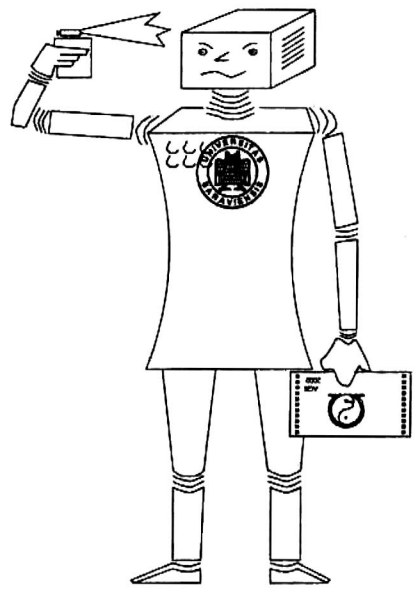
UNIVERSITÄT DES SAARLANDES
FACHBEREICH INFORMATIK
D-66041 SAARBRÜCKEN
GERMANY
WWW: http://www.ags.uni-sb.de/

SEKI Report
ISSN 1437-4447



# The CALCULEMUS Autumn School 2002: Course Notes (Part I)

Christoph Benzmüller and Regine Endsuleit (Eds.)

chris@ags.uni-sb.de, endsulei@ira.uka.de

Fachbereich Informatik
Universität des Saarlandes, Saarbrücken, Germany

Fakultät für Informatik
Universität Karlsruhe (TH), Karlsruhe, Germany

SEKI Report SR-02-07

# The CALCULEMUS Autumn School 2002: Course Notes (Part I)

Christoph Benzmüller and Regine Endsuleit (Eds.)

Fachbereich Informatik
Universität des Saarlandes, Saarbrücken, Germany
chris@ags.uni-sb.de

Fakultät für Informatik
Universität Karlsruhe (TH), Karlsruhe, Germany
endsulei@ira.uka.de

## Sponsors

CALCULEMUS Autumn School 2002 gratefully acknowledges the sponsorship of the following organizations and companies:

- EU CALCULEMUS Network
  http://www.eurice.de/calculemus/
- EU IST Programme, Future and Emerging Technologies (FET)
  http://www.cordis.lu/ist/fethome.htm
- Dipartimento di Matematica, Universitá di Pisa
  http://www.dm.unipi.it/
- German Research Center for Artificial Intelligence GmbH (DFKI)
  http://www.dfki.de/
- Comune di Pisa
  http://www.comune.pisa.it/doc/cittapisa.htm
- Saar Toto GmbH
  http://www.saartoto.de/
- RIACA Research Institute for Applications of Computer Algebra
  http://www.riaca.win.tue.nl/index.html

Contents

1 Alan Bundy
*University of Edinburgh, Scotland*

**Evening Talk: Deduction Systems and Proof Planning**

**Course: Proof Planning**

**Tutorial: Proof Planning with XBARNACLE**


# Deduction Systems and Proof Planning

Alan Bundy
Division of informatics
University of Edinburgh





## Deduction Systems

- Axioms, conjectures and rules represented as logical formulae.
- Deduction system conducts proof search.
- Naive search leads to combinatorial explosion.
- Logical proofs are low-level, long and hard to understand.



## Evidence for Higher-Level Explanations

- Understanding whole proof *vs* understanding details.
- Common structure in proofs.
- Old proofs guide search for new ones.
- Interesting *vs* routine proof steps
- Intuition of theoremhood.
- Varying learning abilities.



## Understanding Mathematical Proofs

- Alan Robinson:

  Proof = Guarantee + Explanation

- Logic provides 'guarantee' and low-level explanation.
- Need high-level explanation too.
- Provided by proof plans.

## Common Structure in Proofs 1: Rippling

**Associativity of Addition**

**Induction Hypothesis:**

$$x + (y + z) = (x + y) + z$$

**Induction Conclusion:**

$$s(x)^{\uparrow} + (y + z) = (\, s(x)^{\uparrow} + y) + z$$

$$s(x + (y + z))^{\uparrow} = s(x + y)^{\uparrow} + z$$

$$s(x + (y + z))^{\uparrow} = s((x + y) + z)^{\uparrow}$$

$$x + (y + z) = (x + y) + z$$

where $s(x) = x + 1$.

**Wave Rules:**

$$s(U)^{\uparrow} + V \Rightarrow s(U + V)^{\uparrow}$$

$$s(U)^{\uparrow} = s(V)^{\uparrow} \Rightarrow U = V$$



## Common Structure in Proofs 2: Rippling

**Additivity of Even Numbers**

**Induction Hypothesis:**

$$even(x) \wedge even(y) \rightarrow even(x + y)$$

**Induction Conclusion:**

$$even(\, s(s(x))^{\uparrow}) \wedge even(y) \rightarrow even(\, s(s(x))^{\uparrow} + y)$$

$$even(x) \wedge even(y) \rightarrow even(\, s(\, s(x)^{\uparrow} + y)^{\uparrow})$$

$$even(x) \wedge even(y) \rightarrow even(\, s(s(x + y))^{\uparrow})$$

$$even(x) \wedge even(y) \rightarrow even(x + y)$$

**Wave Rules:**

$$s(U)^{\uparrow} + V \Rightarrow s(U + V)^{\uparrow}$$

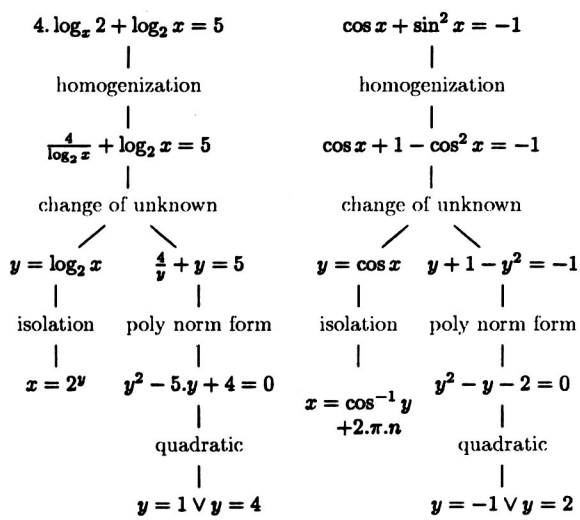
$$s(U)^{\uparrow} = s(V)^{\uparrow} \Rightarrow U = V$$

$$even(\, s(s(U))^{\uparrow}) \Rightarrow even(U)$$



## Common Structure in Proofs 3:

**Equation Solving**






## Proof Plans: What Are They?

- Attempt to capture common structure of family of proofs.
- Used to guide search for new proofs from same family.
- Three parts: tactic, method and critics.

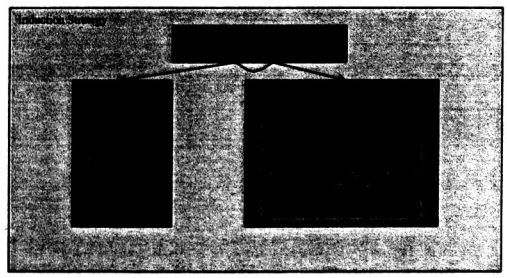
  Tactic is computer program for applying rules of inference.

  Method is meta-logical specification of tactic.

  Critic analyses failure and suggests patch.
- Use AI plan formation to construct special-purpose proof plan for conjecture using general-purpose sub-proof plans.
- Allows flexible application of heuristics.
- Understanding gained suggests extensions of heuristics.

## General-Purpose Proof Plans

**A Strategy for Inductive Proof: ind_strat**



**Preconditions:**

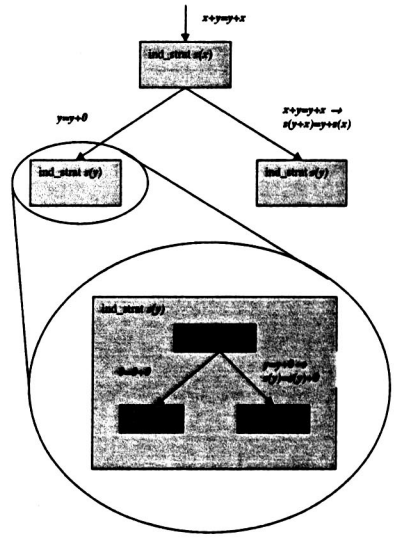
**Declarative:** Rippling must be possible in step cases.

**Procedural:** Look-ahead to choose induction rule that will permit rippling.



## Special-Purpose Proof Plan

**Commutativity of +**





## Critic: Lemma Speculation

- **Conjecture:**

$$rev(rev(L)) = L$$

- **Wave-Rule:**

$$rev(\ H :: T\ ^{\uparrow}) \Rightarrow\ rev(T) <> H :: nil\ ^{\uparrow}$$

- **Induction Conclusion:**

$$rev(rev(\ h :: t\ ^{\uparrow})) =\ h :: t\ ^{\uparrow}$$

$$\underbrace{rev(\ rev(t) <> (h :: nil)\ ^{\uparrow})}_{\textbf{blocked}} =\ h :: t\ ^{\uparrow}$$

- **Pattern Sought:**

$$rev(\ X <> Y\ ^{\uparrow}) \Rightarrow\ F(rev(X), X, Y)\ ^{\uparrow}$$

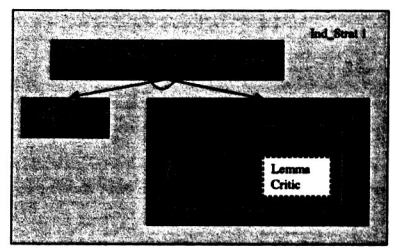
- **Lemma Discovered:**

$$rev(\ X <> Y\ ^{\uparrow}) \Rightarrow\ rev(Y) <> rev(X)\ ^{\uparrow}$$

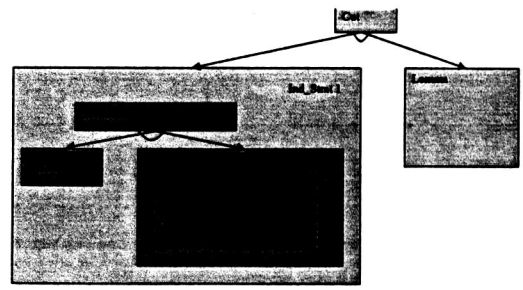


## Overview of Lemma Speculation Critic

**Critic Invocation:**




**Critic Applied:**



## Empirical Success of Proof Planning

- Implemented in *Clam*/$\lambda$*Clam* and Ωmega proof planners.
- Successfully tested on a wide range domains: induction, analysis, residue classes, diagonalisation, summing series, equational reasoning, process algebras, transfinite ordinals, completeness proofs, ...
- Applied outwith mathematics: computer configuration and bridge.
- Solution of eureka problems using critics, *e.g.* lemma discovery and generalisation.
- Applications to software/hardware verification/synthesis/transformation. *e.g.* verification of Gordon computer, construction of loop invariants via critics.
- Ωmega linked to $3^{rd}$ party provers, CAS, constraint solvers, *etc.*



## Criteria for Assessing Proof Plans

**Correctness:** Associated tactic will construct proof step.

**Intuitiveness:** Plan feels right.

**Psychological Validity:** Plan agrees with experiments on humans.

**Expectancy:** The more accurately success can be predicted the better.

**Generality:** The more proofs are accounted for by the plan the better.

**Prescriptiveness:** The less search the tactic generates the better.

**Simplicity:** The simpler the tactic the better.

**Efficiency:** The cheaper the tactic the better.

**Parsimony:** The fewer proof plans the better.



## Explanatory Role of Proof Plans

- Understanding whole proof *vs* understanding details. proof plan *vs* logical proof.
- Common structure in proofs. common proof plans.
- Old proofs guide search for new ones. use proof plan as guide.
- Interesting *vs* routine proof steps outside proof plan *vs* inside.
- Intuition of theoremhood. have proof plan but no logical proof.
- Varying learning abilities. have concepts to build proof plan.



## Advantages of Proof Planning

- Reduction in search: larger steps, fewer options.
- Multi-level proof explanation: supports interaction.
- Non-standard proof exploration, least-commitment devices: meta-variables and constraints. agent-based proof planning.
- Framework for inter-operating reasoners.

## Conclusion

- Proof plans provide multi-level proof structure.
- Used to guide and explain proofs.
- Supports flexible construction of proof.
- Supports inter-operation of reasoners.
- Need for more search in proof methods and critics.
- Need for automatic construction of proof plans.





## Disadvantages of Proof Planning

- Loss of completeness: limited to methods and critics.
- Lack of serendipity: only anticipated patterns.
- Hard work to discover new proof plans,
  need to invent new concepts, *e.g.* wave-fronts,
  which are then a barrier to human understanding.
- Danger of over-tuning,
  a proof plan for every proof.





## Future Directions

- More exploitation of flexible proof construction.
- Use of MathWeb for interacting reasoners.
- More use of search – perhaps at higher levels.
- Automatic proof plan construction.
- Diligent application generality, parsimony, expectancy, *etc.*


# Proof Planning 1:
# Some Problems and a Plan

Alan Bundy
Division of informatics
University of Edinburgh





## The Need for Inductive Inference

Proof by mathematical induction required for reasoning about repetition, *e.g.* in:

- recursive datatypes: numbers, lists, sets, trees, *etc*;
- iterative or recursive computer programs;
- electronic circuits with loops or parameterisation.

So needed for proof obligations in formal methods.



## Theoretical Limitations of Inductive Inference

**Incompleteness:** (Gödel)

formal theories exclude some theorems,

examples found by Kirby & Paris.

**Undecidability of Halting Problem:** (Turing)

no algorithmic test for termination.

**Failure of Cut Elimination:** (Kreisel).

need to generalise and introduce lemmas.



## Recursive Definitions 1

Add Two Numbers:

$$0 + N = N$$
$$s(M) + N = s(M + N)$$

Even Numbers:

$$even(0) \quad even(s(0))$$
$$even(s(s(N))) \leftrightarrow even(N)$$

Append Two Lists:

$$nil <> L = L$$
$$(H :: T) <> L = H :: (T <> L)$$

## Recursive Definitions 2

**Length of a List:**

$$len(nil) = 0$$
$$len(H :: T) = s(len(T))$$

**Reverse a List:**

$$rev(nil) = nil$$
$$rev(H :: T) = rev(T) <> (H :: nil)$$

**(Quickly) Reverse a List:**

$$qrev(nil, L) = L$$
$$qrev(H :: T, L) = qrev(T, H :: L)$$

**Rotate a List:**

$$rot(0, L) = L$$
$$rot(s(N), nil) = nil$$
$$rot(s(N), H :: T) = rot(N, T <> (H :: nil))$$



## Some Induction Rules

**Natural Numbers – One-Step:**

$$\frac{P(0),\quad \forall n{:}nat.\ (P(n) \rightarrow P(s(n)))}{\forall n{:}nat.\ P(n)}$$

**Natural Numbers – Two-Step:**

$$\frac{P(0),\quad P(s(0)),\quad \forall n{:}nat.\ (P(n) \rightarrow P(s(s(n))))}{\forall n{:}nat.\ P(n)}$$

**Lists – One-Step:**

$$\frac{P(nil)\quad \forall h{:}\tau.\forall t{:}list(\tau).\ P(t) \rightarrow P(h :: t)}{\forall l{:}list(\tau).\ P(l)}$$



## (Lack of) Cut Elimination

**Gentzen's Cut Rule:**

$$\frac{A, \Gamma \vdash \Delta,\quad \Gamma \vdash A}{\Gamma \vdash \Delta}$$

lacks subformula property.

**Cut Elimination Theorem:**

Gentzen showed Cut Rule redundant in FOL.

Kreisel showed necessary in inductive theories.

**Practical Consequences:**

Need to generalise conjectures.

Need to introduce lemmas.



## Need for Intermediate Lemmas

**Conjecture:**

$$\forall l{:}list(\tau).\ rev(rev(l)) = l$$

**Rewrite Rules:**

$$rev(nil) \Rightarrow nil$$
$$rev(H :: T) \Rightarrow rev(T) <> (H :: nil)$$

**Step Case:**

$$rev(rev(t)) = t \vdash rev(rev(h :: t)) = h :: t$$
$$\vdash \underbrace{rev(rev(t) <> (h :: nil))}_{\textbf{blocked}} = h :: t$$

## Introducing an Intermediate Lemma

**Lemma Required:**

$$rev(X <> Y) \Rightarrow rev(Y) <> rev(X)$$

**Cut Rule:** introduces this:

**Original:** $\Gamma \vdash rev(rev(l)) = l$

**New:**

$\Gamma, rev(X <> Y) \Rightarrow rev(Y) <> rev(X) \vdash rev(rev(l)) = l$

**Justification:** $\Gamma \vdash rev(X <> Y) \Rightarrow rev(Y) <> rev(X)$

**Heuristics needed:** to speculate lemma.



## Step Case Unblocked

$\vdash rev(rev(t)) = t$

$\vdash rev(rev(t) <> (h :: nil)) = h :: t$

$\vdash rev(h :: nil) <> rev(rev(t)) = h :: t$

$\vdash (rev(nil) <> (h :: nil)) <> rev(rev(t)) = h :: t$

$\vdash (nil <> (h :: nil)) <> rev(rev(t)) = h :: t$

$\vdash (h :: nil) <> rev(rev(t)) = h :: t$

$\vdash h :: (nil <> rev(rev(t))) = h :: t$

$\vdash h :: rev(rev(t)) = h :: t$

$\vdash h = h \wedge rev(rev(t)) = t$

**fertilization now possible.**



## Reverse Example Revisited

**Blocked Step Case:**

$$rev(rev(t)) = t \vdash \underbrace{rev(rev(t) <> (h :: nil))}_{\textbf{blocked}} = h :: t$$

**Weak Fertilization:**

$$rev(rev(t) <> (h :: nil)) = h :: rev(rev(t))$$

**New Step Case:**

$$rev(rev(t') <> (h :: nil)) = h :: rev(rev(t'))$$
$$\vdash rev(rev(h' :: t') <> (h :: nil)) = h :: rev(rev(h' :: t'))$$
$$\vdash \underbrace{rev((rev(t') <> (h' :: nil)) <> (h :: nil))}_{\textbf{blocked}}$$
$$= h :: \underbrace{rev(rev(t') <> (h' :: nil))}_{\textbf{blocked}}$$



## Generalising a Sub-Term

**Generalisation Required:**

$$rev(t'' <> (h :: nil)) = h :: rev(t'')$$

**Step Case Unblocked:**

$rev(t'' <> (h :: nil)) = h :: rev(t'') \vdash$

$rev((h'' :: t'') <> (h :: nil)) = h :: rev(h'' :: t'')$

$rev(h'' :: (t'' <> (h :: nil))) = h :: (rev(t'') <> (h'' :: nil))$

$rev(t'' <> (h :: nil)) <> (h'' :: nil) = (h :: rev(t'')) <> (h'' :: nil)$

$rev(t'' <> (h :: nil)) = h :: rev(t'') \wedge h'' :: nil = h'' :: nil$

NB – <> definition applied right to left.

## A Proof Plan for Induction




Preconditions

**Declarative:** Rippling must be possible in step cases.

**Procedural:** Look-ahead to choose induction rule that will permit rippling.



## Annotated Step Case

$$t <> (Y <> Z) = (t <> Y) <> Z$$

$$\vdash h :: t^{\uparrow} <> (y <> z) = (h :: t^{\uparrow} <> y) <> z$$

$$\vdash h :: t <> (y <> z)^{\uparrow} = h :: t <> y^{\uparrow} <> z$$

$$\vdash h :: t <> (y <> z)^{\uparrow} = h :: (t <> y) <> z^{\uparrow}$$

$$\vdash h = h \wedge t <> (y <> z) = (t <> y) <> z^{\uparrow}$$

- changing bits in *orange boxes* $^{\uparrow}$ (wave-fronts).
- unchanging bits in ***red*** (skeleton).



## Annnotated Rewrite Rules

$$H :: T^{\uparrow} <> L \Rightarrow H :: T <> L^{\uparrow}$$

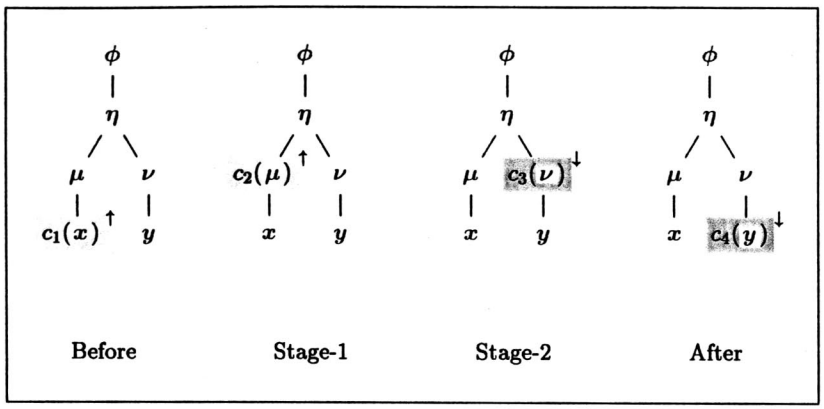
$$X_1 :: X_2^{\uparrow} = Y_1 :: Y_2^{\uparrow} \Rightarrow X_1 = Y_1 \wedge X_2 = Y_2^{\uparrow}$$

- called wave-rules.
- note outward movement of wave-fronts.



## Another View of Rippling

## Sideways Rippling: Another View






## Rippling Sideways and In

$rot(len(t), t <> K) = K <> t$

$\vdash rot(len(\ h :: t^{\uparrow}), h :: t^{\uparrow} <> \lfloor k \rfloor) = \lfloor k \rfloor <> h :: t^{\uparrow}$

$\vdash rot(\ s(len(t))^{\uparrow}, h :: t <> \lfloor k \rfloor^{\uparrow}) = \lfloor k \rfloor <> h :: t^{\uparrow}$

$\vdash rot(len(t), \boxed{t <> \lfloor k \rfloor <> (h :: nil)}^{\downarrow}) = \lfloor k \rfloor <> (\ h :: t^{\uparrow})$

$\vdash rot(len(t), t <> \boxed{\lfloor k \rfloor <> (h :: nil)}^{\downarrow}) = \boxed{\lfloor k \rfloor <> (h :: nil)}^{\downarrow} <> t$

$\vdash rot(len(t), t <> \lfloor k <> (h :: nil) \rfloor) = \lfloor k <> (h :: nil) \rfloor <> t$

- $\lfloor Sinks \rfloor$ provide alternative wave-front destination, available when free variables are in hypothesis.
- Wave-fronts have directions: $out^{\uparrow}$ / $in^{\downarrow}$.



## The Preconditions of Rippling

1. The induction conclusion contains a wave-front,
   *e.g.* $rot(\ s(len(t))^{\uparrow}, h :: t <> \lfloor k \rfloor^{\uparrow}) = \ldots.$
2. A wave-rule applies to this wave-front.
   *e.g.* $rot(\ s(N)^{\uparrow}, H :: T^{\uparrow}) \Rightarrow \ldots.$
3. Any condition is provable.
   *e.g.* $X \neq H \rightarrow X \in H :: T^{\uparrow} \Rightarrow X \in T.$
4. Inserted inwards wave-fronts contain a sink or an outwards wave-front.
   *e.g.* $rot(len(t), t <> \boxed{\lfloor k \rfloor <> (h :: nil)}^{\downarrow}) = \ldots.$



## Sideways and Inwards Wave-rules

$$rot(\ s(N)^{\uparrow}, H :: T^{\uparrow}) \Rightarrow rot(N, \boxed{T <> (H :: nil)}^{\downarrow})$$

$$L <> (\ H :: T^{\uparrow}) \Rightarrow \boxed{L <> (H :: nil)}^{\downarrow} <> T$$

$$\boxed{(X <> Y) <> Z}^{\downarrow} \Rightarrow X <> (\boxed{Y <> Z}^{\downarrow})$$

## Selective Rewriting (with Rippling)

**Wave Rule:** $(X + Y)^{\uparrow} + Z \Rightarrow X + (Y + Z)^{\uparrow}$

**Given:** $a + b = 42$

**Goal:** $((c+d)+a)^{\uparrow} + b = (c+d) + 42^{\uparrow}$

**Ripple:** $(c+d) + (a+b)^{\uparrow} = (c+d) + 42^{\uparrow}$

**Non-Ripples:**

$$((c+d)+a))^{\uparrow} + b = c + (d+42)^{\uparrow}$$

$$c + (d+a)^{\uparrow} + b = (c+d) + 42^{\uparrow}$$





## Summary

- Negative theoretical results create special search problems.
- These problems common in practice:
  induction rule choice, lemmas & generalisations.
- Proof plan for induction based on rippling.
- Rippling: selective; bidirectional; terminating and offers heuristic solution to special problems.





## Advantages of Rippling

**Selective:** not exhaustive rewriting.

**Bi-directional:** rewriting.

**Termination:** of any set of wave-rules,
despite bidirectionality.

**Heuristic basis:** for choosing lemmas, generalisations and inductions.





## Exhaustive Rewriting (without Rippling)

**Rewrite Rule:** $(X + Y) + Z \Rightarrow X + (Y + Z)$

**Given:** $a + b = 42$

**Goal:** $((c + d) + a) + b = (c + d) + 42$

**Rewritings:**

× $((c + d) + a) + b = c + (d + 42)$

× $(c + (d + a)) + b = (c + d) + 42$

√ $(c + d) + (a + b) = (c + d) + 42$


Proof Planning 2:

# Some Heuristic Solutions

Alan Bundy

Division of informatics

University of Edinburgh





## Ripple-Based Heuristics

**Induction Rules:** choose induction which best supports rippling.

**Lemmas:** design wave-rule to unblock ripple.

**Generalisation:** generalise goal to allow wave-rule to apply.



## Induction Variable Selection

$\forall t, l, m{:}list(nat).$

$$t <> (l <> m) = (t <> l) <> m$$

| $\swarrow$ | $\downarrow$ | $\searrow$ |
|---|---|---|
| $t^{\uparrow} <> (l <> m) =$ | $t <> (l^{\uparrow} <> m) =$ | $t <> (l <> m^{\uparrow}) =$ |
| $(t^{\uparrow} <> l) <> m$ | $(t <> l^{\uparrow}) <> m$ | $(t <> l) <> m^{\uparrow}$ |



## Induction Rules Available

$$\frac{P(nil) \quad \forall h{:}\tau.\forall t{:}list(\tau).\ (P(t) \rightarrow P(h :: t^{\uparrow}))}{\forall l{:}list(\tau).\ P(l)} \quad \ldots :: \ldots$$

$$\frac{P(nil) \quad \forall h{:}\tau.\ P(h :: nil) \quad \forall h_1, h_2{:}\tau.\forall t{:}list(\tau).\ (P(t) \rightarrow P(h_1 :: h_2 :: t^{\uparrow}))}{\forall l{:}list(\tau).\ P(l)} \quad \ldots :: \ldots :: \ldots$$

$$\frac{P(nil) \quad \forall h{:}\tau.\ P(h :: nil) \quad \forall t_1, t_2{:}list(\tau).\ (P(t_1) \wedge P(t_2) \rightarrow P(t_1 <> t_2^{\uparrow}))}{\forall l{:}list(\tau).\ P(l)} \quad \ldots <> \ldots$$

$\vdots \qquad \vdots$

## Ripple Analysis

**Wave Rule:**

$$( X :: Y^{\uparrow} ) <> Z \Rightarrow X :: (Y <> Z)^{\uparrow}$$

**Wave Occurrences:**

$$\forall t, l, m{:}list(nat).\ t^{\uparrow} <> ( l^{\uparrow} <> m) = ( t^{\uparrow} <> l) <> m$$

**Induction Variable:**

| variable | occurrences |
|---|---|
| $t$ | all unflawed |
| $l$ | some unflawed, some flawed |
| $m$ | all flawed |

$t$ is best choice since it has the fewest flaws.

**Induction Rule:**

| wave-front | induction rule |
|---|---|
| $X :: \ldots^{\uparrow}$ | $\ldots :: \ldots$ |
| $X_1 :: X_2 :: \ldots^{\uparrow}$ | $\ldots :: \ldots :: \ldots$ |
| $\ldots <> \ldots^{\uparrow}$ | $\ldots <> \ldots$ |

$X :: \ldots^{\uparrow}$ suggests $\ldots :: \ldots$ induction rule.



## Failure of Rippling Analysis

**Conjecture:**

$$\forall t, l{:}list(\tau).\ even(len(t <> l)) \leftrightarrow even(len(l <> t))$$

**Wave-Rules:**

$$X :: Y^{\uparrow} <> Z \Rightarrow X :: (Y <> Z)^{\uparrow}$$

$$even( s(s(X))^{\uparrow} ) \Rightarrow even(X)$$

$$len( H :: T^{\uparrow} ) \Rightarrow s(len(T))^{\uparrow}$$

$$len(L <> H :: T^{\uparrow}) \Rightarrow s(len(L <> T))^{\uparrow}$$

**Step Case:** using $\ldots :: \ldots$ induction on $t$.

$$even(len( h :: t^{\uparrow} <> l)) \leftrightarrow even(len(l <> h :: t^{\uparrow}))$$

$$even(len( h :: (t <> l)^{\uparrow})) \leftrightarrow even( s(len(l <> t))^{\uparrow})$$

$$\underbrace{even( s(len(t <> l))^{\uparrow})}_{\textbf{blocked}} \leftrightarrow \underbrace{even( s(len(l <> t))^{\uparrow})}_{\textbf{blocked}}$$

because only one-level look-ahead.



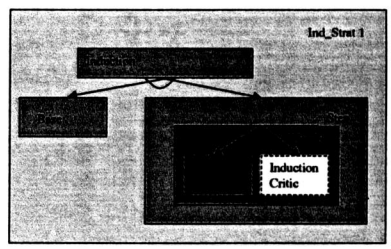
## Failure of Ripple Preconditions

- Precondition 1 is true:
  1. The induction conclusion contains a wave-front.
     $$even( s(len(t <> l))^{\uparrow}) \leftrightarrow \ldots$$
     (other side similar)
- Precondition 2 is false:
  2. A wave-rule applies to this wave-front.
     however, there is a near miss:
     $$even( s(s(X))^{\uparrow}) \Rightarrow even(X)$$
- Preconditions 3 and 4 are inapplicable.
  3. Any condition is provable.
  4. Inserted inwards wave-fronts contain a sink or an outwards wave-front.

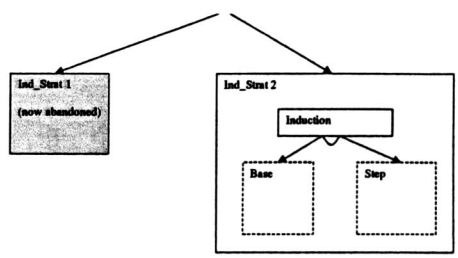


## Overview of Induction Revision Critic

**Critic Invocation:**




**Critic Applied:**



## Patch: Rechoose Induction Rule

Blocked Goal: $even(\ s(len(t <> l))^{\uparrow}\ ) \leftrightarrow \ldots$

Desired Goal: $even(\ s(s(len(t <> l)))^{\uparrow}\ ) \leftrightarrow \ldots$

Inwards Wave-Rules:

$$H :: (T <> L)^{\downarrow} \Rightarrow H :: T^{\downarrow} <> L$$

$$s(len(T))^{\downarrow} \Rightarrow len(\ H :: T^{\downarrow}\ )$$

Calculation of New Induction Term:

$$even(\ s(s(len(\lfloor t \rfloor <> l)))^{\downarrow}\ ) \leftrightarrow \ldots$$

$$even(\ s(len(\ H_2 :: (\lfloor t \rfloor <> l)^{\downarrow}\ ))^{\downarrow}\ ) \leftrightarrow \ldots$$

$$even(len(\ H_1 :: H_2 :: (\lfloor t \rfloor <> l)^{\downarrow}\ )) \leftrightarrow \ldots$$

$$even(len(\ H_1 :: (\lfloor H_2 :: t \rfloor <> l)^{\downarrow}\ )) \leftrightarrow \ldots$$

$$even(len(\lfloor H_1 :: H_2 :: t \rfloor <> l)) \leftrightarrow \ldots$$

suggests $\ldots :: \ldots :: \ldots$ induction on $t$.



## Patch: Apply New Induction Rule

New Induction Rule:

$$\frac{P(nil),\quad \forall h{:}\tau.\ P(h :: nil),\quad \forall h_1{:}\tau.\forall h_2{:}\tau.\forall t{:}list(\tau).\ P(t) \rightarrow P(\ h_1 :: h_2 :: t^{\uparrow}\ )}{\forall l{:}list(\tau).\ P(l)}$$

New Step Case:

$$even(len(\ h_1 :: h_2 :: t^{\uparrow} <> l)) \leftrightarrow even(len(l <>\ h_1 :: h_2 :: t^{\uparrow}\ ))$$

$$even(len(\ h_1 :: (\ h_2 :: t^{\uparrow} <> l)^{\uparrow}\ )) \leftrightarrow even(len(l <>\ h_1 :: h_2 :: t^{\uparrow}\ ))$$

$$even(\ s(len(\ h_2 :: t <> l^{\uparrow}\ ))^{\uparrow}\ ) \leftrightarrow even(\ s(len(l <>\ h_2 :: t^{\uparrow}\ ))^{\uparrow}\ )$$

$$even(\ s(s(len(t <> l)))^{\uparrow}\ ) \leftrightarrow even(\ s(s(len(l <> t)))^{\uparrow}\ )$$

$$even(len(t <> l)) \leftrightarrow even(len(l <> t))$$



## Rippling Failure: Missing Wave-Rule

Conjecture:

$$\forall n{:}nat.\ even(n + n)$$

Wave-Rules:

$$s(X)^{\uparrow} + Y \Rightarrow s(X + Y)^{\uparrow}$$

$$even(\ s(s(X))^{\uparrow}\ ) \Rightarrow even(X)$$

Induction Conclusion:

$$even(\ s(n)^{\uparrow} + s(n)^{\uparrow}\ )$$

$$\underbrace{even(\ s(n + s(n)^{\uparrow})^{\uparrow}\ )}_{\textbf{blocked}}$$



## Failure of Ripple Preconditions

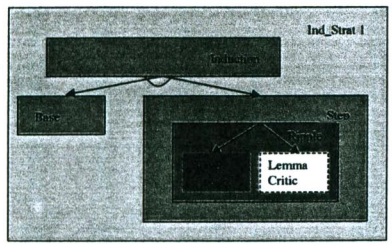
- Precondition 1 is true:
  1. The induction conclusion contains a wave-front.

     $$even(\ s(n + s(n)^{\uparrow})^{\uparrow}\ )$$

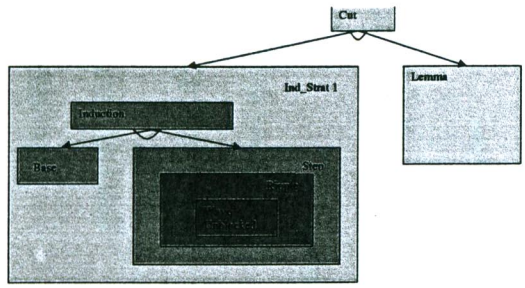
     (in fact, two)
- Precondition 2 is false:
  2. A wave-rule applies to this wave-front.
     (to neither of them)
- Preconditions 3 and 4 are inapplicable.
  3. Any condition is provable.
  4. Inserted inwards wave-fronts contain a sink or an outwards wave-front.

## Overview of Lemma Speculation Critic

Critic Invocation:




Critic Applied:






## Patch: Lemma Speculation

Blocked Goal:

$$even(\ s(n + s(n)^{\uparrow})^{\uparrow}\ )$$

focus on innermost wave-front.

Schematic Wave-Rule:

$$X + s(Y)^{\uparrow} \Rightarrow F(X + Y)^{\uparrow}$$

Continued Ripple:

$$even(\ s(F(n + n))^{\uparrow}\ )$$
$$even(n + n)$$

where $F = s$.

Final Wave-Rule:

$$X + s(Y)^{\uparrow} \Rightarrow s(X + Y)^{\uparrow}$$

which must now be proved.



## Rippling Failure: Missing Sink

Conjecture:

$$\forall t{:}list(\tau).\ rev(t) = qrev(t, nil)$$

Wave-Rules:

$$rev(\ H :: T^{\uparrow}\ ) \Rightarrow rev(T) <> H :: nil^{\uparrow}$$

$$qrev(\ H :: T^{\uparrow}, L) \Rightarrow qrev(T, \boxed{H :: L}^{\downarrow})$$

Induction Conclusion:

$$rev(\ h :: t^{\uparrow}\ ) = qrev(\ h :: t^{\uparrow}, nil)$$

$$rev(t) <> h :: nil^{\uparrow} = \underbrace{qrev(\ h :: t^{\uparrow}, nil)}_{\textbf{missing sink}}$$



## Failure of Ripple Preconditions

- Preconditions 1, 2 and 3 are true:
  1. The induction conclusion contains a wave-front.
     $\ldots = qrev(\ h :: t^{\uparrow}, nil)$
  2. A wave-rule applies to this wave-front.
     $qrev(\ H :: T^{\uparrow}, L) \Rightarrow qrev(T, \boxed{H :: L}^{\downarrow})$
  3. Any condition is provable — trivially, no condition.
- Precondition 4 is false:
  4. Inserted inwards wave-fronts contain a sink or an outwards wave-front.
     $\ldots = qrev(t, \boxed{h :: nil}^{\downarrow})$

## Patch: Instantiating the Meta-Variables

New Step Case:

$$F(rev(\ h :: t\ ^{\uparrow}), \lfloor l \rfloor) = qrev(\ h :: t\ ^{\uparrow}, G(\lfloor l \rfloor))$$

$$F(\ rev(t) <> h :: nil\ ^{\uparrow}, \lfloor l \rfloor) = qrev(t, \boxed{h :: G(\lfloor l \rfloor)}^{\downarrow})$$

$$rev(t) <> (\ \boxed{h :: nil <> F'(\ rev(t) <> (h :: nil)\ ^{\uparrow}, \lfloor l \rfloor)}^{\downarrow}\ ) = qrev(t, \boxed{h :: G(\lfloor l \rfloor)}^{\downarrow})$$

$$rev(t) <> (\ \boxed{h :: F'(\ rev(t) <> (h :: nil)\ ^{\uparrow}, \lfloor l \rfloor)}^{\downarrow}\ ) = qrev(t, \boxed{h :: G(\lfloor l \rfloor)}^{\downarrow})$$

$$rev(t) <> (\lfloor h :: l \rfloor) = qrev(t, \lfloor h :: l \rfloor)$$

where $F = <>$, $F' = \lambda X.\lambda Y.\ Y$ and $G = \lambda X.\ X$.

Key Wave-Rule: $(\ X <> Y\ ^{\uparrow}) <> Z \Rightarrow X <> (\boxed{Y <> Z}^{\downarrow})$

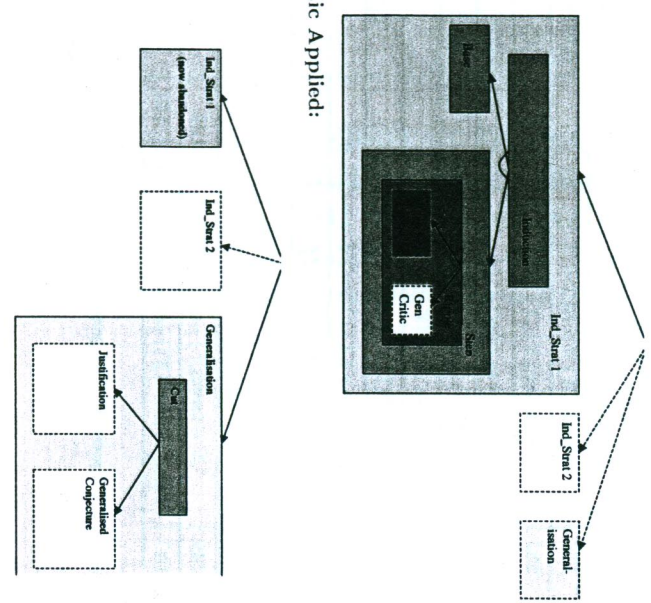
Generalised Conjecture: $\forall t{:}list(\tau).\forall l{:}list(\tau).\ rev(t) <> l = qrev(t, l)$



## Overview of Generalisation Critic

Critic Invocation:

Critic Applied:






## Pattern of Failure Suggests Patch

| Precondition | Generalization | Case analyse | Induction revision | Lemma discovery |
|---|---|---|---|---|
| 1 | ⋆ | ⋆ | ⋆ | ⋆ |
| 2 | ⋆ | ⋆ | ○ | ● |
| 3 | ⋆ | ● | | |
| 4 | ● | | | |

⋆ = success ○ = partial success ● = failure



## Patch: Sink Speculation

Original Conjecture:

$$\forall t{:}list(\tau).\ rev(t) = qrev(t, nil)$$

Disallowed Ripple:

$$\ldots = qrev(t, \boxed{h :: nil}^{\downarrow})$$

Schematic Conjecture:

$$\forall t{:}list(\tau).\forall l{:}list(\tau).\ F(rev(t), l) = qrev(t, G(l))$$

Induction Hypothesis:

$$F(rev(t), L) = qrev(t, G(L))$$

where $F$, $G$ and $L$ are meta-variables.

## Results: Selected Conjectures 1

| No | Conjecture | (i) | (ii) |
|---|---|---|---|
| T3 | $len(X <> Y) = len(Y) + len(X)$ | | L1 |
| T4 | $len(X <> X) = double(len(X))$ | | L2 |
| T5 | $len(rev(X)) = len(X)$ | | L3 |
| T8 | $n^{th}(X, n^{th}(Y, Z)) = n^{th}(Y, n^{th}(X, Z))$ | | L4, L5 |
| T10 | $rev(rev(X)) = X$ | | L8 |
| T11 | $rev(rev(X) <> rev(Y)) = Y <> X$ | | L9, L10 |
| T12 | $qrev(X, Y) = rev(X) <> Y$ | | L11 |
| T13 | $half(X + X) = X$ | | L1 |
| T14 | $ordered(isort(X))$ | | L12 |
| T16 | $even(X + X)$ | | L1 |
| T21 | $rotate(len(X), X <> Y) = Y <> X$ | | L11, L13 |
| T22 | $even(len(X <> Y)) \leftrightarrow even(len(Y <> X))$ | $\star$ | L14 |
| T23 | $half(len(X <> Y)) = half(len(Y <> X))$ | $\star$ | L15 |
| T26 | $half(X + Y) = half(Y + X)$ | $\star$ | L17 |

(i) induction revision, (ii) lemma discovery



## Results: Selected Conjectures 2

| No | Conjecture | (ii) | (iii) | (iv) |
|---|---|---|---|---|
| T27 | $rev(X) = qrev(X, nil)$ | | G1 | |
| T29 | $rev(qrev(X, nil)) = X$ | | G3,G4 | |
| T30 | $rev(rev(X) <> nil) = X$ | | G5,G6 | |
| T32 | $rotate(len(X), X) = X$ | | G9 | |
| T33 | $fac(X) = qfac(X, 1)$ | | G10 | |
| T35 | $exp(X, Y) = qexp(X, Y, 1)$ | | G12 | |
| T36 | $X \in Y \rightarrow X \in (Y <> Z)$ | | | $\star$ |
| T39 | $X \in n^{th}(Y, Z) \rightarrow X \in Z$ | | | $\star$ |
| T40 | $X \subset Y \rightarrow (X \cup Y = Y)$ | | | $\star$ |
| T42 | $X \in Y \rightarrow X \in (Y \cup Z)$ | | | $\star$ |
| T45 | $X \in insert(X, Y)$ | | | $\star$ |
| T48 | $len(isort(X)) = len(X)$ | L18 | | $\star$ |
| T49 | $X \in isort(Y) \rightarrow X \in Y$ | L19 | | $\star$ |
| T50 | $count(X, isort(Y)) = count(X, Y)$ | L20, L21 | | $\star$ |

(ii) lemma discovery, (iii) generalization, (iv) casesplit



## Results: Selected Lemmata

| No | Lemma |
|---|---|
| L1 | $X + s(Y) = s(X + Y)$ |
| L3 | $len(X <> Y :: nil) = s(len(X))$ |
| L5 | $n^{th}(s(V), n^{th}(s(W), X :: Y :: Z)) = n^{th}(s(V), n^{th}(W, X :: Z))$ |
| L8 | $rev(X <> (Y :: nil)) = Y :: rev(X)$ |
| L10 | $rev((X <> Y :: nil) <> nil) = Y :: rev(X <> nil)$ |
| L11 | $(X <> (Y :: nil)) <> Z = X <> (Y :: Z)$ |
| L12 | $ordered(Y) \rightarrow ordered(insert(X, Y))$ |
| L14 | $even(len(W <> Z)) \rightarrow even(len(W <> X :: Y :: Z))$ |
| L15 | $len(X <> Y :: Z :: nil) = s(s(len(X)))$ |
| L16 | $even(X + Y) \rightarrow even(X + s(s(Y)))$ |
| L17 | $half(X + s(s(Y))) = s(half(X + Y))$ |
| L18 | $len(insert(X, Y)) = s(len(Y))$ |
| L19 | $X \neq Y \rightarrow (X \in insert(Y, Z) \rightarrow X \in Z)$ |
| L20 | $count(X, insert(X, Y)) = s(count(X, Y))$ |
| L21 | $X \neq Y \rightarrow (count(X, insert(Y, Z)) = count(X, Z))$ |
| L22 | $(X <> Y) <> Z = X <> (Y <> Z)$ |
| L23 | $(X * Y) * Z = X * (Y * Z)$ |
| L24 | $(X + Y) + Z = X + (Y + Z)$ |



## Results: Generalizations

| No | Generalization | Lemmata |
|---|---|---|
| G1 | $rev(X) <> Y = qrev(X, Y)$ | L22 |
| G2 | $revflat(X) <> Y = qrevflat(X, Y)$ | L22 |
| G3 | $rev(qrev(X, Y)) = rev(Y) <> X$ | L11 |
| G4 | $rev(qrev(X, rev(Y))) = Y <> X$ | L8,L11 |
| G5 | $rev(rev(X) <> Y) = rev(Y) <> X$ | L11 |
| G6 | $rev(rev(X) <> rev(Y)) = Y <> X$ | L8,L11 |
| G7 | $qrev(qrev(X, Y), nil) = rev(Y) <> X$ | L11 |
| G8 | $qrev(qrev(X, rev(Y)), nil) = Y <> X$ | L8,L11 |
| G9 | $rotate(len(X), X <> Y) = Y <> X$ | L11, L22 |
| G10 | $fac(X) * Y = qfac(X, Y)$ | L23 |
| G11 | $(X * Y) + Z = mult(X, Y, Z)$ | L24 |
| G12 | $exp(X, Y) * Z = qexp(X, Y, Z)$ | L23 |

## Summary

- Ripple analysis: **Induction rules chosen to suit rippling.**
- **Different** patterns **of proof breakdown** suggest **different** patches.
- **Ripple breakdowns suggest:** induction **revision;** lemma **speculation or** generalisation.
- **Implemented via proof planning with critics.**

2 Czeslaw Bylinski
*University of Bialystok, Poland*

Course: MIZAR: Development of a Large Mathematical Library

# Mizar

## The development of large mathematical library

Czesław Bylinski – University of Białystok
E-mail: bylinski@math.uwb.edu.pl

project url: http://www.mizar.org

# The Mizar Project

- The Mizar Project is an attempt to create a computer system for writing, checking and accumulating whole mathematics.
- Mizar is the name of the language that is both rigorous and close to mathematical vernacular, designed by Andrzej Trybulec.
- Mizar is the computer system for checking and collecting articles written in Mizar.
- Mizared mathematical knowledge is collected in *the Mizar Mathematical Library* (MML).

# A bit of history

- The project Mizar started almost 30 years ago (1973) in Poland under the leadership of **Andrzej Trybulec.**
- Its original goal was to design and implement of software environment to assist the process of preparing mathematical papers.The project can be seen as an attempt to develop software environment for writing traditional mathematical papers, where classical logic and set theory form the basis of all future developments.
- After several year of experiments with Mizar Languages and its implementations ( 1980 - Mizar 2, Mizar MSE – 1981, 1987 - Mizar 4) in 1988 the design of the language was completed by A. Trybulec. This language is named simply Mizar.

# The elements of the Mizar system

- The system has been equipped with the following:

1. Mizar language
2. Mizar proof checker called verifier is a computer system for verifying correctness and logical validity of Mizar texts.
3. The mathematical knowledge base called Mizar Mathematical Library (MML),
4. Other programs for automated text processing as required to create, develope and review of the Mizar Mathematical Library,
5. The program used to automatic TeX typesetting for Formalized Mathematics , a proof-checked journal.

## The logical basis of Mizar

- The logic of Mizar is classical
- The proofs are written in natural deduction style developed by Stanisław Jaśkowski in 1934,
- Definitions allow for the introduction of constructors for types, terms, adjectives, and atomic formulae.
- Proofs consists of a sequence of steps, each step justified by facts proved in earlier steps or separate lemmas, theorems, or schemas of theorems
- The proof-checker – the notion of an obvious inference instead using some fixed set of inference rules

## The Mizar Language features

- Human readable, machine verificable proofs.
- The Mizar language has been derived from common language used in mathematical proofs and therefore is very rich.
- The Mizar Language includes the following features: comprehensive, definitional facilities, type hierarchy, structured types, attributes, and built-in fragments of arithmetics and set theory.
- The Mizar language is a multi-sorted (typed) first-order predicate logic with functions, schemata, and second-order free variables, both feasible for humans and interpretable for computers.

## The Mizar System

- The PC Mizar system is implemented on an Intel x86 based computers.
- Now we distribute releases for MS Windows and Intel-based Linux and Solaris.
- The whole Mizar system including verifier is coded in Pascal using Free Pascal (Borland Delphi 4.0).
- Current version of the system: 6.1.13
- The Mizar System for download is available at: http://www.mizar.org/system/

## Mizar Mathematical Library

- The main effort in the Mizar Project has been in building the mathematical library of mizared articles (a scripts in mizar language).
- Mizared mathematical knowledge is collected in the Mizar Mathematical Library (MML).
- The development of MML started with axiomatics of Tarski-Grothendieck set theory and axiomatic description of built-in notions.
- We collected till now more than 720 Mizar articles.
- The cross-references allowed in Mizar enable to use proved mathematical facts and defined notions in new articles.

## Preparing a Mizar article

- The user is require to give a full and explicit constuction of formal proof in Mizar Language
- The source text is prepared using any ASCII editor and typically includes from 1500 to 5000 lines.
- The text is run through the **Accomodator** The directives from the *Environment Declaration* guide the production of the environment, specific for the article. The environment is produced from the available data base.
- Now the **Verifier** is ready to start checking. The output gives remarks on unaccepted fragments of the source text.
- These three steps are repeated in a loop until no errors are flagged and the author is satisfied with the resulting text.

