修 士 論 文

型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

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

纓坂 智

2004 年 3 月

修 士 論 文

型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

指導教官　大堀淳 教授

審査委員主査　大堀淳教授
審査委員　田島敬史 助教授
審査委員　片山卓也 教授

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

210016 纓坂 智

提出年月: 2004 年 2 月

概 要

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型理論的な基礎を確立する．まず，パターンを項の部分集合，パターンマッチングをマッチングを行う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表示的意味論を与える．次に，この表示的意味論を表現する木構造を定義し，アルゴリズムを導出する．このアルゴリズムが定義した表示的意味論に関して正しいことを証明する．この証明により，アルゴリズムに新たな機構を導入することなく，アルゴリズムが冗長なパターンや網羅的でないパターン集合を検出できるものであることを示せる．本論文の第二の目的は，以上の型理論的基礎に基づき，実用的なパターンマッチングコンパイラを実装することである．Standard ML のパターン言語のフルセットに対してコンパイルアルゴリズムを構築し，いくつかの高速化技術を開発した．実装したパターンマッチングコンパイラは，現在 JAIST において開発中の次世代 ML コンパイラの一部となる予定である．

目 次

第1章　序論

1.1　パターンマッチング

パターンマッチングは ML[RMH90] や Haskell[eaeH92]，OCaml[Ler97] などの関数型言語が持つ機能の一つである．データの形をパターンを用いて記述することによって，様々なデータ構造に関する分岐やデータの利用を容易にする機能である．

パターンマッチング式は，マッチングの対象となる式と，パターンと式の組 (これをルールと言う) のリストで構成される．パターンマッチングは以下の順番で実行される．まず，マッチングの対象となる式を評価する．評価によって得られたデータを，ルールリスト中の先頭のパターンから順にマッチングを行い，最初にマッチングが成功したパターンと組である式を実行する．Standard ML で定義されているパターンには，各種のデータ構造に対応するパターンと，ワイルドパターン _，変数パターン x がある．ワイルドパターンと変数パターンは任意のデータ構造とマッチし，変数パターンは変数を対応するデータで束縛する．例えば，以下の Standard ML の文法によるパターンマッチングを実行するコードでは，

```
let
  val x = (1, 2)
in
  case x of
    (1, 1) => 2
  | (y, 2) => y + 1
  | _      => 3
end
```

x は二番目のパターン (y，2) と最初にマッチする．したがって二番目のパターンと組である式 y + 1 が実行される．このとき，変数パターン y によって，変数 y は対応するデータ 1 に束縛されている．したがってこのコードの評価結果は 2 となる．また，パターンマッチングが必ず成功するパターン集合のことを「網羅的である」と表現し，いかなる場合もマッチすることのないパターンのことを「冗長である」と表現する．上記のパターンマッチング式は，三番目のパターンがワイルドパターンであり，いかなる場合もパターンマッチングは成功するため，これらによるパターン集合は網羅的であり，また，いかなる場合もマッチすることがないパターンはないため，冗長なパターンはない．

1.2　パターンマッチングコンパイル

パターンマッチングは，構造を持たない基本的なデータだけでなく，直和やプロダクトなど，構造化したデータに関しての場合分け機能も提供する．パターンマッチングコンパイラは，こうした複雑なデータに対する分岐を実現するために，基本データの比較，直和のタグによる比較，プロダクトの各フィールドの取り出しといった，単純な操作へとパターンマッチング式をコンパイルする．例えば，下記のコードは，

上記のパターンマッチング式をコンパイルした例である．

```
let
  val x = (1, 2)
in
  let val x1 = #1 x in
    switch x1 of
      1 => let val x2 = #2 x in
             switch x2 of
               1 => 2
             | 2 => let val y = x1 in y + 1 end
             | _ => 3
           end
    | _ => let val x2 = #2 x in
             switch x2 of
               2 => let val y = x1 in y + 1 end
             | _ => 3
           end
  end
end
```

パターンマッチングコンパイルは，関数型言語の各コンパイルフェーズの中でも複雑なものであり，現在までに様々な方式が提案されている．その代表的なものに，バックトラックオートマトンによる方式と，決定木を作成する方式の二つがある．

1.2.1 バックトラックオートマトン

パターンマッチングコンパイルに用いられている主な方式に，バックトラックオートマトンを生成するものがある [Aug85, Ler92, FM01]．OCaml が採用している方式である．この方式では，分割統治法によってパターンマッチングコンパイルを行う．コンパイルによって生成されたコードは，再帰的に分割された部分的なパターンとのマッチングを行い，マッチングが失敗したら「バックトラック」してマッチングを実行していない他の部分的なパターンとのマッチングを試みる．この方式は再帰的な構造であるため，コンパイラを容易に実装できる．また，以下で記述する決定木モデルと違いパターンのコピーを生成しないため，コンパイル後のコードサイズは小さい．この方式の欠点は，バックトラックを行うとこれまでに実行したテストと同一のテストをくり返すため，コードの実行効率が悪いことである．また，原理的にパターンの冗長性や網羅性の検出を厳密に行うことはできない．そのため，この方式による生成コードはいかなる場合も実行されることのない「デッドコード」を含む可能性があり，また，パターン集合の網羅性を確実に判定するためには別の機構を導入する必要がある．

1.2.2 決定木モデル

パターンマッチングコンパイルにおいて採用されているもう一つの主要な方式に，決定木を作成するものがある [Car84, BM85, Ait92]．Standard ML of New Jersey[AM87] や Haskell がこの方式を採用してい

る．この方式では，コンパイラは項とマッチするパターンを決定する決定木を作成する．同一のテストを二回以上行わないないため，生成後のコードの実行効率はよい．また，パターン集合の冗長性や網羅性を検出できるものである．しかし，既存の研究が対象としている決定木モデルは，一般的な計算モデルであり最小限のものに対する方針を示しているものの，実用的な言語のパターンマッチングコンパイラを実装するには不十分なものであった．様々なパターンを持つ実際の言語においてどのような決定木をどのように構築すればよいのか，実際のコンパイラが検出すべきパターンの冗長性やパターン集合の網羅性をどのように検出するのかといったことは不明瞭であり，また，構築した決定木の正しさも示されてはいなかった．

本研究の成果は，決定木を用いた手法が抱えるこうした問題に対して一つの答えを示すものである．

1.3 本研究の目的と方針

本研究の目的は，パターンマッチングコンパイルの型理論的な基礎を確立し，実用的なパターンマッチングコンパイラを実装することである．目的を達成するための方針を以下に示す．

パターンマッチングの表示的意味論の定義 本研究では，パターン集合を項の全集合を分割するもの，パターンマッチングを項が含まれる部分集合を決定する機構であるとみなし，パターンマッチングの表示的意味論を定義する．表示的意味論を定義することにより，本研究が構築するアルゴリズムの正しさや，必要な諸性質をアルゴリズムが満たしていることを証明できる．

実用的なパターンマッチングアルゴリズムの構築 表示的意味論を表現する木構造を定義し，アルゴリズムを導出する．さらに，効率よいコードを生成するため，また，実用的なパターンマッチングコンパイラへの実装を容易にするために，アルゴリズムに改良を加える．

アルゴリズムの正しさと諸性質の証明 アルゴリズムによって作成した決定木が，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．これによって，アルゴリズムの正しさと，アルゴリズムが，パターンの冗長性や網羅性を検出できることを示す．

アルゴリズムの実装 構築したアルゴリズムを実装する．実装の対象は Standard ML のパターン言語フルセットであり，さらにオアパターンへの対応も加え，拡張可能な方式とする．また，コンパイルの高速化や効率のよい生成コードの生成など，いくつかの最適化技術を導入する．実装するパターンマッチングコンパイラは，現在 JAIST において開発中の次世代 ML コンパイラの一部となる予定であり，ML コンパイラに組込み可能なモジュールとして構築される．

1.4 論文の構成

本論文は以下のように構成される．第 2 章では，パターンマッチングを型に基づいて検証し，パターンマッチングの表示的意味論を定義する．第 3 章では，項集合の木表現を定義し，その定義をしようしてパターンマッチングコンパイルアルゴリズムの基礎を構築する．第 4 章では，第 3 章によって示された決定木，コンパイルアルゴリズムに効率化を図り，実装に耐え得る効率のよいデータ構造とコンパイルアルゴリズムを示す．第 5 章では，第 4 章で構築したアルゴリズムが表示的意味論に関して正しいことを示す．第 6 章では第 4 章で示したアルゴリズムを実装する．第 7 章では関連研究との比較を行う．第 8 章では本論文の結論を述べる．

第2章　パターンマッチング

本論文は型付けされた正格言語 (typed strict language) を対象とし，パターンマッチング中の各パターンは線形パターンであるとする．正格言語とは値による関数呼び出しを行う言語である．線形パターンとは，一つのパターン中に同じ変数が二回以上現れないものである．本章では，本論文が対象とするパターンマッチングに関する種々の定義を示し，パターンマッチングの表示的意味論を定義する．

2.1　型，項，パターン

本論文が対象とするパターンマッチングに関する種々の定義を示す．

以下のような型を考える．

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

t は任意の型を代表する型を表す．関数など，パターンマッチングではその内部構造について言及しないデータ構造に対する型である．b は等価性テストが定義された基本型である．$\tau + \ldots + \tau$ は直和型であり，ML のデータタイプに対応する．$\tau * \ldots * \tau$ はプロダクト型であり，ML の組やレコードの型に対応する．

