プログラム生成システム PAPYRUS

川田秀司[†,*] 坂井公[††] 藤田正幸[†††]
白井康之[†] 大坪透[††††,**]

Curry-Howard Isomorphism に基づくプログラム生成法では，論理式に対しプログラムの仕様としての解釈を定義し，論理式の証明から誤りのないプログラムの生成方式をあたえる．PAPYRUS (PArallel Programs sYnthesis by Reasoning Upon formal System) は，このような方法をベースとしたプログラム生成実験システムであり，大まかには以下の2つの機能を提供する．1つは，論理や表記法，プログラム生成規則の管理と，これらの定義のもとでの証明チェックやプログラム生成の機能で，型論理の1つである CC (Calculus of Constructions) における型推論機能と TRS (Term Rewriting System) の技術により実現される．これらの定義を変更することにより，プログラム生成のためのさまざまな論理や表現方法，実現可能性解釈を利用することができる．もう1つは，証明作成の簡単化を目的とする証明エディタ機能であり，(1) 適用可能な推論規則の表示と，選択された規則による自動証明展開機能，(2)部分証明が構成されるたびに行われる自動証明チェック機能(この機能により完成した証明はその時点で正しさが保証される)，(3)未証明部をプルーバに証明させる機能などがある．本稿では PAPYRUS の原理と構成，機能について述べる．

PAPYRUS: A System for Program Synthesis and Verification by Constructive Logics

Hideji Kawata,[†,*] Kô Sakai,[††] Masayuki Fujita,[†††]
Yasuyuki Shirai[†] and Tôru Ohtubo[††††,**]

PAPYRUS (PArallel Program sYnthesis by Reasoning Upon a formal System) is an experimental error-free program generation system based on the Curry-Howard isomorphism. PAPYRUS consists of two major components. One of the components has two functions: one function defines the logic, description format, and program extraction rules, and the other checks proofs and extracts programs according to the definitions. These functions are based on the TRS (Term Rewriting System) technology and the type inference of CC (Calculus ofConstructions), a type theory. By changing the definitions, various logics, representation methods, and realizability interpretations can be used for program generation. The other component is a proof editor for making proof construction as simple as possible. The proof editor has three major functions: (1) rule indication that can be applied to the present goal, rule selection, and development to subgoals based on the selected rules, (2) automatic checking of each partial proof when it is generated (proofs created with this function are correct by nature), and (3) goal proving by the prover (in addition, the assumptions and lemma to be given can be selected and parameters can be specified). This paper describes the basic principles, configurtaion, functions, and specific operation of PAPYRUS.

† (財)新世代コンピュータ技術開発機構
Institute for New Generation Computer Technology
†† 筑波大学電子・情報工学系
Institute of Information Science and Electronics, University of Tsukuba
††† (株)三菱総合研究所
Mitsubishi Research Institute, Inc.
†††† (株)ヴァンリッチ
VanRich Corporation
* 現在 (株)東芝システム・ソフトウェア生産技術研究所
Presently with Systems and Software Engineering Laboratory, Toshiba Co., Ltd.
** 現在 にっかつ芸術学院映像科

1. はじめに

近年，コンピュータの普及に伴い，膨大な量のプログラムが作成されるようになったが，同時に作成されたシステムのバグによるトラブルなどその信頼性における問題の解決が重要な課題となっている．

これに対し，基礎研究の分野では Curry-Howard Isomorphism[1] など直観主義述語論理とアルゴリズムの関係をベースにしたさまざまなプログラム生成手法が提案されている[2)~4),8)]．これらの方法には，

●仕様としての論理式の意味が厳密に定義されて

いる.

- 論理式に対する証明からプログラムを生成する厳密なアルゴリズムが与えられ，このアルゴリズムで生成されたプログラムは証明が正しければもととなる論理式の仕様を満たすことが保証されている.
- 証明の正しさはチェック可能.

といった特長があり，信頼性の向上が問題となっている分野への適用が重要なテーマと考えられる．しかしながらこれらの方法を実際のプログラミングに適用するような研究は少なく，その実用性について疑問視するものも多い.

PAPYRUS は，論理式の仕様からプログラムを生成するまでの過程に対し，計算機によりさまざまな支援を行うことでそのコストの削減を行い，上記の方法を実用的なものとすることを目標としたプログラム生成実験システムである.

証明からプログラムを生成する方法を定めるものは，そのベースとなる論理やその論理に対する実現可能性解釈であり，現在さまざまなものが提案されている．これらはそれぞれ利点を持ち，さらにこれ以後もより有効なものが現れるであろうと予測できる．よって，このようなシステムではさまざまな論理や実現可能性解釈に柔軟に対応できることが望まれる．このため PAPYRUS では，論理や実現可能性解釈をデータとして管理し，そのデータを置き換えることにより論理や実現可能性解釈の変更が簡単にできるように設計されている．このような機構を実現するためには特に証明の正しさをチェックする方式が問題となるが，PAPYRUS ではこの機構として CC[5] の型チェック機構を利用する．また，プルーバなどその効率上どうしても論理や実現可能性解釈に依存せざるをえないものについては，標準的なインタフェースを定めることで，置換えが可能であるようにした.

以下に PAPYRUS システムの主な特長を記す.

- 論理定義を CC の型データとして管理するため，このデータを変更することでプログラム生成のためのさまざまな論理を利用できる.
- CC による証明のチェックを内蔵し，部分証明が構成されるごとに自動的にチェックを行う.
- プログラム生成規則を項書き換えルールとして管理する．よって，このデータを変更することでさまざまな実現可能性解釈を使用できる.
- 項書き換え系をベースとしたマクロ定義機構により，さまざまな記述形式を使用することができる.
- プルーバを自由に呼び出すことができるため，対話的に証明を構成していくことができる.
- 未証明部へのジャンプ機能など，構造エディタによるさまざまな支援機能を備える.

2章では PAPYRUS のベースとなる理論と表現，証明作成戦略について，3章ではシステムの構成と各サブシステムの機能について証明し，4章では簡単な使用例を示す.

2. ベース理論と証明作成戦略

2.1 証明の扱い

証明からプログラムを生成する方法では，正しい証明からは必ず証明した論理式の仕様としての意味を満たすプログラムが生成されることが保証されている．これは，証明の正しさをチェックする方法があるならばその証明から生成されたプログラムのテストを行う必要がなく，品質を完全に保証できることを示している．よって，証明チェッカの存在は重要であるが，PAPYRUS システムは論理体系を自由に取り換えられることを前提としているため，ある論理体系のもとで証明を記述したとき，その論理体系を示すデータのもとでその証明の正しさをチェックする機構が必要となる．このような機構を実現する方法として型理論を利用する方法があるが[6]，この方法を直接利用することは証明の見易さを損ねるという点で現実的とはいえない．われわれは，型理論の一つである CC と自由に定義を行えるマクロ変換機構を併用することで，見やすさを損ねない記述とチェック機構を実現した．以下に CC と証明チェック方式の概要を記す.

2.1.1 Calculus of Constructions

近年さまざまな型理論が提案され，その性質や表現力などの研究が活発に行われている．CC はその中でも表現力が高く，特に項レベルと型レベル両方の変数を扱えるという特徴を持っている．以下，CC の定義を記す.

以下，x を項レベルの変数，X を型レベルの変数とする.

CC の式は，以下のように定義される.

