ソフトウェア基礎論　17－14
(1986. 7. 4)

構成的証明に基づくプログラミング支援システム
INTELLIGENT PROGRAMMING SYSTEM BASED ON THE CONSTRUCTIVE PROOF

構想とその理論的背景
THE DEVELOPMENT PLAN AND ITS THEORETICAL BACKGROUND

高山　幸秀
Yukihide TAKAYAMA
(財)新世代コンピュータ技術開発機構
Institute of New Generation Computer Technology

あらまし　一階述語論理で書かれた形式的仕様を構成的に証明するだけでプログラム開発ができ，プログラムの再利用支援機能も持つICOT-QJの開発構想についてのべる．

Abstract　We explain the development plan of ICOT-QJ which has the function of extacting program from constructive proof of formal specification written in the first order logic and the function which support reusing the previously developed program modules.

1 はじめに

形式的仕様に対する(構成的)証明からプログラムを導出する研究は比較的古くから行われてきた．この方法の最大の問題点は，(構成的)証明を如何に人間にとって自然なレベルで記述できるようにするかである．言い換えれば証明の行間をある程度補完してくれるような証明記述言語系や証明検証システム（証明チェッカー）がどうしても必要になってくる．"構成的証明に基づくプログラミング支援システム"は，ICOTが研究開発を進めてきた定理証明支援システム([廣瀬 他 86]など)で蓄積された技術を応用した証明チェッカー付きプログラミング支援システムである．

本論文は，"構成的証明に基づくプログラミング支援システム"の構想とその理論的背景や思想，ICOT内部の他のプロジェクト(CAP,TRS,JSL)との関連について紹介する．

2 構成的証明に基づくプログラミング技術の理論的背景

2．1 "構成的証明 = プログラム"

2．1．1 プログラミング・パラダイム

数学における定理の証明とプログラムの大きな相違点の一つとしては，プログラムの方がより手続き的なことを多く記述するのに対して，証明の方は手続きの記述が無い場合があることである．しかしながら，"証明は構成的でなければならない"という条件を付けると，定理の証明はきわめてプログラムに類似したものとなる．

プログラミングを定理の証明として行うことの利点は，

1) 数学的素養を持った人ならば，特殊なプログラミングの知識，とくにプログラミング言語の繁雑な文法や書法をそれほど勉強しなくてもプログラミングに入っていける．
2) 論理によって記述できるのでプログラムの論理的構造が分かりやすい．
3) プログラミングと，プログラムの正当性の確認が一体になっている．
4) 証明検証系技術の観点から，プログラムの検証・合成機能実現の手掛かりがつかめる．

図-1 に証明としてのプログラミングと通常のPrologプログラミングとの対応表を示す．

但し，ここでいう検証には仕様そのものがプログラマの意図に沿うものがどうかをチェックする操作は含まれない．また時制を考慮した通信システムやプロセス概念の入ったプログラムの開発支援は現在のところこのパラダイムの範囲外である．

	証明としてのプログラミング	通常のPrologプログラミング
記述レベル	・一階述語論理あるいはそれにほぼ一致するもの	・Horn論理(一階述語論理の部分クラス)＋制御機構
検証情報	・プログラム(証明)そのものが検証情報になっている	・別途付加する必要あり
検証方法	・証明検証系によりほぼ自動的に行われる(静的検証)	・デバッガを使って具体的なデータについて正しく動作することを確認する(厳密には検証ではない)
プログラマに要求される知識・素養	・数学的素養 ・データ構造に関する初歩的な知識 ・正しい論理の筋道を把握できるだけの注意力	・バックトラック，ユニフィケーション，カットなどのProlog特有の制御構造 ・論理の筋道以外の細部に対する注意力(デバッグ時など)

図-1 :: 通常のPrologプログラミングとの比較

2．1．2 realizability interpretation と realizer

構成的証明を記述するための形式的体系としては直観主義論理のものを用いる．論理の筋道だけを形式化した古典論理に比べて，この形式的体系は，if-then -else，場合分け，λ-式，対化 (pairing)，射影(projection) など，手続きを具体的に記述するための記号が付け加えられており，それらに対する推論規則を与えるところに特徴がある．また，証明のなかの∧，∨，→，∀，∃の構造に注意してそれらの記を組み合わせることによってプログラムが導出できる．これをrealizabili- ty interpretationといい，この操作で導出されたプログラムを realizerと言う．

EONという形式的体系([Beeson 85])を例にrealizability interpretationを考えよう．EON とはHilbert & Ackermanの一階述語論理の直観主義論理版に部分組み合わせ代数の公理系と数学的帰納法の公理を付け加えたものと思ってもよい．

1) 論理式Aに対するrealizerを e として，"e が Aを実現する"ことを表わす新たな論理式を "e q A" と書き表す．
2) 各論理式A に対して"e q A"は次のように定義される．ここでp0，p1は射影を与える定数であり，対化(pairing)を与える定数pとともに

$$pxy\downarrow \wedge\ p0*(pxy) = x \wedge p1*(pxy) = y$$

なる公理が成り立っているものとする．また項tに対してt↓とは，"項tが定義できる"ことを表わす論理式，N(x)とはxが自然数であることを表わす述語とする．

$$e\ q\ A \equiv A \quad : A \text{ がatomicの場合}$$
$$e\ q\ (A \to B) \equiv \forall a(A \wedge a\ q\ A \to ea\downarrow \wedge ca\ q\ B)$$
$$e\ q\ \exists x\ A \equiv A(p1*e) \wedge p0*e\ q\ A(p1*e)$$
$$e\ q\ \forall x\ A \equiv \forall x(ex\downarrow \wedge ex\ q\ A)$$
$$e\ q\ A \vee B \equiv N(p0*e) \wedge [(p0*e = 0 \to A \wedge p1*e\ q\ A) \wedge (p0*e \neq 0 \to B \wedge p1*e\ q\ B)]$$
$$e\ q\ A \wedge B \equiv p0*e\ q\ A \wedge p1*e\ q\ B$$

3) EONに限らず，構成的証明記述用の形式的体系でのrealizerの定義の仕方は本質的に同じである．ここで直観主義論理における論理式の意味の捉え方を下に示す．2)の realizerの定義のところで，"realizer"を"証明"と読み替えると下に示した論理式の意味とほぼ一致する．

A ∨ B の証明 = A の証明　または B の証明
A ∧ B の証明 = A の証明　と　　B の証明
A → B の証明 = 任意のAの証明からBの証明を構成する操作手続き
∃xA(x)の証明 = あるcに対するA(c)の証明
∀xA(x)の証明 = 任意のcに対してA(c)の証明を構成する操作手続き
¬A　　の証明 = A → ⊥の証明
　　(= 任意のAの証明から矛盾を導く操作手続き)

4) 論理演算を行うことによってrealizerも変形する．自然演繹法(NK)風の論理で考えると，例えば

$$\frac{A \qquad B}{A \wedge B}(\wedge\text{-導入})$$

に対して規則

$$\frac{e1\ q\ A \qquad e2\ q\ B}{p(e1,e2)\ q\ A \wedge B}$$

を対応させる．この規則を仮に"R-規則"と呼ぶことにしよう．ここでp(e1,e2)が ∧-導入規則の適用によって新たに生成されたrealizerである．実際2)の定義にあてはめてみると，

p(e1,e2) q　A ∧ B
= p0(p(e1,e2)) q A ∧ p1(p(e1,e2)) q B
= e1 q A　∧ e2 q B

となってR-規則の前提部分に矛盾しない．(realizability interpretationの健全性) このように各推論規則に対してR-規則が対応付けられる．構成的証明の証明図が与えられたとき，それぞれの節(推論規則を適用している部分)に R-規則を対応させて新しい節(仮に"R-規則節" と呼ぶ)を作ってゆく．そのとき証明図の根に対応するR-規則節の結論部分に現われている realizerが，その証明から導出されたプログラムになっている．