上記の型に属する項を以下のように定義する．項とは，パターンマッチングの対象となる式が実行時に持ちうる値のことである．

$$v ::= t \mid c \mid i(v) \mid (v, \ldots, v)$$

t は型 t 自身の項である．パターンマッチングでは型 t の項の内部構造については言及しないため，これで十分である．c は型 b の項である．$i(v)$ は直和型 $\tau_1 + \ldots + \tau_n$ の項であり，v を直和の i 番目の要素へと埋め込んだものである．$(v, \ldots, v)$ はプロダクト型 $\tau_1 * \ldots * \tau_n$ の項である．

本論文では説明を簡潔にするために，以下の仮定を行う．基本型にはただ一つ b のみがあり，型 b を持つ項の集合を $\{c_1, c_2, \ldots\}$ とする．直和 $i(v)$ の型は，明示せずとも $\tau_1 + \ldots + \tau_n$ であると仮定する．これらの仮定は，以下に展開する枠組において，その他実際の言語に含まれる種々の型を導入する上で問題ないものである．

次にパターンを定義する．

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

c は定数，_ はワイルドカード，x は変数，$i(P)$ は i 番目への埋め込みの直和，$(P, \ldots, P)$ はプロダクトを表すパターンである．

パターンマッチング式は以下の形で記述される．

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

本論文では型つき言語を対象としているため，e はある型 τ を持つ．したがって各パターン P_i も同じ型を持つ．パターンとその型を明示する場合，$P : \tau$ と表記する．型が明示されていない場合も，パターンは型 τ を持つ．

2.2 パターンマッチングの意味論

本論文が用いるアプローチは，まずパターンやパターンマッチングの意味論を定義し，そこからコンパイルアルゴリズムを引き出すことである．パターンマッチングの意味論を定義するために必要な定義を以下に示す．

$[\![\tau]\!]$ を以下のように定義する．$[\![\tau]\!]$ は型 τ に属する全ての項の集合を表す．

$$
\begin{aligned}
[\![t]\!] &= \{t\} \\
[\![b]\!] &= \{c_1, c_2, \ldots\} \\
[\![\tau_1 + \ldots + \tau_n]\!] &= \{i(a) \mid a \in [\![\tau_i]\!], 1 \leq i \leq n\} \\
[\![\tau_1 * \ldots * \tau_n]\!] &= \{(a_1, \ldots, a_n) \mid a_i \in [\![\tau_i]\!]\}
\end{aligned}
$$

パターン $P : \tau$ の意味論を下記のように定義する．

$$
\begin{aligned}
[\![\_ : \tau]\!] &= [\![\tau]\!] \\
[\![x : \tau]\!] &= [\![\tau]\!] \\
[\![c : b]\!] &= \{c\} \\
[\![i(P) : \tau_1 + \ldots + \tau_n]\!] &= \{i(a) \mid a \in [\![P : \tau_i]\!]\} \\
[\![(P_1, \ldots, P_n) : \tau_1 * \ldots * \tau_n]\!] &= [\![P_1 : \tau_1]\!] \times \ldots \times [\![P_n : \tau_n]\!]
\end{aligned}
$$

上記の定義を用いると，パターンマッチングの意味論を定義できる．まず以下のようなパターンマッチング式を考える．

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

e の型は τ であると仮定する．以下のように集合 Y_i および X_i を定義する．

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

Y_i は $P_1, \ldots, P_i$ が表す項の部分集合である．X_i は i 番目のルールにマッチングする項の部分集合である．したがってパターンマッチングの意味は，$[\![e]\!] \in X_i$ なるルールを選択し，P_i 中の変数パターンによる変数束縛を伴ってブランチ e_i を実行することである．さらに，上記の定義よりパターンの冗長性，マッチングの網羅性も判定できる．もし，ある X_i について $X_i = \emptyset$ なら対応するパターン P_i は冗長である．もし，$Y_n \neq [\![\tau]\!]$ なら，パターン集合は網羅的でない．

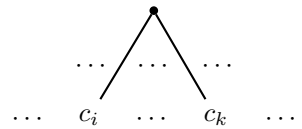
第3章　項集合の木表現

前章で示したように，パターンマッチングではマッチングを行う項が属する部分集合 X_i を決定する必要がある．

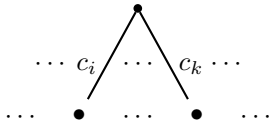
しかし前章で示した表示的意味論は，その効率よい決定方法を導き出すものではない．そこで本章では，効率良い決定方法を導き出すために，前章で定義した様々な項の部分集合を表現する木構造を考える．本章で定義する木は，その構造から、効率よく X_i を決定できる決定木と考えることができる．本章ではさらに，決定木を構築するアルゴリズムを示す．

3.1　型に属する項集合の木表現

まず型 τ に属する項の全体集合 $[\![\tau]\!]$ の木表現を考える．例えば，$[\![b]\!]$ は以下のような無限の枝を持つ木として表現できる．

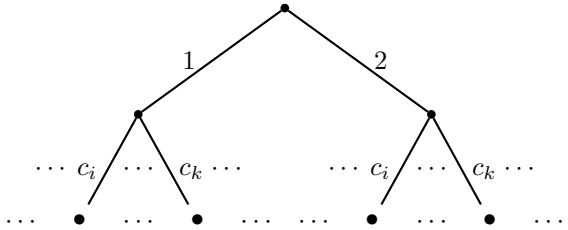


この木は，下記のような木へと修正することで，定数による分岐を表す決定木と考えられる．



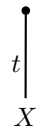
ラベル c_i がついた枝は，項が定数 c_i であるときの分岐と考え，葉 $\bullet$ は分岐後に実行する何かの動作と考える．直和やプロダクトの決定木であれば，さらなる分岐を表す決定木が葉に接続される．

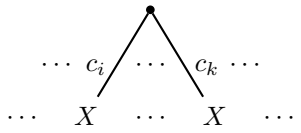
例えば $[\![b+b]\!]$ は，下記のような木として表現できる．

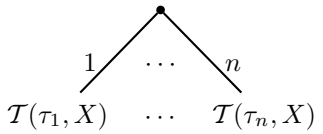


この木は，タグによる分岐を表す決定木のそれぞれの葉に，定数の決定木が関連付けられたものである．この木によって，$b+b$ 型の項による分岐を表現できる．例えば項が $1(c_i)$ であるなら，根から左の枝へ分岐し，さらにラベル c_i を持つ枝へと分岐する．

上記した直和の例のように，直和やプロダクトなど構造化した項の集合に対応する決定木は，葉にさらなる決定木が関連付けられたものとなる．この構造の表現するため，決定木の定義は葉に関連付けるものによってパラメータ化されたものとなる．$[\![\tau]\!]$ を表現し，さらにそのすべての葉が X である決定木を，$\mathcal{T}(\tau, X)$ と表記する．$[\![\tau]\!]$ を表す決定木は $\mathcal{T}(\tau, \bullet)$ である．例に挙げた $[\![b]\!]$ と $[\![b+b]\!]$ に対応する決定木は，それぞれ $\mathcal{T}(b, \bullet)$ と $\mathcal{T}(b+b, \bullet)$ である．

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

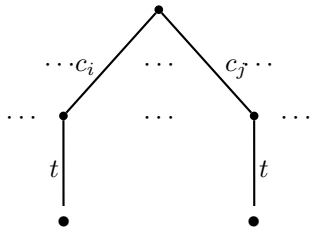
$\mathcal{T}(b, X) =$ 

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

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

図 3.1: 型 τ に属する項の全集合の木表現

図 3.1 に $\mathcal{T}(\tau, X)$ の定義を示す．例えば，$\mathcal{T}(b * t, \bullet)$ は以下のような木となる．



3.2 パターンが表す項集合の木表現

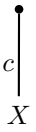
次に，型 τ のパターン P が表す項の集合 $[\![P : \tau]\!]$ の木表現を考える．$[\![P : \tau]\!]$ を表し，その全ての葉が X である決定木を $\mathcal{P}(P : \tau, X)$ と表記する．図 3.2 に $\mathcal{P}(P : \tau, X)$ の定義を示す．

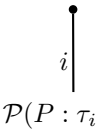
例えば，パターン (c_1, c_2) と $(\_, c_2)$ が意味する項の部分集合の木表現はそれぞれ以下のように表現され，

$$\begin{aligned} \mathcal{P}((c_1, c_2) : b * b, \bullet) &= \mathcal{P}(c_1 : b, \mathcal{P}(c_2 : b, \bullet)) \\ \mathcal{P}((\_, c_2) : b * b, \bullet) &= \mathcal{P}(\_ : b, \mathcal{P}(c_2 : b, \bullet)) = \mathcal{T}(b, \mathcal{P}(c_2 : b, \bullet)) \end{aligned}$$

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

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

$\mathcal{P}(c:b, X) = \quad$ 

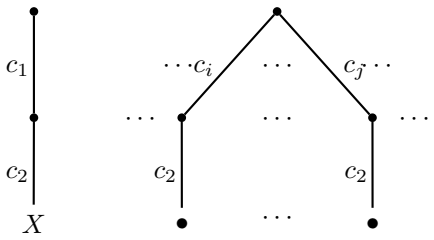
$\mathcal{P}(i(P):\tau_1 + \cdots + \tau_n, X) =$ 

