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)

DRESDEN concept Exzellenz aus Wissenschaft und Kultur

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.

© Prof. U. Aßmann

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

© Prof. U. Aßmann

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.

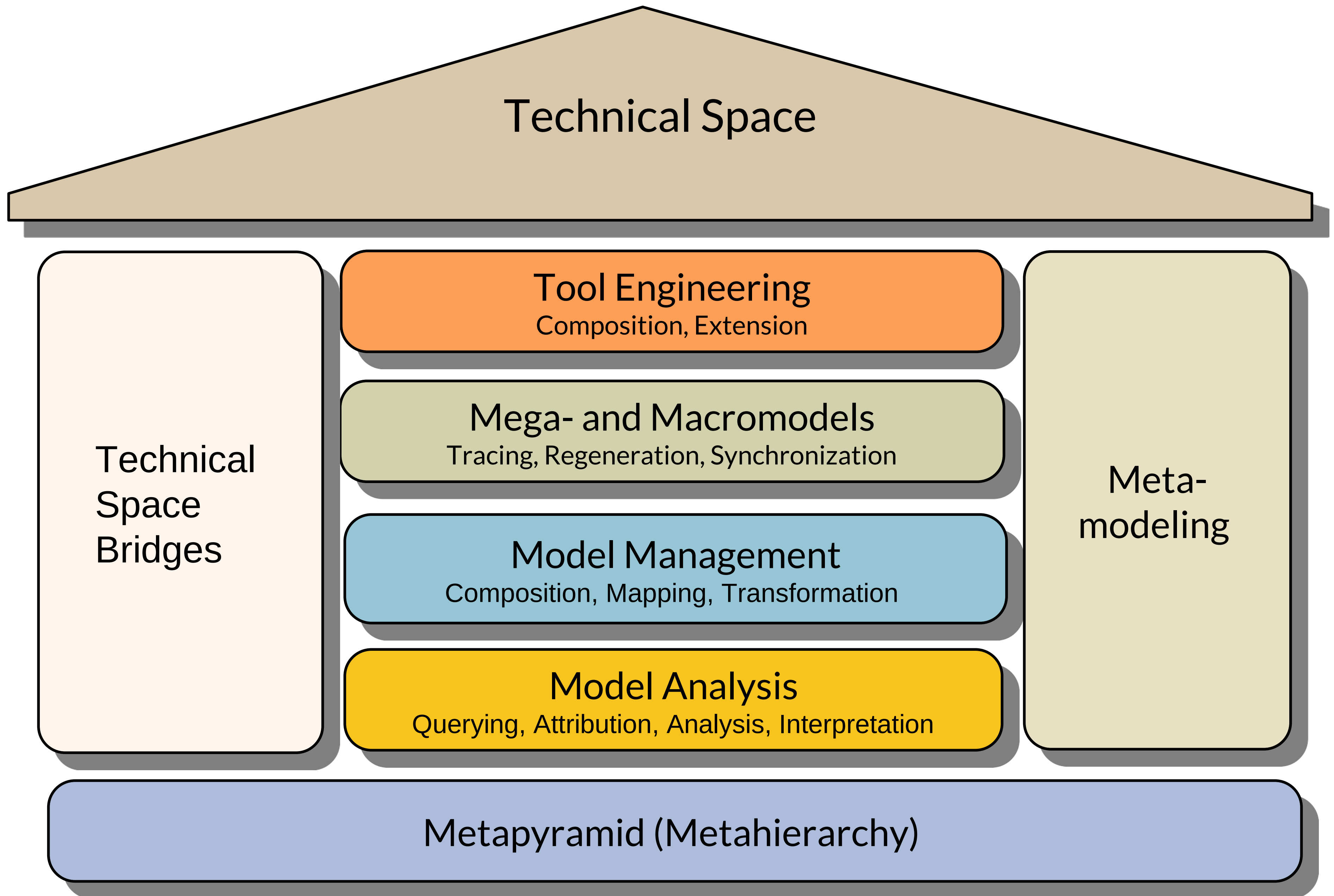
© Prof. U. Aßmann

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.