2．2 Typed Logical Calculus (QJ)

2．2．1 知的プログラミング・システム体系としての QJ

Typed Logical Calculusは佐藤(東北大)が提案している構成的数学向きの論理学 ([Sato 85]) で，その形式的体系はQJと呼ばれる．この理論は形式的仕様，それに対する証明，プログラムを同一の形式的体系の中で記述することを狙っている．さらにrealizability interpretation の手法を使ってプログラムの導出が行える．またプログラムは厳密に型付けがされており

1) リスト,自然数などを表現するために再帰型が導入されている．
2) 型情報を使って正当性検証をするための推論規則群が与えられている
3) 再帰型に関する帰納法の規則が与えられている．

ところに特徴がある．

Typed Logical Calculusの概念図を図-2 に示す．

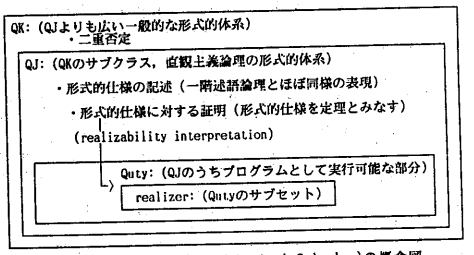

図-2 :: QJ/Quty(Typed Logical Calculus)の概念図

Qutyは佐藤らの開発したプログラミング言語Qute([Sato & Sakurai 84])に型構造を導入した言語である．これは"論理型言語的な書き方もできる関数型言語"とも言うべきものである．以下QJの概略を紹介する．

2．2．2 QJのにおける型

2．2．2．1 型の定義

QJにおける型は基本型 1 と型変数s,t,…によって次の様に帰納的に定義される．

1) s : 型変数　ならば　s は型
2) 1 : 基本型　は　型
3) σ,τ : 型　　ならば　σ × τ　は型
4) σ,τ : 型　　ならば　σ + τ　は型
5) σ,τ : 型　　ならば　σ → τ　は型
6) s : 型変数，σ: 型，s は σに対して"admissible" ならば　μs.σ は型

ここで ×，+，→型構成子で，6)のような型を"再帰型" と呼ぶ．また "admissible"とは，型に対応するデータ領域を考えたとき集合論的矛盾が起きないようにするための制限条件である．

2．2．2．2 型に対応するデータ領域

それぞれの型に対応するデータ領域は，型を定義する型変数に鎖完備半順序集合を対応付けることによって構成される．また型構成子に対応して下に示すような複合データ領域も定義される．特に再帰型に対するデータ領域はScott理論の様に再帰の深さ nに対応するデータ領域 Dnをとって，その極限のかたちで定義される．

〈複合データ領域〉

U + V :　データ領域の直和
U × V :　データ領域の直積
U → V :　部分関数のデータ領域
U∞　　:　再帰関数のデータ領域

また，直和・直積のデータ領域にはleft，right，inl，inr，outl，outrなどの基本関数が定義されている．データ領域とその上の基本関数は2.2.6で述べるQJの表示的意味を与えるのに使われる．

2．2．2．3　抽象データ型の扱い

Typed Logical Calculusにおける型は，後で述べる構成的型理論よりもやや厳しく定義されている．すなわち，

1) プログラムの型は最初から定義されていて型をオブジェクトとして扱えない従って，プログラマが行う型宣言が制限される
2) 型の構成要素は基本型 1 と型変数と決っており，抽象的な対象を表わす型の定義が制限される

といった特徴がある．所謂抽象データ型の扱いが十分でないのだが，それに対してはパラメータを含んだ型を導入するなどの改善が試みられている．

2．2．3 QJにおけるプログラム

QJにおける式は formula(論理式) と term(項) に分類される．termはformula のサブセットでありQJにおけるプログラム(実行可能部分)をあらわす．

termの定義は次の通り．

1) x : 型σの変数　はterm
2) ⊥は任意の型のterm
3) T は基本型 1 のterm
4) a :型 σ　のterm，b : 型σ のterm ならば a = b は基本型 1 のterm
5) a :型 σ　のterm，b : 型τ のterm ならば a ,b　は型 σ × τ の term
6) a :型 σ　のterm，b : 型τ のterm ならば a ∧ b は基本型 1　　の term
7) a :型 σ　のterm，b : 型τ のterm ならば a ; b は型 τ　　の term
8) a :型 σ　のterm　ならば inlσ　a は型σ + τ のterm
9) b :型 τ　のterm　ならば inrτ　a は型σ + τ のterm
10) x :型 σ　のterm，a : 型τ のterm ならば λx.a は型 σ → τのterm
11) a :型σ→τのterm，b:型σのterm　ならば ab は型τのterm
12) σ : b-型，x :型σのterm，a :型σのterm　ならば μx.a は型σのterm
13) a :型σのterm，b :型τのterm，c :型τのterm　ならば if a then b else c は型τのterm
14) x :型σのterm，a:型τのterm　ならば ιx.a は型τのterm

ここで,5)はand-並列 7)はsequential-andを表わす．また inl,inrは型構成子＋を含んだ型のデータ(例えば自然数やリストの要素)を表現するのに使う．10) は関数 11)は関数適用 12) は再帰呼出し 14) は a を満たす唯一のxを求めて返すプログラムをそれぞれ表わしている．また 12)のb-型とはμでバインドされていない型変数(自由変数)をもたないもので，さらに対応するデータ領域が最小限をもつような型のことでμが最小不動点を表わすことからこの条件が付く．

termに対してformulaとは，∧，∨，→，∃，∀に関する通常の論理式と，a =〈 bといった順序に関する表現，及びtermをさすterm以外のformulaの型は基本型 1と定義される．すなわち，基本型 1 は真偽値をもつformulaの型を意味している．

termがプログラムを表現するのに対して,formulaは形式的仕様やそれに対する証明を記述するためのものである．

2．2．4 QJにおける形式的仕様

QJにおける形式的仕様とは

∀x, ∃y. A[x,y]

の形のformulaである．

この形式的仕様を満たすプログラムfとは

∀x. (f(x) ∧ A[x,f(x)])

となる関数を指す．

〈例〉

自然数のリストのソート・プログラムの形式的仕様は，次のように書ける．

```
∀x.∃y.(sorted[y] ∧ perm[y,x])
 where
sorted[x] ≡
        ∀i.∀j.(i=⟨j ∧ j=⟨length(x)
                              → elem(i,x) =⟨ elem(j,x))
elem ≡
       μz.λi,x. if i = 1 then head(x)
                              else z(prd(i),tail(x))
perm[x,y] ≡ ∀i.(occ_no(i,x) = occ_no(i,y))
occ_no ≡ μz.λi,x. if x = [] then 0
          else if head(x) = i then succ(z(i,tail(x))
                              else z(i,tail(x))
```

ただし，変数の型宣言は省略してある．

Typed Logical Calculusにおいては，次の定理が成り立つ．

〈定理〉〉

もし形式的仕様がQJの中での形式的演繹によって証明可能であるならば，必ずそれを満たすプログラムfが存在する．

この定理の証明は構成的なものである．すなわち形式的仕様の証明からプログラムを導出する方法は上の定理の証明によって明示的に与えられている．

2．2．5 QJの規則群

QJは一種の自然演繹系である．しかしながら通常の論理記号(∧,∨,→,∀,∃)に関する推論規則の他に1) 拡張された論理記号に関する規則 2) 型記号に関する規則 3) 順序・等号に関する規則 4) 帰納法に関する規則が与えられている．