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

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

図 3.2: パターンが表す項の部分集合の木表現

以下のような木となる．



3.3 パターンマッチングコンパイルの基本原理

上記の木を用ることで，パターンマッチングコンパイルの基本原理を確立できる．
パターンマッチング式を

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

とし，X_i と Y_i を 2.2 節で定義した集合とする．2.2 節で定義したように，パターンマッチングでは $[\![e]\!] \in X_i$ なる e_i の選択を行う．したがって $\mathcal{T}(\tau, \bullet)$ において，各 X_i を表す部分木の葉に e_i をラベル付けした決定木を作成すればよい．

この決定木は以下の手順で作成できる．

- 木 $T = \mathcal{T}(\tau, \bullet)$ を作成する．
- $i \in \{1, \ldots, n\}$ について，この順番で以下の作業を行う．

- 木 $T_i = \mathcal{P}(P_i : \tau, e_i)$ を用意する．
- T の T_i と一致する部分木について，$\bullet$ でラベル付けされた葉を新たに e_i でラベル付けする．

また，この手順で作成された木 T によって，パターンマッチングの冗長性と網羅性を以下のように検出できる．

- もし，T に e_i でラベル付けされた葉がなかった場合，P_i は冗長なパターンである．
- もし，T に $\bullet$ でラベル付けされた葉があった場合，パターンマッチングは網羅的でない．

上記の処理では，T_i によって T をラベル付けした．このラベル付けは，次節で定義するアルゴリズム $\mathcal{L}$ によって実現できる．

3.4 木のラベル付けアルゴリズム

前節で使用したラベル付けアルゴリズムを $\mathcal{L}$ とする．木 $T_{P:\tau}$ を $\mathcal{P}(P : \tau, e)$，木 T_τ を $\mathcal{T}(\tau, \bullet)$ に 0 回以上アルゴリズム $\mathcal{L}$ を適用して部分的にラベル付けされた木とする．アルゴリズム $\mathcal{L}$ はこの二つの木 $T_{P:\tau}$ と T_τ を受け取り，T_τ の $T_{P:\tau}$ と一致する部分木について，$\bullet$ でラベル付けされた葉を対応する $T_{P:\tau}$ の葉でラベル付けするものである．

木の構造は再帰的であるため，部分木の探索および葉 $\bullet$ のラベル付けも再帰的に定義できる．図 3.3 は，$T_{P:\tau}$ を $\mathcal{P}(P' : \tau', X)$ と表記した場合の，$T_{P:\tau}$ による T_τ のラベル付けアルゴリズム $\mathcal{L}(\mathcal{P}(P' : \tau', X), T_\tau)$ の定義である．図において P' が変数パターン，ワイルドパターン，およびプロダクトパターンにおけるラベル付け規則は，$\mathcal{P}$ の定義によって P' を式変形しているだけであり $\mathcal{L}$ の定義としては不要であるが，次章で示す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上で必要なものである．アルゴリズム $\mathcal{L}$ の説明を以下に示す．

$T_{P:\tau} = e$ **の場合．** 木が葉の場合である．この時 T_τ は $\bullet$ でラベル付けされた葉であるか，すでに別のラベル e' でラベル付けされた葉であるかのいずれかである．$\bullet$ でラベル付けされた葉であれば e でラベル付けした葉 e を返す．そうでなければ T_τ を返す．

$T_{P:\tau} = \mathcal{P}(c : b, X)$ **の場合．** $\mathcal{P}$ の定義より，$\mathcal{P}(c : b, X)$ は部分木 X が枝 c によって根に接続された木である．したがって T_τ から枝 c を持つ部分木 X' を探しだし，再帰的に X' を X でラベル付けする．

$T_{P:\tau} = \mathcal{P}(i(P) : \tau_1 + \ldots + \tau_n, X)$ **の場合．** $\mathcal{P}(c : b, X)$ の場合と同様に，T_τ から枝 i を持つ部分木 X' を探しだし，X' を $\mathcal{P}(P : \tau_i, X)$ でラベル付けする．$\mathcal{T}$ の定義より，X' は $\mathcal{P}(P : \tau_i, X)$ と同じ構造を持つはずである．

$T_{P:\tau} = \mathcal{P}((P_1, \ldots, P_n) : \tau_1 * \ldots * \tau_n, X)$ **の場合．** $\mathcal{P}$ の定義より，$\mathcal{P}((P_1, \ldots, P_n) : \tau_1 * \ldots * \tau_n, X)$ を $\mathcal{P}(P_1 : \tau_1, \mathcal{P}((P_2, \ldots, P_n) : \tau_2 * \ldots * \tau_n, X))$ へと式変形し，これを用いてラベル付けする．これは，まず T_τ の先頭の部分木と P_1 が表す部分木との一致を図り，一致した全ての部分木について $(P_1, \ldots, P_n)$ および X を使用して木の一致とラベル付けを行うことを意味する．

$T_{P:\tau} = \mathcal{P}(\_ : b, X)$ **の場合．** $\mathcal{P}$ の定義より，$\mathcal{P}(\_ : b, X)$ の根は T_τ と同様に，b の全ての定数による分岐を持つ．したがって T_τ の全ての部分木にラベル付けを行う．

$T_{P:\tau} = \mathcal{P}(\_ : \tau_1 + \ldots + \tau_n, X)$ **の場合．** $\mathcal{P}(\_ : b, X)$ の場合と同様に，T_τ の全ての部分木にラベル付けを行う．

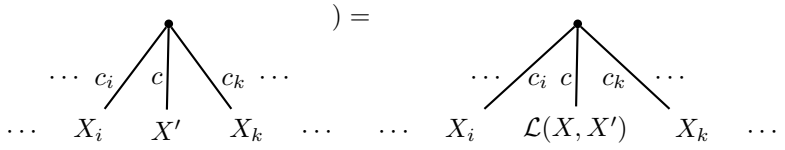
$T_{P:\tau} = \mathcal{P}(\_ : \tau_1 * \ldots * \tau_n, X)$ **の場合．** ワイルドパターン $\_ : \tau_1 * \ldots * \tau_n$ を $\_ : \tau_1$ と残り $\_ : \tau_2 * \ldots * \tau_n$ とに分割して考える．つまり，$\mathcal{P}(\_ : \tau_1 * \ldots * \tau_n, X)$ を $\mathcal{P}(\_ : \tau_1, \mathcal{P}(\_ : \tau_2 * \ldots * \tau_n, X))$ へと式変形し，まず T_τ の τ_1 に対応する部分木に関して一致を図り，その後 $\tau_2 * \ldots * \tau_n$ および X に対応する部分木に関して一致を図り，ラベル付けする．

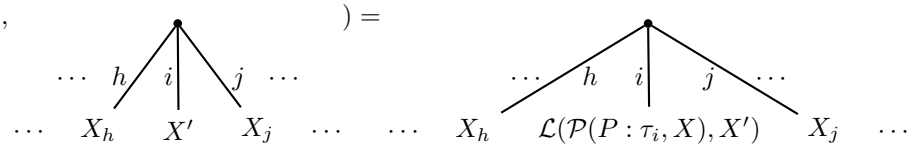
このアルゴリズムによってラベル付けされた木の例を図 3.4 に示す.

$\mathcal{L}(e, \bullet) = e$

$\mathcal{L}(e, e') = e'$

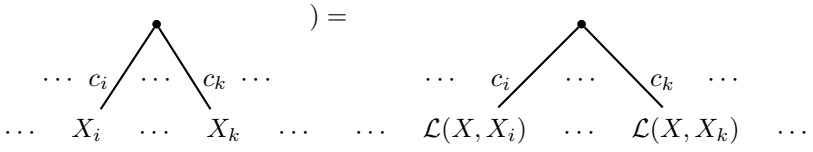
$\mathcal{L}(\mathcal{P}(x : \tau, X), T_\tau) = \mathcal{L}(\mathcal{P}(\_ : \tau, X), T_\tau)$

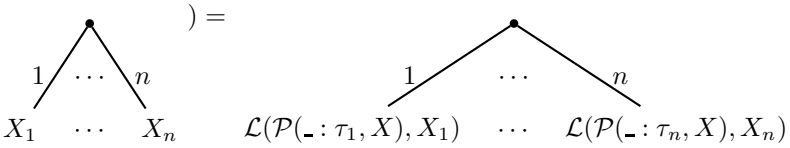
$\mathcal{L}(\mathcal{P}(c : b, X),$  $) =$

$\mathcal{L}(\mathcal{P}(i(P) : \tau_1 + \ldots + \tau_n, X),$  $) =$

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

$\mathcal{L}(\mathcal{P}((P_1, P_2) : \tau_1 * \tau_2, X), T_\tau) = \mathcal{L}(\mathcal{P}(P_1 : \tau_1, \mathcal{P}(P_2 : \tau_1 * \tau_2, X)), T_\tau)$

