

Formale Grundlagen der Informatik 3 –

*3. Spezifizieren, Beweisen und Testen*

Christoph Walther  
TU Darmstadt

# 1 Spezifikationen

## 1.1 Das Kunden-Produzenten Modell

- Kunde und Produzent schließen Vertrag
- Vertragsgegenstand: *Leistungsbeschreibung*
  - Was erhält der Kunde: Produkt
  - Was erhält der Produzent: Entlohnung
  - Geschäftsbedingungen: ...
- Hier von Interesse: *Produktbeschreibung* (= *Spezifikation*)
- *Informell*: “Eine Geldanlage mit hoher Rendite und geringem Verlustrisiko”
- *Halbformal*: Pläne und natürlichsprachliche Beschreibungen
- *Formal*: ?

### Beispiel Hausbau:

- *Formal*: Bauplan/-modell des Architekten,
- *Informell*: Teile der Ausführung (“Anstrich in Pastellfarben”)

## Beispiel Informatik:

- *Informell*: Ein Programm, das feststellt, ob eine Folge von ASCII Zeichen ein syntaktisch korrektes JAVA Programm ist (Parser)
- *Informell*: Ein Programm, das eine Liste natürlicher Zahlen sortiert.

## Wie formale Spezifikation?

### Ansatz:

- (1) Programm = Menge von Datentypen und Prozeduren, d.h. wir sprechen hier nur über Prozeduren, die auf bestimmten Datentypen ausgeführt werden
- (2) Prozeduren berechnen Funktionen:
  - (a) `function fact(x : nat) : nat <= ...` berechnet die Fakultätsfunktion
  - (b) `function sort(k : list[nat]) : list[nat] <= ...` sortiert eine Liste natürlicher Zahlen

**Also:** Kunde bestellt *Prozeduren* (etwa mit Namen `fact` und `sort`), die (bestimmte) *Funktionen berechnen* und auf (bestimmten) Datenstrukturen arbeiten (formal: *Signatur* und *Datentypen*)

**Anmerkung:** Es gibt auch *nicht-berechenbare Funktionen*  
=> Vorlesung *Berechenbarkeitstheorie*

Was sollen diese Funktionen leisten, d.h. *welche Eigenschaften* haben diese *Funktionen*?

**Formal:** Spezifiziere Eigenschaften von Funktionen in einer logischen, d.h. formalen Sprache

=> *Spezifikationsprache*

**Beispiel:** Sprache der Prädikatenlogik 1. Stufe

**Spezifikation** fact:

(1)  $\forall x, y: nat. 1 \leq y \leq x \rightarrow fact(x) \text{ mod } y = 0$ ,

soll heißen: jede natürliche Zahl zwischen 1 und  $x$  teilt  $fact(x)$

**Spezifikation** sort:

(1)  $\forall k : list[nat], i, j : nat. 0 \leq i \leq |k| - 1 \wedge 0 \leq j \leq |k| - 1 \wedge i \leq j \rightarrow element(sort(k), i) \leq element(sort(k), j)$ ,

soll heißen: die Elemente von  $sort(k)$  sind aufsteigend geordnet

**Reicht das zur Spezifikation ?**

## 1.2 Das Validierungsproblem

**Frage:** Erfäßt die Spezifikation den Kundenwunsch ?

**Beispiel:** Produzent liefert Prozedur mit  $fact(n) = 2 * 3 * \dots * n * 2^n$

*Kunde:* Das habe ich nicht gemeint, ich wollte die Fakultätsfunktion :-)

*Produzent:* Das Produkt erfüllt die Spezifikation, ich habe den Vertrag erfüllt :-)

**Beispiel:** Produzent liefert Prozedur mit  $sort(k) = \langle 1, 2, \dots, |k| \rangle$

*Kunde:* Das habe ich nicht gemeint, ich wollte, daß die Elemente von  $k$  sortiert werden :-)

*Produzent:* Das Produkt erfüllt die Spezifikation, ich habe den Vertrag erfüllt :-)

**Lösung des Problems:** Prinzipiell keine !

*Das Validierungsproblem ist prinzipiell unlösbar – wie sollte man auch feststellen, ob das, was man aufgeschrieben hat, auch das ist, was man meint ?*

*Ein gängiges Problem, das oft vor den Gerichten landet !*

**Reales Beispiel:** Die Bahn hatte 60 Neigetechnikzüge *ICE2* stillgelegt, da diese signifikante Betriebsstörungen aufwiesen.

*Kunde (Bahn):* Fordert Schadensersatz.

*Produzent:* Die Züge wurden so gebaut wie es spezifiziert war.

**Fazit:** Spezifikation nachbessern (vor Vertragsschluß!) und hoffen  
(*Validierungsproblem!*), daß die Spezifikation jetzt auch “stimmt”.

**Spezifikation** *fact*:

(1)  $\forall x, y: \text{nat. } 1 \leq y \leq x \rightarrow \text{fact}(x) \text{ mod } y = 0$  **und**

(2)  $\forall x, y: \text{nat. } y \geq 2 \wedge y > x \wedge (\forall z: \text{nat. } 2 \leq z \leq x \rightarrow y \text{ mod } z \neq 0)$   
 $\rightarrow \text{fact}(x) \text{ mod } y \neq 0$

soll heißen:

(1) jede natürliche Zahl zwischen 1 und  $x$  teilt  $\text{fact}(x)$  **und**

(2) jede Zahl  $y \geq 2$  mit  $y > x$ , die kein Vielfaches einer Zahl zwischen 2 und  $x$  ist, teilt nicht  $\text{fact}(x)$ .

**Fazit:** Spezifikation nachbessern (vor Vertragsschluß!) und hoffen  
(*Validierungsproblem!*), daß die Spezifikation jetzt auch “stimmt”.

**Spezifikation** *sort*:

- (1)  $\forall k:\text{list}[\text{nat}], i, j:\text{nat}. 0 \leq i \leq |k| - 1 \wedge 0 \leq j \leq |k| - 1 \wedge i \leq j \rightarrow \text{element}(\text{sort}(k), i) \leq \text{element}(\text{sort}(k), j)$  **und**  
 (2)  $\forall k:\text{list}[\text{nat}], n:\text{nat}. \text{count}(n, k) = \text{count}(n, \text{sort}(k))$

soll heißen:

- (1) die Elemente von *sort(k)* sind aufsteigend geordnet **und**  
 (2) *jede natürliche Zahl n ist genauso oft in k enthalten wie in sort(k), d.h. bei Sortieren werden keine Elemente “vergessen” und keine Elemente werden hinzugefügt.*

**Frage:** Reicht das zur Spezifikation?

Was bedeuten eigentlich “<”, “→”, “∧”, “*mod*”, “=”, “>”, “≠”, “≤”, “|*k*|”, “*element*”, “↔”, “*count*”, ... ?

***Darüber müssen sich Kunde und Produzent einig sein !***

**Lösung:** Bezug auf *Normen, Standards, Gesetze, Verordnungen, Axiome, ...*

**Aber:** Wie detailliert die Begriffe auch beschrieben werden, letztendlich immer Rückgriff auf *natürliche Sprache*. D.h., zu guter Letzt werden *nicht-formal* definierte Begriffe verwendet.

***Das gilt auch für die Mathematik !***

**Beispiel:** “ $A \wedge B$  ist wahr” genau dann, wenn sowohl  $A$  als auch  $B$  wahr sind.

Definiert formal “ist wahr”, aber verwendet die nicht formal definierte Begriffe “*genau dann, wenn*”, “*sowohl als auch*”

**Einspruch:** “ $A \wedge B$  ist wahr” kann man ja auch über eine Wahrheitstafel definieren, und dann ist die Definition ja formal !

**Antwort:** Stimmt !

**Aber:** Was bedeuten eigentlich die Zeilen und Spalten der Wahrheitstafel? Entweder ist das genau so *offensichtlich*, wie die Bedeutung von “genau dann, wenn” und “sowohl als auch” *offensichtlich* ist, oder man definiert die Bedeutung der Wahrheitstafel (*formal*  $\rightarrow$  *formal*  $\rightarrow$  ...  $\rightarrow$  zum Schluß unvermeidbar *informell*).



**Fazit:** Es wird solange formal definiert, bis man auf Begriffe kommt, deren (natürlichsprachliche) Bedeutung *offensichtlich* ist.

**Also:** *Formale Reduktion des Komplexen auf das Einfache* (und Überschaubare).

**Also nochmal Frage:** Was bedeuten eigentlich “<”, “→”, “∧”, “*mod*”, “=”, “>”, “≠”, “≤”, “|*k*|”, “*element*”, “↔”, “*count*”, ... ?

Einige Begriffe werden natürlich-sprachlich definiert, z.B. “→” und “∧”

Einige Begriffe werden formal definiert (gründen dann aber letztendlich auch auf natürlich-sprachlichen Begriffen, formale Reduktion des Komplexen auf das Einfache), z.B. “*mod*”, “|*k*|”, “*element*”

## **Beispiel:** Formale Spezifikation eines Sortierverfahrens

Was brauchen wir?

(1) Auf welchen Datentypen sind die gewünschten Funktionen definiert?

*Also:* Festlegung/Definition der Datentypen

(2) *Was* bilden die gewünschten Funktionen auf *was* ab?

*Also:* Festlegung/Definition der syntaktischen Definitions- und Wertebereiche (formal: *Signatur*)

(3) Welche Eigenschaften soll die gewünschte Funktion besitzen?

Wir brauchen weitere Funktionen (und eventuell auch Datentypen), um Eigenschaften zu spezifizieren, z.B. *ordered* und *count*.

Was ist der Unterschied zwischen *sort*, *ordered* und *count*?

- *Im Kunden-Produzenten Modell:* Es müssen keine Prozeduren für *ordered* und *count* geliefert werden.

**Also:** Der Produzent muß den Nachweis “*sort* berechnet geordnete Permutation” führen und verwendet dazu *ordered* und *count*. Die Prozeduren *ordered* und *count* gehören aber nicht zum Lieferumfang des Auftrags “Sortierverfahren”.

