Formal Specification and Verification Techniques

Prof. Dr. K. Madlener

12. Februar 2009

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

Abstract State Machines (ASMs)

Algebraic Specification

Algebraic Specification: Initial Semantics

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

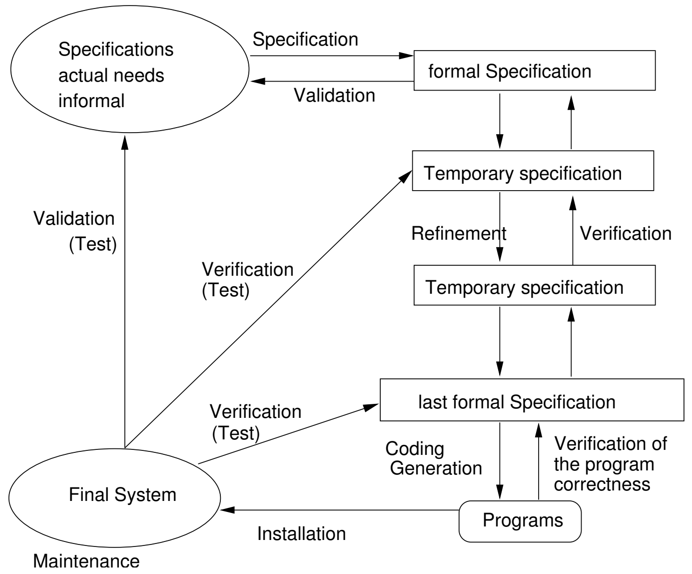
  Goal: The group of documents that includes an executable program.

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)
			models
			algebraic specification
Concept of correctness			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* $\{true, false\}$, $\mathbb{N}$ *natural* $\{0, 1, 2, 3, \dots\}$, $\mathbb{Z}, \mathbb{R}$
- *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

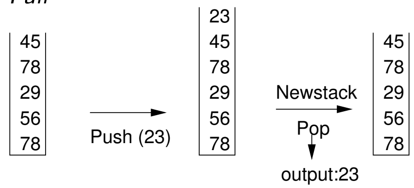
Format:

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

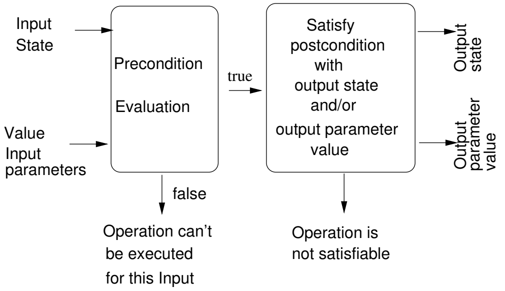
Full( )$b : \mathbb{B}$
ext rd s : Contents
rd n : Max_Stack_Size
pre true
post $b \Leftrightarrow (\text{len } s = n)$

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:

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

alternatively

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

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 : → stack "Constant of the type stack, empty stack"
push : stack nat → stack
pop : stack → stack
top : stack → nat
is_empty? : stack → bool
stack_error : → stack
nat_error : → 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 $\equiv$ 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 $\longrightarrow$ 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.

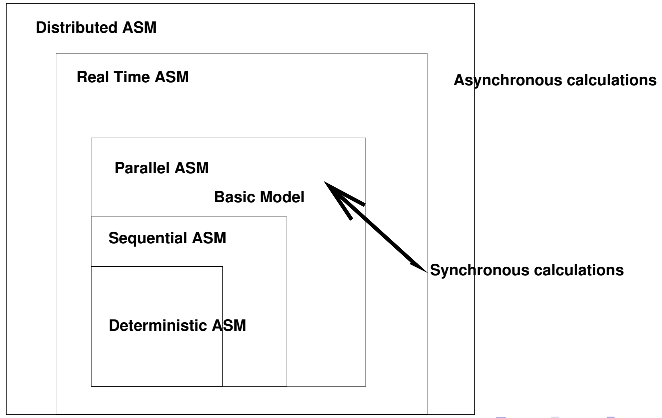
Problem: Simulation is not easy, because there are different abstraction levels of the manipulated objects and different granularity of the steps.

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.

⇝ 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 $\tau_A : S(A) \rightarrow S(A)$, 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 $Edge(x, y) \wedge R(x) \wedge \neg R(y) \qquad R(y) := true$

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

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

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:*

$$\Delta(A, X) \leftrightharpoons \tau_A(X) - X \qquad i.e.\ \tau_A(X) = X + \Delta(A, X)$$

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 $(f, \bar{a})$ of X is reachable when each element in the tuple $\bar{a}$ is reachable.
An update $(f, \bar{a}, b)$ of X is reachable when $(f, \bar{a})$ 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
  .   *Notation: Block (when empty skip)*
  .
  .
  R_k
endpar   *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*

$$\Delta(R,X) \risingdotseq \{(f,(a_1,...,a_j),a_0)\}$$

If R is a par-update rule with components $R_1,...R_k$ then set

$$\Delta(R,X) \risingdotseq \Delta(R1,X) \cup \cdots \cup \Delta(Rk,X).$$

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 $E_X(t1,t2) \risingdotseq Val(t1,X) = Val(t2,X)$ 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 (i.e. containing ground equations, not, and, or) 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 $\tau_\Pi(X) \risingdotseq X + \Delta(\Pi,X)$.*

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 $\{0,1,...,n-1\} \rightarrow \mathbb{R}$ and $i,j,k \in \{0,1,...,n\}$.*
For $i \le j$: $S(i,j) \fallingdotseq \sum_{i \le k < j} A(k)$. In particular $S(i,i) = 0$.

Problem: Compute $S \fallingdotseq max_{i \le j} S(i,j)$.

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

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

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

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

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

$$\begin{array}{l} \textit{if} \quad k \neq n \quad \textit{then} \\ \quad \textit{par} \\ \qquad x := max\{x + A(k), 0\} \\ \qquad y := max\{y, x + A(k)\} \\ \qquad k := k + 1 \\ \textit{else} \quad S := y \end{array}$$

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

 1

Part 1

Abstract states and update sets

 2

Signatures

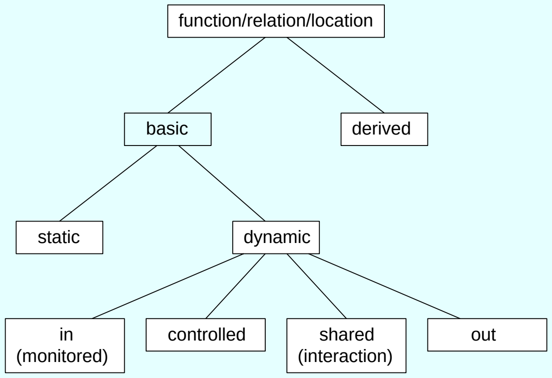
Definition. A *signature* Σ is a finite collection of function names.

- 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*.

 3

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 $f^{\mathfrak{A}}: X^n \to X$.
- 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

$$\boxed{(f, (a_1, \ldots, a_n))}$$

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 $v = w$.

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 $\mathfrak{B}+\alpha(U)$.

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.

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 $t_1, \dots, t_n$ are terms, then $f(t_1, \dots, t_n)$ 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* $[\![t]\!]_\zeta^\mathfrak{A}$ is defined as follows:

- $[\![x]\!]_\zeta^\mathfrak{A} = \zeta(x)$
- $[\![c]\!]_\zeta^\mathfrak{A} = c^\mathfrak{A}$
- $[\![f(t_1, \dots, t_n)]\!]_\zeta^\mathfrak{A} = f^\mathfrak{A}([\![t_1]\!]_\zeta^\mathfrak{A}, \dots, [\![t_n]\!]_\zeta^\mathfrak{A})$

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 $(\varphi \to \psi)$ 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)
$\to$	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 $((\varphi \vee \psi) \vee \chi)$,
$\varphi \wedge \psi \to \chi$ stands for $((\varphi \wedge \psi) \to \chi)$, 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

$$\llbracket s = t \rrbracket_\zeta^{\mathfrak{A}} = \begin{cases} \mathit{true}, & \text{if } \llbracket s \rrbracket_\zeta^{\mathfrak{A}} = \llbracket t \rrbracket_\zeta^{\mathfrak{A}}; \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

$$\llbracket \neg\varphi \rrbracket_\zeta^{\mathfrak{A}} = \begin{cases} \mathit{true}, & \text{if } \llbracket \varphi \rrbracket_\zeta^{\mathfrak{A}} = \mathit{false}; \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

$$\llbracket \varphi \wedge \psi \rrbracket_\zeta^{\mathfrak{A}} = \begin{cases} \mathit{true}, & \text{if } \llbracket \varphi \rrbracket_\zeta^{\mathfrak{A}} = \mathit{true} \text{ and } \llbracket \psi \rrbracket_\zeta^{\mathfrak{A}} = \mathit{true}; \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

$$\llbracket \varphi \vee \psi \rrbracket_\zeta^{\mathfrak{A}} = \begin{cases} \mathit{true}, & \text{if } \llbracket \varphi \rrbracket_\zeta^{\mathfrak{A}} = \mathit{true} \text{ or } \llbracket \psi \rrbracket_\zeta^{\mathfrak{A}} = \mathit{true}; \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

$$\llbracket \varphi \to \psi \rrbracket_\zeta^{\mathfrak{A}} = \begin{cases} \mathit{true}, & \text{if } \llbracket \varphi \rrbracket_\zeta^{\mathfrak{A}} = \mathit{false} \text{ or } \llbracket \psi \rrbracket_\zeta^{\mathfrak{A}} = \mathit{true}; \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

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

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

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 $\llbracket \varphi \rrbracket_\zeta^{\mathfrak{A}} = \llbracket \varphi \rrbracket_\eta^{\mathfrak{A}}$.

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

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

Models

Definition. A state $\mathfrak{A}$ is a *model* of φ (written $\mathfrak{A} \models \varphi$), if $\llbracket \varphi \rrbracket_\zeta^{\mathfrak{A}} = \mathit{true}$ 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: $f(s_1, \dots, s_n) := t$

Meaning: Update the value of f at $(s_1, \dots, s_n)$ 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: $r(t_1, \dots, t_n)$

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

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** $\dots$ **par** P_n
$\{P_1, \dots, P_n\}$	P_1 **par** $\dots$ **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 $a : Index \to Value$

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

Two kinds of non-determinisms:

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

$\textit{choose} \quad x \in \{x_1, x_2, \ldots, x_n\} \ \textit{with} \ \varphi(x) \ \textit{do}$
$\qquad R(x)$

"Don't-know" indeterminism

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

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

Abstract State Machines: ASM- Specification's method
ASM-Specifications

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.

$$\boxed{\textbf{let } x = t \ \underbrace{\textbf{in } P}_{\text{scope of } x}}$$

$$\boxed{\textbf{forall } x \ \underbrace{\textbf{with } \varphi \textbf{ do } P}_{\text{scope of } x}}$$

$$\boxed{\textbf{choose } x \ \underbrace{\textbf{with } \varphi \textbf{ do } P}_{\text{scope of } x}}$$

 28

Abstract State Machines: ASM- Specification's method
ASM-Specifications

Rule declarations

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

$$\boxed{r(x_1, \ldots, x_n) = P}$$

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.

 29

Abstract State Machines: ASM- Specification's method
ASM-Specifications

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.

 30

Semantics of transition rules

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

$$\boxed{\frac{\mathit{Premise}_1 \quad \cdots \quad \mathit{Premise}_n}{\mathit{Conclusion}}\ \mathit{Condition}}$$

The predicate

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

means:

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

Semantics of transition rules (continued)

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

$$\frac{}{\text{yields}(f(s_1, \ldots, s_n) := t, \mathfrak{A}, \zeta, \{(l, v)\})} \quad \text{where } l = (f, (\llbracket s_1 \rrbracket_\zeta^\mathfrak{A}, \ldots, \llbracket s_n \rrbracket_\zeta^\mathfrak{A})) \text{ and } v = \llbracket t \rrbracket_\zeta^\mathfrak{A}$$

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

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta, U)}{\text{yields}(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q, \mathfrak{A}, \zeta, U)} \quad \text{if } \llbracket \varphi \rrbracket_\zeta^\mathfrak{A} = \mathit{true}$$

$$\frac{\text{yields}(Q, \mathfrak{A}, \zeta, V)}{\text{yields}(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q, \mathfrak{A}, \zeta, V)} \quad \text{if } \llbracket \varphi \rrbracket_\zeta^\mathfrak{A} = \mathit{false}$$

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

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

Semantics of transition rules (continued)

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

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

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

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

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

$$\boxed{\mathit{range}(x, \varphi, \mathfrak{A}, \zeta) = \{a \in |\mathfrak{A}| : \llbracket \varphi \rrbracket_{\zeta[x \mapsto a]}^\mathfrak{A} = \mathit{true}\}}$$

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 $a = \llbracket t \rrbracket_\zeta^\mathfrak{A}$. 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:

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

 35

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.

 36

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 → REAL, frontier : EDGE → Bool, tree : EDGE → 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

$$\begin{array}{lll} (P1) & x\,\rho\,x & (\text{reflexive}) \\ (P2) & (x\,\rho\,y \wedge y\,\rho\,x) \rightarrow x = y & (\text{antisymmetric}) \\ (P3) & (x\,\rho\,y \wedge y\,\rho\,z) \rightarrow x\,\rho\,z & (\text{transitive}) \\ (P4) & (x\,\rho\,y \vee y\,\rho\,x) & (\text{linear}) \end{array}$$

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: $< \; = \; \leq \setminus \mathrm{id}_X$
- 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*
  - *$D = X \nrightarrow Y$: set of all the partial functions f with $\mathrm{dom}(f) \subseteq X$ and $\mathrm{cod}(f) \subseteq Y$.*
  - *Let $f, g \in X \nrightarrow Y$.*

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

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:

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

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. $\mathrm{sec}(y) \subseteq X, y \notin X$
4. Contradiction

Well-founded induction

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

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

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. $\mathrm{sec}(z) \subseteq X, z \notin X$
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.

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

Problem

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

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

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

Consider

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

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

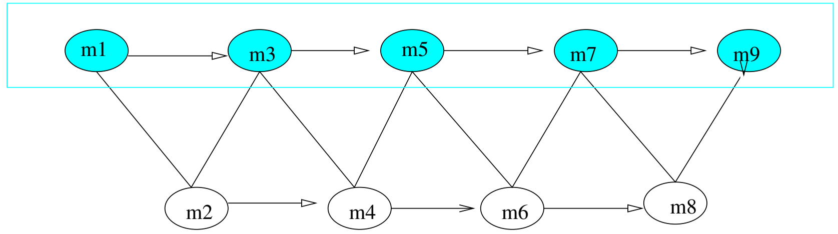
A run of a distributed ASM A is given through a triple $\varrho \rightleftharpoons (M, \lambda, \sigma)$ with the following properties:

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

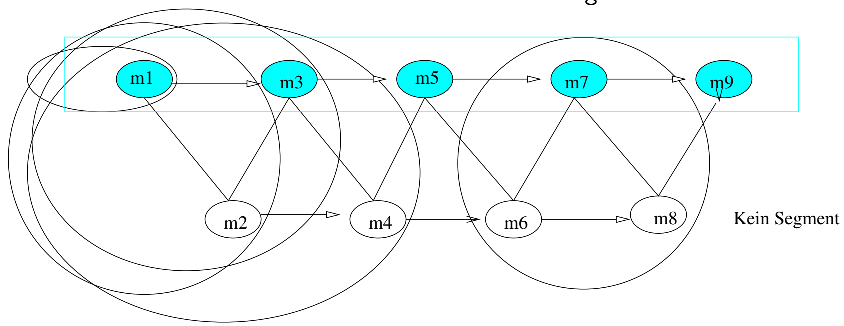
The agents of A modell the concurrent control-threads in the execution of Π_A.
A run can be seen as the common part of the history of the same computation from the point of view of multiple observers.

The role of λ:



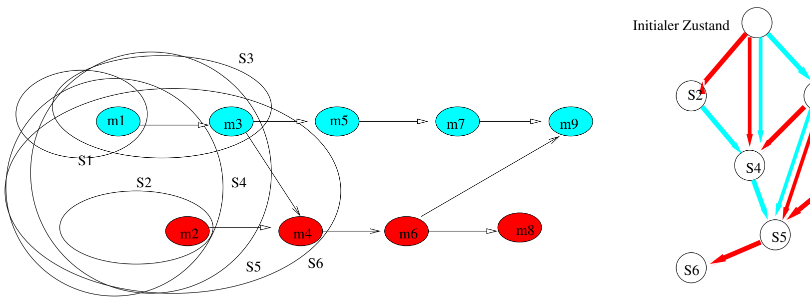
Comment, example (cont.)

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:

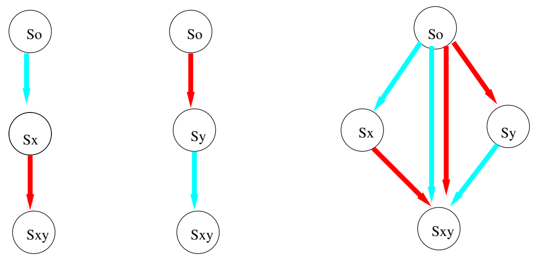
$program_d = door := true$ // move x
$program_w = window := true$ // move y

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)

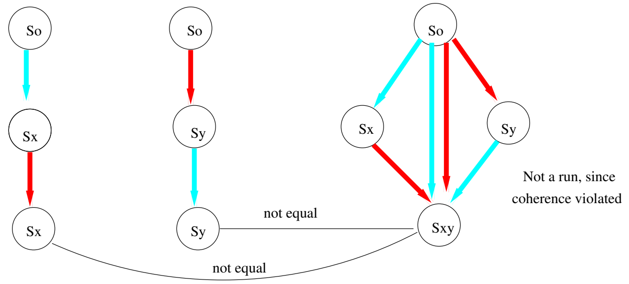


Variants of simple example

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

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

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



More variations

Exercise 4.9. *Consider the following pair of agents* $x, y \in \mathbb{N}$ $\quad (x = 2, y = 1$ *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. $Program_d = \ \ if\ \neg window\ \ then\ \ door := true$ //move x
2. $Program_w = \ \ if\ \neg door\ \ then\ \ window := true$ //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:

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

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

$$\mathit{monitored}\ \ \mathit{now} :\to \mathit{TIME}$$

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.*

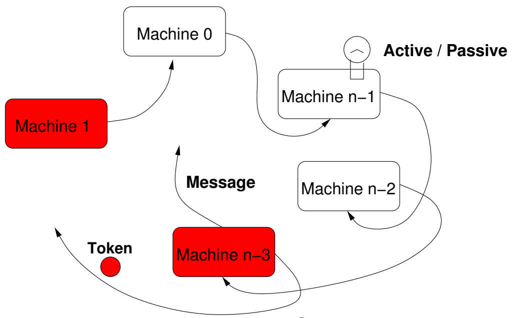
Implement an asynchronous communication model in which timeouts can cancel transactions .

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

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

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

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* =

$$
\begin{array}{l}
ReactOnEvents(me) \\
\text{if} \neg \; Active(me) \wedge \; token(me) \neq \; undef \;\; \text{then} \\
\quad \text{if} \;\; color(me) = \; white \wedge \; token(me) = \; whiteToken \;\; \text{then} \\
\qquad ReportGlobalTermination \\
\quad \text{else} \quad \text{Rule 3} \\
\qquad InitializeMachine(me) \quad \text{Rule 4} \\
\qquad Forward(me, whiteToken) \quad \text{Rule 4}
\end{array}
$$

Distributed Termination Detection

Initial states

$$
\begin{array}{l}
\exists m_0 \in MACHINE \\
\quad (program(m_0) = SupervisorMachineProgram \wedge \\
\quad token(m_0) = redToken \wedge \\
\quad (\forall m \in MACHINE)(m \neq m_0 \Rightarrow \\
\qquad (program(m) = RegularMachineProgram \wedge token(m) = undef)))
\end{array}
$$

Environment constraints For all the executions and all linearizations holds:

$$
\begin{array}{l}
\mathbf{G} \; (\forall m \in MACHINE) \\
\qquad (SendMessageEvent(m) = true \Rightarrow (\mathbf{P}(Active(m)) \;\; \wedge Active(m))) \\
\quad \wedge \;\; ((Active(m) = true \wedge \mathbf{P}(\neg Active(m)) \Rightarrow \\
\qquad (\exists m' \in MACHINE) \;\; (m' \neq m \wedge \;\; SendMessageEvent(m'))))
\end{array}
$$

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)

$\vdots$

Algebraic Specification - Algebras

Specification of data types

Syntax	Equations	Programs
$\left\{ \begin{matrix} \text{signature} \\ \text{axiom} \end{matrix} \right\}$	$\left\{ \begin{matrix} t_1 = t_2 \\ \text{if } \varphi \text{ then } t_1 = t_2 \end{matrix} \right\}$	$\left\{ \begin{matrix} \text{data operations} \\ \text{directed application} \end{matrix} \right\}$

Algebras

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

Single-Sorted Algebras

Example 6.1. *a) Groups*
SORT:: g
SIG:: $\cdot : g, g \rightarrow g \qquad 1 : \rightarrow g \qquad ^{-1} : g \rightarrow g$
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 $\underset{EQN}{=} \; = \; \overset{*}{\underset{R}{\Longleftrightarrow}}$

$$\begin{array}{ll}
x \cdot 1 \rightarrow x & 1 \cdot x \rightarrow x \\
x \cdot x^{-1} \rightarrow 1 & x^{-1} \cdot x \rightarrow 1 \\
1^{-1} \rightarrow 1 & (x^{-1})^{-1} \rightarrow x \\
(x \cdot y)^{-1} \rightarrow y^{-1} \cdot x^{-1} & (x \cdot y) \cdot z \rightarrow x \cdot (y \cdot z) \\
x^{-1} \cdot (x \cdot y) \rightarrow y & x \cdot (x^{-1} \cdot y) \rightarrow y
\end{array}$$

Many-Sorted Algebras

b) Lists over nat-numbers

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

Many-Sorted Algebras

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

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

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

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

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

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

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.:

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

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

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

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.

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

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.

$x + y = y + x :: s^i0 + s^j0 = s^j0 + s^i0$ 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 correctness:

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

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

Consequence closure

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

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

Important concepts:

Consistency: $A \subsetneq L$ $\quad$ A is consistent if $Cl(A) \subsetneq L$
Implementation: A (over L') implements B (over L) $\quad$ (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: $f : s_1, \dots, s_n \to s$

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

Signature - Terms

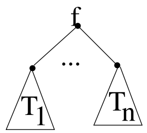
b) Term(F): Set of ground terms over sig and their tree presentation.

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

recursive definition:

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

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

Quotation: terms over sig in the variables V.
(F and τ extended with the set of variables and their sorts).

Intention: for variables it is allowed to use any object of the same sort, i.e. terms of this sort. "Placeholder" 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 for each sort $s \in S$ there is a constant of sort S or a function $f : s_1, \dots, s_n \to s$, so that the s_i are strict. If all the sorts of the signature are strict.$\leadsto$ strict signatures (general assumption)

b) Subterms $(t) = \{t_p \mid p$ location (position) in p, t_p subterm in $p\}$
The positions are represented by 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: $O(t) = \{e\}$ $t_e \equiv t$*
- *$t \equiv f(t_1, \dots, t_n)$ so*
  $O(t) = \{ip \mid 1 \leq i \leq n, p \in O(t_i)\} \cup \{e\}$
  $t_{ip} \equiv t_i|_p$ and $t_e \equiv t$.

