表示的意味論に基づくパターンマッチングコンパイル方式の構築と実装

大堀 淳　纓坂 智

本論文では，パターンマッチングとそのコンパイルを系統的に理解するための表示的意味論を提示し，それを基に，パターンマッチング構文を効率よいコードにコンパイルする系統的なアルゴリズムを提案する．まず，パターンを値の部分集合と見なし，パターンマッチング構文を，マッチングの対象項が含まれる部分集合を決定する機構と見なすことにより，パターンマッチングの集合論的意味論を与える．次に，各パターンの表示的意味を表現する木構造を定義し，それに基づきパターンマッチングのコンパイルアルゴリズムを導出し，このアルゴリズムの正しさを証明する．導出されたアルゴリズムは，分岐の選択に関して正しいばかりでなく，実行時の値のテスト回数に関して最適であり，かつ，冗長なパターンや網羅的でないパターン集合を常に正確に検出することが保証されている．以上構築したアルゴリズムは，Standard ML の拡張言語である SML♯ 言語に対して実装され，SML♯ コンパイラのパターンマッチングコンパイルモジュールとしての使用を通じてその実用性が確認されている．

This paper presents a denotational semantics for patten matching compilation, and develops a simple algorithm to compile pattern matching into efficient code. We interpret a given set of patterns as a disjoint set of subsets of values that partition the domain of a given type, and define a semantics of a pattern matching construct as an operation to determine the subset to which a given value belongs. By developing tree representations for subsets of values, this semantics yields a simple tree-expansion algorithm to compile a pattern matching construct into a sequence of primitive tests. The resulting algorithm is shown to be correct with respect to the denotational semantics. The algorithm produces efficient code performing each test only once, and detects redundancy and non-exhaustiveness of a given set of patterns without introducing any additional mechanism. The presented method has been implemented for SML♯, an extension of the Standard ML language being developed at Research Institute of Electrical Communication, Tohoku University.

1 序論

ML [11] や，OCaml [10]，Haskell [7] などの関数型言語で広く採用されいるパターンマッチングは，処理対象データの選択や部分データの取りだしをパターンを用いて記述する機能である．この機能により，複雑なデータ構造の操作を簡潔に記述することが可能となり，関数型言語の利便性が大きく向上する．しかし一方，コンパイルされたコードの効率を確保するためには，パターンマッチングを用いたプログラムを，値のテストによる分岐や組の要素の取りだしなどのプリミティブ演算からなる効率よいコード列に変換する必要がある．この処理は，型付き関数型言語のコンパイルの中でも最も複雑な処理の一つである．

従来知られているパターンマッチングコンパイルの方式の一つに，バックトラックオートマトンの構成がある [3] [9] [8]．この方式では，候補パターンを順に選び，選択された候補パターンの構造に関して再帰的に

A Fresh Look at Pattern Matching Compilation.

Atsushi Ohori, 東北大学電気通信研究所, Research Institute of Electrical Communication, Tohoku University.

Satoshi Osaka, 株式会社アドバンテスト，Advantest Corporation.

コンピュータソフトウェア, Vol.24, No.2 (2007), pp.113–132.

[論文] 2004 年 10 月 17 日受付.

本研究の一部は，大堀および纓坂が北陸先端科学技術大学院大学に所属していた時になされたものであり，文部科学省リーディングプロジェクト e-Society 基盤ソフトウェアの総合開発「高い生産性をもつ高信頼ソフトウエア作成技術の開発」の一課題「プログラムの自動解析に基づく高信頼ソフトウェアシステム構築技術」として実施されたものである．

各部分のマッチングを試みる．この過程でマッチングが失敗したら「バックトラック」を行い，まだ選ばれていない他の候補パターンとのマッチングを試みる．この方式には，その簡単な再帰的な構造のためコンパイラを容易に実装でき，またコンパイル後のコードサイズも小さいという利点がある一方，パターンの冗長性や網羅性の検出を正確に行うことが困難であるという欠点がある．また，効率に関しても，コードサイズは小さいものの，バックトラックによって同一のテストをくり返す可能性があり，実行速度に関して最適なコードを生成できないという制限がある．

パターンマッチングコンパイルのもう一つの代表的なアプローチは，与えられたパターン集合から，マッチングの成功に至るテストと分岐に関する決定木を構築するというものである．この方式では，同一のテストを二回以上行わないため，実行効率に関して最適なコードを生成することができる．また，決定木の葉の集合を分析することによって，パターン集合の冗長性や網羅性を検出することも可能である．このアプローチは，文献[5] でその概要が提示され，文献[4] [1] などで議論され，Standard ML of New Jersey compiler [2] で実現されている．

決定木は，テストと分岐によって解決できる問題に対する一般的な計算モデルであり，決定木を用いるという考え方それ自身でパターンマッチングコンパイルのアリゴリズムが導かれるわけではない．このアプローチによりパターンマッチングを実現するためには，パターンマッチング構文から決定木を構築するアルゴリズムと，その決定木から目的コードを生成するアルゴリズムを構築する必要がある．Scott と Ramsey は文献[13] で，コンストラクター集合からなる単純な型無しの項集合を対象とし，決定木を用いたマッチコンパイルアルゴリズムの簡潔な記述を与えている．しかし，この研究を含めた既存研究では，現実の言語に現れる種々のパターンを含む決定木の構築の方法や，網羅性のチェックや冗長なパターンの検出方式の詳細は明らかではなく，また，パターンマッチングコンパイルアルゴリズムの正しさの証明もなされていない．現状では，実用言語に対するパターンマッチングコンパイラの系統的な構築方法が十分に確立されているとは言えない．本論文の目的は，strict な意味論をもつ ML 系型付き関数型言語のパターンマッチングコンパイルの意味論を与え，それを基礎に，系統的なパターンマッチングコンパイルアリゴリズムを構築し，その正しさを証明することである．

従来の決定木によるアプローチでは，パターンマッチングを，タグや値のテストを通じて選択肢の可能性を制限していくプロセスとみなす．本論文では，対象言語が静的型システムを装備していることに着目し，従来のこの動的な意味論に代って，各パターンの表示的意味を定義し，パターンマッチングプロセスの静的意味を定義し，それに従ってパターンマッチングのコンパイルアリゴリズムを導出する．

ML 系の型付き言語では，パターンマッチングの対象となる型は，整数などの原子型，直積型構成子，直和型構成子から構成される一階のデータ型であり，各パターンは，それら型を持つ正規項の特定の部分集合の表現と見なせる．すると，パターンマッチング問題は，与えられた項がどの部分集合に属するかを決定する問題と見なすことができる．さらに，各パターンに対応する正規項の集合を表現する木表現を定義することによって，パターンマッチングを行うケース文を，分岐先をラベルとして持つ木で表現出来る．この見方は，パターンマッチング問題への表示的意味を与えるのみならず，パターンマッチングの系統的なコンパイルアルゴリズムも導出する．導出されるアルゴリズムは，木構造を拡張していく簡単な再帰的な構造をしており，パターン集合の網羅性や冗長性の検出機構がすでにそのアルゴリズム自体に備わっている．さらに，この型の表示的意味論を基礎とするアプローチでは，or-パターンなども，なんら特別な機構を追加することなく導入することができる．

本論文では，この考え方に従い，パターンマッチングのコンパイルアルゴリズムを構築し，その正しさを証明する．このアルゴリズムは，Standard ML 言語のフルセットに対して実装され，Standard ML の拡張言語である SML♯ のコンパイラ [14] に組み込み，実際に使用され，その実用性が実証されている．本論文では，この実装の経験を基礎とした種々の最適化を含む実用化技術も合わせて報告する．

もちろん決定木を用いたパターンマッチングコンパイル方式に欠点がないわけではない．例えば，通常の決定木を用いたパターンマッチングコンパイラは，パターンの大きさに関して指数関数的なコードを作る場合がある．本論文で示したアルゴリズムに対しても，そのような例が存在する．そのような例を最小限に押さえるための種々最適化手法が考えられているが，最適な決定木の構成問題は NP-完全であると報告されており [2]，これら結果を考えると，この問題の完全な解決は困難と思われる．これら種々の経験的な手法に基づく最適化手法の導入は重要であるが，それらは，本稿の範囲を越えるので扱わないことにする．

以降の本論文の構造は以下の通りである．第 2 節ではパターンマッチング問題を定義し，その表示的意味を与える．第 3 節ではパターンの意味を表現する木表現を構築し，第 4 節でパターンマッチング式のコンパイルアルゴリズムを構築する．第 5 節でコンパイルアリゴリズムの正しさを示す．第 6 節ではコンパイルアルゴリズムの SML♯ コンパイラへの実装の概要を報告する．第 7 節で今後の課題を，第 8 節で結論を述べる．

2 パターンマッチング問題

本節では，我々が対象とする strict な意味論をもつ関数型言語のパターンマッチング問題を定義する．実際の ML 系言語は，再帰的なラベル付きバリアント型のデータがパターンマッチングの主要な対象であるが，本論文で提案するパターンマッチングアルゴリズムを記述する上では，再帰やラベルは本質的ではない．そこで，本論文では，原子型に加え，直積型と直和型のみを扱う．

2.1 型，項，およびパターンの定義

以下の文法で与えられる型の集合を考える．

$$\tau ::= t \mid b \mid \tau + \cdots + \tau \mid \tau * \cdots * \tau$$

t はパターンマッチングでは分析の対象とならない型を表す．関数型や抽象データ型などが含まれる．b は整数などの原子型を表す．ここでは，簡単のために一つの原子型 b 型とその型をもつ定数集合 $\{c_1, c_2, \ldots\}$ が与えられていると仮定する．$\tau + \cdots + \tau$ は直和型であり，ML 系言語において `datatype` 文で定義される型のモデルである．$\tau * \cdots * \tau$ は直積型であり，ML 系言語のレコード型やタプル型に相当する．1 要素の直積型を要素型そのものと区別しなければならない場合，前者を (τ) と書くことがある．

項の集合を以下の文法で定義する．

$$v ::= t \mid c \mid (i(v) : \tau_1 + \cdots + \tau_n) \mid (v, \ldots, v)$$

t は型 t を持つ項である．パターンマッチングはこの型のデータの内部を分析しないので，この型に対する代表元を一つ項集合に加えれば十分である．そこで，本稿では型 t そのものを用いる．$(i(v) : \tau_1 + \cdots + \tau_n)$ は直和型を持つ項である．この項に含まれる明示的な型宣言は，直和型のデータは唯一の型を持つという ML 系言語の性質をモデル化している．型が重要ではない文脈では，型を省略し $i(v)$ と書くことがある．$(v, \ldots, v)$ は直積型のデータである．

パターンの集合は以下の文法で与えられる．

$$P ::= c \mid _ \mid x \mid i(P) \mid (P, \ldots, P)$$

c は定数，x は変数，“_” は任意の値にマッチするワイルドカードパターンを表す．$i(P)$ は i のタグをもつ直和データを表すパターン，$(P_1, \ldots, P_n)$ は直積データのパターンである．

パターンマッチング構文は以下の文法で与えられる．

$$\texttt{case}\ e\ \texttt{of}\ P_1\ \texttt{=>}\ e_1\ \texttt{|}\ \cdots\ \texttt{|}\ P_n\ \texttt{=>}\ e_n$$