1. 文脈 $\Gamma ::= <> \mid \Gamma, x : A \mid \Gamma, X : K$
   （<>は空列を表すとする）
2. 項 $M ::= x \mid \lambda_{x:A}.M \mid \lambda_{X:K}.M \mid M_1 M_2 \mid M * A$
3. 型 $A ::= X \mid \lambda_{x:A_1}.A_2 \mid \lambda_{X:K}.A \mid \Pi_{x:A_1}.A_2 \mid \Pi_{X:K}.A \mid A * M \mid A_1 * A_2$
4. 種 $K ::= type \mid \Pi_{x:A}.K \mid \Pi_{X:K_1}.K_2$

A_2 に x が自由に現れないとき，$\Pi_{x:A1}.A_2$ を $A_1 \rightarrow A_2$ と表現する．

以下，$K, K_1, K_2, \cdots$ は種，$A, A_1, A_2, \cdots$ は型，$M, M_1, M_2, \cdots$ は項，$B, B_1, B_2, \cdots$ は種または型，$N, N_1, N_2, \cdots$ は型または項，$\Phi_1, \Phi_2, \cdots$ は項レベルまたは型レベルの変数を表すとする．

CC では次の判定を行う．

1. $\vdash \Gamma$ *Context*
2. $\Gamma \vdash K$ *Kind*
3. $\Gamma \vdash N : B$

上の判定が行われたとき，それぞれ

1. Γ は Valid な文脈である．
2. 文脈 Γ のもとで，K は Valid な種である．
3. 文脈 Γ のもとで，$N : B$ は Valid である．

という．CC の判定は以下の推論規則によって行われる．

$$\frac{}{\vdash <> \ Context}$$

$$\frac{\Gamma \vdash K \ Kind \quad X \notin Dom(\Gamma)}{\vdash \Gamma, X : K \ Context}$$

$$\frac{\Gamma \vdash A : typex \quad \notin Dom(\Gamma)}{\vdash \Gamma, x : A \ Context}$$

$$\frac{\vdash \Gamma \ Context}{\Gamma \vdash type \ Kind}$$

$$\frac{\Gamma, \Phi : B \vdash K \ Kind}{\Gamma \vdash \Pi_{\Phi:B}.K \ Kind}$$

$$\frac{\Gamma, \Phi : B \vdash A : type}{\Gamma \vdash \Pi_{\Phi:B}.A : type}$$

$$\frac{\vdash \Gamma \ Context \quad N : B \in Dom(\Gamma)}{\Gamma \vdash N : B}$$

$$\frac{\Gamma, \Phi : B_1 \vdash N : B_2}{\Gamma \vdash \lambda_{\Phi:B_1}.N : \Pi_{\Phi:B_1}.B_2}$$

$$\frac{\Gamma \vdash N_1 : \Pi_{\Phi:B_1}.B_2 \quad \Gamma \vdash N_2 : B_1}{\Gamma \vdash N_1 * N_2 : [N_2/\Phi]B_2}$$

$$\frac{\Gamma \vdash A : K_1 \quad \Gamma \vdash K_2 \ Kind \quad K_1 =_\beta K_2}{\Gamma \vdash A : K_2}$$

$$\frac{\Gamma \vdash M : A_1 \quad \Gamma \vdash A_2 : type \quad A_1 =_\beta A_2}{\Gamma \vdash M : A_2}$$

ここで，$N \in Dom(\Gamma)$ は，列 Γ に，$N : B$ (B はある型，もしくは種) が含まれること，$N : B \in Dom(\Gamma)$ は，列 Γ に $N : B$ が含まれることを表すとする．また，$=_\beta$ は，以下を満たす等関係とする．

$$(\lambda_{N_1:B}.N_2) * N_3 =_\beta [N_3/N_1]N_2$$

さて，CC では，以下のような定理が成り立つ．

1. $\Gamma \vdash N : B_1$，$\Gamma \vdash N : B_2$ ならば，$B_1 =_\beta B_2$
2. $\Gamma \vdash N_1 : B$，$N_1 =_\beta N_2$ ならば，$\Gamma \vdash N_2 : B$
3. $\Gamma \vdash N : B$ ならば，N は強正規化可能かつ Church-Rossor 性を満たす．
4. 判定は決定可能

2.1.2 証明の表現・チェック方式

PAPYRUS における CC を使った証明表現・チェック方式の概略を示す．PAPYRUS では，CC の文脈を2つの概念に分けて使用する．1つは，証明全体に対する文脈で，証明上に現れる論理演算子，推論規則，述語定数，個体定数などを定義するために使用する．もう1つは，証明上で局所的に使用されるものを扱うための文脈で，仮定や eigenvariable などを定義するために使用する．前者を主文脈，後者を副文脈と呼ぶことにする．

●論理式の表現

論理式は，CC の項として型が *prop* であるように定める．ここで，*prop* は *prop* : *type* であるような型とする（主文脈には "prop : type" を定義する）．

例えば，$\forall x : int.x = x$ は，主文脈

$\diamond prop : type$,

$\diamond = : \Pi_{t:type}.t \rightarrow t \rightarrow prop$,

$\diamond \forall : \Pi_{t:type}.t \rightarrow (t \rightarrow prop) \rightarrow prop$,

$\diamond int : type$

のもとで，

$$\forall * int * (\lambda x : int.(= * int * x * x))$$

と表現する．

●証明の表現

CC では項 a に対しその型が B であるかどうかを判定する手続きが存在する．PAPYRUS では，この手続きを証明が正しいかどうかの判定方法として利用する．このため，$true : prop \rightarrow type$ とし，論理式 F に対し，

1. $true * F$ は，F の証明の型．
2. ある CC の項の型が $true * F$ ならばその項は F の証明．

となるように推論規則の型を定める（以下 CC の項で表現された証明を証明項と呼ぶ）．以下に推論規則の型の定め方の概略を示す．

推論規則 r に対し，その上式を，

$$\begin{array}{ccccc} [G_{11}, \cdots, G_{1m_1}] & & [G_{21}, \cdots, G_{2m_2}] & & [G_{n1}, \cdots, G_{nm_n}] \\ \vdots & , & \vdots & , \cdots, & \vdots \\ H_1 & & H_2 & & H_n \end{array}$$

とし，下式を F，この推論規則に含まれるメタ変数を $a_1, \cdots, a_k$ とする（以下 $a_1, \cdots, a_k$ をルール変数と呼ぶ．また具体的に推論が与えられたとき，ルール変数

に対応する項をルール引数と呼ぶ)．このとき各 a_i に対応する型を $A_1, \cdots, A_k$ とし，

$$T_i = (true * G_{i1}) \rightarrow (true * G_{i2}) \rightarrow \cdots \rightarrow (true * G_{im_i}) \rightarrow (true * H_i)$$

とすると，r は以下のように定義される．

$$r : \Pi_{a_1 A_1} \cdots \Pi_{a_k : A_k}. T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow (true * F)$$

例えば，述語論理の ∧ 導入

$$\frac{A \quad B}{A \wedge B}\ \text{All-Intro}$$

では，

$$all\_intro : \Pi_{x\,prop, y : prop}.(true * x) \rightarrow (true * y) \rightarrow (true * (\wedge * x * y))$$

とする（$\wedge : prop \rightarrow prop \rightarrow prop$ は主文脈に登録されているとする)．このとき，A の証明を a，B の証明を b とすると，$A \wedge B$ の証明項は，

$$all\_intro * A * B * a * b$$

となる．同様に，⇒導入，∀導入の規則は，

$$imp\_intro : (\Pi_{x : prop, y : prop}.((true * x) \rightarrow (true * y)) \rightarrow (true * (\Rightarrow * x * y)))$$

$$all\_intro : (\Pi_{t : type, p : t \rightarrow prop}.((\Pi_{x : t}.true * (p * x)) \rightarrow true * (\forall * t * (\lambda_{x : t}.p * x))))$$

となる．

さて，上記のように各推論規則に型が付けられている(主文脈に推論規則が登録されている)とする．このとき，証明の正しさのチェックとは，F を証明する論理式，s をその証明項とすると，s の型が $true * F$ になることを示すことになる．また，部分的に作成された証明のチェックとは，各推論部に対し，その推論部が，

$$\frac{\begin{matrix}[G_{11}, \cdots, G_{1m_1}] & [G_{21}, \cdots, G_{2m_2}] & [G_{n1}, \cdots, G_{nm_n}] \\ \vdots \quad , & \vdots \quad , \cdots, & \vdots \\ H_1 & H_2 & H_n\end{matrix}}{F}\ r$$

であり，そのルール引数は $t_1, \cdots, t_k$，各 t_i に対応する型は $A_1, \cdots, A_k$ とし，

$$T_i = (true * G_{i1}) \rightarrow (true * G_{i2}) \rightarrow \cdots \rightarrow (true * G_{im_i}) \rightarrow (true * H_i)$$

であるとすると，副文脈

$$p_1 : T_1, p_2 : T_2, \cdots, p_n : T_n$$

のもとで，$r * t_1 * t_2 * \cdots * t_k * p_1 * p_2 * \cdots * p_n$ の型が $ture * F$ であることを示すことになる．

