Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document
D-94-08

IGLOO 1.0
Eine grafikunterstützte Beweisentwicklungsumgebung

Benutzerhandbuch

Harald Feibel

November 1993

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, SEMA Group, and Siemens. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct *systems with technical knowledge and common sense* which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- Intelligent Engineering Systems
- Intelligent User Interfaces
- Computer Linguistics
- Programming Systems
- Deduction and Multiagent Systems
- Document Analysis and Office Automation.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Dr. Dr. D. Ruland
Director

IGLOO 1.0 - Eine grafikunterstützte Beweisentwicklungsumgebung Benutzerhandbuch

Harald Feibel

DFKI-D-94-08

Diese Arbeit wurde finanziell unterstützt durch das Bundesministerium für Forschung und Technologie (FKZ ITW 9000 8).

© Deutsches Forschungszentrum für Künstliche Intelligenz 1994

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission

IGLOO 1.0

Eine grafikunterstützte Beweisentwicklungsumgebung

Benutzerhandbuch

Harald Feibel

November 1993

Zusammenfassung

Das System **IGLOO** ist eine interaktive, grafikunterstützte Beweisentwicklungsumgebung für Sequenzenkalküle. Es besteht aus einer Inferenz- und einer Präsentationskomponente. Die Inferenzkomponente beinhaltet einen automatischen Beweiser und einen Mechanismus zum interaktiven und taktikbasierten Beweisen. Sequenzenkalküle unterschiedlicher Logiken können zum Beweisen aktiviert werden. Die Präsentationskomponente dient zur grafischen Darstellung von Ableitungsbäumen. Das interaktive Beweisen ist durch die spezielle Grafikunterstützung ausgezeichnet. Die Taktikgenerierung in der angebotenen Taktiksprache wird durch Möglichkeiten der Beweisanalyse unterstützt. Alle Systemfunktionen werden auf der Fensteroberfläche mit der Maus ausgelöst.

Das System wurde auf einer SOLBOURNE Workstation unter UNIX in SICStus-PROLOG implementiert. X Window System bildet die Grundlage für die fensterorientierte Systemoberfläche. Der Quellcode umfaßt etwa 890 KB.

IGLOO wurde am Deutschen Forschungszentrum für Künstliche Intelligenz (DFKI) in Saarbrücken innerhalb des von Prof. Dr. Wolfgang Wahlster geleiteten Projektes PHI (Planbasierte Hilfesysteme) im Rahmen einer von Dipl.-Inform. Mathias Bauer betreuten Diplomarbeit entwickelt und implementiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Kurzbeschreibung des Systems IGLOO

Das System **IGLOO**[1] wurde als Diplomarbeit im Rahmen des Projektes PHI[2] am DFKI[3] in Saarbrücken implementiert. Es ist ein Beweiser für Sequenzenkalküle[4], mit dem man sowohl automatisch als auch interaktiv Beweise führen kann. Der interaktive Modus ist dabei insbesondere dadurch gekennzeichnet, daß Taktiken benutzt werden können und daß eine grafische Unterstützung gegeben ist. Die Architektur von **IGLOO** wird durch Abbildung 1 beschrieben.

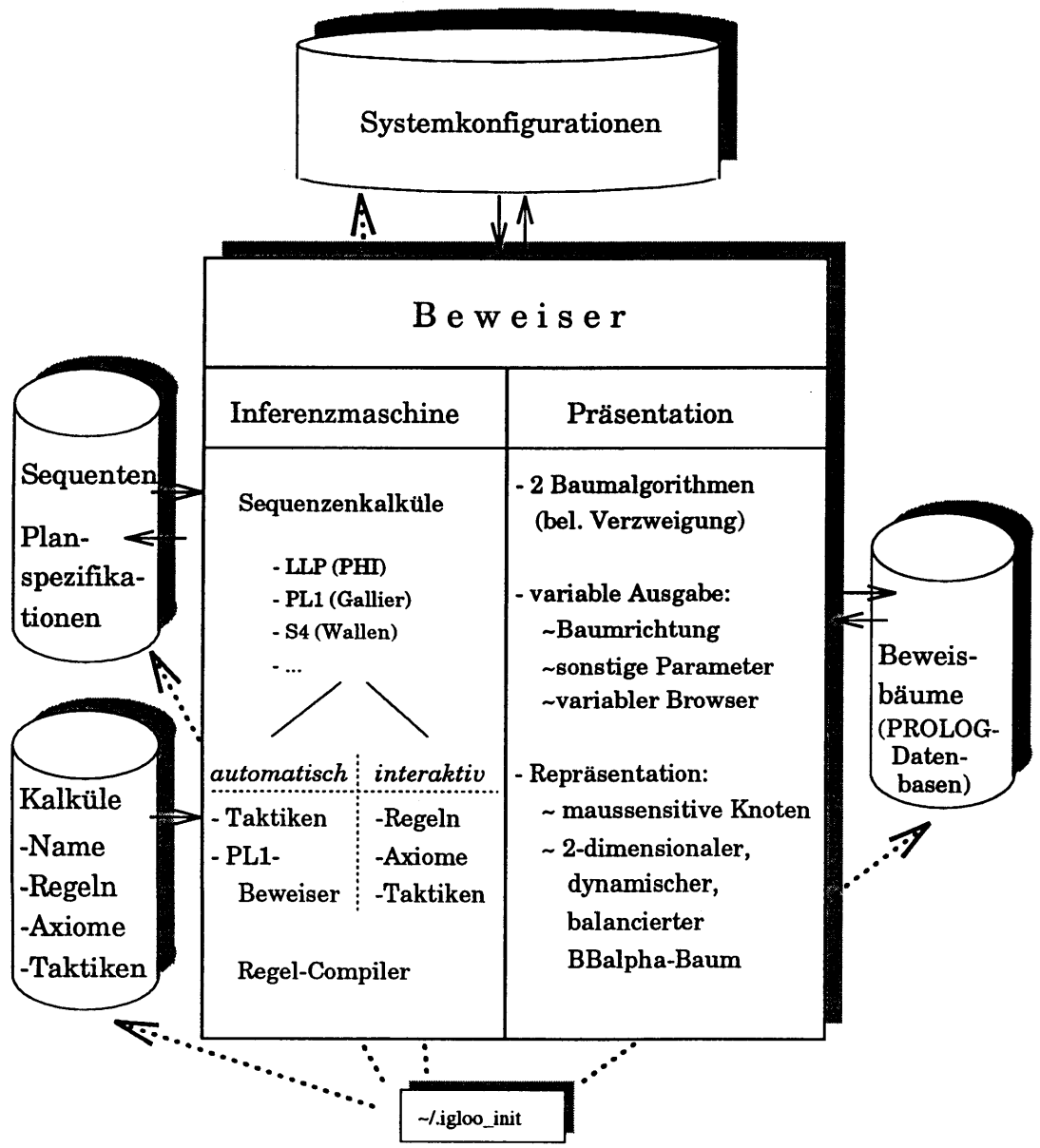


Abbildung 1: Die Architektur von **IGLOO**

[1] An Interactive Graphic Supported Sequent CaLculus TheOrem PrOver
[2] Planbasierte HIlfesysteme (vgl. [BBD+92])
[3] Deutsches Forschungszentrum für Künstliche Intelligenz
[4] Zur Einführung in Sequenzenkalküle, vgl. [Gal87], [Fit90].

Das System besteht aus einem Kern, dem **Beweiser**, und aus einer Reihe von externen Datenbasen mit Schnittstellen zu ihnen. Der Beweiser wiederum besteht aus einer **Inferenzmaschine** und einer **Präsentationskomponente**. Im folgenden werden diese Bestandteile kurz erläutert.

- Die Inferenzmaschine

  Die Inferenzmaschine besteht aus einem kalkülunabhängigen Mechanismus, der es erlaubt, unterschiedliche Mengen von Regeln, Axiomen und Taktiken zum Führen von Beweisen in unterschiedlichen Logiken zu verwenden. In Abbildung 1 sind Logiken aufgeführt, für die entsprechende Kalküle zur Verfügung gestellt werden. In den Abschnitten 2.1, 2.2, und 2.3.3 wird beispielhaft auf den Kalkül für PL1 Bezug genommen. Der Kalkül für S4[5] verhält sich analog. Das Arbeiten mit Kalkülen für LLP[6] wird in Abschnitt 2.4 erläutert. Das Integrieren weiterer Kalküle ist sehr einfach und wird im Abschnitt 2.3.3, "Verwalten von Kalkülen", behandelt.

  Beweise können **automatisch** oder **interaktiv** geführt werden. Besonderes Merkmal beim interaktiven Beweisen ist die Möglichkeit der Anwendung von **Taktiken**.

- Schnittstellen der Inferenzmaschine zu externen Datenbasen

  Vom System aus kann durch Zugriff auf eine eigene Datenbasis die Verwaltung von Sequenten vorgenommen werden, welche deren Abspeichern, Laden und Löschen umfaßt. Ferner kann während des Systemlaufes auch zwischen verschiedenen Kalkülen gewechselt werden, indem auf die Kalküldatenbasis zugegriffen wird. Beim Laden der Regeln eines Kalküles wird ein spezieller Regelcompiler benutzt, der die Regeln von einer abstrakten, benutzerfreundlichen Syntax in Prologprozeduren übersetzt.

- Die Präsentationskomponente

  Die Präsentationskomponente wird zur grafischen Ausgabe von Ableitungsbäumen benutzt. Dabei sind bei der Gestaltung des Layouts der Bäume viele Freiheitsgrade gegeben. Die Knoten der Bäume sind maussensitiv, so daß viele von **IGLOO** angebotene Funktionen durch einfache Selektion von Knoten[7] angestoßen werden können.

- Schnittstelle der Präsentationskomponente zu externer Datenbasis mit Ableitungsbäumen

  Erzeugte Beweisbäume können auf externen Prolog-Datenbasen abgespeichert werden. Dies bietet die Möglichkeit, Beweise zu "konservieren", d.h. die zugehörigen Bäume können jederzeit wieder angezeigt werden, etwa zu Analyse- oder Modifizierungszwecken.

- Systemkonfigurationen

[5] entnommen aus [Wal90]

[6] **L**ogical **L**anguage for **P**lanning (vgl. [BD93])

[7] Zu BB[α]-Bäumen, hier zur eff. Verwaltung von Knotenkoordinaten benutzt, vgl. [Meh84].

Das gesamte **IGLOO**-System ist zu jedem Zeitpunkt in seinem Verhalten durch eine Menge von Parametereinstellungen, die sowohl die Inferenz- als auch die Grafikkomponente betreffen, bestimmt. Einstellungen, die dem Benutzer besonders geeignet erscheinen, können auf Dateien gesichert und später wiederverwendet werden.

- Initiale Umgebung des Systems

  Die beim Programmstart aktive Umgebung des Systems kann vom jeweiligen Benutzer durch die Datei *.igloo_init*, die in seiner Home-Directory stehen muß, bestimmt werden (vgl. Abschnitt 1.3). Darin wird im wesentlichen festgelegt, welche Files mit Sequenten zur Verfügung gestellt werden, welche Kalküle ladbar sind, auf welche Directories mit abgespeicherten Beweisbäumen zugegriffen werden kann und welche Files mit Systemkonfigurationen ladbar sind.

1.2 In der weiteren Darstellung benutzte Notationen

- Da die Bedienung von **IGLOO** im wesentlichen mit der Maus durch Anklicken von Buttons auf der Benutzeroberfläche erfolgt, wird im Text für die Buttonbedienung folgende Notation benutzt:

[redacted]

Ein Beispiel: "... mit ≪*statistics*≫ erhält man nähere Informationen zum geführten Beweis ..."

- Unter dem Begriff *Window* oder *Fenster* wird im Text immer ein (komplexes) Fenster verstanden, das aus mehreren Objekten wie Buttons, Views, ... bestehen kann.

- Unter dem Begriff *View* wird eine Komponente eines Fensters verstanden, in der grafische Objekte wie Linien, Schrift, ... enthalten sein können. Oft sind Views dadurch erkennbar, daß sie von zwei Scrollbars umrandet werden.

- Ein *Buttonstate* ist eine Sammlung von meistens kreisförmigen Buttons. Er repräsentiert alle möglichen Werte eines Parameters, da für jeden Wert ein Button vorhanden ist. Der aktuell eingestellte Wert, ist durch eine grafische Markierung des betreffenden Buttons gekennzeichnet. Bei Anklicken eines Buttons des Buttonstates wird die Markierung von dem zuvor markierten Button gelöscht und auf den selektierten Button übertragen, so daß immer nur der zuletzt angeklickte Button markiert ist und somit den aktuellen Wert des durch den Buttonstate repräsentierten Parameters anzeigt.

- Fenster haben häufig einen Namen in ihrem Kopfteil. Im Text wird dieser Name in Anführungszeichen gesetzt, wenn auf das entsprechende Fenster Bezug genommen wird.

- In der Einleitung und auch im weiteren Text ist von *Sequenten* die Rede. Formal hat ein Sequent folgende Form:

$$\phi_1, \phi_2, ..., \phi_n \longrightarrow \psi_1, ..., \psi_m$$

Dabei sind die Formeln ϕ_i im Antezedens konjunktiv und die Formeln ψ_j im Sukzedens disjunktiv verknüpft. Der Sequenzenpfeil "$\longrightarrow$" kann intuitiv als Implikation interpretiert werden. Die genaue Syntax ist auf Seite 15 unter Punkt **c)** anhand einer kontextfreien Grammatik gegeben.

1.3 Installation und Start des Systems

Das System **IGLOO** wurde auf einer SOLBOURNE Workstation unter UNIX in SICStus-Prolog (Version 2.1, Patch 8) implementiert. X Window System (Version 11, Release 5) bildet die Grundlage für die grafische Systemoberfläche. Das System hat compiliert eine Größe von 2,4 MB. Der Quellcode umfaßt 890 KB.
Das Programmpaket umfaßt folgende Directories und Dateien, wobei davon ausgegangen wird, daß diese in einer Directory (beispielsweise namens `IGLOO/`) wie folgt enthalten sind:

```
GALLIER/                        * <DIR> *
INTERAKTIV/                     * <DIR> *
LLP/                            * <DIR> *
PRETTY/                         * <DIR> *
README                          * Kurzbeschreibung zur Installation*
TREEWIN/                        * <DIR> *
WALLEN/                         * <DIR> *
cpl.pl                          * Ladefile für compilierte Version *
demo_trees/                     * <DIR> *
hf_default                      * Konfigurationsfile *
igloo_day                       * erstes IGLOO-Logo *
igloo_init_expl                 * Beispiel einer Umgebungsdatei *
igloo_night                     * zweites IGLOO-Logo *
load.pl                         * Ladefile für interpretierte Version *

GALLIER/:                       * Beweiser für PL1 *
demo_pl1                        * File mit Sequenten der Logik PL1 *
gallier_help_preds.pl           * Hilfsprädikate *
gallier_prover.pl               * Automatischer Beweiser *
gallier_rules.pl                * Regelmenge des Kalküles für PL1 *
gallier_rules_help.pl           * Hilfsprädikate zu den Regeln *
gen_symbol.pl                   * Zählerverwaltung *
herbrand.pl                     * Erzeugung des Herbrand-Universums *
logsigns.pl                     * Operatordefinitionen für die Junktoren *
pl1_tac.pl                      * Prädikatenlogische Taktiken *
rules_to_code.pl                * Regelcompiler *

INTERAKTIV/:                    * Aktionen auf der Oberfläche "IGLOO" *
browser_pretty_ncd_fonts.pl     * Fontdatei für Regelbrowser *
browser_pretty_slc_fonts.pl     * Fontdatei für Regelbrowser *
default_igloo_init              * Konfigurationsdatei *
df_adapt.pl                     * Regelbrowser *
```

```
int_declarations.pl              * Deklaration von Parametern *
intertree.pl                     * Verwalten der Oberfläche "IGLOO" *
ncd_fonts_hf.pl                  * Fontdatei für "IGLOO" *
seq_pretty.pl                    * ASCII-Prettyprinter für Sequenten *
shell_variables.pl               * Zugriff auf Betriebssystem *
slc_fonts_hf.pl                  * Fontdatei für "IGLOO" *
tac_pretty.pl                    * Prettyprinter für Taktiken *
terminal.pl                      * Laden der richtigen Fontdateien *

LLP/:                            * Prädikate zum Kalkül LLP *
ax_schemata_sort.pl              * Axiomenschemata *
basic_rules.pl                   * Regelmenge zu LLP *
build_hypo.pl                    * Planerkennerprädikate *
buildaxrule.pl                   * Instantiierung von Axiomenschemata *
declarations_for_intertree.pl    * Deklarationsfile *
demo_llp                         * File mit Sequenten der Logik LLP *
dirk.pl                          * Formelmanipulationen *
dirk_tac.pl                      * File mit Taktiken zu LLP *
dirk_utility.pl                  * Formeltransformationen *
display_new.pl                   * Alternative Grafik (hier nicht relevant) *
gen_rules.pl                     * Regelmenge zu LLP *
gen_rules_new.pl                 * Regelmenge zu LLP *
gen_tac.pl                       * File mit Taktiken zu LLP *
gen_tactic.pl                    * File mit Taktiken zu LLP *
gen_utility.pl                   * Axiom-, Formelmanipulationen *
gen_utility_patch.pl             * Axiom-, Formelmanipulationen *
hf_for_ax_schemata_sort.pl       * Handhabung von Sorteninformation *
hf_for_phi.pl                    * Anpassungen an die Oberfläche "IGLOO" *
indexing.pl                      * Indexierung von Termen in LLP *
llp_rules_rest.pl                * Regelmenge zu LLP *
logical_axioms.pl                * Regelmenge zu LLP *
modal_rules.pl                   * Regelmenge zu LLP *
output.pl                        * Plangenerierungstrace *
pg_pretty.pl                     * Plangenerierungstrace *
phisort.pl                       * Sortenhierarchie in LLP *
proof_tree.pl                    * Knotendatenstruktur, Zählerverwaltung *
subform.pl                       * Formelmanipulation *
tac_dynamic.pl                   * File mit Taktiken zu LLP *
tac_plus.pl                      * PG-Trace in "IGLOO" *
tac_util.pl                      * File mit Taktiken zu LLP *
tactic.pl                        * File mit Taktiken zu LLP *
term_manipulation.pl             * Termindexing *
unifsort.pl                      * Sortierte Unifikation zu LLP *

PRETTY/:                         * Prettyprinter für Sequenten *
prettynew.pl                     * Prettyprinter *
prettywin.pl                     * Fenstergröße des Prettyprinters *
```

```
TREEWIN/:                       * eff. Datenstruktur und Grafikkomponente *
bbalpha_pop_up.pl               * Identifizierung selektierter Knoten *
build_bbalpha.pl                * Aufbau der Datenstruktur für Beweisbäume *
delete_bbalpha.pl               * Löschen aus der Datenstruktur *
insert_bbalpha.pl               * Einfügen in die Datenstruktur *
tidytree.pl                     * Strukturieralgorithmus für Bäume *
tidytree_adapt.pl               * Schnittstelle zur Ausgaberoutine *
treewin.pl                      * Zentrieralgorithmus, Ausgaberoutine *
treewin_adapt.pl                * Anpassung an "IGLOO" *

WALLEN/:                        * Kalkül S4 *
demo_s4                         * File mit Sequenten der Logik S4 *
wallen_prop_rules.pl            * Regelmenge zu S4 *
wallen_rules.pl                 * Regelmenge zu S4 *
wallen_tac.pl                   * File mit Taktiken zu S4 *

demo_trees/:                    * Datenbasis für Ableitungsbäume *
all_databases/                  * <DIR> *
dieb_billy943/                  * <DIR> *
dieb_jacky118/                  * <DIR> *
dieb_jacky427/                  * <DIR> *
test1/                          * <DIR> *

demo_trees/all_databases/:      * Register aller gespeicherter Bäume *
0                               * Prologinterne Realisierung einer Datenba- *
if                              * sis. Die Inhalte von 0, if, spec braucht *
```

```
spec                            * weder der Anwender noch der Programmierer *
                                * zu kennen.  Sie werden durch ein Modul *
                                * von SICStus-Prolog erzeugt und verwaltet *
                                * (vgl. [Swe93b], [Swe93a]).  *

demo_trees/dieb_billy943/:      * Abgespeicherter Baum dieb_billy943 *
0                               * s.o. *
if                              * s.o. *
spec                            * s.o. *

demo_trees/dieb_jacky118/:      * Abgespeicherter Baum dieb_jacky118 *
0                               * s.o. *
if                              * s.o. *
spec                            * s.o. *

demo_trees/dieb_jacky427/:      * Abgespeicherter Baum dieb_jacky427 *
0                               * s.o. *
if                              * s.o. *
spec                            * s.o. *
```

```
demo_trees/test1/:          * Abgespeicherter Baum test1 *
0                           * s.o. *
if                          * s.o. *
spec                        * s.o. *
```

Bevor das System gestartet werden kann, müssen folgende Anpassungen vorgenommen werden:

1) Die Datei "igloo_init_expl" muß unter dem Namen ".igloo_init" in die Home-Directory des Benutzers kopiert werden. Diese Datei enthält die initiale Umgebung für das System, die in Form von Prolog-Fakten aufgelistet ist. Konkret sieht die mitgelieferte "igloo_init_expl"-Datei wie folgt aus:

```
sequent_files(['./LLP/demo_llp',
               './GALLIER/demo_pl1',
               './WALLEN/demo_s4']).
db_dirs(['./demo_trees']).
config_files(['./hf_default']).
calculi([ ['pl1',
           [test_pl1_1/1,
            delay_splittings_1/1,
            delay_splittings_2/1,
            one_son_no_quant/3,
            two_sons_no_quant/3,
            quants_introduce_vars/3,
            quants_instantiations/3,
            then_strict_test1/1,
            test_pl1_2/1,
            left_or/1,
            rounds/1],
           ['./GALLIER/gallier_rules',
            './GALLIER/pl1_tac']],
          ['S4',
           [one_son_wallen/3,
            two_sons_wallen/3,
            not_destroy_modal/3,
            wallen_tac/1],
           ['./WALLEN/wallen_rules',
            './WALLEN/wallen_prop_rules',
            './WALLEN/wallen_tac']]
          ['llp:LLP',
           [gentac/1,
            find_plan_with_known_precondition/1,
            close_one_effect_branch/1,
            introduce_missing_precondition/1,
            introduce_before_last_effect/1,
            kill_not_tac/1,
```

```
            introduce_conditional/1,
            disj_precond_elimination/1,
            reconstruct/1,
            establish_frame_cond/1,
            missing_effect_conditional/1,
            or_split_new_tac/1,
            triv_ass_close_tac/1,
            close_triv_ass_sequent/1,
            close_if_triv_ass/1],
           ['./LLP/gen_rules',
            './LLP/gen_rules_new',
            './LLP/llp_rules_rest',
            './LLP/modal_rules',
            './LLP/basic_rules',
            './LLP/logical_axioms',
            './LLP/tac_dynamic',
            './LLP/gen_tac',
            './LLP/dirk_tac',
            './LLP/gen_tactic',
            './LLP/ax_schemata_sort',
            './LLP/hf_for_ax_schemata_sort']]
         ]).
sorting_rules('yes').
```

Das `sequent_files/1`-Fakt enthält eine Liste, in der die Namen der Dateien stehen, in denen Sequenten abgespeichert sind. Die in der vorgegebenen Liste aufgeführten Dateien sind alle in dem Programmpaket vorhanden. Evtl. müssen die Pfadnamen vom Benutzer an seine jeweilige Verzeichnisstruktur angepaßt werden. Sollte ein Benutzer aus irgendwelchen Gründen diese Dateien nicht (mehr) besitzen, so muß er die Liste auf die leere Liste `[]` abändern. Während des Programmes können dann menügesteuert Sequenten, die auf der Oberfläche eingegeben werden, unter beliebigen Dateinamen abgespeichert werden. Vor weiteren Systemstarts kann eine solche Datei auch von Hand in einem Editor erzeugt werden, wobei das Format der einzelnen Sequenten aus den durch den ersten Systemlauf erzeugten Sequent-Dateien ersichtlich ist (natürlich nur, falls der Benutzer auch Sequenten während des Programmlaufes abgespeichert hat). Eine solche in einem Editor außerhalb des Programmes erzeugte Datei muß von dem Benutzer selbst in die Liste der Sequent-Dateien aufgenommen werden.
Das `db_dirs/1`-Fakt enthält eine Liste, in der die Namen der Directories stehen, welche als Datenbasen für Beweisbäume dienen. Wie oben gilt auch hier, daß die aufgeführten Directories im Programmpaket enthalten sind und die Pfadnamen im Präfix je nach Benutzer evtl. abgeändert werden müssen. Neue Directories können in diese Liste aufgenommen werden.
Das `config_files/1`-Fakt enthält eine Liste der Dateien, die Informationen über Systemkonfigurationen enthalten. Die vorgegebene Datei "hf_default" sollte auf jeden Fall vorhanden sein. Hier braucht der Benutzer außer der Pfadanpassung nichts weiter zu edieren, da das Abspeichern und Aktivieren von Konfigurationen aus dem

Programm heraus erfolgt.
Das `calculi/1`-Fakt enthält eine Liste der verschiedenen Kalküle, die von dem System aktivierbar sind. Jeder Kalkül ist durch eine dreielementige Liste [*Name, Taktikliste, Files*] repräsentiert. Die angeführten Kalküle sind im Programmpaket schon vorhanden. Für LLP-Kalküle muß der *Name* die Form '*Name*:LLP' haben (vgl. Abschnitt 2.4). Auch hier braucht der Benutzer wiederum nur die Anpassung der Pfadnamen in der Liste *Files* vorzunehmen.
Das `sorting_rules/1`-Fakt kann als Argument entweder `'no'` oder `'yes'` enthalten. Bei Angabe von `'no'` werden die Regeln des jeweils im Systemlauf geladenen Kalküls in der Reihenfolge angezeigt, in der sie auch in der entsprechenden Regeldatei stehen. Dadurch kann erreicht werden, daß inhaltlich gleichartige Regeln nicht nur in der entsprechenden Datei sondern auch auf der Systemoberfläche nahe beieinander stehen. Bei Angabe von `'yes'` werden sie auf der Systemoberfläche in alphabetischer Reihenfolge angezeigt.