我々が前提とする型付き言語では，マッチングの対象となる式 e は，上で定義されるいずれかの型 τ を持っているはずであり，各パターン P_i もすべて同一の型 τ を持っているはずである．以下，この性質を明示する必要がある場合，P_τ と書くことがある．具体的な型 τ は，この構文を含む文脈の型推論によって決定されるが，その機構はここでは本質的ではない．

2.2 パターンマッチングの表示的意味論

系統的なパターンマッチングコンパイルの基礎として，パターンとパターンマッチングの表示的意味論を定義する．

型 τ の表示的意味 $[\![\tau]\!]$ を以下の様に定義する．

$$\begin{aligned}
[\![t]\!] &= \{t\} \\
[\![b]\!] &= \{c_1, c_2, \ldots\} \\
[\![\tau_1 + \ldots + \tau_n]\!] &= \{(i_k(a) : \tau_1 + \ldots + \tau_n) \mid
\end{aligned}$$

$$a \in [\![\tau_k]\!], 1 \leq k \leq n\}$$
$$[\![\tau_1 * \ldots * \tau_n]\!] = \{(a_1, \ldots, a_n) | a_i \in [\![\tau_i]\!]\}$$

型 τ のパターン P_τ の意味を以下の様に与える．

$$\begin{aligned}
[\![_\tau]\!] &= [\![\tau]\!] \\
[\![x_\tau]\!] &= [\![\tau]\!] \\
[\![c_b]\!] &= \{c\} \\
[\![i(P)_{\tau_1+\ldots+\tau_n}]\!] &= \{i(a) \mid a \in [\![P_{\tau_i}]\!]\} \\
[\![(P^1, \ldots, P^n)_{\tau_1*\ldots*\tau_n}]\!] &= [\![P^1_{\tau_1}]\!] \times \cdots \times [\![P^n_{\tau_n}]\!]
\end{aligned}$$

次にパターンマッチング構文

```
case e of P1 => e1 | ... | Pn => en
```

の意味を定義する．マッチングの対象となる項 e の型を τ として，集合列 X_i と Y_i を以下のように定義する．

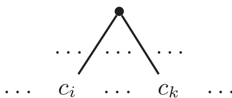
$$\begin{aligned}
Y_0 &= \emptyset \\
Y_i &= Y_{i-1} \cup [\![P_i]\!]_\tau \\
X_i &= [\![P_i]\!]_\tau \setminus Y_{i-1}
\end{aligned}$$

すると，上記のパターンマッチング構文の意味は，e が表現する正規項 v が属する X_i に対応する分岐 e_i を選択する機構と定義できる．さらに，もし $X_i = \emptyset$ なら P_i は冗長パターンであり，また，$Y_n \neq [\![\tau]\!]$ ならパターンの集合は網羅的ではない．

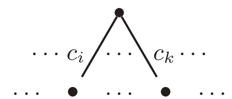
3 項と項集合の木表現

上記の意味論は，パターンマッチング構文の直感的な理解に合致し，その意味で明らかに正しいと確認できるが，これのみからパターンマッチングアルゴリズムを直ちに得ることはできない．我々の目的は，入力が属する X_i を決定する構成的なアルゴリズムを構築することである．このために，まず各集合 $[\![\tau]\!]$ の具体的な表現を定義することを考える．

τ が原子型 b の場合，その値の集合は以下のような木で表現できる．



この木は，原子型 b の値による分岐を表す以下のような無限の決定木と見なすことができる．



$\mathcal{T}(t, X) =$ (t / X)

$\mathcal{T}(b, X) =$ ($\cdots c_i \cdots c_k \cdots$ / $\cdots X \cdots X \cdots$)

$\mathcal{T}(\tau_1 + \cdots + \tau_n, X) =$ ($1 \cdots n$ / $\mathcal{T}(\tau_1, X) \cdots \mathcal{T}(\tau_n, X)$)

$$\mathcal{T}(\tau_1 * \ldots * \tau_n, X) = \mathcal{T}(\tau_1, \mathcal{T}(\tau_2 * \ldots * \tau_n, X))$$

図 1 型領域の木表現

c_i でラベル付けられたグラフの辺は，マッチングの対象式の値が c_i であった時に取るべき分岐を意味しており，● は分岐先で実行する計算を表している．

このように原子型が単独で現れる場合は，その意味を表現する木それ自身を，そのデータ型に対する決定木と見なすことができる．しかし，ある型が複合型の部分型として現れる場合，そのパターンに対応するマッチングの判定は，それを囲む型に対するさらなるテストに継続する必要がある．この継続を表現するために，型の領域に対する木表現を，その木のすべての葉が持つ値をパラメターとして持つ様に一般化する．葉に関連付けられた値が X であるような型 τ の木を $\mathcal{T}(\tau, X)$ と書くことにする．上記の原子型の表現は $\mathcal{T}(b, \bullet)$ と書ける．この一般化により，決定木は，それに継続する計算によってパラメータ化されたものとなり，複合型に対応する決定木の再帰的な定義が可能となる．

直積型の木表現を決定するためには，それぞれの要素を展開する順序を決定する必要がある．この順が，直積型パターンのマッチングを試みる順番になる．本稿では，

$$\tau_1 * \tau_2 * \ldots * \tau_n = \tau_1 * (\tau_2 * \ldots * \tau_n)$$

の同型関係を使い，最左の要素から順に木を生成する方式を取る．別な順序も同様に可能である．

以上の方針に従った $\mathcal{T}(\tau, X)$ の再帰的な定義を図 1 に与える．例えば，$\mathcal{T}(int + int, \{\bullet\})$ と $\mathcal{T}(int * int, \{\bullet\})$ はそれぞれ以下のような木で表現さ

$\mathcal{P}(x:\tau,X) = \mathcal{T}(\tau,X)$

$\mathcal{P}(_:\tau,X) = \mathcal{T}(\tau,X)$

$\mathcal{P}(c:b,X) =$ (tree: edge c to X)

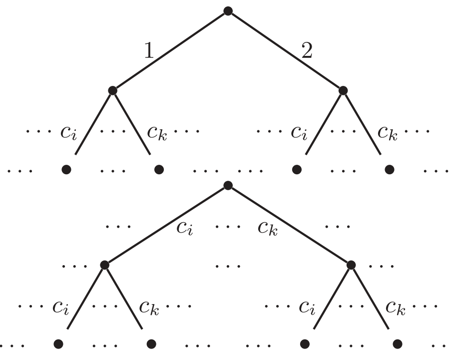
$\mathcal{P}(i(P):\tau_1+\cdots+\tau_n,X) =$ (tree: edge i to $\mathcal{P}(P:\tau_i,X)$)

$\mathcal{P}((P_1,\ldots,P_n):\tau_1*\ldots*\tau_n,X)$

$= \mathcal{P}(P_1:\tau_1,\mathcal{P}((P_2,\ldots,P_n):\tau_2*\ldots*\tau_n,X))$

図 2 パターンの表示的意味の木表現

れる．

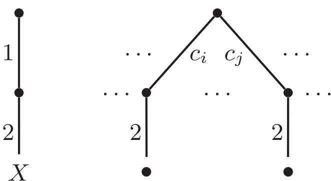


前者の木は，まず直和データのタグチェックを行った後に，int 型のデータで分岐し，さらにパラメータで与えられた決定木に継続する決定木を表している．

次に，葉のラベルを X とする型 τ のパターン P に対する木表現 $\mathcal{P}(P:\tau,X)$ を型に関して再帰的に図 2 のように定義する．例えば，$(1,2)$ と $(_,2)$ の各パターンに対する木表現は以下のように与えられる．

$$\mathcal{P}((1,2):int\times int,\bullet) = \mathcal{P}(1:int,\mathcal{P}(2:int,\bullet))$$
$$\mathcal{P}((_,2):int\times int,\bullet) = \mathcal{P}(_:int,\mathcal{P}(2:int,\bullet))$$

従って，それぞれ以下の木で表現される．



これらの木表現を用いると，構文

case $e:\tau$ of P_1 => e_1 | ... | P_n => e_n

に対するパターンマッチングのコンパイルは，以下のような木のマーク付けプロセスと捉えることができる．

- 可能な値を表す木 $\mathcal{T}(\tau,\bullet)$ を構築する．
- 各 $P_i(i=1,\ldots,n)$ に対して，順に以下の処理を行う．現在の木に含まれるすべてのパスについて，それが $\mathcal{P}(P_i:\tau,\bullet)$ にも含まれておりかつそのパスの葉が未だマークされていなければ，その葉を e_i でマークする．
- もし e_i でマークされた葉が存在しなければ，P_i は冗長なパターンである．
- もし $\bullet$ でマークされた葉が残っていれば，パターン集合は網羅的でない．

この戦略の下で構成的なアルゴリズムを構築するためには，アルゴリズムがある i 番目のパターンを処理している時点でのマーク付けられたパスの集合

$$X_i = [\![P_1]\!]_\tau \cup\cdots\cup [\![P_{i-1}]\!]_\tau$$

とマークがまだ付けられていないパス集合

$$Y_i = [\![\tau]\!] \setminus ([\![P_1]\!]_\tau \cup\cdots\cup [\![P_{i-1}]\!]_\tau)$$

に対する有限な木表現を構築する必要がある．X_i に関しては，その型の構造に従い再帰的に定義できる．特に，マークされたノードがまだ存在しない場合は X_i は空集合であり，定数項 ϕ で表現する．また，$[\![\tau]\!]$ は固定されているので，Y_i は X_i の補集合で表現できる．そこで,「ある有限のラベルの補集合」をラベルとしてもつエッジを導入する．この考えは，[12] で提案された制約付き項の考えと類似のものである．以降，集合 S の補集合を $\overline{S}$ と書く．

以上の洞察の下に，パターンの部分集合によってマークされた木構造 $\mathcal{M}(\tau,X)$ を図 3 のように定義する．パターンマッチングコンパイルの途上に現れる決定木は，ここで定義した木構造のパラメータ X を適当なマークに置き換えられたものに対応する．この木構造を使えば，パターンマッチングコンパイルを，与えられた作成中の決定木の中で $\mathcal{P}(P_i:\tau,\bullet)$ にマッチするパスを P_i に対するコード e_i でマークするプロセスと定義することができる．パターンの順序に基づく優先度は，すでにマーク済みのパスは無視することで

