Formale Grundlagen der Informatik 3 –

2. Spezifikation und Verifikation

Christoph Walther
TU Darmstadt

1 Formale Verifikation

1.1 Korrektheit von Programmen

- Wann ist ein Programm "korrekt"?
- *Informell*: Jedem Programm P kann eine Funktion $\phi_P : in \mapsto out$ zugeordnet werden mit:

$$\boxed{\begin{array}{c}\forall x{:}in, \forall y{:}out.\ \phi_P(x) = y \\ \text{gdw.} \\ P \text{ aufgerufen mit } x \text{ liefert } y \text{ als Ergebnis}\end{array}}$$

- Programm P soll einen bestimmten Zweck erfüllen. Damit: Funktion ϕ_P soll gewisse Eigenschaften $\mathcal{E}_P$ besitzen. Formal

$$\forall x{:}in.\ P(x) \textit{ terminiert } \rightarrow \mathcal{E}_P(x, \phi_P(x)) \qquad (1)$$

  - $\mathcal{E}_P$ beschreibt allgemein eine Beziehung (formal: *Relation*) zwischen den Ein- und Ausgaben des Programms P.
  - Mit $\mathcal{E}_P$ wird P *spezifiziert.*
- Programm P ist *partiell korrekt bzgl. Spezifikation* $\mathcal{E}_P$ gdw. (1) gilt.
- Programm P ist *total korrekt bzgl. Spezifikation* $\mathcal{E}_P$ gdw. (1) gilt **und** P für alle $x{:}in$ *terminiert.*

Beispiel 1 *(Sortieren von Listen natürlicher Zahlen)*

- P ist (irgendein) Sortierverfahren $sort$ (etwa $insertionsort$, $heapsort$, $quicksort$, u.s.w.)
- Forderungen an (jedes) Sortierverfahren: Wenn $sort(\langle n_1, \ldots, n_k\rangle) = \langle m_1, \ldots, m_l\rangle$, dann
  - $m_1 \leq m_2 \leq \ldots \leq m_l$ (Ergebnisliste ist *geordnet*)
  - $k = l$ und $\langle m_1, \ldots, m_l\rangle = \langle n_{\pi(1)}, \ldots, n_{\pi(k)}\rangle$ für eine Permutation $\pi : \{1, \ldots, k\} \rightarrow \{1, \ldots, k\}$ (Ergebnisliste ist *Permutation* der Eingabeliste)
- Damit Spezifikation $\mathcal{E}_{sort}(x, y) := ordered(y) \wedge x \approx y$, d.h. es ist zu zeigen:

$$\boxed{\forall x{:}list[\mathbb{N}].\ ordered(\phi_{sort}(x)) \wedge x \approx \phi_{sort}(x)} \qquad (2)$$

Aufgaben der (formalen) Verifikation für gegebenes Programm P:

- *Spezifiziere* die für P geforderten Eigenschaften $\mathcal{E}_P$
- *Beweise* "$\forall x{:}in.\ \mathcal{E}_P(x, \phi_P(x))$"

Also: Es müssen *formale Beweise* geführt werden.

Was benötigt man dazu?

(1) Die *Semantik* (= *Bedeutung*) eines *Programms* muß formal erfaßt werden. D.h., es muß formal präzise modelliert werden, was das Programm "tut". **Genauer:** Die *Funktion* ϕ_P muß durch *Axiome* einer Logik L präzise definiert werden.

(2) Die *Semantik* (= *Bedeutung*) der *Begriffe der Spezifikation* müssen formal erfaßt werden (was bedeutet "$ordered$", was bedeutet "$\approx$"?). **Genauer:** Die in der *Spezifikation* $\mathcal{E}_P$ verwendeten Begriffe müssen durch *Axiome* einer Logik L präzise definiert werden.

(3) Ausgehend von den Axiomen wird jetzt "$\forall x{:}in.\ \mathcal{E}_P(x, \phi_P(x))$" mittels der Beweisregeln der Logik L *bewiesen.*

1.2 Beweiswerkzeuge

- Formale Korrektheitsnachweise für Programme sind aufwendig (selbst für einfache Programme)
- **Damit:** *Werkzeugunterstützung* unabdingbar.
- Verwendet werden *interaktive Beweissysteme*, etwa
  - *ACL2* (http://www.cs.utexas.edu/users/moore/acl2/)
  - *Isabelle* (http://www.cl.cam.ac.uk/Research/HVG/Isabelle/)
  - *PVS* (http://pvs.csl.sri.com/)
  - *COQ* (http://pauillac.inria.fr/coq/)
  - *VeriFun* (http://www.verifun.org/) – wird hier verwendet
  - ... und andere (http://www.atomseek.com/Math/Logic_and_Foundations/Software/index.html)

Was bedeutet "interaktives" Beweissystem?

- *Generell*: Ein Beweis wird mit Hilfe von *Benutzerunterstützung* berechnet.
- Das System kann als *Beweisassistent* aufgefaßt werden.
- Das System garantiert, daß *nur erlaubte Beweisschritte* angewendet werden.
- Das System kann *bestimmte* Beweisschritte *automatisch* durchführen.
- Insbesondere bei "kreativen" Beweisaufgaben ist *Benutzerhilfe* erforderlich.

Welche Arten von Interaktion (= Benutzereingriffe) gibt es?

(1) *Lemmata* (= "Hilfsaussagen") müssen "*erfunden*" und dann *formuliert* werden (=> **kreative Aufgabe!**)
  - *Gilt für alle interaktiven Systeme. Gewisse Lemmata können automatisch erzeugt werden, aber Interaktion prinzipiell unvermeidbar.*

(2) *Begriffe* (= mathematische Konzepte) müssen "*erfunden*" und dann *definiert* werden, um Lemmata wie unter (1) überhaupt formulieren zu können (=> **kreative Aufgabe!**)
  - *Gilt für alle interaktiven Systeme, gewisse Konzepte können automatisch erzeugt werden, aber Interaktion prinzipiell unvermeidbar.*

(3) Der Benutzer muß eine *Beweisregel explizit anwenden* (wenn das System bei der automatischen Berechnung eines Beweises scheitert).
  - *Gilt für alle interaktiven Systeme. Gewisse Beweise können automatisch berechnet werden, aber Interaktion prinzipiell unvermeidbar.*
  - *Systeme unterscheiden sich im* Grad der Automatisierung.

(4) Der Benutzer steuert das System durch Einstellung bestimmter Kontrollparameter ("versuche für diesen Beweis die Lemmata ... zu verwenden", "ignoriere für diesen Beweis die Lemmata ...", "versuche diesen Beweis durch Anwendung der Beweisregeln ... zu führen", u.s.w.)
  - *Unterschiedliche Konzepte für "Kalibrierung" eines Systems je nach Entwurfsphilosophie.*
  - *In* VeriFun *praktisch keine Eingriffe dieser Art.*
    - Grund: *"Kalibrierung" setzt fundierte Kenntnisse über interne Arbeitsweise eines Systems voraus.*
    - *Entwurfsphilosophie hier jedoch: Benutzer soll sich auf das Beweisproblem und nicht auf die Arbeitsweise des Systems konzentrieren.*
    - *Diese Entwurfsentscheidung erfordert jedoch einen entsprechend hohen Automatisierungsgrad.*