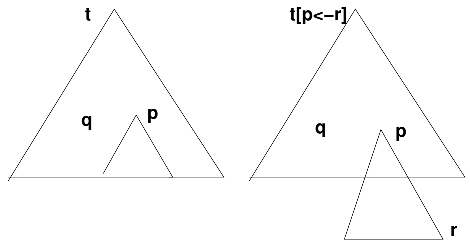
Term replacement

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 by replacing 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. $S = (\mathrm{BOOL}, \mathrm{NAT}, \mathrm{LIST})$, $F = \{true, false, \dots\}$,
$\tau : F \to S^* :: true :\to \mathrm{BOOL}$, $eq : \mathrm{NAT}, \mathrm{NAT} \to \mathrm{BOOL}, \dots$

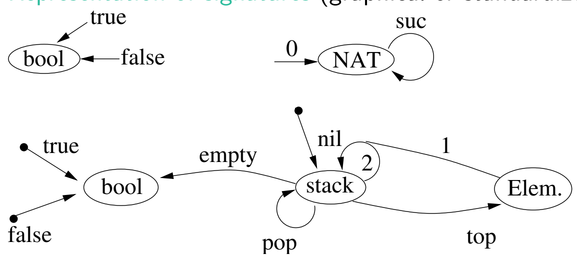
$$V = \underset{\{b_i : i \in \mathbb{N}\}}{\underset{``}{V_{\mathrm{BOOL}}}} \cup \underset{\{x_i : i \in \mathbb{N}\}}{\underset{``}{V_{\mathrm{NAT}}}} \cup \underset{\{l_i : i \in \mathbb{N}\}}{\underset{``}{V_{\mathrm{LIST}}}}$$

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:
<u>sig</u> ...
<u>sorts</u> ...
<u>ops</u> ...
op: $W \to S$
$\mathrm{op}_1, \dots, \mathrm{op}_i : W \to S$

Interpretations: sig-Algebras

Definition 6.5. 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* $f_{\mathfrak{A}} : A_{s_1} \times \dots \times A_{s_n} \to A_s$ *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* $:\rightarrow$ BOOL

$$\left.\begin{array}{llll} \mathfrak{A}_1 & \{0,1\} & true_{\mathfrak{A}_1}=0 & false_{\mathfrak{A}_1}=1 \\ \mathfrak{A}_2 & \{0,1\} & true_{\mathfrak{A}_2}=0 & false_{\mathfrak{A}_2}=0 \\ \mathfrak{A}_3 & \mathbb{N} & true_{\mathfrak{A}_3}=4 & false_{\mathfrak{A}_3}=5 \\ \mathfrak{A}_4 & \{true, false\} & true_{\mathfrak{A}_4}=true & false_{\mathfrak{A}_4}=false \end{array}\right\} \textit{bool-Alg.}$$

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

$$\left\{\begin{array}{cccccc} A_{i_{\mathrm{NAT}}} & \mathbb{N} & \mathbb{Z} & \mathbb{N} & \{true, false\} & \{0, \mathrm{suc}^i(0)\} \\ 0_{\mathfrak{A}_i} & 0 & 0 & 1 & true & 0 \\ \mathrm{suc}_{\mathfrak{A}_i} & \mathrm{suc}_{\mathbb{N}} & \mathrm{pred}_{\mathbb{Z}} & \mathrm{id}_{\mathbb{N}} & \mathrm{suc}(true)=false & \mathrm{suc}(0)=suc(0) \\ & & & & \mathrm{suc}(false)=true & \mathrm{suc}(\mathrm{suc}^i(0))=\mathrm{suc}^{i+1}(0) \end{array}\right.$$

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. $A = \mathrm{Term}(F, V)$
  $F \ni f : s_1, \ldots, s_n \rightarrow s,\ f_{\mathfrak{A}}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$

  $\mathfrak{A}$ *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*
$h : \mathfrak{A} \rightarrow \mathfrak{A}'$ *family of functions*
$h = \{h_s : A_s \rightarrow A'_s : s \in S\}$ *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

$$\begin{tikzcd} \cdot \arrow[r, "f_{\mathfrak{A}}"] \arrow[d, "h"'] & \cdot \arrow[d, "h"] & \text{Algebra } \mathfrak{A} \\ \cdot \arrow[r, "f_{\mathfrak{A}'}"] & \cdot & \text{Algebra } \mathfrak{A}' \end{tikzcd}$$

Canonical homomorphisms

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

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

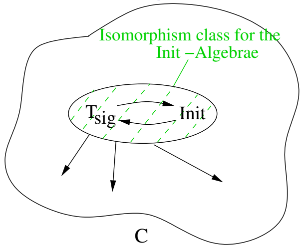
Proof: Just try!!

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, $h : T_{\text{sig}} \to \mathfrak{A}$ interpretation homomorphism.
$\mathfrak{A}$ sig-generated (term-generated) iff
$\forall s \in S \quad h_s : \text{Term}_s(F) \to A_s$ 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 $0 :\to$ int
suc : int $\to$ int
pred : int $\to$ int

Equational specifications

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

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

Write: $u = v$

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

b) *(Equational)-specification: spec $= (\text{sig}, E)$*

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

Notation

Keyword eqns

spec INT	
sorts int	implicit
ops $0 :\to$ int	All-Quantification
suc, pred: int $\to$ int	often also a declaration
eqns $\text{suc}(\text{pred}(x)) = x$	of the sorts
$\text{pred}(\text{suc}(x)) = x$	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 $\mathfrak{A}$: $\varphi_s : V_s \to A_s$ induces a valuation $\varphi : \text{Term}(F, V) \to \mathfrak{A}$ through*

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

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

(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 $\mathfrak{A}$
$\mathfrak{A} \models s = t$: for each assigment φ
$\mathfrak{A} \models_{\varphi} s = t$

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	$0 :\to$ nat
	s : nat $\to$ nat
	$\_ + \_$: nat, nat $\to$ 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

$$
\begin{aligned}
\text{e.g.}\quad \mathfrak{B}: \quad & \varphi(x) = a \quad \varphi(y) = b \quad a, b \in \mathbb{Z} \\
& \varphi(x + 0) = a \hat{+} \hat{0} = a \cdot 1 = a = \varphi(x) \\
& \varphi(x + s(y)) = a \hat{+} \hat{s}(b) = a \cdot (b \cdot 5) \\
& \qquad = (a \cdot b) \cdot 5 = \hat{s}(a \hat{+} b) \\
& \qquad = \varphi(s(x + y))
\end{aligned}
$$

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

$\mathfrak{A} \qquad \mathbb{N}, \mathbb{N}^*$
$\hat{0} = 0 \quad \hat{+} = + \quad \hat{s} = +1$
$\widehat{\text{nil}} = e$ (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, $\mathrm{Term}(F, V)$). $\sigma :: \sigma_s : V_s \to \mathrm{Term}_s(F, V)$,
$\sigma_s(x) \in \mathrm{Term}_s(F, V)$, $x \in V_s$
$\sigma(x) = x$ *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 $= (\mathrm{sig}, E)$
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 $\mathrm{ALG}_{TG}(\mathrm{spec})$.

a) *Semantical equality:* $E \models s = t$
b) *Operational equality:* $t_1 \underset{E}{\vdash\!\dashv} t_2$ *iff*
There is $p \in O(t_1), s = t \in E$*, 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 $\dfrac{t' = t''}{s[t']_p = s[t'']_p} \qquad p \in 0(s)$

(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 Term(F, V) *are stable w.r.t. to substitutions*

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

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

$a \underset{E,\mathfrak{A},s}{\sim} a'$ $\quad (a, a' \in A_s)$ 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 $=$ (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.

example

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

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

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

Birkhoff's Theorem

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

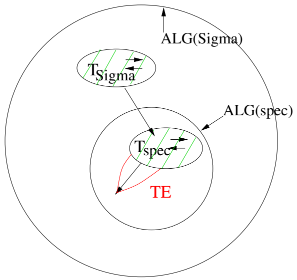
$$E \models s = t \quad \textit{iff} \quad E \vdash s = t \quad (\text{i. e. } s =_E t)$$

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 $T_{\text{sig}}/=_E$.
Write: T_{spec} or $I(E)$



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

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	...

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

Initial algebra

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

Questions:

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

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) <u>spec</u> BOOL
<u>sorts</u> bool
<u>ops</u> $true, false :\longrightarrow$ bool
not : bool $\longrightarrow$ bool
and, or, impl, eqv : bool, bool $\longrightarrow$ bool
if _ then _ else _ : bool, bool, bool $\longrightarrow$ bool

Example (Cont.)

<u>eqns</u> 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''

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

$\rightsquigarrow$ Defined- and constructor-functions.

Example (Cont.)

b) <u>spec</u> SET-OF-CHARACTERS
<u>sorts</u> char, set
<u>ops</u> $a, b, c, \cdots :\longrightarrow$ char
$\emptyset :\longrightarrow$ set
insert : char, set $\longrightarrow$ set
<u>eqns</u> insert(x, insert(x, s)) = insert(x, s)
insert(x, insert(y, s)) = insert(y, insert(x, s))

$(T_{soc})_{char} = \{a, b, c, \ldots\}$
$(T_{soc})_{set} = \{[\emptyset], [\mathrm{insert}(a, \emptyset)], \ldots$
“ “
$\{\emptyset\}\{\mathrm{insert}(a, \mathrm{insert}(a, \ldots, \mathrm{insert}(a, \emptyset)\}$

Example (Cont.)

c)

$$
\begin{array}{ll}
\underline{\text{spec}} & \text{NAT} \\
\underline{\text{sorts}} & \text{nat} \\
\underline{\text{ops}} & 0 :\longrightarrow \text{nat} \\
 & \text{suc} : \text{nat} \longrightarrow \text{nat} \\
 & \_ + \_, \_ * \_ : \text{nat}, \text{nat} \longrightarrow \text{nat} \\
\underline{\text{eqns}} & x + 0 = x \\
 & x + \text{suc}\, y = \text{suc}(x + y) \\
 & x * 0 = 0 \\
 & x * \text{suc}(y) = (x * y) + x
\end{array}
$$

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

Example (Cont.)

d) Binary tree

$$
\begin{array}{ll}
\underline{\text{spec}} & \text{BIN-TREE} \\
\underline{\text{sorts}} & \text{nat}, \text{tree} \\
\underline{\text{ops}} & 0 :\longrightarrow \text{nat} \\
 & \text{suc} : \text{nat} \longrightarrow \text{nat} \\
 & \text{max} : \text{nat}, \text{nat} \longrightarrow \text{nat} \\
 & \text{leaf} :\longrightarrow \text{tree} \\
 & \text{left} : \text{tree} \longrightarrow \text{tree} \\
 & \text{right} : \text{tree} \longrightarrow \text{tree} \\
 & \text{both} : \text{tree}, \text{tree} \longrightarrow \text{tree} \\
 & \text{height} : \text{tree} \longrightarrow \text{nat} \\
 & \text{dleft} : \text{tree} \longrightarrow \text{tree} \\
 & \text{dright} : \text{tree} \longrightarrow \text{tree}
\end{array}
$$

example

Continuation of d) binary tree.

$$
\begin{array}{ll}
\underline{\text{eqns}} & \text{max}(0, n) = n \\
 & \text{max}(n, 0) = n \\
 & \text{max}(\text{suc}(m), \text{suc}(n)) = \text{suc}(\text{max}(m, n)) \\
 & \text{height}(\text{leaf}) = 0 \\
 & \text{height}(\text{both}(t, t')) = \text{suc}(\text{max}(\text{height}(t), \text{height}(t'))) \\
 & \text{height}(\text{left}(t)) = \text{suc}(\text{height}(t)) \\
 & \text{height}(\text{right}(t)) = \text{suc}(\text{height}(t))
\end{array}
$$

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.t. sig). The forget image induces consequently a mapping (functor) between classes of algebras in the following way:

$$\begin{tikzcd}
\text{sig} \arrow[d] \arrow[r, phantom, "\subseteq", "\text{incl.}"'] & \text{sig}' \arrow[d] \\
\text{ALG(sig)} \arrow[r, phantom, "\Cup" description, bend right=0, shift right=0, draw=none] & \text{ALG(sig}') \arrow[l, "|_{\text{sig}}"] \\
\mathfrak{A}|_{\text{sig}} \arrow[u, phantom, "\Cup" rotate=180, draw=none] & \mathfrak{A} \arrow[l, "\text{forgetfulfunctor}"] \arrow[u, phantom, "\Cup", draw=none]
\end{tikzcd}$$

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 $x_1, \dots, x_n$ are the variables in s and t, types $s_1, \dots, s_n$.
$s : (T_{\text{spec}})_{s_1} \times \cdots \times (T_{\text{spec}})_{s_n} \to (T_{\text{spec}})_s$
$s = t \in ITh(E)$ iff s and t induce the same functions $\leadsto$ prove this by induction on the construction of the ground terms.

NAT $0, \text{suc}, +$ $\quad x + y = y + x \quad \in ITH$
$\quad 0 + x = x$

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)	$P_{\text{spec}}(\text{sig}, E, Prop)$
Equations only often do not suffice	Properties that should hold! $\rightsquigarrow$ Verification tasks

Structuring mechanisms

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

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

Here:
Combination, Enrichment, Extension, Modularisation, Parametrisation
$\rightsquigarrow$ **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.*

$\text{comb} = spec_1 + (sig_2, E_2)$ *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: $\quad T_{\text{comb}} := T_{spec}$

The semantics of comb

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

Typical cases:

$S_2 = \emptyset$, 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	$0 :\longrightarrow \text{int}$	
	$\text{suc} : \text{int} \longrightarrow \text{int}$	
	$\cap$	$\cap$
spec	INT2	
use	INT1	$T_{\text{INT2}} \cong (\mathbb{Z}, 0, \text{suc}_{\mathbb{Z}}, \text{pred}_{\mathbb{Z}})$
ops	$\text{pred} : \text{int} \longrightarrow \text{int}$	
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?

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

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

$$\shortparallel$$

$$(\mathbb{Z}, 0, \text{suc}_{\mathbb{Z}}) \qquad \ncong \qquad (\mathbb{N}, 0, \text{suc}_{\mathbb{N}})$$

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

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

$$\left.\begin{array}{l} (T_{\mathrm{comb}})|_{\mathrm{spec}_1} \text{ is } \mathrm{spec}_1 \text{ Algebra} \\ T_{\mathrm{spec}_1} \text{ is initial } \mathrm{spec}_1 \text{ algebra} \end{array}\right\} \rightsquigarrow$$

$\exists!$ 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* comb $= spec_1 + (sig, E)$ *is an extension iff*

$$(T_{\mathrm{comb}})|_{spec_1} \cong T_{spec_1}$$

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

- *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*
$sig_F \subseteq sig_B \qquad E_F \subseteq E_B$.

Notation: Body[Formal]

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

Note: In general it is not 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	$(T_{spec})_{\text{elem}} = \varnothing$
sorts	elem	
ops	next : elem $\longrightarrow$ elem	

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

Example (Cont.)

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

Parameter passing: ELEM $\longrightarrow$ NAT

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

Assignment: formal parameter $\longrightarrow$ current parameter

$$S_F \longrightarrow S_A$$
$$Op \longrightarrow Op_A$$

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 $\longrightarrow$ nat h :: next $\longrightarrow$ suc
Also $\sigma : sig_{\text{BOOL}} \longrightarrow sig_{\text{NAT}}$ with

g ::	*bool* $\longrightarrow$ nat		
h ::	*true* $\longrightarrow$ 0	not $\longrightarrow$ suc	*and* $\longrightarrow$ plus
	false $\longrightarrow$ 0		*or* $\longrightarrow$ 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) $\longrightarrow$ 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 $op : s_1 \dots s_n \longrightarrow s_0 \in$ 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

$$\begin{tikzcd}
\text{Formal} \arrow[r, hook] \arrow[d, "\sigma"] & \text{Body} \arrow[d, dashed, "\sigma'"] \\
\text{Actual} \arrow[r, hook] & \text{Value}
\end{tikzcd}
\qquad
\sigma'(x) = \begin{cases} \sigma(x) & x \in \text{Formal} \\ x' & x \notin \text{Formal} \end{cases}$$

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* $0, \text{suc}, \text{plus}, \text{times}$*)*
$sig' = sig(\text{BOOL})$ $\quad sig = sig(\text{NAT})$
$\sigma : sig' \to sig$ the one considered previously.

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

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

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

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

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

Forgetful functors

$$
\begin{tikzcd}
\mathfrak{A}|_\sigma \arrow[d, "h|_\sigma"] & \mathfrak{A} \arrow[l, "|_\sigma"'] \arrow[d, "h"] \\
\mathfrak{B}|_\sigma & \mathfrak{B} \arrow[l, "|_\sigma"']
\end{tikzcd}
$$

Forgetful functors

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

$$
\begin{tikzcd}
\mathrm{sig}' \arrow[r, "\sigma"] \arrow[d, dash] & \mathrm{sig} \arrow[r, "\sigma'"] \arrow[d, dash] & \mathrm{sig}'' \arrow[d, dash] \\
\mathrm{ALG}(\mathrm{sig}') & \mathrm{ALG}(\mathrm{sig}) \arrow[l, "|_\sigma"'] & \mathrm{ALG}(\mathrm{sig}'') \arrow[l, "|_{\sigma'}"']
\end{tikzcd}
$$

$\Cup \quad \Cup \qquad \Cup \quad \Cup$

$\mathfrak{A}|_\sigma \xrightarrow{h|_\sigma} \mathfrak{B}|_\sigma \qquad \mathfrak{A} \xrightarrow{h} \mathfrak{B}$

Compatible with identity, composition and homomorphisms.

Forgetful functors

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

$$
\begin{tikzcd}
\mathrm{sig}' \arrow[r, "\sigma"] \arrow[rr, bend left, "\sigma' \circ \sigma"] & \mathrm{sig} \arrow[r, "\sigma'"] & \mathrm{sig}'' \\
\mathrm{Alg}(\mathrm{sig}') & \mathrm{Alg}(\mathrm{sig}) \arrow[l, "|_\sigma"'] & \mathrm{Alg}(\mathrm{sig}'') \arrow[l, "|_{\sigma'}"'] \arrow[ll, bend left, "|_{(\sigma' \circ \sigma)}"]
\end{tikzcd}
$$

Parameter Specification *Body*[*Formal*]

$$
\begin{tikzcd}
\mathrm{ALG}(\mathrm{Formal}) & & & \mathrm{ALG}(\mathrm{Body}) \arrow[lll, "|_{\mathrm{incl}}"'] \\
& \mathrm{Formal} \arrow[r, hook, "\mathrm{incl}"'] \arrow[d, "\sigma"] & \mathrm{Body} \arrow[d, "\sigma'"] & \\
& \mathrm{Actual} \arrow[r, hook, "\mathrm{incl'}"'] & \mathrm{Value} & \\
\mathrm{ALG}(\mathrm{Actual}) \arrow[uuu, "|_\sigma"] & & & \mathrm{ALG}(\mathrm{Value}) \arrow[lll, "|_{\mathrm{incl'}}"] \arrow[uuu, "|_{\sigma'}"']
\end{tikzcd}
$$

Semantics of parameter passing (only signature)

Definition 7.17. *Let Body[Formal] be a parameterized specification.* $\sigma : \mathrm{Formal} \longrightarrow \mathrm{Actual}$ *signature morphism.*

Semantics of the the "instantiation" i.e. parameter passing $[\mathrm{Actual}, \sigma]$.

$$\begin{array}{c}\sigma : \mathrm{Formal} \longrightarrow \mathrm{Actual} \\ \downarrow \\ \text{initial semantics of value. i. e.} \\ T_{\mathrm{Body}[\mathrm{Actual},\sigma]}\end{array}$$

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

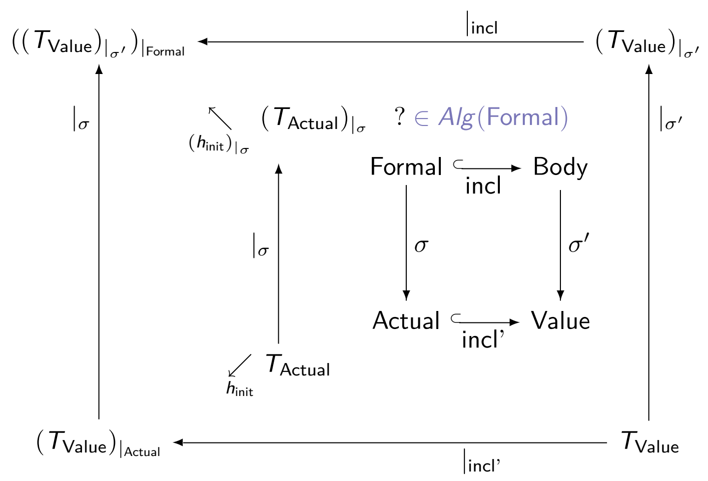
This mapping between initial algebras can be interpreted as correspondence between formal algebras $\longrightarrow$ *body-algebras.*

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

Semantics parameter passing $(T_{\mathrm{Actual}})|_\sigma \mapsto (T_{\mathrm{Body}[\mathrm{Actual},\sigma]})|_{\sigma'}$

$$\begin{tikzcd}
\mathrm{Actual} \arrow[r, hook] \arrow[dddd, "\text{init-Sem.}"'] & \mathrm{Body}[\mathrm{Actual},\sigma] \arrow[d, "\text{init-Sem.}"] \\
& T_{\mathrm{Body}[\mathrm{Actual},\sigma]} \arrow[d, "\text{forget-image}"] \\
& (T_{\mathrm{Body}[\mathrm{Actual},\sigma]})|_{\mathrm{incl}} \arrow[dd, phantom, "\shortparallel"] \\
& \\
h_{\mathrm{init}} : T_{\mathrm{Actual}} \arrow[r] & (T_{\mathrm{Body}[\mathrm{Actual},\sigma]})|_{\mathrm{Actual}}
\end{tikzcd}$$

Mapping between initial algebras



Properties of the signature morphism

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

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

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

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

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

Parameter passing (Actual, σ)

Body[Formal]

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

$$\begin{tikzcd}
\text{Formal} \arrow[r, hook, "\text{incl}"'] \arrow[d, "\sigma"] & \text{Body} \arrow[d, "\sigma'(\text{with renaming})"] \\
\text{Actual} \arrow[r, hook, "\text{incl'}"'] & \text{Value} = \text{Body}[\text{Actual}, \sigma]
\end{tikzcd}$$

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

Parameter passing (Actual, σ)

$$\begin{tikzcd}
((T_{\text{Value}})|_{\sigma'})|_{\text{Formal}} & & & (T_{\text{Value}})|_{\sigma'} \arrow[lll, "|_{\text{incl}}"'] \\
& (T_{\text{Actual}})|_{\sigma} \in \text{Alg}(\text{Formal}) \arrow[ul, "(h_{\text{init}})|_{\sigma}"'] & & \\
& T_{\text{Actual}} \arrow[u, "|_{\sigma}"] \arrow[dl, "h_{\text{init}}"'] & & \\
(T_{\text{Value}})|_{\text{Actual}} \arrow[uuu, "|_{\sigma}"] & & & T_{\text{Value}} \arrow[lll, "|_{\text{incl'}}"] \arrow[uuu, "|_{\sigma'}"']
\end{tikzcd}$$

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

Parameter passing (Actual, σ)

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

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

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

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

sig$'$-homomorphism

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* $\sigma : \text{Formal} \to \text{Actual}$ *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, $\underset{+}{\rightsquigarrow}$ newer languages: - Spectrum, - Troll.