2) Man kann das System entweder im interpretierten oder im compilierten Modus laufen lassen, je nachdem, ob man die Hauptladedatei *load.pl* oder *cpl.pl* nach dem SICStus-Start lädt. Allerdings müssen die in beiden Dateien enthaltenen Pfadnamen evtl. wieder der Verzeichnisstruktur des jeweiligen Benutzers angepaßt werden.

3) Vor dem eigentlichen Start von Prolog muß die Environment-Variable LOCALSTKSIZE mindestens auf den folgenden Wert gesetzt werden:

```
setenv LOCALSTKSIZE 300000
```

Der eigentliche Systemstart geht nun wie folgt vonstatten:

1. Aufruf von SICStus-Prolog durch Eingabe von

   ```
   sicstus
   ```

2. Laden des Systems in compilierter Form durch

   ```
   | ?- compile(cpl).
   ```

   oder im interpretierten Modus durch

   ```
   | ?- [load].
   ```

3. Start des Systems durch Eingabe von

   ```
   | ?- igloo.
   ```

4. Nach Beenden des **IGLOO**-Systems, **vor** Verlassen von Prolog durch

   ```
   | ?- halt.
   ```

   muß zuerst ein

```
| ?- end.
```

eingegeben werden, um den Grafikprozeß zu beenden.

Da der Grafikprozeß ("ispgm") oft mit der Dauer der Systembenutzung langsamer wird, kann man, um schneller arbeiten zu können, das System zwischendurch kurz verlassen, auf Prolog-Ebene ein

```
| ?- end.
```

eingeben und erneut

```
| ?- igloo.
```

aufrufen.

2 Beweisen mit IGLOO

In diesem Kapitel wird gezeigt, wie mit **IGLOO** Beweise in Sequenzenkalkülen geführt werden können. Zuerst wird das automatische, dann das interaktive Beweisen erläutert. Beim interaktiven Beweisen wird auch die Handhabung von Taktiken und die spezielle grafische Unterstützung beschrieben. Nachdem das System gestartet ist (siehe Abschnitt 1.3), erscheint auf dem Bildschirm die in Abbildung 2 dargestellte Oberfläche "IGLOO". Alle Aktionen, die direkt der Beweissteuerung dienen, werden von hier eingeleitet. Zur Präsentation der Beweisbäume gibt es die spezielle Grafikoberfläche "Proof Tree", die in Kapitel 3 dargestellt wird. Eine Kurzanleitung zum Beweisen mit **IGLOO** ist im Anhang unter B in Form von Flußdiagrammen aufgeführt. Zunächst wird die Funktionalität jedes einzelnen Buttons kurz erläutert, so daß hier eine für die Benutzung nützliche, zusammenhängende Referenzstelle im Handbuch vorliegt.

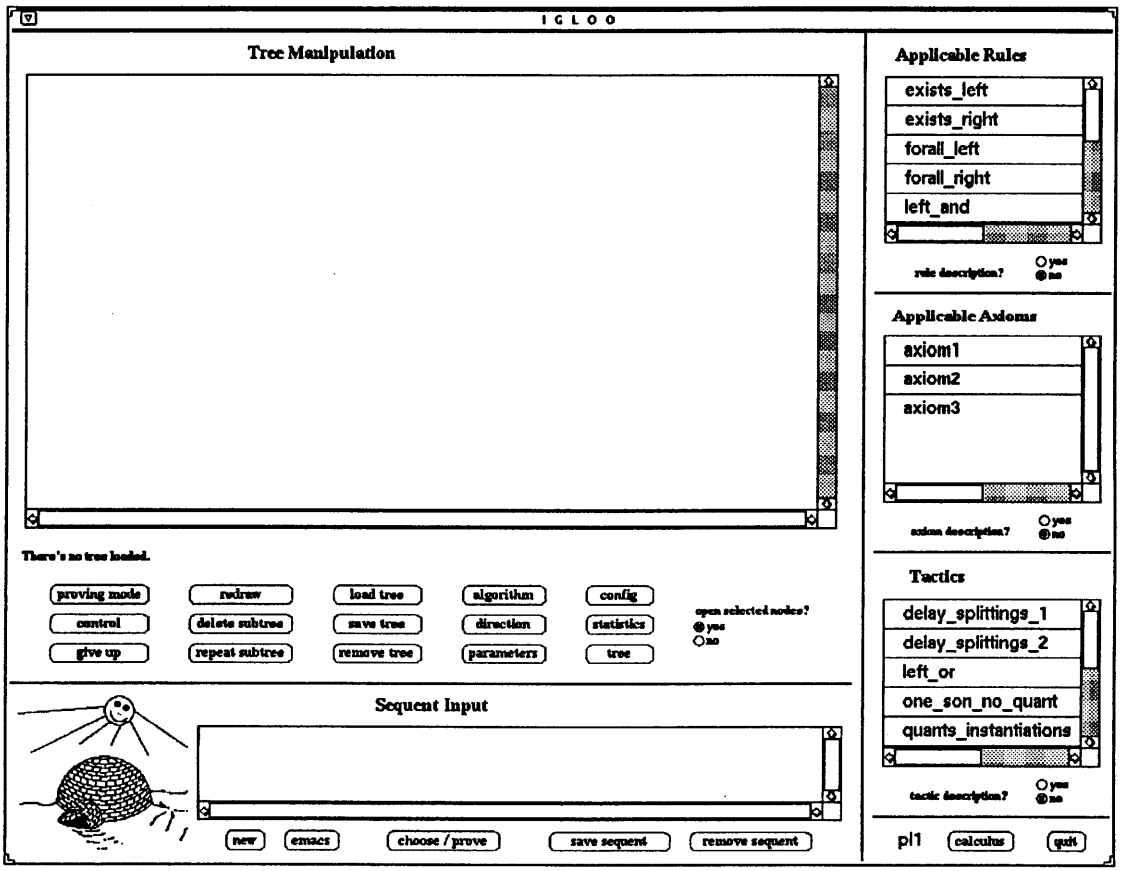


Abbildung 2: Die Hauptoberfläche "IGLOO"

≪*proving mode*≫ Man kann zwischen drei verschiedenen Beweisführungsmodi wählen. Dies sind

- *automatic proving & batch display* In diesem werden die Beweise vom System vollautomatisch geführt, indem es die Regeln in einer bestimmten Systematik

- *automatic proving & incremental display* Dabei wird der Beweis wie im ersten Modus automatisch geführt, jedoch wird nach jedem Ableitungsschritt im Beweis auch der entsprechende Knoten im Beweisbaum grafisch erzeugt, so daß man den Beweisablauf im View *Tree Manipulation* mitverfolgen kann. Der dritte Modus ist

- *interactive proving* Hier führt der Benutzer selbst den Beweis, indem er im View Knoten selektiert und auf diese Regeln, Axiome oder Taktiken anwendet.

≪*control*≫ Das Verhalten beim automatischen Beweisen im Kalkül für PL1 kann in dem Sinne verändert werden, daß man die Termtiefe bei automatischen Instantiierungen begrenzt bzw. daß man die Instantiierungen durch den Benutzer vornehmen läßt. Man hat also eine Zwischenstufe zwischen automatischem und interaktivem Beweisen (vgl. S. 18).

≪*give up*≫ Im vollautomatischen Modus kann ein Beweis jederzeit abgebrochen werden (wichtig bei Nichtterminierung!).

≪*redraw*≫ Mit diesem Button kann man erreichen, daß ein im View vorhandener Ableitungsbaum neu gezeichnet wird.

≪*delete subtree*≫ Man hat die Möglichkeit — besonders nützlich beim interaktiven Beweisen — Teile des Beweisbaumes wieder zu löschen, etwa um andere Beweisschritte durchzuspielen (vgl. S. 23 f.).

≪*repeat subtree*≫ Es kommt relativ häufig vor, daß in verschiedenen Teilen eines Beweisbaumes der Struktur nach gleiche Beweise geführt werden. Beim interaktiven Beweisen wird hierfür eine Unterstützung angeboten, indem man schon einmal im aktuellen Baum enthaltene Beweisstrukturen an anderen Knoten im Beweisbaum wiederholen lassen kann.

≪*load tree*≫ Abgespeicherte Ableitungsbäume können zur weiteren Bearbeitung erneut in den View *Tree Manipulation* geladen werden.

≪*save tree*≫ Ein gerade im View *Tree Manipulation* befindlicher Beweisbaum kann gesichert werden. Dadurch besteht die Möglichkeit, beim interaktiven Beweisen Teilbäume abzuspeichern, die man bei einer anderen Sitzung weiterbearbeiten kann.

≪*remove tree*≫ Ein gespeicherter Baum, der nicht mehr benötigt wird, kann aus der externen Datenbasis gelöscht werden.

≪*algorithm*≫ Es kann zwischen dem Zentrier- und dem Strukturieralgorithmus bei der grafischen Darstellung von Ableitungsbäumen gewählt werden. Die zentrierte Darstellung eines Ableitungsbaumes ist sinnvoll, wenn man sich über die Breite des Baumes klar werden will, die strukturierte Ausgabe macht jedoch die eigentliche Beweisstruktur sichtbar, so daß man beispielsweise ähnliche Ableitungsschritte an verschiedenen Stellen des Beweises optisch leicht erkennen kann.

≪*direction*≫ Ein Ableitungsbaum kann entweder mit der Wurzel oben, den Blättern unten oder mit der Wurzel links und den Blättern rechts dargestellt werden.

≪*parameters*≫ Mit diesem Button lassen sich Einstellungen festlegen, die das optische Erscheinungsbild eines Beweisbaumes bestimmen. So kann man die Knotengröße, den Mindestabstand zwischen Knoten, den im Knoten benutzten Font, die Knotenfarbe (in Graustufen) und das Layout der in Kapitel 3 beschriebenen Grafikoberfläche "Proof Tree" bestimmen. Genauere Erklärungen sind in den Abschnitten 3.1 und 3.2 gegeben.

≪*config*≫ Es wird die aktuelle Konfiguration des Systems angezeigt. Ferner hat man die Möglichkeit, diese abzuspeichern bzw. eine neue zu laden. Vor dem Laden einer neuen Konfiguration wird man darüber informiert, inwieweit sich die aktuelle und die zu ladende voneinander unterscheiden. Eine Konfiguration legt sowohl das Verhalten des Beweisers fest als auch die Art der grafischen Ausgabe der Beweisbäume.

≪*statistics*≫ Zu einem gegenwärtig im View *Tree Manipulation* angezeigten Beweisbaum kann man sich statistische Informationen geben lassen, wie die Anzahl der Knoten im Baum, die Tiefe des Baumes und die Anzahl der Anwendungen der einzelnen Regeln.

≪*tree*≫ Mit diesem Button gelangt man zur Grafikoberfläche "Proof Tree", welche in Kapitel 3 genau beschrieben wird.

Mit dem Buttonstate *open selected nodes?* kann man einstellen, ob bei einer Knotenselektion der im Knoten enthaltene Sequent in einem Fenster angezeigt werden soll (Einstellung *yes*) oder ob lediglich der Knoten ohne Anzeige des zugehörigen Sequenten selektiert werden soll (Einstellung *no*).

≪*new*≫ Wenn ein Beweis durch den Benutzer gestartet wird, wird er immer für den Sequenten durchgeführt, der bei Auslösung des Startes gerade in dem Textview *Sequent Input* angezeigt wird. Durch ≪*new*≫ wird der Inhalt des Textviews gelöscht, so daß ein neuer Sequent eingegeben oder von einer Datei geladen werden kann.

≪*emacs*≫ Da der Textview *Sequent Input* bei der Sequenteneingabe keine korrespondierenden Klammern anzeigt, hat man mit diesem Button die Möglichkeit, den Editor *Emacs* aufzurufen, der vor allem bei größeren Eingaben komfortabler ist und auch den Überblick über einen Sequenten erleichtert.

≪*choose/prove*≫ Es wird ein Beweis für den Sequenten gestartet, der sich gegenwärtig im Textview *Sequent Input* befindet. Ist dort kein Sequent angezeigt, so werden die Dateien angeboten, von denen Sequenten zum Beweisen geladen werden können (vgl. `sequent_files/1`-Fakt S. 7).

≪*save sequent*≫ Ein Sequent, der sich im Textview *Sequent Input* befindet, kann

in einer beliebigen Datei gesichert werden, wo er für spätere Zugriffe dann zur Verfügung steht.

≪*remove sequent*≫ Nicht mehr benötigte Sequenten können von einer Datei gelöscht werden.

≪*calculus*≫ Man kann zwischen verschiedenen Kalkülen wechseln. Der Name des aktuellen Kalküles steht links neben dem Button ≪*calculus*≫. Die Namen der zugehörigen Regeln, Axiome und Taktiken sind in den Views *Applicable Rules*, *Applicable Axioms* und *Tactics* zu sehen. Die Felder dieser Views sind maussensitiv, so daß die Regeln, Axiome und Taktiken beim interaktiven Beweisen durch den Benutzer aktiviert werden können. Eine genaue Beschreibung erfolgt im Abschnitt

Axiom-, und Taktiknamen des gewählten Kalküles ersetzt. Welche Kalküle geladen werden können, wird durch das Fakt `calculi/1` in der Datei "~/.igloo_init" (vgl. S. 7 f.) festgelegt. Die den drei Views zugeordneten Buttonstates haben folgende Bedeutung: Bei Einstellung *no* können beim interaktiven Beweisen Regeln, Axiome

zu 1.: Mit ≪*proving mode*≫ (vgl. S. 11) wählt man den Beweismodus *automatic proving & batch display.*

zu 2.: Bei der Festlegung des Startsequenten[9] bestehen vier alternative Möglichkeiten:

a) Der aktuell im Textview *Sequent Input* angezeigte Sequent soll der Startsequent sein.

b) Aus dem aktuell in *Sequent Input* angezeigten Sequenten soll der Startsequent durch Modifikation erzeugt werden.

c) Der Startsequent soll neu ediert werden.

d) Der Startsequent soll von einer Sequentendatei geladen werden.

Ist der Textview *Sequent Input* nicht leer, so wird mit ≪*choose/prove*≫ (vgl. S. 13) immer für den Sequenten ein Beweis gestartet, der sich gegenwärtig in diesem View befindet (wichtig für die Fälle **a)**, **b)** und **c)**). Voraussetzung für den Fall **d)** ist dagegen ein leerer Textview.

zu a): Mit ≪*choose/prove*≫ wird der Beweis für den in *Sequent Input* stehenden Sequenten gestartet.

zu b): Kleine Modifikationen des Sequenten können direkt im Textview durchgeführt werden. Für größere Änderungen besteht die Möglichkeit, mit ≪*emacs*≫ den Editor *Emacs* aufzurufen, dessen komfortable Ediermöglichkeiten (Anzeigen korrespondierender Klammern, ...) die Modifikation erleichtern. Durch ≪*emacs*≫ wird der aktuelle Inhalt von *Sequent Input* in den Editor kopiert. Nach Verlassen des Editors mit `^X^S`, `^X^C` wird dessen Inhalt wiederum in den Textview übertragen, so daß anschließend mit ≪*choose/prove*≫ der Beweis für den modifizierten Sequenten gestartet werden kann.

zu c): Zum Eingeben eines neuen Sequenten wird zuerst der Inhalt des Textviews *Sequent Input* mittels ≪*new*≫ gelöscht. Die Eingabe und der Beweisstart erfolgen dann analog den Ausführungen zu **b)**. Die bei der Eingabe zu benutzende Syntax wird durch die folgenden Produktionen einer kontextfreien Grammatik festgelegt:

Sequent ::= (*Antezedens* --> *Sukzedens*)
Antezedens ::= *Formel* | *Formel*, *Antezedens*
Sukzedens ::= *Formel* | *Formel*, *Sukzedens*
Formel ::= `Formelsyntax der betreffenden Logik`

Die Formelsyntax der jeweiligen Logik ist in Abschnitt 2.3.3 auf Seite 26 f. für die Prädikatenlogik erster Stufe gegeben. Bei ihr wie auch bei der Syntax von

[9] Vgl. auch Anhang B, Flußdiagramm S. 54.

anderen Logiken darf die dort für Regeln zusätzlich erlaubte Produktion *Formel* ::= `Prologvariable` nicht hinzugenommen werden.

zu d): Um einen Sequenten von einer Datei zu laden, wird zuerst der Inhalt von *Sequent Input* mit ≪*new*≫ gelöscht. Durch ≪*choose/prove*≫ wählt man zuerst die Datei und in der daraufhin angezeigten Auswahl den zu beweisenden Sequenten, den man sich vor dem Beweisstart, welcher durch ≪*prove*≫ im Fenster "Sequent Choice" erfolgt, in einer benutzerfreundlichen Darstellung betrachten kann (vgl. Abbildung 3).

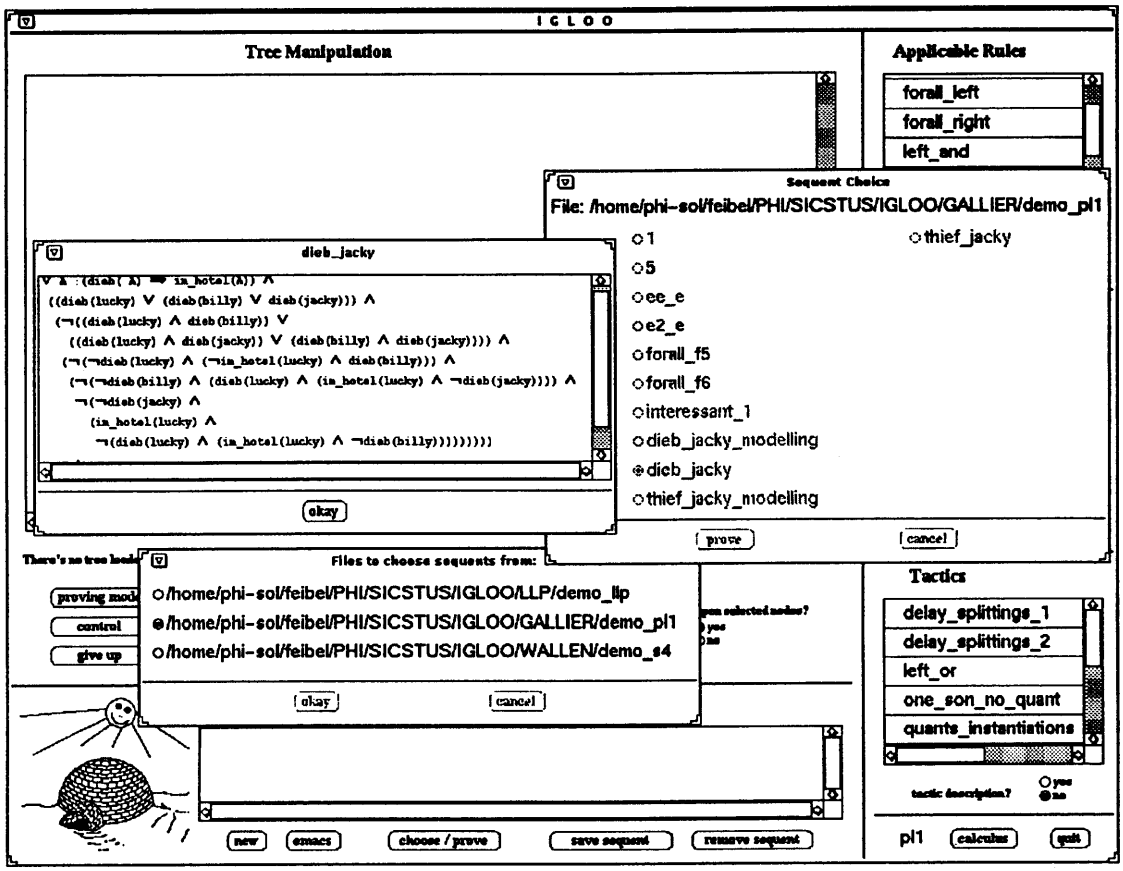


Abbildung 3: Beweisstart für Sequenten aus einer Datei

zu 3.: Nachdem der Beweis beendet worden ist (entweder vom System selbst oder durch den Benutzer mit ≪*give up*≫), meldet das System in dem Fenster "Proof Result", ob der Startsequent bewiesen werden konnte. In diesem Fenster kann man entscheiden, ob die zum Beweis gehörende Grafik erzeugt werden soll. Mit ≪*display prooftree*≫ beginnt das System, die Grafik zu berechnen und zeigt sie nach von der Baumgröße abhängiger Zeit in dem View *Tree Manipulation* an. Ein Blatt eines Baumes wird durch die Färbung seines unteren Teiles in eine von drei Kategorien eingeordnet:

schwarz Der zugehörige Sequent ist ein Axiom (*geschlossenes Blatt*).

dunkelgrau Der Sequent ist kein Axiom, er ist aber durch Regeln weiter expandierbar (*offenes, expandierbares Blatt*).

hellgrau Der Sequent ist kein Axiom, und es kann keine Regel mehr auf ihn angewandt werden (*offenes, nicht expandierbares Blatt*).