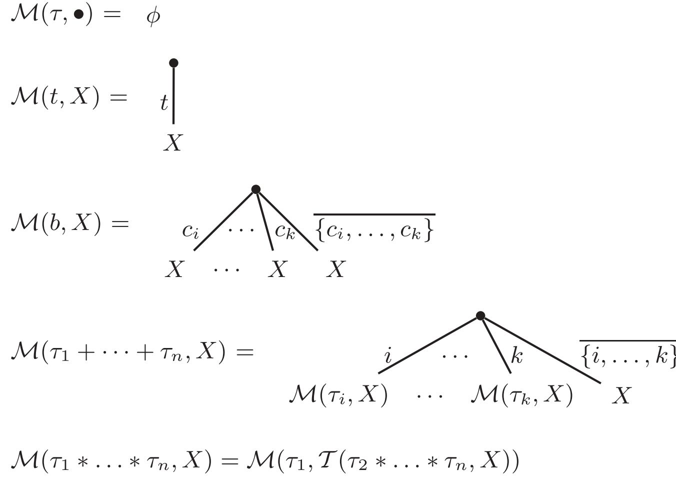


図 3　決定木の効率的な表現

実現できる．次節では，この考え方に基づき，パターンマッチングコンパイルアルゴリズムを構築する．

4　コンパイルアルゴリズム

本節では以上定義したパターンマッチングの意味論を基礎に，パターンマッチングのコンパイルアリゴリズムを構築する．

4.1　決定木とパターンの項表現

パターンマッチングコンパイラは，パターンによる場合分けを行うために，直和や直積などの入れ子構造を成した項の内部構造へアクセスするコードを生成しなければならない．また，変数パターンによる変数束縛を実行するコードも生成する必要がある．パターンマッチングコンパイルアルゴリズムを定義するためには，決定木とパターンの木表現をこれらの情報を加え拡張し，それらをアルゴリズムで操作するための具体的な項表現を与える必要がある．必要な拡張は以下の 3 点である．

1. 各中間ノードへのアクセスパスの追加．ノードが対応する項の部分項へのアクセスパス a を追加する．アクセスパス a は，決定木作成後のコード生成段階で項の部分項を束縛する変数の名前として使用される．アクセスパスが同じノードは，項の対応する部分項も同じである．
2. 明示的なフィールドの記録とフィールドアクセスノードの追加．決定木が部分木である場合，現在処理中のパターンが元々の直和および直積パターンの何番目であるかを記録し，その部分木に対応する処理に制御を移す前に，要素のアクセスコードを追加する必要がある．この処理を明示的に示すために，直和型および直積型を，要素位置を示す自然数のラベル付型で表し，さらに，直積型の決定木の場合は，フィールドアクセスを表現するノードを，部分木の先頭に追加する．
3. 変数の束縛環境の追加．変数パターンによる変数束縛を行うために，葉 e を環境 Γ と e の組へと変更する．環境 Γ は変数パターンによる変数から，変数に対応するアクセスパスへの写像である．

上記のように拡張された決定木は下記のような項で表現できる．

$$
\begin{array}{rcl}
T & ::= & \phi \\
 & | & \texttt{eq}(a, \{c_i : T, \ldots, c_k : T\}, T) \\
 & | & \texttt{tag}(a, \{i{:}\, T, \ldots, k{:}\, T\}, T) \\
 & | & \texttt{prod}(a, n, i{:}\, T) \\
 & | & \texttt{univ}(a, T) \\
 & | & \texttt{leaf}(\Gamma, e)
\end{array}
$$

ϕ は拡張が行われていない空の木を表す．$\texttt{eq}(a, \{c_i : T_i, \ldots, c_k : T_k\}, T_0)$ は型 b の項に関する等価テスト

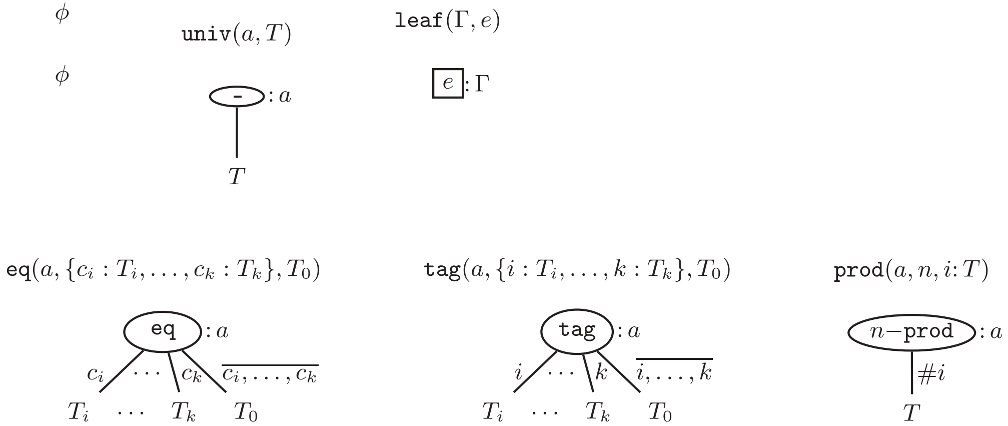


図 4 決定木とその項表現の対応

による分岐を表す．T_0 は $\overline{\{c_i,\ldots,c_k\}}$ による分岐を表す．tag$(a,\{i:T_i,\ldots,k:T_k\},T_0)$ は直和型のタグに対する分岐を表す．eq の場合と同様に，T_0 は $\overline{\{i,\ldots,k\}}$ による分岐を表す．prod$(a,n,i{:}T_i)$ は，T_i の先頭の部分木が n 個のフィールドによって構成される直積の i 番目のフィールドに対応するものであることを表す．これら両者に現れる $i,\ldots,k$ はもともとのパターン中の位置を表す自然数ラベルである．univ(a,T) はワイルドパターンもしくは変数パターンに対応するノードである．leaf(Γ,e) は環境 Γ を伴った e による分岐が選択されたことを表す．図 4 に決定木とその項表現の対応を示す．

次に，パターンをその構造に従い逐次的に処理するために，パターンを表す木に対して，その処理の順序を明示した項表現を定義する．パターンが表す木を Ps とすると，Ps は葉 e であるか，パターン $P:\tau$ の葉が X である木 $\mathcal{P}(P':\tau,X)$ のどちらかである．後者の場合，まず P を処理し，それに引き続き $\mathcal{P}(P':\tau,X)$ を処理することになる．X 自身も Ps であるから，Ps は以下のようなリストとして定義できる．

$$Ps ::= e \mid (P,a) :: Ps$$

e は e をラベルとする葉を表す．$(P,a) :: Ps$ は，この部分項へのアクセスパスが a である木 $\mathcal{P}(P:\tau,Ps)$ を表す．Ps の中に現れる各パターン P は以下のような項で表現する．

$$P ::= c \mid _ \mid x \mid i(P) \mid \{i:P,\ldots,n:P\}$$

$\{i:P,\ldots,n:P\}$ は，もともとの直和型パターンの i 番目から n 番目までの部分パターンを表す直和型パターンである．

4.2 決定木拡張アルゴリズム

パターンマッチングコンパイルアルゴリズムは，まず，領域を表す木を，各パターンを表す木でマーク付けを行い，パターンマッチング構文に対する決定木を構築する．決定木の中でまだマーク付けられていない部分は暗黙に空の木で表現されているため，決定木のマーク付けは，部分的にマーク付けられた決定木 T をパターンを表す木 Ps で拡張する以下の型を持つ関数として与えられる．

$$\mathcal{E} : (Ps * \Gamma * T) \mapsto T$$

Γ は T に付加する葉が持つ環境である．図 5 にアルゴリズム $\mathcal{E}$ の定義を示す．

この定義は，それ自身パターンマッチングメタ表現を用いて記述してある．詳細な条件があるケースに関しては，*where* 節でパターンが満たすべき条件の詳細を記述してある．またこの定義において使用される関数 $getPath(T)$ は，T のルートノードのアクセスパスを返す関数である．また $\uplus$ は枝集合へ枝を加える演算である．$rules \uplus \{i:T_i\}$ は枝集合 $rules$ に部分木 T_i を持つ枝 i を加える．もし $rules$ が枝 i を持つなら，枝に接続された部分木を T_i で置き換える．

$$\mathcal{E}(e,\Gamma,\phi)=\texttt{leaf}(\Gamma,e)$$

$$\mathcal{E}(e,\Gamma,\texttt{leaf}(\Gamma',e'))=\texttt{leaf}(\Gamma',e')$$

$$\mathcal{E}((c,a)::Ps,\Gamma,\phi)=\texttt{eq}(a,\{c{:}\,\mathcal{E}(Ps,\Gamma,\phi)\},\phi)$$

$$\mathcal{E}((c,a)::Ps,\Gamma,\texttt{eq}(a,rules,T_0))=\texttt{eq}(a,rules\uplus\{c:\mathcal{E}(Ps,\Gamma,T')\},T_0)\quad\text{where } T'=T \text{ if } (c:T\in rules) \text{ otherwise } T_0$$

$$\mathcal{E}((c,a)::Ps,\Gamma,\texttt{univ}(a,T_0))=\texttt{eq}(a,\{c:\mathcal{E}(Ps,\Gamma,T_0)\},T_0)$$

$$\mathcal{E}((i(P),a)::Ps,\Gamma,\phi)=\texttt{tag}(a,\{i:\mathcal{E}((P,a')::Ps),\Gamma,\phi)\},\phi)\qquad\text{where } a' \text{ fresh}$$

$$\mathcal{E}((i(P),a)::Ps,\Gamma,\texttt{tag}(a,rules,T_0))=\texttt{tag}(a,rules\uplus\{i:\mathcal{E}((P,getPath(T'))::Ps,\Gamma,T')\},T_0')$$
$$\text{where } T'=\begin{cases}T & (i:T\in rules)\\ T_0 & (i:T\notin rules)\end{cases},\; T_0'=\begin{cases}\phi & (\overline{\{k|k{:}\,T_k\in rules\}\cup\{i\}}=\emptyset)\\ T_0 & (otherwise)\end{cases}$$

$$\mathcal{E}((i(P),a)::Ps,\Gamma,\texttt{univ}(a,T))=\texttt{tag}(a,\{i:\mathcal{E}((P,a')::Ps,\Gamma,\texttt{univ}(a',T))\},T')$$
$$\text{where } a' \text{ fresh}; T'=\begin{cases}\phi & (\overline{\{i\}}=\emptyset)\\ T & (\overline{\{i\}}\neq\emptyset)\end{cases}$$

$$\mathcal{E}((\{i:P_i,\ldots,n:P_n\},a)::Ps,\Gamma,\phi)=$$
$$=\begin{cases}\texttt{prod}(a,n,i{:}\,\mathcal{E}((P_i,a')::(\{i+1:P_{i+1},\ldots,n:P_n\},a)::Ps,\Gamma,\phi)) & (i\neq n; a' \text{ fresh})\\ \texttt{prod}(a,n,n{:}\,\mathcal{E}((P_n,a')::Ps,\Gamma,\phi)) & (i=n; a' \text{ fresh})\end{cases}$$

$$\mathcal{E}((\{i:P_i,\ldots,n:P_n\},a)::Ps,\Gamma,\texttt{prod}(a,n,i{:}\,T_i))$$
$$=\begin{cases}\texttt{prod}(a,n,i{:}\,\mathcal{E}((P_i,getPath(T_i))::(\{i+1:P_{i+1},\ldots,n:P_n\},a)::Ps,\Gamma,T_i)) & (i\neq n)\\ \texttt{prod}(a,n,n{:}\,\mathcal{E}((P_n,getPath(T_n))::Ps,\Gamma,T_n)) & (i=n)\end{cases}$$