- *Formale Spezifikation:*
  - (a) Definiere Begriffe der Lösung (z.B. *sort*) in einer Programmiersprache sowie Begriffe zur Spezifikation (z.B. *ordered* und *count*) in einer Spezifikationsprache
  - (b) Definiere gewünschte Eigenschaften der Lösung (z.B. von *sort*) mittels Begriffen der Spezifikation (z.B. *ordered* und *count*) durch Formeln der Spezifikationsprache. Damit ist Lösung (z.B. *sort*) formal spezifiziert!

Stimmen Kunde und Produzent bzgl. einer solchen Spezifikation überein, so wird diese zum “Vertragsgegenstand” (verbindliche Produktbeschreibung).

**Wir sind noch nicht fertig:** Wie wird der Vertrag durch den Produzenten erfüllt?

- (4) Produzent implementiert Lösung (z.B. *sort*)
- (5) Produzent weist nach, daß die Implementierung der Lösung (z.B. *sort*) die Spezifikation auch *erfüllt*, d.h. daß die (z.B. mittels *ordered* und *count*) spezifizierten Eigenschaften auch tatsächlich gelten

**Validierung** Fallstudie *Insertion Sort* aus **Kapitel 2:**

- Produzent und Kunde “*glauben*”, daß
  - *ordered* entscheidet, ob eine Liste bzgl.  $\leq$  geordnet ist,
  - *count* die Anzahl der Vorkommen einer natürlichen Zahl in einer Liste zählt
- *Aber*: *insert* und *isort* sind nicht komplizierter als *ordered* und *count* !
- *Also*: Dann können wir doch auch gleich “*glauben*”, daß *isort* eine geordnete Permutation berechnet und brauchen keine Verifikation !

**Stimmt** (aber nur, weil dies ein einfaches Beispiel ist):

- *Insertion Sort*:
  - **Implementierung**: 2 Datentypen und 3 Prozeduren
  - **Spezifikation**: 2 Prozeduren
- *Heap Sort*:
  - **Implementierung**: 3 Datentypen und 8 Prozeduren
  - **Spezifikation**: 2 Prozeduren

**Fazit:**

Ohne Verifikation müßten wir bei *Heap Sort* schon sehr *viel mehr* “*glauben*”.

### 1.3 Spezifizieren in *VeriFun*

- Sprache  $\mathcal{L}$  von *VeriFun* ist eine *Programmier- und Spezifikationssprache*
  - *Programme*: Definition von Datentypen und Prozeduren
  - *Spezifikationen*: Definition von Datentypen, Prozeduren und Lemmata
- *Beispiel* Fallstudie *Insertion Sort* aus **Kapitel 2**:
  - *Programm*:
    - \* `structure nat,`
    - \* `structure list,`
    - \* `function  $\leq$ ,`
    - \* `function insert`
    - \* `function isort`
  - *Spezifikation*:
    - \* `function ordered,`
    - \* `function count,`
    - \* `lemma isort_sorts,`
    - \* `lemma isort_permutes`
  - *Restliche Lemmata*: “Hilfs”lemmata zum Nachweis “Programm erfüllt Spezifikation” ( $\Rightarrow$  beweise `isort_sorts` und `isort_permutes`)

**Also:**

In *VeriFun* wird *algorithmisch* (= konstruktiv) *spezifiziert*, denn Begriffe wie “geordnete Liste” und “Anzahl der Vorkommen in einer Liste” werden durch *Prozeduren* ( $\Rightarrow$  ordered und count) *definiert*

**Algorithmische Spezifikation:**

- *Vorteile:*
  - Kein zusätzlicher Aufwand bei Implementierung eines Beweissystems ( $\Rightarrow$  Prozeduren haben wir ja schon)
  - Algorithmische Definitionen unterstützen signifikant *Automatisierung der Beweissuche*
- *Nachteile:*
  - Mitunter umständliche und unübersichtliche Modellierungen
    - \* Beispiel: *Nicht-freie* Datentypen wie *ganze Zahlen, Mengen, ...*
    - \* Beispiel: Modellierung von *Verbänden, Gruppen, Ringen, Körpern, Vektorräumen, ...*

## Alternative:

**Axiomatische Spezifikation** = Begriffe werden durch Axiome definiert

- *Vorteile:*

- Elegante und verständliche Modellierungen auch bei *nicht-freien* Datentypen wie *ganze Zahlen, Mengen, ...* sowie Strukturen wie *Verbände, Gruppen, Ringe, Körper, Vektorräume, ...*

- *Nachteile:*

- Zusätzlicher Aufwand bei Implementierung eines Beweissystems
- Höherer *Interaktionsbedarf* bei Beweissuche

## Fazit:

*Ein Verifikationssystem sollte beide Spezifikationsformen zur Verfügung stellen*

- *VeriFun 3.2.2*: Algorithmische Spezifikationen
- *VeriFun 4* (demnächst verfügbar): Zusätzlich axiomatische Spezifikationen

## Ab jetzt:

- Keine Unterscheidung ob Datentyp- und Prozedurdefinitionen zum Programm oder zur Spezifikation eines Programms gehören.
- *Also*: Wir betrachten ab jetzt nur Programme sowie Lemmata über diese Programme, die bewiesen werden sollen

## 2 Beweisen in *VeriFun*

### 2.1 Der *HPL*-Kalkül

- Beweise werden in *VeriFun* mittels des sogenannten *HPL-Kalküls* erstellt  
( $\Rightarrow$  **H**ypotheses, **P**rograms and **L**emmas)

- (1) **Formeln des *HPL*-Kalküls:** Sequenzen  $seq$  der Form  $\langle H, IH \Vdash goal \rangle$  mit
- $H$  = endliche Menge von *Literalen* (= Menge der *Hypothesen*)
  - $IH$  = endliche Menge von (universell quantifizierten und nicht geschlossenen) *boolschen Termen* (= Menge der *Induktionshypothesen*)
  - $goal$  = boolscher Term (= *Beweisziel*)

- (2) **Beweisregeln des *HPL*-Kalküls:** (Für Sequenzen  $seq, seq_1, \dots, seq_n$ )  
Regeln der Form

$$\frac{seq}{seq_1, \dots, seq_n} \quad (1)$$

- (3) **Herleitungen im *HPL*-Kalkül:**

- *Beweisbäume*, deren Knoten jeweils mit einer *HPL-Sequenz* markiert sind
- *Wurzelmarkierung* des *Beweisbaums* eines *Lemmas* mit Rumpf  $body_{lem}$  :

$$\langle \emptyset, \emptyset \Vdash body_{lem} \rangle$$



- Hat ein Knoten  $k$  des Beweisbaums mit Markierung  $seq$  die Nachfolgerknoten  $k_1, \dots, k_n$  mit den Markierungen  $seq_1, \dots, seq_n$ , so gibt es eine *HPL*-Regel der Form (1)

### Semantik von *HPL*-Sequenzen:

- Eine Sequenz  $seq = \langle H, IH \Vdash goal \rangle$  repräsentiert die Formel<sup>1</sup>  $\varphi_{seq} = \forall \dots \left[ \bigwedge_{h \in H} h \equiv \text{true} \wedge \bigwedge_{\forall \dots ih \in IH} \forall \dots (ih \equiv \text{true}) \rightarrow goal \equiv \text{true} \right]$ .
- Eine Sequenz  $seq$  ist “wahr” gdw.  $\varphi_{seq}$  “wahr” ist.
- $\varphi_{seq}$  ist “wahr”  $\Rightarrow$  **Kapitel 12** (Definition 15).

**Eigenschaften der *HPL*-Regeln:** Für jede *HPL*-Regel wie unter (1) gilt:

- (1)  $seq_1$  ist “wahr”  $\wedge \dots \wedge seq_n$  ist “wahr”  $\Rightarrow seq$  ist “wahr”.
- (2) *Insbesondere gilt:* Jede Sequenz  $\langle H, IH \Vdash \text{true} \rangle$  ist “wahr”.
- (3) *Mit (1) und (2):* Sind alle *Blätter* eines Beweisbaums mit Sequenzen der Form  $\langle H, IH \Vdash \text{true} \rangle$  *markiert*, so ist die Sequenz des *Wurzelknotens* “wahr”.
- (4) *Konsequenz:* Sind alle *Blätter* eines Beweisbaums eines *Lemmas*  $lem$  mit Sequenzen der Form  $\langle H, IH \Vdash \text{true} \rangle$  *markiert*, so ist das *Lemma* “wahr”.

<sup>1</sup> “ $\equiv$ ” ist das Gleichheitszeichen der Prädikatenlogik 1. Stufe, also ein *Prädikatensymbol* ( $\Rightarrow$  **Kapitel 9**, Folie 5).

*Erzeugung von Beweisbäumen durch **Divide-and-Conquer Prinzip** ( $\Rightarrow$  GdI 2):*

- Zerlege Problem solange in Unterprobleme, bis diese (einfach) lösbar sind, und bilde aus der Lösung der Unterprobleme die Lösung des Ausgangsproblems
- *Hier:* Durch Anwendung einer *HPL*-Regel wird aus einer Sequenz *seq* eine Menge  $\{seq_1, \dots, seq_n\}$  von Sequenzen gewonnen, deren Wahrheit die Wahrheit von *seq* impliziert
- *Also:* Zerlege eine Sequenz (mittels der *HPL*-Regeln) solange in Untersequenzen, bis diese (einfach) beweisbar sind ( $\Rightarrow \langle H, IH \Vdash \text{true} \rangle$ ), und bilde aus dem Beweis der Untersequenzen den Beweis der Ausgangssequenz

**Anzeige von *HPL*-Sequenzen in *VeriFun*:**

- *HPL*-Sequenz des Knotens eines Beweisbaums ( $\Rightarrow$  *Proof Window*):
  - Hypothesen  $H \Rightarrow$  Reiter *Hypotheses* im *Proof Viewer*
  - Induktionshypothesen  $IH \Rightarrow$  Reiter *Induction Hypotheses* im *Proof Viewer*
  - Beweisziel  $goal \Rightarrow$  Anzeige *Goal* im *Proof Viewer*

**Beispiel:** *HPL*-Sequenzen im Beweis von Lemma `isort_sorts`

- *Wurzelknoten:*  $\langle \emptyset, \emptyset \Vdash \text{ordered}(\text{isort}(k)) \rangle$
- *Knoten des Induktionsanfangs:*  $\langle \{ ?\emptyset(k) \}, \emptyset \Vdash \text{ordered}(\text{isort}(k)) \rangle$
- *Knoten des Induktionsschritts:*  
 $\langle \{ ? :: (k) \}, \{ \text{ordered}(\text{isort}(\text{tl}(k))) \} \Vdash \text{ordered}(\text{isort}(k)) \rangle$