## Submitting a Mizar article into MML

- When finished, an article is submitted to the *Library Committee of Association of Mizar_Users* for inclusion into the *Mizar Mathematical Library*. http://www.mizar.org/
- The contributed article is subjected to a review and if needed the author must revise his paper.
- The contents of accepted papers are extracted by the **Exporter** utility and incorporated into the public data base (MML) distributed with Mizar system to all Mizar users.

## Basic tools of the Mizar System

- **Accommodator** processes the *Environment Declaration* part of a Mizar text and creates the *Environment* to this text from the Data Base (MML).
- **Verifier** process the *Text Proper* part of a Mizar test and as a result create the report on errors in A article.Verifier has no direct communication with the Data Base and verifies the correctness of the Mizar text using only knowledge stored in the *Environment.*
- **Extractor** separates reusable knowledge from the Mizar text and stores it into Data Base (MML).

## Mizar abstracts

- For every article an abstract was created by a tool called Abstractor.
- An abstract includes a presentation of all the items that can be referenced from other articles. All justifications are removed.
- All abstracts are published in a paper journal *Formalized Mathematics,*
- The electronic journal *Journal of Formalized Mathematics* (http://www.mizar.org/JFM/) allows for browsing the MML through hyperlinks.

## The Mizar Mathematical Library

- The Mizar Mathematical Library is based on Tarski-Grothendieckset theory, which consists of the Zermelo-Fraenkel (ZF) set theory complemented by Tarski's axiom.
- The task of building a rich mathematical library is currently the main effort of the Mizar community.
- The systematic collection of Mizar articles started at the beginning of 1989.
- The Mizar language and associated software evolve.
- The library has evolved; new and improved versions of the entire library are periodically produced.

## The Mizar Mathematical Library Statistics

- A large body of mathematics has alredy been formalized in the Mizar Language and proof-checked by the Mizar verifier
- 32 000 theorems
- 6 000 definitions
- 672 schemes
- 728 articles (50 MB)
- 385 000 cross-references between articles
- 120 authors, about 20 of them active on a long term basis

## The Mizar Mathematical Library

- Most of the library texts formalizes the fundamentals of introductory mathematics. Some of the texts contribute more advanced results.
- The proof of the Jordan curve theory is being continued
- The mizar formalization of a book : A *Compedium of Continous Lattices* by G. Giers, K.H. Hoffman, K. Keimel, j.D. Lawson, M. Mislove, and D. S. Scott, Springer Verlag, 1980. The work is not completed. The current state of formalization is described at: http://megrez.mizar.org/ccl/. The work was done by 16 author in 57 mizar articles.

## Some examples of theorems stored in MML

- Euler's Theorem and Small Fermat's Theorem [EULER_2],
- The Chinese Re
- mainder Theorem [WSIERP_1],
- The de l'Hospital Theorem [L_HOSPIT],
- Average Value Theorems for Real Functions of One Variable (the Rolle Theorem) [ROLLE],
- Heine-Borel's Covering Theorem also known as the Borel-Lebesgue theorem [HEINE],
- Fundamental Theorem of Algebra [POLYNOM5].

## Some examples of theorems stored in MML

- The Theorem of Weierstrass [WEIERSTR],
- Hahn-Banach Theorem in the Vector Space over the Field of Real Numbers przestrzeni rzeczywistych [HAHNBAN],
- The Hahn Banach Theorem in the Vector Space over the Field of Complex Numbers [HAHNBAN1],
- Birkhoff Theorem for Many Sorted Algebras [BIRKHOFF],
- Yoneda Embedding [YONEDA_1],
- Countable Sets and Hessenberg's Theorem [CARD_4],

## Some examples of theorems stored in MML

- The Theorem of Weierstrass [WEIERSTR],
- Fixpoints in Complete Lattices (Tarski-Knastera theorem about the existence of fixpoints) [KNASTER],
- Representation Theorem for Boolean Algebras (Stone's otheorem) [LOPCLSET],
- On Paracompactness of Metrizable Spaces [PCOMPS_2],
- Koonig's Lemma which claims that there is a infinite branch of a finite-order tree if the tree has arbitrary long finite chains) [TREES_2],
- Koonig's Theorem (about the sum and product of any number of cardinals) [CARD_3],

## The Mizar Mathematical Library

- The more detailed information about articles stored in MML is available
  http://www.mizar.org/JFM
- Information on Mizar system is available
  http://www.mizar.org/
  http://mizar.uwb.edu.pl

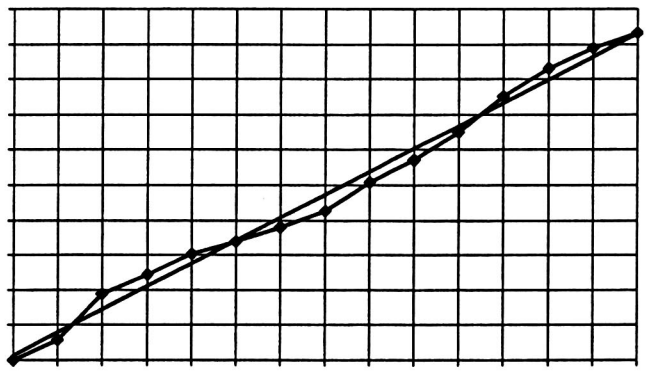
## Axiomatics of the Mizar Mathematical Library

- axioms constitute :
- everything is a set,
- the axiom of extensionality,
- the axiom of pairs,
- the axiom of unions,
- the axiom of power sets
- the axiom of regularity,
- the Fraenkel scheme of substitution, and\item
- the Tarski's axiom

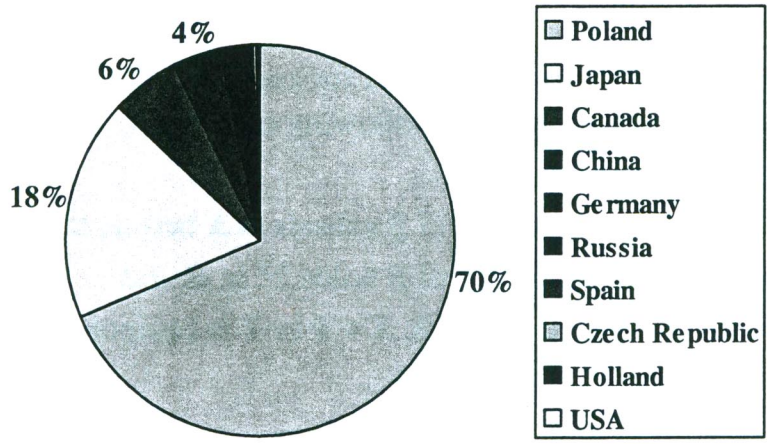
## Tarski Axiom

- The last axiom claims that for every set *A* there is a set *B* satisfying the following conditions:
- A ∈ B,
- for every element *X* of *B* subsets of *X* belong to *B*,
- for every element *X* of *B* holds $2^X$ in *B*, and
- for every subset *X* of *B*, *X* ∈ *B* or *X* is equipotent with *B*.

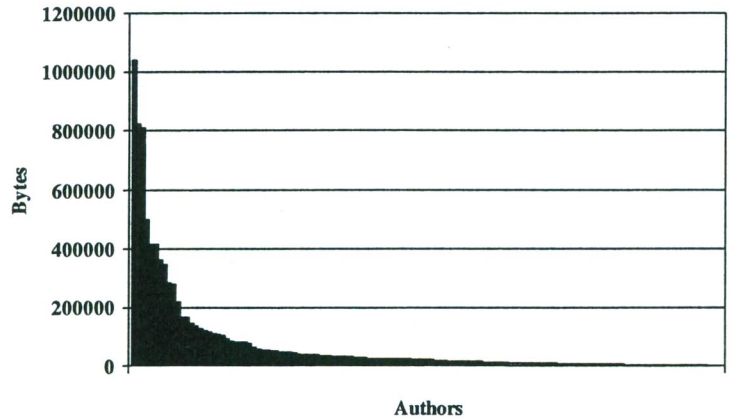
## The Cumulative Contribution in the Consecutive Years



## The Size of Contribution by Countries



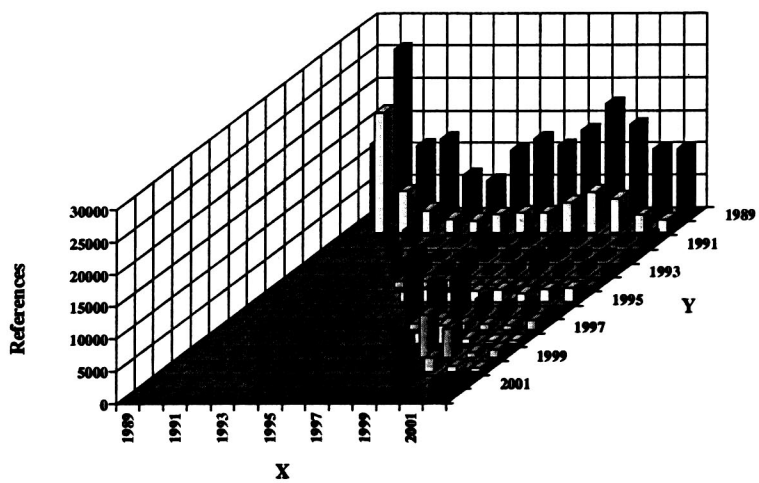

## The Size of Contribution by Authors



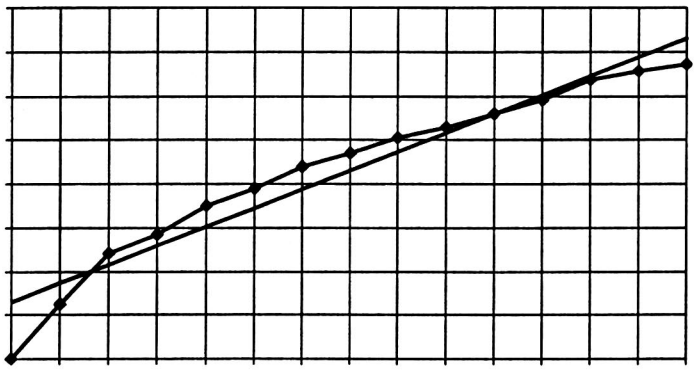
# The Most Active Authors

| | | | |
|---|---|---|---|
| 1. G. Bancerek | 1038979,67 | 12. C. Schwarzweller | 167909,00 |
| 2. Y. Nakamura | 824414,08 | 13. M. Muzalewski | 167303,83 |
| 3. A. Trybulec | 809791,25 | 14. Z. Karno | 146470,00 |
| 4. A. Kornilowicz | 501380,50 | 15. J. Bialas | 136155,50 |
| 5. P. Rudnicki | 416192,58 | 16. A. Naumowicz | 127116,00 |
| 6. R. Milewski | 415386,50 | 17. N. Asamoto | 119951,58 |
| 7. C. Bylinski | 361727,17 | 18. K. Prazmowski | 116099,75 |
| 8. J. Chen | 346127,00 | 19. S. Kobayashi | 111855,50 |
| 9. W. Trybulec | 284702,00 | 20. A. Darmochwal | 108777,33 |
| 10. J. Kotowicz | 280174,33 | 21. Y. Shidama | 103691,33 |
| 11. A. Grabowski | 219682,33 | 22. N. Endou | 95713,00 |

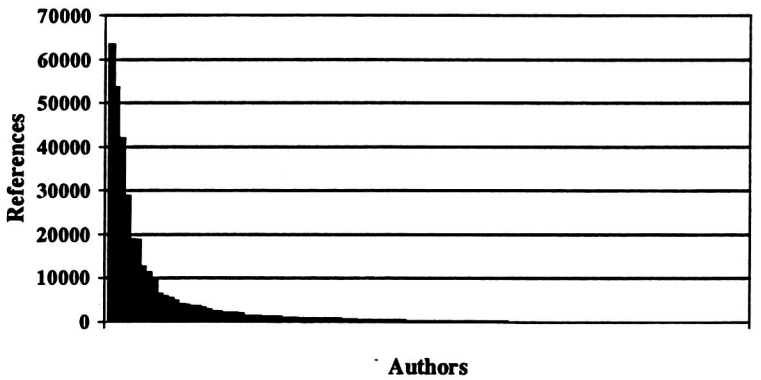
# The Distribution of References



## The Number of the Authors



## The Number of References to the Authors



3 Jacques Calmet
*University of Karlsruhe, Germany*

## Course: Introduction to Computer Algebra Systems

# Introduction to Computer Algebra Systems

Jacques Calmet

*University of Karlsruhe*

## A – Topics

1. Generalized versus specialized CAS,
2. Data structures, normal and canonical forms,
3. Simplification,
4. Basic concepts in algebraic algorithm design.

## B- Literature

Topic 1

Michael J. Wester (ed.) „Computer Algebra Systems - A Practical Guide", John Wiley & Sons (1999).

H. van Hulzen and J. Calmet in „Computer Algebra". Eds B. Buchberger, G.E. Collins and R. Loos, Springer Verlag, 2nd edition (1983).

Topic 2 and 3

B. F. Caviness, „On Canonical Forms and Simplification". PhD thesis, Carnegie-Mellon University, May 1968.

R.J. Fateman, „Essays in Algebraic Simplification". PhD thesis, MIT, Project Mac, April 1972.

Topic 4

J.D. Lipson, „Elements of Algebra and Algebraic Computing". Benjamin/Cummings (1981).

K.O. Geddes, S.R. Czapor and G. Labahn, Algorithms for Computer Algebra), Kluwer Academic Publication (1992).

Many other books are now available. Some authors are: J. von zur Gathen, J.H. Davenport et al. (the first published book on Computer Algebra, R. Zippel, A. Cohen, M. Mignotte, A.G. Akritas), G. Gonnet.

Jacques Calmet, Computer Algebra script (in German) at:
http://iaks-www.ira.uka.de/iaks-calmet/vorlesungen/compalg00/ca96-1.ps.gz

3 Arjeh Cohen
*Technical University Eindhoven, Netherlands*

Course: Integrating Proof and Group Theory

Evening Talk: Mathematical Services on the Web

# An automated proof theory approach to computation with permutation groups

Arjeh M. Cohen
Department of Mathematics
Eindhoven University of Technology
Netherlands

Scott H. Murray
School of Mathematics and Statistics
University of Sydney
Australia

August 28, 2002

**Abstract**

This is an introduction to data structures for permutation groups with a proof theoretic flavour.

## 1 Introduction

In this paper, we introduce the reader to computational permutation group theory. We describe the basic concepts and first results in this area of mathematics, as well as the data structures required to do actual computations. We follow [1], but our aim is to apply the theory to the graph isomorphism problem.

In the spirit of the Calculemus Autumn School, our approach will be proof theoretic. Computer algebra has always had an emphasis on complexity of algorithms, so that bigger and bigger problems could be solved on a given machine. The internet will play an increasingly large role in the exchange of mathematics between people, and we believe this will require a different approach to computational mathematics. As the exchange of mathematics across the World Wide Web becomes easier than solving all problems locally, the management of mathematical queries becomes more



prominent. The problem of verifying the correctness of computations is particularly acute when they are no longer done on local machines with software the user trusts.

In this paper, our *queries* are invocations of permutation group algorithms that have been developed over the years and are implemented as part of the GAP computer algebra package. The *response* to a query is the output of the algorithm, which may have been run on a remote computer which the user knows nothing about. The user has reason to doubt the validity of the response, and so will demand some kind of *verification*. Since our queries are mathematical in nature, this verification should take the form of (an encoding of) a proof.

A classical example is the factorization of a natural number. If a sequence $p_1, p_2, \ldots, p_t$ of numbers is returned as a response to the query "factor the natural number $n$," it is easy for the user to verify whether $n = p_1 \cdots p_t$. In order to verify that each $p_i$ is a prime number, it would be very useful to receive additional data, such as the primality witnesses for each $p_i$. This example has been worked out by Olga Caprotti, Martijn Oostdijk and the first author [2].

We treat computational permutation group theory in a similar manner. We give additional data that allows for a relatively easy check of the correctness of the answer. Of course this requirement may prevent us from using the most efficient possible algorithms and implementations. In general, we use functions in the computer algebra system GAP, which are close to the state of the art. However, on occasion we have been forced to implement simpler methods that allow us to provide the data for a straight forward verification of the result.

This paper will be concerned with providing human readable proofs that could be transformed to a computer checkable proof without too much effort. In this way, we contribute to the integration of computer algebra and proof verification, which is the research focus of the Calculemus project.

## 2 Membership: Prove that the permutation $g$ belongs to the group $G$

In computational permutation group theory, a group $G$ is specified by a set of generating permutations $A$. Suppose that

$$A = \{a_1, a_2, \ldots, a_k\}$$

consists of permutations of the points $\{1,2,\ldots,n\}$, i.e. $A$ is a subset of the symmetric group $\mathrm{Sym}_n$. Hence, by the definition of a generating set, $G$ is the unique smallest subgroup of $\mathrm{Sym}_n$ which contains $A$.

We define a *word* in $A$ to be an expression of the form

$$a_{i_1}^{e_1} a_{i_2}^{e_2} \cdots a_{i_m}^{e_m}$$

where the indices $i_j$ are in the range $1,\ldots,k$ and the exponents $e_j$ are integers. It is now easily shown that the set of words in $A$ form a subgroup of $\mathrm{Sym}_n$, and it is obvious that there is no smaller subgroup containing $A$. Hence a permutation $g \in \mathrm{Sym}_n$ is an element of $G$ if, and only if, it can be expressed as a word in $A$. (A note for those who know combinatorial group theory: since we make little use of words, we are not making the normal distinction between a word in the free group, and its evaluation in the symmetric group).

Writing an arbitrary permutation $g$ as a word in $A$ is a difficult computational problem, which is beyond the scope of this tutorial. Instead we just use the existing methods implemented in GAP without explaining how they work. See Section 8, for a technique to show that a permutation is *not* an element of $G$.

**Example 2.1** *The Mathieu group on* 11 *points,* $M_{11}$*, has generating set* $A = \{a_1, a_2\}$*, where:*

$$\begin{aligned} a_1 &= (1,10)(2,8)(3,11)(5,7), \\ a_2 &= (1,4,7,6)(2,11,10,9). \end{aligned}$$

**Query input**

- A permutation $g$.
- A list $A$ of permutations generating $G$.
- We are given the fact that $g \in \langle A \rangle$.

**GAP input**

```
G := Group(A);
IsIn_Proof(g,G)
```

**GAP output** A word in $A$ that is equal to $g$.

**Query output** By definition, $G$ is generated by $A$ and so $G$ consists of those elements which can be expressed as a word in $A$. In particular $g =$ `IsIn_Proof`$(g,G)$, and so belongs to $G$.



## 3 Subgroup: Prove that $H$ is a subgroup of $G$

Suppose $H$ is another permutation group with generating set $B$. From the definition of a generating set it follows that $H$ is a subgroup of $G$ if, and only if, every element of $B$ is contained in $G$.

**Query input**

- A list $A$ of permutations which generate $G$.
- A list $B$ of permutations which generate $H$.
- We are given the fact that $H$ is a subgroup of $G$.

**GAP input**

```
G := Group(A);
H := Group(B);
IsSubgroup_Proof(H,G);
```

**GAP output** A list of words in $A$ indexed by $B$.

**Query output** In order to show that $H$ is a subgroup of $G$, it suffices to show that each element of the generating set $B$ belongs to $G$. `IsSubgroup_Proof`$(H,G)$ presents, for each element $b$ of $B$, how it can be expressed as a word in $A$. This establishes that each element of $B$ belongs to $G$, and so $H$ is a subgroup of $G$.

## 4 The orbit $X = Gx$ with proof.

The concept of an orbit is used in the construction of sets of coset representatives for stabilisers. Let $G$ be a permutation group on $\{1,\ldots,n\}$. The *orbit* of $x$ under the action of $G$ is

$$xG = \{xg : g \in G\}.$$

Suppose $G$ has generating set $A = \{a_1, a_2, \ldots, a_k\}$ and let $X$ be an orbit of $G$. Then the orbit graph $\mathcal{G}$ has vertex set $X$, label set $\{1,2,\ldots,k\}$ and labeled edges

$$y \xrightarrow{i} z \text{ where } ya_i = z.$$

**Example 4.1** *Consider* $M_{11} = \langle s_1, s_2 \rangle$*, where* $s_1$ *and* $s_2$ *are defined in Example 2.1. The action of* $\{s_1, s_2\}$ *on the orbit* $X = \{1,2,\ldots,11\}$ *is shown in Figure 1, where the dotted lines are labeled by 1 and solid lines by 2.*

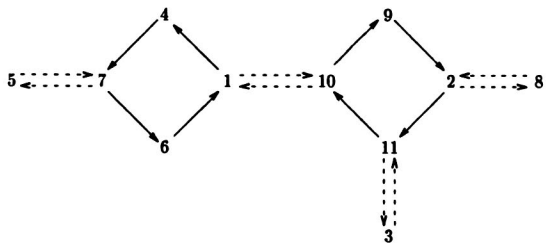



Figure 1: Orbit graph of $M_{11}$

**Algorithm 4.1** *Calculate orbit*

```
(* input:  x ∈ Ω, generating set A = {a_1, a_2, ..., a_k}.
   output: X = xG. *)
begin
    let X = {x};
    for y ∈ X, for a_j ∈ A do
        let y = ya_j;
        if y ∉ X then
            add y to X;
        end if;
    end for;
    return X;
end
```

This algorithm terminates after it has applied every generator to every point in $X$.

### Query Input

- A list $A$ of permutations generating $G$.
- A point $x$.

### GAP input

```
G := Group(A);
Orbit_Proof(G,x);
```



**GAP output** A set of points $X$ and a list $B$ of words in $A$ indexed by $X$. The set $X$ is just the orbit $Gx$. The word in $A$ corresponding to $y \in X$ maps $x$ to $y$.

**Query output** In order to show that $X$ is the $G$-orbit of $x$, we need to show two statements:

1. Each element of $X$ is image of $x$ under an element of $G$. These elements are produced by the table `Orbit_Proof(Group(`$A$`),`$x$`)`.
2. Each element of $A$ leaves the set $X$ invariant. This is a straightforward check that the cycles containing points of $X$ do not contain any points not in $X$.

## 5 The Schreier tree for the orbit $X = xG$.

Stabiliser subgroups are of fundamental importance to both theoretical and computational permutation group theory. The *stabiliser subgroup* in $G$ of $x$ is

$$G_x = \{g \in G : xg = x\}.$$

It is not immediately clear how to compute with this subgroup, since the definition gives us a test for whether $g$ is an element of $G_x$, but does not give us for example a generating set.

The following lemma gives us a one-to-one correspondence between the orbit of a point and the set of cosets of its stabiliser.

**Lemma 5.1 (Orbit-stabiliser lemma)** *If $y \in xG$, then $\{g \in G : xg = y\}$ is a coset of $G_x$. In particular, $|xG| = |G : G_x|$.*

**Proof:** Choose $h \in G$ such that $xh = y$; then

$$\begin{aligned} \{g \in G : x^g = y\} &= \{g \in G : xg = xh\} \\ &= \{g \in G : xgh^{-1} = x\} \\ &= \{g \in G : gh^{-1} \in G_x\} = G_x h. \end{aligned}$$

Hence there is a one-to-one correspondence between the orbit and the cosets, and the second result follows.

Suppose that for every element $y$ of the orbit $xG$, we choose $u_y \in G$ with the property that $xu_y = y$. Then it follows immediately from the orbit-stabiliser lemma that all such $u_y$ form a set of coset representatives for $G_x$ in $G$.

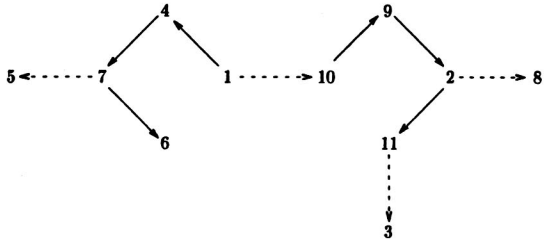



Figure 2: Schreier tree $\mathcal{T}$ for $M_{11}$

It would be inefficient to store all the elements $u_y$, so instead we construct of *Schreier tree.* That is, a subgraph $\mathcal{T}$ of the orbit graph $\mathcal{G}$ which contains, for every $y \in X$, a unique path in $\mathcal{T}$ from $x$ to $y$.

**Example 5.1** *Consider $M_{11}$ with the generating set of Example 2.1. Then $\mathcal{G}$ is the graph shown in Figure 1, where dotted lines are labeled by 1 and solid lines by 2. A Schreier tree $\mathcal{T}$ rooted at 1 is shown in Figure 2.*

In practice, we store this tree in a linearised form, using two vectors $v : X \to \{-m, \ldots, -1, 0, 1, \ldots, m\}$ and $\omega : V \to X \cup \{0\}$ defined by:

$$v(z) = \begin{cases} l & \text{if } y \xrightarrow{l} z \text{ is in } \mathcal{T} \\ -l & \text{if } y \xleftarrow{l} z \text{ is in } \mathcal{T} \\ 0 & \text{if } z = x \end{cases},$$

$$\omega(z) = \begin{cases} y & \text{if } y \xrightarrow{l} z \text{ is in } \mathcal{T} \\ 0 & \text{if } y = x \end{cases}.$$

We call $v$ the *Schreier vector* and $\omega$ the *backpointers.* They can be computed by a modified version of the orbit algorithm.

**Algorithm 5.1** *Calculate orbit*

(* input: $x \in \Omega$, generating set $A = \{a_1, a_2, \ldots, a_k\}$.
output: $X = xG$, Schreier vector $v$, vector of backward pointers $\omega$. *)
**begin**



**let** $X = \{x\}$;
**let** $v(y) = \omega(y) = 0$ **for all** $y$;
**for** $x \in X$, **for** $a_j \in A$ **do**
  **let** $y = xa_j$;
  **if** $y \notin X$ **then**
    **add** $y$ **to** $X$;
    **let** $v(y) = j$, $\omega(y) = x$;
  **end if**;
**end for**;
**return** $X$, $v$, $\omega$;
**end**

**Example 5.2** *The linearised version of the Schreier tree in Figure 2 is given in the following table*

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $v$ | 0 | 2 | 1 | 2 | 1 | 2 | 2 | 1 | 2 | 1 | 2 |
| $\omega$ | 0 | 9 | 11 | 1 | 7 | 7 | 4 | 2 | 10 | 1 | 2 |

**Query Input**

- A list $A$ of permutations generating $G$.
- A point $x$.

**GAP input**

```
G := Group(A);
SchreierData(G,x);
```

**GAP output** A triple $[X, v, \omega]$ of integers sequences consisting of the orbit, the Schreier vector, and the backpointers.

**Query output** Consider the table `SchreierData`$(G, x)$. It has three rows, the first of which represents the orbit $X = Gx$. In order to show that $X$ is indeed the $G$-orbit containing $x$, see Section 4. To show that $v$ is a Schreier vector and $\omega$ is backpointers, consider a column of the table, say $x_j, v_j, \omega_j$. It suffices to show that $x_j a_{v_j} = \omega_j$ if $v_j > 0$ or that $x_j x_{v_j}^{-1} = \omega_j$ if $v_j < 0$. This is a (perhaps tedious but) trivial check.

## 6 The stabiliser $G_x$

Now that we have a set of coset representatives for $G_x$, we can use it to compute a generating set, with the following lemma.

**Lemma 6.1 (Schreier's lemma)** *Suppose $G$ is a group with generating set $A$, and $H$ is a subgroup of $G$. If $U$ is a set of coset representatives for $H$ in $G$, and the function $t : G \to U$ maps an element $g$ of $G$ to the representative of $Hg$, then a generating set for $H$ is given by*

$$\{ua\,t(ua)^{-1} : u \in U, a \in A\}.$$

**Proof:**
Every element $h$ of $H$ can be written in the form $b_1 b_2 \cdots b_l$, where each $b_i$, or its inverse, is in $A$. Let $u_i = t(b_1 b_2 s\, b_i)$, for $i = 0, 1, \ldots, l$. Then $u_0 = t(1) = 1$ and $u_l = t(h) = 1$, so

$$h = u_0 h u_l^{-1} = (u_0 b_1 u_1^{-1})(u_1 b_2 u_2^{-1}) \cdots (u_{l-1} b_l u_l^{-1}).$$

Consider $u_{i-1} b_i u_i$, for $i = 1, 2, \ldots, l$. Now $u_i = t(b_1 b_2 s\, b_i) = t(u_{i-1} b_i)$, since $H b_1 b_2 \cdots b_i = H u_{i-1} b_i$. Let $u = u_{i-1} \in U$ and $b = b_i$; we can now write

$$u_{i-1} b_i u_i^{-1} = ub\,t(ub)^{-1}.$$

This has the desired form if $b \in A$; otherwise, $b = a^{-1}$ for some $a \in A$ and let $v = t(ua^{-1}) \in U$. Since $Hva = Hua^{-1}a$, we have $t(va) = u$, and so the inverse of $ub\,t(ub)^{-1}$ can be written

$$t(ua^{-1})\,au^{-1} = va\,t(va)^{-1},$$

which has the desired form. The result now follows.

**Query Input**

- A list $A$ of permutations generating $G$.
- A point $x$.

**GAP input**

```
G := Group(A);
Stabiliser_Proof(G,x);
```

**GAP output**



- A triple $[X, v, \omega]$ of integers sequences consisting of the orbit, the Schreier vector, and the backpointers.
- A sequence of triples $(y, i, g)$ consisting of a point $y \in X$, an index $i = 1, \ldots, m$, and the Schreier generator $g = t(y)a_i t(t(y)a_i)^{-1}$.

**Query output** The proof that $[X, v, \omega]$ describes a Schreier tree is given in (Section 5). Checking the correctness of the Schreier generators is straightforward.

## 7 Stabilizer chain

Now that we have a stabiliser of a subgroup, we can repeat the process to form a chain of subgroups. A *base* for $G$ is a finite sequence $B = [x_1, \ldots, x_l]$ of distinct points in $\Omega$ such that

$$G_{x_1, x_2, \ldots, x_k} = 1.$$

Hence, the only element of $G$ which fixes all of the points $x_1, x_2, \ldots, x_k$ is the identity. Clearly every permutation group has a base, but not all bases for a given group are of the same length. If we write $G^{(i)} = G_{x_1, x_2, \ldots, x_{i-1}}$, then we have a *chain of stabilisers*

$$G = G^{(1)} \geq G^{(2)} \geq \cdots \geq G^{(k)} \geq G^{(k+1)} = 1.$$

We often require that a base has the additional property that $G^{(i)} \neq G^{(i+1)}$. A *strong generating set* for $G$ with respect to $B$ is a set $S$ of group elements such that, for $i = 1, 2, \ldots, k$,

$$G^{(i)} = \left\langle S \cap G^{(i)} \right\rangle.$$

Note that $S \cap G^{(i)}$ is just the set of elements in $S$ which fix $\beta_1, \beta_2, \ldots, \beta_{i-1}$.

A base can be constructed by starting with $B = [x_1]$, and recursively choosing a letter $x_i$ in a nontrivial $G_{x_1, \ldots, x_{i-1}}$-orbit and appending it to $B$. The construction is finished when $G_{x_1, \ldots, x_i} = 1$.

**Example 7.1** *The Mathieu group on* 11 *points, $M_{11}$, has a base* $[1, 2, 3, 4]$

*and strong generating set* $\{s_1, s_2, \ldots, s_7\}$, *where:*

$$\begin{aligned} s_1 &= (1,10)(2,8)(3,11)(5,7), \\ s_2 &= (1,4,7,6)(2,11,10,9), \\ s_3 &= (2,3)(4,5)(6,11)(8,9), \\ s_4 &= (3,5,7,9)(4,8,11,6), \\ s_5 &= (4,6)(5,11)(7,10)(8,9), \\ s_6 &= (4,10,6,7)(5,9,11,8), \\ s_7 &= (4,11,6,5)(7,8,10,9). \end{aligned}$$

*Note that* $s_1$ *and* $s_2$ *suffice to generate* $M_{11}$.

*The Schreier trees and backpointers for this stabiliser chain are given in the following table*

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $v_1$ | 0 | 2 | 1 | 2 | 1 | 2 | 2 | 1 | 2 | 1 | 2 |
| $\omega_1$ | 0 | 9 | 11 | 1 | 7 | 7 | 4 | 2 | 10 | 1 | 2 |
| $v_2$ | - | 0 | 3 | 7 | 4 | 7 | 4 | 6 | 4 | 6 | 5 |
| $\omega_2$ | - | 0 | 2 | 5 | 3 | 11 | 5 | 11 | 7 | 4 | 5 |
| $v_3$ | - | - | 0 | 7 | 4 | 7 | 4 | 6 | 4 | 6 | 5 |
| $\omega_3$ | - | - | 0 | 5 | 3 | 11 | 5 | 11 | 7 | 4 | 5 |
| $v_4$ | - | - | - | 0 | 7 | 5 | 6 | 6 | 7 | 6 | 7 |
| $\omega_4$ | - | - | - | 0 | 6 | 4 | 6 | 11 | 10 | 4 | 4 |

**Query Input** A list $A$ of permutations generating $G$.

**GAP input** A group $G$ with generating set $A$.

**GAP input**

```
G := Group(A);
StabiliserChain_Proof(G,x);
```

**GAP output** A base $B$ together with strong generators, and the corresponding Schreier trees and Schreier generators for each base point.

**Query output** This is simply the proof of Section 6 repeated inductively.

## 8 Nonmembership: Prove that the permutation $g$ does not belong to $G$

We now have a chain of subgroups

$$G = G^{(1)} \geq G^{(2)} \geq \cdots \geq G^{(k)} \geq G^{(k+1)} = 1.$$



and sets $U^{(i)}$ consisting of coset representatives for $G^{(i+1)}$ in $G^{(i)}$. An element $g$ of $G$ is contained in exactly one coset of $G^{(2)}$ in $G^{(1)}$, so $g = h_2 \cdot u_1$ for some unique $h_2$ in $G^{(2)}$ and $u_1$ in $U^{(1)}$. By induction, we can show that

$$g = u_k \cdot u_{k-1} \cdot \cdots \cdot u_1$$

where each $u_i \in U^{(i)}$ is uniquely determined by $g$. This process, called *sifting* an element, gives a canonical form for the elements of $G$ and underpins most of the more advanced applications of stabiliser chains.

On the other hand, if $g$ is not in $G$, then sifting fails because at some stage we get that $x_i h_i$ is not in the orbit $x_i G^{(i)}$, and so $h_i$ is not in $G^{(i)}$. This gives us our proof on nonmembership.

**Query Input**

- A list $A$ of permutations generating $G$.
- A permutation $g$.
- The fact that $g$ is not in $G$.

**GAP input**

```
G := Group(A);
IsNotIn_Proof ( g, G )
```

**GAP output** A base $B$ together with generators for the stabiliser subgroups, and the corresponding Schreier trees and Schreier generators for each base point. A sequence of permutations $[h_1, \ldots, h_j]$ such that $h_i = g$; $h_i = h_{i+1} u_i$ for $u_i$ in the $i$th set of coset representatives; and $x_j h_j$ is not in the $i$th orbit.

**Query output** A proof that the stabiliser chain is correct is given in Section 7. It is now easy to check that the permutations $h_i$ have the properties claimed.

## 9 The order of $G$

The order of $G$ follows directly from the stabilizer chain

$$\begin{aligned} |G| &= |G_{x_1}| \cdot |x_1 G| \\ &= |G_{x_1}| \cdot |G_{x_2}| \cdot |x_2 G| \\ &= \cdots \\ &= \prod_{i=1}^{t} |x_i G_{\{x_1,\ldots,x_{i-1}\}}| \end{aligned}$$



-45-

where $B = [x_1, \ldots, x_t]$ is a base.

**Example 9.1** *From Example 7.1 it immediately follows that the Mathieu group on* 11 *points has order* $11 \cdot 10 \cdot 9 \cdot 8 = 7920$.

**Query Input** A list $A$ of permutations generating $G$.

**GAP input**

```
G := Group(A);
Order_Proof(G);
```

**GAP output** A stabiliser chain, and the product of the sizes of the orbits.

**Query output** The proof that the stabiliser chain is correct is given in Section 7.

## References


[1] Chapter 8: Working with finite groups of "Some tapas of computer algebra" by Cuypers, Soicher, and Sterk. Springer, 1999.

[2] Caprotti & Oostdijk: Pocklington.

4 Herman Geuvers
*Nijmegen University, Netherlands*

**Course: Type Theory as a Basis for Theorem Proving and Representing Mathematics**

Calculemus Summer School

Type Theory for Formalizing Maths

Herman Geuvers
Nijmegen University, NL



(sort) $\vdash s_1 : s_2$ if $(s_1, s_2) \in \mathcal{A}$ (var) $\dfrac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A}$ if $x \notin \Gamma$

(weak) $\dfrac{\Gamma \vdash A : s \quad \Gamma \vdash M : C}{\Gamma, x{:}A \vdash M : C}$ if $x \notin \Gamma$

($\Pi$) $\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_3}$ if $(s_1, s_2, s_3) \in \mathcal{R}$

($\lambda$) $\dfrac{\Gamma, x{:}A \vdash M : B \quad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}$

(app) $\dfrac{\Gamma \vdash M : \Pi x{:}A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}$

(conv$_\beta$) $\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B}\ A =_\beta B$

Pure Type Systems
Determined by a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ with

- $\mathcal{S}$ the set of sorts
- $\mathcal{A}$ the set of axioms, $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$
- $\mathcal{R}$ the set of rules, $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$

If $s_2 = s_3$ in $(s_1, s_2, s_3) \in \mathcal{R}$, we write $(s_1, s_2) \in \mathcal{R}$.

Pseudoterms:

$$\mathsf{T} ::= \mathcal{S} \mid \mathsf{Var} \mid (\Pi \mathsf{Var{:}T.T}) \mid (\lambda \mathsf{Var{:}T.T}) \mid \mathsf{TT}.$$

Examples of PTSs

| $CC$ |
|---|
| $\mathcal{S}$ Prop, Type |
| $\mathcal{A}$ Prop : Type |
| $\mathcal{R}$ (Prop, Prop), (Prop, Type), (Type, Prop), (Type, Type) |

| $CC^\infty$ |
|---|
| $\mathcal{S}$ Prop, $\{\mathsf{Type}_i\}_{i \in \mathbb{N}}$ |
| $\mathcal{A}$ Prop : Type, $\mathsf{Type}_i$ : $\mathsf{Type}_{i+1}$ |
| $\mathcal{R}$ (Prop, Prop), (Prop, $\mathsf{Type}_i$), ($\mathsf{Type}_i$, Prop) ($\mathsf{Type}_i$, $\mathsf{Type}_j$, $\mathsf{Type}_{\max(i,j)}$) |

Recall that ($\mathsf{Type}_1 \mathsf{Type}_0$, $\mathsf{Type}_0$) is inconsistent ($\lambda U$)
Similarly ($\mathsf{Type}_{i+1} \mathsf{Type}_i$, $\mathsf{Type}_i$)) would be inconsistent.

The Extended Calculus of Constructions has in addition

- Cumulativity: $\mathsf{Prop} \subseteq \mathsf{Type}_0 \subseteq \mathsf{Type}_1 \subseteq \ldots$, so

$$\frac{\Gamma \vdash A : \mathsf{Prop}}{\Gamma \vdash A : \mathsf{Type}_0} \quad \frac{\Gamma \vdash A : \mathsf{Type}_i}{\Gamma \vdash A : \mathsf{Type}_{i+1}}$$

- $\Sigma$-types:

$$\frac{\Gamma \vdash A : \mathsf{Prop} \;\; \Gamma, x{:}A \vdash B : \mathsf{Prop}}{\Gamma \vdash \Sigma x{:}A.B : \mathsf{Prop}} \quad \frac{\Gamma \vdash A : \mathsf{Type}_i \;\; \Gamma, x{:}A \vdash B : \mathsf{Type}_j}{\Gamma \vdash \Sigma x{:}A.B : \mathsf{Type}_{\max(i,j)}}$$

For $\varphi$ : Prop

- We have $\Pi A{:}\mathsf{Type}_i.\varphi$ : Prop, but
- We do not have $\Sigma A{:}\mathsf{Type}_i.\varphi$ : Prop.

Note: The type theory of Coq has in addition Set : Type and rules (Set, Set), ($\mathsf{Type}_i$, Set), (Set, Prop).





The type of groups over $A$, Group($A$), is

$$\begin{aligned}\mathsf{Group}(A) := \Sigma\, \circ{:}A{\to}A{\to}A.\Sigma e{:}A.\Sigma \mathsf{inv}{:}A{\to}A.\\ (\Pi x, y, z{:}A.(x \circ y) \circ z = x \circ (y \circ z)) \wedge\\ (\Pi x{:}A.e \circ x = x) \wedge\\ (\Pi x{:}A.(\mathsf{inv}\ x) \circ x = e).\end{aligned}$$

If $t$ : Group($A$), we can extract the elements of the group structure by projections: $\pi_1 t : A{\to}A{\to}A$, $\pi_1(\pi_2 t) : A$
If $f : A{\to}A{\to}A$, $a : A$ and $h : A{\to}A$ with $p_1, p_2$ and $p_3$ proof-terms of the associated group-axioms, then

$$\langle f, \langle a, \langle h, \langle p_1, \langle p_2, p_3\rangle\rangle\rangle\rangle\rangle : \mathsf{Group}(A).$$



Use $\Sigma$-types for mathematical structures:
theory of groups: Given $A$ : Type, a group over $A$ is a tuple consisting of

$$\begin{aligned}\circ &: A{\to}A{\to}A\\ e &: A\\ \mathsf{inv} &: A{\to}A\end{aligned}$$

such that the following types are inhabited.

$$\begin{aligned}\Pi x, y, z{:}A.(x \circ y) \circ z &= x \circ (y \circ z),\\ \Pi x{:}A.e \circ x &= x,\\ \Pi x{:}A.(\mathsf{inv}\ x) \circ x &= e.\end{aligned}$$

Type of group-structures over $A$, Group-Str($A$), is

$$(A{\to}A{\to}A) \times (A \times (A{\to}A))$$



We would like to use names for the projections:
Coq has labelled record types (type dependent)

- `Record My_type : Set :=`
  `{ l_1 : type_1 ;`
  `l_2 : type_2 ;`
  `l_3 : type_3 }.`
  If `X : My_type`, then `(l_1 X) : type_1`.
- Also with dependent types: `l_1` may occur in `type_2`.
  If `X : My_type`, then

  `(l_2 X) : type_2 [(l_1 X)/l_1]`

```
• Record Group : Type :=
   { crr   :  Set;
     op    :  crr -> crr -> crr;
     unit  :  crr;
     inv   :  crr -> crr;
     assoc :  (x,y,z:crr)
              (op (op x y) z) = (op x (op y z))
     ...      ...
    }.
```

If `X : Group`, then `(op X) : (crr X) -> (crr X) -> (crr X)`.

The record types can be defined in Coq using inductive types.
Note: `Group` is in `Type` and not in `Set`



Allowing

$$\frac{\Gamma \vdash A : \text{Type}\ \ \Gamma, x{:}A \vdash B : \text{Prop}}{\Gamma \vdash \Sigma x{:}A.B : \text{Prop}}$$

leads to inconsistency:

- Define $\Omega := \Sigma A{:}\text{Set}.\Sigma R{:}A{\to}A{\to}\text{Prop}.\text{wf}(R)$
  $\text{wf}(R)$ denotes that $R$ is well-founded.
- Define $<$ on $\Omega$ by

  $(A,R) < (B,Q) :=$ $R$ can be embedded into $Q$ under some $b : B$

  Then

  $-<$ is well-founded on $\Omega$

  – If $(A,R)$ well-founded, then $(A,R) < (\Omega,<)$

  so contradiction: $\ldots < (\Omega,<) < (\Omega,<) < (\Omega,<)$.



Functions and Algorithms

- Set theory (and logic): a function $f : A{\to}B$ is a relation $R \subset A \times B$ such that $\forall x{:}A.\exists! y{:}B.R\,x\,y$.
  "functions as graphs"
- In Type theory, we have functions-as-graphs, but also functions-as-algorithms: $f : A{\to}B$.

Functions as algorithms also compute: $\beta$ and $\iota$ rules:

$$(\lambda x{:}A.M)N \longrightarrow_\beta M[N/x],$$
$$\text{Rec}\,b\,f\,0 \longrightarrow \iota\ b,$$
$$\text{Rec}\,b\,f\,(S\,x) \longrightarrow \iota\ f\,x\,(\text{Rec}\,b\,f\,x).$$

Terms of type $A{\to}B$ denote algorithms, whose operational semantics is given by the reduction rules.
Type theory can be seen as a small programming language esp. if we have inductive types (with primitive recursion)



Poincaré Principle

An equality involving a computation does not require a proof.

In type theory: if $t = q$ by evaluation (computing an algorithm), then this is a trivial equality, proved by reflexivity.
This is made precise by the conversion rule:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : B}\ A =_\beta B$$

Can we actually use the programming power of CIC when formalizing mathematics?

Yes. For automation: replacing a proof obligation by a computation

Reflection Suppose

- We have a class of problems with a syntactic encoding as a data type, say via the type Problem.
  Example: equalities between expressions over a group
  Then the syntactic encoding is

```
Inductive E : Set :=
   evar  : nat -> E
 | eone  : E
 | eop   : E -> E -> E
 | einv  : E -> E
```

- We have a decoding function $[\![-]\!]$ : Problem $\to$ Prop
- We have a decision function Dec : Problem $\to \{0, 1\}$
- We can prove Ok : $\forall p$:Problem$((\text{Dec}(p) = 1) \to [\![p]\!])$



To verify $P$ (from the class of problems):

- Find a $p$ : Problem such that $[\![p]\!] = P$.
- Then Dec$(p)$ yields either $1$ or $0$
- If Dec$(p) = 1$), then we have a proof of $P$ (using Ok)
- If Dec$(p) = 0$, we obtain no information about $P$ (it 'fails')

Note: if Dec is complete:

$$\forall p\text{:Problem}((\text{Dec}(p) = 1) \leftrightarrow [\![p]\!])$$

then Dec$(p) = 0$ yields a proof of $\neg P$.



Implicit Syntax and Coercions

- Implicit Syntax: If the type checker can infer some arguments, we can leave them away:
  Write `(f ? a b)` in stead of `(f S a b)` if `f : (S:Set)S->S->S`
  Also: define `F:=(f ?)` and write `(F a b)`.
- Coercions: The user can tell the type checker to use specific terms as coercions.
  `Coercion k : A >-> B` declares the term `k : A -> B` as a coercion.
  - If `f a` can not be typed, the type checker will try to type check `(k f) a` and `f (k a)`.
  - If we declare a variable `x : A` and `A` is not a type, the type checker will check if `(k A)` is a type.

  Coercions can be composed.



Coercions and structures

```
Record CMonoid : Type :=
  { m_crr    :> CSemi_grp;
    m_proof :  (Commutative m_crr (sg_op m_crr))
       /\ (IsUnit m_crr (sg_unit m_crr) (sg_op m_crr))
  }.
```

- A monoid is now a tuple $\langle\langle\langle S, =_S, r\rangle, a, f, p\rangle, q\rangle$
  If `M : Monoid`, the carrier of `M` is `(crr(sg_crr(m_crr M)))`
  Nasty !!
  $\Rightarrow$ We want to use the structure `M` as synonym for the carrier set `(crr(sg_crr(m_crr M)))`.
  $\Rightarrow$ The maps `crr`, `sg_crr`, `m_crr` should be left implicit.
- The notation `m_crr :> Semi_grp` declares the coercion `m_crr : Monoid >-> Semi_grp`.

Setoids
How to represent the notion of set?
Note: A set is not just a type, because
$M : A$ is decidable whereas $t \in X$ is undecidable

A setoid is a pair $[A, =]$ with

- $A$ : Set,
- $= : A \to (A \to \text{Prop})$ an equivalence relation over $A$

Function space setoid (the setoid of setoid functions)
$[A \xrightarrow{s} B, =_{A \xrightarrow{s} B}]$ is defined by

$$A \xrightarrow{s} B := \Sigma f{:}A \to B.(\Pi x, y{:}A.(x =_A y) \to ((f\ x) =_B (f\ y))),$$
$$f =_{A \xrightarrow{s} B} g := \Pi x, y{:}A.(x =_A y) \to (\pi_1\ f\ x) =_B (\pi_1\ g\ y).$$



Two mathematical constructions: quotient and subset for setoids.

$Q$ is an equivalence relation over the setoid $[A, =_A]$ if

- $Q : A \to (A \to \text{Prop})$ is an equivalence relation,
- $=_A \subset Q$, i.e. $\forall x, y{:}A.(x =_A y) \to (Q\ x\ y)$.

The quotient setoid $[A, =_A]/Q$ is defined as

$$[A, Q]$$

Easy exercise:
If the setoid function $f : [A, =_A] \to [B, =_B]$ respects $Q$
(i.e. $\forall x, y{:}A.(Q\ x\ y) \to ((f\ x) =_B (f\ y))$)
it induces a setoid function from $[A, =_A]/Q$ to $[B, =_B]$.



Given $[A, =_A]$ and predicate $P$ on $A$ define the sub-setoid

$$[A, =_A] \mid P := [\Sigma x{:}A.(P\ x), =_A | P]$$

$=_A | P$ is $=_A$ restricted to $P$: for $q, r : \Sigma x{:}A.(P\ x)$,

$$q\ (=_A | P)\ r := (\pi_1\ q) =_A (\pi_1\ r)$$

NB We do not require the predicate $P$ to respect $=_A$: In taking a subsetoid we may remove elements from the $=$-classes.
Example (rational numbers): Let $A :=$ int $\times$ nat
We define, for $\langle x, p\rangle, \langle y, q\rangle{:}A$,

$$\langle x, p\rangle =_A \langle y, q\rangle := x(q+1) = y(p+1).$$

Take the predicate $P$ on $A$ defined by

$$Q\ \langle x, p\rangle := \gcd(x, p+1) = 1.$$

- The subsetoid $[A, =_A] | P$ is isomorphic to $[A, =_A]$
- All equivalence classes are reduced to a one element set



Objects depending on proofs
What should the type of the reciprocal?
given that the reciprocal of 0 is not defined.

- Let recip : $A \to A$ with the property
  $\forall x{:}A.x \neq 0 \to \text{mult}x(\text{recip}x) = 1$
- Either leave recip 0 undefined (Mizar) or make an arbitrary choice for it (HOL-light). But it should be undefined
- Type theoretic solution

$$\text{recip} : (\Sigma x{:}A.x \neq 0) \to A$$

- Then recip is only defined on the subset of elements that are non-zero:
  recip takes as input a pair $\langle a, p\rangle$ with $p : a \neq 0$ and returns $\text{recip}\langle a, p\rangle : A$.

