22. Deep Analysis in Treeware: Concrete and Abstract Interpretation on M2

How to find out about the semantics of a program or model

Prof. Dr. rer. nat. Uwe Aßmann

Institut für Software- und Multimediatechnik

Lehrstuhl Softwaretechnologie

Fakultät für Informatik

TU Dresden

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

Version 21-0.3, 06.12.21

1) Concrete Interpretation and Abstract Interpretation (AbI)
2) Interpreters on Syntax Trees
3) Laws of Abstract Interpretation
4) Strategy of Iteration (Visit Order)

Other Resources

- Selective reading:
  - Mats Rosendahl. Abstract Interpretation and Attribute Grammars. In: Deransart P., Jourdan M. (eds) Attribute Grammars and their Applications. Lecture Notes in Computer Science, vol 461. Springer, Berlin, Heidelberg
    - https://link.springer.com/chapter/10.1007/3-540-53101-7_11
  - Neil D. Jones and Flemming Nielson. 1995. Abstract interpretation: a semantics-based tool for program analysis. In Handbook of logic in computer science (vol. 4), S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum (Eds.). Oxford University Press, Oxford, UK 527-636.
    - http://dl.acm.org/citation.cfm?id=218637
  - Michael Schwartzbach's Tutorial on Program Analysis
    - http://lara.epfl.ch/dokuwiki/_media/sav08:schwartzbach.pdf
- Patrick Cousot's web site on A.I. http://www.di.ens.fr/~cousot/AI/
- [CC92] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In U. Kastens and P. Pfahler, editors, Proceedings of the International Conference on Compiler Construction (CC), volume 641 of Lecture Notes in Computer Science, pages 125-140, Heidelberg, October 1992. Springer.
- [Kam/Ullmann] John B. Kam and Jeffery D. Ullmann. Global data flow analysis and iterative algorithms. Journal of the ACM, 23:158-171, 1976.

Obligatory Literature

- David Schmidt. Tutorial Lectures on Abstract Interpretation. (Slide set 1.) International Winter School on Semantics and Applications, Montevideo, Uruguay, 21-31 July 2003.
  - http://santos.cis.ksu.edu/schmidt/Escuela03/home.html
- List of analysis tools
  - http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

- [LLL] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. 2008. Comparing software metrics tools. In Proceedings of the 2008 international symposium on Software testing and analysis (ISSTA '08). Association for Computing Machinery, New York, NY, USA, 131–142. DOI:https://doi.org/10.1145/1390630.1390648
- Béatrice Bouchou, Mirian Halfeld Ferrari Alves, Maria Adriana Vidigal de Lima. Attribute Grammar for XML Integrity Constraint Validation. DEXA (1) 2011: 94-109
  - https://link.springer.com/chapter/10.1007/978-3-642-23088-2_7

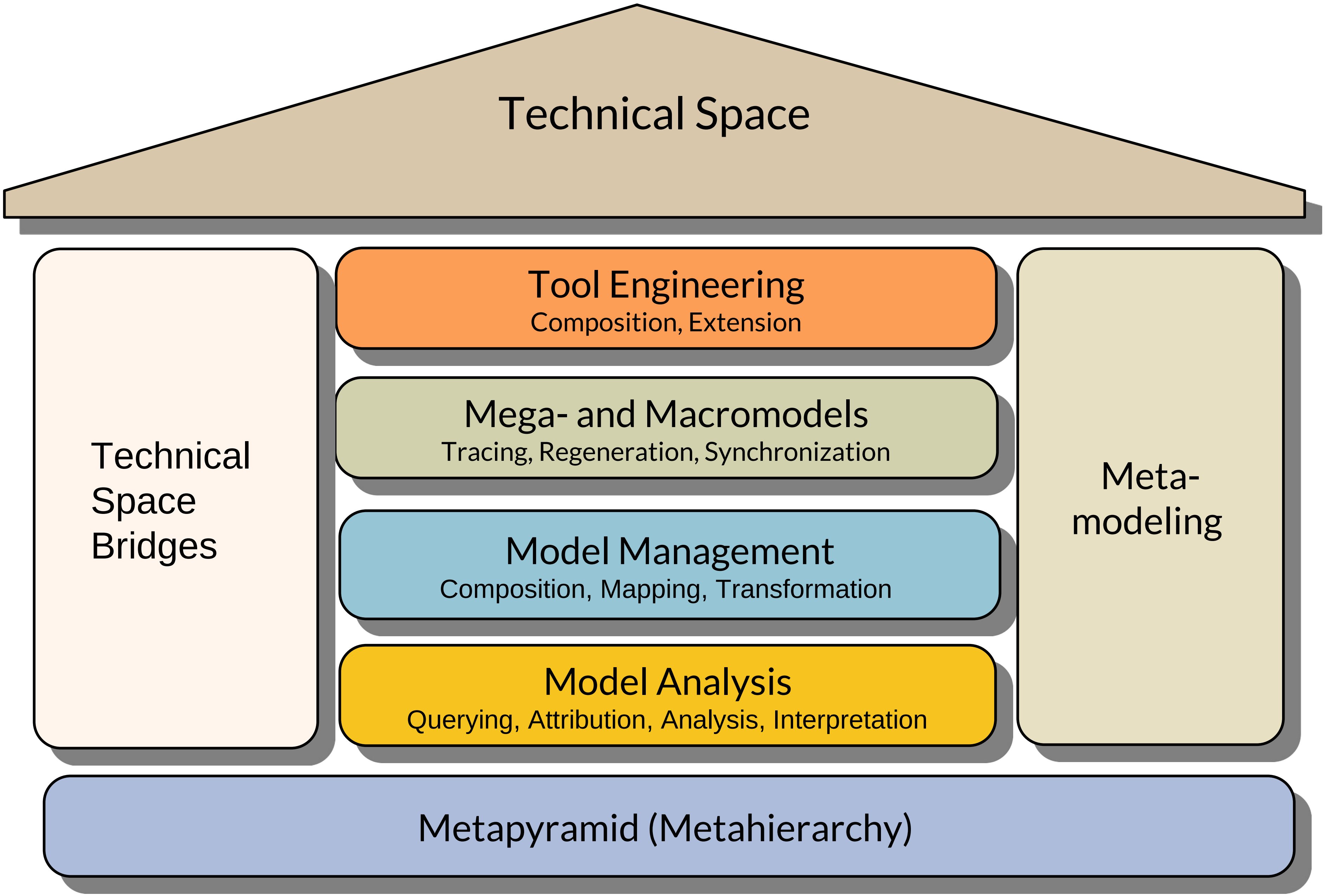
Other Resources

- Selective reading:
  - Neil D. Jones and Flemming Nielson. 1995. Abstract interpretation: a semantics-based tool for program analysis. In Handbook of logic in computer science (vol. 4), S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum (Eds.). Oxford University Press, Oxford, UK 527-636.
    - http://dl.acm.org/citation.cfm?id=218637
  - Michael Schwartzbach's Tutorial on Program Analysis
    - http://lara.epfl.ch/dokuwiki/_media/sav08:schwartzbach.pdf
- Patrick Cousot's web site on A.I. http://www.di.ens.fr/~cousot/AI/
- [CC92] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In U. Kastens and P. Pfahler, editors, Proceedings of the International Conference on Compiler Construction (CC), volume 641 of Lecture Notes in Computer Science, pages 125-140, Heidelberg, October 1992. Springer.
- [Kam/Ullmann] John B. Kam and Jeffery D. Ullmann. Global data flow analysis and iterative algorithms. Journal of the ACM, 23:158-171, 1976.

Literature on Attribute Grammars

- Knuth, D. E. 1968. „Semantics of context-free languages". Theory of Computing Systems 2 (2): 127–145.
- Paakki, Jukka. 1995. „Attribute grammar paradigms—a high-level methodology in language implementation". ACM Comput. Surv. 27 (2) (Juni): 196–255.
- 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. 2011. „Reference Attribute Grammars for Metamodel Semantics". In Software Language Engineering, LNCS 6563:22–41.

Q10: The House of a Technical Space



22.1 Interpretation and Abstract Interpretation (Ab.I.)

Two Forms of Program and Model Analysis

- **Flat Analysis** is the process of finding information about a program, e.g.,
  - finding a pattern in a syntax tree
  - computing attributions
  - computing metrics

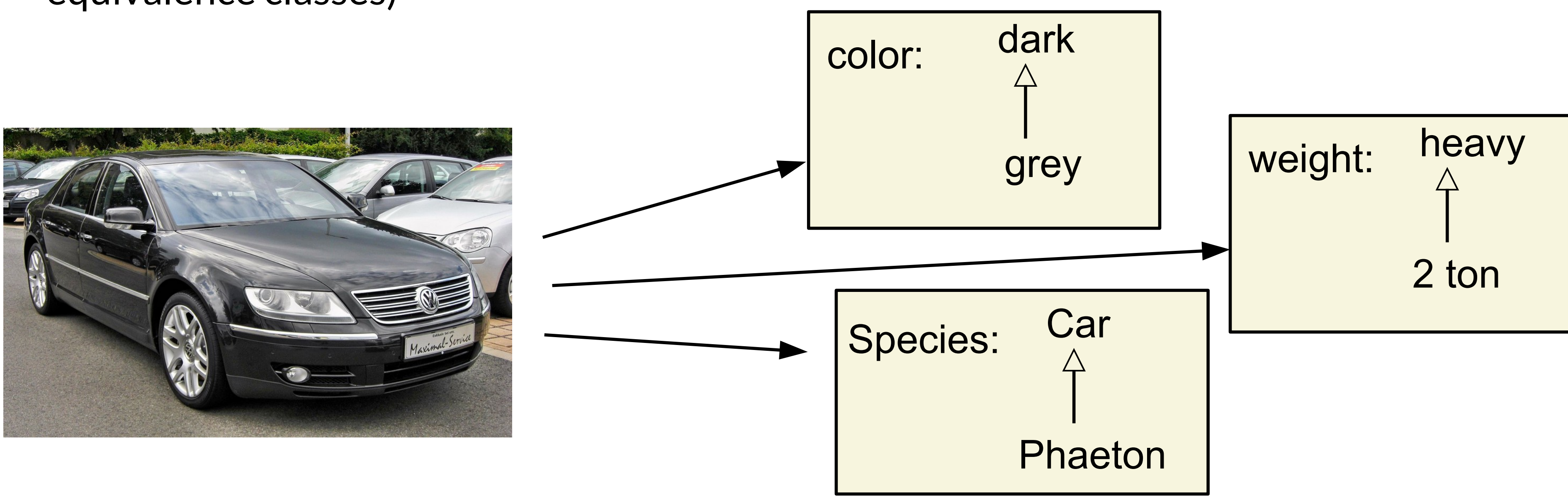
```
X := 10;
Y := if (B < 5) then X;
                    else X+1;
// Y = { X, X+1} // symbolic execution
// Y = { 10 \or|B 11 } // abstract interpretation
```

- **Deep Analysis** is the process of collecting information about *the value flow in* a program and storing it into attributes of the syntax tree so that it can be used for further analysis.
  - Deep analysis *interprets* the program (assigments, expressions) to find out about value computation
  - **Dynamic** interpretation
    - **Simulation** on a virtual machine, not hardware
  - **Static** interpretation
    - **Symbolic execution** (collecting semantics, all-possible-path interpretations)
    - **Abstract interpretation** (possible-value interpretations)

What is Abstraction?