2 *VeriFun*

2.1 Das System

- *Name steht für*: **Veri***fication of* **Fun***ctional Programs*
- *Implementiert unter*: Intensiver studentischen Mitarbeit
  - 15 Studien- und Bachelorarbeiten
  - 14 Diplom- und Masterarbeiten
  - "ungezählte" wiss. Hilfskräfte
- *Ist ein*: Interaktives *Beweissystem* zur *Verifikation* von $\mathcal{L}$ - Programmen
- $\mathcal{L}$ ist eine einfache funktionale Programmiersprache
  - "funktional": Es gibt keine Schleifen und keine Zuweisungen – wir programmieren mit Funktionen (=> vgl. *Scheme*, GdI 1)
  - $\mathcal{L}$ entworfen zur *Modellierung von Algorithmen* + deren *Spezifikation*
- In $\mathcal{L}$ können definiert werden:
  - (freie) *Datentypen* (Datenstrukturen)
  - *Prozeduren*, die auf diesen Datentypen arbeiten
  - *Lemmata*, die Aussagen über Eigenschaften der Datentypen und Prozeduren treffen

Der Weg zu einem bewiesenermaßen korrekten $\mathcal{L}$ - Programm:

(1) Definiere die benötigten Datentypen

(2) Definiere die erforderlichen Prozeduren

(3) Definiere die Lemmata, die zur Spezifikation der (erwünschten) Eigenschaften der Datentypen und Prozeduren erforderlich sind

(4) Beweise (mit Systemunterstützung) die spezifizierten Eigenschaften (= definierten Lemmata)

2.2 *Fallstudie InsertionSort:* Modellierung und Spezifikation

- *Aufgaben*:
  (a) Implementierung eines Sortierverfahrens `isort` für Listen von natürlichen Zahlen nach dem Prinzip "Sortieren durch Einfügen"
  (b) Spezifikation der Korrektheitseigenschaften für "Sortieren" allgemein und damit auch für `isort` (=> geordnete Permutation)
  (c) Formaler Nachweis der Korrektheitseigenschaften

2.2.1 Datentypen

Ein **Datentyp** (kurz: Typ) wird in $\mathcal{L}$ definiert durch Angabe von

- *Typkonstruktoren*
- Datenkonstruktoren, auch Konstruktorfunktionen oder kurz *Konstruktoren* genannt sowie
- Selektorfunktionen, auch kurz *Selektoren* genannt, für jede Argumentposition eines Konstruktors.

Beispiel 2 *(Datentypen)*

- `structure bool <= true, false`
  definiert den *monomorphen* Datentyp `bool`.
  - "`bool`" ist Typkonstruktor,
  - "`true`" und "`false`" sind die (0-stelligen) Konstruktoren von "`bool`".

  Bedeutung:
  - Boolsche Werte (= Wahrheitswerte) werden repräsentiert durch `true` und `false`

Bemerkung 1

- `bool` ist in jedem $\mathcal{L}$ - Programm *vordefiniert*, s. Programfolder *Predefined*.
- `bool` darf *nicht* als *Argumenttyp* in Datentyp- und Prozedurdefinitionen verwendet werden (*Grund*: später).

- `structure nat <= 0, succ(pred : nat)`
  definiert den *monomorphen* Datentyp `nat`.
  - "`nat`" ist Typkonstruktor,
  - "`0`" ist ein (0-stelliger) Konstruktor von "`nat`", "`succ`" ist (1-stelliger) Konstruktor von "`nat`".
  - "`pred`" ist der Selektor von "`succ`" (sprich *predecessor* und *successor*).

  Bedeutung:
  - `nat` repräsentiert die Menge der *natürlichen Zahlen*
  - Natürliche Zahlen $0, 1, 2, \ldots, n, \ldots$ werden repräsentiert durch `0`, `succ(0)`, `succ(succ(0))`, $\ldots$, $\texttt{succ}^{(n)}\texttt{(0)}, \ldots$
  - Den Vorgänger einer natürlichen Zahl $\neq 0$ erhält man mit `pred`, also z.B. `pred(succ(succ(succ(0))))`$=$`succ(succ(0))`

Bemerkung 2

- `nat` ist in jedem $\mathcal{L}$-Programm vordefiniert, s. Programfolder *Predefined*.
- Im *PrettyPrint* von *VeriFun* wird
  $\Rightarrow$ "`nat`" als "$\mathbb{N}$",
  $\Rightarrow$ "`succ(...)`" als "$^{+}$`(...)`" und
  $\Rightarrow$ "`pred(...)`" als "$^{-}$`(...)`"
  angezeigt. Für Eingaben sind beide Schreibweisen zulässig.
- Natürliche Zahlen ≤ 42 können in üblicher Schreibweise eingegeben werden, also z.B. `3` anstatt `succ(succ(succ(0)))`.

- `structure list[@ITEM] <= ø, [infixr,100] ::(hd : @ITEM, tl : list[@ITEM])`
  definiert den *polymorphen* Datentyp `list[@ITEM]`.
  - "`list`" ist Typkonstruktor,
  - `@ITEM` ist eine *Typvariable*,
  - "`ø`" ist ein (0-stelliger) Konstruktor von "`list`", "`::`" ist (2-stelliger) Konstruktor von "`list`".
  - "`hd`" und "`tl`" sind die Selektoren von "`::`" (sprich *head* und *tail*)

  Bedeutung:

  - `list`$[$`@ITEM`$]$ repräsentiert **lineare Listen**
  - Die leere Liste wird durch "`ø`" repräsentiert, eine nicht-leere Liste $\langle e_1, e_2, \ldots, e_n\rangle$ durch e_1 `::` e_2 `::` $\ldots$ `::` e_n `::` `ø`
  - Das erste Element einer nicht-leeren Liste erhält man mit `hd`, also `hd(`e_1 `::` e_2 `::` $\ldots$ `::` e_n `::` `ø)`$= e_1$
  - Für eine nicht-leere Liste erhält man mit `tl` die Liste ohne das erste Element, also `tl(`e_1 `::` e_2 `::` $\ldots$ `::` e_n `::` `ø)`$= e_2$ `::` $\ldots$ `::` e_n `::` `ø`
  - Bei Verwendung von "`list`" darf die Typvariable `@ITEM` durch beliebige Typen (außer `bool`) ersetzt werden:

  - $*$ Mit `list[nat]` erhält man z.B. Listen von natürlichen Zahlen (repräsentiert durch `nat`). `list[nat]` ist ein *monomorpher* Datentyp.
  - $*$ Mit `list[list[@ITEM]]` erhält man z.B. Listen von (*polymorphen*) Listen. `list[list[@ITEM]]` ist ein *polymorpher* Datentyp.

Bemerkung 3

- "`[infixr,100]`" (=> Fixity) gibt an, daß die Konstruktorfunktion `::` in *Infixschreibweise* mit *Bindungspriorität* `100` verwendet wird.
- Die *Fixity* eines Funktionssymbols kann in *VeriFun* jederzeit *geändert* werden (=> Menue *Program*$\backslash$*Rename*).

Definition 1 *(Monomorphe und polymorphe Datentypen)*

- Ein Datentyp τ heißt *monomorph* gdw. τ *keine Typvariablen* enthält.
- Ein Datentyp τ heißt *polymorph* gdw. τ *Typvariable* enthält.

- `structure sexpr[@T] <= atom(data : @T), nil, cons(car : sexpr[@T], cdr : sexpr[@T])`
  definiert den *polymorphen* Datentyp `sexpr[@T]`.
  - "`sexpr`" ist Typkonstruktor,
  - `@T` ist eine *Typvariable*,
  - "`nil`" ist ein (0-stelliger) Konstruktor, "`atom`" ist (1-stelliger) Konstruktor und "`cons`" ist (2-stelliger) Konstruktor von "`sexpr`",
  - "`data`" ist Selektor von "`atom`", "`car`" und "`cdr`" sind die Selektoren von "`cons`".

