Formal Specification and Verification Techniques

Prof. Dr. K. Madlener

20. November 2008

Course of Studies „Informatics“, „Applied Informatics“ and „Master-Inf.“ WS08/09
Prof. Dr. Madlener
TU- Kaiserslautern

Lecture:
Di 08.15–09.45 13/222 Fr 08.15–09.45 42/110
Exercises:??
Fr. 11.45–13.15 11/201 Mo 11.45–13.15 13/370

- Information `http://www-madlener.informatik.uni-kl.de/teaching/ws2008-2009/fsvt/fsvt.html`
- Evaluation method:
  Exercises (efficiency statement) + Final Exam (Credits)
- First final exam: (Written or Oral)
- Exercises (Dates and Registration): See WWW-Site

Bibliography

M. O'Donnell.
Computing in Systems described by Equations, LNCS 58, 1977.
Equational Logic as a Programming language.

J. Avenhaus.
Reduktionssysteme, (Skript), Springer 1995.

Cohen et.al.
The Specification of Complex Systems.

Bergstra et.al.
Algebraic Specification.

Barendregt.
Functional Programming and Lambda Calculus. Handbook of TCS, 321-363, 1990.

Bibliography

Gehani et.al.
Software Specification Techniques.

Huet.
Confluent Reductions: Abstract Properties and Applications to TRS, JACM, 27, 1980.

Nivat, Reynolds.
Algebraic Methods in Semantics.

Loeckx, Ehrich, Wolf.
Specification of Abstract Data Types, Wyley-Teubner, 1996.

J.W. Klop.
Term Rewriting System. Handbook of Logic, INCS, Vol. 2, Abransky, Gabbay, Maibaum.

Bibliography


Ehrig, Mahr.
Fundamentals of Algebraic Specification.


Peyton-Jones.
The Implementation of Functional Programming Language.


Plasmeister, Eekelen.
Functional Programming and Parallel Graph Rewriting.


Astesiano, Kreowski, Krieg-Brückner.
Algebraic Foundations of Systems Specification (IFIP).


N. Nissanke.
Formal Specification Techniques and Applications (Z, VDM, algebraic), Springer 1999.

Bibliography


Turner, McCluskey.
The construction of formal specifications. (Model based (VDM) + Algebraic (OBJ)).


Goguen, Malcom.
Algebraic Semantics of Imperative Programs.


H. Dörr.
Efficient Graph Rewriting and its Implementation.


B. Potter, J. Sinclair, D. Till.
An introduction to Formal Specification and Z. Prentice Hall, 1996.

Bibliography


J. Woodcok, J. Davis.
Using Z: Specification, Refinement and Proof, Prentice Hall 1996.


J.R. Abrial.
The B-Book; Assigning Programs to Meanings. Cambridge U. Press, 1996.


E. Börger, R. Stärk
Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, 2003.


F. Baader, T. Nipkow
Term Rewriting and All That. Cambridge, 1999.

Goals - Contents

General Goals:

Formal foundations of Methods
for Specification, Verification and Implementation

Summary

- The Role of formal Specifications
- Abstract State Machines: ASM-Specification methods
- Algebraic Specification, Equational Systems
- Reduction systems, Term Rewriting Systems
- Equational - Calculus and - Programming
- Related Calculi: λ-Calculus, Combinator- Calculus
- Implementation, Reduction Strategies, Graph Rewriting

Lecture's Contents

Role of formal Specifications
- Motivation
- Properties of Specifications
- Formal Specifications
- Examples

Abstract State Machines (ASMs)

Abstract State Machines: ASM- Specification's method
- Fundamentals
- Sequential algorithms
- ASM-Specifications

Distributed ASM: Concurrency, reactivity, time
- Fundamentals: Orders, CPO's, proof techniques
- Induction
- DASM
- Reactive and time-depending systems

Refinement
- Lecture Börger's ASM-Buch

Algebraic Specification

Algebraic Specification - Equational Calculus
- Fundamentals
- Introduction
- Algebrae
- Algebraic Fundamentals
- Signature - Terms
- Strictness - Positions- Subterms
- Interpretations: sig-algebras
- Canonical homomorphisms
- Equational specifications
- Substitution
- Loose semantics
- Connection between $\models, =_E, \vdash_E$
- Birkhoff's Theorem

Algebraic Specification: Initial Semantics

Initial semantics
- Basic properties
- Correctness and implementation
- Structuring mechanisms
- Signature morphisms - Parameter passing
- Semantics parameter passing
- Specification morphisms

Algebraic Specification: operationalization

Computability and Implementation

Role of formal Specifications

- Software and hardware systems must accomplish well defined tasks (**requirements**).
- Software Engineering has as goal
  - Definition of criteria for the evaluation of SW-Systems
  - Methods and techniques for the development of SW-Systems, that accomplish such criteria
  - Characterization of SW-Systems
  - Development processes for SW-Systems
  - Measures and Supporting Tools
- Simplified view of a SD-Process:
  Definition of a sequence of actions and descriptions for the SW-System to be developed. Process and Product Models

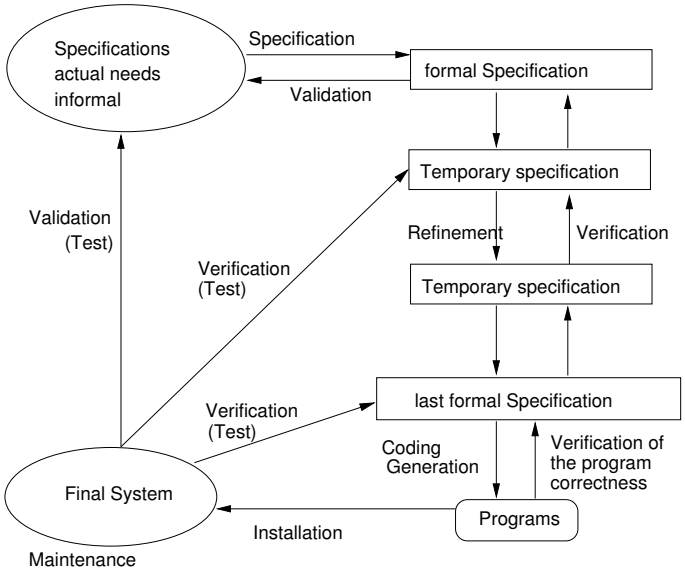
  Goal: The group of documents that includes an executable program.

Models for SW-Development

- Waterfall model, Spiral model,. . .

  Phases $\equiv$ Activities + Product Parts (partial descriptions)
  In each stage of the DP

  Description: a SW specification, that is, a stipulation of what must be achieved, but not always how it is done.



Comment

- First Specification: Global Specification
  Fundament for the Development
  "Contract or Agreement" between Developers and Client
- Intermediate (partial) specifications:
  Base of the Communication between Developers.
- Programs: Final products.

Development paradigms

- Structured Programming
- Design + Program
- Transformation Methods
- ...

Properties of Specifications

Consistency Completeness

- Validation of the global specification regarding the requirements.
- Verification of intermediate specifications regarding the previous one.
- Verification of the programs regarding the specification.
- Verification of the integrated final system with respect to the global specification.
- Activities: Validation, Verification, Testing
  Consistency- and Completeness-Check
- Tool support needed!

Requirements

Functional	-	- non functional
what		time aspects
:		robustness
how		stability
		security
		adaptability
		ergonomics
		maintainability

Properties

Correctness: Does the implemented System fulfill the Requirements?

Test Validate Verify

Validation - Verification

From Wikipedia, the free encyclopedia
In common usage, **validation** is the process of checking if something satisfies a certain criterion. Examples would include checking if a statement is true (validity), if an appliance works as intended, if a computer system is secure, or if computer data are compliant with an open standard. Validation implies one is able to document that a solution or process is correct or is suited for its intended use.
In engineering or as part of a quality management system, **validation** confirms that the needs of an external customer or user of a product, service, or system are met. **Verification** is usually an internal quality process of determining compliance with a regulation, standard, or specification. An easy way of recalling the difference between validation and verification is that
validation is ensuring "you built the right product" and
verification is ensuring "you built the product right."
Validation is testing to confirm that it satisfies user's needs.

Requirements

- The global specification describes, as exact as possible, what must be done.
- **Abstraction of the *how***
  Advantages
  - apriori: Reference document, compact and legible.
  - aposteriori: Possibility to follow and document design decisions $\rightsquigarrow$ **traceability, reusability, maintenance**.
- Problem: Size and complexity of the systems.

  Principles to be supported
  - Refinement principle: Abstraction levels
  - Structuring mechanisms
    Decomposition and modularization principles
  - Object orientation
  - Verification and validation concepts

Requirements Description $\rightsquigarrow$ Specification Language

- Choice of the specification technique depends on the System. Frequently more than a single specification technique is needed. (What – How).
- Type of Systems:
  Pure function oriented (I/O), reactive- embedded- real time-systems.
- Problem : Universal Specification Technique (UST)
  difficult to understand, ambiguities, tools, size ...
  e.g. UML
- Desired: Compact, legible and exact specifications

Here: formal specification techniques

Formal Specifications

- A specification in a formal specification language defines all the possible behaviors of the specified system.
- 3 Aspects: Syntax, Semantics, Inference System
  - Syntax: What's allowed to write: Text with structure, Properties often described by formulas from a logic.
  - Semantics: Which models are associated with the specification, $\rightsquigarrow$ specification models.
  - Inference System: Consequences (Derivation) of properties of the system. $\rightsquigarrow$ Notion of consequence.

Formal Specifications

- Two main classes:

Model oriented	- -	**Property oriented**
(constructive)		(declarative)
e.g.VDM, Z, ASM		signature (functions, predicates)
Construction of a non-ambiguous model from available data structures and construction rules		Properties (formulas, axioms)
Concept of correctness		models algebraic specification AFFIRM, OBJ, ASF,...

- Operational specifications:
  Petri nets, process algebras, automata based (SDL).

Specifications: What for?

- The concept of program correctness is not well defined without a formal specification.
- A verification is not possible without a formal specification.
- Other concepts, like the concept of refinement, simulation become well defined.

Wish List

- Small gap between specification and program:
  Generators, Transformators.
- Not too many different formalisms/notations.
- Tool support.
- Rapid prototyping.
- Rules for "constructing" specifications, that guarantee certain properties (e.g. consistency + completeness).

Formal Specifications

- Advantages:
  - The concepts of correctness, equivalence, completeness, consistency, refinement, composition, etc. are treated in a mathematical way (based on the logic)
  - Tool support is possible and often available
  - The application and interconnection of different tools are possible.
- Disadvantages:

Refinements

Abstraction mechanisms

- Data abstraction (representation)
- Control abstraction (Sequence)
- Procedural abstraction (only I/O description)

Refinement mechanisms

- Choose a data representation (sets by lists)
- Choose a sequence of computation steps
- Develop algorithm (Sorting algorithm)

Concept: Correctness of the implementation

- Observable equivalences
- Behavioral equivalences

Structuring

Problems: Structuring mechanisms

- Horizontal:
  Decomposition/Aggregation/Combination/Extension/
  Parameterization/Instantiation
  (Components)

  Goal: Reduction of complexity, Completeness
- Vertical:
  Realization of Behavior
  Information Hiding/Refinement

  Goal: Efficiency and Correctness

Tool support

- Syntactic support (grammars, parser,...)
- Verification: theorem proving (proof obligations)
- Prototyping (executable specifications)
- Code generation (out of the specifications generate C code)
- Testing (from the specification generate test cases for the program)

Desired:

To generate the tools out of the syntax and semantics of the specification language

Example: declarative

Example 2.1. *Restricted logic: e.g. equational logic*

- *Axioms:* $\forall X\ t_1 = t_2 \qquad t_1, t_2$ *terms.*
- *Rules: Equals are replaced with equals. (directed).*
- *Terms* $\approx$ *names for objects (identifier), structuring, construction of the object.*
- *Abstraction: Terms as elements of an algebra, term algebra.*

Example: declarative

Foundations for the algebraic specification method:

- Axioms induce a congruence on a term algebra
- Independent subtasks
  - Description of properties with equality axioms
  - Representation of the terms
- Operationalization
  - spec, t term give out the „value“ of t, i.e. $t' \in$ Value(spec) with spec $\models t = t'$.
  - $\rightsquigarrow$ Functional programming: LISP, CAML,...
    $F(t_1, \ldots, t_n)$ eval( ) $\rightsquigarrow$ value.

Example: Model-based constructive: VDM

Unambiguous (Unique model), standard (notations),
Independent of the implementation, formally manipulable, abstract,
structured, expressive, consistency by construction

Example 2.2. *Model (state)-based specification technique VDM*

- *Based on naive set theory, PL 1, preconditions and postconditions.*
  Primitive types: $\mathbb{B}$ *Boolean* $\{true, false\}$, $\mathbb{Z}, \mathbb{R}$
  $\mathbb{N}$ *natural* $\{0, 1, 2, 3, \ldots\}$
- *Sets: $\mathbb{B}$-Set: Sets of $\mathbb{B}$-'s.*
- *Operations on sets:* $\in$*: Element, Element-Set* $\rightarrow \mathbb{B}$, $\cup, \cap, \setminus$
- *Sequences: $\mathbb{Z}^*$: Sequences of integer numbers.*
- *Sequence operations:* $\frown$*: Sequences, Sequences* $\rightarrow$ *Sequences.*
  „Concatenation"
  e.g. $[\ ] \frown [true, false, true] = [true, false, true]$
  len: sequences $\rightarrow \mathbb{N}$, *hd: sequences* $\rightsquigarrow$ *elem (partial).*
  tl: sequences $\rightsquigarrow$ *sequences,* *elem: sequences* $\rightarrow$ *Elem-Set.*

Operations in VDM

See e.g.: http://www.vdmportal.org/twiki/bin/view
VDM-SL: System State, Specification of operations

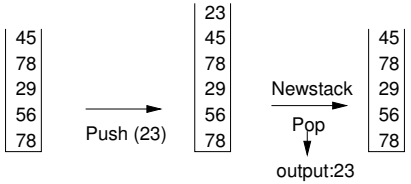
Format:

Operation-Identifier (Input parameters) Output parameters
Pre-Condition
Post-Condition

e.g.
Int_SQR$(x : \mathbb{N}) z : \mathbb{N}$
pre $x \geq 1$
post $(z^2 \leq x) \wedge (x < (z+1)^2)$

Example VDM: Bounded stack

Example 2.3. ▶ *Operations: · Init · Push · Pop · Empty · Full*



Contents $= \mathbb{N}^*$ Max_Stack_Size $= \mathbb{N}$

- STATE STACK OF
  s : Contents
  n : Max_Stack_Size
  inv : mk-STACK$(s, n) \triangleq$ len $s \leq n$
  END

Bounded stack

Init$(size : \mathbb{N})$
ext wr s : Contents
wr n : Max_Stack_Size
pre true
post $s = [\ ] \wedge n = size$

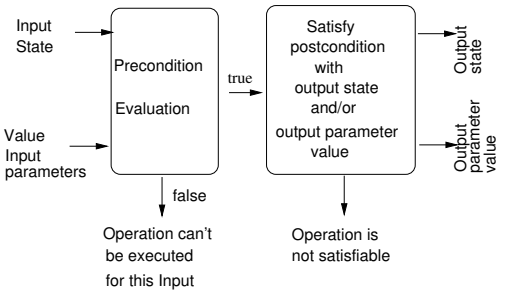
Full$(\ )b : \mathbb{B}$
ext rd s : Contents
rd n : Max_Stack_Size
pre true
post $b \Leftrightarrow (\text{len } s = n)$

Push$(c : \mathbb{N})$
ext wr s : Contens
rd n : Max_Stack_Size
pre len $s < n$
post $s = [c] \frown \overleftarrow{s}$

Pop$(\ )c : \mathbb{N}$
ext wr s : Contens
pre len $s > 0$
post $\overleftarrow{s} = [c] \frown s$

$\rightsquigarrow$ Proof-Obligations

General format for VDM-operations



General form VDM-operations

Proof obligations:
For each acceptable input there's (at least) one acceptable output.

$$\forall s_i, i \cdot (\text{pre-op}(i, s_i) \Rightarrow \exists s_o, o \cdot \text{post-op}(i, s_i, o, s_o))$$

When there are state-invariants at hand:

$$\forall s_i, i \cdot (\text{inv}(s_i) \wedge \text{pre-op}(i, s_i) \Rightarrow \exists s_o, o \cdot (\text{inv}(s_o) \wedge \text{post-op}(i, s_i, o, s_o)))$$

alternatively

$$\forall s_i, i, s_o, o \cdot (\text{inv}(s_i) \wedge \text{pre-op}(i, s_i) \wedge \text{post-op}(i, s_i, o, s_o) \Rightarrow \text{inv}(s_o))$$

See e.g. Turner, McCluskey The Construction of Formal Specifications or Jones C.B. Systematic SW Development using VDM Prentice Hall.

Stack: algebraic specification

Example 2.4. *Elements of an algebraic specification: Signature (sorts, operation names with the arity), Axioms (often only equations)*
SPEC *STACK*
USING *NATURAL, BOOLEAN* *"Names of known SPECs"*
SORT *stack* *"Principal type"*
OPS *init* : $\rightarrow$ *stack* *"Constant of the type stack, empty stack"*
push : *stack nat* $\rightarrow$ *stack*
pop : *stack* $\rightarrow$ *stack*
top : *stack* $\rightarrow$ *nat*
is_empty? : *stack* $\rightarrow$ *bool*
stack_error : $\rightarrow$ *stack*
nat_error : $\rightarrow$ *nat*

(Signature fixed)

Axioms for Stack

FORALL s : stack n : nat
AXIOMS
is_empty? (init) = true
is_empty? (push (s, n)) = false
pop (init) = stack_error
pop (push (s, n)) = s
top (init) = nat_error
top (push (s,n)) = n

Terms or expressions:
top (push (push (init, 2), 3)) "means" 3
How is the "bounded stack" specified algebraically?
Semantics? Operationalization?

Variant: Z and B- Methods: Specification-Development-Programs.

- Covering: Technical specification (what), development through refinement, architecture (layers' architecture), generation of executable code.
- Proofs: Program construction ≡ Proof construction.
  Abstraction, instantiation, decomposition.
- Abstract machines: Encapsulation of information (Modules, Classes, ADT).
- Data and operations: SWS is composed of abstract machines.
  Abstract machines „get " data and „offer" operations.
  Data can only be accessed through operations.

Z- and B- Methods: Specification-Development-Programs.

- Data specification: Sets, relations, functions, sequences, trees. Rules (static) with help of invariants.
- Operator specification: not executable „pseudocode".
  Without loops:
  Precondition + atomic action
  PL1 generalized substitution
- Refinement ($\rightsquigarrow$ implementation).
- Refinement (as specification technique).
- Refinement techniques:
  Elimination of not executable parts, introduction of control structures (cycles).
  Transformation of abstract mathematical structures.

Z- and B- Methods: Specification-Development-Programs.

- Refinement steps: Refinement is done in several steps.
  Abstract machines are newly constructed. Operations for users remain the same, only internal changes.
  In-between steps: Mix code.
- Nested architecture:
  Rule: not too many refinement steps, better apply decomposition.
- Library: Predefined abstract machines, encapsulation of classical DS.
- Reusability
- Code generation: Last abstract machine can be easily translated into a program in an imperative Language.

Z- and B- Methods: Specification-Development-Programs.

Important here:

- Notation: Theory of sets + PL1, standard set operations, Cartesian product, power sets, set restrictions $\{x \mid x \in s \wedge P\}$, P predicate.
- Schemata (Schemes) in Z Models for declaration and constraint {state descriptions}.
- Types.
- Natural Language: Connection Math objects $\rightarrow$ objects of the modeled world.
- See Abrial: The B-Book,
  Potter, Sinclair, Till: An Introduction to Formal Specification and Z,
  Woodcock, Davis: Using Z Specification, Refinement, and Proof $\rightsquigarrow$ Literature

Introduction to ASM: Fundamentals

Adaptable and flexible specification's technique

Modeling in the correct abstraction level

Natural and easy understandable semantics.

Material: See http://www.di.unipi.it/AsmBook/

Theoretical fundaments: ASM Theses

Abstract state machines as computation models

Turing Machines (RAM, part.rec. Fct,..) serve as computation model, e.g. fixing the notion of computable functions. In principle is possible to simulate every algorithmic solution with an appropriate TM.