Example

Example 7.20.

$$\text{Formal} :: \left\{ \begin{array}{ll} \textit{spec} & \text{ELEMENT} \\ \text{use} & \text{BOOL} \\ \text{sorts} & \text{elem} \\ \text{ops} & . \le . : \text{elem}, \text{elem} \to \text{bool} \\ \text{eqns} & x \le x = \textit{true} \\ & \text{imp}(x \le y \textit{ and } y \le z, x \le z) = \textit{true} \\ & x \le y \textit{ or } y \le x = \textit{true} \end{array} \right.$$

Example (Cont.)

$$\begin{array}{ll} \text{spec} & \text{LIST[ELEMENT]} \\ \text{use} & \text{ELEMENT} \\ \text{sorts} & \text{list} \\ \text{ops} & \text{nil} :\to \text{list} \\ & . : \text{elem}, \text{list} \to \text{list} \\ & \text{insert} : \text{elem}, \text{list} \to \text{list} \\ & \text{insertsort} : \text{list} \to \text{list} \\ & \text{case} : \text{bool}, \text{list}, \text{list} \to \text{list} \\ & \text{sorted} : \text{list} \to \text{bool} \end{array}$$

Example (Cont.)

$$\begin{array}{ll} \text{eqns} & \text{case}(\text{true}, l_1, l_2) = l_1 \\ & \text{case}(\text{false}, l_1, l_2) = l_2 \\ & \\ & \text{insert}(x, \text{nil}) = x.\text{nil} \\ & \text{insert}(x, y.l) = \text{case}(x \le y, x.y.l, y.\,\text{insert}(x, l)) \\ & \\ & \text{insertsort}(\text{nil}) = \text{nil} \\ & \text{insertsort}(x.l) = \textit{insert}(x, \textit{insertsort}(l)) \\ & \\ & \text{sorted}(\text{nil}) = \text{true} \\ & \text{sorted}(x.\text{nil}) = \text{true} \\ & \text{sorted}(x.y.l) = \text{if } x \le y \text{ then sorted}(y.l) \text{ else false} \end{array}$$

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!.

Abstract Reduction Systems: Fundamental notions and notations

Definition 8.1. $(U, \to)$ $U \neq \emptyset, \to$ *binary relation is called a reduction system.*

- *Notions:*
- $x \in U$ *reducible iff* $\exists y : x \to y$
  irreducible if not reducible.
- $x \xrightarrow{*} y$ *reflexive, transitive closure,* $x \xrightarrow{+} y$ *transitive closure,* $x \xleftrightarrow{*} y$ *reflexive, symmetrical, transitive closure.*
- $x \xrightarrow{i} y$ $i \in \mathbb{N}$ *defined as usual. Notice* $x \xrightarrow{*} y = \bigcup_{i \in \mathbb{N}} x \xrightarrow{i} y$.
- $x \xrightarrow{*} y$, y *irreducible, then* y *is a normal form for* x. *Abb:: NF*
- $\Delta(x) = \{y \mid x \to y\}$, *the set of direct successors of* x.
- $\Delta^+(x)$ *proper successors,* $\Delta^*(x)$ *successors.*

Notions and notations

- $\Lambda(x) = \max\{i \mid \exists y : x \xrightarrow{i} y\}$ derivational complexity. $\Lambda : U \to \mathbb{N}_\infty$
- $\to$ noetherian (terminating, satisfies the chain condition), in case there is no infinite chain $x_1 \to x_2 \to x_3 \to \cdots$.
- $\to$ bounded, in case that $\Lambda : U \to \mathbb{N}$.
- $\to$ cycle free :: $\neg\exists x \in U : x \xrightarrow{+} x$
- $\to$ locally finite $x \begin{matrix} \nearrow \\ \to \\ \searrow \end{matrix} \Bigg\}$, i.e. $\Delta(x)$ finite for every x.

Notions and notations

Simple properties:

- $\rightarrow$ cycle free, then $\xrightarrow{*}$ partial ordering.
- $\rightarrow$ noetherian, then $\rightarrow$ cycle free.
- $\rightarrow$ bounded, so $\rightarrow$ noetherian.
  but not the other way around!
- $\rightarrow \subset \overset{+}{\Rightarrow}$ and $\Rightarrow$ noetherian, then $\rightarrow$ noetherian.

Principle of the Noetherian Induction

Definition 8.2. *$\rightarrow$ binary relation on U, P predicate on U. P is $\rightarrow$-complete, when*

$$\forall x[(\forall y \in \Delta^+(x) : P(y)) \supset P(x)]$$

Fact:
PNI: If $\rightarrow$ is noetherian and P is $\rightarrow$-complete, then $P(x)$ holds for all $x \in U$.

Applications

Lemma 8.3. *$\rightarrow$ noetherian, then each $x \in U$ has at least one normal form.*

More applications to come.... See e.g. König's lemma.

Definition 8.4. *Main properties for $(U, \rightarrow)$*

- *$\rightarrow$ confluent iff $\overset{*}{\longleftarrow} \circ \overset{*}{\longrightarrow} \subseteq \overset{*}{\longrightarrow} \circ \overset{*}{\longleftarrow}$*
- *$\rightarrow$ Church-Rosser iff $\overset{*}{\longleftrightarrow} \subseteq \overset{*}{\longrightarrow} \circ \overset{*}{\longleftarrow}$*
- *$\rightarrow$ locally-confluent iff $\longleftarrow \circ \longrightarrow \subseteq \overset{*}{\longrightarrow} \circ \overset{*}{\longleftarrow}$*
- *$\rightarrow$ strong-confluent iff $\longleftarrow \circ \longrightarrow \subseteq \overset{*}{\longrightarrow} \circ \overset{\leq 1}{\longleftarrow}$*
- *Abbreviation: joinable $\downarrow$:*

$$\downarrow = \overset{*}{\longrightarrow} \circ \overset{*}{\longleftarrow}$$

Important relations

Lemma 8.5. *$\rightarrow$ confluent iff $\rightarrow$ Church-Rosser.*

Theorem 8.6. *(Newmann Lemma) Let $\rightarrow$ be noetherian, then*

$$\rightarrow \text{ confluent iff } \rightarrow \text{ locally confluent.}$$

Consequence 8.7. a) *Let $\rightarrow$ confluent and $x \overset{*}{\longleftrightarrow} y$.*

i) *If y is irreducible, then $x \overset{*}{\longrightarrow} y$. In particular, when x, y irreducible, then $x = y$.*
ii) *$x \overset{*}{\longleftrightarrow} y$ iff $\Delta^*(x) \cap \Delta^*(y) \neq \emptyset$.*
iii) *If x has a NF, then it is unique.*
iv) *If $\rightarrow$ is noetherian, then each $x \in U$ has exactly one NF: notation $x\downarrow$*

b) *If in $(U, \rightarrow)$ each $x \in U$ has exactly one NF, then $\rightarrow$ is confluent (in general not noetherian).*

Convergent Reduction Systems

Definition 8.8. $(U, \to)$ *convergent* *iff* $\to$ *noetherian and confluent.*

Important since: $x \overset{*}{\longleftrightarrow} y$ *iff* $x \downarrow = y \downarrow$

Hence if $\to$ *effective* $\rightsquigarrow$ *decision procedure for Word Problem (WP):*

For programming: $x \xrightarrow{*} x \downarrow$, $f(t_1, \ldots, t_n) \xrightarrow{*}$ *„value"*

As usual these properties are in general undecidable properties.

Task: Find sufficient computable conditions which guarantee these properties.

Termination and Confluence

Sufficient conditions/techniques

Lemma 8.9. $(U, \to)$, $(M, \succ)$, $\succ$ *well founded (WF) partial ordering.*
If there is $\varphi : U \to M$ *with* $\varphi(x) \succ \varphi(y)$ *for* $x \to y$*, then* $\to$ *is noetherian.*

Example 8.10. *Often* $(\mathbb{N}, >), (\Sigma^*, >)$ *can be used.*
For $w \in \Sigma^*$ *let* $|w|$ *length,* $|w|_a$ *a-length* $a \in \Sigma$.

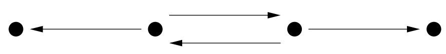
WF-partial orderings on Σ^*

- $x > y$ *iff* $|x| > |y|$
- $x > y$ *iff* $|x|_a > |y|_a$
- $x > y$ *iff* $|x| > |y|$, $|x| = |y| \wedge x \succ_{lex} y$

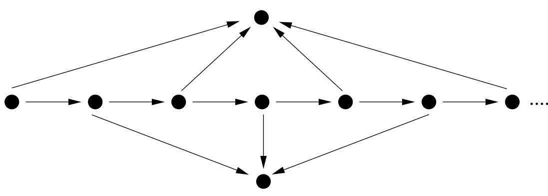
Notice that pure lex-ordering on Σ^* is not noetherian.

Sufficient conditions for confluence

Termination: Confluence *iff* local confluence
Without termination this doesn't hold!



or



Confluence without termination

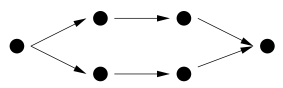
Theorem 8.11. $\to$ *is confluent iff for every* $u \in U$ *holds:*

from $u \to x$ *and* $u \xrightarrow{*} y$ *it follows* $x \downarrow y$.

▷ *one-sided localization of confluence* ◁

Theorem 8.12. *If* $\to$ *is strong confluent, then* $\to$ *is confluent.*

Not a necessary condition:



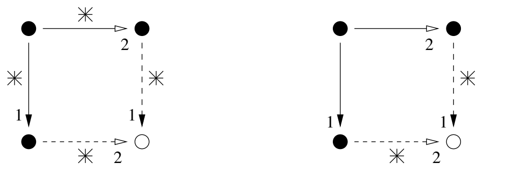
Combination of Relations

Definition 8.13. *Two relations $\rightarrow_1$, $\rightarrow_2$ on U commute, iff*

$$_1\overset{*}{\leftarrow} \circ \overset{*}{\rightarrow}_2 \quad \subseteq \quad \overset{*}{\rightarrow}_2 \circ {}_1\overset{*}{\leftarrow}$$

.

They commute locally iff $_1\leftarrow \circ \rightarrow_2 \quad \subseteq \quad \overset{*}{\rightarrow}_2 \circ {}_1\overset{*}{\leftarrow}$.



commutating   locally commutating

Combination of Relations

Lemma 8.14. *Let* $\rightarrow \; = \; \rightarrow_1 \cup \rightarrow_2$

(1) If $\rightarrow_1$ and $\rightarrow_2$ commute locally and $\rightarrow$ is noetherian, then $\rightarrow_1$ and $\rightarrow_2$ commute.
(2) If $\rightarrow_1$ and $\rightarrow_2$ are confluent and commute, then $\rightarrow$ is also confluent.

Problem: Non-Orientability:

(a) $x + 0 = x, \;\; x + s(y) = s(x + y)$
(b) $x + y = y + x, \;\; (x + y) + z = x + (y + z)$

▷ *Problem: permutative rules like (b)* ◁

Non-Orientability

Definition 8.15. *Let $(U, \rightarrow, \vdash\dashv)$ with $\rightarrow$ a binary relation, $\vdash\dashv$ a symmetrical relation.*

Let $\quad \mid\!\vdash\dashv\!\mid \; = \; \leftrightarrow \cup \vdash\dashv, \qquad \sim \; = \; \overset{*}{\vdash\dashv}, \qquad \approx \; = \; \overset{*}{\mid\!\vdash\dashv\!\mid},$
$\qquad \rightarrow_\sim \; = \; \sim \circ \rightarrow \circ \sim, \qquad \downarrow_\sim \; = \; \overset{*}{\rightarrow} \circ \sim \circ \overset{*}{\leftarrow}.$

If $x \downarrow_\sim y$ holds, then $x, y \in U$ are called joinable modulo $\sim$.

$\rightarrow$ is called Church-Rosser modulo $\sim$ iff $\;\approx \; \subseteq \; \downarrow_\sim$

$\rightarrow$ is called locally confluent modulo $\sim$ iff $\leftarrow \circ \rightarrow \; \subseteq \; \downarrow_\sim$

$\rightarrow$ is called locally coherent modulo $\sim$ iff $\leftarrow \circ \vdash\dashv \; \subseteq \; \downarrow_\sim$

Non-Orientability - Reduction Modulo $\vdash\dashv$

Theorem 8.16. *Let $\rightarrow_\sim$ be terminating. Then $\rightarrow$ is Church-Rosser modulo $\sim$ iff $\sim$ is local confluent modulo $\sim$ and local coherent modulo $\sim$.*

$$\bullet \longleftarrow \bullet \xleftarrow{\vdash\dashv} \bullet \xrightarrow{\vdash\dashv} \bullet \longrightarrow \bullet$$

Most frequent application: Modulo AC (Associativity + Commutativity)

Representation of equivalence relations by convergent reduction relations

Situation: Given: $(U, \vdash\!\dashv)$ and a noetherian PO $>$ on U , find: $(U, \to)$ with
(i) $\to \;\subseteq\; >$, $\to$ convergent on U and
(ii) $\overset{*}{\leftrightarrow} \;=\; \sim$ with $\sim \;=\; \overset{*}{\vdash\!\dashv}$

Idea: Approximation of $\to$ by stepwise transformations

$$(\vdash\!\dashv, \emptyset) = (\vdash\!\dashv_0, \to_0) \vdash (\vdash\!\dashv_1, \to_1) \vdash (\vdash\!\dashv_2, \to_2) \vdash \dots$$

Invariant in i-th. step:

(i) $\sim \;=\; (\vdash\!\dashv_i \cup \leftrightarrow_i)^*$ and
(ii) $\to_i \;\subseteq\; >$

Goal: $\vdash\!\dashv_i = \emptyset$ for an i and $\to_i$ convergent.

Representation of equivalence relations by convergent reduction relations

Allowed operations in i-th. step:

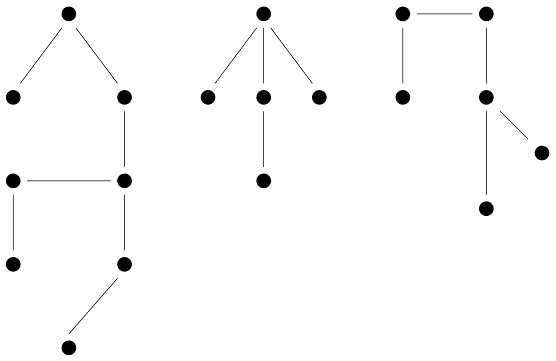
(1) Orient:: $u \to_{i+1} v$, if $u > v$ and $u \vdash\!\dashv_i v$
(2) New equivalences:: $u \vdash\!\dashv_{i+1} v$, if $u \;{}_i\!\leftarrow w \to_i v$
(3) Simplify:: $u \vdash\!\dashv_i v$ to $u \vdash\!\dashv_{i+1} w$, if $v \to_i w$

Goal: Limit system

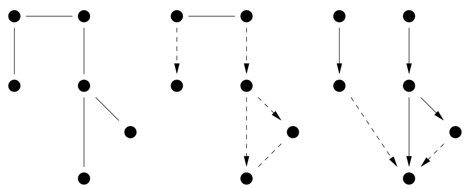
$$\to \;=\; \to_\infty \;=\; \bigcup\{\to_i \mid i \in \mathbb{N}\} \text{ with } \vdash\!\dashv_\infty \;=\; \emptyset$$

Hence:
- $\longrightarrow_\infty \subseteq >$, i.e. noetherian
- $\overset{*}{\longleftrightarrow} = \sim$
- $\longrightarrow_\infty$ convergent !

Grafical representation of an equivalence relation



Transformation of an equivalence relation



(a) (b) (c)

Inference system for the transformation of an equivalence relation

Definition 8.17. *Let $>$ be a noetherian PO on U. The inference system $\mathcal{P}$ on objects $(\vdash\!\dashv, \to)$ contains the following rules:*

(1) *Orient*
$$\frac{(\vdash\!\dashv \cup \{u \vdash\!\dashv v\}, \to)}{(\vdash\!\dashv, \to \cup \{u \to v\})} \text{ if } u > v$$

(2) *Introduce new consequence*
$$\frac{(\vdash\!\dashv, \to)}{(\vdash\!\dashv \cup \{u \vdash\!\dashv v\}, \to)} \text{ if } u \leftarrow \circ \to v$$

(3) *Simplify*
$$\frac{(\vdash\!\dashv \cup \{u \vdash\!\dashv v\}, \to)}{(\vdash\!\dashv \cup \{u \vdash\!\dashv w\}, \to)} \text{ if } v \to w$$

Inference system (Cont.)

(4) Eliminate identities
$$\frac{(\vdash\!\dashv \cup \{u \vdash\!\dashv u\}, \to)}{(\vdash\!\dashv, \to)}$$

$(\vdash\!\dashv, \to) \vdash_{\mathcal{P}} (\vdash\!\dashv', \to')$ if
$(\vdash\!\dashv, \to)$ can be transformed in one step with a rule $\mathcal{P}$ into $(\vdash\!\dashv', \to')$.

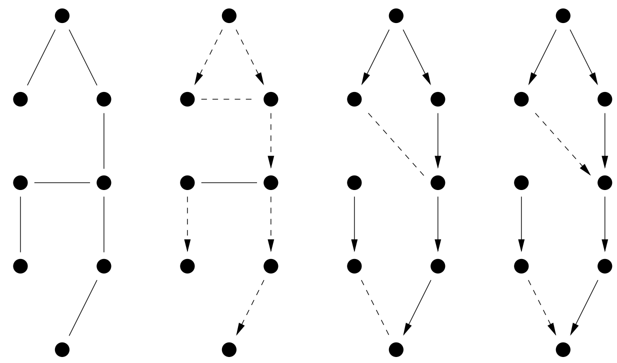
$\vdash^*_{\mathcal{P}}$ transformation relation in finite number of steps with $\mathcal{P}$.

A sequence $((\vdash\!\dashv_i, \to_i))_{i \in \mathbb{N}}$ is called $\mathcal{P}$-derivation, if

$$(\vdash\!\dashv_i, \to_i) \vdash_{\mathcal{P}} (\vdash\!\dashv_{i+1}, \to_{i+1}) \text{ for every } i \in \mathbb{N}$$

.

Transformation with the inference system



(a) (b) (c) (d)

Properties of the inference system

Lemma 8.18. *Let* $(\vdash\!\dashv, \to) \vdash_{\mathcal{P}} (\vdash\!\dashv', \to')$

(a) *If* $\to\ \subseteq\ >$, *then* $\to'\ \subseteq\ >$

(b) $(\vdash\!\dashv \cup \leftrightarrow)^* = (\vdash\!\dashv' \cup \leftrightarrow')^*$

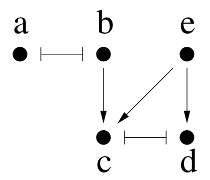
Problem:
When does $\mathcal{P}$ deliver a convergent reduction relation $\to$?
How to measure progress of the transformation?

Idea: Define an ordering $>_{\mathcal{P}}$ on equivalence-proofs, and prove that the inference system $\mathcal{P}$ decreases proofs with respect to $>_{\mathcal{P}}$!

In the proof ordering $\xrightarrow{*} \circ \xleftarrow{*}$ proofs should be minimal.

Equivalence Proofs

Definition 8.19. *Let $(\vdash\!\dashv, \rightarrow)$ be given and $>$ a noetherian PO on U. Furthermore let $(\vdash\!\dashv \cup \leftrightarrow)^* = \sim$.*
A proof for $u \sim v$ is a sequence $u_0 *_1 u_1 *_2 \cdots *_n u_n$ with $*_i \in \{\vdash\!\dashv, \leftarrow, \rightarrow\}$, $u_i \in U$, $u_0 = u$, $u_n = v$ and for every i $u_i *_{i+1} u_{i+1}$ holds.
$P(u) = u$ is proof for $u \sim u$.
A proof of the form $u \xrightarrow{*} z \xleftarrow{*} v$ is called V-proof.



Proofs for $a \sim e$:

$P_1(a,e) = a \vdash\!\dashv b \rightarrow c \vdash\!\dashv d \leftarrow e \qquad P_2(a,e) = a \vdash\!\dashv b \rightarrow c \leftarrow e$

Proof orderings

Two proofs in $(\vdash\!\dashv, \rightarrow)$ are called equivalent, if they prove the equivalence of the same pair (u,v). Hence e.g. $P_1(a,e)$ and $P_2(a,e)$ are equivalent.

Notice: If $P_1(u,v), P_2(v,w)$ and $P_3(w,z)$ are proofs, then $P(u,z) = P_1(u,v)P_2(v,w)P_3(w,z)$ is also a proof.

Definition 8.20. *A proof ordering $>_B$ is a PO on the set of proofs that is monotonic, i.e.. $P >_B Q$ for each subproof, and if $P >_B Q$ then $P_1 P P_2 >_B P_1 Q P_2$.*

Lemma 8.21. *Let $>$ be noetherian PO on U and $(\vdash\!\dashv, \rightarrow)$, then there exist noetherian proof orderings on the set of equivalence proofs.*

Proof: Using multiset orderings.

Multisets and the multiset ordering

Instruments: Multiset ordering

Objects: U, $Mult(U)$ Multisets over U

$A \in Mult(U)$ iff $A : U \rightarrow \mathbb{N}$ with $\{u \mid A(u) > 0\}$ finite.

Operations: $\cup, \cap, -$

$$(A \cup B)(u) := A(u) + B(u)$$
$$(A \cap B)(u) := min\{A(u), B(u)\}$$
$$(A - B)(u) := max\{0, A(u) - B(u)\}$$

Explicit notation:
$U = \{a,b,c\}$ e.g. $A = \{\{a,a,a,b,c,c\}\}, B = \{\{c,c,c\}\}$

Multiset ordering

Definition 8.22. *Extension of $(U, >)$ to $(Mult(U), \gg)$*

$$A \gg B \text{ iff there are } X, Y \in Mult(U) \text{ with } \emptyset \neq X \subseteq A \text{ and } B = (A - X) \cup Y, \text{ so that } \forall y \in Y \ \exists x \in X \ x > y$$

Properties:

(1) $>$ PO $\leadsto$ $\gg$ PO
(2) $\{m_1\} \gg \{m_2\}$ iff $m_1 > m_2$
(3) $>$ total $\leadsto$ $\gg$ total
(4) $A \gg B \leadsto A \cup C \gg B \cup C$
(5) $B \subset A \leadsto A \gg B$
(6) $>$ noetherian iff $\gg$ noetherian

Example: $a < b < c$ then $B \gg A$

Construction of the proof ordering

Let $(\vdash\dashv, \to)$ be given and $>$ a noetherian PO on U with $\to \subset >$
Assign to each „atomic“ proof a complexity

$$c(u * v) = \begin{cases} \{u\} & \text{if } u \to v \\ \{v\} & \text{if } u \leftarrow v \\ \{\{u, v\}\} & \text{if } u \vdash\dashv v \end{cases}$$

Extend this complexity to „composed“ proofs through

$c(P(u)) = \emptyset$
$c(P(u, v)) = \{\{c(u_i *_{i+1} u_{i+1}) \mid i = 0, \dots n-1\}\}$

Notice: $c(P(u, v)) \in Mult(Mult(U))$

Define ordering on proofs through

$$P >_{\mathcal{P}} Q \ \text{ iff } \ c(P) \ggg c(Q)$$

Construction of the proof ordering

Fact : $>_{\mathcal{P}}$ is notherian proof ordering!

