

An Isabelle/HOL Primer

Christian Sternagel

JAIST

May 25, 2012

Ogawa-Hirokawa-Lab Seminar

Overview

- Simplification
- Function Definitions
- Computational Reasoning

Simplification

Example – Term Rewriting

- a set of rules, also called a term rewrite system (TRS)

$$0 + y \rightarrow y$$

$$0 \times y \rightarrow 0$$

$$s(x) + y \rightarrow s(x + y)$$

$$s(x) \times y \rightarrow y + (x \times y)$$

- 'compute' 1×2

$$s(0) \times s^2(0)$$

Example – Term Rewriting

- a set of rules, also called a term rewrite system (TRS)

$$0 + y \rightarrow y$$

$$0 \times y \rightarrow 0$$

$$s(x) + y \rightarrow s(x + y)$$

$$s(x) \times y \rightarrow y + (x \times y)$$

- 'compute' 1×2

$$s(0) \times s^2(0) \rightarrow s^2(0) + (0 \times s^2(0))$$

Example – Term Rewriting

- a set of rules, also called a term rewrite system (TRS)

$$\begin{array}{ll} 0 + y \rightarrow y & 0 \times y \rightarrow 0 \\ s(x) + y \rightarrow s(x + y) & s(x) \times y \rightarrow y + (x \times y) \end{array}$$

- 'compute' 1×2

$$\begin{aligned} s(0) \times s^2(0) &\rightarrow s^2(0) + (0 \times s^2(0)) \\ &\rightarrow s^2(0) + 0 \end{aligned}$$

Example – Term Rewriting

- a set of rules, also called a term rewrite system (TRS)

$$0 + y \rightarrow y$$

$$0 \times y \rightarrow 0$$

$$s(x) + y \rightarrow s(x + y)$$

$$s(x) \times y \rightarrow y + (x \times y)$$

- 'compute' 1×2

$$\begin{aligned} s(0) \times s^2(0) &\rightarrow s^2(0) + (0 \times s^2(0)) \\ &\rightarrow s^2(0) + 0 \\ &\rightarrow s(s(0) + 0) \end{aligned}$$

Example – Term Rewriting

- a set of rules, also called a term rewrite system (TRS)

$$0 + y \rightarrow y$$

$$0 \times y \rightarrow 0$$

$$s(x) + y \rightarrow s(x + y)$$

$$s(x) \times y \rightarrow y + (x \times y)$$

- 'compute' 1×2

$$\begin{aligned} s(0) \times s^2(0) &\rightarrow s^2(0) + (0 \times s^2(0)) \\ &\rightarrow s^2(0) + 0 \\ &\rightarrow s(s(0) + 0) \\ &\rightarrow s(s(0 + 0)) \end{aligned}$$

Example – Term Rewriting

- a set of rules, also called a term rewrite system (TRS)

$$0 + y \rightarrow y$$

$$0 \times y \rightarrow 0$$

$$s(x) + y \rightarrow s(x + y)$$

$$s(x) \times y \rightarrow y + (x \times y)$$

- 'compute' 1×2

$$\begin{aligned} s(0) \times s^2(0) &\rightarrow s^2(0) + (0 \times s^2(0)) \\ &\rightarrow s^2(0) + 0 \\ &\rightarrow s(s(0) + 0) \\ &\rightarrow s(s(0 + 0)) \\ &\rightarrow s^2(0) \end{aligned}$$

In Isabelle

```
datatype num = Zero | Succ num
```

```
notation Zero ("0")
```

```
notation Succ ("s'(_)'")
```

```
primrec
```

```
  add :: "num => num => num" (infixl "+" 65)
```

```
where
```

```
  "(0::num) + y = y" |
```

```
  "s(x)      + y = s(x + y)"
```

```
primrec
```

```
  mul :: "num => num => num" (infixl "×" 70)
```

```
where
```

```
  "(0::num) × y = 0" |
```

```
  "s(x)      × y = y + (x × y)"
```

Explanatory Notes

- 0 and + are overloaded, hence we need type constraint

Explanatory Notes

- 0 and + are overloaded, hence we need **type constraint**

Explanatory Notes

- `0` and `+` are overloaded, hence we need type constraint
- use `'` within syntax annotations to escape characters with special meaning, e.g., `'(` for an opening parenthesis (special meaning: start a group for pretty printing) or `'_` for an underscore (special meaning: argument placeholder)