Abstraction is the neglection of unnecessary detail.
(**Abstraktion** ist das Weglassen von unnötigen Details)

- A thing of the world can be abstracted differently
- This generates mappings from a concrete domain (D) to abstract domains (D#, equivalence classes)

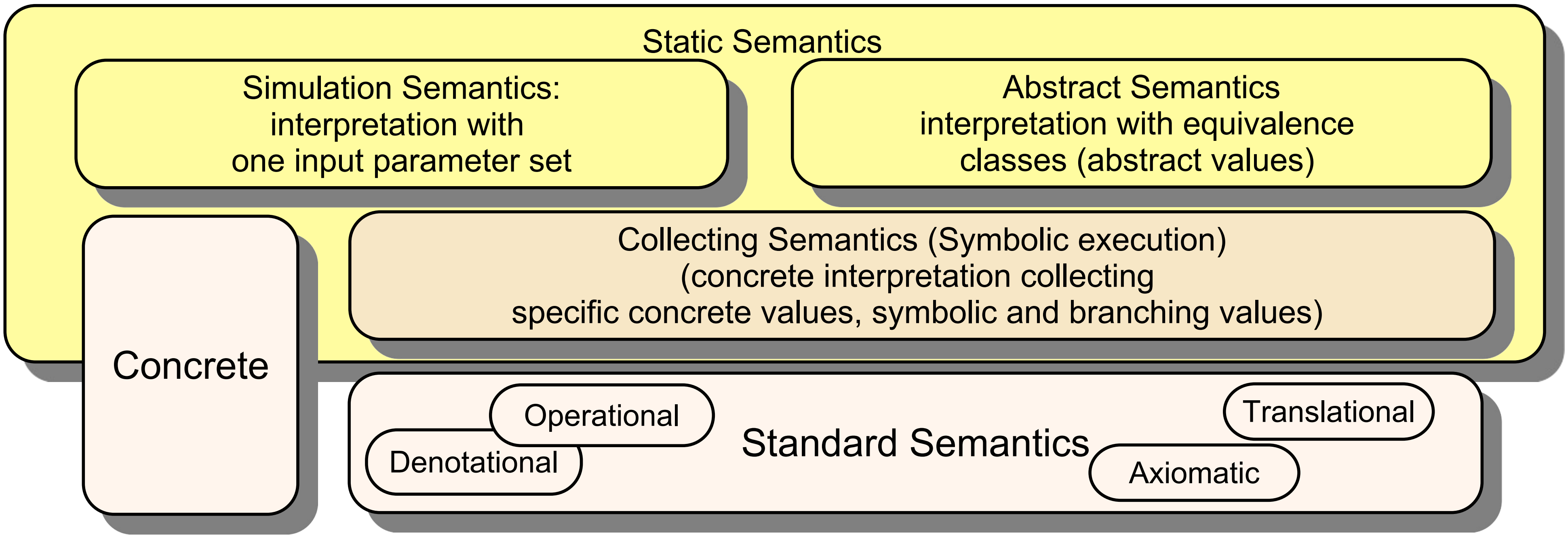


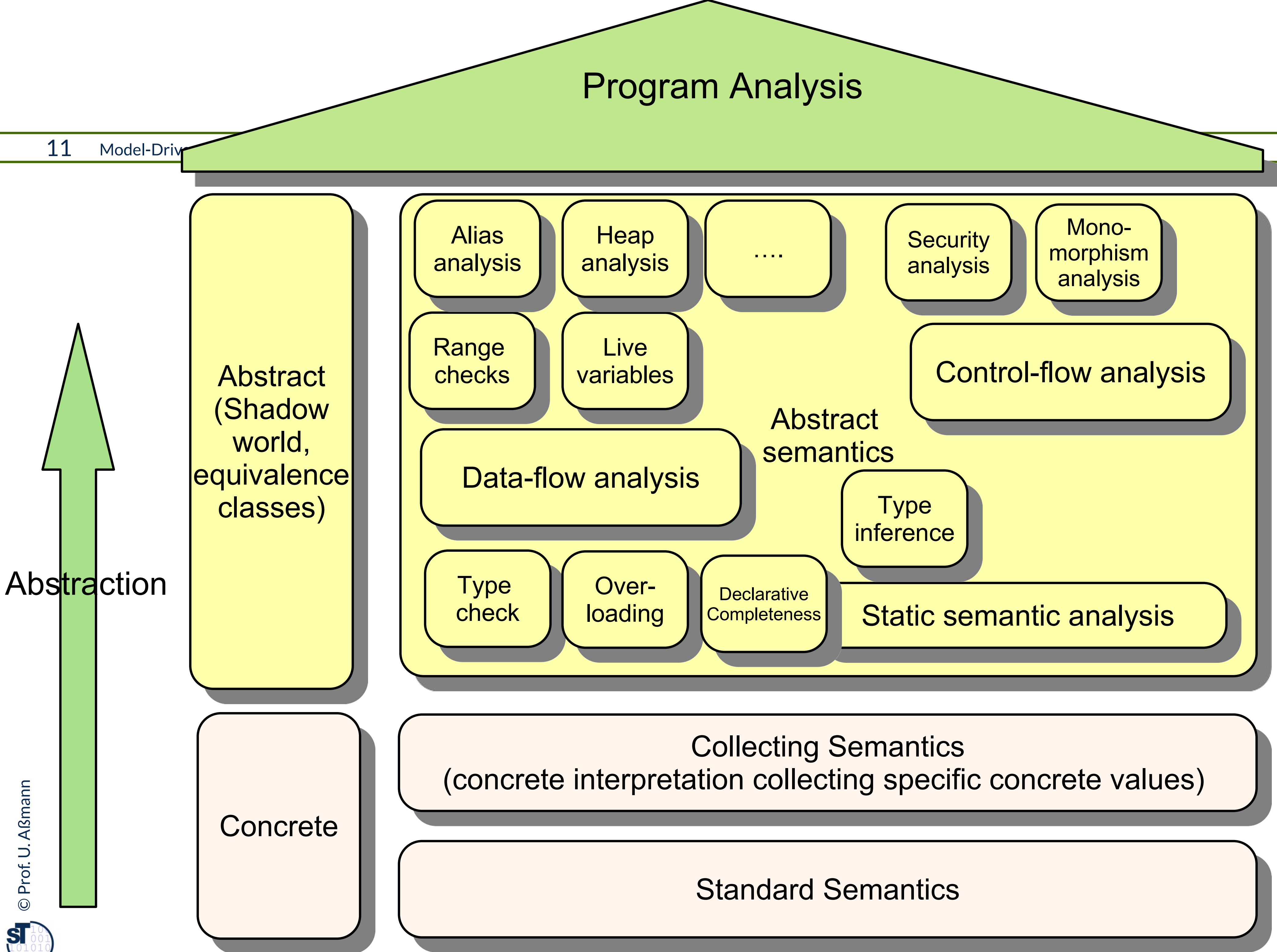
https://commons.wikimedia.org/wiki/File:VW_Phaeton_20090712_front.JPG#/media/Datei:VW_Phaeton_20090712_front.JPG, User M 93

Abstract interpretation is the computing with equivalence classes (abstract domains) instead of concrete numbers

Interpretation and Semantics of Programs

- Given a fixed set of input values, a program has a **concrete standard semantics (dynamic semantics)** based on concrete values
  - **Denotational semantics (result semantics):** The output values
  - **Operational semantics** (interpretative semantics): The set of traces of the execution by an interpreter, and the set of states in these execution traces
  - **Axiomatic semantics**: The set of all true predicates at each execution point
  - **Translational semantics (rewriting semantics):** A translation function (compiler) that returns a program in a lower-level language
- A **collecting semantics (symbolic execution)** selects a subset of interest from the standard semantics, in preparation of the abstract interpretation.
  - The values of the semantics stay concrete, but are replaced by symbols and terms over symbols.
- A **simulation** selects one specific input parameter set and executes the program with it
- An **abstract interpretation** interprets on the **abstract semantics**, an abstraction of the the collecting semantics