2.1.3 マ ク ロ

上記の方法で証明チェックを実現するためには，例えば型付きの直観主義述語論理での $\forall x : A.x = x$ という論理式は，

$$\forall : \Pi_{X : type}.\Pi_{p : \Pi x : X.prop}.prop,$$
$$= \Pi_{X : type}.\Pi_{x : X}.\Pi_{y : X}.prop$$

という主文脈のもとで，

$$\forall * A * (\lambda_{x : A}(= * A * x * x))$$

と表現しなければならない．しかし，この表現を通用することは，読みにくいこと，表記量が膨大になることを考えても現実的とはいえない．そこでわれわれは項書き換え系を利用し，マクロ定義を自由に行えるようにすることによりこの問題を回避することとした．この例では，マクロとして，

$$\forall X : A.B \leftrightarrow \forall * A * \lambda_{X : A}.B$$
$$A = B \leftrightarrow = * @type(A) * A * B$$

($@type(A)$ は，$A : B$ となるような B を意味する関数) と定義することにより通常の表記が可能となる(**図1**参照)．

以下にマクロの使用例を記す．

1. $A * B \leftrightarrow \Pi_{x : type}.(A \rightarrow B \rightarrow x) \rightarrow x$
2. $pair \leftrightarrow \lambda_{a : type, b : type, x : a, y : b, p : type, z : (a \rightarrow b \rightarrow p)}.$ $z * x * y$
3. $\pi_1 \leftrightarrow \lambda_{a : type, b : type, x : a * b}.x * a * (\lambda_{x : a, y : b}\ x)$
4. $\pi_2 \leftrightarrow \lambda_{a : type, b : type, x : a * b}.x * b * (\lambda_{x : a, y : b}.y)$
5. $(A_1, A_2) \leftrightarrow pair * @type(A_1) * @type(A_2) * A_1 * A_2$

このとき，$a : A$，$b : B$ とすると以下が成り立つ．

1. $(a, b) : A * B$
2. $\pi_1 * A * B * (a, b) = a$
3. $\pi_2 * A * B * (a, b) = b$

さて，マクロ定義を利用する場合，一般に次のような問題点が考えられる．

- 定義の適用順序により，変換後の項が違うものになる場合がある．
- 変換操作が停止しない場合がある．

このような問題をさけ，データの一意性を保つため，マクロ定義を以下のように制限する．

マクロ定義 $M_1 \leftrightarrow C_1, \cdots, M_n \leftrightarrow C_n$ に対し，

- C_i にマクロ記述が現れるとき，そのマクロ

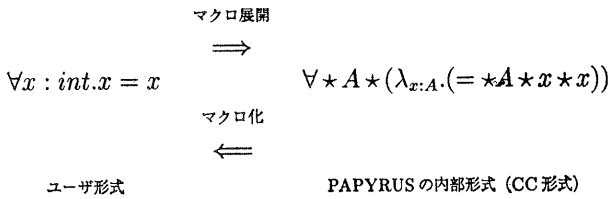


図 1 表現形式の変換
Fig. 1 Translation of expression forms.

記述は $M_j \leftrightarrow C_j (j<i)$ で定義されたものでなければならない.

- M_i に現れるマクロ変数と C_i に現れる *@type()* の引数以外のマクロ変数は一致していなければならない.
- M_i に現れるマクロ変数の集合と C_i に現れる *@type()* の引数以外のマクロ変数の集合は一致していなければならない.
- 任意の M_i $(0<i<n+1)$ に対し, M_i の変数でない部分項は, どの M_j $(0<j<n+1)$ ともユニファイ可能であってはならない.
- すべての M_i $(0<i<n+1)$ に対し, M_i は, $\lambda_A.B$, $\Pi_A.B$, $A*B$ のどれともユニァフイ可能であってはならない.

PAPYRUS では, これらの制限を定義登録時に自動的にチェックし, もしこのどれかに反するならば登録を拒否する. また, ある C_i $(0<i<n+1)$ に対し, C_i の変数でない部分項がある C_j $(0<j<n+1)$ にユニファイ可能である時にはシステムは警告をだす (登録は行われる). これによって起きる不都合はユーザの責任に委ねられる.

マクロ定義の管理など, システムに関する具体的内容は後章で述べることにする.

2.1.4 証明記述言語 PDL

PAPYRUS で使用されている証明記述言語 PDL (Proof Description Language)[7] は, Natural Deduction 方式の証明を表現することを目的として設計された, 論理と独立な型付き証明記述言語である. 多くの論理体系は,

- Sequent Culculus 形式
- Natural Deduction 形式
- Hilbert Type 形式

などに分類される証明形式を持ち, これらの証明形式はそれぞれ利点と欠点を持っている. Natural Deduction 形式の利点は, 証明の記述の流れが人間の直観に適合することであり, 人間が証明作成に介在するようなシステムにはこの証明形式が適している. よって, PDL では Natural Seduction 方式が採用されている.

PDL では証明を, 論理ブロック (FoBk), 仮定 (As), 理由ブロック (SiBk), 証明ラベル (PrLa), 証明ブロック (PrBk), 証明クラスブロック (PrClBk) という単位で扱う.

以下に PDL の構文とその意味の概略を記す.

ここで F は論理式, X は変数, T は型, L は補題名, R は推論規則, N は自然数であるとする.

- FoBk::=F|As ; F
  前者は F が証明可能, 後者は仮定 As のもとで F が証明可能であることを示す.
- As::=let X : T|assume FoBk|let X : T; As
  |assume FoBk ; As
  仮定を表す. let X : T は X の型が T であること, assume FoBk は FoBk であることを仮定することを示す.
- SiBk::=since_by assumption. end_since.|
  since by L. end_since.
  |since_by R. PrClBk end_since.
  直前の論理ブロックの理由を示す. since_by assumption. end_since. は仮定によること, since by L. end_since. は補題 L によること, since_by R. PrClBk end_since. は推論規則 R と PrClBk によることを示す.
- PrLa::=N|PrLa ; N
  証明ブロックに振られるラベル.
- PrBk::=Num. FoBk. SiBk
  論理ブロック FoBk は 理由 SiBk により証明可能であることを示す. またこのとき SiBk は FoBk の理由であるという.
- PrClBk::=PrBk|PrBk. PrClBk
  証明の集まりを示す.

定義より, PDL では論理式や推論規則に対する制約を与えないので, 明らかに論理に対し自由である.

証明図に対する PDL の対応を示すものとして**図 2** に例をあげる.

また, シンタックスシュガーとして, end_since (Num) (Num は, その理由ブロックの前の論理式ブロックの証明ラベル) という形式を許す.

さて, PDL で記述された証明を前記の CC によりチェックするためには, 各推論ステップにおいて推論

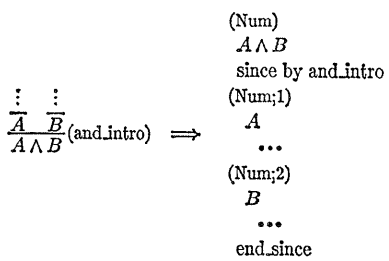


図 2 証明図と PDL
Fig. 2 Relation between proof tree and PDL.