- How to understand the dependency of this object (of type $A$) on the proof $p$?

Possible solution: setoids

- Take a setoid $[A, =_A]$ as the carrier of a field
- The operations on the field are taken to be setoid functions
- The field-properties are now denoted using the setoid equality. For the reciprocal:

  $$\text{recip} : [A, =_A] \mid (\lambda x{:}A.x \neq_A 0) \rightarrow [A, =_A],$$

  a setoid function from the subsetoid of non-zeros to $[A, =_A]$

Note recip still takes a pair of an object and a proof $\langle a, p\rangle$ and returns recip$\langle a, p\rangle : A$.
But recip is now a setoid function which implies

If $p : a \neq_A 0, q : a \neq_A 0$, then recip$\langle a, p\rangle =_A$ recip$\langle a, q\rangle$



- The value of recip$\langle a, p\rangle$ does not depend on the actual $p$
- One only has to ascertain that such a term exists.

Proof Irrelevance principle: for an object $t(p) : A$, with $p : \varphi$ a sub-term of $t$,

$$t(p) = t(q) \text{ for all } p, q : \varphi$$

The setoid equality obeys this principle.



1-53-

Intensionality versus Extensionality
The equality in the side condition in the conv rule is called the definitional equality
It can be intensional or extensional.

Extensional equality requires the following rules:

$$\text{(ext)}\quad \frac{\Gamma \vdash M, N : A{\rightarrow}B \quad \Gamma \vdash p : \Pi x{:}A.(Mx = Nx)}{\Gamma \vdash M = N : A{\rightarrow}B}$$

$$\text{(conv)}\quad \frac{\Gamma \vdash P : A \quad \Gamma \vdash A = B : s}{\Gamma \vdash P : B}$$

- Intensional equality of functions = equality of algorithms (the way the function is presented to us (syntax))
- Extensional equality of functions = equality of graphs (the (set-theoretic) meaning of the function (semantics))



Adding the rule (ext) renders TCP undecidable:

Suppose $H : (A{\rightarrow}B){\rightarrow}$Prop and $x : (H\ f)$; then

$$x : (H\ g) \text{ iff there is a } p : \Pi x{:}A.f\ x = g\ x$$

So, to solve this TCP, we need to solve a TIP.

The interactive theorem prover Nuprl is based on extensional type theory.

**Fundamental Theorem of Algebra:**

Every non-constant polynomial

$$f(z) = a_n z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0$$

with coefficients in $\mathbb{C}$ has a root
(a $z \in \mathbb{C}$ such that $f(z) = 0$).

- Overview / Motivation
- Constructive proof
- Axiomatic reasoning
- Completely formalized in the theorem prover Coq
- Some details of the proof
- Real numbers



1-54-1

Overview of the FTA project (Motivation)

- Formalize a large piece of real mathematics.
- Library for basic algebra and analysis, to be used by others
- Investigate the current limitations
- Try to manage this project. Three sequential/parallel phases:

| | |
|---|---|
| Mathematical proof | LaTeX document (lots of details) |
| Theory development | Coq file (just defs and statements of lemmas) |
| Proof development | Coq file (proofs filled in) |

Try to keep these phases consistent!

- Constructive proof: reals are (potentially) infinite objects; algorithm.



Constructive proof

- Procedure for (always) finding a root.
- Equality on $\mathbb{R}$ is not decidable.
  (Not: $\forall x, y \in \mathbb{R} \quad (x = y) \vee (x \neq y)$)
- Constructively, a polynomial

$$f(z) = a_n z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0$$

is non-constant if

$a_k \# 0$ for some $k > 0$.



Axiomatic reasoning

- First define an algebraic hierarchy
  (semi-groups, monoids, groups, rings, fields, ordered fields)
- Advantages: Reuse of proven results; reuse of notation
- $\mathbb{R}$ is a Cauchy-complete Archimedean ordered field.
- $\mathbb{C} := \mathbb{R} \times \mathbb{R}$.
- A concrete instantiation for $\mathbb{R}$ has been constructed (Niqui)

Completely formalized in the theorem prover Coq

| | |
|---|---|
| Sets and Basics | 41 kb |
| Algebra (upto Ordered Fields) | 165 kb |
| Reals | 52 kb |
| Polynomials | 113 kb |
| Real-valued functions / Basic Analysis | 30 kb |
| Complex numbers | 98 kb |
| FTA proof | 70 kb |
| Construction of $\mathbb{R}$ (Niqui) | 309 kb |
| Rational Tactic | 49 kb |

This is the size of our input files (definitions, lemmas, tactic scripts)



Some details of the proof

- Which proof to formalize?
- There are various constructive proofs of FTA: Weyl; Brouwer-de Loor; Bishop; Kneser (also in Troelstra and van Dalen).
- We have followed the Kneser proof.
- It can be seen as a constructive version of 'the' classical FTA proof.



The classical FTA proof

Suppose $|f(z)|$ is minimal with $|f(z)| \neq 0$.
We construct a $z_0$ with $|f(z_0)| < |f(z)|$.
We may assume that the minimum is reached for $z = 0$.

$$f(x) = a_0 + a_k x^k + O(x^{k+1})$$

with $a_k$ the first coefficient that's not 0.
Now take

$$z_0 := \epsilon \sqrt[k]{-\frac{a_0}{a_k}}$$

with $\epsilon \in \mathbb{R}_{>0}$.
If $\epsilon$ is small enough, the part $O\left(z_0^{k+1}\right)$ will be negligible and we get a $z_0 \neq 0$ for which

$$|f(z_0)| = a_0 + a_k \left(\epsilon \sqrt[k]{-\frac{a_0}{a_k}}\right)^k = a_0(1 - \epsilon^k) < |f(0)|$$



The constructive FTA proof

Define an algorithm
Given $z \in \mathbb{C}$, construct a sequence $z, z_0, z_1, \ldots$ going to the root.
Problem: in the definition

$$z_0 := \epsilon \sqrt[k]{-\frac{a_0}{a_k}}$$

- $\epsilon$ must be small enough to neglect $O\left(z_0^{k+1}\right)$
- $\epsilon$ must be large enough to reach the root.

Solution: write

$$f(x) = a_0 + a_k x^k + \text{other terms}$$

and find $k$ and $z_0$ such that $|a_k||z_0|^k$ is big enough w.r.t. the other terms and small enough compared to $|a_0|$.

Main Lemma

Given $|a_0| > 0$, $|a_1|, \ldots, |a_{n-1}| \geq 0$, $a_n = 1$, there are $r \in \mathbb{R}_{>0}$ and $k \in \{1, \ldots, n\}$ such that

$$r^n < |a_0|$$
$$3^{-2n^2}|a_0| < |a_k|r^k < |a_0|$$
$$\Sigma_{i=1, i\neq k}^{n} |a_i|r^i < (1 - 3^{-n})|a_k|r^k$$

Now take

$$z := r\sqrt[k]{-\frac{a_0}{a_k} \cdot \frac{|a_k|}{|a_0|}}$$

So $a_k z^k$ points opposite to $a_0$. Then

$$|f(z)| < q \cdot |a_0|$$

with $q = 1 - 3^{-2n^2-n}$.



From this one constructs a sequence $(z_i)_{i\in\mathbb{N}}$ in $\mathbb{C}$ such that

$$|f(z_i)| < q^i|a_0|$$
$$|z_{i+1} - z_i| < (q^i|a_0|)^{1/n}$$

This is a Cauchy sequence with limit a root of $f$.



Real Numbers in Coq:

- Axiomatic: a 'Real Number Structure' is a
  Cauchy-complete Archimedean ordered field.
- Prove FTA 'for all real numbers structures'.
- Construct a model to show that real number structures exist.
  (Cauchy sequences over an Arch. ordered field, say $\mathbb{Q}$)
- Prove that any two real number structures are isomorphic.



Axioms for Real Numbers:

- Algebraic hierarchy based on Constructive Setoids.
- Apartness $\#$ as basic

$$x = y \leftrightarrow \neg(x \# y)$$
$$x \# y \rightarrow (x \# z) \vee (y \# z)$$
$$x \# x$$
$$x \# y \rightarrow y \# x$$

- Field operation reciprocal

$$1/- : (\Sigma x{:}F.x \# 0) \rightarrow (\Sigma x{:}F.x \# 0)$$

- Cauchy sequences over Field $F$:
  $g$ : nat $\rightarrow F$ is Cauchy if

$$\forall \epsilon{:}F_{>0}.\exists N{:}\mathbb{N}.\forall m \geq N(|g_m - g_N| < \epsilon)$$

The algebraic hierarchy of the FTA project

- In proving FTA, we have to deal with real numbers, complex numbers and polynomials
- Many of the properties we use are generic and algebraic.
- To be able to reuse results (also for future developments) we have defined a hierarchy of algebraic structures.
- Basic level: constructive setoids, $\langle A, \#_A, =_A \rangle$, with $A$ : Set, $\#_A$ an apartness and $=_A$ an equivalence relation.
- Next level: semi-groups, $\langle S, + \rangle$, with $S$ a setoid and $+$ an associative binary operation on $S$.



Proofs by computation / Reflection

Needed for the proof of FTA:

Proofs of equalities between rational expressions like

$$\frac{1}{x+y} + \frac{1}{x-y} = \frac{2x}{x^2-y^2}$$

are obtained by partial reflection.

Following the reflection method:

$$[-] : E \to \mathbb{R}$$

with $E$ the type of rational expressions. So $E$ contains a constructor `erecip : E -> E`
But in the case of rational expressions, the $[-]$ can not be total.
$\rightsquigarrow$ partial reflection.



Inheritance via Coercions
We have the following coercions.

```
OrdField >-> Field >-> Ring >-> Group
                Group >-> Monoid >-> Semi_grp >-> Setoid
```

- All properties of groups are inherited by rings, fields, etc.
- Also notation is inherited:

  `x[+]y`

  denotes the addition of `x` and `y` for `x,y:G` from any semi-group (or monoid, group, ring,...) `G`.
- The coercions must form a tree, so there is no real multiple inheritance:
  E.g. it is not possible to define rings in such a way that it inherits both from its additive group and its multiplicative monoid.



Axioms for Real Numbers ctd.:

- All Cauchy sequences have a limit:

  $\mathsf{SeqLim} : (\Sigma g{:}\mathsf{nat}\to F.\mathsf{Cauchy}\, g) \to F$

  $\mathsf{CauchyProp} : \forall g{:}\mathsf{nat}\to F.(\mathsf{Cauchy}\, g) \to$
  $\forall \epsilon{:}F_{>0}.\exists N{:}\mathbb{N}.\forall m \geq N.(|g_m - (\mathsf{SeqLim}\, g)| < \epsilon)$
- Axiom of Archimedes: (there are no non-standard elements)

  $$\forall x{:}F.\exists n{:}\mathbb{N}(n > x)$$

NB: The axiom of Archimedes proves that '$\epsilon$-Cauchy sequences' and '$\frac{1}{k}$-Cauchy sequences' coincide (similar for limits):
Viz: $g$ : nat $\to F$ is a $\frac{1}{k}$-Cauchy sequence if

$$\forall k{:}\mathbb{N}.\exists N{:}\mathbb{N}.\forall m \geq N(|g_m - g_N| < \frac{1}{k+1})$$

Construction of a Real Number Structure:

- Construct $\mathbb{Q} := \{\langle p, n\rangle \mid p{:}\mathbb{Z}, n{:}\mathbb{N}\}$ with intended interpretation $\langle p, n\rangle \mapsto \frac{p}{n+1}$.
- Define all operations and relations, turning $\mathbb{Q}$ into an Archimedean constructive ordered field.
- For $F$ an (Archimedean) constructive ordered field, define the Cauchy completion of $F$, $\mathsf{CSeq}_F$.

$$h < g := \exists \epsilon : F_{>0}.\exists N{:}\mathbb{N}\forall m \geq N.(g_m - h_m > \epsilon).$$

$$g_m^{-1} := \begin{cases} 0 & \text{if } m < N \\ \frac{1}{g_m} & \text{if } m \geq N \end{cases}$$

  where $N$ is such that $\forall m \geq N.|g_m| > \epsilon$ for some $\epsilon > 0$.
- Prove that $\mathsf{CSeq}_F$ is a Real Number Structure.



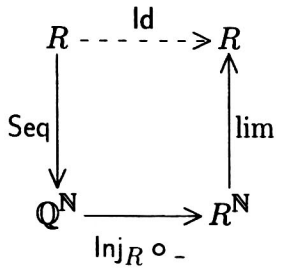
Categoricity of Real Number Structures:

- A morphism between $R_1$ and $R_2$ is a $\varphi : R_1 \to R_2$ that
  1. is strongly extensional: $\forall x, y : R_1.\ \varphi(x)\#\varphi(y) \to x\#y$.
  2. preserves order, addition and multiplication.
- A morphism preserves 'all' structure (Cauchy property, limits, ...)
  (Use: $\forall y \in R_2.\exists x \in R_1(\varphi(x) < y)$.)
- $R_1 \simeq R_2$ if $\varphi : R_1 \to R_2$ for some bijective morphism $\varphi$.



Categoricity of Real Number Structures ctd.:

- All Real Number Structures are isomorphic to $\mathbb{Q}^{\mathbb{N}}$:




- $\mathsf{Inj}_R : \mathbb{Q} \to R$ canonically maps rationals into $R$.
- $\mathsf{Seq} : R \to \mathbb{Q}^{\mathbb{N}}$: For $x : R$, define the $\mathbb{Q}$-sequences $\{q_i\}_{i=0}^{\infty}$ (increasing) and $\{r_i\}_{i=0}^{\infty}$ (decreasing), such that
  1. $\forall n{:}\mathbb{N}(r_n - q_n = (r_0 - q_0)(\frac{2}{3})^n)$
  2. $\forall n{:}\mathbb{N}(\mathsf{Inj}_R(q_n) \leq x \leq \mathsf{Inj}_R(r_n))$



Some Conclusions:

- Real mathematics, involving both a bit of algebra and a bit of analysis can be formalised completely within a theorem prover (Coq).
- Setting up a basic library and some good proof automation procedures is a substantial part of the work.
- Computationally, the behaviour of the algorithm depends mainly on the representation of the reals.
- The non-determinism in the root-finding algorithm (non-continuity of the root-finding function) lies in the taking of a $k$-th root in $\mathbb{C}$.
- How to present a proof?? Curry Howard: proofs-as-terms Converse??

Some Conclusions:

- **Real mathematics, involving both a bit of algebra and a bit of analysis can be formalised completely within a theorem prover (Coq).**
- **Setting up a basic library and some good proof automation procedures is a substantial part of the work.**
- **Computationally, the behaviour of the algorithm depends mainly on the representation of the reals.**
- **The non-determinism in the root-finding algorithm (non-continuity of the root-finding function) lies in the taking of a $k$-th root in $\mathbb{C}$.**
- **How to present a proof??** Curry-Howard: proofs-as-terms
  **Converse:** Hurry-Coward: **present** terms as 'readable' proofs

5 Fausto Giunchiglia, Marco Pistore, Marco Roveri
*ITC-IRST Trento, Italy*

Course: Formal Methods and Modelchecking

Tutorial: The NuSMV System

# Formal Methods and Model Checking

*F. Giunchiglia, M. Pistore, and M. Roveri*

Calculemus Autumn School 2002
Sep 30 - Oct 4, 2002
Pisa (Italy)





## Acknowledgment

These slides are derived from a course on "NuSMV and Symbolic Model Checking" (see `http://nusmv.irst.itc.it/courses/`).

The goals of the course are:

- to provide a practical introduction to symbolic model checking,
- to describe the basic feautures of the NuSMV symbolic model checker.

Authors of the slides:

- Alessandro Cimatti (ITC-irst)
- Marco Pistore (ITC-irst and University of Trento)
- Marco Roveri (ITC-irst)





## The course at a glance

- Part 1: Course Overview
  - Introduction Formal Methods
  - Introduction to Model Checking
- Part 2: Symbolic Model Checking
  - Models for Reactive Systems: Kripke Structures
  - Properties of Reactive Systems: CTL, LTL
  - Symbolic Model Checking Techniques: BDD-based and SAT-based techniques
- Part 3: The NuSMV Model Checker
  - The NuSMV Open Source project
  - The SMV language





# Part 1 - Course Overview

*– Formal Methods and Model Checking–*

*F. Giunchiglia, M. Pistore, and M. Roveri*

Calculemus Autumn School 2002, Sep 30 - Oct 4, 2002, Pisa (Italy)

## The Need for Formal Methods

- Development of Industrial Systems:
  - Growing complexity of environment and required functionalities.
  - Integration among required functionalities.
  - Safety-critical, money-critical systems.
  - Market issues: time-to-market, development costs.
  - Maintenance: requirement may change over time.
  - Quality of resulting product (e.g. requirement traceability).
- System reliability increasingly depends on the "correct" functioning of hardware and software.





## The Need for Formal Methods (II)

- Difficulties with traditional methodologies:
  - Ambiguities (in requirements, architecture, detailed design).
  - Errors is specification/design refinements.
  - The assurance provided by testing can be too limited.
  - Testing came too late in development.
- Consequences:
  - Expensive errors in the early design phases.
  - Infeasibility of achieving high reliability requirements.
  - Low software quality: poor documentation, limited modifiability.





## Formal Methods: Solution and Benefits

- The problem:
  - Certain (sub)systems can bee too complicated/critical to design with traditional techniques.
- Formal Methods:
  - *Formal Specification*: precise, unambiguous description.
  - *Formal Validation & Verification Tools*: exhaustive analysis of the formal specification.
- Potential Benefits:
  - Find design bugs in early design stages.
  - Achieve higher quality standards.
  - Shorten time-to-market reducing manual validation phases.
  - Produce well documented, maintainable products.

Highly recommended by ESA, NASA for the design of safety-critical systems.





## Formal Methods: Potential Problems

- Main Issue: Effective use of Formal Methods
  - debug/verify during the design process;
  - without slowing down the design process;
  - without increasing costs too much.
- Potential Problems of Formal Methods:
  - formal methods can be too costly;
  - formal methods can be not effective;
  - training problems;
  - the verification problem can be too difficult.
- How can we get benefits and avoid these problems?
  - by adapting our technologies and tools to the specific problem at hand;
  - using advanced verification techniques (e.g. model checking).

## Formal Verification Techniques

- Model-based simulation or testing
  - method: test for $\phi$ by exploring possible behaviors.
  - tools: test case generators.
  - applicable if: the system defines an executable model.
- Deductive Methods
  - method: provide a formal proof that $\phi$ holds.
  - tool: theorem prover, proof assistant or proof checker.
  - applicable if: systems can be described as a mathematical theory.
- Model Checking
  - method: systematic check of $\phi$ in all states of the system.
  - tools: model checkers.
  - applicable if: system generates (finite) behavioral model.





## Simulation and Testing

Basic procedure:

- take a model (simulation) or a realization (testing).
- stimulate it with certain inputs, i.e. the test cases.
- observe produce behavior and check whether this is "desired".

Drawback:

- The number of possible behaviors can be too large (or even infinite).
- Unexplored behaviors may contain the fatal bug.

Testing and simulation can show the presence of bugs, *not their absence*.





## Theorem Proving

Basic procedure:

- describe the system as a mathematical theory.
- express the property in the mathematical theory.
- prove that the property is a theorem in the mathematical theory.

Drawback:

- Express the system as a mathematical theory can be difficult.
- Find a proof can require a big effort.

Theorem proving can be used to prove *absence of bugs*.





## Model Checking

Basic procedure:

- describe the system as Finite State Model.
- express properties in Temporal Logic.
- formal V&V by automatic exhaustive search over the state space.

Drawback:

- State space explosion.
- Expressivity – hard to deal with parametrized systems.

Model checking can be used to prove *absence of bugs*.

## Industrial success of Model Checking

- From academics to industry in a decade.
- Easier to integrate within industrial development cycle:
  - input from practical design languages (e.g. VHDL, SDL, StateCharts);
  - expressiveness limited but often sufficient in practice.
- Does not require deep training ("push-button" technology).
  - Easy to explain as exhaustive simulation.
- Powerful debugging capabilities:
  - detect costly problems in early developmemt stages (cfr. Pentium bug);
  - exhaustive, thus effective (often bugs are also in scaled-down problems).
  - provides counterexamples (directs the designer to the problem).





## Model Checking in a nutshell

- Reactive systems represented as a finite state models (in this course, Kripke models).
- System behaviors represented as (possibly) infinite sequences of states.
- Requirements represented as formulae in temporal logics.
- *"The system satisfies the requirement"* represented as truth of the formula in the Kripke model.
- Efficient model checking algorithms based on exhaustive exploration of the Kripke model.





## What is a Model Checker

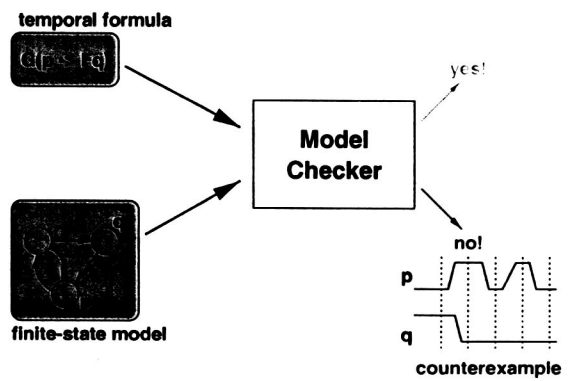
A model checker is a software tool that

- given a description of a Kripke model $M$ ...
- ... and a property $\Phi$,
- decides whether $M \models \Phi$,
- returns "yes" if the property is satisfied,
- otherwise returns "no", and provides a counterexample.





## What is a Model Checker



## What is not covered in this course

- A deep theoretical background. We will focus on practice.
- Advanced model checking techniques:
  - abstraction;
  - compositional, assume-guarantee reasoning;
  - symmetry reduction;
  - approximation techniques (e.g. directed to bug hunting);
  - model transformation techniques (e.g. minimization wrt to bisimulation).
- Many other approaches and tools for model checking.
- Model checking for infinite-state (e.g. hybrid, timed) systems.





# Symbolic Model Checking

*– Formal Methods and Model Checking–*

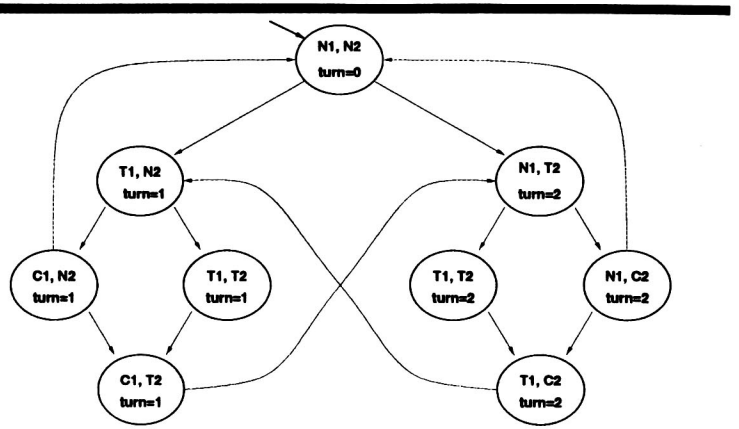
*F. Giunchiglia, M. Pistore, and M. Roveri*

Calculemus Autumn School 2002, Sep 30 - Oct 4, 2002, Pisa (Italy)





## A Kripke model for mutual exclusion

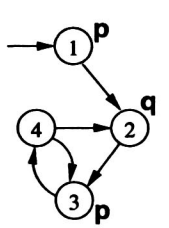







## Modeling the system: Kripke models

- Kripke models are used to describe reactive systems:
  - nonterminating systems with infinite behaviors,
  - e.g. communication protocols, operating systems, hardware circuits;
  - represent dynamic evolution of modeled systems;
  - values to state variables, program counters, content of communication channels.
- Formally, a Kripke model $(S, R, I, L)$ consists of
  - a set of states $S$;
  - a set of initial states $I \subseteq S$;
  - a set of transitions $R \subseteq S \times S$;
  - a labeling $L \subseteq S \times AP$.

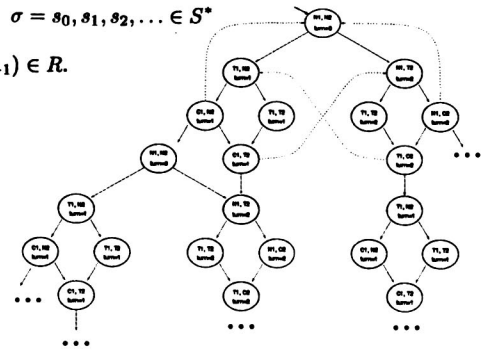


## Path in a Kripke Model

- A path in a Kripke model $M$ is an infinite sequence

$$\sigma = s_0, s_1, s_2, \ldots \in S^*$$

such that $s_0 \in I$ and $(s_i, s_{i+1}) \in R$.



- A state $s$ is reachable in $M$ if there is a path from the initial states to $s$.





## Description languages for Kripke Model

A Kripke model is usually presented using a structured programming language.

Each component is presented by specifying

- state variables: determine the state space $S$ and the labeling $L$.
- initial values for state variables: determine the set of initial states $I$.
- instructions: determine the transition relation $R$.

Components can be combined via

- synchronous composition,
- asynchronous composition.

State explosion problem in model checking:

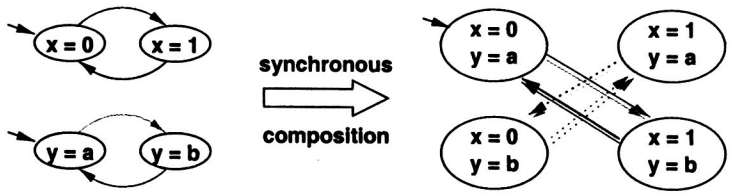
- linear in model size, but model is exponential in number of components.





## Synchronous Composition

- Components evolve in parallel.
- At each time instant, every component performs a transition.



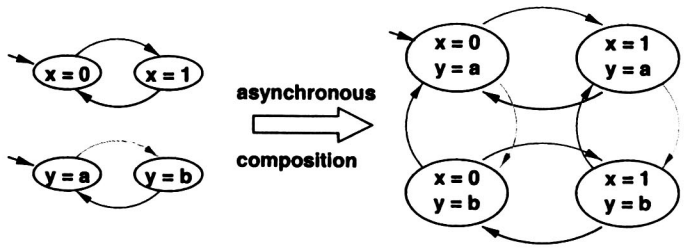

- Typical example: sequential hardware circuits.
- Synchronous composition is the default in NuSMV.





## Asynchronous Composition

- Interleaving of evolution of components.
- At each time instant, one component is selected to perform a transition.

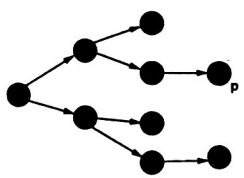



- Typical example: communication protocols.
- Asynchronous composition can be represented with NuSMV processes.

## Properties of Reactive Systems (I)

Safety properties:

- nothing bad ever happens
  - deadlock: two processes waiting for input from each other, the system is unable to perform a transition.
  - no reachable state satisfies a "bad" condition, e.g. never two process in critical section at the same time
- can be refuted by a finite behaviour
- it is never the case that $p$.



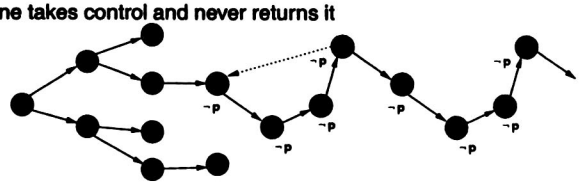





## Properties of Reactive Systems (II)

Liveness properties:

- Something desirable will eventually happen
  - whenever a subroutine takes control, it will always return it (sooner or later)
- can be refuted by infinite behaviour
  - a subroutine takes control and never returns it




  - an infinite behaviour can be presented as a loop





## Temporal Logics

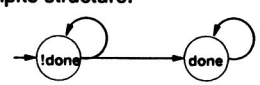
- Express properties of "Reactive Systems"
  - nonterminating behaviours,
  - without explicit reference to time.
- Linear Time Temporal Logic (LTL)
  - intepreted over each path of the Kripke structure
  - linear model of time
  - temporal operators
- Computation Tree Logic (CTL)
  - intepreted over computation tree of Kripke model
  - branching model of time
  - temporal operators plus path quantifiers





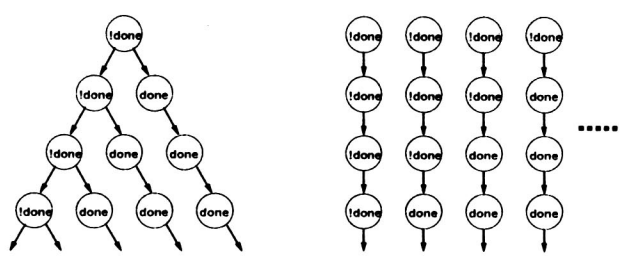
## Computation tree vs. computation paths

☞ Consider the following Kripke structure:




☞ Its execution can be seen as:

- an infinite *computation tree*
- a set of infinite *computation paths*



## Linear Time Temporal Logic (LTL)

LTL properties are evaluated over paths, i.e., over infinite, linear sequences of states:

$$s[0] \rightarrow s[1] \rightarrow \cdots \rightarrow s[t] \rightarrow s[t+1] \rightarrow \cdots$$

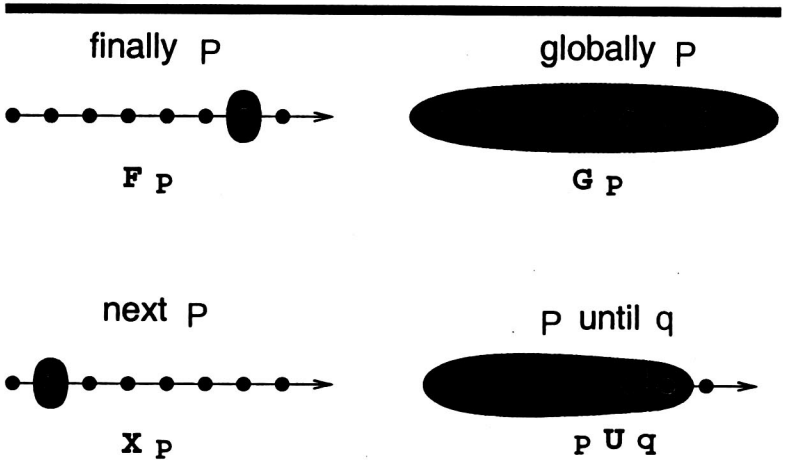
LTL provides the following temporal operators:

- "Finally" (or "future"): $Fp$ is true in $s[t]$ iff $p$ is true in **some** $s[t']$ with $t' \geq t$
- "Globally" (or "always"): $Gp$ is true in $s[t]$ iff $p$ is true in **all** $s[t']$ with $t' \geq t$
- "Next": $Xp$ is true in $s[t]$ iff $p$ is true in $s[t+1]$
- "Until": $pUq$ is true in $s[t]$ iff
  - $q$ is true in some state $s[t']$ with $t' \geq t$
  - $p$ is true in all states $s[t'']$ with $t \leq t'' < t'$





## LTL








## LTL: Examples

- Liveness: "if input, then eventually output"

  $$G(\text{input} \rightarrow F\text{output})$$

- Strong fairness: "infinitely send implies infinitely recv."

  $$GF\text{send} \rightarrow GF\text{recv}$$

- Weak until: "no output before input"

  $$\neg\text{output } W \text{ input}$$

  where $p\ W\ q \ \leftrightarrow \ (p\ U\ q \vee Gp)$

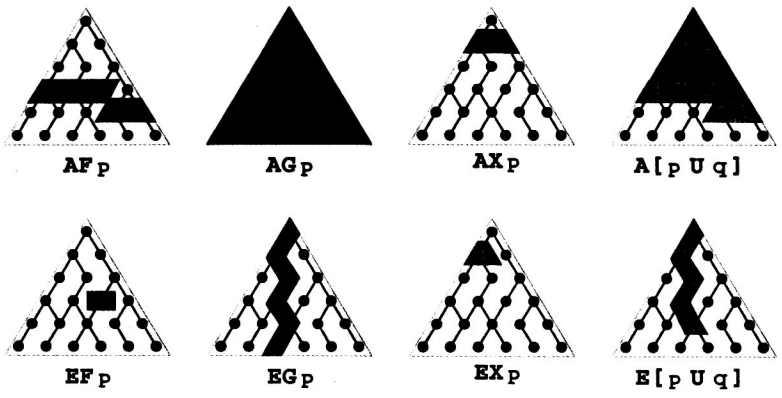




## Computation Tree Logic (CTL)

- CTL properties are evaluated over trees.
- Every temporal operator ($F, G, X, U$) preceded by a path quantifier ($A$ or $E$).
- Universal modalities ($AF, AG, AX, AU$): the temporal formula is true in **all** the paths starting in the current state.
- Existential modalities ($EF, EG, EX, EU$): the temporal formula is true in **some** of the paths starting in the current state.

## CTL








## CTL

- Some dualities:

$$AGp \leftrightarrow \neg EF\neg p$$
$$AFp \leftrightarrow \neg EG\neg p$$
$$AXp \leftrightarrow \neg EX\neg p$$

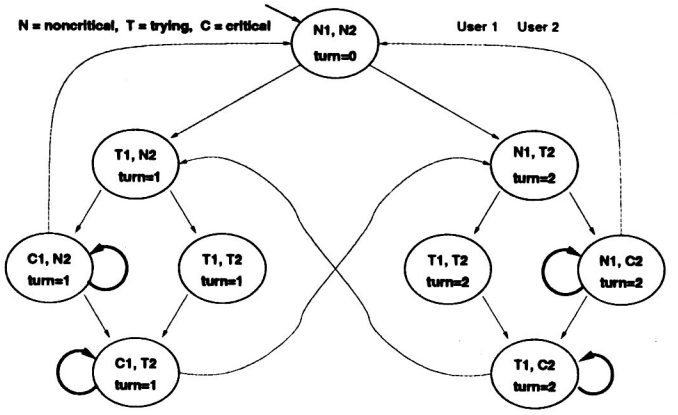
- Example: specifications for the mutual exclusion problem.

| | |
|---|---|
| $AG\neg(C_1 \wedge C_2)$ | mutual exclusion |
| $AG(T_1 \rightarrow AF\, C_1)$ | liveness |
| $AG(N_1 \rightarrow EX\, T_1)$ | non-blocking |





## The need for fairness conditions

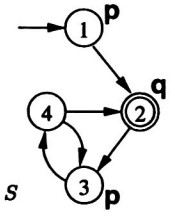







## Fair Kripke models

- Intuitively, fairness conditions are used to eliminate behaviours where a condition never holds
  - e.g. once a process is in critical section, it never exits
- Formally, a Kripke model $(S, R, I, L, F)$ consists of
  - a set of states $S$;
  - a set of initial states $I \subseteq S$;
  - a set of transitions $R \subseteq S \times S$;
  - a labeling $L \subseteq S \times AP$.
  - $\Rightarrow$ a set of fairness conditions $F = \{f_1, \ldots, f_n\}$, with $f_i \subseteq S$

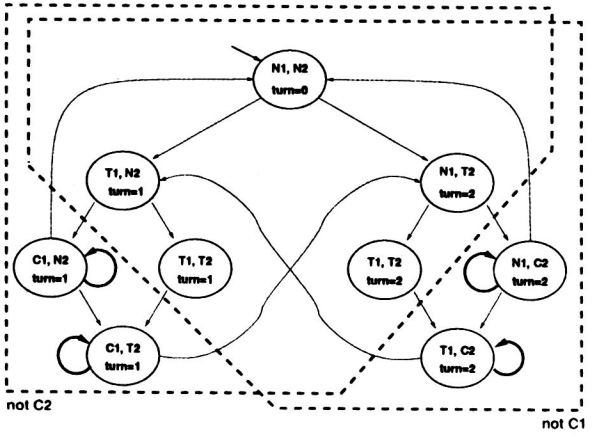



- Fair path: at least one state for each $f_i$ occurs an infinite number of times
- Fair state: a state from which at least one fair path originates



1-69

## Fairness: {{ not C1},{not C2}}



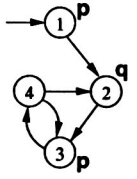





## Model Checking

Model Checking is a formal verification technique where...

- ...the system is represented as Finite State Machine




- ...the properties are expressed as temporal logic formulae

  LTL: **G(p –> Fq)** CTL: **AG(p –> AFq)**

- ...the model checking algorithm checks whether all the executions of the model satisfy the formula.





## The Main Problem: State Space Explosion

The bottleneck:

- Exhaustive analysis may require to store all the states of the Kripke structure
- The state space may be exponential in the number of components
- State Space Explosion: too much memory required

Symbolic Model Checking:

- Symbolic representation
- Different search algorithms





## Symbolic Model Checking

Symbolic representation:

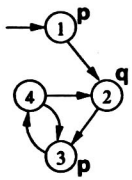
- manipulation of *sets of states* (rather than single states);
- sets of states represented by formulae in propositional logic;
  - set cardinality not directly correlated to size
- expansion of *sets of transitions* (rather than single transitions);
- two main symbolic techniques:
  - Binary Decision Diagrams (BDDs)
  - Propositional Satisfiability Checkers (SAT solvers)

Different model checking algorithms:

- Fix-point Model Checking (historically, for CTL)
- Bounded Model Checking (historically, for LTL)
- Invariant Checking

# CTL Model Checking: Example

Consider a simple system and a specification:




**AG(p -> AFq)**

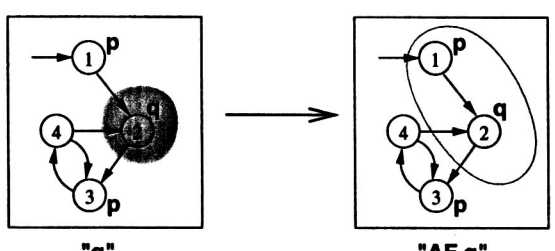
Idea:

- construct the set of states where the formula holds
- proceeding "bottom-up" on the structure of the formula
- **q, AFq, p, p → AF q, AG(p → AF q)**





# CTL Model Checking: Example




**"q"**



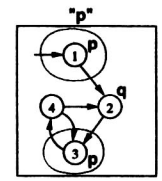

**"AF q"**

**AF q** is the union of **q, AX q, AX AX q, ...**

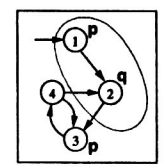




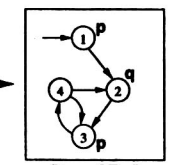
# CTL Model Checking: Example




**"p"**




**"AF q"**

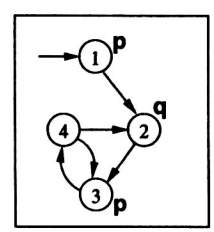



**"p -> AF q"**

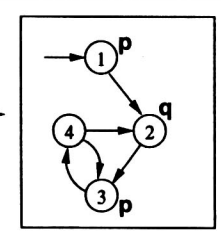




# CTL Model Checking: Example




**"p -> AF q"**




**"AG(p -> AF q)"**

The set of states where the formula holds is empty!

Counterexample reconstruction is based on the intermediate sets.

## Fix-Point Symbolic Model Checking

Model Checking Algorithm for CTL formulae based on fix-point computation:

- traverse formula structure, for each subformula build set of satisfying states; compare result with initial set of states.
- boolean connectives: apply corresponding boolean operation;
- on $\mathrm{AX}\,\Phi$, apply preimage computation
  - $\forall \mathbf{s}'.(\mathcal{T}(\mathbf{s},\mathbf{s}') \rightarrow \Phi(\mathbf{s}'))$
- on $\mathrm{AF}\,\Phi$, compute least fixpoint using
  - $\mathrm{AF}\,\Phi \leftrightarrow (\Phi \vee \mathrm{AX}\,\mathrm{AF}\,\Phi)$
- on $\mathrm{AG}\,\Phi$, compute greatest fixpoint using
  - $\mathrm{AG}\,\Phi \leftrightarrow (\Phi \wedge \mathrm{AX}\,\mathrm{AG}\,\Phi)$





## Bounded Model Checking

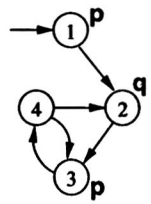
Key ideas:

- looks for counter-example paths of increasing length $k$
  - oriented to finding bugs
- for each $k$, builds a boolean formula that is satisfiable iff there is a counter-example of length $k$
  - can be expressed using $k \cdot |\mathbf{s}|$ variables
  - formula construction is not subject to state explosion
- satisfiability of the boolean formulas is checked using a *SAT procedure*
  - can manage complex formulae on several 100K variables
  - returns satisfying assignment (i.e., a counter-example)





## Bounded Model Checking: Example



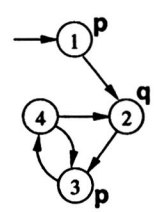

- Formula: **G(p -> Fq)**
- Negated Formula (violation): **F(p & G ! q)**
- $k = 0$: 

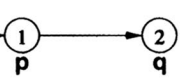
- No counter-example found.





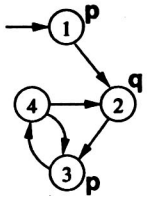
## Bounded Model Checking: Example

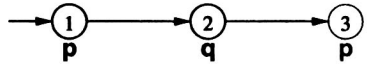



- Formula: **G(p -> Fq)**
- $k = 1$: 

- No counter-example found.

## Bounded Model Checking: Example



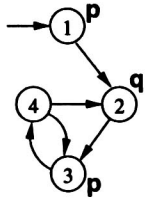

- Formula: **G(p -> Fq)**
- $k = 2$: 

- No counter-example found.

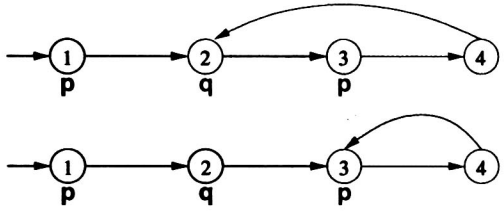




## Bounded Model Checking: Example




- Formula: **G(p -> Fq)**
- $k = 3$:




- The 2nd trace is a counter-example!





## Bounded Model Checking

- *Bounded Model Checking*:
  Given a FSM $\mathcal{M} = \langle \mathcal{S}, \mathcal{I}, \mathcal{T} \rangle$, an LTL property $\phi$ and a bound $k \geq 0$:

$$\mathcal{M} \models_k \phi$$

- This is equivalent to the satisfiability problem on formula:

$$[\mathcal{M}, \phi]_k \equiv [\mathcal{M}]_k \wedge [\phi]_k$$

  where:
  - $[\mathcal{M}]_k$ is a $k$-path compatible with $\mathcal{I}$ and $\mathcal{T}$:

$$\mathcal{I}(s_0) \wedge \mathcal{T}(s_0, s_1) \wedge \ldots \mathcal{T}(s_{k-1}, s_k)$$

  - $[\phi]_k$ says that the $k$-path satisfies $\phi$

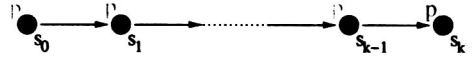




## Bounded Model Checking: Examples

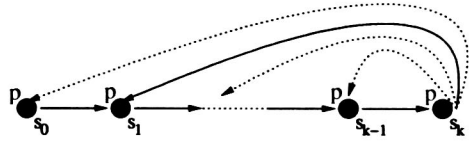
- $\phi = \mathrm{F}\, p$

$$[\mathrm{F}\, p]_k = \bigvee_{i=0}^{k} p(s_i)$$




- $\phi = \mathrm{G}\, p$

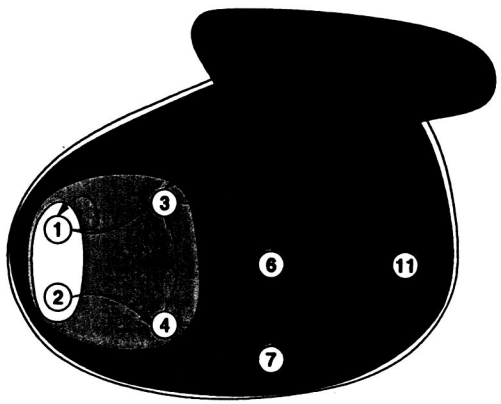
$$[\mathrm{G}\, p]_k = \bigvee_{i=0}^{k} \left( \mathcal{T}(s_k, s_i) \wedge \bigwedge_{i=0}^{k} p(s_i) \right)$$



## Symbolic Model Checking of Invariants

Checking invariant properties (e.g. **AG ! bad** is a reachability problem):

- is there a reachable state that is also a bad state (●)?



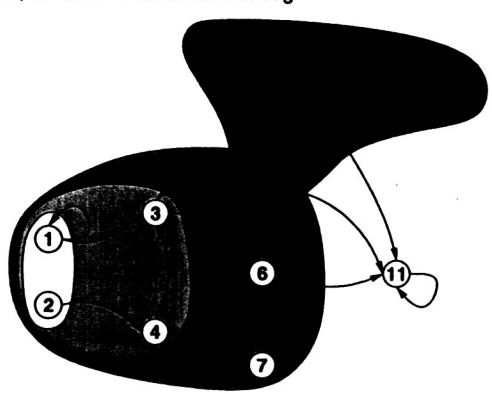





## On the fly Checking of Invariants

Anticipate bug detection:

- at each layer, check if a new state is a bug



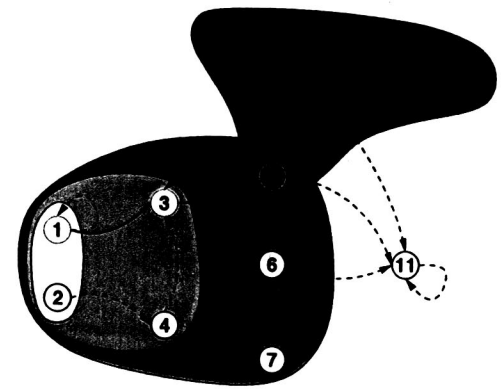





## On the fly Checking of Invariants: Counterexamples

If a bug is found,

- a counterexample can be reconstructed proceeding backwards








## Inductive Reasoning on Invariants

1. If all the initial states are good,
2. and if from any good state we only go to good states

⇒ then we can conclude that the system is correct for all reachable states.

# The NuSMV Model Checker

*– Formal Methods and Model Checking–*

*F. Giunchiglia, M. Pistore, and M. Roveri*

Calculemus Autumn School 2002, Sep 30 - Oct 4, 2002, Pisa (Italy)





## Introduction

- NuSMV is a symbolic model checker developed by ITC-IRST and UniTN with the collaboration of CMU and UniGE.
- The NuSMV project aims at the development of a state-of-the-art model checker that:
  - is robust, open and customizable;
  - can be applied in technology transfer projects;
  - can be used as research tool in different domains.
- NuSMV is *OpenSource*:
  - developed by a distributed community,
  - "Free Software" license.





## History: NuSMV 1

NuSMV is a reimplementation and extension of SMV.

- NuSMV started in 1998 as a joint project between ITC-IRST and CMU:
  - the starting point: SMV version 2.4.4.
  - SMV is the first BDD-based symbolic model checker (McMillan, 90).
- NuSMV version 1 has been released in July 1999.
  - limited to BDD-based model checking
  - extends and upgrades SMV along three dimensions:
    - functionalities (LTL, simulation)
    - architecture
    - implementation
- Results:
  - used for teaching courses and as basis for several PhD theses
  - interest by industrial companies and academics





## History: NuSMV 2

- The NuSMV 2 project started in September 2000 with the following goals:
  - Introduction of SAT-based model checking
  - OpenSource licensing
  - Larger team (Univ. of Trento, Univ. of Genova, ...)
- NuSMV 2 has been released in November 2001.
  - first freely available model checker that combines BDD-based and SAT-based techniques
  - extended functionalities wrt NuSMV 1 (cone of influence, improved conjunctive partitioning, multiple FSM management)
- Results: in the first two months:
  - more than 60 new registrations of NuSMV users
  - more than 300 downloads

## OpenSource License

The idea of OpenSource:

- The System is developed by a distributed community
- Notable examples: Netscape, Apache, Linux
- *Potential* benefits: shared development efforts, faster improvements...

Aim: provide a *publicly available*, *state-of-the-art* symbolic model checker.

- *publicly available*: free usage in research and commercial applications
- *state of the art*: improvements should be made freely available

Distribution license for NuSMV 2: *GNU Lesser General Public License* (LGPL):

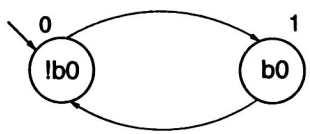
- anyone can freely download, copy, use, modify, and redistribute NuSMV 2
- any modification and extension should be made publicly available under the terms of LGPL ("copyleft")





## The first SMV program

```
MODULE main
 VAR
   b0 : boolean;

 ASSIGN
   init(b0) := 0;
   next(b0) := !b0;
```




An SMV program consists of:

- Declarations of the state variables (`b0` in the example); the state variables determine the state space of the model.
- Assignments that define the valid initial states (`init(b0) := 0`).
- Assignments that define the transition relation (`next(b0) := !b0`).





## Declaring state variables

The SMV language provides booleans, enumerative and bounded integers as data types:

**boolean**:

```
VAR
  x : boolean;
```

**enumerative**:

```
VAR
  st : {ready, busy, waiting, stopped};
```

**bounded integers** (intervals):

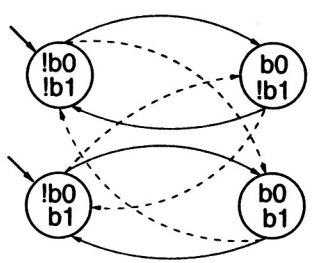
```
VAR
  n : 1..8;
```





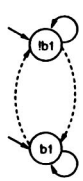
## Adding a state variable

```
MODULE main
VAR
  b0 : boolean;
  b1 : boolean;

ASSIGN
  init(b0) := 0;
  next(b0) := !b0;
```




Remarks:

- The new state space is the artesian product of the ranges of the variables.
- Synchronous composition between the "subsystems" for b0 and b1.



## Declaring the set of initial states

- For each variable, we constrain the values that it can assume in the *initial states*.

```
init(<variable>) := <simple_expression> ;
```

- `<simple_expression>` must evaluate to values in the domain of `<variable>`.
- If the initial value for a variable is not specified, then the variable can initially assume any value in its domain.