$\mathcal{L}(\mathcal{P}(\_ : b, X),$  $) =$

$\mathcal{L}(\mathcal{P}(\_ : \tau_1 + \ldots + \tau_n, X),$  $) =$

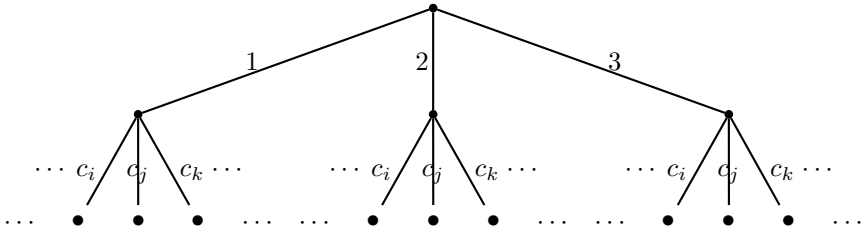
$\mathcal{L}(\mathcal{P}(\_ : \tau_1 * \ldots * \tau_n, X), T_\tau) = \mathcal{L}(\mathcal{P}(\_ : \tau_1, \mathcal{P}(\_ : \tau_2 * \ldots * \tau_n,)), T_\tau)$

$\mathcal{L}(\mathcal{P}(\_ : \tau_1 * \tau_2, X), T_\tau) = \mathcal{L}(\mathcal{P}(\_ : \tau_1, \mathcal{P}(\_ : \tau_1 * \tau_2, X)), T_\tau)$

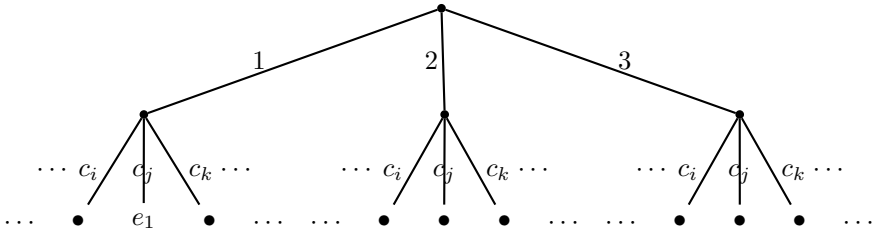
図 3.3: ラベル付けアルゴリズム

$$
\begin{aligned}
T_0 &= \mathcal{T}(b+b+b, \bullet) \\
T_1 &= \mathcal{L}(\mathcal{P}(1(c_j) : b+b+b, e_1), T_0) \\
T_2 &= \mathcal{L}(\mathcal{P}(2(\_) : b+b+b, e_2), T_1) \\
T_3 &= \mathcal{L}(\mathcal{P}(\_ : b+b+b, e_3), T_2)
\end{aligned}
$$

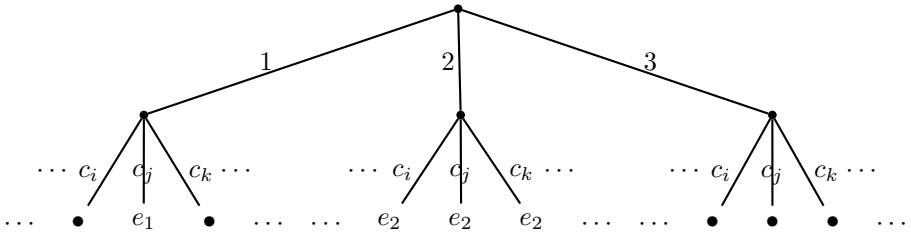
$T_0 =$



$T_1 =$



$T_2 =$



$T_3 =$

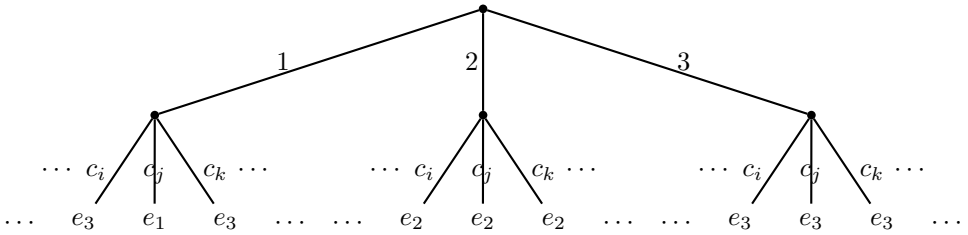


図 3.4: ラベル付けの例

第4章　パターンマッチングコンパイル

本章では，前章にて示された決定木，コンパイルアルゴリズムに効率化を図り，実装に耐え得る効率のよいデータ構造とコンパイルアルゴリズムを示す．

4.1　木の効率のよい表現

前章で示した木のラベル付けアルゴリズム $\mathcal{L}$ は，パターンマッチングを実行する決定木を作成するものであるが，無限の枝を持つパターンが表す木 $T_{P:\tau}$ を用いて，同じく無限の枝を持つ型 τ の木 T_τ をラベル付けするものである．本節では，効率よくこのアルゴリズムを実行するために，$T_{P:\tau}$ および T_τ の効率よい表現を考える．

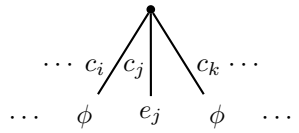
4.1.1　決定木の効率よい表現

アルゴリズム $\mathcal{L}$ は，無限数の枝を持つ木 T_τ に対するものであり，その無限の枝に対する操作を伴うものであった．アルゴリズムの実装を行うには，より効率的な構造の決定木を定義する必要がある．本節では，そのような効率のよい決定木の表現を考える．

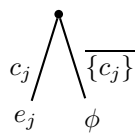
決定木は，部分的にラベル付けされた任意の木を表現でき，かつ枝を有限数で表現し，かつノードの数が少ないものが望ましい．そこで決定木を下記のように変更する．

まず，全ての葉が $\bullet$ である部分木を ϕ に置き換える．これにより，例えば $\mathcal{T}(\tau, \bullet)$ は ϕ と表記でき，木のノードを減らすことができる．

次に，ϕ が接続された枝は一つにまとめる．例えば枝 c_j の葉のみに e_j がラベル付けされ，その他の枝の葉には $\bullet$ がラベル付けされた木

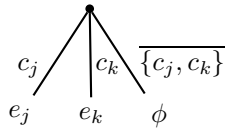


は，ラベル付けされた枝の集合 $\{c_j\}$ の補集合 $\overline{\{c_j\}}$ による枝を用いて，

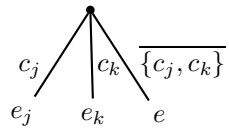


と表記する．これにより枝を有限数で表現できる．上記の木の枝 c_k の葉に新たに e_k をラベル付けする場合

は，枝 c_k を追加して，



という木を作成する．ワイルドパターンによるラベル付けは，ラベル付けされていない枝を補完する働きを持つ．したがってワイルドパターンによって，この木の全ての $\bullet$ でラベル付けされた葉に e をラベル付けする場合は，新たに葉を追加するのではなく補集合 $\overline{\{c_i, c_k\}}$ に対応する枝の葉をラベル付けし，

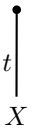


とする．

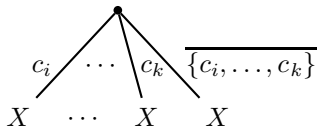
上記の変更を加えた決定木を図 4.1 に示す．この決定木は葉が全て $\bullet$ である部分木は ϕ で表される．そのため，前章での木に対するラベル付けアルゴリズムをこの決定木に対応させると，ϕ で表された部分木に枝を追加して木を拡張する木拡張アルゴリズムとなる．次節ではこの決定木に更なる拡張を行い，4.3 節で木拡張アルゴリズムを示す．

$\mathcal{M}(\tau, \bullet) = \quad \phi$

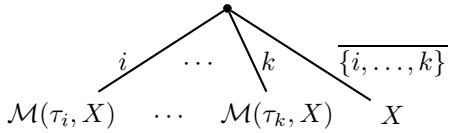
$\mathcal{M}(t, X) =$



$\mathcal{M}(b, X) =$



$\mathcal{M}(\tau_1 + \cdots + \tau_n, X) =$



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

図 4.1: 決定木の効率的な表現

4.1.2 パターンが表す木の効率よい表現

ラベル付けアルゴリズム $\mathcal{L}$ において T_τ のラベル付けに用いられるパターンが表す木の効率よい表現を考える．

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．X 自身も Ps であるから，Ps は以下のような直和として定義できる．

$$Ps ::= e \mid (P:\tau) :: Ps$$

e は葉 e であることを意味し，$(P:\tau) :: Ps$ は木 $\mathcal{P}(P:\tau, Ps)$ を意味する．

一般的に無限の枝を持つパターンが表す木を，このような直和で表すことにより，木を有限な形で効率よく表現でき，この木は，e で終わるパターンのリストと考えることができる．また，$\mathcal{L}$ はパターンが表す木を $\mathcal{P}(P:\tau, X)$ の形で表現したときのラベル付けアルゴリズムを定義したものであるから，$\mathcal{L}$ を Ps によるラベル付けアルゴリズムへと修正することは容易である．

4.2 データ構造の再定義

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，直和やプロダクトなど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．これらの問題に対処するために，4.1.1 節で定義した決定木に以下の三つの修正を加える．

まず各中間ノードに，ノードが対応する項の部分項へとアクセスするアクセスパス 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．次に，プロダクトの各フィールドへとアクセスするために，フィールドに対応する部分木の先頭にノードを追加する．最後に，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．

決定木の作成アルゴリズムを明確に示すために，上記の修正を加えた決定木を，下記のように項表現によって表す．

$$\begin{aligned} T \quad ::= \quad & \phi \\ \mid \quad & \mathtt{eq}(a, \{c_i : T, \ldots, c_k : T\}, T) \\ \mid \quad & \mathtt{tag}(a, \{i{:}\, T, \ldots, k{:}\, T\}, T) \\ \mid \quad & \mathtt{prod}(a, n, i{:}\, T) \\ \mid \quad & \mathtt{univ}(a, T) \\ \mid \quad & \mathtt{leaf}(\Gamma, e) \end{aligned}$$