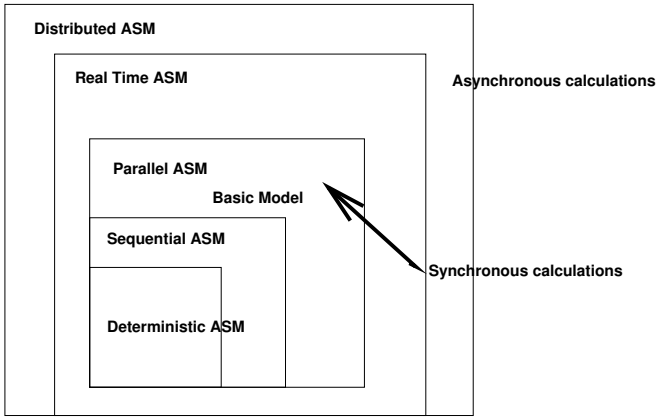
Problem: Simulation is not easy, because there are different abstraction levels of the manipulated objects and different granularity of the steps.

Question: Is it possible to generalize the TM in such a way that every algorithm, independent from it's abstraction level, can be naturally and faithfully simulated with such generalized machine?
How would the states and instructions of such a machine look like?

Easy: If Condition Then Action

ASM Thesis

ASM Thesis The concept of abstract state machine provides a universal computation model with the ability to simulate arbitrary algorithms on their natural levels of abstraction. Yuri Gurevich



Sequential ASM Thesis

- The model of the sequential ASM's is universal for all the sequential algorithms.
- Each sequential algorithm, independent from his abstraction level, can be simulated step by step by a sequential ASM.

To confirm this thesis we need definitions for sequential algorithms and for sequential ASM's.

$\rightsquigarrow$ Postulates for sequentiality

Sequentiality Postulates

- Sequential time:
  Computations are linearly arranged.
- Abstract states:
  Each kind of static mathematical reality can be represented by a structure of the first order logic (PL 1). (Tarski)
- Bounded exploration:
  Each computation step depends only on a finite (depending only on the algorithm) bounded state information.

Y. Gurevich:: Sequential Abstract State Machines Capture Sequential Algorithms, ACM Transactions on Computational Logic, 1, 2000, 77-111.

The postulates in detail: Sequential time

Let A be a sequential algorithm. To A belongs:

- A set (Set of states) $S(A)$ of States of A.
- A subset $I(A)$ of $S(A)$ which elements are called initial states of A.
- A mapping $\tau_A : S(A) \to S(A)$, the one-step-function of A.

An run (or a computation) of A is a finite or infinite sequence of states of A

$$X_0, X_1, X_2, \ldots$$

in which X_0 is an initial state and $\tau_A(X_i) = X_{i+1}$ holds for each i.

Logical time and not physical time.

Abstract States

Definition 3.1 (Equivalent algorithms). *Algorithms A and B are equivalent if $S(A) = S(B)$, $I(A) = I(B)$ and $\tau_A = \tau_B$.*
In particular equivalent algorithms have the same runs.

Let A be a sequential algorithm:

- States of A are first order (PL1) structures.
- All the states of A have the same vocabulary (signature).
- The one-step-function doesn't change the base set (universe) $B(X)$ of a state.
- $S(A)$ and $I(A)$ are closed under isomorphisms and each isomorphism from state X to state Y is also an isomorphism of state $\tau_A(X)$ to $\tau_A(Y)$.

Exercises

States: Signatures, interpretations, universe, terms, ground terms, value ...
Signatures (vocabulary): function- and relation-names, arity ($n \geq 0$)

Assumption: *true*, *false*, *undef* (constants), *Boole* (monadic) and $=$ are contained in every signature.
The interpretation of *true* is different from the one for *false*, *undef*.
Relations are considered as functions with the value of *true*, *false* in the interpretations.
Monadic relations are seen as subsets of the base set of the interpretations.
Let $Val(t, X)$ be the value in state X for a ground term t that is in the vocabulary.
Functions are divided in dynamic and static, according whether they can change or not, when a state transition occurs.
Exercise: Model the states of a TM as an abstract state.
Model the states of the standard Euclidean algorithm.

Bounded exploration

- Unbounded-Parallelism: Consider the following graph-reachability algorithm that iterates the following step. (It is assumed that at the beginning only one node satisfies the unary relation R.)

  do for all x, y with $Edge(x, y) \wedge R(x) \wedge \neg R(y)$ $\quad R(y) := true$

  In each computation step an unbounded number of local changes is made on a global state.
- Unbounded-Step-Information:
  Test for isolated nodes in a graph:

  if $\forall x \exists y\ Edge(x, y)$ then Output := false else Output := true

  In one step only bounded local changes are made, though an unbounded part of the state is considered in one step.
  How can these properties be formalized?$\rightsquigarrow$ Atomic actions

Update sets

Consider the structure X as memory:

If f is a function name of arity j and $\overline{a}$ a j-tuple of base elements from X, then the pair $(f, \overline{a})$ is called a location and $Content_X(f, \overline{a})$ is the value of the interpretation of f for $\overline{a}$ in X.

Is $(f, \overline{a})$ a location of X and b an element of X, then $(f, \overline{a}, b)$ is called an update of X. The update is trivial when $b = Content_X(f, \overline{a})$.

To make (fire) an update, the actual content of the location is replaced by b.

A set of updates of X is consistent when in the set there is no pair of updates with the same location and different values.
A set Δ of updates is executed by making all updates in the set simultaneously (in case the set is consistent, in other case nothing is done).
The result is denoted by $X + \Delta$.

Update sets of algorithms, Reachable elements

Lemma 3.2. *If X, Y are structures over the same signature and with the same base set, then there is a unique consistent set Δ of non-trivial updates of X with $Y = X + \Delta$. Let $\Delta \leftrightharpoons Y - X$.*

Definition 3.3. *Let X be a state of algorithm A. According to the definition, X and $\tau_A(X)$ have the same signature and base set. Set:*

$$\Delta(A, X) \leftrightharpoons \tau_A(X) - X \quad \text{i.e. } \tau_A(X) = X + \Delta(A, X)$$

How can we bring up the elements of the base set in the description of the algorithm at all? $\rightsquigarrow$ Using the ground terms of the signature.

Definition 3.4 (Reachable element)**.** *An element a of a structure X is reachable when $a = Val(t, X)$ for a ground term t in the vocabulary of X. A location $(f, \overline{a})$ of X is reachable when each element in the tuple $\overline{a}$ is reachable.*
An update $(f, \overline{a}, b)$ of X is reachable when $(f, \overline{a})$ and b are reachable.

Bounded exploration postulate

Two structures X and Y with the same vocabulary Sig coincide on a set T of Sig- terms, when $Val(t, X) = Val(t, Y)$ for all $t \in T$. The vocabulary (signature) of an algorithm is the vocabulary of his states.

Let A be a sequential algorithm.

- There exist a finite set T of terms in the vocabulary of A, so that: $\Delta(A, X) = \Delta(A, Y)$, for all states X, Y of A, that coincide on T.

Intuition: Algorithm A examines only the part of a state that is reachable with the set of terms T. If two states coincide on this term-set, then the update-sets of the algorithm for both states should be the same.

The set T is a bounded-exploration witness for A.

Example

Example 3.5. *Consider algorithm A:*

$$\textit{if } P(f) \textit{ then } f := S(f)$$

States with interpretations with base set $\mathbb{N}$, P subset of the natural numbers, for S the successor function and f a constant.

Evidently A fulfills the postulates of sequential time and abstract states.

One could believe that
$T_0 = \{f, P(f), S(f)\}$ is a bounded-exploration witness for A.

Example: Continued

Let X be the canonical state of A with $f = 0$ and $P(0)$ holding.

Set $a \leftrightharpoons Val(true, X)$ and $b \leftrightharpoons Val(false, X)$, so that

$$Val(P(0), X) = Val(true, X) = a.$$

Let Y be the state that is obtained out of X through reinterpretation of *true* as b and *false* as a, i.e. $Val(true, Y) = b$ and $Val(false, Y) = a$.
The values of f and $P(0)$ are left unchanged:

$Val(P(0), Y) = a$, thus $P(0)$ is not valid in Y.

Consequently X, Y coincide on T_0 but $\Delta(A, X) \neq \emptyset = \Delta(A, Y)$.

The set $T = T_0 \cup \{true\}$ is a bounded-exploration witness for A.

Sequential algorithms

Definition 3.6 (Sequential algorithm)**.** *A sequential algorithm is an object A, which fulfills the three postulates.*
In particular A has a vocabulary and a bounded-exploration witness T. Without loss of generality (w.l.o.g.) T is subterm-closed and contains $true, false, undef$. The terms of T are called critical and their interpretations in a state X are called critical values in X.

Lemma 3.7. *If $(f, a_1, ..., a_j, a_0)$ is an update in $\Delta(A, X)$, then all the elements $a_0, a_1, ..., a_j$ are critical values in X.*

Proof: exercise (Proof by contradiction).
The set of the critical terms does not depend of X, thus there is a fixed upper bound for the size of $\Delta(A, X)$ and A changes in every step a bounded number of locations. Each one of the updates in $\Delta(A, X)$ is an atomic action of A. I.e. $\Delta(A, X)$ is a bounded set of atomic actions of A.

Sequential ASM-programs: Update rules

Definition 3.8 (Update rule)**.** *An update rule over the signature Sig has the form*

$$f(t_1, ..., t_j) := t_0$$

in which f is a function and t_i are (ground) terms in Sig. To fire the rule in the Sig-structure X, compute the values $a_i = Val(t_i, X)$ and execute update $((f, a_1, ..., a_j), a_0)$ over X.
Parallel update rule over Sig: Let R_i be update rules over Sig, then

par
  R_1
  R_2
  .
  .
  .
  R_k
endpar

Notation: Block (when empty skip)

fires through simultaneously firing of R_i.

Sequential ASM-programs

Definition 3.9 (Semantics of update rules). *If R is an update rule $f(t_1, ..., t_j) := t_0$ and $a_i = Val(t_i, X)$ then set*
$\Delta(R, X) \leftrightharpoons \{(f, (a_1, ..., a_j), a_0)\}$

If R is a par-update rule with components $R_1, ...R_k$ then set
$\Delta(R, X) \leftrightharpoons \Delta(R1, X) \cup \cdots \cup \Delta(Rk, X)$.

Consequence 3.10. *There exists in particular for each state X a rule R^X that uses only critical terms with* $\Delta(R^X, X) = \Delta(A, X)$.

Notice: If X, Y coincide on the critical terms, then $\Delta(R^X, Y) = \Delta(A, Y)$ holds. If X, Y are states and $\Delta(R^X, Z) = \Delta(A, Z)$ for a state Z, that is isomorphic to Y, then also $\Delta(R^X, Y) = \Delta(A, Y)$ holds.
Consider the equivalence relation $E_X(t1, t2) \leftrightharpoons Val(t1, X) = Val(t2, X)$ on T.
X, Y are T-similar, when $E_X = E_Y \rightsquigarrow \Delta(R^X, Y) = \Delta(A, Y)$. Exercise

Sequential ASM-programs

Definition 3.11. *Let φ be a boolean term over Sig and R_1, R_2 rules over Sig, then*

if φ then R_1
else $R2$
endif *is a rule*

Semantic:: To fire the rule in state X evaluate φ in X. If the result is true, then $\Delta(R, X) = \Delta(R_1, X)$, if not $\Delta(R, X) = \Delta(R_2, X)$.

Definition 3.12 (Sequential ASM program). *A sequential ASM program Π over the signature Sig is a rule over Sig. According to this $\Delta(\Pi, X)$ is well defined for each Sig-structure X. Let $\tau_\Pi(X) \leftrightharpoons X + \Delta(\Pi, X)$.*

Lemma 3.13. ***Basic result:*** *For each sequential algorithm A over Sig there's a sequential ASM-programm Π over Sig with $\Delta(\Pi, X) = \Delta(A, X)$ for all the states X of A.*

Sequential ASM-machines

Definition 3.14 (A sequential abstract-state-machine (seq-ASM)). *A seq-ASM B over the signature Σ is given through:*

- *A sequential ASM-programm Π over Σ.*
- *A set $S(B)$ of interpretations of Σ that is closed under isomorphisms and under the mapping τ_Π .*
- *A subset $I(B) \subset S(B)$, that is closed under isomorphisms.*

Theorem 3.15. *For each sequential algorithm A there is an equivalent sequential ASM.*

Example

Example 3.16. *Maximal interval-sum.[Gries 1990]. Let A be a function from $\{0, 1, ..., n-1\} \to \mathbb{R}$ and $i, j, k \in \{0, 1, ..., n\}$.*
For $i \leq j$: $S(i, j) \leftrightharpoons \sum_{i \leq k < j} A(k)$. In particular $S(i, i) = 0$.

Problem: *Compute* $S \leftrightharpoons max_{i \leq j} S(i, j)$.

Define $y(k) \leftrightharpoons max_{i \leq j \leq k} S(i, j)$. Then $y(0) = 0, y(n) = S$ and

$y(k+1) = max\{max_{i \leq j \leq k} S(i, j), max_{i \leq k+1} S(i, k+1)\} = max\{y(k), x(k+1)\}$

where $x(k) \leftrightharpoons max_{i \leq k} S(i, k)$, thus $x(0) = 0$ and

$$\begin{aligned} x(k+1) &= max\{max_{i \leq k} S(i, k+1), S(k+1, k+1)\} \\ &= max\{max_{i \leq k}(S(i, k) + A(k)), 0\} \\ &= max\{(max_{i \leq k} S(i, k)) + A(k), 0\} \\ &= max\{x(k) + A(k), 0\} \end{aligned}$$

Continuation of the example

Due to $y(k) \geq 0$, we have

$$y(k+1) = max\{y(k), x(k+1)\} = max\{y(k), x(k) + A(k)\}$$

Assumption: The 0-ary dynamic functions k, x, y are 0 in the initial state. The required algorithm is then

if $k \neq n$ *then*
  par
    $x := max\{x + A(k), 0\}$
    $y := max\{y, x + A(k)\}$
    $k := k + 1$
else $S := y$

Exercise 3.17. *Simulation*
Define an ASM, that implements Markov's Normal-algorithms.
e.g. for $ab \rightarrow A,\ ba \rightarrow B,\ c \rightarrow C$

Detailed definition of ASMs

- Part 1: Abstract states and update sets
- Part 2: Mathematical Logic
- Part 3: Transition rules and runs of ASMs
- Part 4: The reserve of ASMs

Part 1

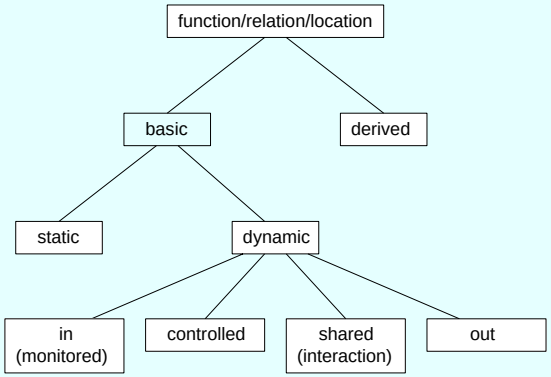
Abstract states and update sets

Signatures

Definition. A *signature* Σ is a finite collection of function names.
- Each function name f has an *arity*, a non-negative integer.
- Nullary function names are called *constants*.
- Function names can be *static* or *dynamic*.
- Every ASM signature contains the static constants $undef$, $true$, $false$.

Signatures are also called *vocabularies*.

Classification of functions



States

Definition. A *state* $\mathfrak{A}$ for the signature Σ is a non-empty set X, the *superuniverse* of $\mathfrak{A}$, together with an *interpretation* $f^{\mathfrak{A}}$ of each function name f of Σ.

- If f is an n-ary function name of Σ, then $f^{\mathfrak{A}}\colon X^n \to X$.
- If c is a constant of Σ, then $c^{\mathfrak{A}} \in X$.
- The superuniverse X of the state $\mathfrak{A}$ is denoted by $|\mathfrak{A}|$.

- The superuniverse is also called the *base set* of the state.
- The *elements* of a state are the elements of the superuniverse.

States (continued)

- The interpretations of $undef$, $true$, $false$ are pairwise different.
- The constant $undef$ represents an undetermined object.
- The *domain* of an n-ary function name f in $\mathfrak{A}$ is the set of all n-tuples $(a_1, \ldots, a_n) \in |\mathfrak{A}|^n$ such that $f^{\mathfrak{A}}(a_1, \ldots, a_n) \neq undef^{\mathfrak{A}}$.
- A *relation* is a function that has the values $true$, $false$ or $undef$.
- We write $a \in R$ as an abbreviation for $R(a) = true$.
- The superuniverse can be divided into *subuniverses* represented by unary relations.

Locations

Definition. A *location* of $\mathfrak{A}$ is a pair

$$(f, (a_1, \ldots, a_n))$$

where f is an n-ary function name and $a_1, \ldots, a_n$ are elements of $\mathfrak{A}$.

- The value $f^{\mathfrak{A}}(a_1, \ldots, a_n)$ is the *content* of the location in $\mathfrak{A}$.
- The *elements* of the location are the elements of the set $\{a_1, \ldots, a_n\}$.
- We write $\mathfrak{A}(l)$ for the content of the location l in $\mathfrak{A}$.

Notation. If $l = (f, (a_1, \ldots, a_n))$ is a location of $\mathfrak{A}$ and α is a function defined on $|\mathfrak{A}|$, then $\alpha(l) = (f, (\alpha(a_1), \ldots, \alpha(a_n)))$.

Updates and update sets

Definition. An *update* for $\mathfrak{A}$ is a pair (l, v), where l is a location of $\mathfrak{A}$ and v is an element of $\mathfrak{A}$.
- The update is *trivial*, if $v = \mathfrak{A}(l)$.
- An *update set* is a set of updates.

Definition. An update set U is *consistent*, if it has no clashing updates, i.e., if for any location l and all elements v, w, if $(l, v) \in U$ and $(l, w) \in U$, then $v = w$.

Firing of updates

Definition. The result of *firing* a consistent update set U in a state $\mathfrak{A}$ is a new state $\mathfrak{A} + U$ with the same superuniverse as $\mathfrak{A}$ such that for every location l of $\mathfrak{A}$:

$$(\mathfrak{A} + U)(l) = \begin{cases} v, & \text{if } (l, v) \in U; \\ \mathfrak{A}(l), & \text{if there is no } v \text{ with } (l, v) \in U. \end{cases}$$

The state $\mathfrak{A} + U$ is called the *sequel* of $\mathfrak{A}$ with respect to U.

Homomorphisms and isomorphisms

Let $\mathfrak{A}$ and $\mathfrak{B}$ be two states over the same signature.

Definition. A *homomorphism* from $\mathfrak{A}$ to $\mathfrak{B}$ is a function α from $|\mathfrak{A}|$ into $|\mathfrak{B}|$ such that $\alpha(\mathfrak{A}(l)) = \mathfrak{B}(\alpha(l))$ for each location l of $\mathfrak{A}$.

Definition. An *isomorphism* from $\mathfrak{A}$ to $\mathfrak{B}$ is a homomorphism from $\mathfrak{A}$ to $\mathfrak{B}$ which is a ono-to-one function from $|\mathfrak{A}|$ onto $|\mathfrak{B}|$.

Lemma (Isomorphism). Let α be an isomorphism from $\mathfrak{A}$ to $\mathfrak{B}$. If U is a consistent update set for $\mathfrak{A}$, then $\alpha(U)$ is a consistent update set for $\mathfrak{B}$ and α is an isomorphism from $\mathfrak{A}+U$ to $\mathfrak{B}+\alpha(U)$.

Composition of update sets

$$U \oplus V = V \cup \{(l, v) \in U \mid \text{there is no } w \text{ with } (l, w) \in V\}$$

Lemma. Let U, V, W be update sets.
- $(U \oplus V) \oplus W = U \oplus (V \oplus W)$
- If U and V are consistent, then $U \oplus V$ is consistent.
- If U and V are consistent, then $\mathfrak{A}+(U \oplus V) = (\mathfrak{A}+U)+V$.

