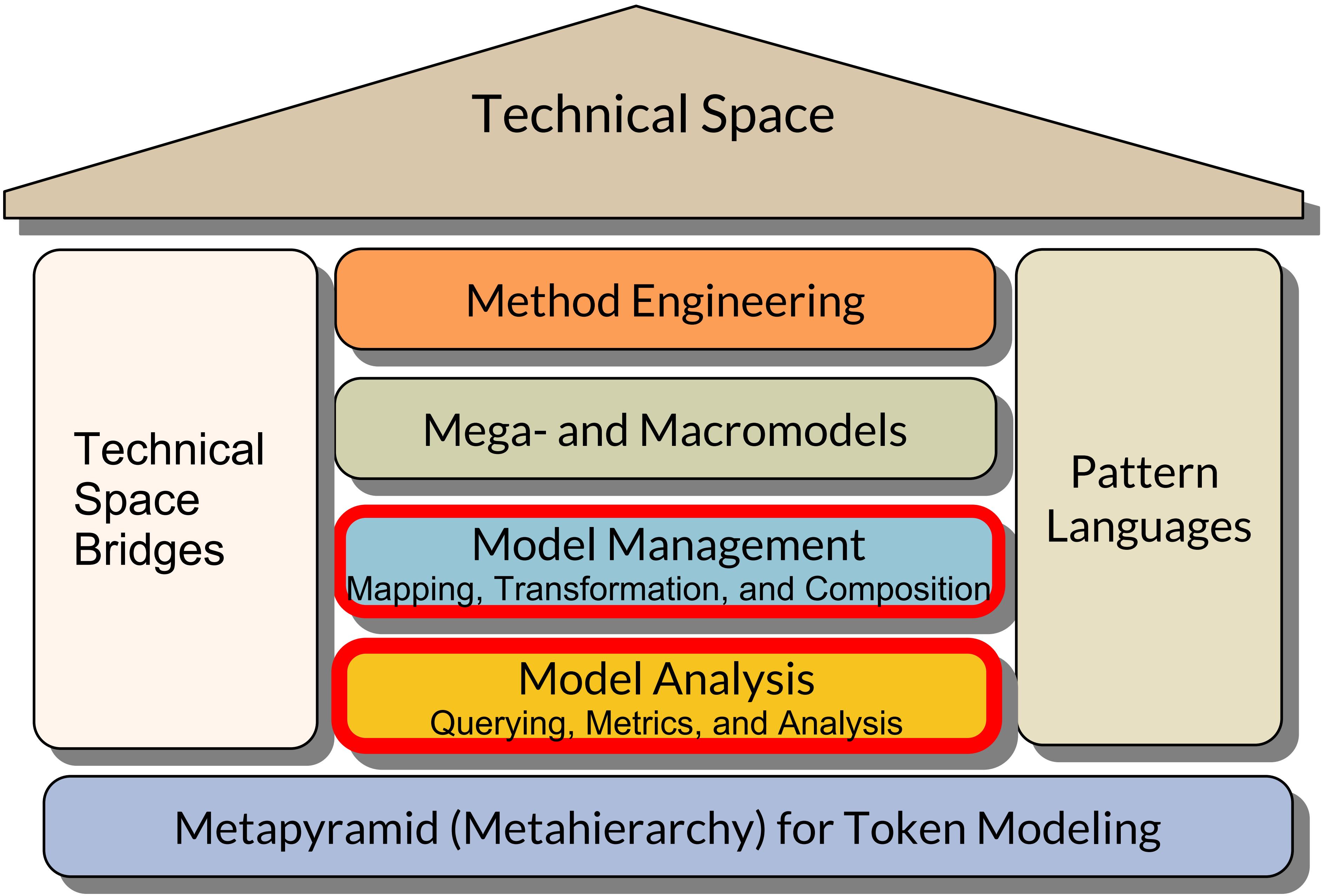
- List of analysis tools
  - http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis
- Paakki, Jukka. 1995. „Attribute grammar paradigms—a high-level methodology in language implementation“. ACM Comput. Surv. 27 (2) (Juni): 196–255.
- [KSV09] Lennart C. L. Kats, Anthony M. Sloane, Eelco Visser. Decorated Attribute Grammars. Attribute Evaluation Meets Strategic Programming (Extended Technical Report). Report TUD-SERG-2008-038a, Delft University
- [SKV09] Anthony M. Sloane, Lennart C. L. Kats, Eelco Visser. A Pure Object-Oriented Embedding of Attribute Grammars. Report TUD-SERG-2009-004, Delft University
- [LLL] Rüdiger Lincke, Jonas Lundberg and Welf Löwe. Comparing Software Metrics Tools

Other Literature on Attribute Grammars

- Knuth, D. E. 1968. „Semantics of context-free languages". Theory of Computing Systems 2 (2): 127–145.
- Hedin, Görel. 2000. „Reference Attributed Grammars". Informatica (Slovenia) 24 (3): 301–317.
- Boyland, John T. 2005. „Remote attribute grammars". Journal of the ACM 52 (4) (Juli): 627–687.
- Bürger, Christoff, Sven Karol, Christian Wende, und Uwe Aßmann. 2021. „Reference Attribute Grammars for Metamodel Semantics". In Software Language Engineering, LNCS 6563:22–41.

