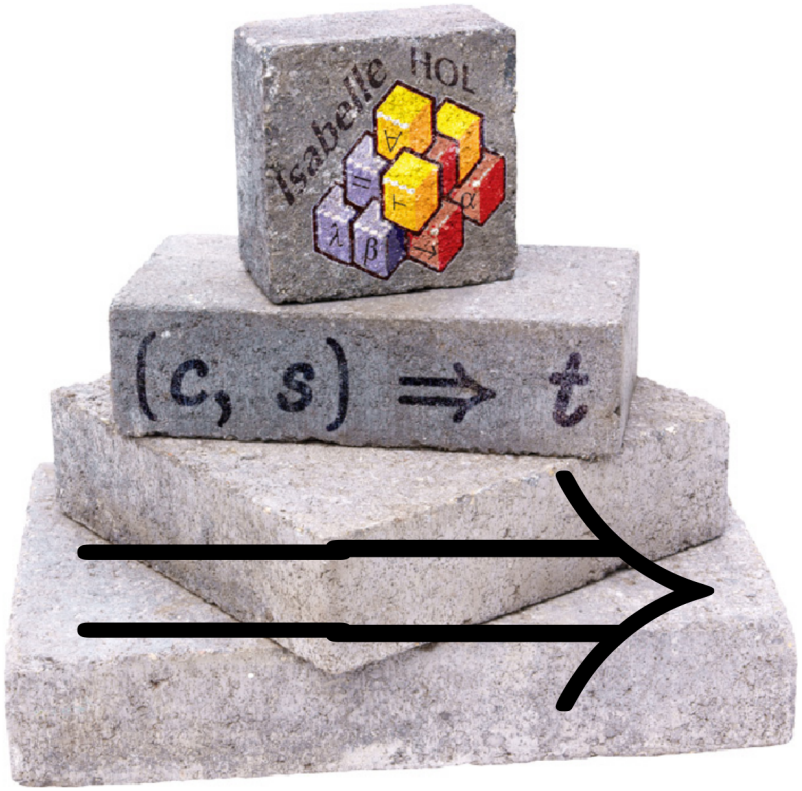


Higher-Order Logics and Interactive Theorem Proving with Isabelle/HOL III

Formal Systems II: Application

Bernhard Beckert · Mattias Ulbrich · Michael Kirsten | Summer Term 2025

Back to HOL

Base: $bool, \Rightarrow, ind \qquad =, \longrightarrow, \varepsilon$

And the rest is definitions:

$$
\begin{array}{lcl}
\text{True} & \equiv & (\lambda x :: bool.\ x) = (\lambda x.\ x) \\
\text{All } P & \equiv & P = (\lambda x.\ \text{True}) \\
\text{Ex } P & \equiv & \forall Q.\ (\forall x.\ P\ x \longrightarrow Q) \longrightarrow Q \\
\text{False} & \equiv & \forall P.\ P \\
\neg P & \equiv & P \longrightarrow \text{False} \\
P \wedge Q & \equiv & \forall R.\ (P \longrightarrow Q \longrightarrow R) \longrightarrow R \\
P \vee Q & \equiv & \forall R.\ (P \longrightarrow R) \longrightarrow (Q \longrightarrow R) \longrightarrow R \\
\text{If } P\ x\ y & \equiv & \text{SOME } z.\ (P = \text{True} \longrightarrow z = x) \wedge (P = \text{False} \longrightarrow z = y) \\
\text{inj } f & \equiv & \forall x\ y.\ f\ x = f\ y \longrightarrow x = y \\
\text{surj } f & \equiv & \forall y.\ \exists x.\ y = f\ x
\end{array}
$$

Institute of Information Security and Dependability (KASTEL)

The Axioms of HOL

$$\frac{}{t = t}\ \text{refl} \qquad \frac{s = t \quad P\ s}{P\ t}\ \text{subst} \qquad \frac{\bigwedge x.\ f\ x = g\ x}{(\lambda x.\ f\ x) = (\lambda x.\ g\ x)}\ \text{ext}$$

$$\frac{P \Longrightarrow Q}{P \longrightarrow Q}\ \text{impl} \qquad \frac{P \longrightarrow Q \quad P}{Q}\ \text{mp}$$

$$\frac{}{(P \longrightarrow Q) \longrightarrow (Q \longrightarrow P) \longrightarrow (P = Q)}\ \text{iff}$$

$$\frac{}{P = \text{True} \vee P = \text{False}}\ \text{True\_or\_False}$$

$$\frac{P\ ?x}{P\ (\text{SOME}\ x.\ P\ x)}\ \text{somel}$$

$$\frac{}{\exists f :: ind \Rightarrow ind.\ \text{inj}\ f \wedge \neg \text{surj}\ f}\ \text{infty}$$

Institute of Information Security and Dependability (KASTEL)

That's it.