1) としては,2.2.3に挙げたtermを表現するための論理記号に関する規則が与えられている．例えば μ-導入規則は次の通りである．

$$\frac{[x =\langle y]\quad a =\langle a[x \leftarrow y]}{\mu x.a}$$

これは，a が変数xに関して単調であるときに a の最小不動点が存在して$\mu x.a$ を定義できることを表わす．

また論理記号に関する規則には，その規則に現れるformulaの型チェックのための規則が付随する．

2) としては例えば型構成子+ に関する規則として +-除去規則などがある．

3) としては等号の規則が重要である.例えば

$$\frac{\mu x.a}{\mu x.a = a[x \leftarrow \mu x.a]}$$

これは再帰呼出しのプログラムを計算するための規則である．

その他にもλ式，if-then-else，ι式に関する計算規則も与えられており，プログラムの実行はすべて推論規則で説明できるようになっている．

4) として特徴的なのは再帰型に関する規則である．再帰型を$\mu s.\sigma$として，unary abstract Aをfixする．このAは帰納法で証明したい命題と思ってよい．これに対してσから下のように構成されるunary abstract $\sigma*$ を定義し，

$$\frac{[\sigma*[x]]\quad A[x]}{\forall x.\ A}$$

という推論規則が導入されている．

〈$\sigma*$ の構成法〉

1) $s*[x] \equiv A[x]$　　s ： 型変数

2) $1*[x] \equiv T$

3) $(\sigma 1 \times \sigma 2)*[x] \equiv \sigma 1*[\mathrm{left}\ x] \wedge \sigma 2*[\mathrm{right}\ x]$

4) $(\sigma 1 + \sigma 2)*[x] \equiv ((\mathrm{outl}\ x) \wedge \sigma 1*[\mathrm{outl}\ x]) \vee ((\mathrm{outr}\ x) \wedge \sigma 2*[\mathrm{outr}\ x])$

ただし，$\mathrm{outl}\ a \equiv \iota x.(a = \mathrm{inl}\ x)$

$\mathrm{outr}\ a \equiv \iota x.(a = \mathrm{inr}\ x)$ $(a : \sigma+\tau)$

5) $(\sigma 1 \rightarrow \sigma 2)*[x] \equiv T$

6) $(\mu t.\sigma)*[x] \equiv T$

この規則を適用する帰納型によって通常の数学的帰納法やリストの構造に関する帰納法の規則などが得られる．その意味で一般性を持った強力な帰納法である．例えば，$\sigma = \mu s.1+s$ すなわち自然数の集合Nの型を考える．Nの要素は$0 \equiv (\mathrm{inl}\ T)$，$1 \equiv (\mathrm{inr}\ (\mathrm{inl}\ T))$，$3 \equiv (\mathrm{inr}\ (\mathrm{inr}\ (\mathrm{inl}\ T)))$,…と定義される．このことから，上の規則は下の様に変形できる．

$$\frac{A[0]\quad [\ A[x]\ ]\ A[\mathrm{succ}(x)]}{\forall x.A[x]}$$

2．2．6 QJの意味論

2．2．6．1　3種類の意味論

QJには表示的意味，公理的意味，操作的意味が与えられている．表示的意味は，多ソート論理のモデルを与えることによって定義する.公理的意味は，2.2.5に示した推論規則による形式的演繹に基づいて定義される．操作的意味はQJのなかの実行可能(executeble)な部分，すなわちプログラム言語 Quty に対して一種の項書換えシステムとして定義される．

2．2．6．2 プログラム・形式的仕様の検証

2.2.6.1の公理的意味論は，形式的仕様を定理自動証明・証明検証の手法で証明するための基礎を与えている．すなわち，操作的意味論の意味で真であることを|= ，公理的意味論の意味で真であることを|-で表わすと，

|- A ならば |= A　(Soundness of QJ)

が証明されているので，Aとして特に形式的仕様を考えれば定理自動証明・証明検証手法によって形式的仕様に対する証明の正当性が完全に保証される．

また，操作的意味論の書換え規則をA =〉 Bと表わすと

A =〉 B　ならば |- A　B

も証明できる．これは，プログラムの実行過程(項書換え)を公理的意味論によって捉え直すことができ，したがって定理自動証明手法によってプログラムの検証が可能であることを意味する．

2．2．7 QJのrealizability interpretation

ここで2.2.4にあげたソートの例を使って，形式的仕様の証明と証明からプログラムが導出される過程と概観してみよう．

ソートの形式的仕様は

∀X ∃Y (sorted(Y) ∧ perm(X,Y))　--- (＊)

であった．

〈略証〉

超限帰納法の規則

$$\frac{[\forall\ Y\ (Y \langle X \rightarrow A[Y])]\quad A[X]\quad WF(\langle)}{\forall\ X.\ A[X]}(\mathrm{TI}\ \langle)$$

(WF は順序〈の条件)

を使って証明する．その前にリストの集合の順序'〈'を次のように定義しておく．

リストL1，L2 に対して

L1 〈 L2 〈=〉 length(L1) 〈 length(L2)

証明の筋書きは次の通りである．

[超限帰納法の仮定]

リストXより小さい任意のリストについて∃ Y0 (sorted(Y0) ∧ perm(X0,Y0) が成り立つ．すなわち，

∀X0 (X0 〈 X → ∃ Y0 (sorted(Y0) ∧ perm(X0,Y0))　--- (1)

[結論]

あるリストX をもってくれば，それについて∃ Y0 (sorted(Y0) ∧perm(X0,Y0)が成り立つ．

∃Y. (sorted[Y]∧ perm[Y,X])　--- (2)

(2)が示せれば[TI〈規則]によって

∀ X . ∃Y (sorted[Y] ∧ perm[Y,X])

となる．

まず自然数リストはnilがそうでないかで場合分けができることが証明できる．すなわち ∀X:μs.(1+(μt.1+t)×s).(X = [] ∨ ¬(X=[])

[注意] QJは直観主義論理なので，A∨¬Aは自明ではなく証明すべきものである

[1] X = [] の場合

Y = []とすれば良い．

実際，sorted([])

≡ ∀i.∀j.(i =〈 j ∧ j =〈 length([])
→ elem(i,[]) =〈 elem(j,[]))

で，length([]) = 0 だから j が0でない自然数であることに反するから矛盾．従って上式は真である．また，perm([],[]) についても同様．よって X = [] の場合 Y = []とすることによって(2)が成立する．

[2] ¬(X = []) の場合

次の補題を参照する．

[補題]

任意の自然数リストX0 および　任意の自然数 I に対して，X0 は I 以上の要素だけを持つ部分リスト(nilもありうる) と I より小さい要素だけを持つ部分リスト(nilもありうる)に分割できる．

i.e.