© Prof. U. Aßmann

Q10: The House of a Technical Space



© Prof. U. Aßmann

22.1 Interpretation and Abstract Interpretation (Ab.I.)

DRESDEN concept
Exzellenz aus Wissenschaft und Kultur

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

- **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)

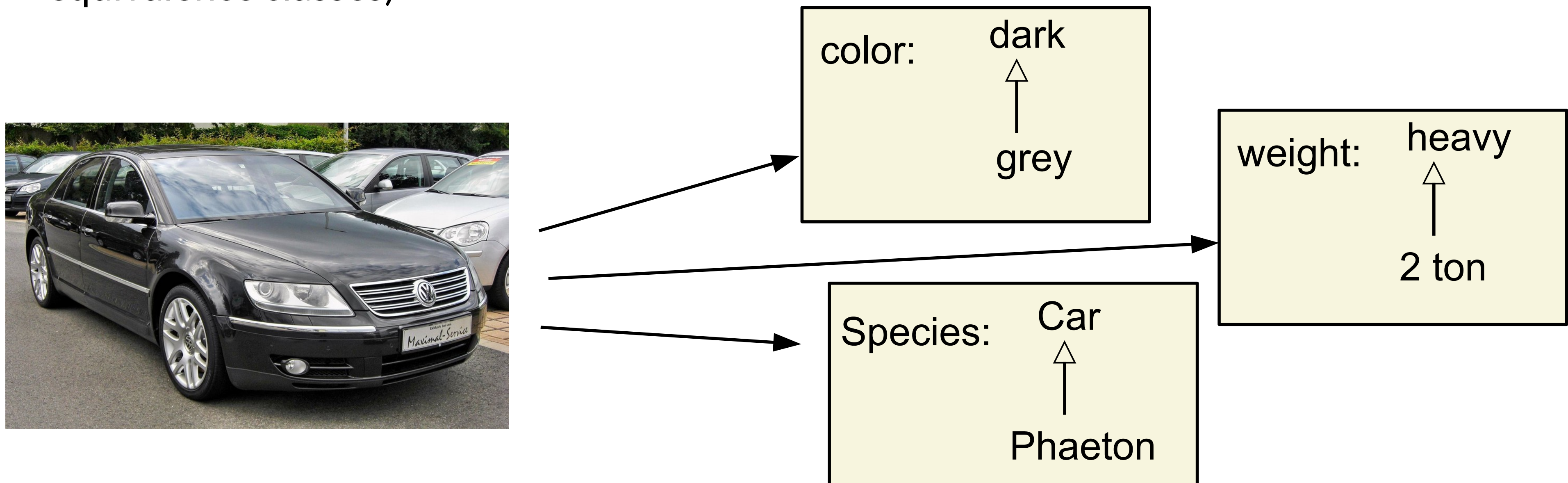
```
X := 10;
Y := if (B < 5) then X;
                else X+1;
// Y = { X, X+1} // symbolic execution
// Y = { 10 \or|B 11 } // abstract interpretation
```

© Prof. U. Aßmann

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)