Da normalerweise nicht der gesamte Baum im View *Tree Manipulation* zu sehen ist, kann man mit den Scrollbars im View navigieren. Im oberen Teil der maussensitiven Knoten eines Ableitungsbaumes steht ihr eindeutiger Name und im unteren Teil der Name der Regel, mit der aus dem Sequenten des Knotens die den Nachfolgeknoten zugehörigen Sequenten abgeleitet worden sind. Durch Selektion eines Knotens mit der Maus wird sein zugehöriger Sequent in einem Fenster, das den Namen des Knotens trägt, angezeigt (vgl. Abbildung 4). In der Grafik sind selektierte Knoten durch eine dickere Umrandung hervorgehoben. Mit ≪*close ALL node windows*≫ können alle geöffneten Knoten geschlossen werden. Durch ≪*close THIS window*≫ oder durch erneutes Anklicken in der Grafik kann man einen einzelnen Knoten schließen. In der Abbildung ist auch die zum Beweis gehörende Statistik (Aufruf durch ≪*statistics*≫) zu sehen.

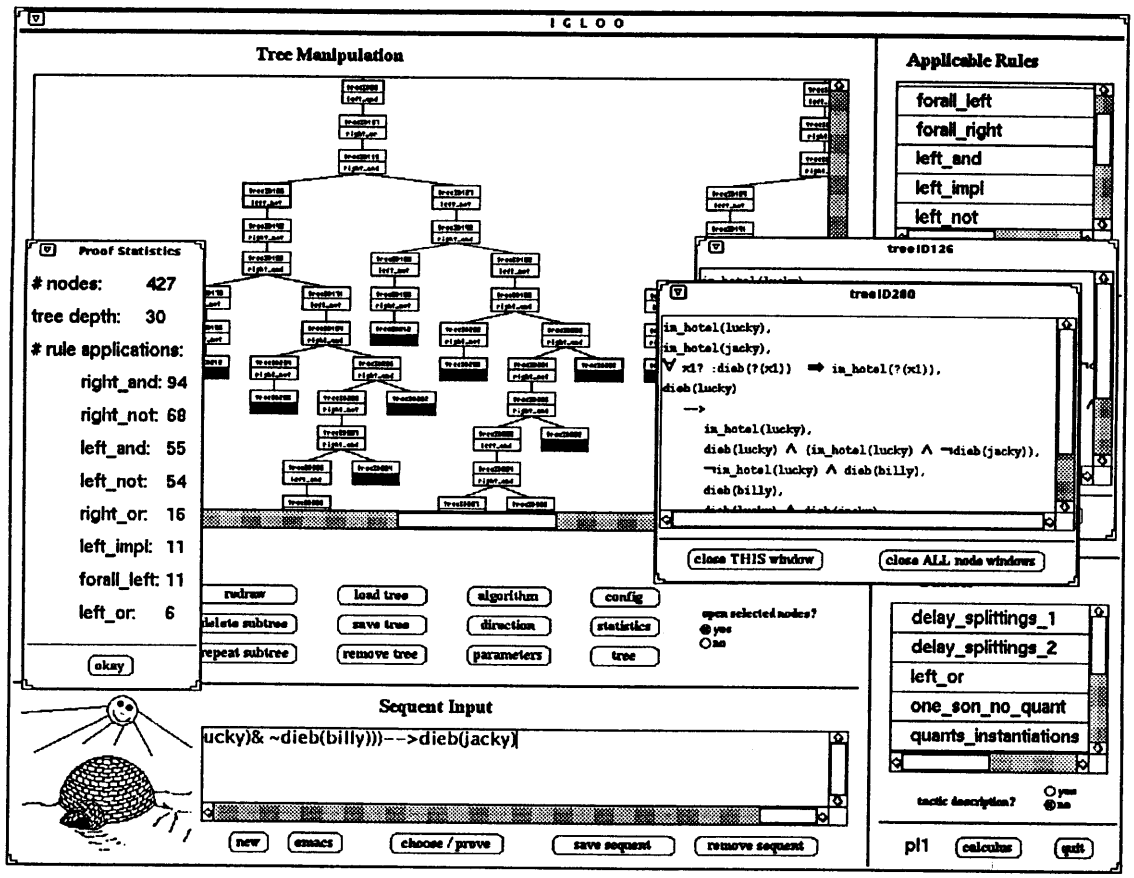


Abbildung 4: Oberfläche nach automatischem Beweis des Sequenten *dieb_jacky* mit zwei geöffneten Knoten und der zugehörigen Statistik

2.1.2 Beweisen im Modus *automatic proving & incremental display*

Folgende Schritte sind beim Beweisen im Modus *automatic proving & incremental display* durchzuführen[10]:

[10]Vgl. auch die Kurzbeschreibung in Anhang B, S. 52.

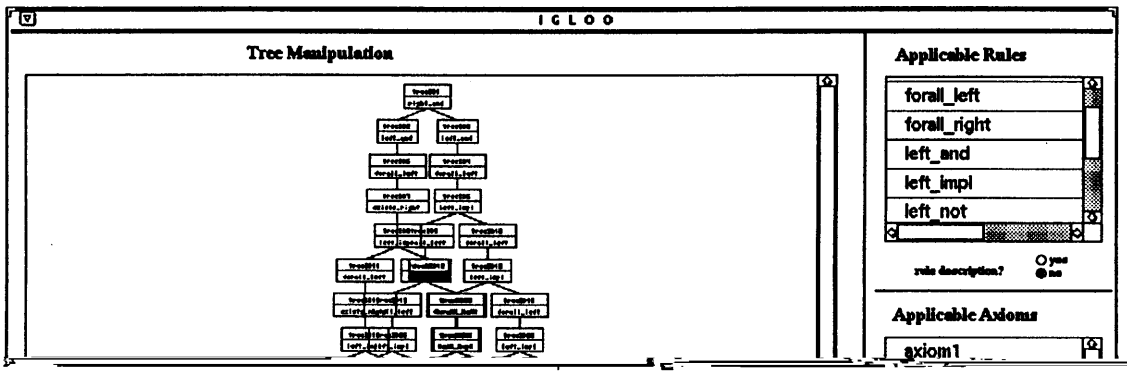
1. Wahl des Beweismodus,

2. Festlegung des Startsequenten und Beweisstart,

3. Neuaufbau der Baumgrafik.

Das Vorgehen bei den Schritten 1 und 2 ist identisch zu dem beim Modus *automatic proving & batch display* (vgl. S. 14 ff.), nur daß hier bei der Moduswahl *automatic proving & incremental display* gewählt wird.

zu 3.: Im Modus *automatic proving & incremental display* wird mit jedem Beweisschritt auch die Grafik entsprechend expandiert, so daß man im View *Tree Manipulation* den Beweis mitverfolgen kann. Im allgemeinen werden sich dabei Kanten und Knoten des Baumes in der Grafik überlappen (vgl. Abbildung 5). Nachdem der Beweis durch System oder Benutzer beendet worden ist, kann man mit ≪*redraw*≫ den Baum in einer optimalen grafischen Form neu aufbauen lassen.



beim automatischen Beweisen sehr groß wird. Diese Regeln instantiieren quantifizierte Formeln, indem sie für die quantifizierten Variablen Terme des Herbrand-Universums in einer bestimmten Systematik einsetzen. Von **IGLOO** werden zwei Möglichkeiten angeboten, diese Einsetzungsproblematik in den Griff zu bekommen. Zum einen wird durch eine Begrenzung der Termtiefe nur ein Teil des Herbrand-Universums zur Instantiierung zugelassen, zum anderen kann dem System die Aufgabe der Instantiierung aber auch ganz abgenommen und dem Anwender übertragen werden. Zwei Beispiele sollen diese Möglichkeiten verdeutlichen.

Begrenzung der Termtiefe: Mit ≪*control*≫ kann man in dem Fenster "proof controls" die maximale Termtiefe festlegen. Bei einer Termtiefe von fünf kann der Sequent *forall_f5*

```
forall A: p(A) ⟶ p(f(f(f(f(f(a))))))
```

vom System bewiesen werden, indem der Reihe nach die Terme

```
a, f(a), ..., f(f(f(f(f(a)))))
```

durch die Regel *forall_left* instantiiert werden. Der Sequent *forall_f6*

```
forall A: p(A) ⟶ p(f(f(f(f(f(f(a)))))))
```

kann bei dieser Termtiefe jedoch nicht bewiesen werden, da die "passende" Instantiierung durch den Term

```
f(f(f(f(f(a)))))
```

der Tiefe 6 nicht vorgenommen wird.

Benutzerinstantiierungen: Wird in dem Buttonstate des durch ≪*control*≫ erreichten Fensters "proof controls" *user* selektiert, so wird bei jeder Anwendung der beiden Instantiierungsregeln der Benutzer durch ein Fenster, wie es in Abbildung 6 dargestellt ist, zur Angabe einer Instantiierung aufgefordert. In dem View dieses Fensters steht der Sequent, auf den die jeweilige Instantiierungsregel angewandt werden soll. Hinter *Quantified Formula* steht die quantifizierte Formel des Sequenten, für die eine Instantiierung gesucht wird. Hinter *Substitution Variable* ist die Variable angegeben, für die ein Term einzusetzen ist. Die Schreibweise der Variable als `x1?` (bzw. `?(x1)` im View) entspricht einer internen Repräsentation durch das System. Bei der Sequenteneingabe, die oben beschrieben worden ist, haben Variablen die gleiche Syntax wie Prologvariablen. In dem Maskenfeld hinter *Substitution Term* wird vom Benutzer der gewünschte Term angegeben. Durch ≪*user substitution*≫ wird die entsprechende Regel mit der angegebenen Instantiierung angewandt. Die Wirkung der Benutzerinstantiierungen kann anhand des Sequenten *interessant_1* beobachtet werden. Bei einem automatischen Beweis mit Instantiierungen, die durch das System selbst vorgenommen werden, besteht der Beweisbaum aus 38 Knoten. Bei erneutem Beweis mit dreimaliger Benutzerinstantiierung (jedesmal mit `g(a)`) kann ein Beweisbaum mit nur 12 Knoten gefunden werden.

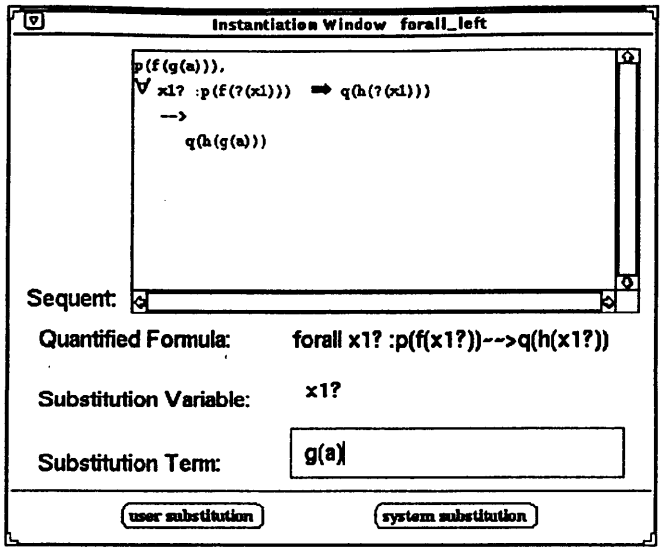


Abbildung 6: Aufforderung zur Benutzerinstantiierung im Beweis des Sequenten *interessant_1*

2.2 Interaktives Beweisen

Folgende Schritte sind beim Beweisen im Modus *interactive proving* durchzuführen[11]:

1. Wahl des Beweismodus,
2. Festlegung des Startsequenten und Beweisstart bzw. Laden eines (teilweise expandierten) Beweisbaumes,
3. interaktive Beweisführung.

zu 1.: Mit ≪*proving mode*≫ wird der Modus *interactive proving* gewählt.

zu 2.: Die Festlegung des Startsequenten geschieht genau wie im Modus *automatic proving & batch display* (vgl. S. 15). Der Beweisstart bewirkt hier, daß die Wurzel des Baumes in dem View *Tree Manipulation* erzeugt wird, wo sie zur weiteren Expansion zur Verfügung steht. Falls sich im View *Tree Manipulation* schon ein Baum befindet, können aber auch an diesem selbst interaktive Beweisschritte durchgeführt werden. Eine weitere Möglichkeit besteht darin, einen Baum aus der Datenbasis mit ≪*load tree*≫ in den Grafikview zu laden und ihn dann interaktiv weiter zu expandieren.

zu 3.: Die interaktive Beweisführung umfaßt folgende Aktionen:

a) Anwendung von Regeln, Axiomen und Taktiken auf selektierte Knoten,

b) Beweiswiederholung,

c) Rücknahme von Beweisschritten.

[11] Vgl. auch Flußdiagramm in Anhang B, S. 53.

zu a): Durch Anklicken mit der Maus können mehrere Blätter für eine anschließende Expansion selektiert werden. Für das jeweils zuletzt selektierte Blatt wird durch Markierungen über und in den Views *Applicable Rules* und *Applicable Axioms* angezeigt, ob, wieviele und welche Regeln bzw. Axiome auf den zugehörigen Sequenten anwendbar sind. In Abbildung 7 wurde die Wurzel selektiert. Aufgrund der Selektion ist über dem Regelview ein kleines, grauhinterlegtes Dreieck erschienen. Die 2 in ihm bedeutet, daß zwei Regeln des aktuellen Kalküles auf das selektierte Blatt anwendbar sind. Im Regelview sind die beiden Regeln durch schwarze Dreiecke vor ihren Namen gekennzeichnet. Für den View *Applicable Axioms* erscheinen bei Anwendbarkeit von Axiomen die analogen Markierungen.
Die Anwendung von Regeln, Axiomen oder Taktiken auf die selektierten Blätter erfolgt nach dem gleichen Schema: Man klickt mit der Maus auf den gewünschten Namen in einem der drei Kalkülviews. Daraufhin werden alle Markierungen dieser Views zurückgenommen, der selektierte Regel-, Axiom- oder Taktikname durch eine graue Umrandung markiert und schließlich die selektierten Blätter — falls möglich — durch die Auswahl expandiert. Nur Selektionen expandierter Blätter werden aufgehoben. Abbildung 8 zeigt das Ergebnis der Expansion des selektierten Blattes aus Abbildung 7 durch Wahl der Regel *right_and*.

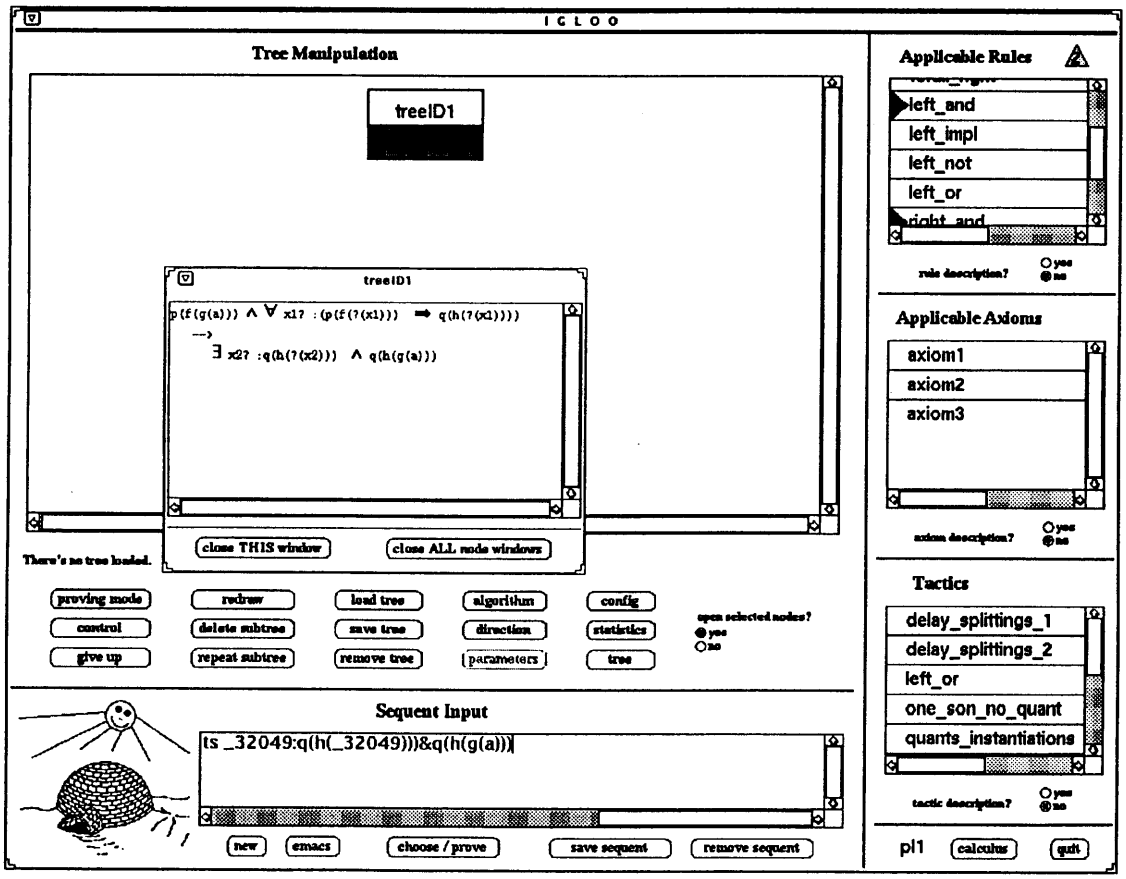
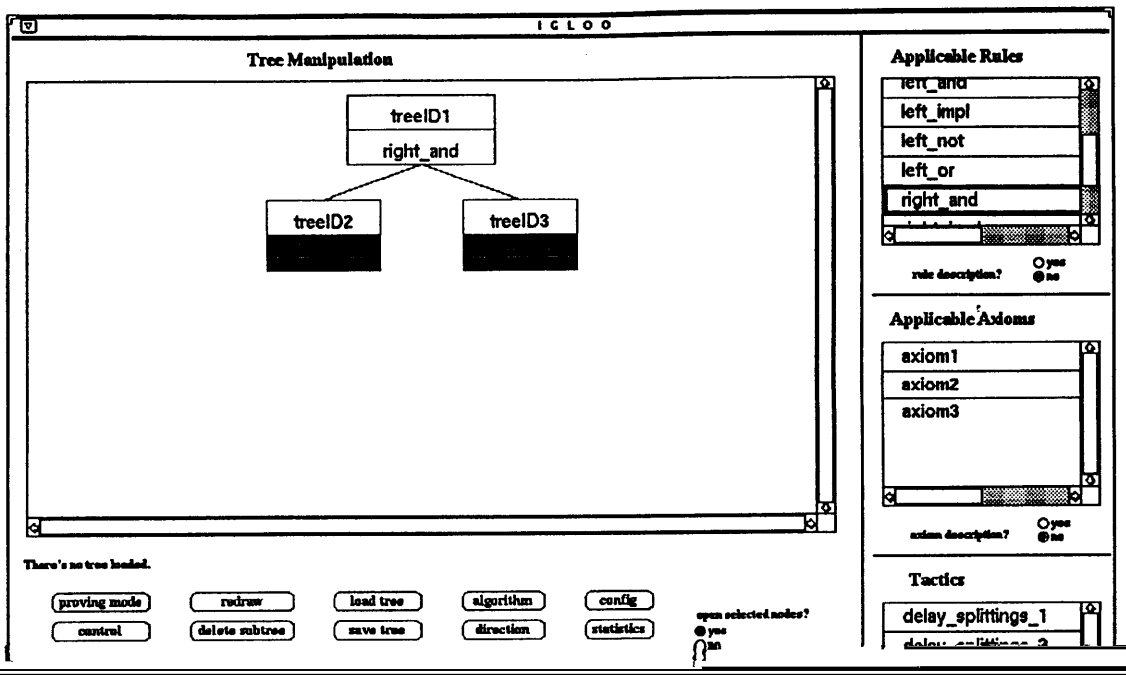


Abbildung 7: Start eines interaktiven Beweises für den Sequenten *interessant_1*


IGLOO
Tree Manipulation
treeID1
right_and
treeID2
treeID3
There's no tree loaded.
proving mode
control
redraw
delete subtree
load tree
save tree
algorithm
direction
config
statistics
open selected nodes?
yes
no
Applicable Rules
left_impl
left_not
left_or
right_and
rule description?
yes
no
Applicable Axioms
axiom1
axiom2
axiom3
axiom description?
yes
no
Tactics
delay_splittings_1

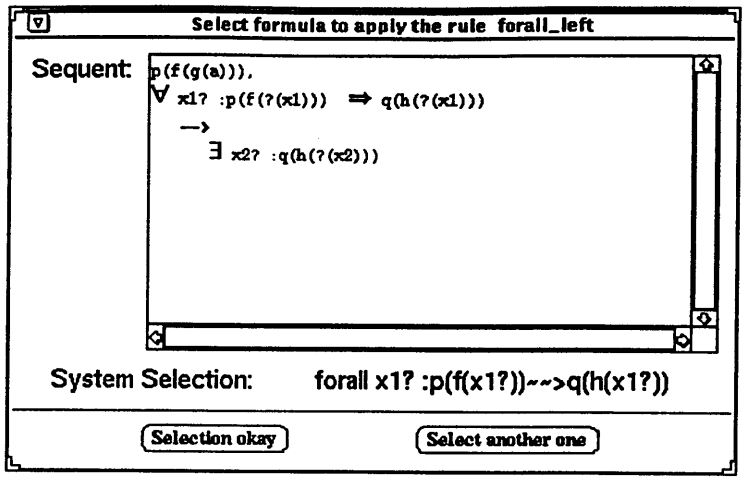


Abbildung 9: Wahl der allquantifizierten Formel, auf die *forall_left* angewandt werden soll.

zu b): Mit der Beweiswiederholung hat man die Möglichkeit, Teile des Beweises, die durch die Struktur von Teilbäumen in der Grafik repäsentiert sind, an mehreren Blättern des aktuellen Beweisbaumes wiederholen zu lassen. Bei der Beweiswiederholung wird der Anwender vom System durch Instruktionen unterstützt. Die folgenden Schritte werden durchgeführt:

i) Start der Beweiswiederholung mit ≪*repeat subtree*≫,
ii) Festlegen der zu wiederholenden Beweisstruktur,
iii) Festlegen der Zielblätter, an denen die Wiederholung gestartet werden soll,
iv) Beendung der Beweiswiederholung durch ≪*end subtree repetition*≫ in einem Instruktionsfenster.

zu ii): Die zu wiederholende Beweisstruktur wird durch Markieren eines Teilbaumes des gegenwärtig im View *Tree Manipulation* abgebildeten Ableitungsbaumes festgelegt[12]. Es werden Wurzel und Blätter des Teilbaumes durch Selektion von Knoten des Ableitungsbaumes festgelegt. Der so identifizierte Teilbaum ist im Grafikview durch Verdickung seiner Kanten und Knotenumrandungen optisch hervorgehoben.

zu iii): Die Blätter, von denen aus die Beweisstruktur wiederholt werden soll, werden der Reihe nach selektiert und nach einer weiteren Bestätigung soweit wie möglich durch die ausgewählte Beweisstruktur expandiert. Bei der Beweiswiederholung an einem Blatt wird die vorgegebene Beweisstruktur wie bei einer Tiefensuche wiederholt. Bei der ersten nicht anwendbaren Regel stoppt die Expansion, so daß evtl. nur Teile der ausgewählten Struktur wiederholt werden.

zu c): Man kann Beweisschritte wieder rückgängig machen, indem man in der Grafik Teilbäume löscht. Folgende Aktionen sind hierzu notwendig:

i) Aktivierung des Löschmodus mit ≪*delete subtree*≫,
ii) Löschen von Teilbäumen,

[12]Vgl. auch Flußdiagramm in Anhang B, S. 55.

iii) Deaktivierung des Löschmodus mit *≪end deletion mode≫* in einem Instruktionsfenster.

zu ii): Solange man im Löschmodus ist, bewirkt eine Selektion von Knoten im View *Tree Manipulation*, daß zugehörige Fenster, wie sie in Abbildung 10 zu sehen sind, geöffnet werden. Von ihnen aus kann man mit *≪Delete from THIS node≫* oder *≪Delete from ALL selected nodes≫* die Teilbäume, deren Wurzeln die selektierten Knoten sind, löschen.