- ➜ 3 basic constants
- ➜ 3 basic types
- ➜ 9 axioms

With this you can define and derive all the rest.

Isabelle knows 2 more axioms:

$$\frac{x = y}{x \equiv y}\ \text{eq\_reflection} \qquad \frac{}{(\text{THE } x.\ x = a) = a}\ \text{the\_eq\_trivial}$$

Institute of Information Security
and Dependability (KASTEL)

Induction: Example for Even Numbers

Informally:

- 0 is even
- If n is even, so is $n + 2$
- These are the only even numbers

In Isabelle/HOL:

inductive $ev :: nat \Rightarrow bool$
where
$ev\ 0$ |
$ev\ n \Longrightarrow ev\ (n + 2)$

An easy proof: $ev\ 4$

$$ev\ 0 \Longrightarrow ev\ 2 \Longrightarrow ev\ 4$$

Institute of Information Security and Dependability (KASTEL)

Consider

fun $evn :: nat \Rightarrow bool$ **where**
$evn\ 0 = True\ |$
$evn\ (Suc\ 0) = False\ |$
$evn\ (Suc\ (Suc\ n)) = evn\ n$

A trickier proof: $ev\ m \Longrightarrow evn\ m$

By induction on the *structure* of the derivation of $ev\ m$

Two cases: $ev\ m$ is proved by

- rule $ev\ 0$
  $\Longrightarrow m = 0 \Longrightarrow evn\ m = True$
- rule $ev\ n \Longrightarrow ev\ (n+2)$
  $\Longrightarrow m = n+2$ and $evn\ n$ (IH)
  $\Longrightarrow evn\ m = evn\ (n+2) = evn\ n = True$

Institute of Information Security and Dependability (KASTEL)

Rule induction for *ev*

To prove

$$ev\ n \Longrightarrow P\ n$$

by *rule induction* on $ev\ n$ we must prove

- $P\ 0$
- $P\ n \Longrightarrow P(n{+}2)$

Rule `ev.induct`:

$$\frac{ev\ n \quad P\ 0 \quad \bigwedge n.\ [\![\ ev\ n;\ P\ n\ ]\!] \Longrightarrow P(n{+}2)}{P\ n}$$

Institute of Information Security
and Dependability (KASTEL)

Format of Inductive Definitions

inductive $I :: \tau \Rightarrow bool$ **where**

$$\llbracket\ I\ a_1;\ \ldots\ ;\ I\ a_n\ \rrbracket \Longrightarrow I\ a\ \ |$$

$$\vdots$$

Note:

- I may have multiple arguments.
- Each rule may also contain *side conditions* not involving I.

Institute of Information Security and Dependability (KASTEL)

Rule Induction in General

To prove

$$I\ x \Longrightarrow P\ x$$

by *rule induction* on $I\ x$
we must prove for every rule

$$[\![\ I\ a_1;\ \ldots\ ;\ I\ a_n\ ]\!] \Longrightarrow I\ a$$

that P is preserved:

$$[\![\ I\ a_1;\ P\ a_1;\ \ldots\ ;\ I\ a_n;\ P\ a_n\ ]\!] \Longrightarrow P\ a$$

Institute of Information Security
and Dependability (KASTEL)

Inductively Defined Set

inductive_set $I :: \tau\ set$ **where**
$\llbracket\ a_1 \in I; \ldots ; a_n \in I\ \rrbracket \Longrightarrow a \in I\ \mid$
$\vdots$

Difference to **inductive**:

- arguments of I are tupled, not curried
- I can later be used with set theoretic operators, eg $I \cup \ldots$

Institute of Information Security and Dependability (KASTEL)

Named Assumptions

In a proof of

$$I \ldots \Longrightarrow A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

by rule induction on $I \ldots$:

In the context of

case R

we have

$R.IH$	the induction hypotheses
$R.hyps$	the assumptions of rule R
$R.prems$	the premises A_i
R	$R.IH + R.hyps + R.prems$

Institute of Information Security and Dependability (KASTEL)

Rule Induction

inductive $I :: \tau \Rightarrow \sigma \Rightarrow bool$
where
$rule_1$: ...
⋮
$rule_n$: ...