Which proof steps are large and which small?

Consider:

(a) $P_1 = x \leftarrow u \to y$, $P_2 = x \vdash\dashv y$

$c(P_1) = \{\{\{u\}, \{u\}\}\} \ggg \{\{x, y\}\} = c(P_2)$ since $u > x$ and $u > y$
$\rightsquigarrow P_1 >_{\mathcal{P}} P_2$

analogously for

(b) $P_1 = x \vdash\dashv y$, $P_2 = x \to y$

(c) $P_1 = u \vdash\dashv v$, $P_2 = u \vdash\dashv w \leftarrow v$

(d) $P_1 = u \vdash\dashv v$, $P_2 = u \to w \leftarrow v$

Fair Deductions in $\mathcal{P}$

Definition 8.23 (Fair deduction). *Let $(\vdash\dashv_i, \to_i)_{i \in \mathbb{N}}$ be a $\mathcal{P}$-deduction. Let*

$$\vdash\dashv^\infty = \bigcup_{i \geq 0} \bigcap_{j \geq i} \vdash\dashv_i \ \text{ and } \to^\infty = \bigcup_{i \geq 0} \to_i .$$

The $\mathcal{P}$-Deduction is called fair, in case

(1) $\vdash\dashv^\infty = \emptyset$ and
(2) If $x \ {}^\infty\!\!\leftarrow u \to^\infty y$, then there exists $k \in \mathbb{N}$ with $x \vdash\dashv_k y$.

Lemma 8.24. *Let $(\vdash\dashv_i, \to_i)_{i \in \mathbb{N}}$ be a fair $\mathcal{P}$-deduction*

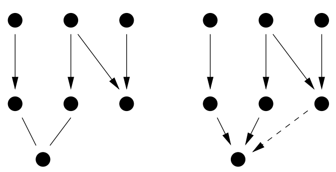
(a) For each proof P in $(\vdash\dashv_i, \to_i)$ there is an equivalent proof P' in $(\vdash\dashv_{i+1}, \to_{i+1})$ with $P \geq_{\mathcal{P}} P'$.

(b) Let $i \in \mathbb{N}$ and P proof in $(\vdash\dashv_i, \to_i)$ which is not a V-proof. Then there exists a $j > i$ and an equivalent proof P' in $(\vdash\dashv_j, \to_j)$ with $P >_{\mathcal{P}} P'$.

Main result

Theorem 8.25. *Let $(\vdash\dashv_i, \to_i)_{i \in \mathbb{N}}$ a fair $\mathcal{P}$-Deduction and $\to \ = \ \to^\infty$. Then*

(a) If $u \sim v$, then there exists an $i \in \mathbb{N}$ with $u \xrightarrow{*}_i \circ {}_i\!\xleftarrow{*} v$
(b) $\to$ is convergent and $\xleftrightarrow{*} \ = \ \sim$



Term Rewriting Systems

Goal: Operationalization of specifications and implementation of functional programming languages

Given $spec = (sig, E)$ when is T_{spec} a computable algebra?

$(T_{spec})_s = \{[t]_{=_E} : t \in Term(sig)_s\}$

T_{spec} is a computable Algebra if there is a computable function

$rep : Term(sig) \to Term(sig)$, with $rep(t) \in [t]_{=_E}$ the "unique representative" in its equivalence class.

Paradigm: Choose as representative the minimal object in the equivalence class with respect to an ordering.

$f(x_1, ..., x_n) : ((T_{spec})_{s_1} \times ...(T_{spec})_{s_n}) \to (T_{spec})_s$
$f([r_1], ..., [r_n]) := [rep(f(rep(r_1), ..., (rep(r_n))]$

Term Rewriting Systems

Definition 9.1. *Rules, rule sets, reduction relation*

- *Sets of variables in terms: For $t \in Term_s(F, V)$ let $V(t)$ be the set of the variables in t (Recursive definition! always finite)*
  Notice: $V(t) = \emptyset$ iff t is ground term.
- *A rule is a pair*
  $(l, r), l, r \in Term_s(F, V)$ $(s \in S)$ with $Var(r) \subseteq Var(l)$
  Write: $l \to r$
- *A rule system R is a set of rules.*
  R defines a reduction relation $\to_R$ over $Term(F, V)$ by:
  $t_1 \to_R t_2$ iff $\exists\ l \to r \in R, p \in O(t_1), \sigma$ substitution : $t_1|_p = \sigma(l) \wedge t_2 = t_1[\sigma(r)]_p$
- *Let $(Term(F, V), \to_R)$ be the reduction system defined by R (term rewriting system).*
- *A rule system R defines a congruence $=_R$ on $Term(F, V)$ just by considering the rules as equations.*

Term Rewriting Systems

Goal: Transform E in R, so that $=_E = \overset{*}{\longleftrightarrow}_R$ holds and $\to_R$ has "sufficiently"good termination and confluence properties.
For instance convergent or confluent. Often it is enough when these properties hold "only" on the set of ground terms.

Notice:

- The condition $V(r) \subseteq V(l)$ in the rule $l \to r$ is necessary for the termination.
  If neither $V(r) \subseteq V(l)$ nor $V(l) \subseteq V(r)$ in an equation $l = r$ of a specification, we have used superfluous variables in some function's definition.
- $\to_R$ is compatible with substitutions and term replacement. i.e.
  From $s \to_R t$ also $\sigma(s) \to_R \sigma(t)$ and $u[s]_p \to_R u[t]_p$
- In particular: $=_R = \overset{*}{\longleftrightarrow}_R$

Matching substitution

Definition 9.2. *Let $l, t \in Term_s(F, V)$. A substitution σ is called a match (matching substitution) of l on t, if $\sigma(l) = t$.*

Consequence 9.3. ***Properties**:*

- *$\forall\ \sigma$ substitution $O(l) \subseteq O(\sigma(l))$.*
- *$\exists \sigma : \sigma(l) = t$ iff for σ defined through*
  $\forall u\ O(l) : l|_u = x \in V \rightsquigarrow u \in O(t) \wedge \sigma(x) = t|_u$
  σ is a substitution $\wedge\ \sigma(l) = t$.
- *If there is such a substitution, then it is unique on $V(l)$. The existence and if possible calculation are effective.*
- *It is decidable whether t is reducible with rule $l \to r$.*
- *If R is finite, then $\Delta(s) = \{t : s \to_R t\}$ is finite and computable.*

Examples

Example 9.4. *Integer numbers*

$$\begin{array}{rl} sig: & 0 :\to int \\ & s,p : int \to int \\ if0 : & int, int, int \to int \\ F : & int, int \to int \end{array} \qquad \begin{array}{rl} eqns: & 1 :: p(0) = 0 \\ & 2 :: p(s(x)) = x \\ & 3 :: if0(0,x,y) = x \\ & 4 :: if0(s(z),x,y) = y \\ & 5 :: F(x,y) = if0(x,0,F(p(x),F(x,y))) \end{array}$$

Interpretation: $\langle \mathbb{N}, \ldots, \rangle$ *spec- Algebra with functions*
$O_{\mathbb{N}} = 0, s_{\mathbb{N}} = \lambda n.\ n+1,$
$p_{\mathbb{N}} = \lambda n.\ if \quad n = 0 \quad then \quad 0 \quad else \quad n-1 \quad fi$
$if0_{\mathbb{N}} = \lambda i,j,k.\ if \quad i = 0\ then \quad j\ else \quad k \quad fi$
$F_{\mathbb{N}} = \lambda m,n.\ 0$

Orient the equations from left to right $\rightsquigarrow$ *rules R (variable condition is fulfilled).*
Is R terminating? Not with a syntactical ordering, since the left side is contained in the right side.

Example (Cont.)

Reduction sequence:

$$F(s(0),0) \to_5 \underbrace{if0(s(0),0,\underbrace{F(\underbrace{p(s(0))}_{2},\underbrace{F(s(0),0)}_{5})})}_{4}$$

$$\to_4 \underbrace{F(\underbrace{p(s(0))},\underbrace{F(s(0),0)})}_{5}$$

$$\to_2 \underbrace{F(0,\underbrace{F(s(0),0)})}_{5}$$

$$\to_5 \underbrace{if0(0,0,\underbrace{F(\underbrace{p(0)},\underbrace{F(0,\underbrace{F(s(0),0)})})})}_{3} \to_3 0$$

Equivalence

Definition 9.5. *Let* $spec = (sig, E), spec' = (sig, E')$ *be specifications. They are equivalent in case* $=_E \quad = \quad =_{E'}$*, i.e..* $T_{spec} = T_{spec'}$*.*
A rule system R over sig is equivalent to E, in case $=_E \quad = \quad \xleftrightarrow{*}_R$*.*

Notice: If R is finite, convergent, equivalent to E, then $=_E$ is decidable

$$s =_E t \quad iff \quad s \downarrow = t \downarrow \text{ i.e.. identical NF}$$

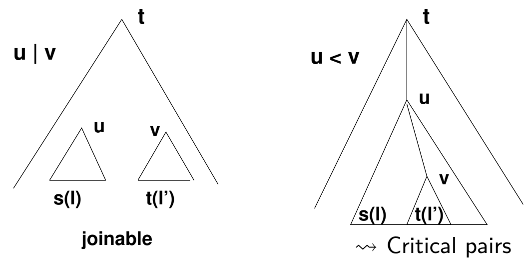
For functional programs and computations in T_{spec} ground convergence is suficient, i.e.. convergence on ground terms.
Problems: Decide whether

- R noetherian (ground noetherian)
- R confluent (ground confluent)
- How can we transform E in an equivalent R with these properties?

Decidability questions

For finite ground term-rewriting-systems the problems are decidable.

For terminating systems deciding local confluence is sufficient, i.e.. out of $t_1 \leftarrow t \rightarrow t_2$ prove $t_1 \downarrow t_2 \rightsquigarrow$ confluent.



Critical pairs

Consider the group axioms:

$$\underbrace{(x' \cdot y') \cdot z}_{l_1} \to x' \cdot (y' \cdot z) \quad \text{and} \quad \underbrace{x \cdot x^{-1}}_{l_2} \to 1.$$

"Overlappings" (Superpositions)

$$\begin{array}{ccc} & (x \cdot x^{-1}) \cdot z & \\ \swarrow_{l_2} & & \searrow_{l_1} \\ 1 \cdot z & & x \cdot (x^{-1} \cdot z) \end{array} \qquad \begin{array}{ccc} & (x \cdot y) \cdot (x \cdot y)^{-1} & \\ \swarrow_{l_2} & & \searrow_{l_1} \\ 1 & & x \cdot (y \cdot (x \cdot y)^{-1}) \end{array}$$

- $l_1|_1$ is "unifiable" with l_2 with substitution $\sigma :: \{x' \leftarrow x, y' \leftarrow x^{-1}, x \leftarrow x\} \leadsto \sigma(l_1|_1) = \sigma(l_2)$
- l_1 "unifiable" with l_2 with substitution $\sigma :: \{x' \leftarrow x, y' \leftarrow y, z \leftarrow (x \cdot y)^{-1}, x \leftarrow x \cdot y\} \leadsto \sigma(l_1) = \sigma(l_2)$

Subsumption, unification

Definition 9.6. *Subsumption ordering on terms:*
$s \preceq t$ iff $\exists \sigma$ substitution $: \sigma(s)$ subterm of t
$s \approx t$ iff $(s \preceq t \wedge t \preceq s)$
$s \succ t$ iff $(t \preceq s \wedge \neg(s \preceq t))$
$\succeq$ *is noetherian partial ordering over* $Term(F, V)$ *Proof!.*

Notice:
$O(\sigma(t)) = O(t) \cup \bigcup_{w \in O(t) : t|_w = x \in V} \{wv : v \in O(\sigma(x))\}$

Compatibility properties:
$t|_u = t' \leadsto \sigma(t)|_u = \sigma(t')$
$t|_u = x \in V \leadsto \sigma(t)|_{uv} = \sigma(x)|_v \quad (v \in O(\sigma(x)))$
$\sigma(t)[\sigma(t')]_u = \sigma(t[t']_u)$ for $u \in O(t)$

Definition 9.7. *$s, t \in Term(F, V)$ are unifiable iff there is a substitution σ with $\sigma(s) = \sigma(t)$. σ is called a unifier of s and t.*

Unification, Most General Unifier

Definition 9.8. *Let $V' \subseteq V, \sigma, \tau$ be substitutions.*

- $\sigma \preceq \tau \ (V')$ iff $\exists \rho$ substitution $: \rho \circ \sigma|_{V'} = \tau|_{V'}$
  Quote: σ is more general than τ over V'
- $\sigma \approx \tau \ (V')$ iff $\sigma \preceq \tau \ (V') \wedge \tau \preceq \sigma \ (V')$
- $\sigma \prec \tau \ (V')$ iff $\tau \preceq \sigma \ (V') \wedge \neg(\sigma \preceq \tau \ (V'))$
- *Notice: $\prec$ is noetherian partial ordering on the substitutions.*

Question: *Let s, t be unifiable. Is there a most general unifier $mgu(s, t)$ over $V = Var(s) \cup Var(t)$?*
i.e.. for any unifier σ of s, t always $mgu(s, t) \preceq \sigma \ (V)$ holds.

Is $mgu(s, t)$ unique? (up to variable renaming).

Unification's problem and its solution

Definition 9.9.

- *A unification's problem is given by a set $E = \{s_i \stackrel{?}{=} t_i : i = 1, ..., n\}$ of equations.*
- *σ is called a solution (or a unifier) in case that $\sigma(s_i) = \sigma(t_i)$ for $i = 1, ..., n$.*
- *If $\tau \succeq \sigma \ (Var(E))$ holds for each solution τ of E, then $mgu(E) := \sigma$ most general solution or most general unifier.*
- *Let $Sol(E)$ be the set of the solutions of E.*
  E and E' are equivalent, if $Sol(E) = Sol(E')$.
- *E' is in solved form, in case that*
  $E' = \{x_j \stackrel{?}{=} t_j : x_i \neq x_j \ (i \neq j), \ x_i \notin Var(t_j) \ (1 \leq i \leq j \leq m)\}$
- *E' is a solved form for E, if E' is in solved form and equivalent to E with $Var(E') \subseteq Var(E)$.*

Examples

Example 9.10. *Consider*

- $s = f(x, g(x,a)) \stackrel{?}{=} f(g(y,y),z) = t$
  $$\begin{array}{lll} \rightsquigarrow x \stackrel{?}{=} g(y,y) & g(x,a) \stackrel{?}{=} z & \textit{split} \\ \rightsquigarrow x \stackrel{?}{=} g(y,y) & g(g(y,y),a) \stackrel{?}{=} z & \textit{merge} \\ \rightsquigarrow \sigma :: x \leftarrow g(y,y) & z \leftarrow g(g(y,y),a) & y \leftarrow y \end{array}$$
- $f(x,a) \stackrel{?}{=} g(a,z)$ *unsolvable (not unifiable).*
- $x \stackrel{?}{=} f(x,y)$ *unsolvable, since* $f(x,y)$ *not* x *free.*
- $x \stackrel{?}{=} f(a,y) \rightsquigarrow$ *solution* $\sigma :: x \leftarrow f(a,y)$ *is the most general solution.*

Inference system for the unification

Definition 9.11. *Calculus* ***UNIFY****. Let* $\sigma =$ *be the binding set.*

(1) *Erase* $$\frac{(E \cup \{s \stackrel{?}{=} s\}, \sigma)}{(E, \sigma)}$$

(2) *Split (Decompose)* $$\frac{(E \cup \{f(s_1,\dots,s_m) \stackrel{?}{=} g(t_1,\dots,t_n)\}, \sigma)}{\lightning \ (\textit{unsolvable})} \quad \textit{if } f \neq g$$

$$\frac{(E \cup \{f(s_1,\dots,s_m) \stackrel{?}{=} f(t_1,\dots,t_m)\}, \sigma)}{(E \cup \{s_i \stackrel{?}{=} t_i : i = 1,\dots,m\}, \sigma)}$$

(3) *Merge (Solve)* $$\frac{(E \cup \{x \stackrel{?}{=} t\}, \sigma)}{(\tau(E), \sigma \cup \tau)} \quad \textit{if } x \notin Var(t), \tau = \{x \stackrel{?}{=} t\}$$

"occur check" $$\frac{(E \cup \{x \stackrel{?}{=} t\}, \sigma)}{\lightning \ (\textit{unsolvable})} \quad \textit{if } x \in Var(t) \wedge x \neq t$$

Unification algorithms

Unification algorithms based on UNIFY start always with $(E_0, S_0) := (E, \emptyset)$ and return a sequence $(E_0, S_0) \vdash_{UNIFY} \dots \vdash_{UNIFY} (E_n, S_n)$
They are successful in case they end with $E_n = \emptyset$, unsuccessful in case they end with $S_n = \lightning$. S_n defines a substitution σ which represents $Sol(S_n)$ and consequently also $Sol(E)$.

Lemma 9.12. *Correctness.*
Each sequence $(E_0, S_0) \vdash_{UNIFY} \dots \vdash_{UNIFY} (E_n, S_n)$ *terminates: either with* $\lightning$ *(unsolvable, not unifiable) or with* $(\emptyset, S)$ *and* S *is a solved form for* E.

Notice: Representations in solved form can be quite different (Complexity!!)
$s \stackrel{?}{=} f(x_1,\dots,x_n)$ $\quad t \stackrel{?}{=} f(g(x_0,x_0),\dots,g(x_{n-1},x_{n-1}))$
$S = \{x_i \stackrel{?}{=} g(x_{i-1},x_{i-1}) : i = 1,\dots,n\}$ and
$S_1 = \{x_{i+1} \stackrel{?}{=} t_i : t_0 = g(x_0,x_0), t_{i+1} = g(t_i,t_i) \ i = 0,\dots,n-1\}$
are both in solved form. The size of t_i grows exponentially with i.

Example

Example 9.13. *Execution:*

$$f(x, g(a,b)) \stackrel{?}{=} f(g(y,b),x)$$

E_i	S_i	*rule*
$f(x,g(a,b)) \stackrel{?}{=} f(g(y,b),x)$	$\emptyset$	
$x \stackrel{?}{=} g(y,b), x \stackrel{?}{=} g(a,b)$	$\emptyset$	*split*
$g(y,b) \stackrel{?}{=} g(a,b)$	$x \stackrel{?}{=} g(a,b)$	*solve*
$y \stackrel{?}{=} a, b \stackrel{?}{=} b$	$x \stackrel{?}{=} g(a,b)$	*split*
$b \stackrel{?}{=} b$	$x \stackrel{?}{=} g(a,b), y \stackrel{?}{=} a$	*solve*
	$x \stackrel{?}{=} g(a,b), y \stackrel{?}{=} a$	*delete*

Solution: $mgu = \sigma = \{x \leftarrow g(a,b), y \leftarrow a\}$

Critical pairs - Local confluence

Definition 9.14. *Let R be a rule system and $l_1 \to r_1, l_2 \to r_2 \in R$ with $V(l_1) \cap V(l_2) = \emptyset$ (renaming of variables if necessary, $l_1 \approx l_2$ resp. $l_1 \to r_1 \approx l_2 \to r_2$ are allowed).*

Let $u \in O(l_1)$ with $l_1|_u \notin V$ s.t. $\sigma = mgu(l_1|_u, l_2)$ exists.

$\sigma(l_1)$ is called then a overlap (superposition) of $l_2 \to r_2$ in $l_1 \to r_1$ and $(\sigma(r_1), \sigma(l_1[r_2]_u))$ is the associated critical pair to the overlap $l_1 \to r_1, l_2 \to r_2, u \in O(l_1)$, provided that $\sigma(r_1) \neq \sigma(l_1[r_2]_u)$.

Let $CP(R)$ be the set of all the critical pairs that can be constructed with rules of R.

Notice: The overlaps and consequently the set of critical pairs is unique up to renaming of the variables.

Examples

Example 9.15. *Consider*

- $f(f(\underline{x}, \underline{y}), z) \to f(x, f(y, z)) \qquad f(\underline{f(x', y')}, \underline{z'}) \to f(x', f(y', z'))$
  unifiable with $x \leftarrow f(x', y'), y \leftarrow z'$

$$f(f(f(x', y'), z'), z)$$
$$\swarrow \qquad \searrow$$
$$t_1 = f(f(x', y'), f(z', z)) \qquad\qquad f(f(x', f(y', z')), z) = t_2$$

- $t = f(x, g(x, a)) \to h(x) \qquad h(x') \to g(x', x'), t|_1 = t|_{21} = x$
  no critical pairs. Consider variable overlaps:

$$f(h(z), g(h(z), a)))$$
$$\swarrow \qquad \searrow$$
$$t_1 = h(h(z)) \qquad\qquad f(g(z, z), g(h(z), a)) = t_2$$
$$\searrow \qquad\qquad \downarrow$$
$$f(g(z, z), g(g(z, z), a))$$
$$\swarrow$$
$$h(g(z, z))$$

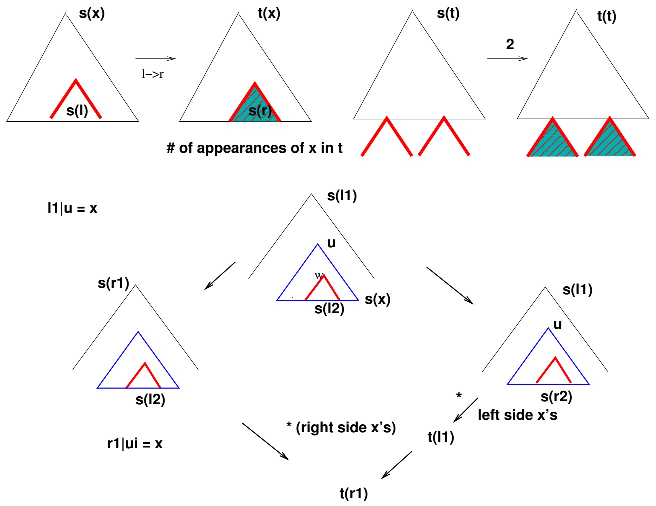
Properties

- Let σ, τ be substitutions, $x \in V$, $\sigma(y) = \tau(y)$ for $y \neq x$ and $\sigma(x) \to_R \tau(x)$. Then for each term t holds:
$$\sigma(t) \xrightarrow{*}_R \tau(t)$$
- Let $l_1 \to r_1, l_2 \to r_2$ be rules, $u \in O(l_1), l_1|_u = x \in V$. Let $\sigma(x)|_w = \sigma(l_2)$, i.e.. $\sigma(l_2)$ is introduced by $\sigma(x)$.
  Then $\quad t_1 \downarrow_R t_2$ holds for
$$t_1 := \sigma(r_1) \leftarrow \sigma(l_1) \to \sigma(l_1)[\sigma(r_2)]_{uw} =: t_2$$

Lemma 9.16. *Critical-Pair Lemma of Knuth/Bendix*
Let R be a rule system. Then the following holds:

from $t_1 \leftarrow_R t \to_R t_2$ either $t_1 \downarrow_R t_2$ or $t_1 \leftrightarrow_{CP(R)} t_2$ hold.