- Examples on: www.jastemf.org

Q10: The House of a Technical Space



Glossary for Automated Rewriting on Strings, Terms and Graphs

- **Rewrite rule:** rule (left, right hand side) to match left-hand side in the graph and to transform it to the right-hand side
- **Rewrite system (RS):** set of graph rewrite rules
- **Start data (axiom):** input data to rewriting process
- **Rewrite problem:** a rewrite system applied to a start data
- **Manipulated data (host data):** data which is rewritten in rewrite problem
- **Redex (reducible expression):** application place of a rule in the manipulated data
- **Rule mapping:** the mapping of a rule to a redex
- **Normal form:** result data of rewriting; manipulated data without further redex
- **Derivation:** a sequence of rewrite steps on the manipulated graph, starting from the start data and ending in the normal form
- **Unique normal form:** unique result of a rewrite system, applied to one start data
- **Deterministic RS:** rewrite system with one normal form
- **Terminating RS:** rewrite system that stops after finite number of rewrites
- **Confluent RS:** two derivations always can be commuted, resp. joined together to one result
- **Strong confluent RS:** all pairs of rewrite steps can be commuted
- **Convergent RS:** terminating deterministic rewrite system that always yields unique results (equivalent to terminating and confluent)

21.1 Simplification - Rewritings with the Stratego Term Rewriting System

Term and Tree Rewrite Systems (Termersetzungssysteme, TRS)

- Rewrite Systems enable the specification of **transformative semantics (reductive semantics)**
  - They reduce a data structure to a normal form, i.e., “give it a semantics”
  - They apply rewrite rules until a fixpoint
- **Term rewrite systems (Termersetzungssysteme)** transform tree- or term data structures
  - Can be used to rewrite an (abstract) syntax tree (AST)
  - Based on RTG
  - If pattern is a unordered tree, we speak of **tree rewriting**
  - If pattern is a term (ordered tree), we speak of **term rewriting**
- Use:
  - **Identification** of tree patterns (pattern matching)
  - **Simplifications** such as peephole optimization, constant folding
  - **Normalisations**, such as expanding abbreviations
  - **Inlining** and **outlining** of functions