## 2.2 Editieren von *HPL*-Beweisen (= Erzeugen von Beweisbäumen)

- Folgende *HPL*-Regeln stehen zur Verfügung  
(System => Menue *Proof\Proof Rules*; Details => *VeriFun User Guide*):
  - (1) **Simplification**: Ruft *symbolischen Interpreter* (= *Symbolic Evaluator*) zur “Vereinfachung” des Beweiszieles *goal* einer Sequenz auf (=> **Kapitel 10**)
  - (2) **Weak Simplification**: Wie (1), jedoch mit anderen Aufrufparametern.
  - (3) **Normalization**: Wie (1), jedoch mit anderen Aufrufparametern.
  - (4) **Weak Normalization**: Wie (1), jedoch mit anderen Aufrufparametern.
  - (5) **Inconsistency**: Ruft den *Symbolic Evaluator* zur Widerlegung der Hypothesenmenge *H* einer Sequenz auf (=> **Kapitel 10**)  
  
=> *Sprechweise*: Die *HPL*-Regeln (1) – (5) werden auch als *Computed Proof Rules* bezeichnet
  - (6) **Case Analysis**: Führt eine Fallunterscheidung im Beweisziel *goal* einer Sequenz durch.
  - (7) **Use Lemma**: Wendet ein Lemma oder eine Induktionshypothese an  
=> *nicht bewiesene* Lemmata (= Status *ready*) dürfen angewendet werden

- (8) **Unfold Procedure:** Ersetzt einen Prozeduraufruf im Beweisziel *goal* einer Sequenz durch den instantiierten Prozedurrumpf
- (9) **Apply Equation:** Wendet eine Gleichung aus einer Hypothese, einer Induktionshypothese oder einem Lemma an  
⇒ auch Gleichungen aus *nicht bewiesenen* Lemmata (= Status *ready*) dürfen angewendet werden
- (10) **Purge:** Ersetzt *let*-Bindungen im Beweisziel *goal* einer Sequenz  
⇒ *let*-Ausdrücke werden in Abschnitt 1.3.2 von **Kapitel 5** definiert
- (11) **Induction:** Bildet *Induktionsformeln* (= *Induktionsanfänge* und *Schrittfälle*); Details in **Kapitel 7**, Abschnitt 4
- (12) **Insert Induction Hypotheses:** Bildet Instanzen von Induktionshypothesen  
⇒ wird selten benötigt, brauchen wir hier nicht
- (13) **Insert Hypotheses:** Führt eine Fallunterscheidung in der Hypothesenmenge *H* einer Sequenz durch
- (14) **Move Hypotheses:** Verschiebt Hypothesen aus der Hypothesenmenge *H* einer Sequenz in das Beweisziel *goal*
- (15) **Delete Hypotheses:** Löscht Hypothesen aus der Hypothesenmenge *H* einer Sequenz

**Mit den Beweisregeln des *HPL*-Kalküls werden *Beweise* für  $\mathcal{L}$ -Lemmata “editiert” und damit *Beweisbäume* erstellt ( $\Rightarrow$  *Proof Window*)**

- Nach Eingabe eines Lemmas *lem* ( $\Rightarrow$  Menue *Program\Insert Element*) erzeugt System *initialen Beweisbaum* (= nur Wurzelknoten) für *lem*
  - *Voraussetzung*: Lemma erhält Status “ready” (= *blaues* Lemma-Icon); falls im Lemmarumpf Prozeduren aufgerufen werden, deren *Terminierung* ( $\Rightarrow$  **Kapitel 8**) *nicht nachgewiesen* wurde, so gilt Lemmastatus  $\neq$  “ready” !
- *Erstellen* von Beweisen in *VeriFun*
  - **Benutzer**: Selektiert im *Program Window* ein Lemma
  - **Benutzer**: Öffnet mit *Program\Proof* den Beweisbaum des Lemmas im *Proof Window*
  - **Benutzer**: Selektiert im *Proof Window* ein Blatt des Beweisbaums
  - **Benutzer**: Wählt in *Proof\Proof Rules* eine *HPL*-Regel aus
  - **Benutzer**: Editiert Parameter für die ausgewählte Regel (falls erforderlich)
  - **System**: Überprüft Anwendbarkeit der Regel und Gültigkeit der angegebenen Parameter
  - **System**: Erweitert Beweisbaum durch Anfügen von Sohnknoten an dem ausgewählten Beweisbaumblatt

## 2.3 Die *Verify*- und die *NextRule*-Taktiken

Die *Verify*- und die *NextRule*-Taktiken

- berechnen Anwendungsfolgen von *HPL*-Regeln, die “oft” zum Erfolg führen
- Die *Verify*-Taktik wird durch *Benutzer* mit *Program\Verify* für ein Lemma gestartet, das im *Program Window* selektiert ist
- Die *NextRule*-Taktik wird *automatisch* nach jeder interaktiven Anwendung einer *HPL*-Regel gestartet
- **Zweck beider Taktiken:** *Entlastung* des *Benutzers* von *Interaktionen*  
=> Auswahl von *HPL*-Regeln und Eingabe von Regelparametern
- *Anzeige* im *Beweisbaum*:  $\langle \text{Regelname} \rangle^*$ ,  
=> Suffix “\*” zeigt an, daß die Regel automatisch ausgewählt wurde
- Beide Taktiken sind mittels *Heuristiken* implementiert  
=> können also auch Unbrauchbares produzieren
- **Deshalb:**
  - *NextRule*-Taktik kann in *Options\User Settings* abgestellt werden
  - beide Taktiken können in *Window\Open Proof Control* abgebrochen werden  
=> *Selektiere* Task im *Proof Control Window* mit Maus-rechts  
und wähle dann *Cancel*

## Was wird hier unter *Heuristik* verstanden ?

- bezeichnet Kriterium zur *Auswahl* unter mehreren möglichen Folgeschritten bei *unentscheidbaren Problemen*  
=> Indeterminismus determinieren
- **Zweck:** “Möglichst gute” Bestimmung eines Folgeschritts  
=> “möglichst gut” bzgl. der Berechnung der Lösung eines Problems
- Heuristiken sind
  - **nie** *richtig* oder *falsch*
  - entweder *gut* (= oft erfolgreich) oder *schlecht* (= selten erfolgreich)
- *Güte* einer Heuristik kann nur *empirisch* festgestellt werden  
=> analysiere große Anzahl von Lösungsversuchen
- **Beispiel:**  
Erfolgsquote bei heuristisch gesteuerter automatischer Auswahl der *HPL*-Regel *Induction* in *VeriFun*: > 95%



**Beispiel:** Güte der Heuristiken in der Fallstudie *InsertionSort*

- Erfolgsquote *Verify*- und *NextRule*-Taktik = 100%
- D.h. keine Benutzerinteraktion zum Erstellen der Beweisbäume notwendig (wenn die erforderlichen Lemmata formuliert sind)
- Verwendete *HPL*-Regeln hier nur *Induction* und *Simplification*
- *Aber:* Bei *InsertionSort* gibt es auch keine sooo schwierigen Beweisprobleme

**Beispiel:** Güte der Heuristiken in der Fallstudie *HeapSort*

- Erfolgsquote *Verify*- und *NextRule*-Taktik = 92,7%
- 15 Benutzerinteraktion (= 7.3%) zum Erstellen der Beweisbäume notwendig (wenn die erforderlichen Lemmata formuliert sind)
- Verwendete *HPL*-Regeln hier neben *Induction* und *Simplification*
  - 3× *Case Analysis*
  - 5× *Use Lemma*
  - 2× *Unfold Procedure*
  - 5× *Apply Equation*

### 3 Testen in *VeriFun*

- *Programme* sind oft *falsch*
- *Behauptungen über Programmeigenschaften* (hier: *Lemmata*) sind oft *falsch*
- **Dann:** Vergeudete Zeit und Frustration bei vergeblichen Beweisversuchen
- **Konsequenz:** *Testen, Testen, Testen, ...*
- Mit *Testen* versucht man *Fehler* in Programmen und/oder *Lemmata* zu *finden*
  - *erfolglos, falls ohne Fehler*
- Mit *Beweisen* versucht man zu *zeigen*, daß *keine Fehler* in Programmen und/oder *Lemmata existieren*
  - *erfolglos, falls mit Fehlern*
- ***VeriFun*:** Testen mittels
  - des *symbolischen Interpreters* (= *Symbolic Evaluator*)
  - des *Disprovers* (= *Widerlegungsbeweiser*)