Part 2

Mathematical Logic

Terms

Let Σ be a signature.

Definition. The *terms* of Σ are syntactic expressions generated as follows:
- Variables x, y, z, ... are terms.
- Constants c of Σ are terms.
- If f is an n-ary function name of Σ, $n > 0$, and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

- A term which does not contain variables is called a *ground term*.
- A term is called *static*, if it contains static function names only.
- By $t\frac{s}{x}$ we denote the result of replacing the variable x in term t everywhere by the term s (*substitution* of s for x in t).

Variable assignments

Let $\mathfrak{A}$ be a state.

Definition. A *variable assignment* for $\mathfrak{A}$ is a finite function ζ which assigns elements of $|\mathfrak{A}|$ to a finite number of variables.

- We write $\zeta[x \mapsto a]$ for the variable assignment which coincides with ζ except that it assigns the element a to the variable x:

$$\zeta[x \mapsto a](y) = \begin{cases} a, & \text{if } y = x; \\ \zeta(y), & \text{otherwise.} \end{cases}$$

- Variable assignments are also called *environments*.

Evaluation of terms

Definition. Let $\mathfrak{A}$ be a state of Σ.
Let ζ be a variable assignment for $\mathfrak{A}$.
Let t be a term of Σ such that all variables of t are defined in ζ.
The *value* $[\![t]\!]^{\mathfrak{A}}_{\zeta}$ is defined as follows:
- $[\![x]\!]^{\mathfrak{A}}_{\zeta} = \zeta(x)$
- $[\![c]\!]^{\mathfrak{A}}_{\zeta} = c^{\mathfrak{A}}$
- $[\![f(t_1, \ldots, t_n)]\!]^{\mathfrak{A}}_{\zeta} = f^{\mathfrak{A}}([\![t_1]\!]^{\mathfrak{A}}_{\zeta}, \ldots, [\![t_n]\!]^{\mathfrak{A}}_{\zeta})$

Evaluation of terms (continued)

Lemma (Coincidence). If ζ and η are two variable assignments for t such that $\zeta(x) = \eta(x)$ for all variables x of t, then $[\![t]\!]^{\mathfrak{A}}_{\zeta} = [\![t]\!]^{\mathfrak{A}}_{\eta}$.

Lemma (Homomorphism). If α is a homomorphism from $\mathfrak{A}$ to $\mathfrak{B}$, then $\alpha([\![t]\!]^{\mathfrak{A}}_{\zeta}) = [\![t]\!]^{\mathfrak{B}}_{\alpha\circ\zeta}$ for each term t.

Lemma (Substitution). Let $a = [\![s]\!]^{\mathfrak{A}}_{\zeta}$.
Then $[\![t\frac{s}{x}]\!]^{\mathfrak{A}}_{\zeta} = [\![t]\!]^{\mathfrak{A}}_{\zeta[x\mapsto a]}$.

Formulas

Let Σ be a signature.

Definition. The *formulas* of Σ are generated as follows:
- If s and t are terms of Σ, then $s = t$ is a formula.
- If φ is a formula, then $\neg\varphi$ is a formula.
- If φ and ψ are formulas, then $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ and $(\varphi \rightarrow \psi)$ are formulas.
- If φ is a formula and x a variable, then $(\forall x\, \varphi)$ and $(\exists x\, \varphi)$ are formulas.

- A formula $s = t$ is called an *equation*.
- The expression $s \neq t$ is an abbreviation for $\neg(s = t)$.

Formulas (continued)

symbol	name	meaning
$\neg$	negation	not
$\wedge$	conjunction	and
$\vee$	disjunction	or (inclusive)
$\rightarrow$	implication	if-then
$\forall$	universal quantification	for all
$\exists$	existential quantification	there is

Formulas (continued)

$\varphi \wedge \psi \wedge \chi$ stands for $((\varphi \wedge \psi) \wedge \chi)$,
$\varphi \vee \psi \vee \chi$ stands for $((\varphi \vee \psi) \vee \chi)$,
$\varphi \wedge \psi \rightarrow \chi$ stands for $((\varphi \wedge \psi) \rightarrow \chi)$, etc.

- The variable x is *bound* by the quantifier $\forall$ ($\exists$) in $\forall x\, \varphi$ ($\exists x\, \varphi$).
- The *scope* of x in $\forall x\, \varphi$ ($\exists x\, \varphi$) is the formula φ.
- A variable x occurs *free* in a formula, if it is not in the scope of a quantifier $\forall x$ or $\exists x$.
- By $\varphi\frac{t}{x}$ we denote the result of replacing all free occurrences of the variable x in φ by the term t. (Bound variables are renamed.)

Semantics of formulas

$$[\![s = t]\!]^{\mathfrak{A}}_{\zeta} = \begin{cases} true, & \text{if } [\![s]\!]^{\mathfrak{A}}_{\zeta} = [\![t]\!]^{\mathfrak{A}}_{\zeta}; \\ false, & \text{otherwise.} \end{cases}$$

$$[\![\neg\varphi]\!]^{\mathfrak{A}}_{\zeta} = \begin{cases} true, & \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = false; \\ false, & \text{otherwise.} \end{cases}$$

$$[\![\varphi \wedge \psi]\!]^{\mathfrak{A}}_{\zeta} = \begin{cases} true, & \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = true \text{ and } [\![\psi]\!]^{\mathfrak{A}}_{\zeta} = true; \\ false, & \text{otherwise.} \end{cases}$$

$$[\![\varphi \vee \psi]\!]^{\mathfrak{A}}_{\zeta} = \begin{cases} true, & \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = true \text{ or } [\![\psi]\!]^{\mathfrak{A}}_{\zeta} = true; \\ false, & \text{otherwise.} \end{cases}$$

$$[\![\varphi \rightarrow \psi]\!]^{\mathfrak{A}}_{\zeta} = \begin{cases} true, & \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = false \text{ or } [\![\psi]\!]^{\mathfrak{A}}_{\zeta} = true; \\ false, & \text{otherwise.} \end{cases}$$

$$[\![\forall x\, \varphi]\!]^{\mathfrak{A}}_{\zeta} = \begin{cases} true, & \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta[x \mapsto a]} = true \text{ for every } a \in |\mathfrak{A}|; \\ false, & \text{otherwise.} \end{cases}$$

$$[\![\exists x\, \varphi]\!]^{\mathfrak{A}}_{\zeta} = \begin{cases} true, & \text{if there exists an } a \in |\mathfrak{A}| \text{ with } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta[x \mapsto a]} = true; \\ false, & \text{otherwise.} \end{cases}$$

Coincidence, Substitution, Isomorphism

Lemma (Coincidence). If ζ and η are two variable assignments for φ such that $\zeta(x) = \eta(x)$ for all free variables x of φ, then $[\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = [\![\varphi]\!]^{\mathfrak{A}}_{\eta}$.

Lemma (Substitution). Let t be a term and $a = [\![t]\!]^{\mathfrak{A}}_{\zeta}$. Then $[\![\varphi\frac{t}{x}]\!]^{\mathfrak{A}}_{\zeta} = [\![\varphi]\!]^{\mathfrak{A}}_{\zeta[x \mapsto a]}$.

Lemma (Isomorphism). Let α be an isomorphism from $\mathfrak{A}$ to $\mathfrak{B}$. Then $[\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = [\![\varphi]\!]^{\mathfrak{B}}_{\alpha \circ \zeta}$.

Models

Definition. A state $\mathfrak{A}$ is a *model* of φ (written $\mathfrak{A} \models \varphi$), if $[\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = true$ for all variable assignments ζ for φ.

Part 3

Transition rules and runs of ASMs

Transition rules

Skip Rule: **skip**

Meaning: Do nothing

Update Rule: $f(s_1, \ldots, s_n) := t$

Meaning: Update the value of f at $(s_1, \ldots, s_n)$ to t.

Block Rule: P **par** Q

Meaning: P and Q are executed in parallel.

Conditional Rule: **if** φ **then** P **else** Q

Meaning: If φ is true, then execute P, otherwise execute Q.

Let Rule: **let** $x = t$ **in** P

Meaning: Assign the value of t to x and then execute P.

Transition rules (continued)

Forall Rule: **forall** x **with** φ **do** P

Meaning: Execute P in parallel for each x satisfying φ.

Choose Rule: **choose** x **with** φ **do** P

Meaning: Choose an x satisfying φ and then execute P.

Sequence Rule: P **seq** Q

Meaning: P and Q are executed sequentially, first P and then Q.

Call Rule: $r(t_1, \ldots, t_n)$

Meaning: Call transition rule r with parameters $t_1, \ldots, t_n$.

Variations of the syntax

if φ **then**
P
else
Q
endif
	if φ **then** P **else** Q

[**do in-parallel**] P_1 $\vdots$ P_n [**enddo**]	P_1 **par** ... **par** P_n
$\{P_1, \ldots, P_n\}$	P_1 **par** ... **par** P_n

Variations of the syntax (continued)

do forall x: φ P **enddo**	**forall** x **with** φ **do** P
choose x: φ P **endchoose**	**choose** x **with** φ **do** P
step P **step** Q	P **seq** Q

Example

Example 3.18. *Sorting of linear data structures in-place, one-swap-a-time.*
Let $a : Index \to Value$

$$\begin{array}{l} \textit{choose} \quad x, y \in \textit{Index} : x < y \wedge a(x) > a(y) \\ \quad \textit{do} \quad \textit{in} - \textit{parallel} \\ \qquad a(x) := a(y) \\ \qquad a(y) := a(x) \end{array}$$

Two kinds of non-determinisms:

"Don't-care" non-determinism: random choice

$$\begin{array}{l} \textit{choose} \quad x \in \{x_1, x_2, ..., x_n\} \ \textit{with} \ \ \varphi(x) \ \ \textit{do} \\ \qquad R(x) \end{array}$$

"Don't-know" indeterminism

Extern controlled actions and events (e.g. input actions)

$$\textit{monitored} \quad f : X \to Y$$

Free and bound variables

Definition. An occurrence of a variable x is *free* in a transition rule, if it is not in the scope of a **let** x, **forall** x or **choose** x.

let $x = t$ **in** P
scope of x

forall x **with** φ **do** P
scope of x

choose x **with** φ **do** P
scope of x

Rule declarations

Definition. A *rule declaration* for a rule name r of arity n is an expression

$$r(x_1, \ldots, x_n) = P$$

where

- P is a transition rule and
- the free variables of P are contained in the list $x_1, \ldots, x_n$.

Remark: Recursive rule declarations are allowed.

Abstract State Machines

Definition. An *abstract state machine* M consists of

- a signature Σ,
- a set of initial states for Σ,
- a set of rule declarations,
- a distinguished rule name of arity zero called the *main rule name* of the machine.

Semantics of transition rules

The semantics of transition rules is defined in a calculus by rules:

$$\frac{Premise_1 \ \cdots \ Premise_n}{Conclusion} \ Condition$$

The predicate

$$\mathsf{yields}(P, \mathfrak{A}, \zeta, U)$$

means:

The transition rule P yields the update set U in state $\mathfrak{A}$ under the variable assignment ζ.

Semantics of transition rules (continued)

$$\frac{}{\mathsf{yields}(\textbf{skip}, \mathfrak{A}, \zeta, \emptyset)}$$

$$\frac{}{\mathsf{yields}(f(s_1, \ldots, s_n) := t, \mathfrak{A}, \zeta, \{(l, v)\})} \quad \text{where } l = (f, ([\![s_1]\!]^{\mathfrak{A}}_{\zeta}, \ldots, [\![s_n]\!]^{\mathfrak{A}}_{\zeta})) \text{ and } v = [\![t]\!]^{\mathfrak{A}}_{\zeta}$$

$$\frac{\mathsf{yields}(P, \mathfrak{A}, \zeta, U) \quad \mathsf{yields}(Q, \mathfrak{A}, \zeta, V)}{\mathsf{yields}(P \ \textbf{par} \ Q, \mathfrak{A}, \zeta, U \cup V)}$$

$$\frac{\mathsf{yields}(P, \mathfrak{A}, \zeta, U)}{\mathsf{yields}(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q, \mathfrak{A}, \zeta, U)} \quad \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = true$$

$$\frac{\mathsf{yields}(Q, \mathfrak{A}, \zeta, V)}{\mathsf{yields}(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q, \mathfrak{A}, \zeta, V)} \quad \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = false$$

$$\frac{\mathsf{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U)}{\mathsf{yields}(\textbf{let } x = t \textbf{ in } P, \mathfrak{A}, \zeta, U)} \quad \text{where } a = [\![t]\!]^{\mathfrak{A}}_{\zeta}$$

$$\frac{\mathsf{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U_a) \quad \text{for each } a \in I}{\mathsf{yields}(\textbf{forall } x \textbf{ with } \varphi \textbf{ do } P, \mathfrak{A}, \zeta, \bigcup_{a \in I} U_a)} \quad \text{where } I = range(x, \varphi, \mathfrak{A}, \zeta)$$

Semantics of transition rules (continued)

$$\frac{\mathsf{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U)}{\mathsf{yields}(\textbf{choose } x \textbf{ with } \varphi \textbf{ do } P, \mathfrak{A}, \zeta, U)} \quad \text{if } a \in range(x, \varphi, \mathfrak{A}, \zeta)$$

$$\frac{}{\mathsf{yields}(\textbf{choose } x \textbf{ with } \varphi \textbf{ do } P, \mathfrak{A}, \zeta, \emptyset)} \quad \text{if } range(x, \varphi, \mathfrak{A}, \zeta) = \emptyset$$

$$\frac{\mathsf{yields}(P, \mathfrak{A}, \zeta, U) \quad \mathsf{yields}(Q, \mathfrak{A} + U, \zeta, V)}{\mathsf{yields}(P \ \textbf{seq} \ Q, \mathfrak{A}, \zeta, U \oplus V)} \quad \text{if } U \text{ is consistent}$$

$$\frac{\mathsf{yields}(P, \mathfrak{A}, \zeta, U)}{\mathsf{yields}(P \ \textbf{seq} \ Q, \mathfrak{A}, \zeta, U)} \quad \text{if } U \text{ is inconsistent}$$

$$\frac{\mathsf{yields}(P\frac{t_1 \cdots t_n}{x_1 \cdots x_n}, \mathfrak{A}, \zeta, U)}{\mathsf{yields}(r(t_1, \ldots, t_n), \mathfrak{A}, \zeta, U)} \quad \text{where } r(x_1, \ldots, x_n) = P \text{ is a rule declaration of } M$$

$$range(x, \varphi, \mathfrak{A}, \zeta) = \{a \in |\mathfrak{A}| : [\![\varphi]\!]^{\mathfrak{A}}_{\zeta[x \mapsto a]} = true\}$$

Coincidence, Substitution, Isomorphisms

Lemma (Coincidence). If $\zeta(x) = \eta(x)$ for all free variables x of a transition rule P and P yields U in $\mathfrak{A}$ under ζ, then P yields U in $\mathfrak{A}$ under η.

Lemma (Substitution). Let t be a static term and $a = [\![t]\!]^{\mathfrak{A}}_{\zeta}$. Then the rule $P\frac{t}{x}$ yields the update set U in state $\mathfrak{A}$ under ζ iff P yields U in $\mathfrak{A}$ under $\zeta[x \mapsto a]$.

Lemma (Isomorphism). If α is an isomorphism from $\mathfrak{A}$ to $\mathfrak{B}$ and P yields U in $\mathfrak{A}$ under ζ, then P yields $\alpha(U)$ in $\mathfrak{B}$ under $\alpha \circ \zeta$.

Move of an ASM

Definition. A machine M can make a *move* from state $\mathfrak{A}$ to $\mathfrak{B}$ (written $\mathfrak{A} \overset{M}{\Longrightarrow} \mathfrak{B}$), if the main rule of M yields a consistent update set U in state $\mathfrak{A}$ and $\mathfrak{B} = \mathfrak{A} + U$.

- The updates in U are called *internal updates*.
- $\mathfrak{B}$ is called the *next internal state*.

If α is an isomorphism from $\mathfrak{A}$ to $\mathfrak{A}'$, the following diagram commutes:

$$\begin{array}{ccc} \mathfrak{A} & \overset{M}{\Longrightarrow} & \mathfrak{B} \\ \alpha \downarrow & & \downarrow \alpha \\ \mathfrak{A}' & \overset{M}{\Longrightarrow} & \mathfrak{B}' \end{array}$$

Run of an ASM

Let M be an ASM with signature Σ.

A *run* of M is a finite or infinite sequence $\mathfrak{A}_0, \mathfrak{A}_1, \ldots$ of states for Σ such that

- $\mathfrak{A}_0$ is an initial state of M
- for each n,
  - either M can make a move from $\mathfrak{A}_n$ into the next internal state $\mathfrak{A}'_n$ and the environment produces a consistent set of external or shared updates U such that $\mathfrak{A}_{n+1} = \mathfrak{A}'_n + U$,
  - or M cannot make a move in state $\mathfrak{A}_n$ and $\mathfrak{A}_n$ is the last state in the run.

- In *internal* runs, the environment makes no moves.
- In *interactive* runs, the environment produces updates.

Example

Example 3.19. *Minimal spanning tree:: **Prim's algorithm***

Two separated phases: initial, run

Signature: Weighted graph (connected, without loops) given by sets NODE, EDGE, ... functions
weight : *EDGE* $\to$ *REAL*, *frontier* : *EDGE* $\to$ *Bool*, *tree* : *EDGE* $\to$ *Bool*

```
if mode = initial then
    choose p : NODE
        Selected(p) := true
        forall e : EDGE : p ∈ endpoints(e)
            frontier(e) := true
    mode := run
```


Example: Prim's algorithm (Cont.)

```
if mode = run then
    choose e : EDGE : frontier(e) ∧
            ((∀f ∈ EDGE) : frontier(f) ⇒ weight(f) ≥ weight(e))
        tree(e) := true
        choose p : NODE : p ∈ endpoints(e) ∧ ¬Selected(p)
            Selected(p) := true
            forall f : EDGE : p ∈ endpoints(f)
                frontier(f) := ¬frontier(f)
    ifnone mode := done
```

How can we prove the correctness, termination?

Exercise 3.20. *Construct an ASM-Machine that implements Kruskal's algorithm.*

Part 4

The reserve of ASMs

Importing new elements from the reserve

Import rule: **import** x **do** P

Meaning: Choose an element x from the reserve, delete it from the reserve and execute P.

let $x = new(X)$ **in** P abbreviates

import x **do**
$X(x) := true$
P

The reserve of a state

- New dynamic relation $Reserve$.
- $Reserve$ is updated by the system, not by rules.
- $Res(\mathfrak{A}) = \{a \in |\mathfrak{A}| : Reserve^{\mathfrak{A}}(a) = true\}$
- The reserve elements of a state are not allowed to be in the domain and range of any basic function of the state.

Definition. A state $\mathfrak{A}$ satisfies the *reserve condition* with respect to an environment ζ, if the following two conditions hold for each element $a \in Res(\mathfrak{A}) \setminus ran(\zeta)$:
- The element a is not the content of a location of $\mathfrak{A}$.
- If a is an element of a location l of $\mathfrak{A}$ which is not a location for $Reserve$, then the content of l in $\mathfrak{A}$ is $undef$.

Semantics of ASMs with a reserve

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U)}{\text{yields}(\textbf{import } x \textbf{ do } P, \mathfrak{A}, \zeta, V)} \quad \text{if } a \in Res(\mathfrak{A}) \setminus ran(\zeta) \text{ and } V = U \cup \{((Reserve, a), false)\}$$

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta, U) \quad \text{yields}(Q, \mathfrak{A}, \zeta, V)}{\text{yields}(P \textbf{ par } Q, \mathfrak{A}, \zeta, U \cup V)} \quad \text{if } Res(\mathfrak{A}) \cap El(U) \cap El(V) \subseteq ran(\zeta)$$

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U_a) \quad \text{for each } a \in I}{\text{yields}(\textbf{forall } x \textbf{ with } \varphi \textbf{ do } P, \mathfrak{A}, \zeta, \bigcup_{a \in I} U_a)} \quad \text{if } I = range(x, \varphi, \mathfrak{A}, \zeta) \text{ and for } a \neq b \; Res(\mathfrak{A}) \cap El(U_a) \cap El(U_b) \subseteq ran(\zeta)$$

- $El(U)$ is the set of elements that occur in the updates of U.
- The elements of an update (l, v) are the value v and the elements of the location l.