$$\mathcal{E}((\{i:P_i,\ldots,n:P_n\},a)::Ps,\Gamma,\texttt{univ}(a,T))$$
$$=\begin{cases}\texttt{prod}(a,n,i{:}\,\mathcal{E}((P_i,a')::(\{i+1:P_{i+1},\ldots,n:P_n\},a)::Ps,\Gamma,\texttt{univ}(a',\texttt{univ}(a,T)))) & (i\neq n; a' \text{ fresh})\\ \texttt{prod}(a,n,n{:}\,\mathcal{E}((P_n,a')::Ps,\Gamma,\texttt{univ}(a',T))) & (i=n; a' \text{ fresh})\end{cases}$$

$$\mathcal{E}((_,a)::Ps,\Gamma,\phi)=\texttt{univ}(a,\mathcal{E}(Ps,\Gamma,\phi))$$

$$\mathcal{E}((_,a)::Ps,\Gamma,\texttt{eq}(a,rules,T_0))=\texttt{eq}(a,\{c_i{:}\,\mathcal{E}(Ps,\Gamma,T_i)|c_i{:}\,T_i\in rules\},\mathcal{E}(Ps,\Gamma,T_o))$$

$$\mathcal{E}((_,a)::Ps,\Gamma,\texttt{tag}(a,rules,T_0))=\texttt{tag}(a,\{i{:}\,\mathcal{E}((_,getPath(T_i))::Ps,\Gamma,T_i)|i{:}\,T_i\in rules\},T_0')$$
$$\text{where } T_0'=\begin{cases}\phi & (\overline{\{i|i{:}\,T_i\in rules\}}=\emptyset)\\ \mathcal{E}(Ps,\Gamma,T_o) & (otherwise)\end{cases}$$

$$\mathcal{E}((_,a)::Ps,\Gamma,\texttt{prod}(a,n,i{:}\,T_i))=\begin{cases}\texttt{prod}(a,n,i{:}\,\mathcal{E}((_,getPath(T_i))::(_,a)::Ps,\Gamma,T_i)) & (i\neq n)\\ \texttt{prod}(a,n,n{:}\,\mathcal{E}((_,getPath(T_n))::Ps,\Gamma,T_n)) & (i=n)\end{cases}$$

$$\mathcal{E}((_,a)::Ps,\Gamma,\texttt{univ}(a,T_0))=\texttt{univ}(a,\mathcal{E}(Ps,\Gamma,T_0))$$

$$\mathcal{E}((x,a)::Ps,\Gamma,T)=\mathcal{E}((_,a)::Ps,\Gamma\{x:a\},T)$$

図 5 木拡張アルゴリズム

以下，Ps の先頭の構造の種類毎に，図 5 に与えた $\mathcal{E}(Ps, \Gamma, T)$ の処理の概要を説明する．

$Ps = e$ の場合．e に対応するパターンによるマッチングが成功した時の処理である．もし葉 T が ϕ であるなら，$\texttt{leaf}(\Gamma, e)$ を付加する．T がすでに別の葉を持つなら，この場合において e に対応するパターンは冗長であるため，葉に変更を加えない．

$Ps = (P, a) :: Ps'$ の場合．P の種類に応じて以下のような処理を行う．

$P = c$ の場合．T の枝 c に接続された部分木を，Ps' と Γ で拡張する．もし T が ϕ である場合，これは型 b の値全てに関して ϕ への分岐があることを意味する．そこで拡張して得られる部分木を持つ枝 c と，ϕ を持つ $\{c\}$ の補集合の枝の二つを持つ eq 接点を作成する．もし T が $\texttt{univ}(a, T')$ である場合，これは型 b の値全てに関して T' への分岐があることを意味する．よって T が ϕ である場合と同様に T' の拡張を行い，eq ノードを作成する．もし T が eq ノードであるなら，その枝集合から c を選び出し，その部分木を拡張する．

$P = i(P')$ の場合．$P = c$ の場合と同様に考える．T の枝 i に接続された部分木を拡張する．部分木は，まず P' に関して拡張を行い次いで Ps' によって拡張する．

$P = \{i : P_i, \ldots, n : P_n\}$ の場合．P_i を用いて，n 次の直積の i 番目のフィールドの拡張を行う．$i+1, \ldots, n$ に関する拡張は P_i による拡張の後に行う．

$P =$ _ の場合．ワイルドパターンは対応する部分木を補完する働きをする．つまり，対応する部分木の全ての枝を再帰的に拡張する．

$P = x$ の場合．変数パターンによる変数束縛を生成する．変数の参照先はアクセスパス a である．Γ を $\{x : a\}$ で拡張する．拡張された環境は $\mathcal{E}$ アルゴリズムによって葉へと伝搬される．

4.3 目的コードの生成

パターンマッチングコンパイルアルゴリズムを完成させるためには，前節のアルゴリズムによって作成された決定木で表現される動作を実現する目的コードを生成するアリゴリズムを与える必要がある．

決定木は，それ自身テストと分岐を行うアルゴリズムを表しており，木を再帰的にたどれば，ルートから葉に至るパスに対応するパターンに対する分岐コードが得られる．したがって，目的コードは，この分岐コードに続き，対応する葉 $\texttt{leaf}(\Gamma, e)$ に格納されている式 e を環境 Γ の下で実行するコードを生成すれば実現できる．しかしながら，決定木拡張アルゴリズムは，決定木の作成過程において部分木の複製を生成することがあり，上記のコード生成を単純に実行すると，実行式 e_i が複数現れる可能性がある．この問題を解決するには，以下の方針に従い，実行式 e を関数で表現すればよい．

場合分構文に現れる各ルール P => e に対して，P に含まれる自由変数集合 $\{x_1, \ldots, x_m\}$ を求め，変数に対する一定の線形順序を仮定し，e を実行する関数 fn x_1 => $\cdots$ fn x_m => e を構築し，この関数を新しい名前 X で束縛したのち，e の代わりに X を葉として決定木を構成する．構成された決定木に含まれる葉は $\texttt{leaf}(\Gamma, X)$ の形をしており，Γ には，アルゴリズムの正しさから，P の自由変数集合の束縛が生成されているはずである．そこで，この葉に対して，変数の順序に従い関数適応式 X $\Gamma(x_1)$ $\cdots$ $\Gamma(x_1)$ を構築する．この方式は，関数クロージャの作成と関数呼び出しのオーバーヘッドを伴う．この問題の解決方法は，6.3.1 および第 7 節で論じる．

以上の方針の下，パターンマッチングコンパイルアルゴリズムを定義する．図 6 にターゲット言語の定義を与える．switch 文は定数による分岐である．case 文は，ソース言語と違い，トップレベルのタグによる分岐のみを行う組み込み構文である．図 7 はパターンマッチング式 e を，ターゲット言語へ変換するアルゴリズム $\mathcal{ML}[\![e]\!]$ の定義である．同図中の $\mathcal{TL}[\![T]\!]$ は $\mathcal{E}$ アルゴリズムによって作成した決定木 T からターゲットコードを生成するアルゴリズムである．図 8 に $\mathcal{TL}[\![T]\!]$ の定義を示す．

アルゴリズム $\mathcal{ML}$ によって決定木を作成する過程で，冗長なパターンを検出することができる．各パターンによって決定木を拡張する際，拡張後の木に一

$$
\begin{array}{rcl}
e & ::= & c^b \mid x \mid \texttt{\#}i\ e \mid \{i:e,\ldots,k:e\} \mid i(e) \mid \texttt{fn}\ x\ \texttt{=>}\ e \mid \texttt{raise}\ e \mid e\ \texttt{handle}\ e \\
 & \mid & \texttt{let}\ x\ \texttt{=}\ e\ \ldots x\ \texttt{=}\ e\ \texttt{in}\ e\ \texttt{end} \\
 & \mid & \texttt{switch}\ e\ \texttt{of}\ c_i\ \texttt{=>}\ e\ \mid\ \ldots\ \mid c_k\ \texttt{=>}\ e \mid\ _\ \texttt{=>}\ e \\
 & \mid & \texttt{case}\ e\ \texttt{of}\ i(x)\ \texttt{=>}\ e\ \mid\ \ldots\ \mid k(x)\ \texttt{=>}\ e \mid\ _\ \texttt{=>}\ e
\end{array}
$$

図 6　対象言語

$$
\begin{array}{l}
\mathcal{ML}[\![\texttt{case}\ e\ \texttt{of}\ P_1\ \texttt{=>}\ e_1\ \texttt{|}\ \cdots\ \texttt{|}\ P_n\ \texttt{=>}\ e_n]\!] \\
\quad = \texttt{let}\ a\ \texttt{=}\ e\ \texttt{in} \\
\qquad \texttt{let} \\
\qquad\quad X_1\ \texttt{= fn}\ x_1^1\ \texttt{=>}\ \cdots\ \texttt{fn}\ x_{m_1}^1\ \texttt{=>}e_1 \qquad (\{x_1^1,\ldots,x_{m_1}^1\} = FV(P_1)) \\
\qquad\quad \ldots \\
\qquad\quad X_n\ \texttt{= fn}\ x_1^n\ \texttt{=>}\ \cdots\ \texttt{fn}\ x_{m_n}^n\ \texttt{=>}\ e_n \qquad (\{x_1^n,\ldots,x_{m_n}^n\} = FV(P_n)) \\
\qquad\quad X_0\ \texttt{= raise Match} \\
\qquad \texttt{in} \\
\qquad\quad \mathcal{TL}[\![\mathcal{E}((_,a) :: X_0,\emptyset,\mathcal{E}((P_n,a) :: X_n,\emptyset,\ldots\mathcal{E}((P_1,a) :: X_1,\emptyset,\phi)\ldots))]\!] \\
\qquad \texttt{end}
\end{array}
$$

図 7　パターンマッチングコンパイルアルゴリズム

$\mathcal{TL}[\![\phi]\!] = \texttt{raise Match}$

$\mathcal{TL}[\![\texttt{eq}(a,\{\{i:T_i,\ldots,k:T_k\}\},T_0)]\!]$
$\quad = \ \texttt{switch}\ a\ \texttt{of}\ i\ \texttt{=>}\ \mathcal{TL}[\![T_i]\!]\ \texttt{|}\ \ldots\ \texttt{|}\ k\ \texttt{=>}\ \mathcal{TL}[\![T_k]\!]\ \texttt{|}\ _\ \texttt{=>}\ \mathcal{TL}[\![T_0]\!]$

$\mathcal{TL}[\![\texttt{tag}(a,\{\{i:T_i,\ldots,k:T_k\}\},T_0)]\!]$
$\quad = \ \texttt{case}\ a\ \texttt{of}\ i(a_i)\ \texttt{=>}\ \mathcal{TL}[\![T_i]\!]\ \texttt{|}\ \ldots\ \texttt{|}\ k(a_k)\ \texttt{=>}\ \mathcal{TL}[\![T_k]\!]\ \texttt{|}\ _\ \texttt{=>}\ \mathcal{TL}[\![T_0]\!]$
$\qquad\qquad$ *where* $a_j = getPath(T_j) \quad (i \le j \le k)$

$\mathcal{TL}[\![\texttt{prod}(a,n,i{:}T_i)]\!] = \ \texttt{let}\ a_i\ \texttt{=}\ \texttt{\#}i\ a\ \texttt{in}\ \mathcal{TL}[\![T_i]\!]\ \texttt{end}$ $\qquad$ *where* $a_i = getPath(T_i)$

