テキスト処理言語における文字列のための正規表現型

○ 田渕直† 住井英二郎† 米澤明憲†

Perl，Ruby，Python 等に代表されるスクリプト言語は，CGI のような文字列操作が中心となるアプリケーションの開発に広く使用されており，また開発効率を高めるために有用であることが経験的に知られている．しかしながら，これらの言語はその柔軟すぎる記述力のため，プログラムの挙動が実際の実行時までわからない事が多く，デバッグ・保守が困難になりがちであるという問題を抱えている．

一方，プログラムの性質を静的に検証するという目的では，型システムを利用することが有効であるが，Java・ML など既存の言語の型システムは，スクリプト言語のプログラミングスタイルと必ずしも相性が良くないという問題がある．

本研究では，これらのスクリプト言語のプログラミングスタイルとして「正規表現を用いた文字列操作」が多用されるという点に着目し，正規表現を一種の「型」とみなして文字列に対する型付けを行う型システムを提案する．我々の手法は，文字列の内容や，文字列操作の結果を静的に検査し，プログラムの信頼性を向上させることができる．また，正規表現を用いることにより，古典的な型システムでは達成できない記述力と柔軟性を得ることができる．

Regular Expression Types for Strings in a Text Processing Language

○ Naoshi Tabuchi,† Eijiro Sumii† and Akinori Yonezawa†

Scripting language such as Perl, Ruby, Python, etc. are now widely used in developing string-manipulating software including CGIs. Although they are known to be useful for rapid application development, their excessive flexibility often makes it difficult to debug and maintain programs.

On the other hand, type systems are known to be a good tool for static verification of programs. However, existing type systems as in Java, ML, etc. are not necessarily compatible with scripting languages.

To overcome this problem, we propose a new type system in which "types" consist of regular expressions, observing that an important feature of these scripting languages is to manipulate string values by means of regular expressions. Our method enables programmers to statically verify string values and operations on them, and thereby improve reliability of software. Using regular expressions, our approach achieves higher degree of expressiveness and flexibility than traditional type systems.

1. はじめに

背景

Perl[13]，Ruby[9),12)]，Python[8)] などのスクリプト言語は，いわゆる RAD（Rapid Application Development）の為の強力なツールとして，特に web プログラミングのような文字列・テキスト処理が中心となる開発ドメインにおいて広く受け入れられている．しかしながら，確かにこれらの言語ではプログラムを書くことそのものは簡単になるものの，一度書いたプログラムのデバッグ・保守という観点から見ると，いくつかの問題点を抱えている．スクリプト言語を使った開発の典型的な工程では（1）スクリプト言語の記述力を活かした高速なコーディングの後（2）プログラムを実行させると実行時エラーに遭遇し（運が悪ければ何の出力も得られないこともある）（3）苦痛の伴うデバッグ作業に多大な時間を費やす，というステップを辿ることになる．さらに悪いことには，テストの段階で見落とされた境界条件が原因となってプログラムの運用開始後に始めてバグが発見されるというケースも多い．Web のようなオープンな環境において，この種のバグは致命的なセキュリティホールとなり得る．よく知られた例では，HTML のメタ文字をエスケープ

† 東京大学情報理工学系研究所コンピュータ科学専攻
Department of Computer Science, University of Tokyo, Graduate School of Infomation Science and Technology

し忘れることによる CGI プログラムのクロスサイトスクリプティング問題[1] などを挙げることができる．

一方，プログラムの実行時エラーを防止するための有効なアプローチとしては静的型システムによる型付けがある．しかし，ML に見られるような標準的な型システムは必ずしも上述の問題に対する有効な解決策とはならない．なぜなら，これらの型システムはルールが厳格にすぎるため古いプログラム（あるいは古いプログラマ）との相性が必ずしも良くないからである．さらに標準的な型システムでは，テキストデータは単に `string` という型のみが与えられ，それ以上の情報を得ることができないという点も重要である．入力のテキストを解析して，より構造化されたデータの形にして操作するという方法ももちろん考えられるが，それではスクリプト言語の簡便さというメリットを損なってしまう．

我々のアプローチ

以上のような手軽さと頑健さのジレンマを克服する手段として，我々は文字列の型を正規表現として表した文字列のための正規表現型を提案する．例えば，`a`，`aa`，`aaa` といった文字列定数は皆 $\mathtt{a}^*$ という型を付けることができる．他にも空文字列（長さ 0 の文字列）を ε，文字列の連結操作を ^ と書くことにすると，$f(n) = \mathsf{if}\ n \leq 0\ \mathsf{then}\ \varepsilon\ \mathsf{else}\ \mathtt{a}\ \hat{}\ f(n-1)\ \hat{}\ \mathtt{b}$ という再帰関数には $\mathsf{int} \to \mathtt{a}^*\mathtt{b}^*$ という型を付けることができるだろう．プログラムがより複雑になってくると，このような文字列の形式に関する情報はデバッグ・保守の有用な手掛かりとなることが期待される．

ここで，なぜ文脈自由文法など他のクラスの形式言語ではなく，正規表現を採用するのかという疑問が生じるかもしれない．正規表現の代わりに “文脈自由文法型” なるものを採用すれば，上の例での関数 f の値域は $\{\mathtt{a}^n\mathtt{b}^n \mid n = 0, 1, 2, \ldots\}$ のようにより正確に表すことができる．我々が正規表現を選んだ一つの理由は，その構文・意味論的な基礎が確立されており，しかもプログラマにとって馴染みの深いものだからである．より重要な理由として，正規表現が交差・和・差・商など多くの基本的な演算に関して閉じているということがある．そして，等価性・包含など最も基本的な判定問題に関しても正規表現ならば決定可能である．これらの特質は我々の型システムにおいて，効率的な型チェックの実現のために不可欠である．他のクラスの形式言語はこれほど扱い易い特徴を持っていない．例えば広く知られているように文脈自由言語の等価性（それゆえ包含も）の判定は決定不能である[4]．

我々の貢献

上述のアイデアを形式的，かつ簡易に示すための必要最小限の計算体系として我々は言語 λ^{re}，及びその上での文字列の正規表現型を定義する．λ^{re} は極めて小さな体系であるが，正規表現の持つ柔軟性により，得られる結果の多くはより現実的な言語に拡張できると期待される．例えば，文字列に評価されるあらゆる式は，最悪でも $.^*$（“全ての文字列”）として型付けできるし，多くの場合にはもっと詳細にできるだろう．

データの形やプログラムの挙動を表現するために正規表現，あるいは他の形式言語を用いるのは全く新しいアイデアではない．我々の知る限り最も関連の深い研究として，XML 文書処理のための言語である XDuce[5),6)] がある．XDuce では XML 文書をラベル付きの木とみなして，それに対する正規表現型を与えている．理論的にみれば文字列は木の特別な場合であるから（Scheme でのリストの cons エンコーディングを考えればよい），XDuce は本論文の内容を包括していると考えられるかもしれない．しかしながら，本論文は以下の点で独立の価値があると我々は考える．(1) 一般的な木構造ではなく文字列に焦点を当てているため，技術的な詳細が簡略化され，問題点を明確にできる (2) 我々の研究は XDuce の結果を一部，拡張している．詳細は後の章に譲るが，主要な拡張は (i) 副作用の正規表現による型付けと (ii) 末尾以外の場所での as パターンに対する型推論である．

本論文では型推論の完全なアルゴリズムや効率的な実装については言及していない（ただし 4 章でパターンマッチの変数束縛に関する部分的な型推論を，6 章で完全な型推論に向けての大まかなアイデアを紹介する）．プログラマが大きな負担を払わずに型システムの恩恵を受けるために，型推論は重要な要素ではあるが，それ自身で一つの大きなテーマとなる問題であるため，今回は将来の課題とした．

本論文の以降の構成は次の通りである．2 章で λ^{re} の構文を，続いて 3 章で操作的意味論を定義する．4 章で型システムを定義し，5 章では λ^{re} によるプログラムの例を紹介する．6 章で関連研究と今後の課題についての考察を与え，結論とする．