## 3.1 Testen durch symbolische Interpretation

### 3.1.1 Symbolische Interpretation von Termen

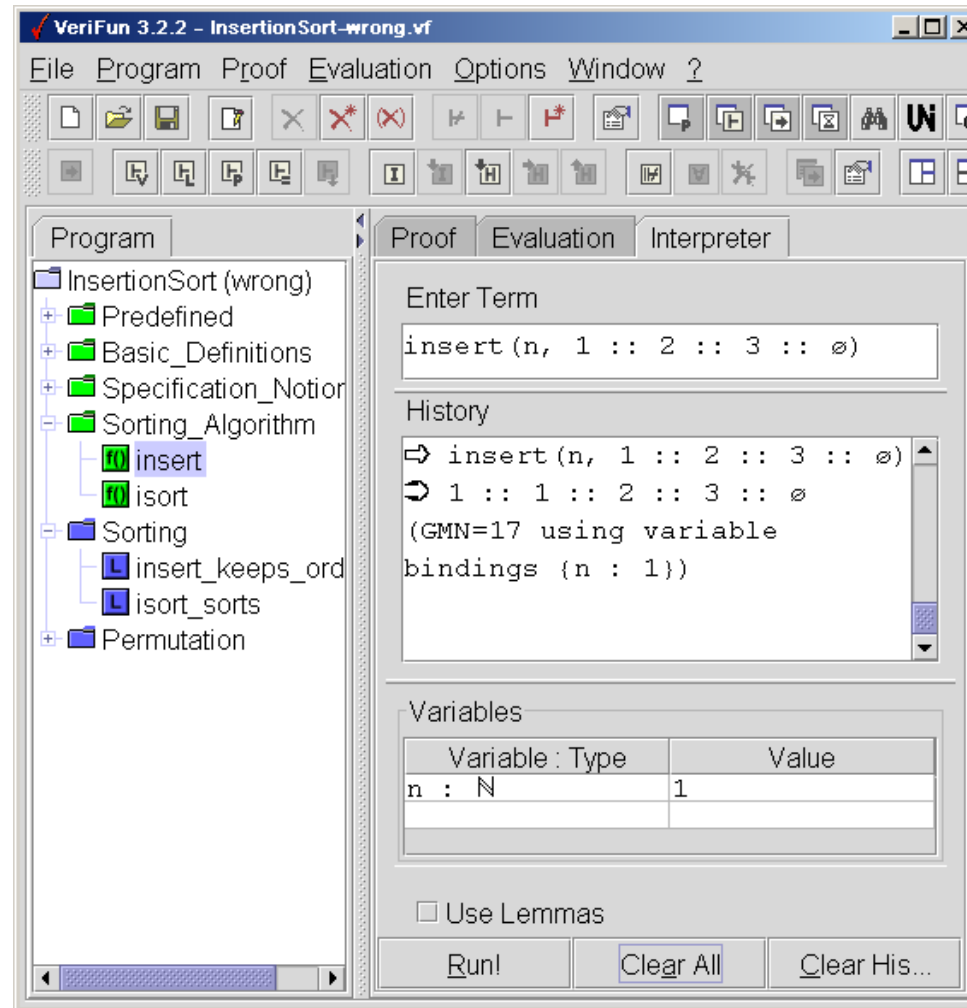
- Mit dem symbolischen Interpreter (= *Symbolic Evaluator*) können Terme durch “Ausrechnen” *vereinfacht* werden
- “*symbolisch*”: Auch Terme, die *Variable* enthalten, können “ausgerechnet” werden
- **Beispiel:**

```
function [infixr,20] +(x:ℕ, y:ℕ):ℕ <=  
if ?0(x) then y else +( -(x) + y) end_if
```

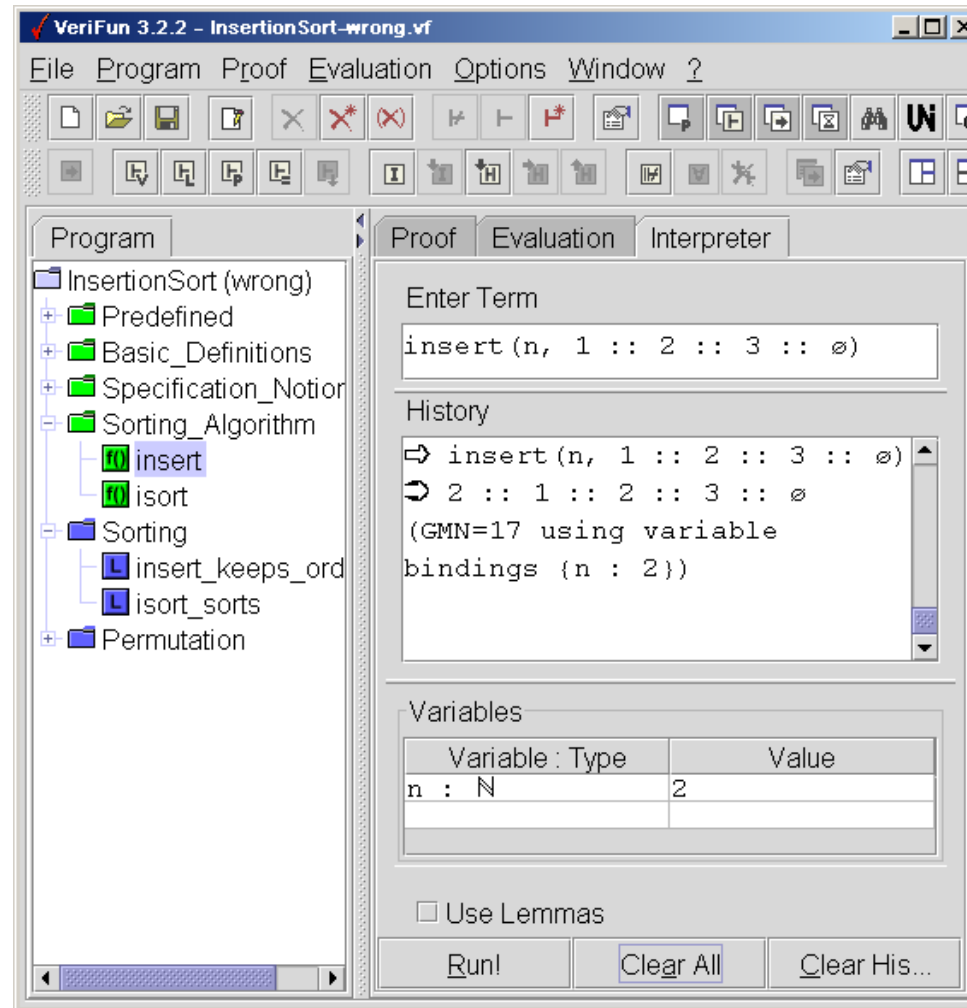
- *Eingabe*:  $3 + y$  ; *Ergebnis*:  $+(+(+(y)))$  (‘+’ wurde “ausgerechnet”)
  - *Eingabe*:  $y + 3$  ; *Ergebnis*:  $y + 3$  (‘ $y + 3$ ’ kann nicht “ausgerechnet” werden)
- Um Terme symbolisch “auszurechnen”, werden diese im *Interpreter Window* eingegeben

- **Beispiel:** Fehlerhafte Implementierung von insert:

```
function insert(n : ℕ, k : list[ℕ]) : list[ℕ] <=
if ?∅(k)
  then n :: ∅
  else if hd(k) ≤ n
    then n :: k
    else hd(k) :: insert(n, tl(k))
  end_if
end_if
```



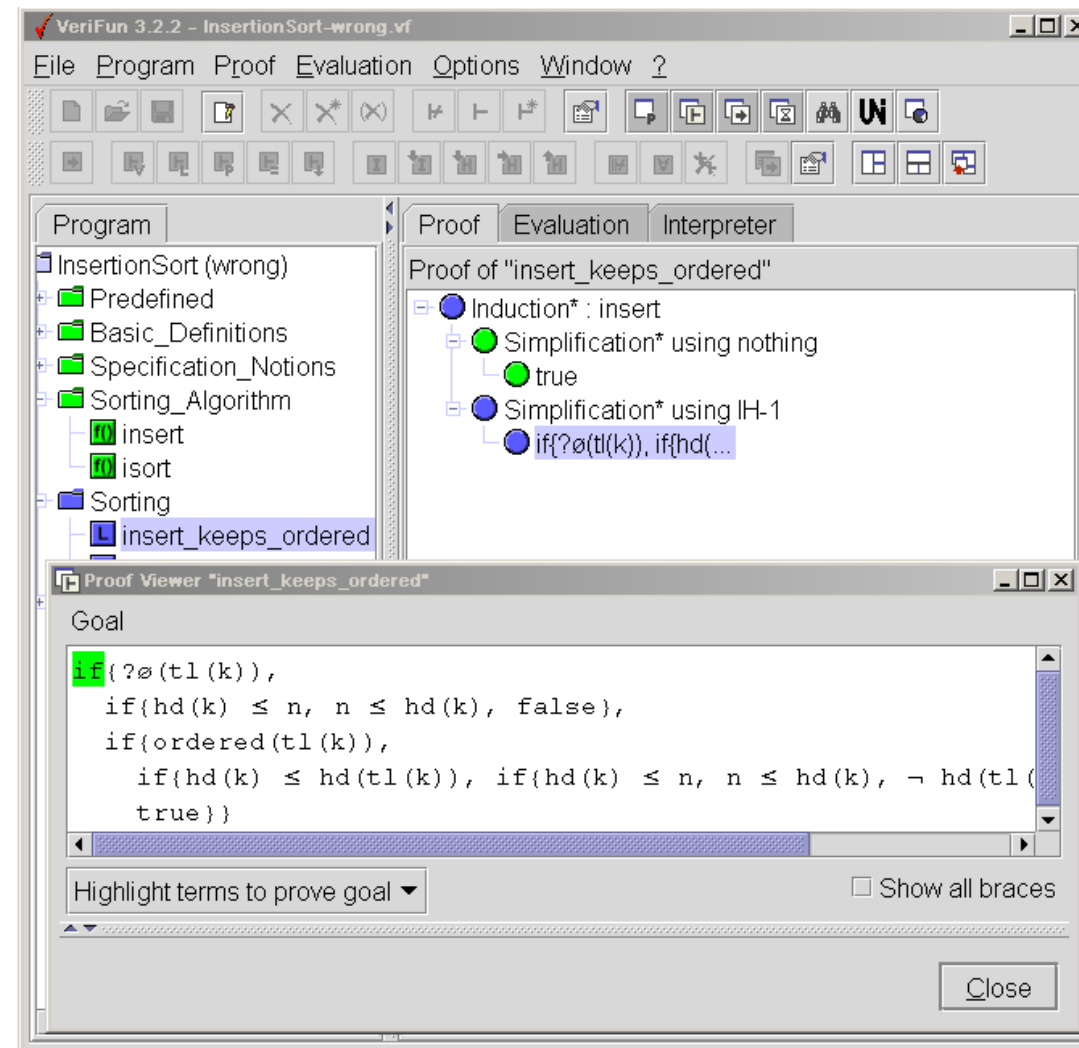
- Wir fügen 1 in die Liste  $1 :: 2 :: 3 :: \emptyset$  ein und erhalten als Ergebnis  $1 :: 1 :: 2 :: 3 :: \emptyset$
- *Test erfolglos!*
- Aber vorsichtshalber noch ein Versuch ...