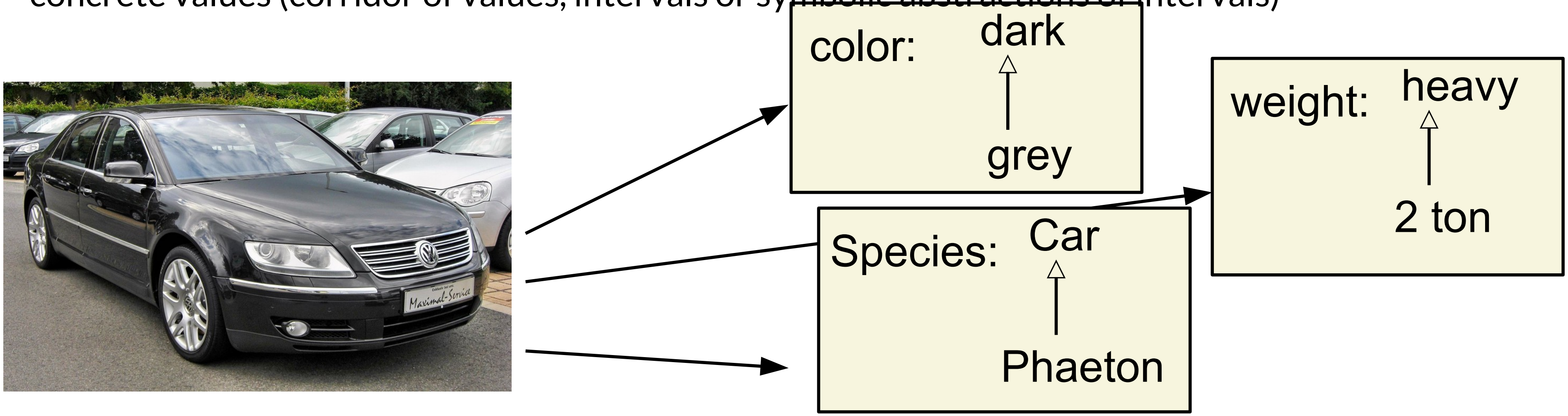

Program Analysis
Abstract (Shadow world, equivalence classes)
Alias analysis
Heap analysis
....
Security analysis
Mono-morphism analysis
Range checks
Live variables
Control-flow analysis
Abstract semantics
Data-flow analysis
Type inference
Type check
Over-loading
Declarative Completeness
Static semantic analysis
Abstraction
Concrete
Collecting Semantics
(concrete interpretation collecting specific concrete values)
Standard Semantics

What is an Interpreter?

- An **interpreter** executes a program (tree) on a set of input data and realizes an operational semantics
  - An interpreter is based on an operational semantics over state
  - For an object-oriented language, for all metaclasses of the language on M2, interpretation functions have to be given
- **The interpreter annotates every statement of a program syntax tree (ST, AST) or graph (SG, ASG) with attributes holing the values at every point**
- ==> the interpreter is an *attribute evaluator of the program*
- A **symbolic execution interpreter** interprets the program with symbolic values
  - The values are *or-ed* on branch of control flow
- A **simulator** interprets the program with one set of concrete input parameters. Often, a specific platform interpreter is used
- An **abstract interpreter (possible-value interpreter)** is the twin of an interpreter, interpreting on abstract values (equivalence classes, “shadows” in the shadow world)
  - **The abstract interpreter annotates every statement of a program graph (AST, ASG) with attributes holing the possible values, i.e., abstract values (equivalence classes) at every point**
  - ==> the abstract interpreter is an *attribute evaluator of the program*

Abstract Interpretation

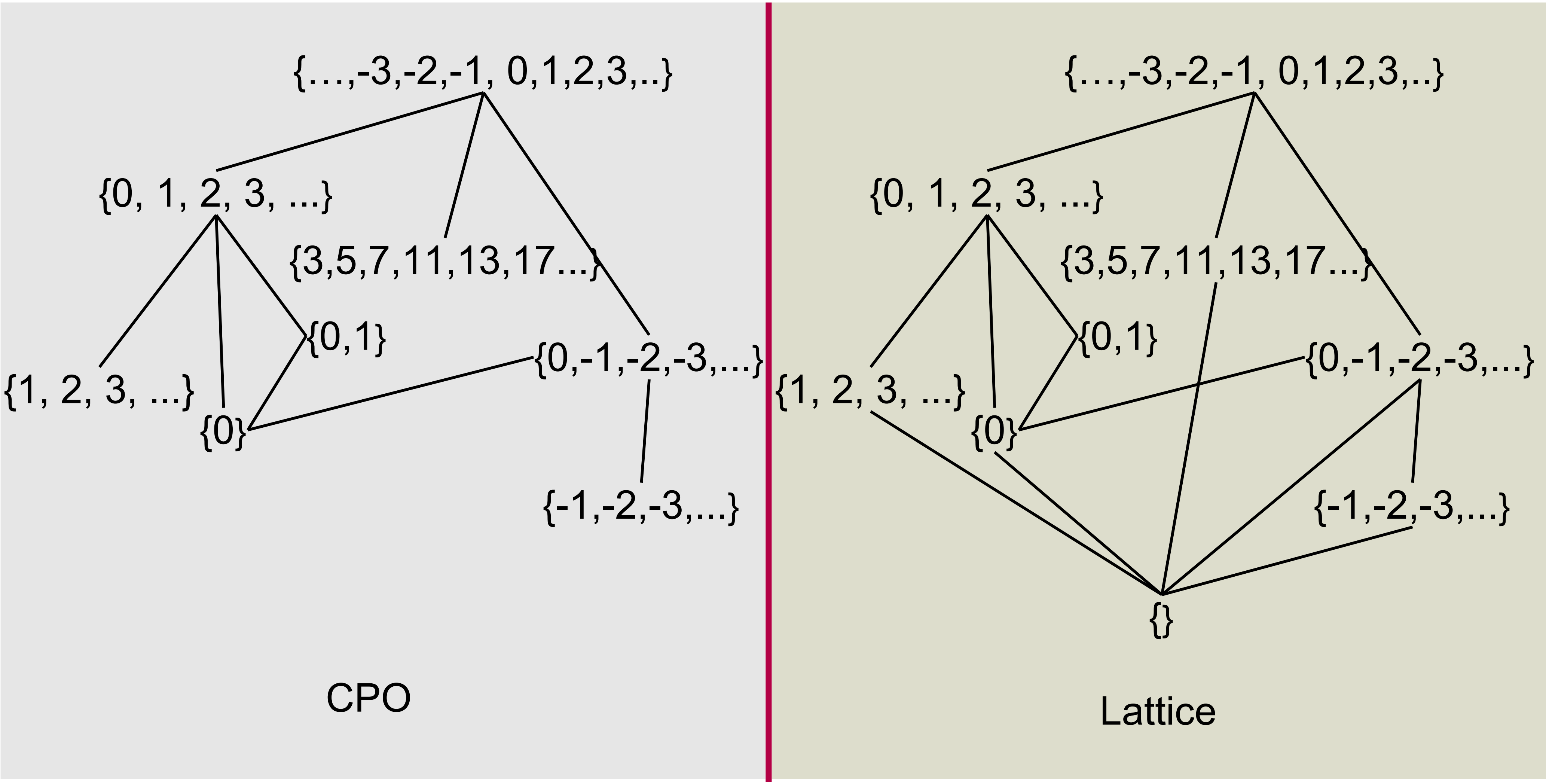
- ***Abstract interpretation*** *is static symbolic execution* of the program with *abstract symbolic values (equivalence classes containing possible values)*
  - Since the values cannot be concrete we must abstract them to "easier" values, i.e., simpler domains of *finite* count, height, or breadth, or equivalence classes
- Values are taken from the *abstract domains (equivalence class domains)* (called `D#`)
  - complete partial orders (cpo, with “or” or “subset”),
  - semi-lattices (cpo with some top elements) or
  - lattices (semi-lattice with top and bottom element)
  - The suprenum operation of the cpo expresses the “unknown”, i.e., the unknown decisions at control flow decision points (if's)
- An abstract interpreter works in a *shadow world*, corridor-oriented, i.e., on a shadow of the concrete values (corridor of values, intervals or symbolic abstractions of intervals)



https://commons.wikimedia.org/wiki/File:VW_Phaeton_20090712_front.JPG#/media/Datei:VW_Phaeton_20090712_front.JPG, User M 93

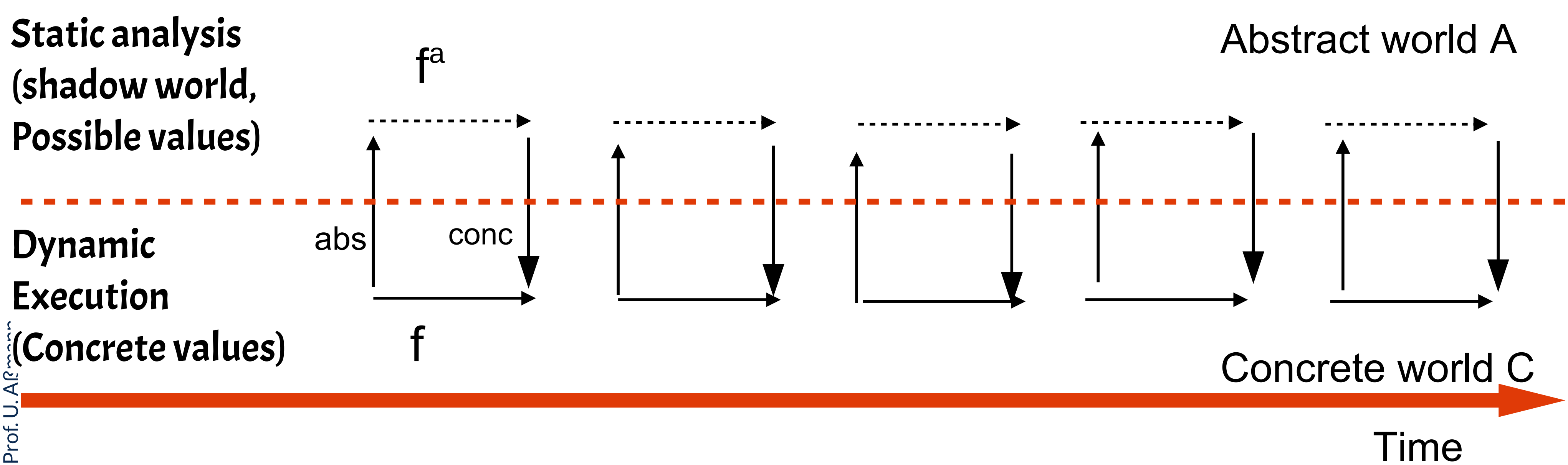
Complete Partial Orders (CPO) and Lattices with the Example of Integer Equivalence Classes