ϕ は拡張が行われていない，空の木を表す．$\mathtt{eq}(a, \{c_i : T_i, \ldots, c_k : T_k\}, T_0)$ 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．$\mathtt{tag}(a, \{i : T_i, \ldots, k : T_k\}, T_0)$ は直和型のタグに対する分岐を表す．$i, \ldots, k$ はタグを表す整数である．eq の場合と同様に，T_0 は $\overline{\{i, \ldots, k\}}$ による分岐を表す．$\mathtt{prod}(a, n, i{:}\, T_i)$ 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プロダクトの i 番目のフィールドに対応するものであることを表す．$\mathtt{univ}(a, T)$ はワイルドパターンもしくは変数パターンによるノードを表す．$\mathtt{leaf}(\Gamma, e)$ 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 4.2 に決定木とその項表現の対応を示す．

次に前節で定義した Ps をここで以下のような変更を加え，再定義する．

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

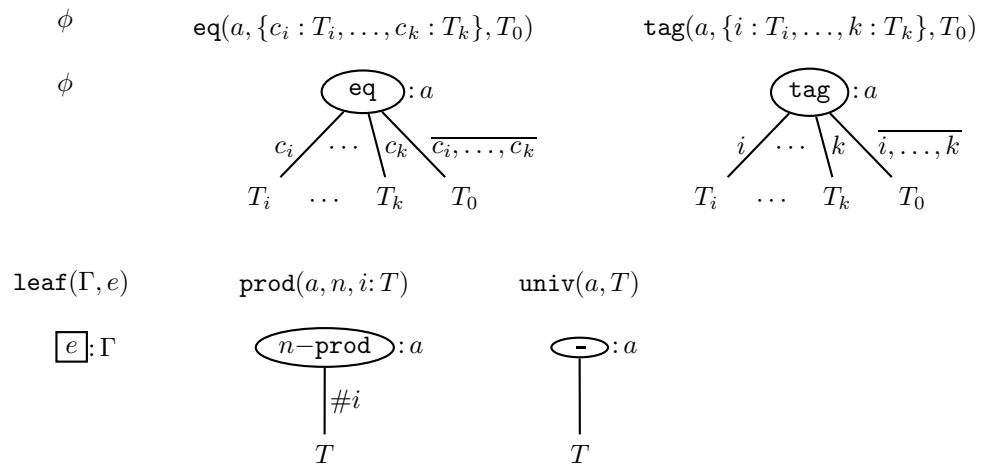


図 4.2: 決定木とその項表現の対応

a はパターン P に対応する部分項へのアクセスパスである．次節で示すアルゴリズムで木を拡張する際に必要となる．

続いて，パターン集合を以下のように再定義する．

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

以前の定義において組として表現していたプロダクトを，整数をラベルとするレコードで表現する．

4.3 木拡張アルゴリズム

前章で定義した木のラベル付けアルゴリズム $\mathcal{L}(\mathcal{P}(P : \tau, X), \mathcal{T}(\tau, X'))$ は，$\mathcal{T}(\tau, X')$ を $\mathcal{P}(P : \tau, X)$ でラベル付けするものであった．$\mathcal{P}(P : \tau, X)$ と $\mathcal{T}(\tau, X')$ は前節でそれぞれ Ps と T へと修正，再定義された．本節では，ラベル付けアルゴリズム $\mathcal{L}$ を基に，Ps を用いて T を拡張するアルゴリズム $\mathcal{E}$ を定義する．

アルゴリズム $\mathcal{E}$ は部分的に拡張された木 T を Ps で拡張する以下の型を持つ関数である．

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

Γ は T に付加する葉が持つ環境である．

図 4.3 にアルゴリズム $\mathcal{E}$ の定義を示す．

図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．

以下に $\mathcal{E}(Ps, \Gamma, T)$ の概要を Ps に関して場合分けして説明する．

$Ps = e$ の場合 この場合，e に対応するパターンによるマッチングが成功したことを意味する．もし葉 T が ϕ であるなら，$\mathtt{leaf}(\Gamma, e)$ を付加する．T がすでに別の葉を持つなら，この場合に置いて 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 に関して場合分けする．

$P = c$ **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-プロダクトの 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．

4.4 ターゲットコード生成アルゴリズム

前節のアルゴリズムによって作成された決定木は，項の値による場合分けや変数束縛などターゲットコードを生成するための全ての情報を含んでおり，決定木からターゲットであるラムダ式への変換は再帰的に容易に行うことができる．しかしこの方針によって生成されたコードは，式の重複を伴う可能性がある．この問題は以下の理由によるものである．パターンマッチング式のマッチング時に実行する式は，決定木において葉に対応し，決定木はその作成過程において部分木の複製を生成することがある．したがって決定木は一つのマッチング時の実行式に対応する葉を複数持つことがあり，決定木から生成されたコードも一つの実行式が複数現れる可能性がある．

この問題の一般的な解決法は，マッチング時の実行式を関数にする方法である．この方法は，実行式をパターン中の変数を引数として受け取る関数として，マッチング時に関数適用を行う方法である．この方法は式の重複が生成される問題を解決するものであるが，関数適用時のオーバーヘッドを伴うものである．

この問題を解決するため，本論文では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，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．Hashimoto と Ohori による context calcurus[HO01] は，letterm に用いた文脈 $\delta X.e$ と穴埋め操作 $e1@e2$ を含んでおり，letterm を含むターゲット言語の操作的意味論と型の健全性を定義することができる．

letterm を含むターゲット言語を以下に示す．

$$\begin{aligned} e \quad ::= \quad & c^b \mid x \mid \texttt{\#}i \ \ e \mid \{i : e, \ldots, k : e\} \mid i(e) \\ \mid \quad & \texttt{let } x \texttt{ = } e \ \ \ldots x \texttt{ = } e \texttt{ in } e \texttt{ end} \end{aligned}$$

| letterm x = e ...x = e in e end

| switch e of c_i => e | ... | c_k => e | _ => e

| case e of $i(x)$ => e | ... | $k(x)$ => e | _ => e

| raise e

| e handle e

以上によりパターンマッチングコンパイルアルゴリズムを定義できる．図 4.4 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 4.5 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．

4.5 コンパイル例

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

本論文のコンパイルアルゴリズムは下記の計算を行う．

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

= `let` a `= {1 = 2, 2 = B(5)} in`

`letterm` X_1 `= 3` X_2 `= x` X_0 `= raise MatchFail in`

$\mathcal{TL}[\![\mathcal{E}((\{$`1 = 1, 2 = A(2)`$\}, a) :: X_1, \emptyset, \mathcal{E}((\{$`1 = _, 2 = B(x)`$\}, a) :: X_2, \emptyset, \phi))]\!]$ 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 4.6 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，最終的に以下のターゲットコードが生成される．

```
let a = {1 = 2, 2 = B(5)} in
  letterm X1 = 3 X2 = x X0 = raise MatchFail 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
```

```
                        | _ => X0
                  | B(a4) => let x = a4 in X2 end
                end
        | _ => let a5 = #2 a in
                  case a5 of
                    B(a6) => let x = a6 in X2 end
                  | _ => X0
                end
      end
    end
  end
```

この出力結果は，例えばlet x = a4 のように，冗長な変数束縛を含んでいる．また，決定木の生成過程において，拡張する余地のない部分木に不必要に拡張を試みることがある．6 章ではこのような冗長性を取り除き，より効率的なコードを生成するコンパイラについて延べる．

4.6 パターンの拡張

本論文がこれまでに考慮しなかったパターンに，オアパターンとレイヤードパターンがある．本論文で示したアルゴリズムは，新たな機構を導入することなく，これらのパターンに対応できる．

オアパターンは $(P_1 \mid P_2)$ という形のパターンであり，P_1 もしくは P_2 とのマッチングを図るパターンである．よって木を 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．このパターンに対応するために，$\mathcal{E}$ アルゴリズムに以下のパターンを追加する．

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

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り，P に対応する部分項で x を束縛するパターンである．x による束縛を環境に追加した後に，P による木の拡張を行えばよい．$\mathcal{E}$ アルゴリズムに以下の拡張を加える．

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

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

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

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

$$\mathcal{E}((c,a) :: Ps,\Gamma,\texttt{eq}(a,rules,T_0)) = \texttt{eq}(a, rules \uplus \{c : \mathcal{E}(Ps,\Gamma,T')\}, T_0)$$
$$\text{where}\quad T' = \begin{cases} T & (c : T \in rules) \\ T_0 & (c : T \notin rules) \end{cases}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

図 4.3: 木拡張アルゴリズム

$\mathcal{ML}[\![$`case` e `of` P_1 `=>` e_1 `|` $\cdots$ `|` P_n `=>` $e_n]\!]$

$=$ `let` a `=` e `in`

`letterm`

X_1 `=` e_1 $\ldots$ X_n `=` e_n X_0 `=` `raise MatchFail`

`in`

$\mathcal{TL}[\![\mathcal{E}((\_, a) :: X_0, \emptyset, \mathcal{E}((P_n, a) :: X_n, \emptyset, \ldots \mathcal{E}((P_1, a) :: X_1, \emptyset, \phi)\ldots))]\!]$