∀X0 ∀I ∃X1 ∃X2

(below[X1,I]∧above[X2,I]∧perm[append[X1,X2),X0])　--- (3)

where below[X,I] ≡ ∀J (J =〈 length(X) → elem(J,X) 〈 I)

above[X,I] ≡ ∀J (J =〈 length(X) → I =〈 elem(J,X))

[証明]

省略．

¬(X=[])ゆえ head(X) も tail(X) も存在する．そこでそれぞれをI，X0 とおくと，[補題]の(3)より，

∃X1 ∃ X2 (below[X1,I] ∧ above[X2,I]∧
perm[append(X1,X2), X0])
--- (4)
一方X1 (X (すなわち length(X1) ⟨ length(X)) ゆえ (1) より
∃Y1 (sorted(Y1) ∧ perm(Y1,X1)) --- (5)
同様に
∃Y2 (sorted(Y2) ∧ perm(Y2,X2)) --- (6)
そこで Y = append(Y1,[I,Y2])とおけば
sorted(Y) = sorted(append(Y1,[I,Y2))
ここで(5)(6)より
sorted(Y1) --- (7) sorted(Y2) --- (8)
である．また同じく(5)(6)よりperm(Y1,X1),
perm(Y2,X2)だから，X1,X2はそれぞれY1,Y2の要素を適当に並べ変えたものにすぎない．さらに (4)によりbelow(X1,I), above(X2,I)だから，
below(Y1,I) --- (9) above(Y2,I) --- (10)
が成り立つ．(7)(8)(9)(10)より結局append(Y1,[I,Y2])は既にソートされているとわかる．すなわち
sorted(append(Y1,[I,Y2]))
が成り立っている．
一方perm(Y,X) = perm(append(Y1,[I,Y2]),X)を考えると，
perm(append(Y1,[I,Y2]),X)
= ∀i.(occ_no(i,append(Y1,[I,Y2])) = occ_no(i,X)
= ∀i.(occ_no(i,[I.append(Y1,Y2)) = occ_no(i,[I.X0]))
= ∀i.(occ_no(i,append(Y1,Y2)) = occ_no(i,X0))
ここでperm(Y1,X1), perm(Y2,X2)ゆえ，
∀i.(occ_no(i,append(Y1,Y2)) = occ_no(i,append(X1,X2)))
従って
= ∀i.(occ_no(i,append(X1,X2)) = occ_no(i,X0))
≡ perm(append(X1,X2),X0)
最後の式は，(3)により成り立つ．よって
perm(append(Y1,[I.Y2]),X)
が成り立っている．
以上により
sorted(append(Y1,[I.Y2])) ∧ perm(append(Y1,[I.Y2],X)
すなわち
sorted[Y] ∧ perm[Y,X]
を得る．
(3) 結論
(1) (2) によりどんな場合にも(2)が成り立つことが言えた．
〈証終〉

以上によりソートの形式的仕様の証明を得た．つぎにプログラムの導出の様子を見てみよう．まず，〔補題〕の証明から導出されるプログラムは

g1≡μZ1.λX0.if outl(if outl(X0) then L else R)
then λI.[] else λI.if head(tail(X0))(I then
[head(X0).Z1(tail(X0))(I)] else Z1(tail(X0))(I)
g2≡μZ2.λX0.if outl(if outl(X0) then L else R)
then λI.[] else λI.if I=(head(tail(X0)) then
[head(X0).Z2(tail(X0))(I)] else Z2(tail(X0))(I)

で それぞれ，below[g1(X0), I] と above[g2(X0), I] を満たしている．すなわち X0 から〔補題〕で定義したX1, X2 を計算する関数である．ここで，

outl(if outl(X0) then L else R)

の部分はX0 = []か否かの判定をしていると思えば良い．
上の証明から導出される．仕様(＊)に対応するプログラムは

qsort≡μZ.λX.if X=[] then []
else append(Z(g1(tail(X))),[head(X).Z(g2(tail(X)))])

となる．
これを導出するときのrealizability interpretationは，大まかに言って次のようになる．
1) 証明(1)の部分で "X = [] ならば Y = []" の部分だけを取り出して記憶しておく．
2) 証明(2)の(4)の部分は∃X1 ∃X2 (…)の形の論理式だが，実際にg1,g2を使って X1 = g1(tail(X)) X2 = g2(tail(X)) とできる
3) (5)の部分は∃Y1 (…)の形であるが，このY1は今求めようとしている関数を変数Z で表わせば Y1 = Z X1 = Z (g1(tail(X))) とできる．
4) 同様に(6)の部分についても Y2 = Z X = Z (g2(tail(X))) とできる．
5) 証明(2)の最後にappend(Y1,[I,Y2])が仕様(＊)を満たすためのYであるとしているので
Y = append(Z(g1(tail(X))), [head(X). Z(g2(tail(X)))])
を得るのでこれを記憶する．
6) 証明(3)にて(1)(2)に対して(∨-除去規則)で証明しているので，
1) 5)で得たものを if-then-elseで繋ぎ
if X=[] then []
else append(Z(g1(tail(X))),[head(X),Z(g2(tail(X)))])
を作る．
7) 最後に超限帰納法(TI ⟨⟩)を使っているので，6)で得た項を再帰呼出しの関数にして結局上のqsortを得る．

この様にQJにおけるプログラム導出は，構成的証明を解析してプログラムとして使える部分を捜し出し，それらを組み合わせるという方法をとる．これは2.3で述べる構成的型理論に基づくものとはやや違っている．

2．3 構成的型理論

2．3．1 構成的型理論の特徴

直観主義論理や構成的数学の流れの一つに構成的型理論(Constructive Type Theory) がある．これはとくに1970年代からMartin-Löfやde Bruijnらの手によって発展してきた論理学であり．論理定数や形式的演繹の意味を直観主義の考え方で捉えなおし，構成的数学を十分記述できるような形式的体系の構築を狙ったところに特徴がある．

例としてMartin-Löfの体系([Martin-Löf 82][Martin-Löf 84])について考えると，集合は
1) canonical element の形式とその等号条件
2) non-canonical elementの評価(evaluation)規則
を与えることによって決定されるものと定義され，さらに集合から新しい集合を構成する規則が与えられている．すなわちCartesian積(Π)と直和(Σ)に対して(1)構成規則 (2)導入規則 (3)除去規則 (4)等号規則 を与えている．構成規則は集合の構成規則で，集合の構成の仕方および2つの集合が等しくなるための条件を示している．導入規則は集合の canonical elementの構成条件および等号条件，除去規則は集合のelementに対する基本関数の構成規則，等ゲ規則はその基本関数の実行(評価)規則をそれぞれ与えている．また ΠとΣの規則を"命題=型"という解釈で捉え直すことによって，通常の論理学の論理定数に関する公理系・推論規則が導かれる．また，この2つの規則から通常のプログラミング言語で使われるif-then-elseや射影，対化，超限帰納法などのスキーマも自然に導かれる．

2．3．2 構成的型理論における証明とプログラム

[Smith 82]ではMartin-Löfの体系の中でクイック・ソートのプログラムが導出する例を示している．Martin-Löfの形式的体系では原始帰納的関数のみが許される．これは形式的体系の意味論を簡単にし，プログラムの検証を容易にするための配慮である．ほとんどのプログラムが原始帰納関数で比較的簡単に書き表せるが，クイック・ソートの場合はやや困難である．[Smith 82]では course-of-value equation の解を原始帰納関数で表わす手法がMartin-Lofの形式的体系の中で証明できることを示し，それを使ってプログラムを導出している．

以下では[Smith 82]に従った例を示す．証明は非形式的な記述法によった．例で使う Martin-Löfの形式的体系における推論規則を下に示す．

〔→ 導入〕

$$\frac{\begin{array}{c}[x \in A]\\ b(x) \in B\end{array}}{(\lambda x)b(x) \in A \to B}$$

〔→ 除去〕

$$\frac{a \in A \quad (\forall x \in A)B(x)}{B(a)}$$

〔Π-導入〕

$$\frac{\begin{array}{c}[\,x \in A\,]\\ b(x) \in B(x)\end{array}}{(\lambda x)b(x) \in (\Pi x \in A)B(x)}$$