規則 r に対するルール引数が必要となる．そこでPAPYRUS では，SiBk の記述において推論規則 r のかわりに $r*t_1*\cdots*t_m$ ($t_1,\cdots,t_m$ はルール引数とする）と記述することと定める．

2.1.5 プログラム生成方式

PAPYRUS のプログラム生成方式の概略を記す．

一般にプログラム生成規則は各推論規則に対応する関数と考えられ，上式に対応するプログラムから下式に対応するプログラムを生成する．$Ext(S)$ を証明木 S に対応するプログラムを表すとすると，例えば推論規則 and- intro に対応するプログラム生成規則は

$$Ext\left(\frac{\begin{matrix}\vdots & \vdots\\ A & B\end{matrix}}{A\wedge B}\text{and-intro}\right)=\left(Ext\left(\begin{matrix}\vdots\\ A\end{matrix}\right),\ Ext\left(\begin{matrix}\vdots\\ B\end{matrix}\right)\right)$$

となる．

さて，2.1.2 項で示した証明項について考えてみよう．今，p を論理式 F の証明に対応するる証明項，R を F を直接導いた推論規則とすると，その構成方法から以下が成り立つ．

$$p=\underbrace{R*t_1*\cdots*t_m}_{R\text{のルール引数}}*\underbrace{p_1*\cdots*p_n}_{R\text{の上式に対する証明項}}$$

例えば R が and-intro のとき証明項は以下のようになる（$proof(A)$ を A の証明項とする）．

$$proof(A\wedge B)$$
$$=and\_intro*A*B*proof(A)*proof(B)$$

このことから，証明項は，プログラム生成規則に必要な情報をすべて含んでいると考えられ，かつ各プログラム生成規則は証明項に対する単純な項書換え規則で表現できることがわかる．例えば，上記の推論規則 and-intro に対するプログラム生成規則は，

$$and\_intro*\alpha_1*\alpha_2*\alpha_3*\alpha_4\rightarrow(\alpha_3,\alpha_4)$$

という項書換え規則として表現すればよい．

PAPYRUS では，プログラム生成規則を上記のような証明項に対する項書換え規則として定義する．

2.2 証明作成戦略

証明からプログラムを作成するためには，その証明が抜けのない完全な証明であり，かつ形式的な言語により表現されている必要がある．しかし，完全な証明の作成は，対応するプログラムが簡単なものであっても，一般に証明の規模は大きくなること，同じ証明を何箇所もしなければならない状況がおきることがあること，明らかと思われる部分もすべて完全に証明しなければならないことなど，作成者に対する負担が大きい．

証明を作成するには，人が直接記述する方法のほかにプルーバを利用する方法がある．しかし，例えば直観主義述語論理などほとんどの論理は決定可能でなく，さらに帰納法の入った体系では完全性さえ保証できないことなどから，プルーバにすべてをまかせることは現実的とはいえない．

PAPYRUS では，証明作成を人とプルーバとが協調して進めることを前提とする．

プルーバの役割は，簡単なゴール，サブゴールに対する完全な証明を人の手を，煩わすことなく高速に作成することであり，人は証明の基幹となる部分の作成を担当する．つまり証明作成者は，ゴールをさまざまな方法でプルーバに扱えそうなサブゴールまで分割し，プルーバに与えることで証明を作成していくわけである．もしそのサブゴールをプルーバが解けない場合にはさらに細かく問題を分けて与え直す．

このような方針のもとで PAPYRUS では以下の機能を提供する．

- 補題を登録しておく機能を持ち，呼び出しが自由に行える．
- 指定したゴールに対し適用可能な推論規則を表示でき，適用に必要な項の入力により，そのゴールをサブゴールに展開することができる．
- 指定したゴールに対しプルーバを簡単に呼び出すことができ，さらにそのゴールでの仮定や補題からプルーバに与えるものを選択できる．
- プルーバに与えるパラメータを自由に変えることができ，さらに簡単な操作でプルーバを強制停止させることができる．

これらの機能をうまく利用し，例えば NJ で，

$$\frac{\begin{matrix}[A]\\ \vdots\\ A\quad B\end{matrix}}{B}$$

のような推論規則を他の規則とともに定義して中間ゴールを導入することで，プルーバの検索空間を小さくするという方法なども考えられる．

3. PAPYRUS システム

PAPYRUS システムは論理型オブジェクト指向言語 ESP によりインプリメントされ，次のようなモジュールにより構成される．

- 補題ライブラリモジュール
- マクロ変換モジュール

- 証明エディタ
- プログラム生成モジュール
- 証明チェッカモジュール
- 定義管理モジュール
- 証明木表示モジュール
- 証明展開モジュール

  ゴールに適用可能な推論規則を検索したり，指定された推論規則によりゴールをサブゴールに展開するモジュール．エディタにおける証明展開機能を実現する．
- プルーバモジュール
- filter モジュール

  生成されたプログラムの最適化や形式の変換などを行うモジュール．プログラム生成規則などでは表現しきれないものがあるとき使用する．
- プログラム実行モジュール

ここで，証明展開モジュール，プルーバモジュール，filter モジュール，プログラム実行モジュールは，論理や実現可能性解釈に依存するものとしている．これは，現在の技術では効率などの面で汎用化することのメリットよりデメリットの方が大きいと判断したためである．これらに対しては，汎用的なインタフェースが定められており，このインタフェースを満たすモジュールを作成しシステムのパラメータを変更することにより，既存のモジュールと置換えが可能である．

PAPYRUS のさまざまな支援機能は，上記のモジュールを有機的に結合することにより実現されている．

3.1 データ管理機構

PAPYRUS におけるデータは，補題ライブラリモジュールと定義管理モジュールによって管理されている．補題ライブラリモジュールは補題，定義管理モジュールは推論規則・定数データ，マクロデータ，プログラム生成規則を管理する．

3.1.1 データの種類

各データの意味と形式について述べる．

1. 補題データ

   それ以前に作成された補題のデータであり，論理式ブロックとその証明，その証明から作成されたプログラム，論理体系名などからなる．
2. 推論規則・定数データ

   CC の形式で表現された各推論規則の定義や定数の定義からなる．
3. マクロデータ

   マクロ変換の定義データ．項書き換えルールの形式で表現される．
4. プログラム生成規則

   証明項からプログラムへの変換規則や項書き換えルールの形式で表現される．

3.1.2 補題ライブラリモジュール

一般に証明を行う場合，過去に証明した定理を利用したり，その証明を行うためにあらかじめいくつかの補題を証明しておきそれを利用して証明を完成させるといった手順を踏む場合が多い．特にその証明が巨大なものである場合，このような手法は必須である．このような手法を実際に行うなめの機構を提供するものが補題ライブラリである（図 3）．補題ライブラリでは，会話的な補題登録機能と，さまざまなキーによる検索機能を用意し，証明エディタにおける補題プールとの連動により証明手続き時間の短縮を実現する．また，CC の機能を利用することで，汎用性の優れたスキーマやメタな定理を矛盾なく使用できる環境を提供している．補題ライブラリでは，基本的な補題を提供するシステムライブラリと，各ユーザが管理するライブラリが提供される．一般のユーザは自分のライブラリに必要な補題を登録しなければならないが，他のライブラリの補題を利用したり自分のライブラリに複写したりすることは許される．

3.1.3 定義管理モジュール

PAPYRUS では，推論規則・定数データ，マクロデータ，プログラム生成規則を定義管理モジュールにより管理する．各データはそれぞれクラス，パッケー

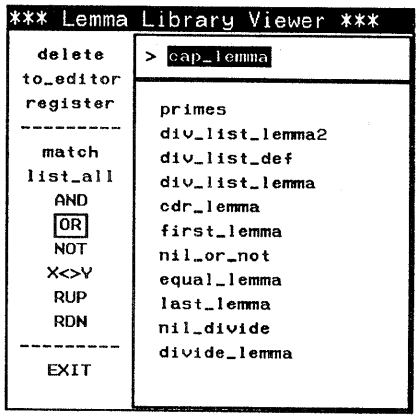


図 3 補題ライブラリ
Fig. 3 Lemma library.

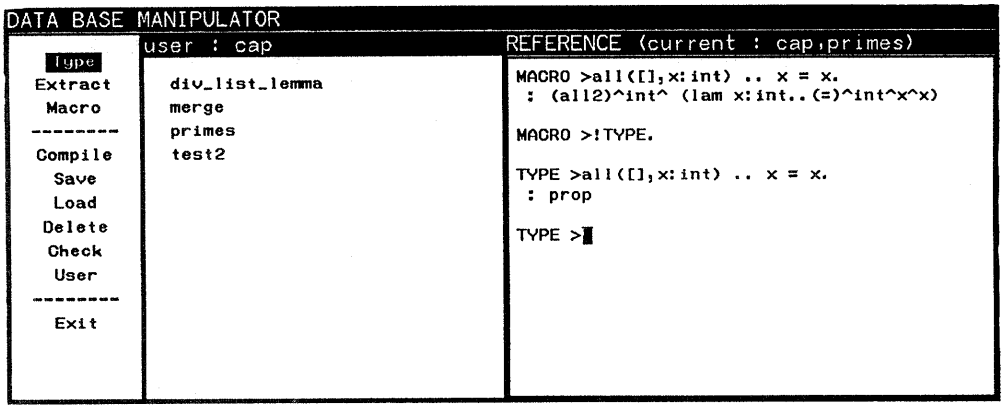


図 4 定義ウインドウ
Fig. 4 Definition window.

