14) Abstract Interpretation of Behavior Specification Languages

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 11-1.2, 10.12.11

1) Abstract Interpretation (AI)
2) Iteration

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

Other Resources

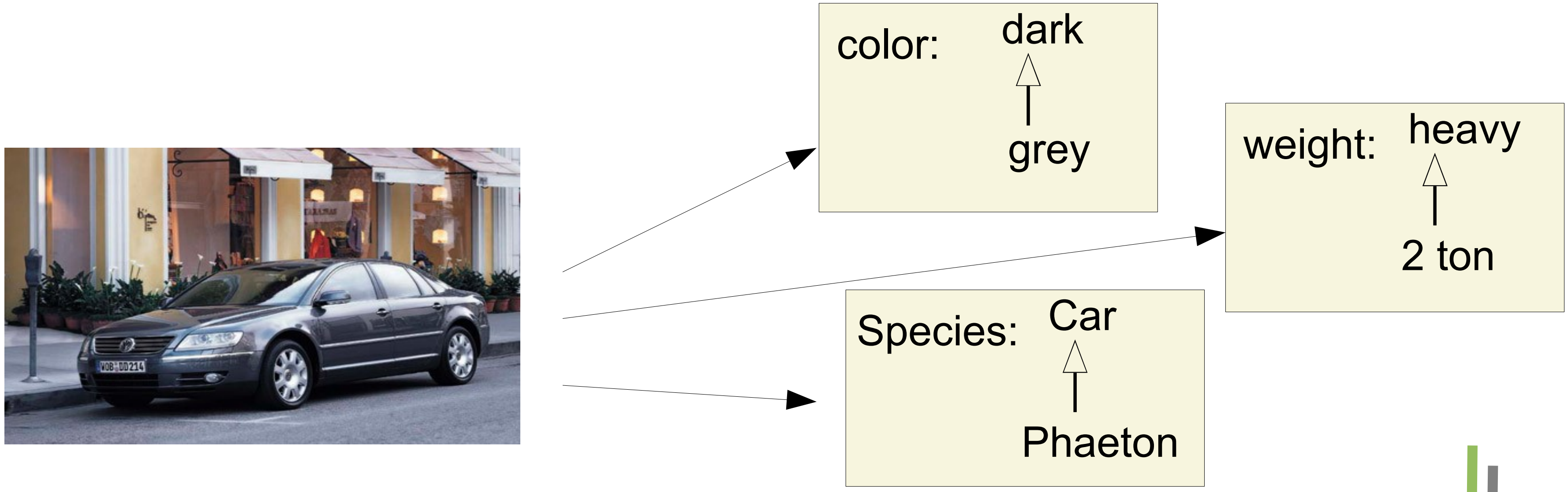
- Selective reading:
  - Neil D. Jones and Flemming Nielson. 1995. Abstract interpretation: a semantics-based tool for program analysis. In Handbook of logic in computer science (vol. 4), S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum (Eds.). Oxford University Press, Oxford, UK 527-636.
    - http://dl.acm.org/citation.cfm?id=218637
  - 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.

14.1 Abstract Interpretation (A.I.)

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#)