- Wir fügen 2 in die Liste  $1 :: 2 :: 3 :: \emptyset$  ein und erhalten als Ergebnis  $2 :: 1 :: 2 :: 3 :: \emptyset$
- *Test erfolgreich*, 2 wurde nicht an der richtigen Stelle eingefügt !
- **Jetzt:** Fehler in *insert* *suchen* und *reparieren* und danach wieder *testen*

### 3.1.2 Symbolische Interpretation von *HPL*-Sequenzen

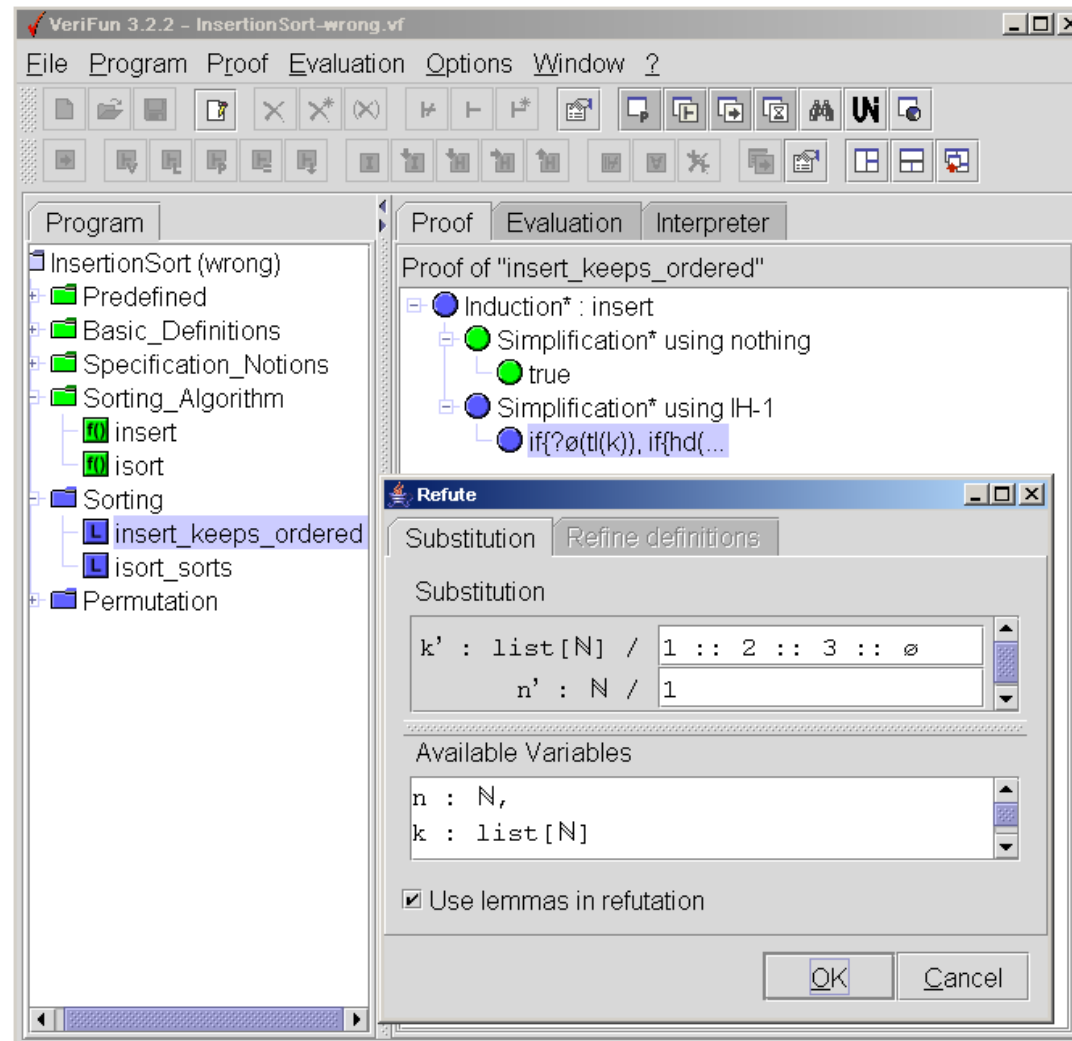
- *HPL*-Sequenzen  $seq$  können getestet werden, indem Variable  $x:\tau$  (aus  $seq$ ) durch Terme  $t:\tau$  ersetzt werden und die so gebildete Instanz  $seq'$  von  $seq$  durch den symbolischen Interpreter “ausgerechnet” wird
  - $Ergebnis = \text{true}$ : Test erfolglos
  - $Ergebnis = \text{false}$ : Test erfolgreich, die Terme  $t:\tau$  bilden ein *Gegenbeispiel* für die Gültigkeit der Sequenz
  - $Ergebnis \notin \{\text{true}, \text{false}\}$ : Test weder erfolgreich noch fehlgeschlagen (=  $\Rightarrow$  andere Terme  $t:\tau$  wählen)
- Instanzen einer *HPL*-Sequenz werden mittels des Kommandos *Refute* aus dem *Proof Menue* eingegeben

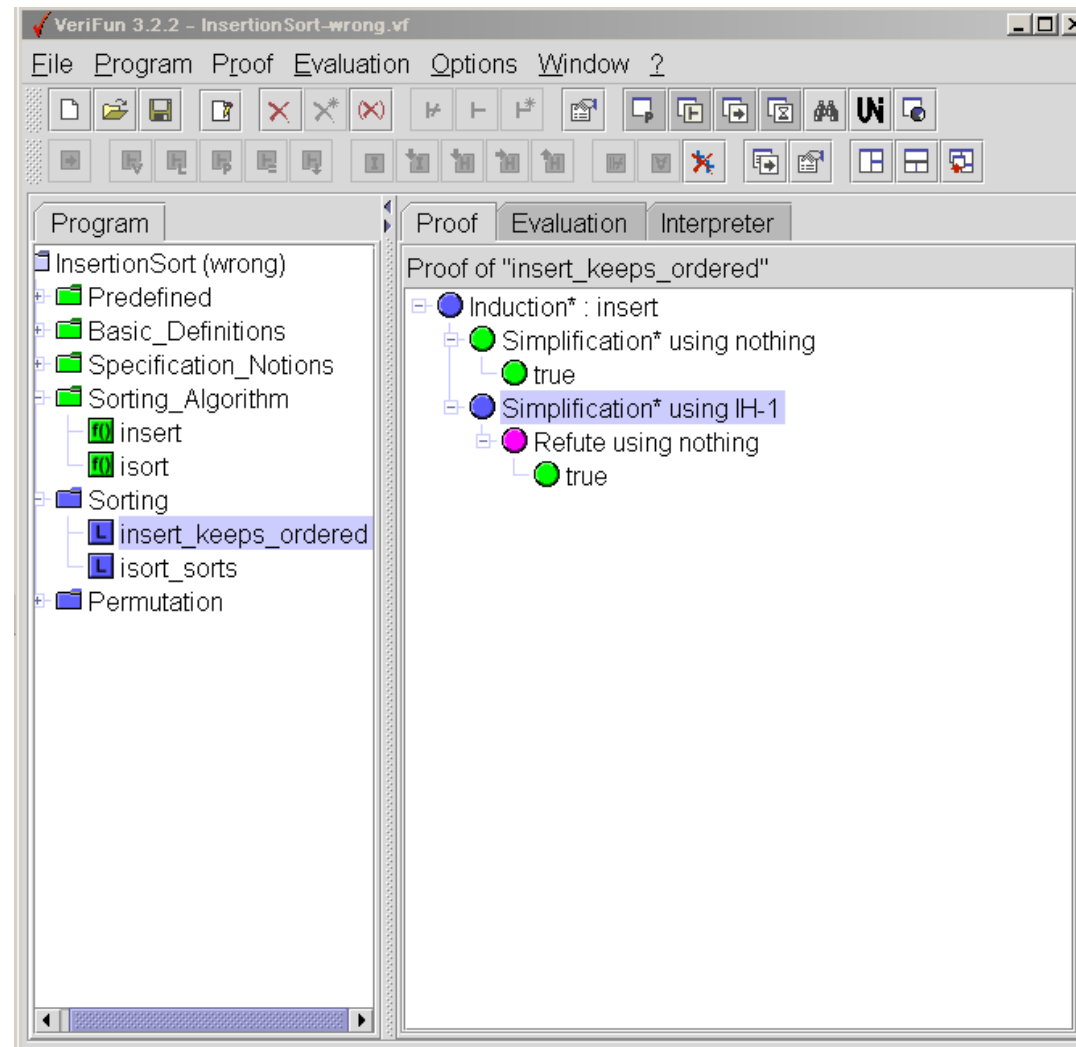


- Beweis von Lemma `insert_keeps_ordered` scheitert (wegen fehlerhaftem `insert`)



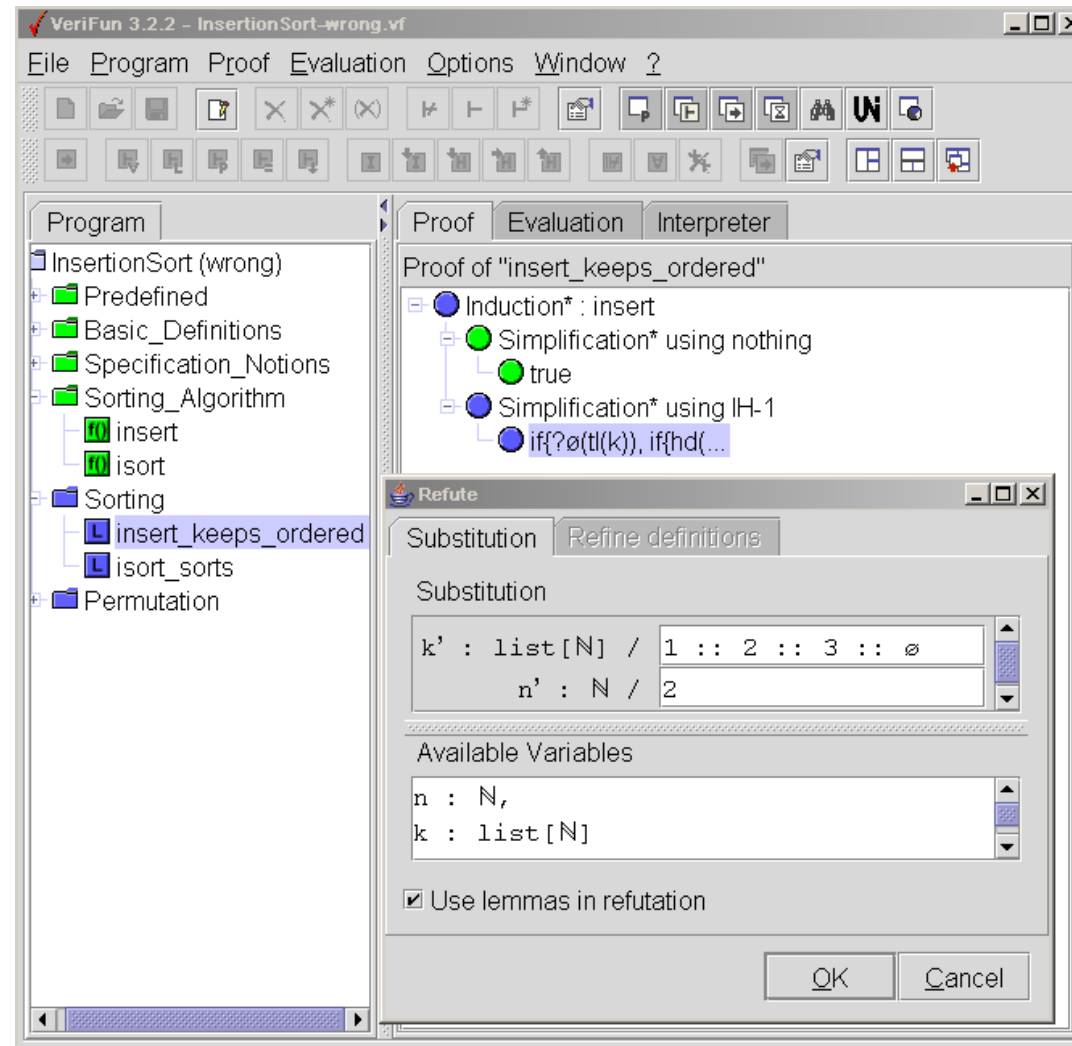
- Wir testen die *HPL*-Sequenz mit  $k := 1 :: 2 :: 3 :: \emptyset$  und  $n := 1$