Explanatory Notes

- `0` and `+` are overloaded, hence we need type constraint
- use `'` within **syntax annotations** to escape characters with special meaning, e.g., `'(` for an opening parenthesis (special meaning: start a group for pretty printing) or `'_` for an underscore (special meaning: argument placeholder)

Explanatory Notes

- `0` and `+` are overloaded, hence we need type constraint
- use `'` within syntax annotations to escape characters with special meaning, e.g., `'(` for an opening parenthesis (special meaning: start a group for pretty printing) or `'_` for an underscore (special meaning: argument placeholder)
- to get symbols like \times use Unicode Tokens (see next slide)

Explanatory Notes

- `0` and `+` are overloaded, hence we need type constraint
- use `'` within syntax annotations to escape characters with special meaning, e.g., `'(` for an opening parenthesis (special meaning: start a group for pretty printing) or `'_` for an underscore (special meaning: argument placeholder)
- to get symbols like \times use **Unicode Tokens** (see next slide)

Explanatory Notes

- `0` and `+` are overloaded, hence we need type constraint
- use `'` within syntax annotations to escape characters with special meaning, e.g., `'(` for an opening parenthesis (special meaning: start a group for pretty printing) or `'_` for an underscore (special meaning: argument placeholder)
- to get symbols like \times use Unicode Tokens (see next slide)
- we automatically get lemmas `add.simps` and `mul.simps`

Unicode Tokens

ASCII	Unicode Token	shown as	ASCII	Unicode Token	shown as
=>	<code>\<Rightarrow></code>	\Rightarrow	ALL	<code>\<forall></code>	\forall
-->	<code>\<longrightarrow></code>	\longrightarrow	EX	<code>\<exists></code>	\exists
==>	<code>\<Longrightarrow></code>	\Longrightarrow	&	<code>\<and></code>	\wedge
!!	<code>\<And></code>	\bigwedge		<code>\<or></code>	\vee
==	<code>\<equiv></code>	\equiv	~	<code>\<not></code>	\neg
~=	<code>\<noteq></code>	\neq	%	<code>\<lambda></code>	λ
:	<code>\<in></code>	\in	*	<code>\<times></code>	\times
~:	<code>\<notin></code>	\notin	o	<code>\<circ></code>	\circ
Un	<code>\<union></code>	\cup	[<code>\<lbrakk></code>	\llbracket
Int	<code>\<inter></code>	\cap]	<code>\<rbrakk></code>	\rrbracket
Union	<code>\<Union></code>	\bigcup	<=	<code>\<subseteq></code>	\subseteq
Inter	<code>\<Inter></code>	\bigcap	<	<code>\<subset></code>	\subset

Using Simplification Rules Automatically

```
lemma "s(s(0)) × s(s(0)) = s(s(s(s(0))))" by simp
```

Using Simplification Rules Automatically

```
lemma "s(s(0)) × s(s(0)) = s(s(s(s(0))))" by simp
```

Using Simplification Rules Explicitly

```
lemma "s(s(0)) × s(s(0)) = s(s(s(s(0))))"  
  unfolding add.simps mul.simps by (rule refl)
```

Modifying the Simpset

Modifying the Simpset

- simpset is set of simplification rules currently in use

Modifying the Simpset

- `simpset` is set of simplification rules currently in use

Modifying the Simpset

- simpset is set of simplification rules currently in use
- adding a lemma to the simpset

```
declare <theorem-name> [simp]
```


Modifying the Simpset

- simpset is set of simplification rules currently in use
- adding a lemma to the simpset
`declare <theorem-name> [simp]`
- deleting a lemma from the simpset
`declare <theorem-name> [simp del]`

Modifying the Simpset

- simpset is set of simplification rules currently in use
- adding a lemma to the simpset
`declare <theorem-name> [simp]`
- deleting a lemma from the simpset
`declare <theorem-name> [simp del]`

Example

```
declare add.simps [simp del]
lemma "0 + s(0) = s(0)" oops
```

A More Complete Grammar for Proofs

proof $\stackrel{\text{def}}{=}$ *prefix** **proof** *method*? *statement** **qed** *method*?
| *prefix** **by** *method* *method*?