尚，紙幅の都合により証明の詳細な部分については省略した．これに関しては WWW で `http://www.yl.is.s.u-tokyo.ac.jp/~tabee/xperl/` から参照することができる．

2. λ^{re} の構文

λ^{re} の項の構文は図 1 で定義される．x，y，z，f，g，

M	(term)	::=	x	(variable)
		|	$\mathsf{fix}(f, x, M)$	(recursive function)
		|	$M_1 M_2$	(function application)
		|	s	(constant string)
		|	$M_1 \hat{} M_2$	(string concatenation)
		|	match M with $P_1 \Rightarrow M_1 \mid \ldots \mid P_n \Rightarrow M_n$	(pattern matching)
		|	print M	(output)
P	(pattern)	::=	s	(constant string)
		|	$P_1 \mid P_2$	(choice)
		|	x as P	(variable binding)
		|	P^*	(repetition)
		|	$P_1 P_2$	(sequence)

図 1 項とパターンの構文
Fig. 1 Syntax of Terms and Patterns.

h 等は変数名であり，これらは加算無限個存在するとする．$\mathsf{fix}(f, x, M)$ は $f(x) = M$ で定義される再帰関数 f を表すものとし，これを含めて最初の 3 つの要素は標準的なものである．$\lambda x.\, M$ を f が M の自由変数に表れない場合の $\mathsf{fix}(f, x, M)$ の略記，$\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2$ を $(\lambda x.\, M_2)M_1$ の略記と定める．ただし，自由変数の定義についてはやはり標準的なものに従うこととし，ここでは省略する．

文字列を操作するためのプリミティブとして，以下の 4 つの要素を導入する．(1) 文字列そのものを表す文字列定数 s．(2) 2 つの文字列を連結して新しい文字列を作る $M_1 \hat{} M_2$．(3) 文字列に対して正規表現のパターンマッチを行う match M with $P_1 \Rightarrow M_1 \mid \ldots \mid P_n \Rightarrow M_n$．(4) そして文字列を出力して副作用を引き起こす print M である．4 章で見るように，これが我々の正規表現 effect system の対象となる．

パターンの構文はほぼ標準的な正規表現に準ずるが，変数束縛のための x as P という構文を導入して拡張している．これは ML における as パターンと同様に，最初に入力文字列 s を パターン P にマッチさせ，成功すれば x に s を束縛する．以降，P に現れる全ての変数の集合を表すのに $var(P)$ と書くこととする．

3. λ^{re} の操作的意味論

λ^{re} の項の意味は図 **2** にある通り，small-step の簡約意味論として定義される．Big-step ではなく small-step の意味論を用いるのは，発散するプログラムについても副作用に関しては停止するプログラムと同等に扱えるようにするためである．意味論は $M_1 \xrightarrow{s} M_2$ の形の関係で表され，これは “項 M_1 は 項 M_2 に簡約され，その際文字列 s を出力する” と読むことができる．副作用の扱いを簡単にするため，評価順序は call-by-value かつ left-to-right で固定する．

規則 (R-App)，(R-Cat) 及び (R-Ctx) はそれぞれ標準的な (再帰的かもしれない) 関数適用，文字列結合，および評価コンテキストを表している．(R-Print) は print s という項を評価すると文字列 s を出力し，ダミーの値 (空文字列 ε) に簡約されることを表している．(R-Match) では λ^{re} におけるパターンマッチの意味を，$s \triangleright P \Rightarrow \theta$ という関係を使って定義している．直観的には $s \triangleright P \Rightarrow \theta$ という関係は，文字列 s をパターン P にマッチさせると，失敗するか ($\theta = \bot$ の場合) 変数から文字列への置換 θ を生じる ($\theta \neq \bot$ の場合) ことを意味する．すなわち，(R-Match) は match s with $P_1 \Rightarrow M_1 \mid \ldots \mid P_n \Rightarrow M_n$ という項が θM_m に簡約され，ただし P_m は文字列 s にマッチする (そして置換 θ を生む) 最初のパターンであることを言っている．

パターンマッチの関係 $s \triangleright P \Rightarrow \theta$ は図 **3** で定義されている．以降，$s \triangleright P \Rightarrow \theta$ なる θ $(\neq \bot)$ が存在する時単に $s \triangleright P$ と書くこととする．直観的には，図 3 の規則はパターンマッチの (素朴な) アルゴリズムをボトムアップの形で記述したものに相当し，基本的にはパターン P の構造に関して帰納的に定義されている．(M-Const-Succ) と (M-Const-Fail) はパターンが文字列定数 s の場合の規則，(M-Choice-Fst) と (M-Choice-Snd) はパターンが $P_1 \mid P_2$ という選択の形の場合の規則，(M-Bind) は変数束縛 x as P の為の規則，そして (M-Rep) はパターンが繰り返し P^* の場合の規則である．(M-Bind) において $\theta \uplus \{x \mapsto s\}$

$$
\begin{array}{rlcl}
v & \text{(value)} & ::= & \mathsf{fix}(f,x,M) \\
 & & \mid & s \\
C[\,] & \text{(evaluation context)} & ::= & [\,]M \\
 & & \mid & v[\,] \\
 & & \mid & [\,]\hat{}\,M \\
 & & \mid & v\,\hat{}\,[\,] \\
 & & \mid & \mathsf{match}\ [\,]\ \mathsf{with}\ P_1 \Rightarrow M_1 \mid \ldots \mid P_n \Rightarrow M_n \\
 & & \mid & \mathsf{print}\ [\,]
\end{array}
$$

$$\frac{}{\mathsf{fix}(f,x,M)v \xrightarrow{\varepsilon} [v/x][\mathsf{fix}(f,x,M)/f]M}(\text{R-App}) \qquad \frac{}{s_1\,\hat{}\,s_2 \xrightarrow{\varepsilon} s_1 + s_2}(\text{R-Cat})$$

$$\frac{}{\mathsf{print}\ s \xrightarrow{s} \varepsilon}(\text{R-Print}) \qquad \frac{\forall i < m.\, s \rhd P_i \Rightarrow \bot \quad s \rhd P_m \Rightarrow \theta \quad \theta \neq \bot}{\mathsf{match}\ s\ \mathsf{with}\ P_1 \Rightarrow M_1 \mid \ldots \mid P_n \Rightarrow M_n \xrightarrow{\varepsilon} \theta M_m}(\text{R-Match})$$

$$\frac{M_1 \xrightarrow{s} M_2}{C[M_1] \xrightarrow{s} C[M_2]}(\text{R-Ctx})$$

図 2 項の操作的意味
Fig. 2 Operational Semantics of Terms.

$$\frac{}{s \rhd s \Rightarrow \emptyset}(\text{M-Const-Succ}) \qquad \frac{s \neq s'}{s \rhd s' \Rightarrow \bot}(\text{M-Const-Fail})$$

$$\frac{s \rhd P_1 \Rightarrow \theta \quad \theta \neq \bot}{s \rhd P_1 \mid P_2 \Rightarrow \theta}(\text{M-Choice-Fst}) \qquad \frac{s \rhd P_1 \Rightarrow \bot \quad s \rhd P_2 \Rightarrow \theta}{s \rhd P_1 \mid P_2 \Rightarrow \theta}(\text{M-Choice-Snd})$$

$$\frac{s \rhd P \Rightarrow \theta}{s \rhd x\ \mathsf{as}\ P \Rightarrow \theta \uplus \{x \mapsto s\}}(\text{M-Bind}) \qquad \frac{s \rhd PP^* \mid \varepsilon \Rightarrow \theta}{s \rhd P^* \Rightarrow \theta}(\text{M-Rep})$$

$$\frac{s_2^{-1}s_1 \rhd P \Rightarrow \theta}{s_1 \rhd s_2 P \Rightarrow \theta}(\text{M-Seq-Const-Succ}) \qquad \frac{s_2^{-1}s_1 \text{ not exist}}{s_1 \rhd s_2 P \Rightarrow \bot}(\text{M-Seq-Const-Fail})$$

$$\frac{s \rhd P_1P_3 \mid P_2P_3 \Rightarrow \theta}{s \rhd (P_1 \mid P_2)P_3 \Rightarrow \theta}(\text{M-Seq-Choice})$$

$$\frac{y \notin var(P_1P_2) \quad s_1 \rhd P_1(y\ \mathsf{as}\ P_2) \Rightarrow \theta \uplus \{y \mapsto s_2\}}{s_1 \rhd (x\ \mathsf{as}\ P_1)P_2 \Rightarrow \theta \uplus \{x \mapsto s_1 s_2^{-1}\}}(\text{M-Seq-Bind})$$

$$\frac{s \rhd (P_1P_1^* \mid \varepsilon)P_2 \Rightarrow \theta}{s \rhd P_1^*P_2 \Rightarrow \theta}(\text{M-Seq-Rep}) \qquad \frac{s \rhd P_1(P_2P_3) \Rightarrow \theta}{s \rhd (P_1P_2)P_3 \Rightarrow \theta}(\text{M-Seq-Seq})$$

図 3 パターンの操作的意味
Fig. 3 Operational Semantics of Patterns.