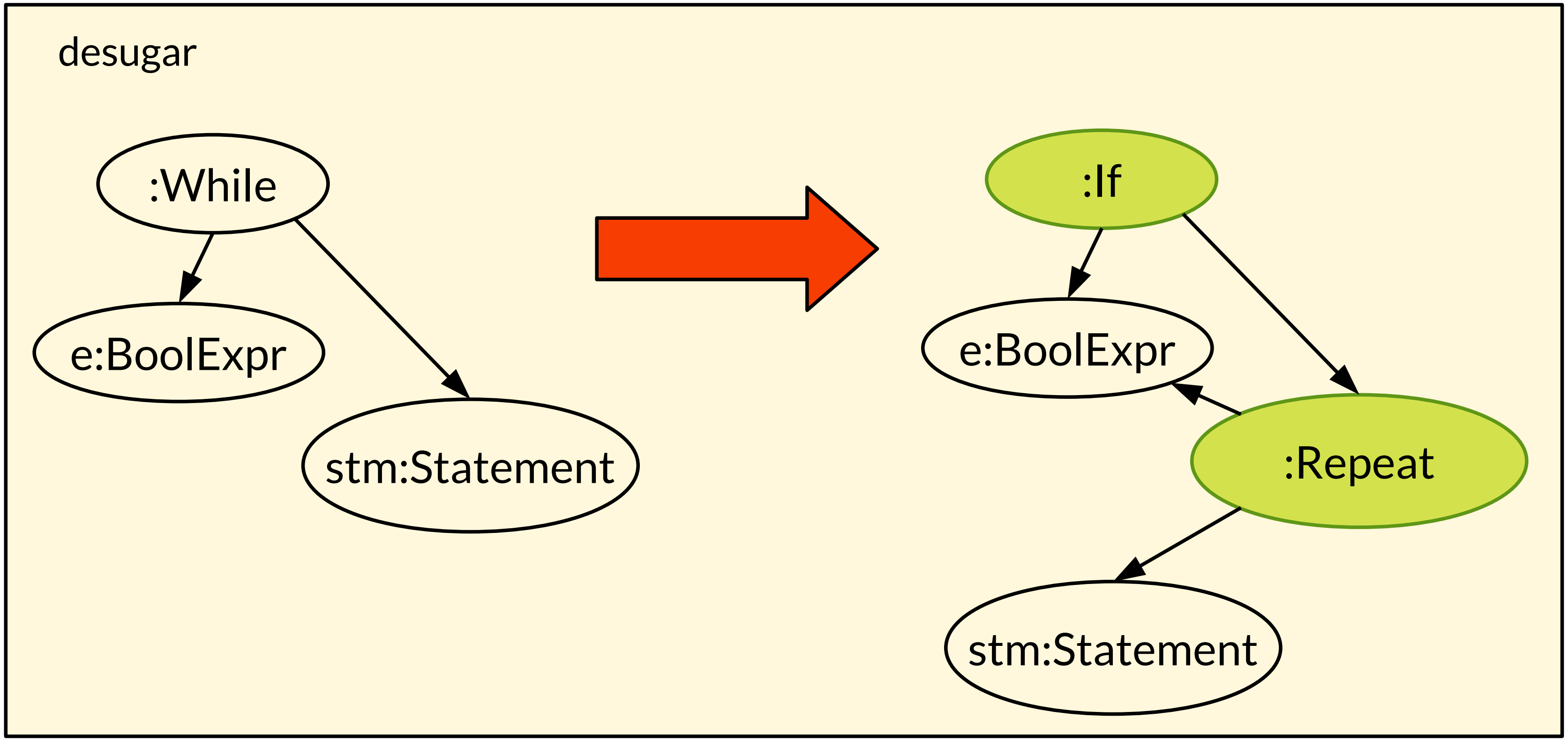
Stratego Term Rewrite System

https://strategoxt.org

- Syntax of a Stratego rewrite rule is based on RTG patterns

Name : RTG-Pattern „->“ Pattern

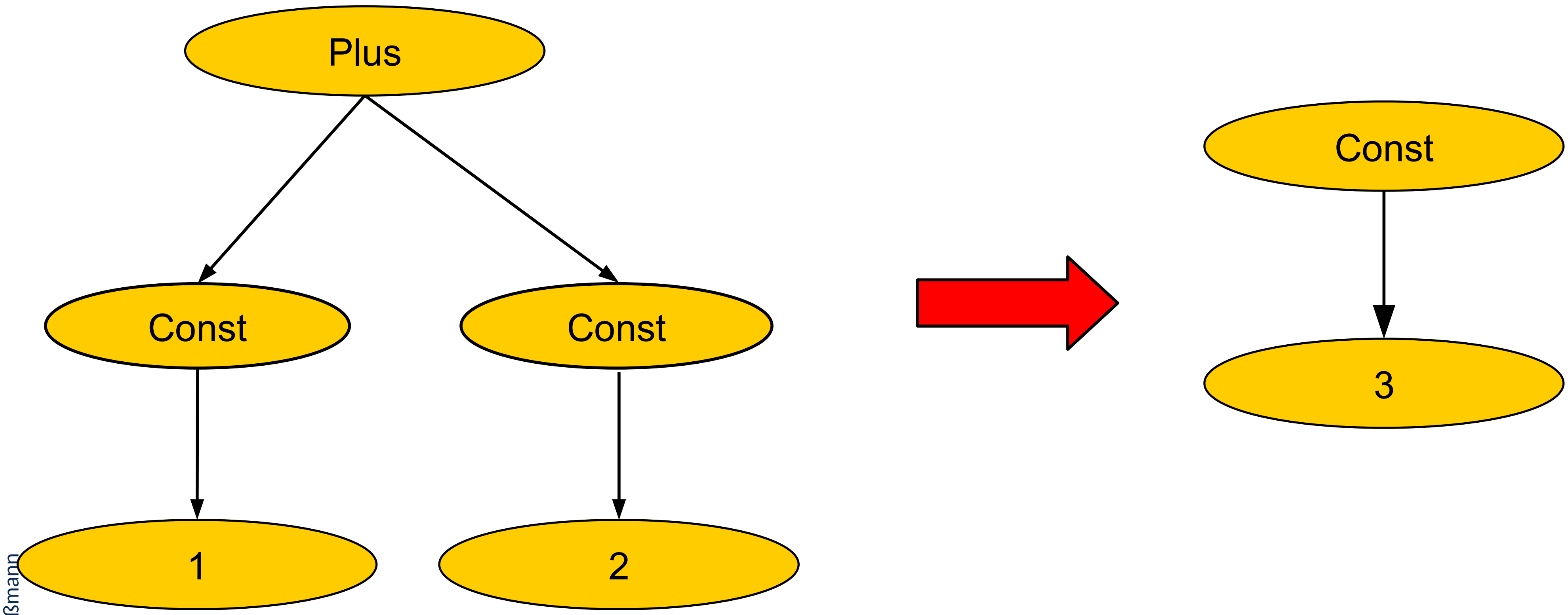
```
// Example: lowering all While statements to If statement with Repeats
desugar : While(e, stm) -> If(e, Repeat(stm, e))
```



