NICTA

COMP 4161

NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein
Formal Methods

$\longrightarrow$

CONTENT

NICTA

- ➜ Intro & motivation, getting started with Isabelle
- ➜ **Foundations & Principles**
  - Lambda Calculus
  - **Higher Order Logic, natural deduction**
  - **Term rewriting**
- ➜ Proof & Specification Techniques
  - Inductively defined sets, rule induction
  - Datatypes, recursion, induction
  - Calculational reasoning, mathematics style proofs
  - Hoare logic, proofs about programs

LAST TIME ON HOL

- ➜ Defining HOL
- ➜ Higher Order Abstract Syntax
- ➜ Deriving proof rules
- ➜ More automation

NICTA

➜ **Axioms**:

Expample: **axioms** refl: "$t = t$"

Do not use. Evil. Can make your logic inconsistent.

➜ **Definitions:**

Example: **defs** inj_def: "inj $f \equiv \forall x\ y.\ f\ x = f\ y \longrightarrow x = y$"

➜ **Proofs:**

Example: **lemma** "inj $(\lambda x.\ x + 1)$"

The harder, but safe choice.

THE THREE BASIC WAYS OF INTRODUCING TYPES

- **typedecl**: by name only

  Example: **typedecl** names

  Introduces new type *names* without any further assumptions

- **types**: by abbreviation

  Example: **types** α rel = "$\alpha \Rightarrow \alpha \Rightarrow bool$"

  Introduces abbreviation *rel* for existing type $\alpha \Rightarrow \alpha \Rightarrow bool$