  Bedeutung:

  - `sexpr`$[$`@T`$]$ repräsentiert *Binärbäume*, die Daten in den *Blättern* speichern.
  - Bezeichnungen aus der Programmiersprache **LISP**, `sexpr` steht für "symbolic expression" (=> *Scheme*, GdI 1)
  - "`nil`" bezeichnet den "leeren" Baum, `atom(d)` bezeichnet ein Blatt, das das Datum `d` speichert, `cons(l,r)` bezeichnet einen inneren Knoten mit linkem Sohn `l` und rechtem Sohn `r`
  - Das Datum eines Blatts erhält man mit `data`, also `data(atom(`d`))`$= d$

- Den linken / rechten Sohn eines inneren Knotens erhält man mit `car` / `cdr`, also $\texttt{car(cons(}l,r\texttt{))} = l$ und $\texttt{cdr(cons(}l,r\texttt{))} = r$
- Bei Verwendung von "`sexpr`" darf die Typvariable $\mathbb{@T}$ durch beliebige Typen (außer `bool`) ersetzt werden.
- **Modellierung:** Für $\texttt{sexpr}[\texttt{nat}]$ ist der Konstruktor `atom` ein *Typkonverter* $\texttt{nat} \rightarrow \texttt{sexpr}[\texttt{nat}]$
- *Informell:* `nil` ist ein `sexpr[nat]`, jede natürliche Zahl ist ein `sexpr[nat]` und $\texttt{cons}(\texttt{l},\texttt{r})$ ist ein `sexpr[nat]`, falls `l` und `r` vom Typ `sexpr[nat]` sind.
- Kann in $\mathcal{L}$ nicht modelliert werden, da dort *Disjunktheit von Typen* gefordert wird.
- Also werden anstatt natürlicher Zahlen $0, 1, 2, \ldots$ Ausdrücke der Form $\texttt{atom(0)}, \texttt{atom(1)}, \texttt{atom(2)}, \ldots$ verwendet.

Bemerkung 4

Der Datentyp $\texttt{sexpr}[\mathbb{@T}]$ *wird nicht für "Sortieren durch Einfügen" benötigt und dient nur als ein weiteres Beispiel für Datentypdefinitionen in* $\mathcal{L}$.

Durch Datentypdefinitionen implizit definierte Funktionssymbole

- Für jeden Datentyp $\tau \neq \texttt{bool}$ ist in $\mathcal{L}$ eine Funktion $\texttt{eq}_\tau\texttt{:}\tau \times \tau \rightarrow \texttt{bool}$ definiert.
  - *Bedeutung*: Gleichheit in τ – formal präzise definiert in **Kapitel 5**
  - *Schreibweise*: `x = y` anstatt $\texttt{eq}_\tau\texttt{(x, y)}$ (der Index "τ" wird bei Eingaben und im *PrettyPrint* weggelassen)
- Für jeden Datentyp τ ist in $\mathcal{L}$ eine Funktion $\texttt{if}_\tau\texttt{:bool} \times \tau \times \tau \rightarrow \tau$ definiert.
  - Bedeutung "offensichtlich" – formal präzise definiert in **Kapitel 5**
  - Der Index "τ" wird bei Eingaben und im *PrettyPrint* weggelassen
  - Erlaubte (gleichwertige) Schreibweisen:
    - $\texttt{if}\{x,y,z\}$ (=> verwendet im *PrettyPrint* von Lemmata)
    - $\texttt{if(}x,y,z\texttt{)}$
    - $\texttt{if}\ x\ \texttt{then}\ y\ \texttt{else}\ z\ \texttt{end\_if}$ (=> verwendet im *PrettyPrint* v. Prozeduren)
    - $\texttt{if}\ x\ \texttt{then}\ y\ \texttt{else}\ z\ \texttt{end}$
  - **Aber:** Keine Mischformen – verboten sind z.B. $\texttt{if}\{x,y,z\texttt{)}$, $\texttt{if(}x,y,z\}$, $\texttt{if}\{x,y,z\ \texttt{end}$, ...

2.2.2 Prozeduren

Eine **Prozedur** wird in $\mathcal{L}$ definiert durch Angabe

- des *Prozedurbezeichners* (= Name)
- der *formalen Parameter* mit ihren *Typen* ($\neq \texttt{bool}$)
- des *Ergebnistyps*
- des *Prozedurrumpfs* = Term über
  - den *formalen Parametern*,
  - dem *Prozedurbezeichner* (= Rekursion !) sowie
  - den *Funktionssymbolen*, die durch vorherige Definition von Datentypen und Prozeduren eingeführt wurden.

Beispiel 3 *(Prozeduren)*

- ```
  function [infixl,20] ≤(n : ℕ, m : ℕ) : bool <=
  if ?0(n)
   then true
   else if ?0(m)
          then false
          else ⁻(n) ≤ ⁻(m)
        end_if
  end_if
  ```

  berechnet die "übliche" $\leq$-Relation auf natürlichen Zahlen.