ジという2つの階層によって管理される．クラスとは幾つかのデータをまとめたもの（1つのデータでもかまわない）で，PAPYRUS での最小のデータ単位である．パッケージとはクラスの集まりで，例えばそれらのクラスの管理者ごとに作成することで管理が容易になる．各クラスにはクラス名，パッケージにはユーザ名と呼ばれるラベルをつける．パッケージとユーザ名は1対1に対応しなければならないが，クラスに関してはその所属するパッケージが異なる場合には同一のクラス名を使用してもかまわない．

システムはあらかじめ，“system” というユーザ名のパッケージを用意するが，このパッケージは，各ユーザが共通して使用するクラスを登録するためのものである．

データのメンテナンスは，定義管理モジュールが提供するユーザインタフェースである定義ウインドウ（図4参照）によって行う．

定義ウィンドウで利用できる主な機能を以下に記す．

- 新たなパッケージの作成．
- 新たなクラスの作成（定義を記述したファイルからクラスを作成する）．
- クラスに対するデータの追加や削除．
- 各パッケージに属するクラスの表示．
- クラスが含む定義の表示．
- クラスの削除やコピー．

さらに，

- マクロ定義のクラスを指定すれば，入力した項のマクロ展開やマクロ化を行った結果を表示できる．
- 各推論規則・定数データのクラスを指定すれば，入力した項の型推論結果や正規形などを表示できる（マクロ定義のクラスを指定しておけば，入力する項にマクロを使用することもできる）．
- プログラム生成規則を指定すれば，入力した証明項に対応するプログラムを表示できる．

また，クラスを定義する際，継承機能を利用することができる．継承機能とは，既存のクラスの定義を包含するクラスを作るときに利用するものであり，定義を記述したファイルをクラスに変換する際に必要なクラスを指定することにより，指定したクラスの定義とそのファイルに書かれた定義を合わせたクラスを作成することができる．あるクラスを変更すると，そのクラスを継承しているクラスも自動的に変更される．この際，データの整合性はユーザが保証しなければならない．

3.2 証明エディタと証明支援機構

証明エディタは，PAPYRUS のユーザインタフェースの中心に位置するものであり，補題ライブラリなどのモジュールの呼び出し機能，証明の作成や編集を行う機能を提供する．特に証明の作成，編集では，各モジュールの機能を利用し，さまざまな支援機能を実現している．

PAPYRUS では，使用する項や推論規則の定義，証明記述方式が明確であり，さらに推論自体が厳密な定義によるものであるため，システムによるさまざまな支援を行えるというメリットがある．

ここでは，証明エディタのデータ管理方法と証明支援機構の概略を述べることにする．

3.2.1 証明エディタにおけるデータ管理機構

証明エディタでは各種のデータをバッファ単位で管

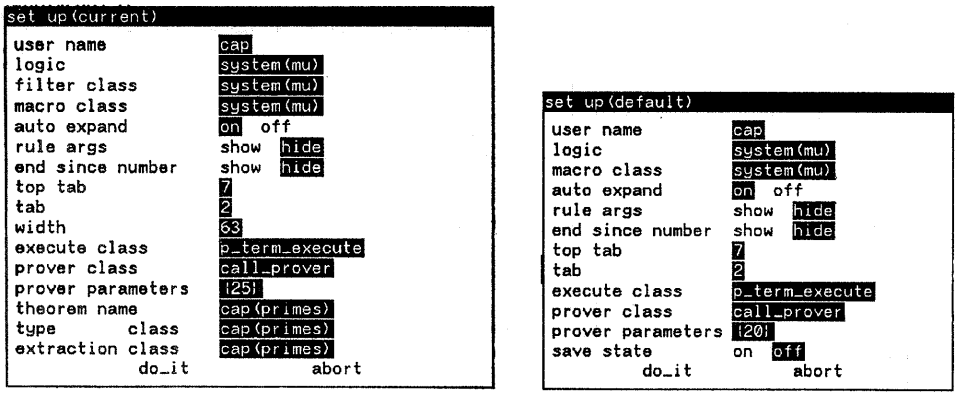


図 5 Set UP ウインドウ
Fig. 5 Set up windows.

理している．各バッファは，

1. ユーザ名（使用者を表す文字列）
2. 定理名（定理を表す文字列）
3. 論理体系名（論理体系を表す文字列）
4. filter クラス（filter モジュールのクラス名．filter モジュールを使用しない時は nilを 設定しておく）
5. execute クラス（プログラム実行モジュールのクラス名）
6. 推論規則・定数データ（3.1 節参照．未定義の定数が使用された場合には，システムはその定義をユーザに質問し，定義の正当性チェックの後，追加をする）
7. マクロクラス（3.1 節を参照）
8. 展開規則クラス（証明展開モジュールのクラス名）
9. プログラム生成規則（3.1 節を参照）
10. 変数データ（証明木における，変数名とその導入位置などのデータ）
11. プルーバクラス（プルーバモジュールのクラス名）
12. 利用する補題データ（証明展開，自動証明で利用するための補題データ．補題プールで管理される．証明で利用が予想される補題を補題ライブラリから登録しておく）
13. 証明（証明図本体．証明図におけるブロック単位に分けて管理される）
14. モード（現在の編集モード，表示モードと展開モード）

を持ち，複数の証明を同時に編集することができる．各バッファの管理するデータは Set Up ウィンドウ（図5参照）により変更することができる．このデータのうち，ユーザ名，論理体系名，利用するプルーバ名，execute クラス，表示モード，展開モードに関しては，デフォルト値を登録しておくことができる（図5参照）．また，推論規則，展開規則，プログラム生成規則は，論理体系名に対し，自動的に設定される（変更も可能）．

3.2.2 証明支援機能

PAPYRUS には大きくわけて2つの証明がある．編集モード1つは構造モードと呼ばれ，すべての情報を証明の構造レベルでシステムが管理する．ユーザは，構造単位で証明の更新，追加を行う．このモードが PAPYRUS の基本モードであり，システムはすべての情報を管理できるため，さまざまな支援を行うことができる．もう一つは，テキストモードと呼ばれ，ある構造ブロックを完全にユーザに開放する．ユーザはその構造を通常のエディタを使う要領で変更し，変更後システムによる整合性のチェックを経て構造モードに復帰するという手順をふむ．開放する構造ブロックの情報の管理は不可能であるため支援を行うことができない．

- テキストモード

テキストモードは，ある理由ブロックの中に構造モードの提供する編集機能では処理できない状況が発生したとき，ユーザがそのブロックを指定しモード変更を要求することにより移行する．この際，そのブロック以外の部分はマスクされユーザはそのブロック内のみ，通常の EMACS エディタと同様の編集を行うことができる．編集の終了後，ユーザはシステムに構造モードへの移行の要求を行う．この際システムはそのブロックに対