〔Π-除去〕

$$\frac{a \in A \quad c \in (\Pi x \in A)B(x)}{ap(c,a) \in B(a)}$$

但し ap は関数適用を表わす．

〔List 除去〕

$$\frac{l \in List(A) \quad c \in C(nil) \quad [x \in A,\ y \in List(A),\ z \in C(y)]\ e(x,y,z) \in C(x.y)}{listrec(l,c,e) \in C(l)}$$

但し listrec(nil,c,e) = c,
listrec(a.l,c,e) = e(a,l,listrec(l,c,e))

〔N 除去〕

$$\frac{n \in N \quad c \in C(0) \quad [\,x \in N,\ y \in C(x)\,]\ e(x,y) \in C(succ(x))}{rec(n,c,e) \in C(n)}$$

但し rec(0,c,e) = c, rec(succ(a),c,e) = e(a,rec(a,c,e))

〈問題〉

任意のリスト l （その要素の型はAとする）に対してそれをソートしたリストが存在することを証明せよ．

この問題は実際にソートしたリストを構成して見せれば解けたことになる．この問題を形式的に

C(l) 〔 l ∈ List(A) 〕

という述語の形で書き表す． これは述語論理を使ってもう少し詳しく書くと

C(l) ≡ ∃s ∈ List(A) 〔 perm(l,s) ∧ ordered(s) 〕

というふうに表せる．

以下($x_1, x_2 \cdots x_n$)AはAのn-ary abstractionを表わすものとする．

〈補題-1〉

任意のリストl ，s に対して，それらを繋ぎ合せたリストが存在する．

〈補題-1の証明〉

詳細省略．canonical proof は concat(l,x) ≡listrec(l,s,(x,y,z)(x.z))

〈補題-2〉

任意のリスト v（要素の型はAとする）とAの要素uに対してvの要素のうち u以上の大きさのものだけを集めたリストが存在する．その長さは length(v)以下である．

〈補題-2の証明〉

詳細省略．canonical proof は filter=〈(u,v) ≡ listrec(v,nil,(x,y,z)(if x=〈v then x.z else z) length(filter=〈(u,v)) =〈 length(v)

〈補題-3〉

任意のリスト v（要素の型はAとする）とAの要素uに対してvの要素のうち uより小さいものだけを集めたリストが存在する．その長さはlength(v)以下である．

〈補題-3の証明〉

詳細省略．canonical proof は filter〉(u,v) ≡ listrec(v,nil,(x,y,z)(if x =〈 v then z else x.z) length(filter〉(u,v)) =〈 length(v)

〈補題-4〉

任意の自然数 n を固定した時，長さ n 以下の任意のリスト l（要素の型はAとする）に対して C(l) が成り立つ．

この補題を形式化して次のようにする．

P(n) ≡ (Π l∈List(A))(length(l) =〈 n → C(l))

〈P(n)の証明〉

証明はP(n)を型とみなして，その型に対する証明を構成すればよい．

〔1〕 P(0) の証明

長さ0以下のリストとしてはnilリストしかない．nilリストをソートしたリストはnilに他ならない．従ってC(nil)の証明はnilに他ならない．

nil ∈ C(nil) ---- (1)

そこでlength(nil) =〈 0であることの証明を'p'とおけば，〔→導入規則〕により

(λp)nil ∈ (length(l) =〈 0 → C(l))

さらに，l ∈ List(A)であることから 〔Π-導入規則〕により

(λl)(λp)nil ∈ (Π l∈List(A))(length(l) =〈 0 → C(l))

すなわちP(0)の証明(λl)(λp)nilが得られた．

〔2〕 P(x)を仮定してP(succ(x))の証明を導こう

まずP(x) に対する証明を'y'とおく．

y ∈ (Π l∈List(A))(length(l) =〈 x → C(l) --- (2)

そこで

Q(l) ≡ length(l) =〈 succ(x) → C(l)

とおいてまずこれを証明しよう．

1) Q(nil) の証明

length(nil)(= 0) =〈 succ(x)の証明を'q'とおけば，(1) と〔→導入規則〕により

(λq)nil ∈ length(l) =〈 succ(x) → C(nil)

すなわちQ(nil)の証明(λq)nilが得られた．

2) Q(v) を仮定したときの Q(u.v) の証明 〔ただし u ∈ A, v ∈ List(A)〕 u.v の長さがsucc(x)以下であることが分かっているとして，その証明を'r'とする．すなわち

r ∈ length(u.v) =〈 succ(x) --- (3)

このとき

succ(length(v)) = length(u.v) =〈 succ(x)

よって両辺のsuccを取って

length(v) =〈 x

このことと補題-2,3により

length(filter=〈(u,v)) =〈 length(v)

length(filter〉(u,v)) =〈 length(v)

であることから =〈 の推移性によりある証明's1','s2'が存在して

s1 ∈ length(filter=〈(u,v)) =〈 x --- (4)

s2 ∈ length(filter〉(u,v)) =〈 x --- (5)

さらに補題-2,3により

filter=〈(u,v) ∈ List(A) , filter〉(u,v) ∈ List(A)

ゆえ(2)により〔Π-除去規則〕を使って

ap(y,filter=〈(u,v)) ∈ (length(filter=〈(u,v)) =〈 x → C(filter=〈(u,v))) --- (6)

ap(y,filter〉(u,v)) ∈ (length(filter〉(u,v)) =〈 x → C(filter〉(u,v)) --- (7)

そこで(4)(5)により(6)(7)に対して〔→除去規則〕を使って

ap(ap(y,filter=〈(u,v)),s1) ∈ C(filter=〈(u,v)) --- (8)

ap(ap(y,filter〉(u,v)),s2) ∈ C(filter〉(u,v)) --- (9)

ところでQ(v) ≡ length(v) =〈 succ(x) → C(v)を仮定していたから，その証明，とくにC(v) の証明が存在するはずである．それを'g'とおく．ところが〈補題-1〉の 'concat'を使って(8)(9) よりconcat(ap(ap(y,filter=〈(u,v)),s1),ap(ap(y,filter〉(u,v)),s2))を考えればこれが実はC(v)の証明になっていることがわかる．また，これから

concat(ap(ap(y,filter=〈(u,v)),s1), u.ap(ap(y,filter〉(u,v)),s2)) ∈ C(u.v) --- (10)

が証明できる．

そこで(3)(10)に〔→導入規則〕を適用して

(λr)concat(ap(ap(y,filter=〈(u,v)),s1), u.ap(ap(y,filter〉(u,v)),s2)) ∈ length(u.v) =〈 succ(x) → C(u.v)

を得る．

3) 1)2)から〔List-除去規則〕により

listrec(l,(λq)nil, (u,v,w)(λr)concat(ap(ap(y,filter=〈(u,v)),s1), u.ap(ap(y,filter〉(u,v)),s2)) ∈ (length(l) =〈 succ(x) → C(l)) ≡ Q(l)

を得る．すなわちQ(l)の証明が得られた．

そこで l ∈ List(A)より〔Π-導入規則〕により

(λl)listrec(l,(λq)nil, (u,v,w)(λr)concat(ap(ap(y,filter=〈(u,v)),s1), u.ap(ap(y,filter〉(u,v)),s2)) ∈ (Π l∈List(A))(length(l) =〈 succ(x) → C(l)) ≡ P(succ(x))