Bemerkung 5

- "`[infixl,20]`" (=> Fixity) gibt an, daß die Prozedur $\leq$ in *Infixschreibweise* mit *Bindungspriorität* 20 verwendet wird.
- "`?0(n)`" steht für "`n = 0`". Beide Schreibweisen sind bei Eingabe zulässig.
  ```

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

  fügt die natürliche Zahl `n` "geordnet" in die Liste `k` von natürlichen Zahlen ein.

Bemerkung 6

– "`?ø(k)`" steht für "`k = ø `". Beide Schreibweisen sind bei Eingabe zulässig.

- `function isort(k : list[ℕ]) : list[ℕ] <=`
  `if ?ø(k)`
  `then ø`
  `else insert(hd(k), isort(tl(k)))`
  `end_if`

  sortiert die Liste `k` von natürlichen Zahlen durch "geordnetes Einfügen".
- `function ordered(k : list[ℕ]) : bool <=`
  `if ?ø(k)`
  `then true`
  `else if ?ø(tl(k))`
  `then true`
  `else if hd(k) ≤ hd(tl(k))`
  `then ordered(tl(k))`
  `else false`
  `end_if`
  `end_if`
  `end_if`

  entscheidet, ob die Liste `k` von natürlichen Zahlen bzgl. $\leq$ geordnet ist.

2.2.3 Einschub: Warum Sonderregelung für `bool`?

- In $\mathcal{L}$ gibt es keine *Prädikatssymbole* sondern nur *Funktionssymbole*
- **Grund:** Einheitliche Beweisregeln; andernfalls eigener Regelsatz für Prädikate erforderlich
- **Daher:** Prädikate werden in $\mathcal{L}$ durch Funktionen mit Ergebnistyp `bool` repräsentiert
- **Beispiel:** $\leq$ und `ordered` sind keine *Prädikatssymbole*, sondern *Funktionssymbole* mit den Signaturen
  – $\leq$: `nat` × `nat` → `bool`
  – `ordered` : `list`[`nat`] → `bool`
- In der Prädikatenlogik dürfen *Prädikatssymbole nicht in Argumenten von Funktionsanwendungen* vorkommen (=> FGdI 2)
- **Daher in $\mathcal{L}$**:
  – `bool` darf *nicht* in Definitionen von *Datentypen* verwendet werden
  – `bool` darf *nicht* als Typ eines Parameters einer *Prozedur* verwendet werden
  – Typvariable dürfen nicht durch `bool` instantiiert werden
  – *Gleichheit* "`=`" ist auf `bool` *nicht* definiert

Wem das nicht paßt ...

- kann ja sein "eigenes `bool`" definieren, etwa

  `structure MyBool <= MyTrue, MyFalse`

- denn die Einschränkungen gelten nur für das vordefinierte `bool` aus *Predefined*

2.2.4 Lemmata

- In ***VeriFun*** werden *Formeln* durch Terme (= Ausdrücke) vom Typ `bool` repräsentiert.
- *Atomare* Formeln (kurz: Atome) sind
  - `true` und `false` sowie
  - Aufrufe von Prozeduren mit Ergebnistyp `bool`, z.B. `ordered(k)`.
- *Zusammengesetzte* Formeln nur durch $\texttt{if}_{\texttt{bool}}$:
  - $\texttt{if}\{a, b, c\}$ steht für $(a \Rightarrow b) \wedge (\neg a \Rightarrow c)$.

Damit beispielsweise:

- $\texttt{if}\{\texttt{a,true,b}\}$ steht für $a \vee b$
- $\texttt{if}\{\texttt{a,b,false}\}$ steht für $a \wedge b$
- $\texttt{if}\{\texttt{a,b,true}\}$ steht für $a \Rightarrow b$
- $\texttt{if}\{\texttt{a,false,true}\}$ steht für $\neg a$

Bemerkung 7

(1) Im *PrettyPrint* ¬a anstatt $\texttt{if}\{\texttt{a,false,true}\}$.
Bei Eingaben sind beide Schreibweisen zulässig.

(2) Wie üblich nennt man ein *Atom* sowie dessen *Negat* ein *Literal*, d.h. boolsche Terme der Form a oder ¬a mit "a ist Atom" werden auch *Literale* genannt.

Eigenschaften von Datentypen und Prozeduren werden in $\mathcal{L}$ durch sogenannte *Lemmata* (= Formeln mit einem "Namen") spezifiziert.

Ein **Lemma** wird in $\mathcal{L}$ definiert durch Angabe
- eines *Lemmabezeichners* (= Name)
- einer Liste ℓ von all-quantifizierten *Variablen* zusammen mit ihren *Typen*
- des *Lemmarumpfs* = *boolschem Term* über
  - den *Variablen aus* ℓ sowie
  - den *Funktionssymbolen*, die durch vorherige Definition von Datentypen und Prozeduren eingeführt wurden.

Beispiel 4 *(Lemmata)*

- `lemma isort_sorts <= ∀ k : list[ℕ] ordered(isort(k))`

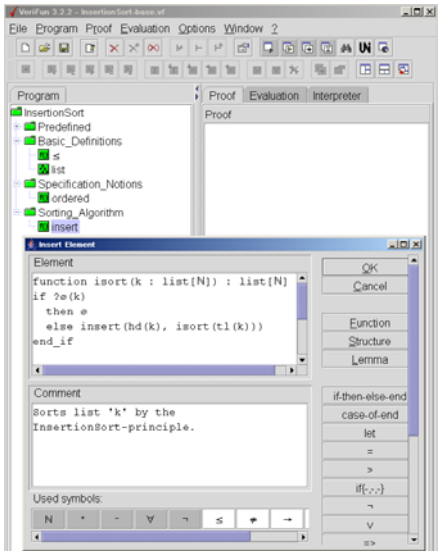
  gibt an, daß das Ergebnis von `isort` angewendet auf eine *beliebige* Liste `k` von natürlichen Zahlen eine geordnete Liste (i.S.v. `ordered`) ist.

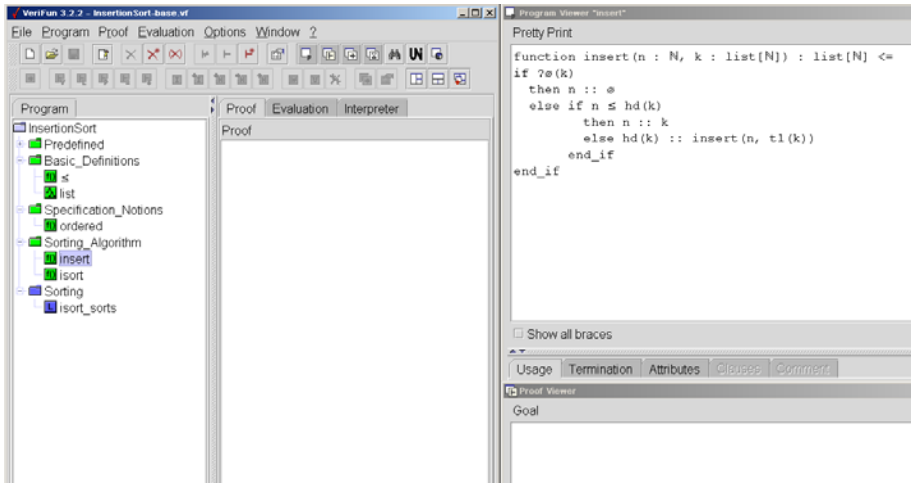