## Expressions

- Arithmetic operators:
  `+` `-` `*` `/` `mod` - (unary)
- Comparison operators:
  `=` `!=` `>` `<` `<=` `>=`
- Logic operators:
  `&` `|` `xor` `!` (not) `->` `<->`
- Conditional expression:

```
case
  c1 : e1;
  c2 : e2;
  ...
  1  : en;
esac
```

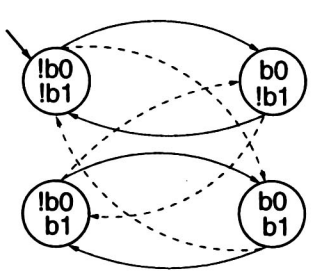
  if `c1` then `e1` else if `c2` then `e2` else if ... else `en`
- Set operators:
  `{v1,v2,...,vn}` (enumeration) `in` (set inclusion) `union` (set union)





## Declaring the set of initial states

```
MODULE main
 VAR
   b0 : boolean;
   b1 : boolean;

 ASSIGN
   init(b0) := 0;
   next(b0) := !b0;

   init(b1) := 0;
```








## Expressions

- Expressions in SMV do not necessarily evaluate to one value. In general, they can represent a set of possible values.

```
init(var) := {a,b,c} union {x,y,z} ;
```

- The meaning of `:=` in assignments is that the lhs can assume non-deterministically a value in the set of values represented by the rhs.
- A constant `c` is considered as a syntactic abbreviation for `{c}` (the singleton containing `c`).

## Declaring the transition relation

- The transition relation is specified by constraining the values that variables can assume in the *next state*.

```
next(<variable>) := <next_expression> ;
```

- `<next_expression>` must evaluate to values in the domain of `<variable>`.
- `<next_expression>` depends on "current" and "next" variables:

```
next(a) := { a, a+1 } ;
next(b) := b + (next(a) - a) ;
```

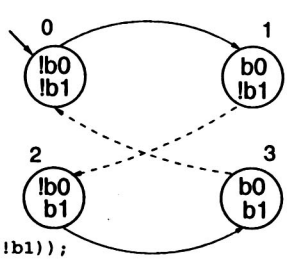
- If no `next()` assignment is specified for a variable, then the variable can evolve non deterministically, i.e. it is unconstrained.
  Unconstrained variables can be used to model non-deterministic *inputs* to the system.





## Declaring the transition relation

```
MODULE main
 VAR
   b0 : boolean;
   b1 : boolean;

 ASSIGN
   init(b0) := 0;
   next(b0) := !b0;

   init(b1) := 0;
   next(b1) := ((!b0 & b1) | (b0 & !b1));
```








## Specifying normal assignments

- Normal assignments constrain the *current value* of a variable to the current values of other variables.
- They can be used to model *outputs* of the system.

```
<variable> := <simple_expression> ;
```

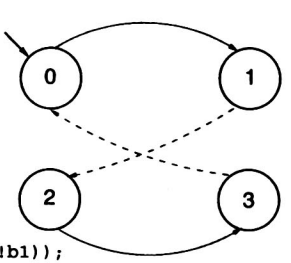
- `<simple_expression>` must evaluate to values in the domain of the `<variable>`.





## Specifying normal assignments

```
MODULE main
 VAR
   b0 : boolean;
   b1 : boolean;
   out : 0..3;

 ASSIGN
   init(b0) := 0;
   next(b0) := !b0;

   init(b1) := 0;
   next(b1) := ((!b0 & b1) | (b0 & !b1));

   out := b0 + 2*b1;
```



## Restrictions on the ASSIGN

For technical reasons, the transition relation must be *total*, i.e., for every state there must be at least one successor state.

In order to guarantee that the transition relation is total, the following restrictions are applied to the SMV programs:

- Double assignments rule – Each variable may be assigned only once in the program.
- Circular dependencies rule – A variable cannot have "cycles" in its dependency graph that are not broken by delays.

If an SMV program does not respect these restrictions, an error is reported by NuSMV.





## Double assignments rule

***Each variable may be assigned only once in the program.***

All of the following combinations of assignments are illegal:

```
init(status) := ready;
init(status) := busy;

next(status) := ready;
next(status) := busy;

status := ready;
status := busy;

init(status) := ready;
status := busy;

next(status) := ready;
status := busy;
```





## Circular dependencies rule

***A variable cannot have "cycles" in its dependency graph that are not broken by delays.***

All the following combinations of assignments are illegal:

```
x := (x + 1) mod 2;

x := (y + 1) mod 2;
y := (x + 1) mod 2;

next(x) := x & next(x);

next(x) := x & next(y);
next(y) := y & next(x);
```

The following example is *legal*, instead:

```
next(x) := x & next(y);
next(y) := y & x;
```

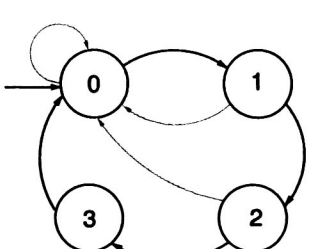




## The modulo 4 counter with reset

The counter can be reset by an external "uncontrollable" reset signal.

```
MODULE main
 VAR
   b0    : boolean;
   b1    : boolean;
   reset : boolean;
   out   : 0..3;

 ASSIGN
   init(b0) := 0;
   next(b0) := case
                 reset = 1 : 0;
                 reset = 0 : !b0;
               esac;

   init(b1) := 0;
   next(b1) := case
                 reset : 0;
                 1     : (((!b0 & b1) | (b0 & !b1));
               esac;

   out := b0 + 2*b1;
```



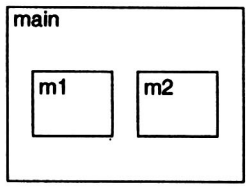



1 2 9 1

## Modules

An SMV program can consist of one or more *module declarations*.

```
MODULE mod
  VAR out: 0..9;
  ASSIGN next(out) :=
                (out + 1) mod 10;
MODULE main
  VAR m1 : mod;
      m2 : mod;
      sum: 0..18;
  ASSIGN sum := m1.out + m2.out;
```




- Modules are instantiated in other modules. The instantiation is performed inside the `VAR` declaration of the parent module.
- In each SMV specification there must be a module `main`. It is the top-most module.
- All the variables declared in a module instance are visible in the module in which it has been instantiated via the dot notation (e.g., `m1.out`, `m2.out`).

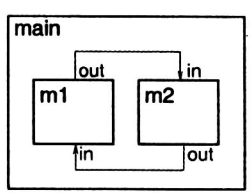




## Module parameters

Module declarations may be *parametric*.

```
MODULE mod(in)
  VAR out: 0..9;
  ...
MODULE main
  VAR m1 : mod(m2.out);
      m2 : mod(m1.out);
  ...
```




- *Formal parameters* (`in`) are substituted with the *actual parameters* (`m2.out`, `m1.out`) when the module is instantiated.
- Actual parameters can be any legal expression.
- Actual parameters are passed by reference.





## Example: The modulo 8 counter revisited

```
MODULE counter_cell(tick)

  VAR
    value : boolean;
    done  : boolean;

  ASSIGN
    init(value) := 0;
    next(value) := case
      tick = 0 : value;
      tick = 1 : (value + 1) mod 2;
    esac;

    done := tick & (((value + 1) mod 2) = 0);
```

Remarks:

- `tick` is the formal parameter of module `counter_cell`.





## Example: The modulo 8 counter revisited

```
MODULE main
  VAR
    bit0 : counter_cell(1);
    bit1 : counter_cell(bit0.done);
    bit2 : counter_cell(bit1.done);
    out  : 0..7;

  ASSIGN
    out := bit0.value + 2*bit1.value + 4*bit2.value;
```

Remarks:

- Module `counter_cell` is instantiated three times.
- In the instance `bit0`, the formal parameter `tick` is replaced with the actual parameter `1`.
- When a module is instantiated, all variables/symbols defined in it are preceded by the module instance name, so that they are unique to the instance.



1-80-1

## Module hierarchies

A module can contain instances of others modules, that can contain instances of other modules... provided the module references are not circular.

```
MODULE counter_8 (tick)
  VAR
    bit0 : counter_cell(tick);
    bit1 : counter_cell(bit0.done);
    bit2 : counter_cell(bit1.done);
    out  : 0..7;
    done : boolean;
  ASSIGN
    out  := bit0.value + 2*bit1.value + 4*bit2.value;
    done := bit2.done;

MODULE counter_512(tick) -- A counter modulo 512
  VAR
    b0 : counter_8(tick);
    b1 : counter_8(b0.done);
    b2 : counter_8(b1.done);
    out : 0..511;
 ASSIGN
    out := b0.out + 8*b1.out + 64*b2.out;
```





## Specifications

- The SMV language allows for the specification of different kinds of properties:
  - invariants,
  - CTL formulas,
  - LTL formulas...
- Specifications can be added in any module of the program.
- Each specification is verified separately by NuSMV.





## Specifications

In the SMV language:

- Specifications can be added in any module of the program.
- Each property is verified separately.
- Different kinds of properties are allowed:
  - Properties on the reachable states
    - *invariants* (`INVARSPEC`)
  - Properties on the computation paths (*linear time* logics):
    - LTL (`LTLSPEC`)
    - qualitative characteristics of models (`COMPUTE`)
  - Properties on the computation tree (*branching time* logics):
    - CTL (`SPEC`)
    - Real-time CTL (`SPEC`)





## Invariant specifications

- Invariant properties are specified via the keyword `INVARSPEC`:

```
INVARSPEC <simple_expression>
```

- Example:

```
MODULE counter_cell(tick)
  ...
MODULE counter_8 (tick)
  VAR
    bit0 : counter_cell(tick);
    bit1 : counter_cell(bit0.done);
    bit2 : counter_cell(bit1.done);
    out  : 0..7;
    done : boolean;
  ASSIGN
    out  := bit0.value + 2*bit1.value + 4*bit2.value;
    done := bit2.done;

  INVARSPEC
    done <-> (bit0.done & bit1.done & bit2.done)
```

## LTL specifications

- LTL properties are specified via the keyword `LTLSPEC`:

  `LTLSPEC <ltl_expression>`

  where `<ltl_expression>` can contain the following temporal operators:

  `X _ F _ G _ _ U _`

- A state in which `out = 3` is eventually reached.

  ```
  LTLSPEC F out = 3
  ```

- Condition `out = 0` holds until `reset` becomes false.

  ```
  LTLSPEC (out = 0) U (!reset)
  ```

- Even time a state with `out = 2` is reached, a state with `out = 3` is reached afterwards.

  ```
  LTLSPEC G (out = 2 -> F out = 3)
  ```





## CTL properties

- CTL properties are specified via the keyword `SPEC`:

  `SPEC <ctl_expression>`

  where `<ctl_expression>` can contain the following temporal operators:

  `AX _ AF _ AG _ A[_ U _]`
  `EX _ EF _ EG _ E[_ U _]`

- It is possible to reach a state in which `out = 3`.

  ```
  SPEC EF out = 3
  ```

- A state in which `out = 3` is always reached.

  ```
  SPEC AF out = 3
  ```

- It is always possible to reach a state in which `out = 3`.

  ```
  SPEC AG EF out = 3
  ```

- Even time a state with `out = 2` is reached, a state with `out = 3` is reached afterwards.

  ```
  SPEC AG (out = 2 -> AF out = 3)
  ```





## Quantitative characteristics computations

It is possible to compute the minimum and maximum length of the paths between two specified conditions.

- Quantitative characteristics are specified via the keyword `COMPUTE`:

  `COMPUTE MIN/MAX [ <simple_expression> , <simple_expression> ]`

- For instance, the shortest path between a state in which `out = 0` and a state in which `out = 3` is computed with

  ```
  COMPUTE
    MIN [ out = 0 , out = 3]
  ```

- The length of the longest path between a state in which `out = 0` and a state in which `out = 3`.

  ```
  COMPUTE
    MAX [ out = 0 , out = 3]
  ```





## Bounded CTL specifications

NuSMV provides *bounded CTL* (or *real-time CTL*) operators.

- There is no state that is reachable in 3 steps where `out = 3` holds.

  ```
  SPEC
    !EBF 0..3 out = 3
  ```

- A state in which `out = 3` is reached in 2 steps.

  ```
  SPEC
    ABF 0..2 out = 3
  ```

- From any reachable state, a state in which `out = 3` is reached in 3 steps.

  ```
  SPEC
    AG ABF 0..3 out = 3
  ```



1 28 1

## Fairness Constraints

Let us consider again the counter with reset.

- The specification `AF out = 1` is not verified.
- On the path where `reset` is always `1`, then the system loops on a state where `out = 0`, since the counter is always reset:
  `reset = 1,1,1,1,1,1,1...`
  `out = 0,0,0,0,0,0,0...`
- Similar considerations hold for the property `AF out = 2`. For instance, the sequence:
  `reset = 0,1,0,1,0,1,0...`
  generates the loop:
  `out = 0,1,0,1,0,1,0...`
  which is a counterexample to the given formula.





## Fairness Constraints

- NuSMV allows to specify *fairness* constraints.
- Fairness constraints are formulas which are assumed to be true infinitely often in all the execution paths of interest.
- During the verification of properties, NuSMV considers path quantifiers to apply only to fair paths.
- Fairness constraints are specified as follows:

```
FAIRNESS <simple_expression>
```





## Fairness Constraints

- With the fairness constraint

```
FAIRNESS
  out = 1
```

  we restrict our analysis to paths in which the property `out = 1` is true infinitely often.
- The property `AF out = 1` under this fairness constraint is now verified.
- The property `AF out = 2` is still not verified.
- Adding the fairness constraint `out = 2`, then also the property `AF out = 2` is verified.





## The `DEFINE` declaration

In the following example, the values of variables `out` and `done` are defined by the values of the other variables in the model.

```
MODULE main          -- counter_8
VAR
  b0   : boolean;
  b1   : boolean;
  b2   : boolean;
  out  : 0..8;
  done : boolean;

ASSIGN
  init(b0) := 0;
  init(b1) := 0;
  init(b2) := 0;

  next(b0) := !b0;
  next(b1) := (!b0 & b1) | (b0 & !b1);
  next(b2) := ((b0 & b1) & !b2) | (!(b0 & b1) & b2);

  out  := b0 + 2*b1 + 4*b2;
  done := b0 & b1 & b2;
```

## The DEFINE declaration

DEFINE declarations can be used to define *abbreviations*:

```
MODULE main          -- counter_8
VAR
  b0 : boolean;
  b1 : boolean;
  b2 : boolean;

ASSIGN
  init(b0) := 0;
  init(b1) := 0;
  init(b2) := 0;

  next(b0) := !b0;
  next(b1) := (!b0 & b1) | (b0 & !b1);
  next(b2) := ((b0 & b1) & !b2) | (!(b0 & b1) & b2);

DEFINE
  out  := b0 + 2*b1 + 4*b2;
  done := b0 & b1 & b2;
```





## The DEFINE declaration

- The syntax of DEFINE declarations is the following:

  DEFINE <id> := <simple_expression> ;

- They are similar to macro definitions.
- No new state variable is created for defined symbols (hence, no added complexity to model checking).
- Each occurrence of a defined symbol is replaced with the body of the definition.





## Arrays

The SMV language provides also the possibility to define *arrays*.

```
VAR
  x : array 0..10 of booleans;
  y : array 2..4 of 0..10;
  z : array 0..10 of array 0..5 of {red, green, orange};
ASSIGN
  init(x[5]) := 1;
  init(y[2]) := {0,2,4,6,8,10};
  init(z[3][2]) := {green, orange};
```

- Remark: Array indexes in SMV *must be constants*.





## Records

Records can be defined as modules without parameters and assignments.

```
MODULE point
  VAR x: -10..10;
      y: -10..10;

MODULE circle
  VAR center: point;
      radius: 0..10;

MODULE main
  VAR c: circle;
  ASSIGN
    init(c.center.x) := 0;
    init(c.center.y) := 0;
    init(c.radius)   := 5;
```

## The constraint style of model specification

The following SMV program:

```
MODULE main
VAR request : boolean;
    state   : {ready,busy};
ASSIGN
  init(state) := ready;
  next(state) := case
      state = ready & request : busy;
      1                       : {ready,busy};
  esac;
```

can be alternatively defined in a *constraint style*, as follows:

```
MODULE main
VAR request : boolean;
    state   : {ready,busy};
INIT
  state = ready
TRANS
  (state = ready & request) -> next(state) = busy
```





## The constraint style of model specification

- The SMV language allows for specifying the model by defining constraints on:
  - the *states*:
    `INVAR <simple_expression>`
  - the *initial states*:
    `INIT <simple_expression>`
  - the *transitions*:
    `TRANS <next_expression>`
- There can be zero, one, or more constraints in each module, and constraints can be mixed with assignments.
- Any propositional formula is allowed in constraints.
- Very useful for writing translators from other languages to NuSMV.
- `INVAR p` is equivalent to `INIT p` and `TRANS next(p)`, but is more efficient.
- Risk of defining *inconsistent models* (`INIT p & !p`).





## Assignments versus constraints

- Any `ASSIGN`-based specification can be easily rewritten as an equivalent constraint-based specification:

```
ASSIGN
  init(state) := {ready,busy};      INIT  state in {ready,busy}
  next(state) := ready;             TRANS next(state) = ready
  out := b0 + 2*b1;                 INVAR out = b0 + 2*b1
```

- The converse is not true: constraint

```
TRANS
  next(b0) + 2*next(b1) + 4*next(b2) =
                        (b0 + 2*b1 + 4*b2 + tick) mod 8
```

cannot be easily rewritten in terms of `ASSIGN`s.





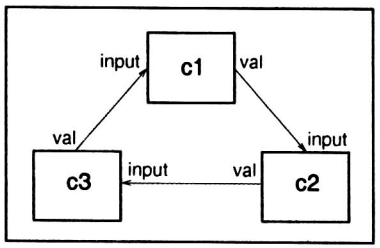
## Assignments versus constraints

- Models written in **assignment style**:
  - by construction, there is always *at least one initial state*;
  - by construction, all states have *at least one next state*;
  - *non-determinism is apparent* (unassigned variables, set assignments...).
- Models written in **constraint style**:
  - `INIT` constraints *can be inconsistent*:
    - inconsistent model: no initial state,
    - any specification (also `SPEC 0`) is vacuously true.
  - `TRANS` constraints *can be inconsistent*:
    - the transition relation is not total (there are deadlock states),
    - NuSMV detects and reports this case.
  - *non-determinism is hidden* in the constraints:
    `TRANS (state = ready & request) -> next(state) = busy`

# Synchronous composition

☞ By default, composition of modules is **synchronous**:
*all modules move at each step.*

```
MODULE cell(input)
  VAR
    val : {red, green, blue};
  ASSIGN
    next(val) := {val, input};

MODULE main
  VAR
    c1 : cell(c3.val);
    c2 : cell(c1.val);
    c3 : cell(c2.val);
```








# Synchronous composition

A possible execution:

| *step* | *c1.val* | *c2.val* | *c3.val* |
|---|---|---|---|
| 0 | **red** | **green** | **blue** |
| 1 | red | **red** | **green** |
| 2 | **green** | red | green |
| 3 | green | red | green |
| 4 | green | red | **red** |
| 5 | **red** | **green** | red |
| 6 | red | **red** | red |
| 7 | red | red | red |
| 8 | red | red | red |
| 9 | red | red | red |
| 10 | red | red | red |





# Asynchronous composition

☞ **Asynchronous** composition can be obtained using keyword `process`.

☞ In asynchronous composition *one process moves at each step.*

☞ Boolean variable `running` is defined in each process:

- it is true when that process is selected;
- it can be used to guarantee a fair scheduling of processes.

```
MODULE cell(input)
  VAR
    val : {red, green, blue};
  ASSIGN
    next(val) := {val, input};
  FAIRNESS
    running

MODULE main
  VAR
    c1 : process cell(c3.val);
    c2 : process cell(c1.val);
    c3 : process cell(c2.val);
```





# Asynchronous composition

A possible execution:

| *step* | *runnig* | *c1.val* | *c2.val* | *c3.val* |
|---|---|---|---|---|
| 0 | - | **red** | **green** | **blue** |
| 1 | c2 | red | **red** | blue |
| 2 | c1 | **blue** | red | blue |
| 3 | c1 | blue | red | blue |
| 4 | c2 | blue | red | blue |
| 5 | c3 | blue | red | **red** |
| 6 | c2 | blue | **blue** | red |
| 7 | c1 | blue | blue | red |
| 8 | c1 | **red** | blue | red |
| 9 | c3 | red | blue | **blue** |
| 10 | c3 | red | blue | blue |



28-1

# NuSMV resources

- NuSMV home page:
  - `http://nusmv.irst.itc.it/`
- Mailing lists:
  - `nusmv-users@irst.itc.it` (public discussions)
  - `nusmv-announce@irst.itc.it` (announces of new releases)
  - `nusmv@irst.itc.it` (the development team)
  - to subscribe: `http://nusmv.irst.itc.it/mail.html`
- Course notes and slides:
  - `http://nusmv.irst.itc.it/courses/`

6 Dieter Hutter, Werner Stephan
*DFKI Saarbrücken, Germany*

**Course: Formal Methods with INKA/VSE**

# Formal Methods with INKA/VSE

Calculemus Autumn School
Pisa

2002

Dieter Hutter, Werner Stephan,
DFKI Saarbrücken



- Mathematical Models
- Formal Methods
- Temporal Logic
- Temporal Logic of Actions (TLA)



## Mathematical Models





Sets, Relations, Functions
more specific: Axiomatic Set Theory

- $A = \{ a_0, a_1, a_2, \ldots \}$ finite/infinite (basic) sets
  - $= \{ 0, 1, 2, \ldots \}$ example: natural numbers
- $A_0 \times A_1 \times \ldots \times A_{n-1}$ cross product (n-fold)
  - $< a_0, a_1, \ldots, a_{n-1} > \in A_0 \times A_1 \times \ldots \times A_{n-1}$ n-tuple
- $R \subseteq A_0 \times A_1 \times \ldots \times A_{n-1}$ n-ary relation
- $f : A_0 \times A_1 \times \ldots \times A_{n-1} \rightarrow B$ n-ary (total) function





Mathematical Description of
Syntax and Semantics

- Syntax: description of the system
  - Each description is finite (finite set)
  - (program-) texts: grammars
  - structural description: finite (mathematical) structures ⇒ represented in the computer
  - tools: visualization
  - syntactic analysis: is the description correct? ⇒ syntax analysis, "typecheck"





Mathematical Description of
Syntax and Semantics

- Semantics: Description of the behavior of a given system
  - (typically) infinite mathematical structures
  - mathematical methods for the description of semantic assignments

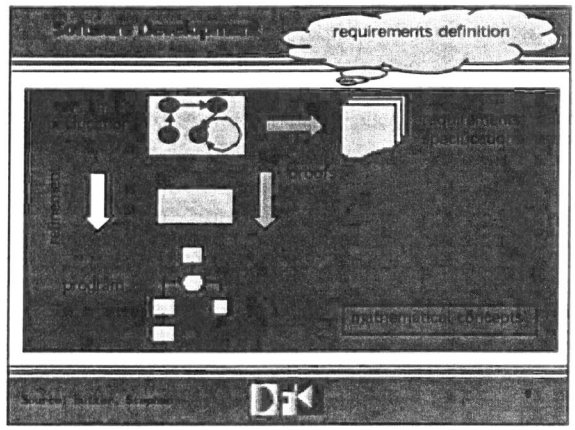
  Analysis: The proof of semantic properties (of a behavior) is the central concern of "Formal Methods" ⇒ Uniqueness

- Finite State Transition Systems (Automata)
  - Finite Transductors : basic concepts
  - State-Charts
  - Communicating Sequential Processes (CSP)
- Programming Languages (Semantics)
  - concurrent imperative programs
- Data Structures : later



# Formal Methods








Predictable safety properties

- exclusion of unsafe states
- failsafe performance
  ⇒ correct functioning in spite of failures
- (Data) Security
  ⇒ protection against abuse (attacks)

Unpredicted Behavior



## Description Techniques

The software engineer as Creator?
**Objectifying through restriction of expressiveness**

- Structured text
- Enrichment through diagrams / formulas
- Formal syntax
- Formal (syntax and) semantics
  ⇒ basis for analysis

objectifying



## Structured Text

*Transaction Collection and Posting*

**Collection**

A specification must be published for how system operators must interface with each other for data collection and transmission.

- A cycle must be run regularly to collect, process and report product activity.
- The system must be able to process files of "on us" and "not on us" transactions and transmit the not on us transactions to the appropriate processor.

**Validation and Update**

The following functions are required:

*Transaction Exceptions*

- Transactions with errors will be reported back to acquirers, as appropriate.
- Errors will result form an unreadable BIN, an invalid MAC or if the transaction is found to be duplicate.
- Transactions with errors will be recorded as not settled, and sent to the system operator/acquirer in the reconciliation report.

*Transaction Validation*

...

## Diagrams/ Formulas

5.4.2 Beschreibung der Transaktion (Version 1)

Wir unterscheiden bei der TP-Transaktion die folgenden drei Phasen:

(1) Übermittlung von $B_{SLA}$

(2) Bestätigung Empfang von $B_{SLA}$

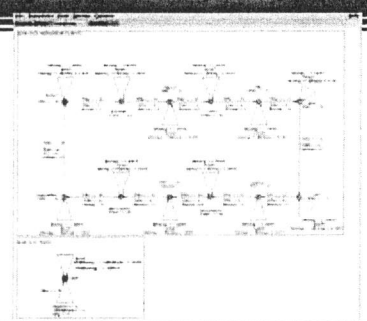
(3) Erhöhen/Verringern des realen elektronischen Guthabens des Handlers um $(\sum B_{SV,A})$, und Verringern (Erhöhen) des realen elektronischen Guthabens des Acquirers um $(\sum B_{SV,A})$, formal: $K'_{SH} = K_{SH} + (\sum B_{SV,A})$, $B'_{SH} = B_{SH} \setminus B_{SH,A}$; $K'_A = K_A - (\sum B_{SV,A})$, $B'_A = B_A \cup B_{SH,A}$



## Formal Syntax (ARIS)



## Formal Syntax + Semantics (UPPAAL)





## Formal Methods in the Software Development Cycle

Controlling the Complexity

- System Description + Requirements (Specification)
  - Abstraction
  - Mathematical Objectifying
- Technical realization (refinements)
  - Proof of correctness
  - Intermediate stages
- Decomposition into independent units (modularization)



## Formal Methods in the Software Development Cycle

System Description and Requirements

- Abstraction: abandon from irrelevant details of the technical (Software, Hardware) realization
  ⇒ Decisive mistakes are often made in earlier stages of the development (concept phase).
- Objectifying
  - Mathematical models
  - Formal description and proof of intended properties
  ⇒ *Computation* (cf. other disciplines of engineering)



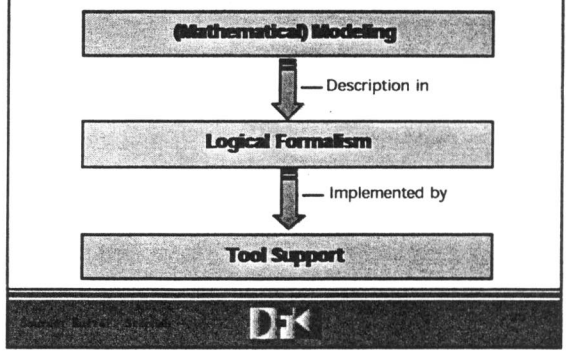
## Formal Methods in the Software Development Cycle

From Requirements Specification to Implementation

- Inclusion of technical concepts
  - Control structures, definite data structures, communication
  - Correctness proof (verification)
  ⇒ *All Inputs*
- Step-by-step strategies
  - Several abstract intermediate levels
  - Only the last level corresponds to the technical platform

Modularization: Decomposition into sub-problems

- Formal Basics of Modularization
  - Independent Subsystems
  - Mechanisms of cooperation
  - Correctness
- Modular Development
  - Modular specification und validation




## Temporal Logic

- Semantic interpretation of (certain) symbols depends on *worlds*, $w \in W$.
  - ⇒ Distinguish statements that are incidentally true (in some worlds) from statements that are *necessarily* true (in all reachable/visible worlds).
- Formulas $\varphi$ are interpreted in worlds: $[\varphi]_{val\ w}$, where $val : V \to D$ is a value assignment for *global* (or rigid) variables.
  - ⇒ Interpretation over a fixed domain $D$.
  - ⇒ In some modal systems domains (quantification!) depend on worlds.

- *Reachability* relation $R \subseteq W \times W$.
- Syntax: Extend rules (for propositional or first-order) formulas by : If $\varphi$ is a *formula* so *is* $\Box\varphi$.
  - ⇒ "$\varphi$ is necessarily true."
- $(W, R), val \models_w \Box\varphi \Leftrightarrow$ for all worlds $w'$ satisfying $wRw'$ : $(W, R), val \models_{w'} \varphi$
- The dual (modal) operator : $\Diamond\varphi \leftrightarrow \neg\Box\neg\varphi$
  - ⇒ "$\varphi$ is possibly true."

$(W, R), val \models \varphi \Leftrightarrow (W, R), val \models_w \varphi$ for all $w$

$(W, R) \models \varphi \Leftrightarrow (W, R), val \models \varphi$ for all $val$

$\models \varphi \Leftrightarrow (W, R) \models \varphi$ for all structures $(W, R)$

## Modal Logic

- Valid formulas ( $\Box\varphi$ ):

  - $\Box(\varphi \wedge \psi) \leftrightarrow \Box\varphi \wedge \Box\psi$

  - $(\Box\varphi \vee \Box\psi) \rightarrow \Box(\varphi \vee \psi)$

  - $\Box(\varphi \rightarrow \psi) \rightarrow \Box\varphi \rightarrow \Box\psi$



## Modal Logic

- Rule of *necessitation* (Hilbert-style axiomatization):

$$\frac{\varphi}{\Box\varphi}$$

  $\Rightarrow$ Not: $\varphi \rightarrow \Box\varphi$

- *Barcan* formula (quantification over global variables using a fixed domain):

  $(\forall v)\,\Box\varphi \rightarrow \Box(\forall v)\,\varphi$



## Modal Logic

- Main topic of modal logic: Characterization of properties of $R$ by (modal) axioms.
  $\Rightarrow$ reflexivity, transitivity, symmetry

- Reflexivity: For all structures $(W, R)$, where $R$ is reflexive $\Box\varphi \rightarrow \varphi$ is a valid formula, that is $(W, R) \models \Box\varphi \rightarrow \varphi$, (and vice versa).

- Transitivity: For all structures $(W, R)$, where $R$ is transitive $\Box\varphi \rightarrow \Box\Box\varphi$ is a valid formula (and vice versa).



## Temporal Structures

- Application of temporal logic to formal development:
  Compution sequences (of programs or abstract systems)
  - abstract specifications: *tlspec* a formula that describes the behaviour of a system
  - programs: *tlprog* a formula that describes the execution sequences of a (concurrent) program
  - properties (of systems and programs): *tlspec* $\rightarrow$ *prop*, *tlprog* $\rightarrow$ *prop*
  - refinement (implementation): *tlprog* $\rightarrow$ *tlspec*, $tlspec_0 \rightarrow tlspec_1$



## Temporal Structures

- Example (property): „The *local* (flexible, state dependent) variable $y$ always becomes 0 after $x$ has been set to 0.

  $\ldots \rightarrow [x = 0] \rightarrow \ldots \rightarrow [y = 0] \rightarrow \ldots$

  $\rightarrow [x = 0] \rightarrow \ldots \rightarrow [y = 0] \rightarrow \ldots$

  Formalization: $\Box(x = 0 \rightarrow \Diamond y = 0))$



## Temporal Structures

- Discrete versus continuous:

  - discrete: time as a totally ordered set $(S, <)$ is isomorphic to $(\mathbb{N}, <)$.
    $\Rightarrow$ used in most temporal logics (and this course)

  - continuous: dense structures (isomorphic to $(\mathbb{R}, <)$)
    $\Rightarrow$ sometimes used for quantitative analysis in real-time systems

- Branching versus linear time:
  - linear-time: at each moment (state) there is only one future (= sequence of future states)
    ⇒ in this course: Linear Temporal Logic (LTL)
  - branching-time: at each moment (state) there can be a split into several futures
    ⇒ tree-like structures
    ⇒ excurs

- Time structure: For a set of *states* $s \in S$ consider infinite sequences of states $\mathbf{s} : , \rightarrow S$. We use $\mathbf{s}_i$ for $\mathbf{s}(i)$.
  $\mathbf{s}|_i$ denotes the $i^{th}$ *suffix* of $\mathbf{s}$.
  Properties: discrete, there exists an initial state with no predecessors, infinite into the future
- Linearity: Each possible computation of a possibly indeterministic system is considered separately.
  ⇒ Implicit quantification over all computations
  ⇒ We are reasoning about a fixed computation but we do not know which it is. The next state cannot be "computed" like in formal execution.

- Example: The program $x := x + 1 \parallel y := y + 1$ generates the sequences:

  $[x = 0, y = 0] \rightarrow [x = 1, y = 0] \rightarrow [x = 1, y = 1]$
  and
  $[x = 0, y = 0] \rightarrow [x = 0, y = 1] \rightarrow [x = 1, x = 1]$

- Knowing $x = 0 \wedge y = 0$ what is the next state? In linear time logics we cannot consider *both* alternatives.

Semantics (only future)

- In addition to the variables $v \in V$ some symbols have a fixed meaning in all states. Use a first-order structure (model) $M$ to interpret these symbols.
- Satisfaction: $\mathbf{s}, (M, val) , \varphi$
- For first-order formulas $\phi$ :
  $\mathbf{s}, (M, val) , \varphi \Leftrightarrow [\phi]_{(M, val), \mathbf{s}_0} = t$
  First-order formulas are interpreted in the initial state.

Semantics of (some) basic future temporal operators (( $M$, $val$ ) omitted):

- always:
  $\mathbf{s} , \Box\varphi \Leftrightarrow \mathbf{s}|_i , \varphi$ for all $i$
- atnext:
  $\mathbf{s} , \bigcirc\varphi \Leftrightarrow \mathbf{s}|_1 , \varphi$
- unless (waiting for):
  $\mathbf{s} , (\varphi \; W \; \psi) \Leftrightarrow \mathbf{s}|_i , \varphi$ for all $i$ or
  for some $j$ $\mathbf{s}|_j , \psi$ and
  $\mathbf{s}|_k , \varphi$ for all $k < j$

Semantics of (some) derived future temporal operators:

- eventually:
  $\mathbf{s} , \Diamond\varphi \Leftrightarrow \mathbf{s}|_i , \varphi$ for some $i$
- until:
  $\mathbf{s} , (\varphi \; U \; \psi) \Leftrightarrow$ for some $j$ $\mathbf{s}|_j , \psi$ and
  $\mathbf{s}|_k , \varphi$ for all $k < j$
- can be defined by the basic operators.

Additional important valid formulas:

$\bullet \neg\varphi \leftrightarrow \neg \bullet\varphi$

$\bullet\varphi \rightarrow \bullet\varphi$ $\qquad \varphi \rightarrow \bullet\varphi$

$\psi \mathcal{U} \varphi \rightarrow \bullet\varphi$ $\qquad \bullet\varphi \rightarrow \bullet\varphi$

$\bullet\varphi \rightarrow \bullet\varphi$ $\qquad \bullet\varphi \rightarrow \bullet\bullet\varphi$

$\bullet\bullet\varphi \leftrightarrow \bullet\varphi$ $\qquad \bullet\bullet\varphi \leftrightarrow \bullet\varphi$

$\bullet\bullet\varphi \leftrightarrow \bullet\bullet\varphi$ $\qquad \bullet\bullet\varphi \leftrightarrow \bullet\bullet\varphi$

$(\bullet\psi) \mathcal{U} (\bullet\varphi) \leftrightarrow \bullet(\psi \mathcal{U} \varphi)$

$\bullet(\varphi \vee \psi) \leftrightarrow \bullet\varphi \vee \bullet\psi$ $\qquad (\varphi \wedge \psi) \mathcal{U} \theta \leftrightarrow (\varphi \mathcal{U} \theta) \wedge (\psi \mathcal{U} \theta)$

$\varphi \mathcal{U} (\theta \vee \psi) \leftrightarrow (\varphi \mathcal{U} \theta) \vee (\varphi \mathcal{U} \psi)$

Additional important valid formulas:

$\bullet(\varphi \vee \psi) \leftrightarrow \bullet\varphi \vee \bullet\psi$ $\qquad \bullet(\varphi \wedge \psi) \leftrightarrow \bullet\varphi \wedge \bullet\psi$

$\bullet(\varphi \rightarrow \psi) \leftrightarrow \bullet\varphi \rightarrow \bullet\psi$

$\bullet(\varphi \wedge \psi) \rightarrow \bullet\varphi \wedge \bullet\psi$ $\qquad (\varphi \mathcal{U} \theta) \vee (\psi \mathcal{U} \theta) \rightarrow (\varphi \vee \psi) \mathcal{U} \theta$

$\varphi \mathcal{U} (\theta \wedge \psi) \rightarrow (\varphi \mathcal{U} \theta) \wedge (\varphi \mathcal{U} \psi)$

$\bullet(\varphi \rightarrow \psi) \rightarrow (\bullet\varphi \rightarrow \bullet\psi)$ $\qquad \bullet(\varphi \rightarrow \psi) \rightarrow (\bullet\varphi \rightarrow \bullet\psi)$

$\bullet(\varphi \rightarrow \psi) \rightarrow ((\varphi \mathcal{U} \theta) \rightarrow (\psi \mathcal{U} \theta))$ $\qquad \bullet(\varphi \rightarrow \psi) \rightarrow ((\theta \mathcal{U} \varphi) \rightarrow (\theta \mathcal{U} \psi))$

$\bullet\varphi \leftrightarrow (\varphi \vee \bullet\bullet\varphi)$ $\qquad \bullet\varphi \leftrightarrow (\varphi \wedge \bullet\bullet\varphi)$

$\varphi \mathcal{U} \psi \leftrightarrow \psi \vee (\varphi \wedge \bullet(\varphi \mathcal{U} \psi))$

Semantics (also past):

- Consider $s, (M, val) \models_i \varphi$ and define
  $s, (M, val) \models \varphi \Leftrightarrow s, (M, val) \models_0 \varphi$
  (the anchored version)
- Rewrite semantics of future operators using $s \models_i \varphi$.
  For example:
  $s \models_j \bullet\varphi \Leftrightarrow s \models_i \varphi$ for all $i \geq j$

Semantics of (some) basic past temporal operators ($M, val$) omitted):

- uptilnow (has-always-been):
  $s \models_i \bullet\varphi \Leftrightarrow s \models_j \varphi$ for all $0 \leq j \leq i$
- atlast (previous):
  $s \models_i \bullet\varphi \Leftrightarrow i > 0$ and $s \models_{i-1} \varphi$
- backto:
  $s \models_i (\varphi B \psi) \Leftrightarrow s \models_j \varphi$ for all $0 \leq j \leq i$ or
  for some $0 \leq l \leq i$ $s \models_l \psi$ and
  $s \models_k \varphi$ for all $l < k \leq i$

Need past operators:

- Safety: A *safety* formula is a formula equivalent to a formula of the form $\Box\varphi$, where $\varphi$ is a past formula.
- Guarantee: A *guarantee* formula is a formula equivalent to a formula of the form $\Diamond\varphi$, where $\varphi$ is a past formula.
- Obligation: An *obligation* formula is a formula equivalent to a formula of the form $\bigwedge_{0 \leq i \leq n} (\Box\varphi_i \vee \Diamond\psi_i)$, where the $\varphi_i$ and $\psi_i$ are past formulas.

- Response: A *response* formula is a formula equivalent to a formula of the form $\Box\Diamond\varphi$, where $\varphi$ is a past formula.
- Persistence: A *persistence* formula is a formula equivalent to a formula of the form $\Diamond\Box\varphi$, where $\varphi$ is a past formula.
- Reactivity: A *reactivity* formula is a formula equivalent to a formula of the form $\bigwedge_{0 \leq i \leq n} (\Box\Diamond\varphi_i \vee \Diamond\Box\psi_i)$, where the $\varphi_i$ and $\psi_i$ are past formulas.

# The Temporal Logic of Actions (TLA)

- Imperative programs transform *states* (*St*):

  $C[c] \subseteq St \times St$

  - $s : X \to D$, where $X$ is a set of *program variables* and $D$ is a domain of values
  - $C[c]$ is typically a *partial function*. Desired property: *termination* for certain initial states.
  - Logics: Floyd-Hoare Logic, Dynamic Logic.
  - No intermediate states.

- Concurrent programs ($c_0 \parallel c_1$), distributed and reactive systems: *infinite* sequences of states:
  $\langle s_0, s_1, s_2, \ldots \rangle$ (*behaviours*)

  - No termination (operating systems, control systems)
  - A *transition* $\langle s_i, s_{i+1} \rangle$ is an execution step (of one of the components).
  - TLA: specifying behaviours, reasoning about behaviours.

- Notation
  - $St^\omega$ the set of infinite sequences of states $\langle s_0, s_1, s_2, \ldots \rangle$
  - $St^n$ the set of sequences of length $n$: $\langle s_0, \ldots, s_{n-1} \rangle$

    $St^0 = \{\langle\rangle\}$
  - $St^* = \bigcup_{n \in \omega} St^n$
  - For $\mathbf{s} \in St^n \cup St^\omega$ and $i < n : \mathbf{s}_i = \mathbf{s}_i$ (the i'th element of $\mathbf{s}$)

- For $\mathbf{s} \in St^n \cup St^\omega$:

  $\mathbf{s}|_n = \langle s_j, s_{j+1}, \ldots \rangle$ for $\mathbf{s} \in St^\omega$
  and $\mathbf{s}|_n = \langle s_j, \ldots, s_{n-1} \rangle$ for $\mathbf{s} \in St^n$ and $j \leq n$
- For $\mathbf{s}_1 \in St^n$ and $\mathbf{s}_2 \in St^\omega \cup St^m$:

  $\mathbf{s}_1 \cdot \mathbf{s}_2 = \langle s_{1_0}, \ldots, s_{1_{n-1}}, s_{2_0}, \ldots, s_{2_1}, \ldots \rangle$ for $\mathbf{s}_2 \in St^\omega$
  and $\mathbf{s}_1 \cdot \mathbf{s}_2 = \langle s_{1_0}, \ldots, s_{1_{n-1}}, s_{2_0}, \ldots, s_{2_{m-1}} \rangle$
  for $\mathbf{s}_2 \in St^m$
- *Stuttering equivalence* is the finest equivalence relation $\sim \subseteq St^\omega \times St^\omega$, such that for all $\mathbf{s}_1, \mathbf{s}_2 \in St^\omega$ and all $s \in St : \mathbf{s}_1 \cdot \langle s \rangle \cdot \mathbf{s}_2 \sim \mathbf{s}_1 \cdot \langle s, s \rangle \cdot \mathbf{s}_2$

- A *language* (for TLA) is given by $\mathcal{L} = (Sig, V, X)$, where
  - $Sig = (Z, F, R)$ is a first-order signature
  - $V$ is a $Z$-sorted set of *rigid* (logical, global) variables
  - $X$ is a $Z$-sorted set of *flexible* (local) variables, such that

    $V \cap X \cap \bigcup_{z \in Z} F_{\to z} = \emptyset$
  - Rigid variables are interpreted (evaluated) under states $s$.

- For a given set of flexible variables $X$, $X' = \{x' \mid x \in X\}$ is the corresponding set of *primed* variables.
  - $X'$ is not just an additional set of variables but an additional set $Y$ *together* with an one-to-one mapping $prime : X \to Y$.
  - Primed variables refer to the second state of a state transition.
  - Compare $Z$-notions.
  - The set of variables (of $\mathcal{L}$) is $U = V \cup X \cup X'$.

- A *transition function* is a first-order term (expression) over *Sig* and $U$, $Tm(Sig, U)$.
  - Transition functions are evaluated (to a value from some domain) under a pair of states $\langle s, s' \rangle$ (two elements interval).
  - Example:

    $max(x', gcd(y, z))$

- A *transition predicate* is a first-order formula over *Sig* and $U$, $Fm(Sig, U)$.
  - First-order quantification over $U = V \cup X \cup X'$ allowed!
  - Transition predicates are evaluated (to a truth value) under a pair of states. They describe steps of a computation (*actions*).
  - Example:

    $z' = max(x', gcd(y, z))$

- A *state function* is a transition function that contains no primed variables.
  - First-order term over *Sig* and $U \setminus X'$.
  - State-functions are evaluated under a state.
  - Example:

    $gcd(y, z)$

- A *state predicate* is a transition predicate that contains no primed variables.
  - First-order formula over *Sig* and $U \setminus X'$.
  - State-predicates are evaluated under a state. They describe properties of states (assertions).
  - Example:

    $x \geq y$

- A *constant function* is a state function with no flexible variables.
- A *constant predicate* is a state predicate with no flexible variables.

## Syntax of TLA

- The set of (simple) *behaviour predicates* (TLA-formulas) is the smallest set such that:
  - If $\Pi$ is a state predicate, then $\Pi$ is a behaviour predicate.
  - If $A$ is a transition predicate and $\phi$ is a state function, then $\cdot[A]_\phi$ is a behaviour predicate.
  - If $\Phi$ is a behaviour predicate, then $\neg\Phi$ is a behaviour predicate.
  - If $\Phi_0, \Phi_1$ are behaviour predicates, then $(\Phi_0 \wedge \Phi_1)$, $(\Phi_0 \vee \Phi_1)$, $(\Phi_0 \rightarrow \Phi_1)$, and $(\Phi_0 \leftrightarrow \Phi_1)$ are behaviour predicates.

## Syntax of TLA

- Behaviour predicates ctd.
  - If $\Phi$ is a behaviour predicate, then $\cdot\Phi$ is a behaviour predicate.
  - If $v \in V$ and $\Phi$ is a behaviour predicate, then $\exists v \cdot \Phi$ and $\forall v \cdot \Phi$ are behaviour predicates.

## Syntactic Notions

- For transition functions (state functions) $\phi$ and transition predicates (state predicates) $\Pi$ the sets of *free variables*, $FV_{tr}(\phi)$, $FV_{tr}(\Pi) \subseteq U$ are defined as in first-order logic.
- For transition functions $\phi$ (state functions) and transition predicates (state predicates) $\Pi$ the *substitution* of a function $\psi$ for all free occurrences of $u \in U$, $\phi[\psi/u]$ and $\Pi[\psi/u]$, respectively, is defined as in first-order logic.

## Syntactic Notions

- For state function $\phi$ and state predicates $\Pi$

$$\phi' := \phi[x'_1/x_1] \ldots [x'_n/x_n] \text{ and}$$
$$\Pi' := \Pi[y'_1/y_n] \ldots [y'_m/y_m],$$

where

$$FV_{tr}(\phi) \cap X = \{x_1, \ldots, x_n\} \text{ and}$$
$$FV_{tr}(\Pi) \cap X = \{y_1, \ldots, y_m\}$$

## Syntactic Notions

- For behaviour predicates $\Phi$ the set of *free variables*, $FV_{temp}(\Phi) \subseteq V \cup X$ is defined recursively by
  - $FV_{temp}(\Pi) = FV_{tr}(\Pi)$
  - $FV_{temp}(\cdot[A]_\phi) = \{x \mid x \in X \text{ and } x \in FV_{tr}(A) \text{ or } x' \in FV_{tr}(A)\} \cup FV_{tr}(\phi)$
  - $FV_{temp}(\neg\Phi) = FV_{temp}(\Phi)$
  - $FV_{temp}(\Phi_0 \cdot \Phi_1) = FV_{temp}(\Phi_0) \cup FV_{temp}(\Phi_1)$
  - $FV_{temp}(\cdot\Phi) = FV_{temp}(\Phi)$
  - $FV_{temp}(\exists v \cdot \Phi) = FV_{temp}(\Phi) \setminus \{v\}$

## Semantics of TLA

State functions and predicates:

- $[\phi]' : Val \times St \rightarrow D$
  $[\phi]'_{vals} = [\phi]^{tr}_{vals,s}$
- $[\Pi]'_{vals} : Val \times St \rightarrow \{t, f\}$
  $[\Pi]'_{vals} = [\Pi]^{tr}_{vals,s}$

## Semantics of TLA

Transition functions and predicates:

- $[\phi]^{tr} : Val \times St \times St \to D$

  $[\phi]^{tr}_{val\, s_0, s_1} = [\phi]_{E\,(val, s_0, s_1)}$

- $[\Phi]^{tr} : Val \times St \times St \to \{t, f\}$

  $[\Phi]^{tr}_{val\, s_0, s_1} = [\Phi]_{E\,(val, s_0, s_1)}$



## Semantics of TLA

- For a language $\mathcal{L} = (Sig, V, X)$ as above let $M = (D, I)$ be a $Sig$-model. $Val = [V \to D]$ (the set of all value assignments)
- For states $s_0, s_1 \in St$ and value assignment $val \in Val$ the combined assignment $E\,(val, s_0, s_1) : U \to D$ is defined by:

$$E\,(val, s_0, s_1)\,(x) = s_0(x)$$
$$E\,(val, s_0, s_1)\,(x') = s_1(x)$$
$$E\,(val, s_0, s_1)\,(v) = val\,(v)$$



## Semantics of TLA

Semantics of behaviour predicates:

$[\Phi]^b : Val \times St^{\omega} \to \{t, f\}$

- $[\,\Pi\,]^b_{val\, s} = [\,\Pi\,]^{tr}_{val\, s_0}$
- $[\,\Box[\,A\,]_\phi\,]^b_{val\, s} = t \Leftrightarrow$ for all $l \geq 0$ :

  $[A \vee \phi = \phi']^{tr}_{val\, s_l, s_{l+1}} = t$
- $[\neg\Phi\,]^b_{val\, s} = t \Leftrightarrow [\Phi]^b_{val\, s} = f$
- $[\Phi_0 \wedge \Phi_1\,]^b_{val\, s} = t \Leftrightarrow$

  $[\Phi_0]^b_{val\, s} = t$ and $[\Phi_1\,]^b_{val\, s} = t$



## Semantics of TLA

- $[\,\Box\Phi\,]^b_{val\, s} = t \Leftrightarrow$ for all $j \geq 0$ : $[\Phi]^b_{val\, s_j} = t$
- $[\exists v.\ \Phi]^b_{val\, s} = t \Leftrightarrow$ for some $val' =_v val$ :

  $[\Phi]_{val', s} = t$ where

  $val' =_v val : \Leftrightarrow$ for all $v \neq v$ : $val'(u) = val\,(u)$



## Abbreviations

- Stuttering indices : $\Box[A\,]_{\phi_1, \ldots, \phi_n} : \Leftrightarrow \Box[A\,]_{\phi_1} \wedge \ldots \wedge \Box[A\,]_{\phi_n}$

  $\Rightarrow$ typically $\phi_i = x_i$
- $\langle A \rangle_{\phi_1, \ldots, \phi_n} : \Leftrightarrow \neg\Box[\neg A\,]_{\phi_1, \ldots, \phi_n}$
- Eventually: $\Diamond\Phi : \Leftrightarrow \neg\Box\neg\Phi$
- Enabled predicate:

  $Enabled\ (A) : \Leftrightarrow \exists x'_1 \ldots x'_n . A$, where

  $FV_{tr}(A) \cap X' = \{x'_1, \ldots, x'_n\}$
- $\Phi \rightsquigarrow \Psi : \Leftrightarrow \Box(\Phi \to \Diamond\Psi)$ (leadsto)



## Invariance under Stuttering

Theorem:

If $\mathbf{s}$ and $\mathbf{s}'$ are stuttering equivalent,

then $[\Phi]^b_{val\, \mathbf{s}} = [\Phi]^b_{val\, \mathbf{s}'}$.