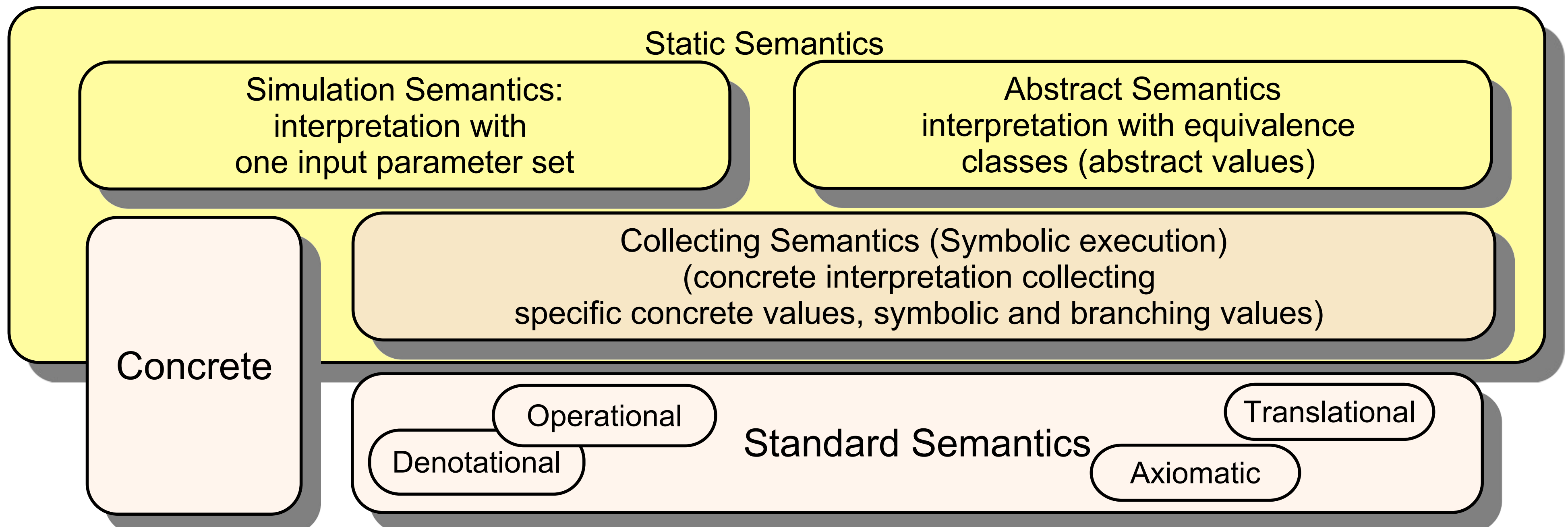
https://commons.wikimedia.org/wiki/File:VW_Phaeton_20090712_front.JPG#/media/Datei:VW_Phaeton_20090712_front.JPG, User M 93

© Prof. U. Aßmann

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

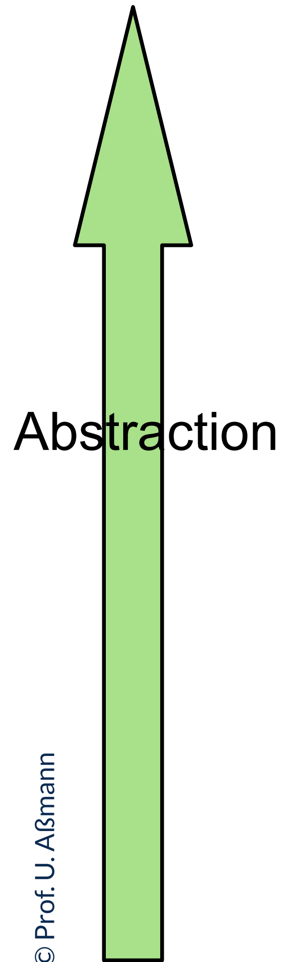
Interpretation and Semantics of Programs

- 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



© Prof. U. Aßmann

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

Concrete

Collecting Semantics (concrete interpretation collecting specific concrete values)

Standard Semantics

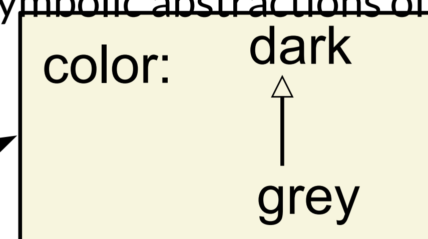
What is an Interpreter?