Constant Folding as Subtractive TRS

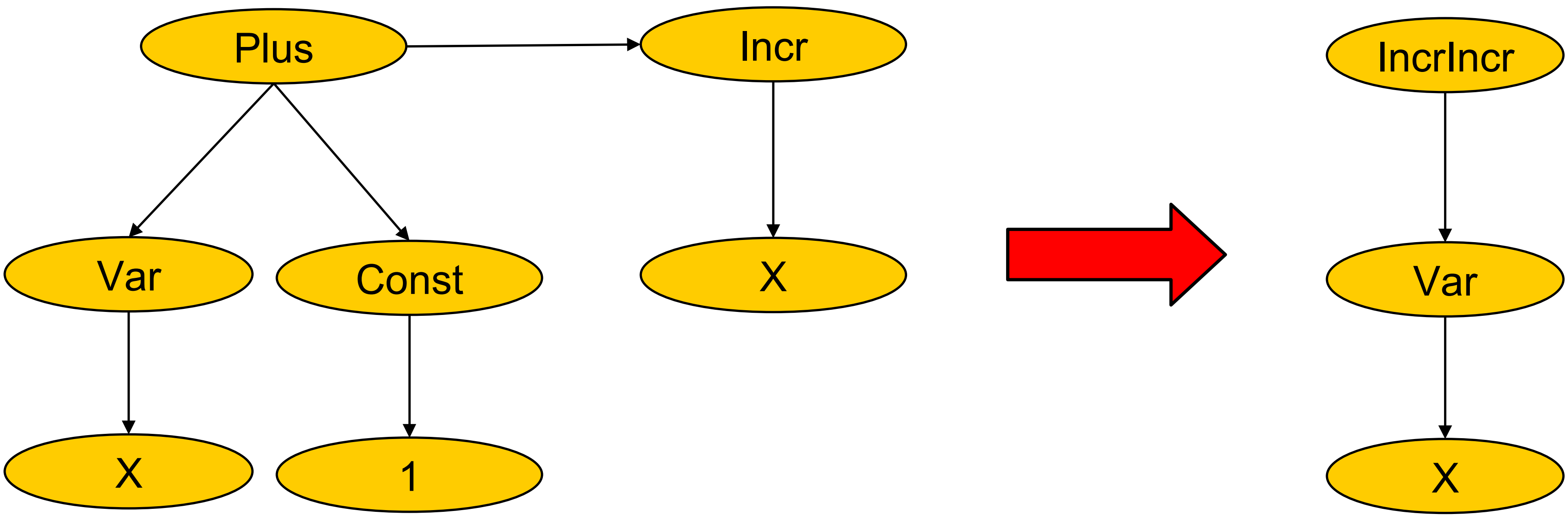
// Example: a special case of constant folding as RTG pattern rewriting

foldPlus : Plus(Const(1), Const(2)) -> Const(3)



Peephole Optimization as Subtractive TRS

// Example: a special case of peephole optimization
peepPlusIncr : next(Plus(Var(X),Const(1)), Incr(X)) -> IncrIncr(Var(X))



An Constant Folder Programmed in Stratego

- Constant folding in the TIL language
- http://hydra.nixos.org/build/23332578/download/1/manual/chunk-chapter/examples.html

```
// constant folding in Stratego
module til-eval
imports TIL
rules
compare(s) = if s then !True() else !False() end
  EvalAdd :    Add(Int(i), Int(j)) -> Int(<addS>(i,j))
  EvalAdd :    Add(String(i), String(j)) -> String(<conc-strings>(i,j))
  EvalSub :    Sub(Int(i), Int(j)) -> Int(<subtS>(i,j))
  EvalMul :    Mul(Int(i), Int(j)) -> Int(<mulS>(i,j))
  EvalDiv :    Div(Int(i), Int(j)) -> Int(<divS>(i,j))
  EvalMod :    Mod(Int(i), Int(j)) -> Int(<modS>(i,j))
  EvalLt :     Lt(Int(i), Int(j)) -> <compare(ltS)>(i,j)
  EvalGt :     Gt(Int(i), Int(j)) -> <compare(gtS)>(i,j)
  EvalLeq :    Leq(Int(i), Int(j)) -> <compare(leqS)>(i,j)
  EvalGeq :    Geq(Int(i), Int(j)) -> <compare(geqS)>(i,j)
  EvalEqu :    Equ(Int(i), Int(j)) -> <compare(eq)>(i,j)
  EvalOr :     Or(True(), e) -> True()
  EvalOr :     Or(False(), e) -> e
  EvalAnd :    And(True(), e) -> e
  EvalAnd :    And(False(), e) -> False()
  AddZero :    Add(e, Int("0")) -> e
  AddZero :    Add(Int("0"), e) -> e
  MulOne :     Mul(e, Int("1")) -> e
  MulOne :     Mul(Int("1"), e) -> e
  EvalS2I :    FunCall("string2int", [String(x)]) -> Int(x)
    where <string-to-int> x

  EvalI2S :    FunCall("int2string", [Int(i)]) -> String(i)
  EvalIf :     IfElse(False(), st1*, st2*) -> Block(st2*)
  EvalIf :     IfElse(True(), st1*, st2*) -> Block(st1*)
  EvalWhile : While(False(), st*) -> Block([])
```

http://hydra.nixos.org/build/23332578/download/1/manual/chunk-chapter/demo-rewriting.html#ref-til-sim-til-eval.str