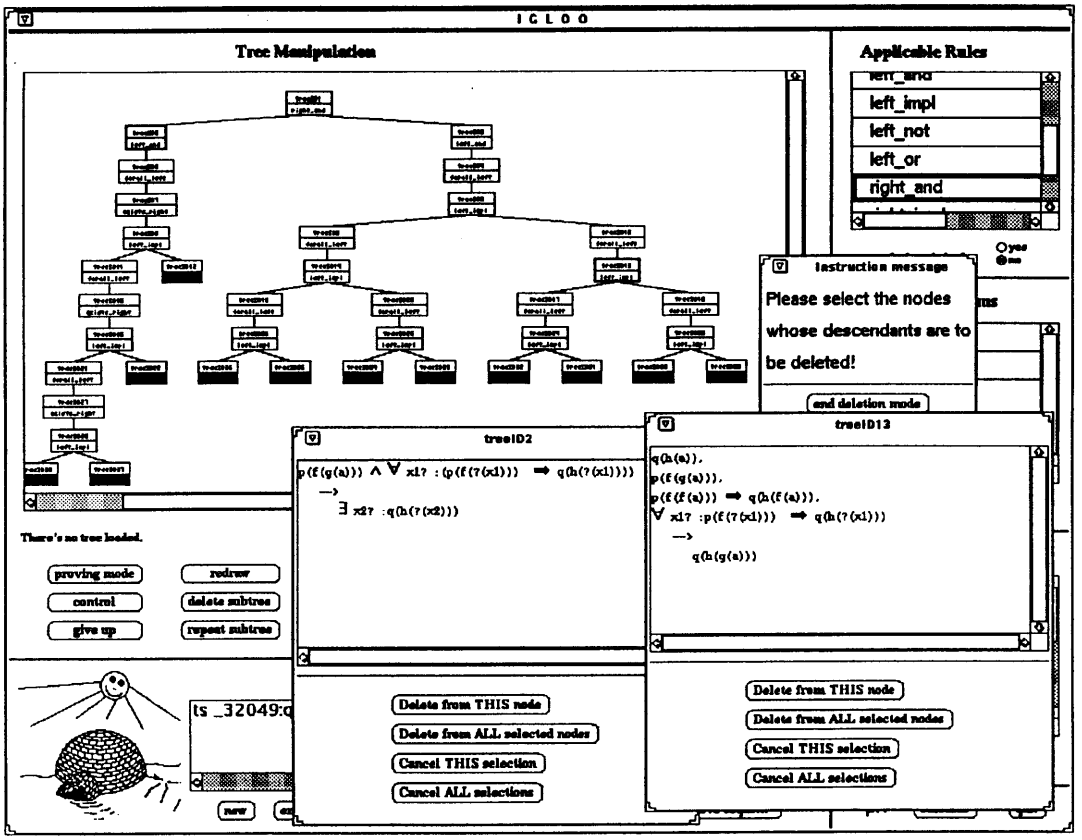


Abbildung 10: Selektion der zu löschenden Teilbäume

2.3 Zugriff auf externe Datenbasen des Systems

Die Abbildung zur Systemarchitektur auf Seite 1 zeigt, daß **IGLOO** auf Datenbasen für Beweisbäume, Sequenten, Kalküle und Systemkonfigurationen zugreifen kann. In diesem Abschnitt werden diese Verwaltungsoperationen beschrieben.

2.3.1 Verwalten von Ableitungsbäumen

Die Verwaltung von Ableitungsbäumen umfaßt folgende menügesteuerten Funktionen:

1. Abspeichern von Bäumen auf externe Prologdatenbasen.

zu 1.: Einen im View *Tree Manipulation* befindlichen Baum kann man mit ≪*save tree*≫ abspeichern. Dabei kann man als Datenbasis eine der durch das Fakt `db_dirs/1` in der Datei "~/.igloo_init" (vgl. S. 7) angegebenen Directories wählen oder eine neue Directory angeben. Diese wird vor Systemende bei der Aktualisierung der "~/.igloo_init"-Datei von **IGLOO** automatisch in die Liste des `db_dirs/1`-Faktes aufgenommen. In beiden Fällen ist die Existenz der Directories Voraussetzung für das Abspeichern. Beim Abspeichern wird vom Benutzer die Angabe eines Namens für den Baum verlangt. Ferner kann er in einem Maskenfeld *Proved By* eine weitere Information zum Beweisbaum angeben.

zu 2.: Durch das Laden eines Beweisbaumes wird der Inhalt des Views *Tree Manipulation* durch den gewünschten Baum ersetzt. Durch ≪*load tree*≫ kann man zunächst die Datenbasis, von der man einen Baum laden will, wählen. In der daraufhin angezeigten Liste der Bäume kann man sich durch Anklicken der Baumnamen die entsprechenden Startsequenten mit den beim Abspeichern im Maskenfeld *Proved By* angegebenen Informationen (s.o.) ansehen. Durch ≪*build*≫ aus einem Untermenü wird ein Baum geladen. Die Anzahl der Knoten des Baumes bestimmt, wie lange der Grafikgenerierungsprozeß dauert.

zu 3.: Mit ≪*remove tree*≫ kann man Bäume aus den Datenbasen löschen. Das Festlegen des zu löschenden Baumes geschieht analog zum Laden von Bäumen.

2.3.2 Verwalten von Sequenten

Neben dem Edieren von Sequenten, das auf Seite 15 unter Punkt **c)** beschrieben worden ist, umfaßt die Verwaltung von Sequenten folgende menügesteuerten Funktionen:

1. Abspeichern von Sequenten auf Dateien,
2. Laden von Sequenten von Dateien,
3. Löschen von Sequenten aus Dateien.

zu 1.: Ein im Textview *Sequent Input* befindlicher Sequent kann mit ≪*save sequent*≫ auf eine Datei abgespeichert werden. Die zur Auswahl stehenden Dateien sind durch das Fakt `sequent_files/1` in der Datei "~/.igloo_init" (vgl. S. 7) festgelegt. Man kann aber auch eine neue Datei zum Abspeichern angeben. Diese wird vor Systemende bei der Aktualisierung der "~/.igloo_init"-Datei von **IGLOO** automatisch in die Liste des `sequent_files/1`-Faktes aufgenommen. Vor dem Abspeichern wird der Benutzer vom System aufgefordert, einen Namen für den Sequenten anzugeben.

zu 2.: Das Laden eines Sequenten von einer Datei wurde auf Seite 16 unter Punkt **d)** erklärt.

zu 3.: Nach ≪*remove sequent*≫ wählt man zunächst die Datei, von der man Sequenten löschen will. In der daraufhin angezeigten Liste von Sequentennamen kann

man sich durch Anklicken die zugehörigen Sequenten in einem extra Fenster zeigen lassen. Von diesem löst man mit ≪*remove*≫ das Löschen aus.

2.3.3 Verwalten von Kalkülen

Beim Umgang mit Kalkülen sind die folgenden Punkte zu beachten:

1. Erstellen von Kalkülen,
2. Aufnahme der Kalküle in das System,
3. Laden von Kalkülen beim Systemlauf.

zu 1.: Kalküle werden außerhalb von **IGLOO** auf Unix-Ebene in einem Editor erstellt. Ein Kalkül setzt sich aus einer Menge von Regeln, Axiomen und Taktiken zusammen. Die Syntax der Regeln und Axiome wird durch die Produktionen einer kontextfreien Grammatik angegeben. Der Aufbau von Taktiken wird anschließend beispielhaft an einer Taktik gezeigt, um ein intuitives Verständnis zu vermitteln. Die genauen Definitionen der durch die Taktiksprache angebotenen Konstrukte sind in Anhang A angegeben.

Regeln und Axiome werden in der gleichen abstrakten Syntax ediert:

Rule ::= *Rulename* :: *KonPräm* ==> *Prämissenliste*
::- `Prologbody`.
Rulename ::= `Prologatom`
KonPräm ::= (*Antezedens* --> *Sukzedens*)
Antezedens ::= `Prologvariable` | `conj(`*ConjOrDisj+*`)`
Sukzedens ::= `Prologvariable` | `disj(`*ConjOrDisj+*`)`
ConjOrDisj+ ::= *Formel*, *ConjOrDisj** | `$Prologvariable`, *ConjOrDisj** |
Formel, `$Prologvariable`, *ConjOrDisj**
Formel ::= `Formelsyntax der betreffenden Logik`
*ConjOrDisj** ::= ϵ | *Formel*, *ConjOrDisj** |
[redacted]

Formeln sind so definiert:

Formel ::= *Atomare Formel* | *Komplexe Formel*
Atomare Formel ::= `true` | `false` | `Prologatom` | *Prädikatensymbol*(*TermArgs*)
Prädikatensymbol ::= `Prologatom`
Komplexe Formel ::= ∼(*Formel*)
Komplexe Formel ::= (*Formel* `&` *Formel*)
Komplexe Formel ::= (*Formel* `or` *Formel*)
Komplexe Formel ::= (*Formel* ∼∼> *Formel*)
Komplexe Formel ::= (*Formel* `iff` *Formel*)
Komplexe Formel ::= (`forall` *Variable* : *Formel*)
Komplexe Formel ::= (`exists` *Variable* : *Formel*)

Ein Beispiel für eine Formel ist:

```
(forall X : ((p(X) & (~ p(1))) ~~> false))
```

Da sich eine Regel im allgemeinen auf eine Klasse von Formeln beziehen soll, die auf Toplevel eine gewisse Struktur haben und nicht nur auf eine ganz spezielle Formel, muß bei der Regeldefinition die Möglichkeit bestehen, eine Art Platzhalter für Formeln anzugeben. Dies wird erreicht, indem bei der Regeldefinition zusätzlich folgende Ergänzung zur kontextfreien Grammatik der jeweiligen Logik gemacht wird:

Formel ::= `Prologvariable`

Nach diesen Definitionen werden ein paar Regeln des Kalküles für PL1 angegeben. Das komplette Kalkül kann man sich in der Datei *IGLOO/GALLIER/gallier_rules.pl* ansehen.

```
• forall_right ::  (G --> disj($ X, forall Var :  A, $ Y)) ==>
                   [(G --> disj($ X, A_Instance, $ Y))]
                   ::- ^@ replace_free_variable_by_new_variable_
                         in_formula(Var, A, A_Instance).
```

An dieser Regel sieht man, daß mittels der Prologvariablen `G` der gesamte Antezedens der Konklusion (auf sie wird ja die Regel angewandt) in die Prämisse übernommen wird. Neben solchen "normalen" Prologvariablen werden bei den Regeln auch die sogenannten *Sequenzenvariablen* der Form `$Prologvariable` benutzt. Sie stehen stellvertretend für beliebig viele Formeln. In diesem Beispiel werden mit `$ X` alle Formeln, die im Sukzedens der Konklusion vor der allquantifizierten Formel stehen, in den Sukzedens der einzigen Prämisse übernommen. Hinter "`::-`" kann ein beliebiger Prologrumpf angegeben werden. Er wird bei Ausführung der Regel zwischen der Analyse der Konklusion und der Synthese der Prämissen ausgeführt. Vor jedes Prologliteral, das im Rumpf angegeben wird, kann man den Operator "`^@`" schreiben. Falls er vor einem Literal angegeben wird, so wird das Literal nur bei der tatsächlichen Regelanwendung beachtet, nicht aber beim Test auf Anwendbarkeit einer Regel, der beim interaktiven Beweisen nach jeder Knotenselektion für alle Regeln durchgeführt wird. Diese Möglichkeit ist besonders bei Seiteneffekten sinnvoll, also bei *assert*-Anweisungen oder Ähnlichem im Prologrumpf, die nur dann ausgeführt werden sollen, wenn die Regel auch wirklich angewandt wird und nicht

schon bei ihrem Test. Einige Regeln benötigen den Prologrumpf nicht. In diesem Fall wird einfach `true` angegeben.
Obige Regel wäre beispielsweise auf folgende Sequenten anwendbar:

1. `p1 --> p2, p3, forall A : q(A), p4.` Bei diesem Sequenten würde `X` an die Liste `[p2, p3]` und `Y` an die Liste `[p4]` gebunden werden. Als Ergebnissequent käme `p1 --> p2, p3, q(Z1), p4` heraus.

2. `p1 --> forall A : q(A).` Hier würden `X` und `Y` an die leere Liste `[]` gebunden werden, und als Prämisse käme `p1 --> q(Z2)` heraus.

```
• right_and ::  (G --> disj($ X, A & B, $ Y)) ==>
                [(G --> disj($ X, A, $ Y)),
                 (G --> disj($ X, B, $ Y))] ::- true.
```

Diese Regel stellt einen Fall dar, bei dem zwei Prämissen erzeugt werden. Hier wird deutlich, wie mittels der Sequenzenvariablen vom Sukzedens der Konklusion alle Formeln eines Sequenten außer der Konjunktion `A & B` in die Sukzedenten beider Prämissen übernommen werden. Aus der Konjunktion selbst wird aber nur jeweils ein Konjunkt in jede Prämisse übernommen. Der Prologrumpf ist hier trivial (`true`).

• Auch logische Axiome werden auf diese Weise erstellt. Ein Beispiel ist

```
axiom1 ::  (conj($ _, A, $ _) --> disj($ _, A, $ _)) ==>
           [(conj(false) --> disj(true))] ::- true.
```

Dieses Axiom testet, ob es eine Formel `A` gibt, die sowohl im Antezedens als auch im Sukzedens eines Sequenten auftritt. Dabei kann sie an beliebiger Stelle dort stehen. Wenn man den Sequenzenpfeil "`-->`" als Implikation, die Formeln im Antezedens als konjunktiv, die im Sukzedens als disjunktiv verknüpft interpretiert, so sieht man schnell ein, daß es sich hierbei um ein Axiom handelt.

In Abbildung 11 ist ein Beispiel einer prädikatenlogischen Taktik zu sehen. Die verwendeten Sprachkonstrukte wie *orelse*, *call_tac* oder *then* sind mit den übrigen in Anhang A beschrieben.

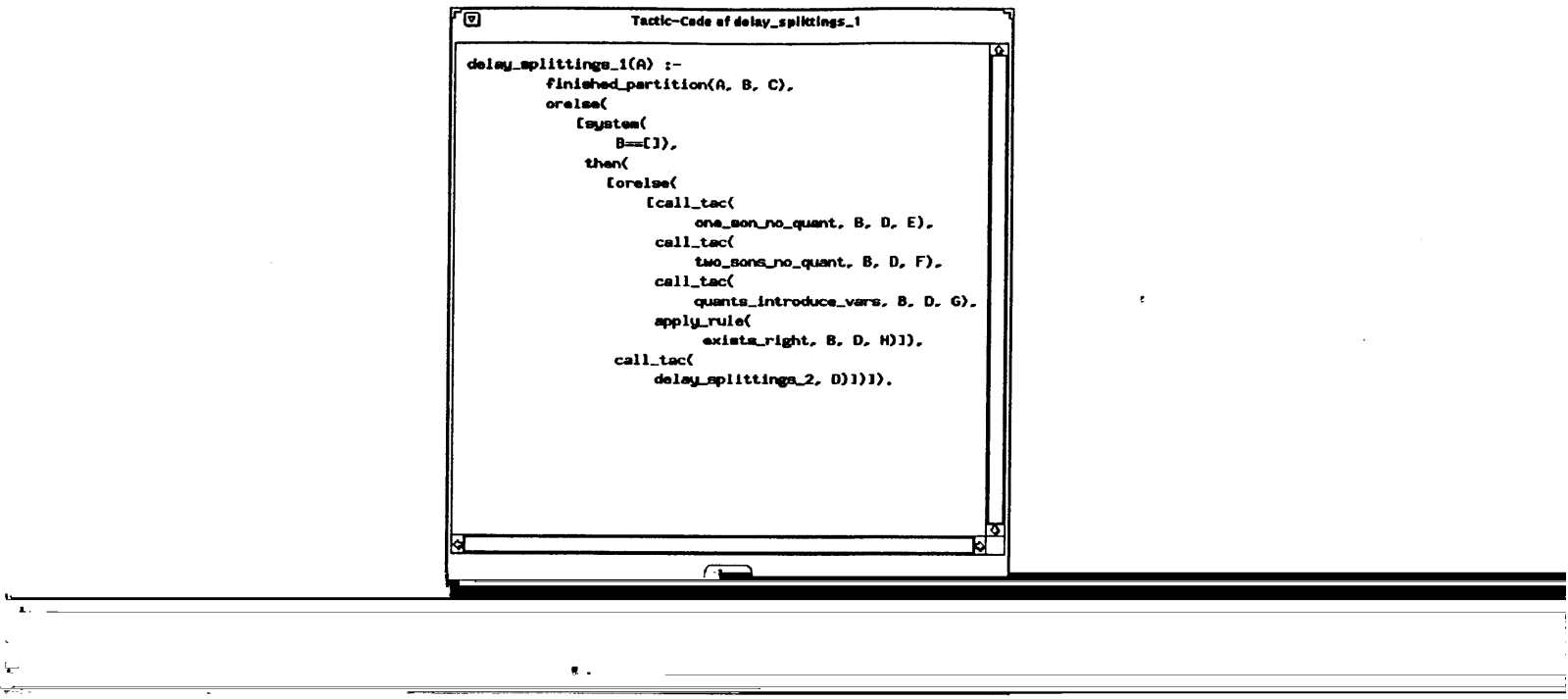


Abbildung 11: Eine prädikatenlogische Taktik

Die abgebildete Taktik verfolgt die Idee, aussagenlogische Regeln immer den den Suchraum extrem vergrößernden Instantiierungsregeln vorzuziehen. Den Effekt, den man durch geschickte Taktiken erreichen kann, kann man erkennen, wenn man die in Abbildung 12 gezeigte Statistik des Taktikbeweises für den Sequenten *dieb_jacky* mit der angegebenen Taktik und die Statistik zum automatischen Beweis dieses Sequenten in Abbildung 4 (vgl. S. 17) miteinander vergleicht (Statistiken zu Beweisbäumen im View *Tree Manipulation* kann man sich mit ≪*statistics*≫ anzeigen lassen). Die Anzahl der Knoten im Taktikbeweis hat sich gegenüber der des automatischen Beweises fast auf ein Viertel verkleinert.

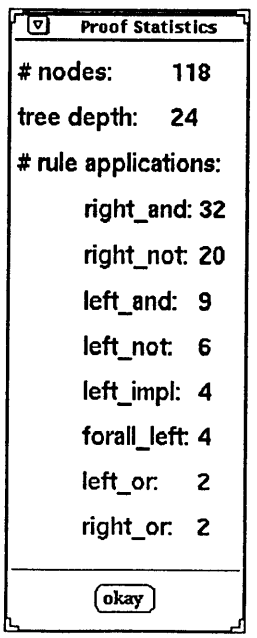


Abbildung 12: Statistik zu einem Taktikbeweis für den Sequenten *dieb_jacky*

zu 2.: Ein Kalkül darf auf mehrere Dateien verteilt sein. Man kann z.B. alle Regeln und Axiome auf eine Datei schreiben und alle Taktiken des Kalküles auf eine zweite Datei. In der Datei, in der die Taktiken ediert werden, muß am Anfang für jede definierte Taktik eine *multifile*- und eine *dynamic*-Deklaration stehen. Vor dem Laden des Systems muß jedes neue Kalkül, das während des Sytemlaufes ladbar sein soll, in die Liste des Faktes `calculi/1` der Datei "~/.igloo_init" (vgl. S. 7 ff.) aufgenommen werden. Ein Kalkül wird dabei durch eine dreielementige Liste [*Kalkülname, Taktikliste, Kalküldateien*] repräsentiert. *Taktikliste* enthält für jede definierte Taktik einen Eintrag *Name/Stelligkeit*. Für obige Beispieltaktik würde `delay_splittings_1/1` der Eintrag sein. *Kalküldateien* ist die Liste der vollständigen Pfadnamen aller Dateien, die das Kalkül konstituieren (incl. der Dateien, welche Hilfsprozeduren für das entsprechende Kalkül enthalten).

zu 3.: Während des Systemlaufes kann zwischen verschiedenen Kalkülen gewechselt werden. Zur Auswahl stehen die in der Liste des Faktes `calculi/1` in der Datei "~/.igloo_init" (vgl. S. 7 ff.) aufgeführten Kalküle. Mit ≪*calculus*≫ kann man einen zum Laden wählen. Nach dem Laden wird links neben dem Button ≪*calculus*≫ der Name des gewählten Kalküles angezeigt, und die Inhalte der drei Kalkülviews *Applicable Rules*, *Applicable Axioms* und *Tactics* werden durch die Namen der neuen Regeln, Axiome und Taktiken ersetzt.