- CPO must have some “top elements”; lattice must have one top and one bottom element



Functions for Abstract Interpretation

- $f: C \rightarrow C$, run-time semantics of the program (**interpreter**)
- $abs: C \rightarrow A$, **abstraction function** from concrete to abstract
- $conc: A \rightarrow C$, **concretization function** from abstract to concrete
- $f^a: A \rightarrow A$, **abstract interpreter** (abstract semantic function, flow/transfer function)
  - The abstract interpreter is an over-approximization of the real values (safe corridor which includes the real value)
  - f^a is like a *shadow* of f



The Purpose of Abstract Interpretation

An abstract interpreter finds out where a value ***may flow*** (data flow analysis, value flow analysis, program flow analysis, model flow analysis)

- What is the type of this variable? (type inference, type checking)
- Are there competitive writes on shared variables?
- Can an expression be moved out of a loop?
- Can an expression be eliminated because it is use-less?
- How long does a program execute (worst-case execution time analysis)

More Precisely: Abstract Interpreters are Sets of Abstract Interpretation Functions

- For an abstract interpretation, for all node types 1..k in the control flow graph (or metaclasses in the language), set up *interpretation functions (transfer functions)*, each for one statement of the program
  - They form the core of the abstract interpreter

Real interpreter functions

f: Instruction x State → State

f(Statement, State) → State
f(Expression, State) → State

Abstract interpreter functions (transfer functions)

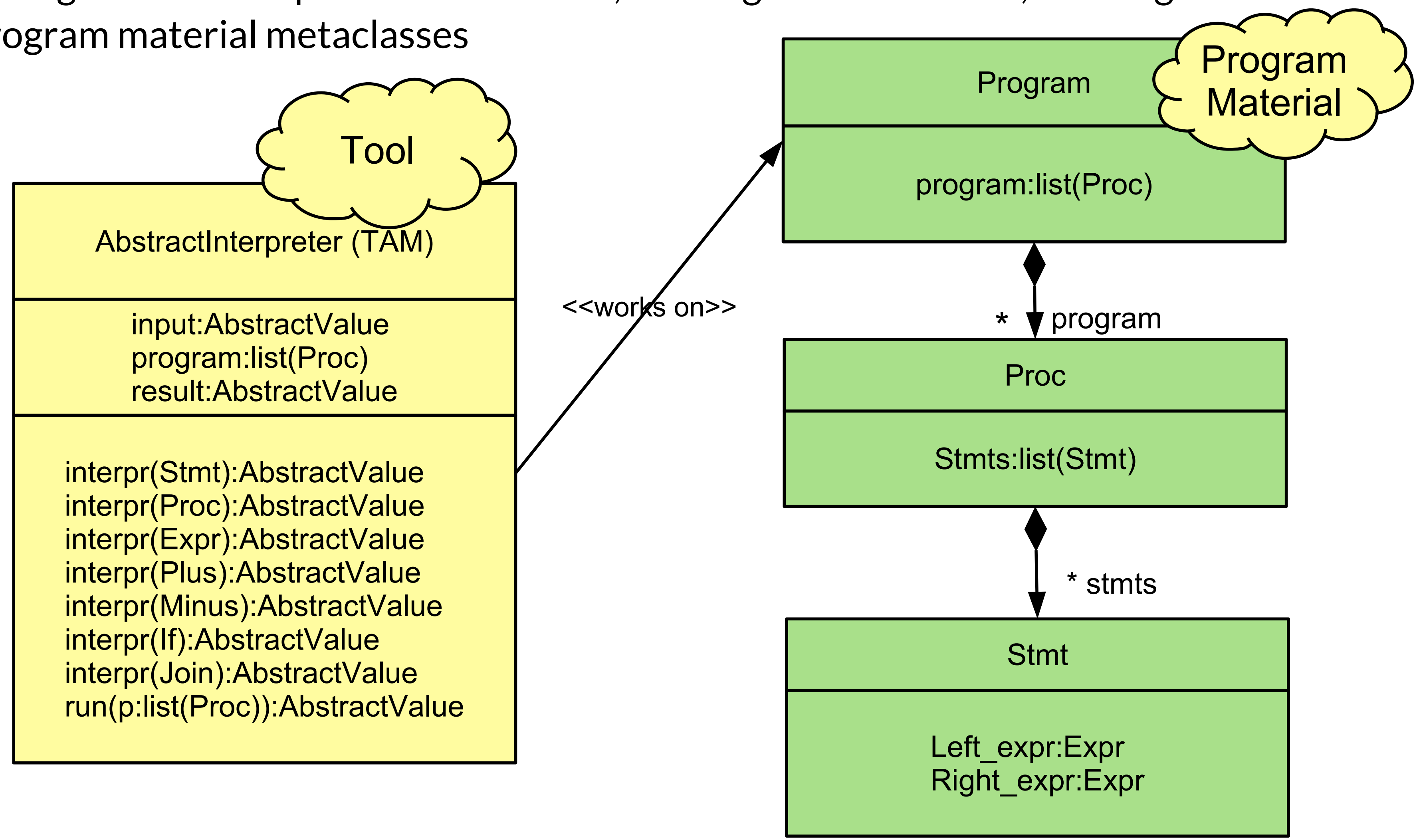
f: Instruction x AbstractState → AbstractState

f(Statement, AbstractState) → AbstractState
f(Expression, AbstractState) → AbstractState

22.2 Implementation Patterns for Interpreters and Abstract Interpreters

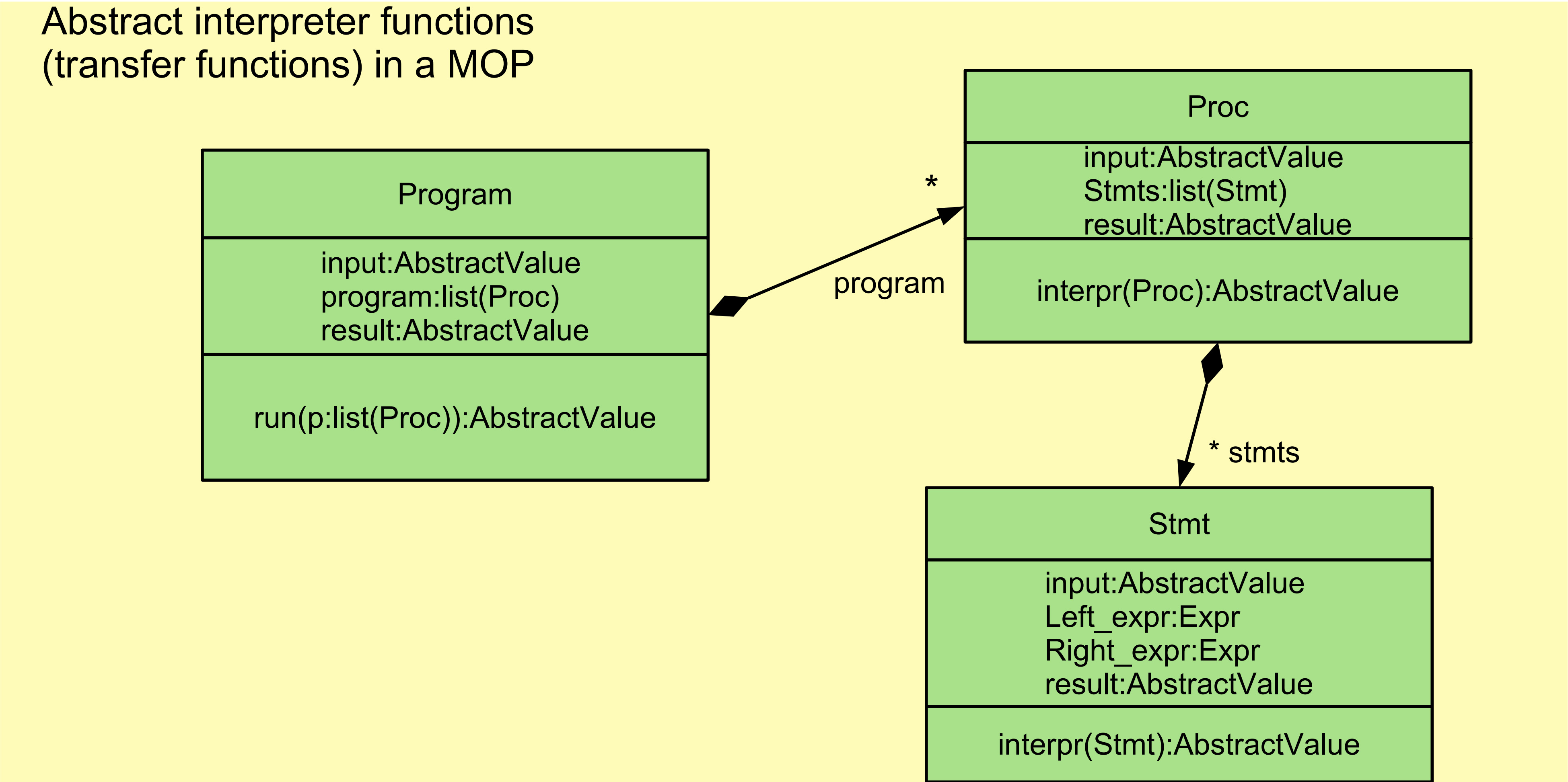
Implementation Pattern 1 (TAM-based Interpreters): An Abstract Interpreter is a Tool on Program Materials

- The *interpretation functions (transfer functions)* of an abstract interpretation may be arranged in an interpreter class on M2, forming a *tool metaclass*, working on the program material metaclasses