は $\theta'(x) = s$, $\theta'(y) = \theta(y)$ $(y \neq x)$ を満たす θ' を表すこととする (ただし $\bot \uplus \{x \mapsto s\} = \bot$ と定める) . また , $\emptyset$ は恒等置換を表すとする .

注意すべき点として , (M-Choise-Snd) 規則は (M-Choice-Fst) 規則が適用されなかった場合にのみ適用される . これによって , いわゆる *first-match* を実現できる . 同様に (M-Rep) 規則は P^* を $\varepsilon \mid PP^*$ ではなく $PP^* \mid \varepsilon$ に展開する . これと *first-match* の原則

からいわゆる *longest-match* を実現できる．

パターンが P_1P_2 という連接の形をしている場合，(M-Seq-...) 規則に見られるように P_1 の形に従ってさらに 6 つの場合に分ける．入力文字列 s を，単に s_1 が P_1 にマッチし，s_2 が P_2 にマッチする，という分割をしてしまうとパターンマッチに非決定性の挙動が含まれてしまうためである．6 つの規則はそれぞれ P_1P_2 を，導出の前件部で P_1 の形がより “小さく” なるように P_1 の意味に従って書き換える（ただし繰り返しの場合を除く）．(M-Seq-Const-...) と (M-Seq-Bind) 規則に見られる $s_1^{-1}s_2$，$s_1s_2^{-1}$ という表記は，$s_1s = s_2$ 或いは $s_1 = ss_2$ を満たすような文字列 s を（もしあれば）表す記法とする．

例 1 文字列 a をパターン $(x \text{ as } \mathsf{a}^*)\mathsf{a}^*$ にマッチさせると置換 $x \mapsto \mathsf{a}$ を生じる．この導出の例を図 4 に示す．first-match，longest-match のポリシーがどのように働いているかに注意されたい．

ところで，このパターンマッチの導出が “無限ループ” に陥って停止しなくなってしまうケースがある．空文字列の繰り返しがあるパターン，すなわちパターンが P^* で P が ε にマッチするような場合がそれに相当する．もう少し厳密な言い方をすれば，上述の（帰納的に定義された）規則で，例えば $\varepsilon \triangleright \varepsilon^* \Rightarrow \emptyset$ のような導出をすることはできない．幸いにも，このような繰り返しは前処理によってパターンの意味を変えることなく排除することができる．この技法については既に研究がなされている．例えば文献[3] の 4 章などを参照されたい．よって本論文では以降，このようなパターンは出現しないことを仮定する．

注意深い読者は，$s \triangleright P \Rightarrow \bot$ を単に “$s \triangleright P \Rightarrow \theta$ を満たす θ が存在しない” ことで定義しない事に疑問を持たれるかもしれない．しかしながらこの定義では停止しないこととパターンマッチに失敗することが区別できないという問題がある．のみならず，このやり方ではパターンマッチの関係の定義そのものが実は成立しない．というのも (M-Choice-Snd) 規則が前提の $s \triangleright P_1 \Rightarrow \bot$ のために帰納的な定義でなくなってしまうからである☆．

4. 型システム

λ^{re} の型 τ は図 5 の通りに定義される．関数型 $\tau_1 \xrightarrow{T} \tau_2$ は τ_1 型の引数を取り，T 型の文字列を出力し，返り値があればその型は τ_2 となるような関数を表す．文字列型 T は $[\![T]\!]$ と表記される文字列の集合を表す．これは文字列定数 s，選択 $T_1 \mid T_2$，繰り返し T^*，連接 T_1T_2 のような正規表現であるか，交差 $T_1 \cap T_2$，差分 $T_1 \cap \overline{T_2}$，left quotient $T_1^{-1}T_2$，right quotient $T_1T_2^{-1}$ のような正規表現に対する演算の形をとる．正則言語の集合はこれらの演算に関して閉じていることが知られている[4]．しかしながら，このような演算は型の構文として定義する方が，型付け規則を与える上で有用である．ところで，型の構文は変数の束縛を除いてパターンの構文を含んでいる．そこで以降，パターン P から全ての変数を除去して得られる型を $novar(P)$ と書くことにする．すなわち，$novar(s) = s$，$novar(P_1 \mid P_2) = novar(P_1) \mid novar(P_2)$，$novar(x \text{ as } P) = novar(P)$，$novar(P^*) = novar(P)^*$，$novar(P_1P_2) = novar(P_1)novar(P_2)$ と定める．さらに，型の交差 $\cap$ と 選択 $\mid$ の定義を文字列型 T から一般の型 τ に拡張する．関数型 $\tau_1 \xrightarrow{T} \tau_2$ についてはそれぞれの演算を以下のように定義する．

$$(\tau_1 \xrightarrow{T_1} \tau_1') \mid (\tau_2 \xrightarrow{T_2} \tau_2') = (\tau_1 \cap \tau_2) \xrightarrow{T_1 \mid T_2} (\tau_1' \mid \tau_2')$$
$$(\tau_1 \xrightarrow{T_1} \tau_1') \cap (\tau_2 \xrightarrow{T_2} \tau_2') = (\tau_1 \mid \tau_2) \xrightarrow{T_1 \cap T_2} (\tau_1' \cap \tau_2')$$

このような定義の変わりに，$\tau_1 \cap \tau_2$ や $\tau_1 \mid \tau_2$ も型の構文として定義し，本物の intersection 型（つまりオーバーロード），union 型を考えることもできる．同様のアイデアが参考文献 2) で研究されており，この成果を本研究に取り入れることも可能ではあるが，ここでは簡潔さのためその方法は取らない．

型付け規則は図 6 で与えられる．規則 (S-Str) にあるように，文字列型 T_1 と T_2 の間の部分型関係 $T_1 \leq T_2$ は，型の表示の包含関係として定義する．型の表示は正則集合なのでその包含関係は決定可能である（ただし，効率的な実装に関しては本論分の範囲を超える）．文字列型の部分型関係は一般の型の間に $\tau_1 \leq \tau_2$ として拡張される．関数型に関しては (S-Fun) 規則にあるように標準的な部分型の定義を用いる．

型判定 $\Gamma \vdash M : \tau, T$ は “型環境 Γ の下で項 M は型 τ と副作用 T を持つ” と読むことができる．型付け規則は副作用とパターンマッチの部分を除いて標準的なものである．副作用に関しては，例えば (T-Print) 規則を考えると print M という項は最初に M を評価して結果の文字列 s を出力し，空文字列 ε に評価されるので，M が文字列型 T と副作用 T' を持つならば print M は文字列型 ε と副作用 $T'T$ で型付けされる．副作用の型付けは他の規則でも同様の推論に基づいている．