すなわちP(succ(x))の証明が得られた．

〔3〕 結論部

〔1〕〔2〕により〔N-除去規則〕から

Q(n) ≡

rec(n, (λl)(λq)nil, (x,y)(λl)listrec(l,(λq)nil, (u,v,w1,w2)(λt(ap(ap(y,filter=〈(u,v)),s1), u.ap(ap(y,filter〉(u,v)),s2)))) ∈ (Π l∈List(A))(length(l) =〈 → C(l))

where

n ∈ N

q ∈ length(l) =〈 succ(x)

r ∈ length(u.v) =〈 succ(x)

s1 ∈ length(filter=〈(u,v))=〈 x

s2 ∈ length(filter〉(u,v))=〈 x

すなわちP(n)の証明が得られた．

〈補題P(n)証明終〉

〈問題の証明〉

〈補題P(n)〉により

Q(n) ∈ (Π l∈List(A))(length(l) =〈 n → C(l))

であるが，ここで

l ∈ List(A)

をとり

n = length(l)

とおくと〔Π-除去規則〕により

ap(Q(length(l)),l) ∈ length(l) =〈 length(l) → C(l)

すなわち

ap(Q(length(l)),l) ∈ C(l)

よって問題の証明は

f(l) ≡

ap(rec(length(n), (λl)(λq)nil, (x,y)(λl)listrec(l,(λq)nil, (u,v,w1,w2)(λr)concat(ap(ap(y,filter=〈(u,v)),s1), u.ap(ap(y,filter〉(u,v)),s2))), l)

となる．

〈問題の証明終〉

上の証明で得られたf(l)は，次のcourse-of-value equationを満たすことに注意する．

f(nil) = nil

f(a.l) = concat(f(filter=〈(a,l)),a.f(filter〉(a,l)))

この様に構成的型理論においては，形式的仕様に対する構成的証明を行うことが即ちプログラム導出になっている．また"命題=型"

の考え方により，構成的型理論において必要に応じて命題そのものを扱ったり，あるいは命題の(canonical)proof を陽に扱ったりして証明が記述できる．これはプログラムの性質を証明する場合には非常に便利である．

2．3．3　構成的型理論における一般化

構成的型理論のもう一つの応用としてプログラム・モジュールの一般化がある．これは最近おもに萩谷([萩谷 85]〔萩谷 86〕)によって研究されているテーマである．プログラム・モジュールの一般化は，ソフトウェアの再利用技術の基礎という観点から重要である．

例えば，自然数リストの各要素に1を加えたリストを返す関数を考える．

```
fix (λ map (λ x (if  (null x ) (nil)
        (cons (add1 (car x ))(map (cdr x ))) )))
                        ∈ (list natnum) → (list natnum)
```

ここで fix は不動点オペレータであり，(list **)は型**の要素をもつリストの型をあらわす．構成的型理論ではプログラム(項)が全てある型の証明として扱われるので，上のプログラムの各項は型付けが成されている．すなわち

```
map ∈ (list natnum → natnum)
fix ∈ (Πx∈TYPE)((x → x) → x)
bool ∈ TYPE
null ∈ (Πx∈TYPE)((list x) → bool)
if ∈ (Πx∈TYPE)(bool → x → x → x)
add1 ∈ natnum → natnum
list ∈ TYPE → TYPE
nil ∈ (Πx∈TYPE)(list x)
car ∈ (Πx∈TYPE)( (list x) → x )
cdr ∈ (Πx∈TYPE)( (list x) → (list x))
cons ∈ (Πx∈TYPE)(x → (list x) → (list x)
```

但し，ここでTYPEとはおおざっぱに言って型全体の集合と考えてよい．この型付けを陽にプログラムとして記述すれば；

```
fix (λ map:((list natnum)→(list natnum))(λ x:(list natnum)
          (if:_ (null:_ x:_ ) (nil:(list natnum))
              (cons :_ (add1:(natnum → natnum)
                       (car:_ x:_ ))(map:_ (cdr:_ x:_ )))
   ))) ∈ (list natnum) → (list natnum)
                                          --- (1)
```

(ただし'_'は簡単のため省略)

ここで(1)の型を型変数 P ∈ TYPE, Q ∈ TYPEを使って

```
(list P) → (list Q)
```

とパラメータ化してみる．

そして add1の部分を型 P → Q の変数Fで置き換えてみるとプログラムは

```
fix (λ map: ((list P)→(list Q))(λ x : (list P)
          (if :_  (null:_  x:_  ) (nil:(list P))
              (cons:_ (F:(P → Q)  (
             car:_ x:_ ))(map:_ (cdr:_ x:_ )))
   ))) ∈ (list P) → (list Q)          --- (2)
```

となる．この場合(2)のプログラムは(1)のプログラムの一般化になっている．ここで型に関する規則が無制限なパラメータ化を防いでいる．しかしながら一つのプログラムに関してその一般化は複数存在し，単に局所的なパラメータ化だけをやっていては意味のない一般化ができてしまう．型に関する規則だけではそれを解決できないところにこの手法の難しさがある．萩谷はこの問題をプログラマによる検証情報の記述，ヒューリスチックの導入などで解決しようとしている．また萩谷の手法は，構成的型理論が豊かな型構造をもつということの他に，型をオブジェクトとして自由に扱えるという特徴が本質的に利いている．

3 いくつかの考察

以上のことがらをまとめて構成的証明に基づくプログラミング技術に関する基礎的考察を試みてみる．

3．3．1　Martin-Löfの体系とQJ

Martin-Löf流の方式とQJ流の方式を，比較してみると次のようになる．

1) Martin-Löfの形式的体系は，一般帰納的関数の記述は許していない．一方，QJ の方は一般帰納的関数の記述が許されている．
2) Martin-Löfの形式的体系では関数は全て全域関数として扱っている．一方，QJ は関数は全て部分的関数としている．
3) Martin-Löfの構成的型理論では，型をオブジェクトとして扱っているのでプログラム言語 ，証明/仕様記述言語系としては型宣言や型の操作の自由度が大きい．一方，QJの型機構では型をオブジェクトとして扱うことは許さない．

以上のことから，Martin-Löfの体系に基づくプログラム言語系は明解な意味論と単純ではあるが強力な型機構を持っているという意味で理論的に非常に興味深いものではあるが，現実のプログラミングの現象をカバーしきっていないきらいがある．それに対してQJの意味論は Martin-Löfのものに比べて単純ではなく，型の扱いの自由度に制限はあるものの一般帰納的関数や部分的関数が扱えるという点からより現実のプログラミングに近いものになっている．

3．3．2　非決定的処理プログラムの問題

証明から導出されるプログラムは，Martin-Löf流のものもQJによるものもいずれも決定的処理を記述したものである．すなわち構成的証明は，特に存在の証明を具体的にそのものの構成方法を明示することによって行う．従って構成的証明から導出されたプログラムは手続きを決定的に記述したものになる．それに対して証明の概念・記述法を拡張することによって非決定的処理を記述したプログラムが導出できる可能性は否定できない．しかし非決定的処理の記述に対する意味論がプログラムの計算機による検証・変換・合成に応用できる程数学的に厳密なものにまで発展するにはまだ時間がかかることが予想される．以上のことから，

1) 構成的証明に基づくプログラミング支援システムを論理型プログラミングについて考えた場合，それは論理型プログラムの決定的処理の部分を支援するものになるであろう．
2) 実際的な応用としては，数式処理の分野が考えられる．

3．3．3　関数型言語と論理型言語

Martin-Löf流の方法とQJの方法に共通していることは，

1) プログラムのデータ構造や制御構造を記述するrec，inl，if-then-elseなどを使って構成的証明を記述すること
2) 証明から導出されたプログラムは結局それらを適当に組み合わせたものになっていることである．