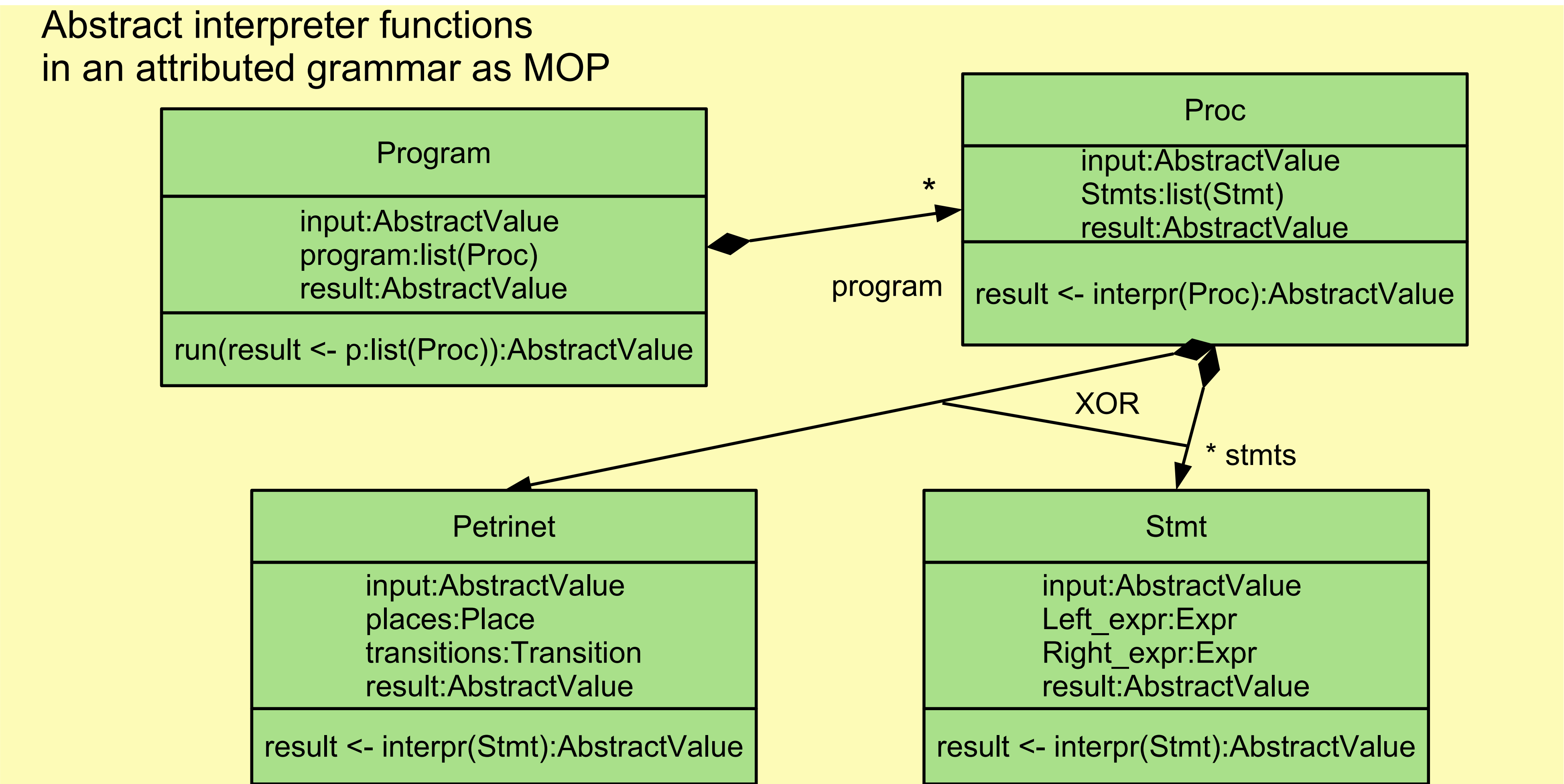
Implementation Pattern 2 (MOP-based Interpreters): Object-Oriented Abstract Interpreters are Sets of Abstract Interpretation Functions Encapsulated in Metaclasses

- The *interpretation functions (transfer functions)* of an abstract interpretation may be arranged **in the metaclasses of M2** (the language concepts)
- Then, we call the abstract interpreter a **abstract meta-object-protocol (aMOP),** and we do not distinguish tools and materials

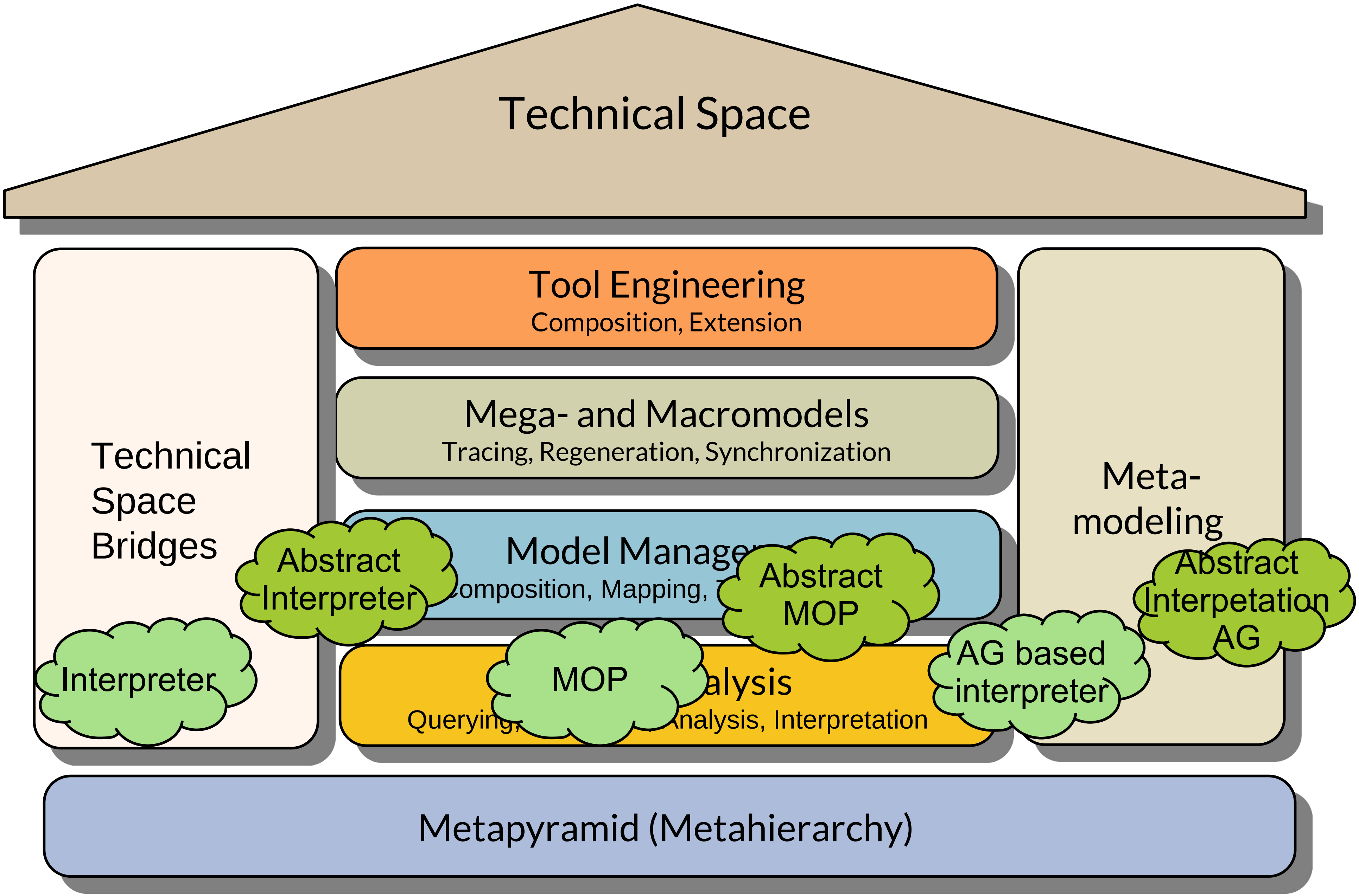


Implementation Pattern III (MOP-AG-Interpreters): Abstract Interpreters can be Specified by AG

- The *interpretation functions (transfer functions)* of an abstract interpretation may be arranged **in the metaclasses of an attributed grammar M2**
  - Then, the syntax trees (hierarchic) are described by a grammar
- Then, we call the abstract interpreter a **abstract-interpretation attribute grammar**
  - storing the results in attributes of the tree.