パターンマッチに関しては，(T-Match) 規則を参

☆ 実のところ XDuce の論文[5] のパターンマッチの定義はこの問題を抱えている．

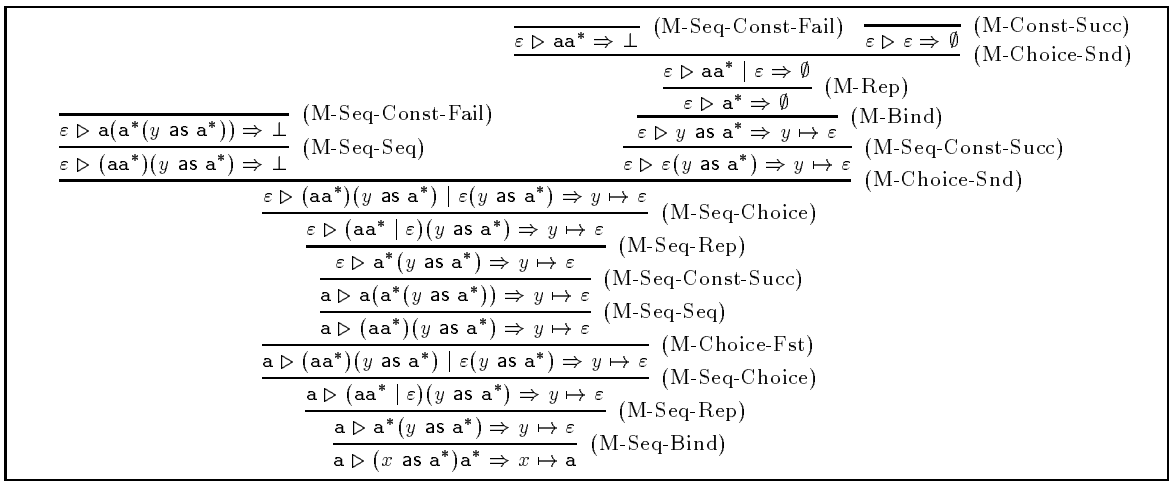


図 4 パターンマッチの例
Fig. 4 Example of Pattern Matching.

τ	(type)	::=	$\tau_1 \xrightarrow{T} \tau_2$			
		|	T			
T	(string type)	::=	s	$[\![s]\!]$	=	$\{s\}$
		|	$T_1 \mid T_2$	$[\![T_1 \mid T_2]\!]$	=	$[\![T_1]\!] \cup [\![T_2]\!]$
		|	T^*	$[\![T^*]\!]$	=	$\{s_1 + \cdots + s_n \mid s_i \in [\![T]\!]\ (i = 1, \ldots, n)\}$
		|	$T_1 T_2$	$[\![T_1 T_2]\!]$	=	$\{s_1 + s_2 \mid s_1 \in [\![T_1]\!] \wedge s_2 \in [\![T_2]\!]\}$
		|	$T_1 \cap T_2$	$[\![T_1 \cap T_2]\!]$	=	$[\![T_1]\!] \cap [\![T_2]\!]$
		|	$T_1 \cap \overline{T_2}$	$[\![T_1 \cap \overline{T_2}]\!]$	=	$[\![T_1]\!] \setminus [\![T_2]\!]$
		|	$T_1^{-1} T_2$	$[\![T_1^{-1} T_2]\!]$	=	$\{s' \mid s \in [\![T_1]\!] \wedge s + s' \in [\![T_2]\!]\}$
		|	$T_1 T_2^{-1}$	$[\![T_1 T_2^{-1}]\!]$	=	$\{s \mid s + s' \in [\![T_1]\!] \wedge s' \in [\![T_2]\!]\}$

図 5 型の構文と意味
Fig. 5 Syntax and Semantics of Types.

照されたい．まず，入力 M の型は文字列型 T_1 でなければならない．T_1 型の入力文字列に対し，最初のパターン P_1 とのマッチを試みる．もしこのマッチが成功すれば，何らかの変数束縛が生成される．この時生じた束縛変数の型を計算するため，補助的な関係として $T \rightsquigarrow P \Rightarrow \Gamma$ を導入する．この関係は“もし型 T を持つ文字列がパターン P にマッチしたならば，各束縛変数 x の型は $\Gamma(x)$ に従う”と読むことができる．このようにして，型環境 Γ_1 が T_1 と P_1 から $T_1 \rightsquigarrow P_1 \Rightarrow \Gamma_1$ によって計算され，P_1 に対応する項 M_1 が新しい型環境の下で $\Gamma, \Gamma_1 \vdash M_1 : \tau_1, T_1'$ のように型付けされる．ところで，もし T_1 型のどんな文字列も P_1 にマッチしない，すなわち，T_1 型の文字列で P_1 にマッチするものの集合 $T_1 \cap novar(P_1)$ が空ならば，このパターンマッチが成功することはあり得ないので P_1 は冗長である．このことが型システムの健全性に直接影響を与えることはないが，我々はプログラマの利便のため，この種の冗長性に対して警告を与えることにする．ただし，冗長性を直ちにエラーとみなすことはできない．このようにすると後の subject reduction の性質が成り立たなくなってしまうからである．さらに，我々は first-match のポリシーを採用したので，P_1 にマッチする入力文字列は次のパターン P_2 の入力とはならない．そこで，P_2 に対する入力文字列の型 T_2 は，T_1 と P_1（にマッチする文字列の型）の差分 $T_1 \cap \overline{novar(P_1)}$ とする．このようにして各パターン P_m と対応する項 M_m $(m = 1, \ldots, n)$ に対する型付けが行われる．最後に，パターンマッチが exhaustive，つまり入力がいずれかのパターンに必ずマッチすることを保証するため，入力文字列のうちどのパターンにもマッチしないものの型 T_{n+1} は空でなければならない．ここで空の型（その表示が空集合と

$$\frac{[\![T_1]\!] \subseteq [\![T_2]\!]}{T_1 \leq T_2}(\text{S-Str}) \qquad \frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2' \quad T_1 \leq T_2}{\tau_1 \xrightarrow{T_1} \tau_2 \leq \tau_1' \xrightarrow{T_2} \tau_2'}(\text{S-Fun})$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau, \varepsilon}(\text{T-Var}) \qquad \frac{\Gamma, f : \tau_1 \xrightarrow{T} \tau_2, x : \tau_1 \vdash M : \tau_2, T' \quad T' \leq T}{\Gamma \vdash \mathsf{fix}(f, x, M) : \tau_1 \xrightarrow{T} \tau_2, \varepsilon}(\text{T-Fix})$$

$$\frac{\begin{array}{c}\Gamma \vdash M_1 : \tau_2 \xrightarrow{T} \tau_1, T_1 \\ \Gamma \vdash M_2 : \tau_2', T_2 \quad \tau_2' \leq \tau_2\end{array}}{\Gamma \vdash M_1 M_2 : \tau_1, T_1 T_2 T}(\text{T-App}) \qquad \frac{}{\Gamma \vdash s : s, \varepsilon}(\text{T-Const})$$

$$\frac{\Gamma \vdash M_1 : T_1, T_1' \quad \Gamma \vdash M_2 : T_2, T_2'}{\Gamma \vdash M_1 \hat{} M_2 : T_1 T_2, T_1' T_2'}(\text{T-Cat}) \qquad \frac{\Gamma \vdash M : T, T'}{\Gamma \vdash \mathsf{print}\ M : \varepsilon, T'T}(\text{T-Print})$$

$$\frac{\begin{array}{c}\Gamma \vdash M : T_1, T' \\ T_m \rightsquigarrow P_m \Rightarrow \Gamma_m \quad \text{warn of redundancy if } T_m \cap novar(P_m) \leq \emptyset \\ \Gamma, \Gamma_m \vdash M_m : \tau_m, T_m' \quad T_{m+1} = T_m \cap \overline{novar(P_m)} \quad (m = 1, \ldots, n) \\ T_{n+1} \leq \emptyset\end{array}}{\Gamma \vdash \mathsf{match}\ M\ \mathsf{with}\ P_1 \Rightarrow M_1 \mid \ldots \mid P_n \Rightarrow M_n : \tau_1 \mid \ldots \mid \tau_m, T'(T_1' \mid \ldots \mid T_m')}(\text{T-Match})$$

$$\frac{([\![T]\!], P) \in \Pi \quad \forall x \in dom(\Gamma) = var(P).\, \Gamma(x) = \emptyset}{\Pi \vdash T \rightsquigarrow P \Rightarrow \Gamma}(\text{P-Mem}) \qquad \frac{}{\Pi \vdash T \rightsquigarrow s \Rightarrow \emptyset}(\text{P-Const})$$