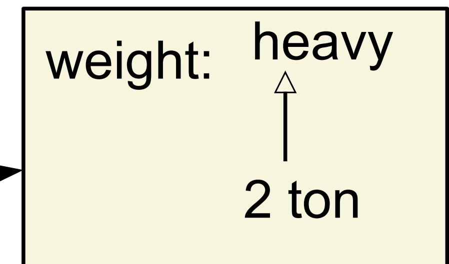
- An **interpreter** executes a program (tree) on a set of input data and realizes an operational semantics
  - An interpreter is based on an operational semantics over state
  - For an object-oriented language, for all metaclasses of the language on M2, interpretation functions have to be given
- **The 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*

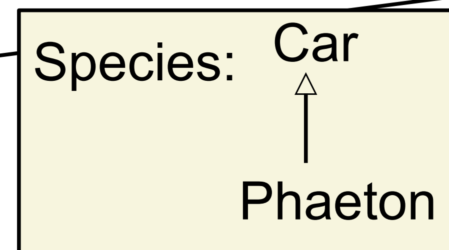
© Prof. U. Aßmann

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)





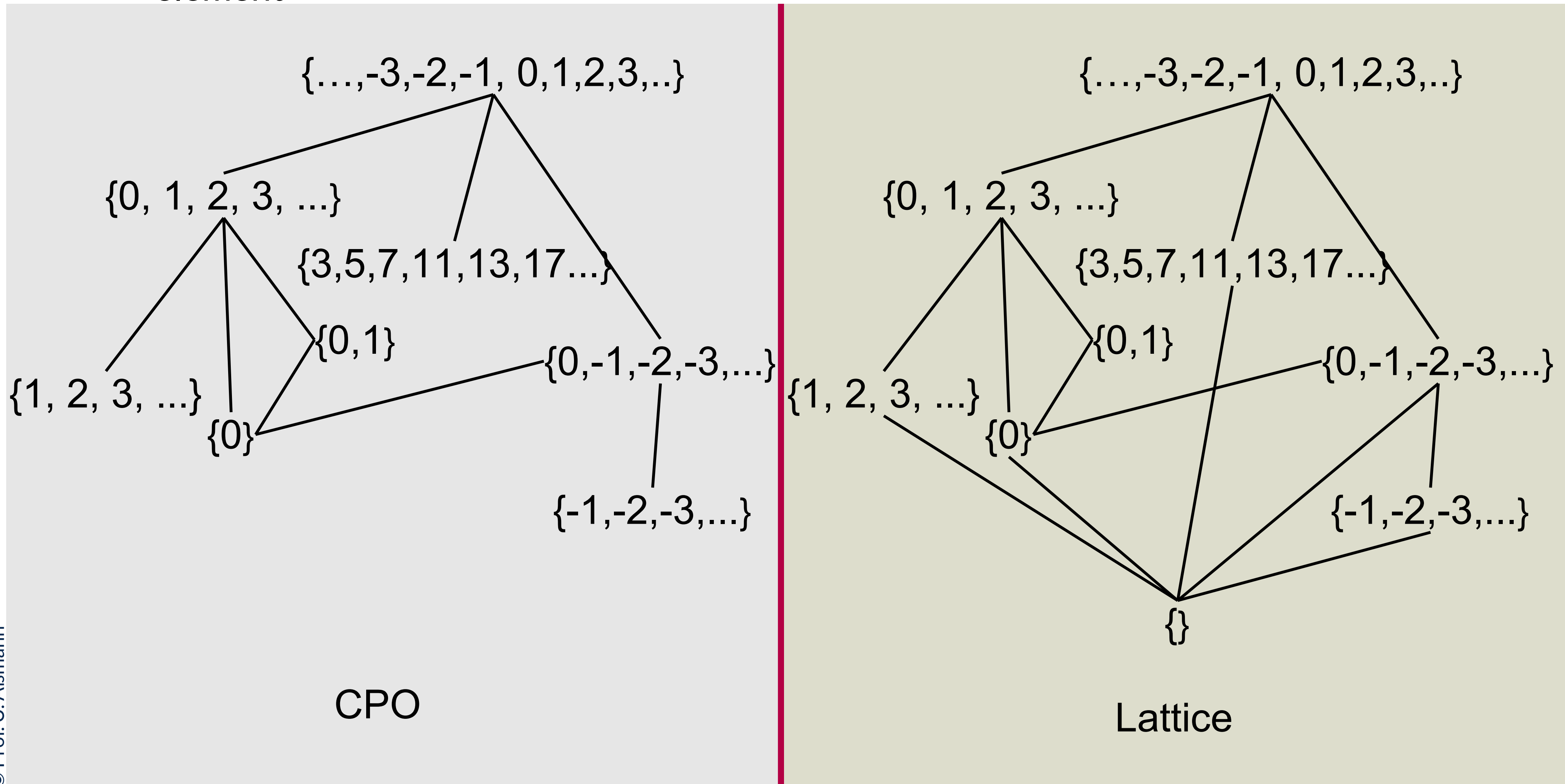


https://commons.wikimedia.org/wiki/File:VW_Phaeton_20090712_front.JPG#/media/Datei:VW_Phaeton_20090712_front.JPG, User M 93

