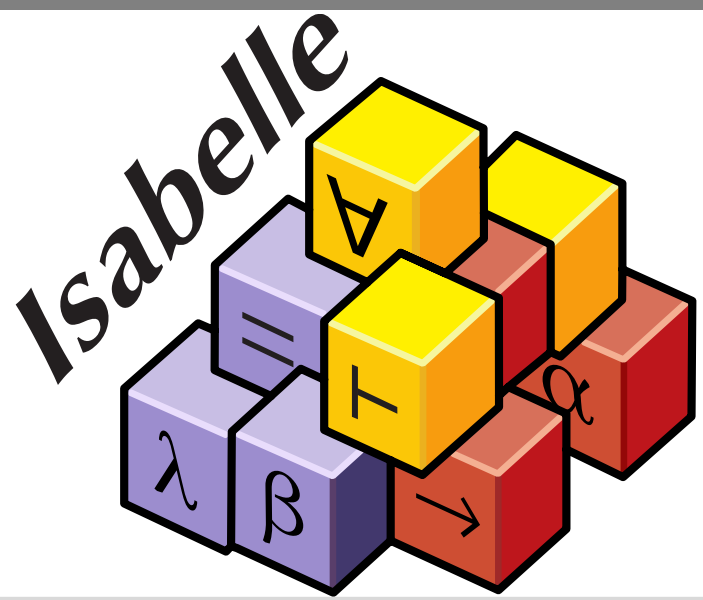
Karlsruher Institut für Technologie

Theorembeweiserpraktikum

Anwendungen in der Sprachtechnologie

LEHRSTUHL PROGRAMMIERPARADIGMEN



Karlsruher Institut für Technologie

Teil XXVIII

Typedef

Eigene Typen in HOL definieren

In Isabelle/HOL können eigene Typen definiert werden. Dazu benötigt man

- eine Teilmenge eines existierenden Typs sowie
- ein Beweis, dass diese Teilmenge nicht leer ist.

(Leere Typen würden HOL inkonsistent machen, d.h. man könnte `False` beweisen.)

Eigene Typen in Isabelle/HOL definieren: typedef

Syntax

typedef *typname* = *"Menge"* **morphisms** *rep_fun abs_fun* **by** *proof*

- *typname* ist der Name des neuen Typs. Hier dürfen auch Typvariablen verwendet werden (*('a, 'b) typname*).
- *Menge* ist ein Ausdruck vom Typ *irgendwas set*.
- Morphismen konvertieren zwischen der Menge und dem neuen Typ: *rep_fun :: typname ⇒ irgendwas* und *abs_fun :: irgendwas ⇒ typname*
- Default-Morphismennamen: *Rep_typname* und *Abs_typname*.
- Das Beweisziel ist $\exists x.\ x \in Menge$.
- Erzeugt (u. a. und v. a.) diese Lemmas:

```
rep_fun:          rep_fun ?x ∈ Menge
rep_fun_inverse:  abs_fun (rep_fun ?x) = ?x
rep_abs_inverse:  ?y ∈ Menge ⟹ rep_fun (abs_fun ?y) = ?y
```

Beispiel: Nicht-Leere Liste

Wir erstellen einen Typ für nicht-leere Listen und beginnen mit der Typ-Definition:

Beispiel: Nicht-Leere Liste

Wir erstellen einen Typ für nicht-leere Listen und beginnen mit der Typ-Definition:

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

Beispiel: Nicht-Leere Liste

Wir erstellen einen Typ für nicht-leere Listen und beginnen mit der Typ-Definition:

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

Weiter ein paar Funktionen auf nicht-leeren Listen:

```
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)"

definition tail :: "'a ne ⇒ 'a ne"
  where "tail l = Abs_ne (tl (Rep_ne l))"
```

Beispiel: Lemmas zu Nicht-Leeren Liste

Bei Append kommt der Head der Liste immer von der linken Liste (für allgemeine Listen nicht wahr!):

```
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
```

Beispiel: Mehr Lemmas zu Nicht-Leeren Liste

Head und Tail ergeben wieder die gesamte Liste:

```
lemma "append (singleton (head l)) (tail l) = l"
  unfolding head_def append_def singleton_def tail_def
  apply (subst Abs_ne_inverse)
  apply simp
  apply (subst Abs_ne_inverse)
  defer
  using Rep_ne[of l]
  apply simp
  apply (rule Rep_ne_inverse)
  apply simp
  oops
```

Beispiel: Mehr Lemmas zu Nicht-Leeren Liste

Head und Tail ergeben wieder die gesamte Liste:

```
lemma "append (singleton (head l)) (tail l) = l"
  unfolding head_def append_def singleton_def tail_def
  apply (subst Abs_ne_inverse)
  apply simp
  apply (subst Abs_ne_inverse)
  defer
  using Rep_ne[of l]
  apply simp
  apply (rule Rep_ne_inverse)
  apply simp
  oops
```

Problem: Das Lemma ist „eigentlich“ richtig, aber `tail [a]` ist undefiniert, da keine nicht-leere Liste.

Beispiel: Richtige Lemmas zu Nicht-Leeren Liste

Tail muss eine „normale“ Liste zurückgeben:

```
definition tail' :: "'a ne ⇒ 'a list"
  where "tail' l = tl (Rep_ne l)"

definition append' :: "'a ne ⇒ 'a list ⇒ 'a ne"
  where "append' l1 l2 = Abs_ne (Rep_ne l1 @ l2)"

lemma "append' (singleton (head l)) (tail' l) = l"
  unfolding head_def append'_def singleton_def tail'_def
  apply (subst Abs_ne_inverse, simp)
  using Rep_ne[of l, simplified]
  apply simp
  apply (rule Rep_ne_inverse)
  done
```

Karlsruher Institut für Technologie

Teil XXIX

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 Repräsentationsfunktionen ist mühsam 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 Datentyp ist und der *Beweis* beweist dass die Typ-Invarianten respektiert 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 Datentyp ist und der *Beweis* beweist dass die Typ-Invarianten respektiert 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 exI[where x = "[]"]) simp

setup_lifting type_definition_slist
```

Beispiel: Sortierte Listen

Typ registrieren:

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

setup_lifting type_definition_slist
```

Definitionen:

```
lift_definition Singleton :: "nat ⇒ slist" is "λx. [x]" by simp
lift_definition set_of :: "slist ⇒ nat set" is "List.set" .
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 abstrakten 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 abstrakten 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 abstrakten 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 abstrakten 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** *(simp add: set_of_smerge)*

Karlsruher Institut für Technologie

Teil XXX

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.

Datatype Refinement mit Invarianten

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 take list_of set_of*
  in *Haskell*

Manuell: Auch möglich, dann mit **code_datatype**, `[code abstype]` und `[code abstract]` arbeiten.
$\Longrightarrow$ siehe `isabelle doc codegen`.

Code-Präprozessor

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

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