Proofs



Confluence test

Theorem 9.17. *Main result:* *Let R be a rule system.*

- *R is locally confluent iff all the pairs $(t_1, t_2) \in CP(R)$ are joinable.*
- *If R is terminating, then:*
  R confluent iff $(t_1, t_2) \in CP(R) \rightsquigarrow t_1 \downarrow t_2$.
- *Let R be linear (i.e.. for $l, r \in l \to r \in R$ variables appear at most once). If $CP(R) = \emptyset$, then R is confluent.*

Example 9.18.
- *Let $R = \{f(x,x) \to a, f(x, s(x)) \to b, a \to s(a)\}$.*
  R is locally confluent,but not confluent:

$$a \leftarrow f(a,a) \to f(a, s(a)) \to b$$

but not $a \downarrow b$. R is neither terminating nor left-linear.

Example (Cont.)

- $R = \{f(f(x)) \to g(x)\}$

$$t_1 = g(f(x)) \leftarrow f(f(f(x))) \to f(g(x)) = t_2$$

It doesn't hold $t_1 \downarrow_R t_2 \rightsquigarrow R$ not confluent.
Add rule $t_1 \to t_2$ to R. R_1 is equivalent to R, terminating and confluent.

$$\begin{tikzcd}
 & g(f(f(x))) \arrow[dl] \arrow[dr] & \\
f(g(f(x))) \arrow[dr] & & g(g(x)) \\
 & f(f(g(x))) \arrow[ur] &
\end{tikzcd}$$

- $R = \{x + 0 \to x, x + s(y) \to s(x + y)\}$, linear without critical pairs $\rightsquigarrow$ confluent.
- $R = \{f(x) \to a, f(x) \to g(f(x)), g(f(x)) \to f(h(x)), g(f(x)) \to b\}$ is locally confluent but not confluent.

Confluence without Termination

Definition 9.19. *$\epsilon - \epsilon$ - Properties.* *Let $\xrightarrow{\epsilon} = \xrightarrow{0} \cup \xrightarrow{1}$.*

- *R is called $\epsilon - \epsilon$ closed , in case that for each critical pair $(t_1, t_2) \in CP(R)$ there exists a t with $t_1 \underset{R}{\xrightarrow{\epsilon}} t \underset{R}{\xleftarrow{\epsilon}} t_2$.*
- *R is called $\epsilon - \epsilon$ confluent iff $\underset{R}{\leftarrow} \circ \underset{R}{\to} \subseteq \underset{R}{\xrightarrow{\epsilon}} \circ \underset{R}{\xleftarrow{\epsilon}}$*

Consequence 9.20.
- *$\to$ $\epsilon - \epsilon$ confluent $\rightsquigarrow$ $\to$ strong-confluent.*
- *R $\epsilon - \epsilon$ closed $\not\Rightarrow$ R $\epsilon - \epsilon$ confluent*
  $R = \{f(x,x) \to a, f(x, g(x)) \to b, c \to g(c)\}$. $CP(R) = \emptyset$, i.e..
  R $\epsilon - \epsilon$ closed but $a \leftarrow f(c,c) \to f(c, g(c)) \to b$, i.e.. R not confluent $\lightning$.
- *If R is linear and $\epsilon - \epsilon$ closed , then R is strong-confluent, thus confluent (prove that R is $\epsilon - \epsilon$ confluent).*

These conditions are unfortunately too restricting for programming.

Example

Example 9.21. *R left linear $\epsilon - \epsilon$ closed* *is not sufficient:*
$R = \{f(a,a) \to g(b,b), a \to a', f(a', x) \to f(x,x), f(x, a') \to f(x,x),$
$g(b,b) \to f(a,a), b \to b', g(b', x) \to g(x,x), g(x, b`) \to g(x,x)\}$

It holds $f(a', a') \underset{R}{\overset{*}{\longleftrightarrow}} g(b', b')$ but not $f(a', a') \downarrow_R g(b', b')$.
R left linear $\epsilon - \epsilon$ closed :

$$\begin{tikzcd}
 & f(a,a) \arrow[dl] \arrow[d] \arrow[dr] & \\
g(b,b) \arrow[dr] & f(a',a) \arrow[d] \arrow[dr] & f(a,a') \arrow[d] \\
 & f(a,a) & f(a',a') \arrow[d] \\
 & & f(a',a')
\end{tikzcd} .$$

Parallel reduction

Notice: Let $\to, \Rightarrow$ with $\xrightarrow{*} = \overset{*}{\Rightarrow}$. (Often: $\to \subseteq \Rightarrow \subseteq \xrightarrow{*}$).
Then $\to$ is confluent iff $\Rightarrow$ confluent.

Definition 9.22. *Let R be a rule system.*

- *The parallel reduction, $\mapsto_R$, is defined through $t \mapsto_R t'$ iff $\exists U \subset O(t) : \forall u_i, u_j (u_i \neq u_j \leadsto u_i | u_j)$ $\exists l_i \to r_i \in R, \sigma_i$ with $t|_{u_i} = \sigma_i(l_i) :: t' = t[\sigma_i(r_i)]_{u_i} (u_i \in U)$ $(t[u_1 \leftarrow \sigma_1(r_1)]...t[u_n \leftarrow \sigma_1(r_n)])$.*
- *A critical pair of R : $(\sigma(r_1), \sigma(l_1[r_2]_u))$ is parallel 0-joinable in case that $\sigma(l_1[r_2]_u) \mapsto_R \sigma(r_1)$.*
- *R is parallel 0-closed in case that each critical pair of R is parallel 0-joinable.*

Properties: $\mapsto_R$ is stable and monotone. It holds $\overset{*}{\mapsto}_R = \xrightarrow{*}_R$ and consequently, if $\mapsto_R$ is confluent then $\to_R$ too.

Parallel reduction

Theorem 9.23. *If R is left-linear and parallel 0-closed, then $\mapsto_R$ is strong-confluent, thus confluent, and consequently R is also confluent.*

Consequence 9.24.

- *If R fulfills the O'Donnel condition, then R is confluent. O'Donnel's condition: R left-linear, $CP(R) = \emptyset$, R left-sequential (Redexes are unambiguous when reading the terms from left to right: $f(g(x,a),y) \to 0, g(b,c) \to 1$ has not this property).*
  By regrouping of the arguments, the property can frequently be achieved, for instance $f(g(a,x),y) \to 0, g(b,c) \to 1$
- *Orthogonal systems:: R left-linear and $CP(R) = \emptyset$, so R confluent. (In the literature denominated also as regular systems).*
- *Variations: R is strongly-closed, in case that for each critical pair (s,t) there are terms u, v with $s \xrightarrow{*} u \xleftarrow{\leq 1} t$ and $s \xrightarrow{\leq 1} v \xleftarrow{*} t$.*
  R linear and strongly-closed, so R strong-confluent.

Consequences

- Does confluence follow from $CP(R) = \emptyset$? **No.**
  $R = \{f(x,x) \to a, g(x) \to f(x, g(x)), b \to g(b)\}$.
  Consider $g(b) \to f(b, g(b)) \to f(g(b), g(b)) \to a$
  "Outermost" reduction.
  $g(b) \to g(g(b)) \xrightarrow{*} g(a) \to f(a, g(a))$ not joinable.
- Regular systems can be non terminating:
  $\{f(x,b) \to d, a \to b, c \to c\}$. Evidently $CP = \emptyset$.
  $f(c,a) \to f(c,b) \to d$
  $\downarrow *$
  $f(c,a) \to f(c,b)$. Notice that $f(c,a)$ has a normal form. $\leadsto$
  Reduction strategies that are normalizing or that deliver shortest reduction sequences.
- A context is a term with "holes" $\square$, e.g. $f(g(\square, s(0)), \square, h(\square))$ as "tree pattern" (pattern) for rule $f(g(x, s(0)), y, h(z)) \to x$. The holes can be filled freely. Sequentiality is defined using this notion.

Termination-Criteria

Theorem 9.25. *R is terminating iff there is a noetherian partial ordering $\succ$ over the ground terms $Term(F)$, that is monotone, so that $\sigma(l) \succ \sigma(r)$ holds for each rule $l \to r \in R$ and ground substitution σ.*

Proof:$\curvearrowright$ Define $s \succ t$ iff $s \xrightarrow{+} t$ $(s, t \in Term(F))$
$\curvearrowleft$ Asume that $\to_R$ not terminating, $t_0 \to t_1 \to ...(V(t_i) \subseteq V(t_0))$.
Let σ be a ground substitution with $V(t_0) \subset D(\sigma)$, then $\sigma(t_0) \succ \sigma(t_1) \succ ... \lightning$.
Problem: infinite test.

Definition 9.26. *A* ***reduction ordering*** *is partial ordering $\succ$ over $Term(F, V)$ with*
(i) $\succ$ is noetherian (ii) $\succ$ is stable and (iii) $\succ$ is monotone.

Theorem 9.27. *R is noetherian iff there exists a reduction ordering $\succ$ with $l \succ r$ for every $l \to r \in R$*

Termination's criteria

Notice: There are no total reduction orderings for terms with variables..
$x \succ y? \rightsquigarrow \sigma(x) \succ \sigma(y)$
$f(x,y) \succ f(y,x)$? commutativity cannot be oriented.
Examples for reduction orderings:
Knuth-Bendix ordering: Weight for each function symbol and precedence over F.
Recursive path ordering (RPO): precedence over F is recursively extended to paths (words) in the terms that are to be compared.
Lexicographic path ordering(LPO), polynomial interpretations, etc.

$f(f(g(x)))$	$=$	$f(h(x))$	$f(f(x))$	$=$	$g(h(g(x)))$	$f(h(x))$	$=$	$h(g(x)$
KB	$\rightarrow$	$l(f)=3$	$l(g)=2$	$\rightarrow$	$l(h)=$	1	$\rightarrow$	
RPO	$\leftarrow$	$g>h$	$>f$	$\leftarrow$			$\leftarrow$	

Confluence modulo equivalence relation (e.g. AC):
$R :: f(x,x) \to g(x)$ $\quad G :: \{(a,b)\}$ $\quad g(a) \leftarrow f(a,a) \sim f(a,b)$ but not $g(a) \downarrow_\sim f(a,b)$.

Knuth-Bendix Completion method

Input: E set of equations, $\succ$ reduction ordering, $R = \emptyset$.

Repeat while E not empty
(1) Remove $t = s$ of E with $t \succ s$, $R := R \cup \{t \to s\}$ else abort
(2) Bring the right side of the rules to normal form with R
(3) Extend E with every normalized critical pair generated by $t \to s$ with R
(4) Remove all the rules from R, whose left side is properly larger than t w.r.t. to the subsumption ordering.
(5) Use R to normalize both sides of equations of E.
Remove identities.

Output: 1) Termination with R convergent, equivalent to E. 2) Abortion 3) not termination (it runs infinitely).

Examples for Knuth-Bendix-Procedure

Example 9.28. ▶ *SRS::* $\Sigma = \{a,b,c\}, E = \{a^2 = \lambda, b^2 = \lambda, ab = c\}$
$u < v$ *iff* $|u| < |v|$ *or* $|u| = |v|$ *and* $u <_{lex} v$ *with* $a <_{lex} b <_{lex} c$

$E_0 = \{a^2 = \lambda, b^2 = \lambda, ab = c\}, R_0 = \emptyset$

$E_1 = \{b^2 = \lambda, ab = c\}, R_1 = \{a^2 \to \lambda\}, CP_1 = \emptyset$

$E_2 = \{ab = c\}, R_2 = \{a^2 \to \lambda, b^2 \to \lambda\}, CP_2 = \emptyset$

$R_3 = \{a^2 \to \lambda, b^2 \to \lambda, ab \to c\}, NCP_3 = \{(b, ac), (a, cb)\}$
$E_3 = \{b = ac, a = cb\}$

$R_4 = \{a^2 \to \lambda, b^2 \to \lambda, ab \to c, ac \to b\}, NCP_4 = \emptyset, E_4 = \{a = cb\}$

$R_5 = \{a^2 \to \lambda, b^2 \to \lambda, ab \to c, ac \to b, cb \to a\}, NCP_5 = \emptyset, E_5 = \emptyset$

Examples for Knuth-Bendix-Completion

- $E = \{ffg(x) = h(x), ff(x) = x, fh(x) = g(x)\}$ $\quad >: KBO(3,2,1)$
  $R_0 = \emptyset, E_0 = E$
  $R_1 = \{ffg(x) \to h(x)\}, KP_1 = \emptyset.E_1 = \{ff(x) = x, fh(x) = g(x)\}$
  $R_2 = \{ffg(x) \to h(x), ff(x) \to x\}, NKP_2 = \{(g(x), h(x))\},$
  $E_2 = \{fh(x) = g(x), g(x) = h(x)\}, R_2 = \{ff(x) \to x\}$
  $R_3 = \{ff(x) \to x, fh(x) \to g(x)\}, NKP_3 = \{(h(x), fg(x))\}, E_3 = \{g(x) = h(x), h(x) = fg(x)\}$
  $R_4 = \{ff(x) \to x, fh(x) \to h(x), g(x) \to h(x)\}, NKP_3 = \emptyset, E_4 = \emptyset$
- $E = \{fgf(x) = gfg(x)\}$ $\quad >: LL :: f > g$
  $R_0 = \emptyset, E_0 = E$
  $R_1 = \{fgf(x) \to gfg(x)\}, NKP_1 = \{(gfggf(x), fggfg(x))\}, E_1 = \{gfggf(x) = fggfg(x)\}$
  $R_1 = \{fgf(x) \to gfg(x), fggfg(x) \to gfggf(x)\}, NKP_2 = \{(gfggfggfg(x), fgggfggfg(x), ..\}...$

Refined Inference system for Completion

Definition 9.29. *Let $>$ be a noetherian PO over $Term(F,V)$. The inference system $\mathcal{P}_{TES}$ is composed by the following rules:*

(1) *Orientate* $\quad \dfrac{(E \cup \{s \doteq t\}, R)}{(E, R \cup \{s \to t\})}$ *in case that* $s > t$

(2) *Generate* $\quad \dfrac{(E, R)}{(E \cup \{s \doteq t\}, R)}$ *in case that* $s \leftarrow_R \circ \to_R t$

(3) *Simplify EQ* $\quad \dfrac{(E \cup \{s \doteq t\}, R)}{(E \cup \{u \doteq t\}, R)}$ *in case that* $s \to_R u$

(4) *Simplify RS* $\quad \dfrac{(E, R \cup \{s \to t\})}{(E, R \cup \{s \to u\})}$ *in case that* $t \to_R u$

(5) *Simplify LS* $\quad \dfrac{(E, R \cup \{s \to t\})}{(E \cup \{u \doteq t\}, R)}$ *in case that* $s \to_R u$ *with* $l \to r$ *and* $s \succ l$ *(SubSumOrd.)*

(6) *Delete identities*

Equational implementations

Programming = Description of algorithms in a formal system

Definition 10.1. *Let $f : M_1 \times ... \times M_n \rightsquigarrow M_{n+1}$ be a (partial) function. Let $T_i, 1 = 1...n+1$ be decidable sets of ground terms over Σ, $\hat{f}$ n-ary function symbol, E set of equations.*

A data interpretation $\mathfrak{I}$ is a function $\mathfrak{I} : T_i \to M_i$.

$\hat{f}$ implements f under the interpretation $\mathfrak{I}$ in E iff
1) $\mathfrak{I}(T_i) = M_i \quad (i = 1...n+1)$
2) $f(\mathfrak{I}(t_1), ..., \mathfrak{I}(t_n)) = \mathfrak{I}(t_{n+1})$ iff $\hat{f}(t_1, ..., t_n) =_E t_{n+1}$ $(\forall t_i \in T_i)$

$$\begin{array}{ccc} T_1 \times ... \times T_n & \xrightarrow{\hat{f}} & T_{n+1} \\ \mathfrak{I}\downarrow \quad\quad \mathfrak{I}\downarrow & & \mathfrak{I}\downarrow \\ M_1 \times ... \times M_n & \xrightarrow{f} & M_{n+1} \end{array}$$

Abbreviation: $(\hat{f}, E, \mathfrak{I})$ implements f.

Equational implementations

Theorem 10.2. *Let E be set of equations or rules (same notations). For every $i = 1, ..., n+1$ assume*
1) $\mathfrak{I}(T_i) = M_i$
2a) $f(\mathfrak{I}(t_1), ..., \mathfrak{I}(t_n)) = \mathfrak{I}(t_{n+1}) \rightsquigarrow \hat{f}(t_1, ..., t_n) =_E t_{n+1}$ $(\forall t_i \in T_i)$

$\hat{f}$ implements the total function f under $\mathfrak{I}$ in E when one of the following conditions holds:

a) $\forall t, t' \in T_{n+1} : t =_E t' \rightsquigarrow \mathfrak{I}(t) = \mathfrak{I}(t')$
b) E confluent and $\forall t \in T_{n+1} : t \to_E t' \rightsquigarrow t' \in T_{n+1} \wedge \mathfrak{I}(t) = \mathfrak{I}(t')$
c) E confluent and T_{n+1} contains only E-irreducible terms.

Application: Assume $(\hat{f}, E, \mathfrak{I})$ implements the total function f. If E is extended by E_0 under retention of $\mathfrak{I}$, then 1 and 2a still hold. If one of the criteria a, b, c are fullfiled for $E \cup E_0$, then $(\hat{f}, E \cup E_0, \mathfrak{I})$ implements also the function f. This holds specially when $E \cup E_0$ is confluent and T_{n+1} contains only $E \cup E_0$ irreducible terms.

Equational implementations

Theorem 10.3. *Let $(\hat{f}, E, \mathfrak{I})$ implement the (partial) function f. Then*

a) $\forall t, t' \in T_{n+1} :: \mathfrak{I}(t) = \mathfrak{I}(t') \wedge \mathfrak{I}(t) \in Image(f) \rightsquigarrow t =_E t'$
b) Let E be confluent and T_{n+1} contains only normal forms of E. Then $\mathfrak{I}$ is injective on $\{t \in T_{n+1} : \mathfrak{I}(t) \in Image(f)\}$.

Theorem 10.4. *Criterion for the implementation of total functions. Assume*

1) $\mathfrak{I}(T_i) = M_i \quad (i = 1, ..., n+1)$
2) $\forall t, t' \in T_{n+1} :: \mathfrak{I}(t) = \mathfrak{I}(t')$ iff $t =_E t'$
3) $\forall_{1 \leq i \leq n} \;\; t_i \in T_i \;\; \exists t_{n+1} \in T_{n+1} ::$

$$\hat{f}(t_1, ..., t_n) =_E t_{n+1} \wedge f(\mathfrak{I}(t_1), ... \mathfrak{I}(t_n)) = \mathfrak{I}(t_{n+1})$$

Then $\hat{f}$ implements the function f under $\mathfrak{I}$ in E and f is total.

Notice: If T_{n+1} contains only normal forms and E is confluent, so 2) is fulfilled, in case $\mathfrak{I}$ is injective on T_{n+1}.

Equational implementations

Theorem 10.5. *Let $(\hat{f}, E, \mathfrak{I})$ implement $f : M_1 \times \ldots \times M_n \to M_{n+1}$. Let $S_i = \{t \in T_i :: \exists t_0 \in T_i : t \neq t_0, \mathfrak{I}(t) = \mathfrak{I}(t_0) \quad t \xrightarrow{+}_E t_0\}$ be recursive sets.*
Then $\hat{f}$ implements also f with term sets $T_i' = T_i \backslash S_i$ under $\mathfrak{I}|_{T_i'}$ in E.

So we can delete terms of T_i that are reducible to other terms of T_i with the same $\mathfrak{I}$-value. Consequently the restriction to E-normal forms is allowed.

Consequence 10.6.

- *Implementations can be composed.*
- *If we extend E by E- consequences then the implementation property is preserved.*
  This is important for the KB-Completion since only E-consequences are added.

Examples: Propositional logic, natural numbers

Example 10.7. *Convention: Equations define the signature. Occasionally variadic functions and overloading. Single sorted.*

Boolean algebra: Let $M = \{true, false\}$ with $\wedge, \vee, \neg, \supset, \ldots$.
Constants tt, ff. Term set $Bool := \{tt, ff\}$, $\mathfrak{I}(tt) = true, \mathfrak{I}(ff) = false$.
Strategy: Avoid rules with tt or ff as left side. According to theorem 10.2 c) we can add equations with these restrictions without influencing the implementation property, as long as confluence is achieved.
Consider the following rules:

(1) $cond(tt, x, y) \to x$ (2) $cond(ff, x, y) \to y$. (help function).
(3) $x \ vel \ y \to cond(x, tt, y)$
$E = \{(1), (2), (3)\}$ is confluent. Hence: $tt \ vel \ y =_E cond(tt, tt, y) =_E tt$ holds, i.e.

$$(*_1) \quad tt \ vel \ y = tt \text{ and } \quad (*_2) \quad x \ vel \ tt = cond(x, tt, tt)$$

$x \ vel \ tt = tt$ cannot be deduced out of E.
However vel implements the function $\vee$ with E.

Examples: Propositional logic

According to theorem 10.4, we must prove the conditions (1), (2), (3):
$\forall t, t' \in \ Bool \ \exists \bar{t} \in \ Bool :: \mathfrak{I}(t) \vee \mathfrak{I}(t') = \mathfrak{I}(\bar{t}) \wedge t \ vel \ t' =_E \bar{t}$
For $t = tt$ $(*_1)$ and $t = ff$ (2) since $ff \ vel \ t' \to_E cond(ff, tt, t') \to_E t'$
Thus $x \ vel \ tt \neq_E \ tt$ but $tt \ vel \ tt =_E \ tt$, $\ ff \ vel \ tt =_E \ tt$.

MC Carthy's rules for *cond*:

$$(1) \ cond(tt, x, y) = x \quad (2) \ cond(ff, x, y) = y \quad (*) \ cond(x, \ tt, \ tt) = \ tt$$

Notice Not identical with *cond* in Lisp. Difference: Evaluation strategy.
Consider
$(**)$ $cond(x, cond(x, y, z), u) \to cond(x, y, u)$
$\rightsquigarrow$ $E' = \{(1), (2), (3), (*), (**)\}$ is terminating and confluent.
Conventions: Sets of equations contain always (1), (2), (3) and $x \ et \ y \to cond(x, y, \ ff)$.
Notation: $cond(x, y, z) :: [x \to y, z]$ or
$[x \to y_1, x_2 \to y_2, \ldots, x_n \to y_n, z]$ for $[x \to \ [\ldots]\ldots, z]$

Examples: Semantical arguments

Properties of the implementing functions:
$(vel, E, \mathfrak{I})$ implements $\vee$ of BOOL.

Statement: *vel* is associative on *Bool*.
Prove: $\forall t_1, t_2, t_3 \in Bool : t_1 \ vel \ (t_2 \ vel \ t_3) =_E (t_1 \ vel \ t_2) \ vel \ t_3$

There exist $t, t', T, T' \in Bool$ with
$\mathfrak{I}(t_2) \vee \mathfrak{I}(t_3) = \mathfrak{I}(t)$ and $\mathfrak{I}(t_1) \vee \mathfrak{I}(t_2) = \mathfrak{I}(t')$ as well as
$\mathfrak{I}(t_1) \vee \mathfrak{I}(t) = \mathfrak{I}(T)$ and $\mathfrak{I}(t') \vee \mathfrak{I}(t_3) = \mathfrak{I}(T')$

Because of the semantical valid associativity of $\vee$
$\mathfrak{I}(T) = \mathfrak{I}(t_1) \vee \mathfrak{I}(t_2) \vee \mathfrak{I}(t_3) = \mathfrak{I}(T')$ holds.