それに対して論理型言語特にpur-PROLOGなどはプログラムの制御構造を陽に記述するものが無く，従ってプログラミングが容易であるという利点がある．その半面構成的証明からプログラムを導出する手法が直接的には使えないという問題がある．また，決定的処理を記述する際には，プログラムの制御機構を陽に指定したい場合が多いことを考えると論理型言語で決定的処理は必ずしも書きやすいとは言えない．

一方，論理型言語はプログラムのデータ構造を表現する項表現や項の扱いが単純すぎるので数学の記述などは必ずしも書きやすいとは言えない．

以上のことから次のことが言える．

1) 論理型言語に豊かな項表現や適当な制御構造の記述を導入し関数型言語的な記述ができるものに拡張することが必要である．
2) しかし論理型言語と関数型言語のプラグマチックな融合は，言語の意味論の明解さや厳密さを損ない，計算機による検証・変換・合成技術への可能性を閉ざしてしまう危険がある．従って理論と実践の両面からの融合が必要である．

3．3．4　最適化の問題

数学の証明とプログラムの記述の大きな違いのひとつとして，数学には操作の節約すなわち最適化の概念が無いことである．これについては次のように考えられる．

1) アルゴリズムを高い視点から見直すことによって本質的な最適化ができる．プログラムを証明として記述することが具体的にこの問題にどれほど有効であるか?
2) 証明の戦略と導出されたプログラム効率の関係が明確でない("証明の最適化")
3) 証明図の正規化や，また最近盛んになってきている証明の複雑さの研究が"証明の最適化" にどれほど有効であるか?
4) 導出されたプログラムをプログラムの等価変換技術を応用して最適化した場合，証明とプログラム，そして導出されたプログラムと最適化されたプログラムとの明解な対応関係が付けられるか?

いずれにしても最適化の問題は未知の部分が多い．

またコンパイラ・レベルの最適化を考えると，一般に型機構があれば最適化がしやすくなると言われているが，これは a ∈ A (a は項，Aは型) のチェックが自動的に行えることが前提になる．従来の型機構付言語(Pascalなど)はこの条件を満たすように作られており，Quty も同様である．しかし，構成的型理論の場合は (Πx∈A)B(x) なる型が中心的な位置を占めていて，B(x)としては特に制限を設けていないので a ∈ (Πx∈A)B(x) が決定的(decidable)でなくなる．したがって，構成的型理論に基づくプログラム言語の場合，型機構をつかったコンパイラ・レベルの最適化は難しくなると思われる．

3．3．5　モジュール知識ベースとパラメータ化

基本アルゴリズム程度の大きさのプログラミング支援を中心に考えたとしても，開発済みのプログラムやそれに類似したプログラムを使って新しいプログラムを開発する必要は起こる．証明について考えれば，証明済みの補題を参照する必要が頻繁に起こるし，以前に行ったものと類似の方法を使って証明をしたいことも起こる．そこでモジュール知識ベースとパラメータ化の問題が出てくる．

ここでパラメータ化とは2.3.3で述べたような"プログラムの一般化"と，逆に一般化されているプログラムのパラメータ部分に，その型情報に合致した適当なプログラムを代入することにより必要なプログラムを得る"プログラムの特殊化"の2つを指す．

プログラム開発を行う時，以前の作成した，あるいは，他のプログラマが開発したプログラムをそのまま利用することは実際には困難なことが多い．そこで次のことが要求されてくる．

1) プログラムに対する明解で統一された形式的仕様
2) プログラムの入出力，さらにはそのプログラムの構造に対する形式的な記法
3) 2) に基づいて，プログラムを一般的な形にして蓄えるためのシステマチックな方法
4) 3) により一般化されたプログラムを必要に応じて特殊化して開発者が望んでいるようなプログラムに作り直すためのシステマチックな方法
5) 4) を支援する機能として，開発者が望む機能仕様・入出力仕様・構造を持った開発済みプログラムに対する柔軟な検索機能

これらの要求すなわちパラメータ化機能と柔軟な検索機能を満たすものをモジュール知識ベースと呼ぶ．モジュール知識ベースを構築するためには知識ベース技術の他に，プログラムの型付けが不可欠である．この方面では，2.3.3に述べた萩谷の方法が参考になりそうだが，型をオブジェクトとして扱うことを許していないQJ で同様の手法が展開できるかどうかはこれからの課題である．

3．3．6 ICOT-QJ

ICOT-QJは，現在わかっている理論をもとに構成的証明に基づくプログラミング環境のプロトタイプをインプリメントするプロジェクトである．これは次の問題意識によるものである．

1) 構成的証明によるプログラミングの実際的な経験が十分でない．
2) 定理証明技術がプログラミング・システムにどこまで応用できるかを確認したい

ICOT-QJは2.2に述べたTyped Logical Calculusをベースにしたプログラミング支援システムである．その具体的内容については'5. システム全体像' のところで紹介する．

4 ICOTの他のプロジェクトとの関連

4．1 JSL

4．1．1 大規模プログラミングの問題

構成的証明に基づくプログラミング支援システムをより大規模なプログラムに対応できるためにパラメータ化とモジュール知識ベースの技術やモジュラー・プログラミングの手法や抽象データ型の記述の導入などが必要である．

一方，ICOTで研究・開発が進められているJSL(日本語仕様記述言語)([田中 他 86])は，大規模プログラミングの観点から構成的証明に基づくプログラミング支援システムをサポートするものとして捉えられる．

一般に大規模なプログラムを開発する場合，形式的仕様以前の曖昧で正確さを欠く非形式的仕様から考え初め，それを段階的に詳細化してゆく．その中でモジュール分割が行われ，各モジュールの機能が具体化していく．ICOT-QJから見たJSLはプログラマのこのような段階での作業をサポートするものとして捉えられる．すなわち，プログラマが問題に直面してからそれを形式的仕様にまで落とす作業までを JSLがサポートする．

4．1．2 JSLとのインタフェース

JSLとICOT-QJとの間のインタフェースは，QJにおける形式的仕様あるいはそれに基づいたものを想定している．このICOT-QJの形式的仕様記述は一般に小規模なプログラムあるいは，大きなトップ・レベル仕様から分割された仕様(モジュール仕様/サブルーチン仕様)を記述するに使われる． 従って，分割された仕様から ICOT-QJ によって生成されたプログラムを繋ぎあわせて初めに与えられた問題を解くプログラムを作り出す機能がJSLには必要である．

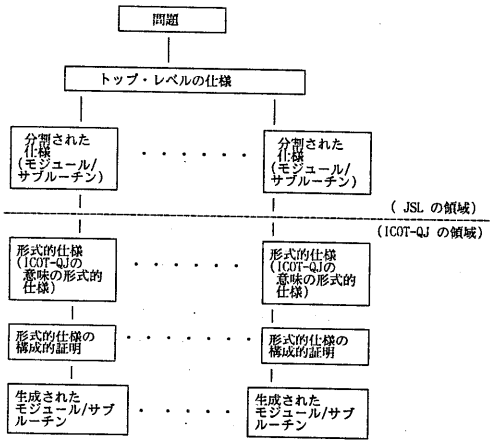

図-3::JSLとICOT-QJ

4．2 CAP-LA と TRS

CAP-LAは現在，線形代数に限らない一般性を持った証明検証系を目指し

1) 現バージョンCAP-LA(PSI上に実装済み)の方式評価
2) 証明記述言語PDL(Proof Description Language)([筧 他 86])の改良
3) TRS(項書換えシステム)機能の強化