© Prof. U. Aßmann

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

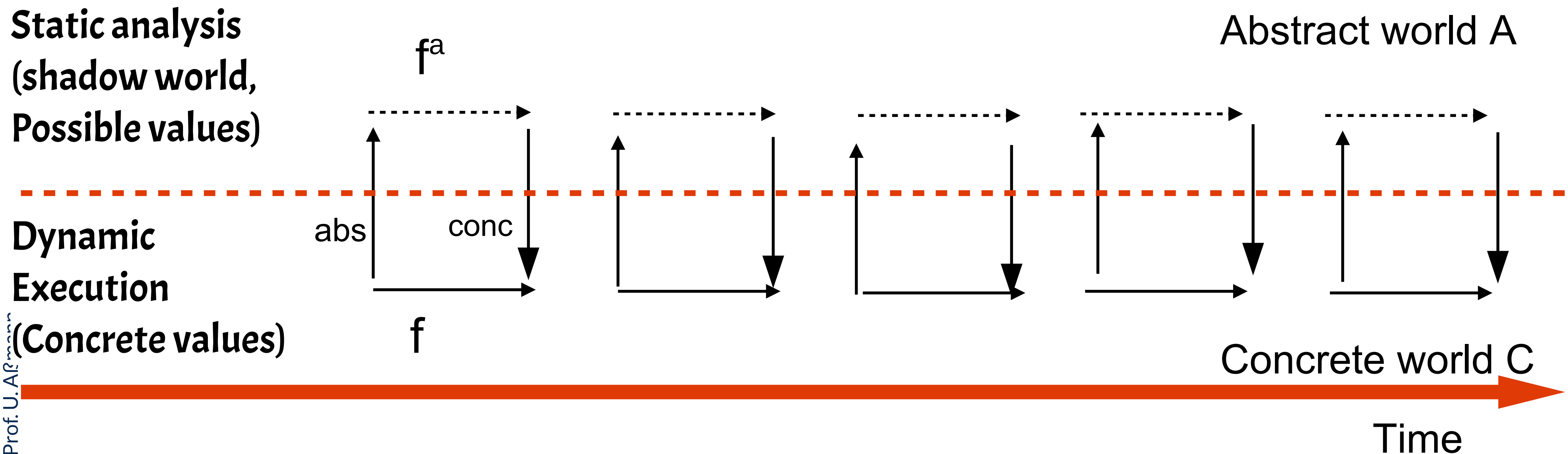
- CPO must have some “top elements”; lattice must have one top and one bottom element



© Prof. U. Aßmann

Functions for Abstract Interpretation

- f: C → C, run-time semantics of the program (**interpreter**)
- abs: C → A, **abstraction function** from concrete to abstract
- conc: A → C, **concretization function** from abstract to concrete
- f^a:A → 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



© Prof. U. Aßmann

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)