2.3.4 Verwalten von Systemkonfigurationen

Systemkonfigurationen sind durch Parameterwerte, welche sowohl die Beweis- als auch die Grafikkomponente von **IGLOO** beeinflussen, gegeben. Konfigurationen können auf Dateien gespeichert werden. Eine Datei enthält dabei genau eine Konfiguration. Die Verwaltung der Konfigurationen umfaßt folgende Funktionen:

1. Abspeichern von Konfigurationen,

2. Aktivieren von Konfigurationen,

3. Festlegen der beim Systemstart initialen Konfiguration.

zu 1.: Durch ≪*config*≫bekommt man in einem Fenster die aktuelle Konfigura-

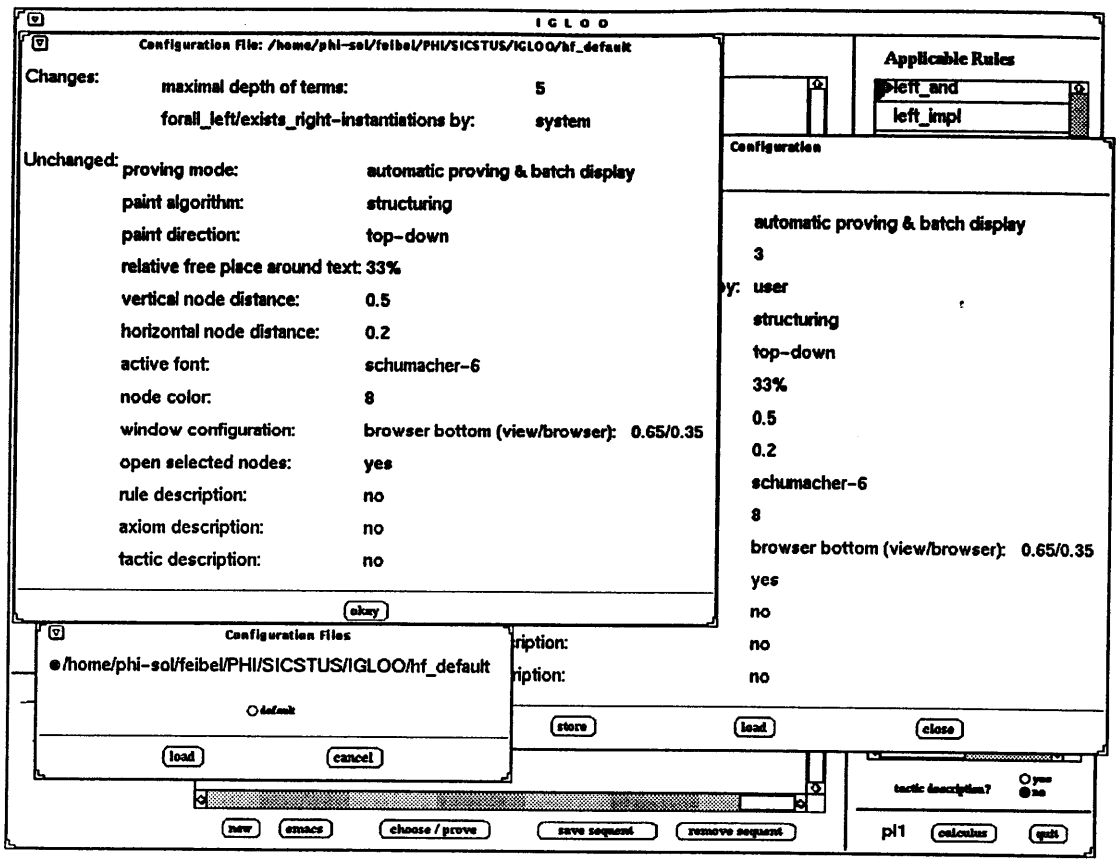


Abbildung 13: Laden einer abgespeicherten Konfiguration

zu 2.: Zuerst läßt man sich mit ≪*config*≫ die aktuelle Konfiguration anzeigen. In diesem Fenster leitet man mit ≪*load*≫ das Laden einer neuen Konfiguration ein. Anschließend kann man sich die Parametereinstellungen der angebotenen Konfigurationsdateien bzw. einer Standardkonfiguration *default* anzeigen lassen. Bei der Auflistung werden zuerst die Werte der Parameter gezeigt, die sich von der aktuellen Einstellung unterscheiden und darunter die unveränderten Werte (vgl. Abbildung 13). Nach dem Laden werden die Werte aller Parameter durch die der gewählten Konfiguration ersetzt.

zu 3.: Beim Systemstart wird immer die Konfiguration aktiviert, deren zugehöriger Dateiname in der Liste des Faktes `config_files/1` in der Datei "~/.igloo_init" an erster Stelle steht. Soll eine bestimmte Konfigurationsdatei die initiale Konfiguration festlegen, so muß die Datei "~/.igloo_init" entsprechend geändert werden.

2.4 Arbeiten mit LLP-Kalkülen

Beim Arbeiten mit Kalkülen für LLP (vgl. [BD93] u. [BBD+92]) sind bei den folgenden Schritten Besonderheiten zu beachten:

1. Erstellen von Regeln,

2. Aufnahme eines Kalküles in die Datei "~/.igloo_init",

3. Versehen der Sequenten/Planspezifikationen mit Sorteninformationen.

zu 1.: Bei der Anwendung von Regeln aus Kalkülen für LLP werden häufig in den zugehörigen Prologrümpfen Seiteneffekte erzeugt, z.B. durch `assert`- oder `retract`-Operationen. Diese Seiteneffekte müssen von **IGLOO** mitprotokolliert werden können, damit die Prologdatenbasis — z.B. nach dem Löschen von Teilbäumen — immer den in der Grafik vorhandenen Knoten des Baumes entspricht. Im Rumpf von Regeln und in Prozeduren, die aus dem Rumpf einer Regel direkt oder indirekt aufgerufen werden, müssen darum folgende Ersetzungen vorgenommen werden:

```
assert     ⇝ rb_assert
asserta    ⇝ rb_asserta
assertz    ⇝ rb_assertz
retract    ⇝ rb_retract
retractall ⇝ rb_retractall
```

`rb_assert` ruft ein entsprechendes `assert` auf, erledigt aber zusätzlich die erforderliche Protokollierungsaufgabe. Analog verhält es sich mit den anderen Anweisungen.

Des weiteren können Seiteneffekte durch Prozeduren auftreten, die im Rahmen der Anwendung von Axiomenschemata aufgerufen werden. In solchen Fällen muß zuerst untersucht werden, welche Prädikate dabei auf die Prologdatenbasis abgelegt oder von ihr gelöscht werden. Hat man diese Menge ermittelt, so sind zwei Schritte durchzuführen:

i) Die Prädikate müssen dem System mitgeteilt werden. Einige Prädikate kennt das System schon. Dies sind `to_replace_for/2`, `to_replace_for_pre/2`, `effect_of/2`, `action_name_of/2`, `to_replace_for/3`, und `effect_intro_entry/1`. Für die dem System noch nicht bekannten Prädikate muß in einer der Dateien, die in der Liste *Files* des zugehörigen Kalküles (vgl. Datei "~/.igloo_init", S. 7 ff.) namentlich aufgeführt sind, noch folgendes ergänzt werden, wobei hier angenommen wird, daß es zwei solche Prädikate `fakt1/3` und `fakt2/2` gibt:

```
:- retract(possible_llp_side_effects(Known_Facts_List)),
   assert(possible_llp_side_effects([fakt1(_,_,_), fakt2(_,_)|
                                     Known_Facts_List])).
```

ii) Jedes `assert`, `asserta`, `assertz`, `retract`, und `retractall`, das ein neues oder schon bekanntes solches Prädikat als Argument hat, muß wie folgt ergänzt werden (am Beispiel von `asserta` und `fakt1/3`):

```
asserta(fakt1(a,b,c))   ⇝   asserta(fakt1(a,b,c)),
                            update_node_dependencies(asserta,
                                                     fakt1(a,b,c))
```

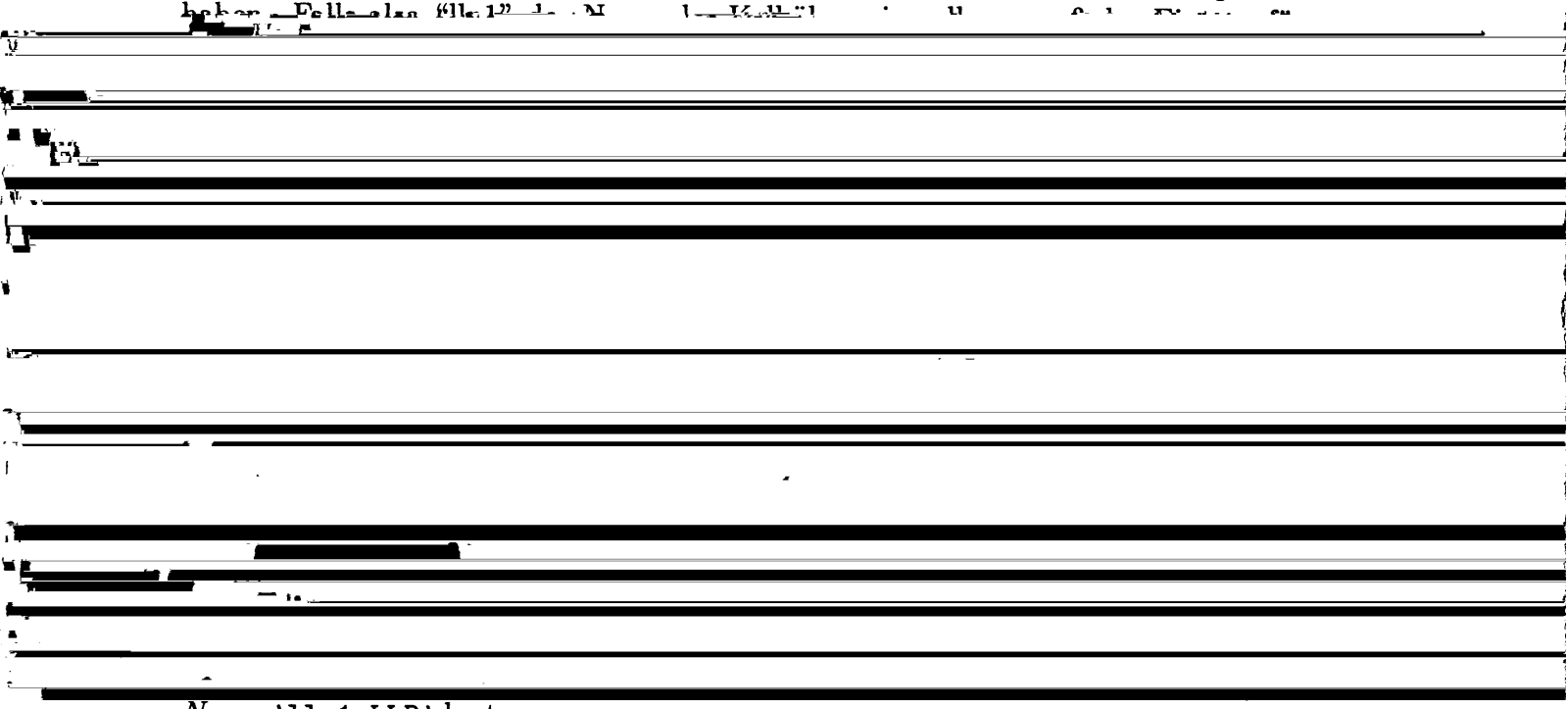
Generell sollte man Seiteneffekte durch Regelanwendungen möglichst vermeiden. Der Gebrauch des Operators "`^@`", der insbesondere bei Operationen wie `assert` und `retractall` bewirkt, daß diese beim Regeltest nicht ausgeführt werden, wurde schon im Abschnitt 2.3.3 auf Seite 27 beschrieben. Folgendes ist dabei aber noch

zu beachten: Eine `retract`-Operation im Rumpf einer Regel stellt nicht nur eine Seiteneffektoperation, sondern auch eine Bedingung dar, denn sie liefert `fail`, falls das entsprechende Prädikat nicht existiert. Setzt man vor eine solche Operation den Operator "`^@`", so erreicht man zwar, daß die Seiteneffektoperation beim Anwendbarkeitstest der Regel nicht ausgeführt wird, jedoch wird der Test selbst verfälscht. Dieses Problem kan gelöst werden, indem man eine Ersetzung wie im folgenden Beispiel vornimmt:

```
retract(my_fact(X,a)) ⇝ on_exception(_, my_fact(X,a), fail),
                         ^@ rb_retract(my_fact(X,a))
```

Auf diese Weise wird beim Regeltest sowohl geprüft, ob das entsprechende Prädikat existiert, als auch der Seiteneffekt verhindert.

zu 2.: In der Liste des Faktes `calculi/1` (vgl. Datei "~/.igloo_init", S. 7 ff.) werden Kalküle durch eine Liste der Form [*Name*, *Taktiken*, *Files*] repräsentiert. *Name* ist dabei ein Prologatom. Bei Kalkülen für LLP muß dieses Atom die Endung `:LLP`



Name `'llp1:LLP'` lauten.

zu 3.: Jedem Startsequenten bzw. jeder Planspezifikation sind in den Kalkülen der Logik LLP eigene Sortendeklarationen zugeordnet. Beim Laden eines Sequenten von einer Datei (vgl. Fakt `sequent_files/1` in der Datei "~/.igloo_init", S. 7 ff.) übernimmt das System die entsprechende Sorteninformation aus der Repräsentation des Sequenten in der Datei. Das Format, in dem ein Sequent abgespeichert ist, muß ein Benutzer nur dann kennen, wenn er mit einem Kalkül für die Logik LLP arbeitet. Es sieht wie folgt aus:

`plan_spec(`*Seq_Name*, *List_of_Sort_Declarations*, *Sequent*`)`.

Bei Startsequenten von Logiken ungleich LLP ist die *List_of_Sort_Declarations* leer. Bei einem LLP-Startsequenten kann diese Liste z.B. so aussehen:

```
[global_var(x, integer),
 global_var(y, integer),
 local_var(system_mbox, mbox),
 local_var(screen_display, list),
 constant(joe, sender)]
```

In beiden Fällen kann man anschließend vom System aus den Sequenten von der Datei laden, wobei dann auch die Sorteninformationen zur Verfügung stehen. Beim Edieren einer Planspezifikation sollte für die Plan-Metavariable `meta(metaps1)` angegeben werden. Weitere Details über die Logik LLP können [BD93] entnommen werden.

3 Grafische Darstellung von Beweisbäumen

Die Oberfläche "Proof Tree" (vgl. Abb. 14) dient zur grafischen Darstellung von Beweisbäumen. Sie wird von "IGLOO" mit $\ll tree \gg$ erreicht. Ihre wesentlichen Komponenten sind

- der Ausschnittsview *Selected Tree Part*,
- der Browserview *Tree Browser*,
- die Buttonleiste.

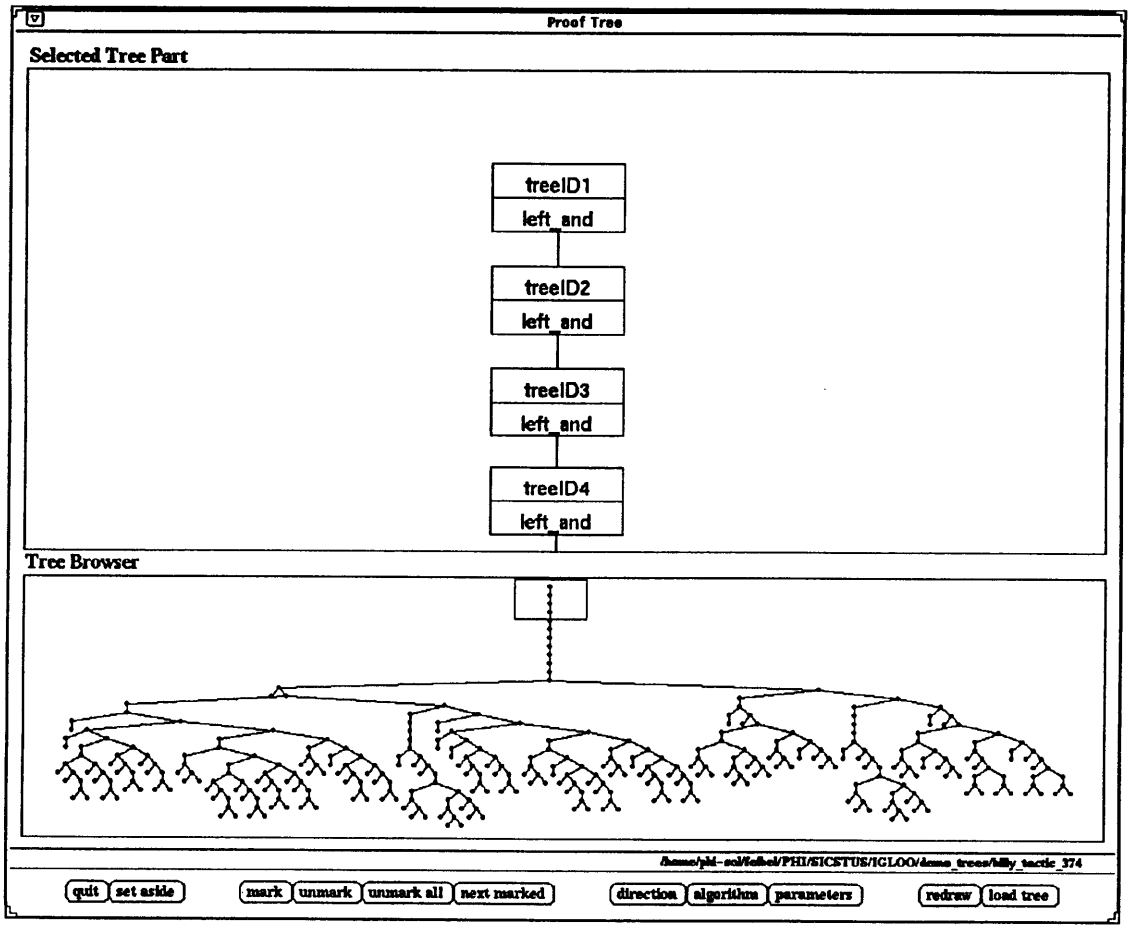


Abbildung 14: Die Grafikoberfläche "Proof Tree"

In dem Browserview *Tree Browser* ist ein Baum immer in seiner Gesamtgestalt zu sehen. Das kleinere Rechteck, das man im Browserview erkennt, bestimmt den Teil des Baumes, der in dem Ausschnittsview *Selected Tree Part* so dargestellt wird, wie man es von der Oberfläche "IGLOO" her kennt. Mit dem Rechteck kann wie folgt navigiert werden: Durch Anklicken eines Punktes im View *Tree Browser* wird das Rechteck so verschoben, daß sein Zentrum genau auf der selektierten Position liegt. Der so bestimmte Teil des Gesamtbaumes wird im Ausschnittsview angezeigt. Die Form des Rechteckes ist sowohl durch die Konstellation des Fensters "Proof Tree" (vgl. S. 37 ff.) als auch durch die aktuellen Knotenparameter (vgl. S. 40 ff.) bestimmt. Es enthält immer genau die Knoten des Baumes, die man auch im View

Selected Tree Part sieht. Die Knoten des Ausschnittsviews sind maussensitiv. Die Buttons werden nun kurz erläutert:

≪*quit*≫ Mit diesem Button wird das Fenster "Proof Tree" geschlossen. Der zu diesem Zeitpunkt in seinen Views dargestellte Baum wird auf der auf diese Art und Weise wieder erreichten "IGLOO"-Oberfläche dann neu aufgebaut, wenn er sich in seiner grafischen Gestalt von dem Baum, der sich eventuell schon im View *Tree Manipulation* des Fensters "IGLOO" befindet, unterscheidet.

≪*set aside*≫ Mit diesem Button kann man ähnlich wie mit ≪*quit*≫ von "Proof Tree" zu "IGLOO" wechseln. Seine Nutzung ist dann sinnvoll, wenn man erwartet, daß man bald wieder zu "Proof Tree" zurückwechseln wird. Mit ≪*set aside*≫ wird das Fenster nur ikonifiziert, so daß bei erneutem ≪*tree*≫ der Baum in den Views von "Proof Tree" nicht neu erzeugt zu werden braucht, sondern einfach das Ikon wieder zu einem Fenster, welches genauso aussieht wie vorher, hergestellt wird. Wurde vor dem Wechsel der Baum auf "IGLOO"-Seite modifiziert, so kann man die Änderung jetzt auch hier mit ≪*redraw*≫ wirksam machen.

≪*mark*≫ Hiermit kann man den zuletzt selektierten Knoten des Views *Selected Tree Part* markieren, so daß man ihn leicht wieder finden kann (vgl. ≪*next marked*≫). Die Markierung wird grafisch durch die Verdickung der Linie, die den Knotennamen von dem Regelnamen trennt, dargestellt. Zu einem Zeitpunkt können beliebig viele Knoten markiert sein.

≪*unmark*≫ Vom zuletzt angeklickten Knoten wird die Markierung wieder gelöscht.

≪*unmark all*≫ Alle Knotenmarkierungen werden gelöscht.

≪*next marked*≫ Der nächste markierte Knoten erscheint im Zentrum des Views *Selected Tree Part*. Das Rechteck des Browsers wird automatisch auf die richtige Position im View *Tree Browser* verschoben. Die markierten Knoten werden in einer zirkulären Liste verwaltet. Zum ersten Knoten dieser Liste wird navigiert, und anschließend wird er vom Anfang ans Ende der Liste gestellt.

≪*direction*≫, ≪*algorithm*≫, ≪*parameters*≫ Die Erklärung zu diesen Buttons kann man auf Seite 12 nachlesen, da sie auch auf der Oberfläche "IGLOO" vorhanden sind.

≪*redraw*≫ Dieser Button wird nur unmittelbar nach dem Wechsel von "IGLOO" zu "Proof Tree" durch ≪*tree*≫ gebraucht. Dabei sind zwei Fälle zu unterscheiden:

1 Beim Wechsel existiert kein "Proof Tree" Ikon

≪*load tree*≫ einen anderen Baum aus den Datenbasen.
In Fall 2 wird das Ikon "Proof Tree" wieder zu der entsprechenden Oberfläche hergestellt. Die Views haben den gleichen Inhalt wie im Moment der Ikonifizierung durch ≪*set aside*≫. Wurde dieser Baum aber auf der "IGLOO"-Oberfläche verändert, so kann man nun mit ≪*redraw*≫ die beiden Views aktualisieren.

≪*load tree*≫ Dieser Button hat die gleiche Bedeutung wie auf der Oberfläche "IGLOO" (vgl. S. 12).