$\mathcal{TL}[\![\texttt{univ}(a,T)]\!] = \mathcal{TL}[\![T]\!]$

$\mathcal{TL}[\![\texttt{leaf}(\emptyset,e)]\!] = e$

$\mathcal{TL}[\![\texttt{leaf}(\{x_1:a_1,\ldots,x_n:a_n\},X)]\!] = X\ a_1\ \cdots\ a_n$

図 8　ターゲットコード生成アルゴリズム

つも葉が付加されなかった場合，そのパターンは冗長であることが分かる．また，アルゴリズム $\mathcal{ML}$ は木の拡張の最後に，ワイルドパターンを用いて拡張を行う．ワイルドパターンによる拡張は木を補完するものであるため，この拡張によって新たに葉が追加された場合，パターン集合が網羅的でないことが分かる．

本論文によるパターンマッチングコンパイルの例を示す．ここでは Standard ML の文法を用いる．以下の入力コードを考える．

```
datatype foo = A of int | B of int
```

$\mathcal{ML}[\![$`case (2,B(5)) of (1,A(2)) => 3 | (_,B(x)) => x`$]\!]$

$= let\ a_1 = fresh\ path\ name$

$\quad T = \mathcal{E}(_, X_0, \emptyset, \mathcal{E}(($`_,B(x)`$), X_2, \emptyset, \mathcal{E}(($`1,A(1)`$), X_1, \emptyset)\phi))$

$\ in$

`let` a_1 `= (2,B(5)) in`

`let`

X_1 `= 3,` X_2 `= x,` X_0 `= raise Match`

`in` $\mathcal{TL}[\![T]\!]$ `end`

`end`

図 9 コンパイル例 (1)：アルゴリズムの呼び出し構造

$$
\begin{aligned}
T_1 &= \phi \\
T_2 &= \mathcal{E}((\{\texttt{1 = 1, 2 = A(2)}\}, a) :: X_1, \emptyset, T_1) \\
T_3 &= \mathcal{E}((\{\texttt{1 = _, 2 = B(x)}\}, a) :: X_2, \emptyset, T_2) \\
T_4 &= \mathcal{E}((_, a) :: X_0, \emptyset, T_3)
\end{aligned}
$$

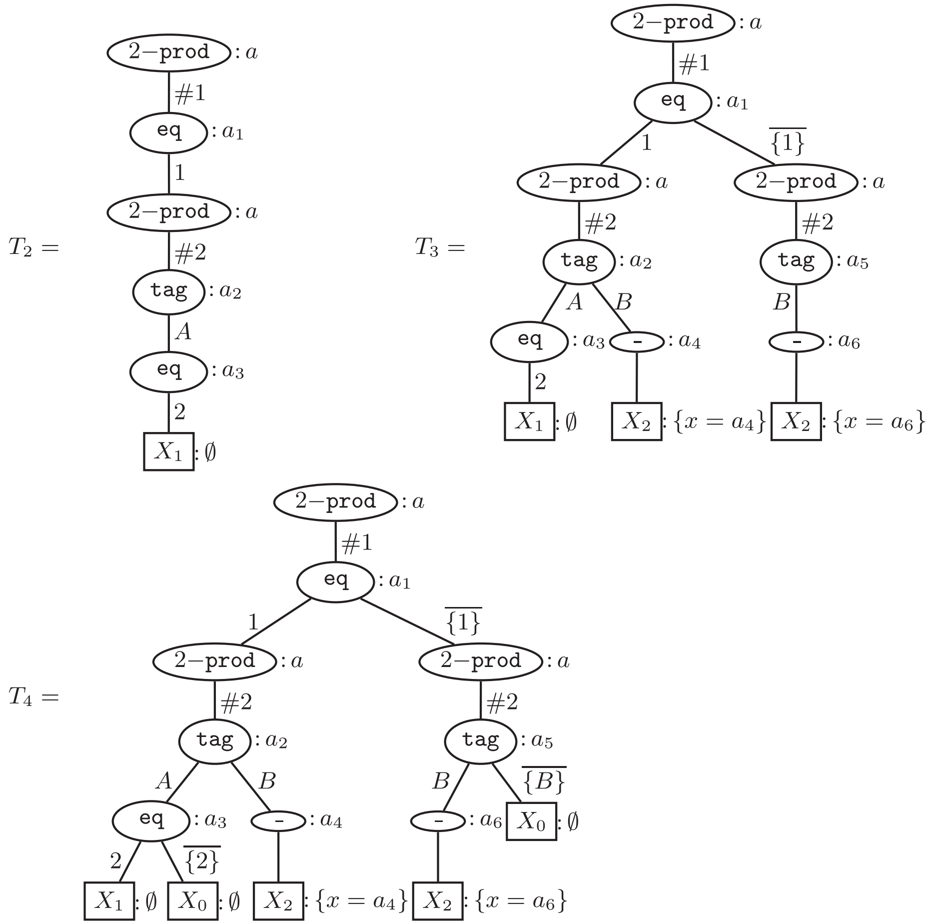


図 10 コンパイル例 (2)：木の拡張過程

```
let a = {1 = 2, 2 = B(5)} in
  let
     X1 = fn () => 3
     X2 = fn x => x
  in
    let a1 = #1 a in
      switch a1 of
        1 => let a2 = #2 a in
               case a2 of
                 A(a3) =>
                    switch a3 of
                      2 => X1 ()
                    | _ => raise Match
               | B(a4) =>  X2 a4
             end
      | _ => let a5 = #2 a in
               case a5 of
                 B(a6) => X2 a6
               | _ => raise Match
             end
    end
  end
end
```

図 11　コンパイル例 (3)：コンパイル結果

```
case (2, B(5)) of (1, A(2)) => 3
                | (_, B(x)) => x
```

本論文で構築したアルゴリズムがこのコードに対して行う計算を図 9 に示す．この計算は，空の木 ϕ に対して，くり返しアルゴリズム $\mathcal{E}$ を三回適用して木の拡張を行う．図 10 は各段階での木の状態を示すものである．$\mathcal{E}$ によって作成した木にコード生成アルゴリズム $\mathcal{TL}$ を適用することにより，最終的に図 11 に示すターゲットコードが生成される．

5　アルゴリズムの正しさ

本節では，前節で定義した木拡張アルゴリズム $\mathcal{E}$ によって作成された決定木がパターンマッチングの表示的意味論に関して正しいことを示す．そのためにまず，木の型，型付け規則，および木が表す項の部分集合等を定義し，$\mathcal{E}$ が型を保存することを示し，それを用いて木拡張アリゴリズムの健全性と完全性を示す．その特殊な場合として，パターンマッチングアルゴリズムの正しが導かれる．

5.1　木の型付けと型保存定理

木が項集合の表現となっているためには，木は整合性を持たなければならない．そのための条件として，木の型および型付け規則を定義する．

決定木とパターンが表す木の型 σ は以下のように定義する．

$$\sigma ::= \bullet \mid \tau :: \sigma$$

決定木 T およびパターンが表す木 Ps が型 σ を持つことをそれぞれ $T : \sigma$ および $Ps : \sigma$ と書く．図 12 および図 13 にこれら型判定を導出する規則の集合を与える．

前節で定義した木拡張アルゴリズム $\mathcal{E}$ の正しさの証明には $\mathcal{E}$ に関する以下の性質が有用である．

定理 1 (型保存定理) もし $Ps : \sigma, T' : \sigma$ なら，任意の Γ について，$T = \mathcal{E}(Ps, \Gamma, T')$ とすると $T : \sigma$ である．

証明 (概要):　Ps の構造に関する帰納法で証明する．Γ を任意の環境とし，Ps と T' に関して場合分けを

$$\texttt{leaf}(\Gamma, e) : \bullet$$

$$\frac{\phi : \sigma \quad T_i : \sigma \quad \cdots \quad T_k : \sigma \quad T_0 : \sigma}{\texttt{eq}(a, \{c_i : T_i, \ldots, c_k : T_k\}, T_0) : b :: \sigma}$$

$$\frac{T_i : \tau_i :: \sigma \quad \cdots \quad T_k : \tau_k :: \sigma \quad T_0 : \sigma}{\texttt{tag}(a, \{i : T_i, \ldots, k : T_k\}, T_0) : \tau_1 + \ldots + \tau_n :: \sigma}$$

$$\frac{T_n : \tau_n :: \sigma}{\texttt{prod}(a, n, n : T_n) : (\tau_n) :: \sigma}$$

$$\frac{T_i : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma}{\texttt{prod}(a, n, i : T_i) : \tau_i * \ldots * \tau_n :: \sigma} \quad (i \neq n)$$

$$\frac{T : \sigma}{\texttt{univ}(a, T) : \tau :: \sigma} \quad (T \neq \phi)$$

図 12　決定木の型付け規則

$c : b$

$_ : \tau$

$x : \tau$

$$\frac{P : \tau_i}{i(P) : \tau_1 + \cdots + \tau_n}$$

$$\frac{P_i : \tau_i \quad \cdots \quad P_n : \tau_n}{\{i : P_i, \ldots, n : P_n\} : \tau_1 + \cdots + \tau_n}$$

$e : \bullet$

$$\frac{P : \tau \quad Ps : \sigma}{(P, a) :: Ps : \tau :: \sigma}$$

図 13 パターンおよびパターン木の型付け規則

行う．以下にいくつかの場合についての証明を与える．他の場合も同様に示すことができる．

$Ps = e$ の場合．型付け規則より，$Ps : \bullet$ である．仮定と型付け規則より，ある Γ', e' があって，$T' = \texttt{leaf}(\Gamma', e') : \bullet$ か $T' = \phi : \bullet$ のいずれかである．$T' = \texttt{leaf}(\Gamma', e) : \bullet$ なら，$\mathcal{E}$ の定義より，$T = T' : \bullet$ である．$T' = \phi : \bullet$ なら，$\mathcal{E}$ の定義と型付け規則より，$T = \texttt{leaf}(\Gamma, e) : \bullet$ である．

$Ps = (P, a) :: Ps'$ の場合．P と T' に関して場合分けする．

$P = c : b, T' = \phi$ の場合．$Ps : \sigma, P : b$ であるから，$Ps' : \sigma'$ とすると型付け規則より $\sigma = b :: \sigma'$ である．$\mathcal{E}$ の定義より，$T = \texttt{eq}(a, \{c : \mathcal{E}(Ps', \Gamma, \phi)\}, \phi)$ である．ここで $T'_c = \mathcal{E}(Ps', \Gamma, \phi)$ とする．$Ps' : \sigma'$ であり，型付け規則より ϕ は任意の型でよいので，$\phi : \sigma'$ とおくと，帰納法の仮定より，$T'_c : \sigma'$ である．よって $T'_c : \sigma', \phi : \sigma'$ であるから，型付け規則より，$T = \texttt{eq}(a, \{c : \mathcal{E}(Ps', \Gamma, \phi)\}, \phi) : b :: \sigma'$ である．