Since *vel* implements $\vee$ it follows:
$t_1 \ vel \ (t_2 \ vel \ t_3) =_E t_1 \ vel \ t =_E T =_E T' =_E t' \ vel \ t_3 =_E (t_1 \ vel \ t_2) \ vel \ t_3$

Examples: Natural numbers

Function symbols: $\hat{0}, \hat{s}$ Ground terms: $\{\hat{s}^n(\hat{0})\ (n \geq 0)\}$
$\mathfrak{I}$ Interpretation $\mathfrak{I}(\hat{0}) = 0, \mathfrak{I}(\hat{s}) = \lambda x.x + 1$, i.e. $\mathfrak{I}(\hat{s}^n(\hat{0})) = n\ (n \geq 0)$.
Abbreviation: $\widehat{n+1} := \hat{s}(\hat{n})\ (n \geq 0)$
Number terms. $NAT = \{\hat{n} : n \geq 0\}$ normal forms (Theorem 10.2 c holds).

Important help functions over NAT:

Let $E = \{is\_null(\hat{0}) \to\ tt, is\_null(\hat{s}(x)) \to\ ff\}$.
is_null implements the predicate $Is\_Null : \mathbb{N} \to \{true, false\}$ Zero-test.
Extend E with (non terminating rules)
$\hat{g}(x) \to [is\_null(x) \to \hat{0}, \hat{g}(x)]$, $\quad \hat{f}(x) \to [is\_null(x) \to \hat{g}(x), \hat{0}]$

Statement:It holds under the standard interpretation $\mathfrak{I}$
$\hat{f}$ implements the null function $f(x) = 0\ \ (x \in \mathbb{N})$ and
$\hat{g}$ implements the function $g(0) = 0$ else undefined.
Because of $\hat{f}(\hat{0}) \to [is\_null(\hat{0}) \to \hat{g}(\hat{0}), \hat{0}] \xrightarrow{*} \hat{g}(\hat{0}) \to [...] \xrightarrow{*} \hat{0}$ and
$\hat{f}(\hat{s}(x)) \to [is\_null(\hat{s}(x)) \to \hat{g}(\hat{s}(x)), \hat{0}] \xrightarrow{*} \hat{0}$ (follows from theorem 10.4).

Examples: Natural numbers

Extension of E to E' with rule:

$\hat{f}(x, y) = [is\_null(x) \to y, \hat{0}]$ $\quad$ ($\hat{f}$ overloaded).

$\hat{f}$ implements the function $F : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$

$$F(x, y) = \begin{cases} y & x = 0 \\ 0 & x \neq 0 \end{cases} \qquad \begin{array}{l} \hat{f}(\hat{0}, \hat{y}) \xrightarrow{*} \hat{y} \\ \hat{f}(\hat{s}(x), \hat{y}) \xrightarrow{*} \hat{0} \end{array}$$

Nevertheless it holds:

$$\hat{f}(x, \hat{g}(x)) =_{E'} [is\_null(x) \to \hat{g}(x), \hat{0}]) =_{E'} \hat{f}(x)$$

But $f(n) = F(n, g(n))$ for $n > 0$ is not true.

If one wants to implement all the computable functions, then the recursion equations of Kleene cannot be directly used, since the composition of partial functions would be needed for it.

Representation of primitive recursive functions

The class $\mathfrak{P}$ contains the functions
$s = \lambda x.x + 1, \pi_i^n = \lambda x_1, ..., x_n.x_i,$ as well as $c = \lambda x.0$ on $\mathbb{N}$ and is closed w.r. to composition and primitive recursion, i.e.

$f(x_1, ..., x_n) = g(h_1(x_1, ..., x_n), ..., h_r(x_1, ..., x_n))$ $\qquad$ resp.

$f(x_1, ..., x_n, 0) = g(x_1, ..., x_n)$
$f(x_1, ..., x_n, y + 1) = h(x_1, ..., x_n, y, f(x_1, ..., x_n, y))$

Statement: $f \in \mathfrak{P}$ is implementable by $(\hat{f}, E_{\hat{f}}, \mathfrak{I})$

Idea: Show for suitable $E_{\hat{f}}$:

$\hat{f}(\hat{k_1}, ..., \hat{k_n}) \xrightarrow{*}_{E_{\hat{f}}} \widehat{f(k_1, ..., k_n)}$ with $E_{\hat{f}}$ confluent and terminating.

Assumption: $FUNKT$ (signature) contains for every $n \in \mathbb{N}$ a countable number of function symbols of arity n.

Implementation of primitive recursive functions

Theorem 10.8. *For each finite set $A \subset FUNKT \setminus \{\hat{0}, \hat{s}\}$ the exception set, and each function $f : \mathbb{N}^n \to \mathbb{N}, f \in \mathfrak{P}$ there exist $\hat{f} \in FUNKT$ and $E_{\hat{f}}$ finite, confluent and terminating such that $(\hat{f}, E_{\hat{f}}, \mathfrak{I})$ implements f and none of the equations in $E_{\hat{f}}$ contains function symbols from A.*

Proof: Induction over construction of $\mathfrak{P}$: $\hat{0}, \hat{s} \notin A$. Set $A' = A \cup \{\hat{0}, \hat{s}\}$

- $\hat{s}$ implements s with $E_{\hat{s}} = \emptyset$
- $\hat{\pi}_i^n \in FUNKT^n \setminus A'$ implem. π_i^n with $E_{\hat{\pi}_i^n} = \{\hat{\pi}_i^n(x_1, ..., x_n) \to x_i\}$
- $\hat{c} \in FUNKT^1 \setminus A'$ implements c with $E_{\hat{c}} = \{\hat{c}(x) \to 0\}$
- Composition: $[\hat{g}, E_{\hat{g}}, A_0]$, $\quad [\hat{h}_i, E_{\hat{h}_i}, A_i]$ with
$A_i = A_{i-1} \cup \{f \in FUNKT : f \in E_{\hat{h}_{i-1}}\} \setminus \{\hat{0}, \hat{s}\}$. Let $\hat{f} \in FUNKT \setminus A'_r$
and $E_{\hat{f}} = E_{\hat{g}} \cup \bigcup_1^r E_{\hat{h}_i} \cup \{\hat{f}(x_1, ..., x_n) \to \hat{g}(\hat{h}_1(...), ..., \hat{h}_r(...))\}$
- Primitive recursion: Analogously with the defining equations.

Implementation of primitive recursive functions

All the rules are left-linear without overlappings $\rightsquigarrow$ confluence.
Termination criteria: Let $\mathfrak{J} : \ FUNKT \to (\mathbb{N}^* \to \mathbb{N})$, i.e
$\mathfrak{J}(f) : \mathbb{N}^{st(f)} \to \mathbb{N}$, strictly monotonous in all the arguments. If E is a rule system, $l \to r \in E, b : VAR \to \mathbb{N}$ (assignment), if $\mathfrak{J}[b](l) > \mathfrak{J}[b](r)$ holds, then E terminates.
Idea: Use the Ackermann function as bound:
$A(0,y) = y+1, A(x+1,0) = A(x,1), A(x+1,y+1) = A(x, A(x+1,y))$
A is strictly monotonic,
$A(1,x) = x+2, A(x,y+1) \leq A(x+1,y), A(2,x) = 2x+3$
For each $n \in \mathbb{N}$ there is a β_n with $\quad \sum_1^n A(x_i, x) \leq A(\beta_n(x_1, ..., x_n), x)$

Define $\mathfrak{J}$ through $\mathfrak{J}(\hat{f})(k_1, ..., k_n) = A(p_{\hat{f}}, \sum k_i)$ with suitable $p_{\hat{f}} \in \mathbb{N}$.

- $p_{\hat{s}} := 1 :: \mathfrak{J}[b](\hat{s}(x)) = A(1, b(x)) = b(x) + 2 > b(x) + 1 = \mathfrak{J}[b](x \hat{+} 1)$
- $p_{\hat{\pi}_i^n} := 1 :: \mathfrak{J}[b](\hat{\pi}_i^n(x_1, ..., x_n)) = A(1, \sum_1^n b(x_i)) > b(x_i) = \mathfrak{J}[b](x_i)$
- $p_{\hat{c}} := 1 :: \mathfrak{J}[b](\hat{c}(x)) = A(1, b(x)) > 0 = \mathfrak{J}[b](\hat{0})$

Implementation of primitive recursive functions

- Composition: $f(x_1, ..., x_n) = g(h_1(...), ..., h_r(...))$.
  Set $c^* = \beta_r(p_{\hat{h}_1}, ..., p_{\hat{h}_r})$ and $p_{\hat{f}} := p_{\hat{g}} + c^* + 2$. Check that
  $\mathfrak{J}[b](\hat{f}(x_1, ..., x_n)) > \mathfrak{J}[b](\hat{g}(\hat{h}_1(x_1, ..., x_n), ..., \hat{h}_r(x_1, ..., x_n)))$
- Primitive recursion:
  Set $m = \ max(p_{\hat{g}}, p_{\hat{f}})$ and $p_{\hat{f}} := m + 3$. Check that
  $\mathfrak{J}[b](\hat{f}(x_1, ..., x_n, 0)) > \mathfrak{J}[b](\hat{g}(x_1, ..., x_n))$ and
  $\mathfrak{J}[b](\hat{f}(x_1, ..., x_n, \hat{s}(y))) > \mathfrak{J}[b](\hat{g}(....))$.
  Apply $A(m+3, k+3) > A(p_{\hat{h}}, k + A(p_{\hat{f}}, k))$
- By induction show that
  $\hat{f}(\hat{k}_1, ..., \hat{k}_n) \xrightarrow{*}_{E_{\hat{f}}} \widehat{f(k_1, ..., k_n)}$
- From the theorem 10.4 the statement follows.

Representation of recursive functions

Minimization:: μ-Operator $\mu_y[g(x_1, ..., x_n, y) = 0] = z$ iff
i) $g(x_1, ..., x_n, i)$ defined $\neq 0$ for $0 \leq i < z \quad$ ii) $g(x_1, ..., x_n, z) = 0$

Regular minimization: μ is applied to total functions for which
$\forall x_1, ..., x_n \exists y : g(x_1, ..., x_n, y) = 0$

$\mathfrak{R}$ is closed w.r. to composition, primitive recursion and regular minimization.

Show that: regular minimization is implementable with exception set A.
Assume $\hat{g}, E_{\hat{g}}$ implement g where $\hat{g}(\hat{k}_1, ..., \hat{k}_{n+1}) \xrightarrow{*}_{E_{\hat{g}}} \widehat{g(k_1, ..., k_{n+1})}$
Let $\hat{f}, \hat{f}^+, \hat{f}^*$ be new and $\quad E_{\hat{f}} := E_{\hat{g}} \cup \{\hat{f}(x_1, ..., x_n) \to \hat{f}^*(x_1, ..., x_n, \hat{0}),$
$\hat{f}^*(x_1, ..., x_n, y) \to \hat{f}^+(\hat{g}(x_1, ..., x_n, y), x_1, ..., x_n, y),$
$\hat{f}^+(\hat{0}, x_1, ..., x_n, y) \to y, \hat{f}^+(\hat{s}(x), x_1, ..., x_n, y) \to \hat{f}^*(x_1, ..., x_n, \hat{s}(y))\}$

Claim: $(\hat{f}, E_{\hat{f}})$ implements the minimization of g.

Implementation of recursive functions

Assumption: For each $k_1, ..., k_n \in \mathbb{N}$ there is a smallest $k \in \mathbb{N}$ with $g(k_1, ..., k_n, k) = 0$
Claim: For every $i \in \mathbb{N}, i \leq k \quad \hat{f}^*(\hat{k}_1, ..., \hat{k}_n, (\widehat{k - i})) \xrightarrow{*}_{E_{\hat{f}}} \hat{k}$ holds
Proof: induction over i:

- $i = 0 :: \hat{f}^*(\hat{k}_1, ..., \hat{k}_n, \hat{k}) \to \hat{f}^+(\hat{g}(\hat{k}_1, ..., \hat{k}_n, \hat{k}), \hat{k}_1, ..., \hat{k}_n, \hat{k}) \xrightarrow{*}_{E_{\hat{g}}}$
  $\hat{f}^+(\widehat{g(k_1, ..., k_n, k)}, \hat{k}_1, ..., \hat{k}_n, \hat{k}) \to \hat{k}$
- $i > 0 :: \hat{f}^*(\hat{k}_1, ..., \hat{k}_n, k - (\hat{i} + 1)) \to$
  $\hat{f}^+(\hat{g}(\hat{k}_1, ..., \hat{k}_n, k - (\hat{i} + 1)), \hat{k}_1, ..., \hat{k}_n, k - (\hat{i} + 1)) \xrightarrow{*}_{E_{\hat{g}}}$
  $\hat{f}^+(\hat{s}(\hat{x}), \hat{k}_1, ..., \hat{k}_n, k - (\hat{i} + 1) \to \hat{f}^*(\hat{k}_1, ..., \hat{k}_n, \hat{s}(k - (\hat{i} + 1))) =$
  $\hat{f}^*(\hat{k}_1, ..., \hat{k}_n, \widehat{k - i})) \xrightarrow{*}_{E_{\hat{g}}} \hat{k}$
  For appropiate x and Induction hypothesis.
- $E_{\hat{f}}$ is confluent and according to Theorem 10.4, $(\hat{f}, E_{\hat{f}})$ implements the total function f.
- $E_{\hat{f}}$ is not terminating.$g(k,m) = \delta_{k,m} \rightsquigarrow \hat{f}^*(\hat{k}, k \hat{+} 1)$ leads to NT-chain. Termination is achievable!.

Representation of partial recursive functions

Problem: Recursion equations (Kleene's normal form) cannot be directly used. Arguments must have "number" as value. (See example). Some arguments can be saved:

Example 10.9.
$f(x,y) = g(h_1(x,y), h_2(x,y), h_3(x,y))$. Let g, h_1, h_2, h_3 be implementable by sets of equations as partial functions.

Claim: f is implementable. Let $\hat{f}, \hat{f}_1, \hat{f}_2$ be new and set:

$\hat{f}(x,y) =$
$\hat{f}_1(\hat{h}_1(x,y), \hat{h}_2(x,y), \hat{h}_3(x,y), \hat{f}_2(\hat{h}_1(x,y)), \hat{f}_2(\hat{h}_2(x,y)), \hat{f}_2(\hat{h}_3(x,y)))$

$\hat{f}_1(x_1, x_2, x_3, \hat{0}, \hat{0}, \hat{0}) = \hat{g}(x_1, x_2, x_3), \quad \hat{f}_2(\hat{0}) = \hat{0}, \quad \hat{f}_2(\hat{s}(x)) = \hat{f}_2(x)$

$(\hat{f}, E_{\hat{g}}, E_{\hat{h}_1}, E_{\hat{h}_2}, E_{\hat{h}_3} \cup \ REST)$ implements f.

Theorem 10.4 cannot be applied!!.

$(\hat{f}, E_{\hat{g}}, E_{\hat{h}_1}, E_{\hat{h}_2}, E_{\hat{h}_3} \cup \ REST)$ implements f.

Apply definition 10.1:
$\curvearrowright$ For number-terms let $f(\mathfrak{I}(t_1), \mathfrak{I}(t_2)) = \mathfrak{I}(t)$. There are number-terms T_i $(i = 1,2,3)$ with
$g(\mathfrak{I}(T_1), \mathfrak{I}(T_2), \mathfrak{I}(T_3)) = \mathfrak{I}(t)$ and $h_i(\mathfrak{I}(t_1), \mathfrak{I}(t_2)) = \mathfrak{I}(T_i)$.
Assumption: $\hat{g}(T_1, T_2, T_3) =_{E_{\hat{f}}} t$ and $\hat{h}_i(t_1, t_2) =_{E_{\hat{f}}} T_i (i = 1,2,3)$. The T_i are number-terms:: $\hat{f}_2(T_i) =_{E_{\hat{f}}} \hat{0}$ i.e. $\hat{f}_2(\hat{h}_i(t_1, t_2)) =_{E_{\hat{f}}} \hat{0}$ $(i = 1,2,3)$.
Hence
$\hat{f}(t_1, t_2) =_{E_{\hat{f}}} \hat{f}_1(T_1, T_2, T_3, \hat{0}, \hat{0}, \hat{0}) \rightsquigarrow \hat{f}(t_1, t_2) =_{E_{\hat{f}}} t (=_{E_{\hat{f}}} \hat{g}(T_1, T_2, T_3))$
$\curvearrowleft$ For number-terms t_1, t_2, t let $\hat{f}(t_1, t_2) =_{E_{\hat{f}}} t$, so
$\hat{f}_1(\hat{h}_1(t_1, t_2), \hat{h}_2(t_1, t_2), \hat{h}_3(t_1, t_2), \hat{f}_2(\hat{h}_1(t_1, t_2), ...) =_{E_{\hat{f}}} t$. If for an $i = 1,2,3$ $\quad \hat{f}_2(\hat{h}_i(t_1, t_2))$ would not be $E_{\hat{f}}$ equal to $\hat{0}$, then the $E_{\hat{f}}$ equivalence class contains only $\hat{f}_1$ terms. So there are number-terms T_1, T_2, T_3 with $\hat{h}_i(t_1, t_2) =_{E_{\hat{f}}} = T_i$ $(i = 1,2,3)$ (Otherwise only $\hat{f}_2$ terms equivalent to $\hat{f}_2(\hat{h}_i(t_1, t_2)$). From Assumption:
$\rightsquigarrow h_i(\mathfrak{I}(T_1), \mathfrak{I}(T_2)) = \mathfrak{I}(T_i), \qquad g(\mathfrak{I}(T_1), \mathfrak{I}(T_2), \mathfrak{I}(T_3)) = \mathfrak{I}(t)$

$\mathfrak{R}_p$ and normalized register machines

Definition 10.10. *Program terms for RM: P_n $(n \in \mathbb{N})$ Let $0 \leq i \leq n$*
Function symbols: a_i, s_i constants , $\circ$ binary , W^i unary
Intended interpretation:
a_i :: Increase in one the value of the contents on register i.
s_i :: Decrease in one the value of the contents on register i.$(\dot{-}1)$
$\circ(M_1, M_2)$ :: Concatenation $M_1 M_2$ (First M_1, then M_2)
$W^i(M)$ :: While contents of register i not 0, execute M Abbr.: $(M)_i$

Note: $P_n \subseteq P_m$ for $n \leq m$

Semantics through partial functions: $M_e : P_n \times \mathbb{N}^n \to \mathbb{N}^n$

- $M_e(a_i, \langle x_1, ..., x_n \rangle) = \langle ...x_{i-1}, x_i + 1, x_{i+1}... \rangle$ $(s_i :: x_i \dot{-} 1)$
- $M_e(M_1 M_2, \langle x_1, ..., x_n \rangle) = M_e(M_2, M_e(M_1, \langle x_1, ..., x_n \rangle))$
- $M_e((M)_i, \langle x_1, ..., x_n \rangle) = \begin{cases} \langle x_1, ..., x_n \rangle & x_i = 0 \\ M_e((M)_i, M_e(M, \langle x_1, ..., x_n \rangle)) & \text{otherwise} \end{cases}$

Implementation of normalized register machines

Lemma 10.11. *M_e can be implemented by a system of equations.*

Proof: Let tup_n be n-ary function symbol. For $t_i \in \mathbb{N}$ $(0 < i \leq n)$ let $\langle t_1, ..., t_n \rangle$ be the interpretation for $tup_n(\hat{t}_1, ..., \hat{t}_n)$. Program terms are interpreted by themselves (since they are terms). For $m \geq n$ ::

$$\begin{array}{lll} P_n & tup_m(\hat{t}_1, ..., \hat{t}_m) & \text{syntactical level} \\ \mathfrak{I} \downarrow & \mathfrak{I} \downarrow & \\ P_n & \langle t_1, ..., t_m \rangle & \text{Interpretation} \end{array}$$

Let $eval$ be a binary function symbol for the implementation of M_e and $i \leq n$. Define $E_n := \{$
$eval(a_i, \ tup_n(x_1, ..., x_n)) \to \ tup_n(x_1, ..., x_{i-1}, \hat{s}(x_i), x_{i+1}, ..., x_n)$
$eval(s_i, \ tup_n(..., x_{i-1}, \hat{0}, x_{i+1}...)) \to \ tup_n(..., x_{i-1}, \hat{0}, x_{i+1}...)$
$eval(s_i, \ tup_n(..., x_{i-1}, \hat{s}(x), x_{i+1}...)) \to \ tup_n(..., x_{i-1}, x, x_{i+1}...)$
$eval(x_1 x_2, t) \to \ eval(x_2, \ eval(x_1, t))$
$eval((x)_i, \ tup_n(..., x_{i-1}, \hat{0}, x_{i+1}...)) \to \ tup_n(..., x_{i-1}, \hat{0}, x_{i+1}...)$
$eval((x)_i, \ tup_n(..., x_{i-1}, \hat{s}(y), x_{i+1}...) \to$
$\qquad eval((x)_i, eval(x, \ tup_n(..., x_{i-1}, \hat{s}(y), x_{i+1}...))\}$

$(eval, E_n, \mathfrak{I})$ implements M_e

Consider program terms that contain at most registers with $1 \le i \le n$.

- E_n is confluent (left-linear, without critical pairs).
- Theorem 10.4 not applicable, since M_e is not total.
  Prove conditions of the Definition 10.1.