Bemerkung 8 *Anstatt "∀" darf alternativ "*`all`*" geschrieben werden.*

Zusammenfassung
- *Datentypen*
  - Wir haben die monomorphen Datentypen `bool` und `nat` sowie den polymorphen Datentyp `list` definiert.
  - Damit steht insbesondere auch der monomorphe Datentyp `list[nat]` zur Verfügung.
  - Somit stehen alle Datentypen zur Verfügung, die für "Sortieren durch Einfügen" erforderlich sind.
- *Prozeduren*
  - Wir haben die Prozeduren $\leq$, `insert` und `isort` definiert und damit "Sortieren durch Einfügen" implementiert.
  - Wir haben die Prozedur `ordered` definiert. Diese Prozedur benötigen wir *nicht* zur Implementierung von "Sortieren durch Einfügen".
    Prozedur `ordered` ist vielmehr ein *Begriff der Spezifikation*: Mit `ordered` *spezifizieren* wir die Ordnungseigenschaft von `isort`.
- *Lemmata*
  - Wir haben das Lemma `isort_sorts` formuliert. Dieses Lemma *spezifiziert*, daß Prozedur `isort` die für Sortieralgorithmen geforderte *Ordnungseigenschaft* erfüllt.

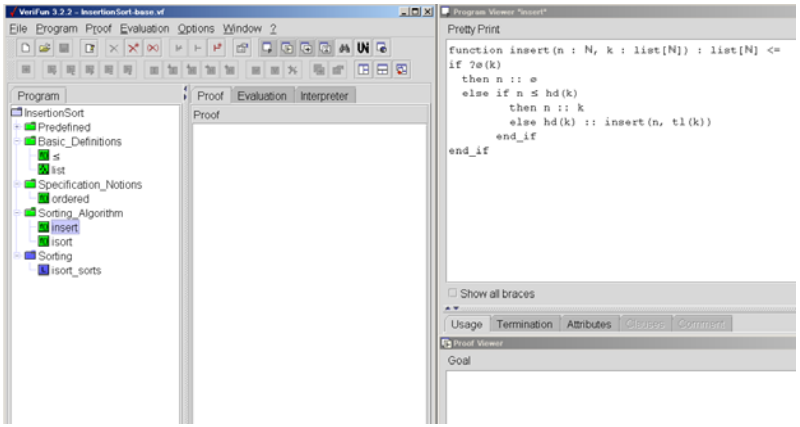
2.3 *Fallstudie InsertionSort:* Formale Verifikation

Neue Programmelemente (= Datentypen, Prozeduren, Lemmata) werden mittels *Program**Insert Element* editiert:

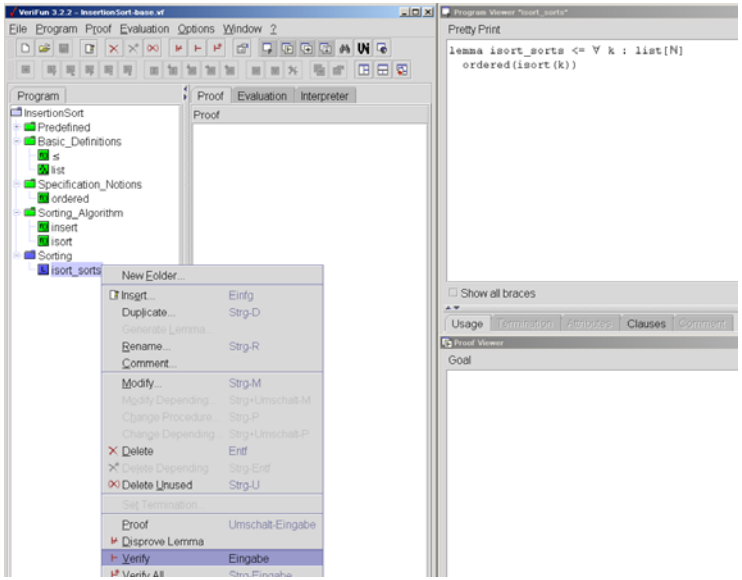




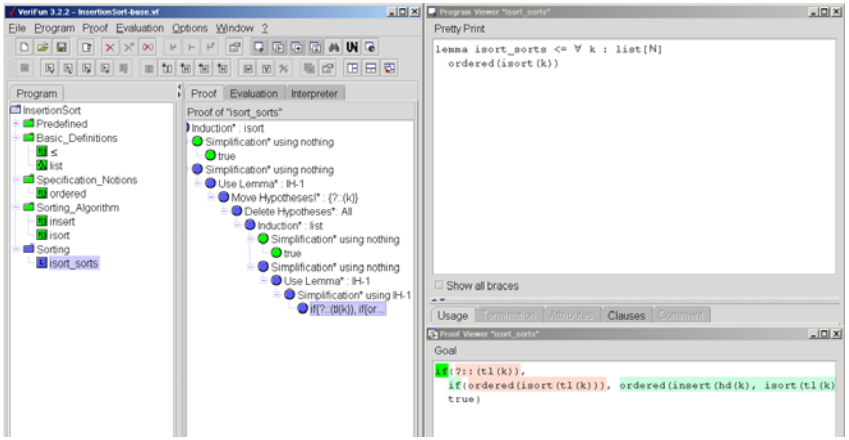
- Das aktuelle $\mathcal{L}$-Programm wird im *Program Window* angezeigt
- Im *Program Window* selektierte Programmelemente werden im *Program Viewer* angezeigt



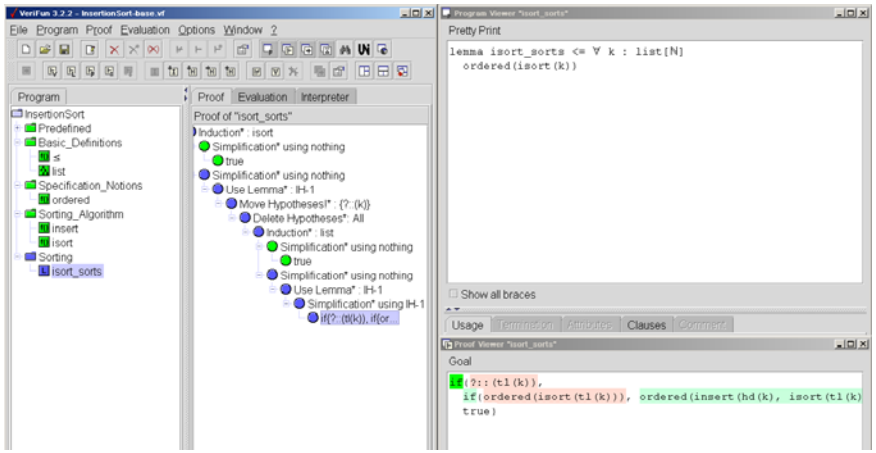
- Bedeutung der Farben (= *Status* von Programmelementen)
  - Datentypen “grün” (Status *verified*) => gilt immer
  - Prozedur “grün” (Status *verified*) => Terminierung wurde bewiesen
  - Lemma “grün” (Status *verified*) => Lemma wurde bewiesen
  - Lemma “blau” (Status *ready*) => Beweis kann begonnen/fortgesetzt werden



- Bei Aufruf von *Program**Verify* startet das System einen Beweisversuch

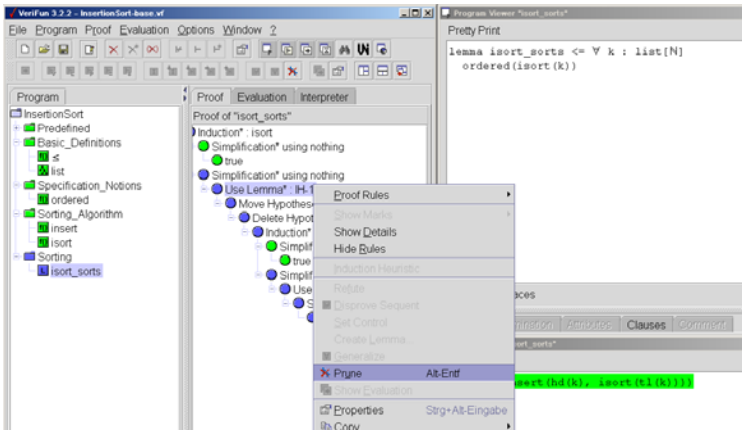


- Ergebnis ist ein *Beweisbaum*, der im *Proof Window* angezeigt wird
- Die inneren Knoten des Beweisbaums sind mit Namen der angewendeten *Beweisregeln* markiert
- Die Blätter des Beweisbaums sind mit *Beweiszielen* markiert.
- Beweisziele von im *Proof Window* selektierten Knoten werden im *Proof Viewer* angezeigt.



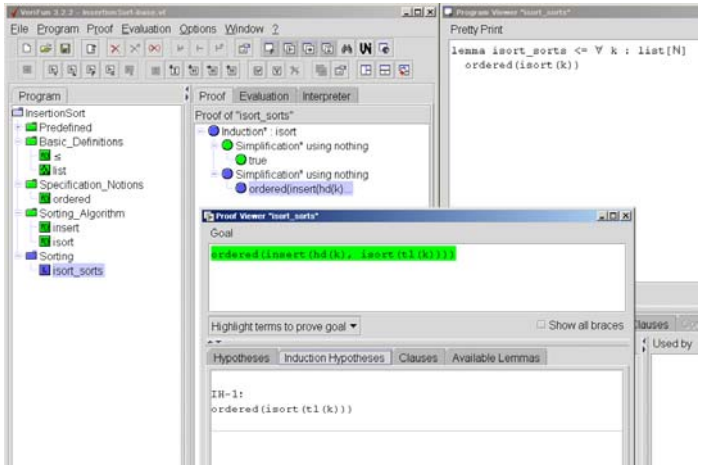
Was ist hier passiert?

- Das System versucht einen *Induktionsbeweis* (s. Wurzelknoten)
- Der *Induktionsanfang* (=> Basisfall) konnte mit der Beweisregel *Simplification* bewiesen werden.
- Im *Induktionsschritt* (=> Schrittfall) scheitert das System – erfolgloser Versuch eines weiteren Induktionsbeweises



Was jetzt?

- Wir schneiden den überflüssigen Teil des Beweisbaums mit *Proof\Prune* ab
- Wir analysieren das Beweisziel des Beweisbaumblatts



- Zu zeigen ist: `ordered(insert(hd(k), isort(tl(k))))`
- Zur Verfügung steht Induktionshypothese: `ordered(isort(tl(k)))`
- Es gilt: `k` $\neq$ `ø` (denn wir sind im Schrittfall)

Zu zeigen ist also

$$\forall k{:}list[nat] \quad k \neq \emptyset \wedge ordered(isort(tl(k))) \Rightarrow ordered(insert(hd(k), isort(tl(k)))) \tag{3}$$

- Hier benötigen wir ein "Hilfs"-Lemma **!**

2.4 Einschub: Durch Generalisierung Lemmata "erfinden"

- *Generalisierung* ist eine *Methode* zum "Erfinden" *nützlicher Lemmata*
- Ein *Term* wird generalisiert durch:
  (a) *Ersetzen* von *Termen* durch *Variable*
  (b) *Eliminination* von *Prämissen* in Implikationen

Im Beispiel:

(1) $isort(tl(k)) \rightsquigarrow \ell, hd(k) \rightsquigarrow n$ (2) weg mit "$k \neq \emptyset$"

Man erhält:

$$\forall n{:}nat, \ell{:}list[nat]\ ordered(\ell) \Rightarrow ordered(insert(n, \ell)) \tag{4}$$

- Formel (4) ist eine *Generalisierung* von Formel (3)
- Forderung an jede Generalisierung φ' von φ:
  Es gilt $\models \varphi' \Rightarrow \varphi$, d.h. $\varphi' \Rightarrow \varphi$ ist *allgemeingültig* (=> FGdI 2)
- **Damit**: Mit Beweis von φ' erhält man auch einen Beweis für φ

Achtung: Man kann auch "*Übergeneralisieren*"