$$\frac{\begin{array}{c}\Pi \vdash T \rightsquigarrow P_1 \Rightarrow \Gamma_1 \quad \Pi \vdash T \cap \overline{novar(P_1)} \rightsquigarrow P_2 \Rightarrow \Gamma_2 \\ \forall x \in dom(\Gamma) = dom(\Gamma_1) = dom(\Gamma_2).\, \Gamma(x) = \Gamma_1(x) \mid \Gamma_2(x)\end{array}}{\Pi \vdash T \rightsquigarrow P_1 \mid P_2 \Rightarrow \Gamma}(\text{P-Choice})$$

$$\frac{\Pi \vdash T \rightsquigarrow P \Rightarrow \Gamma}{\Pi \vdash T \rightsquigarrow x\ \mathsf{as}\ P \Rightarrow \Gamma \uplus \{x : \Gamma(P) \cap T\}}(\text{P-Bind}) \qquad \frac{\Pi \uplus \{([\![T]\!], P^*)\} \vdash T \rightsquigarrow PP^* \mid \varepsilon \Rightarrow \Gamma}{\Pi \vdash T \rightsquigarrow P^* \Rightarrow \Gamma}(\text{P-Rep})$$

$$\frac{\Pi \vdash s^{-1}T \rightsquigarrow P \Rightarrow \Gamma}{\Pi \vdash T \rightsquigarrow sP \Rightarrow \Gamma}(\text{P-Seq-Const}) \qquad \frac{\Pi \vdash T \rightsquigarrow P_1P_3 \mid P_2P_3 \Rightarrow \Gamma}{\Pi \vdash T \rightsquigarrow (P_1 \mid P_2)P_3 \Rightarrow \Gamma}(\text{P-Seq-Choice})$$

$$\frac{\forall([\![T]\!], P) \in \Pi.\, y \notin var(P) \quad y \notin var(P_1P_2) \quad \Pi \vdash T_1 \rightsquigarrow P_1(y\ \mathsf{as}\ P_2) \Rightarrow \Gamma \uplus \{y : T_2\}}{\Pi \vdash T_1 \rightsquigarrow (x\ \mathsf{as}\ P_1)P_2 \Rightarrow \Gamma \uplus \{x : \Gamma(P_1) \cap T_1 T_2^{-1}\}}(\text{P-Seq-Bind})$$

$$\frac{\Pi \uplus ([\![T]\!], P_1{}^*P_2) \vdash T \rightsquigarrow (P_1P_1{}^* \mid \varepsilon)P_2 \Rightarrow \Gamma}{\Pi \vdash T \rightsquigarrow P_1{}^*P_2 \Rightarrow \Gamma}(\text{P-Seq-Rep}) \qquad \frac{\Pi \vdash T \rightsquigarrow P_1(P_2P_3) \Rightarrow \Gamma}{\Pi \vdash T \rightsquigarrow (P_1P_2)P_3 \Rightarrow \Gamma}(\text{P-Seq-Seq})$$

図 6 型付け規則
Fig. 6 Typing Rules.

なる型）は型 ε（表示が一要素 ε のみからなる集合の型）と区別するように注意されたい．以後，空の型のことを $\emptyset$ と表記することとする．空の型としてどのような型を具体的にとるかは問題ではないが，例えば $\varepsilon \cap \overline{\varepsilon}$ 等があるだろう．

パターンマッチの型付け $T \rightsquigarrow P \Rightarrow \Gamma$ は操作的意味論 $s \triangleright P \Rightarrow \theta$ と似た形で定義される．ここでも各規則はボトムアップに，パターン P と型 T を取って結果の型環境 Γ を計算するアルゴリズムとみなせる．文字列型は文字列の（正則）集合を表すので，パターンの型付け規則は実際，パターンの操作的意味論に非常に近い（図 3 参照）．しかしながら，個別の文字列

の代わりに文字列の型を取る，という明らかな違いに加えて，大きく 2 つの違いがある．

一つは型推論の停止性を保証するために “memoization” の手法を利用している点である．具体的には，導出の関係を $\Pi \vdash T \rightsquigarrow P \Rightarrow \Gamma$ の形に一般化する．ここで Π は 文字列型 T (の表示) とパターン P のペア $(\llbracket T \rrbracket, P)$ の集合である．$\Pi = \emptyset$ の時単に $T \rightsquigarrow P \Rightarrow \Gamma$ と書くことにする．直観的には，Π は型付けの過程で “既に一度出現した” 繰り返しのパターンと型の組を記憶している．規則 (P-Req)，(P-Seq-Rep) 等を参照されたい．アルゴリズムの途中で再びこのような組に遭遇すると，規則 (P-Mem) にあるように全ての変数に空の型を割り当てた型環境を返す．この規則がないと，繰り返しを含むパターンに関する型付けは常に停止しなくなってしまう．

もう一つの違いは，この型付けそのものはパターンマッチが成功することを保証しない点にある．型付けは単にマッチが成功したならばどのような文字列が変数に束縛されるかを推論するにすぎない．これが最も明確に現れるのは (P-Const) 規則で，例えば $\emptyset \vdash \mathtt{a}^* \rightsquigarrow \mathtt{b} \Rightarrow \emptyset$ のような導出が可能である．このことの意味は “$\mathtt{a}^*$ 型の文字列を定数文字列のパターン b にマッチさせると変数に対する束縛は起こらない” ということであるが，当然このパターンマッチが成功することはあり得ない．このような定義の仕方は明快さを損ねるようにも思えるが，実は必然的なものである．特に $P_1 \mid P_2$ のような選択のパターンで，入力が常に P_1 にマッチすることを (P-Choice) で要求してしまうと，制約が強くなりすぎてしまう．一例として，パターン $.^*$ を match M with $P \Rightarrow M_1 \mid x$ as $.^* \Rightarrow M_2$ のように “デフォルト” として使うことを考えよう (ただし . は全ての文字の選択 $\mathtt{a} \mid \mathtt{b} \mid \ldots$ の省略記法とする)．$.^*$ への入力文字列が任意の形を取り得るわけではないが，このパターンをデフォルトとして用いるのは極めて自然な発想であろう．とはいえ，今の例では P にマッチする文字列は x に束縛されないことは確かなのだから，x の型にはその事実を何らかの形で反映させたい．それを実現するのが (P-Bind) 規則で，ここで入力の型 T と型環境 Γ の元でのパターン P の値域 $\Gamma(P)$ の交差をとって変数の型としている．ただし，ここで $\Gamma(P)$ は $\Gamma(s) = s$，$\Gamma(P_1 \mid P_2) = \Gamma(P_1) \mid \Gamma(P_2)$，$\Gamma(x \text{ as } P) = \Gamma(x) \cap \Gamma(P)$，$\Gamma(P^*) = \Gamma(P)^*$，$\Gamma(P_1 P_2) = \Gamma(P_1)\Gamma(P_2)$ で定義される型である．同じ発想で (P-Seq-Bind) 規則も定義されている☆．尚，型環境に新しい要素を追加する際には $\uplus$ を使って保守的な形でのみ拡張を許している．すなわち，追加しようとしている要素と同じ変数に関する要素がすでに型環境に含まれていた場合，拡張の結果は未定義としているので，パターン中の変数については線形性が要求される．

以降，簡単のため $\llbracket T \rrbracket = \llbracket T' \rrbracket$ の時に型 T を別の型 T' に暗黙に置換えて使用することがある．これによって型付けの結果が影響を受けることはない．

例 2　型 $\mathtt{aa}^*$ の文字列をパターン $(x \text{ as } \mathtt{a}^*)(y \text{ as } \mathtt{a}^*)$ にマッチさせると，x の型は $\mathtt{aa}^*$，y の型は ε となる．すなわち，$\mathtt{aa}^* \rightsquigarrow (x \text{ as } \mathtt{a}^*)(y \text{ as } \mathtt{a}^*) \Rightarrow x : \mathtt{aa}^*, y : \varepsilon$ という導出ができる．この導出の例を図 7 に示す．この型付けが first-match，longest-match のポリシーを反映していることに注意されたい．