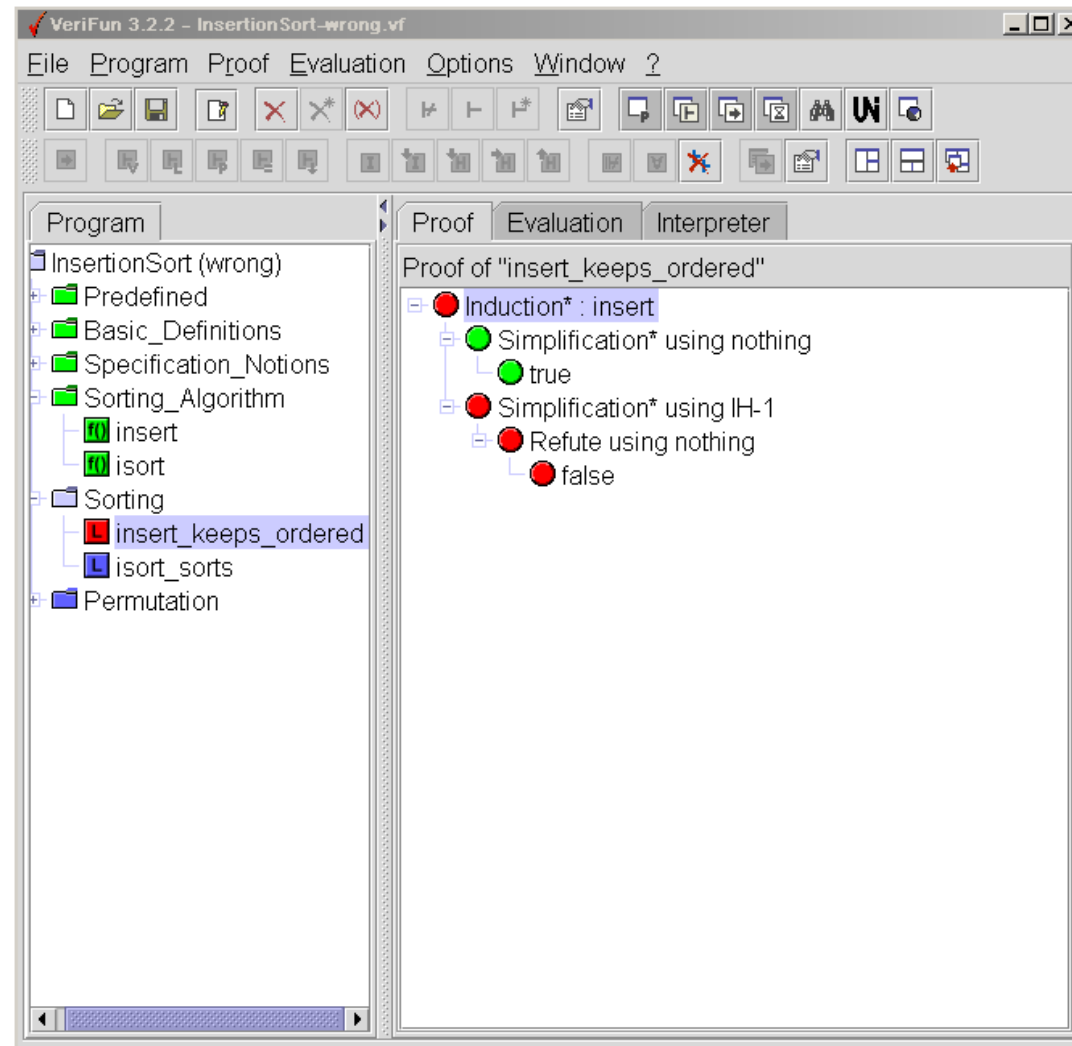




- *Test erfolglos !*
- Aber vorsichtshalber noch ein Versuch ...

- Wir testen die *HPL*-Sequenz mit  $k := 1 :: 2 :: 3 :: \emptyset$  und  $n := 2$



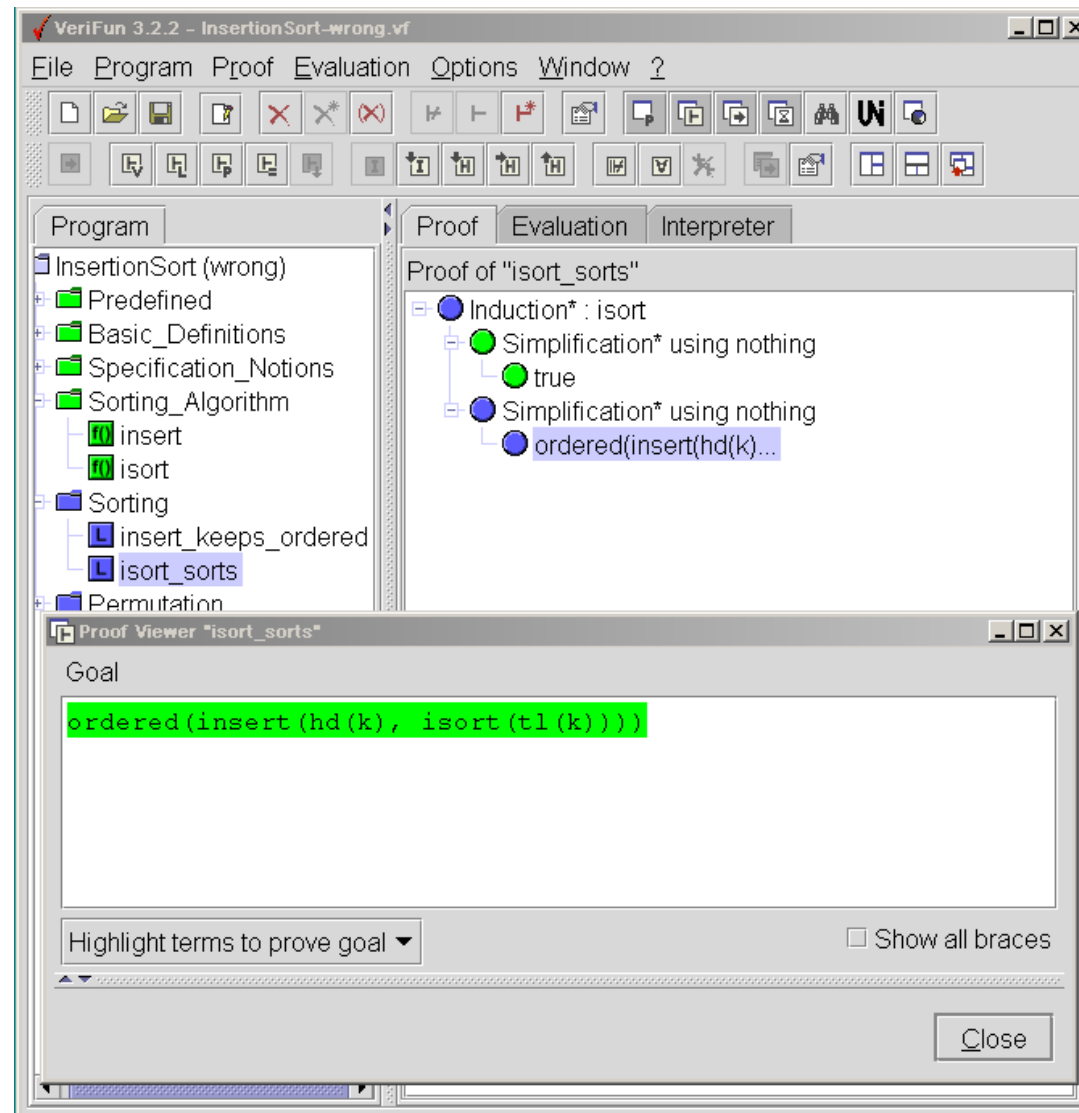


- *Test erfolgreich* – mögliche Fehler
  - insert und/oder ordered falsch implementiert
  - Behauptung von insert\_keeps\_ordered ist falsch
- **Jetzt:** Fehler *suchen* und *reparieren* und danach wieder *testen*