color: dark
grey

weight: heavy
2 ton

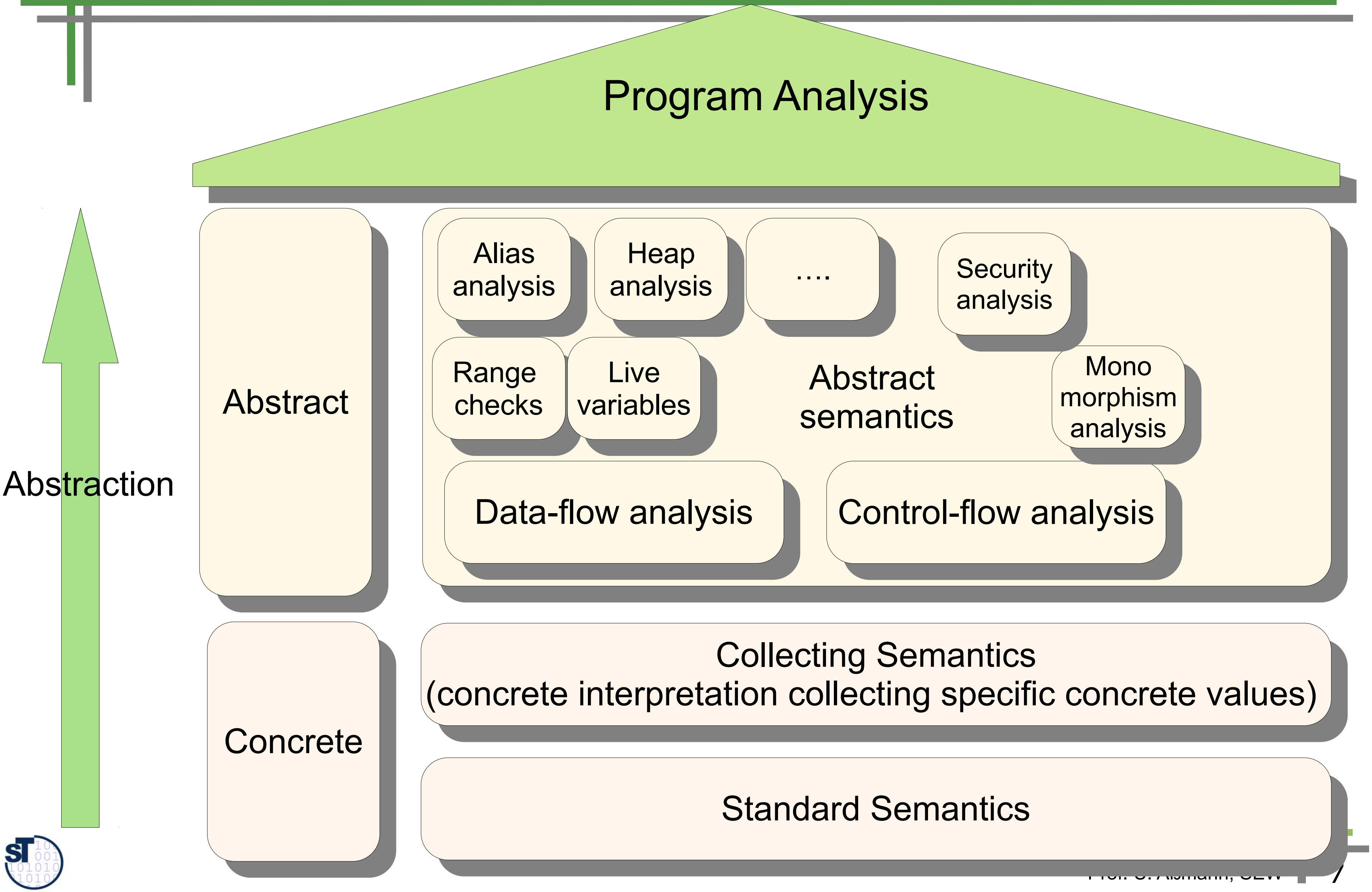
Species: Car
Phaeton

[VW factory]

Interpretation and Semantics of Programs

- Given a fixed set of input values, a program has a *concrete standard semantics.*
  - Denotational semantics (result semantics):
    - The output values
  - Operational semantics:
    - The set of traces of the execution
    - The set of states in the execution traces
  - Axiomatic semantics:
    - The set of all true predicates at each execution point

- A **collecting semantics** selects a subset of interest from the standard semantics, in preparation of the abstract interpretation. The values stay concrete.
- An **abstract interpretation** interprets on the **abstract semantics**, an abstraction of the the collecting semantics



Program Analysis

Abstraction

Abstract

Alias analysis

Heap analysis

….

Security analysis

Range checks

Live variables

Abstract semantics

Mono morphism analysis

Data-flow analysis

Control-flow analysis

Concrete

Collecting Semantics
(concrete interpretation collecting specific concrete values)