  Type abbreviations are immediatly expanded internally

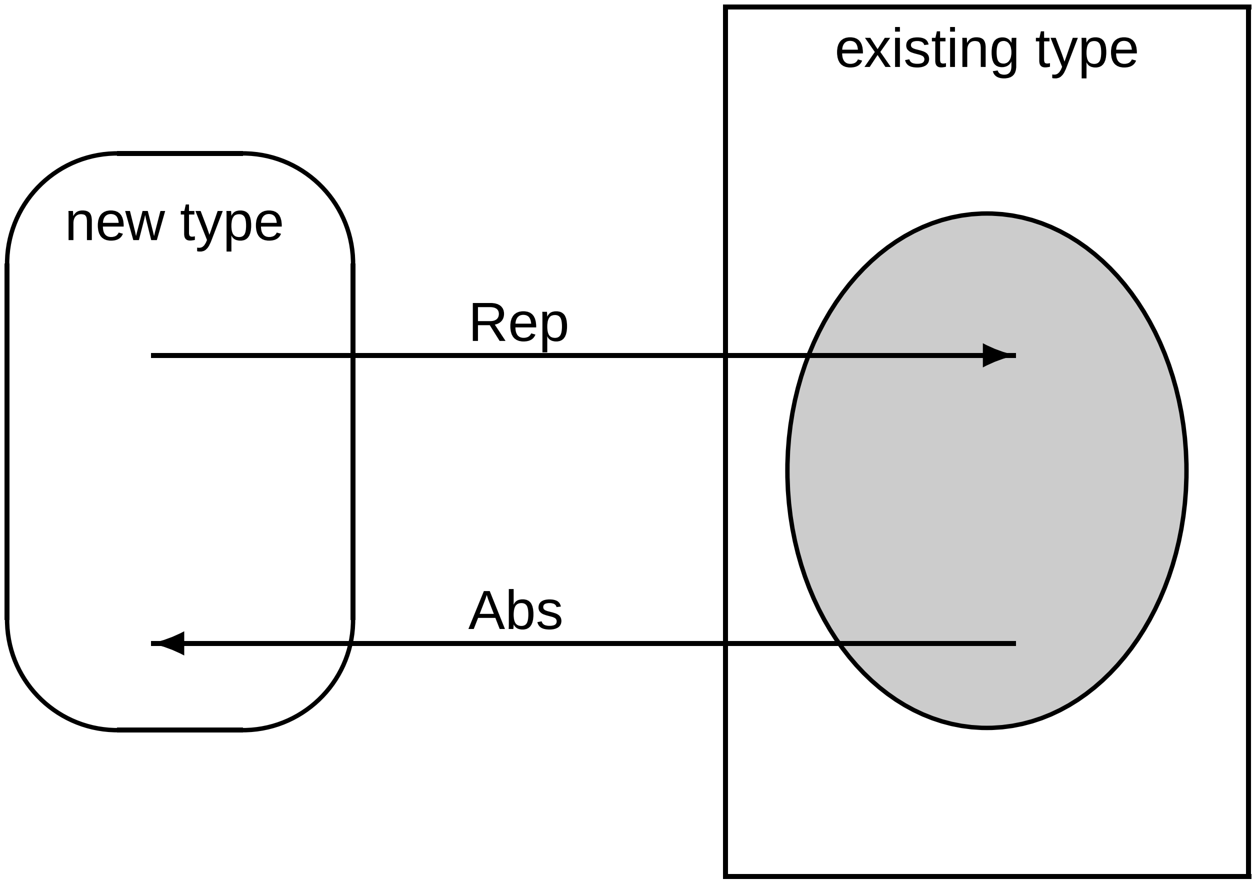
- **typedef**: by definiton as a set

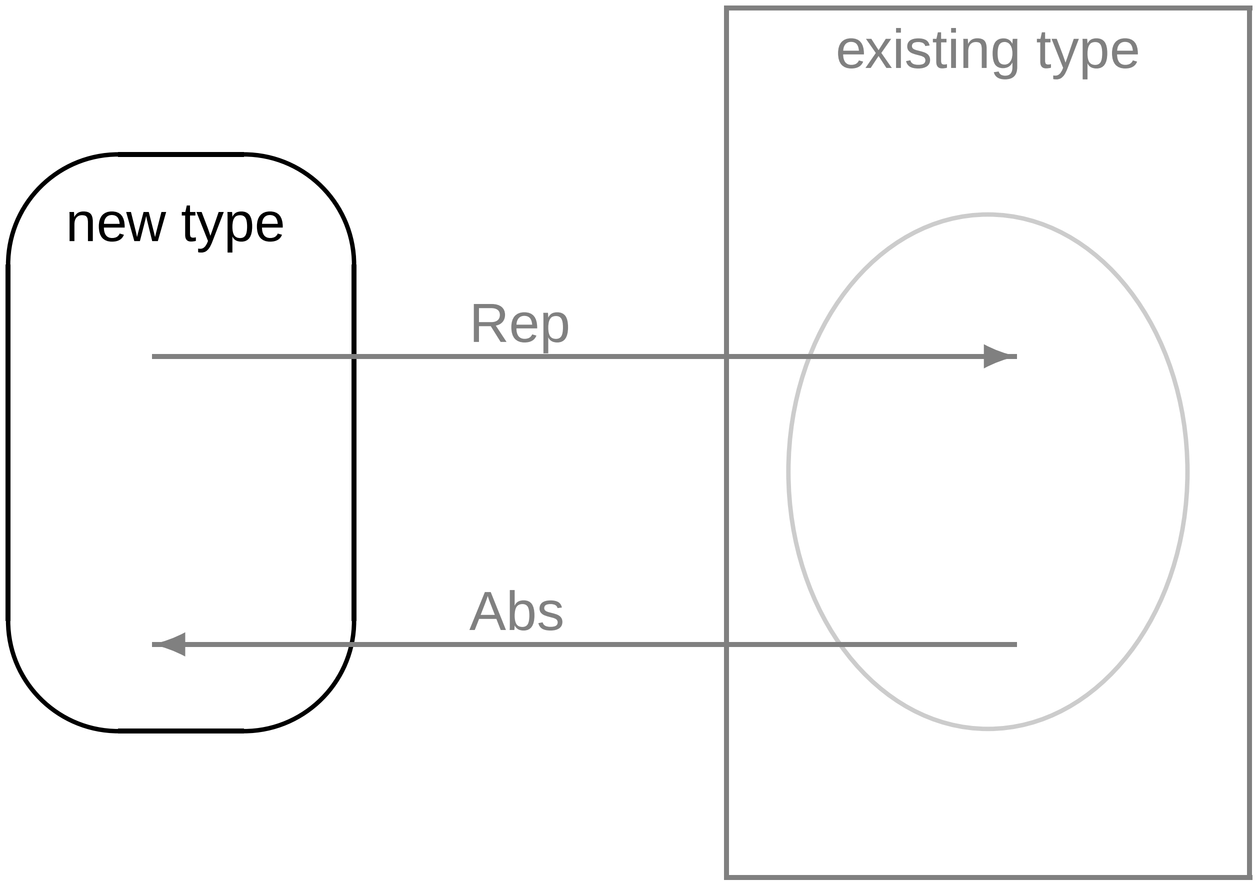
  Example: **typdef** new_type = "$\{$some set$\}$" <proof>