$\Rightarrow$ Behaviour predicates (as defined above) cannot distinguish between stuttering equivalent behaviours.

## A Simple Clock

- Hour clock (HC) : a clock that displays (only) hours.
- Data:
  - *Nat* (natural numbers)
  - *mod* : $Nat \times Nat \rightarrow Nat$
    *mod* (*n*, *v*) : signed remainder after division by non-zero *v*
  - *in_hourp* : *Nat*
    *in_hourp* (*v*) : *v* is an admissible (display) value

## A Simple Clock

- A behaviour
  $[hr = 11] \rightarrow [hr = 12] \rightarrow [hr = 1] \rightarrow [hr = 2] \rightarrow \ldots$ where
  - $hr \in X$ (a flexible variable).
  - $[hr = 11]$ represents a state $s \in St$, where $s(hr) = 11$.
    The remaining flexible variables have arbitrary values and change in an arbitrary way.

    If $s_0 = [hr = 11]$ and $s_1 = [hr = 12]$ and $s_0 \rightarrow s_1$,
    then $s_0(x) \neq s_1(x)$ is possible.

## A Simple Clock

- A tick of the clock is given by the transition predicate (action)
  $HC\_nxt :: = hr' = (mod(hr, 12) + 1)$,
  where 12 stands for (the constant function)
  *succ* (*succ* ( ... *succ* (0) ... ))
- Describing the behaviour of the clock:
  $in\_hourp(hr) \wedge \blacksquare[HC\_nxt]_{hr}$
  - *in_hourp* (*hr*) describes the initial state.
  - $\blacksquare[HC\_nxt]_{hr}$ describes the state transitions.
- A (safety) property of the hour clock: $\blacksquare in\_hourp(hr)$

## A Simple Clock

- Hour minute clock (MC): a clock that displays hours and minutes.
- Data:
  - *Nat* (natural numbers)
  - *mod* : $Nat \times Nat \rightarrow Nat$
    *mod* (*n*, *v*) : signed remainder after division by non-zero *v*.
  - *in_hourp* : *Nat*
    *in_hourp* (*v*) : *v* is an admissible (display) value of hours.
  - *in_minutep* : *Nat*
    *in_minutep* (*v*) : *v* is an admissible (display) value of minutes.

## A Simple Clock

- A behaviour

| | | | |
|---|---|---|---|
| $hr = 11$ | $hr = 11$ | $hr = 12$ | $hr = 12$ |
| $min = 58$ | $min = 59$ | $min = 0$ | $min = 1$ |

where
- $hr, min \in X$ (flexible variables)
- $hr = 11$
  $min = 58$ represents a state $s \in St$,
  where $s(hr) = 11$ and $s(min) = 58$

## A Simple Clock

- A tick of the clock is given by the transition predicate (action)
  $MC\_nxt \vee MC\_min$, where
  $MC\_nxt :: = min = 59 \wedge HC\_nxt \wedge min' = 0$ and
  $MC\_min :: = min \neq 59 \wedge min' = (min + 1) \wedge Unchanged(hr)$.
- Describing the behaviour of the hour clock:
  $(in\_hourp(hr) \wedge in\_minutep(min)) \wedge$
  $\blacksquare[MC\_nxt \vee MC\_min]_{hr,min}$
  - $(in\_hourp(hr) \wedge in\_minutep(min))$ describes the initial state.
  - $\blacksquare[MC\_nxt \vee MC\_min]_{hr,min}$ describes the state transitions.

## A Simple Clock

- *Weak Fairness* :

  $\Box((\Box\, executed) \vee (\Box\, impossible)) \leftrightarrow$

  $\neg\Box((\Box\, possible \wedge (\Box \neg executed))$

  - It cannot happen that an operation is (from some point on) constantly possible but never executed (taken).
  - Using $\Box\Box(\Phi_0 \wedge \Phi_1) \leftrightarrow \Box\Box\Phi_0 \wedge \Box\Box\Phi_1$ (linearity) we get $\Box\Box impossible \vee \Box\Box executed$.
  - $impossible : \neg Enabled(\langle A \rangle_{\phi_1,\ldots,\phi_n})$, executed : $\langle A \rangle_{\phi_1,\ldots,\phi_n}$
  - Weak Fairness in TLA (for an action $A$ ):

  $WF_{\phi_1,\ldots,\phi_n}(A) \leftrightarrow \Box\Box\neg Enabled(\langle A \rangle_{\phi_1,\ldots,\phi_n}) \vee \Box\Box \langle A \rangle_{\phi_1,\ldots,\phi_n}$



## A Simple Clock

- *Strong Fairness* :

  $\Box((\Box\, executed) \vee (\Box\, impossible)) \leftrightarrow$

  $\neg\Box((\Box\, possible \wedge (\Box \neg executed))$

  - It cannot happen that an operation is (from some point on) is infinitely often possible but never taken.
  - Rewriting yield: $(\Box\Box impossible) \vee (\Box\Box executed)$
  - Strong Fairness in TLA (for an action $A$ ):

  $SF_{\phi_1,\ldots,\phi_n}(A) \leftrightarrow \Box\Box\neg Enabled(\langle A \rangle_{\phi_1,\ldots,\phi_n}) \vee \Box\Box \langle A \rangle_{\phi_1,\ldots,\phi_n}$



## Fairness

- Problem: $\Box\Box hr = 12$ (*liveness*) does not hold for all behaviours (of the hour clock).
  - $[HC\_nxt]_{hr}$ allows for steps where $hr = hr'$.
  - $[hr = 11] \rightarrow [hr = 11] \rightarrow [hr = 11] \rightarrow [hr = 12] \rightarrow \ldots$ does not matter.
  - But

    $[hr = 11] \rightarrow [hr = 11] \rightarrow [hr = 11] \rightarrow [hr = 11] \rightarrow \ldots$

    yields a problem.
  - $\Box HC\_nxt$ is not a TLA-formula!



## Fairness

- Solution (first attempt): add

  $\Box\Box hr = 1 \wedge \ldots \wedge \Box\Box hr = 12$ or

  $\Box\Box\langle HC\_nxt \rangle_{hr}$ to the specification.
  - Adding arbitrary formulas (of this kind) leads to new problems.

    $\Rightarrow$ machine closed behaviours.
  - A general solution: introduce a kind of *scheduler*
    - has to be invariant under stuttering,
    - has to be machine closed,
    - has to be as liberal as possible.



## Fairness for Clocks

- Hour clock
  - $HC\_nxt$ is always enabled.
  - $WF_{hr}(HC\_nxt) \leftrightarrow \Box\Box\langle HC\_nxt \rangle_{hr}$
- Hour minute clock
  - $(MC\_nxt \vee MC\_min)$ is always enabled.
  - $WF_{hr,min}(MC\_nxt \vee MC\_min) \leftrightarrow$

    $\Box\Box\langle MC\_nxt \rangle_{hr,min} \vee \Box\Box\langle MC\_nxt \rangle_{hr,min}$



## Standard Form of TLA Specification

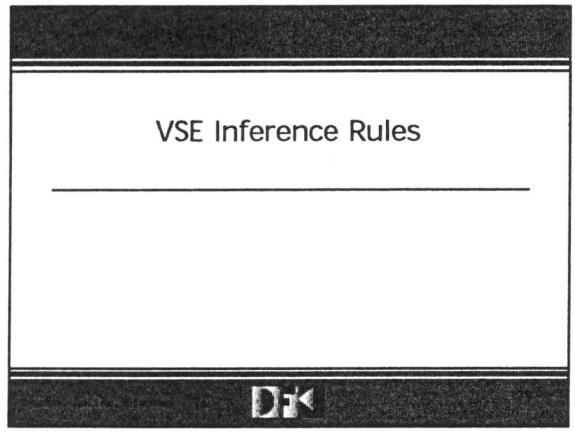
- $Init \wedge \Box[A]_{\phi_1,\ldots,\phi_n} \wedge F$
  - $Init$ is a state predicate.
  - $A$ is the next-state relation. Often $A = A_1 \vee \ldots \vee A_m$.
  - $\phi_1,\ldots,\phi_n$ is the stuttering index.

    Often $\phi_i = x_i$ for all $1 \leq i \leq n$.
  - $F$ describes constrains. $F$ is a conjunction of

    $WF_{\phi_1,\ldots,\phi_n}(A)$ and $SF_{\phi_1,\ldots,\phi_n}(A)$ formulas.

    In cases where $A = A_1 \vee \ldots \vee A_m$ each conjunct is $WF_{\phi_1,\ldots,\phi_n}(A_i)$ or $SF_{\phi_1,\ldots,\phi_n}(A_i)$ for $1 \leq i \leq n$.

## VSE Inference Rules



## Sequent Calculus Rules

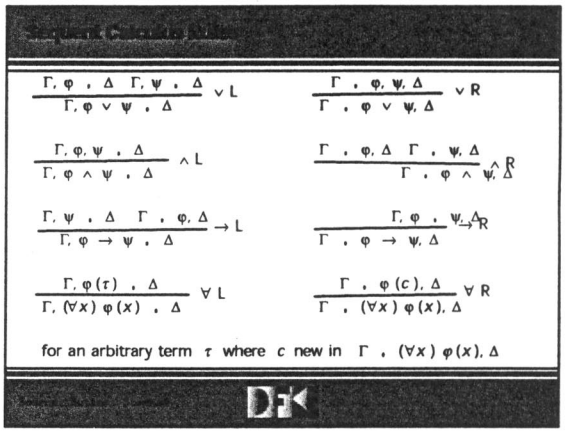
$$\frac{\Gamma, \varphi \ , \ \Delta \quad \Gamma, \psi \ , \ \Delta}{\Gamma, \varphi \vee \psi \ , \ \Delta} \vee L \qquad \frac{\Gamma \ , \ \varphi, \psi, \Delta}{\Gamma \ , \ \varphi \vee \psi, \Delta} \vee R$$

$$\frac{\Gamma, \varphi, \psi \ , \ \Delta}{\Gamma, \varphi \wedge \psi \ , \ \Delta} \wedge L \qquad \frac{\Gamma \ , \ \varphi, \Delta \quad \Gamma \ , \ \psi, \Delta}{\Gamma \ , \ \varphi \wedge \psi, \Delta} \wedge R$$

$$\frac{\Gamma, \psi \ , \ \Delta \quad \Gamma \ , \ \varphi, \Delta}{\Gamma, \varphi \rightarrow \psi \ , \ \Delta} \rightarrow L \qquad \frac{\Gamma, \varphi \ , \ \psi, \Delta}{\Gamma \ , \ \varphi \rightarrow \psi, \Delta} \rightarrow R$$

$$\frac{\Gamma, \varphi(\tau) \ , \ \Delta}{\Gamma, (\forall x)\, \varphi(x) \ , \ \Delta} \forall L \qquad \frac{\Gamma \ , \ \varphi(c), \Delta}{\Gamma \ , \ (\forall x)\, \varphi(x), \Delta} \forall R$$

for an arbitrary term $\tau$ where $c$ new in $\Gamma \ , \ (\forall x)\, \varphi(x), \Delta$



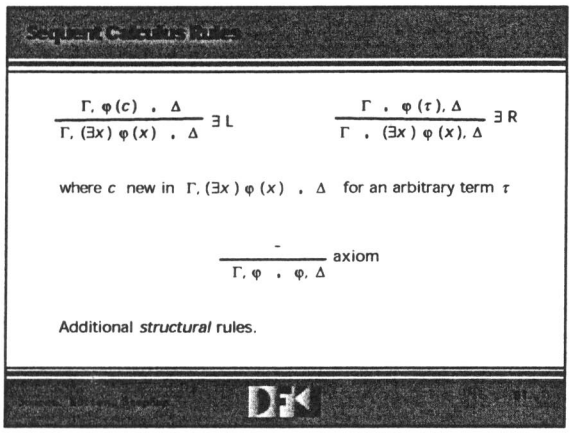
## Sequent Calculus Rules

$$\frac{\Gamma, \varphi(c) \ , \ \Delta}{\Gamma, (\exists x)\, \varphi(x) \ , \ \Delta} \exists L \qquad \frac{\Gamma \ , \ \varphi(\tau), \Delta}{\Gamma \ , \ (\exists x)\, \varphi(x), \Delta} \exists R$$

where $c$ new in $\Gamma, (\exists x)\, \varphi(x) \ , \ \Delta$ for an arbitrary term $\tau$

$$\frac{}{\Gamma, \varphi \ , \ \varphi, \Delta} \text{ axiom}$$

Additional *structural* rules.



## Temporal Rules in VSE-II

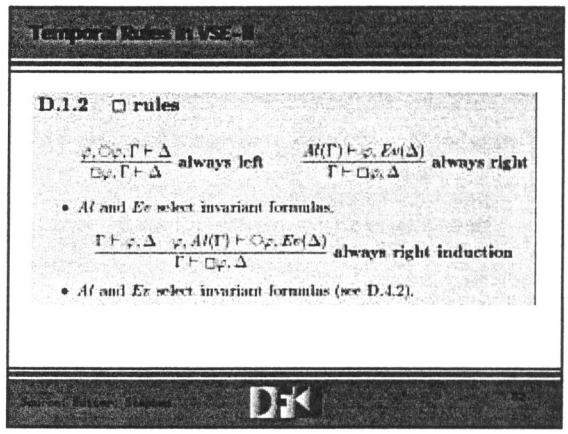
**D.1.2 □ rules**

$$\frac{\varphi, \bigcirc\Box\varphi, \Gamma \vdash \Delta}{\Box\varphi, \Gamma \vdash \Delta} \textbf{ always left} \qquad \frac{\mathit{At}(\Gamma) \vdash \varphi, \mathit{Ev}(\Delta)}{\Gamma \vdash \Box\varphi, \Delta} \textbf{ always right}$$

- *At* and *Ev* select invariant formulas.

$$\frac{\Gamma \vdash \varphi, \Delta \quad \varphi, \mathit{At}(\Gamma) \vdash \bigcirc\varphi, \mathit{Ev}(\Delta)}{\Gamma \vdash \Box\varphi, \Delta} \textbf{ always right induction}$$

- *At* and *Ev* select invariant formulas (see D.4.2).



## Temporal Rules in VSE-II

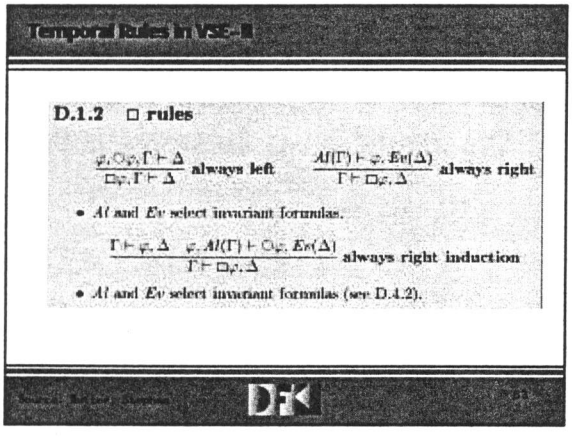
**D.1.2 □ rules**

$$\frac{\varphi, \bigcirc\Box\varphi, \Gamma \vdash \Delta}{\Box\varphi, \Gamma \vdash \Delta} \textbf{ always left} \qquad \frac{\mathit{At}(\Gamma) \vdash \varphi, \mathit{Ev}(\Delta)}{\Gamma \vdash \Box\varphi, \Delta} \textbf{ always right}$$

- *At* and *Ev* select invariant formulas.

$$\frac{\Gamma \vdash \varphi, \Delta \quad \varphi, \mathit{At}(\Gamma) \vdash \bigcirc\Box\varphi, \mathit{Ev}(\Delta)}{\Gamma \vdash \Box\varphi, \Delta} \textbf{ always right induction}$$

- *At* and *Ev* select invariant formulas (see D.4.2).



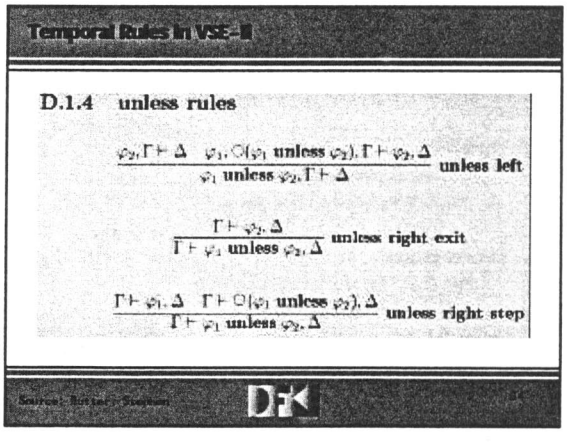
## Temporal Rules in VSE-II

**D.1.4 unless rules**

$$\frac{\varphi_2, \Gamma \vdash \Delta \quad \varphi_1, \bigcirc(\varphi_1 \textbf{ unless } \varphi_2), \Gamma \vdash \varphi_2, \Delta}{\varphi_1 \textbf{ unless } \varphi_2, \Gamma \vdash \Delta} \textbf{ unless left}$$

$$\frac{\Gamma \vdash \varphi_2, \Delta}{\Gamma \vdash \varphi_1 \textbf{ unless } \varphi_2, \Delta} \textbf{ unless right exit}$$

$$\frac{\Gamma \vdash \varphi_1, \Delta \quad \Gamma \vdash \bigcirc(\varphi_1 \textbf{ unless } \varphi_2), \Delta}{\Gamma \vdash \varphi_1 \textbf{ unless } \varphi_2, \Delta} \textbf{ unless right step}$$



## Temporal Rules in VSE-II

$$\frac{\Gamma \vdash \varphi_1, \varphi_2, \Delta \quad \varphi_1, All\mathit{Inl}_{\varphi_1}(\Gamma) \vdash \varphi_2, \bigcirc(\varphi_1 \vee \varphi_2), Ev(\Delta)}{\Gamma \vdash \varphi_1 \textbf{ unless } \varphi_2, \Delta}$$ **unless right induction**

- $All\mathit{Inl}_{\varphi_1}$ selects invariant formulas, and also preserves unless formulas.
- $Ev$ selects invariant formulas of the succedent.



## Temporal Rules in VSE-II

### D.1.5 Step rule

$$\frac{All_w(\Gamma) \vdash Ex_w(\Delta)}{\Gamma \vdash \Delta}$$ **step**

- $w$ maps all free flexible, unprimed variables onto new rigid variables.
- $All_w$ and $Ex_w$ select invariant formulas, and transfers formulas which are not invariant to next state.



## Temporal Rules in VSE-II

### D.1.6 Induction rule

$$\frac{\Box\forall v, (v \ll v_0 \rightarrow \varphi[v]), All(\Gamma) \vdash \varphi[v_0], Ev(\Delta)}{\Gamma \vdash \forall v, \varphi[v], \Delta}$$ **temporal induction 1**

- a size function $\ll$ for $v$ must exist
- $v_0$ is new rigid variable and
- $\varphi[v]$ indicates that the variable $v$ occurs in the formula $\varphi$.



## Temporal Rules in VSE-II

### D.1.7 Quantifiers

$$\frac{\varphi[x], \Gamma \vdash \Delta}{\exists v, \varphi[v], \Gamma \vdash \Delta}$$ **hide left**

- $\varphi$ must be invariant under stuttering
- $x$ is new flexible variable

$$\frac{\Gamma \vdash \varphi[\tau], \Delta}{\Gamma \vdash \exists v, \varphi[v], \Delta}$$ **hide right**

- $\varphi[\tau]$ is substitution of $\varphi$ by term $\tau$; $\tau$ may contain flexible and primed variables; substituting terms in temporal logic is different from substitution in predicate logic (s. [ZM])



## Temporal Rules in VSE-II

### D.1.8 Confluence rule

$$\frac{\Diamond\Box[\varphi \wedge \psi], \Gamma \vdash \Delta}{\Diamond\Box\varphi, \Diamond\Box\psi, \Gamma \vdash \Delta}$$ **confluence left**

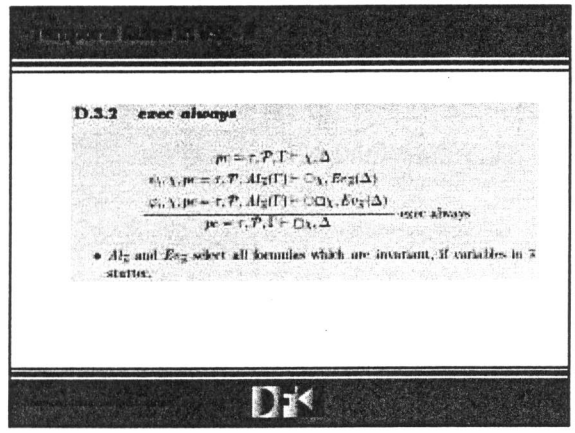


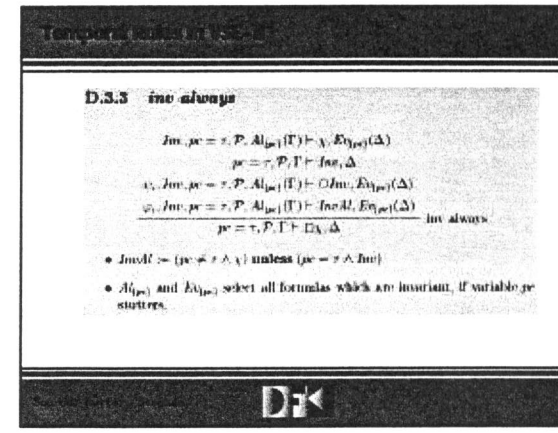
## Temporal Rules in VSE-II

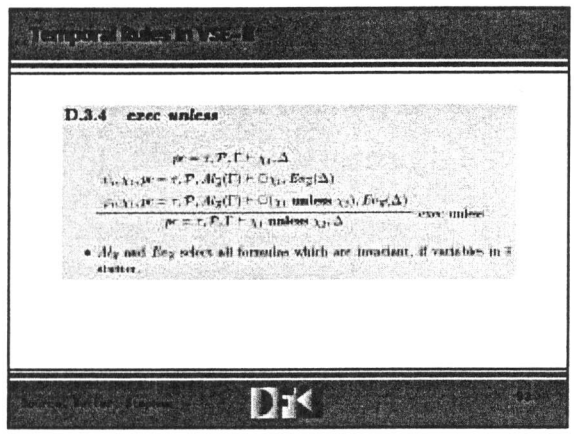
### D.2 Transition Rule

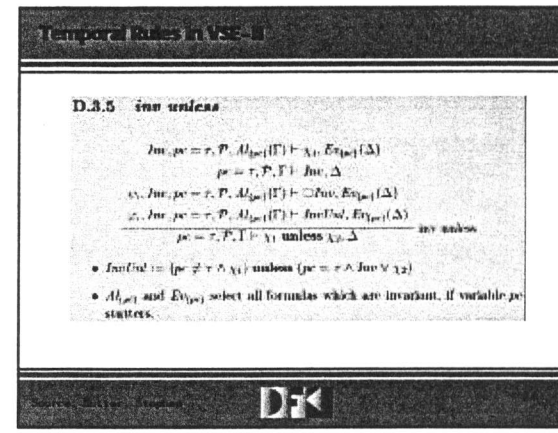
$$\frac{\begin{array}{c} Isc, All(\Gamma) \vdash \chi, Ev(\Delta) \\ \Gamma \vdash Ins, \Delta \\ Ax_w(Inv(), Xi_{w_i}(c_i), All(\Gamma) \vdash Ins', Ev(\Delta) \end{array}}{\Box(\chi \vee \ldots \vee v_n), \Gamma \vdash \Box\chi, \Delta}$$ **canonical always right**

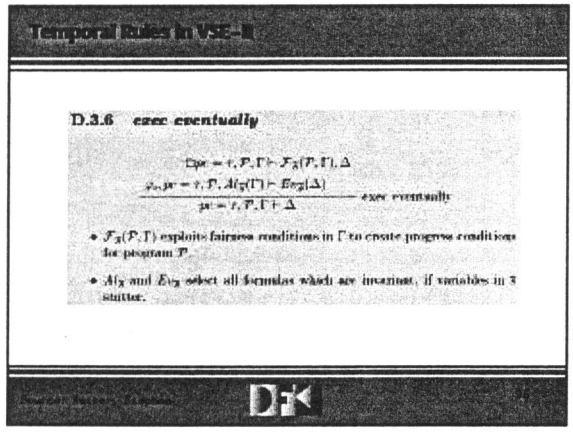
- for each $v_i$, $i = 1, \ldots, n$ a separate premise is created
- $\chi$, $Ins$ are predicate logic formulas and contain no primed variables
- $v_1, \ldots, v_n$ are predicate logic formulas
- $w$ maps all free flexible, unprimed variables onto new rigid variables
- $Ax_w$ transfers formulas to next state

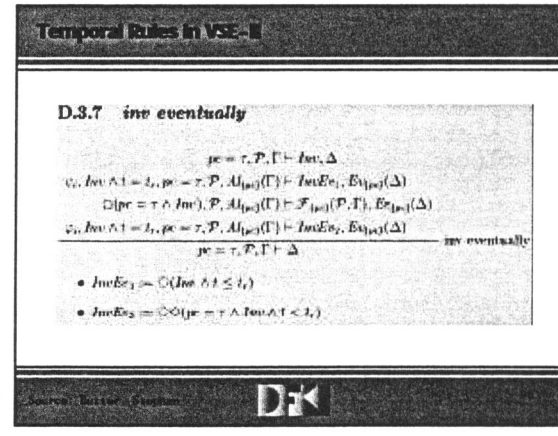

D.3.2 exec always
$pc = \tau, \mathcal{P}, \Gamma \vdash \chi, \Delta$
$\varphi_1, \chi, pc = \tau, \mathcal{P}, Al_\Xi(\Gamma) \vdash \bigcirc\chi, Ev_\Xi(\Delta)$
$\varphi_2, \chi, pc = \tau, \mathcal{P}, Al_\Xi(\Gamma) \vdash \bigcirc\Box\chi, Ev_\Xi(\Delta)$
$pc = \tau, \mathcal{P}, \Gamma \vdash \Box\chi, \Delta$
exec always
• $Al_\Xi$ and $Ev_\Xi$ select all formulas which are invariant, if variables in $\Xi$ stutter.


D.3.3 inv always
$Inv, pc = \tau, \mathcal{P}, Al_{\{pc\}}(\Gamma) \vdash \chi, Ev_{\{pc\}}(\Delta)$
$pc = \tau, \mathcal{P}, \Gamma \vdash Inv, \Delta$
$\varphi_1, Inv, pc = \tau, \mathcal{P}, Al_{\{pc\}}(\Gamma) \vdash \bigcirc Inv, Ev_{\{pc\}}(\Delta)$
$\varphi_2, Inv, pc = \tau, \mathcal{P}, Al_{\{pc\}}(\Gamma) \vdash InvAl, Ev_{\{pc\}}(\Delta)$
$pc = \tau, \mathcal{P}, \Gamma \vdash \Box\chi, \Delta$
inv always
• $InvAl := (pc \neq \tau \wedge \chi)$ unless $(pc = \tau \wedge Inv)$
• $Al_{\{pc\}}$ and $Ev_{\{pc\}}$ select all formulas which are invariant, if variable $pc$ stutters.


D.3.4 exec unless
$pc = \tau, \mathcal{P}, \Gamma \vdash \chi_1, \Delta$
$\varphi_1, \chi_1, pc = \tau, \mathcal{P}, Al_\Xi(\Gamma) \vdash \bigcirc\chi_1, Ev_\Xi(\Delta)$
$\varphi_2, \chi_1, pc = \tau, \mathcal{P}, Al_\Xi(\Gamma) \vdash \bigcirc(\chi_1 \text{ unless } \chi_2), Ev_\Xi(\Delta)$
$pc = \tau, \mathcal{P}, \Gamma \vdash \chi_1$ unless $\chi_2, \Delta$
exec unless
• $Al_\Xi$ and $Ev_\Xi$ select all formulas which are invariant, if variables in $\Xi$ stutter.


D.3.5 inv unless
$Inv, pc = \tau, \mathcal{P}, Al_{\{pc\}}(\Gamma) \vdash \chi_1, Ev_{\{pc\}}(\Delta)$
$pc = \tau, \mathcal{P}, \Gamma \vdash Inv, \Delta$
$\varphi_1, Inv, pc = \tau, \mathcal{P}, Al_{\{pc\}}(\Gamma) \vdash \bigcirc Inv, Ev_{\{pc\}}(\Delta)$
$\varphi_2, Inv, pc = \tau, \mathcal{P}, Al_{\{pc\}}(\Gamma) \vdash InvUnl, Ev_{\{pc\}}(\Delta)$
$pc = \tau, \mathcal{P}, \Gamma \vdash \chi_1$ unless $\chi_2, \Delta$
inv unless
• $InvUnl := (pc \neq \tau \wedge \chi_1)$ unless $(pc = \tau \wedge Inv \vee \chi_2)$
• $Al_{\{pc\}}$ and $Ev_{\{pc\}}$ select all formulas which are invariant, if variable $pc$ stutters.


D.3.6 exec eventually
$\Box pc = \tau, \mathcal{P}, \Gamma \vdash \mathcal{F}_\Xi(\mathcal{P}, \Gamma), \Delta$
$\varphi_1, pc = \tau, \mathcal{P}, Al_\Xi(\Gamma) \vdash Ev_\Xi(\Delta)$
$pc = \tau, \mathcal{P}, \Gamma \vdash \Delta$
exec eventually
• $\mathcal{F}_\Xi(\mathcal{P}, \Gamma)$ exploits fairness conditions in $\Gamma$ to create progress conditions for program $\mathcal{P}$.
• $Al_\Xi$ and $Ev_\Xi$ select all formulas which are invariant, if variables in $\Xi$ stutter.


D.3.7 inv eventually
$pc = \tau, \mathcal{P}, \Gamma \vdash Inv, \Delta$
$\varphi_1, Inv \wedge \lambda = \lambda_1, pc = \tau, \mathcal{P}, Al_{\{pc\}}(\Gamma) \vdash InvEv_1, Ev_{\{pc\}}(\Delta)$
$\Box(pc = \tau \wedge Inv), \mathcal{P}, Al_{\{pc\}}(\Gamma) \vdash \mathcal{F}_{\{pc\}}(\mathcal{P}, \Gamma), Ev_{\{pc\}}(\Delta)$
$\varphi_2, Inv \wedge \lambda = \lambda_1, pc = \tau, \mathcal{P}, Al_{\{pc\}}(\Gamma) \vdash InvEv_2, Ev_{\{pc\}}(\Delta)$
$pc = \tau, \mathcal{P}, \Gamma \vdash \Delta$
inv eventually
• $InvEv_1 := \bigcirc(Inv \wedge \lambda \leq \lambda_1)$
• $InvEv_2 := \bigcirc\Diamond(pc = \tau \wedge Inv \wedge \lambda < \lambda_1)$

## Temporal Rules in VSE-II

**Filters which also map into next state**

$Al_m$ is defined as follows

$$\begin{aligned} Al_m(\Box\varphi) &= \Box\varphi \\ Al_m(SF(\varphi)\{\overline{x}\}) &= SF(\varphi)\{\overline{x}\} \\ Al_m(WF(\varphi)\{\overline{x}\}) &= WF(\varphi)\{\overline{x}\} \\ Al_m(\varphi) &= Nx_m(\varphi) \quad \text{otherwise} \end{aligned}$$

$Ev_m$ is defined as follows

$$\begin{aligned} Ev_m(\Diamond\varphi) &= \Diamond\varphi \\ Ev_m(\varphi) &= Nx_m(\varphi) \quad \text{otherwise} \end{aligned}$$



## Temporal Rules in VSE-II

**Filters which respect stuttering variables**

$Al_S$ is defined as follows

$$\begin{aligned} Al_S(\varphi \odot \Gamma) &= Al_S(\varphi) \odot Al_S(\Gamma) \\ Al_S(\Box\varphi) &= \Box\varphi \\ Al_S(SF(\varphi)\{\overline{x}\}) &= SF(\varphi)\{\overline{x}\} \\ Al_S(WF(\varphi)\{\overline{x}\}) &= WF(\varphi)\{\overline{x}\} \\ Al_S(\varphi) &= \varphi \quad \text{if } \varphi \text{ a predicate logic formula and } \mathit{flexvars}(\varphi) \subseteq \overline{x} \\ Al_S(\varphi) &= nil \quad \text{otherwise} \end{aligned}$$

with $\odot \in \{\wedge, \vee, \ldots\}$.



## Temporal Rules in VSE-II

**Filter which preserve unless formulas**

$AllUnl_\chi$ is defined as follows

$$\begin{aligned} AllUnl_\chi(\Box\varphi) &= \Box\varphi \\ AllUnl_\chi(SF(\varphi)\{\overline{x}\}) &= SF(\varphi)\{\overline{x}\} \\ AllUnl_\chi(WF(\varphi)\{\overline{x}\}) &= WF(\varphi)\{\overline{x}\} \\ AllUnl_\chi(\varphi) &= \varphi \quad \varphi \text{ is rigid} \\ AllUnl_\chi(\varphi_1 \textbf{ unless } \varphi_2) &= (\Box(\varphi_2 \rightarrow \chi)) \rightarrow (\varphi_1 \textbf{ unless } \varphi_2) \\ AllUnl_\chi(\varphi) &= nil \quad \text{otherwise} \end{aligned}$$



## Proof Search
## Annotated Reasoning



## Automating Deduction
## 50 Years of Proof Search and Proof Tasks

- Proof capabilities are still not fully mechanized
- Deduction migrated from the heart of AI to an assistant of formal methods

- Improvements come in small steps
- „Deduction engineering" (Loveland):
  - Strategy formulation
  - Augumenting existing systems
  - Composition of various techniques



## Automated Theorem Proving

- Development of calculi:
  e.g. resolution, matrices, paramodulation, superposition
- Avoiding redundancies in calculi
  e.g. basic superposition, restart model elimination, hyperresolution
- Efficent computations, efficient access to data
  e.g. normalforms, term indexing

"Strategic knowledge" is encoded into the basic proof engine

- „Inefficient" (but „human-readable) calculi
  e.g. sequent-calculi, ND-calculi
- Calculus as an abstract machine, embedding of logics
- Tactics as macros / programs operating on this abstract machine
- User as a meta-tactic

„Strategic knowledge" is encoded into tactics

- Proof planning using tactics
- Additional declarative specification of tactics
- Methods = Tactics + Pre- and Postconditions

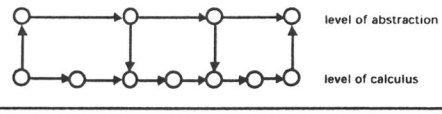
- But:
  Problem of weak postconditions
  e.g. only type information is available: $\bar{A} \vdash \bar{A}$

  → integrated plan generation and execution

- Possible solution: abstractions

## Abstractions in Theorem Proving

- Abstractions
  - Formalizing pre- and postconditions
  - Planning of abstract intermediate goals




## PI-Abstractions

PI-abstractions:
*„The abstraction of a proof denotes an abstract proof"*
Example:

- Use of term-rewriting-systems at calculus and abstract level:
- Correspondence between concrete and abstract rewriting rules
- → Abstraction and substitutions commute
- → Abstraction and subterm-relation commute
- ⇒ abstraction on signature

## Abstractions Using Additional Information

Rippling to guide inductive proofs

Inductionhypothesis: $(x + y) + z = x + (y + z)$
Inductionconclusion: $(s(x) + y) + z = s(x) + (y + z)$

Modification of the conclusion:
$(s(x) + y) + z = s(x) + (y + z)$
$s(x + y) + z = s(x) + (y + z)$
$s((x + y) + z) = s(x) + (y + z)$
$s((x + y) + z) = s(x+ (y + z))$

## Abstraction Using Additional Information

- Abstraction is based on colors of symbols
- Success is measured by the location of red dots

$((\cdot x + y) + z = (\cdot x + (y + z)$
$(\cdot(x + y) + z = (\cdot x + (y + z)$
$\cdot((x + y) + z) = (\cdot x + (y + z)$
$\cdot((x + y) + z) = \cdot(x+ (y + z))$

- Abstract rules: $(\cdot x + y) = \cdot(x + y)$

## Encoding and Maintaining Useful Information

- Abstractions are definiert on
  term + strategic information
- Additional information changes during proof search

⇒ Integration of terms and strategic information
⇒ Terms annotated by additional information
⇒ Encoding of strategic information in logic
⇒ Maintaining terms and strategic information with the help of a uniform calculus



## Knowledge Representation Using Logic

- How to encode strategic knowledge in logic?
- Labelled Deductive Systems (Gabbay):
  - a label *t* is attached to a formula *A*
    calculi operate on pairs *t:A*
  - no labels for subterms
  - no fixed language for labels
  - no general support for using labels



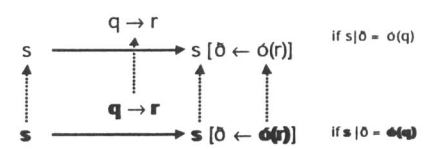
## Logic to Maintain Knowledge

- Annotations
  - Each occurrence of a symbol can store information
  - Information is encoded in 1st order terms (annotation signature and variables)
  - Encoding information into annotations of individual symbol occurrences
  - uniform maintenance within an annotated logic



## Maintaining Information

Idea: information is maintained by an enlarged inferernce mechanism






## Annotated Substitution

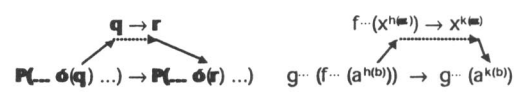
- annotated substitutions correspond to substitutions in the underlying calculus (Soundness):
  $|\phi(t)| = |\phi|(|t|) = \acute{o}(t)$
- annotated substitutions instantiate also annotations (strategic information)
- annotated substitution is composed of
  - substitution for terms
  - substitutions for annotations

  however: both parts are not independent!

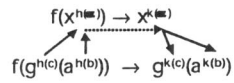


## Flow of Information

- Annotation of the rewrite rules and the notion of substitution determine the information flow
- q and r contain pattern of information that is instantiated during rewriting



- Logic for annotations and terms are independent
- Annotations and terms can be instantiated independently
- Annotations of variables determine the combination



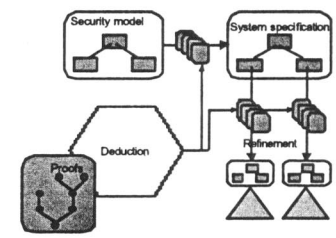
- Relations between different parts of formulas:
  - rippling, difference -reduction-approaches
  - window-technique (Isabelle)
  - reuse of proofs
  - origin-tracking
- Management of information during proof search
  - normalform of terms
  - efficient simplification
- Linguistics

Calculi:

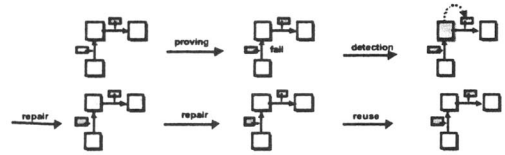
- 1st-order logic + Colors
  (Hutter, Journal of Automated Reasoning 1997)
- Lambda-Calculus + Colors
  (Hutter + Kohlhase, Journal of Automated Reasoning 2000)
- 1st-order logic (Lambda-Calculus) + general annotations
  (Hutter, Annals of Mathematics and Artificial Intelligence 2001)

- Generic mechanism to integrate strategic information
  (without changing the underlying calculus)
- Definition of information flow by annotating (rewrite) rules
  - no soundness problems
  - high degree of flexibility without modification of the underlying calculus
  - logical frameworks for strategic information

## Maintaining Formal Developments



## Techniques to Support Revision Process

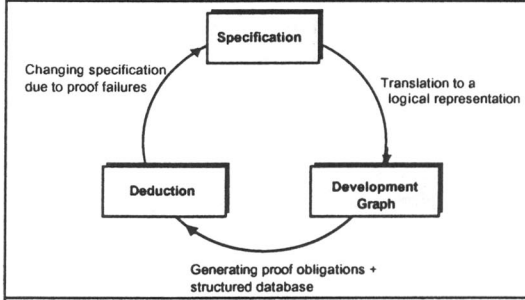



**Solutions in RiPE:**

- Management of change for development graphs
- Analogical replay and reuse of proofs
- Building set for the change of formal developments



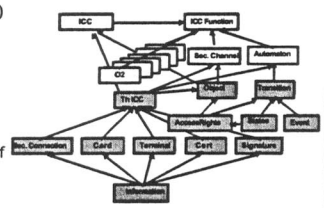
## Specification & Verification Circle






## Developments in Practice

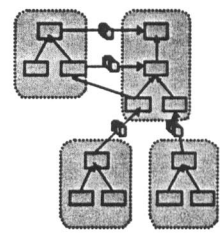
- Large number of specified theories (usually > 100)
- Complex dependencies between theories
- Different structuring mechanisms of specification language and its logic encoding
- Automatic generation of proof obligations






## Development Graph

- Logic based representation of structured formal developments
- Specifications and implementations as theories (consequence relations)
- Formal relations between parts of developments (morphisms)
- supports different formalisms (logics) to represent different parts
- supports hiding





## Semantics of Developmentgraphs

Let N be a node in a developmentgraph

Define class $Mod_S(N)$ of $\Sigma_N$- models n of N by

- n satisfies local axioms of N
- for all global $K \xrightarrow{\sigma} N$, $n|_\sigma$ is a K-model
- for all local $K \xrightarrow{\sigma} N$:
  - $n|_\sigma$ satisfies local axioms of K,
- for all $K \xleftarrow{\sigma} N$:
  - n has a $\sigma$ - expansion k ($k|_\sigma = n$)



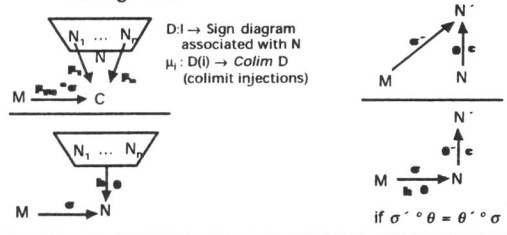
## Semantics of Developmentgraphs

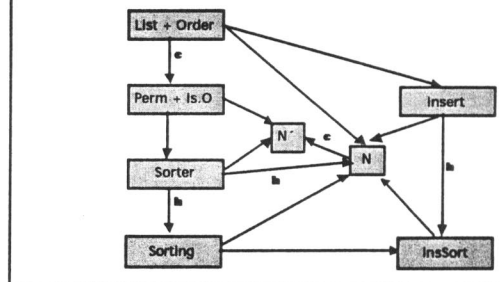
Let N be a node in a developmentgraph

Define class $Mod_S(N)$ of $\Sigma_N$- models n of N by

- n satisfies local axioms of N
- for all global $K \xrightarrow{\sigma} N$, $n|_\sigma$ is a K-model
- for all local $K \xrightarrow{\sigma} N$:
  - $n|_\sigma$ satisfies local axioms of K,
- for all $K \xleftarrow{\sigma} N$:
  - n has a $\sigma$ - expansion k ($k|_\sigma = n$)

Idea: move theorem-links out of the scope of hiding-links




```
spec SORTING [TotalOrder] =
{ LIST [sort Elem]
then
  preds  is_ordered : List[Elem];
         perm: List[Elem] x List[Elem];
  vars x, y:Elem; L, L1,L2 : List[Elem]
  is_ordered([])
  is_ordered([x])
  is_ordered(x::(y::L)) = ...
  perm(L1, L2) = ...
then
  ops sorter : List[Elem] → List[Elem]
  vars L : List[Elem]
  is_ordered(sorter(L))
  perm(L, Sorter(L))
}
hide is_ordered, perm
end
```

```
spec INSERTSORT [TotalOrder] =
{ LIST [sort Elem]
then
  ops insert: Elem x List[Elem] → List[Elem]
      sorter: List[Elem] → List[Elem]
  vars x, y : Elem  L : List[Elem]
  insert(x, []) = [x]
  insert(x, y::L) = x::(y::L) when x < y
                    else y::insert(x, L)
  sorter([]) = []
  sorter(x::L) = insert(x, sorter(L))
}
hide insert
```

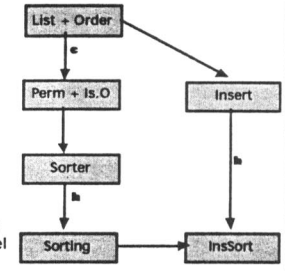
Developmentgraphs contain also hiding (definition) links:

M —σ→ N, σ : $\Sigma^N \to \Sigma^M$

and also hiding theorem links:

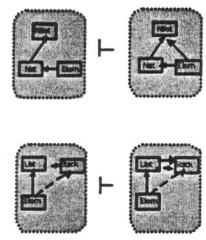
M —σ→ N, σ : $\Sigma^N \to \Sigma^M$

i.e for all m ∈ $Mod_S(M)$, $m|_\sigma$ has a θ-expansion to some N-model



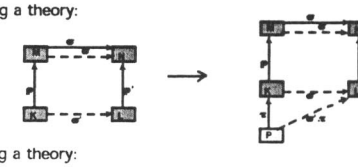

## Global Management of Change

- Calculus to reason about relations between theories:
  - reduction of the amount of (real) proof obligations
  - localization of proof obligations to theories causing them
  - transformation of graphs to capture hiding

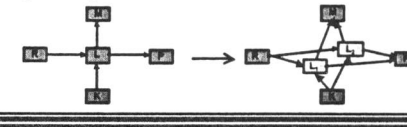


## Basic Operations to Preserve Proofs

Inserting a theory:

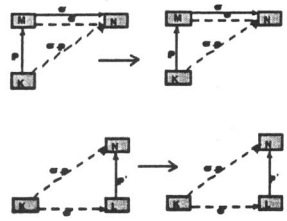


Splitting a theory:



## Changing the Development Graph

- Reduction of relations:
  instead "global" relations
  Th(N) |- σ(Th(M))
  use of "local" relations:
  Th(N) |- σ(Ax(M))
  and Th(N) |- σ(ρ(Ax(K)))
- Reduction by reachability:
  links are subsumed if
  σ(ρ(Ax(K))) = ρ'(σ'(Ax(K)))



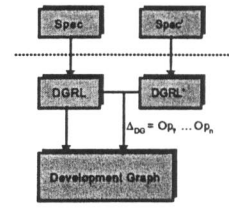

Source: Autor, Stephan

DFKI

133

## From Specifications to Development Graphs

- Uniform intermediate representation language (DGRL)
- Independent of concrete specification language
- Support for various specification languages, e.g. VSE-SL, CASL
- Incremental change of DG
  - Heuristics to compute differences of DGRL´s
  - Computation of a Workplan
- Allows for distributed formal development




Source: Autor, Stephan

DFKI

134

## Computing a Workplan for DGRL-Differences

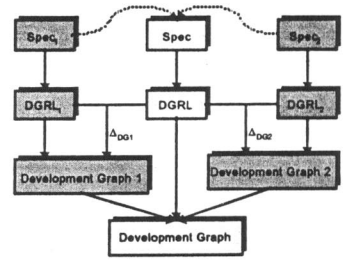
- Differences on DGRL causes computation of a workplan
- Workplan consists of a series of basic operations:
  - e.g. insert new theory, delete theory, split theory
  - e.g. insert new axiom, delete axiom
- Basic operations preserve as much proof work as possible
- Open approach:
  - insertion of new basic operations possible
  - e.g. FM-DIN gave rise to additional basic operations

Source: Autor, Stephan

DFKI

135

## Supporting Distributed Formal Software Development



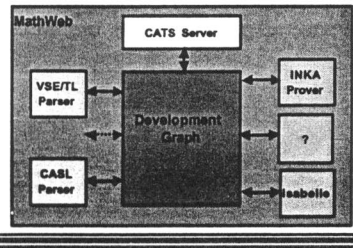

Source: Autor, Stephan

DFKI

136

## MAYA / INKA

- Tool to maintain structured formal developments




Source: Autor, Stephan

DFKI

137

7 Christoph Kreitz
*Cornell University, Ithaca, USA*

**Course: The NuPrl Proof Development System**

# The Nuprl Proof Development System

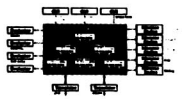
Christoph Kreitz

Department of Computer Science, Cornell University
Ithaca, NY 14853

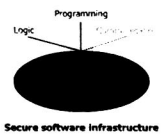
http://www.nuprl.org

## The Nuprl Project at Cornell

- **Computational formal logics** — Type Theory
- **Proof & program development systems**
  - The Nuprl Logical Programming Environment
  - Fast inference engines + proof search techniques
  - Natural language generation from formal mathematics
  - Program extraction + automated complexity analysis



- **Application to reliable, high-performance networks**
  - Assigning precise semantics to system software
  - Performance Optimizations
  - Assurance for reliability (verification)
  - Verified System Design






## Nuprl's Type Theory

- **Constructive higher-order logic**
  - Reasoning about types, elements, propositions, proofs, functions ...
- **Functional programming language**
  - Similar to core **ML**: polymorphic, with partial recursive functions
- **Expressive data type system**
  - Function, product, disjoint union, Π- & Σ-types, atoms, void, top
  - Integers, lists, inductive types, universes
  - Propositions as types, equality type, subsets, subtyping, quotient types
  - (Dependent) intersection, union, records, modules
- **Open-ended**
  - new types can be added if needed
- **User-defined extensions possible**



## The Nuprl Proof Development System

- **Beginnings in 1984**
  - **Nuprl 1** (Symbolics): proof & program refinement in Type Theory
  - Book: *Implementing Mathematics ...* (1986)
  - **Nuprl 2**: Unix Version
- **Nuprl 3: Mathematical problem solving** (1987–1994)
  - Constructive machine proofs for unsolved mathematical problems
- **Nuprl 4: System verification and optimization** (1993–2001)
  - Verification of logic synthesis tools & SCI cache coherency protocol
  - Optimization/verification of the **Ensemble** group communication system
- **Nuprl 5: Open distributed architecture** (2000–... )
  - Cooperating proof processes centered around persistent knowledge base
  - Asynchronous, concurrent, and external proof engines

  ⇝ Interactive digital libraries of formal algorithmic knowledge

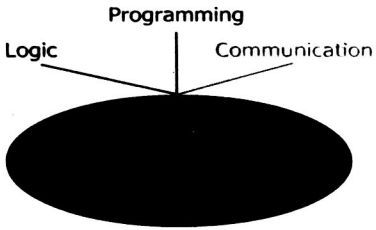
## Applications: Mathematics & Programming

- **Formalized mathematical theories**
  - Elementary number theory, real analysis, group theory
  - Discrete mathematics (Allen, 1994 –...)
  - General algebra (Jackson, 1994)
  - Finite and general automata (Constable, Naumov Uribe 1997, Bickford, 2001)
  - Basics of Turing machines (Naumov, 1998 ...)
  - Formal mathematical textbook (Constable, Allen 1999)

  `http://www.nuprl.org/Nuprl4.2/Libraries/Welcome.html`
- **Machine proof for unsolved problems**
  - Girard's paradox (Howe 1987)
  - Higman's Lemma (Murthy 1990)
- **Algorithms and programming languages**
  - Synthesis of elementary algorithms: square-root, sorting, ...
  - Simple imperative programming (Naumov, 1997)
  - Programming semantics & complexity analysis (Benzinger, 2000)
  - Type-theoretical semantics of large OCaml fragment (Kreitz 1997/2002)



## Applications: System Verification and Optimization




**Secure software infrastructure**

- **Verification of a logic synthesis tool** (Aagaard & Leeser 1993)
- **Verification of the SCI cache coherency protocol** (Howe 1996)
- **Ensemble group communication toolkit**
  - Optimization of application protocol stacks (by factor 3–10) (Kreitz, Hayden, Hickey, Liu, van Renessee 1999)
  - Verification of protocol layers (Bickford 1999)
  - Formal design of new adaptive protocols (Bickford, Kreitz, Liu, van Renessee 2001)
- **MediaNet stream computation network**
  - Validation of real-time schedules wrt. resource limitations (ongoing)



## After more than 15 years ...

- **Insights**
  - Type theory expressive enough to formalize today's software systems
  - Formal optimization can significantly improve practical performance
  - Formal verification reveals errors even in well-explored designs
  - Formal design reveals hidden assumptions and limitations for use of software
- **Ingredients for success in applications...**
  - Precise semantics for implementation language of a system
  - Formal models of: application domain, system model, programming language
  - Knowledge-based formal reasoning tools
  - Collaboration between systems and formal reasoning groups



## Purpose of this course

- Understand Nuprl's theoretical foundation
- Understand features of the Nuprl proof development system
- Learn how to formalize mathematics and computer science

**Additional material can be found at ....**

`http://www.nuprl.org`

`http://www.cs.cornell.edu/home/kreitz/Abstracts/02calculemus-nuprl.html`

## Overview

☑ **Introduction**

**1. NUPRL's Type Theory**
- Distinguishing Features
- Standard NUPRL Types

**2. The NUPRL Proof Development System**
- Architecture and Feature Demonstration

**3. Proof Automation in NUPRL**
- Tactics & Rewriting
- Decision Procedures
- External Proof Systems

**4. Building Formal Theories**
- (Dependent) Records, Algebra, Abstract Data Types

**5. Future Directions**



# Part I:

# Nuprl's Type Theory



## The NuPRL Type Theory
## An Extension of Martin-Löf Type Theory

- **Foundation for computational mathematics**
  - Higher-order logic + programming language + data type system
  - Focus on constructive reasoning
  - Reasoning about types, elements, and equality ...
- **Open-ended, expressive type system**
  - Function, product, disjoint union, Π- & Σ-types, atoms ⇝ programming
  - Integers, lists, inductive types ⇝ inductive definition
  - Propositions as types, equality type, void, top, universes ⇝ logic
  - Subsets, subtyping, quotient types ⇝ mathematics
  - (Dependent) intersection, union, records ⇝ modules, program composition

  *New types can/will be added as needed*
- **Self-contained**
  - Based on "formalized intuition", not on other theories



## Distinguishing Features of NUPRL's Type Theory

- **Uniform internal notation**
  - Independent display forms support flexible term display ⇝ free syntax
- **Expressions defined independently of their types**
  - No restriction on expressions that can be defined ⇝ Y combinator
  - Expressions *in proofs* must be typeable ⇝ "total" functions
- **Semantics based on values of expressions**
  - Judgments state what is true ⇝ computational semantics
- **Refinement calculus**
  - Top-down sequent calculus ⇝ interactive proof development
  - Proof expressions linked to inference rules ⇝ program extraction
  - Computation rules ⇝ program evaluation
- **User-defined extensions possible**
  - User-defined expressions and inference rules ⇝ abstractions & tactics

## Syntax Issues

- **Uniform notation:** $opid\{p_i:F_i\}(x_{11},..,x_{m_1 1}.t_1;\ldots;x_{1n},..,x_{m_n n}.t_n)$
  - Operator name *opid* listed in operator tables
  - Parameters $p_i:F_i$ for base terms (variables, numbers, tokens...)
  - Sub-terms $t_j$ may contain bound variables $x_{1j},..,x_{m_j j}$
  - No syntactical distinction between types, members, propositions ...