Stratego System

- TRS compiled to C
  - Terms represented with the C-based Aterm library
- TRS compiled to Java
  - Rewriting the Java-based syntax trees of Eclipse JDT

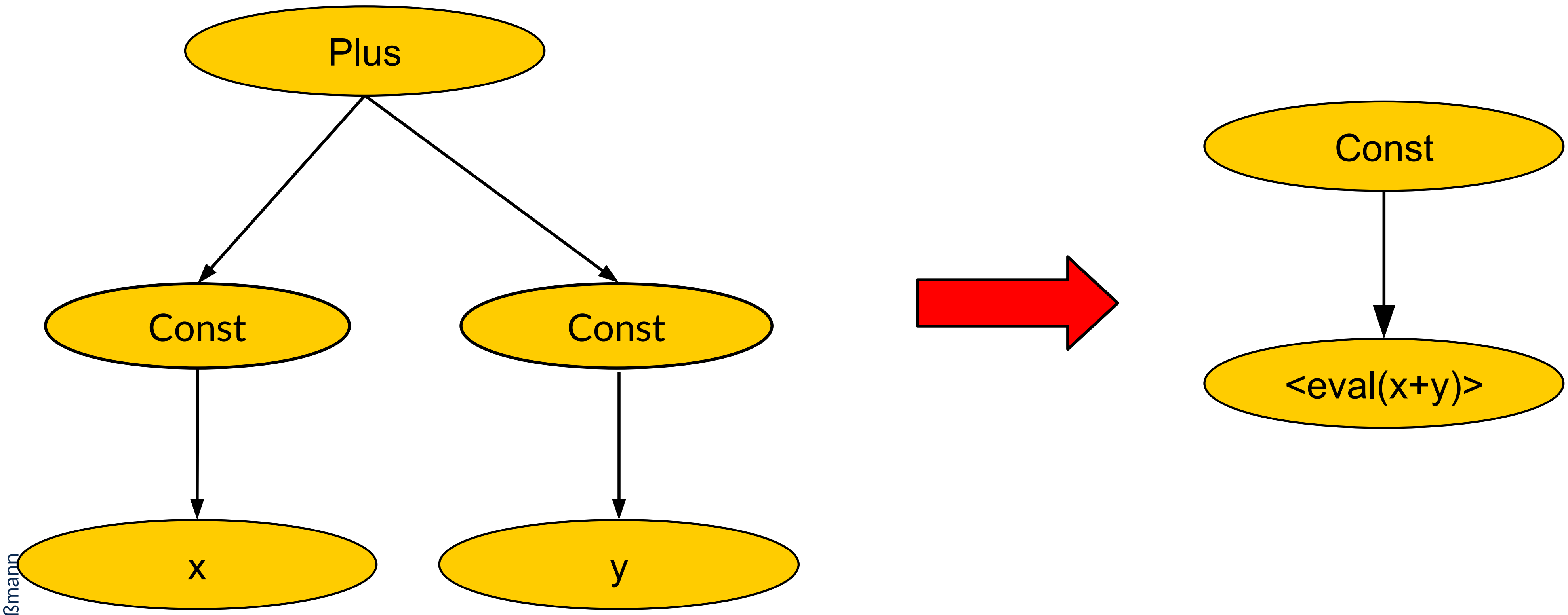
Rewriting Strategies

- ***Free (chaotic) rewriting***: all rules are applied until a fixpoint, the point of no change
- ***Confluent rewriting:*** when free rewriting ends up always with the same result (same normalform), the rewriting is confluent
- ***Strategies*** are second order rules that can steer the application of normal, first-order rules:
  - Top-down — topdown(r)
  - Bottom-up — bottumup(try(r))
  - Left-to-right depth-first / breadth-first
  - Right-to-left depth-first / breadth-first
  - Try a rule — try(r) = r <+ id
- Strategies are important for non-confluent rewriting problems
- Ex.: Das alternierende Suchen und Löschen von gefundenen Informationen.