$P = c : b, T' = \texttt{eq}(a, \{c_i : T_i, \ldots, c : T_c, \ldots, c_k : T_k\}, T_0)$ の場合．仮定と型付け規則より，$T_j : \sigma' \quad (j \in \{c_i, \ldots, c, \ldots, c_k\})$ かつ $T_0 : \sigma'$ である．$\mathcal{E}$ の定義より，$T = \texttt{eq}(a, \{c_i : T_i, \ldots, c : \mathcal{E}(Ps', \Gamma, T_c), \ldots, c_k : T_k\}, T_0)$ である．ここで $T'_c = \mathcal{E}(Ps', \Gamma, T_c)$ とおく．$Ps' : \sigma'$ かつ $T_c : \sigma'$ であるから，帰納法の仮定より $T'_c : \sigma'$ である．よって型付け規則より，$T = \texttt{eq}(a, \{c_i : T_i, \ldots, c : T'_c, \ldots, c_k : T_k\}, T_0) : b :: \sigma'$ である．

$P = i(P') : \tau_1 + \ldots + \tau_n, T' = \texttt{tag}(a, \{h : T_h, \ldots, i : T_i, \ldots, k : T_k\}, T_0)$ の場合．型付け規則より，$Ps' : \sigma'$ とすると，$T_j : \tau_j :: \sigma', h \leq j \leq k, T_0 : \sigma'$ である．$\mathcal{E}$ の定義より，$T = \texttt{tag}(a, \{h : T_h, \ldots, i : \mathcal{E}(P' :: Ps', \Gamma, T_i), \ldots, k : T_k\}, T_0)$ である．ここで $T'_i = \mathcal{E}(P' :: Ps', \Gamma, T_i)$ とすると，$P' :: Ps' : \tau_i :: \sigma', T_i : \sigma'$ であるから，帰納法の仮定より，$T'_i : \sigma'$ である．よって型付け規則より，$T : \tau_1 + \ldots + \tau_n :: \sigma'$ である．□

5.2 木の表示的意味と $\mathcal{E}$ の正しさの証明

型 σ が表す項集合 $[\![\sigma]\!]$ と $Ps : \sigma$ が表す項の集合 $[\![Ps : \sigma]\!]$ を以下のように定義する．

$$\begin{aligned}
[\![\bullet]\!] &= \{\bullet\} \\
[\![\tau :: \sigma]\!] &= [\![\tau]\!] \times [\![\sigma]\!] \\
[\![e : \bullet]\!] &= \{\bullet\} \\
[\![(P, a) :: Ps) : \tau :: \sigma]\!] &= [\![P : \tau]\!] \times [\![Ps : \sigma]\!]
\end{aligned}$$

集合 S と $S \times \{\bullet\}$ は同型である．以降，この二つを同一視する．

木 $T : \sigma$ が表す項集合 $\overline{T : \sigma}$ の定義を図 14 に示す．e をソースプログラムに現れる分岐先の式とする．さらに，与えられたプログラムの中の各分岐式は互いに異なる式と見なす．より厳密には，自然数によるラベル付け式構成子を導入し，case 文の各分岐式はその分岐位置を表す自然数でラベル付けられていると見なせばよい．木 $T : \sigma$ の，葉が e である部分木 $T/e : \sigma$ の定義を図 15 に示す．

以下の補題は，$\mathcal{E}$ が意味論的に健全かつ完全であること，すなわち，$\mathcal{E}$ によって拡張される枝はパターンにマッチしかつその拡張はパターンにマッチするすべての枝を網羅することを示す．

補題 1 ($\mathcal{E}$ の健全性かつ完全性) もし $Ps : \sigma$ が e で終わり，かつ $T' : \sigma$ かつ $\overline{T'/e : \sigma} = \emptyset$ であり，任意の Γ について $T = \mathcal{E}(Ps, \Gamma, T')$ なら，

$$\overline{T : \sigma} = [\![Ps : \sigma]\!] \cup \overline{T' : \sigma}$$

かつ

$$\overline{T/e : \sigma} = [\![Ps : \tau]\!] \setminus \overline{T' : \sigma}$$

である．

証明 (概要): Ps の構造に関する帰納法で証明する．

$\overline{\phi : \sigma} = \emptyset$

$\overline{\texttt{leaf}(\Gamma, e) : \bullet} = \{\bullet\}$

$\overline{\texttt{eq}(a, \{c_i : T_i, \ldots, c_k : T_k\}, T_0) : b :: \sigma} = (\{c_i\} \times \overline{T_i : \sigma}) \cup \ldots \cup (\{c_k\} \times \overline{T_k : \sigma}) \cup (\overline{\{c_i, \ldots, c_k\}} \times \overline{T_0 : \sigma})$

$$\begin{aligned}&\overline{\texttt{tag}(a, \{i : T_i, \ldots, k : T_k\}, T_0) : \tau_1 + \ldots + \tau_n :: \sigma}\\ &\qquad = \{(j(t_1), t_2) \mid j \in \{i, \ldots, k\}, (t_1, t_2) \in \overline{T_j : \tau_j :: \sigma}\}\\ &\qquad\qquad \cup \{j(t) \mid j \notin \{i, \ldots, k\}, t \in [\![\tau_j]\!]\} \times \overline{T_0 : \sigma}\end{aligned}$$

$\overline{\texttt{prod}(a, n, n{:}T_n) : \tau_n :: \sigma} = \{((t_n), t) \mid (t_n, t) \in \overline{T_n : \tau_n :: \sigma}\}$

$$\begin{aligned}&\overline{\texttt{prod}(a, n, i{:}T) : \tau_i * \ldots * \tau_n :: \sigma}\\ &\qquad = \{((t_i, t_{i+1}, \ldots, t_n), t) \mid (t_i, ((t_{i+1}, \ldots, t_n), t)) \in \overline{T_i : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma}\} \quad (i \neq n)\end{aligned}$$

$\overline{\texttt{univ}(a, T) : \tau :: \sigma} = [\![\tau]\!] \times \overline{T : \sigma}$

図 14 木が表す項集合

$\phi/e : \sigma = \phi : \sigma$

$\texttt{leaf}(\Gamma, e)/e : \bullet = \texttt{leaf}(\Gamma, e) : \bullet$

$\texttt{leaf}(\Gamma, e')/e : \bullet = \phi : \bullet \qquad (e \neq e')$

$\texttt{eq}(a, rules, T_0)/e : b :: \sigma = \texttt{eq}(a, \{c : T/e : \sigma \mid c : T \in rules\}, T_0/e : \sigma)$

$\texttt{tag}(a, rules, T_0)/e : \tau_1 + \ldots + \tau_n :: \sigma = \texttt{tag}(a, \{i : T/e : \sigma \mid i : T \in rules\}, T_0/e : \sigma)$

$\texttt{prod}(a, n, i{:}T_i)/e : (\tau_i, \ldots, \tau_n) :: \sigma = \texttt{prod}(a, n, i{:}T_i/e : \tau_i * \ldots * \tau_n :: \sigma) : \tau_i * \ldots * \tau_n :: \sigma$

$\texttt{univ}(a, T)/e : \tau :: \sigma = \texttt{univ}(a, T/e : \sigma) : \tau :: \sigma$

図 15 T の部分木 $T/e : \sigma$

Γ を任意の環境と仮定し，Ps と T' に関して場合分けを行う．以下にいくつかの場合についての証明を与える．他の場合も同様の手法で証明することができる．

$Ps = e$ の場合．パターン木の型付け規則より $Ps : \bullet$ である．決定木の型付け規則より，ある Γ', e' があって $T' = \texttt{leaf}(\Gamma', e') : \bullet$ であるか，$T' = \phi : \bullet$ であるかのどちらかである．$T' = \texttt{leaf}(\Gamma', e') : \bullet$ の場合，$\overline{T'/e : \sigma} = \emptyset$ であるから，$e' \neq e$ である．$\mathcal{E}$ の定義より，$T = T' : \bullet$ である．よって以下の等式が成立する．

$$\begin{aligned}[\![Ps : \bullet]\!] \cup \overline{T' : \bullet} &= \{\bullet\} \cup \{\bullet\}\\ &= \{\bullet\}\\ &= \overline{T : \bullet}\end{aligned}$$

$$\begin{aligned}[\![Ps : \bullet]\!] \setminus \overline{T' : \bullet} &= \{\bullet\} \setminus \{\bullet\}\\ &= \emptyset\\ &= \overline{T/e : \bullet}\end{aligned}$$

$T' = \phi : \bullet$ の場合，$\mathcal{E}$ の定義より $T = \texttt{leaf}(\Gamma, e) : \bullet$ である．よって以下の等式が成立する．

$$\begin{aligned}[\![Ps : \bullet]\!] \cup \overline{T' : \bullet} &= \{\bullet\} \cup \emptyset\\ &= \{\bullet\}\\ &= \overline{T : \bullet}\\ [\![Ps : \bullet]\!] \setminus \overline{T'/e : \bullet} &= \{\bullet\} \setminus \emptyset\\ &= \{\bullet\}\\ &= \overline{T/e : \bullet}\end{aligned}$$

$Ps = (P, a) :: Ps'$ の場合．P と T' に関して場合分けを行う．

$P = c : b,\ T' = \phi$ の場合．$Ps : \sigma, P : b$ であるから，ある σ' があって，$\sigma = b :: \sigma'$ かつ $Ps' : \sigma'$ である．$\mathcal{E}$ の定義より，$T = \mathtt{eq}(a, \{c : \mathcal{E}(Ps', \Gamma, \phi)\}, \phi)$ である．$T'_c = \mathcal{E}(Ps', \Gamma, \phi)$ とする．Ps が e で終わり，$Ps = (P, a) :: Ps'$ であるから，$Ps' : \sigma'$ も e で終わる．定義より $\phi : \sigma'$ であり，$\overline{\phi/e : \sigma'} = \emptyset$ である．また，型保存定理より，$T'_c : \sigma'$ である．よって帰納法の仮定より，$\overline{T'_c : \sigma'} = \llbracket Ps' : \sigma' \rrbracket \cup \overline{\phi : \sigma'}$ かつ $\overline{T'_c/e : \sigma'} = \llbracket Ps' : \sigma' \rrbracket \setminus \overline{\phi : \sigma'}$ である．よって，以下の等式が成立する．

$$\begin{aligned}
\overline{T : b :: \sigma'} &= \overline{\mathtt{eq}(a, \{c : T'_c\}, \phi) : b :: \sigma'} \\
&= \{c\} \times \overline{T'_c : \sigma'} \\
&= \llbracket (c, a) :: Ps' : b :: \sigma' \rrbracket \cup \emptyset \\
&= \llbracket Ps : b :: \sigma' \rrbracket \cup \overline{T' : b :: \sigma'} \\
\overline{T/e : b :: \sigma'} &= \overline{\mathtt{eq}(a, \{c : T'_c\}, \phi)/e : b :: \sigma'} \\
&= \overline{\mathtt{eq}(a, \{c : T'_c/e : \sigma'\}, \phi) : b :: \sigma'} \\
&= \{c\} \times \overline{T'_c/e : \sigma'} \\
&= \{c\} \times (\llbracket Ps' : \sigma \rrbracket \setminus \overline{\phi : \sigma'}) \\
&= \{c\} \times \llbracket Ps' : \sigma' \rrbracket \\
&= \llbracket (c, a) :: Ps' : b :: \sigma' \rrbracket \setminus \emptyset \\
&= \llbracket Ps : b :: \sigma' \rrbracket \setminus \overline{T' : b :: \sigma'}
\end{aligned}$$