`end`

図 4.4: パターンマッチングコンパイルアルゴリズム

$\mathcal{TL}[\![\phi]\!] =$ `raise MatchFail`

$\mathcal{TL}[\![$`eq`$(a, \{\{i : T_i, \ldots, k : T_k\}\}, T_0)]\!]$

$=$ `switch` a `of` i `=>` $\mathcal{TL}[\![T_i]\!]$ `|` $\ldots$ `|` k `=>` $\mathcal{TL}[\![T_k]\!]$ `|` `_` `=>` $\mathcal{TL}[\![T_0]\!]$

$\mathcal{TL}[\![$`tag`$(a, \{\{i : T_i, \ldots, k : T_k\}\}, T_0)]\!]$

$=$ `case` a `of` $i(a_i)$ `=>` $\mathcal{TL}[\![T_i]\!]$ `|` $\ldots$ `|` $k(a_k)$ `=>` $\mathcal{TL}[\![T_k]\!]$ `|` `_` `=>` $\mathcal{TL}[\![T_0]\!]$

where $a_j = getPath(T_j) \quad (i \leq j \leq k)$

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

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

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

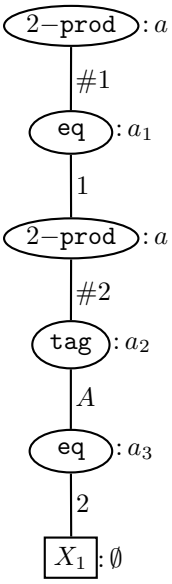
$\mathcal{TL}[\![$`leaf`$(\{x_1 : a_1, \ldots, x_n : a_n\}, e)]\!]$

$=$ `let` x_1 `=` a_1 $\ldots$ x_n `=` a_n `in` e `end`

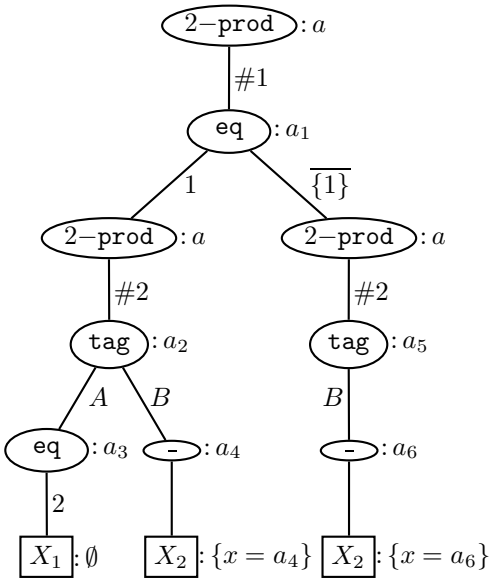
図 4.5: ターゲットコード生成アルゴリズム

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

$T_2 =$



$T_3 =$



$T_4 =$

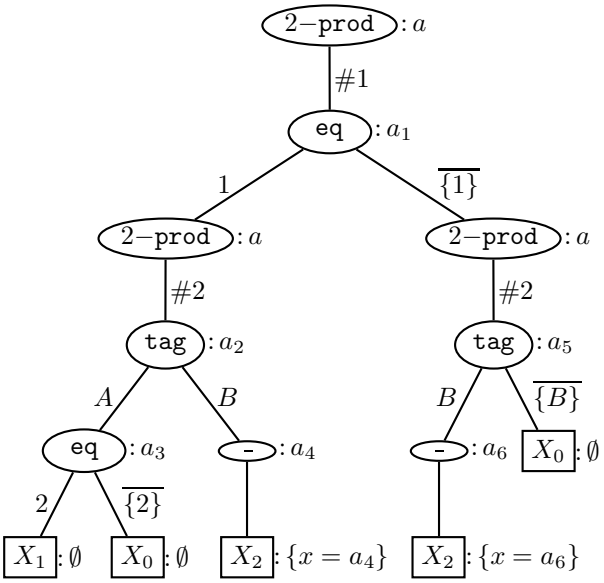


図 4.6: コンパイル例

第5章　アルゴリズムの正しさ

本章では，前章で定義した木拡張アルゴリズム $\mathcal{E}$ の正しさを示す．この正しさは，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ことで証明される．本章ではまず，正しさの証明に必要な，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する．次に $\mathcal{E}$ が型を保存することを示し，これらを用いてアルゴリズムの正しさを示す．

5.1　木の型と型付け規則

木は項集合を表す．木が項集合を正しく表すために，木は整合性を持たなければならない．木の整合性を規定するために，木の型および型付け規則を定義する．

決定木 T とパターンが表す木 Ps の型 σ は以下のように定義される．

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

木 $T : \sigma$ の型付け規則を図 5.1 に示す．

$$\text{(leaf)}\quad \mathtt{leaf}(\Gamma, e) \colon \bullet$$

$$\text{(empty)}\quad \phi \colon \sigma$$

$$\text{(const)}\quad \frac{T_i : \sigma \quad \cdots \quad T_k : \sigma \quad T_0 : \sigma}{\mathtt{eq}(a, \{c_i : T_i, \ldots, c_k : T_k\}, T_0) \colon b :: \sigma}$$

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

$$\text{(prod1)}\quad \frac{T_n : \tau_n :: \sigma}{\mathtt{prod}(a, n, n \colon T_n) \colon \tau_n :: \sigma}$$

$$\text{(prod2)}\quad \frac{T_i : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma}{\mathtt{prod}(a, n, i \colon T_i) \colon \tau_i * \ldots * \tau_n :: \sigma} \quad (i \neq n)$$

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

図 5.1: 決定木の型付け規則

$Ps : \sigma$ の型付け規則を図 5.2 に示す．

これらの型付け規則によって型を導出できた T と Ps のみが，項の部分集合を正しく表す．型付け規則によって型 σ を導出できた T と Ps をそれぞれ $T : \sigma$，$Ps : \sigma$ と表記する．

$$\text{(end)} \quad e : \bullet$$

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

図 5.2: 継続の型付け規則

5.2 項集合の定義

型 σ が表す項集合 $[\![\sigma]\!]$ は以下のように定義できる．

$$\begin{aligned} [\![\bullet]\!] &= \emptyset \\ [\![\tau :: \sigma]\!] &= [\![\tau]\!] \otimes [\![\sigma]\!] \end{aligned}$$

ここで $\otimes$ は以下の定義の集合演算であり，以降で用いる場合も同様の定義とする．

$$\begin{aligned} S_1 \otimes \emptyset &= S_1 \\ S_1 \otimes S_2 &= S_1 \times S_2 \qquad (S_2 \neq \emptyset) \end{aligned}$$

木 $T : \sigma$ が表す項集合 $\overline{T : \sigma}$ の定義を図 5.3 に示す．

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

$$\overline{\mathtt{eq}(a, \{c_i : T_i, \ldots, c_k : T_k\}, T_0) : b :: \bullet} = \begin{cases} \{c_i, \ldots, c_k\} & (T_0 = \phi) \\ [\![b]\!] & (T_0 \neq \phi) \end{cases}$$

$$\begin{aligned} &\overline{\mathtt{eq}(a, \{c_i : T_i, \ldots, c_k : T_k\}, T_0) : b :: \sigma} \\ &\quad = (\{c_i\} \times \overline{T_i : \sigma}) \cup \ldots \cup (\{c_k\} \times \overline{T_k : \sigma}) \cup (\overline{\{c_i, \ldots, c_k\}} \times \overline{T_0 : \sigma}) \qquad (\sigma \neq \bullet) \end{aligned}$$

$$\overline{\mathtt{tag}(a, \{i : T_i, \ldots, k : T_k\}, T_0) : \tau_1 + \ldots + \tau_n :: \bullet} = \{j(t) \mid j \in \{i, \ldots, k\}, t \in \overline{T_j : \tau_j :: \bullet}\} \cup X$$
$$\text{where } X = \begin{cases} \emptyset & (T_o = \phi) \\ \{j(t) \mid j \notin \{i, \ldots, k\}, t \in [\![\tau_j]\!]\} & (T_o \neq \phi) \end{cases}$$

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

$\overline{\mathtt{prod}(a, n, n{:}T_n) : \sigma} = \overline{T_n : \sigma}$

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

$\overline{\mathtt{univ}(a, T) : \tau :: \bullet} = [\![\tau]\!]$

$\overline{\mathtt{univ}(a, T) : \tau :: \sigma} = [\![\tau]\!] \times \overline{T : \sigma} \qquad (\sigma \neq \bullet)$

図 5.3: 木が表す項集合

$Ps : \sigma$ が表す項の部分集合 $[\![Ps : \sigma]\!]$ は以下のように定義される．

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

5.3 決定木 $T : \sigma$ のラベルによる分割

木 $T : \sigma$ の，葉が e である部分木 $T/e : \sigma$ の定義を図 5.4 に示す．

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

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

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

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

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

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

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

図 5.4: T の部分木 $T/e : \sigma$

5.4 アルゴリズム $\mathcal{E}$ の型保存定理

前節で定義した木拡張アルゴリズム $\mathcal{E}$ の正しさを示すには，$\mathcal{E}$ が型を保存することを示さなければならない．これは以下の定理で示せる．

定理 1 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): $\mathcal{E}$ アルゴリズムに関する帰納法で証明できる．証明は Γ を任意の環境と仮定して行う．Ps と T' に関して場合分けして証明する．以下にいくつかの場合についての証明を与える．以下で示さない場合についても，以下で証明するものと同様の手法で証明することができる．