Constant Folding with Strategic Rewriting

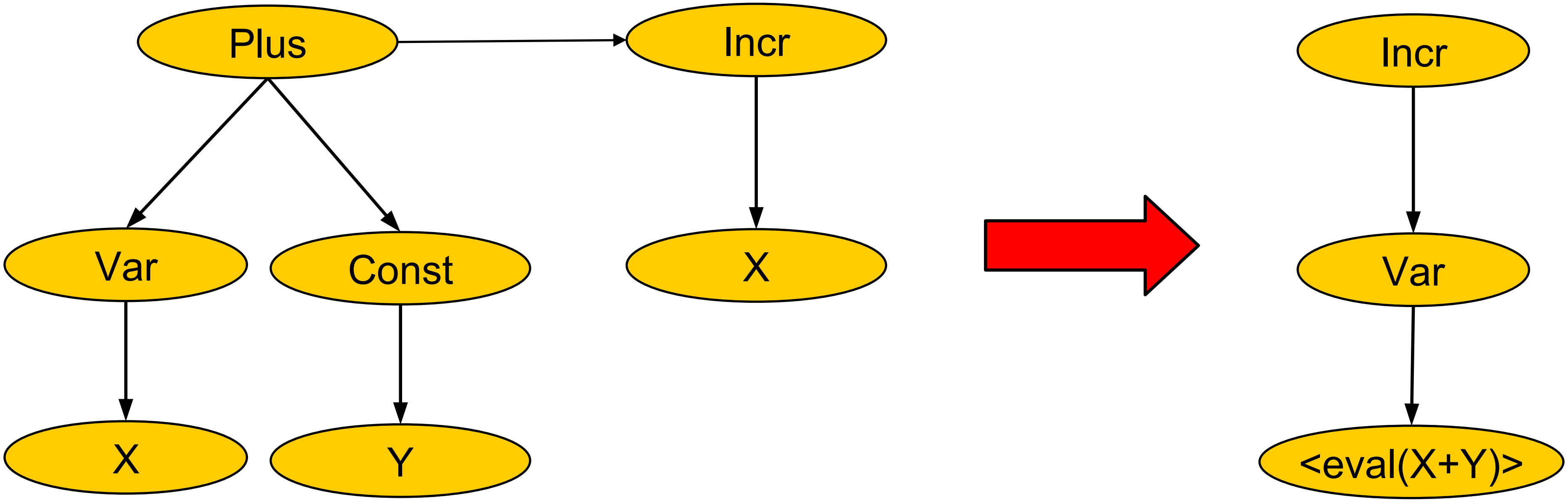
// Evaluation goes bottum-up of a (possibly big) term

foldPlusOperations: bottomup(Plus(Const(x), Const(y)) -> Const(<eval(x+y)>)



Peephole Optimization with Topdown Rewriting

// Example: a more general case of peephole optimization
peepConstants:
topdown(next(Plus(Var(X),Const(Y)), Incr(X))) -> Incr(Var(<eval(X+Y-1))



21.1.2 The TXL Tool

- <External slide set from TXL distribution: TXLintro.pdf>
  - http://www.txl.ca/docs/TXLintro.pdf
- Exercise: Download the TXL distribution (10.6) from
  - http://www.txl.ca/ndownload.html
- Investigate the TXL Pascal Grammar.
  - examples/analysis/tracing/Txl/Pascal.Grm
- Then look at the ptract.txl specification, which adds tracing statements to a Pascal program:
  - examples/analysis/tracing/Txl/ptrace.Txl
- Try to run it and see the result.

21.2 Analysis in an MDSD Tool

- **Analysis** in an MDSD tool requires queries, metrics, and deep analysis

The Hierarchy of Analyses

Logic queries (on graphs)

Attribute analysis on graphs

Attribute analysis on reducible graphs

Attribute analysis on link trees and XML trees

Attribute analysis on trees
(queries, metrics, attribute-based interpretations)

21.2.1 Metric Analysis in an MDSD Tool

- **Analysis** in an MDSD tool requires queries, metrics, and deep analysis
- Queries are done in a query language, see later

Metrics

[LLL]

