# Teil XXIV

# *Dokumentenerzeugung*

# Dokumentenerzeugung

Isabelle kann Theorien mit LaTeX schön setzen.

Dazu muss man eine *Sitzung* definieren. Am einfachsten geht das mit `isabelle mkroot -d name`.

Die Datei `ROOT` führt alle verwendeten Theorien auf.
Die Datei `document/root.tex` enthält den LaTeX-Rahmen.

Man lässt Isabelle mit
`isabelle build -D .`
die Theorien verarbeiten und die PDF-Dateien erzeugen.

# LaTeX in Isabelle-Theorien

Normaler Text (einschließlich LaTeX-Makros) kann mittels

**txt** *{* bla bla *}*

eingefügt werden.

Innerhalb eines **proof**-Blocks heißt der Befehl **txt**.

Kommentare (*(* bla bla *)*) erscheinen *nicht* im Dokument!

# Strukturierung

Statt LaTeX-Befehle wie `\section`, `\subsection` etc. in **text**-Blöcke einzubauen kann man die entsprechenden Isabelle-Befehle

- **chapter**
- **section**
- **subsection**
- **subsubsection**

oder die Varianten für **proof**-Blöcke, **sect**, **subsect** und **subsubsect**, verwenden.

# Anti-Quotations

Man kann Ausdrücke verschiedener Art von Isabelle in das Dokument einfügen lassen:
Nach

```
definition N :: nat where "N = 0"
theorem great_result: "N = N * P" unfolding N_def by simp
```

wird aus

```
text {*
  After defining @{thm N_def} we were finally able
  to prove @{thm great_result}.
*}
```

in der Dokumentausgabe

After defining $N = 0$ we were finally able to prove $N = N * ?P$.

# Weitere Anti-Quotations

Neben `@{thm ...}` sind noch nützlich:

- `@{theory ...}` verweist auf einen (importierten) Theorie-Namen,
- `@{term ...}` setzt einen Term,
- `@{term_type ...}` ebenso, aber mit Typ,
- `@{typ ...}` setzt einen Typ,
- `@{value ...}` evaluiert einen Term und zeigt das Ergebnis,
- `@{text ...}` setzt beliebigen Text im Isabelle-Stil.

Während `@{thm ...}` garantiert, dass nur bewiesenes gedruckt wird, überprüfen die anderen nur die Typisierung, und mit `@{text ...}` lässt sich alles ausgeben.

Beim Ausgeben von Lemmas ist oft `@{thm great_result[no_vars]}` schöner als `@{thm great_result}`.

# Reihenfolge der Dateien

Standardmäßig enthält `document/root.tex` den Befehl `\input{session}` und `session.tex` (von Isabelle erstellt) enthält für jede Theorie *`foo`* eine Zeile `\input{Example.tex}`.

Man kann natürlich auch die Theorie-Dateien direkt in `document/root.tex` einbinden, etwa um dazwischen noch Text wie Kapitelüberschriften oder Einleitungen zu setzten.

Auch will man vielleicht in der Einleitung schon auf alle Definitionen und Ergebnisse vorgreifen. Dazu erstellt man z.B. eine Theorie *`Introduction`* und bindet diese in `document/root.tex` am Anfang ein.

Für Theorien, die in `ROOT` mit der Option `document = false` versehen sind, werden nicht in das Dokument aufgenommen (die trotzdem erzeugte `.tex`-Datei ist leer).

# Mehr Informationen

zu mehr Anti-Quotations siehe das Isabelle Referenz-Handbuch (`isabelle doc isar-ref`).

Für mehr LaTeX-Spielereien wie z.B. die Ausgabe

$$\frac{P\ 0 \qquad \bigwedge nat.\ \dfrac{P\ nat}{P\ (Suc\ nat)}}{P\ nat}$$

für

```
text {* \begin{center}
  @{thm[mode=Rule] nat.induct[no_vars]}
\end{center} *}
```

siehe `isabelle doc sugar`.

Karlsruher Institut für Technologie

# Teil XXV

# *Erzeugung von ausführbarem Code*

# Code exportieren

Isabelle kann Formalisierungen nach **SML**, **OCaml**, **Haskell** bzw. **Scala** exportieren.

$\Longrightarrow$ Dadurch sind verifizierte *ausführbare* Programme möglich.

- Jede HOL-Funktion wird in eine entsprechende Funktion der Zielsprache übersetzt.
- Jeder HOL-Typ wird ein entsprechenden Typ der Zielsprache übersetzt.
- **Basis:** Code-Gleichungen

Code-Erzeugung mit Befehl:

**export_code** *f* **in** *Sprache* **module_name** *Modul* **file** *Datei*

Die definierenden HOL-Gleichungen von *f* werden 1:1 in die Zielsprache übersetzt.

# Nicht ausführbare Definitionen

Nicht alle HOL-Funktionen können direkt übersetzt werden.

## Beispiel

Wie kann die Funktion

```
doubled xs =
  (if (∃ys. xs = ys @ ys) then Some (THE ys. xs = ys @ ys) else None)
```

übersetzt werden?

# Nicht ausführbare Definitionen

Nicht alle HOL-Funktionen können direkt übersetzt werden.