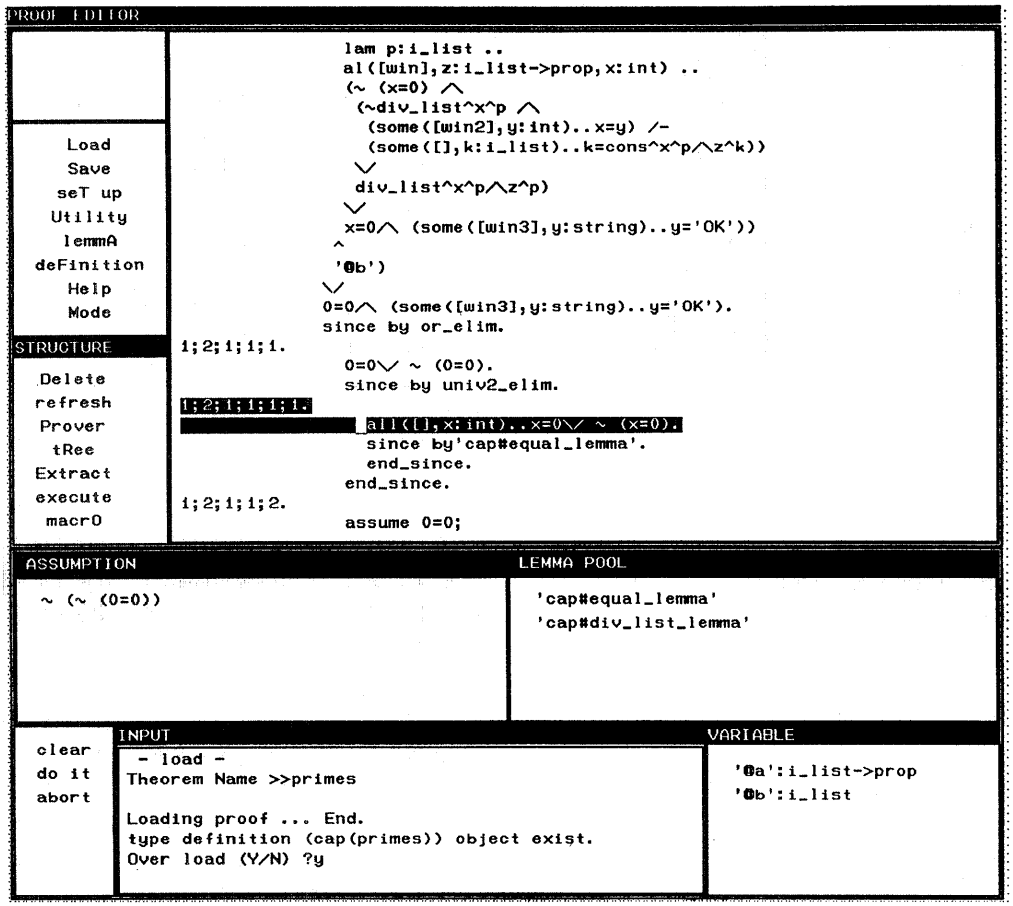


図 6 領域を指定したところ
Fig. 6 Selecting a region of a proof.

し，項 PD Lとしての構文チェックを行い，その後 CC による証明木としての意味レベルチェックを行う．これらのチェックが成功しなければ構造モードには移行できない．

- 構造モード

PAPYRUS の標準編集モードである．このモードでは証明の構造を保持する．編集モードで提供する機能を以下に記す．

(a) プリティプリント機能
PDL による証明図をバッファに登録された表示モードに従い，清書を行う機能．PDL の構文チェックを通るものであればテキストモードでも利用できる．

(b) ブロック指定機能
PDL での論理式ブロック，証明ブロックをマウスにより指定する機能．指定されたブロックは反転表示に変わる（**図 6** 参照）．まずマウスの右ボタンをクリックすることで，マウスのさしているポイントを含む最小のブロックが指定される．以後左ボタンのダブルクリックにより，そのブロックを含む最小のブロックに指定されたブロックを広げる．また，中ボタンのダブルクリックにより，それ以前の左ボタンのダブルクリックにより広げられた範囲をもとの範囲に戻すことができる．

(c) 削除機能
(b)によりあるブロックが指定されているとき，そのブロックを含む最小の証明ブロックのうちの最大の理由ブロックを削除する（このときユーザに確認をとる）．

(d) ジャンプ

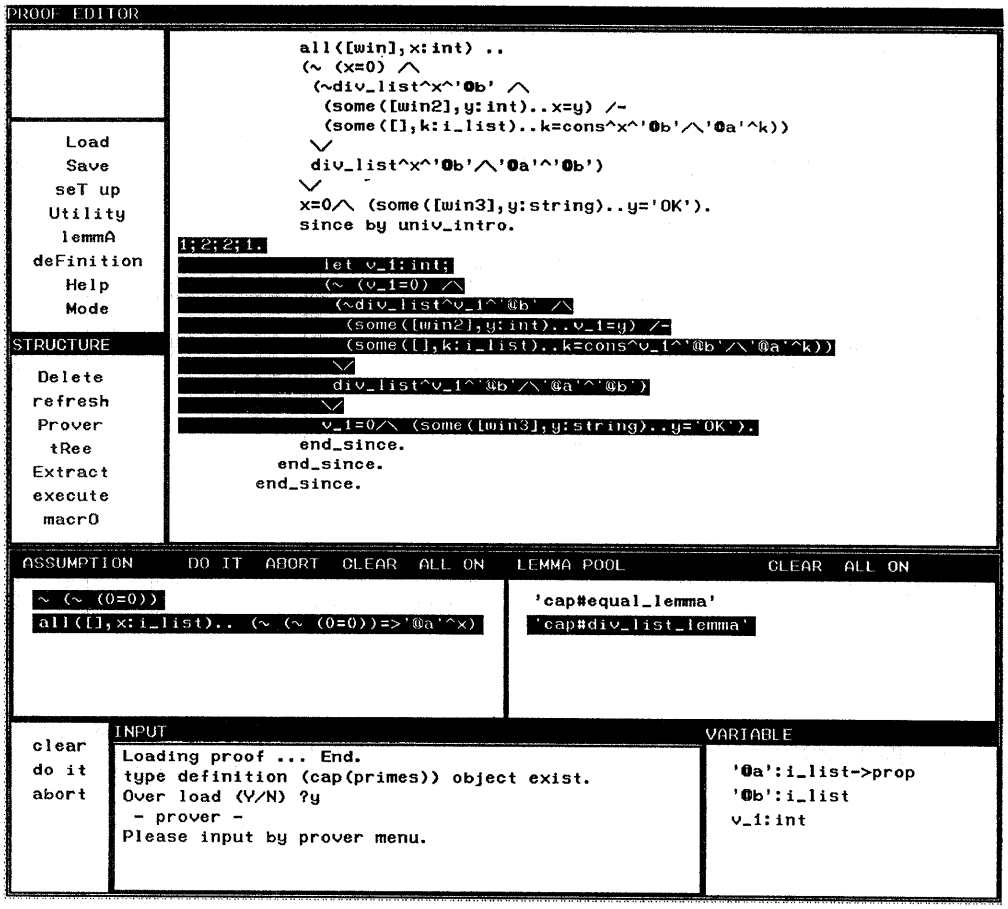


図 7 プルーバの起動と仮定，定理の指定
Fig. 7 Running a prover and choosing assumptions and lemmas.

証明図の構造をもとに，カーソルを移動する機能であり，以下の種類がある．

◇現在のカーソルより下に位置する未証明の論理式ブロックに移動する機能．

◇現在のカーソルより上に位置する未証明の論理式ブロックに移動する機能．

◇最上部に位置する未証明の論理式ブロックに移動する機能．

◇現在のカーソルのあるブロックの内側の証明クラスブロックの最上部に位置する論理式ブロックに移動する機能．

◇現在のカーソルのあるブロックと同じレベルの論理式ブロックで，カーソルより下に位置するブロックに移動する機能．

◇現在のカーソルのあるブロックと同じレベルの論理式ブロックで，カーソルより上に位置するブロックに移動する機能．

◇現在のカーソルのあるブロックの一つ外側の論理式ブロックに移動する機能．

(e) 自動証明

PAPYRUS には，プルーバとのインタフェースが用意されている．このインタフェースの仕様を満たすプルーバが用意され，かつそのプルーバクラスがバッファに登録されているとき，自動証明を行う論理式ブロックを指定することで，自動証明機能を利用することができる．自動証明が成功すると，成功がユーザに通知され証明が自動的にテキスト上に展開される（図7参照）．