© Prof. U. Aßmann

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

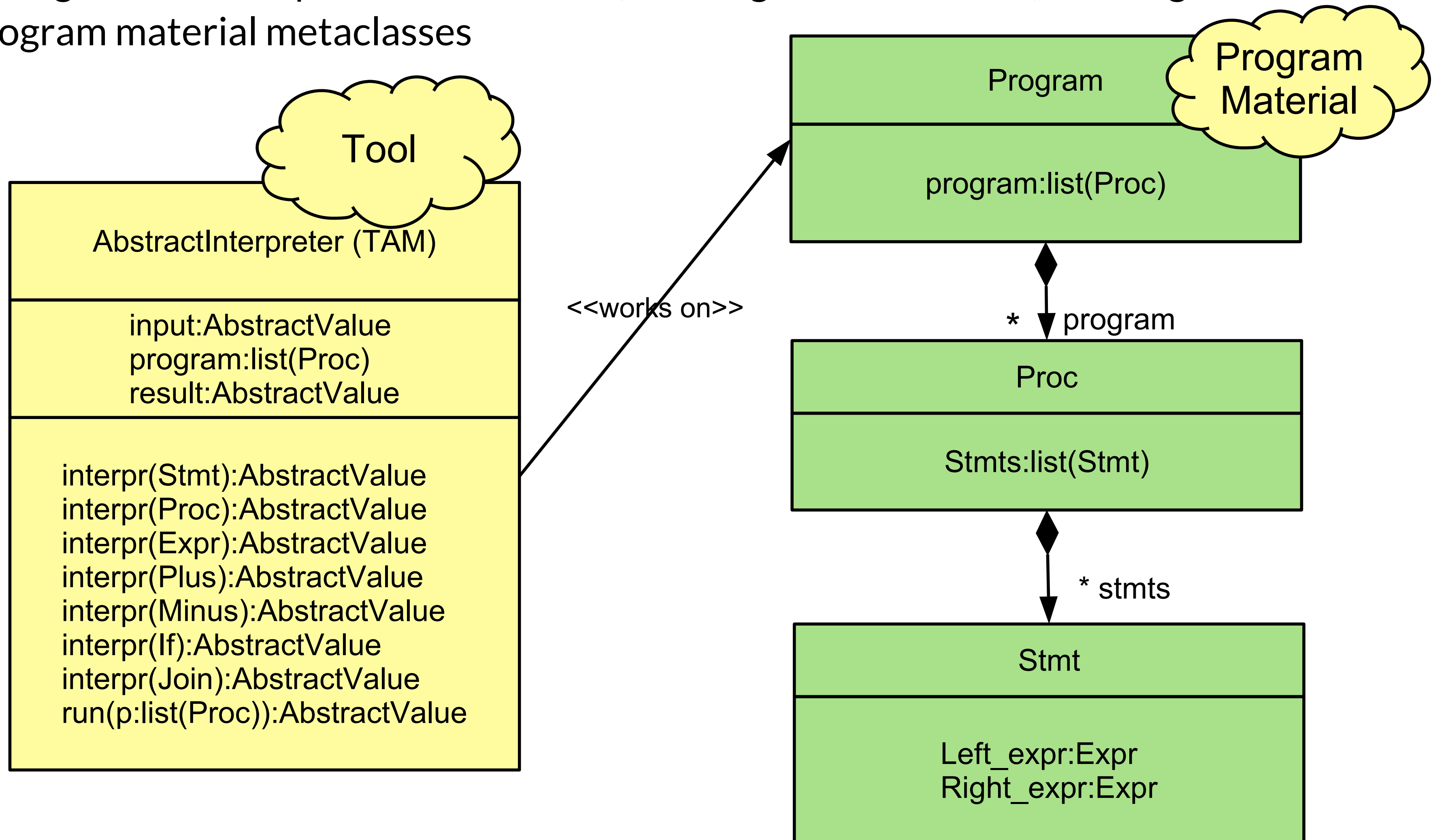
f(Statement, AbstractState) → AbstractState
f(Expression, AbstractState) → AbstractState