## Beispiel

Wie kann die Funktion

```
doubled xs =
  (if (∃ys. xs = ys @ ys) then Some (THE ys. xs = ys @ ys) else None)
```

übersetzt werden?

**Problem:** Existenzquantor nur für enum-Typen ausführbar.

# Nicht ausführbare Definitionen

Nicht alle HOL-Funktionen können direkt übersetzt werden.

## Beispiel

Wie kann die Funktion

```
doubled xs =
  (if (∃ys. xs = ys @ ys) then Some (THE ys. xs = ys @ ys) else None)
```

übersetzt werden?

**Problem:** Existenzquantor nur für enum-Typen ausführbar.

**Lösung:** Beweise alternative Code-Gleichung:

```
lemma doubled_code [code]: "doubled xs =
  (let ys = take (length xs div 2) xs in
    (if (xs = ys @ ys) then Some ys else None)"
```

# Code-Gleichungen

Eine Gleichung kann als Code-Gleichung für `f` verwendet werden, wenn

- `f` das oberste (und einzige) Funktionssymbol im linken Term ist,
- Patternmatching auf die Parameter von `f` nur via Datentyp-Konstruktoren erfolgt, und
- für alle Funktionssymbole auf der rechten Seite Code-Gleichungen existieren.

Insbesondere ist es nicht (direkt) möglich “partielle” Code-Gleichungen anzugeben.

Späteres hinzufügen einer Gleichung als Code-Gleichung mit

**declare** *`lemma [code]`*

möglich.

# Beispiel

Siehe Formalisierung

# Diverses

- Das Kommando

  **value** *[code]* `"t"`

  übersetzt den Term `t` und wertet ihn aus.

- **eval** ist eine Beweis-Taktik, welche versucht, das aktuelle Ziel durch "ausrechnen" (Brute-Force) zu zeigen.

- **code_thms** `f` zeigt alle registrierten Code-Gleichungen an, die zur Auswertung von `f` benötigt werden.

- **print_codesetup** zeigt *alle* registrierten Code-Gleichungen an.

Karlsruher Institut für Technologie

# Teil XXVI

# *Kurz angeschnitten – Weitere Möglichkeiten des Codegenerators*

# Datatype Refinement

Siehe Beispiel in der Formalisierung.

**Stichwort: code_datatype**

$\Longrightarrow$ Pattern-Matching auf Code-Datentyp Konstruktor jetzt möglich.

# Datatype Refinement mit Invarianten

In Zusammenhang mit **lifting** und **transfer** transparent (Out-Of-The-Box).

*Manuell:* Auch möglich, dann mit `[code abstype]` und `[code abstract]` arbeiten.

$\Longrightarrow$ siehe z.B.: `isabelle doc codegen`

# Code-Präprozessor

Vor Anwendung der Code-Gleichungen werden diese vom Code-Präprozessor bearbeitet.

- Rewrite-System mit ähnlicher Mächtigkeit wie Simplifier
- Attribut **code_unfold** verwenden, um Gleichungen zu registrieren
- **print_codeproc** zeigt das Präprozessor-Setup an

Karlsruher Institut für Technologie

# Teil XXVII

# *Lifting und Transfer*

# Rückblick: Eigene Typen in HOL definieren

```
typedef 'a ne = "{xs :: 'a list . xs ≠ []}"
  by (rule exI[where x = "[undefined]"], simp)

definition singleton :: "'a ⇒ 'a ne"
  where "singleton x = Abs_ne [x]"
definition append :: "'a ne ⇒ 'a ne ⇒ 'a ne"
  where "append l1 l2 = Abs_ne (Rep_ne l1 @ Rep_ne l2)"
definition head :: "'a ne ⇒ 'a"
  where "head l = hd (Rep_ne l)"

lemma "head (append l1 l2) = head l1"
  unfolding head_def append_def
  apply (subst Abs_ne_inverse)
  using Rep_ne[of l1] apply simp
  using Rep_ne[of l1] apply simp
  done
```

# Heute: lifting und transfer

Das Beweisen mit den Abstraktions- und Represenentationsfunktionen ist mühsahm und unnatürlich: So wird die erhaltung einer Invariante beim Verwenden der Funktion bewiesen, und nicht beim definieren (siehe `tail`).

Die Isabelle-Pakete Lifting und Transfer erlauben es, Funktionen einmal bei der Definition als „korrekt“ zu beweisen und Lemmas mit einem Methodenaufruf in die Welt der zugrundeliegenden Repräsentation zu übertragen und dann dort zu beweisen.

# Lifting und Transfer verwenden

1. Typ registrieren:

   **setup_lifting** *type_definition_typname*

# Lifting und Transfer verwenden

1. Typ registrieren:

   **setup_lifting** *type_definition_typname*

2. Definitionen liften:

   **lift_definition** *name :: type* **is** *"ausdruck"*

   *Beweis*

   wobei *ausdruck* die Definition von *name* auf den konkreten Datetyp ist und der *Beweis* beweist dass die Typ-Invarianten respektiet werden.