そのバッファ上の論理とプルーバの整合性はユーザが保証しなければならない（作成された証明のチェックは自動的に行われるため，

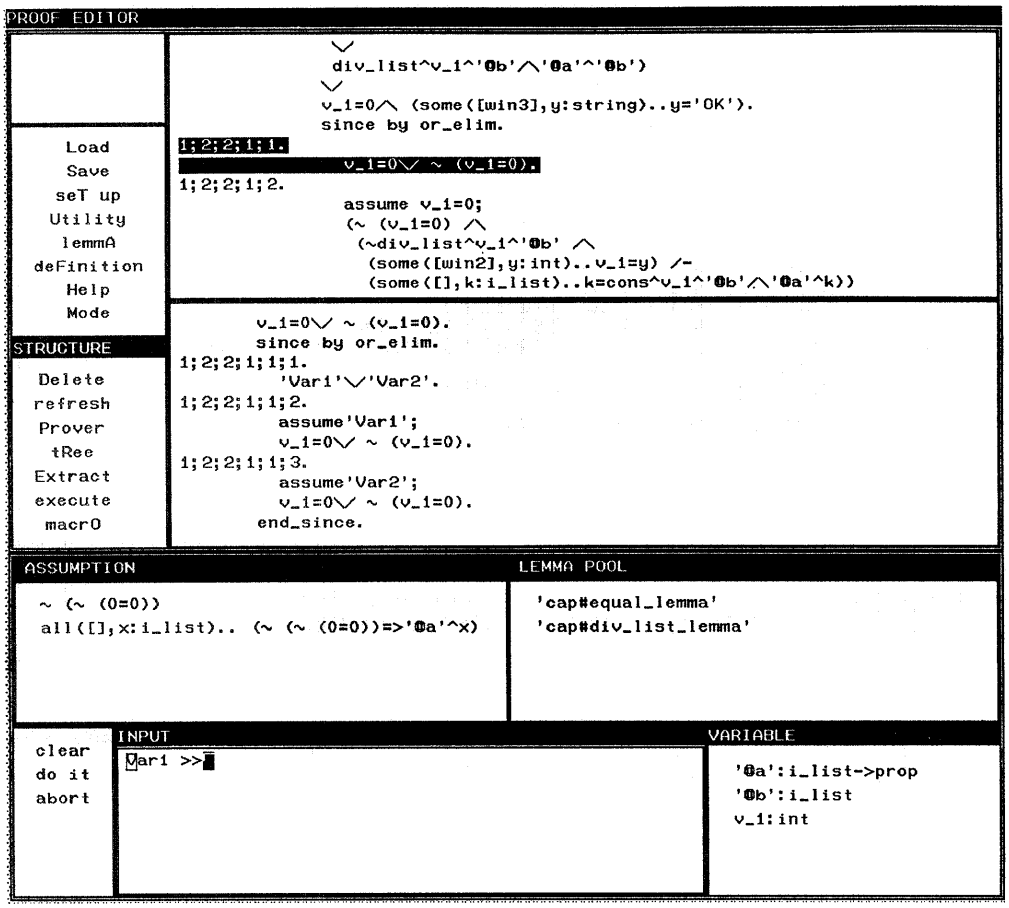


図 8 証明展開
Fig. 8 Automatid refinement of a proof.

プルーバが誤った証明図を返した場合にはテキスト上に展開されない).

(f) 証明展開

PAPYRUS では，登録された証明展開クラスにより証明展開を行う機能を提供する(図8参照)．証明展開は以下のような手順で行う．

i．展開する論理式ブロックにマウスマーカを移動し左ダブルクリックを行う．

ii．その論理式ブロックが未証明でないとき，その理由ブロックを消去してよいかをユーザに尋ねる．ユーザが消去を拒否するならば終了する．

iii．その論理式ブロックが未証明か，未証明でなくかつユーザが消去を拒否しなかったとき，システムは適用可能な推論規則，仮定，補題（補題プールに登録されているもの）を調べる．

iv．自動展開モードが ON で，仮定，もしくは補題 L のどちらかが適用可能であるなら，それぞれ，Since by assumption, Since by 補題 L に展開し終了する．それ以外のとき，適用できる推論規則，仮定，補題をメニューとして表示する．このとき，ユーザが推論規則が表示されたメニューのある規則を左ダブルクリックすると，その規則の適用によりそのブロックがどのように展開されるかを表示されることができる（右クリックによりもとの表示にもどる)．

v．推論規則の表示されたメニューのある規則を左クリックすると展開に必要な情報を

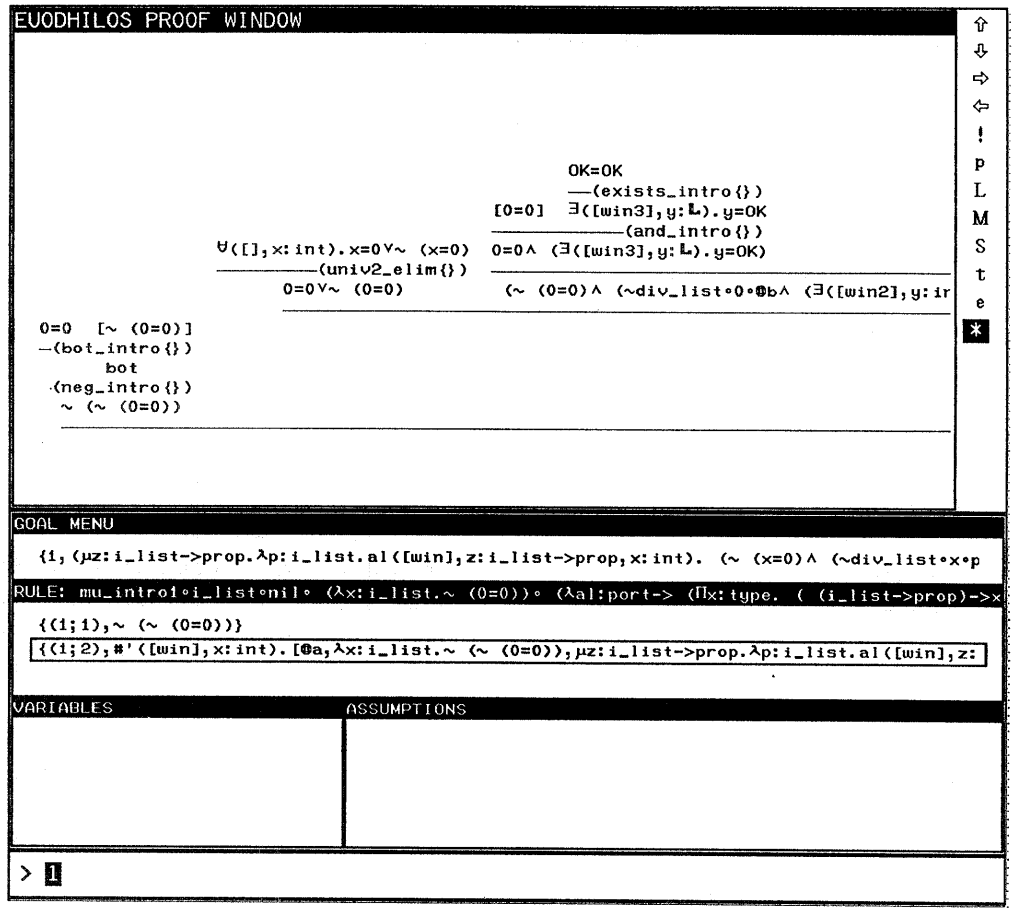


図 9 証明木表示ウインドウ
Fig. 9 Tree window.

ユーザに尋ねた後そのブロックを展開する．このとき，自動展開モードが ON であるならば展開によって作られる新たなブロックに対し，仮定，補題で適用可能かどうかを調べ，適用可能なものがあれば，自動的に展開を行う．

(g) 証明木の表示 (EUODHILOS ウンィドウ)
証明図の木構造を専用ウィンドウに構造図として表示する機能 (**図 9** 参照)．表示する木構造は全体，部分のどちらかから選択できる．また，表示された構造図上の各論理式ブロックを表すノードをマウスクリックすることで，エディタが表示している証明図をそのブロックの場所へスクロールさせると同時に，その理由ブロックの持つ規則，サブゴールなどの情報をウィンドウに表示する．

(h) プログラム生成
バッファに登録されたプログラム生成規則に従いプログラムを生成する機能 (filter クラスが登録されているときにはさらにそのクラスによりプログラムの変換を行う)．

(i) 実行
バッファに登録された execute クラスに，生成されたプログラムの実行を行わせる機能．

4. 例 題

以下の例題をもとに，PAPYRUS システムの操作例を示す．

- 直観主義インプリケーション・ロジックで論理式 a=>(a=>b)=>b を証明し，プログラムを取り出す．

1. PAPYRUS で直観主義インプリケーション・ロジックを利用するための準備を行う．

(a) データ・ベース作成

直観主義インプリケーション・ロジックのデータ・ベースを作成する．データ・ベース作成は次のように行う．まず，定義ファイルを作成する．ファイル内容は以下のとおり．次に，定義ウィンドウ(3.1.3 項)により各ファイルをコンパイルする．

- マクロデータ

A=>B<-(=>)^A^B.