$P = \{i : P_i, \ldots, n : P_n\} : \tau_i * \ldots * \tau_n, T' = \mathtt{prod}(a, n, i{:}T_i), i \neq n$ の場合．$Ps = (P, a) :: Ps' : \sigma$ かつ $P : \tau_i * \ldots * \tau_n$ であるから，ある σ' があって，$\sigma = \tau_i * \ldots * \tau_n :: \sigma'$ かつ $Ps' : \sigma'$ である．また $i \neq n$ より $T_i : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma'$ である．$\mathcal{E}$ の定義より，$T = \mathtt{prod}(a, n, i{:}\mathcal{E}((P_i, a') :: (\{i+1 : P_{i+1}, \ldots, n : P_n\}, a) :: Ps', \Gamma, T_i))$ である．ここで $T'_i = \mathcal{E}((P_i, a') :: (\{i+1 : P_{i+1}, \ldots, n : P_n\}, a) :: Ps', \Gamma, T_i)$ とする．型保存定理より $T'_i : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma'$ である．Ps が e で終わるから，Ps' も e で終わる．$\overline{T'/e : \sigma} = \emptyset$ であるから，定義より $\overline{T_i/e : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma'} = \emptyset$ である．したがって帰納法の仮定と $\llbracket Ps \rrbracket$ の定義より，

$$\begin{aligned}
&\overline{T'_i : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma'} \\
&\quad = \llbracket (P_i, a') :: (\{i+1 : P_{i+1}, \ldots, n : P_n\}, a) :: Ps' \rrbracket \\
&\qquad \cup \overline{T_i : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma'} \\
&\quad = \llbracket P_i \rrbracket \times ((\llbracket P_{i+1}, \ldots \llbracket P_n \rrbracket \rrbracket) \times \llbracket Ps' \rrbracket) \\
&\qquad \cup \overline{T_i : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma'}
\end{aligned}$$

かつ

$$\begin{aligned}
&\overline{T'_i/e : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma'} \\
&\quad = \llbracket P_i \rrbracket \times ((\llbracket P_{i+1}, \ldots \llbracket P_n \rrbracket \rrbracket) \times \llbracket Ps' \rrbracket) \\
&\qquad \setminus \overline{T_i : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma'}
\end{aligned}$$

である．ここで $i \neq n$ であるから定義より，

$$\begin{aligned}
&\overline{T : \sigma} \\
&\quad = \overline{\mathtt{prod}(a, n, i{:}T'_i) : \sigma} \\
&\quad = \{((t_i, \ldots, t_n), t) \mid w = (t_i, ((t_{i+1}, \ldots, t_n), t)) \\
&\qquad\quad w \in \overline{T'_i : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma'}\}
\end{aligned}$$

である．ここで

$$\begin{aligned}
&\overline{T'_i : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma'} \\
&\quad = \llbracket P_i \rrbracket \times ((\llbracket P_{i+1}, \ldots \llbracket P_n \rrbracket \rrbracket) \times \llbracket Ps' \rrbracket) \\
&\qquad \cup \overline{T_i : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma'}
\end{aligned}$$

であるから，

$$\begin{aligned}
&\overline{T : \sigma} \\
&\quad = (\llbracket P_i \rrbracket \times \ldots \llbracket P_n \rrbracket) \times \llbracket Ps' \rrbracket \\
&\qquad \cup \{((t_i, \ldots, t_n), t) \mid w = (t_i, ((t_{i+1}, \ldots, t_n), t)) \\
&\qquad\quad w \in \overline{T_i : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma'}\} \\
&\quad = \llbracket (\{i : P_i, \ldots, n : P_n\}, a) :: Ps' : \tau_i * \ldots * \tau_n :: \sigma' \rrbracket \\
&\qquad \cup \overline{T' : \tau_i * \ldots * \tau_n :: \sigma'}
\end{aligned}$$

である．$\overline{T/e : \sigma}$ に関する等式も同様に示せる． □

以上の補題を用いて $\mathcal{E}$ の正しさを証明できる．

以下のパターンマッチング式を考える．

`case` $e : \tau$ `of` P_1 `=>` e_1 `|` ... `|` P_n `=>` e_n

上記の式に対し，アルゴリズム $\mathcal{ML}$ 内でアルゴリズム $\mathcal{E}$ は以下のような木を計算する．

$$\begin{aligned}
T_0 &= \phi \\
T_i &= \mathcal{E}((P_i, a) :: e_i, \emptyset, T_{i-1}) \\
&\qquad (1 \leq i \leq n;\ a \text{ は } e \text{ に束縛される})
\end{aligned}$$

以上から，アルゴリズムの正しさを示すには，以下の性質を証明すれば十分である．

定理 2 1. $\overline{T_i : \tau :: \bullet} = \llbracket (P_i, a) :: e_i : \tau :: \bullet \rrbracket \cup \overline{T_{i-1} : \tau :: \bullet} \quad (1 \leq i \leq n)$

2. $\overline{T_i/e_i : \tau :: \bullet} = \llbracket (P_i, a) :: e_i : \tau :: \bullet \rrbracket \setminus \overline{T_{i-1} : \tau :: \bullet} \quad (1 \leq i \leq n)$

証明: これは補題 1 において，$Ps = (P_i, a) :: e_i$ の場合である．

アルゴリズムが，パターンの冗長性とパターン集合の網羅性を正確に検出できることも，アルゴリズムの

$$\mathcal{E}((((P_1|P_2),a) :: Ps,\Gamma,T) = \mathcal{E}((P_2,a) :: Ps,\Gamma,\mathcal{E}((P_1,a) :: Ps,\Gamma,T))$$
$$\mathcal{E}((x \text{ as } P,a) :: Ps,\Gamma,T) = \mathcal{E}((P,a) :: Ps,\Gamma\{x:a\},T)$$

図 16 パターン言語と木拡張とアルゴリズムの拡張

定義と補題 1 の証明から導くことができる．

6 SML♯ への実装

以上構築したアルゴリズムを，実用的なプログラミング言語である SML♯ を対象に実装した．SML♯ 言語は，Standard ML の定義[11] を包摂する関数型言語であり，そのパターン言語は文献[11] で定義されているものに，or パターンを追加して拡張したものである．実装したアルゴリズムは SML♯ コンパイラの の一部として使用され，関数型言語の標準的なベンチマークを含む 3000 以上のテストケースを正しくコンパイルすることが確認されている．本節では，実装の概要およびその過程で開発した最適化手法の概要を報告する．

6.1 拡張パターン言語

SML♯ 言語は Standard ML の上位互換性を保っており，パターンマッチングは，場合分構文以外の以下の構文にも書くことができる．

- 関数定義文，関数式
- 例外処理構文
- 値の宣言文

これら構文の内，最初の二つは型に依存しない前処理フェーズによって，3 番目の値の宣言文は型推論モジュールによって，それぞれ場合分式とパターンを含まない構文の組み合わせに変換されている．

パターン言語自体も，本論文で考慮したパターンに加え，直積型と直和型はそれぞれラベル付き直積型とラベル付き直和型に一般化されており，さらに新しいパターンとして or-パターンとレイヤードパターンを含む．本論文の冒頭で述べたとおり，ラベル付き直積型とラベル付き直和型への一般化は何ら技術的な問題をふくまない．or-パターンとレイヤードパターンは，以下のように対応している．

or-パターンは $(P_1 \mid P_2)$ という形のパターンであり，P_1 もしくは P_2 とマッチする．その表示的意味はそれぞれのパターンの表示的意味の和集合であるから，木を P_1 によって拡張した後に，P_2 による拡張を行えばよい．レイヤードパターンは x as P という形のパターンであり，P とのマッチングを図るとともに，成功すれば，P に対応する部分項で x を束縛する．その表示的意味は P と同一であり，決定木の構造に影響を与えない．必要名処理は，P による木の拡張の前に，x による束縛を環境に追加するのみである．

実装システムでは，これらパターンに対応するため，図 16 に示すケースを加え拡張した $\mathcal{E}$ アルゴリズムを用いた[†1]．この拡張されたアルゴリズムに対しても，定理 2 が成立することを容易に確かめることができる．また，これら拡張は，網羅性と冗長性のチェックに影響を与えない．

6.2 実装の概要

SML♯ コンパイラは現在 12 の中間言語を用いた 13 フェーズからなるが，本研究で構築したパターンマッチングコンパイラは，その中の 7 番目のフェーズとして組み込まれている．このフェーズでは，型推論およびモジュールコンパイルが終了した後のフラットな型付き多相型言語をパターンマッチングの機能をもたない多相型レコード計算に変換する処理を行う．

フラットな型付き多相型言語は，パターンマッチングを行う CASE 構文を含む言語であり，本論文で考察した入力言語に対応している．

パターンマッチングコンパイラは，Standard ML を使い，`MATCH_COMPILER` シグネチャをもつ `MatchCompile` ストラクチャとして実装されている．`MATCH_COMPILER` は以下の簡単なものである．

†1 SML♯ アルファリリースにはこの拡張は含めていない．

```
  sig
    val compile : tfpdecl list
        -> rcdecl list * errorInfo list
  end
```

tfpdecl はフラットな多相型言語の宣言のリスト，rcdecl は多相型レコード計算の宣言のリスト，errorInfo list は網羅性や冗長性に関するメッセージである．MatchCompile は，tfpdecl を再帰的にたどり，CASE 構文に対して，本論文で示したアルゴリズムを適用している．

これら実装の実際の構造とソースコードは，[14] の中の「リソース」,「SML ＃内部構造」の順にページをたどり，MatchCompiler および MATCH COMPILER のタグをクリックすることで見ることができる．

6.3 最適化

実用的なコンパイラを実装するためには，コンパイルの各段階において，コンパイル時間の短縮および効率的なターゲットコードの生成を可能にする最適化を行う必要がある．パターンマッチングコンパイル処理も例外ではない．実用的な言語コンパイラを意図する SML♯ コンパイラにおける実装でも，数多くの最適化処理を検討し実現している．以下，我々が実際に実現した最適化の中で特に重要と思われるものの概要を示す．

6.3.1 分岐先コード呼び出しの最適化

4.1 節で指摘した通り，マッチが成功した時に実行される分岐先の式を関数として実現すると，関数クロージャの生成と関数呼び出しのオーバヘッドを伴う．ML 系言語において，パターンマッチングは最も基本的な制御機構であり，if 文などの条件分岐もパターンマッチングで表現されていることを考慮すると，このオーバヘッドは無視できないものである．本実装では，コードの大きさを考慮したインライニングと併用することによってこの問題を軽減している．第 4.3 で示した例に対して実際に SML♯ コンパイラが生成するコードは図 17 に示すものである．