(1) $\mathfrak{I}(T_i) = M_i$ according to the definition.
(2) $M_e(p, \langle k_1, ..., k_n\rangle) = \langle m_1, ..., m_n\rangle$ iff
$eval(p,\ tup_n(\hat{k}_1, ..., \hat{k}_n)) =_{E_n} tup_n(\hat{m}_1, ..., \hat{m}_n)$
$\curvearrowright$ out of the def. of M_e res. E_n. induction on construction of p.
$\curvearrowleft$ Structural induction on p ::
1. $p = a_i(s_i)$ :: $\hat{k}_j = \hat{m}_j (j \neq i), \hat{s}(\hat{k}_i) = \hat{m}_i$ res. $\hat{k}_i = \hat{m}_i = \hat{0}$
   $(\hat{k}_i = \hat{s}(\hat{m}_i))$ for s_i
2.Let $p = p_1 p_2$ and
$eval(p_2, eval(p_1,\ tup_n(\hat{k}_1, ..., \hat{k}_n))) \xrightarrow{*}_{E_n} tup_n(\hat{m}_1, ..., \hat{m}_n)$
Because of the rules in E_n it holds:

$(eval, E_n, \mathfrak{I})$ implements M_e

There are $i_1, ..., i_n \in \mathbb{N}$ with $eval(p_1, tup_n(\hat{k}_1, ..., \hat{k}_n)) \xrightarrow{*}_{E_n} tup_n(\hat{i}_1, ..., \hat{i}_n)$
hence
$eval(p_2, tup_n(\hat{i}_1, ..., \hat{i}_n)) \xrightarrow{*}_{E_n} tup_n(\hat{m}_1, ..., \hat{m}_n)$
According to the induction hypothesis (2-times) the statement holds.
3. Let $p = (p_1)_i$. Then:
$eval((p_1)_i, tup_n(\hat{k}_1, ..., \hat{k}_n)) \xrightarrow{*}_{E_n} tup_n(\hat{m}_1, ..., \hat{m}_n)$
There exists a finite sequence $(t_j)_{1 \le j \le l}$ with
$t_1 = eval((p_1)_i, tup_n(\hat{k}_1, ..., \hat{k}_n)),\quad t_j \to t_{j+1},\quad t_l = tup_n(\hat{m}_1, ..., \hat{m}_n)$
There exists subsequence $(T_j)_{1 \le j \le m}$ of form $eval((p_1)_i, tup_n(\hat{i}_{1,j}, ..., \hat{i}_{n,j}))$
For T_m $i_{i,m} = 0$ holds, i.e. $i_{1,m} = m_1, ..., i_{i,m} = 0 = m_i, ..., i_{n,m} = m_n$.
For $j < m$ always $i_{i,j} \neq 0$ holds and
$eval(p_1, tup_n(\hat{i}_{1,j}, ..., \hat{i}_{n,j}) \xrightarrow{*}_{E_n} tup_n(\hat{i}_{1,j+1}, ..., \hat{i}_{n,j+1})$.
The induction hypothesis gives:
$M_e(p_1, \langle i_{1,j}, ..., i_{n,j}\rangle) = \langle i_{1,j+1}, ..., i_{n,j+1}\rangle$ for $j = 1, ..., m$.
But then $M_e((p_1)_i, \langle i_{1,j}, ..., i_{n,j}\rangle) = \langle m_1, ..., m_n\rangle \quad (1 \le j < m)$

Implementation of $\mathfrak{R}_p$

For $f \in \mathfrak{R}_p^{n,1}$ there are $r \in \mathbb{N}$, program term p with at most r-registers $(n+1 \le r)$, so that for every $k_1, ..., k_n, k \in \mathbb{N}$ holds:
$f(k_1, ..., k_n) = k$ iff $\forall m \ge 0$

$$eval(p, tup_{r+m}(\hat{k}_1, ..., \hat{k}_n, \hat{0}, \hat{0}, ..., \hat{0}, \hat{x}_1, ..., \hat{x}_m)) =_{E_{r+m}}$$
$$tup_{r+m}(\hat{k}_1, ..., \hat{k}_n, \hat{k}, \hat{0}, ..., \hat{0}, \hat{x}_1, ..., \hat{x}_m) \qquad \text{iff}$$

$$eval(p, tup_r(\hat{k}_1, ..., \hat{k}_n, \hat{0}, \hat{0}, ..., \hat{0})) =_{E_r} tup_r(\hat{k}_1, ..., \hat{k}_n, \hat{k}, \hat{0}, ..., \hat{0})$$

Note: $E_r \sqsubset E_{r+m}$ via $tup_r(...) \blacktriangleright tup_{r+m}(..., \hat{0}, ..., \hat{0})$.

Let $\hat{f}, \hat{R}$ be new function symbols, p program for f. Extend E_r by
$\hat{f}(y_1, ..., y_n) \to \hat{R}(eval(p, tup_r(y_1, ..., y_n), \hat{0}, ..., \hat{0}))$ and
$\hat{R}(tup_r(y_1, ..., y_r)) = y_{n+1}$ to $E_{ext(f)}$.

Theorem 10.12. *$f \in \mathfrak{R}_p^{n,1}$ is implemented by $(\hat{f}, E_{ext(f)}, \mathfrak{I})$.*

Non computable functions

Let E be recursive, T_i recursive. Then the predicate

$$P(t_1, ..., t_n, t_{n+1}) \text{ iff } \hat{f}(t_1, ..., t_n) =_E t_{n+1}$$

is a r.a. predicate on $T_1 \times ... \times T_n \times T_{n+1}$
If the function $\hat{f}$ implements f, then P represents the graph of the function $f \rightsquigarrow\ f \in \mathfrak{R}_p$.
Kleene's normal form theorem:
$f(x_1, ..., x_n) = U(\mu_y[T_n(p, x_1, ..., x_n, y) = 0])$
Let h be the total non recursive function, defined by:

$$h(x) = \begin{cases} \mu_y[T_1(x, x, y) = 0] & \text{in case that } \exists y : T_1(x, x, y) = 0 \\ 0 & \text{otherwise} \end{cases}$$

h is uniquely defined through the following predicate:
(1) $(T_1(x, x, y) = 0 \wedge \forall z(z < y \rightsquigarrow T_1(x, x, z) \neq 0)) \rightsquigarrow h(x) = y$
(2) $(\forall z(z < y \wedge T_1(x, x, z) \neq 0)) \rightsquigarrow (h(x) = 0 \vee h(x) \ge y)$
If $h(x)$ is replaced by u, then these are prim. rec. predicates in x, y, u.

Non computable functions

There are primitive recursive functions P_1, P_2 in x, y, u, so that

$$(1') \quad P_1(x, y, h(x)) = 0 \text{ and } (2') \quad P_2(x, y, h(x)) = 0$$

represent (1) and (2).
Hence there are an equational system E and function symbols $\hat{P}_1, \hat{P}_2$, that implement P_1, P_2 under the standard interpretation.
(As prim. rec. functions in the Var. x, y, u)
Let $\hat{h}$ be fresh. Add to E the equations

$$\hat{P}_1(x, y, \hat{h}(x)) = \hat{0} \text{ and } \hat{P}_2(x, y, \hat{h}(x)) = \hat{0}.$$

The equational system is consistent (there are models) and $\hat{h}$ is interpreted by the function h on the natural numbers. $\leadsto$
It is possible to specify non recursive functions implicitly with a finite set of equations, in case arbitrary models are accepted as interpretations.
Through non recursive sets of equations any function can be implemented by a confluent, terminating ground system :
$E = \{\hat{h}(\hat{t}) = \hat{t}' : t, t' \in \mathbb{N}, h(t) = t'\}$ (Rule application is not effective).

Computable algebras

Definition 10.13.
- *A sig-Algebra $\mathfrak{A}$ is recursive (effective, computable), if the base sets are recursive and all operations are recursive functions.*
- *A specification spec $= (sig, E)$ is recursive, if T_{spec} is recursive.*

Example 10.14. *Let $sig = (\{nat, even\}, odd :\to even, 0 :\to nat, s : nat \to nat, red : nat \to even)$.*
As sig-Algebra $\mathfrak{A}$ choose: $A_{even} = \{2n : n \in \mathbb{N}\} \cup \{1\}, A_{nat} = \mathbb{N}$ with odd as 1, red as $\lambda x.if\ x\ even\ then\ x\ else\ 1$, s successor
Claim: There is no finite (init-Algebra) specification for $\mathfrak{A}$

- *No equations of the sort nat.*
- *$odd, red(s^n(0)), red(s^n(x))$ $(n \geq 0)$ terms of sort even. No equations of the form $red(s^n(x)) = red(s^m(x)$ $(n \neq m)$ are possible.*
- *Infinite number of ground equations are needed.*

Computable algebras

Solution: Enrichment of the signature with:
$even : nat \to nat$ and $cond : nat\ even\ even \to even$ with interpretation

$$\lambda x.\ if\ x\ even\ then\ 0\ else\ 1, \qquad \lambda x, y, z.\ if\ x = 0\ then\ y\ else\ z$$

Equations:
$even(0) = 0, \quad even(s(0)) = s(0), \quad even(s(s(x)) = even(x)$
$cond(0, y, z) = y, \quad cond(s(x), y, z) = z$
$red(x) = cond(even(x), red(x), odd)$

Alternative: Conditional equations:
$red(s(0)) = odd, \ red(s(s(x)) = odd\ if\ red(x) = odd$

Conditional equational systems (term replacement systems) are more "expressive" as pure equational systems. They also define reduction relations. Confluence and termination criteria can be derived. Negated equations in the conditions lead to problems with the initial semantics (non Horn-clause specifications).

Computable algebras: Results

Theorem 10.15. *Let $\mathfrak{A}$ be a recursive term generated sig- Algebra. Then there is a finite enrichment sig' of sig and a finite specification $spec' = (sig', E)$ with $T_{spec'}|_{sig} \cong \mathfrak{A}$.*

Theorem 10.16. *Let $\mathfrak{A}$ be a term generated sig- Algebra. Then there are equivalent:*

- *$\mathfrak{A}$ is recursive.*
- *There is a finite enrichment (without new sorts) sig' of sig and a finite convergent rule system R, so that $\mathfrak{A} \cong T_{spec'}|_{sig}$ for $spec' = (sig', R)$*

See Bergstra, Tucker: Characterization of Computable Data Types (Math. Center Amsterdam 79).

Attention: Does not hold for signatures with only unary function symbols.

Reduction strategies for replacement systems

Main implementation problems for functional programming languages.
Which reduction strategies guarantee the calculation of normal forms, in case these exist. Let R be TES, $t \in term(\Sigma)$.
Assuming that there is $\bar{t}$ irreducible with $t \xrightarrow{*}_R \bar{t}$.

- Which choice of the redexes guarantees a "computation" of $\bar{t}$?
- Which choice of the redexes delivers the "shortest" derivation sequence?
- Let R be terminating. Is there a reduction strategy that delivers always the shortest derivation sequence? How much does it cost?

For $SKI-$calculus and $\lambda-$calculus the Left-Most-Outermost strategy (normal strategy) is normalizing, i.e. calculates a normal form of a term if it exists. It doesn't deliver the shortest derivation sequences. Though it holds: If $t \xrightarrow{k} \bar{t}$ is a shortest derivation sequence, then $t \xrightarrow{\leq 2^k}_{LMOM} \bar{t}$. By using structure-sharing-methods, the bounds for LMOM can be lowered.

Functional computability models

- Partial recursive functions (Basic functions + Operators)
- Term rewriting systems (Algebraic Specification)
- λ-Calculus and Combinator Calculus
- Graph replacement Systems (Implementation + efficiency)

Central Notion: Application:

Expressions represent (denote) functions.
Application of functions on functions $\rightsquigarrow$ Self application problem

See e.g. Barendregt: Functional Programming and λ-Calculus Handbook of Theoretical Computer Science.

λ-Calculus und Combinator Calculus: Informal

Basic operations:

- Application:: For "expressions" F, A:: $F.A$ or (FA)
  F as program term is "applied" on A as argument term.
- Abstraction:: For an "expression" M, Variable x :: $\lambda x.M$
  Denotes a function which maps x into M, M can "depend"on x .
- Example: $(\lambda x.2 * x + 1).3$ should give as result $2 * 3 + 1$, hence 7.
- β-Equation:: $(\lambda x.M[x])N = M[x := N]$
  "Free" occurrences of x in M are "replaced" by N. β-Conversion
  $$(yx(\lambda x.x))[x := N] \equiv (yN(\lambda x.x))$$
  Notice: Free occurrences of variables in N remain free.
  Renaming of (bound) variables if necessary
  $$(\lambda x.y)[y := xx] \equiv \lambda z.xx \ z \text{ "new"}$$

λ-Calculus und Combinator Calculus: Informal

- α-Equation:: $\lambda x.M = \lambda y.M[x := y]$ with y "new"
  $\lambda x.x = \lambda y.y$. Same effect as "Functions" α-Conversion
- Set of λ- terms in C and V::
  $$\Lambda(C, V) = C|V|(\Lambda\Lambda)|(\lambda V.\Lambda)$$
- Set of free variables of M:: $FV(M)$
- M is closed (Combinator) if $FV(M) = \emptyset$
- Standard Combinators:: $I \equiv \lambda x.x$ $K \equiv \lambda xy.x \equiv \lambda x.(\lambda y.x)$
  $B \equiv \lambda xyz.x(yz)$ $K_* \equiv \lambda xy.y$ $S \equiv \lambda xyz.xz(yz)$
- Following equalities hold:
  $IM = M$ $KMN = M$ $K_*MN = N$ $SMNL = ML(NL)$
  $BLMN = L(M(N))$ left parenthesis !
- Fixpoint Theorem:: $\forall F \exists X$ $FX = X$ with e.g.
  $X \equiv WW$ and $W \equiv \lambda x.F(xx)$

λ-Calculus und Combinator Calculus: Informal

- Representation of functions, numbers $c_n \equiv \lambda fx.f^n(x)$
  F combinator represents f iff $Fz_{n1}...z_{nk} = z_{f(n1,...,nk)}$
- f is partial recursive iff f is represented by a combinator.
- Theorem of Scott: Let $A \subset \Lambda, A$ non trivial and closed under $=$, then A not recursively decidable.
- β-Reduction:: $(\lambda x.M)N \rightarrow_\beta M[x := N]$
- NF = Set of terms which have a normal form is not recursive.
- $(\lambda x.xx)y$ is not in normal form, yy is in normal form.
- $(\lambda x.xx)(\lambda x.xx)$ has no normal form.
- Church Rosser Theorem:: $\rightarrow_\beta$ ist confluent
- Theorem of Curry If M has a normal form then $M \rightarrow_l^* N$, i.e. Leftmost Reduction is normalizing.

Reduction strategies for replacement systems

Definition 11.1. *Let R be a TES.*

- *A one-step reduction strategy $\mathfrak{S}$ for R is a mapping $\mathfrak{S} : term(R, V) \rightarrow term(R, V)$ with $t = \mathfrak{S}(t)$ in case that t is in normal form and $t \rightarrow_R \mathfrak{S}(t)$ otherwise.*
- *$\mathfrak{S}$ is a multiple-step-reduction strategy for R if $t = \mathfrak{S}(t)$ in case that t is in normal form and $t \xrightarrow{+}_R \mathfrak{S}(t)$ otherwise.*
- *A reduction strategy $\mathfrak{S}$ is called normalizing for R, if for each term t with a R- normal form, the sequence $(\mathfrak{S}^n(t))_{n\geq 0}$ contains a normal form. (Contains in particular a finite number of terms).*
- *A reduction strategy $\mathfrak{S}$ is called cofinal for R, if for each t and $r \in \Delta^*(t)$ there is a $n \in \mathbb{N}$ with $r \xrightarrow{*}_R \mathfrak{S}^n(t)$.*

Cofinal reduction strategies are optimal in the following sense: they deliver maximal information gain.
Assuming that normal forms contain always maximal information.

Known reduction strategies

Definition 11.2. *Reduction strategies:*

- *Leftmost-Innermost (Call-by-Value). One-step-RS, the redex that appears most left in the term and that contains no proper redex is reduced.*
- *Paralell-Innermost. Multiple-step-RS. $PI(t) = \bar{t}$, at which $t \mapsto \bar{t}$ (All the innermost redexes are reduced).*
- *Leftmost-Outermost (Call-by-Name). One-step-RS.*
- *Parallel-Outermost. Multiple-step-RS. $PO(t) = \bar{t}$, at which $t \mapsto \bar{t}$ (All the disjoint outermost redexes are reduced).*
- *Fair-LMOM. A left-most outermost redex in a red-sequence is eventually reduced. (A LMOR in such a strategy doesn't remain unreduced for ever). (Lazy strategy).*

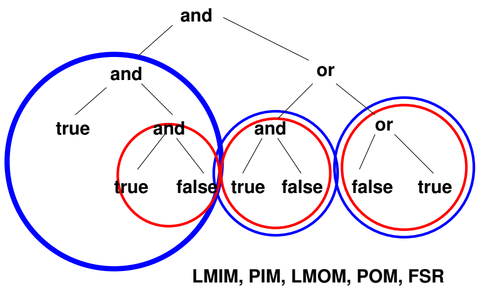
Known reduction strategies

- Full-substitution-rule. (Only for orthogonal systems). Multiple-step-RS. $GK(t) :: t \xrightarrow{+} GK(t)$ all the redexes in t are reduced, in case they're not disjunct, then the residuals of the redexes are also reduced.
- Call-By-Need. One-step-RS. It reduces always a necessary redex. A redex in t is necessary, when it must be reduced in order to compute the normal form. (Only for certain TES e.g. LMOM for SKI calculus) Problem: How can one decide whether a redex is necessary or not?
- Variable-Delay-Strategy: One-step-RS. Reduce redex, that doesn't appear as redex in the instance of a variable of another redex.

Examples

Example 11.3. *:*

- $and(true, x) \to x,\ \ and(false, x) \to\ false,$
  $or(true, x) \to\ true,\ \ or(false, x) \to x$
  Orthogonal, strong left sequential (constants "before" the variables).



LMIM, PIM, LMOM, POM, FSR

Examples

- $\Sigma = \{0, s, p, if0, F\}, R = \{p(0) \to 0, p(s(x)) \to x, if0(0, x, y) \to x, if0(s(z), x, y) \to y, F(x, y) \to if0(x, 0, F(p(x), F(x, y)))\}$
  Left-linear, without overlaps. (orthogonal).
  $F(0, 0) \to if0(0, 0, F(p(0), F(0, 0))) \stackrel{OM}{\to} 0$
  $\downarrow PIM$
  $if0(0, 0, F(0, if0(0, 0, F(p(0), F(0, 0)))))$
  No IM-strategy is for all orthogonal systems normalizing or cofinal.
- FSR (Full-Substitution-Rule): Choose all the redexes in the term and reduce them from innermost to outermost (notice no redex is destroyed). Cofinal for orthogonal systems.
- $\Sigma = \{a, b, c, d_i : i \in \mathbb{N}\}$
  $R := \{a \to b, d_k(x) \to d_{k+1}(x), c(d_k(b)) \to b$
  confluent (left linear parallel 0-closed).
  $c(d_0(a)) \to_1 c(d_1(a)) \to_1 ....$ not normalizing (POM).
  $c(d_0(a)) \to_{1,1} c(d_0(b)) \to_0 b$

Examples

- $\Sigma = \{a, b_i, c, d : i \in \mathbb{N}\}$. Non confluent SRS:
  $R = \{ab_0c \to acb_0, ab_0d \to ad, c \to d, cb_i \to d, b_i \to b_{i+1} (i \geq 1)\}$
  $ab_0c \to_{11} ab_0d \to ad$
  $ab_0c \to_0 acb_0 \to_{11} acb_1 \to adb_1 \to ...$
- $\Sigma = \{f, a, b, c, d\}$ $R = \{f(x, b) \to d, a \to b, c \to c\}$ Orthogonal.
  LMOM must not be normalizing:
  $f(c, a) \to f(c, a) \to ....$ but $f(c, a) \to f(c, b) \to d$
- $f(a, f(x, y)) \to f(x, f(x, f(b, b)))$ left linear with overlaps.
  $f(a, f(a, f(b, b))) \to_{OUT} f(a, f(a, f(b, b))) \to_{OUT} ....$
  $\downarrow INN$
  $f(a, f(b, f(b, f(b, b)))) \to f(b, f(b, f(b, b)))$
- $R = \{f(g(x), c) \to h(x, d), b \to c\}$
  $f(g(f(a, f(a, \underline{b}))), c) \to_{VD} h(f(a, f(a, \underline{b})), d) \to_{VD}$
  $h(f(a, f(a, c)), d)$

Strategies for orthogonal systems

Theorem 11.4. *For orthogonal systems the following holds:*

- *Full-Substitution-Rule is a cofinal reduction strategy.*
- *POM is a normalizing reduction strategy.*
- *LMOM is normalizing for λ-calculus and CL-calculus.*
- *Every fair-outermost strategy is normalizing.*
- ***Main tools****:*
  Elementary reduction diagrams,residuals and *reduction diagrams*

$$\begin{array}{ccc}
Sab(Ic) & \to & a(Ic)(b(Ic)) \\
\downarrow & & \downarrow \\
 & & ac(b(Ic)) \\
 & & \downarrow \\
Sabc & \to & ac(bc)
\end{array}
\qquad
\begin{array}{ccc}
Ka(Ib) & \to & Kab \\
\downarrow & & \downarrow \\
a & \to_{\emptyset} & a
\end{array}$$

$$\begin{array}{ccc}
Ia & \to & a \\
\downarrow_{\emptyset} & & \downarrow_{\emptyset} \\
Ia & \to & a
\end{array}
\qquad
\begin{array}{ccc}
Ia & \to & a \\
\downarrow & & \downarrow_{\emptyset} \\
a & \to_{\emptyset} & a
\end{array}
\qquad
\begin{array}{ccc}
a & \to & a \\
 & & \\
a & & a
\end{array}$$

Composition of E-reduction diagrams

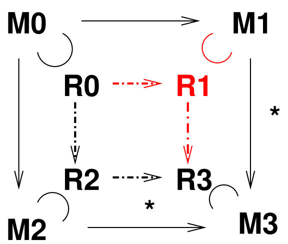
Reduction diagrams and projections:

$$
\begin{array}{cccccccc}
t_0 & \to & t_1 & \to & & \dots & \to & t_n \\
\downarrow & & \downarrow * & \xrightarrow{*} & & \xrightarrow{*} & & \downarrow * \\
t_1' & \xrightarrow{*} & & \xrightarrow{*} & & \xrightarrow{*} & & \downarrow * \\
\downarrow & & \downarrow * & \xrightarrow{*} & & \xrightarrow{*} & & \downarrow * \quad R_4 = R_2 / R_1 \\
\downarrow & & \downarrow * & \xrightarrow{*} & & \xrightarrow{*} & & \downarrow * \\
\vdots & \vdots & \dots & \dots & & & & \downarrow * \\
\downarrow & & \downarrow * & \xrightarrow{*} & & \xrightarrow{*} & & \downarrow * \\
t_m' & \xrightarrow{*} & & \xrightarrow{*} & & & \xrightarrow{*} & \downarrow * \\
 & & & & R_3 = R_1 / R_2 & & & \text{projections}
\end{array}
$$

Let $R_1 :: t \xrightarrow{+} t'$ and $R_2 :: t \xrightarrow{+} t'$ be two reduction sequences of r from t to t'. They are equivalent $R_1 \cong R_2$ iff $R_1 / R_2 = R_2 / R_1 = \emptyset$.

Strategies for orthogonal systems

Lemma 11.5. *Let D be an elementary reduction diagram for orthogonal systems, $R_i \subseteq M_i$ $(i = 0, 2, 3)$ redexes with $R_0 -.-.\to R_2 -.-.\xrightarrow{*} R_3$ i.e. R_2 is residual of R_0 and R_3 is residual of R_2. Then there is a unique redex $R_1 \subseteq M_1$ with $R_0 -.-.\to R_1 -.-.\xrightarrow{*} R_3$, i.e.*



Notice, *that in the reduction sequences $M_1 \xrightarrow{*} M_3$ and $M_2 \xrightarrow{*} M_3$ only residuals of the corresponding used redex in the reduction in M_0 are reduced.*

Property of elementary reduction diagrams!

Strategies for orthogonal systems

Definition 11.6. *Let Π be a predicate over term pairs M, R so that $R \subseteq M$ and R is redex (e.g. LMOM, LMIM,...).*

i) Π has property I when for a D like in the lemma it holds:

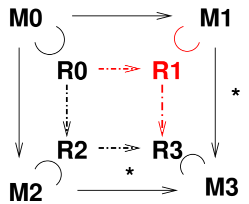
$$\Pi(M_0, R_0) \wedge \Pi(M_2, R_2) \wedge \Pi(M_3, R_3) \rightsquigarrow \Pi(M_1, R_1)$$

ii) Π has property II if in each reduction step $M \to^R M'$ with $\neg\Pi(M, R)$, each redex $S' \subseteq M'$ with $\Pi(M', S')$ has an ancestor-redex $S \subseteq M$ with $\Pi(M, S)$. (i.e. $\neg\Pi$ steps introduce no new Π-redexes).

