Experiments in Verification

SS 2011

Christian Sternagel

Computational Logic
Institute of Computer Science
University of Innsbruck

March 25, 2011

Today’s Topics

- Simplification
- Function Definitions
- Calculational Reasoning

Simplification

Example – Term Rewriting

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

$$0 + y \to y \qquad\qquad 0 \times y \to 0$$
$$s(x) + y \to s(x + y) \qquad\qquad s(x) \times y \to y + (x \times y)$$

- 'compute' 1×2

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

Example – Term Rewriting

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

$$\begin{aligned} 0 + y &\to y & \qquad 0 \times y &\to 0 \\ \mathsf{s}(x) + y &\to \mathsf{s}(x + y) & \qquad \mathsf{s}(x) \times y &\to y + (x \times y) \end{aligned}$$

- 'compute' 1×2

$$\mathsf{s}(0) \times \mathsf{s}^2(0) \quad \to \mathsf{s}^2(0) + (0 \times \mathsf{s}^2(0))$$

Example – Term Rewriting

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

$$0 + y \to y \qquad\qquad 0 \times y \to 0$$
$$\mathsf{s}(x) + y \to \mathsf{s}(x + y) \qquad \mathsf{s}(x) \times y \to y + (x \times y)$$

- 'compute' 1×2

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

Example – Term Rewriting

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

$$0 + y \to y \qquad\qquad 0 \times y \to 0$$
$$\mathsf{s}(x) + y \to \mathsf{s}(x + y) \qquad\qquad \mathsf{s}(x) \times y \to y + (x \times y)$$

- 'compute' 1×2

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

Example – Term Rewriting

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

$$0 + y \to y \qquad\qquad 0 \times y \to 0$$
$$\mathsf{s}(x) + y \to \mathsf{s}(x + y) \qquad\qquad \mathsf{s}(x) \times y \to y + (x \times y)$$

- 'compute' 1×2

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

Example – Term Rewriting

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

$$0 + y \to y \qquad\qquad 0 \times y \to 0$$
$$\mathsf{s}(x) + y \to \mathsf{s}(x + y) \qquad\qquad \mathsf{s}(x) \times y \to y + (x \times y)$$

- 'compute' 1×2

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

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 (Emacs)

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

Unicode Tokens (Emacs)

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

- Emacs: `Proof-General` → `Quick Options` → `Display` → `Unicode Tokens`

Unicode Tokens (Emacs)

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

- Emacs: Proof-General $\rightarrow$ Quick Options $\rightarrow$ Display $\rightarrow$ Unicode Tokens
- jEdit: several predefined abbreviations achieve a similar effect

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

$$
\begin{aligned}
\textit{proof} &\overset{\text{def}}{=} \textit{prefix}^* \ \mathtt{proof}\ \textit{method}^? \ \textit{statement}^* \ \mathtt{qed}\ \textit{method}^? \\
&\mid \ \textit{prefix}^* \ \mathtt{by}\ \textit{method}\ \textit{method}^? \\[1em]
\textit{prefix} &\overset{\text{def}}{=} \mathtt{apply}\ \textit{method} \\
&\mid \ \mathtt{using}\ \textit{fact}^* \\
&\mid \ \mathtt{unfolding}\ \textit{fact}^* \\[1em]
\textit{statement} &\overset{\text{def}}{=} \mathtt{fix}\ \textit{variables} \\
&\mid \ \mathtt{assume}\ \textit{proposition}^+ \\
&\mid \ (\mathtt{from}\ \textit{fact}^+)^? \ (\mathtt{show} \mid \mathtt{have})\ \textit{proposition}\ \textit{proof} \\[1em]
\textit{proposition} &\overset{\text{def}}{=} (\textit{label}\,\mathtt{:})^? \ \mathtt{"}\textit{term}\mathtt{"} \\[1em]
\textit{fact} &\overset{\text{def}}{=} \textit{label} \\
&\mid \ \mathtt{`}\textit{term}\mathtt{`}
\end{aligned}
$$

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

$$
\begin{array}{rcl}
\textit{theorem} & \stackrel{\text{def}}{=} & \textit{kind goal} \\
& \mid & \textit{kind name}:\ \textit{goal} \\
& \mid & \textit{kind}\ \texttt{[}\textit{attributes}\texttt{]}\texttt{:}\ \textit{goal} \\
& \mid & \textit{kind name}\,\texttt{[}\textit{attributes}\texttt{]}\texttt{:}\ \textit{goal} \\
\\
\textit{kind} & \stackrel{\text{def}}{=} & \texttt{theorem} \mid \texttt{lemma} \mid \texttt{corollary} \\
\\
\textit{goal} & \stackrel{\text{def}}{=} & (\texttt{fixes}\ \textit{variables})^?\ (\texttt{assumes}\ \textit{prop}^+)^?\ \texttt{shows}\ \textit{prop}^+ \\
& \mid & \textit{prop}^+ \\
\\
\textit{prop} & \stackrel{\text{def}}{=} & (\textit{label}\,\texttt{:})^?\ \texttt{"}\textit{term}\texttt{"}
\end{array}
$$

Example

```
lemma some_lemma[simp]:
  fixes A :: "bool" (*"A" has type "bool"*)
  assumes AnA: "A ∧ 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 "∀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

- set `Isabelle` $\longrightarrow$ `Settings` $\longrightarrow$ `Tracing` $\longrightarrow$ `Trace Simplifier`

Tracing

- set `Isabelle` $\longrightarrow$ `Settings` $\longrightarrow$ `Tracing` $\longrightarrow$ `Trace Simplifier`
- useful to get a feeling for simplification rules

Tracing

- set `Isabelle` $\longrightarrow$ `Settings` $\longrightarrow$ `Tracing` $\longrightarrow$ `Trace Simplifier`
- useful to get a feeling for simplification rules
- see which rules are applied

Tracing

- set `Isabelle` → `Settings` → `Tracing` → `Trace Simplifier`
- 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 < 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