- **Display forms describe visual appearance of terms**

| *Internal Term Structure* | *Display Form* |
|---|---|
| **variable**$\{x:\mathrm{v}\}()$ | $x$ |
| **function**$\{\}(S;x.T)$ | $x:S\to T$ |
| **function**$\{\}(S;.T)$ | $S\to T$ |
| ⋮ | ⋮ |
| **lambda**$\{\}(x.t)$ | $\lambda x.t$ |
| **apply**$\{\}(f;t)$ | $f\ t$ |
| ⋮ | ⋮ |

⇝ conventional notation, information hiding, auto-parenthesizing, aliases, ...



## Semantics models proof, not denotation

- **(Lazy) evaluation of expressions**
  - Identify canonical expressions (values)
  - Identify principal arguments of non-canonical expressions
  - Define reducible non-canonical expressions (redex)
  - Define reduction steps in redex–contracta table

| *canonical* | *non-canonical* | *Redex* | *Contractum* |
|---|---|---|---|
| $S\to T$ | | | |
| $\lambda x.t$ | $f\ t$ | $\lambda x.u\ t$ | $\xrightarrow{\beta}\ u[t/x]$ |

- **Judgments: semantical truths about expressions**
  - 4 categories: Typehood ($T$ Type), Type Equality ($S{=}T$), Membership ($t\in T$), Member equality ($s{=}t$ in $T$)
  - Semantics tables define judgments for values of expressions

$S_1\to T_1 = S_2\to T_2$ iff $S_1{=}S_2$ and $T_1{=}T_2$

$\lambda x_1.t_1 = \lambda x_2.t_2$ in $S\to T$ iff $S\to T$ Type and $t_1[s_1/x_1] = t_2[s_2/x_2]$ in $T$ for all $s_1, s_2$ with $s_1{=}s_2\in S$

⋮ ⋮



## Nuprl's Proof Theory

- **Sequent** $x_1:T_1,\ldots,x_n:T_n \vdash C$ ⌈ext $t$⌋
  *"If $x_i$ are variables of type $T_i$ then $C$ has a (yet unknown) member $t$"*
  - A judgment $t\in T$ is represented as $T$ ⌈ext $t$⌋ ⇝ proof term construction
  - Equality is represented as type $s{=}t\in T$ ⌈ext Ax⌋ ⇝ propositions as types
  - Typehood represented by (cumulative) universes $\mathbf{U}_i$ ⌈ext $T$⌋

- **Refinement calculus**
  - Top-down decomposition of proof goal ⇝ interactive proof development
  - Bottom-up construction of proof terms ⇝ program extraction

$\Gamma\vdash S\to T$ ⌈ext $\lambda x.e$⌋ **by lambda-formation** $x$
$\Gamma, x:S\vdash T$ ⌈ext $e$⌋
$\Gamma\vdash S{=}S\in\mathbf{U}_i$ ⌈ext Ax⌋

  - Computation rules ⇝ program evaluation

About 8–10 inference rules for each Nuprl type



## Executing a Formal Proof Step

| | |
|---|---|
| *Theorem name* | |
| *Status + position in proof* | `# top 1` |
| *Hypothesis of main goal* | `1. x:ℕ` |
| *Conclusion* | `⊢ ∃y:ℕ. y²≤x ∧ x<(y+1)²` |
| *Inference rule* | `BY natE 1` |
| *First subgoal – status, conclusion* | `1# ⊢ ∃y:ℕ. y²≤0 ∧ 0<(y+1)²` |
| *Second subgoal – status, new hypotheses* | `2# 2. n:ℕ`<br>`3. 0<n`<br>`4. v: ∃y:ℕ. y²≤n-1 ∧ n-1<(y+1)²` |
| *conclusion* | `⊢ ∃y:ℕ. y²≤n ∧ n<(y+1)²` |

## Methodology for building types

- **Syntax:**
  - Define canonical type
  - Define canonical members of the type
  - Define noncanonical expressions corresponding to the type
- **Semantics**
  - Introduce evaluation rules for non-canonical expressions
  - Define type equality judgment for the type
    The typehood judgment is a special case of type equality
  - Define member equality judgment for canonical members
    The membership judgment is a special case of member equality

  *Define judgments only in terms of the new expressions* $\rightsquigarrow$ consistency
- **Proof Theory**
  - Introduce proof rules that are consistent with the semantics



## Methodology for defining proof rules

- **Type Formation rules:**
  - When are two types equal? (*type*Equality) $\Gamma \vdash S = T \in \mathbf{U}_j$
  - How to build the type? (*type*Formation) $\Gamma \vdash \mathbf{U}_j$ ext $T_j$
- **Canonical rules:**
  - When are two members equal? (*member*Equality) $\Gamma \vdash s = t \in T$
  - How to build members? (*member*Formation) $\Gamma \vdash T$ ext $t_j$
- **Noncanonical rules:**
  - When does a term inhabit a type? (*noncanonical*Equality) $\Gamma \vdash s = t \in T$
  - How to use a variable of the type (*type*Elimination) $\Gamma, x{:}S, \Delta \vdash T$ ext $t_j$
- **Computation rules:**
  - Reduction of redices in an equality (*noncanonical*Reduce*) $\Gamma \vdash redex = t \in T$
- **Special purpose rules**



## Proof Rules for the Function Type

$\Gamma \vdash \mathbf{U}_j$ ext $x{:}S \rightarrow T_j$
**by dependent_functionFormation** $x$ $S$
$\Gamma \vdash S \in \mathbf{U}_j$ ext Ax
$\Gamma, x{:}S \vdash \mathbf{U}_j$ ext $T_j$

$\Gamma \vdash \lambda x_1.t_1 = \lambda x_2.t_2 \in x{:}S \rightarrow T$ ext Ax
**by lambdaEquality** $j$ $x'$
$\Gamma, x'{:}S \vdash t_1[x'/x_1] = t_2[x'/x_2] \in T[x'/x]$ ext Ax
$\Gamma \vdash S \in \mathbf{U}_j$ ext Ax

$\Gamma \vdash f_1 t_1 = f_2 t_2 \in T[t_1/x]$ ext Ax
**by applyEquality** $x{:}S \rightarrow T$
$\Gamma \vdash f_1 = f_2 \in x{:}S \rightarrow T$ ext Ax
$\Gamma \vdash t_1 = t_2 \in S$ ext Ax

$\Gamma \vdash (\lambda x.t)\, s = t_2 \in T$ ext Ax
**by applyReduce**
$\Gamma \vdash t[s/x] = t_2 \in T$ ext Ax

$\Gamma \vdash x_1{:}S_1 \rightarrow T_1 = x_2{:}S_2 \rightarrow T_2 \in \mathbf{U}_j$ ext Ax
**by functionEquality** $x$
$\Gamma \vdash S_1 = S_2 \in \mathbf{U}_j$ ext Ax
$\Gamma, x{:}S_1 \vdash T_1[x/x_1] = T_2[x/x_2] \in \mathbf{U}_j$ ext Ax

$\Gamma \vdash x{:}S \rightarrow T$ ext $\lambda x'.t_j$
**by lambdaFormation** $j$ $x'$
$\Gamma, x'{:}S \vdash T[x'/x]$ ext $t_j$
$\Gamma \vdash S \in \mathbf{U}_j$ ext Ax

$\Gamma, f{:}x{:}S \rightarrow T, \Delta \vdash C$ ext $t[fs, \mathrm{Ax}/y, z]$
**by dependent_functionElimination** $i$ $s$ $y$ $z$
$\Gamma, f{:}x{:}S \rightarrow T, \Delta \vdash s \in S$ ext Ax
$\Gamma, f{:}x{:}S \rightarrow T, y{:}T[s/x], z{:}y = fs \in T[s/x], \Delta \vdash C$ ext $t_j$

$\Gamma \vdash f_1 = f_2 \in x{:}S \rightarrow T$ ext $t_j$
**by functionExtensionality** $j$ $x_1{:}S_1 \rightarrow T_1$ $x_2{:}S_2 \rightarrow T_2$ $x'$
$\Gamma, x'{:}S \vdash f_1 x' = f_2 x' \in T[x'/x]$ ext $t_j$
$\Gamma \vdash S \in \mathbf{U}_j$ ext Ax
$\Gamma \vdash f_1 \in x_1{:}S_1 \rightarrow T_1$ ext Ax
$\Gamma \vdash f_2 \in x_2{:}S_2 \rightarrow T_2$ ext Ax

Note: $e = e \in T$ is usually abbreviated by $e \in T$



## User-defined Extensions

- **Conservative extension of the formal language**
  = **Abstraction**: *new-opid*{*parms*}(*sub-terms*) $\equiv$ *expr*[*parms, sub-terms*]
  e.g. **exists**{}$(T; x.A[x]) \equiv x{:}T \times A[x]$
  + **Display Form** for newly defined term
  e.g. $\exists x{:}T.A[x] \equiv$ **exists**{}$(T; x.A[x])$

  Library contains many standard extensions of Type Theory
  e.g. Intuitionistic logic, Number Theory, List Theory, Algebra, ...
- **Tactics: User-defined inference rules**
  - Meta-level programs built using basic inference rules and existing tactics
  - May include meta-level analysis of the goal to *find* a proof
  - Always result in a valid proof

  Library contains many standard tactics and proof search procedures

## Standard Nuprl Types

| | | |
|---|---|---|
| Function Space | $S \to T$, $x:S \to T$ | $\lambda x.t$, $f\,t$ |
| Product Space | $S \times T$, $x:S \times T$ | $\langle s,t\rangle$, let $\langle x,y\rangle = e$ in $u$ |
| Disjoint Union | $S+T$ | inl$(s)$, inr$(t)$, case $e$ of inl$(x) \mapsto u$ \| inr$(y) \mapsto v$ |
| Universes | $\mathbf{U}_j$ | *— types of level $j$ —* |
| Equality | $s = t \in T$ | Ax |
| Empty Type | Void | any$(x)$, *— no members —* |
| Atoms | Atom | "*token*", if $a{=}b$ then $s$ else $t$ |
| Numbers | $\mathbf{Z}$ | 0,1,-1,2,-2,... $s{+}t$, $s{-}t$, $s{*}t$, $s \div t$, $s$ rem $t$, if $a{=}b$ then $s$ else $t$, if $i{<}j$ then $s$ else $t$ ind$(u; x, f_x.s;$ *base*$;$ $y, f_y.t)$ |
| | $i<j$ | Ax |
| Lists | $S$ list | $[]$, $t::l$, rec-case $L$ of $[] \mapsto$ *base* \| $x::l \mapsto [f_l].t$ |
| Inductive Types | rectype $X = T[X]$ | let* $f(x) = t$ in $f(e)$, *— members defined by $T[X]$ —* |
| Subset | $\{x:S \mid P[x]\}$, | *— some members of $S$ —* |
| Intersection | $\cap x:S.T[x]$, | *— members that occur in all $T[x]$ —* |
| | $x:S \cap T[x]$ | *— members $x$ that occur $S$ and $T[x]$ —* |
| Union | $\cup x:S.T[x]$ | *— members that occur in some $T[x]$, tricky equality—* |
| Quotient | $x,y : S//E[x,y]$ | *— members of $S$, new equality —* |
| Very Dep. Functions | $\{f \mid x:S \to T[f,x]\}$ | |
| Squiggle Equality | $s \sim t$ | *— a "simpler" equality* |



## Functions: Basic Programming Concepts

### Syntax:

Canonical: $S \to T$, $\lambda x.e$

Noncanonical: $e_1\,e_2$

### Evaluation:

$\boxed{\lambda x.u}\,t \xrightarrow{\beta} u[t/x]$

### Semantics:

- $S \to T$ is a type if $S$ and $T$ are
- $\lambda x_1.e_1 = \lambda x_2.e_2$ in $S \to T$ if $S \to T$ type and $e_1[s_1/x_1] = e_2[s_2/x_2]$ in $T$ for all $s_1, s_2$ with $s_1 = s_2 \in S$

### Proof System: — see above —



## Cartesian Products: Building Data Structures

### Syntax:

Canonical: $S \times T$, $\langle e_1, e_2\rangle$

Noncanonical: let $\langle x,y\rangle = e$ in $u$

### Evaluation:

let $\langle x,y\rangle = \boxed{\langle e_1, e_2\rangle}$ in $u \xrightarrow{\beta} u[e_1, e_2 / x, y]$

### Semantics:

- $S \times T$ is a type if $S$ and $T$ are
- $\langle e_1, e_2\rangle = \langle e_1', e_2'\rangle$ in $S \times T$ if $S \times T$ type, $e_1 = e_1'$ in $S$, and $e_2 = e_2'$ in $T$

### Library Concepts: $e.1$, $e.2$



## Disjoint Union: Case Distinctions

### Syntax:

Canonical: $S+T$, inl$(e)$, inr$(e)$

Noncanonical: case $e$ of inl$(x) \mapsto u$ \| inr$(y) \mapsto v$

### Evaluation:

case $\boxed{\text{inl}(e')}$ of inl$(x) \mapsto u$ \| inr$(y) \mapsto v$ $\xrightarrow{\beta}$ $u[e' / x]$

case $\boxed{\text{inr}(e')}$ of inl$(x) \mapsto u$ \| inr$(y) \mapsto v$ $\xrightarrow{\beta}$ $v[e' / y]$

### Semantics:

- $S+T$ is a type if $S$ and $T$ are
- inl$(e)$ = inl$(e')$ in $S+T$ if $S+T$ type, $e = e'$ in $S$
- inr$(e)$ = inr$(e')$ in $S+T$ if $S+T$ type, $e = e'$ in $T$

### Library Concepts: —

## The Curry-Howard Isomorphism, formally

**Propositions are represented as types**

| Proposition | | Type |
|---|---|---|
| $P \wedge Q$ | $\equiv$ | $P \times Q$ |
| $P \vee Q$ | $\equiv$ | $P + Q$ |
| $P \Rightarrow Q$ | $\equiv$ | $P \rightarrow Q$ |
| $\neg P$ | $\equiv$ | $P \rightarrow \mathsf{Void}$ |
| $\exists x{:}T\,.\,P[x]$ | $\equiv$ | $x{:}T \times P[x]$ |
| $\forall x{:}T\,.\,P[x]$ | $\equiv$ | $x{:}T \rightarrow P[x]$ |

**Need an empty type to represent "falsehood"**
**Need dependent types to represent quantifiers**



## Empty Type

**Syntax:**
Canonical: Void – *no canonical elements* –
Noncanonical: any(*e*)

**Evaluation:** – *no reduction rules* –

**Semantics:**
· Void is a type
· *e* = *e'* in Void *never holds*

**Library Concepts:** —

**Warning**: rules for Void allows proving semantical nonsense like
x:Void ⊢ 0=1∈2 or ⊢ Void→2 type



## Singleton Type

**Syntax:**
Canonical: Unit, Ax
Noncanonical: – *no noncanonical expressions* –

**Evaluation:** – *no reduction rules* –

**Semantics:**
· Unit is a type
· Ax = Ax in Unit

**Library Concepts:** —

Defined type in NUPRL, see the library theory `core_1` for further details



## Dependent types

- **Allow representing logical quantifiers as type constructs**
- **Allow typing functions like** `λx. if x=0 then λx.x else λx,y.x`
- **Allow expressing mathematical concepts such as finite automata**
  – $(Q,\Sigma,q_0\delta,F)$, where $q_0 \in Q$, $\delta{:}Q\times\Sigma\rightarrow Q$, $F \subseteq Q$.
- **Allow representing dependent structures in programming languages**
  – Record types $[f_1{:}T_1;\ \ldots;\ f_n{:}T_n]$
  – Variant records
  `type date = January of 1..31 | February of 1..28 | ...`
- **Nuprl had them from the beginning**
  – Other systems have recently adopted them (PVS, SPECWARE, ...)

## Dependent Functions

**Subsumes independent function type**
∀ generalizes ⇒

### Syntax:

Canonical: $x{:}S{\to}T$, $\lambda x.e$

Noncanonical: $e_1\, e_2$

### Evaluation:

$\boxed{\lambda x.u}\, t \xrightarrow{\beta} u[t/x]$

### Semantics:

- $x{:}S{\to}T$ is a type if $S$ is a type and $T[e/x]$ is a type for all $e$ in $S$
- $\lambda x_1.e_1 = \lambda x_2.e_2$ in $x{:}S{\to}T$ if $x{:}S{\to}T$ type and
  $e_1[s_1/x_1]{=}e_2[s_2/x_2]$ in $T[s_1/x]$ for all $s_1, s_2$ with $s_1{=}s_2 \in S$



## Dependent Products

**Subsumes (independent) cartesian product**
∃ generalizes ∧

### Syntax:

Canonical: $x{:}S{\times}T$, $\langle e_1, e_2\rangle$

Noncanonical: **let** $\langle x, y\rangle = e$ **in** $u$

### Evaluation:

**let** $\langle x, y\rangle = \boxed{\langle e_1, e_2\rangle}$ **in** $u \xrightarrow{\beta} u[e_1, e_2 / x, y]$

### Semantics:

- $x{:}S{\times}T$ is a type if $S$ is a type and $T[e/x]$ is a type for all $e$ in $S$
- $\langle e_1, e_2\rangle = \langle e_1', e_2'\rangle$ in $x{:}S{\times}T$ if $x{:}S{\times}T$ type,
  $e_1{=}e_1'$ in $S$, and $e_2{=}e_2'$ in $T[e_1/x]$



## Well-formedness Issues

- **Formation rules for dependent type require checking**
  $x'{:}S \vdash T[x'/x]$ **type**
  - $T$ is a function from $S$ to types that could involve complex computations,
    e.g. $T[i]$ ≡ `if M_i(i) halts then ℕ else Void`

**Well-formedness is undecidable in theories with dependent types**

- **Programming languages must restrict dependencies**
  - Only allow finite dependencies ⇝ decidable typechecking
- **Typechecking in Nuprl cannot be fully automated**
  - Typechecking becomes part of the proof process ⇝ heuristic typechecking
- **Additional problem**
  - What is the type of a function from ℕ to types? ⇝ Girard Paradox



## Universes

- **Syntactical representation of typehood**
  - $T$ type expressed as $T \in \mathbf{U}$ — $S{=}T$ expressed as $S{=}T \in \mathbf{U}$
- **Universes are object-level terms**
  - $\mathbf{U}$ is a type and a universe
  - Girard's Paradox: a theory with dependent types and $\mathbf{U} \in \mathbf{U}$ is inconsistent
    ↦ *No single universe can capture the notion of typehood*
  - Typehood ≙ cumulative hierarchy of universes $\mathbf{U}{=}\mathbf{U}_1 \overset{\in}{\subset} \mathbf{U}_2 \overset{\in}{\subset} \mathbf{U}_3 \overset{\in}{\subset} \ldots$

### Syntax:

Canonical: $\mathbf{U}_j$

Noncanonical: —

### Semantics:

- $\mathbf{U}_j$ is a type for every positive integer $j$
- $S = T$ in $\mathbf{U}_j$ if ...mimic semantics for $S = T$ as types...
- $\mathbf{U}_{j_1} = \mathbf{U}_{j_2}$ in $\mathbf{U}_j$ if $j_1{=}j_2{<}j$

## Integers: Basic Arithmetic

**Syntax:**

Canonical: $\mathbf{Z}$, 0, 1, -1, 2, -2, ... $i<j$, Ax

Noncanonical: rec-case $i$ of $x<0 \mapsto [f_x].s \mid 0 \mapsto b \mid y>0 \mapsto [f_y].t$,
$s+t$, $s-t$, $s*t$, $s \div t$, $s$ rem $t$,
if $i=j$ then $s$ else $t$, if $i<j$ then $s$ else $t$,

**Evaluation:**

rec-case $\boxed{0}$ of $x<0 \mapsto [f_x].s \mid 0 \mapsto b \mid y>0 \mapsto [f_y].t \xrightarrow{\beta} b$

rec-case $\boxed{i}$ of $x<0 \mapsto [f_x].s \mid 0 \mapsto b \mid y>0 \mapsto [f_y].t$ where $i > 0$
$\xrightarrow{\beta} t[i, \text{rec-case } i-1 \text{ of } x<0 \mapsto [f_x].s \mid 0 \mapsto b \mid y>0 \mapsto [f_y].t \ / \ x, f_x]$

rec-case $\boxed{i}$ of $x<0 \mapsto [f_x].s \mid 0 \mapsto b \mid y>0 \mapsto [f_y].t$ where $i < 0$
$\xrightarrow{\beta} s[i, \text{rec-case } i+1 \text{ of } x<0 \mapsto [f_x].s \mid 0 \mapsto b \mid y>0 \mapsto [f_y].t \ / \ x, f_x]$

*other noncanonical expressions evaluate as usual*

**Semantics:**

- $\mathbf{Z}$ is a type
- $i < j$ is a type if $i \in \mathbf{Z}$ and $j \in \mathbf{Z}$
- $i = i$ in $\mathbf{Z}$ for all integer constants $i$
- Ax = Ax in $i<j$ if $i, j$ are integers with $i < j$

**Library Concepts:** see the library theories `int_1`, `int_2`, and `num_thy` ...



## Lists: Basic Data Containers

**Syntax:**

Canonical: $T$ list, [], $e_1::e_2$

Noncanonical: rec-case $e$ of [] $\mapsto$ *base* | $x::l \mapsto [f_{xl}].up$

**Evaluation:**

rec-case $\boxed{[]}$ of [] $\mapsto$ *base* | $x::l \mapsto [f_{xl}].up \xrightarrow{\beta}$ *base*

rec-case $\boxed{e_1::e_2}$ of [] $\mapsto$ *base* | $x::l \mapsto [f_{xl}].up$
$\xrightarrow{\beta} up[e_1, e_2, \text{rec-case } e_2 \text{ of } [] \mapsto base \mid x::l \mapsto [f_{xl}].up \ / \ x, l, f_{xl}]$

**Semantics:**

- $T$ list is a type if $T$ is
- [] = [] in $T$ list if $T$ list is a type
- $e_1::e_2 = e_1'::e_2'$ in $T$ list if $T$ list type, $e_1=e_1'$ in $T$, and $e_2=e_2'$ in $T$ list

**Library Concepts:**

`hd(e)`, `tl(e)`, $e_1$@$e_2$, `length(e)`, `map(f;e)`, `rev(e)`, `e[i]`, `e[i..j^-]`, ...



## Inductive Types: Recursive Definition

- **Representation of recursively defined datatypes**
  - Recursive type definition $X = T[X]$
  - Canonical elements determined by unrolling $T[X]$
  - Noncanonical form for inductive evaluation of elements
- **Recursion must be well-founded**
  - Least fixed point semantics
  - $T[X]$ must contain a "base" case
  - $X$ must only occur positively in $T[X]$
- **Extensions possible**
  - Parameterizatized, simultaneous recursion
    rectype $X_1(x_1) = T[X_1]$ and ... $X_n(x_n) = T[X_n]$ select $X_i(a_i)$
  - Co-inductive type inftype $X = T_X$: greatest fixed point semantics
  - Partial recursive functions $S \not\to T$: unrestricted recursive induction



## Inductive Types, formally

**Syntax:**

Canonical: rectype $X = T_X$

Noncanonical: let* $f(x) = t$ in $f(e)$

**Evaluation:**

let* $f(x) = t$ in $f(e) \xrightarrow{\beta} t[\lambda y.\text{let* } f(x) = t \text{ in } f(y), e \ / \ f, x]$

*Termination of* let* $f(x) = t$ in $f(e)$ *requires* $e$ *in* rectype $X = T[X]$

**Semantics:**

- rectype $X_1 = T_{X1}$ = rectype $X_2 = T_{X2}$
  if $T_{X1}[X/X_1] = T_{X2}[X/X_2]$ for all types $X$
- $s = t$ in rectype $X = T_X$ if rectype $X = T_X$ type and
  $s = t$ in $T_X$[rectype $X = T_X / X$]

## Subset Types: Hiding Computational Content

**• Representation of mathematical concept of subsets**

- $\{x{:}S \mid T[x]\}$ formally similar to dependent product $x{:}S{\times}T[x]$

  ...but ...

- Members are elements of $s \in S$, not pairs $\langle s,t\rangle$
- Only implicit evidence for $T[s]$ but no explicit proof component

**Syntax:**

Canonical: $\{x{:}S \mid T\}$, $\{S \mid T\}$

Noncanonical: —

**Semantics:**

· $\{x_1{:}S_1 \mid T_1\} = \{x_2{:}S_2 \mid T_2\}$ if $S_1{=}S_2$ and there are terms $p_1$, $p_2$ and a variable $x$, which occurs neither in $T_1$ nor in $T_2$ such that

$p_1$ in $\forall x{:}S_1.\ T_1[x/x_1] \Rightarrow T_2[x/x_2]$

and $p_2$ in $\forall x{:}S_1.\ T_2[x/x_2] \Rightarrow T_1[x/x_1]$. (violates separation principle)

· $s = t$ in $\{x{:}S \mid T\}$ if $\{x{:}S \mid T\}$ type,

$s{=}t$ in $S$, and there is some $p$ in $T[s/x]$.



## Subset Types: Proof theory

**Proof rules must manage implicit information**

- We "know" $T[s]$ if $s$ in $\{x{:}S \mid T\}$
- We cannot use the proof term for $T[s]$ computationally
- Proof term for $T[s]$ must be available in non-computational proof parts
- Some refinement rules generate hidden assumptions

$\Gamma, z{:}\{x{:}S \mid T\}, \Delta \vdash C$ ext $(\lambda y.t)\ z$

**by setElimination** $i\ y\ v$

$\Gamma, z{:}\{x{:}S \mid T\}, y{:}S, [\![v]\!]{:}T[y/x], \Delta[y/z] \vdash C[y/z]$ ext $t$

- Hidden assumptions made visible by refinement rules with extract term Ax



## Intersection Types: Polymorphism without parameters

**• Represent mathematical concept of intersection**

- $\cap x{:}S.T[x]$ formally similar to dependent functions $x{:}S{\to}T[x]$

  ...but ...

- Members are elements of all $T[s]$ with $s \in S$, not functions
- "Range parameter" $s \in S$ only implicitly present

**Syntax:**

Canonical: $\cap x{:}S.T[x]$

Noncanonical: —

**Evaluation:** —

**Semantics:**

· $\cap x{:}S.T[x]$ is a type if $S$ is a type and $T[e/x]$ is a type for all $e$ in $S$

· $s = t$ in $\cap x{:}S.T[x]$ if $\cap x{:}S.T[x]$ type and

$s = t$ in $T[e/x]$ for all $e$ in $S$



## Quotient Types: User-Defined Equality

**• Representation of equivalence classes**

- Members of $x,y : T//E$ are elements of $T$ (but $x,y : T//E \not\subseteq T$)
- Equality $s{=}t$ redefined as $E[s,t/x,y]$
- $E$ must be type of an equivalence relation

**Syntax:**

Canonical: $x,y : T//E$

Noncanonical: —

**Semantics:**

· $x_1,y_1 : T_1//E_1 = x_2,y_2 : T_2//E_2$ if $T_1 = T_2$ and there are terms $p_1p_2r$, $s,t$ and variables $x, y, z$, which occur neither in $E_1$ nor in $E_2$ such that

$p_1$ in $\forall x{:}T_1.\forall y{:}T_1.\ E_1[x,y/x_1,y_1] \Rightarrow E_2[x,y/x_2,y_2]$,

$p_2$ in $\forall x{:}T_1.\forall y{:}T_1.\ E_2[x,y/x_2,y_2] \Rightarrow E_1[x,y/x_1,y_1]$,

$r$ in $\forall x{:}T_1.\ E_1[x,x/x_1,y_1]$,

$s$ in $\forall x{:}T_1.\forall y{:}T_1.\ E_1[x,y/x_1,y_1] \Rightarrow E_1[y,x/x_1,y_1]$,

and $t$ in $\forall x{:}T_1.\forall y{:}T_1.\forall z{:}T_1.\ E_1[x,y/x_1,y_1] \Rightarrow E_1[y,z/x_1,y_1] \Rightarrow E_1[x,z/x_1,y_1]$

· $s = t$ in $x,y : T//E$ if $x,y : T//E$ type, $s$ in $T$, $t$ in $T$,

and there is some term $p$ in $E[s,t/x,y]$

## Quotient Types: Proof theory

**Proof rules must manage implicit information**

- We "know" $E[s,t/x,y]$ if $s = t$ in $x,y : T//E$
- Proof term for $E[s,t/x,y]$ can only be used non-computationally
- Hidden assumptions generated by decomposing equalities in hypotheses

$\Gamma, v{:}s = t \in x,y : T//E, \Delta \vdash C$ ext $u_j$
by quotient_equalityElimination $i$ $j$ $v'$
$\Gamma, v{:}s = t \in x,y : T//E, [v']{:}E[s,t/x,y], \Delta \vdash C$ ext $u_j$
$\Gamma, v{:}s = t \in x,y : T//E, \Delta \vdash E[s,t/x,y] \in \mathbf{U}_j$ A$_j$

**User-predicates may require type-squashing**

- $\downarrow P \equiv \{\mathtt{x{:}Top} \mid P\}$: reduce $P$ to it's truth content
- Necessary if there is too much structure on $x,y : T//E$



## Dependent Intersection

- **Intersection with self-reference**
  - $x{:}S\cap T$ somewhat similar to dependent products $x{:}S\times T[x]$
    ... but ...
  - Members are elements $s \in S$ with $s \in T[s]$ ("very dependent pairs")

**Syntax:**

Canonical: $x{:}S\cap T$

Noncanonical: —

**Evaluation:** —

**Semantics:**

- $x{:}S\cap T$ is a type if $S$ is a type and $T[e/x]$ is a type for all $e$ in $S$
- $s = t$ in $x{:}S\cap T$ if $x{:}S\cap T$ type, $s = t$ in $S$, and $s = t$ in $T[s]$

**Useful for representing dependent records, ADT's, objects, etc.**



## Important Defined Types

- Integer ranges: $\mathbb{N} \equiv \{\mathtt{i{:}Z \mid 0{\le}i}\}$, $\{\mathtt{j...}\} \equiv \{\mathtt{i{:}Z \mid j{\le}i}\}$,
  $\mathbb{N}^+ \equiv \{\mathtt{i{:}Z \mid 0{<}i}\}$, $\{\mathtt{...j}\} \equiv \{\mathtt{i{:}Z \mid i{\le}j}\}$
- Logic: $\forall$ $\exists$ $\wedge$ $\vee$ $\Rightarrow$ $\neg$ `True` `False` (Curry-Howard isomorphism)
- Singleton type: `Unit` $\equiv$ $0 \in \mathbf{Z}$
- Boolean: $\mathbf{B} \equiv$ `Unit + Unit`, $\uparrow b \equiv$ `if` $b$ `then True else False`
- Top type: `Top` $\equiv \cap$`x:Void.Void`
- Subtyping: $S \sqsubseteq T \equiv \forall \mathtt{x{:}}S.\ \mathtt{x} \in T$
- Type squashing: $\downarrow P \equiv \{\mathtt{True} \mid P\}$
- Recursive functions: $\mathbf{Y} \equiv$ `λf. (λx.f (x x)) (λx.f (x x))`
- (Dependent) records $\{x_1{:}T_1;\ x_2{:}T_2[x_1]; \ldots;\ x_n{:}T_n[x_1..x_{n-1}]\}$ (→ part IV)

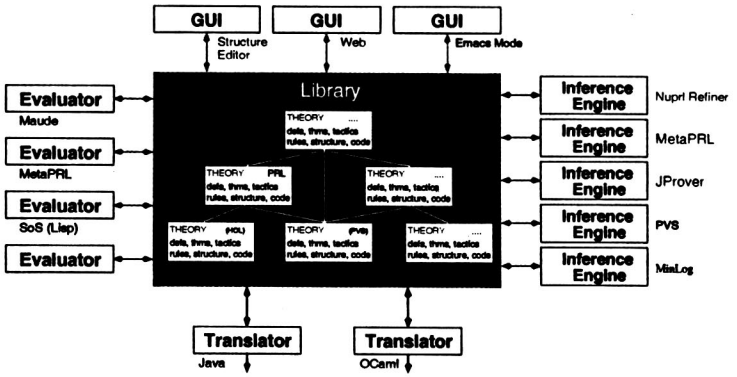
See the standard theories of NUPRL 5 for further details



# Part II:

# The Nuprl System

## Nuprl's Automated Reasoning Environment

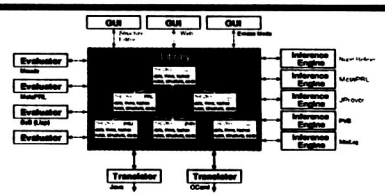



**Interactive proof development**

- Supports program extraction and evaluation
- Proof automation through tactics & decision procedures
- Highly customizable: conservative language extensions, term display, ...
- Supports cooperation with other proof systems



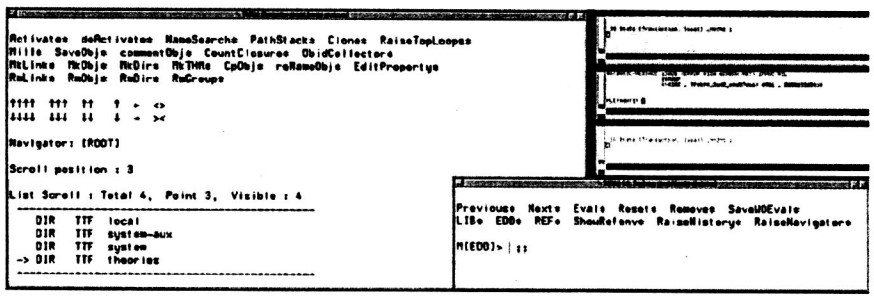
## System Architecture (Allen et. al, 2000)



- **Collection of cooperating processes**
  - ⇝ Asynchronous, distributed & collaborative theorem proving ("interoperability")
- **Centered around a common knowledge base**
  - Library of formal algorithmic knowledge
  - Persistent data base, version control, dependency tracking ⇝ accountability
- **Connected to external systems**
  - MetaPRL (fast rewriting, multiple logics) (Hickey & Nogin, 1999)
  - JProver (matrix-based intuitionistic theorem prover) (IJCAR 2001)
- **Multiple user interfaces**
  - Structure editor, web browser ⇝ collaborative proving
- **Reflective system structure**
  - System designed within the system's library ⇝ customizability



## Initial Nuprl 5 screen




- **Navigator** for browsing and invoking editors
- **ML top loop** for entering meta-level commands
- 3 windows for library, refiner, and editor Lisp processes



## Features of the Proof Development System

- **Interactive proof editor** ⇝ readable proofs
- **Flexible definition mechanism** ⇝ user-defined terms
- **Customizable term display** ⇝ flexible notation
- **Structure editor for terms** ⇝ no ambiguities
- **Tactics & decision procedures** ⇝ user-defined inferences
- **Proof objects, program extraction** ⇝ program synthesis
- **Program evaluation**
- **Library mechanism** ⇝ user-theories
  - Large mathematical libraries & tactics collection
- **Command interface: navigator + ML top loops**
- **Formal documentation mechanism** ⇝ LaTeX, HTML

## Basic Navigator Operations

- **Creating, copying, renaming, removing, printing objects, directories, and links**
  - Objects will never be destroyed – only references to objects change
- **Browsing and searching the library**
- **Invoking editors on objects**
- **Checking theories**
- **Importing and exporting theories**
- **Invoking operations on collections of objects**



## The Proof Editor

- **Invoke proof editor by opening an object of kind THM**
- **State theorem as top goal, using structured term editor**
- **Prove a goal by entering proof tactics and parameters after the BY**
- **Proof editor refines goal and displays remaining subgoals**
  - Proof steps are immediately committed to library
  - Proof engine may be invoked asynchronously
- **User can move into subgoal nodes if necessary**

```
* top
∀A,B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY D 0
* 1
1. A:ℙ
⊢ ∀B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY Auto
* 1 1
2. B:ℙ
3. (¬A) ∨ (¬B)
⊢ ¬(A ∧ B)
BY D 0 THEN D 3 THEN Auto
* 2
.....wf.....
ℙ ∈ U'
BY Auto
```

- **Proof editor may generate extract terms from complete proofs**



## The Structured Term Editor

**Edit internal structure of terms while showing external display**

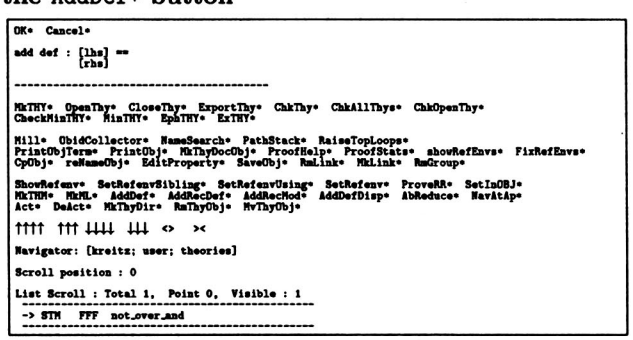
- **Invoke by opening or entering a term slot**
- **Entering *opid* of term opens template**
  e.g. ⌞`exists`↵⌟ generates the template ⌜∃[var]:[type]. [prop]⌝
  - [type] and prop] are new term slots, [var] is a text slot
- **Users may navigate through term tree and edit subterms**
  - Motion by mouse or **emacs**-like key combinations (`m-p`, `m-b`, `m-f`, `m-n`)
  - Cutting and pasting of terms possible (`c-k`, `m-k`, `c-y`)
  - Text oriented editing possible as well
  - Insert non-ASCII characters with `c-#`*num*
- **Internal structure can be made visible**
  - Explode (`c-x ex`) and implode (`c-x im`) terms
  - Entirely new terms can be inserted by entering *opid*{*parms*}(arity)
- **Terms have hyperlinks to abstractions and display forms**
  - Use `c-x ab` / Mouse-Right and `c-x df` / Mouse-Middle



## Creating Definitions

**Define new terms in terms of existing ones**

- **Click the AddDef* button**




- **Insert a new term into the [lhs] slot** ⌜`exists_uni(T; x.P[x])`⌝
- **Enter its definition into [rhs]** ⌜∃x:T. P[x] ∧ (∀y:T. P[y] ⇒ y=x∈T)⌝
  - All free variables of the new term must occur
- **Edit the generated display form and wellformedness theorem**



1 v2 s 1

## Modifying the Term Display

- **Open display form object for the term**
  - create a new one if necessary

```
EdAlias exists_uni ::
  exists_uni(<T:T:*>;<x:var:*>.<P:P:*>)
  == exists_uni(<T>;<x>.<P>)
```

- **Edit text on left hand side of ==**
  - Special characters may be inserted, e.g. `c-# 163` inserts ∃
  - Template slots may be moved or deleted (mark with `m-p`)
  - Slot description between colons may be modified
  - Precedences for use of parentheses may be described after last colon

```
EdAlias exists_uni ::
  ∃!<x:var:*>:<T:type:*>. <P:prop:*>)
  == exists_uni(<T>;<x>.<P>)
```

- **Add additional display forms for iteration and special cases**
  - Iteration: instead of ⌜∀x:T. ∀y:T. P⌝ display ⌜∀x,y:T. P⌝
  - Special cases: instead of ⌜x=y∈Z⌝ display ⌜x=y⌝ (delete the type slot)



## Evaluation of Terms

- **Invoke the term evaluator on a Nuprl term by entering `␣view_showc` *name term*␣ into the editor ML top loop**

```
Compute1*  Compute5*  Compute10*  ComputeAll*

((β * 4) - 5) + 6
```

- **Click the buttons to perform one top-level reduction steps**
  - Use `c-_` to undo a step



## Extracting Programs from Proofs

- **Generate extract term of completed proof**
  - Close proof editor with `c-z` instead of `c-q`
- **Make extract term available for editing**
  - Enter `␣require_termof (ioid` *obid*`)␣` into the editor ML top loop
  - *obid* is abstract identifier of proof object
    mark in navigator with left mouse and copy into top loop with `c-y`
- **Open term evaluator on extract term**
  - Enter `␣view_show_co` *obid*␣ into the editor ML top loop

```
Compute1*  Compute5*  Compute10*  ComputeAll*

TERMOF{intsqrt:o, \\v:1}
```

- **Evaluate one step to see the extract**
  - Edit term to supply arguments to a Nuprl function, if desired

**Should be simplified in the future**



# Part III:

# Proof Automation in Nuprl

## Automating the Construction of Proofs

- **Tactics:** Programmed application of inference rules
  - Easy to implement, even by users
  - Flexible, guaranteed to be correct
- **Rewriting:** Replace terms by equivalent ones
  - Computational and definitional equality
  - Derived equivalences in lemmata and hypotheses
- **Decision Procedures:** Solve problems in narrow application domains
  - Translate proof goal into different problem domain
  - Use efficient algorithms for checking translated problems
- **Proof Search Procedures:** Compact representation of proof tree
  - "Unintuitive", but efficient proof procedure
  - Only for "small" theories
  - Correct integration into interactive proof system?



## Tactics: User-defined inference rules

- **Meta-level programs built using**
  - Basic inference rules
  - Predefined tacticals ...
  - Meta-level analysis of the proof goal and its context
  - Large collection of standard tactics in the library
- **May produce incomplete proofs**
  - ↦ User has to complete the proof by calling other tactics
- **May not terminate**
  - ↦ User has to interrupt execution

but

**Applying a tactic always results in a valid proof**



## Basic Tactics

**Subsume primitive inferences under a common name**

| | |
|---|---|
| `Hypothesis`: | *Prove* $\ulcorner \ldots C \ldots \vdash C' \urcorner$ *where* $C'$ *α-equal to* $C$ |
| `Declaration`: | *Prove* $\ulcorner \ldots x{:}T \ldots \vdash x \in T' \urcorner$ *where* $T'$ *α-equal to* $T$ |
| | *Variants:* `NthHyp` *i*, `NthDecl` *i* |
| `D` *c*: | *Decompose the outermost connective of clause c* |
| `EqD` *c*: | *Decompose immediate subterms of an equality in clause c* |
| `MemD` *c*: | *Decompose subterm of a membership term in clause c* |
| | Variants: `EqCD` , `EqHD` *i*, `MemCD` , `MemHD` *i* |
| `EqTypeD` *c*: | *Decompose type subterm of an equality in clause c* |
| `MemTypeD` *c*: | *Decompose type subterm of a membership term in clause c* |
| | *Variants:* `EqTypeCD` , `EqTypeHD` *i*, `MemTypeCD` , `MemTypeHD` *i* |
| `Assert` *t*: | *Assert (or cut) term t as last hypothesis* |
| `Auto`: | *Apply trivial reasoning, decomposition, decision procedures ...* |
| `Reduce` *c*: | *Reduce all primitive redices in clause c* |



## Parameters in Tactics

| | |
|---|---|
| **Position of a hypothesis to be used** | `NthHyp` $\boxed{i}$ |
| **Names for newly created variables** | `New` $[x]$ `(D 0)` |
| **Type of some subterm in the goal** | `With` $\boxed{x{:}S \to T}$ `(MemD 0)` |
| **Term to instantiate a variable** | `With` $\boxed{s}$ `(D 0)` |
| **Universe level of a type** | `At` $\boxed{j}$ `(D 0)` |
| **Dependency of a term instance $C[z]$ on a variable $z$** | `Using` $\boxed{[z, C]}$ `(D 0)` |

## TACTICALS

### Compose tactics into new ones

| | |
|---|---|
| $tac_1$ `THEN` $tac_2$: | *Apply $tac_2$ to all subgoals created by $tac_1$* |
| $t$ `THENL` $[tac_1; \ldots; tac_n]$: | *Apply $tac_i$ to the i-th subgoal created by t* |
| $tac_1$ `THENA` $tac_2$: | *Apply $tac_2$ to all auxiliary subgoals created by $tac_1$* |
| $tac_1$ `THENW` $tac_2$: | *Apply $tac_2$ to all wf subgoals created by $tac_1$* |
| $tac_1$ `ORELSE` $tac_2$: | *Apply $tac_1$. If this fails apply $tac_2$ instead* |
| `Try` *tac:* | *Apply tac. If this fails leave the proof unchanged* |
| `Complete` *tac:* | *Apply tac only if this completes the proof* |
| `Progress` *tac:* | *Apply tac only if that causes the goal to change* |
| `Repeat` *tac:* | *Repeat tac until it fails* |
| `RepeatFor` *i tac:* | *Repeat tac exactly i times* |
| `AllHyps` *tac:* | *Try to apply tac to all hypotheses* |
| `OnSomHyp` *tac:* | *Apply tac to the first possible hypotheses* |



## ADVANCED TACTICS

- **Induction**
  - `NatInd` *i: standard natural-number induction on hypothesis i*
  - `IntInd`, `NSubsetInd`, `ListInd`*: induction on* $\mathbb{Z}$, $\mathbb{N}$ *subranges, lists*
  - `CompNatInd` *i: complete natural-number induction on hypothesis i*
- **Case Analysis**
  - `BoolCases` *i: case split over boolean variable in hypothesis i*
  - `Cases` $[t_1; \ldots; t_n]$*: n-way case split over terms* $t_i$
  - `Decide` *P: case split over (decidable) proposition P and its negation*
- **Chaining**
  - `InstHyp` $[t_1; \ldots; t_n]$ *i: instantiate hypothesis i with terms* $t_1 \ldots t_n$
  - `FHyp` *i* $[h_1; \ldots; h_n]$*: forward chain through hypothesis i matching its antecedents against any of the hypotheses* $h_1 \ldots h_n$
  - `BHyp` *i: backward chain through hypothesis i matching its consequent against the conclusion of the proof*
  - `Backchain` *bc_names: backchain repeatedly through lemmas and hypotheses*

*Variants:* `InstLemma name` $[t_1; \ldots; t_n]$, `FLemma name` $[h_1; \ldots; h_n]$, `BLemma name`.