Beispiel 5 *(Übergeneralisierung)*

(1) $insert(hd(k), isort(tl(k))) \rightsquigarrow h, isort(tl(k)) \rightsquigarrow \ell$

(2) weg mit "$k \neq$ø"

Ergebnis:

$$\forall \ell, h{:}list[nat]\ ordered(\ell) \Rightarrow ordered(h) \qquad (5)$$

- Zwar gilt $\models (5) \Rightarrow (3)$ (und damit ist (5) Generalisierung von (3))
- aber (5) ist *falsch*, also *nicht beweisbar*,
- also ist Generalisierung (5) *unbrauchbar*.

Warum sind Generalisierungen erforderlich?

- Durch Ersetzung von Termen t durch Variable x werden *Induktionsbeweise* über x *ermöglicht*
- Durch Elimination von Prämissen werden *Beweisziele vereinfacht*
  – damit *einfachere Beweise* oder sogar überhaupt erst *Beweisbarkeit*

Gefundene Generalisierung:

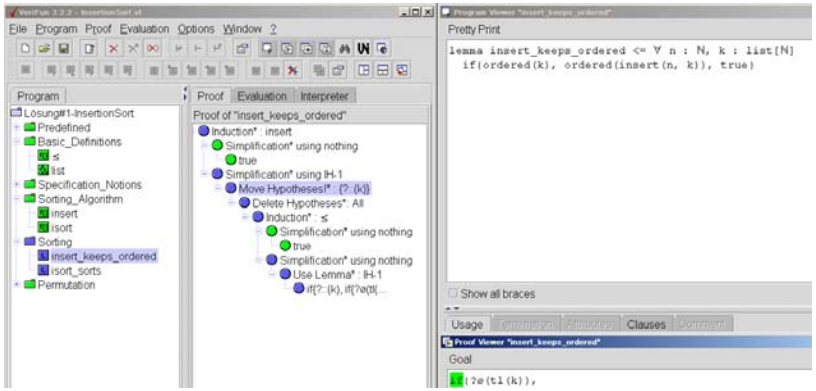
$$\forall n{:}nat, \ell{:}list[nat]\ ordered(\ell) \Rightarrow ordered(insert(n, \ell)) \qquad (4)$$

- **Behauptet:** Einfügen mittels `insert` in *geordnete* Liste ergibt *geordnete* Liste
- **Damit:** Generalisierung (4) repräsentiert *Korrektheitsbedingung* für `insert`

Jetzt weiter:

- Generalisierung (4) als $\mathcal{L}$-Lemma formulieren (=> *Program**Insert Element*)
- Beweisversuch mittels *Program**Verify* starten
- bei Mißerfolg Ergebnis analysieren

2.5 *Fallstudie InsertionSort:* Formale Verifikation (Fortsetzung 1)

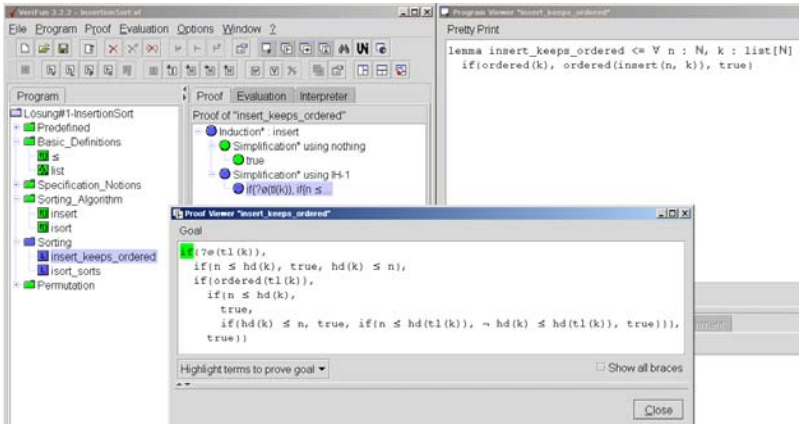


Ergebnis wie zuvor:

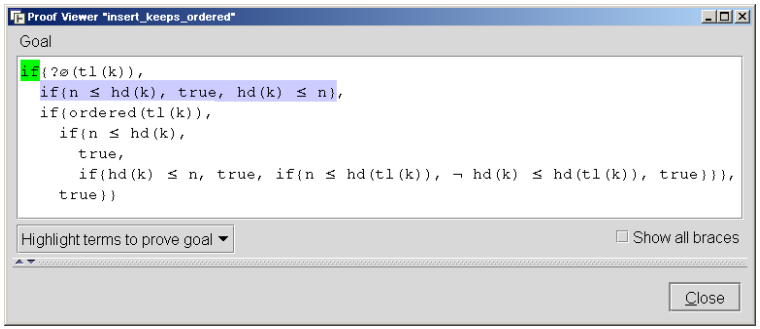
- Das System versucht einen *Induktionsbeweis* (s. Wurzelknoten)
- Der *Induktionsanfang* (= Basisfall) konnte mit der Beweisregel *Simplification* bewiesen werden.
- Im *Schrittfall* scheitert das System – erfolgloser Versuch eines weiteren Induktionsbeweises

Also:

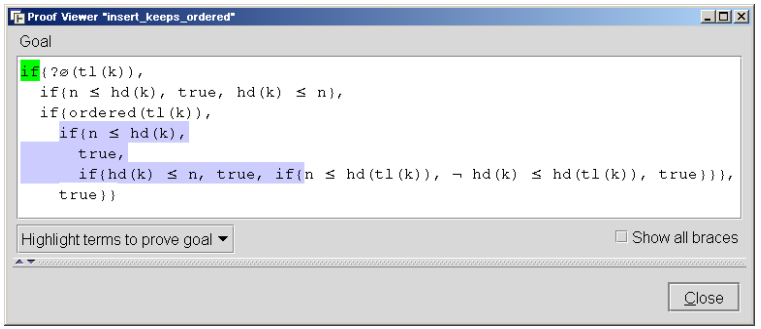
- Beweisbaum abschneiden
- Beweisziel analysieren
- Generalisieren



Analyse von



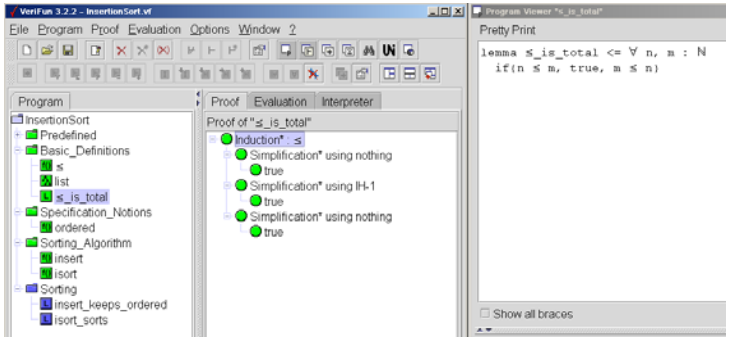
- Sieht komplizierter aus als es eigentlich ist **!**
- *then*-Teil des äußersten `if :` `if{n ≤ hd(k), true, hd(k) ≤ n}`
- Generalisiert mit $hd(k) \rightsquigarrow m$: `if{n ≤ m, true, m ≤ n}`
- In "üblicher" Schreibweise: $n \leq m \vee m \leq n$
- Also erforderliches Lemma: *Relation* $\leq$ *ist* ***total***



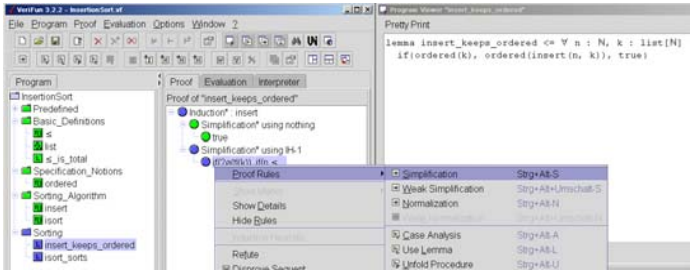
- *then*-Teil des *else*-Teils des äußersten `if :`
  `if{n ≤ hd(k), true, if{hd(k) ≤ n, true, if{...}}}`
- Generalisiert mit $hd(k) \rightsquigarrow m$:
  `if{n ≤ m, true, if{m ≤ n, true, if{...}}}`
- In "üblicher" Schreibweise: $n \leq m \vee m \leq n \vee if\{\ldots\}$
- Also erforderliches Lemma auch hier: *Relation* $\leq$ *ist* ***total***

Jetzt weiter:

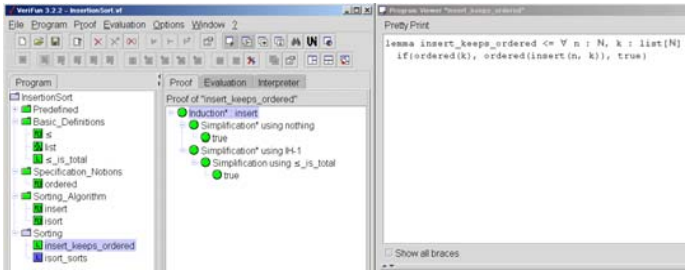
- "$\leq$ ist *total*" als $\mathcal{L}$-Lemma formulieren (=> *Program**Insert Element*)
- Beweisversuch mittels *Program**Verify* starten



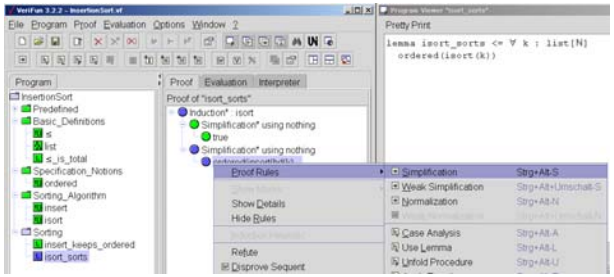
- *Bewiesen*, also zurück zum letzten offenen Beweisproblem



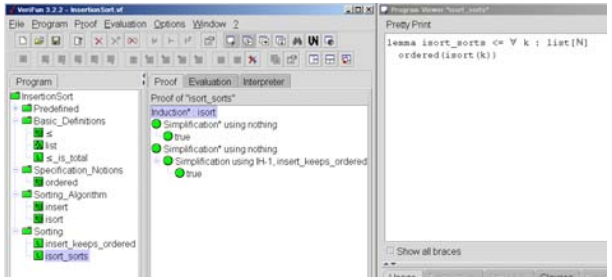
- Beweisregel *Simplification* anwenden



- *Bewiesen*, also zurück zum letzten offenen Beweisproblem



- Beweisregel *Simplification* anwenden



- Bewiesen, **fertig !**

2.6 *Fallstudie InsertionSort:* Zwischenbilanz

2.6.1 Wie sind wir vorgegangen?

(1) Wir haben "Sortieren durch Einfügen" als $\mathcal{L}$-Prozedur `isort` implementiert

(2) Mit der Prozedur `ordered` und dem Lemma `isort_sorts` haben wir eine Korrektheitseigenschaft von `isort` (= Ergebnisliste ist geordnet) spezifiziert

(3) Der automatische Beweisversuch von `isort_sorts` scheitert.
Grund: Es fehlt eine Korrektheitsaussage über `insert`

(4) Die fehlende Korrektheitsaussage haben wir nach Analyse des Beweisfehlschlags mittels *Generalisierung* ermittelt und als $\mathcal{L}$-Lemma `insert_keeps_ordered` formuliert

(5) Der automatische Beweisversuch von `insert_keeps_ordered` scheitert ebenfalls. *Grund*: Es fehlt eine Korrektheitsaussage über $\leq$