Q10: The House of a Technical Space

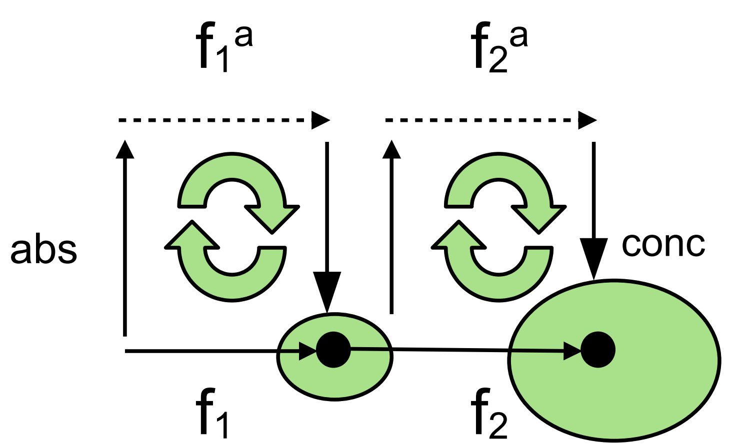


22.3. The Laws of Abstract Interpretation for Deep Analysis of Programs

The Iron Law of Abstract Interpretation: Faithfullness

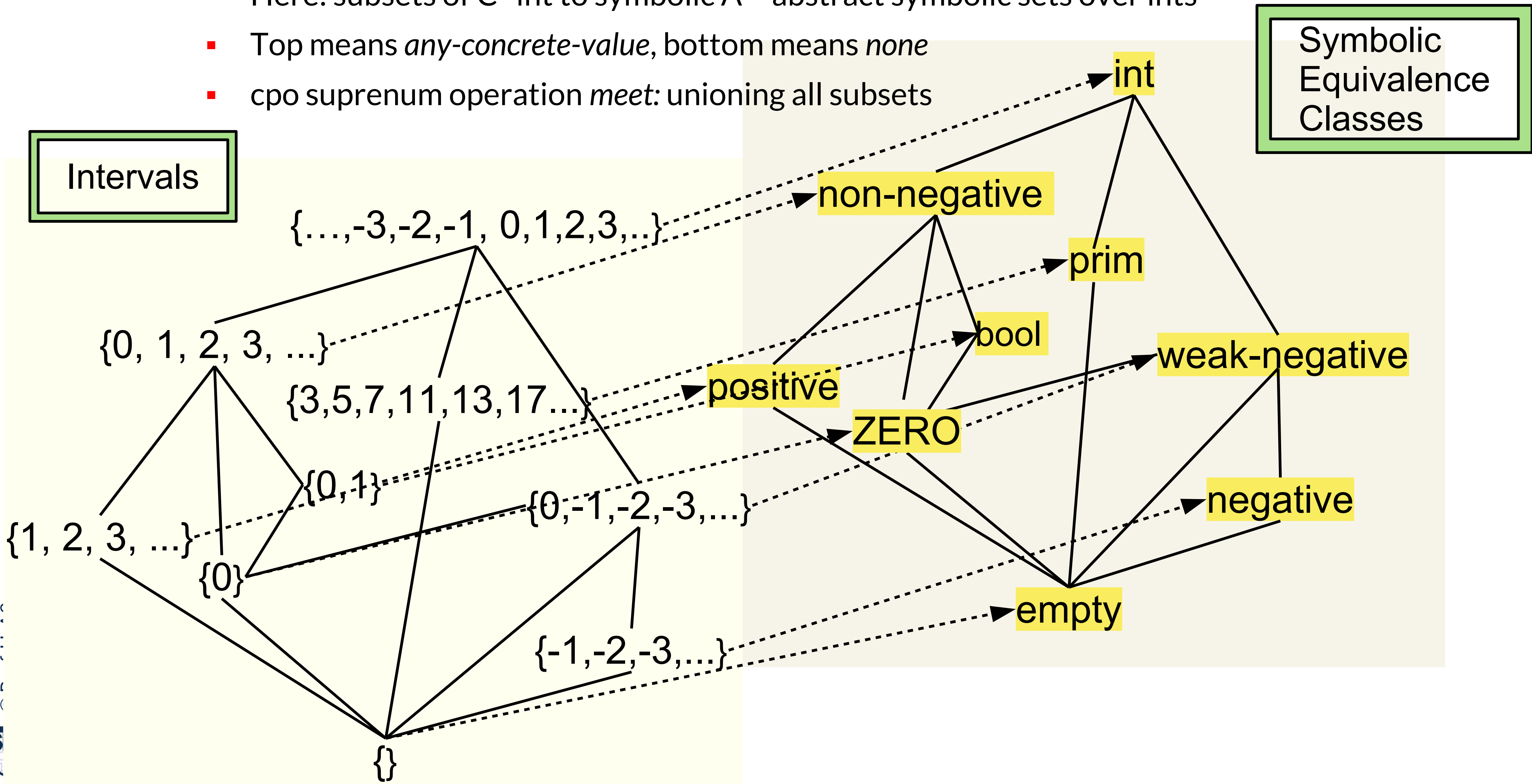
The abstract interpretation must be *correct (conservative)*, i.e., faithfully abstracting the run-time behavior of the program („reality proof"):
$f \subset conc \circ f^a \circ abs$

- The shadow must be faithful; the ***corridor of possible values must contain the real value***
- abs (abstraction function), conc (concretization function), and f^a (abstract interpretation function) must form a commuting diagram
  - The abstract interpretation should deliver all correct values, but may be more
  - They must be "interchangeable", formally: a Gaulois connection
- The interpretation must be a subset of the abstract interpretation:
  - $f \subset conc \circ f^a \circ abs$
  - The concrete semantics must be a subset of the concretization of the abstract semantics (conservative approximation)
  - $conc \circ f^a \circ abs \supset f$
  - The abstract semantic value must be a superset of the concrete semantic value after application of the transfer function
  - The concrete value of f must be a subset of the abstracted value after application of the transfer function



Ex. Concrete and Abstract Values (Equivalence Classes) over Integers

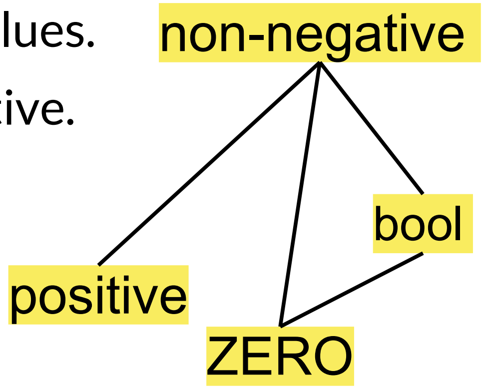
- A program variable v has a value from a concrete domain C (here Integers)
- At a point in the program, v can be typed by a subset of C (an equivalence class)
- This concrete domain C is mapped to symbolic abstract domain A
  - Here: subsets of C=int to symbolic A=”abstract symbolic sets over ints”
  - Top means *any-concrete-value*, bottom means *none*
  - cpo suprenum operation *meet:* unioning all subsets



Law of Join of Control Flow in an Abstract Interpreter

When the abstract interpreter does not know what the type of a variable will be from 2 or n incoming control-flow paths at a join,
it takes the suprenum („union") of the equivalence classes of the abstract domain

- In a *join point* of the control flow (at the end of an If, Switch, While, Loop, Call), an abstract interpreter will not know from which incoming path it should select the value
  - **If**: two paths
  - **Switch:** finitely many paths
  - **While, Loop:** infinitely many paths
  - **Call:** from a return of the called procedure
- In order to proceed, the interpreter chooses the *suprenum* of the equivalence-class values of all paths (the *meet* of all values of all incoming paths), i.e. it will choose the union or the most simple abstraction of all equivalence-class values.
- Ex.: in a Switch the values of the branches are ZERO, bool, positive.
  - The interpreter will choose "non-negative", to cover all.



Ubiquituous Abstract Interpretation for Deep Analysis of Programs and Models

- Any program in any programming or specification language can be interpreted abstractly, if
  - A syntax tree (link tree, or a graph model) is given
  - An abstract semantics is given, mapping the tree nodes to interpretation functions over abstract values
- The abstract interpreter is an implementation of the metaclasses of the M2 metamodel
- Examples:
  - Imperative Programs: A.I. of embedded C, C++, Java, C#, Scala programs
  - Rule-based Programs: A.I. of Prolog rule sets, A.I. of ECA-rule bases
  - Models: A.I. of state machines, A.I. of Petri Nets
- Functional analysis
  - Value analysis ("data-flow analysis") for numeric values and pointers
    - Range check analysis, null check analysis
    - Heap analysis, alias analysis
- Quality analyses:
  - Worst case execution time analysis (WCETA)
  - Worst case energy analysis (WCENA)
  - Security analysis

22.4 Iteration Strategy of Abstract Interpreters (Intra- and Interprocedural Visit Order)

Example: Interpretation of a Procedure with a Worklist Algorithm

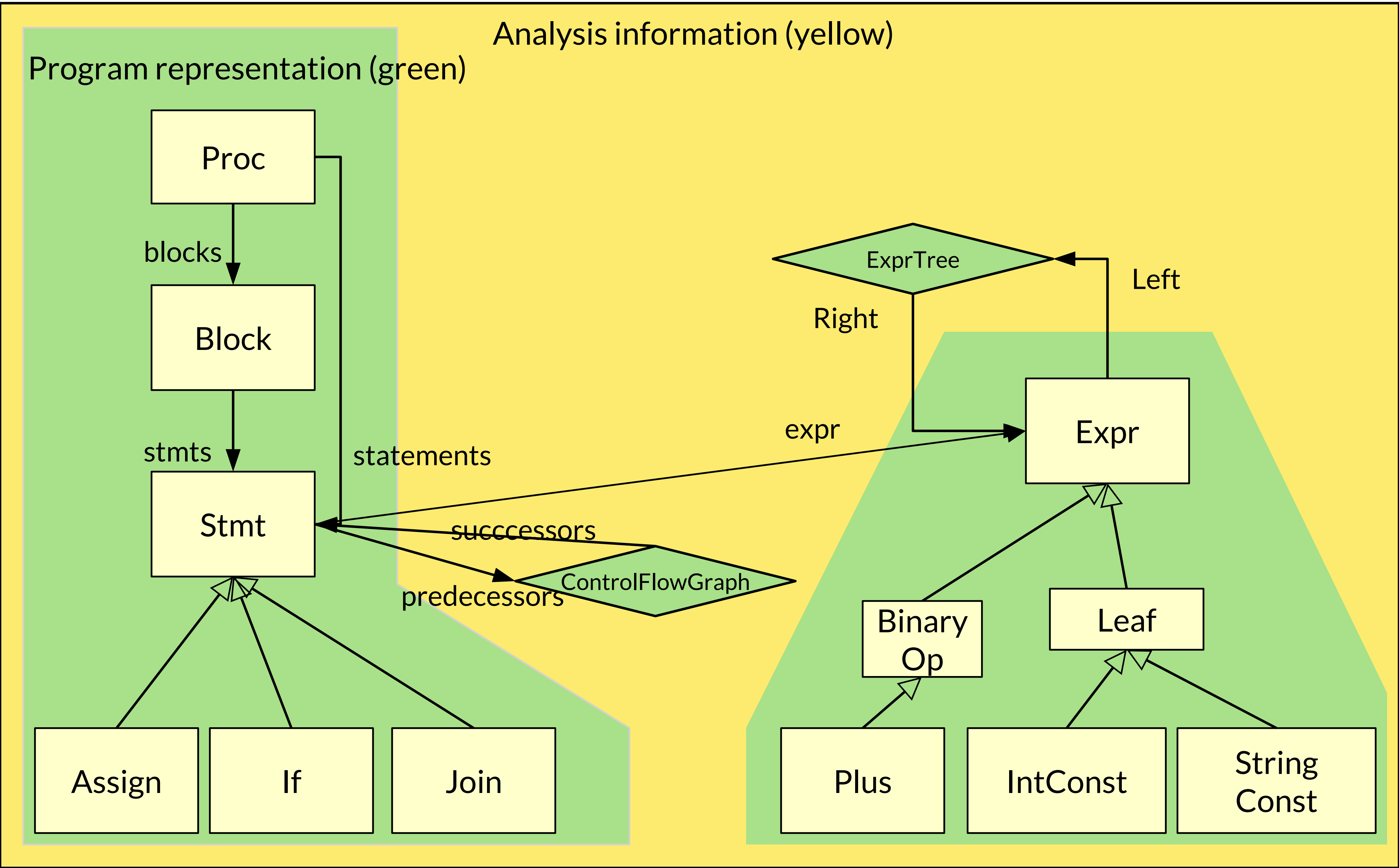
- Iteration can be done *forward* over a worklist of statements that contains “nodes of the syntax tree not finished”
- The abstract interpretation functions `fª(p)` are applied as long as there are changes in the attributes
- For a AG this means: application of attribution functions is free-choice

```
worklist := nodes of syntax tree;
WHILE (worklist != NULL) DO
SELECT n:node FROM worklist;
// forward propagation from predecessors to n
            FORALL p in n.ControlFlowGraph.predecessors
               X := meet( fª(p.abstract_value()) );
            // test fixpoint condition
            IF (X != n.abstract_value()) THEN
               // reattribution
               n.abstract_value() = X;
               worklist += n.ControlFlowGraph.successors;
           END
END
```