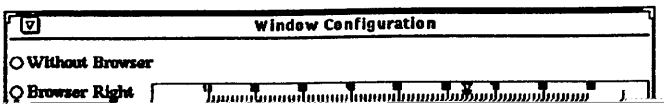
3.1 Individuelles Layout der Grafikoberfläche

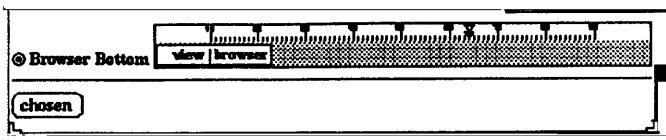
Der Aufbau der Grafikoberfläche kann variiert werden. Folgende Möglichkeiten sind gegeben:

- Kein Browserview, Navigieren mit Scrollbars,

In den beiden letzten Fällen kann eingestellt werden, welchen Anteil die Views jeweils an der gesamten Fensterfläche haben. Das Layout wird mit folgenden Schritten festgelegt:

1. Öffnen des Fensters "Parameter Manipulation" durch ≪*parameters*≫ (vgl. Abb. 15),


Window Configuration
Without Browser
Browser Right


Browser Bottom
view
browser
chosen

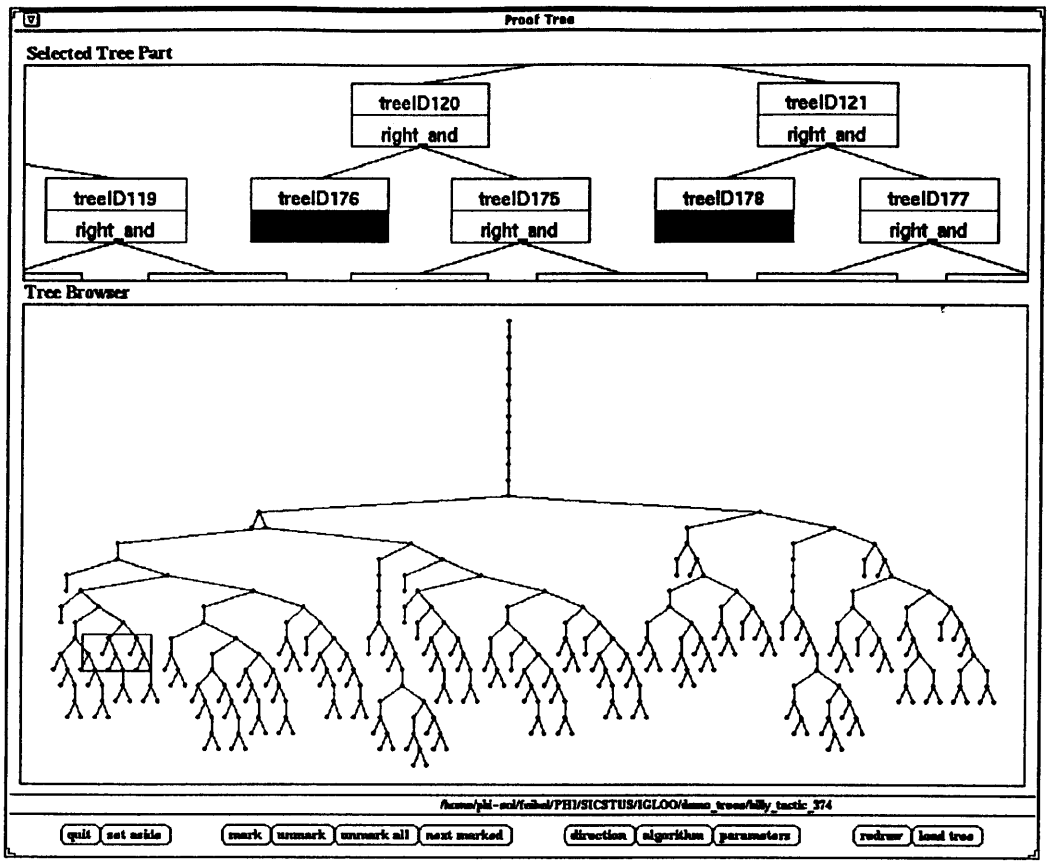


Abbildung 17: *Browser Bottom*, Slider: 30

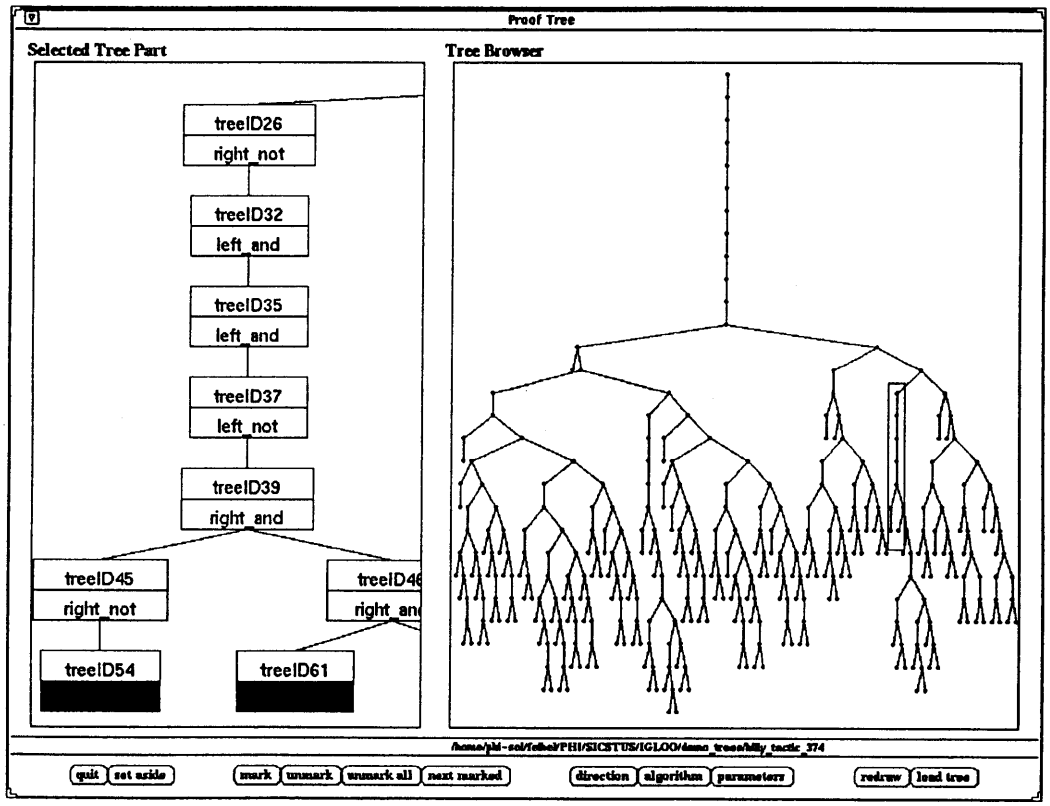


Abbildung 18: *Browser Right*, Slider: 40

3.2 Die Parameter der Baumgrafik

Die Parameter, die das grafische Bild eines Baumes festlegen, gehören jeweils genau einer der folgenden Klassen an:

1. Parameter, die die Gestalt der Knoten betreffen,
2. Parameter, die das Aussehen eines gesamten Baumes betreffen.

zu 1.: Um die Knotenparameter zu setzen, sind folgende Schritte durchzuführen:

a) Öffnen des Fensters "Parameter Manipulation" durch ≪*parameters*≫ (vgl. Abb. 15),

b) Auswahl der gewünschten Parameter,

c) Bestätigung der Änderungen im Fenster "Parameter Manipulation".

zu a): Die oberen fünf rechteckigen Buttons des Fensters "Parameter Manipulation" repräsentieren je einen Knotenparameter. Sie werden nun kurz erläutert:

≪*relative free place around text*≫ Man kann zwischen sechs Werten auswählen, die festlegen, wie groß der Platz zwischen dem Textzug in einem Knoten und den Knotenrändern sein soll.

≪*vertical node distance*≫ Der Abstand zwischen den Knoten zweier aufeinanderfolgender Levels des Baumes wird als Vielfaches der Knotenhöhe festgelegt. Zur Auswahl stehen neun Werte.

≪*horizontal node distance*≫ Der Abstand, der zwischen zwei Knoten desselben Levels im Baum mindestens vorhanden sein muß, wird als Vielfaches der maximalen Knotenbreite festgelegt. Zur Auswahl stehen elf Werte.

≪*active font*≫ In jedem Knoten steht ein identifizierender Namen und eventuell ein Regelname. Es wird einer von vier Fonts gewählt, mit dem diese beiden Schriftzüge dargestellt werden.

≪*node color*≫ Die Farbe der Knoten wird festgelegt. Man kann zwischen 9 verschiedenen Graustufen (von *weiß* bis *schwarz*) wählen. Mit der Wahl *weiß* werden die Grafiken am schnellsten erzeugt.

zu b): In den Fenstern, die durch die eben beschriebenen Buttons geöffnet werden, wird der gewünschte Parameterwert mit der Maus selektiert und durch ≪*chosen*≫ in demselben Fenster die Auswahl abgeschlossen.

zu c): Nachdem alle gewünschten Parameter verändert worden sind, kann man die Übernahme der neuen Werte durch das System entweder durch ≪*set*≫ oder durch ≪*set & redraw*≫ in dem Fenster "Parameter Manipulation" erreichen. ≪*set*≫ bewirkt nur die Übernahme, ≪*set & redraw*≫ bewirkt zusätzlich den Neuaufbau der Grafik mit den neuen Parameterwerten.

zu 2.: Es gibt zwei Parameter, die das Aussehen des Gesamtbaumes betreffen:

a) der Algorithmus-Parameter,

b) der Direction-Parameter.

zu a): Mit dem Algorithmusparameter kann man einen von zwei Algorithmen, die die Verteilung der Knoten des Baumes in der Ebene festlegen, auswählen:

Der Zentrieralgorithmus Er ordnet alle Knoten eines Levels äquidistant auf dem Level an, wobei die Wurzel des Baumes die Symmetrieachse der Levels bestimmt. Diese Darstellung ist zur *quantitativen Analyse* eines Beweisbaumes gut geeignet, da man mit ihr sehr gut sieht, welche Breite (in der Anzahl der Knoten) der Baum auf jedem Level hat.

Der Strukturieralgorithmus[13] Zur Unterstützung der *strukturellen Analyse* eines Beweisbaumes sollte der Baum mit diesem Algorithmus erzeugt werden. Teilbeweise gleicher Struktur sind in so generierten Bäumen sehr leicht an ihrer gleichen grafischen Darstellung zu erkennen.

Mit ≪*algorithm*≫ wählt man den gewünschten Algorithmus aus. Durch ≪*set*≫ oder ≪*set & redraw*≫ in dem entsprechenden Auswahlfenster übernimmt das System den gewählten Wert bzw. baut zusätzlich den aktuell in den Views dargestellten Baum mit dem gewählten Algorithmus neu auf. In allen bisherigen Abbildungen wurde der *Strukturier*algorithmus benutzt. Abbildung 19 zeigt eine Anwendung des *Zentrier*algorithmus.

zu b): Mit dem Direction-Parameter kann man festlegen, wo die Wurzel des Baumes und wo seine Blätter liegen sollen. Die beiden Werte, die man auswählen kann, sind:

top-down Die Wurzel des Baumes ist oben, seine Blätter befinden sich unten. Diese Darstellung ist *tiefenorientiert*, d.h. mit ihr kann man die meisten aufeinanderfolgenden Levels eines Baumes gleichzeitig im View *Selected Tree Part* sehen.

left-right Die Wurzel des Baumes ist links, seine Blätter liegen rechts. Diese Darstellung ist *breitenorientiert*, d.h. man kann so die meisten Knoten eines Levels gleichzeitig im View *Selected Tree Part* sehen.

Die Auswahl erfolgt analog zur Algorithmuswahl (s.o.) durch ≪*direction*≫. Abbildung 20 zeigt einen Baum mit Einstellung *left-right*.

[13] Zu den Grundlagen des Strukturieralgorithmus vgl. [DT90], [DT91], [M...]

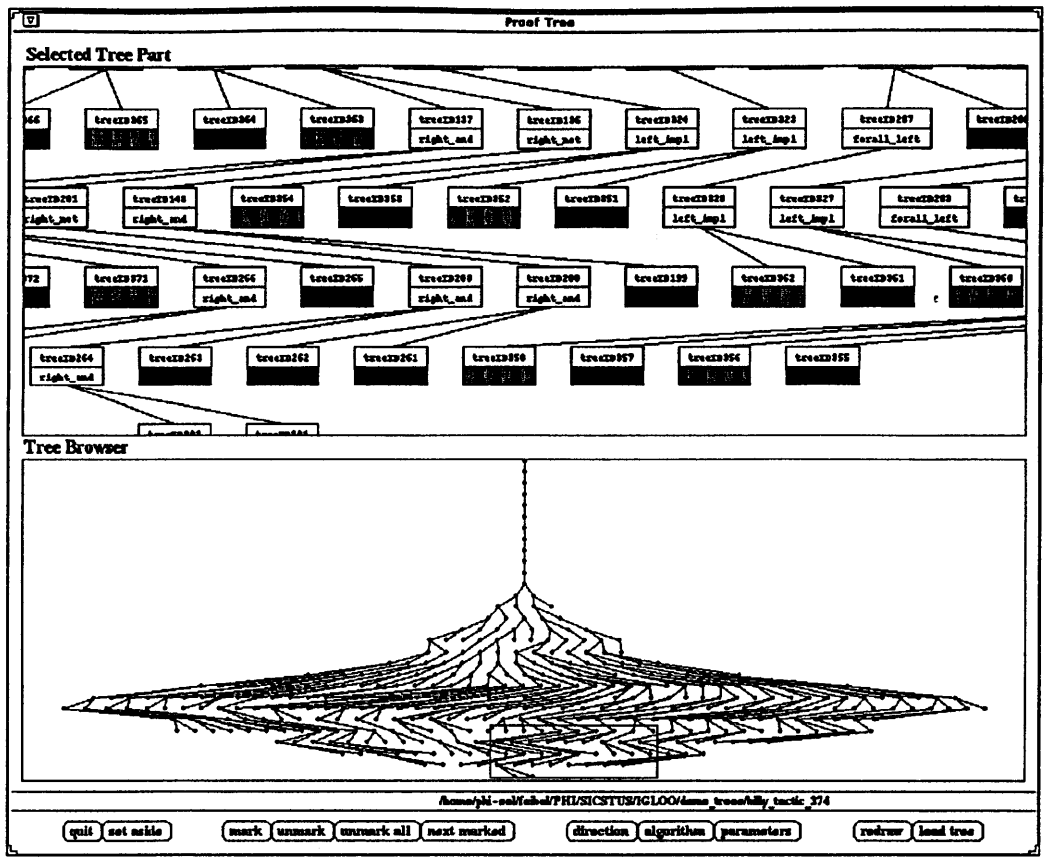


Abbildung 19: Baumdarstellung mit dem Zentrieralgorithmus

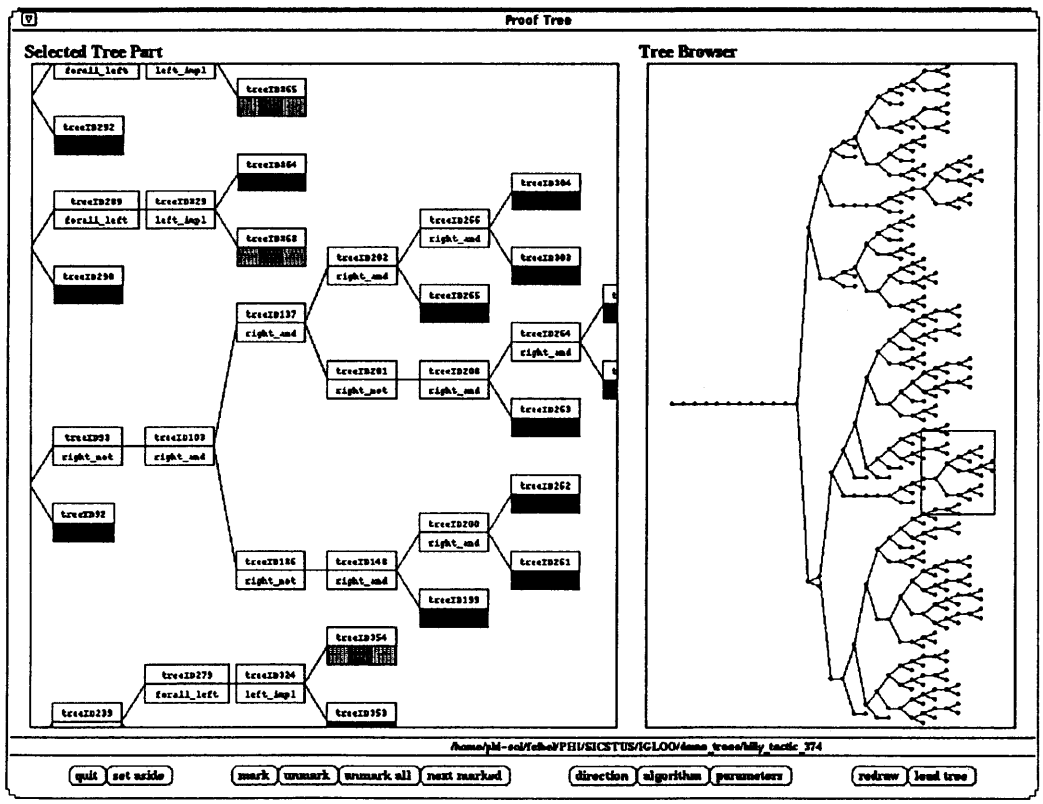


Abbildung 20: *left_right*-Baumdarstellung

3.3 Große Ableitungsbäume als Grafiken

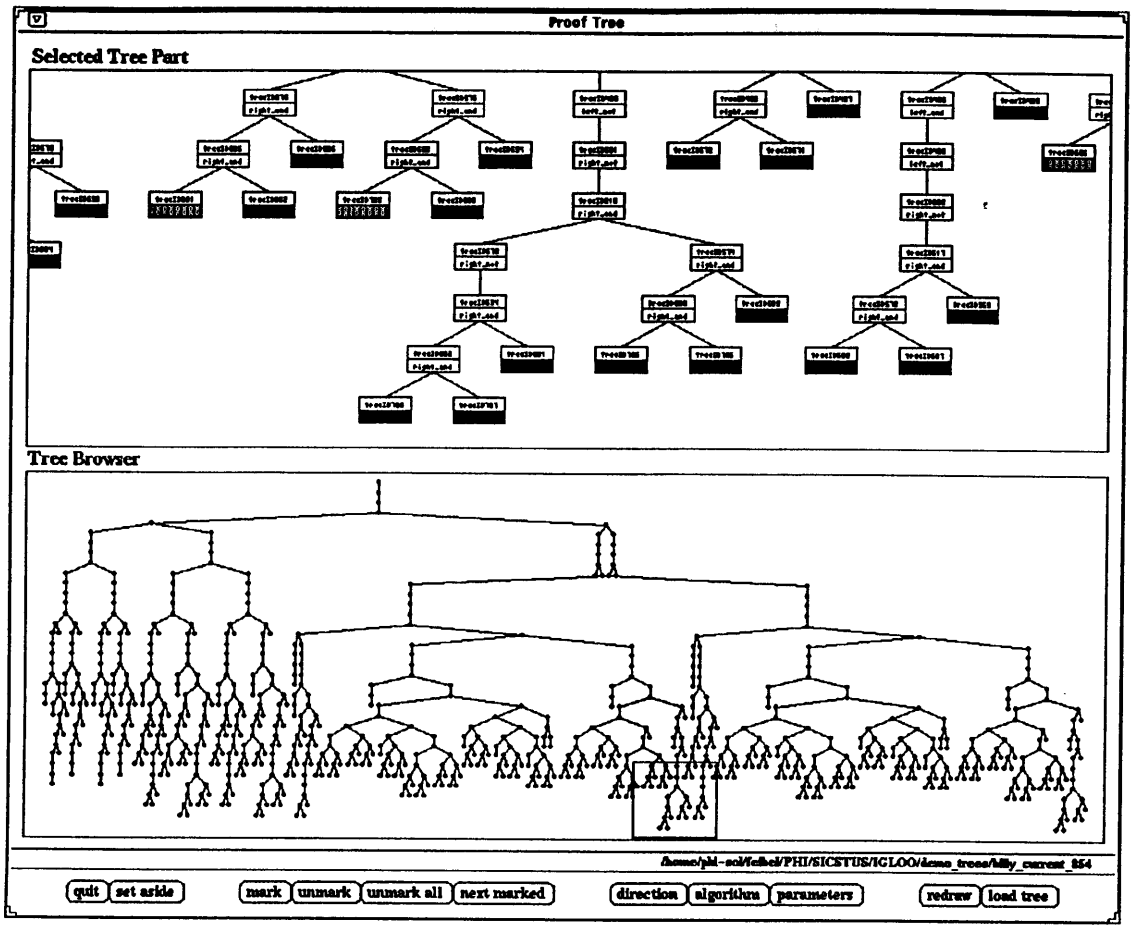


Abbildung 21: Großer Baum in strukturierter Darstellung

In diesem Abschnitt werden noch ein paar Beispiele für die grafische Darstellung von großen Bäumen mit **IGLOO**s Grafikoberfläche gegeben. Abbildung 21 zeigt einen Ableitungsbaum mit 854 Knoten. Die Bedeutung eines Browsers wird bei diesem großen Baum besonders deutlich, wenn man beachtet, welch kleine Fläche des gesamten Baumes hier durch das Rechteck des Views *Tree Browser* nur erfaßt werden kann. Ferner kann man sich hier auch leicht darüber klar werden, welchen Nutzen die Möglichkeit bringt, Knoten markieren zu können, um dann zwischen ihnen hin- und herspringen zu können. Des weiteren erkennt man im Browser auch sehr gut, wo der Baum gleiche Strukturen aufweist, was häufig ein Indiz für ähnlich geführte Teilbeweise ist. In der nächsten Abbildung 22 ist derselbe Baum mit dem *Zentrier*algorithmus dargestellt. Die Grafik im Browserview zeigt sehr deutlich, wie sich die Breite des Baumes von Level zu Level verändert. Im Ausschnittsfenster wird aber deutlich, daß eine strukturelle Beweisanalyse mit dieser Darstellung nicht möglich ist.