  Introduces a new type as a subset of an existing type.

  The proof shows that the set on the rhs in non-empty.

HOW TYPEDEF WORKS




existing type
new type
Rep
Abs

EXAMPLE: PAIRS

(α, β) Prod

① Pick existing type: $\alpha \Rightarrow \beta \Rightarrow$ bool

② Identify subset:

(α, β) Prod $= \{f.\ \exists a\ b.\ f = \lambda(x :: \alpha)\ (y :: \beta).\ x = a \wedge y = b\}$

③ We get from Isabelle:

- functions Abs_Prod, Rep_Prod
- both injective
- Abs_Prod (Rep_Prod x) $= x$

④ We now can:

- define constants Pair, fst, snd in terms of Abs_Prod and Rep_Prod
- derive all characteristic theorems
- forget about Rep/Abs, use characteristic theorems instead

Demo: Introducting new Types

Term Rewriting

Given a set of equations

$$
\begin{aligned}
l_1 &= r_1 \\
l_2 &= r_2 \\
&\vdots \\
l_n &= r_n
\end{aligned}
$$

does equation $l = r$ hold?

Applications in:

- ➜ **Mathematics** (algebra, group theory, etc)
- ➜ **Functional Programming** (model of execution)
- ➜ **Theorem Proving** (dealing with equations, simplifying statements)

NICTA

use equations as reduction rules

$$l_1 \longrightarrow r_1$$

$$l_2 \longrightarrow r_2$$

$$\vdots$$

$$l_n \longrightarrow r_n$$

decide $l = r$ **by deciding** $l \stackrel{*}{\longleftrightarrow} r$

Arrow Cheat Sheet

$\xrightarrow{0}$	$=$	$\{(x,y) \mid x = y\}$	identity
$\xrightarrow{n+1}$	$=$	$\xrightarrow{n} \circ \longrightarrow$	n+1 fold composition
$\xrightarrow{+}$	$=$	$\bigcup_{i>0} \xrightarrow{i}$	transitive closure
$\xrightarrow{*}$	$=$	$\xrightarrow{+} \cup \xrightarrow{0}$	reflexive transitive closure
$\xrightarrow{=}$	$=$	$\longrightarrow \cup \xrightarrow{0}$	reflexive closure
$\xrightarrow{-1}$	$=$	$\{(y,x) \mid x \longrightarrow y\}$	inverse
$\longleftarrow$	$=$	$\xrightarrow{-1}$	inverse
$\longleftrightarrow$	$=$	$\longleftarrow \cup \longrightarrow$	symmetric closure
$\xleftrightarrow{+}$	$=$	$\bigcup_{i>0} \xleftrightarrow{i}$	transitive symmetric closure
$\xleftrightarrow{*}$	$=$	$\xleftrightarrow{+} \cup \xleftrightarrow{0}$	reflexive transitive symmetric closure

How to Decide $l \stackrel{*}{\longleftrightarrow} r$

Same idea as for β: look for n such that $l \stackrel{*}{\longrightarrow} n$ and $r \stackrel{*}{\longrightarrow} n$

Does this always work?