上述の型システムの健全性は以下のように証明される．議論の多くは一般的なものであるが，パターンマッチの性質についていくつか言及している．証明の詳細については紙幅の都合上割愛するが，第 1 章で紹介した web サイトに掲載しているので，そちらを参照されたい．

定理 3 (型と副作用の健全性)　$\vdash M : \tau, T$ かつ $M \xrightarrow{s_1} M_1 \xrightarrow{s_2} M_2 \xrightarrow{s_3} \ldots \xrightarrow{s_n} M_n$ ならば，$s_1 s_2 s_3 \ldots s_n$ はある文字列 $s \in \llbracket T \rrbracket$ の prefix である．さらに，M_n がこれ以上簡約できない ($M_n \xrightarrow{s} M'$ なる s，M' が存在しない) ならば，M_n は値でその型は τ の部分型であり (すなわち，ある v，τ' について，$M_n = v$，$\vdash v : \tau', \varepsilon, \tau' \leq \tau$)，同時に $s_1 s_2 s_3 \ldots s_n \in \llbracket T \rrbracket$ が成り立つ．

証明. 下の progress と subject reduction の補題より直ちに導かれる．　□

補題 4 (**Progress**)　$\vdash M : \tau, T$ で M が値でないならば，M は簡約できる (すなわち，$M \xrightarrow{s} M'$ なる s と M' が存在する)．

証明. $\vdash M : \tau, T$ の導出に関する帰納法．ただし (T-Match) の場合に以下の 2 つの補題を使う．　□

補題 5 (パターンマッチの決定性・停止性)　$s \triangleright P \Rightarrow \theta_1$ かつ $s \triangleright P \Rightarrow \theta_2$ ならば $\theta_1 = \theta_2$．さらに，$s \in \llbracket novar(P) \rrbracket$ ならば $s \triangleright P$．逆に $s \notin \llbracket novar(P) \rrbracket$ ならば $s \triangleright P \Rightarrow \bot$．

証明概略. 証明は見掛けよりもやや複雑なものとなる．単純にパターンの構造に関して帰納的に証明する方法では，規則 (M-Rep)，(M-Seq-Rep) の所でパターン

☆ これとは逆に，XDuce[5] では末尾以外の場所に現れる as パターンについて正確な型を与えることができない．Regular tree と正規表現は理論的には近い位置にあるため，我々のアプローチは XDuce にも適用できるかもしれないと考えている．

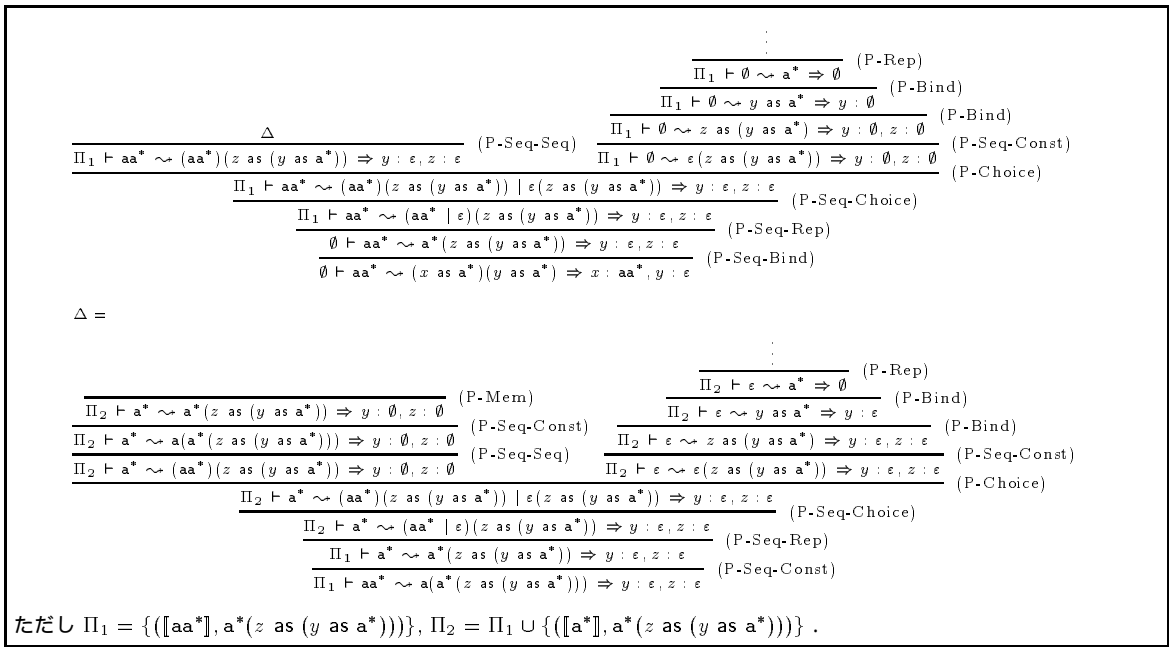


図 7 導出の例
Fig. 7 Example of Derivation.

の構造が結論部分よりも仮定部分において大きくなってしまうためうまくいかない．我々の証明では，まず最初にパターンマッチのもう一つのアルゴリズムを定義し，新しいアルゴリズムの決定性・停止性と元々のアルゴリズムとの互換性を示す，という段階を踏む．新しいアルゴリズムでは first/longest-match を含む全ての可能なマッチの集合が計算される．このアルゴリズムは文献 14) のものに基づいているが，停止性を保証するためにパターンが 3 章で述べたように前処理[3] されている必要がある，という点で文献の物とは多少異なっている．

補題 6 (パターンマッチの **Exhaustiveness**) $\Gamma \vdash \mathsf{match}\ s\ \mathsf{with}\ P_1 \Rightarrow M_1 \mid \ldots \mid P_n \Rightarrow M_n : \tau, T$ ならば，ある $1 \leq i \leq n$ について $s \rhd P_i$ ．

補題 7 (**Subject Reduction**) $\Gamma \vdash M : \tau, T$ かつ $M \xrightarrow{s} M'$ ならば，ある τ', T' について $\Gamma \vdash M' : \tau', T'$ であり，τ' ，T' は $\tau' \leq \tau$ と $sT' \leq T$ を満たす．

証明． $\Gamma \vdash M : \tau, T$ の導出に関する帰納法．(T-App) と (T-Match) の場合に以下の 2 つの補題をそれぞれ使う． □

補題 8 (**Substitution**) $\Gamma, x : \tau' \vdash M : \tau, T$ かつ $\Gamma \vdash v : \tau', \varepsilon$ ならば $\Gamma \vdash [v/x]M : \tau, T$ ．

補題 9 (パターンマッチの型推論の健全性・完全性) $T \rightsquigarrow P \Rightarrow \Gamma$ ならば，各々の $x \in var(P) = dom(\Gamma)$ について $[\![\Gamma(x)]\!] = \{\theta(x) \mid s \in [\![T]\!] \wedge s \rhd P \Rightarrow \theta \wedge \theta \neq \bot\}$ ．

最後の補題は見掛けよりもかなり巧妙な証明を要する (特に $\supseteq$ の部分について) ．例えば，もしパターンに ε^* のような空 (かもしれない) 文字列の繰り返しを認めると，この性質は成り立たなくなる．反例としては $\mathsf{a} \rightsquigarrow \varepsilon^*(y\ \mathsf{as}\ \mathsf{a}) \Rightarrow y : \emptyset$ 等を挙げることができる．このようなケースでは型システムの健全性が破壊される結果を生んでしまう (実際には 3 章で述べたように，この種のパターンは一般性を失わずあらかじめ排除できるので，起こりえないケースである) ．以上のような事情があるため，証明は複雑なものとなる．少し具体的には，Π 関していくつかの補助的な関係を定義した上で，成り立つ不変条件について言及するという方法をとる．