Standard Semantics

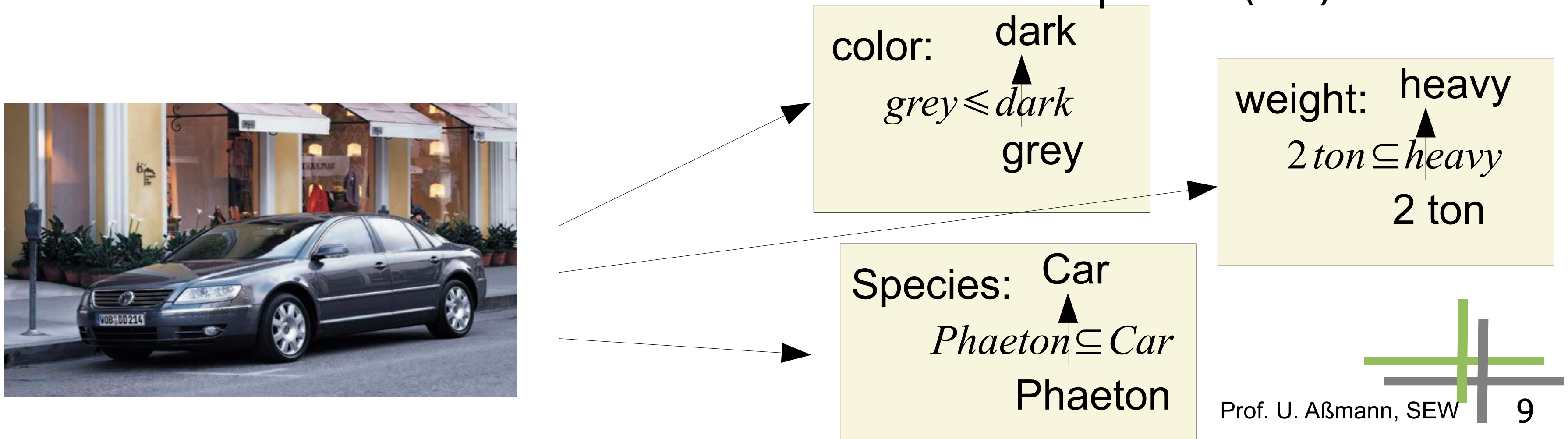
What is a Procedure?

- A procedure is a parameterized code component (code template) for a behavior in a BSL
  - a named lambda abstraction with parameters
  - has a return instruction that returns a return value
  - is a schema for a runtime instance, an activation record
  - a schema for execution traces
  - an abbreviation for code that is called from different reference points (call sites)
- Code templates (parameterized components) can be found in any specification or programming language
  - Z, ..
  - Generic classes in Generic Java
- However, procedures form the component model of the chip, because they can be compiled isomorphically to the chip's instruction pair JUMP-SUBROUTINE-RETURN
- Therefore, procedures are coarse-grain instructions of the chip

Abstract Interpretation

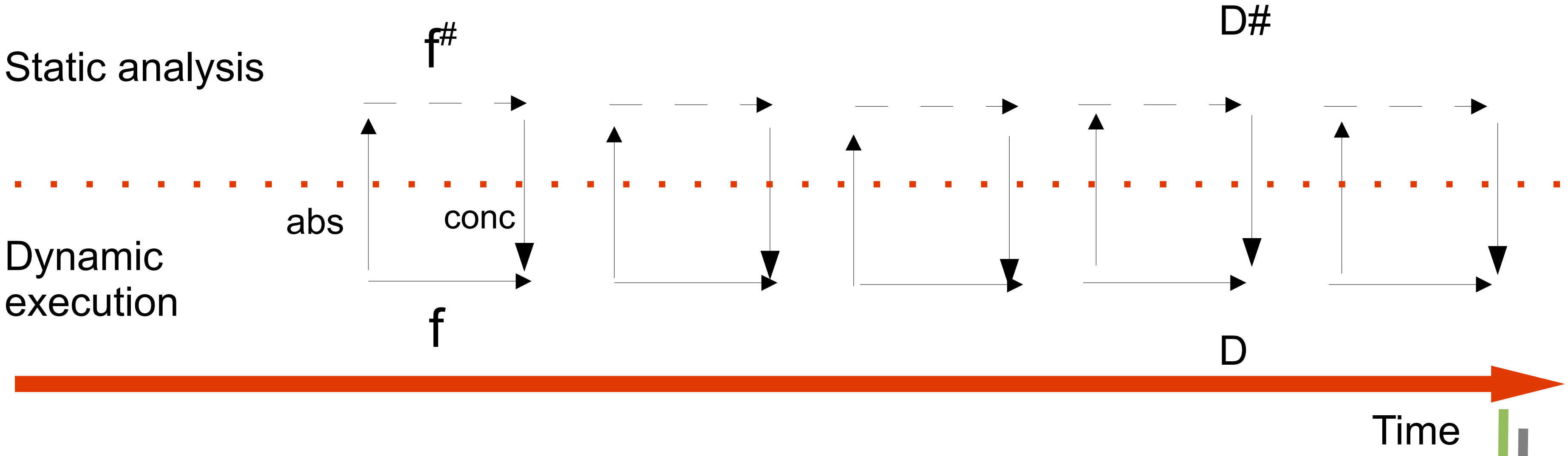
- ***Abstract interpretation*** *is static symbolic execution* of the program with *abstract symbolic* values
  - Since the values cannot be concrete we must abstract them to "easier" values, i.e., simpler domains of *finite* count, height, or breadth
- Values are taken from the *abstract domains* (called 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)