# Lifting und Transfer verwenden

1. Typ registrieren:

   **setup_lifting** *type_definition_typname*
2. Definitionen liften:

   **lift_definition** *name :: type* **is** *"ausdruck"*

   *Beweis*

   wobei *ausdruck* die Definition von *name* auf den konkreten Datetyp ist und der *Beweis* beweist dass die Typ-Invarianten respektiet werden.
3. Aussagen auf die konkreten Typen übertragen: **apply** *transfer*
   Ersetzt das aktuelle Ziel durch ein gleichwertiges auf dem konkreten Datentyp, indem die per **lift_definition** definierten Funktionen durch ihre konkrete Definition ersetzt werden.

# Beispiel: Sortierte Listen

Typ registrieren:

```
typedef slist = "{xs. sorted xs}" morphisms list_of as_sorted
  by (rule_tac x="[]" in exI) simp

setup_lifting type_definition_slist
```

# Beispiel: Sortierte Listen

Typ registrieren:

```
typedef slist = "{xs. sorted xs}" morphisms list_of as_sorted
  by (rule_tac x="[]" in exI) simp

setup_lifting type_definition_slist
```

Definitionen:

```
lift_definition Singleton :: "nat ⇒ slist" is "λx. [x]" by simp
lift_definition hd :: "slist ⇒ nat" is "List.hd" ..
lift_definition take :: "nat ⇒ slist ⇒ slist" is "List.take" ..
lift_definition smerge :: "slist ⇒ slist ⇒ slist" is "Scratch.merge" by
(rule sorted_merge_sorted)
```

# Beispiel: Sortierte Listen

Lemmas zu Definitionen auf dem abstrakten Typ:

**lemma** *set_of_Singleton [simp]: "set_of (Singleton x) = {x}"*
  Aktuelles Ziel: *set_of (Singleton x) = {x}*
**apply** *transfer*
  Aktuelles Ziel: *⋀x. set [x] = {x}*
**apply** *simp*
  Aktuelles Ziel: *No subgoals!*
**done**

oder gleich

**by** *transfer simp*

# Beispiel: Sortierte Listen

Lemmas können Invarianten nutzen:

**lemma** *"list_of xs = a#b#ys* $\Longrightarrow$ *a* $\leq$ *b"*

Aktuelles Ziel: *list_of xs = a # b # ys* $\Longrightarrow$ *a* $\leq$ *b*

**apply** *transfer*

Aktuelles Ziel: $\bigwedge$*xs a b ys.* $[\![$*sorted xs; xs = a # b # ys*$]\!]$ $\Longrightarrow$ *a* $\leq$ *b*

**apply** *simp*

Aktuelles Ziel: *No subgoals!*

**done**

# Beispiel: Sortierte Listen

Lemmas mit rein abstrakte Definitionen:

**definition** *insert :: "nat ⇒ slist ⇒ slist"*
  **where** *"insert x xs = smerge xs (Singleton x)"*

**lemma** *set_of_insert [simp]: "x ∈ set_of (insert x xs)"*

# Beispiel: Sortierte Listen

Lemmas mit rein abstrakte Definitionen:

**definition** *insert :: "nat ⇒ slist ⇒ slist"*
  **where** *"insert x xs = smerge xs (Singleton x)"*

**lemma** *set_of_insert [simp]: "x ∈ set_of (insert x xs)"*

Erster Versuch:

  **apply** *transfer*

Hier bringt *transfer* einen nicht weiter!

# Beispiel: Sortierte Listen

Lemmas mit rein abstrakte Definitionen:

**definition** *insert :: "nat ⇒ slist ⇒ slist"*
  **where** *"insert x xs = smerge xs (Singleton x)"*

**lemma** *set_of_insert [simp]: "x ∈ set_of (insert x xs)"*

Erster Versuch:

  **apply** *transfer*

Hier bringt *transfer* einen nicht weiter!

Zweiter Versuch:

  **unfolding** *insert_def* **by** *transfer simp*

# Beispiel: Sortierte Listen

Lemmas mit rein abstrakte Definitionen:

**definition** *insert :: "nat ⇒ slist ⇒ slist"*
  **where** *"insert x xs = smerge xs (Singleton x)"*

**lemma** *set_of_insert [simp]: "x ∈ set_of (insert x xs)"*

Erster Versuch:

  **apply** *transfer*

Hier bringt *transfer* einen nicht weiter!

Zweiter Versuch:

  **unfolding** *insert_def* **by** *transfer simp*

Schöner ist:

**lemma** *set_of_smerge: "set_of (smerge xs ys) = set_of xs ∪ set_of ys"*
  **by** *transfer simp*

und dann

  **unfolding** *insert_def* **by** *transfer (simp add: set_of_smerge)*

# Beispiel: Sortierte Listen

Lifting arbeitet gut mit dem Code-Generator zusammen: Es registriert *as_sorted* als Konstruktor für den Typ *slist* und definierte alle Operationen darauf. Man kann keine Code-Gleichung angeben die mittels *as_sorted x* ein Wert vom Typ *slist* konstruiert, ohne bewiesen zu haben, dass *sorted x* gilt.

```
export_code insert hd take list_of set_of
  in Haskell
  file "-"
```