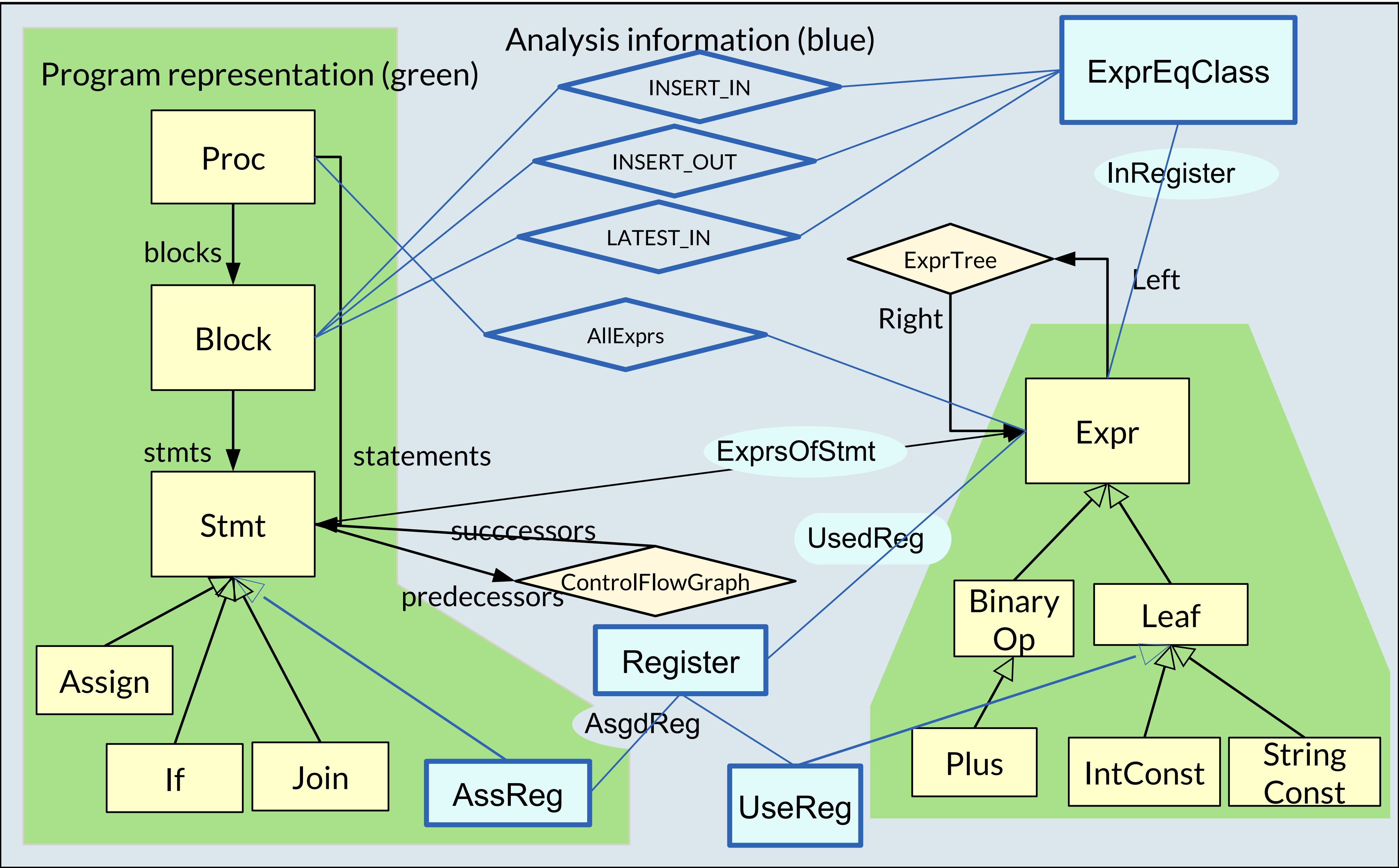
Building Abstract Interpreters on M2

- In the TAM style, the interpreter works basically with Design Pattern “`Interpreter`”, as from the Gamma book
- What has to be modeled:
  - A model of the program (program representation), with Class, Proc, Stmt, Expr, etc
    - Most often, this is a syntax tree (with links)
  - A model of the analysis information
    - ControlFlowGraph: has inserted Join nodes representing control flow joins in If#s and While's
    - AbstractValue domains: e.g., abstract integers, abstract intervals and ranges, abstract heap configurations
    - Environments binding variables to abstract values

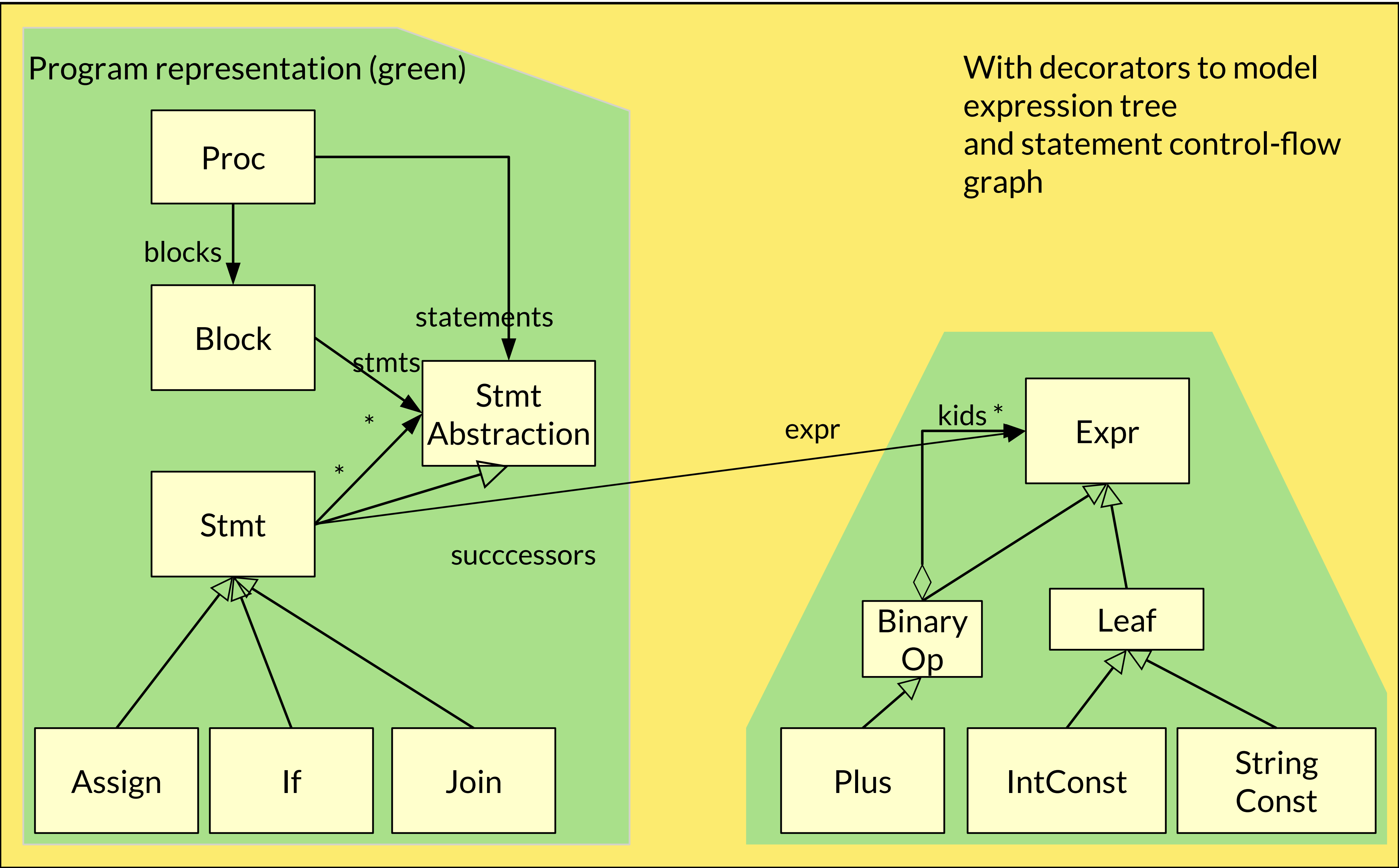
A Simple Intraprocedural Program (Code) Model (Schema) in MOF