[VW factory]



Functions for Abstract Interpretation

- f: D → D, run-time semantics of the program (**interpreter**)
- abs: D → D#, **abstraction function** from concrete to abstract
- conc: D# → D, **concretization function** from abstract to concrete
- f#:D# → D#, **abstract interpretation function** (abstract semantic function, flow/transfer function)
- f# is like a *shadow* of f



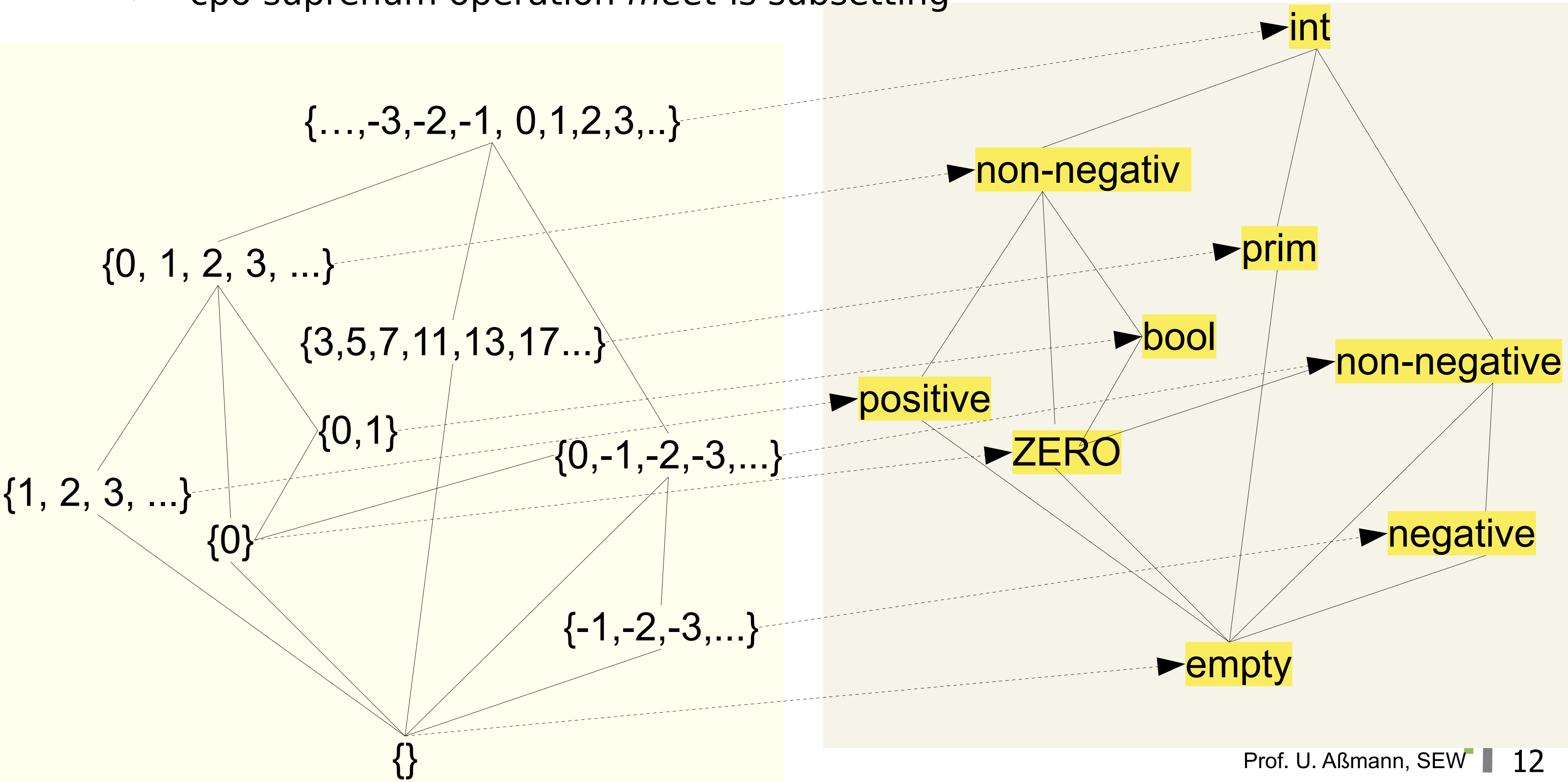
The Iron Law of Abstract Interpretation

- The abstract interpretation must be **correct**, i.e., faithfully abstracting the run-time behavior of the program
- Abs (**abstraction function**), conc (**concretization function**), and f# (**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 ◦ f# ◦ abs
  - The concrete semantics must be a subset of the concretization of the abstract semantics (conservative approximation)
  - The abstract semantic value is 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

$$\begin{tikzcd}
{} \arrow[r, dashed, "f^{\#}"] & {} \arrow[d, "conc"] \\
{} \arrow[u, "abs"] \arrow[r, "f"'] & {}
\end{tikzcd}$$

Ex. Concrete and Abstract Values over int

- Concrete Domain is mapped to abstract domain
- Here: subsets of D=int to symbolic D#=”abstract sets over ints”
- Top means *any-concrete-value*, bottom means *none*
- cpo suprenum operation *meet* is subsetting



Sets of Interpretation Functions

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

$$\{ f_n : L \longrightarrow L \}$$

$$\Leftrightarrow$$

$$f_1 : L \longrightarrow L$$