証明概略． まず，なぜ単純な方法での証明では巧くいかないのかを考察する．今，我々が示したい事は “$\emptyset \vdash T \rightsquigarrow P \Rightarrow \Gamma$ ならば $[\![\Gamma(x)]\!]$ は実際に x に束縛され得る文字列の集合 $\{\theta(x) \mid s \in [\![T]\!] \wedge s \rhd P \Rightarrow \theta \wedge \theta \neq \bot\}$ に一致する” という事である．この証明に，単純に $\emptyset \vdash T \rightsquigarrow P \Rightarrow \Gamma$ の導出に関する帰納法などを適用することはできない．なぜならば，規則 (P-Rep) などで Π は $\emptyset$ ではなくなってしまい，帰納法の仮定が適用できなくなるからである．次に考えられるのは捕題全体の言明を “$\Pi \vdash T \rightsquigarrow P \Rightarrow \Gamma$ ならば…” のように強めるやり方であるが，この方法もうまくいかな

い．なぜならば，Π の要素として任意のものを許してしまうと，Π の中に導出に悪影響を与える「ごみ」が含まれ得るためである．そのような「ごみ」によって正しくない結果を生じてしまう導出の例を図 8 に示す．ここで変数 x の型は $\emptyset$ であると推論されているが，一見して分かるように x の正しい型は $\mathtt{a}$ になるはずである．以上の事柄を踏まえ，証明に先立って導出の各過程で Π の中にこのような「ごみ」が含まれないことを保証しなければならない．そのために新しい導出規則と Π に関する述語を導入する．定義の詳細は前出の web サイトに譲るが，以下にそれぞれを簡単に解説する．

導出の拡張

まず，導出 $\Pi \vdash T \rightsquigarrow P \Rightarrow \Gamma$ を $\Pi \vdash T \rightsquigarrow P \Rightarrow \Gamma; \Pi'$ の形に拡張する．ここで Π' は Π の要素のうち，(P-Mem) 規則によって実際に使われた要素を記録する集合である．

Π の正当性

次に，導出 $\Pi \vdash T \rightsquigarrow P \Rightarrow \Gamma; \Pi'$ において Π の中の要素が正当なものである，ということを定義するための関係を導入する．$\Pi \overset{P}{\bowtie} T$ と書かれる関係によって，Π の中のどの要素も，少なくとも T，P に関して悪影響を持たないということを表す．この関係が Π の中に「ごみ」が含まれていない，ということの定義となる．

証明の流れ

以上の定義を導入した上で，証明は以下のようになされる．(1) 最初に，導出 $\Pi \vdash T \rightsquigarrow P \Rightarrow \Gamma; \Pi'$ において $\Pi \overset{P}{\bowtie} T$ が成り立っているならば，導出の仮定の段階でも同様の性質が成り立つことを示す．すなわち，導出の帰結において Π が「ごみ」を含まないならば，この性質は導出全体の不変条件となることを証明する．(2) 導出が (1) の性質を満たすならば，実際に使われた要素の集合 Π' も特定の性質を満たすことを示す．(3) 最後に，導出において Π' が (2) の性質を満たすならば型推論の健全性・完全性が成り立つことを示して証明の完了となる．結局，最終的に証明されるのは以下の言明となる：“$\Pi \vdash T \rightsquigarrow P \Rightarrow \Gamma; \Pi'$ かつ $\Pi \overset{P}{\bowtie} T$ ならば健全性・完全性が (Π' に関して少し強められた形で) 満たされる．” $\Pi = \emptyset$ の時，直観的には $\emptyset$ に「ごみ」は含まれないので) 任意の T, P について $\emptyset \overset{P}{\bowtie} T$ が成り立つ．このことから，補題 9 は上の言明の系として示される．

5. プログラム例

本章では λ^{re} によるプログラムの小さな例 (とその型付け) を紹介する．

ブール値

文字列 $\mathtt{t}$ と $\mathtt{f}$ を true, false とみなせば，ブール代数は λ^{re} で表現できる．条件式 if M_0 then M_1 else M_2 はパターンマッチを使って match M_0 with $\mathtt{t} \Rightarrow M_1 \mid \mathtt{f} \Rightarrow M_2$ と表すことができる．型 bool はそれゆえ $\mathtt{t} \mid \mathtt{f}$ として定義できる．これと再帰関数を組み合わせれば，λ^{re} は Turing 完全である．

リスト

型 T を，文字 , (カンマ) を中に含まない任意の文字列を表す型としよう．この時，T 型の文字列のリストは $(T,)^*$ 型の文字列として表現できる．その考え方は以下の通りである: 空リストは空文字列 ε で表し，要素 s とリスト ℓ の cons は $s\,\hat{}\,,\hat{}\,\ell$ で表す．リスト ℓ を分解するコードは，パターンマッチを使って match ℓ with $\varepsilon \Rightarrow M_1 \mid (x \text{ as } T),(y \text{ as } .^*) \Rightarrow M_2$ のようにして実現できる．x は ℓ の先頭要素 (“car”) に，y は残りの要素 (“cdr”) に対応する．y のパターンはプログラム上は $.^*$ と書かれているが，ℓ の型が $(T,)^*$ であれば，y の型は $(T,)^*$ と正確に，自動的に推論される．この例が末尾以外の場所の as パターンの例にもなっていることに注意されたい．これは先行研究[5] と比較した我々の型システムの利点である．

フィルタリング

(上の方法で表現された) リストに対しては，“grep” のような簡単なフィルタリングが以下のように明らかな方法で実現できる．ただしフィルタリングに使用する正規表現 T' はコンパイル時に静的に与えられるものとする．

$$
\begin{array}{l}
\mathsf{fix}(f, x, \mathsf{match}\ x\ \mathsf{with} \\
\qquad \varepsilon \Rightarrow \varepsilon \\
\qquad \mid (y \text{ as } T'),(z \text{ as } .^*) \Rightarrow y\,\hat{}\,,\hat{}\,f(z) \\
\qquad \mid T,(z \text{ as } .^*) \Rightarrow f(z)
\end{array}
$$

このフィルタ関数の型は期待通り $(T,)^* \overset{\varepsilon}{\to} ((T \cap T'),)^*$ となる．

パーズと整形表示

自然数とその上での基本的な演算が (簡単のため厳密な定義抜きに) 組込み済みであるとすると，自然数とその文字列による表記の間の相互変換は下のプログラムで実現できる．

$$\dfrac{\dfrac{(\llbracket \mathtt{ab} \rrbracket, \mathtt{a}(y\ \mathtt{as}\ \mathtt{b})) \in \{(\llbracket \mathtt{ab} \rrbracket, \mathtt{a}(y\ \mathtt{as}\ \mathtt{b}))\}}{\{(\llbracket \mathtt{ab} \rrbracket, \mathtt{a}(y\ \mathtt{as}\ \mathtt{b}))\} \vdash \mathtt{ab} \rightsquigarrow \mathtt{a}(y\ \mathtt{as}\ \mathtt{b}) \Rightarrow \{y : \emptyset\}}\,(\text{P-Mem})}{\{(\llbracket \mathtt{ab} \rrbracket, \mathtt{a}(y\ \mathtt{as}\ \mathtt{b}))\} \vdash \mathtt{ab} \rightsquigarrow (x\ \mathtt{as}\ \mathtt{a})\mathtt{b} \Rightarrow \{x : \emptyset\}}\,(\text{P-Seq-Bind})$$

図 8 正しくない導出の例
Fig. 8 Example of Wrong Derivation.

$$\begin{array}{l}
\mathsf{let}\ \mathit{dig2chr} = \lambda d. \\
\quad \mathsf{if}\ d = 9\ \mathsf{then}\ 9\ \mathsf{else} \\
\quad \mathsf{if}\ d = 8\ \mathsf{then}\ 8\ \mathsf{else} \\
\quad \ldots \\
\quad \mathsf{if}\ d = 1\ \mathsf{then}\ 1\ \mathsf{else} \\
\quad 0\ \mathsf{in} \\
\mathsf{fix}(\mathit{nat2str}, n, \\
\quad \mathsf{if}\ n < 10\ \mathsf{then}\ \mathit{dig2chr}(n)\ \mathsf{else} \\
\quad \mathit{nat2str}(n\ \mathsf{div}\ 10)\ \hat{}\ \mathit{dig2chr}(n\ \mathsf{mod}\ 10))
\end{array}$$