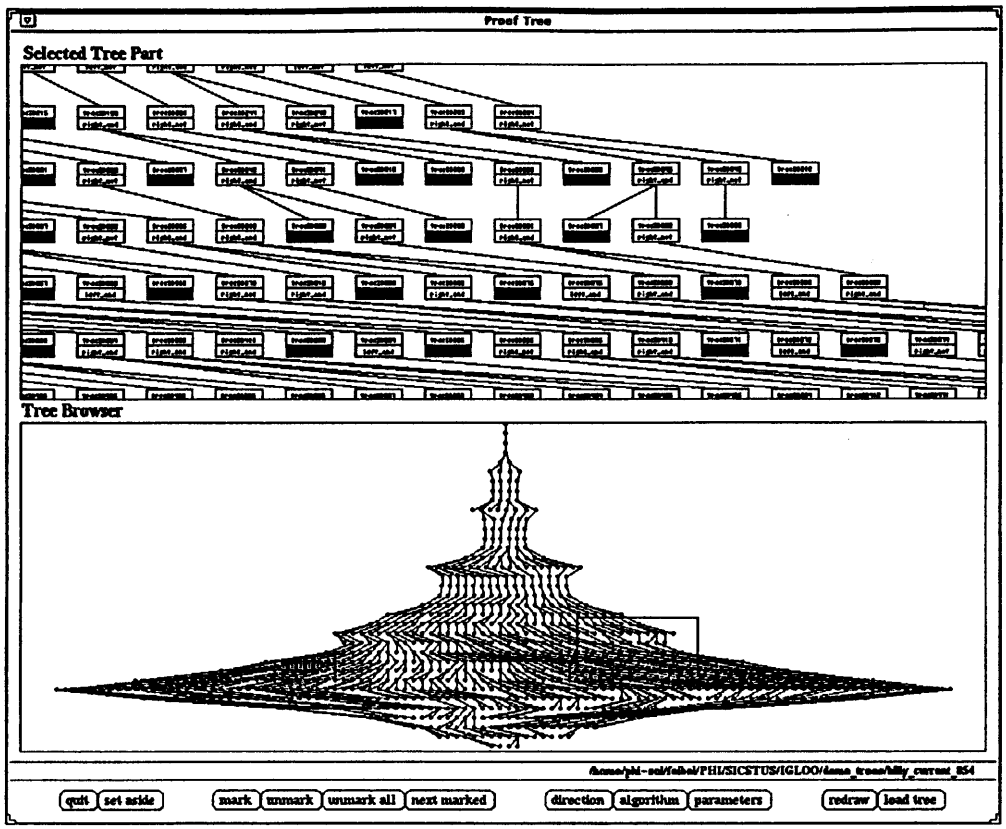


Abbildung 22: Großer Baum in zentrierter Darstellung

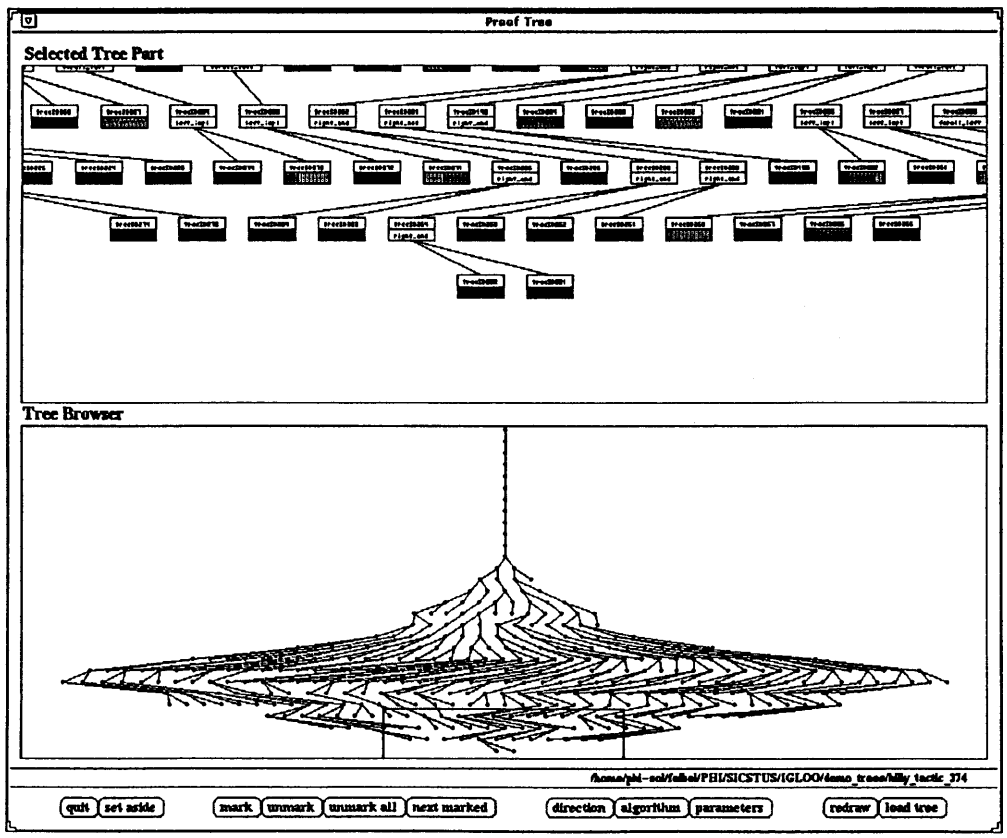


Abbildung 23: Baum eines Taktikbeweises

Abbildung 23 zeigt einen Beweisversuch desselben Startsequenten wie in den beiden vorherigen Darstellungen, allerdings ist der Beweisbaum durch die Verwendung einer Taktik um mehr als die Hälfte kleiner geworden. Man sieht auch hier ein ähnliches Verhalten bei der Entwicklung der Baumbreite von Level zu Level wie in der vorherigen Abbildung.

A Die Taktiksprache

Die Anwendung der Sequenzenregeln in einem Beweis kann durch Taktiken gesteuert werden. Eine Taktik kann wie folgt aufgebaut sein: Sie besteht aus

- einer *primitiven Taktik* oder
- ist durch eine Kombination von *Tacticals*, mit denen man Kontrollstrukturen wie *Folge*, *Verzweigung* und *Iteration* realisieren kann, definiert.

Eine primitive Taktik erzeugt einen kleinen Teil des Beweises, z.B. durch Anwendung einer Sequenzenregel auf verschiedene Blätter des aktuellen Beweisbaumes. Durch die Tacticals kann man Taktiken definieren, die große Teile eines Beweises oder sogar den ganzen Beweis erzeugen.

Die Sprachkonstrukte der primitiven Taktiken:

`apply_rule`(+*Rulename*, +*Insequents*, −*Outsequents*, −*Blockedsequents*)
Die Sequenzenregel namens *Rulename* wird auf jeden Sequenten der Liste *Insequents* angewandt. Die dadurch neu erhaltenen Sequenten werden in *Outsequents* — Sequenten zur weiteren Expansion — und *Blockedsequents* — Sequenten, deren Expansion zeitlich verschoben werden muß — partitioniert. Die Liste *Blockedsequents* ist hier wie auch in allen folgenden Beschreibungen nur für die Plankalküle der Logik LLP zu beachten. In den übrigen Kalkülen werden alle Sequenten, die aus einer Regelanwendung resultieren, in *Outsequents* gesammelt. *Blockedsequents* ist dann immer gleich der leeren Liste. Falls die Anwendung der Sequenzenregel fehlschlägt, wird der unveränderte Sequent ebenfalls in die Liste *Outsequents* aufgenommen. In den Kalkülen zu LLP ist das Wissen darüber, welche resultierenden Sequenten nicht direkt weiter expandiert werden dürfen, in der Repräsentation der Sequenzenregeln kodiert.

`apply_rule_strict`(+*Rulename*, +*Insequents*, −*Outsequents*,
−*Blockedsequents*)
Der Aufruf ist nur erfolgreich, wenn die Regel auf jeden Sequenten aus *Insequents* wirklich anwendbar ist.

`apply_rule_wo_tree`, `apply_rule_strict_wo_tree`
Die sind Versionen der obigen Taktiken, welche die für die grafische Erzeugung des Beweisbaumes notwendigen Informationen nicht generieren.

`apply_ax_schema`(+*Actionname*, +*Insequent*, −*Outsequent*)
Diese Taktik wird ausschließlich in den Kalkülen zur Logik LLP benutzt. Die Taktik ist anwendbar, wenn *Insequent* durch eine entsprechende Instanz des Axiomenschemas für die Aktion *Actionname* expandiert werden kann.

Die Tacticals zum Aufbau komplexer Taktiken:

`then`(+*List_of_tactics*, +*Insequents*, −*Outsequents*, −*Blockedsequents*)

Das then-Tactical realisiert die Kontrollstruktur der Folge zur Hintereinanderausführung von Taktiken. Das Abarbeitungsschema sieht wie folgt aus:

then([$tac_1, \ldots, tac_n$] ,In,Out,Block) $\equiv$
tac_1(In,Out_1,Block_1), ..., tac_n(Out_{n-1},Out,Block_n),
Block = $\bigcup_{i=1}^{n}$ Block_i

Es existiert auch eine schwache Version von then — then(+*List_of_tactics*) — wobei keine Argumentweitergabe unterstützt wird.
Es gibt eine Abkürzung für die Hintereinanderausführung von apply_rule*-Taktiken (s.o.):
then(+*List_of_Rulenames*, +*Insequents*, −*Outsequents*, −*Blockedsequents*)
then_strict, then_wo_tree, then_strict_wo_tree haben die entsprechende Bedeutung.

orelse(+*List_of_tactics*, +*Insequents*, −*Outsequents*, −*Blockedsequents*)
Das orelse-Tactical realisiert eine Art deterministischer Auswahl. Es werden alle möglichen Alternativen aus *List_of_tactics* der Reihe nach probiert, bis eine von ihnen anwendbar ist, dann ist auch orelse anwendbar.
orelse(+*List_of_tactics*) ist die schwache Version von orelse, welche die Weitergabe von Argumenten nicht unterstützt.
Auch hier existiert eine Abkürzung hinsichtlich der apply_rule*-Taktiken:
orelse(+*List_of_Rulenames*, +*Insequents*, −*Outsequents*, −*Blockedsequents*)
Die strict und wo_tree Versionen existieren analog.

if_tac(+*TestTactic*, +*Tactic1*, +*Tactic2*)
Mit diesem Tactical wird die Kontrollstruktur der Verzweigung realisiert. Wenn *TestTactic* zu *true* evaluiert, dann wird *Tactic1* sonst *Tactic2* ausgeführt.

map_tac(+*Tactic*, +*Insequents*)
map_tac(+*Tactic*, +*Insequents*, −*Outsequents*)
map_tac(+*Tactic*, +*Insequents*, −*Outsequents*, −*Blockedsequents*)
Tactic wird sukzessive auf jedes Element von *Insequents* angewandt. Alle Aufrufe sind voneinander unabhängig. Die jeweiligen Ergebnisse können gesammelt werden.

repeat(+*Tactic*, +*Insequents*, −*Outsequents*, −*Blockedsequents*)
Dieses Konstrukt realisiert die Kontrollstruktur der Iteration. Das Abarbeitungsschema sieht wie folgt aus:

```
repeat(Tac,In,Out,B) ≡ if_tac(then(Tac,In,Out1,B1),
                               then([map_tac(repeat(Tac),Out1,Out,B2),
                                     append(B1,B2,B)]),
                               then([Out1=Out, B1=B]))
```

Es gibt auch die Versionen repeat(+*Tactic*, +*Insequents*), und
repeat(+*Tactic*, +*Insequents*, −*Outsequents*)

Es gibt einen Definitionsmechanismus für neue Taktiken:

`define_tac`(+*NewTacticname*, +*ArgumentList*, +*Tacticbody*)
Tacticbody ist wiederum eine Taktik.
Selbst definierte Taktiken können mit dem Tactical `call_tac` aufgerufen werden:
`call_tac`(+*Tactic*, +*Insequents*)
`call_tac`(+*Tactic*, +*Insequents*, −*Outsequents*)
`call_tac`(+*Tactic*, +*Insequents*, −*Outsequents*, −*Blockedsequents*)
Der Mechanismus unterstützt rekursive Taktiken.

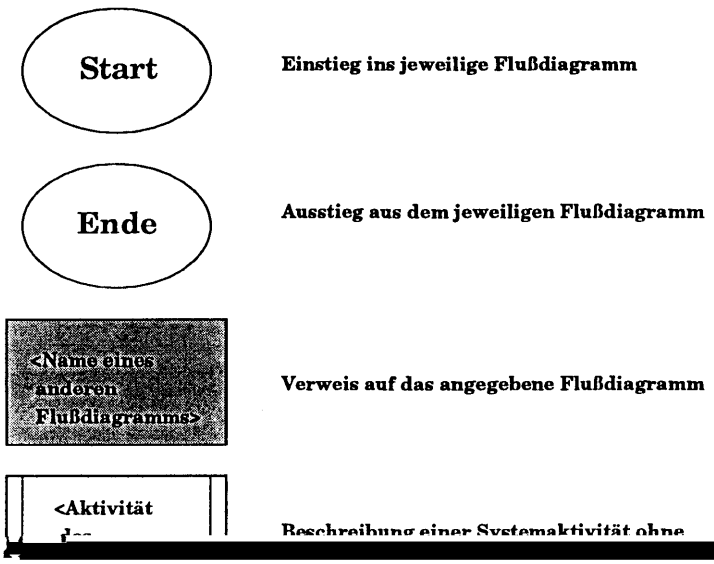
`system`(+*Object_language_predicate*)
Hiermit wird das Einbeziehen "normaler" Prologprozeduren ermöglicht. Der Aufruf ist erfolgreich, wenn *Object_language_predicate* zu *true* evaluiert.

`select`(+*Testpredicate*, +*List_of_Sequents*, −*First_Candidate*, −*Rest*)
First_Candidate ist das erste Element (von links) in *List_of_Sequents*, für welches *Testpredicate* zu *true* evaluiert. Der Aufruf liefert *fail*, wenn kein solches Element existiert.

Der Interpreter für die Taktiksprache ist durch einen Prolog Meta-Interpreter realisiert. Ein Beispiel für eine Taktik ist in Abbildung 11 auf Seite 29 gegeben. Ein mit dieser Taktik erzeugter Beweis ist in Abbildung 23 auf Seite 44 anhand des zugehörigen Ableitungsbaumes zu sehen.

B Kurzanleitungen

Zum Beweisen in den Modi *automatic proving & batch display*, *automatic proving & incremental display* und *interactive proving* sind hier Vorgangsbeschreibungen in Form von Flußdiagrammen gegeben. In Abbildung 24 sind die in den Diagrammen verwendeten Symbole erklärt.



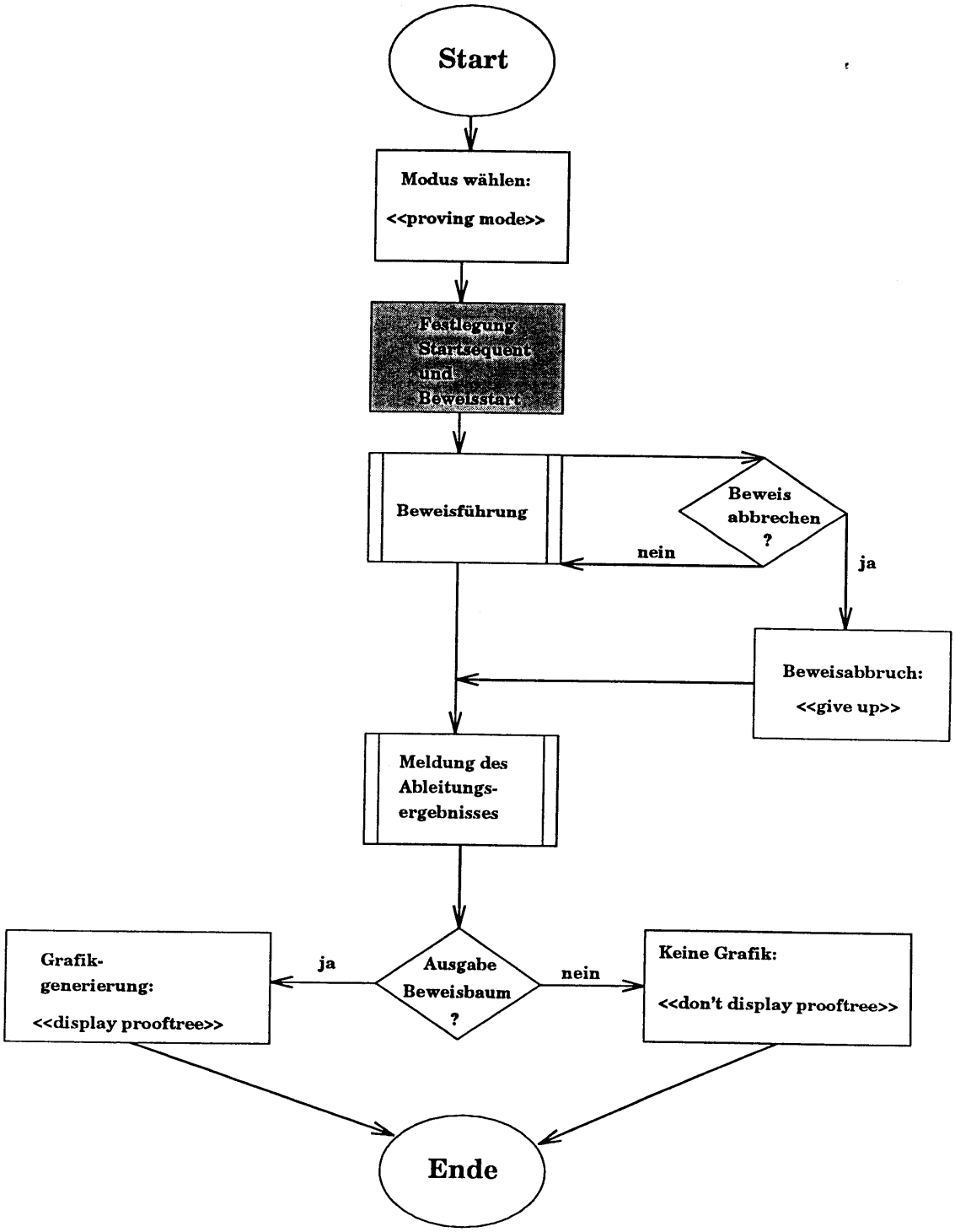


Abbildung 25: Beweisen im Modus *automatic proving & batch display*

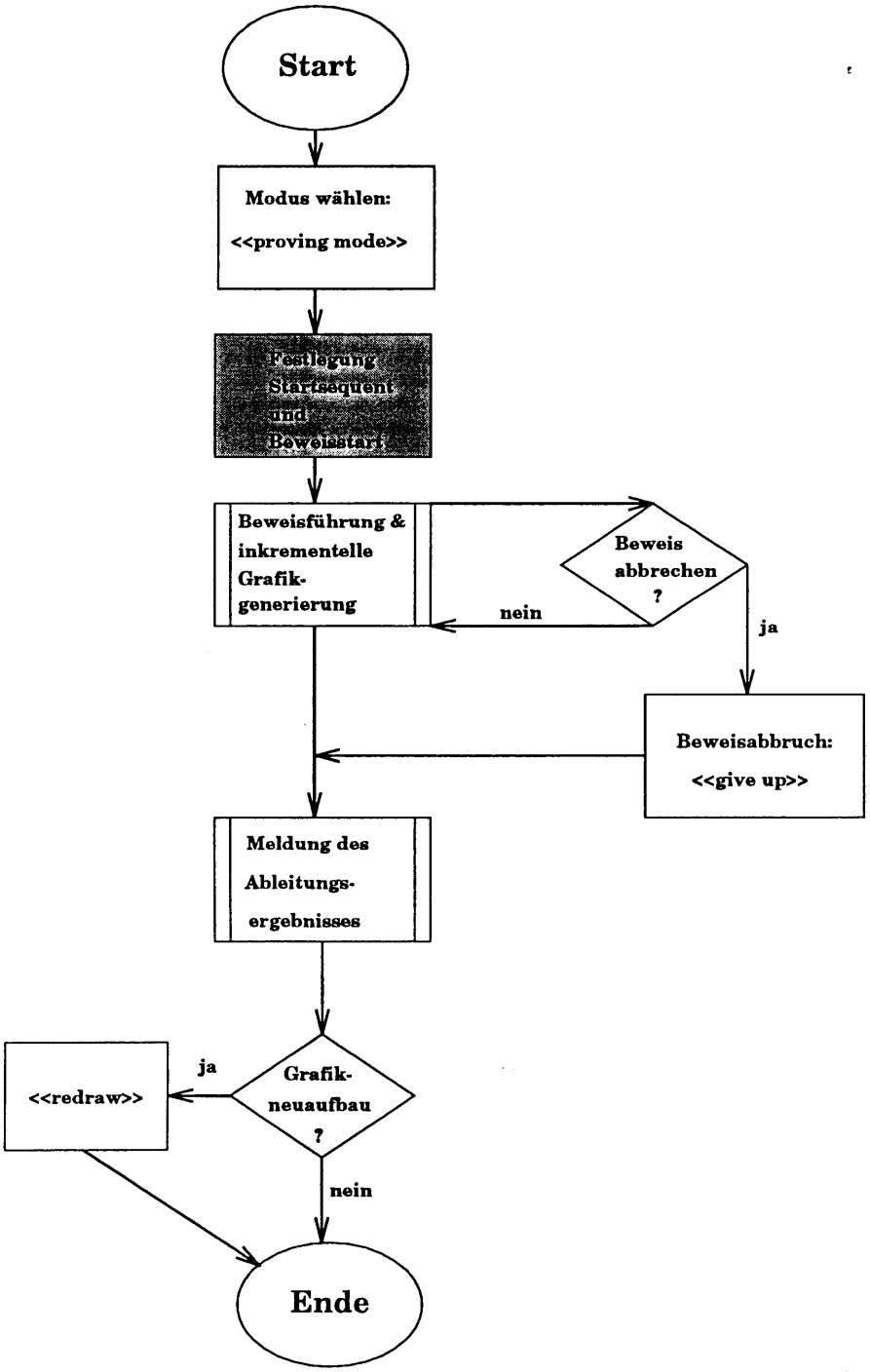
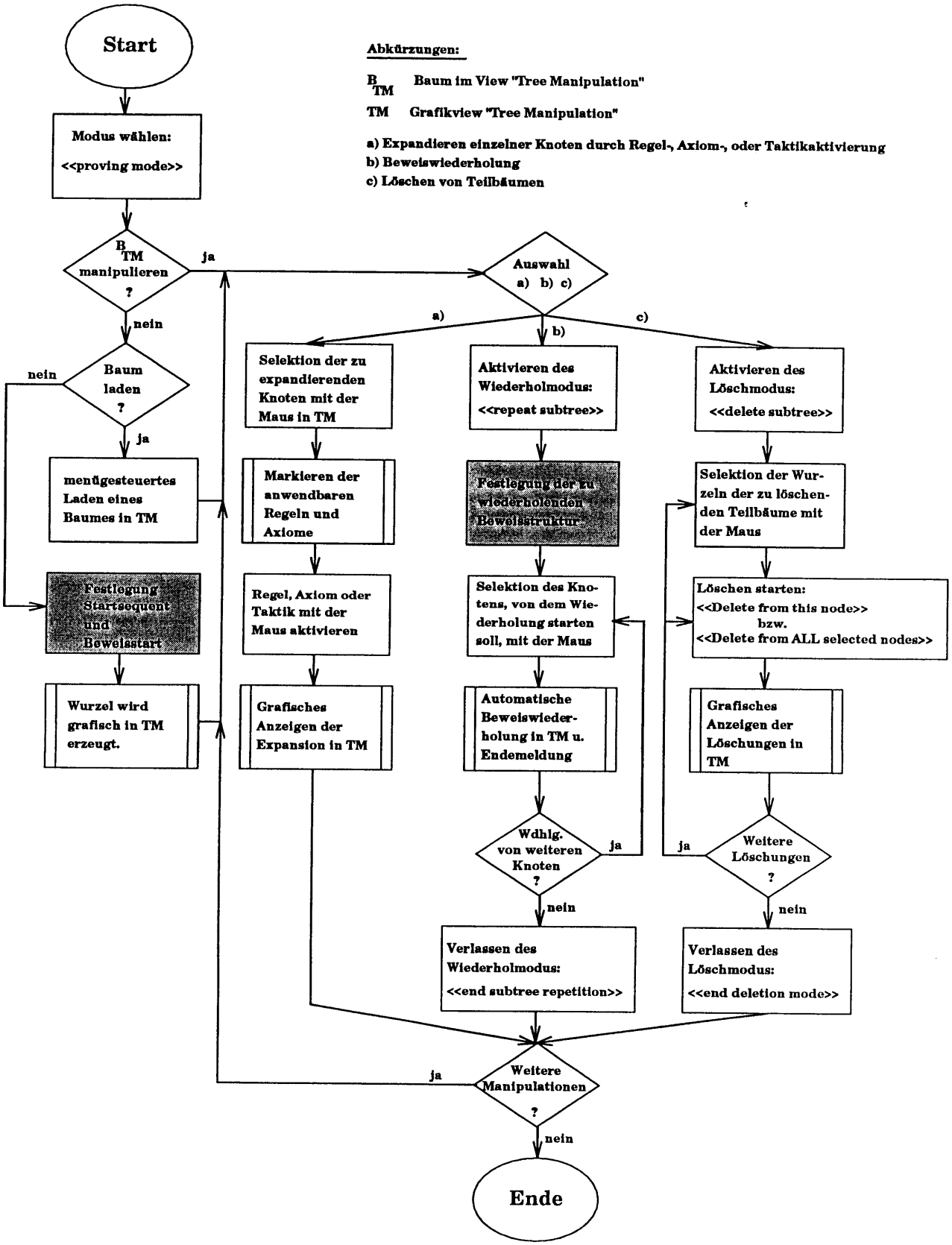