show $I\ x\ y \Longrightarrow P\ x\ y$
proof $(induction\ rule:\ I.induct)$
  case $rule_1$
  ...
  show $?case$
next
⋮
next
  case $rule_n$
  ...
  show $?case$
qed

Institute of Information Security
and Dependability (KASTEL)

Style Remark

- **case** $(Suc\ n)$ ... **show** $?case$
  is easy to write and maintain
- **fix** n **assume** $formula$ ... **show** $formula'$
  is easier to read:
  - all information is shown locally
  - no contextual references (e.g. $?case$)

Institute of Information Security
and Dependability (KASTEL)

More on (Data) Types

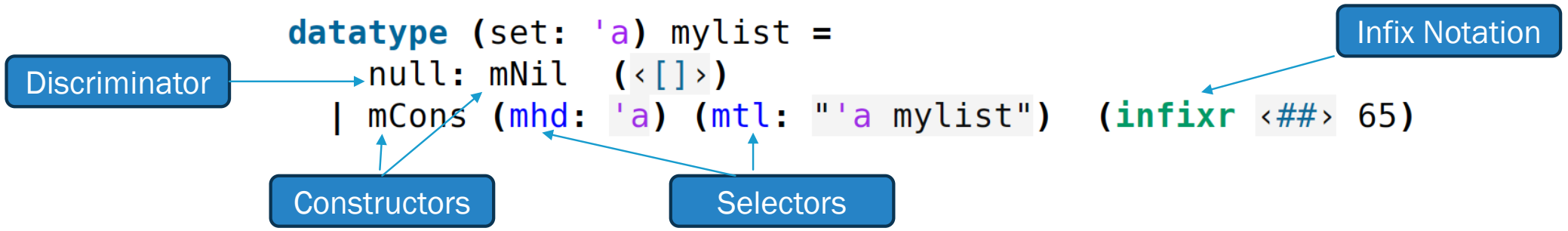
- One common use case of datatypes is an option datatype

```
datatype 'a option = None | Some 'a
```

- Datatypes can be parameterised by multiple types:

```
datatype ('a, 'b, 'c) three = Three 'a 'b 'c
```

- Datatypes can also be annotated:



- The datatypes (and co-datatypes) tutorial has significantly more information.

Type Synonyms, Declarations, and Definitions

- A type synonym can be useful to make a formalisation more readable/descriptive. E.g.

```
type_synonym 'a edge = "'a set"
```

  - declares a parameterised edge type which is the same as a set
- A type declaration declares a new type without defining it

```
typedecl Test
```

- A type definition allows you to define a new type

```
typedef three = "{0:: nat, 1, 2}"
  apply (intro exI[of _ 0]) (* Goal must show RHS is non-empty *)
  by simp
```

  - You must prove the type is not empty
  - Introduces Rep and Abs properties to convert between reasoning on base type and new type (then you need to establish useful properties)…
  - Or in this case just use a datatype which does the setup for you!

Institute of Information Security and Dependability (KASTEL)

Application: Programming Language Semantics

Language with only arithmetic expressions and assignments:

```
a ::= 6;; b ::= 7;; x = a + b
```


Institute of Information Security
and Dependability (KASTEL)

Application: Programming Language Semantics

Language with only arithmetic expressions and assignments:

```
a ::= 6;; b ::= 7;; x = a + b
```

```
type_synonym vname = string
datatype aexp = Num int | Val vname | Plus aexp aexp

datatype
  com =  Assign vname aexp       (‹_ ::= _› [1000, 61] 61)
      | Seq    com  com         (‹_;;/ _›  [60, 61] 60)
```


Institute of Information Security
and Dependability (KASTEL)

Application: Programming Language Semantics

Predicate $(c, s) \Rightarrow s'$: program c executed in state s yields state s'.

```
type_synonym val = int
type_synonym state = "vname ⇒ val"

fun aval :: "aexp ⇒ state ⇒ val" where
"aval (Num n) s = n" |
"aval (Val x) s = s x" |
"aval (Plus a₁ a₂) s = aval a₁ s + aval a₂ s"

inductive
  big_step :: "com × state ⇒ state ⇒ bool" (infix ‹⇒› 55) where
  Assign: "(x ::= a,s) ⇒ s(x := aval a s)" |
  Seq:    "⟦ (c₁,s₁) ⇒ s₂;  (c₂,s₂) ⇒ s₃ ⟧ ⟹ (c₁;;c₂, s₁) ⇒ s₃"
```


Institute of Information Security
and Dependability (KASTEL)