Problem

Problem 1: New elements that are imported in parallel must be different.

import x **do** $parent(x) = root$
import y **do** $parent(y) = root$

Problem 2: Hiding of bound variables.

import x **do**
  $f(x) := 0$
  let $x = 1$ **in**
    import y **do** $f(y) := x$

Syntactic constraint. In the scope of a bound variable the same variable should not be used again as a bound variable (**let**, **forall**, **choose**, **import**).

Preservation of the reserve condition

Lemma (Preservation of the reserve condition).
If a state $\mathfrak{A}$ satisfies the reserve condition wrt. ζ and P yields a consistent update set U in $\mathfrak{A}$ under ζ, then

- the sequel $\mathfrak{A} + U$ satisfies the reserve condition wrt. ζ,
- $Res(\mathfrak{A} + U) \setminus ran(\zeta)$ is contained in $Res(\mathfrak{A}) \setminus El(U)$.

Permutation of the reserve

Lemma (Permutation of the reserve). Let $\mathfrak{A}$ be a state that satisfies the reserve condition wrt. ζ. If α is a function from $|\mathfrak{A}|$ to $|\mathfrak{A}|$ that permutes the elements in $Res(\mathfrak{A}) \setminus ran(\zeta)$ and is the identity on non-reserve elements of $\mathfrak{A}$ and on elements in the range of ζ, then α is an isomorphism from $\mathfrak{A}$ to $\mathfrak{A}$.

Independence of the choice of reserve elements

Lemma (Independence).
Let P be a rule of an ASM without **choose**. If

- $\mathfrak{A}$ satisfies the reserve condition wrt. ζ,
- the bound variables of P are not in the domain of ζ,
- P yields U in $\mathfrak{A}$ under ζ,
- P yields U' in $\mathfrak{A}$ under ζ,

then there exists a permutation α of $Res(\mathfrak{A}) \setminus ran(\zeta)$ such that $\alpha(U) = U'$.

Example: Abstract Data Types (ADT)

Example 3.21. *Double-linked lists*

See ASM-Buch.

Exercise 3.22. *Give an ASM-Specification for the data structure bounded stack.*

Distributed ASM: Concurrency, reactivity, time

Distributed ASM (DASM)

- Computation model:
  - Asynchronous computations
  - Autonomous operating agents
- A finite set of autonomous ASM-agents, each with a program of his own.
- Agents interact through reading and writing common locations of global machine states.
- Potential conflicts are solved through the underlying semantic model, according to the definition of (partial-ordered) runs.

Foundations: Orders, CPO's, Proof techniques

Properties of binary relations

- X set
- $\rho \subseteq X \times X$ binary relation
- Properties

(P1)	$x \rho x$	(reflexive)
(P2)	$(x \rho y \wedge y \rho x) \rightarrow x = y$	(antisymmetric)
(P3)	$(x \rho y \wedge y \rho z) \rightarrow x \rho z$	(transitive)
(P4)	$(x \rho y \vee y \rho x)$	(linear)

Quasi-Orders

- $\lesssim \subseteq X \times X$ Quasi-order iff $\lesssim$ reflexive and transitive.
- Kernel:

$$\approx \; = \; \lesssim \cap \lesssim^{-1}$$

- Strict part: $< \; = \; \lesssim \setminus \approx$
- $Y \subseteq X$ left-closed (in respect of $\lesssim$) iff

$$(\forall y \in Y : (\forall x \in X : x \lesssim y \rightarrow x \in Y))$$

- Notation: Quasi-order $(X, \lesssim)$

Partial-Orders

- $\leq \subseteq X \times X$ partial-order iff $\leq$ reflexive, antisymmetric and transitive.
- Kernel: Following holds

$$\mathrm{id}_X = \leq \cap \leq^{-1}$$

- Strict part: $< \quad = \quad \leq \setminus \mathrm{id}_X$
- Often: $<$ Partial-order iff $<$ irreflexive, transitive.
- Notation: Partial-order $(X, \leq)$

Well-founded Orderings

- Partial-order $\leq \subseteq X \times X$ well-founded iff

$$(\forall Y \subseteq X : Y \neq \emptyset \rightarrow (\exists y \in Y : y \text{ minimal in } Y \text{ in respect of } \leq))$$

- Quasi-order $\lesssim$ well-founded iff strict part of $\lesssim$ is well-founded.
- Initial segment: $Y \subseteq X$, left-closed
- Initial section of x: $\mathrm{sec}(x) = \{y : y < x\}$

Supremum

- Let $(X, \leq)$ be a partial-order and $Y \subseteq X$
- $S \subseteq X$ is a chain iff elements of S are linearly ordered through $\leq$.
- y is an upper bound of Y iff

$$\forall y' \in Y : y' \leq y$$

- Supremum: y is a supremum of Y iff y is an upper bound of Y and

$$\forall y' \in X : ((y' \text{ upper bound of } Y) \rightarrow y \leq y')$$

- Analog: lower bound, Infimum $\inf(Y)$

CPO

- A Partial-order $(D, \sqsubseteq)$ is a complete partial ordering (CPO) iff
  - $\exists$ the smallest element $\perp$ of D (with respect of $\sqsubseteq$)
  - Each chain S has a supremum $\sup(S)$.

Example

Example 4.1. ▸ $(\mathcal{P}(X), \subseteq)$ *is CPO.*

- $(D, \sqsubseteq)$ *is CPO with*
  - $D = X \rightharpoonup Y$*: set of all the partial functions* f *with* $\mathrm{dom}(f) \subseteq X$ *and* $\mathrm{cod}(f) \subseteq Y$.
  - *Let* $f, g \in X \rightharpoonup Y$.

$$f \sqsubseteq g \text{ iff } \mathrm{dom}(f) \subseteq \mathrm{dom}(g) \wedge (\forall x \in \mathrm{dom}(f) : f(x) = g(x))$$

Monotonous, continuous

- $(D, \sqsubseteq)$, $(E, \sqsubseteq')$ CPOs
- $f : D \to E$ monotonous iff

$$(\forall d, d' \in D : d \sqsubseteq d' \to f(d) \sqsubseteq' f(d'))$$

- $f : D \to E$ continuous iff f monotonous and

$$(\forall S \subseteq D : S \text{ chain } \to f(\sup(S)) = \sup(f(S)))$$

- $X \subseteq D$ is admissible iff

$$(\forall S \subseteq X : S \text{ chain } \to \sup(S) \in X)$$

Fixpoint

- $(D, \sqsubseteq)$ CPO, $f : D \to D$
- $d \in D$ fixpoint of f iff

$$f(d) = d$$

- $d \in D$ smallest fixpoint of f iff d fixpoint of f and

$$(\forall d' \in D : d' \text{ fixpoint } \to d \sqsubseteq d')$$

Fixpoint-Theorem

Theorem 4.2 (Fixpoint-Theorem:). $(D, \sqsubseteq)$ *CPO,* $f : D \to D$ *continuous, then* f *has a smallest fixpoint* μf *and*

$$\mu f = \sup\{f^i(\bot) : i \in \mathbb{N}\}$$

Proof: (Sketch)

- $\sup\{f^i(\bot) : i \in \mathbb{N}\}$ fixpoint:

$$\begin{aligned} f(\sup\{f^i(\bot) : i \in \mathbb{N}\}) &= \sup\{f^{i+1}(\bot) : i \in \mathbb{N}\} \\ &\quad \text{(continuous)} \\ &= \sup\{\sup\{f^{i+1}(\bot) : i \in \mathbb{N}\}, \bot\} \\ &= \sup\{f^i(\bot) : i \in \mathbb{N}\} \end{aligned}$$

Fixpoint-Theorem (Cont.)

Fixpoint-Theorem: $(D, \sqsubseteq)$ CPO, $f : D \to D$ continuous, then f has a smallest fixpoint μf and

$$\mu f = \sup\{f^i(\bot) : i \in \mathbb{N}\}$$

Proof: (Continuation)

- $\sup\{f^i(\bot) : i \in \mathbb{N}\}$ smallest fixpoint:
  1. d' fixpoint of f
  2. $\bot \sqsubseteq d'$
  3. f monotonous, d' FP: $f(\bot) \sqsubseteq f(d') = d'$
  4. Induction: $\forall i \in \mathbb{N} : f^i(\bot) \sqsubseteq f^i(d') = d'$
  5. $\sup\{f^i(\bot) : i \in \mathbb{N}\} \sqsubseteq d'$

Induction over $\mathbb{N}$

Induction's principle:

$$(\forall X \subseteq \mathbb{N} : ((0 \in X \wedge (\forall x \in X : x \in X \to x + 1 \in X))) \to X = \mathbb{N})$$

Correctness:

1. Let's assume no, so $\exists X \subseteq \mathbb{N} : \mathbb{N} \setminus X \neq \emptyset$
2. Let y be minimum in $\mathbb{N} \setminus X$ (with respect to $<$).
3. $y \neq 0$
4. $y - 1 \in X \wedge y \notin X$
5. Contradiction

Induction over $\mathbb{N}$ (Alternative)

Induction's principle:

$$(\forall X \subseteq \mathbb{N} : (\forall x \in \mathbb{N} : \text{sec}(x) \subseteq X \to x \in X) \to X = \mathbb{N})$$

Correctness:

1. Let's assume no, so $\exists X \subseteq \mathbb{N} : \mathbb{N} \setminus X \neq \emptyset$
2. Let y be minimum in $\mathbb{N} \setminus X$ (with respect to $<$).
3. $\text{sec}(y) \subseteq X, y \notin X$
4. Contradiction

Well-founded induction

Induction's principle: Let $(Z, \leq)$ be a well-founded partial order.

$$(\forall X \subseteq Z : (\forall x \in Z : \text{sec}(x) \subseteq X \to x \in X) \to X = Z)$$

Correctness:

1. Let's assume no, so $Z \setminus X \neq \emptyset$
2. Let z be minimum in $Z \setminus X$ (in respect of $\leq$).
3. $\text{sec}(z) \subseteq X, z \notin X$
4. Contradiction

FP-Induction: Proving properties of fixpoints

Induction's principle: Let $(D, \sqsubseteq)$ CPO, $f : D \to D$ continuous.

$$(\forall X \subseteq D \text{ admissible} : (\bot \in X \wedge (\forall y : y \in X \to f(y) \in X)) \to \mu f \in X)$$

Correctness: Let $X \subseteq D$ admissible.

$$\begin{array}{lll} \mu f \in X & \Leftrightarrow & \sup\{f^i(\bot) : i \in \mathbb{N}\} \in X \qquad \text{(FP-theorem)} \\ & \Leftarrow & \forall i \in \mathbb{N} : f^i(\bot) \in X \qquad (X \text{ admissible }) \\ & \Leftarrow & \bot \in X \wedge (\forall n \in \mathbb{N} : f^n(\bot) \in X \to f(f^n(\bot)) \in X) \\ & & \qquad \text{(Induction } \mathbb{N}) \\ & \Leftarrow & \bot \in X \wedge (\forall y \in X \to f(y) \in X) \qquad \text{(Ass.)} \end{array}$$

Problem

Exercise 4.3. *Let $(D, \sqsubseteq)$ CPO with*

- $X = Y = \mathbb{N}$
- $D = X \rightharpoonup Y$*: set all partial functions f with* $\mathrm{dom}(f) \subseteq X$ *and* $\mathrm{cod}(f) \subseteq Y$.
- *Let* $f, g \in X \rightharpoonup Y$.

$$f \sqsubseteq g \text{ iff } \mathrm{dom}(f) \subseteq \mathrm{dom}(g) \wedge (\forall x \in \mathrm{dom}(f) : f(x) = g(x))$$

Consider

$$\begin{array}{rcl} F : \quad D & \to & \mathcal{P}(\mathbb{N} \times \mathbb{N}) \\ g & \mapsto & \begin{cases} \{(0,1)\} & g = \emptyset \\ \{(x, x \cdot g(x-1)) : x - 1 \in \mathrm{dom}(g)\} \cup \{(0,1)\} & \textit{otherwise} \end{cases} \end{array}$$

Problem

Prove:

1. $\forall g \in D : F(g) \in D$, i.e. $F : D \to D$
2. $F : D \to D$ continuous
3. $\forall n \in \mathbb{N} : \mu F(n) = n!$

Note:

- μF can be understood as the semantics of a function's definition

$$\begin{array}{l} \text{function Fac}(n : \mathbb{N}_\bot) : \mathbb{N}_\bot =_{\text{def}} \\ \quad \text{if } n = 0 \text{ then } 1 \\ \quad \text{else } n \cdot \text{Fac}(n-1) \end{array}$$

- Keyword: 'derived functions' in ASM

Problem

Exercise 4.4. *Prove: Let $G = (V, E)$ be an infinite directed graph with*

- *G has finitely many roots (nodes without incoming edges).*
- *Each node has finite out-degree.*
- *Each node is reachable from a root.*

There exists an infinite path that begins on a root.

Distributed ASM

Definition 4.5. *A DASM A over a signature (vocabulary) Σ is given through:*

- *A distributed programm Π_A over Σ.*
- *A non-empty set I_A of initial states*
  An initial state defines a possible interpretation of Σ over a potential infinite base set X.

A contains in the signature a dynamic relation's symbol AGENT, that is interpreted as a finite set of autonomous operating agents.

- *The behaviour of an agent a in state S of A is defined through $program_S(a)$.*
- *An agent can be ended through the definition of $program_S(a) := undef$ (representation of an invalid programm).*

Partially ordered runs

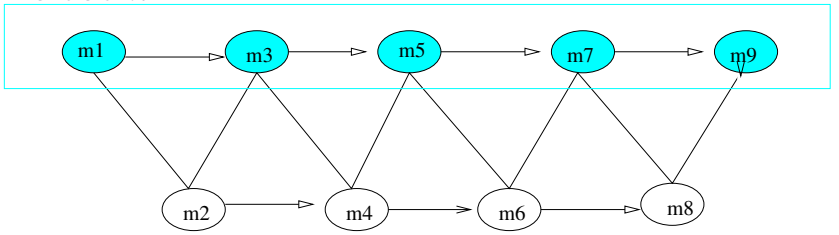
A run of a distributed ASM A is given through a triple $\varrho \rightleftharpoons (M, \lambda, \sigma)$ with the following properties:

1. M is a partial ordered set of "moves", in which each move has only a finite number of predecessors.
2. λ is a function on M, that assigns an agent to each move, so that the moves of a particular agent are always linearly ordered.
3. σ asociates a state of A with each finite initial segment Y of M. Intended meaning:: $\sigma(Y)$ is the "result of the execution of all moves in Y". $\sigma(Y)$ is an initial state when Y is empty.
4. The coherence condition is satisfied:
   If max is a set of maximal elements in a finite initial segment X of M and $Y = X \setminus max$, then for $x \in max$:: $\lambda(x)$ is an agent in $\sigma(Y)$ and we get $\sigma(X)$ from $\sigma(Y)$ by firing $\{\lambda(x) : x \in max\}$ (their programs) in $\sigma(Y)$.

Comment, example

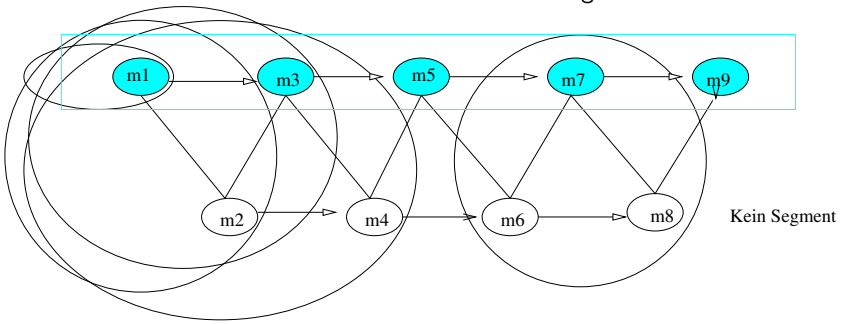
The agents of A modell the concurrent control-threads in the execution of Π_A.
A run can be seen as the common part of the history of the same computation from the point of view of multiple observers.

The role of λ:



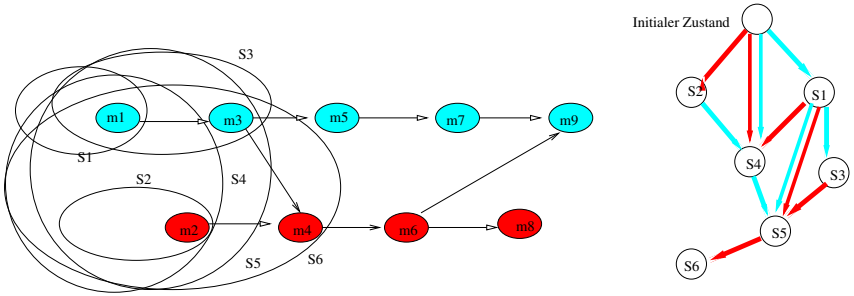
Comment, example (cont.)

The role of σ: Snap-shots of the computation are the initial segments of the partial ordered set M. To each initial segment a state of A is assigned (interpretation of Σ), that reflects the execution of the programs of the agents that appear in the segment.
$\rightsquigarrow$"Result of the execution of all the moves" in the segment.



Coherence condition, example

If *max* is a set of maximal elements in a finite initial segment X of M and $Y = X \setminus max$, then for $x \in max$:: $\lambda(x)$ is an agent in $\sigma(Y)$ and we get $\sigma(X)$ from $\sigma(Y)$ by firing $\{\lambda(x) : x \in max\}$ (their programs) in $\sigma(Y)$.



Consequences of the coherence condition

Lemma 4.6. *All the linearizations of an initial segment (i.e. respecting the partial ordering) of a run ϱ lead to the same "final" state.*

Lemma 4.7. *A property P is valid in all the reachable states of a run ϱ, iff it is valid in each of the reachable states of the linearizations of ϱ.*

Simple example

Example 4.8. *Let $\{door, window\}$ be propositional-logic constants in the signature with natural meaning:*
door = true means " door open " and analog for window.

The program has two agents, a door-manager d and a window-manager w with the following programs:

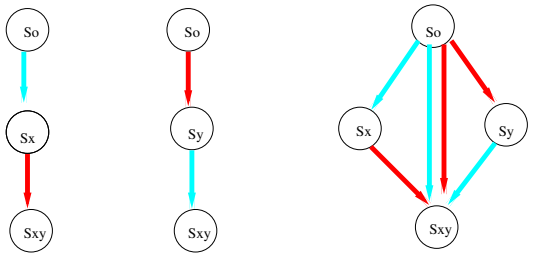
$program_d$ = *door := true* // *move x*
$program_w$ = *window := true* // *move y*

In the initial state S_0 let the door and window be closed, let d and w be in the agent set.

Which are the possible runs?

Simple example (Cont.)

Let $\varrho_1 = ((\{x, y\}, x < y), id, \sigma), \varrho_2 = ((\{x, y\}, y < x), id, \sigma)$,
$\varrho_3 = ((\{x, y\}, <>), id, \sigma)$ (coarsest partial order)

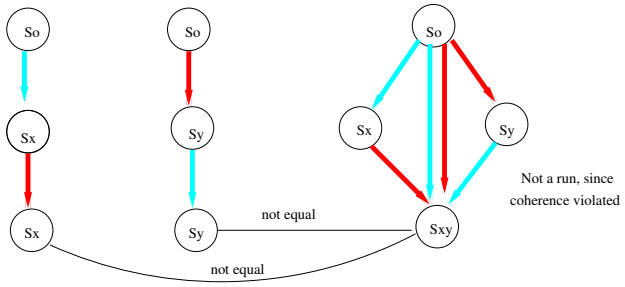


Variants of simple example

The program consists of two agents, a door-Manager d and a window-manager w with the following programs:

$program_d = if \neg window \ then \ \ door := true \quad // \text{ move } x$
$program_w = if \neg door \ then \ \ window := true \ // \text{ move } y$

In the initial state S_0 let the door and window be closed, let d and w be in the agent set. How do the runs look like? Same ϱ's as before.