$Ps = e$ **の場合** 型付け規則 (end) より，$Ps : \bullet$ である．よって仮定と型付け規則より，T' はある Γ', e' があって $\mathtt{leaf}(\Gamma', e') : \bullet$ であるか，$\phi : \bullet$ であるかのどちらかである．$T' = \mathtt{leaf}(\Gamma', e) : \bullet$ の場合，$\mathcal{E}$ の定義より，$T = T' : \bullet$ である．$T' = \phi : \bullet$ の場合，$\mathcal{E}$ の定義と型付け規則 (leaf) より，$T = \mathtt{leaf}(\Gamma, e) : \bullet$ である．よって成り立つ．

$Ps = (P, a) :: Ps'$ **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，$Ps' : \sigma'$ とすると規則 (cont) より $\sigma = b :: \sigma'$ である．$\mathcal{E}$ の定義より，$T = \mathtt{eq}(a, \{c : \mathcal{E}(Ps', \Gamma, \phi)\}, \phi)$ である．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 (const) より，$T = \mathtt{eq}(a, \{c : \mathcal{E}(Ps', \Gamma, \phi)\}, \phi) : b :: \sigma'$ である．よって成り立つ．

$P = c : b, T' = \mathtt{eq}(a, \{c_i : T_i, \ldots, c : T_c, \ldots, c_k : T_k\}, T_0)$ **の場合** 仮定と規則 (const) 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，

$$T = \mathtt{eq}(a, \{c_i : T_i, \ldots, c : \mathcal{E}(Ps', \Gamma, T_c), \ldots, c_k : T_k\}, T_0)$$

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．よって規則 (const) より，

$$T = \mathtt{eq}(a, \{c_i : T_i, \ldots, c : T'_c, \ldots, c_k : T_k\}, T_0) : b :: \sigma'$$

である．よって成り立つ．

$P = i(P') : \tau_1 + \ldots + \tau_n, T' = \mathtt{tag}(a, \{h : T_h, \ldots, i : T_i, \ldots, k : T_k\}, T_0)$ **の場合** 規則 (tag) 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，

$$T = \mathtt{tag}(a, \{h : T_h, \ldots, i : \mathcal{E}(P' :: Ps', \Gamma, T_i), \ldots, k : T_k\}, T_0)$$

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．よって規則 (tag) より，$T : \tau_1 + \ldots + \tau_n :: \sigma'$ である．よって成り立つ．

5.5 アルゴリズム $\mathcal{E}$ の正しさの証明

$\mathcal{E}$ の正しさを証明するために，以下の補題を証明する．以下の補題では，$T = \mathcal{E}(Ps : \sigma, \Gamma, T' : \sigma)$ なる T について，T の型が σ であるとしている．これは前節の型保存定理によって証明されている．

補題 1 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): 型保存定理と同様に，$\mathcal{E}$ アルゴリズムに関する帰納法で証明できる．証明は Γ を任意の環境と仮定して行う．Ps と T' に関して場合分けして証明する．以下にいくつかの場合についての証明を与える．以下で示さない場合についても，以下で証明するものと同様の手法で証明することができる．

$Ps = e$ **の場合** 規則 (end) より $Ps : \bullet$ である．よって仮定と型付け規則より，T' はある Γ', e' があって $\mathtt{leaf}(\Gamma', e') : \bullet$ であるか，$\phi : \bullet$ であるかのどちらかである．$T' = \mathtt{leaf}(\Gamma', e) : \bullet$ の場合，$\mathcal{E}$ の定義より，$T = T' : \bullet$ であり，定義より $\overline{T : \bullet} = [\![Ps : \bullet]\!] \cup \overline{T' : \bullet} = [\![Ps : \bullet]\!] \setminus \overline{T' : \bullet} = \emptyset$ である．$T' = \phi : \bullet$ の場合も，$\mathcal{E}$ の定義と型付け規則 (leaf) より，$T = \mathtt{leaf}(\Gamma, e) : \bullet$ であり，定義より $\overline{T : \bullet} = [\![Ps : \bullet]\!] \cup \overline{T' : \bullet} = [\![Ps : \bullet]\!] \setminus \overline{T' : \bullet} = \emptyset$ である．よって成り立つ．

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

$P = c : b, T' = \phi, Ps' \neq e$ **の場合** $Ps : \sigma, P : b$ であるから，$Ps' : \sigma'$ とすると規則 (cont) より $\sigma = b :: \sigma'$ である．また，$Ps' \neq e$ より，$\sigma' \neq \bullet$ である．$\mathcal{E}$ の定義より，$T = \mathtt{eq}(a, \{c : \mathcal{E}(Ps', \Gamma, \phi)\}, \phi)$ であり，ここで $T'_c = \mathcal{E}(Ps', \Gamma, \phi)$ 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，$\overline{T'_c : \sigma'} = [\![Ps' : \sigma']\!] \cup \overline{\phi : \sigma'}$ かつ $\overline{T'_c/e : \sigma'} = [\![Ps' : \sigma']\!] \setminus \overline{\phi : \sigma'}$ である．

一方，$\sigma' \neq \bullet$ であるから，木が表す項集合の定義より，

$$\overline{T : b :: \sigma'} = \overline{\mathtt{eq}(a, \{c : T'_c\}, \phi) : b :: \sigma'} = \{c\} \times \overline{T'_c : \sigma'}$$

である．よって，$\overline{T : b :: \sigma'} = \{c\} \times ([\![Ps' : \sigma']\!] \cup \overline{\phi : \sigma'}) = \{c\} \times ([\![Ps' : \sigma']\!])$ である．ここで定義より，$[\![Ps : b :: \sigma']\!] \cup \overline{T' : b :: \sigma'} = [\![(c, a) :: Ps' : b :: \sigma']\!] \cup \emptyset = \{c\} \times [\![Ps' : \sigma']\!]$ である．よって $\overline{T : \sigma} = [\![Ps : \sigma]\!] \cup \overline{T' : \sigma}$ である．

また，$\sigma' \neq \bullet$ であるから，定義より，

$$\overline{T/e : b :: \sigma'} = \overline{\mathtt{eq}(a, \{c : T'_c\}, \phi)/e : b :: \sigma'} = \overline{\mathtt{eq}(a, \{c : T'_c/e : \sigma'\}, \phi) : b :: \sigma'} = \{c\} \times \overline{T'_c/ :: e}\sigma'$$

である．よって，$\overline{T : b :: \sigma'} = \{c\} \times ([\![Ps' : \sigma]\!] \setminus \overline{\phi : \sigma'}) = \{c\} \times [\![Ps' : \sigma']\!]$ である．ここで定義より，$[\![Ps : b :: \sigma']\!] \setminus \overline{T' : b :: \sigma'} = [\![(c, a) :: Ps' : b :: \sigma']\!] \setminus \emptyset = \{c\} \times [\![Ps' : \sigma']\!]$ である．よって $\overline{T/\sigma :=} [\![Ps : \sigma]\!] \setminus \overline{T' : \sigma}$ である．

以上により成り立つ．

$P = \{i : P_i, \ldots, n : P_n\} : \tau_i * \ldots * \tau_n, T' = \mathtt{prod}(a, n, i{:}\, T_i), Ps' \neq e, i \neq n$ **の場合** $Ps = (P, a) :: Ps' : \sigma, P : \tau_1 * \ldots * \tau_n, Ps' \neq e$ であるから，$Ps' : \sigma'$ とすると規則 (cont) より，$\sigma = \tau_i * \ldots * \tau_n :: \sigma', \sigma' \neq \bullet$ である．また，$i \neq n$ と規則 (prod2) より，$T_i : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma'$ である．$i \neq n$ であるから，$\mathcal{E}$ の定義より，

$$T = \mathtt{prod}(a, n, i{:}\, \mathcal{E}((P_i, a') :: \tau_{i+1}\tau_n :: Ps' * \ldots *, \Gamma, T_i))$$

である $(a' = getPath(T_i))$．ここで $T'_i = \mathcal{E}((P_i, a') :: \tau_{i+1}\tau_n :: Ps' * \ldots *, \Gamma, T_i)$ とする．Ps が e で終わるから，Ps' も e で終わり，$\overline{T'/e : \sigma} = \emptyset$ 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．また型保存定理より $T'_i : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma'$ である．したがって帰納法の仮定と $[\![Ps]\!]$ の定義より，

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

かつ

$$\overline{T'_i/e : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma'} = [\![(P_i, a') :: \tau_{i+1}\tau_n :: Ps' * \ldots *]\!] \setminus \overline{T_i : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma'}$$

である．ここで $i \neq n, \sigma' \neq \bullet$ であるから定義より，

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

である．ここで

$$\overline{T'_i : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma'} = [\![P_i]\!] \times (([\![P_{i+1}, \ldots [\![P_n]\!]]\!]) \times [\![Ps']\!]) \cup \overline{T_i : \tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma'}$$

であるから，

$$\overline{T:\sigma} = ([\![P_i]\!]\times\ldots[\![P_n]\!])\times[\![Ps']\!]\cup\{((t_i,\ldots,t_n),t) \mid (t_i,((t_{i+1},\ldots,t_n),t)) \in \overline{T_i:\tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma'}\}$$