(6) Die fehlende Korrektheitsaussage haben wir nach Analyse des Beweisfehlschlags mittels *Generalisierung* ermittelt und als $\mathcal{L}$-Lemma $\leq$ `_is_total` formuliert

(7) Das $\mathcal{L}$-Lemma $\leq$ `_is_total` kann automatisch bewiesen werden.

(8) Mit dem Beweis von $\leq$ `_is_total` können wir den Beweis von `insert_keeps_ordered` erfolgreich abschließen

(9) Mit dem Beweis von `insert_keeps_ordered` können wir schließlich den Beweis von `isort_sorts` erfolgreich abschließen

Was war hier "schwer" und was war "einfach"?

- Schritte (4) und (6) (= "nützliches" Lemma erfinden) sind die "kreativen" Schritte – **hier muß man nachdenken !**
- Der Rest ist Routine

2.6.2 Warum müssen wir Lemmata "erfinden"?

- Ausgangspunkt ist immer eine "Haupt"-Prozedur `p` (z.B. `isort`) und deren spezifizierte Korrektheitseigenschaften (z.B. `isort_sorts`).
- Prozeduren `p` verwenden i.A. "Hilfs"-Prozeduren `q` (`isort` verwendet Prozedur `insert`, `insert` verwendet Prozedur $\leq$).
- Prozeduren `q` müssen gewissen Korrektheitsbedingungen genügen, um die Korrektheit der aufrufenden Prozedur `p` sicherzustellen: Wäre dies nicht so, so könnte man ja `p` mit *jeder beliebigen* Prozedur `q` als "Hilfs"-Prozedur implementieren – also bräuchte man `q` überhaupt nicht **!**
- Wir müssen also Lemmata erfinden, die diejenigen Eigenschaften einer Prozedur `q` repräsentieren, die für den Korrektheitsnachweis von `p` erforderlich sind.
- Das gilt für alle Systeme: Das korrekte Funktionieren eines Systems (hier: *Prozedur*) ist abhängig vom korrekten Funktionieren aller Systembausteine (hier: *"Hilfs"-Prozeduren*)

Typischer "Anfängerfehler":

- Übersehen von erforderlichen Lemmata
- **Beispiel $\leq$:** Das kennen wir schon aus der Schule – "$\leq$" ist *reflexiv*, *transitiv*, *antisymmetrisch* und *total*, das ist ja klar.
- **Frage:** Warum verwendet dann das System nicht die Totalität von $\leq$ im Beweis von `insert_keeps_ordered` ???
- **Antwort:** $\leq$ ist nicht die Relation, die wir alle kennen, sondern eine $\mathcal{L}$-Prozedur, die wir geschrieben haben. Das System *errät nicht* Eigenschaften von $\leq$. Diese muß ein Benutzer als $\mathcal{L}$-Lemma formulieren – erst *nach deren Beweis* werden solche Lemmata verwendet.

Bemerkung 9

Man muß nicht für jede Verifikationsaufgabe "das Rad neu erfinden".

- Prozedur $>$ (für die übliche $>$-Relation auf $\mathbb{N}$) und Lemmata darüber sind schon im Folder *Predefined* vordefiniert.
- Man kann *Beweisbibliotheken* anlegen und daraus Definition und Lemmata mit ihren Beweisen in eine aktuelle Fallstudie *importieren* (=> Menue *File\Open Import*)

2.7 *Fallstudie InsertionSort:* Formale Verifikation (Fortsetzung 2)

Wir sind noch nicht fertig:

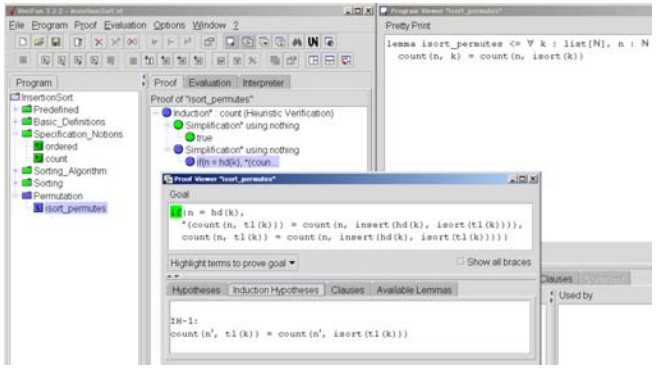
- jetzt Nachweis der Permutationseigenschaft von `isort`
-
```
function count(i : @ITEM, k : list[@ITEM]) : ℕ <=
if ?ø(k)
 then 0
 else if i = hd(k)
        then +(count(i, tl(k)))
        else count(i, tl(k))
       end_if
end_if
```

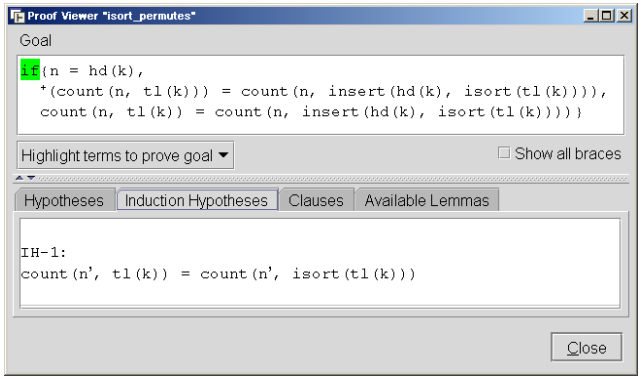
  berechnet die Anzahl der Vorkommen von `i` in der Liste `k`
- `lemma isort_permutes <=`
  `∀ k:list[ℕ], n:ℕ  count(n, k) = count(n, isort(k))`

  gibt an, daß jede natürliche Zahl `n` genauso oft in einer Liste `k` vorkommt, wie in der sortierten Liste `isort(k)`.

Also jetzt:

- Prozedur `count` und Lemma `isort_permutes` eingeben (=> Menue *Program\Insert Element*)
- Mit *Program\Verify* Beweisversuch für `isort_permutes` starten
- Nach Fehlschlag überflüssigen Teil des Beweisbaums abschneiden (=> Menue *Proof\Prune*)





- Jetzt Beweisziel analysieren
- Generalisieren und damit erforderliches Lemma "erfinden"
- Beweisversuch mit *Program\Verify* starten (sollte automatisch gelingen)
- Beweis von `isort_permutes` mit *Proof\Proof Rules\Simplification* fertigstellen (sollte gelingen)
- **Fertig !**

Tip:

- Man benötigt ein Lemma über `count` und `insert`. Der gesuchte Lemmarumpf hat die Form