prefix $\stackrel{\text{def}}{=}$ **apply** *method*
| **using** *fact**
| **unfolding** *fact**

statement $\stackrel{\text{def}}{=}$ **fix** *variables*
| **assume** *proposition*⁺
| (**from** *fact*⁺)? (**show** | **have**) *proposition* *proof*

proposition $\stackrel{\text{def}}{=}$ (*label*:)? "term"

fact $\stackrel{\text{def}}{=}$ *label*
| `term`

A Proof by Hand

```
lemma "s(s(0)) × s(s(0)) = s(s(s(s(0))))"
proof -
  have "s(s(0)) × s(s(0)) =
        s(s(0)) + s(0) × s(s(0))"
    unfolding mul.simps by (rule refl)
  from this have "s(s(0)) × s(s(0)) =
        s(s(0)) + (s(s(0)) + 0 × s(s(0)))"
    unfolding mul.simps .
  from this have "s(s(0)) × s(s(0)) =
        s(s(0)) + (s(s(0)) + 0)"
    unfolding mul.simps .
  from this show ?thesis unfolding add.simps .
qed
```

The simp Method – General Format

`simp` *<list of modifiers>*

The simp Method – General Format

`simp` *<list of modifiers>*

Modifiers

The simp Method – General Format

`simp` *<list of modifiers>*

Modifiers

- `add:` *<list of theorem names>*

The simp Method – General Format

`simp` *<list of modifiers>*

Modifiers

- `add`: *<list of theorem names>*
- `del`: *<list of theorem names>*

The simp Method – General Format

`simp` *<list of modifiers>*

Modifiers

- `add`: *<list of theorem names>*
- `del`: *<list of theorem names>*
- `only`: *<list of theorem names>*

The simp Method – General Format

`simp` *<list of modifiers>*

Modifiers

- `add`: *<list of theorem names>*
- `del`: *<list of theorem names>*
- `only`: *<list of theorem names>*

Example

```
lemma "s(s(0)) × s(s(0)) = s(s(s(s(0))))"  
  by (simp only: add.simps mul.simps)
```

A General Format for Stating Theorems

theorem $\stackrel{\text{def}}{=} \begin{array}{l} \textit{kind} \ \textit{goal} \\ | \\ \textit{kind} \ \textit{name} : \ \textit{goal} \\ | \\ \textit{kind} \ [\textit{attributes}] : \ \textit{goal} \\ | \\ \textit{kind} \ \textit{name}[\textit{attributes}] : \ \textit{goal} \end{array}$

kind $\stackrel{\text{def}}{=} \text{theorem} \mid \text{lemma} \mid \text{corollary}$

goal $\stackrel{\text{def}}{=} \begin{array}{l} (\text{fixes} \ \textit{variables})? \ (\text{assumes} \ \textit{prop}^+)? \ \text{shows} \ \textit{prop}^+ \\ | \\ \textit{prop}^+ \end{array}$

prop $\stackrel{\text{def}}{=} (\text{label}:)? \ \text{"term"}$

Example

```
lemma some_lemma[simp]:  
  fixes A :: "bool" (*"A" has type "bool"*)  
  assumes AnA: "A  $\wedge$  A" (*give the name "AnA"*)  
  shows "A"  
  using AnA by simp
```

Assumptions

Assumptions

- by default assumptions are used as simplification rules + assumptions are simplified themselves

```
lemma
  assumes "xs @ zs = ys @ xs"
    and "[] @ xs = [] @ []"
  shows "ys = zs"
  using assms by simp
```

Assumptions

- by default assumptions are used as simplification rules + assumptions are simplified themselves

```
lemma
  assumes "xs @ zs = ys @ xs"
    and "[] @ xs = [] @ []"
  shows "ys = zs"
  using assms by simp
```

- this can lead to nontermination

```
lemma
  assumes " $\forall x. f\ x = g\ (f\ (g\ x))$ "
  shows "f [] = f [] @ []"
  using assms by simp
```

The simp Method – More Modifiers

The simp Method – More Modifiers

- `(no_asm)` assumptions are ignored

The simp Method – More Modifiers

- `(no_asm)` assumptions are ignored
- `(no_asm_simps)` assumptions are not simplified themselves

The simp Method – More Modifiers

- `(no_asm)` assumptions are ignored
- `(no_asm_simps)` assumptions are not simplified themselves
- `(no_asm_use)` assumptions are simplified but not added to simpset

Tracing

Tracing

- `declare [[simp_trace]]`

Tracing

- `declare [[simp_trace]]`
- useful to get a feeling for simplification rules

Tracing

- `declare [[simp_trace]]`
- useful to get a feeling for simplification rules
- see which rules are applied