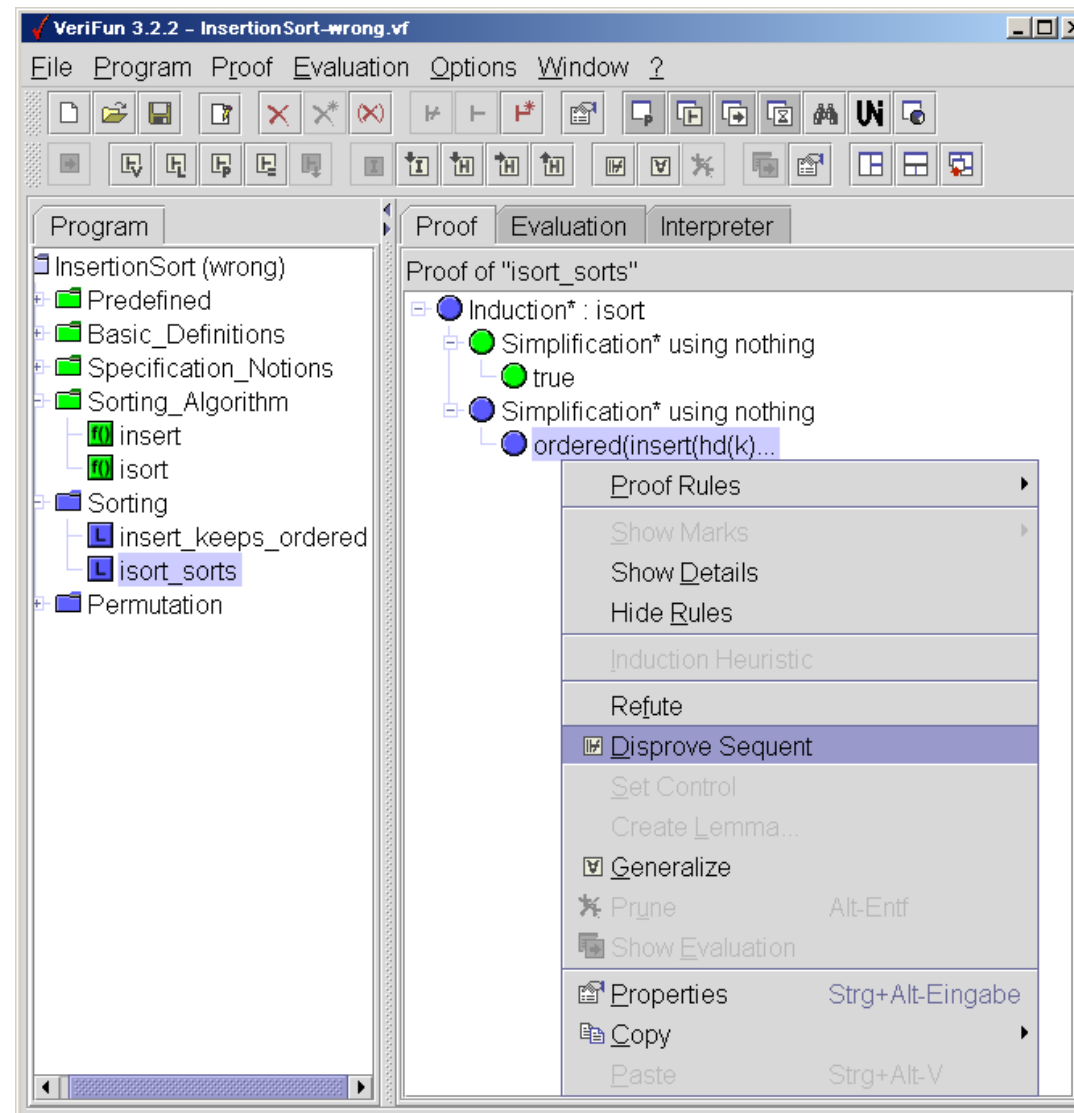
## 3.2 Testen mit dem *Disprover*

### 3.2.1 Widerlegung von *HPL*-Sequenzen

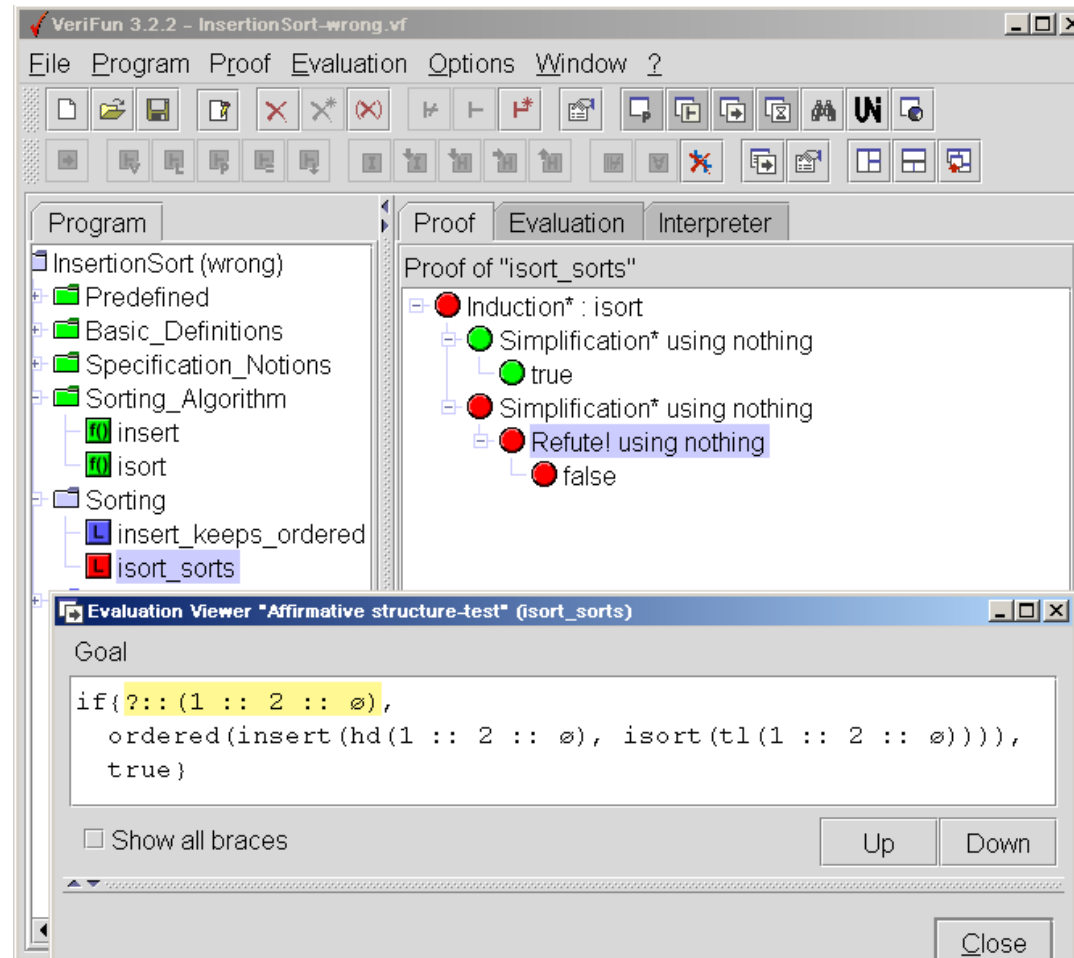
- Der *Disprover* ist ein *automatischer Widerlegungsbeweiser*
- Versucht eine *HPL*-Sequenz *seq* zu *widerlegen*
- **Vorgehen:** *Finde* für Variable  $x:\tau$  in *seq* einen Term  $t:\tau$ , so daß *seq* nach Ersetzung der Variablen  $x$  durch  $t$  falsch wird
- **Damit:** Der *Disprover* versucht *automatisch Eingaben* für *Refute* zu finden, für die ein Test erfolgreich ist



- Beweis von Lemma `isort_sorts` scheitert (bei fehlerhaftem `insert`)
- Wir testen die *HPL*-Sequenz mit dem *Disprover*



- Aufruf des *Disprovers* mittels *Disprove Sequent* im *Proof* Menue

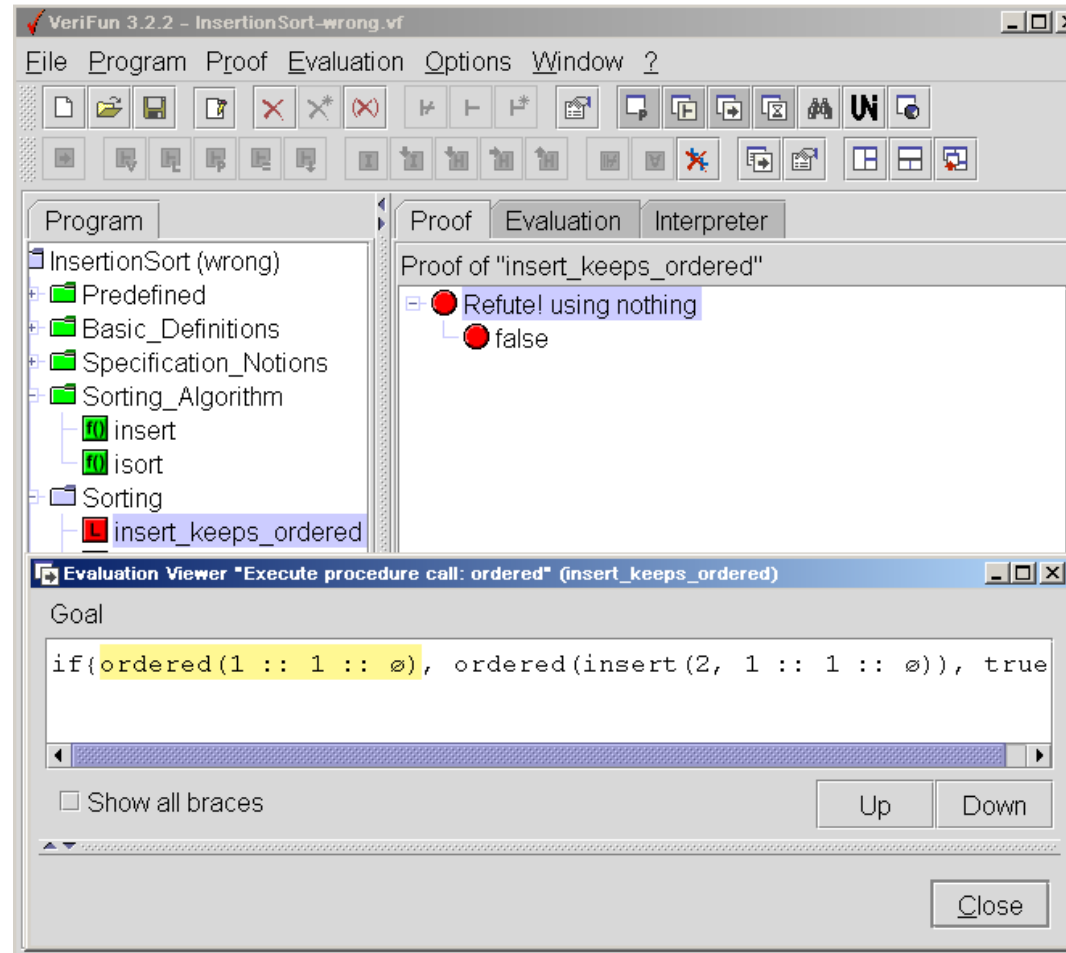


- Anzeige des Ergebnisses im *Evaluation Viewer*  
(=> Doppelklick auf *Refute!*-Knoten im Beweisbaum)
- Der *Disprover* hat die Instanz  $1 :: 2 :: \emptyset$  für  $k$  gefunden, d.h. für  $k := 1 :: 2 :: \emptyset$  berechnet *isort keine* sortierte Liste