## WRITING A TACTIC-BASED PROOF SEARCH PROCEDURE IS EASY

### Sort rule applications by cost of induced proof search

```
let simple_prover = Repeat
                    (       hypotheses
                     ORELSE contradiction
                     ORELSE InstantiateAll
                     ORELSE InstantiateEx
                     ORELSE conjunctionE
                     ORELSE existentialE
                     ORELSE nondangerousI
                     ORELSE disjunctionE
                     ORELSE not_chain
                     ORELSE iff_chain
                     ORELSE imp_chain
                    );;

letrec prover = simple_prover
                THEN Try (        Complete (orI1 THEN prover)
                          ORELSE (Complete (orI2 THEN prover))
                ;;
```



## simple_prover: COMPONENT TACTICS

```
let contradiction    = TryAllHyps falseE    is_false_term
and conjunctionE     = TryAllHyps andE      is_and_term
and existentialE     = TryAllHyps exE       is_ex_term
and disjunctionE     = TryAllHyps orE       is_or_term

and nondangerousI pf = let kind = operator_id_of_term (conclusion pf)
                       in
                          if mem mkind ['all'; 'not'; 'implies';
                                        'rev_implies'; 'iff'; 'and']
                             then Run (termkind ^ 'R') pf
                             else failwith 'tactic inappropriate'
                       ;;

let imp_chain pf  = Chain impE (select_hyps is_imp_term pf) hypotheses pf
                    ;;
let not_chain     = TryAllHyps (\pos. notE pos THEN imp_chain) is_not_term
                    ;;
let iff_chain     = TryAllHyps (\pos. (iffE   pos THEN (imp_chain
                                                        ORELSE not_chain))
                                      ORELSE
                                      (iffE_b pos THEN (imp_chain
                                                        ORELSE not_chain))
                                   ) is_iff_term
                    ;;
```

## simple_prover: Matching and instantiation

```
let InstantiateAll =
   let InstAll_aux pos pf =
        let concl = conclusion pf
        and qterm = type_of_hyp pos pf          in
           let sigma = match_subAll qterm concl in
              let terms = map snd sigma         in
                 (allEon pos terms THEN (OnLastHyp hypothesis)) pf
   in
      TryAllHyps InstAll_aux  is_all_term
;;
let InstantiateEx =
  let InstEx_aux pos pf =
       let qterm = conclusion pf
       and hyp = type_of_hyp pos pf           in
          let sigma = match_subEx qterm hyp   in
             let terms = map snd sigma        in
                (exIon terms THEN (hypothesis pos)) pf
 in
      TryAllHyps InstEx_aux (\h.true)
;;
```



## Rewriting: replace terms by equivalent ones

- **Simple rewrite tactics**
  - `Fold` *name c:* *fold abstraction name in clause c*
  - `Unfold` *name c:* *unfold abstraction name in clause c*
  - `Subst` $t_1 = t_2 \in T$ *c:* *substitute $t_1$ by $t_2$ in clause c*
  - `Reduce` *c:* *repeatedly evaluate redices in clause c*
- **Nuprl's rewrite package**
  - Functions for creating and applying term rewrite rules
  - Supports various equivalence relations
  - Based on tactics for applying conversions to clauses in proofs
- **Conversions**
  - Language for systematically building rewrite rules
  - Transform terms and provide justifications
  - Need to be supported by various kinds of lemmata
  - Organized like tactics: atomic conversions, conversionals, advanced conversions



## Atomic Conversions

- **Folding and Unfolding Abstractions**
  - `UnfoldC` *abs*: *Unfold all occurrences of abstraction abs*
  - `FoldC` *abs* : *Fold all instances of abstraction abs*

  *Versions for (un)folding specific instances available as well*
- **Evaluating Redices**
  - `ReduceC`: *contract all primitive redices*
  - `AbReduceC`: *contract primitive and abstract (user-defined) redices*
- **Applying Lemmata and Hypotheses**
  - Universally quantified formulas with consequent *a r b*
  - `HypC` *i:* *rewrite instances of a into instances of b*
  - `RevHypC` *i:* *rewrite instances of b into instances of a*

  *Variants:* `LemmaC` *name*, `RevLemmaC` *name*



## Building Rewrite Tactics

- **Construct advanced Conversions using Conversionals**
  - `ANDTHENC`, `ORTHENC`, `ORELSEC`, `RepeatC`, `ProgressC`, `TryC`
  - `SubC`, `NthSubC`, `AddrC`, `SweepUpC`, `SweepDnC`, `DepthC`, `AllC`, `SomeC`, `FirstC`
- **Define Macro Conversions**
  - `MacroC` *name* $c_1$ $t_1$ $c_2$ $t_2$ : *Rewrite instance of $t_1$ into instance of $t_2$*

    $c_1$ and $c_2$ must rewrite $t_1$ and $t_2$ into the same term, *name* is a failure token
  - `SimpleMacroC` *name* $t_1$ $t_2$ *abs* : *Rewrite $t_1$ into $t_2$ by unfolding abstractions from abs and contracting primitive redices*
- **Transform Conversions into Tactics**
  - `Rewrite` *c i: Apply conversion c to clause i*

  *Variants:* `RewriteType` *c i*, `RWAddr` *addr c i*, `RWU`, `RWD`



1-1-29.

## Decision Procedures

- **Decide problems in narrow application domains**
  - Translate proof goal into different problem domain
  - Decide translated problem using efficient standard algorithms
  - Implement directly in NUPRL or connect as external proof tool
- **Currently available**
  - `ProveProp`: simple propositional reasoning
  - `Eq`: trivial equality reasoning (limited congruence closure algorithm)
  - `RelRST`: exploit properties of binary relations (find shortest path in relation graph)
  - `Arith`: standard, induction-free arithmetic
  - `SupInf`: solve linear inequalities over **Z**



## Arith: induction-free arithmetic

- **Input sequent:** $H \vdash C_1 \vee \ldots \vee C_m$
  - $C_i$ is an arithmetic relation over **Z** built from $<, \leq, >, \geq, =, \neq$, and $\neg$
- **Theory covered:**
  - ring axioms for + and *
  - total order axioms of <
  - reflexivity, symmetry and transitivity of =
  - limited substitutivity
- **Proof procedure:**
  - Translate sequent into a directed graph whose egdes are labeled with natural numbers
  - Check if the graph contains positive cycles
- **Implemented as NUPRL procedure (Lisp level)**
- **Integrated into the tactic `Auto`**



## SupInf: linear inequalities over **Z**

- **Adaptation of Bledsoe's Sup-Inf method**
  - Complete only for the rationals
  - Sound for integers
- **Proof procedure:**
  - Convert sequent into conjunction of terms $0 \leq e_i$ where each $e_i$ is a linear expression over **Q** in variables $x_1 \ldots x_n$
  - Check if some assignment of values to the $x_j$ satisfies the conjunction
  - Determine upper and lower bounds for each variable in turn
  - Identify counter-examples if no assignment exists
- **Implemented as NUPRL procedure (ML level)**
- **Integrated into the tactic `Auto'`**



## Proving the Existence of an Integer Square Root

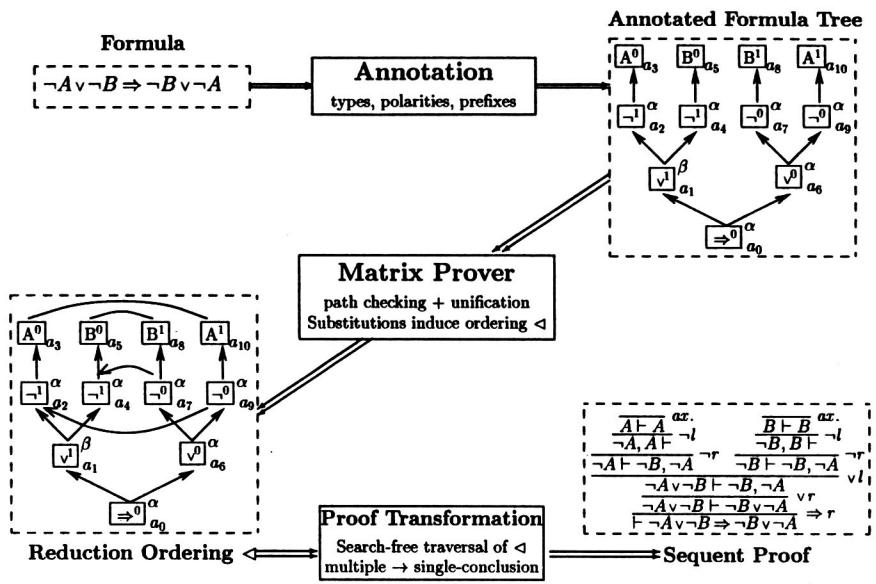
```
* top      ∀n:ℕ. ∃r:ℕ. r² ≤ n < (r+1)²
           BY allR

* 1        1. n : ℕ
           ⊢ ∃r:ℕ. r² ≤ n < (r+1)²
           BY NatInd 1

* 1 1      .....basecase.....
           ∃r:ℕ. r² ≤ 0 < (r+1)²
           BY Witness 0 (D 0) THEN Auto

* 1 2      .....upcase.....
           1. i : ℕ
           2. 0 < i
           3. r : ℕ
           4. r² ≤ i-1 < (r+1)²
           ⊢ ∃r:ℕ. r² ≤ i < (r+1)²
           BY Decide ⌜(r+1)² ≤ i⌝ THENW Auto

* 1 2 1    5. (r+1)² ≤ i
           ⊢ ∃r:ℕ. r² ≤ i < (r+1)²
           BY Witness ⌜r+1⌝ (D 0) THEN Auto'

* 1 2 2    5. ¬((r+1)² ≤ i)
           ⊢ ∃r:ℕ. r² ≤ i < (r+1)²
           BY Witness ⌜r⌝ (D 0) THEN Auto
```

## Integrating Complete Proof Search Procedures

- **Tactic-based proof search has limitations**
  - Many proofs require some "lookahead"
  - Proof search must perform meta-level analysis first
- **Complete proof search procedures are "unintuitive"**
  - Proof search tree represented in compact form
  - Link similar subformulas that may represent leafs of a sequent proof
  - Proof search checks if all leaves can be covered by connections and if parameters all connected subformulas can be unified
- **JProver: inutionistic proof search for NUPRL**
  - Find matrix proof of goal sequent and convert it into sequent proof

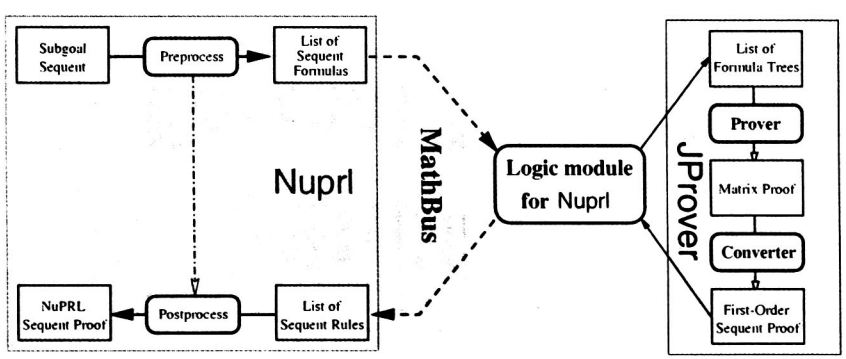


## JProver: Proof Methodology (Kreitz, Otten, Schmitt 1995–2000)





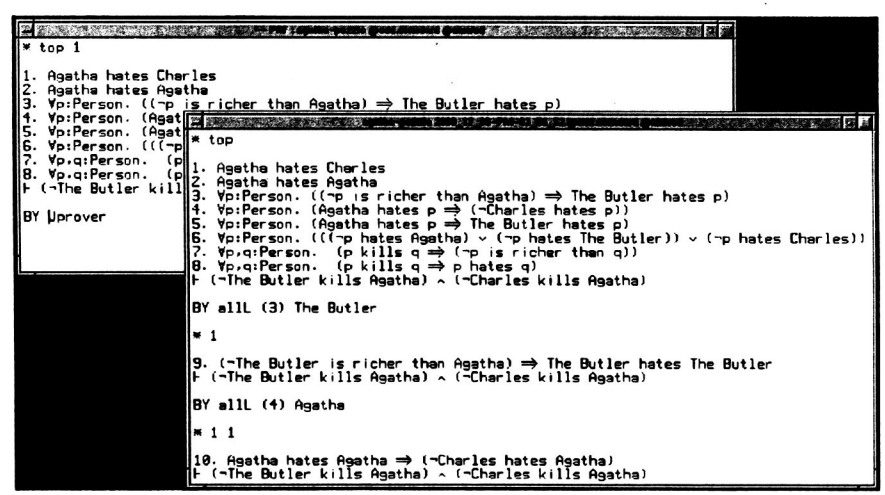

## JProver: Integration Architecture (Schmitt, et. al 2001)




- Communicate formulas in uniform format (MathBus) over INET sockets
- Logic module converts between internal term representations
- Pre- and postprocessing in NUPRL widens range of applicability



## Solving the "Agatha Murder Puzzle"




JProver can run in trusted mode or with all proof details expanded

# Part IV:

# Building Formal Theories



## An elegant account of Record Types

- **Express records as (dependent) functions from labels to types**
  - $\{x_1{:}T_1;\ ..\ ;\ x_n{:}T_n\}$ ≡ `l:Labels → if l=`$x_i$ `then` $T_i$ `else Top`
  - $\{x_1{=}t_1;\ ..\ ;\ x_n{=}t_n\}$ ≡ `λl.if z=`$x_i$ `then` $t_i$ `else ()`
  - $r.l$ ≡ $(r\ l)$
- **Dependent Records** $\{x_1{:}T_1;\ x_2{:}T_2[x_1];..\ ;\ x_n{:}T_n[x_1;..x_{n-1}]\}$
  - Type $T_i$ may depend on value of components $x_1;..x_{i-1}$
  - Used for describing algebra, abstract data types, inheritance, ...
- **Use (dependent) intersection to formalize both**

| | |
|---|---|
| $\{x{:}T\}$ | ≡ `z:Labels → if z=`$x$ `then` $T$ `else Top` |
| $\{R_1;\ R_2\}$ | ≡ $R_1 \cap R_2$ |
| $\{x{:}S;\ y{:}T[x]\}$ | ≡ `r:`$\{x{:}S\} \cap \{y{:}T[r.x]\}$ |
| $r.l$ | ≡ $(r\ l)$ |
| $r.l{<}{-}t$ | ≡ `λz. if z=`$l$ `then` $t$ `else r.z` |
| $\{\}$ | ≡ `λl.()` |
| $\{r;\ l{=}t\}$ | ≡ $r.l{<}{-}t$ |

**⇝ Subtyping** $\{x_1{:}T\} \sqsubseteq \{x_1{:}T_1;\ x_2{:}T_2[x_1]\}$ **is easy to prove**

**Syntax of iterations can be adjusted using display forms**



## Formal Algebra: Semigroups

**Tuple $(M, \circ)$ where $M$ is a type and $\circ{:}M{\times}M{\to}M$ associative**

- **Formalization as dependent product**

`SemiGroup ≡ M:U × ∘:M×M→M × ∀x,y,z:M. x∘(y∘z) = (x∘y)∘z ∈ M`

⇝ **semigroups represented as triples** $(M, \circ, assoc\_pf)$

- **Formalization via set types**

```
SemiGroupSig ≡ M:U × ∘:M×M→M
SemiGroup    ≡ {sg:SemiGroupSig | ∀x,y,z:M_sg. x∘_sg(y∘_sg z) = (x∘_sg y)∘_sg z ∈ M_sg}
```

⇝ **tedious to access components or use associativity in proofs**

- **Formalization via dependent records**

```
SemiGroupSig ≡ {M:U; ∘:M×M→M}
SemiGroup    ≡ {SemiGroupSig; assoc: ↓(∀x,y,z:M. x∘(y∘z) = (x∘y)∘z ∈ M)}
```

⇝ **Accessing components and properties straightforward**

⇝ **Type squashing suppresses explicit proof component**

⇝ **Subtyping relation `SemiGroup` ⊑ `SemiGroupSig` easy to prove**



## Formal Algebra: Monoids and Groups

- **Monoid: semigroup with identity**

```
MonoidSig ≡ { SemiGroupSig; e:M}
Monoid    ≡ { SemiGroup; MonoidSig; id: ↓(∀x:M. e∘x=x∈M)}
```

⇝ **natural use of multiple inheritance**

- **Group: monoid with inverse**

```
GroupSig ≡ { MonoidSig; ⁻¹:M→M}
Group    ≡ { Monoid;    GroupSig; inv: ↓(∀x:M.x∘x⁻¹=e∈M)}
```

⇝ **refinement hierarchy follows directly from definitions**

```
SemiGroup ⊑ SemiGroupSig
   ⊑           ⊑
Monoid    ⊑ MonoidSig
   ⊑           ⊑
Group     ⊑ GroupSig
```

## Formalization: Abstract Data Types

- **Abstract Data Type for stacks over a type $T$**

```
TYPES        Stack
OPERATORS    empty:   Stack
             push:    Stack×T → Stack
             pop:     {s:Stack|s≠empty} → Stack×T
AXIOMS       pushpop: ∀s:Stack.∀t:T. pop(push(s,a)) = (s,a)
```

- **Formalization**
  - Dependent products unsuited for same reason as above
  - Dependent records lead to "natural formalization"

```
STACKSIG(T) ≡ { Stack:U
              ; empty:   Stack
              ; push:    Stack×T → Stack
              ; pop:     {s:Stack|s≠empty} → Stack×T}
STACK(T)    ≡ {STACKSIG(T); pf: ↓(∀s:Stack.∀t:T. pop(push(s,a))=(s,a) ∈M)}
```

- **Formalizing the implementation of stacks through lists**

```
list-as-stack(T) ≡ { Stack = T list
                   ; empty = []
                   ; push  = λs,t. t::s
                   ; pop   = λs. <hd(s),tl(s)> }
```

⇝ `list-as-stack(T) ∈ STACK(T)` **easy to prove**



# Part V:

# Future Directions



## Challenges for Automated Theorem Proving

- **A more expressive theory**
  - Reflection: reasoning about syntax and semantics simultaneously
  - Reasoning about objects, inheritance, liveness, distributed processes, ...
- **A more widely applicable system**
  - Digital Libraries of Formal Knowledge
  - Cooperation between different proof systems
- **Learn more from larg scale applications**
  - Synthesize, verify, and optimize high-assurance software systems
  - Target "unclean" but popular programming languages
  - Aim at pushbutton technology



## Directions in Theory: Reflection

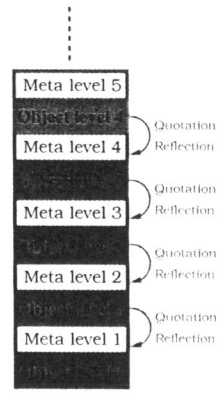
- **Embed meta-level of type theory into type theory**
  - Reason about relation between syntactical form and semantical value
    - evaluation, resources, complexity
    - semantical effects of syntactical transformations (reordering, renaming,...)
    - proofs, tactic applications, dependencies (e.g. proofs ↔ library contents)
    - relations between different formal theories

    ...from within the logic
- **Extremely powerful, but little utilization**
- **Approach: mirror type theory as recursive type**
  - Logically satisfactory, not efficient enough for practical purposes (LICS 1990)
- **New: primitive type of intensional representations**
  - Type `Term`, closed under quotation (Cornell 2001)
  - Theoretically challenging, but much more efficient

## Reflection – basic methodology

- **Represent object and meta level in type theory**
  - Represent meta-logical concepts as NUPRL terms
  - Express specific object logic in represented meta logic
  - Build hierarchy: level $i$ contains meta level for level $i+1$

  $\mapsto$ Reasoning about both levels from the "outside"




- **Link object logic and meta-logic**
  - Embed object level terms using quotation (operator)
  - Embed object level provability using reflection rule

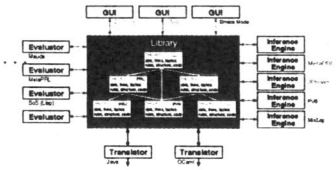
$$H \vdash_{i+1} A \qquad \texttt{by reflection } i$$
$$\vdash_i \exists \texttt{p:Proof}_i.\ \texttt{goal(p)} = \lceil H \vdash_{i+1} A \rceil$$

- **Use same reasoning apparatus for object and meta level**



## Digital Libraries of Formal Algorithmic Knowledge

- **Library as platform for cooperating reasoning tools**
- **Connect**
  - Additional proof engines: PVS, HOL, MinLog, ...
  - Multiple browsers (ASCII, web, ...) and editors (structured, Emacs-mode, ...)




- **Provide new features**
  - Archival capacities (documentation & certification, version control)
  - Embedding external library contents (needs data conversion, proof replay, ...)
  - A variety of justifications (levels of trust)
  - Creation of formal and textual documents
  - Asynchronous and distributed mode of operation
  - Meta-reasoning (e.g. about relations between theories) and reflection

**Improve cooperation between research groups**

$\Downarrow$

**Authoritative reference for reliable software construction**



## Areas for Study & Research

- **EmFormal Logics & Type Theory**
  - Classes & inheritance, recursive & partial objects, concurrency, real-time
  - Meta-reasoning, reflection, relating different logics, ...
- **Theorem Proving Environments**
  - Logical accounting, theory modules, interfaces, proof presentation, ...
- **Automated Proof Search Procedures**
  - Matrix methods, inductive theorem proving, rewriting, proof planning
  - Decision procedures, extended type inference, cooperating provers
  - Proof reuse, analogy, distributed proof procedures, ...
- **Applications**
  - Formal CS knowledge: graph theory, automata, trees, arrays, ...
  - Strategies for program synthesis, verification, and optimization
  - Modeling programming languages (OCaml, Java, ..)

8 Ursula Martin
*University of St. Andrews, Scotland*

## Evening Talk: Adding Mathematics to the Semantic Web

# Computational math: the new challenge for computational logic

Ursula Martin
University of St Andrews
www-theory.dcs.st-and.ac.uk/~um

Thanks to SRI, Waterloo Maple, NAG Ltd

- Dream of computational logic
  - produce significant new theorems -- Fermat, Goldbach....
  - archives of definitive proofs of old ones
- Reality
  - effective (semi) decision procedures for certain problems
  - can formalise any math argument if....
  - but informal math more effective for
    - human checking and communication
    - mathematical progress
- What is mathematics?
  - Proof? Knowledge? Process?



All you ever wanted to know about

computational mathematics

but were afraid to ask

- Symbolic computation
- Numerical computation
- Computational logic

- Symbolic computation — Maple, Mathematica
  - Solve $x^2 - 2x - 4a = 0$ — Soln: $x = 1 + \sqrt{(1 + 4a)}, 1 - \sqrt{(1 + 4a)}$
  - Differentiate $\sin(\cos(x))$ — Soln: $-\sin(x)\ \cos(\cos(x))$
- Numerical computation
- Computational logic

- Symbolic computation — Maple, Mathematica1 mill users
  - Solve $x^2 - 2x - 4a = 0$ — Soln: $x = 1 + \sqrt{(1 + 4a)}, 1 - \sqrt{(1 + 4a)}$
  - Differentiate $\sin(\cos(x))$ — Soln: $-\sin(x)\ \cos(\cos(x))$
  - There is not always a (feasible) symbolic solution!
  - Computer algebra systems are buggy and unpredictable
  - Eventually you need a number....
- Numerical computation
- Computational logic

- Symbolic computation — Maple, Mathematica
  - Solve $x^2 - 2x - 4a = 0$ — Soln: $x = 1 + \sqrt{(1 + 4a)}$, $1 - \sqrt{(1 + 4a)}$
  - Differentiate $\sin(\cos(x))$ — Soln: $-\sin(x) \cos(\cos(x))$
  - There is not always a (feasible) symbolic solution!
  - Computer algebra systems are buggy and unpredictable
  - Eventually you need a number.....
- Numerical computation — MATLAB, NAG library
  - Solve $x^2 - 2x - 4 = 0$ — Soln: $x = 3.236, -1.236$
  - Integrate $\cos(x)$ between $0, \pi/2$ — Soln: 1.0
- Computational logic

- Symbolic computation — Maple, Mathematica 1 mill users
  - Solve $x^2 - 2x - 4a = 0$ — Soln: $x = 1 + \sqrt{(1 + 4a)}$, $1 - \sqrt{(1 + 4a)}$
  - Differentiate $\sin(\cos(x))$ — Soln: $-\sin(x) \cos(\cos(x))$
  - There is not always a (feasible) symbolic solution!
  - Computer algebra systems are buggy and unpredictable
  - Eventually you need a number.....
- Numerical computation — MATLAB, NAG library
  - Solve $x^2 - 2x - 4 = 0$ — Soln: $x = 3.236, -1.236$
  - Integrate $\cos(x)$ between $0, \pi/2$ — Soln: 1.0
  - Workhorse for science, engineering, modelling...
- Computational logic

- Symbolic computation — Maple, Mathematica
  - Solve $x^2 - 2x - 4a = 0$ — Soln: $x = 1 + \sqrt{(1 + 4a)}$, $1 - \sqrt{(1 + 4a)}$
  - Differentiate $\sin(\cos(x))$ — Soln: $-\sin(x) \cos(\cos(x))$
  - There is not always a (feasible) symbolic solution
  - Computer algebra systems are buggy and unpredictable
  - Eventually you need a number.....
- Numerical computation — MATLAB, NAG library
  - Solve $x^2 - 2x - 4 = 0$ — Soln: $x = 3.236, -1.236$
  - Integrate $\cos(x)$ between $0, \pi/2$ — Soln: 1.0
  - Workhorse for science, engineering, modelling, e-science ;+)...
- Computational logic — HOL, PVS
  - Prove that $x^2 - 2x - 4a = 0$ has a real solution for $a > -1/4$
  - Prove that $x = 3.236$ is a "solution" of $x^2 - 2x - 4 = 0$ with error ...
  - Prove that this implementation of Newton-Raphson is....

All you ever wanted to know about
applied mathematics
but were afraid to ask

- **What is applied mathematics?**
- **Continuous mathematics**
- predator prey model: differential equations

$$\frac{dx}{dt} = \alpha x - \beta xy$$

$$\frac{dy}{dt} = \gamma xy - \delta y$$

$x, y$ numbers of foxes and rabbits

$\alpha, \beta, \gamma, \delta$ parameters

  - solve numerically or symbolically for $x, y$
  - explore behavior for different values of $\alpha, \beta, \gamma, \delta$
    - reachability
    - stability
    - phase plane analysis (chaotic behaviour)
  - proof obligations

- **Mathematical modeling**

{model world} — abstraction of reality

⇩

{model} — predator prey equations

⇩

{implementation} — numbers, graphs.....

- **Mathematical modeling**

{reality} — fox, rabbit, grass, relativity....

⇩

{purposes} — are fox/rabbit numbers cyclical?

⇩

{model world} — simplified abstraction of reality

⇩

{model} — predator prey equations

⇩

{implementation} — numbers, graphs....

⇙ ⇘

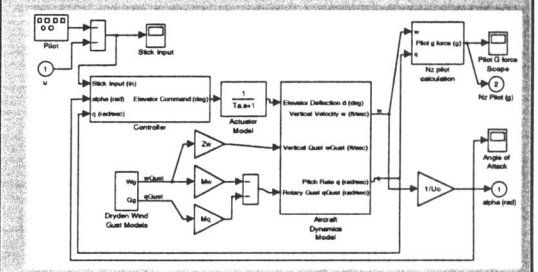
{assessment} experiment — {application} prediction

- **Mathematical modeling**

continuous mathematics

predictions from data+model+assumptions

MATLAB Simulink diagram

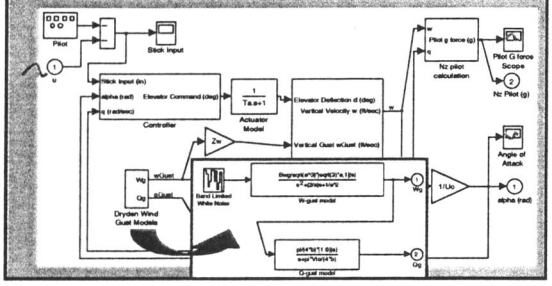



- **Mathematical modeling**

continuous mathematics

predictions from data+model+assumptions

MATLAB Simulink diagram



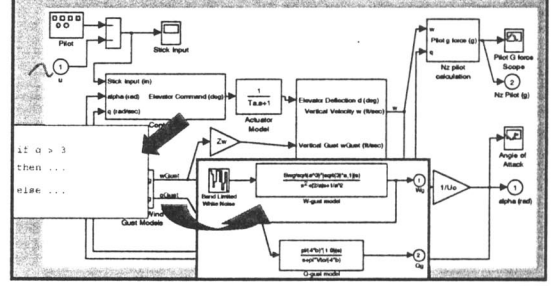

- **Application: embedded systems design**

Current: simulate and test at design stage, manual coding

Ford power train control: 120 "features" + 2000 blocks

Goal: automatic large scale code generation

DARPA MOBIES project




- **Much richer landscape than pure mathematics!**
- **Theories**

Dynamical systems — theories of existence of solutions etc

- **Much richer landscape than pure mathematics!**
- **Theories**

Dynamical systems — theories of existence of solutions etc

Numerical/algebraic analysis — algorithms plus error analysis

Control theory — dynamical systems plus feedback

- **Software**

Numerical methods/computer algebra

robust implementations

Depend on ...

floating point, algebraic primitives

Software systems -- MATLAB, NAG libraries, Maple

- **Applications**

Simulation and design -- models assumptions and requirements

System development -- embedded systems and control

How can we use computational logic to support computational mathematics

Invisible computational logic:

work in familiar informal CM, check/verify in CL

- St Andrews CAAR project: CL to support CM
  - Maple-PVS interface
  - Precondition and result verification for differential equations
  - VSDITLU verified symbolic definite integral table look-up
  - Light-FM for assertions, assumptions and requirements

- Maple-PVS interface

CM Maple 6: symbolic computation + numerics
experiment / solve / simulate / prototype
formulate proof obligations

CL PVS: restricted invocation of highly automated
theorem proving support for simple proofs
strategies for continuity, convergence...
d'base of lemmas about elementary functions (cos, ln, exp...)

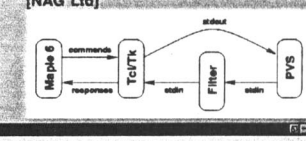
- **Maple/PVS interface** [NAG Ltd]

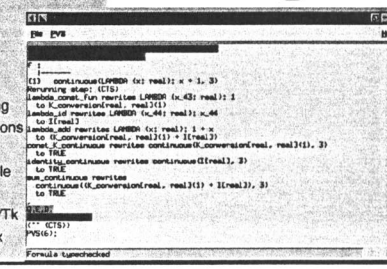
Maple: calls to PVS
PVS:return yes/fail/more...

6 Maple primitives

PVS:
automation
continuity
convergence
equality testing
d'base el'y functions

Maple 80 L maple
380 L C
pvs-ctl 370 L Tcl/Tk
pvs-filter80 L Lex




- **Precondition and result verification for differential equations**

- Solve $\frac{dy}{dx} + f(x).y = g(x), y(a) = b$ in $[c, d]$
  a, b, c, d P K under constraints $C$ over K

Solve in interval [p,q]

$$\frac{dy}{dx} + \frac{y}{2\sqrt{x-c}} = \exp(-\sqrt{x-c})\ln(d-x), \quad y(0) = 0$$

`Maple returns`

$$y = \exp(-\sqrt{x-c})((d-x)\ln(d-x) + x - d\ln(d))$$

What if [p,q] is [1,3], c = 5, d = 7?

Query and result meaningless -- Maple does not check this!

Query and result only defined for d > x > c

- **Precondition and result verification for differential equations**

- Solve $\frac{dy}{dx} + f(x).y = g(x), y(a) = b$ in $[c, d]$
  a, b, c, d P K under constraints $C$ over K

Maple returns

$y(x) = h(x)$

- To check solution exists need to show
  $f(x)$, $g(x)$ continuous in $[c,d]$ under $C$
  $a$ P $[c,d]$

- To check the result need to show
  $h(x)$ satisfies the equation algebraically
  $h(x)$ defined and continuous in $[c,d]$ under $C$

- **Maple harness for dsolve** qsolve(1/x|x-2,1,2,a,3,{a<2,-a<0},{});

PVS calls supported by : automation of el'y functions, continuity

```
% Solve  [diff(y(x),x)+y(x)/x-x-2 = 0, y(1) = 2]
% in the interval (a,3), under constraints {a-2 < 0, -a < 0}
% Maple solution is  y(x) = 1/3*(x^3+3*x^2+2)/x
...
% PVS call to check solution valid
solcheck_ex3_1 : LEMMA
  FORALL (a : real) :
    FORALL (x : I2(a,3)) :
     (a-2 < 0 and -a < 0) IMPLIES
     (y=lambda x: (1/3*(x^3+3*x^2+2)/x)
       IMPLIES(diff1(y,x,x+2-y/x) and y(1) = 2)))
...
% REQUIRES (a-2 < 0 and -a < 0) implies 1/x is
%  continuous in [a,3]
continterval_ex3_1 : LEMMA
  FORALL (a : real) :
  FORALL (zi : I2(a,3)) :
    (a-2 < 0 and -a < 0) IMPLIES
      continuous(LAMBDA (xi : I2(a,3)) : 1/xi, zi)
```

- **VSDITLU verified symbolic definite integral table look-up**

  verified look up table for definite integration

  use PVS to discharge side conditions

  Typical table entry

$$\int_a^b \frac{dx}{x-c} = \begin{cases} \ln(b-c)/(a-c) & \text{if } c < a < b \\ \ln(b-c)/(a-c) & \text{if } a < b < c \\ \textit{undefined} & \text{if } a < c < b \end{cases}$$

- **VSDITLU verified symbolic definite integral table look-up**

  verified look up table

  use PVS to discharge side conditions

  Typical table entry

$$\int_a^b \frac{dx}{x-c} = \begin{cases} \ln(b-c)/(a-c) & c < a < b \\ \ln(b-c)/(a-c) & a < b < c \\ \textit{undefined} & a < c < b \end{cases}$$

  Typical query

$$\int_{t+1}^{t+2} \frac{dx}{x-t}$$ returns **ln(2)**

  use PVS to discharge side conditions when a = t + 1, b = t + 2, c = t

- **Light-FM for assertions, assumptions and requirements**
  - Aldor: NAG/Maple internal development language
    category/domain object model
  - Aldor-FMlite project
    annotate code with **assertions**
    interface specifications as high level operational semantics for trusted components
    `uses/requires/modifies/ensures/assumes`
    tools for verification condition generation
  - Applications
    locate type system bugs
    analysis of pre/side conditions eg continuity
    smart documentation
    method selection
    document and reason about assumptions
  - Case study: bugs in Aldor object model

**ADT for sorting**

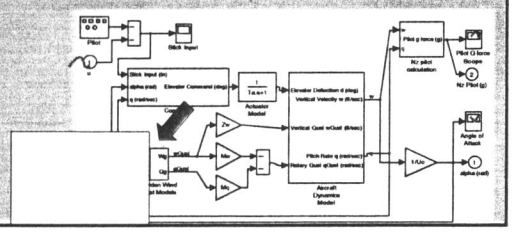
```
Sorting(E,C,<) : trait
includes
   StrictTotalOrder(<,E)
   ...
introduces
   permuted : C, C -> Bool
   sort     : C   -> C
   ordered  : C   -> Bool
   ...
asserts
   (∀ c, d : C ) c = sort(d) ==
   ordered (c) ∧ permuted(c,d));
```

**Quicksort in Aldor-FMlite**

```
qSort(L:List S): List S == {
++                  uses Sorting(S,
    List S, <);
++                  requires
++              StrictTotalOrder(<, S);
++                  modifies
    nothing;
++                  ensures
    result = sort(L);
...
(L < 2) => L
P       == first (L);
lo      == [e for e in L | e < P] ;
mid     == [e for e in L | e = P] ;
hi      == [e for e in L |
              ~ ((e < P) or (e = P))] ;
concat(qSort(lo), concat(mid,
   qSort(hi)));
...
```

What next

- **Dependable continuous mathematics** St Andrews/SRI/NASA/DERA
- Treat Simulink diagram as a specification
- Reason about model and environment for control system
  abstractions for continuous math
- Proof oriented techniques for Statecharts
- Code generation



- **Challenges**
- **Immediate applications**
  problem oriented libraries and methods for automation/re-use
  complex numbers, special functions, DEs,
  continuity, limits, solve, table look-up, side conditions
  Models assumptions and requirements
  Embedded systems and control
- **Longer term**
  We can do anything we want in CL
  How do we exploit this in mathematics, engineering, science?
  - representation
  - validation
  - communication

9 Erica Melis
*DFKI Saarbrücken, Germany*

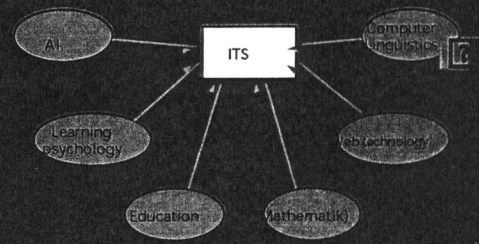
Course: Maths Learning Environments

Tutorial: The ACTIVEMATH System

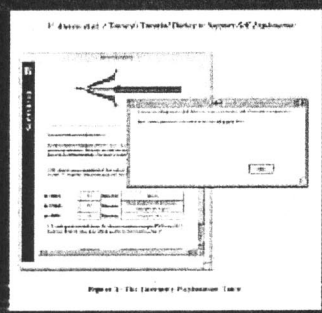
## Intelligent Learning Systems

**Erica Melis**
Competence Center for e-learning
**German Research Center for Artificial Intelligence Saarbrücken**

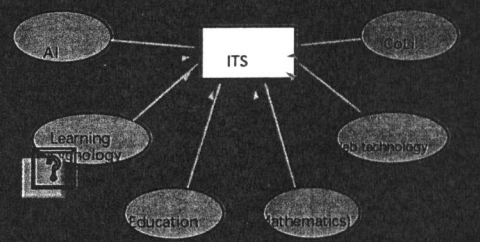
## Interdisciplinary Work




## Geometry Explanation Tutor, example *



## Interdisciplinary Work*

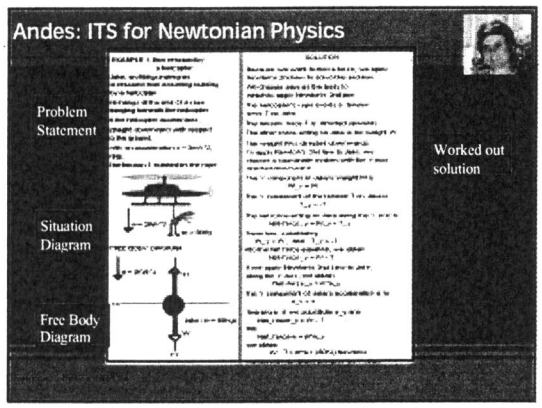


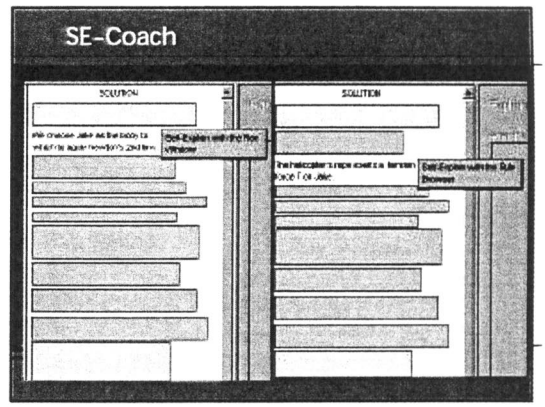

## Learning psychology...

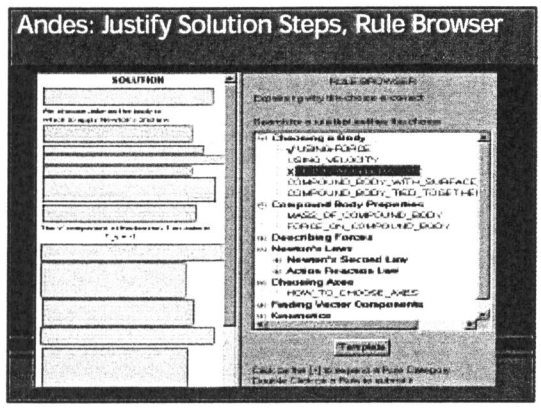
| Traditional Instruction<br>Transmission/teaching | Constructivist learning |
|---|---|
| passive learner<br>• external motivation<br>• coach according to expert solution<br>• no discussion<br>• isolated training<br>• judge from solutions only<br>• static learning goal | • Active learner, interaction<br>• internal motivation<br>• self-responsible<br>• personalized<br>• exploration, discovery<br>• authentic szenarios<br>• learning in context cooperative learning<br>• dynamic goals |

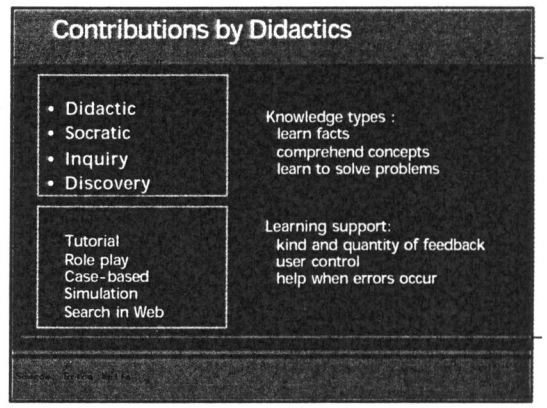
## Cognitive Psychology

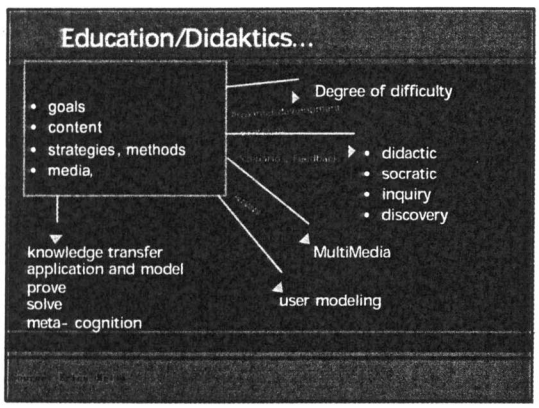
- Feedback
  - *ask, don't tell*
  - motivating: competent, self-guided, socially embedded
  - gender -spezific
- adaptive support
- presentation of solöutions
- Meta-cognition *

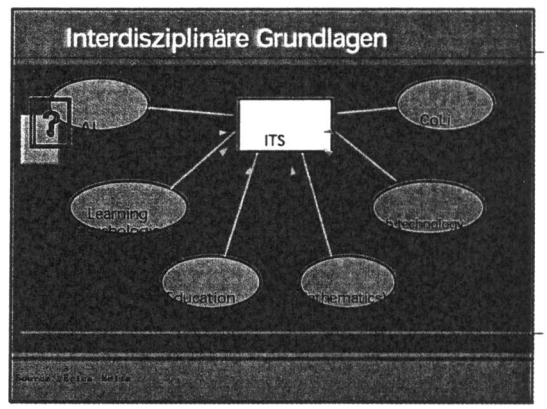

Andes: ITS for Newtonian Physics
Problem Statement
Situation Diagram
Free Body Diagram
Worked out solution


SE-Coach
SOLUTION
SOLUTION


Andes: Justify Solution Steps, Rule Browser
SOLUTION
RULE BROWSER
Describing Forces
Newton's Laws
Newton's Second Law
Action Reactions Law
Choosing Axes
Finding Vector Components
Kinematics
Templates


Contributions by Didactics
Didactic
Socratic
Inquiry
Discovery
Knowledge types :
learn facts
comprehend concepts
learn to solve problems
Tutorial
Role play
Case-based
Simulation
Search in Web
Learning support:
kind and quantity of feedback
user control
help when errors occur


Education/Didaktics...
goals
content
strategies, methods
media,
Degree of difficulty
didactic
socratic
inquiry
discovery
knowledge transfer
application and model
prove
solve
meta- cognition
MultiMedia
user modeling


Interdisziplinäre Grundlagen
AI
ITS
CoLi
Learning
Psychology
Education
Mathematics

## Artificial Intelligence*

- User modeling .
- Presentation planning
- adaptive user interfaces
- Problemsolving systems
- knowledge representation
- error diagnosis .
- agent-basiertes Feedback
- natural language processing

## ActiveMath Learning Environment

## Pedagogical Goals (first phase) **

- Personalization
  - content, feedback
  - presentation
  - notes
- Reading and active, exploratory learning
- realistic complexity
- dependencies
- some self-responsibility
- help by feedback
- make explicit some implikcit knowledge

## Technical Goals

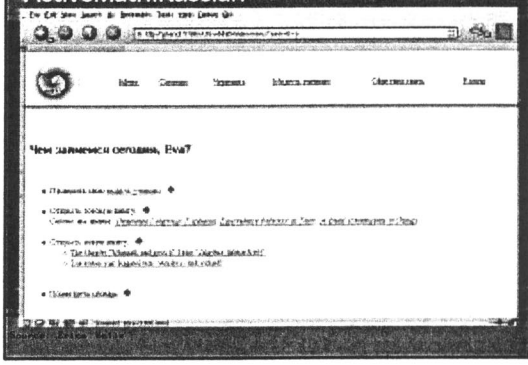
- re-use of content, standard repr .
- Platform independence
- several kinds of presentation
- separation of knowledge and functionality
- open distributed architecture
- modular design, reusable components
- standard XML-communication

## ActiveMath **

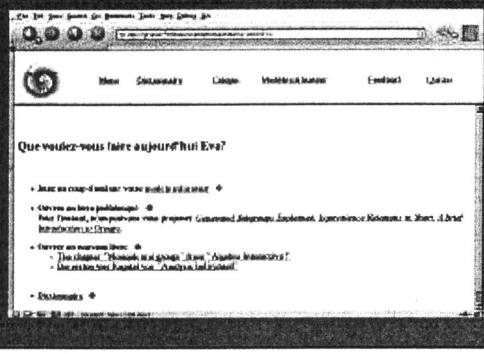
- personalized content asnd presentation
- internationalized (German, English, Russian,...)
- dynamic suggestions
- interactive exercises with service-systems
- web-based, distributed architecture
- semantic XML-representation

http://www.activemath.org

## ActiveMath:Russian

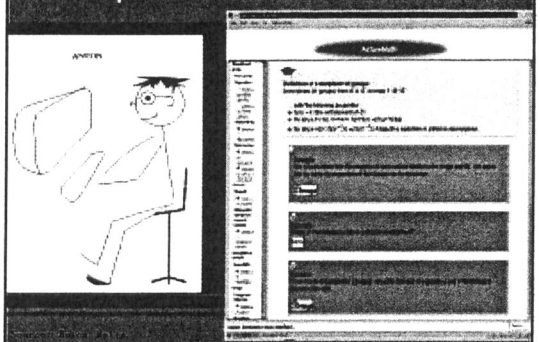


## ActiveMath:French

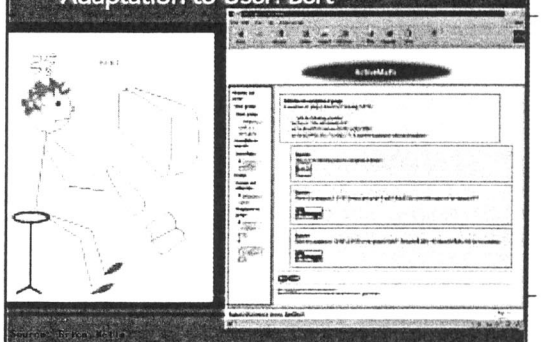



## ActiveMath


## Adaptation to User: Anton



## Adaptation to User: Bert



## Dynamic Generation of Presentation

- Choice of content , examples, and exercises
- assemblance of pages
- availability of external systems
- presentation-features

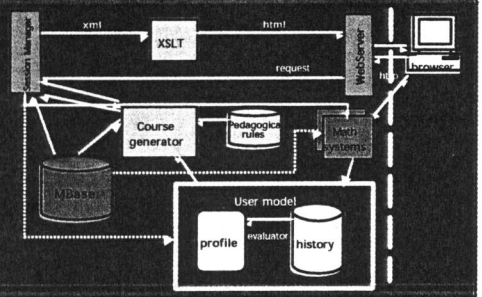
Dependent on user model and pedagogical rules:

- goals
- scenario
- preferences
- knowledge mastery...

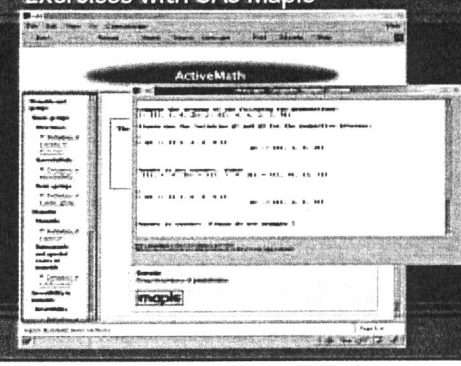
## Pedagogical Rules

| | | |
|---|---|---|
| IF field(user) = ?F | THEN | addEx(?C, field)=?F |
| IF learnGoal=appl AND uk(?C low) | THEN | addEx(?C,diff)=(1,1,2,3) |
| IF learnGoal=appl AND NOT uk(?C low) | THEN | addEx(?C, diff)=(3) |
| IF scenario(overview) | THEN | pattern=(concept,exm,ex) |
| IF scenario(detail) | THEN | pattern=(motiv,intro, concept,exm,ex) |
| IF nextbest AND navigation(crazy ?start) | THEN | addPointer(?start) |

## Architecture of ActiveMath




## Exercises with CAS Maple

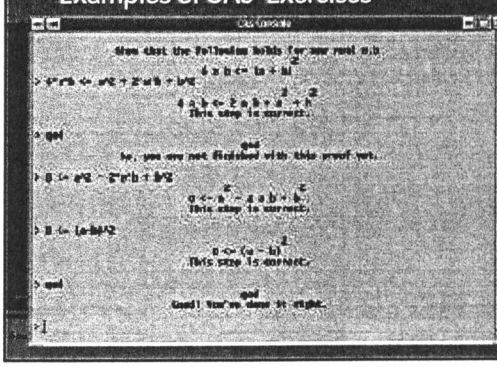



## Feedback: Classes of Test Conditions *

- test
- valid ? type?
- equivalent expressions?
- equal solution set?
- match?
- equal to pre-defined solution (if author insists)

- qed?