If $l \stackrel{*}{\longrightarrow} n$ and $r \stackrel{*}{\longrightarrow} n$ then $l \stackrel{*}{\longleftrightarrow} r$. Ok.

If $l \stackrel{*}{\longleftrightarrow} r$, will there always be a suitable n? **No!**

Example:

Rules: $f\ x \longrightarrow a$, $g\ x \longrightarrow b$, $f\ (g\ x) \longrightarrow b$

$f\ x \stackrel{*}{\longleftrightarrow} g\ x$ because $f\ x \longrightarrow a \longleftarrow f\ (g\ x) \longrightarrow b \longleftarrow g\ x$

But: $f\ x \longrightarrow a$ and $g\ x \longrightarrow b$ and a, b in normal form

Works only for systems with **Church-Rosser** property:

$$l \stackrel{*}{\longleftrightarrow} r \Longrightarrow \exists n.\ l \stackrel{*}{\longrightarrow} n \wedge r \stackrel{*}{\longrightarrow} n$$

Fact: $\longrightarrow$ is Church-Rosser iff it is confluent.

CONFLUENCE

$$\begin{tikzcd} & s \arrow[dl, "*"'] \arrow[dr, "*"] & \\ x \arrow[dr, dashed, "*"'] & & y \arrow[dl, dashed, "*"] \\ & t & \end{tikzcd}$$

Problem:

is a given set of reduction rules confluent?

undecidable

Local Confluence

$$\begin{tikzcd} & s \arrow[dl] \arrow[dr] & \\ x \arrow[dr, dashed, "*"'] & & y \arrow[dl, dashed, "*"] \\ & t & \end{tikzcd}$$

Fact: local confluence and termination $\Longrightarrow$ confluence

TERMINATION

$\longrightarrow$ is **terminating** if there are no infinite reduction chains

$\longrightarrow$ is **normalizing** if each element has a normal form

$\longrightarrow$ is **convergent** if it is terminating and confluent

Example:

$\longrightarrow_\beta$ in λ is not terminating, but confluent

$\longrightarrow_\beta$ in $\lambda^\rightarrow$ is terminating and confluent, i.e. convergent

Problem: is a given set of reduction rules terminating?

undecidable

WHEN IS $\longrightarrow$ TERMINATING?

Basic Idea: when the r_i are in some way simpler then the l_i

More formally: $\longrightarrow$ is terminating when

there is a well founded order $<$ in which $r_i < l_i$ for all rules.

(well founded = no infinite decreasing chains $a_1 > a_2 > \ldots$)

Example: $f\ (g\ x) \longrightarrow g\ x$, $g\ (f\ x) \longrightarrow f\ x$

This system always terminates. Reduction order:

$s <_r t$ iff $size(s) < size(t)$ with

$size(s) =$ numer of function symbols in s

① $g\ x <_r f\ (g\ x)$ and $f\ x <_r g\ (f\ x)$

② $<_r$ is well founded, because $<$ is well founded on $\mathbb{N}$

TERM REWRITING IN ISABELLE

Term rewriting engine in Isabelle is called **Simplifier**

apply simp

- ➜ uses simplification rules
- ➜ (almost) blindly from left to right
- ➜ until no rule is applicable.

termination: not guaranteed
(may loop)

confluence: not guaranteed
(result may depend on which rule is used first)

CONTROL

➜ Equations turned into simplifaction rules with **[simp]** attribute

➜ Adding/deleting equations locally:
apply (simp add: $<$rules$>$) and **apply** (simp del: $<$rules$>$)

➜ Using only the specified set of equations:
apply (simp only: $<$rules$>$)

DEMO