More variations

Exercise 4.9. *Consider the following pair of agents* $x, y \in \mathbb{N}$ *($x = 2, y = 1$ in the initial state)*

1. $a = x := x + 1$ *and* $b = x := x + 1$
2. $a = x := x + 1$ *and* $b = x := x - 1$
3. $a = x := y$ *and* $b = y := x$

Which runs are possible with partial-ordered sets containing two elements?

Try to characterize all the runs.

More variations

Consider the following agents with the conventional interpretation:

1. $Program_d = \ if \ \neg window \ \ then \ \ door := true \quad //$move x
2. $Program_w = \ if \ \neg door \ \ then \ \ window := true \quad //$move y
3. $Program_l = \ if \ \neg light \wedge (\neg door \vee \neg window) \ \ then \ //$move z
   $light := true$
   $door := false$
   $window := false$

Which end states are possible, when in the initial state the three constants are false?

Further exercises

Consumer-producer problem: Assume a single producer agent and two or more consumer agents operating concurrently on a global shared structure. This data structure is linearly organized and the producer adds items at the one end side while the consumers can remove items at the opposite end of the data structure. For manipulating the data structure, assume operations *insert* and *remove* as introduced below.

$insert : \ \ Item \times \ \ ItemList \rightarrow \ \ ItemList$
$remove : \ \ ItemList \rightarrow \ \ (Item \times \ \ ItemList)$

(1) Which kind of potential conflicts do you see?
(2) How does the semantic model of partially ordered runs resolve such conflicts?

Environment

Reactive systems are characterized by their interaction with the environment. This can be modeled with the help of an environment-agent. The runs can then contain this agent (with λ), λ must define in this case the update-set of the environment in the corresponding move.
The coherence condition must also be valid for such runs.

For externally controlled functions this surely doesn't lead to inconsistencies in the update-set, the behaviour of the internal agents can of course be influenced. Inconsistent update-sets can arise in shared functions when there's a simultaneous execution of moves by an internal agent and the environment agent.

Often certain assumptions or restrictions (suppositions) concerning the environment are done.
In this aspect there are a lot of possibilities: the environment will be only observed or the environment meets stipulated integrity conditions.

Time

The description of real-time behaviour must consider explicitly time aspects. This can be done successfully with help of timers (see SDL), global system time or local system time.

- The reactions can be instantaneous (the firing of the rules by the agents don't need time)
- Actions need time

Concerning the global time consideration, we assume, that there is on hand a linear ordered domain *TIME*, for instance with the following declarations:

$$domain \quad (TIME, \leq), \quad (TIME, \leq) \subset (\mathbb{R}, \leq)$$

In these cases the time will be measured with a discrete system watch: e.g.

$$monitored \quad now :\rightarrow TIME$$

ATM (Automatic Teller Machine)

Exercise 4.10. *Abstract modeling of a cash terminal:*
Three agents are in the model: ct-manager, authentication-manager, account-manager. To withdraw an amount from an account, the following logical operations must be executed:

1. *Input the card (number) and the PIN.*
2. *Check the validity of the card and the PIN (AU-manager).*
3. *Input the amount.*
4. *Check if the amount can be withdrawn from the account (ACC-manager).*
5. *If OK, update the account's stand and give out the amount.*
6. *If it is not OK, show the corresponding message.*

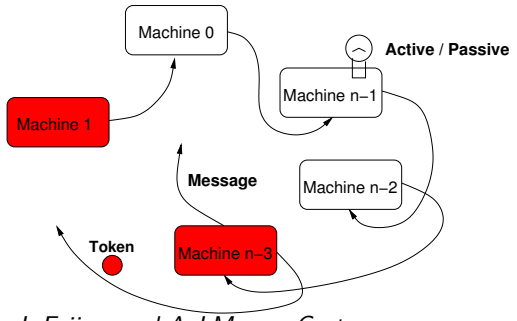
Implement an asynchronous communication model in which timeouts can cancel transactions .

Distributed Termination Detection

Example 4.11. *Implement the following termination detection protocol:*

A passive machine becomes active, iff it receives a message from another machine.

Only active machines can send messages.



Edsger W. Dijkstra, W. H. J. Feijen, and A.J.M. van Gasteren. Derivation of a Termination Detection Algorithm for Distributed Computations. IPL 16 (1983).

Assumptions for distributed termination detection

Rules for a probe

Rule 0 When active, $Machine_{i+1}$ keeps the token; when passive, it hands over the token to $Machine_i$.

Rule 1 A machine sending a message makes itself red.

Rule 2 When $Machine_{i+1}$ propagates the probe, it hands over a red token to $Machine_i$ when it is red itself, whereas while being white it leaves the color of the token unchanged.

Rule 3 After the completion of an unsuccessful probe, $Machine_0$ initiates a next probe.

Rule 4 $Machine_0$ initiates a probe by making itself white and sending to $Machine_{n-1}$ a white token.

Rule 5 Upon transmission of the token to $Machine_i$, $Machine_{i+1}$becomes white. (Notice that the original color of $Machine_{i+1}$ may have affected the color of the token).

Distributed Termination Detection: Procedure

Signature:

static
$COLOR = \{red, white\}$ $TOKEN = \{redToken, whiteToken\}$
$MACHINE = \{0, 1, 2, \ldots, n-1\}$
$next : MACHINE \rightarrow MACHINE$
e.g. with $next(0) = n-1, next(n-1) = n-2, \ldots, next(1) = 0$

controlled
$color : MACHINE \rightarrow COLOR$ $token : MACHINE \rightarrow TOKEN$
$RedTokenEvent, WhiteTokenEvent : MACHINE \rightarrow BOOL$

monitored $Active : MACHINE \rightarrow BOOL$
$SendMessageEvent : MACHINE \rightarrow BOOL$

Distributed Termination Detection: Procedure

Macros: (Rule definitions)

- $ReactOnEvents(m : MACHINE) =$
  if $RedTokenEvent(m)$ *then*
  $token(m) := redToken$
  $RedTokenEvent(m) :=$ *undef*
  if $WhiteTokenEvent(m)$ *then*
  $token(m) := whiteToken$
  $WhiteTokenEvent(m) :=$ *undef*
  if $SendMessageEvent(m)$ *then* $color(m) := red$ Rule 1
- $Forward(m : MACHINE, t : TOKEN) =$
  if $t = whiteToken$ *then*
  $WhiteTokenEvent(next(m)) :=$ *true*
  else
  $RedTokenEvent(next(m)) :=$ *true*

Distributed Termination Detection: Procedure

Programs

- $RegularMachineProgram =$

  $ReactOnEvents(me)$
  if $\neg$ $Active(me) \wedge token(me) \neq$ *undef* *then* Rule 0
  $InitializeMachine(me)$ Rule 5
  if $color(me) =$ *red* *then*
  $Forward(me, redToken)$ Rule 2
  else
  $Forward(me, token(me))$ Rule 2
- With $InitializeMachine(m : MACHINE) =$

  $token(m) := undef$
  $color(m) := white$

Distributed Termination Detection: Procedure

Programs

- *SupervisorMachineProgram* =

ReactOnEvents(*me*)
if ¬ *Active*(*me*) ∧ *token*(*me*) ≠ *undef* *then*
  if *color*(*me*) = *white* ∧ *token*(*me*) = *whiteToken* *then*
    ReportGlobalTermination
  else Rule 3
    InitializeMachine(*me*) Rule 4
    Forward(*me*, *whiteToken*) Rule 4

Distributed Termination Detection

Initial states

$\exists m_0 \in MACHINE$
$(program(m_0) = SupervisorMachineProgram \wedge$
$token(m_0) = redToken \wedge$
$(\forall m \in MACHINE)(m \neq m_0 \Rightarrow$
$(program(m) = RegularMachineProgram \wedge token(m) = undef)))$

Environment constraints For all the executions and all linearizations holds:

$\mathbf{G}\ (\forall m \in MACHINE)$
$(SendMessageEvent(m) = true \Rightarrow (\mathbf{P}(Active(m)) \wedge Active(m)))$
$\wedge\ ((Active(m) = true \wedge \mathbf{P}(\neg Active(m)) \Rightarrow$
$(\exists m' \in MACHINE)\ (m' \neq m \wedge\ SendMessageEvent(m'))))$

Nextconstraints

Distributed Termination Detection

Correctness of the abstract version: Dijkstra
Suppositions: The machines constitute a closed system, i.e. messages can only be dispatched among each other (no outside messages). The system in the initial state can have any color and several machines can be active. The token is located in the 0'th. machine. The given rules describe the transfer of the token and the coloration of the machines upon certain activities.
The task is to determine a state in which all the machines are passive (not active). This is a stable state of the system, because only active machines can dispatch messages and passive machines can only become active by receiving a message.
The invariant: Let t be the position on which the token is, then following invariant holds

$(\forall i : t < i < n\ \ Machine_i \text{ is passive}) \vee (\exists j : 0 \leq j \leq t\ \ Machine_j \text{ is red}) \vee$
$(Token \text{ is red})$

Distributed Termination Detection

$(\forall i : t < i < n\ \ Machine_i \text{ is passive}) \vee (\exists j : 0 \leq j \leq t\ \ Machine_j \text{ is red}) \vee$
$(Token \text{ is red})$

Correctness argument
When the token reaches $Machine_o$, $t = 0$ and the invariant holds.
If
$(Machine_o \text{ is passive}) \wedge (Machine_o \text{ is white}) \wedge (Token \text{ is white})$
then
$(\forall i : 0 < i < n\ \ Machine_i \text{ is passive})$ must hold, i.e. termination.

Proof of the invariant Induction over t:
The case t = n - 1 is easy.
Assume the invariant is valid for $0 < t < n$, prove it is valid for $t - 1$.

Distributed Termination Detection

Is the invariant valid in all the states of all the linearizations of the runs of the DASM ? **No**

- **Problem 1** The red coloration of an active machine (that forwards a message) occurs in a later state. It should occur in the same state in which the message-receiving machine turns active. (Instantaneous message passing)
  Solution *color* is a shared function. Instead of using $SendMessageEvent(m)$ to set the color, it will be set by the environment: $color(m) = red$.
- **Problem 2** There are states in which none of the machines has the token:: The machine that has the token, initializes itself and sets an event, that leads to a state in which none of the machines has the token.
  Solution Instead of using *FarbTokenEvent* to reset, it is directly properly set: $token(next(m))$.
- **Result** More abstract machine. The environment controls the activity of the machines, message passing and coloration.

Refinement's concepts for ASM's

Question: Is in the termination detection example the given DASM a refinement of the abstracter DASM? $\rightsquigarrow$

General refinement concepts for ASM's

- Refinements are normally defined for BASM, i.e. the executions are linear ordered runs, this makes the definition of refinements easier.
- Refinements allow abstractions, realization of data and procedures.
- ASM refinements are usually problem-oriented: Depending on the application a flexible notion of refinement should be used.
- Proof tasks become structured and easier with help of correct and complete refinements.

See ASM-Buch.
Example Shortest Path

Algebraic Specification - Equational Logic

Specification techniques' requirements:

- Abstraction (refinement)
- Structuring mechanisms
  Partition-aggregation, combination, extension-instantiation
- Clear (explicit and plausible) semantics
- Support of the „verify while develop"-principle
- Expressiveness (all the partial recursive functions representable)
- Readability (adequacy) (suitability)

⋮

Algebraic Specification - Algebras

Specification of data types

Syntax	Equations	Programs
signature axiom	$t_1 = t_2$ if φ then $t_1 = t_2$	data operations directed application

Algebras

heterogeneous	order-sorted	homogeneous
(Many-Sorted)	(Many-Sorted)	(Single-Sorted)

Single-Sorted Algebras

Example 6.1. *a) Groups*
SORT:: g
SIG:: $\cdot : g, g \to g \qquad 1 :\to g \qquad ^{-1} : g \to g$
EQN:: $x \cdot 1 = x \qquad x \cdot x^{-1} = 1 \qquad (x \cdot y) \cdot z = x \cdot (y \cdot z)$
All-quantified equations

Models are groups

Question: Which equations are valid in all groups,
i.e. $EQN \models t_1 = t_2$

$$1 \cdot x = x \qquad x^{-1} \cdot x = 1 \qquad (x^{-1})^{-1} = x$$

Single-Sorted Algebras

Equational Logic: Replace „equals“ with „equals“
Problem: cycles, non-termination
Solution: Directed equations $\rightsquigarrow$ Term rewriting systems

Find R „convergent“ with $\underset{EQN}{=} = \overset{*}{\underset{R}{\Longleftrightarrow}}$

$x \cdot 1 \to x$	$1 \cdot x \to x$
$x \cdot x^{-1} \to 1$	$x^{-1} \cdot x \to 1$
$1^{-1} \to 1$	$(x^{-1})^{-1} \to x$
$(x \cdot y)^{-1} \to y^{-1} \cdot x^{-1}$	$(x \cdot y) \cdot z \to x \cdot (y \cdot z)$
$x^{-1} \cdot (x \cdot y) \to y$	$x \cdot (x^{-1} \cdot y) \to y$

Many-Sorted Algebras

b) Lists over nat-numbers

SIG: BOOL, NAT, LIST Sorts
true, false: $\to$ BOOL
$0 \to$ NAT
suc: NAT $\to$ NAT
+: NAT, NAT $\to$ NAT
eq: NAT, NAT $\to$ BOOL
nil: $\to$ LIST
. : NAT, LIST $\to$ LIST
app: LIST, LIST $\to$ LIST
rev: LIST $\to$ LIST

Many-Sorted Algebras

Axioms are all-quantified equations, i.e.
$\forall x_1, ..., x_n, y_1, ..., y_m : \quad t_1(x_1, ..., x_n) = t_2(y_1, ..., y_m)$ where

$t_1(x_1, ..., x_n), t_2(y_1, ..., y_m)$ Terms of the same sort over the signature.

EQN : $n + 0 = n \quad n + \text{suc}(m) = \text{suc}(n + m)$

$\text{eq}(0, 0) = \text{true} \quad \text{eq}(0, \text{suc}(n)) = \text{false}$
$\text{eq}(\text{suc}(n), 0) = \text{false}$
$\text{eq}(\text{suc}(n), \text{suc}(m)) = \text{eq}(n, m)$

$\text{app}(\text{nil}, l) = l \quad \text{app}(n.l_1, l_2) = n.\,\text{app}(l_1, l_2)$

$\text{rev}(\text{nil}) = \text{nil} \quad \text{rev}(n.l) = \text{app}(\text{rev}(l), n.\text{nil})$

Many-Sorted Algebras

Terms of type BOOL, NAT, LIST as identifiers for elements.
(standard definition!)
Which algebra is specified? How can we compute in this algebra?
Direct the equations $\rightsquigarrow$ term-rewriting system R. Evidently e.g.:

$$s^i(0) + s^j(0) \xrightarrow[R]{*} s^{i+j}(0)$$

$$\text{app}(3.1.\text{nil}, \text{app}(5.\text{nil}, 1.2.3.\text{nil})) \xrightarrow[R]{*} 3.1.5.1.2.3.\text{nil}$$

$$\begin{aligned} \text{rev}(3.1.\text{nil}) &\rightarrow \text{app}(\text{rev}(1.\text{nil}), 3.\text{nil}) \\ &\rightarrow \text{app}(\text{app}(\text{rev}(\text{nil}), 1.\text{nil}), 3.\text{nil}) \\ &\rightarrow \text{app}(\text{app}(\text{nil}, 1.\text{nil}), 3.\text{nil}) \\ &\rightarrow \text{app}(1.\text{nil}, 3.\text{nil}) \xrightarrow{*} 1.3.\text{nil} \end{aligned}$$

Question: Is $\text{app}(x.y.\text{nil}, z.\text{nil}) =_E \text{app}(x.\text{nil}, y.z.\text{nil})$ true?

Many-Sorted Algebras

Some equations are not valid in all the models of EQN$=$ E.
e.g.

$$\begin{gathered} x + y \neq_E y + x \\ \text{app}(x, \text{app}(y, z)) \neq_E \text{app}(\text{app}(x, y), z) \\ \text{rev}(\text{rev}(x)) \neq_E x \end{gathered}$$

The pairs of terms cannot be joined via rewriting.

Distinction:

- Equations that are valid in all the models of E.
- Equations that are valid in data models of E.

$x + y = y + x :: s^i 0 + s^j 0 = s^j 0 + s^i 0$ all i, j
$\text{rev}(\text{rev}(x)) = x$ for $x \equiv s^{i_1}0.s^{i_2}0.\ldots s^{i_n}0.\text{nil}$

Thesis: Data types are Algebras

ADT: Abstract data types. Independent of the data representation.

Specification of abstract data types:

Concepts from Logic/universal Algebra
Objective: common language for specification and implementation.

Methods for proving the correctness:

Syntax, L formulae (P-Logic,Hoare,...)

Cl: Consequence closure (e.g. $\models$, $Th(A), \ldots$)

Consequence closure

$Cl : \mathbb{P}(L) \rightarrow \mathbb{P}(L)$ (subsets of L) with

a) $A \subset L \rightsquigarrow A \subset Cl(A)$
b) $A, B \subset L, A \subseteq B \rightsquigarrow Cl(A) \subseteq Cl(B)$ (Monotony)
c) $Cl(A) = Cl(Cl(A))$ (Maximality)

Important concepts:

Consistency: $A \subsetneq L$ A is consistent if $Cl(A) \subsetneq L$
Implementation: A implements B (Refinement)

$$L \subset L', Cl(B) \subseteq Cl(A)$$

Related to implication.

Signature - Terms

Definition 6.2. *a) Signature is a triple* sig $= (S, F, \tau)$ *(abbreviated:* Σ*)*

- S *finite set of sorts*
- F *set of operators (function symbols)*
- $\tau : F \to S^+$ *arity function, i.e.*
  $\tau(f) = s_1 \cdots s_n\ s$, $n \geq 0$, s_i *argument's sorts*, s *target sort.*

Write: $f : s_1, \ldots, s_n \to s$

(Notice that $n = 0$*) is possible, constants of sort* S*.*

Signature - Terms

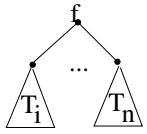
b) Term(F): Set of ground terms over sig and their tree presentation.

$$\mathrm{Term}(F) := \dot{\bigcup_{s \in S}} \mathrm{Term}_s(F)$$

recursive def.

- $f :\to s$, so $f \in \mathrm{Term}_s(F)$ representation: $\cdot f$
- $f : s_1, \ldots, s_n \to s$, $t_i \in \mathrm{Term}_{s_i}(F)$ with Rep. T_i so
  $f(t_1, \ldots, t_n) \in \mathrm{Term}_s(F)$ with Rep.

Consider the representation by ordered trees



Signature - Terms

c) $V = \dot{\bigcup_{s \in S}} V_s$ system of variables $V \cap F = \varnothing$.
Each $x \in V_s$ has functionality $x :\to s$

Set: $\mathrm{Term}(F, V) := \mathrm{Term}(F \cup V)$.

Quotation: terms over sig in the variables V.
(F and τ suitable enhanced with the variables and their sorts).

Intention: for variables is allowed to use any object of the same sort, i.e. terms of this sort. "Identifier" for an arbitrary object of this sort.

Strictness - Positions- Subterms

Definition 6.3. *a)* $s \in S$ *strict, if* $\mathrm{Term}_s(F) \neq \varnothing$
If there's for each sort $s \in S$ *a constant of sort* S *or a function* $f : s_1, \ldots, s_n \to s$*, so that the* s_i *are strict, then all the sorts of the signature are strict.*$\rightsquigarrow$ *strict signatures (general assumption)*

b) Subterms $(t) = \{t_p \mid p$ *location (position) in* p*,* t_p *subterm in* $p\}$
The positions are represented through sequences over $\mathbb{N}$
(elements of $\mathbb{N}^*$*,* e *the empty sequence).*
$O(t)$ *Set of positions in* t*,*
For $p \in O(t)$ t_p *(or* $t|_p$*) subterm of* t *in position* p

- t *constant or variable:* $O(t) = \{e\}$ $t_e \equiv t$
- $t \equiv f(t_1, \ldots, t_n)$ *so*
  $O(t) = \{ip \mid 1 \leq i \leq n, p \in O(t_i)\} \cup \{e\}$
  $t_{ip} \equiv t_i|_p$ *and* $t_e \equiv t$.