## Examples of CAS-Exercises

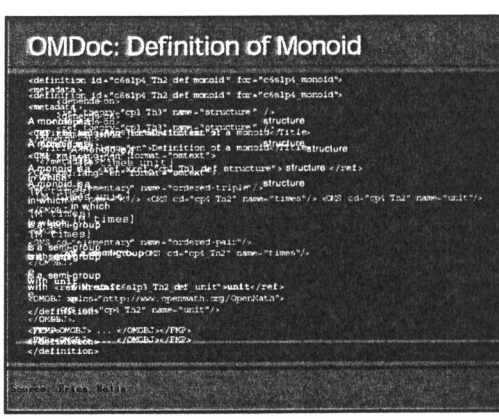


## Knowledge Representation in ActiveMath

- OMDoc XML-language:
  - structures
  - semantics (OpenMath, MathML)
- Ontology for Mathematics

- ActiveMath: extensions by
- didactical metadata
- relations
- several verbosities (book, slide, summary)
- etc.

## OMDoc: Definition of Monoid



## Research: ActiveMath+ for Learning

- Formalization of didactic strategies etc
- pro-constructivist content
- support of Metakognition, error discovery construction of examples and counter-examples
- tutorial dialogues
- *motivational state*
- emotions
- reactive suggestions
- elaborate user modeling
- usability
- Integration of statistics tools etc.
- Mixed-initiative proof planning
- empirical tests!!!!


## ActiveMath+ for (empirical) Research

- effect of didactic strategies
- effect of different problem presentations
- effect of help
- effect of meta-cognitive support
- dependence of learner features and suitable system parameters
- how to plan learning
- how to learn with CASs, proof planner
- effect of different sollution presentations
  - structure
  - meta-variables
  - method- and control knowledge
- Poor Man's Eye Tracker DFKeye



## Conclusion: Demo www.activemath.org

- ITS: interdisciplinary field with commercial relevance
  - o Distance learning
  - o Virtual universities
  - o Training on the job
  - o Military training
  - o Training for disabled
  - o use the Web as learning resource
- **ActiveMath:** *open source* project

10 Tobias Nipkow
*Technical University München, Germany*

**Course: The ISABELLE System**

# A Compact Overview of Isabelle/HOL

Tobias Nipkow
Institut für Informatik, TU München
http://www.in.tum.de/~nipkow/

This document is a very compact introduction to the proof assistant Isabelle/HOL and is based directly on the published tutorial [1] where full explanations and a wealth of additional material can be found.

While reading this document it is recommended to single-step through the corresponding theories using Isabelle/HOL to follow the proofs.

## 1 Functional Programming/Modelling

### 1.1 An Introductory Theory

```
theory FP0 = PreList:

datatype 'a list = Nil                          ("[]")
                 | Cons 'a "'a list"            (infixr "#" 65)

consts app :: "'a list ⇒ 'a list ⇒ 'a list"   (infixr "@" 65)
       rev :: "'a list ⇒ 'a list"

primrec
"[] @ ys       = ys"
"(x # xs) @ ys = x # (xs @ ys)"

primrec
"rev []        = []"
"rev (x # xs)  = (rev xs) @ (x # [])"

theorem rev_rev [simp]: "rev(rev xs) = xs"

end
```

**Exercise 1.1** Define a datatype of binary trees and a function `mirror` that mirrors a binary tree by swapping subtrees recursively. Prove `mirror(mirror t) = t`.

Define a function `flatten` that flattens a tree into a list by traversing it in infix order. Prove `flatten(mirror t) = rev(flatten t)`.



### 1.2 More Constructs

```
lemma "if xs = ys
       then rev xs = rev ys
       else rev xs ≠ rev ys"
by auto

lemma "case xs of
         []   ⇒ tl xs = xs
       | y#ys ⇒ tl xs ≠ xs"
apply(case_tac xs)
by auto
```

### 1.3 More Types

#### 1.3.1 Natural Numbers

```
consts sum :: "nat ⇒ nat"
primrec "sum 0 = 0"
        "sum (Suc n) = Suc n + sum n"

lemma "sum n + sum n = n*(Suc n)"
apply(induct_tac n)
apply(auto)
done
```

Some examples of linear arithmetic:

```
lemma "[ ¬ m < n; m < n+(1::int) ] ⟹ m = n"
by(auto)

lemma "min i (max j k) = max (min k i) (min i (j::nat))"
by(arith)
```

Not proved automatically because it involves multiplication:

```
lemma "n*n = n ⟹ n=0 ∨ n=(1::int)"
```

#### 1.3.2 Pairs

```
lemma "fst(x,y) = snd(z,x)"
by auto
```

### 1.4 Definitions

```
consts xor :: "bool ⇒ bool ⇒ bool"
defs xor_def: "xor x y ≡ x ∧ ¬y ∨ ¬x ∧ y"

constdefs nand :: "bool ⇒ bool ⇒ bool"
          "nand x y ≡ ¬(x ∧ y)"
```

```
lemma "¬ xor x x"
apply(unfold xor_def)
by auto
```

## 1.5 Simplification

### 1.5.1 Simplification Rules

```
lemma fst_conv[simp]: "fst(x,y) = x"
by auto
```

Setting and resetting the *simp* attribute:

```
declare fst_conv[simp]
declare fst_conv[simp del]
```

### 1.5.2 The Simplification Method

```
lemma "x*(y+1) = y*(x+1::nat)"
apply simp
```

### 1.5.3 Adding and Deleting Simplification Rules

```
lemma "∀x::nat. x*(y+z) = r"
apply (simp add: add_mult_distrib2)

lemma "rev(rev(xs @ [])) = xs"
apply (simp del: rev_rev_ident)
```

### 1.5.4 Rewriting with Definitions

```
lemma "xor A (¬A)"
apply(simp only: xor_def)
apply simp
done
```

### 1.5.5 Conditional Equations

A pre-proved simplification rule: $xs \neq [] \Longrightarrow hd\ xs\ \#\ tl\ xs = xs$

```
lemma "hd(xs @ [x]) # tl(xs @ [x]) = xs @ [x]"
by simp
```

### 1.5.6 Automatic Case Splits

```
lemma "∀xs. if xs = [] then A else B"
apply simp
```

Case-expressions are only split on demand.



### 1.5.7 Arithmetic

Only simple arithmetic:

```
lemma "[ ¬ m < n; m < n+(1::nat) ] ⟹ m = n"
by simp
```

Complex goals need `arith`-method.

## 1.6 Case Study: Compiling Expressions

### 1.6.1 Expressions

```
types 'v binop = "'v ⇒ 'v ⇒ 'v"

datatype ('a,'v)expr = Cex 'v
                     | Vex 'a
                     | Bex "'v binop"  "('a,'v)expr"  "('a,'v)expr"

consts value :: "('a,'v)expr ⇒ ('a ⇒ 'v) ⇒ 'v"
primrec
"value (Cex v) env = v"
"value (Vex a) env = env a"
"value (Bex f e1 e2) env = f (value e1 env) (value e2 env)"
```

### 1.6.2 The Stack Machine

```
datatype ('a,'v) instr = Const 'v
                       | Load 'a
                       | Apply "'v binop"

consts exec :: "('a,'v)instr list ⇒ ('a⇒'v) ⇒ 'v list ⇒ 'v list"
primrec
"exec [] s vs = vs"
"exec (i#is) s vs = (case i of
    Const v  ⇒ exec is s (v#vs)
  | Load a   ⇒ exec is s ((s a)#vs)
  | Apply f  ⇒ exec is s ((f (hd vs) (hd(tl vs)))#(tl(tl vs))))"
```

### 1.6.3 The Compiler

```
consts comp :: "('a,'v)expr ⇒ ('a,'v)instr list"
primrec
"comp (Cex v)       = [Const v]"
"comp (Vex a)       = [Load a]"
"comp (Bex f e1 e2) = (comp e2) @ (comp e1) @ [Apply f]"

theorem "exec (comp e) s [] = [value e s]"
```

## 1.7 Advanced Datatypes

### 1.7.1 Mutual Recursion

```
datatype 'a aexp = IF   "'a bexp" "'a aexp" "'a aexp"
                 | Sum  "'a aexp" "'a aexp"
                 | Var 'a
                 | Num nat
and      'a bexp = Less "'a aexp" "'a aexp"
                 | And  "'a bexp" "'a bexp"
                 | Neg  "'a bexp"


consts  evala :: "'a aexp ⇒ ('a ⇒ nat) ⇒ nat"
        evalb :: "'a bexp ⇒ ('a ⇒ nat) ⇒ bool"

primrec
  "evala (IF b a1 a2) env =
     (if evalb b env then evala a1 env else evala a2 env)"
  "evala (Sum a1 a2) env = evala a1 env + evala a2 env"
  "evala (Var v) env = env v"
  "evala (Num n) env = n"

  "evalb (Less a1 a2) env = (evala a1 env < evala a2 env)"
  "evalb (And b1 b2) env = (evalb b1 env ∧ evalb b2 env)"
  "evalb (Neg b) env = (¬ evalb b env)"

consts substa :: "('a ⇒ 'b aexp) ⇒ 'a aexp ⇒ 'b aexp"
       substb :: "('a ⇒ 'b aexp) ⇒ 'a bexp ⇒ 'b bexp"

primrec
  "substa s (IF b a1 a2) =
     IF (substb s b) (substa s a1) (substa s a2)"
  "substa s (Sum a1 a2) = Sum (substa s a1) (substa s a2)"
  "substa s (Var v) = s v"
  "substa s (Num n) = Num n"

  "substb s (Less a1 a2) = Less (substa s a1) (substa s a2)"
  "substb s (And b1 b2) = And (substb s b1) (substb s b2)"
  "substb s (Neg b) = Neg (substb s b)"

lemma substitution_lemma:
 "evala (substa s a) env = evala a (λx. evala (s x) env) ∧
  evalb (substb s b) env = evalb b (λx. evala (s x) env)"
apply(induct_tac a and b)
by simp_all
```

### 1.7.2 Nested Recursion

```
datatype tree = C "tree list"
```



Some trees:

```
term "C []"
term "C [C [C []], C []]"

consts
mirror :: "tree ⇒ tree"
mirrors:: "tree list ⇒ tree list"

primrec
  "mirror(C ts) = C(mirrors ts)"

  "mirrors [] = []"
  "mirrors (t # ts) = mirrors ts @ [mirror t]"

lemma "mirror(mirror t) = t ∧ mirrors(mirrors ts) = ts"
apply(induct_tac t and ts)
apply simp_all
```

**Exercise 1.2** Complete the above proof.

### 1.7.3 Datatypes Involving Functions

```
datatype ('a,'i)bigtree = Tip | Br 'a "'i ⇒ ('a,'i)bigtree"
```

A big tree:

```
term "Br 0 (λi. Br i (λn. Tip))"

consts map_bt :: "('a ⇒ 'b) ⇒ ('a,'i)bigtree ⇒ ('b,'i)bigtree"
primrec
"map_bt f Tip      = Tip"
"map_bt f (Br a F) = Br (f a) (λi. map_bt f (F i))"

lemma "map_bt (g o f) T = map_bt g (map_bt f T)"
apply(induct_tac T, rename_tac[2] F)
apply simp_all
done
```

This is *not* allowed:

```
datatype lambda = C "lambda => lambda"
```

**Exercise 1.3** Define a datatype of ordinals and the ordinal $\Gamma_0$.

## 1.8 Wellfounded Recursion

### 1.8.1 Examples

```
consts fib :: "nat ⇒ nat"
recdef fib "measure(λn. n)"
  "fib 0 = 0"
  "fib (Suc 0) = 1"
  "fib (Suc(Suc x)) = fib x + fib (Suc x)"

consts sep :: "'a × 'a list ⇒ 'a list"
recdef sep "measure (λ(a,xs). length xs)"
  "sep(a, [])     = []"
  "sep(a, [x])    = [x]"
  "sep(a, x#y#zs) = x # a # sep(a,y#zs)"

consts last :: "'a list ⇒ 'a"
recdef last "measure (λxs. length xs)"
  "last [x]      = x"
  "last (x#y#zs) = last (y#zs)"

consts sep1 :: "'a × 'a list ⇒ 'a list"
recdef sep1 "measure (λ(a,xs). length xs)"
  "sep1(a, x#y#zs) = x # a # sep1(a,y#zs)"
  "sep1(a, xs)     = xs"
```

This is what the rules for *sep1* are turned into:

```
sep1 (a, x # y # zs) = x # a # sep1 (a, y # zs)
sep1 (a, []) = []
sep1 (a, [w]) = [w]
```

Pattern matching is also useful for nonrecursive functions:

```
consts swap12 :: "'a list ⇒ 'a list"
recdef swap12 "{}"
  "swap12 (x#y#zs) = y#x#zs"
  "swap12 zs       = zs"
```

### 1.8.2 Beyond Measure

The lexicographic product of two relations:

```
consts ack :: "nat×nat ⇒ nat"
recdef ack "measure(λm. m) <*lex*> measure(λn. n)"
  "ack(0,n)         = Suc n"
  "ack(Suc m,0)     = ack(m, 1)"
  "ack(Suc m,Suc n) = ack(m,ack(Suc m,n))"
```



### 1.8.3 Induction

Every recursive definition provides an induction theorem, for example `sep.induct`:

```
⟦⋀a. P a []; ⋀a x. P a [x];
   ⋀a x y zs. P a (y # zs) ⟹ P a (x # y # zs)⟧
⟹ P u v
```

```
lemma "map f (sep(x,xs)) = sep(f x, map f xs)"
apply(induct_tac x xs rule: sep.induct)
apply simp_all
done

lemma ack_incr2: "n < ack(m,n)"
apply(induct_tac m n rule: ack.induct)
apply simp_all
done
```

### 1.8.4 Recursion Over Nested Datatypes

```
datatype tree = C "tree list"

lemma termi_lem: "t ∈ set ts ⟶ size t < Suc(tree_list_size ts)"
by(induct_tac ts, auto)

consts mirror :: "tree ⇒ tree"
recdef mirror "measure size"
 "mirror(C ts) = C(rev(map mirror ts))"
(hints recdef_simp: termi_lem)

lemma "mirror(mirror t) = t"
apply(induct_tac t rule: mirror.induct)
apply(simp add: rev_map sym[OF map_compose] cong: map_cong)
done
```

Figure out how that proof worked!

**Exercise 1.4** Define a function for merging two ordered lists (of natural numbers) and show that if the two input lists are ordered, so is the output.

# 2 The Rules of the Game

## 2.1 Introduction Rules

Introduction rules for propositional logic:

```
impI    (P ⟹ Q) ⟹ P ⟶ Q
conjI   [P; Q] ⟹ P ∧ Q
disjI1  P ⟹ P ∨ Q
disjI2  Q ⟹ P ∨ Q
iffI    [P ⟹ Q; Q ⟹ P] ⟹ P = Q
```

```
lemma "A ⟹ B ⟶ A ∧ (B ∧ A)"
apply(rule impI)
apply(rule conjI)
 apply assumption
apply(rule conjI)
 apply assumption
apply assumption
done
```

## 2.2 Elimination Rules

Elimination rules for propositional logic:

```
impE   [P ⟶ Q; P; Q ⟹ R] ⟹ R
mp     [P ⟶ Q; P] ⟹ Q
conjE  [P ∧ Q; [P; Q] ⟹ R] ⟹ R
disjE  [P ∨ Q; P ⟹ R; Q ⟹ R] ⟹ R
```

```
lemma disj_swap: "P ∨ Q ⟹ Q ∨ P"
apply (erule disjE)
 apply (rule disjI2)
 apply assumption
apply (rule disjI1)
apply assumption
done
```

## 2.3 Negation

```
notI    (P ⟹ False) ⟹ ¬ P
notE    [¬ P; P] ⟹ R
ccontr  (¬ P ⟹ False) ⟹ P
```

```
lemma "¬¬P ⟹ P"
apply (rule ccontr)
apply (erule notE)
apply assumption
done
```

A simple exercise:

```
lemma "¬P ∨ ¬Q ⟹ ¬(P ∧ Q)"
```



## 2.4 Quantifiers

Quantifier rules:

```
allI   (⋀x. P x) ⟹ ∀x. P x
exI    P x ⟹ ∃x. P x
allE   [∀x. P x; P x ⟹ R] ⟹ R
exE    [∃x. P x; ⋀x. P x ⟹ Q] ⟹ Q
spec   ∀x. P x ⟹ P x
```

Another classic exercise:

```
lemma "∃x.∀y. P x y ⟹ ∀y.∃x. P x y"
```

## 2.5 Instantiation

```
lemma "∃xs. xs @ xs = xs"
apply(rule_tac x = "[]" in exI)
by simp
```

```
lemma "∀xs. f(xs @ xs) = xs ⟹ f [] = []"
apply(erule_tac x = "[]" in allE)
by simp
```

## 2.6 Automation

```
lemma "(∀x. honest(x) ∧ industrious(x) ⟶ healthy(x)) ∧
      ¬ (∃x. grocer(x) ∧ healthy(x)) ∧
      (∀x. industrious(x) ∧ grocer(x) ⟶ honest(x)) ∧
      (∀x. cyclist(x) ⟶ industrious(x)) ∧
      (∀x. ¬healthy(x) ∧ cyclist(x) ⟶ ¬honest(x))
      ⟶ (∀x. grocer(x) ⟶ ¬cyclist(x))"
by blast
```

```
lemma "(⋃i∈I. A(i)) ∩ (⋃j∈J. B(j)) =
      (⋃i∈I. ⋃j∈J. A(i) ∩ B(j))"
by fast
```

```
lemma "∃x.∀y. P x y ⟹ ∀y.∃x. P x y"
apply(clarify)
by blast
```

## 2.7 Forward Proof

Instantiation of unknowns (in left-to-right order):

```
append_self_conv                (?xs @ ?ys = ?xs) = (?ys = [])
append_self_conv [ of _ "[]" ]  (?xs @ [] = ?xs) = ([] = [])
```

Applying one theorem to another by unifying the premise of one theorem with the conclusion of another:

```
sym                              ?s = ?t ⟹ ?t = ?s
sym [ OF append_self_conv ]      (?ys1 = []) = (?xs1 @ ?ys1 = ?xs1)
append_self_conv [ THEN sym ]    (?ys1 = []) = (?xs1 @ ?ys1 = ?xs1)
```

### 2.8 Further Useful Methods

```
lemma "n ≤ 1 ∧ n ≠ 0 ⟹ n^n = n"
apply(subgoal_tac "n=1")
 apply simp
apply arith
done
```

And a cute example:

```
lemma "[ 2 ∈ Q; sqrt 2 ∉ Q;
        ∀x y z. sqrt x * sqrt x = x ∧
                x ^ 2 = x * x ∧
                (x ^ y) ^ z = x ^ (y*z)
      ] ⟹ ∃a b. a ∉ Q ∧ b ∉ Q ∧ a ^ b ∈ Q"
```

## 3 Sets, Functions and Relations

### 3.1 Set Notation

```
A ∪ B      A ∩ B      A - B
a ∈ A      b ∉ A
{a, b}     {x. P x}
⋃M     ⋃a ∈ A. F a
```

### 3.2 Some Functions

```
id ≡ λx. x
f ∘ g ≡ λx. f (g x)
f ‘ A ≡ {y. ∃x∈A. y = f x}
f -‘ B ≡ {x. f x ∈ B}
```

### 3.3 Some Relations

```
Id ≡ {p. ∃x. p = (x, x)}
r⁻¹ ≡ {(y, x). (x, y) ∈ r}
r `` s ≡ {y. ∃x∈s. (x, y) ∈ r}
(a, a) ∈ r*
[(a, b) ∈ r*; (b, c) ∈ r] ⟹ (a, c) ∈ r*
r⁺ ≡ r O r*
```



### 3.4 Wellfoundedness

```
wf r ≡ ∀P. (∀x. (∀y. (y, x) ∈ r ⟶ P y) ⟶ P x) ⟶ (∀x. P x)
wf r = (¬ (∃f. ∀i. (f (Suc i), f i) ∈ r))
```

### 3.5 Fixed Point Operators

```
lfp f ≡ ⋂{u. f u ⊆ u}
mono f ⟹ lfp f = f (lfp f)
[a ∈ lfp f; mono f; ⋀x. x ∈ f (lfp f ∩ Collect P) ⟹ P x] ⟹ P a
```

### 3.6 Case Study: Verified Model Checking

```
typedecl state

consts M :: "(state × state)set"

typedecl atom

consts L :: "state ⇒ atom set"

datatype formula = Atom atom
                 | Neg formula
                 | And formula formula
                 | AX formula
                 | EF formula

consts valid :: "state ⇒ formula ⇒ bool"   ("(_ ⊨ _)" [80,80] 80)

primrec
"s ⊨ Atom a  = (a ∈ L s)"
"s ⊨ Neg f   = (¬(s ⊨ f))"
"s ⊨ And f g = (s ⊨ f ∧ s ⊨ g)"
"s ⊨ AX f    = (∀t. (s,t) ∈ M ⟶ t ⊨ f)"
"s ⊨ EF f    = (∃t. (s,t) ∈ M* ∧ t ⊨ f)"

consts mc :: "formula ⇒ state set"
primrec
"mc(Atom a)  = {s. a ∈ L s}"
"mc(Neg f)   = -mc f"
"mc(And f g) = mc f ∩ mc g"
"mc(AX f)    = {s. ∀t. (s,t) ∈ M  ⟶ t ∈ mc f}"
"mc(EF f)    = lfp(λT. mc f ∪ (M⁻¹ `` T))"

lemma mono_ef: "mono(λT. A ∪ (M⁻¹ `` T))"
```

```
apply(rule monoI)
apply blast
done

lemma EF_lemma:
  "lfp(λT. A ∪ (M⁻¹ `` T)) = {s. ∃t. (s,t) ∈ M* ∧ t ∈ A}"
apply(rule equalityI)
 thm lfp_lowerbound
 apply(rule lfp_lowerbound)
 apply(blast intro: rtrancl_trans)
apply(rule subsetI)
apply clarsimp
apply(erule converse_rtrancl_induct)
thm lfp_unfold[OF mono_ef]
 apply(subst lfp_unfold[OF mono_ef])
 apply(blast)
apply(subst lfp_unfold[OF mono_ef])
apply(blast)
done

theorem "mc f = {s. s ⊨ f}"
apply(induct_tac f)
apply(auto simp add: EF_lemma)
done
```

**Exercise 3.1** *AX* has a dual operator *EN*[1] ("there exists a next state such that") with the intended semantics

```
s ⊨ EN f = (∃t. (s, t) ∈ M ∧ t ⊨ f)
```

Fortunately, *EN f* can already be expressed as a PDL formula. How?

Show that the semantics for *EF* satisfies the following recursion equation:

```
s ⊨ EF f = (s ⊨ f ∨ s ⊨ EN (EF f))
```

# 4 Inductive Definitions

## 4.1 Even Numbers

### 4.1.1 The Definition

```
consts even :: "nat set"
inductive even
intros
zero[intro!]: "0 ∈ even"
step[intro!]: "n ∈ even ⟹ Suc(Suc n) ∈ even"

lemma [simp,intro!]: "2 dvd n ⟹ 2 dvd Suc(Suc n)"
apply (unfold dvd_def)
apply clarify
apply (rule_tac x = "Suc k" in exI, simp)
done
```

### 4.1.2 Rule Induction

Rule induction for set *even*, *even.induct*:

```
[xa ∈ even; P 0; ⋀n. [n ∈ even; P n] ⟹ P (Suc (Suc n))] ⟹ P xa


lemma even_imp_dvd: "n ∈ even ⟹ 2 dvd n"
apply (erule even.induct)
apply simp_all
done
```

### 4.1.3 Rule Inversion

```
inductive_cases Suc_Suc_case [elim!]: "Suc(Suc n) ∈ even"

[Suc (Suc n) ∈ even; n ∈ even ⟹ P] ⟹ P


lemma "Suc(Suc n) ∈ even ⟹ n ∈ even"
by blast
```

## 4.2 Mutually Inductive Definitions

```
consts evn :: "nat set"
       odd :: "nat set"

inductive evn odd
intros
zero: "0 ∈ evn"
evnI: "n ∈ odd ⟹ Suc n ∈ evn"
oddI: "n ∈ evn ⟹ Suc n ∈ odd"

lemma "(m ∈ evn ⟶ 2 dvd m) ∧ (n ∈ odd ⟶ 2 dvd (Suc n))"
apply(rule evn_odd.induct)
by simp_all
```

[1] We cannot use the customary *EX* as that is the ASCII-equivalent of ∃





-951-

### 4.3 The Reflexive Transitive Closure

```
consts rtc :: "('a × 'a)set ⇒ ('a × 'a)set"   ("_*" [1000] 999)
inductive "r*"
intros
refl[iff]:  "(x,x) ∈ r*"
step:       "[ (x,y) ∈ r; (y,z) ∈ r* ] ⟹ (x,z) ∈ r*"

lemma [intro]: "(x,y) : r ⟹ (x,y) ∈ r*"
by(blast intro: rtc.step)

lemma rtc_trans: "[ (x,y) ∈ r*; (y,z) ∈ r* ] ⟹ (x,z) ∈ r*"
apply(erule rtc.induct)
oops

lemma rtc_trans[rule_format]:
  "(x,y) ∈ r* ⟹ (y,z) ∈ r* ⟶ (x,z) ∈ r*"
apply(erule rtc.induct)
 apply(blast)
apply(blast intro: rtc.step)
done
```

**Exercise 4.1** Show that the converse of `rtc.step` also holds:

```
[(x, y) ∈ r*; (y, z) ∈ r] ⟹ (x, z) ∈ r*
```

### 4.4 The accessible part of a relation

```
consts  acc :: "('a × 'a) set ⇒ 'a set"
inductive "acc r"
intros
  "(∀y. (y,x) ∈ r ⟶ y ∈ acc r) ⟹ x ∈ acc r"

lemma "wf{(x,y). (x,y) ∈ r ∧ y ∈ acc r}"
thm wfI
apply(rule_tac A = "acc r" in wfI)
 apply (blast elim: acc.elims)
apply(simp(no_asm_use))
thm acc.induct
apply(erule acc.induct)
by blast

consts  accs :: "('a × 'a) set ⇒ 'a set"
inductive "accs r"
intros
 "r''{x} ∈ Pow(accs r) ⟹ x ∈ accs r"
```



```
monos Pow_mono
```

## References

[1] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2002. http://www.in.tum.de/~nipkow/LNCS2283/.

# A Compact Overview of Isabelle/HOL

Tobias Nipkow
Institut für Informatik, TU München
http://www.in.tum.de/~nipkow/

This document is a very compact introduction to the proof assistant Isabelle/HOL and is based directly on the published tutorial [1] where full explanations and a wealth of additional material can be found.

While reading this document it is recommended to single-step through the corresponding theories using Isabelle/HOL to follow the proofs.

## 1 Functional Programming/Modelling

### 1.1 An Introductory Theory

```
theory FP0 = PreList:

datatype 'a list = Nil                          ("[]")
                 | Cons 'a "'a list"            (infixr "#" 65)

consts app :: "'a list ⇒ 'a list ⇒ 'a list"   (infixr "@" 65)
       rev :: "'a list ⇒ 'a list"

primrec
"[] @ ys       = ys"
"(x # xs) @ ys = x # (xs @ ys)"

primrec
"rev []        = []"
"rev (x # xs)  = (rev xs) @ (x # [])"

theorem rev_rev [simp]: "rev(rev xs) = xs"

end
```

**Exercise 1.1** Define a datatype of binary trees and a function `mirror` that mirrors a binary tree by swapping subtrees recursively. Prove `mirror(mirror t) = t`.

Define a function `flatten` that flattens a tree into a list by traversing it in infix order. Prove `flatten(mirror t) = rev(flatten t)`.



### 1.2 More Constructs

```
lemma "if xs = ys
       then rev xs = rev ys
       else rev xs ≠ rev ys"
by auto

lemma "case xs of
         []   ⇒ tl xs = xs
       | y#ys ⇒ tl xs ≠ xs"
apply(case_tac xs)
by auto
```

### 1.3 More Types

#### 1.3.1 Natural Numbers

```
consts sum :: "nat ⇒ nat"
primrec "sum 0 = 0"
        "sum (Suc n) = Suc n + sum n"

lemma "sum n + sum n = n*(Suc n)"
apply(induct_tac n)
apply(auto)
done
```

Some examples of linear arithmetic:

```
lemma "[ ¬ m < n; m < n+(1::int) ] ⟹ m = n"
by(auto)

lemma "min i (max j k) = max (min i j) (min i (k::nat))"
by(arith)
```

Not proved automatically because it involves multiplication:

```
lemma "n*n = n ⟹ n=0 ∨ n=(1::int)"
```

#### 1.3.2 Pairs

```
lemma "fst(x,y) = snd(z,x)"
by auto
```

### 1.4 Definitions

```
consts xor :: "bool ⇒ bool ⇒ bool"
defs xor_def: "xor x y ≡ x ∧ ¬y ∨ ¬x ∧ y"

constdefs nand :: "bool ⇒ bool ⇒ bool"
          "nand x y ≡ ¬(x ∧ y)"
```

```
lemma "¬ xor x x"
apply(unfold xor_def)
by auto
```

## 1.5 Simplification

### 1.5.1 Simplification Rules

```
lemma fst_conv[simp]: "fst(x,y) = x"
by auto
```

Setting and resetting the *simp* attribute:

```
declare fst_conv[simp]
declare fst_conv[simp del]
```

### 1.5.2 The Simplification Method

```
lemma "x*(y+1) = y*(x+1::nat)"
apply simp
```

### 1.5.3 Adding and Deleting Simplification Rules

```
lemma "∀x::nat. x*(y+z) = r"
apply (simp add: add_mult_distrib2)

lemma "rev(rev(xs @ [])) = xs"
apply (simp del: rev_rev_ident)
```

### 1.5.4 Rewriting with Definitions

```
lemma "xor A (¬A)"
apply(simp only: xor_def)
apply simp
done
```

### 1.5.5 Conditional Equations

A pre-proved simplification rule: $xs \neq [] \Longrightarrow hd\ xs\ \#\ tl\ xs = xs$

```
lemma "hd(xs @ [x]) # tl(xs @ [x]) = xs @ [x]"
by simp
```

### 1.5.6 Automatic Case Splits

```
lemma "∀xs. if xs = [] then A else B"
apply simp
```

Case-expressions are only split on demand.



### 1.5.7 Arithmetic

Only simple arithmetic:

```
lemma "[ ¬ m < n; m < n+(1::nat) ] ⟹ m = n"
by simp
```

Complex goals need `arith`-method.

## 1.6 Case Study: Compiling Expressions

### 1.6.1 Expressions

```
types 'v binop = "'v ⇒ 'v ⇒ 'v"

datatype ('a,'v)expr = Cex 'v
                     | Vex 'a
                     | Bex "'v binop"  "('a,'v)expr"  "('a,'v)expr"

consts value :: "('a,'v)expr ⇒ ('a ⇒ 'v) ⇒ 'v"
primrec
"value (Cex v) env = v"
"value (Vex a) env = env a"
"value (Bex f e1 e2) env = f (value e1 env) (value e2 env)"
```

### 1.6.2 The Stack Machine

```
datatype ('a,'v) instr = Const 'v
                       | Load 'a
                       | Apply "'v binop"

consts exec :: "('a,'v)instr list ⇒ ('a⇒'v) ⇒ 'v list ⇒ 'v list"
primrec
"exec [] s vs = vs"
"exec (i#is) s vs = (case i of
    Const v  ⇒ exec is s (v#vs)
  | Load a   ⇒ exec is s ((s a)#vs)
  | Apply f  ⇒ exec is s ((f (hd vs) (hd(tl vs)))#(tl(tl vs))))"
```

### 1.6.3 The Compiler

```
consts comp :: "('a,'v)expr ⇒ ('a,'v)instr list"
primrec
"comp (Cex v)       = [Const v]"
"comp (Vex a)       = [Load a]"
"comp (Bex f e1 e2) = (comp e2) @ (comp e1) @ [Apply f]"

theorem "exec (comp e) s [] = [value e s]"
```

## 1.7 Advanced Datatypes

### 1.7.1 Mutual Recursion

```
datatype 'a aexp = IF   "'a bexp" "'a aexp" "'a aexp"
                 | Sum  "'a aexp" "'a aexp"
                 | Var 'a
                 | Num nat
and      'a bexp = Less "'a aexp" "'a aexp"
                 | And  "'a bexp" "'a bexp"
                 | Neg  "'a bexp"


consts  evala :: "'a aexp ⇒ ('a ⇒ nat) ⇒ nat"
        evalb :: "'a bexp ⇒ ('a ⇒ nat) ⇒ bool"

primrec
  "evala (IF b a1 a2) env =
     (if evalb b env then evala a1 env else evala a2 env)"
  "evala (Sum a1 a2) env = evala a1 env + evala a2 env"
  "evala (Var v) env = env v"
  "evala (Num n) env = n"

  "evalb (Less a1 a2) env = (evala a1 env < evala a2 env)"
  "evalb (And b1 b2) env = (evalb b1 env ∧ evalb b2 env)"
  "evalb (Neg b) env = (¬ evalb b env)"

consts substa :: "('a ⇒ 'b aexp) ⇒ 'a aexp ⇒ 'b aexp"
       substb :: "('a ⇒ 'b aexp) ⇒ 'a bexp ⇒ 'b bexp"

primrec
  "substa s (IF b a1 a2) =
     IF (substb s b) (substa s a1) (substa s a2)"
  "substa s (Sum a1 a2) = Sum (substa s a1) (substa s a2)"
  "substa s (Var v) = s v"
  "substa s (Num n) = Num n"

  "substb s (Less a1 a2) = Less (substa s a1) (substa s a2)"
  "substb s (And b1 b2) = And (substb s b1) (substb s b2)"
  "substb s (Neg b) = Neg (substb s b)"

lemma substitution_lemma:
 "evala (substa s a) env = evala a (λx. evala (s x) env) ∧
  evalb (substb s b) env = evalb b (λx. evala (s x) env)"
apply(induct_tac a and b)
by simp_all
```

### 1.7.2 Nested Recursion

```
datatype tree = C "tree list"
```



Some trees:

```
term "C []"
term "C [C [C []], C []]"

consts
mirror :: "tree ⇒ tree"
mirrors:: "tree list ⇒ tree list"

primrec
  "mirror(C ts) = C(mirrors ts)"

  "mirrors [] = []"
  "mirrors (t # ts) = mirrors ts @ [mirror t]"

lemma "mirror(mirror t) = t ∧ mirrors(mirrors ts) = ts"
apply(induct_tac t and ts)
apply simp_all
```

**Exercise 1.2** Complete the above proof.

### 1.7.3 Datatypes Involving Functions

```
datatype ('a,'i)bigtree = Tip | Br 'a "'i ⇒ ('a,'i)bigtree"
```

A big tree:

```
term "Br 0 (λi. Br i (λn. Tip))"

consts map_bt :: "('a ⇒ 'b) ⇒ ('a,'i)bigtree ⇒ ('b,'i)bigtree"
primrec
"map_bt f Tip      = Tip"
"map_bt f (Br a F) = Br (f a) (λi. map_bt f (F i))"

lemma "map_bt (g o f) T = map_bt g (map_bt f T)"
apply(induct_tac T, rename_tac[2] F)
apply simp_all
done
```

This is *not* allowed:

```
datatype lambda = C "lambda => lambda"
```

**Exercise 1.3** Define a datatype of ordinals and the ordinal $\Gamma_0$.

## 1.8 Wellfounded Recursion

### 1.8.1 Examples

```
consts fib :: "nat ⇒ nat"
recdef fib "measure(λn. n)"
  "fib 0 = 0"
  "fib (Suc 0) = 1"
  "fib (Suc(Suc x)) = fib x + fib (Suc x)"

consts sep :: "'a × 'a list ⇒ 'a list"
recdef sep "measure (λ(a,xs). length xs)"
  "sep(a, [])      = []"
  "sep(a, [x])     = [x]"
  "sep(a, x#y#zs) = x # a # sep(a,y#zs)"

consts last :: "'a list ⇒ 'a"
recdef last "measure (λxs. length xs)"
  "last [x]      = x"
  "last (x#y#zs) = last (y#zs)"

consts sep1 :: "'a × 'a list ⇒ 'a list"
recdef sep1 "measure (λ(a,xs). length xs)"
  "sep1(a, x#y#zs) = x # a # sep1(a,y#zs)"
  "sep1(a, xs)     = xs"
```

This is what the rules for `sep1` are turned into:

```
sep1 (a, x # y # zs) = x # a # sep1 (a, y # zs)
sep1 (a, []) = []
sep1 (a, [w]) = [w]
```

Pattern matching is also useful for nonrecursive functions:

```
consts swap12 :: "'a list ⇒ 'a list"
recdef swap12 "{}"
  "swap12 (x#y#zs) = y#x#zs"
  "swap12 zs       = zs"
```

### 1.8.2 Beyond Measure

The lexicographic product of two relations:

```
consts ack :: "nat×nat ⇒ nat"
recdef ack "measure(λm. m) <*lex*> measure(λn. n)"
  "ack(0,n)          = Suc n"
  "ack(Suc m,0)      = ack(m, 1)"
  "ack(Suc m,Suc n) = ack(m,ack(Suc m,n))"
```



### 1.8.3 Induction

Every recursive definition provides an induction theorem, for example `sep.induct`:

```
⟦⋀a. P a []; ⋀a x. P a [x];
   ⋀a x y zs. P a (y # zs) ⟹ P a (x # y # zs)⟧
⟹ P u v
```

```
lemma "map f (sep(x,xs)) = sep(f x, map f xs)"
apply(induct_tac x xs rule: sep.induct)
apply simp_all
done

lemma ack_incr2: "n < ack(m,n)"
apply(induct_tac m n rule: ack.induct)
apply simp_all
done
```

### 1.8.4 Recursion Over Nested Datatypes

```
datatype tree = C "tree list"

lemma termi_lem: "t ∈ set ts ⟶ size t < Suc(tree_list_size ts)"
by(induct_tac ts, auto)

consts mirror :: "tree ⇒ tree"
recdef mirror "measure size"
 "mirror(C ts) = C(rev(map mirror ts))"
(hints recdef_simp: termi_lem)

lemma "mirror(mirror t) = t"
apply(induct_tac t rule: mirror.induct)
apply(simp add: rev_map sym[OF map_compose] cong: map_cong)
done
```

Figure out how that proof worked!

**Exercise 1.4** Define a function for merging two ordered lists (of natural numbers) and show that if the two input lists are ordered, so is the output.

# 2 The Rules of the Game

## 2.1 Introduction Rules

Introduction rules for propositional logic:

```
impI    (P ⟹ Q) ⟹ P ⟶ Q
conjI   [P; Q] ⟹ P ∧ Q
disjI1  P ⟹ P ∨ Q
disjI2  Q ⟹ P ∨ Q
iffI    [P ⟹ Q; Q ⟹ P] ⟹ P = Q
```

```
lemma "A ⟹ B ⟶ A ∧ (B ∧ A)"
apply(rule impI)
apply(rule conjI)
 apply assumption
apply(rule conjI)
 apply assumption
apply assumption
done
```

## 2.2 Elimination Rules

Elimination rules for propositional logic:

```
impE    [P ⟶ Q; P; Q ⟹ R] ⟹ R
mp      [P ⟶ Q; P] ⟹ Q
conjE   [P ∧ Q; [P; Q] ⟹ R] ⟹ R
disjE   [P ∨ Q; P ⟹ R; Q ⟹ R] ⟹ R
```

```
lemma disj_swap: "P ∨ Q ⟹ Q ∨ P"
apply (erule disjE)
 apply (rule disjI2)
 apply assumption
apply (rule disjI1)
apply assumption
done
```

## 2.3 Negation

```
notI    (P ⟹ False) ⟹ ¬ P
notE    [¬ P; P] ⟹ R
ccontr  (¬ P ⟹ False) ⟹ P
```

```
lemma "¬¬P ⟹ P"
apply (rule ccontr)
apply (erule notE)
apply assumption
done
```

A simple exercise:

```
lemma "¬P ∨ ¬Q ⟹ ¬(P ∧ Q)"
```



## 2.4 Quantifiers

Quantifier rules:

```
allI   (⋀x. P x) ⟹ ∀x. P x
exI    P x ⟹ ∃x. P x
allE   [∀x. P x; P x ⟹ R] ⟹ R
exE    [∃x. P x; ⋀x. P x ⟹ Q] ⟹ Q
spec   ∀x. P x ⟹ P x
```

Another classic exercise:

```
lemma "∃x.∀y. P x y ⟹ ∀y.∃x. P x y"
```

## 2.5 Instantiation

```
lemma "∃xs. xs @ xs = xs"
apply(rule_tac x = "[]" in exI)
by simp
```

```
lemma "∀xs. f(xs @ xs) = xs ⟹ f [] = []"
apply(erule_tac x = "[]" in allE)
by simp
```

## 2.6 Automation

```
lemma "(∀x. honest(x) ∧ industrious(x) ⟶ healthy(x)) ∧
      ¬ (∃x. grocer(x) ∧ healthy(x)) ∧
      (∀x. industrious(x) ∧ grocer(x) ⟶ honest(x)) ∧
      (∀x. cyclist(x) ⟶ industrious(x)) ∧
      (∀x. ¬healthy(x) ∧ cyclist(x) ⟶ ¬honest(x))
      ⟶ (∀x. grocer(x) ⟶ ¬cyclist(x))"
by blast
```

```
lemma "(⋃i∈I. A(i)) ∩ (⋃j∈J. B(j)) =
      (⋃i∈I. ⋃j∈J. A(i) ∩ B(j))"
by fast
```

```
lemma "∃x.∀y. P x y ⟹ ∀y.∃x. P x y"
apply(clarify)
by blast
```

## 2.7 Forward Proof

Instantiation of unknowns (in left-to-right order):

```
append_self_conv                (?xs @ ?ys = ?xs) = (?ys = [])
append_self_conv [ of _ "[]" ]  (?xs @ [] = ?xs) = ([] = [])
```

Applying one theorem to another by unifying the premise of one theorem with the conclusion of another:

```
sym                              ?s = ?t ⟹ ?t = ?s
sym [ OF append_self_conv ]      (?ys1 = []) = (?xs1 @ ?ys1 = ?xs1)
append_self_conv [ THEN sym ]    (?ys1 = []) = (?xs1 @ ?ys1 = ?xs1)
```

### 2.8 Further Useful Methods

```
lemma "n ≤ 1 ∧ n ≠ 0 ⟹ n^n = n"
apply(subgoal_tac "n=1")
 apply simp
apply arith
done
```

And a cute example:

```
lemma "[ 2 ∈ Q; sqrt 2 ∉ Q;
        ∀x y z. sqrt x * sqrt x = x ∧
                x ^ 2 = x * x ∧
                (x ^ y) ^ z = x ^ (y*z)
      ] ⟹ ∃a b. a ∉ Q ∧ b ∉ Q ∧ a ^ b ∈ Q"
```

## 3 Sets, Functions and Relations

### 3.1 Set Notation

```
A ∪ B      A ∩ B      A - B
a ∈ A      b ∉ A
{a, b}     {x. P x}
⋃M     ⋃a ∈ A. F a
```

### 3.2 Some Functions

```
id ≡ λx. x
f ∘ g ≡ λx. f (g x)
f ` A ≡ {y. ∃x∈A. y = f x}
f -` B ≡ {x. f x ∈ B}
```

### 3.3 Some Relations

```
Id ≡ {p. ∃x. p = (x, x)}
r⁻¹ ≡ {(y, x). (x, y) ∈ r}
r `` s ≡ {y. ∃x∈s. (x, y) ∈ r}
(a, a) ∈ r*
[(a, b) ∈ r*; (b, c) ∈ r] ⟹ (a, c) ∈ r*
r⁺ ≡ r O r*
```



### 3.4 Wellfoundedness

```
wf r ≡ ∀P. (∀x. (∀y. (y, x) ∈ r ⟶ P y) ⟶ P x) ⟶ (∀x. P x)
wf r = (¬ (∃f. ∀i. (f (Suc i), f i) ∈ r))
```

### 3.5 Fixed Point Operators

```
lfp f ≡ ⋂{u. f u ⊆ u}
mono f ⟹ lfp f = f (lfp f)
[a ∈ lfp f; mono f; ⋀x. x ∈ f (lfp f ∩ Collect P) ⟹ P x] ⟹ P a
```

### 3.6 Case Study: Verified Model Checking

```
typedecl state

consts M :: "(state × state)set"

typedecl atom

consts L :: "state ⇒ atom set"

datatype formula = Atom atom
                 | Neg formula
                 | And formula formula
                 | AX formula
                 | EF formula

consts valid :: "state ⇒ formula ⇒ bool"   ("(_ ⊨ _)" [80,80] 80)

primrec
"s ⊨ Atom a  = (a ∈ L s)"
"s ⊨ Neg f   = (¬(s ⊨ f))"
"s ⊨ And f g = (s ⊨ f ∧ s ⊨ g)"
"s ⊨ AX f    = (∀t. (s,t) ∈ M ⟶ t ⊨ f)"
"s ⊨ EF f    = (∃t. (s,t) ∈ M* ∧ t ⊨ f)"

consts mc :: "formula ⇒ state set"
primrec
"mc(Atom a)  = {s. a ∈ L s}"
"mc(Neg f)   = -mc f"
"mc(And f g) = mc f ∩ mc g"
"mc(AX f)    = {s. ∀t. (s,t) ∈ M ⟶ t ∈ mc f}"
"mc(EF f)    = lfp(λT. mc f ∪ (M⁻¹ `` T))"

lemma mono_ef: "mono(λT. A ∪ (M⁻¹ `` T))"
```

```
apply(rule monoI)
apply blast
done

lemma EF_lemma:
  "lfp(λT. A ∪ (M⁻¹ `` T)) = {s. ∃t. (s,t) ∈ M* ∧ t ∈ A}"
apply(rule equalityI)
 thm lfp_lowerbound
 apply(rule lfp_lowerbound)
 apply(blast intro: rtrancl_trans)
apply(rule subsetI)
apply clarsimp
apply(erule converse_rtrancl_induct)
thm lfp_unfold[OF mono_ef]
 apply(subst lfp_unfold[OF mono_ef])
 apply(blast)
apply(subst lfp_unfold[OF mono_ef])
apply(blast)
done

theorem "mc f = {s. s ⊨ f}"
apply(induct_tac f)
apply(auto simp add: EF_lemma)
done
```

**Exercise 3.1** *AX* has a dual operator *EN*[1] ("there exists a next state such that") with the intended semantics

```
s ⊨ EN f = (∃t. (s, t) ∈ M ∧ t ⊨ f)
```

Fortunately, *EN f* can already be expressed as a PDL formula. How?

Show that the semantics for *EF* satisfies the following recursion equation:

```
s ⊨ EF f = (s ⊨ f ∨ s ⊨ EN (EF f))
```

# 4 Inductive Definitions

## 4.1 Even Numbers

### 4.1.1 The Definition

```
consts even :: "nat set"
inductive even
intros
zero[intro!]: "0 ∈ even"
step[intro!]: "n ∈ even ⟹ Suc(Suc n) ∈ even"

lemma [simp,intro!]: "2 dvd n ⟹ 2 dvd Suc(Suc n)"
apply (unfold dvd_def)
apply clarify
apply (rule_tac x = "Suc k" in exI, simp)
done
```

[1] We cannot use the customary *EX* as that is the ASCII-equivalent of ∃



### 4.1.2 Rule Induction

Rule induction for set *even*, *even.induct*:

```
[xa ∈ even; P 0; ⋀n. [n ∈ even; P n] ⟹ P (Suc (Suc n))] ⟹ P xa


lemma even_imp_dvd: "n ∈ even ⟹ 2 dvd n"
apply (erule even.induct)
apply simp_all
done
```

### 4.1.3 Rule Inversion

```
inductive_cases Suc_Suc_case [elim!]: "Suc(Suc n) ∈ even"

[Suc (Suc n) ∈ even; n ∈ even ⟹ P] ⟹ P


lemma "Suc(Suc n) ∈ even ⟹ n ∈ even"
by blast
```

## 4.2 Mutually Inductive Definitions

```
consts evn :: "nat set"
       odd :: "nat set"

inductive evn odd
intros
zero: "0 ∈ evn"
evnI: "n ∈ odd ⟹ Suc n ∈ evn"
oddI: "n ∈ evn ⟹ Suc n ∈ odd"

lemma "(m ∈ evn ⟶ 2 dvd m) ∧ (n ∈ odd ⟶ 2 dvd (Suc n))"
apply(rule evn_odd.induct)
by simp_all
```

### 4.3 The Reflexive Transitive Closure

```
consts rtc :: "('a × 'a)set ⇒ ('a × 'a)set"   ("_*" [1000] 999)
inductive "r*"
intros
refl[iff]:  "(x,x) ∈ r*"
step:       "[ (x,y) ∈ r; (y,z) ∈ r* ] ⟹ (x,z) ∈ r*"

lemma [intro]: "(x,y) : r ⟹ (x,y) ∈ r*"
by(blast intro: rtc.step)

lemma rtc_trans: "[ (x,y) ∈ r*; (y,z) ∈ r* ] ⟹ (x,z) ∈ r*"
apply(erule rtc.induct)
oops

lemma rtc_trans[rule_format]:
  "(x,y) ∈ r* ⟹ (y,z) ∈ r* ⟶ (x,z) ∈ r*"
apply(erule rtc.induct)
 apply(blast)
apply(blast intro: rtc.step)
done
```

**Exercise 4.1** Show that the converse of `rtc.step` also holds:

```
[(x, y) ∈ r*; (y, z) ∈ r] ⟹ (x, z) ∈ r*
```

### 4.4 The accessible part of a relation

```
consts  acc :: "('a × 'a) set ⇒ 'a set"
inductive "acc r"
intros
  "(∀y. (y,x) ∈ r ⟶ y ∈ acc r) ⟹ x ∈ acc r"

lemma "wf{(x,y). (x,y) ∈ r ∧ y ∈ acc r}"
thm wfI
apply(rule_tac A = "acc r" in wfI)
 apply (blast elim: acc.elims)
apply(simp(no_asm_use))
thm acc.induct
apply(erule acc.induct)
by blast

consts  accs :: "('a × 'a) set ⇒ 'a set"
inductive "accs r"
intros
 "r''{x} ∈ Pow(accs r) ⟹ x ∈ accs r"
```



```
monos Pow_mono
```

## References

[1] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2002. http://www.in.tum.de/~nipkow/LNCS2283/.

Course Notes Part II (to appear)

Alessandro Armando
*University of Genua, Italy*

Jörg Siekmann, Christoph Benzmüller
*Saarland University and DFKI, Saarbrücken, Germany*

Bruno Buchberger, Wolfgang Windsteiger
*RISC Linz, Austria*

James Davenport
*University of Bath, England*

Manfred Kerber
*University of Birmingham, England*

Michael Kohlhase
*Carnegie Mellon University, Pittsburgh, USA*

Andreas Meier, Volker Sorge
*Saarland University, Saarbrücken, Germany*

Tom Kelsey
*University of St.Andrews, Scotland*

Olga Caprotti
*(RISC Linz, Austria)*