$$\ldots$$

$$f_k : L \longrightarrow L$$

Ubiquituous A.I.

- Any program in any programming or specification language can be interpreted abstractly, if a collecting semantics is given.
- Examples:
  - A.I. of embedded C programs
  - A.I. of Prolog rule sets
  - A.I. of ECA-rule bases
  - A.I. of state machines (looks like model checking, see later)
  - A.I. of Petri Nets
- Quality analyses:
  - Worst case execution time analysis (WCETA)
  - Worst case energy analysis (WCENA)
  - Security analysis
- Functional analysis
  - Value analysis (“data-flow analysis”)
  - Range check analysis, null check analysis
  - Heap analysis, alias analysis

14.2 Iteration of Abstract Interpreters

Intraprocedural Coincidence Theorem

[Kam/Ullman] Interprocedural 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 a node n (MOP(n))

- Forall n:Node: $MFP(n,f_n) = MOP(n,f_n)$
- Means
  - No matter how the abstract-interpretation functions are iterated, if they stop, 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)

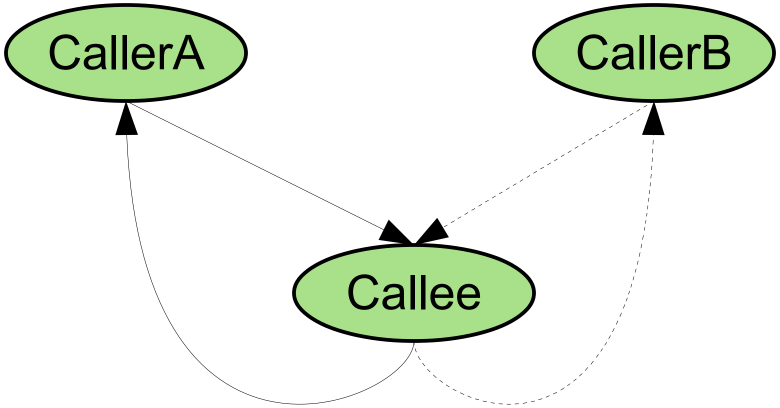
Example: Interpretation with Worklist Algorithms

- Iteration can be done with many strategies
- E.g., iterating *forward* over a worklist that contains “nodes not finished”

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

Interprocedural Control Flow Graphs and Valid Paths

- Flow Functions f# can be on Nodes f#(n), or 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



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