Abbildung 26: Beweisen im Modus [illegible]



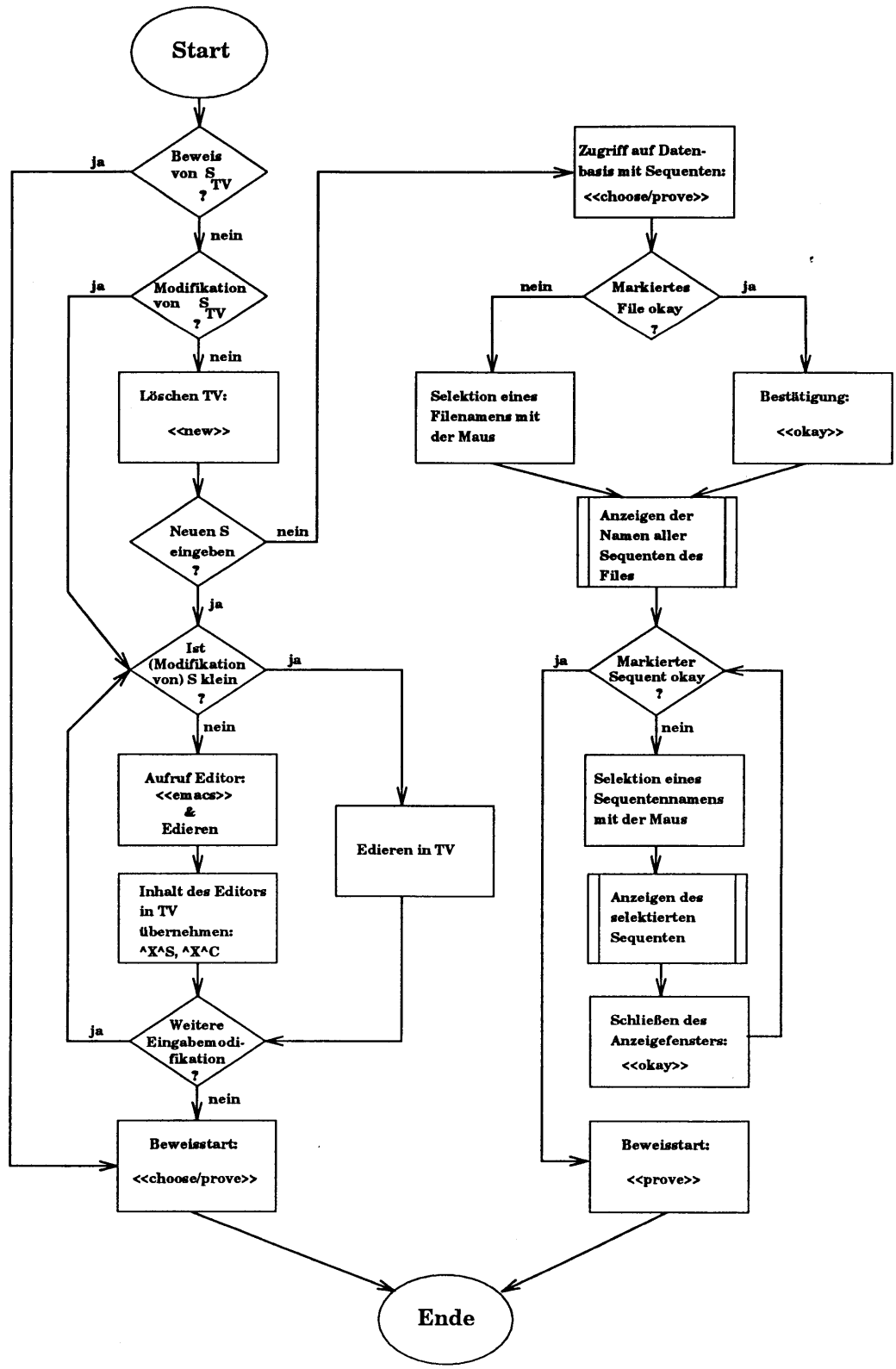
Abkürzungen:

B_{TM} Baum im View "Tree Manipulation"

TM Grafikview "Tree Manipulation"

a) Expandieren einzelner Knoten durch Regel-, Axiom-, oder Taktikaktivierung
b) Beweiswiederholung
c) Löschen von Teilbäumen

Abbildung 27. Bewe[illegible]



Abkürzungen:

S_{TV} aktuell im Textview "Sequent Input" stehender Sequent

TV Textview "Sequent Input"

S Sequent

Abbildung 28: Festlegung Startsequent und Beweisstart

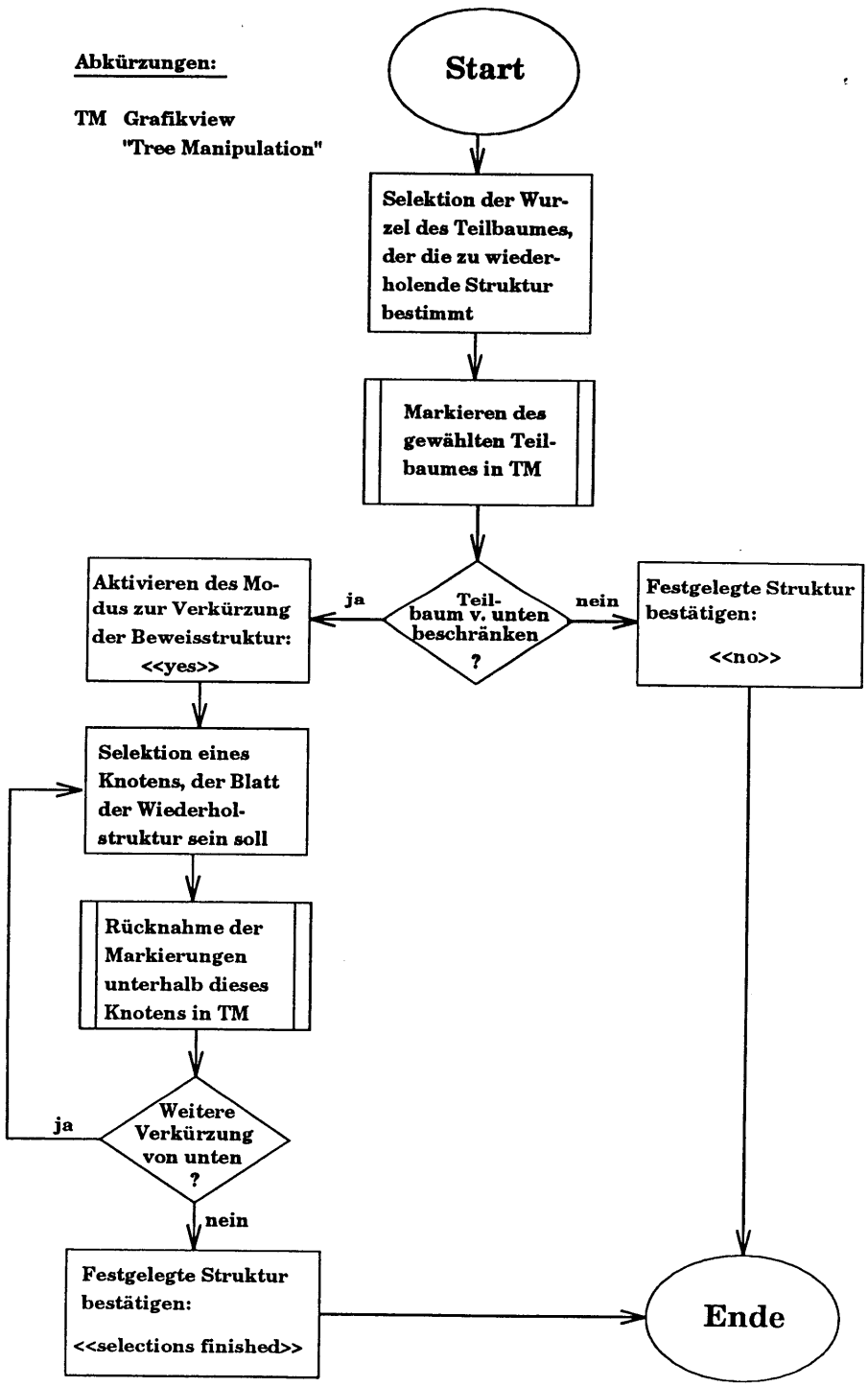


Abbildung 29: Festlegung der zu wiederholenden Beweisstruktur

Literaturverzeichnis

[BBD+92] Mathias Bauer, Susanne Biundo, Dietmar Dengler, Jana Köhler und Gaby Paul. PHI, A Logic-Based Tool for Intelligent Help Systems. Research Report RR-92-52, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, W-6600 Saarbrücken 11, Germany, December 1992.

[BD93] Susanne Biundo und Dietmar Dengler. The logical language for planning LLP. DFKI Research Report, German Research Center for Artificial Intelligence, 1993. to appear.

[ET89] Peter Eades und Roberto Tamassia. Algorithms For Drawing Graphs: An Annotated Bibliography. Technical Report CS-89-09, Department of Computer Science, University of Queensland, St.Lucia, Queensland 4067, Australia; Brown University, Providence, Rhode Island 02912-1910, USA, October 1989.

[Fit90] Melvin Fitting. *First Order Logic And Automated Theorem Proving.* Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[Gal87] Jean H. Gallier. *Logic for Computer Science.* Foundations of Automated Theorem Proving. John Wiley & Sons, Inc., 1987.

[Meh84] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching,* Seite: 189–199. Springer-Verlag, 1984.

[Mes] Frank Mestmäcker. Graphische Darstellung von Bäumen. Fortgeschrittenenpraktikum am Lehrstuhl von Professor Dr. Kurt Mehlhorn, Universität des Saarlandes, Stuhlsatzenhausweg, W-6600 Saarbrücken 11.

[RT81] Edward M. Reingold und John S. Tilford. Tidier drawing of trees. In *IEEE Transactions of Software Engineering,* Seite: 223–228, March 1981.

[Swe93a] Swedish Institute of Computer Science, Sweden. *SICStus Prolog Library Manual,* January 1993.

[Swe93b] Swedish Institute of Computer Science, Sweden. *SICStus Prolog User's Manual,* January 1993.

[Wal90] Lincoln A. Wallen. *Automated Proof Search in Non-classical Logics: Efficient Matrix Proof Methods for Modal and Intuitionistic Logics.* MIT-Press, 1990.

Register

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
67608 Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse oder per anonymem ftp von ftp.dfki.uni-kl.de (131.246.241.100) unter pub/Publications bezogen werden.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of all published papers so far are obtainable from the above address or via anonymous ftp from ftp.dfki.uni-kl.de (131.246.241.100) under pub/Publications.
The reports are distributed free of charge except if otherwise indicated.

DFKI Research Reports

RR-93-05
Franz Baader, Klaus Schulz: Combination Techniques and Decision Problems for Disunification
29 pages

RR-93-06
Hans-Jürgen Bürckert, Bernhard Hollunder, Armin Laux: On Skolemization in Constrained Logics
40 pages

RR-93-07
Hans-Jürgen Bürckert, Bernhard Hollunder, Armin Laux: Concept Logics with Function Symbols
36 pages

RR-93-08
Harold Boley, Philipp Hanschke, Knut Hinkelmann, Manfred Meyer: COLAB: A Hybrid Knowledge Representation and Compilation Laboratory
64 pages

RR-93-09
Philipp Hanschke, Jörg Würtz:
Satisfiability of the Smallest Binary Program
8 pages

RR-93-10
Martin Buchheit, Francesco M. Donini, Andrea Schaerf: Decidable Reasoning in Terminological Knowledge Representation Systems
35 pages

RR-93-11
Bernhard Nebel, Hans-Jürgen Bürckert:
Reasoning about Temporal Relations:
A Maximal Tractable Subclass of Allen's Interval Algebra
28 pages

RR-93-12
Pierre Sablayrolles: A Two-Level Semantics for French Expressions of Motion
51 pages

RR-93-13
Franz Baader, Karl Schlechta:
A Semantics for Open Normal Defaults via a Modified Preferential Approach
25 pages

RR-93-14
Joachim Niehren, Andreas Podelski, Ralf Treinen:
Equational and Membership Constraints for Infinite Trees
33 pages

RR-93-15
Frank Berger, Thomas Fehrle, Kristof Klöckner, Volker Schölles, Markus A. Thies, Wolfgang Wahlster: PLUS - Plan-based User Support
Final Project Report
33 pages

RR-93-16
Gert Smolka, Martin Henz, Jörg Würtz: Object-Oriented Concurrent Constraint Programming in Oz
17 pages

RR-93-17
Rolf Backofen:
Regular Path Expressions in Feature Logic
37 pages

RR-93-18
Klaus Schild: Terminological Cycles and the Propositional μ-Calculus
32 pages

RR-93-20
Franz Baader, Bernhard Hollunder:
Embedding Defaults into Terminological Knowledge Representation Formalisms
34 pages

RR-93-22
Manfred Meyer, Jörg Müller:
Weak Looking-Ahead and its Application in Computer-Aided Process Planning
17 pages

RR-93-23
Andreas Dengel, Ottmar Lutzy:
Comparative Study of Connectionist Simulators
20 pages

RR-93-24
Rainer Hoch, Andreas Dengel:
Document Highlighting — Message Classification in Printed Business Letters
17 pages

RR-93-25
Klaus Fischer, Norbert Kuhn: A DAI Approach to Modeling the Transportation Domain
93 pages

RR-93-26
Jörg P. Müller, Markus Pischel: The Agent Architecture InteRRaP: Concept and Application
99 pages

RR-93-27
Hans-Ulrich Krieger:
Derivation Without Lexical Rules
33 pages

RR-93-28
Hans-Ulrich Krieger, John Nerbonne, Hannes Pirker: Feature-Based Allomorphy
8 pages

RR-93-29
Armin Laux: Representing Belief in Multi-Agent Worlds viaTerminological Logics
35 pages

RR-93-30
Stephen P. Spackman, Elizabeth A. Hinkelman:
Corporate Agents
14 pages

RR-93-31
Elizabeth A. Hinkelman, Stephen P. Spackman:
Abductive Speech Act Recognition, Corporate Agents and the COSMA System
34 pages

RR-93-32
David R. Traum, Elizabeth A. Hinkelman:
Conversation Acts in Task-Oriented Spoken Dialogue
3[illegible]

RR-93-34
Wolfgang Wahlster:
Verbmobil Translation of Face-To-Face Dialogs
10 pages

RR-93-35
Harold Boley, François Bry, Ulrich Geske (Eds.):
Neuere Entwicklungen der deklarativen KI-Programmierung — *Proceedings*
150 Seiten
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

RR-93-36
Michael M. Richter, Bernd Bachmann, Ansgar Bernardi, Christoph Klauck, Ralf Legleitner, Gabriele Schmidt: Von IDA bis IMCOD: Expertensysteme im CIM-Umfeld
13 Seiten

RR-93-38
Stephan Baumann: Document Recognition of Printed Scores and Transformation into MIDI
24 pages

RR-93-40
Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, Werner Nutt, Andrea Schaerf:
Queries, Rules and Definitions as Epistemic Statements in Concept Languages
23 pages

RR-93-41
Winfried H. Graf: LAYLAB: A Constraint-Based Layout Manager for Multimedia Presentations
9 pages

RR-93-42
Hubert Comon, Ralf Treinen:
The First-Order Theory of Lexicographic Path Orderings is Undecidable
9 pages

RR-93-43
M. Bauer, G. Paul: Logic-based Plan Recognition for Intelligent Help Systems
15 pages

RR-93-44
Martin Buchheit, Manfred A. Jeusfeld, Werner Nutt, Martin Staudt: Subsumption between Queries to Object-Oriented Databases
36 pages

RR-93-45
Rainer Hoch: On Virtual Partitioning of Large Dictionaries for Contextual Post-Processing to

RR-93-48
Franz Baader, Martin Buchheit, Bernhard Hollunder:
Cardinality Restrictions on Concepts
20 pages

RR-94-01
Elisabeth André, Thomas Rist:
Multimedia Presentations:
The Support of Passive and Active Viewing
15 pages

RR-94-02
Elisabeth André, Thomas Rist:
Von Textgeneratoren zu Intellimedia-Präsentationssystemen
22 Seiten

RR-94-03
Gert Smolka:
A Calculus for Higher-Order Concurrent Constraint Programming with Deep Guards
34 pages

RR-94-05
Franz Schmalhofer,
J.Stuart Aitken, Lyle E. Bourne jr.:
Beyond the Knowledge Level: Descriptions of Rational Behavior for Sharing and Reuse
81 pages

RR-94-06
Dietmar Dengler:
An Adaptive Deductive Planning System
17 pages

RR-94-07
Harold Boley: Finite Domains and Exclusions as First-Class Citizens
25 pages

RR-94-08
Otto Kühn, Björn Höfling: Conserving Corporate Knowledge for Crankshaft Design
17 pages

RR-94-10
Knut Hinkelmann, Helge Hintze:
Computing Cost Estimates for Proof Strategies
22 pages

RR-94-11
Knut Hinkelmann: A Consequence Finding Approach for Feature Recognition in CAPP
18 pages

RR-94-12
Hubert Comon, Ralf Treinen:
Ordering Constraints on Trees
34 pages

RR-94-14
Harold Boley, Ulrich Buhrmann, Christof Kremer:
Towards a Sharable Knowledge Base on Recyclable Plastics
14 pages

DFKI Technical Memos

TM-92-04
Jürgen Müller, Jörg Müller, Markus Pischel, Ralf Scheidhauer:
On the Representation of Temporal Knowledge
61 pages

TM-92-05
Franz Schmalhofer, Christoph Globig, Jörg Thoben:
The refitting of plans by a human expert
10 pages

TM-92-06
Otto Kühn, Franz Schmalhofer: Hierarchical skeletal plan refinement: Task- and inference structures
14 pages

TM-92-08
Anne Kilger: Realization of Tree Adjoining Grammars with Unification
27 pages

TM-93-01
Otto Kühn, Andreas Birk: Reconstructive Integrated Explanation of Lathe Production Plans
20 pages

TM-93-02
Pierre Sablayrolles, Achim Schupeta:
Conlfict Resolving Negotiation for COoperative Schedule Management
21 pages

TM-93-03
Harold Boley, Ulrich Buhrmann, Christof Kremer:
Konzeption einer deklarativen Wissensbasis über recyclingrelevante Materialien
11 pages

TM-93-04
Hans-Günther Hein:
Propagation Techniques in WAM-based Architectures — The FIDO-III Approach
105 pages

TM-93-05
Michael Sintek: Indexing PROLOG Procedures into DAGs by Heuristic Classification
64 pages

TM-94-01
Rainer Bleisinger, Klaus-Peter Gores:
Text Skimming as a Part in Paper Document Understanding
14 pages

TM-94-02
Rainer Bleisinger, Berthold Kröll:
Representation of Non-Convex Time Intervals and Propagation of Non-Convex Relations
11 pages

DFKI Documents

D-93-07
Klaus-Peter Gores, Rainer Bleisinger:
Ein erwartungsgesteuerter Koordinator zur partiellen Textanalyse
53 Seiten

D-93-08
Thomas Kieninger, Rainer Hoch:
Ein Generator mit Anfragesystem für strukturierte Wörterbücher zur Unterstützung von Texterkennung und Textanalyse
125 Seiten

D-93-09
Hans-Ulrich Krieger, Ulrich Schäfer:
TDL ExtraLight User's Guide
35 pages

D-93-10
Elizabeth Hinkelman, Markus Vonerden, Christoph Jung: Natural Language Software Registry
(Second Edition)
174 pages

D-93-11
Knut Hinkelmann, Armin Laux (Eds.):
DFKI Workshop on Knowledge Representation Techniques — Proceedings
88 pages

D-93-12
Harold Boley, Klaus Elsbernd, Michael Herfert, Michael Sintek, Werner Stein:
RELFUN Guide: Programming with Relations and Functions Made Easy
86 pages

D-93-14
Manfred Meyer (Ed.): Constraint Processing – Proceedings of the International Workshop at CSAM'93, July 20-21, 1993
264 pages
Note: This document is available only for a

D-93-21
Dennis Drollinger:
Intelligentes Backtracking in Inferenzsystemen am Beispiel Terminologischer Logiken
53 Seiten

D-93-22
Andreas Abecker:
Implementierung graphischer Benutzungsoberflächen mit Tcl/Tk und Common Lisp
44 Seiten

D-93-24
Brigitte Krenn, Martin Volk:
DiTo-Datenbank: Datendokumentation zu Funktionsverbgefügen und Relativsätzen
66 Seiten

D-93-25
Hans-Jürgen Bürckert, Werner Nutt (Eds.):
Modeling Epistemic Propositions
118 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-93-26
Frank Peters: Unterstützung des Experten bei der Formalisierung von Textwissen
INFOCOM:
Eine interaktive Formalisierungskomponente
58 Seiten

D-93-27
Rolf Backofen, Hans-Ulrich Krieger, Stephen P. Spackman, Hans Uszkoreit (Eds.):
Report of theEAGLES Workshop on Implemented Formalisms at DFKI, Saarbrücken
110 pages

D-94-01
Josua Boon (Ed.):
DFKI-Publications: The First Four Years
1990 - 1993
75 pages

D-94-02

IGLOO 1.0 - Eine grafikunterstützte Beweisentwicklungsumgebung

Harald Feibel

D-94-08
Document