この例からわかる通り，各ブランチ式が小さいかまたは一カ所にしか現れなければ，最適なコードを実現できる．しかしながらより大規模なプログラムの場合，このような単純な戦略では，関数生成と適用のオーバーヘッドを完全に取り除くことはできない．この問題の解決は重要な今後の課題である．これについては第 8 節で取り上げる．

```
let a = {1 = 2, 2 = B5} in
  let a1 = #1 a in
    switch a1 of
      1 => let a2 = #2 a in
             case a2 of
               A a3 => switch a3 of
                         2 => 3
                       | _ => raise Match
             | B x => x
             | _ => raise MatchCompBug
           end
    | _ => let a4 = #2 a in
             case a4 of
               B x => x
             | _ => raise Match
           end
  end
end
```

図 17 SML♯ が実際に生成する最適化されたコード例

6.3.2 不必要な木の拡張の抑制

木 $T:\sigma$ が表す項集合 $\overline{T:\sigma}$ は $[\![\sigma]\!]$ の部分集合である．したがって $\overline{T:\sigma}$ が $[\![\sigma]\!]$ と一致する時，木 T はこれ以上拡張の余地のないものである．4.2 節で示した木拡張アルゴリズム $\mathcal{E}$ は，このような拡張する余地のない木に対しても拡張を試みる．この冗長性を取り除くために，以下の修正を加える．木 $T:\sigma$ について，$\overline{T:\sigma} = [\![\sigma]\!]$ の時，木 T が「閉じている」と表現する．木の各中間ノードに，木が閉じているか否かを示すクローズドフラグを追加し，$\mathcal{E}$ は，拡張して得られた木の全ての部分木が閉じている時，木のルートノードのクローズドフラグをセットする．また，$\mathcal{E}$ は受け取った木が閉じていた場合，つまり木のルートノードのクローズドフラグがセットされた場合，それ以上拡張を行わないように修正を加える．以上の修正

によって，不必要な木の拡張を一部抑制できる．

6.3.3 決定木の並列拡張

決定木の拡張アルゴリズム $\mathcal{E}$ は，決定木を 1 つのパターンで拡張する．パターンマッチングコンパイラは，与えられた n 個のパターンを含む場合分構文に対して，各パターンを順番に拡張していく処理を行う．しかし，このような逐次的な処理では，すでに拡張された決定木を繰り返したどる処理が発生しうる．この問題を理解するために，以下のような場合分構文を考えてみよう．

```
case exp of
     (1,(2,3),(true,x)))) => E1
      ⋮
  | (1,(2,3),(false,y)))) => E2
      ⋮
```

以下，簡単のために，ワイルドカードパターンおよび変数パターンはあらかじめ処理され，それらは含まれていないと仮定する．この構文を本論文で定義したパターンマッチングコンパイラによって処理すると，まず，パターン (1,(2,3),(true,x)))) によって空の決定木が拡張され，その拡張された木に対してさらに拡張が行われていく．パターン (1,(2,3),(false,y)))) の処理では，すでに存在する 1，2，3 に対するテストノードをたどり，その子ノードに対して false テストを行うノードを付け加える処理を行うことになる．しかしながら，この二つのパターンが共通部分を含むことを考えると，存在する 1，2，3 に対するテストノードをたどる処理は，これら二つのパターンによる拡張を同時に行っていけば避けることができる冗長な処理である．

この冗長性を取り除き，パターンマッチングコンパイルの処理効率を高めるために，実装したシステムでは，$\mathcal{E}$ アルゴリズムを，以下の方針に従い，複数のパターンのリストを一度に平行して処理するアルゴリズムに改良している．

1. 与えられたパターンのリストの先頭のノードを決定する．
2. それと同一のノードをもつパターンをすべて抜き出し，先頭ノードを作成する処理を行い，継続パターンを計算する．
3. 継続パターンのリストに対して再帰的に処理を行う．

同一の先頭ノードをもつパターンの表示的意味はそれ以外のパターンとは共通部分を持たないことに注意すると，上記の処理は，本論文で与えた逐次処理の意味論を保存すること，したがって，この並列処理の結果作成される決定木は，網羅性や冗長性のチェックの結果を含め同一であることを容易に確かめることができる．この改良によって，一度の再帰的な処理によって，すべてのパターンを平行に処理することが可能となった．

7 更なる課題

実用的なパターンマッチングコンパイラを構築する上で課題は数多く残されているが，その中で最も重要と思われるものはマッチが成功した時実行される式 (決定木における葉の位置にある式 e) の呼び出しの最適化である．この問題をほぼ完全に解決するための我々の戦略は，式 e をその自由変数を引数とする関数にコンパイルする代わりに，自由変数を含む式そのものを束縛する機構とそれを利用する機構をターゲット言語に入れるというものである．

式 e が呼び出されるすべての場合で，e に含まれる自由変数の動的な束縛環境は同一のはずである．そこで，e を利用するコードは，それぞれの時点で束縛を生成した後，関数呼び出しのように束縛環境をスイッチすることなく，現在の環境のまま e のコードにジャンプすればよいはずである．このジャンプにより e の自由変数の環境を与えるので，項の等式理論の観点から考察すると，e のコードへのジャンプは，式 e の呼び出しを，e 自身で置き換える機構と見なすことができる．この機構を実現するために，ターゲット言語に以下の特殊な式を追加する．

$$\texttt{letterm}\ X\ \texttt{=}\ e_1\ \texttt{in}\ e_2[X]$$

ここで $e[\ ]$ は式に穴 $[\ ]$ が開いたものであり，ラムダ計算の文献では式の文脈と呼ばれるものである．$e[X]$ は文脈の穴に X を埋め込んで得られる式である．値を束縛して式を評価する ML における let 式とは異なり，letterm 式は X を式 e_1 で束縛して $e_2[X]$ を評価

する．e_1 内の自由変数は e_2 内の X の文脈において捕捉，束縛される．したがってこの式は，直感的には，式 $e_2[e_1]$ を意味する．この機構を含む型付き計算系の基礎は，Hashimoto と Ohori による文脈計算 [6] に求めることができる．この文献では，letterm に用いた文脈 $\delta X.e$ と穴埋め操作 $e1@e2$ を含んだ言語に対して型システムと操作的意味論を定義し，型の健全性が示されている．この文献では，一般的な評価機構を実現するために，複雑な制御機構が必要になっているが，我々のコンパイルアルゴリズムが生成する `letterm` に限定すれば，必要な処理は，各文脈変数 X が現れる時の変数の束縛順序の同等性の保証と同一の束縛環境下でのコンテクストスイッチを伴わないジャンプとリターン機構のみである．SML♯ のパターンマッチングコンパイラは，すでにこの機構を実装しているが，サブルーチン呼び出し機構を実現するコード生成と実行時処理系は現在構築中である．

もちろん，同様の静的なジャンプが表現できる機構であれば，他の機構でも代用することは可能である．例えば[9] では，ターゲット言語のなかで静的なジャンプを一種の静的な例外処理機構として表現し同様の効果を実現している．本研究で文脈計算を用いた主な動機は，堅牢な型主導のコンパイル実現である．SML♯ では，パターンマッチングコンパイルに続き，多段階の型主導コンパイルを行う必要がある．そのために，パターンマッチングコンパイルの出力言語に対しても，その操作的意味のみならず，その型システムが明確に定義され，その型と操作的意味の関係が確立していることが望まれる．

8 結論

本論文では，パターンマッチングとそのコンパイルを系統的に理解するための表示的意味論を提示し，それを基に，パターンマッチング構文を効率よいコードにコンパイルする系統的なアルゴリズムを提案し，その正しさを証明した．このアルゴリズムは，分岐の選択に関して正しいばかりでなく，実行時の値のテスト回数に関して最適であり，かつ，冗長なパターンや網羅的でないパターン集合を常に正確に検出することが保証されている．以上構築したアルゴリズムは，Standard ML のパターン言語のフルセットに対して実装され，Standard ML の拡張言語である SML♯ 言語コンパイラのパターンマッチングコンパイラモジュールとして使用されており，その実用性が確認されている．実装したコンパイラは[14] に公開されている．パターンマッチングコンパイルの結果は，`--xprintRC=yes --xswitchTrace=yes` のスイッチとともにコンパイラを起動することによって確認することができる．

我々の経験によれば，本論文で報告したパターンマッチングコンパイラは，その処理の複雑さにも関わらず，SML♯ コンパイラの中でも群を抜いた堅牢さを示している．例えば，SML♯ コンパイラのほぼ 2 年間にわたる稼働の中で発見された 242 件のバグの中でマッチコンパイラのバグはたった 1 件のみであり，それも SML♯ に特有な多相型レコードパターンの処理に関するものである．この堅牢さは，パターンマッチングコンパイラが，本論文で示された系統的で細部まで正しさが確認されたアルゴリズムに基づき忠実に実装されたことに依ると考える．

本論文で報告したパターンマッチングコンパイル方式とその実装方法は，前節で論じた課題の解決などを通じてさらに改良を加えていけば，パターンマッチングを含む関数型言語の最適化コンパイラの系統的な実現に貢献する技術となると期待される．

謝辞

本論文を丁寧に読んで頂き，アルゴリズムの細部に関する表記の不正確さの指摘や他の処理系における関連する最適化の扱いなどを含む数多い有益な指摘をして下さった査読者に深謝致します．

参 考 文 献

[1] Aitken, W.: SML/NJ match compiler notes, Standard ML of New Jersey, 1992.

[2] Appel, A. W. and MacQueen, D. B.: A Standard ML Compiler, in *Proc. Conference on Functional Programming Languages and Computer Architecture*, Vol. 274, Springer, Berlin, 1987, pp. 301–324.

[3] Augustsson, L.: Compiling pattern matching, in *Functional Programming and Computer Architechture.* ACM, 1985.

[4] Baudinet, M. and MacQueen, D.: Tree pattern matching for ML, Draft, AT& T Bell Laboratories, 1985.

[5] Cardelli, L.: Compiling a functional language, in *Proc. International Conferecen on LISP and Functional Programming*, 1984, pp. 208–217.

[6] Hashimoto, M. and Ohori, A.: A typed context calculus, *Theoretical Computer Science*, Vol. 226, No. 1–2(2001), pp. 352–366.

[7] Hudak, P., Jones, S. P., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzman, M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W. and Perterson, J.: Report on programming language Haskel a non-strict, purely functional language version 1.2, *SIGPLAN Notices, Haskell special issue*, Vol 27, No. 5, 1992.

[8] Le Fessant, F. and Maranget, L.: Optimizing pattern matching, in *Proc. International Conference on Functional Programming*, 2001, pp. 26–37.

[9] Leroy, X.: The ZINC experiment: an economical implementation of the ML language, Technical Report 117, INRIA, 1992.

[10] Leroy, X.: *The Objective Caml User's Manual*. INRIA Rocquencourt, France, 1997.

[11] Milner, R., Tofte, M. and Harper, R.: *The Definition of Standard ML*, The MIT Press, 1990.

[12] Puel, L. and Suarez, A.: Compiling pattern matching by term decomposition, *J. Symbolic Computation*, Vol. 1(1993), pp. 1–26.

[13] Scott, K. and Ramsey, N.: When do match-compilation heuristics matter? Technical report CS-2000-13, Dept. of Computer Science, University of Virginia, 2000.

[14] SML# home page, http://www.riec.tohoku.ac.jp/smlsharp/, 2006.