Application: Toy Assembly and Compiler

Simple assembly language with four instructions:

```
datatype instr =
  LOADI int | LOAD vname | ADD | STORE vname

fun acomp :: "aexp ⇒ instr list" where
"acomp (Num n) = [LOADI n]" |
"acomp (Val x) = [LOAD x]" |
"acomp (Plus a1 a2) = acomp a1 @ acomp a2 @ [ADD]"

fun ccomp :: "com ⇒ instr list" where
"ccomp (x ::= a) = acomp a @ [STORE x]" |
"ccomp (c₁;;c₂) = ccomp c₁ @ ccomp c₂"
```


Institute of Information Security
and Dependability (KASTEL)

Application: Execution

Predicate: $instr \vdash (i, s, stk) \rightarrow^* (i', s', stk')$

"*instr* executed on *stk* with state *s* at instruction counter *i* leads to state s' on stk' with new instruction counter i'."

```
type_synonym stack = "val list"
type_synonym config = "int × state × stack"

fun iexec :: "instr ⇒ config ⇒ config" where
"iexec instr (i,s,stk) = (case instr of
  LOADI n ⇒ (i+1,s, n#stk) |
  LOAD x ⇒ (i+1,s, s x # stk) |
  ADD ⇒ (i+1,s, (hd2 stk + hd stk) # tl2 stk) |
  STORE x ⇒ (i+1,s(x := hd stk),tl stk))"
```


Institute of Information Security
and Dependability (KASTEL)

Application: Correctness Proof

Download `Tiny_Semantics.thy` from ILIAS.

Institute of Information Security
and Dependability (KASTEL)

IMP: A Small Imperative Language

Commands:

datatype com	=	SKIP	
	|	Assign vname aexp	(_ := _)
	|	Semi com com	(_; _)
	|	Cond bexp com com	(IF _ THEN _ ELSE _)
	|	While bexp com	(WHILE _ DO _ OD)

type_synonym vname = string
type_synonym state = vname ⇒ nat

type_synonym aexp = state ⇒ nat
type_synonym bexp = state ⇒ bool

Institute of Information Security and Dependability (KASTEL)

Example Program

Usual syntax:

$$
\begin{array}{l}
B := 1; \\
\text{WHILE } A \neq 0 \text{ DO} \\
\quad B := B * A; \\
\quad A := A - 1 \\
\text{OD}
\end{array}
$$

Expressions are functions from state to bool or nat:

$$
\begin{array}{l}
B := (\lambda\sigma.\ 1); \\
\text{WHILE } (\lambda\sigma.\ \sigma\ A \neq 0) \text{ DO} \\
\quad B := (\lambda\sigma.\ \sigma\ B * \sigma\ A); \\
\quad A := (\lambda\sigma.\ \sigma\ A - 1) \\
\text{OD}
\end{array}
$$

Institute of Information Security
and Dependability (KASTEL)

Structural Operational Semantics

$$\frac{}{\langle \text{SKIP}, \sigma \rangle \to \sigma}$$

$$\frac{e\ \sigma = v}{\langle \text{x} := \text{e}, \sigma \rangle \to \sigma[x \mapsto v]}$$

$$\frac{\langle c_1, \sigma \rangle \to \sigma' \quad \langle c_2, \sigma' \rangle \to \sigma''}{\langle c_1; c_2, \sigma \rangle \to \sigma''}$$

$$\frac{b\ \sigma = \text{True} \quad \langle c_1, \sigma \rangle \to \sigma'}{\langle \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, \sigma \rangle \to \sigma'}$$

$$\frac{b\ \sigma = \text{False} \quad \langle c_2, \sigma \rangle \to \sigma'}{\langle \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, \sigma \rangle \to \sigma'}$$

Structural Operational Semantics Cont'd

$$\frac{b\ \sigma = \text{False}}{\langle \text{WHILE } b \text{ DO } c \text{ OD}, \sigma \rangle \to \sigma}$$

$$\frac{b\ \sigma = \text{True} \quad \langle c, \sigma \rangle \to \sigma' \quad \langle \text{WHILE } b \text{ DO } c \text{ OD}, \sigma' \rangle \to \sigma''}{\langle \text{WHILE } b \text{ DO } c \text{ OD}, \sigma \rangle \to \sigma''}$$

Isabelle's Module System: Locales

- Locales are Isabelle's module system. From a logical perspective, they are simply persistent contexts.