Lemma 11.7. *Separability of developments. Assume Π has property II. Then each development $\mathfrak{R} :: M_0 \to \dots \to M_n$ can be partitioned in a Π-part followed by a $\neg\Pi$-part.*
More precisely: There are reduction sequences
$\mathfrak{R}_\Pi :: M_0 = N_0 \to^{R_0} \dots \to^{R_{k-1}} N_k$ with $\Pi(N_i, R_i)$ $(i < k)$ and
$\mathfrak{R}_{\neg\Pi} :: N_k \to^{R_k} \dots \to^{R_{k+l-1}} N_{k+l}$ with $\neg\Pi(N_j, R_j)$ $(k \le j < k + l)$ and $\mathfrak{R}$ is equivalent to $\mathfrak{R}_\Pi \times \mathfrak{R}_{\neg\Pi}$.

Example 11.8.

- *$\Pi(M, R)$ iff R is redex in M. I and II hold.*
- *$\Pi(M, R)$ iff R is an outermost redex in M. Then properties I and II hold: To I*



R_0, R_2, R_3 outermost redexes
Let S_i be the redex in $M_0 \to M_i$
Assuming that is not OM $\rightsquigarrow$ In M_1 a redex (P) is generated by the reduction of S_1, that contains R_1.

In $M_1 \to\!> M_3$ R_1 becomes again outermost. i.e. P is reduced: But in $M_1 \to\!> M_3$ only residuals of S_2 are reduced and P is not residual, since was newly introduced.$\lightning$. II is clear.

- *$\Pi(M, R)$ iff R is left-most redex in M. I holds. II not always:*
$F(x, b) \to d, a \to b, c \to c :: F(c, a) \to F(c, b)$

Descendants of redexes (residuals)

Definition 11.9. *Traces in reduction sequences:*

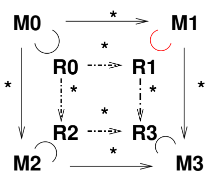
- *Let $\mathfrak{R} :: M_0 \to M_1 \to \dots$ be a reduction sequence. Let M_j be fixed and $L_i \subseteq M_i$ $(i \geq j)$ (provided that M_i exists) redexes with $L_j - . - . \to L_{j+1} - . - . \to \dots$. The sequence $\mathfrak{L} = (L_{j+i})_{i \geq 0}$ is a trace of descendants (residuals) of redexes in M_j.*
- *$\mathfrak{L}$ is called Π-trace, in case that $\forall i \geq j \;\; \Pi(M_i, L_i)$.*
- *Let R be a reduction sequence, Π a predicate. R is Π-fair, if R has no infinite Π-Traces.*

Results from Bergstra, Klop :: Conditional Rewrite Rules: Confluence and Termination. JCSS 32 (1986)

Properties of Traces

Lemma 11.10. *Let Π be a predicate with property I.*

- *Let $\mathfrak{D}$ be a reduction diagram with $R_i \subseteq M_i, R_0 - . - . \to> R_2 - . - . \to> R_3$ is Π trace.*



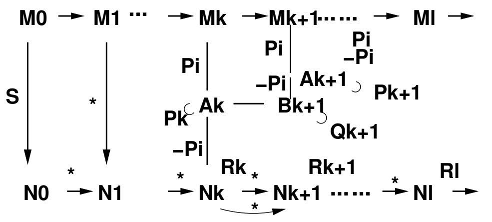
Then $R_0 - . - . \to> R_1 - . - . \to> R_3$ via M_1 also a Π trace

- *Let $\mathfrak{R}, \mathfrak{R}'$ be equivalent reduction sequences from M_0 to M. $S \subseteq M_0, S' \subseteq M$ redexes, so that a Π-trace $S - . - . \to> S'$ via $\mathfrak{R}$ exists. Then there is a unique Π-trace $S - . - . \to> S'$ via $\mathfrak{R}'$.*

Main Theorem of O'Donnell 77

Theorem 11.11. *Let Π be a predicate with properties I,II. Then the class of Π-fair reduction sequences is closed w.r. to projections.*

Proof Idea:



Let $\mathfrak{R} :: M_0 \to \dots$ be Π-fair and $\mathfrak{R}' :: N_0 \xrightarrow{*}$ a projection.
$\forall k \exists M_k \overset{\Pi}{\to}> A_k \overset{\neg\Pi}{\to}> N_k$ equivalent to the complete development $M_k \to> N_k$. In the resulting rearrangement both derivations between N_k and N_{k+1} are equivalent. In particular the Π-Traces remain the same.
Results in an echelon form: $A_k - B_{k+1} - A_{k+1} - B_{k+2} - \dots$

Main Theorem: Proof

This echelon reaches $\mathfrak{R}$ after a finite number of steps, let's say in M_l::
If not $\mathfrak{R}$ would have an infinite trace of S residuals with property Π.

Let's assume that $\mathfrak{R}'$ is not Π fair. Hence it contains an infinite Π -trace $R_k, \dots, R_{k+1} \dots$ that starts from N_k.

There are Π-ancestors $P_k \subseteq A_k$ from the Π-redex $R_k \subseteq N_k$, i.e with $\Pi(A_k, P_k)$. Then the Π-trace $P_k - . - . \to> R_k - . - . \to> R_{k+1}$ can be lifted via B_{k+1} to the Π-trace $P_k - . - . \to> Q_{k+1} - . - . \to> R_{k+1}$.

Iterating this construction until M_l, a redex P_l that is predecessor of R_l with $\Pi(M_l, P_l)$ is obtained. This argument can be now continued with R_{l+1}.
Consequently $\mathfrak{R}$ is not Π-fair.$\lightning$.

Consequences

Lemma 11.12. *Let $\mathfrak{R} :: M_0 \to M_1 \to \ldots$ be an infinite sequence of reductions with infinitely outermost redex-reductions. Let $S \subseteq M_0$ be a redex. Then $\mathfrak{R}' = \mathfrak{R} / \{S\}$ is also infinite.*

Proof: Assume that $\mathfrak{R}'$ is finite with length k. Let $l \geq k$ and R_l be the redex in the reduction of $M_l \to M_{l+1}$ and let $\mathfrak{R}_l$ the reduction sequence from M_l to M'_l

• If R_l is outermost, then $M'_l \xrightarrow{*} M'_{l+1}$ can only be empty if R_l is one of the residuals of S which are reduced in $\mathfrak{R}_l$. Thus $\mathfrak{R}_{l+1}$ has one step less than $\mathfrak{R}_l$.

• Otherwise R_l is properly contained in the residual of S reduced in $\mathfrak{R}_l$.

However given that $\mathfrak{R}$ must contain infinitely many outermost redex-reductions then $\mathfrak{R}_q$ would become empty. Consequently $\mathfrak{R}'$ must coincide with $\mathfrak{R}$ from some position on, hence it is also infinite.

Consequences for orthogonal systems

Theorem 11.13. *Let $\Pi(M, R)$ iff R is outermost redex in M.*

- *The fair outermost reduction sequences are terminating, when they start from a term which has a normal form.*
- *Parallel-Outermost is normalizing for orthogonal systems.*

Proof: If t has a normal form, then there is no infinite Π-fair reduction sequence that starts with t.
Let $\mathfrak{R} :: t \to t_1 \to \ldots. \to$ be an infinite Π-fair and $\mathfrak{R}' :: t \to t'_1 \to \ldots \to \bar{t}$ a normal form.
$\mathfrak{R}$ contains infinitely many outermost reduction steps (otherwise it would not Π-fair). Then $\mathfrak{R} / \mathfrak{R}'$ also infinite. $\lightning$.

Observe that: The theorem doesn't hold for LMOM-strategy: property II is not fulfilled. Consider for this purpose $a \to b, c \to c, f(x, b) \to d$.

Consequences for orthogonal systems

Definition 11.14. *Let R be orthogonal, $l \to r \in R$ is called left normal, if in l all the function symbols appear left of the variables. R is left normal, if all the rules in R are left normal.*

Consequence 11.15. *Let R be left normal. Then the following holds:*

- *Fair leftmost reduction sequences are terminating for terms with a normal form.*
- *The LMOM-strategy is normalizing.*

Proof: Let $\Pi(M, L)$ iff L is LMO-redex in M. Then the properties I and II hold. For II left normal is needed.
According to theorem 11.11 the Π-fair reduction sequences are closed under projections. From Lemma 11.12 the statement follows.

Summary

A strategy is called perpetual if it can induce infinite reduction sequences.

Strategy	Orthogonal	LN-Ortogonal	Orthogonal-NE
LMIM	*p*	*p*	*p n*
PIM	*p*	*p*	*p n*
LMOM		*n*	*p n*
POM	*n*	*n*	*p n*
FSR	*n c*	*n c*	*p n c*

Classification of TES according to appearances of variables

Definition 11.16. *Let R be TES, $Var(r) \subseteq Var(l)$ for $l \to r \in R, x \in Var(l)$.*

- *R is called variable reducing, if for every $l \to r \in R, |l|_x > |r|_x$*
  R is called variable preserving, if for every $l \to r \in R, |l|_x = |r|_x$
  R is called variable augmenting, if for every $l \to r \in R, |l|_x \leq |r|_x$
- *Let $D[t, t']$ be a derivation from t to t'. Let $|D[t, t']|$ the length of the reduction sequence. $D[t, t']$ is optimal if it has the minimal length among all the derivations from t to t'.*

Lemma 11.17. *Let R be orthogonal, variable preserving. Then every redex remains in each reduction sequence, unless it is reduced. Each derivation sequence is optimal.*

Proof: Exchange technique: residuals remain as residuals, as long as they are not reduced, i.e. the reduction steps can be interchanged.

Examples

Example 11.18. *Lengths of derivations:*

- *Variable preserving:*
  $R :: f(x, y) \to g(h(x), y)), g(x, y) \to l(x, y), a \to c, b \to d$.
  Consider the term $f(a, b)$ and its derivations.
  All derivation sequences to the normal form are of the same length (4).
- *Variable augmenting (non erasing):*
  $R :: f(x, b) \to g(x, x), a \to b, c \to d$. Consider the term $f(c, a)$ and its derivations.
  Innermost derivation sequences are shorter than the outermost ones.

Further Results

Lemma 11.19. *Let R be overlap free, variable augmenting. Then an innermost redex remains until it is reduced.*

Theorem 11.20. *Let R be orthogonal variable augmenting (ne). Let $D[t, t']$ be a derivation sequence from t to its normal form t', which is non-innermost. Then there is an innermost derivation $D'[t, t']$ with $|D'| \leq |D|$.*

Proof: Let $L(D)$ = derivation length from the first non-innermost reduction in D to t'.
Induction over $L(D) :: t \to t_1 \to \dots \to t_i \xrightarrow{S} \dots \to t_j \xrightarrow{*} t'$.
Let i be this position.
S is non-innermost in t_i, hence it contains an innermost redex S_i that must be reduced later on, let's say in the reduction of t_j. Consider the reduction sequence $D' :: t \to t_1 \to \dots \to t_i \xrightarrow{S_i} t'_{i+1} \xrightarrow{S} \dots t'_j \overset{<}{\xrightarrow{*}} t'$
$|D'| \leq |D|, L(D') < L(D) \rightsquigarrow$ there is a derivation D' with $L(D') = 0$.

Further Results

Theorem 11.21. *Let R be overlap free, variable augmenting. Every two innermost derivations to a normal form are equally long.*

Sure! given that innermost redexes are disjoint and remain preserved as long as they are not reduced.
Consequence:Let R be left linear, variable augmenting. Then innermost derivations are optimal. Especially LMIM is optimal.

Example 11.22. *If there are several outermost redexes, then the length of the derivation sequences depend on the choice of the redexes.*
Consider:
$f(x, c) \to d, a \to d, b \to c$ and the derivations:
$f(\underline{a}, b) \to f(d, \underline{b}) \to \underline{f(d, c)} \to d$ and respectively $f(a, \underline{b}) \to \underline{f(a, c)} \to d$
$\rightsquigarrow$ variable delay strategy. If an outermost redex after a reduction step is no longer outermost, then it is located below a variable of a redex originated in the reduction. If this rule deletes this variable, then the redex must not be reduced.

Further Results

Theorem 11.23. *Let R be overlap free.*

- *Let D be an outermost derivation and L a non-variable outermost redex in D. Then L remains a non-variable outermost redex until it is reduced.*
- *Let R be linear. For each outermost derivation* $D[t, t']$, t' *normal form, exists a variable delaying derivation* $D'[t, t']$ *with* $|D'| \leq |D|$. *Consequently the variable delaying derivations are optimal.*

Theorem 11.24. *Ke Li. The following problem is NP-complete:*

Input: *A convergent TES R, term t and* $D[t, t\downarrow]$.
Question: *Is there a derivation* $D'[t, t\downarrow]$ *with* $|D'| < |D|$.

Proof Idea: Reduce 3-SAT to this problem.

Computable Strategies

Definition 11.25. *A reduction strategy* $\mathfrak{S}$ *is computable, if the mapping* $\mathfrak{S} : Term \rightarrow Term$ *with* $t \xrightarrow{*} \mathfrak{S}(t)$ *is recursive.*

Observe that: The strategies LMIM, PIM, LMOM, POM, FSR are polynomially computable.

Question: Is there a one-step computable normalizing strategy for orthogonal systems ?.

Example 11.26.

- *(Berry) CL-calculus extended by rules* $FABx \rightarrow C, FBxA \rightarrow C, FxAB \rightarrow C$ *is orthogonal, non-left-normal. Which argument does one choose for the reduction of FMNL? Each argument can be evaluated to A resp. B, however this is undecidable in CL.*
- *Consider* $or(true, x) \rightarrow true, or(x, true) \rightarrow true + CL$. *Parallel evaluation seems to be necessary!*

Computable Strategies: Counterexample

Example 11.27. *Signature: Constants:* $S, K, S', K', C, 0, 1$
unary: A, *activate* *binary:* ap, ap' *ternary:* B

Rules:
$ap(ap(ap(S, x), y), z) \rightarrow ap(ap(x, y), ap(y, z))$
$ap(ap(K, x), y) \rightarrow x$
$activate(S') \rightarrow S, \quad activate(K') \rightarrow K$
$activate(ap'(x, y)) \rightarrow ap(activate(x), activate(y))$
$A(x) \rightarrow B(0, x, activate(x)), \quad A(x) \rightarrow B(1, x, activate(x))$
$B(0, x, S) \rightarrow C, \quad B(1, x, K) \rightarrow C, \quad B(x, y, z) \rightarrow A(y)$

Terms: *Starting with terms of form* $A(t)$ *where t is constructed from* S', K' *and* ap'.

Claim: *R is confluent and has no computable one step strategy which is normalizing.*

A sequential Strategy for paror Systems

Example 11.28. *Let* $f, g : \mathbb{N}^+ \rightarrow \mathbb{N}$ *recursive functions. Define a "term rewriting system" R on* $\mathbb{N} \times \mathbb{N}$ *with rules:*

- $(x, y) \rightarrow (f(x), y)$ *if* $x, y > 0$
- $(x, y) \rightarrow (x, g(y))$ *if* $x, y > 0$
- $(x, 0) \rightarrow (0, 0)$ *if* $x > 0$
- $(0, y) \rightarrow (0, 0)$ *if* $y > 0$

Obviously R is confluent. Unique normal form is $(0, 0)$ *and for* $x, y > 0$,

$$(x, y) \text{ has a normal form iff } \exists n.\ f^n(x) = 0 \vee g^n(x) = 0.$$

A one step reductions strategy must choose among the application of f res. g in the first res. second argument.
Such a reduction strategy cannot compute first the zeros of $f^n(x)$ *res.* $g^n(y)$ *in order to choose the corresponding argument. One could expect, that there are appropriate functions f and g for which no computable one step strategy exists. But this is not the case!!*

A sequential strategy for paror systems

There exists a computable one step reduction strategy which is normalizing.

Lemma 11.29. *Let $(x, y) \in \mathbb{N} \times \mathbb{N}$. Then:*

- $x < y$:: *For n either $f^n(x) = 0$ or $f^n(x) \geq y$ or there exists an $i < n$ with $f^n(x) = f^i(x) \neq 0$ holds. Choose n minimal with this property. The three alternatives are mutually excluding.*
  If one of the first two holds then $\mathfrak{S}(x, y) = L$ else R
- $x \geq y$:: *For n either $g^n(y) = 0$ or $g^n(y) > x$ or there exists an $i < n$ with $g^n(y) = g^i(y) \neq 0$. Choose n minimal with this property. The three alternatives are mutually excluding. If one of the first two holds then $\mathfrak{S}(x, y) = R$ else L*
- *Claim: $\mathfrak{S}$ is a computable one step reduction strategy for R which is normalizing. (Proof: Exercise)*

Computable Strategies

Theorem 11.30. *Kennaway (Annals of Pure and Applied Logic 43(89)) For each orthogonal system there is a computable sequential (one step) normalising reduction strategy.*

Definition 11.31. *Standard reduction sequences*
Let $\mathfrak{R} :: t_0 \to t_1 \to \ldots$ be a reduction sequence in the TES R. Mark in each step in $\mathfrak{R}$ all top-symbols of redexes that appear on the left side of the reduced redex. $\mathfrak{R}$ is a standard reduction sequence if no redex with marked top-symbol is ever reduced.

Theorem 11.32.
Standardization theorem for left-normal orthogonal TES.
Let R be LNO.
If $t \xrightarrow{*} s$ holds, then there exists a standard reduction sequence in R with $t \xrightarrow{*}_{ST} s$.
Especially LMOM is normalizing.

Sequential Orthogonal TES

Example 11.33. *For applicative TES:: $PxQ \to xx, R \to S, Ix \to x$*
Consider $\mathfrak{R} :: PR(\underline{IQ}) \to \underline{PRQ} \to \underline{R}R \to SR$
There exists no standard reduction sequence from $PR(IQ)$ to SR

Fact: λ-Calculus and CL-Calculus are sequential, i.e. always needed redexes are reduced for computing the normal form.

Definition 11.34. *Let R be orthogonal, $t \in Term(R)$ with normal form $t\downarrow$. A redex $s \subseteq t$ is a* ***needed*** *redex, if in every reduction sequence $t \to \ldots \to t\downarrow$ some residual of s is reduced (contracted).*

Sequential Orthogonal TES: Call-by-need

Theorem 11.35. *Huet- Levy (1979) Let R be orthogonal*

- *Let t with a normal form but reducible , then t contains a needed redex*
- *"Call-by-need" Strategy (needed redexes are contracted) is normalizing*
- *Fair needed-redex reduction sequences are terminating for terms with a normal form.*

Lemma 11.36. *Let R be orthogonal, $t \in Term(R)$, s, s' redexes in t s.t. $s \subseteq s'$. If s is needed, then also s' is.*
In particular:: If t is not in normal form, then a outermost redex is a needed redex.

Let $C[\ldots, \ldots, \ldots]$ be a context with n-places (holes), σ a substitution of the redexes $s_1, \ldots, s_n$ in places $1, \ldots, n$. The Lemma implies the following property:
$\forall C[\ldots, \ldots, \ldots]$ in normal form, $\forall \sigma \exists i. s_i$ needed in $C[s_1, \ldots, s_n]$.
Which one of the s_i is needed, depends on σ .

Sequential Orthogonal TES

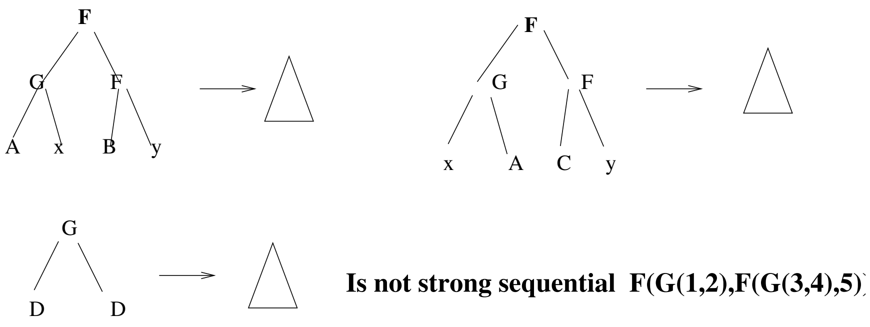
Definition 11.37. *Let R be orthogonal.*

- *R is sequential* iff* $\forall C[\ldots,\ldots,\ldots]$ *in normal form* $\exists i \forall \sigma . s_i$ *is needed in* $C[s_1,\ldots,s_n]$
  Unfortunately this property is undecidable
- *Let* $C[\ldots]$ *context. The reduction relation* $\rightarrow_?$ *(possible reduction) is defined by*

  $$C[s] \rightarrow_? C[r] \text{ for each redex } s \text{ and arbitrary term } r$$

  $\rightarrow_?^*$ *and residuals defined in analogy.*
- *A redex s in t is called* ***strongly needed*** *if in every reduction sequence* $t \rightarrow_? \ldots \rightarrow_? t'$*, where* t' *is a normal form, some descendant of s gets reduced.*
- *R is* ***strongly sequential*** *if* $\forall C[\ldots,\ldots,..]$ *in normal form* $\exists i \forall \sigma . s_i$ *is strongly needed.*

Example



Strong Sequentiality

Lemma 11.38. *Let R be orthogonal.*

- *The property of being strongly sequential is decidable. The needed index i is computable.*
  Proof: See e.g. Huet-Levy
- *Call-by-need is a computable one step reduction strategy for such systems.*

Summary: Formal Specification and Verification Techniques

- What have we learned? $\rightsquigarrow$ See contents of lecture.
- Which were the important notions about FSVT?
- Are formal methods helpful for better software development?
- Can formal methods be integrated in SD-Process models?
- What is needed in order to understand and use formal methods?
- Are there criteria for evaluating formal methods?
- The importance of knowing what one does....

Principles to make a formal method a useful tool in system development

- formal syntax
- formal semantics
- clear conceptual system model
- uniform notion of an interface
- sufficient expressiveness and descriptive power
- concept of development techniques with a proper notion of refinement and implementation

Model oriented specification techniques

- ASM
- VDM
- Z and B-Methods
- SDL
- STATECHARTS
- CSP, Petri-Nets (concurrent)
- ....

Property oriented specification techniques

- Algebraic Specification Techniques (equational logic)
- Logical Specification Techniques (Prolog, temporal- and modal logics)
- Hybrids
- LARCH, OBJ, MAUDE,....
- Tools: http://rewriting.loria.fr/
- ....

Interesting reading:
http://www.comp.lancs.ac.uk/computing/resources/IanS/SE6/Slides/PDF/ch9.
http://libra.msra.cn/ConferenceDetail.aspx?id=1618

Verification techniques

Important: What and where should something hold...
What to do when it does not hold?
Use the proper tools depending on the abstraction level.

- Equational Logic (Term Rewriting ...)
- Equational properties in a single model (Induction methods....)
- First order Logics (General theorem provers...)
- First order properties of single models (Inductive methods...)
- Temporal and modal logics (Propositional part...Model checking)
- Propositional logics (Sat solvers, Davis Putman, tableaux,...)

FSVT

- **Thanks for your attention**