などの研究活動が行われている．

CAP-LAは数学のみを対象に考えているが，原理的にはICOT-QJの証明検証系，証明記述言語系と同じであることが予想される．従って，CAP-LAの成果が直接的に ICOT-QJに応用できることが期待され，設計もCAP-LAシステムを参考にして行われつつある．

5．システム全体像

ICOT-QJのシステム・イメージを簡単に紹介する．ICOT-QJはSIMPOS/PSI上にインプリメントすることを想定している．

1) 証明検証系

・証明記述言語によって記述された構成的証明の正当性をチェックする．方式としては， CAP-LAと同様に([永田 他 86])証明木に対するbackward reasoningと項書換えシステムを使ったものを想定．QJではプログラムの動きすべてが論理的演繹として扱われるが，そのなかでも ∧,∨ ,→,¬,∀,∃などの論理定数に関する規則はbackward reasoning で行い，プログラムの形式的変換に相当する部分は項書換えシステムで行うことを考えている．

・エラー箇所の指摘，エラー理由の表示機能，エディタと直結した会話的なインタフェース

・簡単な事実でもQJによる完全な証明となるとかなり繁雑なものになる．従ってより人間の通常の証明記述(計算機から見てかなりギャップのあるもの)が許される機能が必要である．証明補完機能の実現は項書換えシステムを使って行うことを想定している．

・補題の参照ができる機能はモジュール知識ベースと直結させて実現する．

・証明検証系は一般に対象言語(この場合ならQuty)自身で記述するほうがシステムの拡張性に富み，証明検証系の研究材料として非常に望ましいが， PSI上の実装ということを考慮して ESPで記述する．

2) realizability interpreter

・検証済みの構成的証明の証明図を解析してrealizerを生成する．方式としては，Typed Logical Calculusの"Soundness of realizability interpre tation Theorem"の証明に示されている手法を使う．エディタから呼び出して動かせるものを想定している．

3) 対象言語および言語処理系

・Qutyあるいはそれに基づくもの．これはESPのサブセットの部分(pure-PROLOG)を取り出してそれを拡張・強化したものという位置付けで捉えている

・処理系はESP上のインタプリタ．当面は，realizerとして証明から生成される部分(Qutyのサブセット)を中心に考える．デバッガは原理的には不要だが，このシステムの開発・保守ツールとしては有るほうがよい．

4) 仕様記述言語および証明記述言語

・QJまたはそれに基づくもの．項の表現能力が強化された一階述語論理．CAP-LAで開発された証明記述言語PDLをベースに設計することを想定している．

5) 仕様/証明記述用エディタ

・CAP-LAシステムと同じく，システム・コントローラと直結して，エディター内部から検証系 /realizability interpreter，モジュール検索・一般化支援系などが呼び出せるもの．

・強力な項書換え機能をもち，簡単の論理演算や機械的な変換はエディタからのコマンド操作によりその場で実行できるようなもの．

・SIMPOSのPMACS-エディタに専用コマンドを追加することにより実装する．

6) 最適化系

・realizerとして導出されたプログラムを等価変換によって最適化する．但し，変換後のプログラムは，realizerとして導出可能なものの範囲に限定することにより，構成的証明とプログラムの対応関係を保つ．

・証明図の正規化による証明レベルの最適化．(lemma を参照している部分をlemma の証明で置き換えた上で証明全体を正規化する)

・最適化は実用化システムとして磨き上げる段階での課題とし当面は基礎検討だけにとどめる．

7) モジュール知識ベースとモジュール再利用支援系

・型付けされたプログラム・モジュール,仕様，変換・証明情報を格納する．

・モジュール名，型による検索

・世代管理，一貫性管理，クロス・リファレンス，変換過程のログの管理

・型付けに基づいて一般化（パラメータ化）されたプログラム・モジュールの格納

・型の階層構造による類似検索

・プログラマはモジュールを直接見ることは無く，補題とその証明の形で参照する．したがって，モジュールのパラメータ化と補題との対応付けが必要

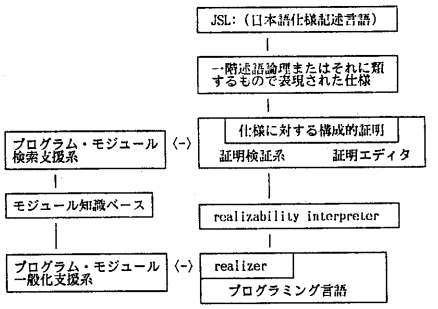

図-4 ::ICOT-QJの全体像

6 むすび

構成的数学とプログラミングの関連が理論家の間で盛んに論じられるようになって10数年が経ったが，実際にインプリメントされたシステムはそれほど多くない．それゆえ"証明 = プログラム"のパラダイムでのプログラミングの研究とくに実験科学的研究は盛んであるとは言い難いと思われる．こうしたなかで，日本でも実働システムを設計し一日も早く実験用のプログラミング環境が整備されることが望まれる．

ICOT-QJ自体はまだ設計の初期段階である．設計の詳細が決り次第稿を改めて報告する予定である．

〈謝辞〉

本研究の機会を与えて下さった，(財)新世代コンピュータ技術開発機構の淵所長および横井俊夫第2研究室長に心から感謝致します．また，日頃指導下さる早稲田大学渕瀬健教授，東北大学佐藤雅彦教授をはじめとするICOT CAPワーキング・グループの皆様に感謝致します．

7. 文献

〔Beeson 85〕 Beeson,M.,"Foundations of Constructive Mathematics", Springer 1985.

〔萩谷 85〕萩谷,"構成的型理論における一般化について",日本ソフトウェア科学会第2回大会論文集,pp189-192, 1985.

〔萩谷 86〕萩谷,"構成的型理論における一般化について",理化学研究所シンポジウム - 関数型プログラミング - 論文集, pp61-67, 1986.

〔廣瀬 他 86〕廣瀬, 横井, 横田, 坂井, 玉井,"CAP プロジェクト(1) ねらいと構想", 情報処理学会第32回全国大会予稿集,1986.

〔筧 他 86〕筧, 坂井, 西山,"CAP プロジェクト(3) 証明記述言語", 情報処理学会第 32回全国大会予稿集, 1986.

〔Martin-Löf 82〕 Martin-Lof,P.,"Constuctive mathematics and computer programming", in 'Logic, Methodology, and Philosophy of Science VI(ed: Chohen,L.J.et al)" pp153-179, North-Holland,Amstermam, 1982.

〔Martin-Löf 84〕Martin-Lof,P.,"Intuitionistic Type Theory", Biblioplis, Napoli, 1984.

〔永田 他 86〕永田, 高山, 西山,"CAP プロジェクト(4) 証明の検証",情報処理学会第32回全国大会予稿集, 1986.

〔Sato 85〕Sato,M.,"Typed Logical Calculus (Provisional version)",Proceedings of the 10th.IBM Symposium of Mathematical Information Science ,1985.

〔Sato & Sakurai 84〕Sato,M. and Sakurai,T.,"Qute:A Functional Language based on Unification", Proc. of the International Conference on Fifth Generation Computer Systems", pp157-165, 1984.

〔Smith 82〕Smith,J.,"The identification of propositions and types in Martin- Löf's type theory: a programming example", Lecture Notes in Comp. Science 158, Springer, 1982.

〔田中 他 86〕田中,滝塚,横井,"JSL:日本語仕様記述言語", ICOTテクニカル・メモ ,(to appear 1986)