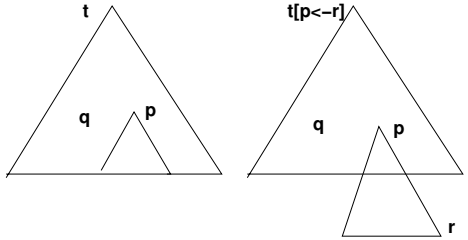
Term replacement

c) Term replacement: $t, r \in \mathrm{Term}(F, V)$
$p \in O(t)$: with $r, t_p \in \mathrm{Term}_s(F, V)$ for a sort s.

Then
$t[r]_p$, $t[p \leftarrow r]$ respectively t_p^r is the term, that is obtained from t through replacement of subterm t_p by r.

So $t[p \leftarrow r]_q = t_q$ for $q \mid p$ and

$$t[p \leftarrow r]_p = r$$



Signatures - terms

Example 6.4. $S = (\mathrm{BOOL}, \mathrm{NAT}, \mathrm{LIST})$, $F = \{true, false, \ldots\}$,
$\tau : F \rightarrow S^* :: true :\rightarrow \mathrm{BOOL}$, $eq : \mathrm{NAT}, \mathrm{NAT} \rightarrow \mathrm{BOOL}, \ldots$

$$V = \underset{\text{“}}{V_{\mathrm{BOOL}}} \cup \underset{\text{“}}{V_{\mathrm{NAT}}} \cup \underset{\text{“}}{V_{\mathrm{LIST}}}$$

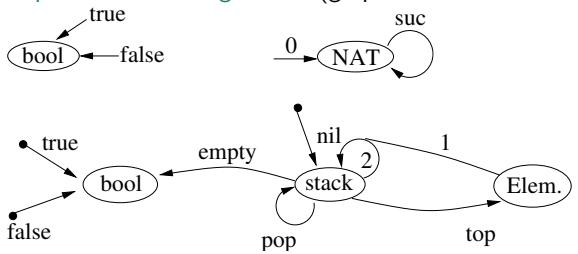
$$\{b_i : i \in \mathbb{N}\} \quad \{x_i : i \in \mathbb{N}\} \quad \{l_i : i \in \mathbb{N}\}$$

Ground terms:
$true, false, eq(0, \mathrm{suc}(0)) \in \mathrm{Term}_{\mathrm{BOOL}}(S)$
$0, \mathrm{suc}(0), \mathrm{suc}(0) + (\mathrm{suc}(\mathrm{suc}(0)) + 0) \in \mathrm{Term}_{\mathrm{NAT}}(S)$
$\mathrm{app}(nil, \mathrm{suc}(0).(\mathrm{suc}(\mathrm{suc}(0)).nil) \in \mathrm{Term}_{\mathrm{LIST}}(S)$
$0.\mathrm{suc}(0), eq(true, false), \mathrm{rev}(0)$ *no terms.*

General terms:
$eq(x_1, x_2) \in \mathrm{Term}_{\mathrm{BOOLE}}(F, V), suc(x_1) + (x_2 + \mathrm{suc}(0)) \in \mathrm{Term}_{\mathrm{NAT}}(F, V)$
$\mathrm{app}(l_1, x_1.l_0) \in \mathrm{Term}_{\mathrm{LIST}}(F, V)$
$\mathrm{rev}(x_1.l) \in \mathrm{Term}_{\mathrm{LIST}}(F, V)$
$\mathrm{app}(x_1, l_2)$ *no term.*

Signatures

Representation of signatures (graphical or standardized)



Notations:
sig ...
sorts ...
ops ...
op: $W \rightarrow S$
$\mathrm{op}_1, \ldots, \mathrm{op}_i : W \rightarrow S$

Interpretations: sig-Algebras

Definition 6.5. $\mathrm{sig} = (S, F, \tau)$ *signature. A sig-Algebra* $\mathfrak{A}$ *is composed of*

1) *Set of support* $A = \bigcup_{s \in S} A_s$, $A_s \neq \varnothing$ *set of support of sort s.*
2) *Function system* $F_{\mathfrak{A}} = \{f_{\mathfrak{A}} : f \in F\}$ *with*
$f_{\mathfrak{A}} : A_{s_1} \times \cdots \times A_{s_n} \rightarrow A_s$ *function and* $\tau(f) = s_1 \cdots s_n s$.

Notice: The $f_{\mathfrak{A}}$ are total functions.

The precondition $A_s \neq \varnothing$ is not mandatory.

Interpretations: sig-Algebras

Example 6.6. *a)* sig $\equiv$ BOOL-*algebras*, *true*, *false* $:\longrightarrow$ BOOL

$\mathfrak{A}_1$	$\{0,1\}$	$true_{\mathfrak{A}_1} = 0$	$false_{\mathfrak{A}_1} = 1$
$\mathfrak{A}_2$	$\{0,1\}$	$true_{\mathfrak{A}_2} = 0$	$false_{\mathfrak{A}_2} = 0$
$\mathfrak{A}_3$	$\mathbb{N}$	$true_{\mathfrak{A}_3} = 4$	$false_{\mathfrak{A}_3} = 5$
$\mathfrak{A}_4$	$\{true, false\}$	$true_{\mathfrak{A}_4} = true$	$false_{\mathfrak{A}_4} = false$

bool-Alg.

b) sig $\equiv$ NAT, 0, suc

$A_{i_{\mathrm{NAT}}}$	$\mathbb{N}$	$\mathbb{Z}$	$\mathbb{N}$	$\{true, false\}$	$\{0, \mathrm{suc}^i(0)\}$
$0_{\mathfrak{A}_i}$	0	0	1	*true*	0
$\mathrm{suc}_{\mathfrak{A}_i}$	$\mathrm{suc}_{\mathbb{N}}$	$\mathrm{pred}_{\mathbb{Z}}$	$\mathrm{id}_{\mathbb{N}}$	$\mathrm{suc}(true) = false$	$\mathrm{suc}(0) = suc(0)$
				$\mathrm{suc}(false) = true$	$\mathrm{suc}(\mathrm{suc}^i(0)) = \mathrm{suc}^{i+1}(0)$

Free sig-algebra generated by V

Definition 6.7.

- $\mathfrak{A} = (A, F_{\mathfrak{A}})$ *with:* $A = \bigcup_{s \in S} A_s$ $A_s = \mathrm{Term}_s(F, V)$, *i.e.* $A = \mathrm{Term}(F, V)$
  $F \ni f : s_1, \ldots, s_n \to s$, $f_{\mathfrak{A}}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$

  $\mathfrak{A}$ *is sig-Algebra::* $T_{\mathrm{sig}}(V)$
  the free termalgebra in the variables V *generated by* V
- $V = \varnothing$: $A_s = \mathrm{Term}_s(F)$ *set of ground terms*
  ($A_s \neq \varnothing$, *because sig is strict).*

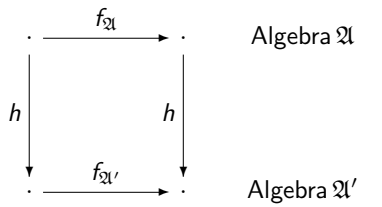
  $\mathfrak{A}$ *ground termalgebra::* T_{sig}

Homomorphisms

Definition 6.8 (sig-homomorphism). $\mathfrak{A}, \mathfrak{A}'$ *sig-algebras*
$h : \mathfrak{A} \to \mathfrak{A}'$ *family of functions*
$h = \{h_s : A_s \to A'_s : s \in S\}$ *is sig-homomorphism when*

$$h_s(f_{\mathfrak{A}}(a_1, \ldots, a_n)) = f_{\mathfrak{A}'}(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$$

As always: injective, surjective, bijective, isomorphism



Canonical homomorphisms

Lemma 6.9. $\mathfrak{A}$ *sig-Algebra,* T_{sig} *ground term algebra*

a) *The family of canonical interpretation functions* $h_s : \mathrm{Term}_s(F) \to A_s$ *defined through*

$$h_s(f(t_1, \ldots, t_n)) = f_{\mathfrak{A}}(h_{s_1}(t_1), \ldots, h_{s_n}(t_n))$$

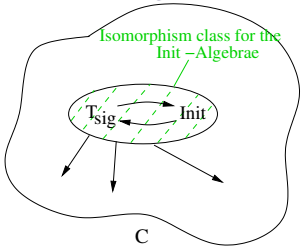
with $h_s(c) = c_{\mathfrak{A}}$ *is a sig-homomorphism.*

b) *There is no other sig-homomorphism from* T_{sig} *to* $\mathfrak{A}$. *Uniqueness!*

Proof: Just try!!

Initial algebras

Definition 6.10 (Initial algebras). *A sig-Algebra $\mathfrak{A}$ is called initial in a class C of sig-algebras, if for each sig-Algebra $\mathfrak{A}' \in C$ exists exactly one sig-homomorphism $h : \mathfrak{A} \to \mathfrak{A}'$.*
***Notice**: T_{sig} is initial in the class of all sig-algebras (Lemma 6.9).*
Fact: Initial algebras are isomorphic.



The final algebras can be defined analogously.

Canonical homomorphisms

$\mathfrak{A}$ sig-Algebra, $h : T_{sig} \to \mathfrak{A}$ interpretation homomorphism.
$\mathfrak{A}$ sig-generated (term-generated) iff
$\forall s \in S \quad h_s : \mathrm{Term}_s(F) \to A_s$ surjective

The ground termalgebra is sig-generated.

ADT requirements:

- Independent of the representation (isomorphism class)
- Generated by the operations (sig-generated)
  Often: constructor subset

Thesis: An ADT is the isomorphism class of an initial algebra.

Ground termalgebras as initial algebras are ADT.

Notice by the properties of free termalgebras : functions from V in $\mathfrak{A}$ can be extended to unique homomorphisms from $T_{sig}(V)$ in $\mathfrak{A}$.

Equational specifications

For Specification's formalisms:

Classes of algebras that have initial algebras.

$\rightsquigarrow$ Horn-Logic (See bibliography)

sig INT sorts int
ops 0 :⟶ int
suc : int ⟶ int
pred : int ⟶ int

Equational specifications

Definition 6.11. sig $= (S, F, \tau)$ *signature, V system of variables.*

a) *Equation:* $(u, v) \in \mathrm{Term}_s(F, V) \times \mathrm{Term}_s(F, V)$

Write: $u = v$

Equational system E over sig, V: Set of equations E

b) *(Equational)-specification:* spec $=$ (sig, E)

where E is an equational system over $F \cup V$.

Notation

Keyword eqns

spec INT		
sorts int		implicit
ops	$0 :\longrightarrow$ int	All-Quantification
	suc, pred: int $\rightarrow$ int	often also a declaration
eqns	$\mathrm{suc}(\mathrm{pred}(x)) = x$	of the sorts
	$\mathrm{pred}(\mathrm{suc}(x)) = x$	of the variables

Semantics::

- loose all models (PL1)
- tight (special model initial, final)
- operational (equational calculus + induction principle)

Models of spec = (sig, E)

Definition 6.12. $\mathfrak{A}$ *sig-Algebra,* $V(S)$*- system of variables*

a) *Assignment function* φ *for* $\mathfrak{A}$*:* $\varphi_s : V_s \rightarrow A_s$ *induces a valuation* $\varphi : \mathrm{Term}(F, V) \rightarrow \mathfrak{A}$ *through*

$\varphi(f) = f_{\mathfrak{A}}$, f *constant,* $\varphi(x) := \varphi_s(x)$, $x \in V_s$
$\varphi(f(t_1, \ldots, t_n)) = f_{\mathfrak{A}}(\varphi(t_1), \ldots, \varphi(t_n))$

$$\begin{array}{ccc} V_s & \xrightarrow{\varphi_s} & A_s \\ \mathrm{Term}_s(F,V) & \xrightarrow{\varphi_s} & A_s \\ \mathrm{Term}(F,V) & \xrightarrow{\varphi} & \mathfrak{A} \end{array} \quad \textit{homomorphism}$$

(Proof!)

Models of spec = (sig, E)

b) $s = t$ equation over sig, V
$\mathfrak{A} \models_{\varphi} s = t$: $\mathfrak{A}$ satisfies $s = t$ with assignment φ iff $\varphi(s) = \varphi(t)$, equality in A.

c) $\mathfrak{A}$ satisfies $s = t$ or $s = t$ holds in $\mathfrak{A}$
$\mathfrak{A} \models_{\varphi} s = t$: for each assigment φ
$\mathfrak{A} \models s = t$

d) $\mathfrak{A}$ is model of spec = (sig, E)
iff $\mathfrak{A}$ satisfies each equation of E
$\mathfrak{A} \models E$ ALG(spec) class of the models of spec.

Examples

Example 6.13. *1)*

spec	NAT
sorts	nat
ops	$0 :\longrightarrow$ nat
	s : nat $\rightarrow$ nat
	$\_ + \_$: nat, nat $\rightarrow$ nat
eqns	$x + 0 = x$
	$x + s(y) = s(x + y)$

Examples

sig-algebras

a) $\mathfrak{A} = (\mathbb{N}, \hat{0}, \hat{+}, \hat{s})$
$\hat{0} = 0 \quad \hat{s}(n) = n + 1 \quad n \hat{+} m = n + m$

b) $\mathfrak{B} = (\mathbb{Z}, \hat{0}, \hat{+}, \hat{s})$
$\hat{0} = 1 \quad \hat{s}(i) = i \cdot 5 \quad i \hat{+} j = i \cdot j$

c) $\mathfrak{C} = (\{\text{true}, \text{false}\}, \hat{0}, \hat{+}, \hat{s})$
$\hat{0} = \text{false} \quad \hat{s}(\text{true}) = \text{false} \quad \hat{s}(\text{false}) = \text{true}$
$i \hat{+} j = i \vee j$

Examples

$\mathfrak{A}, \mathfrak{B}, \mathfrak{C}$ are models of spec NAT
e.g. $\mathfrak{B}$: $\varphi(x) = a \quad \varphi(y) = b \quad a, b \in \mathbb{Z}$

$$\varphi(x + 0) = a \hat{+} \hat{0} = a \cdot 1 = a = \varphi(x)$$

$$\begin{aligned} \varphi(x + s(y)) &= a \hat{+} \hat{s}(b) = a \cdot (b \cdot 5) \\ &= (a \cdot b) \cdot 5 = \hat{s}(a \hat{+} b) \\ &= \varphi(s(x + y)) \end{aligned}$$

Examples

2)

spec	LIST(NAT)
use	NAT
sorts	nat, list
ops	nil :⟶ list
	_._ : nat, list ⟶ list
	app : list, list ⟶ list
eqns	$\text{app}(\text{nil}, q_2) = q_2$
	$\text{app}(x.q_1, q_2) = x.\,\text{app}(q_1, q_2)$

Examples

spec-Algebra

$\mathfrak{A} \qquad \mathbb{N}, \mathbb{N}^*$
$\hat{0} = 0 \quad \hat{+} = + \quad \hat{s} = +1$
$\hat{\text{nil}} = e \qquad (\text{emptyword})$
$\hat{.}\ (i, z) = i\ z$
$\widehat{\text{app}}(z_1, z_2) = z_1 z_2$ (concatenation)

Examples

3) spec INT $\quad \mathrm{suc}(\mathrm{pred}(x)) = x \quad \mathrm{pred}(\mathrm{suc}(x)) = x$

	1	2	3
A_{int}	$\mathbb{Z}$	$\mathbb{N}$	$\{\mathrm{true}, \mathrm{false}\}$
$0_{\mathfrak{A}_i}$	0	0	true
$\mathrm{suc}_{\mathfrak{A}_i}$	$\mathrm{suc}_{\mathbb{Z}}$	$\mathrm{suc}_{\mathbb{N}}$	$\left\{\begin{array}{c} \mathrm{true} \to \mathrm{false} \\ \mathrm{false} \to \mathrm{true} \end{array}\right\}$
$\mathrm{pred}_{\mathfrak{A}_i}$	$\mathrm{pred}_{\mathbb{Z}}$	$\left\{\begin{array}{c} n+1 \to n \\ 0 \to 0 \end{array}\right\}$	$\left\{\begin{array}{c} \mathrm{true} \to \mathrm{false} \\ \mathrm{false} \to \mathrm{true} \end{array}\right\}$
	$+$	$-$	$+$

Examples

	4	5	6
A_{int}	$\{a, b\}^* \cup \mathbb{Z}$	$\{1\}^+ \cup \{0\}^+ \cup \{z\}$	!
$0_{\mathfrak{A}_i}$	0	z	!
$\mathrm{suc}_{\mathfrak{A}_i}$	$\mathrm{suc}_{\mathbb{Z}}$	$\left\{\begin{array}{c} 1^n \to 1^{n+1} \\ z \to 1 \\ 0^{n+1} \to 0^n \\ 0 \to z \end{array}\right\}$	id
$\mathrm{pred}_{\mathfrak{A}_i}$	$\mathrm{pred}_{\mathbb{Z}}$	$\left\{\begin{array}{c} 1^{n+1} \to 1^n \\ 1 \to z \\ z \to 0 \\ 0^n \to 0^{n+1} \end{array}\right\}$	id
	$-$	$+$	$+$

Substitution

Definition 6.14 (sig, $\mathrm{Term}(F, V)$). *σ:: $\sigma_s : V_s \to \mathrm{Term}_s(F, V)$, $\sigma_s(x) \in \mathrm{Term}_s(F, V)$, $x \in V_s$*
$\sigma(x) = x$ for almost every $x \in V$

$D(\sigma) = \{x \mid \sigma(x) \neq x\}$ finite:: domain of σ

Write $\sigma = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$

Extension to homomorphism $\sigma : \mathrm{Term}(F, V) \to \mathrm{Term}(F, V)$

$$\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$$

Ground substitution: $t_i \in \mathrm{Term}_S(F) \quad x_i \in D(\sigma)_S$

Lose semantics

Definition 6.15. spec = (sig, E)
ALG(spec) $= \{\mathfrak{A} \mid$ *sig-Algebra*, $\mathfrak{A} \models E\}$ *sometimes alternatively*

ALG_{TG}(spec) $= \{\mathfrak{A} \mid$ *term-generated sig-Algebra*, $\mathfrak{A} \models E\}$

Find: Characterizations of equations that are valid in ALG(spec) or $\mathrm{ALG}_{\mathrm{TG}}(\mathrm{spec})$.

a) *Semantical equality:* $E \models s = t$
b) *Operational equality:* $t_1 \underset{E}{\vdash\!\dashv} t_2$ *iff*
There is $p \in 0(t_1), s = t \in E$, *substitution* σ *with*
$t_1|_p \equiv \sigma(s)$, $t_2 \equiv t_1[\sigma(t)]_p(t_1[p \leftarrow \sigma(t)])$
or $t_1|_p \equiv \sigma(t)$, $t_2 \equiv t_1[\sigma(s)]_p$
$t_1 =_E t_2$ *iff* $t_1 \overset{*}{\underset{E}{\vdash\!\dashv}} t_2$
Formalization of replace equals $\leftrightarrow$ *equals*

Equality calculus

c) Equality calculus: Inference rules (deductive)

Reflexivity $\dfrac{}{t = t}$

Symmetry $\dfrac{t = t'}{t' = t}$

Transitivity $\dfrac{t = t', t' = t''}{t = t''}$

Replacement $\dfrac{t' = t''}{s[t']_p = s[t'']_p} \quad p \in 0(s)$

(frequently also with substitution σ)

Equality calculus

$E \vdash s = t$ iff there is a proof P for $s = t$ out of E, i.e.

P = sequence of equations that ends with $s = t$, such that for $t_1 = t_2 \in P$.

i) $t_1 = t_2 \in \sigma(E)$ for a Substitution σ:

ii) $t_1 = t_2 \ldots$ out of precedent equations in P by application of one of the inference rules.