© Prof. U. Aßmann

22.2 Implementation Patterns for Interpreters and Abstract Interpreters

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

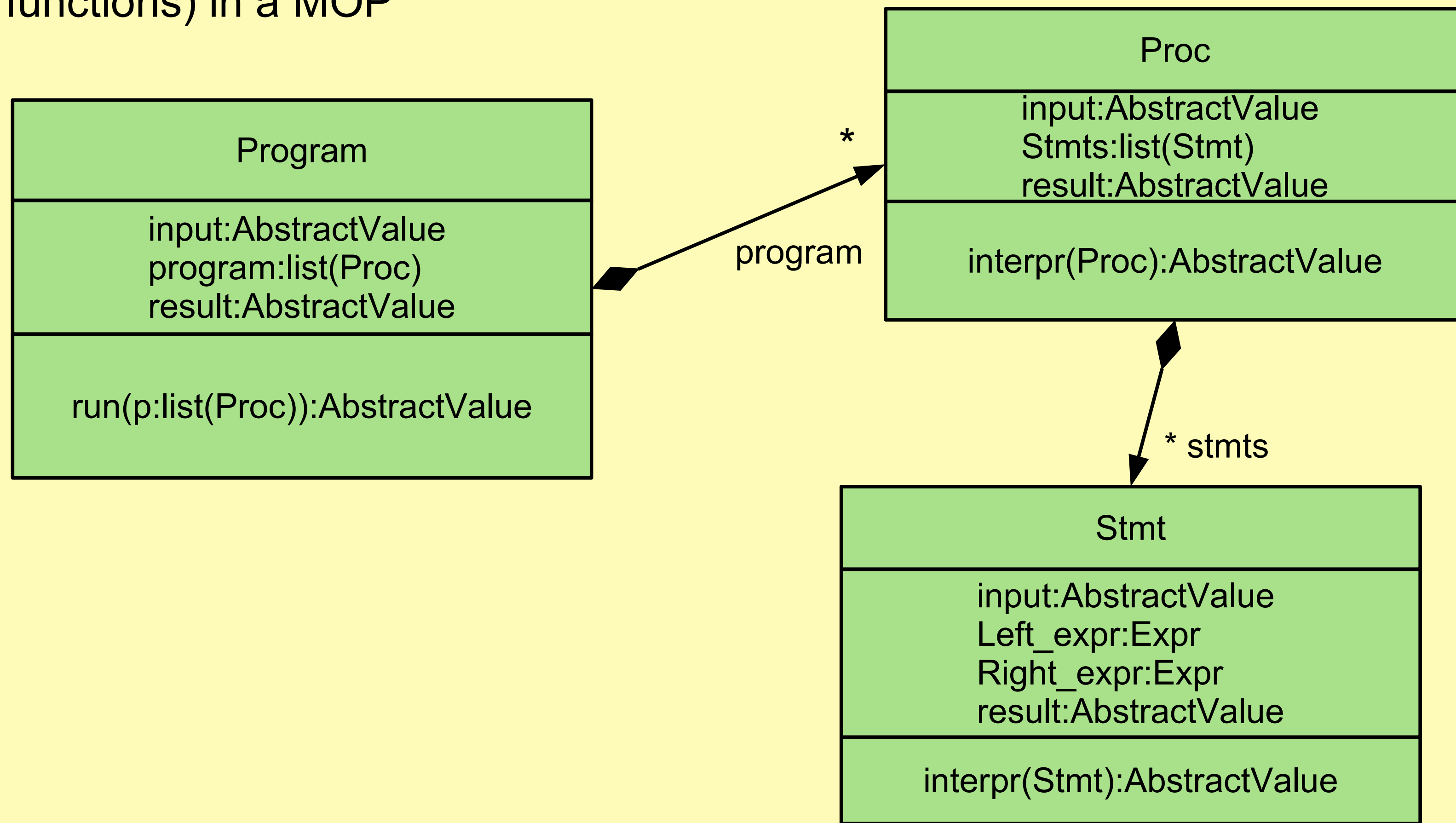
- The *interpretation functions (transfer functions)* of an abstract interpretation may be arranged in an interpreter class on M2, forming a *tool metaclass*, working on the program material metaclasses



© Prof. U. Aßmann

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



© Prof. U. Aßmann