Tracing

- `declare [[simp_trace]]`
- useful to get a feeling for simplification rules
- see which rules are applied
- find out why simplification loops

Digression – Finding Theorems

Start Search

- either by keyboard shortcut (only Emacs) `Ctrl+C`, `Ctrl+F`, or
- entering the command `find_theorems`

Digression – Finding Theorems

Start Search

- either by keyboard shortcut (only Emacs) `Ctrl+C`, `Ctrl+F`, or
- entering the command `find_theorems`

Search Criteria

Digression – Finding Theorems

Start Search

- either by keyboard shortcut (only Emacs) `Ctrl+C`, `Ctrl+F`, or
- entering the command `find_theorems`

Search Criteria

- a number in parenthesis specifies number of shown results

Digression – Finding Theorems

Start Search

- either by keyboard shortcut (only Emacs) `Ctrl+C`, `Ctrl+F`, or
- entering the command `find_theorems`

Search Criteria

- a number in parenthesis specifies number of shown results
- a pattern in quotes specifies the term to be searched for

Digression – Finding Theorems

Start Search

- either by keyboard shortcut (only Emacs) `Ctrl+C`, `Ctrl+F`, or
- entering the command `find_theorems`

Search Criteria

- a number in parenthesis specifies number of shown results
- a pattern in quotes specifies the term to be searched for
- a pattern may contain wild cards `'_'`, and type constraints

Digression – Finding Theorems

Start Search

- either by keyboard shortcut (only Emacs) `Ctrl+C`, `Ctrl+F`, or
- entering the command `find_theorems`

Search Criteria

- a number in parenthesis specifies number of shown results
- a pattern in quotes specifies the term to be searched for
- a pattern may contain wild cards `'_'`, and type constraints
- precede a pattern by `simp:` to only search for theorems that could simplify the specified term at the root

Digression – Finding Theorems

Start Search

- either by keyboard shortcut (only Emacs) `Ctrl+C`, `Ctrl+F`, or
- entering the command `find_theorems`

Search Criteria

- a number in parenthesis specifies number of shown results
- a pattern in quotes specifies the term to be searched for
- a pattern may contain wild cards `'_'`, and type constraints
- precede a pattern by `simp:` to only search for theorems that could simplify the specified term at the root
- to search for part of a name use `name: "some string"`

Digression – Finding Theorems

Start Search

- either by keyboard shortcut (only Emacs) `Ctrl+C`, `Ctrl+F`, or
- entering the command `find_theorems`

Search Criteria

- a number in parenthesis specifies number of shown results
- a pattern in quotes specifies the term to be searched for
- a pattern may contain wild cards `'_'`, and type constraints
- precede a pattern by `simp:` to only search for theorems that could simplify the specified term at the root
- to search for part of a name use `name: "<some string>"`
- negate a search criterion by prefixing a minus, e.g., `-name:`

Function Definitions

Example

```
fun fib :: "nat => nat" where
  "fib 0 = Suc 0" |
  "fib (Suc 0) = Suc 0" |
  "fib (Suc (Suc n)) = fib n + fib (Suc n)"
```

Example

```
fun fib :: "nat => nat" where
  "fib 0 = Suc 0" |
  "fib (Suc 0) = Suc 0" |
  "fib (Suc (Suc n)) = fib n + fib (Suc n)"
```

Lemma

$0 < \text{fib } n$

Abbreviations

Abbreviations

- `this`: the previous proposition proved or assumed

Abbreviations

- `this`: the previous proposition proved or assumed
- `then: from this`

Abbreviations

- `this`: the previous proposition proved or assumed
- `then: from this`
- `hence: then have`

Abbreviations

- `this`: the previous proposition proved or assumed
- `then: from this`
- `hence: then have`
- `thus: then show`

Abbreviations

- **this**: the previous proposition proved or assumed
- **then: from this**
- **hence: then have**
- **thus: then show**
- **with *<facts>*: from *<facts>* this**

The Command fun

The Command fun

- in principle arbitrary pattern matching on left-hand sides

The Command fun

- in principle arbitrary pattern matching on left-hand sides
- patterns are matched top to bottom

The Command `fun`

- in principle arbitrary pattern matching on left-hand sides
- patterns are matched top to bottom
- `fun` tries to prove termination automatically (current method: lexicographic orders)