及び

$$\begin{array}{l}
\mathsf{let}\ \mathit{chr2dig} = \lambda c. \\
\quad \mathsf{match}\ c\ \mathsf{with}\ 9 \Rightarrow 9 \\
\quad \mid \ldots \\
\quad \mid 0 \Rightarrow 0\ \mathsf{in} \\
\mathsf{fix}(\mathit{str2int}, s, \\
\quad \mathsf{match}\ s\ \mathsf{with}\ (d\ \mathsf{as}\ .) \Rightarrow \mathit{chr2dig}(d) \\
\quad \mid (d\ \mathsf{as}\ .)(t\ \mathsf{as}\ .^*) \Rightarrow \\
\quad \mathit{chr2dig}(d) \times 10 + \mathit{str2int}(t)
\end{array}$$

．それぞれの型は $\mathsf{nat} \xrightarrow{\varepsilon} \iota\iota^*$ と $\iota\iota^* \xrightarrow{\varepsilon} \mathsf{nat}$ である．ただし $\iota = 0 \mid \ldots \mid 9$．しかしながら，我々の型システムは $\mathit{nat2str}$ の値域の型を $0 \mid (\iota \cap \overline{0})\iota^*$ と型付けできるほど強力ではない．実際には 00 のような結果は起こりえないのであるが，それを型システムで表現するには nat に対する dependent type を必要とするだろう．

6. 結　　論

我々はテキスト処理の為の小さな言語として λ^{re} を定義し，文字列と副作用に対して正規表現を用いた型システムを導入した．

XDuce の他には五十嵐・小林による *resource usage analysis*[7]（及び先行研究）がおそらく我々と最も関連の深い研究である．Resource usage analysis は，ファイル・メモリ領域など種々の計算機資源に対してプログラムがどのようなパターンでアクセスするかを静的に解析（そして検証）するための一般的な枠組みである．我々の正規表現による effect system は彼らの枠組みの簡単な例と見なすことができる．しかしながら彼らの枠組みでは，アクセスパターンの記述言語は再帰を含んでしまうため等価性と包含の判定は決定不能であり，型チェックも一般には決定不能となってしまう．一方，我々の体系では文字列の型は正則であるため，fix の束縛変数に型の注釈が付いていれば型チェックは決定可能である．

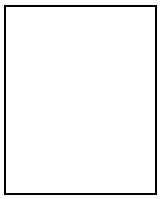
本論文では正規表現による型システムの定義とその健全性という，理論的な基礎となる部分について紹介したが，実用化のためには他にも解決すべき課題が残されている．とりわけ，型チェック，そしてより重要な型推論の効率的なアルゴリズムは実用にあたって本質的な位置を占めるが，現在はまだ用意されていない．技術的に重大な問題として，制約解消に基づく標準的な型推論の手法では，制約の中に再帰を含んでしまうため正規表現よりも表現力の強い（少なくとも文脈自由文法程度の）解が必要になるという点が挙げられる．第 1 章で見た $f(n) = \mathsf{if}\ n \leq 0\ \mathsf{then}\ \varepsilon\ \mathsf{else}\ \mathtt{a}\hat{}f(n-1)\hat{}\mathtt{b}$ という関数の例を思い出すと，型推論の過程で $\varepsilon \mid \mathtt{a}\alpha\mathtt{b} \leq \alpha$ のような制約が得られるが，これは正規表現の範囲では最小の解を持たない．なぜなら $\mathtt{a}^*\mathtt{b}^* \geq (\varepsilon \mid \mathtt{aa}^*\mathtt{b}^*\mathtt{b}) \geq (\varepsilon \mid \mathtt{ab} \mid \mathtt{aaa}^*\mathtt{b}^*\mathtt{bb}) \geq (\varepsilon \mid \mathtt{ab} \mid \mathtt{aabb} \mid \mathtt{aaaa}^*\mathtt{b}^*\mathtt{bbb}) \geq \ldots$ 等が全て有効な解となるからである．我々は目下 “適度な” 近似解を与える型推論アルゴリズムの構築に向けて，自然言語処理の分野から文献 10)，11) など類似の研究の成果を取り入れながら作業中である．

参 考 文 献

1) *CERT Advisory CA-2000-02*, http://www.cert.org/advisories/CA-2000-02.html (2000).
2) Frisch, A., Castagna, G. and Benzaken, V.: Semantic subtyping, *Logic in Computer Science*, Los Alamitos, CA, USA, IEEE Computer Society, pp. 137–146 (2002).
3) Harper, R.: Proof-Directed Debugging, *Journal of Functional Programming*, Vol. 9, No. 4, pp. 463–469 (1999).
4) Hopcroft, J.E., Motwani, R. and Ullman, J.D.: *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Publishing, 2nd edition (2000).
5) Hosoya, H. and Pierce, B. C.: Regular ex-

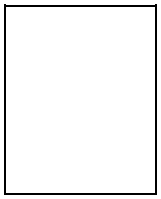
pression pattern matching for XML, *Journal of Functional Programming* (2002). To appear. Extended abstract in *Symposium on Principles of Programming Languages*, pages 67–80, 2001.
6) Hosoya, H., Vouillon, J. and Pierce, B. C.: Regular expression types for XML, *International Conference on Functional Programming*, pp. 11–22 (2000).
7) Igarashi, A. and Kobayashi, N.: Resource Usage Analysis, *Symposium on Principles of Programming Languages* (2002).
8) Lutz, M.: *Programming Python*, O'Reilly & Associates, 2nd edition (2001).
9) Matsumoto, Y. M. and Ishituka, K.: *The Ruby Programming Language*, Addison-Wesley Publishing (2002).
10) Mohri, M. and Nederhof, M.-J.: Regular approximation of context-free grammars through transformation, *Robustness in Language and Speech Technology* (Junqua, J.-C. and Noord, G. V.(eds.)), Kluwer Academic Publishers (2001).
11) Nederhof, M.-J.: Regular Approximation of CFLs: A Grammatical View, *Advances in Probabilistic and Other Parsing Technologies* (Bunt, H. C. and Nijholt, A.(eds.)), Kluwer Academic Publishers (2000).
12) Thomas, D., Hunt, A. and Thomas, D.: *Programming Ruby: A Pragmatic Programmer's Guide*, Addison-Wesley Publishing (2000).
13) Wall, L. et al.: *Programming Perl*, O'Reilly & Associates, 3rd edition (2000).
14) 山本篤：正規表現の関数としての定義とパターンマッチアルゴリズム, http://aglaia.c.u-tokyo.ac.jp/~yamamoto/regex_function/ (2001).

(平成？年？月？日受付)
(平成？年？月？日採録)



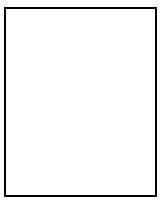
田渕　直

1977 年生．2001 年 3 月京都大学文学部心理学専攻卒業．同年 4 月東京大学情報理工学系研究科コンピュータ科学専攻修士課程進学．ソフトウェアの自動検証等の領域において，型システムの理論及び応用に関する研究に従事．



住井英二郎

1975 年生. 2000 年 3 月東京大学大学院理学系研究科情報科学専攻修士課程修了. 同年 4 月、同専攻博士課程進学. 同年より翌年にかけて, 米国ペンシルバニア大学訪問研究員. 2001 年 4 月より東京大学大学院情報理工学系研究科コンピュータ科学専攻助手 (2003 年 3 月まで同大学院情報学環へ流動). 関数型言語, プロセス計算, 部分評価, 情報セキュリティ等の領域における, 先進的な型システムの理論と応用に関する研究に従事. 日本ソフトウェア科学会員. ACM 会員.



米澤　明憲 (正会員)

1947 年生．1977 年 Ph.D. in Computer Science (MIT)．1989 年より東京大学理学部情報科学科教授．超並列・分散ソフトウェアアーキテクチャなどに興味を持つ．共著書「モデルと表現」等 (岩波書店)，編著書「ABCL」(MIT Press) 等がある．1992–1996 年ドイツ国立情報処理研究所 (GMD) 科学顧問，ACM Transaction on Programming Languages and Systems 副編集長，IEEE Parallel & Distributed Technology および Computer 編集委員などを歴任，元ソフトウェア科学会理事長，ACM Fellow．