Q14: A Simple Intraprocedural Program (Code) Model (Schema) in MOF

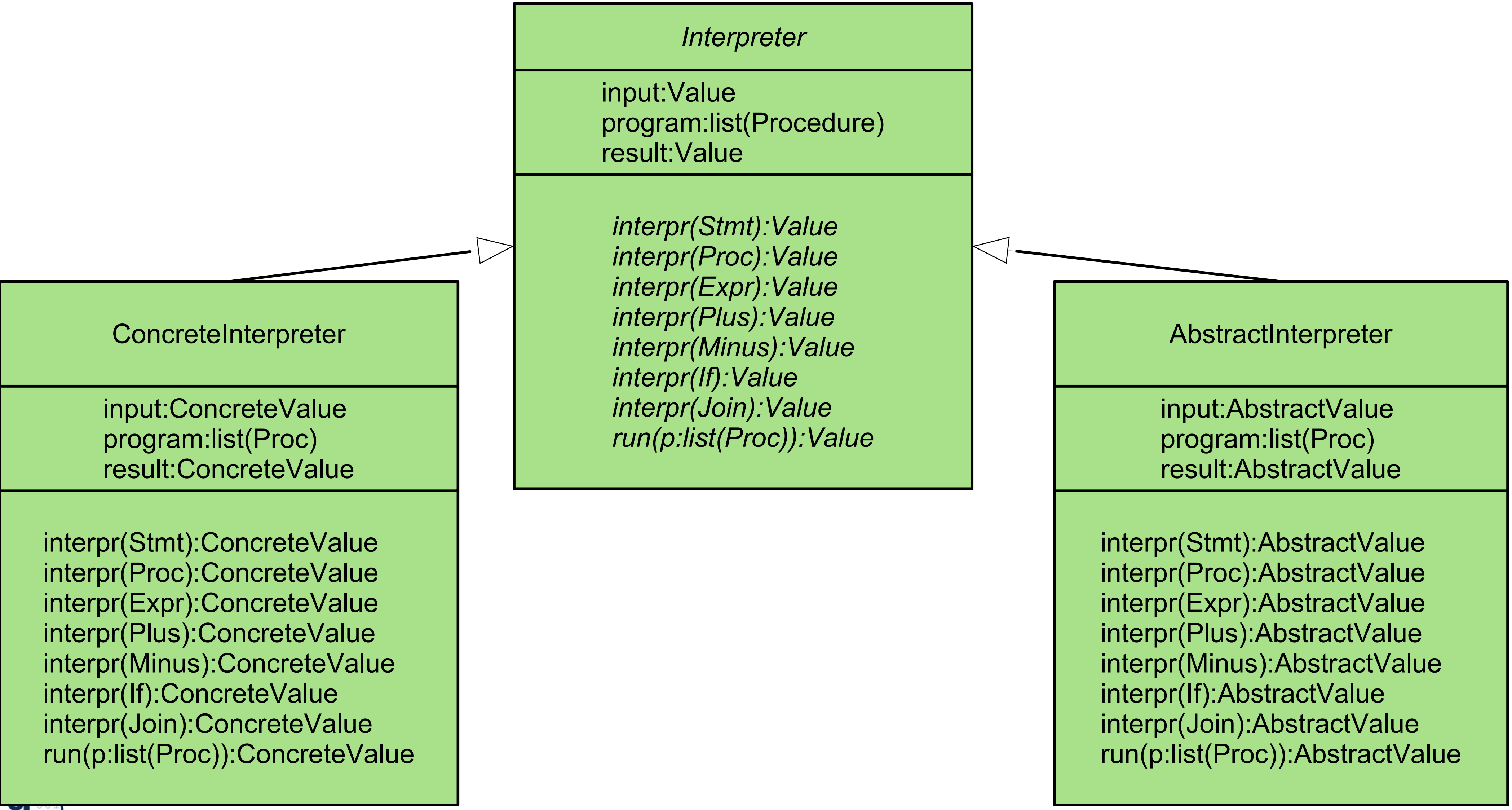


A Simple Program (Code) Model (Schema) in EMOF



An TAM-Design of an Interpreter Family of a Programming Language

- Concrete and abstract interpreters are “twins”, i.e., have the same interface but working on concrete vs abstract values



Example: TAM-Interpretation of a Procedure with a Worklist Algorithm

- Simplified assumption: one value per statement is computed by the abstract interpreter.
- The value at the return statement of the interpreted procedure is the final result of the abstract interpretation

```
CLASS AbstractInterpreter EXTENDS Interpreter {
…
  FUNCTION interpr(p:Proc):AbstractValue {
    worklist:list(Statement) := p.statements;
    WHILE (worklist != NULL) {
      SELECT current:Statement FROM worklist;
      // forward propagation from current.predecessors to current
      FORALL pred in current.ControlFlowGraph.predecessors {
        NewValue := meet( pred.abstract_value() );
      }
      // test whether fixpoint is reached
      IF (NewValue != current.abstract_value()) {
        current.abstract_value() := NewValue;
        worklist += current.ControlFlowGraph.successors;
      }
    }
    RETURN p.statements.last.abstract_value;
  }
}
```

22.2.2 Free-Choice Visit Theorem (Intraprocedural Coincidence Theorem) for Abstract Interpreters

[Kam/Ullman] Intraprocedural Coincidence Theorem:

The maximum fixpoint of an iterative evaluation of the system of abstract-interpretation functions f_n at a node n
is equal
to the value of the meet-over-all-paths to the node n (MOP(n)).

- Forall n:Node: $MFP(n,f_n) = MOP(n,f_n)$
- The theorem means, that no matter how the abstract-interpretation functions are iterated over a procedure (free-choice visit), if they stop at a fixpoint, they stop at the meet-over-all-paths
  - Any iteration algorithm can be used to reach the abstract values at each node (i.e., the maximal fixpoint of the function system)
  - The paths through a procedure need not be formed (there may be infinitely many), instead, free iteration can be used until the fixpoint is found (until termination of the iteration)
- The application of an attribution function is similar to a free rewriting step

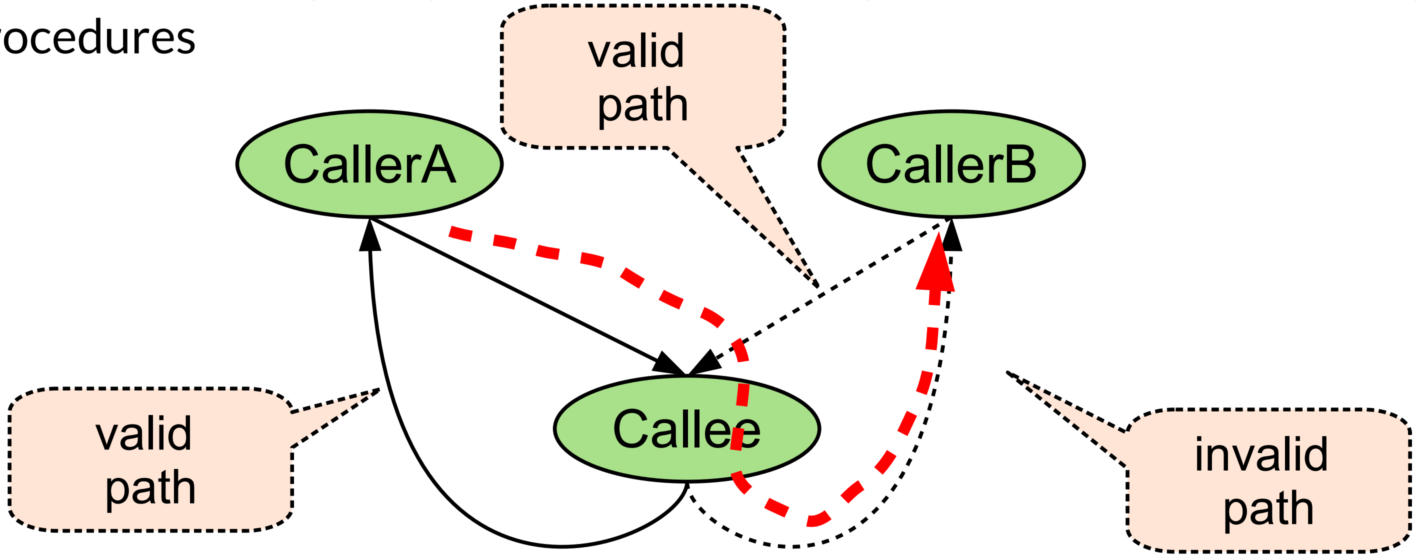
Example: Backward TAM-Interpretation with Worklist Algorithm

- Iteration can be done with many strategies
- E.g., iterating *backward* over a worklist that contains "nodes not finished"
- Other alternatives: innermost-outermost, lazy, etc.

```
CLASS AbstractInterpreter EXTENDS Interpreter {
…
  FUNCTION interpr(p:Procedure):AbstractValue {
    worklist:list(Statement) := p.statements;
    WHILE (worklist != NULL) {
      SELECT current:Statement FROM worklist;
      // backward propagation from current.successors to current
      FORALL succ in current.ControlFlowGraph.successors {
        NewValue := meet( succ.abstract_value() );
      }
      // test whether fixpoint is reached
      IF (NewValue != current.abstract_value()) {
        current.abstract_value() := NewValue;
        worklist += current.ControlFlowGraph.predecessors;
      }
    }
    RETURN p.statements.last.abstract_value();
  }
}
```

Interprocedural Control Flow Graphs and Valid Paths

- Transfer Functions f# can be defined on Nodes f#(n), or even on Edges f#(e)
- **Interprocedural edges** are call edges from caller to callee
- **Local edges** are within a procedure from "call" to "return"
- Problem: not all interprocedural paths will be taken at the run time of the program
  - Call and return are *symmetric*
  - From whereever I enter a procedure, to there I leave
- An **interprocedurally valid path** respects the symmetry of call/return
- Important in program graphs, sequence diagrams, communication diagrams, Petri-net procedures



Interprocedural Problems

- Non-valid interprocedural paths invalidate the coincidence for the interprocedural case
- Knoop found a restricted one [CC92]:
  - No global parameters of functions
  - Restricted return behavior

The End

- Explain the differences of an interpreter and an abstract interpreter!
- Why are interpreters and abstract interpreters specified on an abstract syntax tree specified by an RTG?
- Can models be interpreted?
- What are the differences of an abstract interpreter and an attributed grammar?
- Why is a reference attributed grammar (RAG) more expressive than a pure AG?
- What happens at a control-flow join during an abstract interpretation?
- Explain abstract domains and the iron law of abstract interpretation.