The Command `fun`

- in principle arbitrary pattern matching on left-hand sides
- patterns are matched top to bottom
- `fun` tries to prove termination automatically (current method: lexicographic orders)
- use `function` instead of `fun` to provide a manual termination prove

The Command `fun`

- in principle arbitrary pattern matching on left-hand sides
- patterns are matched top to bottom
- `fun` tries to prove termination automatically (current method: lexicographic orders)
- use `function` instead of `fun` to provide a manual termination prove
- for further information: `isabelle doc functions`

Calculational Reasoning

Additional Commands

Additional Commands

- **also**: to apply transitivity automatically

Additional Commands

- **also**: to apply transitivity automatically
- **finally**: to reconsider first left-hand side

Additional Commands

- **also**: to apply transitivity automatically
- **finally**: to reconsider first left-hand side
- **...:** to abbreviate previous right-hand side

An Example Proof (Base Case)

```
primrec sum :: "nat => nat" where
  "sum 0          = 0" |
  "sum (Suc n) = Suc n + sum n"

lemma "sum n = (n * (Suc n)) div (Suc (Suc 0))"
proof (induct n)
  case 0 show ?case by simp
next
```

An Example Proof (Step Case)

```
case (Suc n)
hence IH: "sum n = (n*(Suc n)) div (Suc(Suc 0))" .
have "sum(Suc n) = Suc n + sum n" by simp
also
  have "... = Suc n + ((n*(Suc n)) div (Suc(Suc 0)))"
    unfolding IH by simp
also have "... = ((Suc(Suc 0)*Suc n) div Suc(Suc 0)) +
              ((n*(Suc n)) div Suc(Suc 0))" by arith
also have "... = (Suc(Suc 0)*Suc n + n*(Suc n)) div
              Suc(Suc 0)" by arith
also
  have "... = ((Suc(Suc 0) + n)*Suc n) div Suc(Suc 0)"
    unfolding add_mult_distrib by simp
also have "... = (Suc(Suc n) * Suc n) div Suc(Suc 0)"
  by simp
finally show ?case by simp
qed
```

Remarks

Remarks

- cases are named by the corresponding `datatype` constructors

Remarks

- cases are named by the corresponding **datatype** constructors
- **?case** is an abbreviation installed for the current goal in each case of an induction proof

Remarks

- cases are named by the corresponding **datatype** constructors
- **?case** is an abbreviation installed for the current goal in each case of an induction proof
- **case 0** sets up the assumption corresponding to the base case (i.e., no assumption at all)

Remarks

- cases are named by the corresponding **datatype** constructors
- **?case** is an abbreviation installed for the current goal in each case of an induction proof
- **case 0** sets up the assumption corresponding to the base case (i.e., no assumption at all)
- **case (Suc n)** sets up the corresponding assumption

```
fix n assume "sum n = (n*Suc n) div Suc(Suc 0)"
```

Remarks

- cases are named by the corresponding **datatype** constructors
 - **?case** is an abbreviation installed for the current goal in each case of an induction proof
 - **case 0** sets up the assumption corresponding to the base case (i.e., no assumption at all)
 - **case (Suc n)** sets up the corresponding assumption
- ```
fix n assume "sum n = (n*Suc n) div Suc(Suc 0)"
```
- **arith** is a decision procedure for Presburger Arithmetic

## Remarks

- cases are named by the corresponding **datatype** constructors
  - **?case** is an abbreviation installed for the current goal in each case of an induction proof
  - **case 0** sets up the assumption corresponding to the base case (i.e., no assumption at all)
  - **case (Suc n)** sets up the corresponding assumption
- ```
fix n assume "sum n = (n*Suc n) div Suc(Suc 0)"
```
- **arith** is a decision procedure for **Presburger Arithmetic**

Remarks

- cases are named by the corresponding **datatype** constructors
- **?case** is an abbreviation installed for the current goal in each case of an induction proof
- **case 0** sets up the assumption corresponding to the base case (i.e., no assumption at all)
- **case (Suc n)** sets up the corresponding assumption

```
fix n assume "sum n = (n*Suc n) div Suc(Suc 0)"
```
- **arith** is a decision procedure for Presburger Arithmetic
- **.** abbreviates **by assumption**

Exercises

<http://isabelle.in.tum.de/exercises/arith/powSum/ex.pdf>