Properties and examples

Consequence 6.16 (Properties and Examples). a) *If either* $E \models s = t$ *or* $s =_E t$ *or* $E \vdash s = t$ *holds, then*

i) *If* σ *is a substitution, then also*

$E \models \sigma(s) = \sigma(t)$ / $\sigma(s) =_E \sigma(t)$ / $E \models \sigma(s) = \sigma(t)$
i.e. the induced equivalence relations on $\mathrm{Term}(F, V)$ *are stable w.r. to substitutions*

ii) $r \in \mathrm{Term}(F, V)$, $p \in 0(r)$, $r|_p$, $s, t \in \mathrm{Term}_{s'}(F, V)$ *then*

$E \models r[s]_p = r[t]_p$ / $r[s]_p =_E r[t]_p$ / $E \vdash r[s]_p = r[t]_p$
replacement property (monotonicity)

$\rightsquigarrow$ *Congruence on* $\mathrm{Term}(F, V)$ *which is stable.*

Congruences / Quotient algebras

b) $\mathfrak{A} = (A, F_{\mathfrak{A}})$ sig-Algebra. $\sim$ bin. relation on A is congruence relation over $\mathfrak{A}$, iff

i) $a \sim b \rightsquigarrow \exists s \in S : a, b \in A_s$ (sort compatible)
ii) $\sim$ is equivalence relation
iii) $a_i \sim b_i$ $(i = 1, \ldots, n)$, $f_{\mathfrak{A}}(a_1, \ldots, a_n)$ defined
$\rightsquigarrow f_{\mathfrak{A}}(a_1, \ldots, a_n) \sim f_{\mathfrak{A}}(b_1, \ldots, b_n)$ (monotonic)

$\mathfrak{A}/\sim$ quotient algebra:
$A/\sim = \bigcup_{s \in S}(A_s/\sim)_s$ with $(A_s/\sim)_s = \{[a]_\sim : a \in A_s\}$ and $f_{\mathfrak{A}/\sim}$ with $f_{\mathfrak{A}/\sim}([a_1], \ldots, [a_n]) = [f_{\mathfrak{A}}(a_1, \ldots, a_n)]$

well defined, i.e. $\mathfrak{A}/\sim$ is sig-Algebra. Abbreviated $\mathfrak{A}_\sim$

$\varphi : \mathfrak{A} \to \mathfrak{A}_\sim$ with $\varphi_s(a) = [a]_\sim$ is a surjective homomorphism, the canonical homomorphism.

Connections between $\models, =_E, \vdash_E$

c) $\mathfrak{A}, \mathfrak{A}'$ sig-algebras $\varphi : \mathfrak{A} \to \mathfrak{A}'$ surjective homomorphism.
Then

$$\mathfrak{A} \models s = t \rightsquigarrow \mathfrak{A}' \models s = t$$

d) spec $= (\text{sig}, E)$:

$$s =_E t \text{ iff } E \vdash s = t$$

e) $\mathfrak{A}$ sig-Algebra, R a sort compatible bin. relation over $\mathfrak{A}$.
Then there is a smallest congruence $\equiv_R$ over $\mathfrak{A}$ that contains R, i.e.
$R \subseteq \equiv_R$

$\equiv_R$ the congruence generated by R

Proofs: Don't give up...

example

spec :: INT with $\text{pred}(\text{suc}(x)) = x$, $\text{suc}(\text{pred}(x)) = x$

$$\begin{aligned}(T_{\text{INT}}/=_E)_{\text{int}} = \quad & \{[0] = \{0, \text{pred}(\text{suc}(0)), \text{suc}(\text{pred}(0)), \ldots \\ & [\text{suc}(0)] = \{\text{suc}(0), \text{pred}(\text{suc}(\text{suc}(0))), \ldots \\ & [\text{suc}(\text{suc}(0))] = \{\cdots \\ & [\text{pred}(0)] = \{\text{pred}(0), \text{suc}(\text{pred}(\text{pred}(0))) \ldots\end{aligned}$$

$$\begin{aligned}\text{suc}_{T_{\text{INT}}/=_E} \quad & ([\text{pred}(\text{suc}(0))]) = [\text{suc}(\text{pred}(\text{suc}(0)))] \\ & = [\text{suc}(0)] \\ & = \text{suc}_{T_{\text{INT}}/=_E}([0])\end{aligned}$$

Connections between $\models, =_E, \vdash_E$

f) $\mathfrak{A}$ sig-Algebra, E equational system over (sig, V).
E induces a relation $\underset{E,\mathfrak{A}}{\sim}$ on $\mathfrak{A}$ where
$a \underset{E,\mathfrak{A},s}{\sim} a'$ $(a, a' \in A_s)$ iff there is $t = t' \in E$ and an assignment
$\varphi : V \to \mathfrak{A}$ with $\varphi(t) = a$, $\varphi(t') = a'$
This relation is sort compatible.
Fact: Let $\equiv$ be a congruence over $\mathfrak{A}$ that contains $\underset{E,\mathfrak{A}}{\sim}$, then $\mathfrak{A}/\equiv$ is a spec $= (\text{sig}, E)$-Algebra, i.e. model of E.

g) Existence: $\mathfrak{A} = T_{\text{sig}}$ the (ground) term algebra, then $=_E$ is on T_{sig} the smallest congruence that contains $\underset{E,\mathfrak{A}}{\sim}$.
In particular $T_{\text{sig}}/=_E$ is a term-generated model of E.

Birkhoff's Theorem

Theorem 6.17 (Birkhoff). *For each specification spec $= (sig, E)$ the following holds*

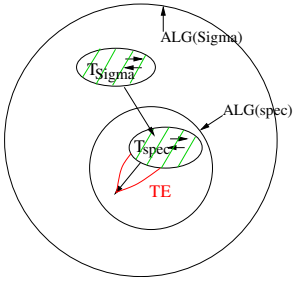
$$E \models s = t \quad \textit{iff} \quad E \vdash s = t \quad (\text{i.e. } s =_E t)$$

Definition 6.18. *Initial semantics*
Let spec $= (sig, E)$, sig strict.
The algebra $T_{sig}/=_E$ (Quotient term algebra)
($=_E$ the smallest congruence relation on T_{sig} generated by E)
is defined as initial algebra semantics of spec $= (sig, E)$.

It is term-generated and initial in $ALG(spec)$!

Initial Algebra semantics

Initial Algebra semantics assigns to each equational specification spec the isomorphism class of the (initial) quotient term algebra $T_{sig}/=_E$.
Write: T_{spec} or $I(E)$



sig $= \Sigma$, spec $= (\Sigma, E)$

Initial algebra

spec $= (\text{sig}, E)$ Initial algebra T_{spec} $\quad (I(E))$

Questions:

- Is T_{spec} computable?
- Is the word problem $(T_{sig}, =_E)$ solvable?
- Is there an "operationalization" of T_{spec}?
- Which (PL1-) properties are valid in T_{spec} ?
- How can we prove this properties? Are there general methods?

Quotient term algebras

Quotient term algebras are ADT.

Example 7.1. *(Continuation)* *spec = INT*

A^i_{int}	$\mathbb{Z}$	$\{true, false\}$	$\{1\}^+ \cup \{0\}^+ \cup \{z\}$
0_{A^i}	0	*true*	*z*
suc_{A^i}	$\text{suc}_{\mathbb{Z}}$	not	...
pred_{A^i}	$\text{pred}_{\mathbb{Z}}$	not	...

$T_{\text{INT}}/=_E$ $\quad [0] \mapsto true \quad [\text{suc}^{2n}(0)] \mapsto true$
$[\text{suc}^{2n+1}(0)] \mapsto false \quad [\text{pred}^{2n+1}(0)] \mapsto false$
$[\text{pred}^{2n}(0)] \mapsto true$

Equational theory / Inductive (equational-) theory

Definition 7.2. *Properties of equations*

a) $TH(E) = \{s = t : E \models s = t\}$ *Equational theory*
Equations that are valid in all spec-algebras.

b) $ITH(E) = \{s = t : T_{spec} \models s = t\}$ *inductive (=)-theory*
Equations that are valid in all term generated spec-algebras.

Equational theory / Inductive (equational-) theory

Consequence 7.3. *Basic properties*

a) $TH(E) \subseteq ITH(E)$, *since* T_{spec} *is a model of* E.

b) *Generally* $TH(E) \subsetneq ITH(E)$
$=$ hence E *is* ω*-complete*
$\rightsquigarrow$ *proofs by consistency inductionless induction*
E *recursively enumerable (r.e.), so* $TH(E)$ *r.e., but* $ITH(E)$ *generally not r.e.*

c) $T_{spec} \models s = t$ *iff* $\sigma(s) =_E \sigma(t)$ *for each ground substitution of the Var. in* s, t. $\rightsquigarrow$ *inductive proof methods, coverset induction*

d) $E : x + 0 = x \qquad x + s(y) = s(x + y)$
$\rightsquigarrow x + y = y + x \in ITH(E) - TH(E)$
$(x + y) + z = x + (y + z)$ *Proof !*

Examples

Example 7.4. *Basic examples*

a) spec	BOOL
sorts	bool
ops	$true, false :\rightarrow$ bool
	not : bool $\rightarrow$ bool
	and, or, impl, eqv : bool, bool $\rightarrow$ bool
	if _ then _ else _ : bool, bool, bool $\rightarrow$ bool

Example (Cont.)

eqns	not(true) = false
	not(false) = true
	and(true, b) = b
	and(false, b) = false
	or(b, b') = not(and(not(b), not(b')))
	impl(b, b') = or(not(b), b')
	eqv(b, b') = and(impl(b, b'), impl(b', b))
	if true b' else b'' = b'
	if false b' else b'' = b''

$(T_{\text{BOOL}})_{\text{bool}} = \{[\text{true}], [\text{false}]\}$ (Proof!)

$\rightsquigarrow$ Defined- and constructor-functions.

Example (Cont.)

b) spec	SET-OF-CHARACTERS
sorts	char, set
ops	$a, b, c, \cdots :\rightarrow$ char
	$\varnothing :\rightarrow$ set
	insert : char, set $\rightarrow$ set
eqns	insert(x, insert(x, s)) = insert(x, s)
	insert(x, insert(y, s)) = insert(y, insert(x, s))

$(T_{\text{soc}})_{\text{char}} = \{a, b, c, \dots\}$
$(T_{\text{soc}})_{\text{set}} = \{[\varnothing], [\text{insert}(a, \varnothing)], \dots$
$\{\varnothing\}\{\text{insert}(a, \text{insert}(a, ..., \text{insert}(a, \varnothing)\}$

Example (Cont.)

c)

spec NAT

sorts nat

ops $0 :\longrightarrow \text{nat}$
$\text{suc} : \text{nat} \longrightarrow \text{nat}$
$\_ + \_, \_ * \_ : \text{nat}, \text{nat} \longrightarrow \text{nat}$

eqns $x + 0 = x$
$x + \text{suc}\, y = \text{suc}(x + y)$
$x * 0 = 0$
$x * \text{suc}(y) = (x * y) + x$

$(T_{\text{NAT}})_{\text{nat}} = \{\ [0, 0 + 0, 0 * 0, \ldots$
$[\text{suc}\, 0, 0 + \text{suc}\, 0, \ldots$
$[\text{suc}(\text{suc}(0)), \ldots$

Example (Cont.)

d) Binary tree

spec BIN-TREE

sorts nat, tree

ops $0 :\longrightarrow \text{nat}$
$\text{suc} : \text{nat} \longrightarrow \text{nat}$
$\text{max} : \text{nat}, \text{nat} \longrightarrow \text{nat}$
$\text{leaf} :\longrightarrow \text{tree}$
$\text{left} : \text{tree} \longrightarrow \text{tree}$
$\text{right} : \text{tree} \longrightarrow \text{tree}$
$\text{both} : \text{tree}, \text{tree} \longrightarrow \text{tree}$
$\text{height} : \text{tree} \longrightarrow \text{nat}$
$\text{dleft} : \text{tree} \longrightarrow \text{tree}$
$\text{dright} : \text{tree} \longrightarrow \text{tree}$

example

Continuation of d) binary tree.

eqns $\max(0, n) = n$
$\max(n, 0) = n$
$\max(\text{suc}(m), \text{suc}(n)) = \text{suc}(\max(m, n))$
$\text{height}(\text{leaf}) = 0$
$\text{height}(\text{both}(t, t')) = \text{suc}(\max(\text{height}(t), \text{height}(t')))$
$\text{height}(\text{left}(t)) = \text{suc}(\text{height}(t))$
$\text{height}(\text{right}(t)) = \text{suc}(\text{height}(t))$

Correctness

Definition 7.5. *A specification* $spec = (sig, E)$ *is sig-correct for a sig-Algebra* $\mathfrak{A}$ *iff* $T_{spec} \cong \mathfrak{A}$ *(i.e. the unique homomorphism is a bijection).*

Example 7.6. *Application:*
INT correct for $\mathbb{Z}$*, BOOL correct for* $\mathbb{B}$

Note: The concept is restricted to initial semantics!

Restrictions/Forgetful functors

Definition 7.7. *Restrictions/Forget-images*

a) *sig* $= (S, F, \tau)$, *sig*$' = (S', F', \tau')$ *signatures with sig* $\subseteq$ *sig*$'$, *i.e.* $(S \subseteq S', F \subseteq F', \tau \subseteq \tau')$.

For each sig$'$*-algebra* $\mathfrak{A}$ *let the sig-part* $\mathfrak{A}|_{sig}$ *of* $\mathfrak{A}$ *be the sig-Algebra with*

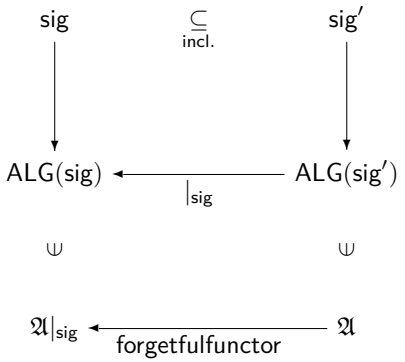
i) $(\mathfrak{A}|_{sig})_s = A_s$ *for* $s \in S$
ii) $f_{\mathfrak{A}|_{sig}} = f_{\mathfrak{A}}$ *for* $f \in F$

Note: $\mathfrak{A}|_{\text{sig}}$ is sig - algebra. The restriction of $\mathfrak{A}$ to the signature sig.

$\mathfrak{A}|_{\text{sig}}$ is also called forget-image of $\mathfrak{A}$ (with respect to sig).

Restrictions/Forgetful functors

$\mathfrak{A}|_{\text{sig}}$ forget-image of $\mathfrak{A}$ (w.r. to sig). The forget image induces consequently a mapping (functor) between classes of algebras in the following way:



Restrictions/Forgetful functor

b) A specification spec $= (\text{sig}', E)$ with sig $\subseteq$ sig$'$ is correct for a sig-algebra $\mathfrak{A}$ iff

$$(T_{\text{spec}})|_{\text{sig}} \cong \mathfrak{A}$$

c) A specification spec$' = (\text{sig}', E')$ implements a specification spec $= (\text{sig}, E)$ iff

$$\text{sig} \subseteq \text{sig}' \text{ and } (T_{\text{spec}'})|_{\text{sig}} \cong T_{\text{spec}}$$

Note:

- A consistency-concept is not necessary for $=$-specification. ((initial) models always exist !).
- The general implementation concept $(Cl(\text{spec}) \subseteq Cl(\text{spec}'))$ reduces here to $=$ of the valid equations in the smaller language. „complete“ theories.

Problems

Verification of $s = t \in Th(E)$ or $\in ITH(E)$.

For $Th(E)$ find $=_E$ an equivalent, convergent term rewriting system (see group example).

For $ITH(E)$ induction's methods:

s, t induce functions to T_{spec}. If $x_1, \ldots, x_n$ are the variables in s and t, types $s_1, \ldots, s_n$.
$s : (T_{\text{spec}})_{s_1} \times \cdots \times (T_{\text{spec}})_{s_n} \to (T_{\text{spec}})_s$
$s = t \in ITh(E)$ iff s and t induce the same functions $\rightsquigarrow$ prove this by induction on the construction of the ground terms.
NAT $0, \text{suc}, +$ $x + y = y + x$ $\in ITH$
$0 + x = x$

Problems

- $0 + 0 = 0$ Ass. : $0 + a = a$
  $0 + Sa =_E S(0 + a) =_I S(a)$
- $x + 0 = 0 + x$ Ass. : $x + a = a + x$
  $x + Sa =_E S(x + a) =_I S(a + x) =_E a + Sx \stackrel{?}{=} Sa + x$
- $x + Sy = Sx + y$
  $x + S0 =_E S(x + 0) =_E Sx =_E Sx + 0$
  $x + SSa =_E S(x + Sa) =_I S(Sx + a) =_E Sx + Sa$

spec(sig, E) — Equations only often do not suffice

P_{spec}(sig, E, $Prop$) — Properties that should hold! ⇝ Verification tasks

Structuring mechanisms

Horizontal: - Decomposition, - Combination,
- Extension, - Instantiation

Vertical: - Realisation, - Information hiding,
- Vertical composition

Here:
Combination, Enrichment, Extension, Modularisation, Parametrisation
⇝ **Reusability.**

Structuring mechanisms

BIN-TREE

1) spec NAT
   sorts nat
   ops $0 :\longrightarrow$ nat
   suc : nat $\longrightarrow$ nat

2) spec NAT1
   use NAT
   ops max : nat, nat $\longrightarrow$ nat
   eqns $\max(0, n) = n$
   $\max(n, 0) = n$
   $\max(s(m), s(n)) = s(\max(m, n))$

Structuring mechanisms

BIN-TREE (Cont.)

3) spec BINTREE1
   sorts bintree
   ops leaf $:\longrightarrow$ bintree
   left, right : bintree $\longrightarrow$ bintree
   both : bintree, bintree $\longrightarrow$ bintree

4) spec BINTREE2
   use NAT1, BINTREE1
   ops height : bintree $\longrightarrow$ nat
   eqns $\vdots$

Combination

Definition 7.8 (Combination). *Let $spec_1 = (sig_1, E_1)$, with $sig_1 = (S_1, F_1, \tau_1)$ be a signature and $sig_2 = [S_2, F_2, \tau_2]$ a triple, E_2 set of equations.*

comb $= spec_1 + (sig_2, E_2)$ *is called combination*
iff
$spec = ((S_1 \cup S_2), (F_1 \cup F_2), (\tau_1 \cup \tau_2)), E_1 \cup E_2)$ *is a specification.*

In particular $((S_1 \cup S_2), (F_1 \cup F_2), (\tau_1 \cup \tau_2))$ is a signature and E_2 contains „syntactically correct“ equations.

The semantics of comb: $T_{\text{comb}} := T_{spec}$

The semantics of comb

$T_{\text{comb}} := T_{\text{spec}}$

Typical cases:

$S_2 = \varnothing$, F_2 new function symbols with arities τ_2 (in old sorts).

S_2 new sorts, F_2 new function symbols.
τ_2 arities in new + old sorts.

E_2 only „new“ equations.

Notations: use, include (protected)

Example

Example 7.9. a) *Step-by-step design of integer numbers*

		semantics
spec	INT1	
sorts	int	$T_{\text{INT1}} \cong (\mathbb{N}, 0, \text{suc}_{\mathbb{N}})$
ops	$0 :\longrightarrow$ int	
	suc : int $\longrightarrow$ int	
	$\cap$	$\cap$
spec	INT2	
use	INT1	$T_{\text{INT2}} \cong (\mathbb{Z}, 0, \text{suc}_{\mathbb{Z}}, \text{pred}_{\mathbb{Z}})$
ops	pred : int $\longrightarrow$ int	
eqns	$\text{pred}(\text{suc}(x)) = x$	
	$\text{suc}(\text{pred}(x)) = x$	

Example (Cont.)

Question: Is the INT1-part of T_{INT2} equal to T_{INT1}??
Does INT2 implement INT1?

$$(T_{\text{INT2}})|_{\text{INT1}} \cong T_{\text{INT1}}$$

$$(\mathbb{Z}, 0, \text{suc}_{\mathbb{Z}}, \text{pred}_{\mathbb{Z}})|_{\text{INT1}}$$
$$\shortparallel$$
$$(\mathbb{Z}, 0, \text{suc}_{\mathbb{Z}}) \quad \ncong \quad (\mathbb{N}, 0, \text{suc}_{\mathbb{N}})$$

Caution: Not always the proper data is specified!
Here new data objects of sort int were introduced.