```
if{?1,
   count(n, insert(m, k)) = ?2,
   count(n, insert(m, k)) = ?3}
```

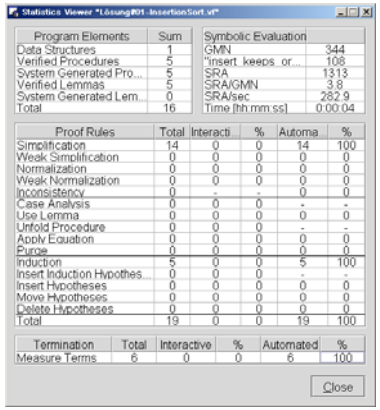

2.8 *Fallstudie InsertionSort:* Benutzeraufwand

Benutzeraufwand *InsertionSort*

Modellierung und Spezifikation:

- 1 Datentyp (=> `list`)
- 5 Prozeduren (=> $\leq$, `insert`, `isort`, `ordered`, `count`)
- 2 Lemmata (=> `isort_sorts`, `isort_permutes`)

Benutzerinteraktionen => Menue *Window\Open Statistics Viewer* :



Also:

- 3 Lemmata mußten “erfunden” werden
- alle Beweise *automatisch* (wenn erforderliche Lemmata gegeben)

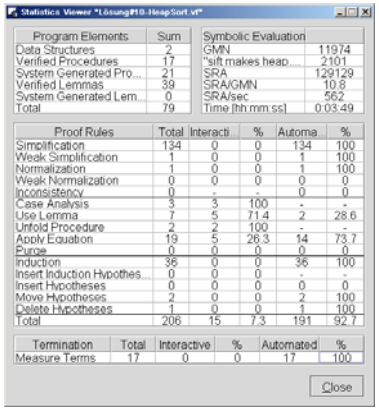
=> **Das war einfach !**

Zum Vergleich: **Benutzeraufwand** *HeapSort*

Modellierung und Spezifikation:

- 2 Datentypen (=> `list`, `tree`)
- 10 Prozeduren
- 2 Lemmata (=> `heapsort sorts`, `heapsort permutes`)

Benutzerinteraktionen => Menue *Window\Open Statistics Viewer* :



Program Elements	Sum
Data Structures	2
Verified Procedures	17
System Generated Pro...	21
Verified Lemmas	39
System Generated Lem...	0
Total	79

Symbolic Evaluation	
GMN	11974
"sift makes heap....	2101
SRA	129129
SRA/GMN	10.8
SRA/sec	562
Time [hh:mm:ss]	0:03:49

Proof Rules	Total	Interacti	%	Automa	%
Simplification	134	0	0	134	100
Weak Simplification	1	0	0	1	100
Normalization	1	0	0	1	100
Weak Normalization	0	0	0	0	0
Inconsistency	0	-		0	0
Case Analysis	3	3	100	-	-
Use Lemma	7	5	71.4	2	28.6
Unfold Procedure	2	2	100	-	-
Apply Equation	19	5	26.3	14	73.7
Purge	0	0	0	0	0
Induction	36	0	0	36	100
Insert Induction Hypothes...	0	0	0	-	-
Insert Hypotheses	0	0	0	0	0
Move Hypotheses	2	0	0	2	100
Delete Hypotheses	1	0	0	1	100
Total	206	15	7.3	191	92.7

Termination	Total	Interactive	%	Automated	%
Measure Terms	17	0	0	17	100

Also:

- 7 Prozeduren mußten “erfunden” werden (um Lemmata zu formulieren)
- 37 Lemmata mußten “erfunden” werden
- 15 Beweisregeln mußten angegeben werden (wenn alle Lemmata gegeben)

  => **Das ist schon aufwendiger !**