### 3.2.2 Widerlegung von Lemmata

- Lemmata  $lem$  können auch “direkt” durch den *Disprover* widerlegt werden
- **Voraussetzung:**  
Der Beweisbaum von  $lem$  besteht *nur* aus dem *Wurzelknoten* (= initialer Beweisbaum)  
=> gegebenenfalls Baum mittels *Proof\Prune* an der Wurzel abschneiden
- **Aufruf des *Disprovers*:**
  - Selektiere Lemma-Icon im *Programm Window*
  - Wähle *Disprove Lemma* im *Program Menue*
- **Reaktion:**  
System wendet *Disprove Sequent* auf den Wurzelknoten des Beweisbaums an



- Anzeige des Ergebnisses im *Evaluation Viewer*  
(=> Doppelklick auf *Refute!*-Knoten im Beweisbaum)
- Der *Disprover* hat die Instanz  $1 :: 1 :: \emptyset$  für  $k$  und  $2$  für  $n$  gefunden, d.h. für diese Werte von  $k$  und  $n$  ist `insert_keeps_ordered` *widerlegt* !

### 3.3 Hinweise zum Testen in *VeriFun*

- *Prozeduren* testet man am besten mit geeigneten Testeingaben mittels des *symbolischen Interpreters* (vgl. Folie 27 ff.)
- *Lemmata* testet man am besten zunächst mit *Disprove Lemma* (s. Folie 41)
  - Mögliche Ergebnisse:
    1. *Disprover* findet Gegenbeispiel  
⇒ jetzt Fehler suchen
    2. *Disprover* hält ohne Gegenbeispiel mit entsprechender Meldung im *System Log*  
⇒ In *Options\User Settings* andere *Suchstrategie* für den *Disprover* einstellen *oder*  
⇒ mit *Proof\Refute* selbst Kandidat für Gegenbeispiel formulieren  
(*Achtung*: Falls das Lemma wahr ist, so werden der *Disprover* und man selbst **nie** erfolgreich sein !)
    3. *Disprover* hält nicht  
⇒ *Proof Control* mit *Window\Open Proof Control* öffnen,  
*Disprover* in *Proof Control* beenden,  
weiter wie bei 2.

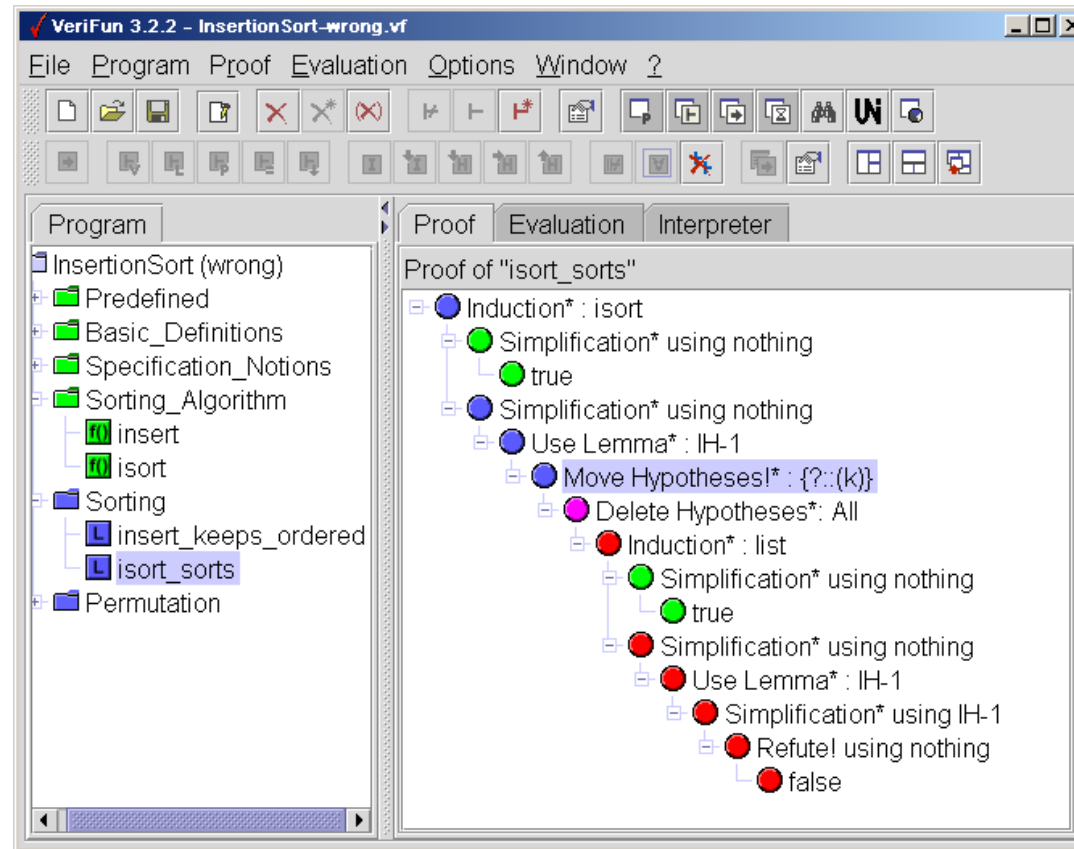
## Widerlegungen mittels *Disprove Sequent*

- Die Berechnung eines Gegenbeispiels für ein Lemma mittels *Disprove Lemma* kann wegen der Größe des Suchraums scheitern
- Da man kein Gegenbeispiel kennt, versucht man das Lemma zu beweisen
- Durch Anwendung der *HPL*-Regeln erhält man Sequenzen, die den *Suchraum* für den *Disprover einschränken*
- **Konsequenz:** *Disprove Sequent* kann *erfolgreich* sein (und damit ein Lemma widerlegen) auch wenn *Disprove Lemma* *scheitert*
- **Beispiel:** *Disprove Lemma* scheitert bei `isort_sorts`, *Disprove Sequent* ist jedoch im *Induktionsschritt* erfolgreich ( $\Rightarrow$  ausprobieren)

## Anwendungen von *Disprove Sequent*

- Die *Verify*- und *NextRule*-Taktiken erzeugen meist Beweisbäume mit geschachtelter Induktion
- In den meisten Fällen ist es sinnvoll, den Beweisbaum an dem ersten Knoten, der kein *Simplification*\*-Knoten ist, mittels *Proof\Prune* abzuschneiden und dann *Disprove Sequent* auf das resultierende Blatt des Beweisbaums anzuwenden

- **Achtung:** Ein *Delete Hypotheses*-Knoten sollte *nicht oberhalb* der Anwendung von *Disprove Sequent* stehen.
- **Grund:** Für alle *HPL*-Regeln  $R$  **außer** *Delete Hypotheses* gilt:  
*Ist eine HPL-Sequenz  $seq$  “wahr”, so sind auch alle  $seq_1, \dots, seq_n$  (die aus  $seq$  durch Anwendung von  $R$  entstehen) “wahr”.*
- **Damit:** *Ist eine der HPL-Sequenzen  $seq_i$  “falsch”, so ist auch  $seq$  “falsch”* (und damit die *HPL*-Sequenz an der *Wurzel* des Beweisbaums und damit das *Lemma*, zu dem der Beweisbaum gehört)
- **Aber:** Entsteht die *HPL*-Sequenz  $seq'$  aus  $seq$  durch Anwendung von *Delete Hypotheses*, so **gilt nicht** ‘ $seq$  “wahr”  $\rightarrow seq'$  “wahr”’, und damit **nicht** ‘ $seq'$  “falsch”  $\rightarrow seq$  “falsch”’
- **Konsequenz:** Die Widerlegung einer *HPL*-Sequenz  $seq$  wird im Beweisbaum nur bis zum ersten *Delete Hypotheses*-Knoten oberhalb von  $seq$  (und nicht bis zur Wurzel) propagiert; ein Lemma kann so nicht widerlegt werden !



- Widerlegungen von *HPL*-Sequenzen werden nur bis zum ersten *Delete Hypotheses*-Knoten propagiert

## Fehleranalyse – “Wo steckt der Fehler ?”

- Die Berechnungen des symbolischen Interpreters werden im *Evaluation Viewer* angezeigt
- Öffnen des *Evaluation Viewer*:
  - bei Eingabe im *Interpreter Window*: *Open Evaluation Viewer* im *Window Menue*
  - bei Widerlegung einer *HPL*-Sequenz: Doppelklick auf *Refute*-Knoten im Beweisbaum des *Proof Windows*
  - bei Widerlegung eines Lemmas: Beweisbaum mit *Program\Proof* im *Proof Window* öffnen und dann Doppelklick auf *Refute*-Knoten (= Wurzel) im Beweisbaum
- Im *Evaluation Viewer* können die einzelnen Rechenschritte mittels der *Up/Down*-Buttons oder der  $\downarrow / \uparrow$ -Tasten nachvollzogen werden
- Dabei die einzelnen Schritte *analysieren*:  
“*Ist Ergebnis so, wie man es erwartet ?*”
  - *Nein* – hier steckt der Fehler ( $\Rightarrow$  fehlerhafte Implementierung einer Prozedur; fehlerhafte Behauptung eines Lemmas)
  - *Ja* – nächster Schritt ( $\Rightarrow$  *Down*-Button oder  $\downarrow$ -Taste)