- 推論規則・定数データ

```
prop : type.
(=>) : prop->prop->prop.
true : prop->type.
imp_intro : (pi p : prop.. pi q : prop..
                (true^p->true^q)->true^(p=>q)).
imp_elim : (pi p : prop.. pi q : prop..
    true^(p=>q)->true^p->true^q).
```

- プログラム生成規則

```
imp_intro^_^_^(lam C : D.. E)<-lam(C, E).
imp_elim^_^_^C^D<-app(D, C).
```

(b) 証明展開モジュール作成

証明展開モジュールーとして，直観主義インプリケーション・ロジックを定められたインタフェースに従って ESP クラスの形式で定義する（**図 10** 参照）．また，バッファの管理する展開規則クラスにそのクラス名を設定する．この設定により，展開時には規則として定義したクラスが呼び出される．

```
remove_operator(('=>')).
add_operator(('=>'),xfy,700).
use_package(system_pr_ref).

class implication_logic has
:rule_name(_,Goal,RuleName,_,_,_):-
    rule_name(Goal,RuleName);
:rule(_,Goal,RuleName,RuleArg,UserList,SubList,1,Num,Num,[],_,_,_):- !,
    rule(Goal,RuleName,RuleArg,UserList,SubList);

local
rule((A=>B),imp_intro,[A,B],[],
  [{[],[A],B}]);
rule(B,imp_elim,[A,B],[A],[{[],[],(A => B)},{[],[],A}]);
rule_name(A => B,imp_intro);
rule_name(_,imp_elim);
end.
```

図 10 展開規則クラス定義の例
Fig. 10 Example definition of a proof refinement class.

2. 証明を作成する．

まず，証明する論理式 $a\Rightarrow(a\Rightarrow b)\Rightarrow b$ をエディタに入力し，1で定義したデータ・ベースを継承させる．バッファの管理する推論規則・定数データ名に存在しない名前を設定すると，そのクラスをエディタが自動的に生成する．その際，継承クラスをきいてくる．ユーザが何も指定しなくとも，自動的に論理体系名と同名のクラスを継承する．プログラム生成規則についても同様．また，この証明固有の定数として以下のものを定数データに追加定義する．

```
a : prop
b : prop
```

3. 証明を展開し，完成させる．

証明を作成する方法には証明展開機能によるものと自動証明機能によるものと2通りがある（3.2.2 項参照）．直観主義インプリケーション・ロジック専用のプルーバを定義してある場合は，これを利用する．システムでは NJ，μ[8] の2つのプルーバを用意している．直観主義インプリケーション・ロジックは NJ に含まれるので，NJ を使用して証明を生成できる．この例題の場合は，システムの用意したプルーバによりトップ・レベルから証

```
1.
      a=> (a=>b)=>b.
      since by imp_intro.
1;1.
        assume a;
         (a=>b)=>b.
        since by imp_intro.
1;1;1.
          assume a=>b;
          b.
          since by imp_elim.
1;1;1;1.
            a=>b.
            since by assumption.
            end_since.
1;1;1;2.
            a.
            since by assumption.
            end_since.
          end_since.
        end_since.
      end_since.
```

図 11 $a\Rightarrow(a\Rightarrow b)\Rightarrow b$ の証明
Fig. 11 Tne proof of $a\Rightarrow(a\Rightarrow b)\Rightarrow b$.

明可能である.

完成した証明を**図11**に記す.

4. プログラムを生成し execute クラスに渡す (execute クラスが存在する場合).

5. おわりに

本稿では, Curry-Howard Isomorphism を利用したプログラム生成方法を現実のソフトウエア生産に利用するための実験システムとして作成した PAPYRUS について述べた. また, ベースとなる理論を示し, システムの構成, 機能の概略について説明した.

PAPYRUS における支援機能を使用することによってプログラム生産効率がどのように向上するかを示すことは難しいが, 証明作成に限っていえば, 例えば, 素数フィルタ (2, 3, 4, 5, …という自然数列から素数の列を生成するプログラム) の仕様を示す論理式の証明は, 単にエディタを使用した場合に比べ 20 分の1の作成時間で作成できた. もちろんこの数式は作成者の能力やシステムに対する知識により変化するであろうが, 少なくともかなりの効率向上が見込まれることは間違いないであろう.

さて, PAPYRUS では仕様からプログラム生成する過程を支援するが, Curry-Howard Isomorphism を利用した方法を, 実際のソフトウェア生産方法として確立するには, さらに以下のものが必要であると思われる.

1. 概念としての仕様から論理式としての仕様を作成するための支援環境.
2. 作成されたプログラムコードを効率の良いプログラムコードに変換する方法.

1に関しては, ソフトウェア工学における仕様の詳細化手法などが有効と思われる. また, 2については, lisp のコードの最適化方法, 特に副作用のないコードから, 副作用のあるコードへの変換方法の研究が重要であると思われるのでぜひ検討したい.

PAPYRUS システム自体について現在さらにさまざまな改良と機能拡張が検討されているが, 特に, TRS を使った E-unification や2階のマッチングを利用した証明構成支援機能, 構造モードでの証明の切り貼り機能は, インプリメントしたいと考えている.

参考文献

1) Howard, W. A.: The Formulae-as-types Notion of Constructon, in *Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, pp. 479-490 (1980).
2) Martin-Löf, P.: Constructive Mathematics and Computer Programming, in *Logic, Methodology, and Philosophy of Science VI* (Cohen, L. J. et al. eds), North-Holland, pp. 153-179 (1982).
3) Sato, M.: QJ: A Constructive Logical System with Types, *France-Japan Artificial Intelligence and Computer Science Symposium* 86, Tokyo (1986).
4) Hayashi, S. and Nakano, H.: *PX: A Computational Logic*, RIMS-573, RIMS, Kyoto University (1987).
5) Coquand, T. and Huet, G.: The Calculus of Constructions, *Information and Computation*, Vol. 76, pp. 95-120 (1988).
6) Harper, R., Honsell, F. and Plotkin, G.: A Framework for Defining Logics, *Proceedings of the Second Symposium on Logic in Computer Science* (1987).
7) Sakai, K.: Toward Mechanization of Mathematies—Proof Checker and Term Rewriting System—, *Proceedings of the First Franco-Japanese Symposium on Programming of Future Generation Computing*, pp. 335-390 (1986).
8) Kawata, H. et al.: *Parallel Process Systhesis from Proofs on Logic μ*, ICOT TR-803 (1992).

川田 秀司

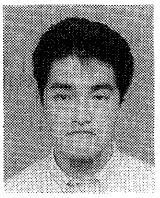
1961年生. 1986年学習院大学理学部数学科卒業. 同年(株)東芝入社. 1989年1月より1992年9月まで(財)新世代コンピュータ技術開発機構出向. 現在, (株)東芝システム・ソフトウェア生産技術研究所にてソフトウェア検証技術の研究に従事. 日本ソフトウェア科学会会員.

坂井　公（正会員）

1976年東京工業大学理学部情報科学科卒業．1978年同大大学院修士課程修了．同年日本電気(株)入社．1982年10月 ICOT 出向．1992年筑波大学電子情報工学系講師．理学博士．理論計算機科学，特に自動定理証明．項書き換えシステム，構成的数学，型理論，制約処理など，数学基礎論の計算機応用の研究に従事．日本ソフトウェア科学会，EATCS 各会員．

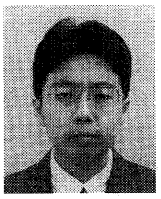
藤田　正幸（正会員）

正昭32年生．昭和55年東京大学教養学部基礎科学科卒業．昭和57年同理学系大学院修士課程修了．昭和58年同大学院博士課程中退，(株)三菱電機総合研究所入社．昭和64年～平成4年財団法人新世代コンピュータ技術開発機械へ出向．定理証明，プログラム合成等の研究に従事．ソフトウェア科学会会員．



白井　康之

昭和39年生．昭和62年東京工業大学理学部数学科卒業．平成元年東京工業大学大学院総合理工学研究科システム科学専攻修士課程修了．平成元年4月より(株)三菱総合研究所，平成4年7月より，(財)新世代コンピュータ技術開発機構出向，現在に至る．主な研究分野：並列定理証明，制約充足問題．



大坪　透

1968年生．1989年日本工学院専門学校卒業．同年(株)ヴァンリッチ入社．1993年，同社退職．現在，にっかつ芸術学院映像科学生．