$$\bigwedge x_1 \ldots x_n. [\![A_1; \ldots; A_m]\!] \Rightarrow C.$$

- Provides fixed type and term variables and contextual assumptions within a local context.
- Type classes use and can interact with the underlying locale infrastructure.

```
locale semigroup_orig =
  fixes mult :: "'a ⇒ 'a ⇒ 'a" (infixl"⊗" 70)
  assumes assoc: "(x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)"
```

Same params/assumptions as before

Locale inheritance

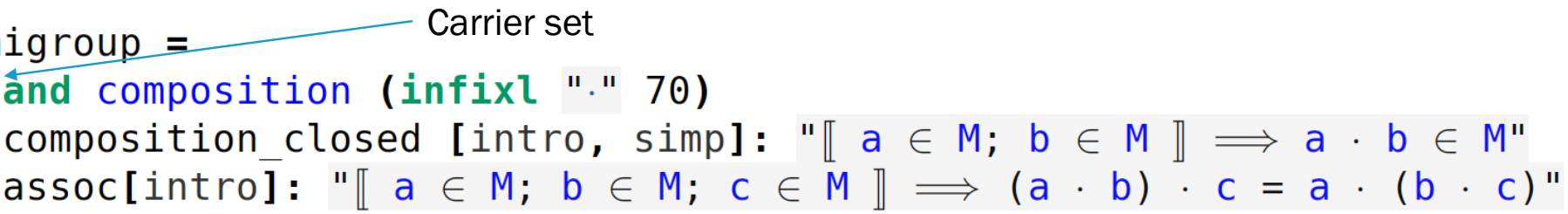
Class

```
class semigroup_orig_add = plus +
  assumes add_assoc: "(a + b) + c = a + (b + c)"
begin

sublocale add: semigroup_orig plus
  by standard (fact add_assoc)

end
```


Institute of Information Security and Dependability (KASTEL)

Locales Cont'd

- Locales allow us to work explicitly with "carrier sets" (if we want to)

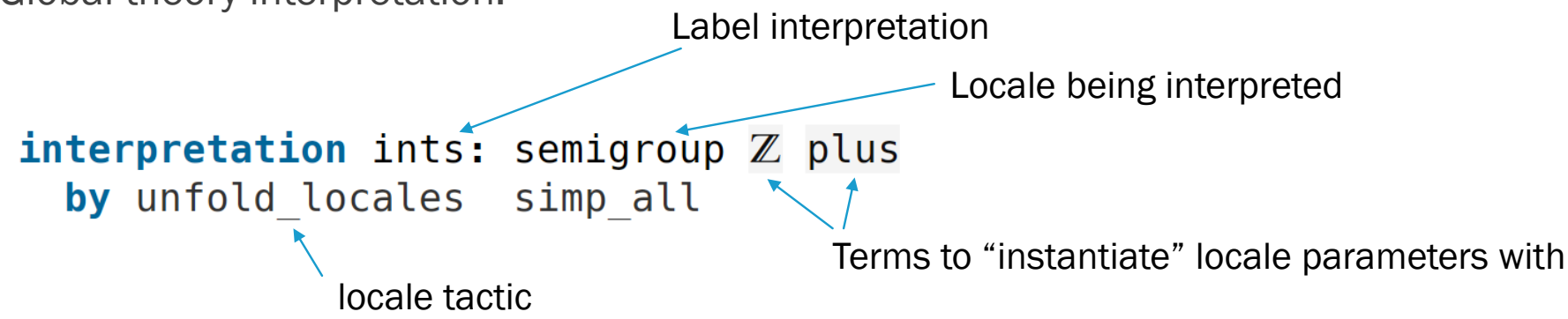


- Think of locales as more of a set-based rather than type-based approach.

Institute of Information Security
and Dependability (KASTEL)

Interpreting a Locale

- Global theory interpretation:



- Can also now use inherited locale properties outside locale context

```
lemma "(1 + 2) + (3 ::int) = 1 + (2 + 3)"
  using ints.assoc by simp
```

Must reference named interpretation

Other Facets of Isabelle

- Document preparation: you can generate LaTeX documents from your theories
- Axiomatic type classes: a general approach to polymorphism and overloading when there are shared laws
- Code generation: you can generate executable code from the formal functional programs you have verified
  $\Rightarrow$ Algorithms can be verified and then executed (ML, Haskell, Scala)
- Archive of Formal Proofs: a massive collection of proven und useful theories, which you can extend!
- More on Program Verification: Imperative HOL and refinement to efficient compiler code using separation logic

Institute of Information Security
and Dependability (KASTEL)