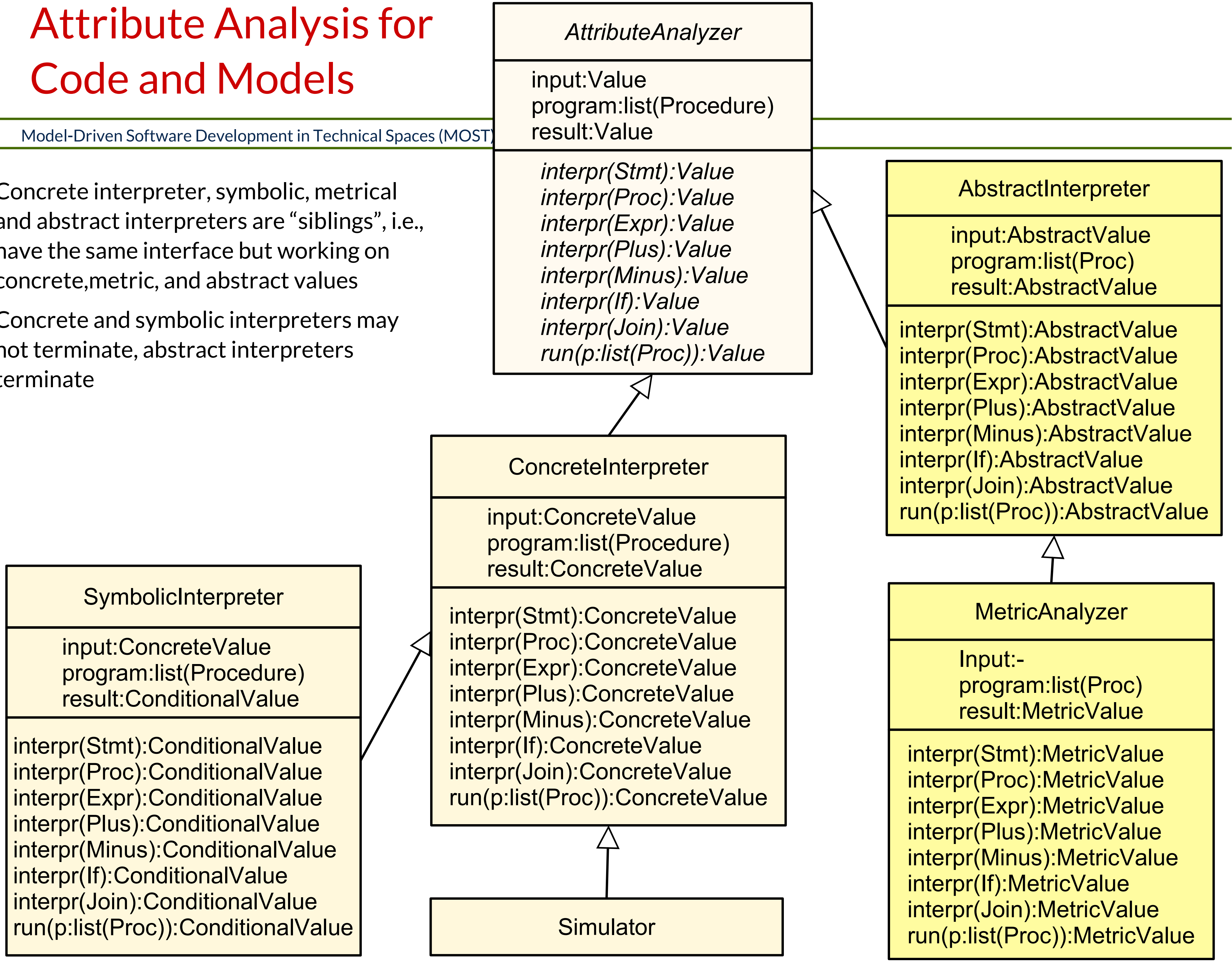
- **Coupling metrics** measure the coupling of two packages, classes or modules
  - CBO: “Coupling between object classes” counts links to other classes
  - RFC: “Response for a class” counts the number of methods called in response to a message to an object
- **Cohesion metrics** measure the cohesion of one package, class, or module
  - LCOM “Lack of cohesion of methods” in a unit
- **Inheritance metrics**
  - DIT: “Depth of inheritance tree”
  - BIT: “Breadth of inheritance tree”
- **Size and complexity metrics**
  - LOC: “lines of code” - quite weak metrics
  - EXC: “expression count” in a method
  - NOM “number of methods” in a class
  - WMC “weighted methods per class” with cyclomatic complexity

21.2.2 Attribute-based Analysis and Interpretation

- In ***attribute-based analysis*****,** the code or the model stays invariant, but ***attributes*** are evaluated on the code, based on *stencil functions (transfer functions, attribution functions)*
- A ***symbolic interpreter*** executes the program, but builds up a lookup table, under which conditions which values are produced. The table may be infinitely large.

Attribute Analysis for Code and Models

- Concrete interpreter, symbolic, metrical and abstract interpreters are "siblings", i.e., have the same interface but working on concrete,metric, and abstract values
- Concrete and symbolic interpreters may not terminate, abstract interpreters terminate



21.3 Attribute(d) Grammars for Interpretation, Simulation, Metric, and Abstract Interpretation

Attribute(d) Grammars (AG) for Interpreters on Piecelists and Trees

- An **attribute(d) grammar** is a regular tree grammar (RTG) in which all nonterminals are adorned with attributes, and every rule contains a set of **attribution functions (attribute equations, stencil functions)**
  - AG are declarative and partition the stencil function space with their tree nodes (tree-node-specific functions)
- Attributed grammars describe calculations on part lists (piece lists, Stücklisten)
  - Metrics on piecelists in information systems
  - Concrete interpretation, metric, and abstract interpretation on syntax trees of programs
- Instead of rewriting a tree for interpretation, AG compute their results as functions over attributes
  - Data-Driven Programming: the data structure is in primary focus
- AG are **stencil computation systems,** because they keep the syntax trees invariant, but compute with stencil functions attributes over them
  - The tree grammar computes the part list (phase 1)
  - The stencil functions compute the attributes (phase 2)
  - The rewrites rebuilt the part list (optional phase 3)

Interpretation of Programs with Attribute(d) Grammars (AG)