Example (Cont.)

b) spec NAT2
use NAT
eqns $\mathrm{suc}(\mathrm{suc}(x)) = x$

$$(T_{\mathsf{NAT2}})|_{\mathsf{NAT}} = (\mathbb{N} \bmod 2)|_{\mathsf{NAT}} = \mathbb{N} \bmod 2 \quad \not\cong \quad \mathbb{N} = T_{\mathsf{NAT}}$$

Problem: Adding new or identifying old elements.

Problems with the combination

Let

$$\mathrm{comb} = \mathrm{spec}_1 + (\mathrm{sig}, E)$$

$(T_{\mathrm{comb}})|_{\mathrm{spec}_1}$ is spec_1 Algebra
T_{spec_1} is initial spec_1 algebra
$\Big\} \rightsquigarrow$

$\exists!$ homomorphism $h : T_{\mathrm{spec}_1} \to (T_{\mathrm{comb}})|_{\mathrm{spec}_1}$

Properties of

h: not injective / not surjective / bijective.

e.g. $(T_{\mathsf{BINTREE2}})|_{\mathsf{NAT}} \cong T_{\mathsf{NAT}}$.

Extension and enrichment

Definition 7.10. a) *A combination* $\mathrm{comb} = \mathit{spec}_1 + (\mathit{sig}, E)$ *is an extension iff*

$$(T_{\mathrm{comb}})|_{\mathit{spec}_1} \cong T_{\mathit{spec}_1}$$

b) *An extension is called enrichment when sig does not include new sorts, i.e.* $\mathit{sig} = [\varnothing, F_2, \tau_2]$

- *Find sufficient conditions (syntactical or semantical) that guarantee that a combination is an extension*

Parameterisation

Definition 7.11 (Parameterised Specifications)**.** *A parameterised specification Parameter=(Formal, Body) consist of two specifications: formal and body with formal* $\subseteq$ *body.*

i.e. Formal=(sig_F, E_F)*, Body=*(sig_B, E_B)*, where* $\mathit{sig}_F \subseteq \mathit{sig}_B \qquad E_F \subseteq E_B$.

Notation: Body[Formal]

Syntactically: Body = Formal $+(\mathit{sig}', E')$ *is a combination.*

Note: In general it is not be required that Formal or Body[Formal] have an initial semantics.
It is not necessary that there exist ground terms for all the sorts in Formal.
Only until a concrete specification is "substituted", this requirement will be fulfilled.

Example

Example 7.12. *spec* ELEM $(T_{spec})_{elem} = \varnothing$
sorts elem
ops next : elem → elem

spec STRING[ELEM] $(T_{spec})_{string} = \{[empty]\}$
use ELEM
sorts string
ops empty :→ string
unit : elem → string
concat : string, string → string
ladd : elem, string → string
radd : string, elem → string

Example (Cont.)

eqns concat$(s, \text{empty}) = s$
concat$(\text{empty}, s) = s$
concat$(\text{concat}(s_1, s_2), s_3) = \text{concat}(s_1, \text{concat}(s_2, s_3))$
ladd$(e, s) = \text{concat}(\text{unit}(e), s)$
radd$(s, e) = \text{concat}(s, \text{unit}(e))$

Parameter passing: ELEM → NAT

$$\text{STRING[ELEM]} \to \text{STRING[NAT]}$$

Assignment: formal parameter → current parameter

$$S_F \to S_A$$
$$Op \to Op_A$$

Mapping of the sorts and functions, semantics?

Signature morphisms - Parameter passing

Definition 7.13. a) *Let* $sig_i = (S_i, F_i, \tau_i)$ $i = 1, 2$ *be signatures. A pair of functions* $\sigma = (g, h)$ *with* $g : S_1 \to S_2, h : F_1 \to F_2$ *is a signature morphism, in case that for every* $f \in F_1$

$$\tau_2(hf) = g(\tau_1 f)$$

(g *extended to* $g : S_1^* \to S_2^*$).

In the example g :: elem → nat h :: next → suc
Also $\sigma : sig_{BOOL} \to sig_{NAT}$ *with*

g :: *bool* → nat
h :: *true* → 0 not → suc *and* → plus
false → 0 *or* → times

is a signature morphism.

Signature morphisms - Parameter passing

b) spec = Body[Formal] parameterised specification and *Actual* a standard specification (i.e. with an initial semantics).
A parameter passing is a signature morphism
σ : sig(Formal) → sig(Actual) in which Actual is called the current parameter specification.

(Actual, σ) defines a specification VALUE through the following syntactical changes to Body:

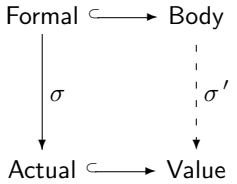
1) Replace Formal with Actual: Body[Actual].
2) Replace in the arities of $op : s_1 \dots s_n \to s_0 \in$ Body, which are not in Formal, $s_i \in$ Formal with $\sigma(s_i)$.
3) Replace in each not-formal equation $L = R$ of Body each $o_P \in$ Formal with $\sigma(o_P)$.
4) Interprete each variable of a type s with $s \in$ Formal as variable of type $\sigma(s)$.
5) Avoid name conflicts between actual and Body/Formal by renaming properly.

Parameter passing

Notation:

$$\text{Value} = \text{Body}[\text{Actual}, \sigma]$$

Consequently for $\sigma : \text{sig}(\text{Formal}) \to \text{sig}(\text{Actual})$ we get a a signature morphism
$\sigma' : \text{sig}(\text{Body}[\text{Formal}]) \to \text{sig}(\text{Body}[\text{Actual}, \sigma]$ with



$$\sigma'(x) = \begin{cases} \sigma(x) & x \in \text{Formal} \\ x' & x \notin \text{Formal} \end{cases}$$

Where x' is a renaming, if there are naming conflicts.

Signature morphisms (Cont.)

Definition 7.14. *Let $\sigma : sig' \to sig$ be a signature morphism.*

Then for each sig-Algebra $\mathfrak{A}$ define $\mathfrak{A}|_\sigma$ a sig'-Algebra, in which for $sig' = (S', F', \tau')$

$(\mathfrak{A}|_\sigma)_s = A_{\sigma(s)}$ $s \in S'$ *and* $f_{\mathfrak{A}|_\sigma} = \sigma(f)_{\mathfrak{A}}$ $f \in F'$.

$\mathfrak{A}|_\sigma$ *is called forget-image of $\mathfrak{A}$ along σ*

Hence $|_\sigma$ is a "mapping" from sig-Algebras into sig'-Algebras.

(Special case: $sig' \subseteq sig :\hookrightarrow$) $|_{sig'}$

Example

Example 7.15. $\mathfrak{A} = T_{\text{NAT}}$ *(with* $0, \text{suc}, \text{plus}, \text{times}$*)*
$sig' = sig(\text{BOOL})$ $sig = sig(\text{NAT})$
$\sigma : sig' \to sig$ *the one considered previously.*

$$\begin{aligned} ((T_{\text{NAT}})|_\sigma)_{\text{bool}} &= (T_{\text{NAT}})_{\sigma(\text{bool})} = (T_{\text{NAT}})_{\text{nat}} \\ &= \{[0], [\text{suc}(0)], \dots\} \end{aligned}$$

$$\begin{array}{lcll} true_{(T_{\text{NAT}})|_\sigma} & = & \sigma(true)_{T_{\text{NAT}}} & = [0] \\ false_{(T_{\text{NAT}})|_\sigma} & = & \sigma(false)_{T_{\text{NAT}}} & = [0] \\ not_{(T_{\text{NAT}})|_\sigma} & = & \sigma(not)_{T_{\text{NAT}}} & = \text{suc}_{T_{\text{NAT}}} \\ and_{(T_{\text{NAT}})|_\sigma} & = & \sigma(and)_{T_{\text{NAT}}} & = \text{plus}_{T_{\text{NAT}}} \\ or_{(T_{\text{NAT}})|_\sigma} & = & \sigma(or)_{T_{\text{NAT}}} & = \text{times}_{T_{\text{NAT}}} \end{array}$$

Forget images of homomorphisms

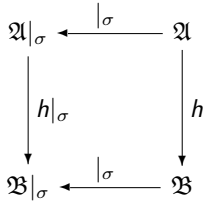
Definition 7.16. *Let $\sigma : sig' \to sig$ a signature morphism, $\mathfrak{A}, \mathfrak{B}$ sig-algebras and $h : \mathfrak{A} \to \mathfrak{B}$ a sig-homomorphism, then*

$h|_\sigma := \{h_{\sigma(s)} \mid s \in S'\}$ *(with $sig' = (S', F', \tau')$) is a sig'-homomorphism from $\mathfrak{A}|_\sigma \to \mathfrak{B}|_\sigma$ by setting*

$$\begin{array}{rccc} (h|_\sigma)_s = h_{\sigma(s)} : & A_{\sigma(s)} & \to & B_{\sigma(s)} \\ & \| & & \| \\ & (A|_\sigma)_s & \to & (B|_\sigma)_s \end{array}$$

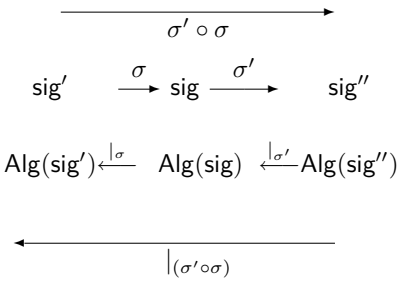
$h|_\sigma$ is called the forget image of h along σ

Forgetful functors



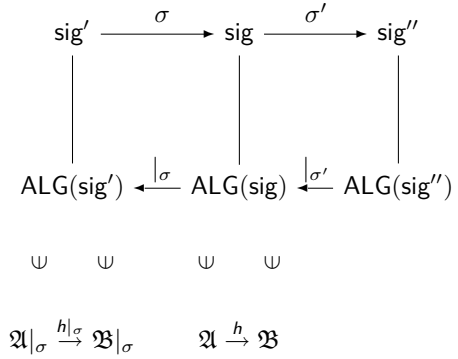
Forgetful functors

Let $\sigma : \text{sig}' \to \text{sig}$, $\mathfrak{A}, \mathfrak{B}$, sig-algebras, $h : \mathfrak{A} \to \mathfrak{B}$, sig-homomorphism.
$h|_\sigma = \{h_{\sigma(s)} \mid s \in S'\}$, $\text{sig}' = (S', F', \tau')$, with
$h|_\sigma : A|_\sigma \to B|_\sigma$ forget image of h along σ.



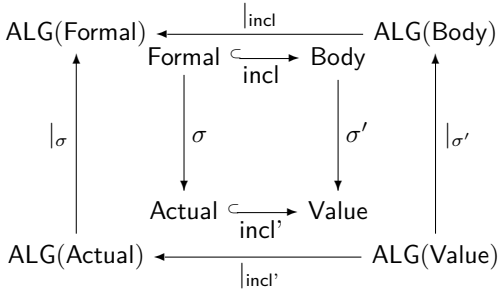
Forgetful functors

Properties of $h|_\sigma$ (forget image of h along σ)



Compatible with identity, composition and homomorphisms.

Parameter Specification *Body*[*Formal*]



Semantics of parameter passing (only signature)

Definition 7.17. *Let Body[Formal] be a parameterized specification.* σ : Formal $\to$ Actual *signature morphism.*

Semantics of the the "instantiation" i.e. parameter passing [Actual, σ].

$$\sigma : \text{Formal} \to \text{Actual}$$
$$\downarrow$$
initial semantics of value. i. e.
$$T_{\text{Body[Actual},\sigma]}$$

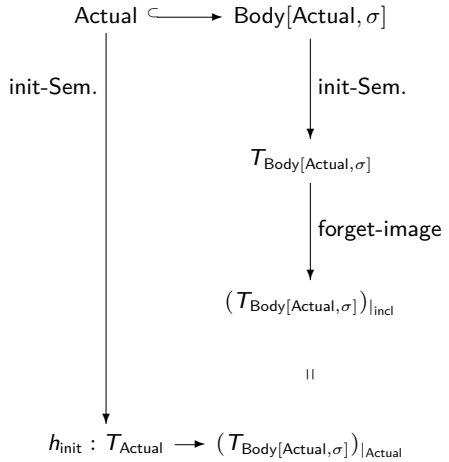
Can be seen as a mapping : $S ::(T_{\text{Actual}}, \sigma) \mapsto T_{\text{Body[Actual},\sigma]}$

This mapping between initial algebras can be interpreted as correspondence between formal algebras $\to$ body-algebras.

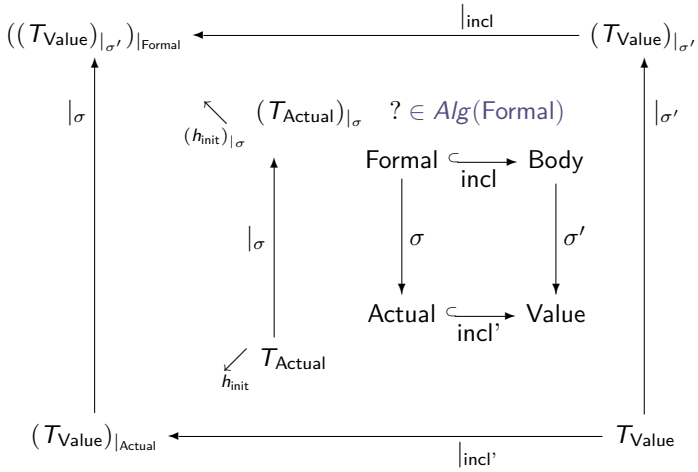
$$(T_{\text{Actual}})|_\sigma \mapsto (T_{\text{Body[Actual},\sigma]})|_{\sigma'}$$

Semantics parameter passing

$(T_{\text{Actual}})|_\sigma \mapsto (T_{\text{Body[Actual},\sigma]})|_{\sigma'}$



Mapping between initial algebras



Properties of the signature morphism

Formal		$\xrightarrow{\sigma}$	Actual	
sorts	elem	elem $\to$ nat	sorts	nat
ops	$a, b :\to$ elem	$a \to 0$	ops	$0, 1 :\to$ nat
eqns	$a = b$	$b \to 1$	eqns	

$\mathfrak{A} = T_{\text{Actual}}$ $A_{\text{nat}} = \{0, 1\}$

$\mathfrak{A}|_\sigma \in \text{Alg}(\text{sig Formal})$ $(A|_\sigma)_{\text{elem}} = \{0, 1\}$

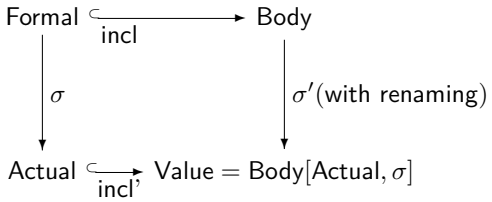
$a|_{\mathfrak{A}|_\sigma} = 0 \neq 1 = b|_{\mathfrak{A}|_\sigma}$

Equation from Formal is not fulfilled! i.e. $\mathfrak{A}|_\sigma \notin \text{Alg(Formal)}$.

Parameter passing (Actual, σ)

Body[Formal]

$$\sigma : \text{sig(Formal)} \to \text{sig(Actual)}$$
signature morphism



Precondition: sig(Actual) and sig(Value) strict.

Parameter passing (Actual, σ)

Forgetful functor: $|_\sigma : \text{Alg(sig)} \to \text{Alg(sig}')$

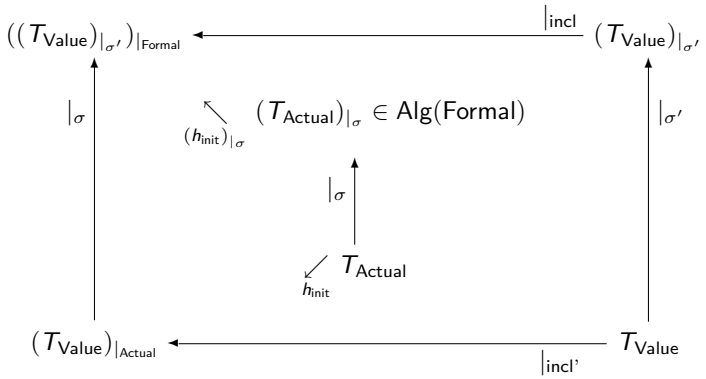
$$\mathfrak{A}|_\sigma \text{ for } \sigma : \text{sig}' \to \text{sig}$$

$h : \mathfrak{A} \to \mathfrak{B}$ sig-homomorphism

$$h|_\sigma : \mathfrak{A}|_\sigma \to \mathfrak{B}|_\sigma$$

sig′-homomorphism

Parameter passing (Actual, σ)



Problems: 1) $(T_{\text{Actual}})|_\sigma \notin \text{Alg(Formal)}$, 2) h_{init} is not a bijection.

Specification morphisms

Definition 7.18. *Let $spec' = (sig', E'), spec = (sig, E)$ (general) specifications.*
A signature morphism $\sigma : sig' \to sig$ is called a specification morphism, if $\sigma(s) = \sigma(t) \in Th(E)$ for every $s = t \in E'$ holds.

Write: $\sigma : spec' \to spec$

Fact: If $\mathfrak{A} \in \text{Alg}(spec)$ then $\mathfrak{A}|_\sigma \in \text{Alg}(spec')$
i.e. $|_\sigma : \text{Alg}(spec) \to \text{Alg}(spec')$*!*

Often „only"the weaker condition $\sigma(s) = \sigma(t) \in ITh(E)$ is demanded in above definition. More spec morphisms!

Semantically correct parameter passing

Definition 7.19. *A parameter passing for Body[Formal] is a pair* (Actual, σ)*: Actual an equational specification and* σ : Formal $\to$ Actual *a specification morphism.*

Hence:: $(T_{\text{Actual}})|_\sigma \in \text{Alg}(\text{Formal})$

- Demand also h_{init} bijection. Proof tasks become easier.

There are syntactical restrictions that guarantee this.

Algebraic Specification languages

CLEAR, Act-one, -Cip-C, Affirm, ASL, Aspik, OBJ, ASF, $\overset{\rightsquigarrow}{+}$ newer languages: - Spectrum, - Troll.

Example

Example 7.20.

Formal ::
spec	ELEMENT
use	BOOL
sorts	elem
ops	$. \leq . :$ elem, elem $\to$ bool
eqns	$x \leq x = \mathit{true}$
	$\text{imp}(x \leq y \ \mathit{and}\ y \leq z, x \leq z) = \mathit{true}$
	$x \leq y \ \mathit{or}\ y \leq x = \mathit{true}$

Example (Cont.)

spec	LIST[ELEMENT]
use	ELEMENT
sorts	list
ops	nil :$\to$ list
	. : elem, list $\to$ list
	insert : elem, list $\to$ list
	insertsort : list $\to$ list
	case : bool, list, list $\to$ list
	sorted : list $\to$ bool

Example (Cont.)

eqns	$\text{case}(\text{true}, l_1, l_2) = l_1$
	$\text{case}(\text{false}, l_1, l_2) = l_2$
	$\text{insert}(x, \text{nil}) = x.\text{nil}$
	$\text{insert}(x, y.l) = \text{case}(x \leq y, x.y.l, y.\,\text{insert}(x, l))$
	$\text{insertsort}(\text{nil}) = \text{nil}$
	$\text{insertsort}(x.l) = \mathit{insert}(x, \mathit{insertsort}(l))$
	$\text{sorted}(\text{nil}) = \text{true}$
	$\text{sorted}(x.\text{nil}) = \text{true}$
	$\text{sorted}(x.y.l) = \text{if } x \leq y \text{ then sorted}(y.l) \text{ else false}$

Property: $\text{sorted}(\text{insertsort}(l)) = \text{true}$

Example (Cont.)

ACTUAL $\equiv$ BOOL

σ : elem $\longrightarrow$ bool, bool $\longrightarrow$ bool
. $\leq$. $\longrightarrow$ impl

The equations of ELEMENT are in Th(BOOL)

$\rightsquigarrow$ Specification morphism

Example (Cont.)

ACTUAL $\equiv$ NAT

σ : bool $\longrightarrow$ nat elem $\longrightarrow$ nat
true $\longrightarrow$ suc(0) not allowed
false $\longrightarrow$ 0
not $\longrightarrow$ suc
or $\longrightarrow$ plus
and $\longrightarrow$ times
$\vdots$
. $\leq$. $\longrightarrow \cdots$

is not a specification morphism
not(false) = true
not(true) = false does not hold!.