- An **attribute(d) grammar** describes an interpreter on a syntax tree (a hierarchical program representation)
  - The syntax tree is described by a Regular Tree Grammar (or string grammar)
  - The nodes of the program in the syntax tree are augmented with values, **attributes**. The resulting data structure is called **attributed syntax tree (AST)**
    - Graph representations are not possible in pure AGs
  - There is a set of **attribution functions (attribution rules, attribute equations)** which define **interpretation functions** on all nodes of the syntax tree
  - Usually, the rules are interpreted with recursion along the attributed syntax tree
- Because the underlying program representation is hierarchic, often
  - AG-based interpreters can be proven to terminate
  - can be compiled to code, instead of interpreted (pretty fast)

AG-based concrete interpreters can analyze syntax trees by concrete interpretation evaluating their attribution functions

The Pattern-Major Form of AG (Node-Based Form, Window Form)

- In the **pattern-major form (window form)** of an AG, the tree node patterns of the RTG used to describe the tree form the *windows* (the major groups of attribute definitions)
- Attribution Functions are written in a functional language. They take node attributes as parameters and results.
- A **stencil** is an assignment of an attribute by a attribution function.
    - In one window, many stencils may appear.

```
Interpretation evalArithmeticExpr(Tree      Tree)
 in pattern-major form {
  Attribute definitions of Root(st) {
          this.result := st.result;
          <println(„Result is %S“, this.result)>
  }
  Attribute definitions of Plus(st1,st2) {
          this.result := <st1.result + st2.result>
  }
  Attribute definitions of Minus(st1,st2) {
          this.result := <st1.result - st2.result>
  }
  Attribute definitions of Mult(st1,st2) {
          this.result := <st1.result * st2.result>
  }
  Attribute definitions of Div(st1,st2) {
          this.result := if (st2 == 0) then {
              <println(„Error, div by zero“)>;
              -999 }
            else <st1.result / st2.result>
  }
  Attribute definitions of  Leaf(value:Integer) {
          this.result := value
  }
}
```

Global Minima

[KSV09]

- Transforming a tree to a new tree with leaf nodes carrying the global minimum of all leaf nodes

```
Transformation repmin(Tree       Tree) in pattern-major form
{
  Attribute definitions of Root(st) {
            st.global-min := st.min
            this.min := st.min
            this.replace := Root(st.replace)
  }
  Attribute definitions of Pair(st1,st2) {
            st1.global-min := this.global-min
            st2.global-min := this.global-min
            this.min := <min(st1.min,st2.min)>
            this.replace := Pair(st1.replace,st2.replace)
  }
  Attribute definitions of  Leaf(value:Integer) {
            this.min := value
            this.replace := Leaf(this.global-min)
  }
}
```

Attribute-Major Format

- Attribution functions are re-sorted along attributes, while pattern matching is used inside the definition of the attribution function

```
Transformation repmin(Tree      Tree) in attribute-major form {
  Attribute definitions for min: {
     Root(st)                this.min := st.min
     Pair(st1,st2)           this.min := <min(st1.min,st2.min)>
     Leaf(value:Integer)     this.min := value
  }
  Attribute definitions for global-min: {
    Root(st)                 st.global-min := st.min
    Pair(st1,st2)            st1.global-min := this.global-min
                             st2.global-min := this.global-min
    Leaf(value:Integer)      st.global-min := st.min
  }
  Attribute definitions for replace { // tree-valued attribute
    Root(st)                 this.replace := Root(st.replace)
    Pair(st1,st2)            this.replace := Pair(st1.replace,st2.replace)
    Leaf(value:Integer)      this.replace := Leaf(this.global-min)
  }
}
```

Attribute Grammars (AG) can Specify Metric Analyzers

- An attribute grammar can describe a **metric analyzer**, if the values are from a domain of a software metrics
- Then, the set of attribution rules (attribute equations) define a software metrics interpretation functions on the syntax tree

AG-based abstract interpreters can analyze syntax trees by **metric interpretation**

Attribute Grammars (AG) Can Specify Abstract Interpreters

- An attribute grammar can describes an **abstract interpreter**, if the values are from an abstract domain (a system of equivalence classes)
  - e.g., from a set of types, a type system, interval ranges, etc.
  - Then, the set of attribution rules (attribute equations) define abstract interpretation functions computing on equivalence classes
- Example: **Type analysis and checking**
  - The analysis of expressions on their types (int, real, char, string, etc) and the check whether their types are compatible is an abstract interpretation
  - Finitely many types ((int, real, char, string, user types)
  - Inclusion and compatibility rules for types
    - Char < int < real
    - Range < int
    - Person < Object

AG-based abstract interpreters can analyze syntax trees by **abstract interpretation**

The End

- Explain the differences of a concrete interpreter, a metric analyzer, and an abstract interpreter
- What are the differences of an abstract interpreter and an attribute grammar?
- Why is a reference attribute grammar more expressive than a pure AG?
- What happens at a control-flow join during an interpretation?
- Why is *metric interpretation* important?
- Explain how RTG and AG are related
- Explain the difference of pattern-major and attribute-major form
- What is the difference of a functional program and an AG?
- Why is an abstract interpreter a functional program?