である．ここで定義より，
$[\![(\{i:P_i,\ldots,n:P_n\},a) :: Ps' : \tau_i * \ldots * \tau_n :: \sigma']\!] \cup \overline{T' : \tau_i * \ldots * \tau_n :: \sigma'}$
$= ([\![P_i]\!]\times\ldots[\![P_n]\!])\times[\![Ps']\!]\cup\{((t_i,\ldots,t_n),t) \mid (t_i,((t_{i+1},\ldots,t_n),t)) \in \overline{T_i:\tau_i :: \tau_{i+1} * \ldots * \tau_n :: \sigma'}\}$
となる．よって，$\overline{T:\sigma} = [\![(\{i:P_i,\ldots,n:P_n\},a) :: Ps' : \tau_i * \ldots * \tau_n :: \sigma']\!] \cup \overline{T' : \tau_i * \ldots * \tau_n :: \sigma'}$
である．
$\overline{T/e:\sigma}$ も同様である．よって成り立つ．

以上の補題を用いて $\mathcal{E}$ の正しさを証明できる．
まず以下のパターンマッチング式を考える．

$$\texttt{case}\ e:\tau\ \texttt{of}\ P_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ P_n\ \texttt{=>}\ e_n$$

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．

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

以上の定義から，アルゴリズムの正しさを示すには，以下の定理を証明すれば十分である．

定理 2　1. $\overline{T_i : \tau :: \bullet} = [\![(P_i,a) :: e_i : \tau :: \bullet]\!] \cup \overline{T_{i-1} : \tau :: \bullet} \qquad (1 \le i \le n)$

2. $\overline{T_i/e_i : \tau :: \bullet} = [\![(P_i,a) :: e_i : \tau :: \bullet]\!] \setminus \overline{T_{i-1} : \tau :: \bullet} \qquad (1 \le i \le n)$

証明:　この定理は補題 1 において，$Ps = (P_i,a) :: e_i$ の場合である．補題 1 は証明されたので，この定理も証明できる．

上記の定理によって，アルゴリズムが作成した木が表示的意味論に関して正しいことを示せた．よって，アルゴリズムが，パターンの冗長性，パターン集合の網羅性を検出できることは明らかである．

第6章　実装

本論文では，実装に加え，効率よい実装を行う上での技術的問題と解決策を検討した．本章では，それら問題点と解決策を論じ，それらを導入した実装の概要を報告する．

6.1　最適化

本節では実装時に施した最適化について延べる．最適化は，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目的とするものである．

6.1.1　不必要な木の拡張の抑制

木 $T:\sigma$ が表す項集合 $\overline{\sigma:T}$ は $[\![\sigma]\!]$ の部分集合である．したがって $\overline{\sigma:T}$ が $[\![\sigma]\!]$ と一致するとき，木 T はこれ以上拡張の余地のないものである．木 $T:\sigma$ について，$\overline{\sigma:T}=[\![\sigma]\!]$ の時，木 T が「閉じている」と表現する．

4.3 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，閉じた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抑制できる．

6.1.2　不必要な変数束縛の抑制

4.4 節で示したコード生成アルゴリズム $\mathcal{TL}$ は，`let x = a` という形の冗長な変数束縛を生成する．このような冗長な束縛の生成は以下の二つの理由によって必要である．一つは，決定木の中間ノードが持つアクセスパスが項の同一の部分項を指す場合も，中間ノードが異なればアクセスパスの名前が異なる場合があることである．もう一つは，異なるパターンでは項の同一の部分項を束縛する場合も，一般的に異なる名前の変数を用いることである．

したがってこの冗長性を取り除くために以下の処理を行う．まず，アルゴリズム $\mathcal{E}$ において新たにアクセスパスを生成する場合，項の同一の部分項を指すアクセスパスは，同じ名前を持つようにする．次に，パターンマッチング式内の分岐時の実行式 e_i に $\alpha-$ 改名を施す．e_i の $\alpha-$ 改名では，e_i 内の自由変数でパターン中の変数によって束縛される変数を，変数に対応するアクセスパスの名前で置き換える．

例えば，4.5 節で示した入力コードに対し，実装したコンパイラは以下のようなコードを生成する．

```
let a = {1 = 2, 2 = B(5)} in
  letterm X1 = 3 X2 = a4 X0 = raise MatchFail 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
                          | _ => X0
               | B(a4) => X2
             end
      | _ => let a2 = #2 a in
               case a5 of
                 B(a4) => X2
               | _ => X0
             end
    end
  end
end
```

6.2 冗長性と網羅性の検出

本節では冗長なパターンの検出およびパターン集合の網羅性の判定の実現方法について述べる．

冗長なパターンの検出のために，コンパイラは新しく葉が作られたかどうかを表すフラグを導入する．アルゴリズム $\mathcal{ML}$ がパターンマッチング中の各パターンを処理する際に，このフラグをリセットしてからパターンによる木の拡張を行う．木拡張アルゴリズム $\mathcal{E}$ は，新しく葉が作られた場合，フラグをセットする．もしフラグがセットされていなければ，そのパターンは冗長なパターンであることが分かる．

また，パターン集合の網羅性は，$\mathcal{ML}$ が行うワイルドパターンによる最後の木の拡張の際に，フラグがセットされているかどうかで判定できる．ワイルドパターンによる拡張によってフラグがセットされた場合は，パターン集合が網羅的でないことを意味し，フラグがリセットされたままの場合は，パターン集合が網羅的であることを意味する．

6.3 実装システム

本節では，実装したシステムを説明する．システムの概要を以下に示す．

実装対象 Standard ML で定義されているパターン言語のフルセットである．

実装言語 Standard ML によって実装を行った．

実装機能 実装したシステムは，以下の機能を実現するものである．

パターンマッチングコンパイル パターンマッチング式のコンパイルを行う．

パターンの拡張 パターンの拡張が可能であり，現在 Standard ML で定義されていないオアパターンにも対応している．

冗長なパターンの検出とパターン集合の網羅性の判定 入力パターンマッチング式中から，冗長なパターンを検出し，また，入力パターン集合の網羅性の判定を行う．

エラー出力 冗長なパターンを検出した場合や，入力パターン集合が網羅的でなかった場合，エラー出力を行う．

第7章　関連研究との比較

既存のパターンマッチングコンパイラには，決定木を使用するものとバックトラックオートマトンを作成するものの二つがある．本章では，これらを用いた既存研究との比較を行う．

7.1　他の決定木モデルとの比較

決定木を用いたパターンマッチングコンパイル手法の概略は，[Car84, SR00] で述べられている．また，[Ait92, BM85, SRR92] によって様々なコンパイル手法が考案されている．しかし，これらはパターンマッチングコンパイルの本質的なもののみを取り扱っており，実際にコンパイラを作成するには不十分なものである．これらの論文で用いられている決定木の多くはコンストラクタパターンにのみ対応するものであり，様々なデータ構造を取り扱う実際のコンパイラを作成するには，実装者がアドホックに決定木を拡張しなければならない．また，これらの手法は，コンパイルアルゴリズムの正しさや，パターンマッチングの冗長性や網羅性の検出方法など，パターンマッチングコンパイルが持つ様々な特性を明らかにしていない．本研究は，以上のようにこれまでの手法が抱えていた問題への一つの解決法を示すものである．

また，これまでに提案された手法では，プロダクトの各フィールドのテスト順番をヒューリスティックに選ぶことによって生成コードのテスト回数を低く抑える手法がいくつか考案されている．本論文では，これらヒューリスティックな手法は本論文の主題から外れるため導入していない．しかし，型に基づいた構築を持つ本論文による決定木に，これらヒューリスティックな手法を適用できると考えられ，さらに型情報を利用した新たな手法も適用できることが期待できる．

7.2　バックトラックオートマトンモデルとの比較

パターンマッチング式をバックトラックオートマトンへとコンパイルする手法は [Wad87, Aug85, Ler92, FM01] で研究されている．特に，[FM01] はバックトラックオートマトンを用いる手法での最適化およびその効果を述べている．同一のテストを複数回行う可能性のあるこれらの手法と比べて，本研究による手法は一般的に効率のよいコードを生成するが，他の決定木を用いる手法と同様，生成コードのコードサイズとコンパイル時間が指数関数的に増大する場合がある．

第8章　結論と今後の課題

8.1　結論

本論文は，パターンマッチングコンパイルの型に基づいた体系的な説明を示した．さらに，パターンマッチング式を効率的なコードへとコンパイルするアルゴリズムの構築，その正しさの証明，および実装を行った．本論文では，パターンマッチング式中のパターン集合を，項の全体集合を各パターンが表す部分集合へと分割したものとみなし，パターンマッチングを，マッチングを行う項が含まれる部分集合を決定する機構であると考えた．上記の考え方および項の部分集合を木の形で表現することにより，パターンマッチングコンパイルを木のラベル付けアルゴリズムとみなすことができる．本論文が構築したアルゴリズムによって作成された決定木は，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した．また，アルゴリズムが最終的に生成するコードは同一のテストを多くとも一度しか行わないため，効率よいものである．本論文が示したコンパイルアルゴリズムは，パターンの冗長性とパターン集合の網羅性の検出やオアパターンやレイヤードパターンへの対応など，実際のパターンマッチングコンパイラが満たすべき性質を，新たな機構を導入することなく実行できるものとなっている．また，本論文が示したアルゴリズムは，Standard ML のパターン言語のフルセットを対象としてすでに実装され，現在 JAIST において開発中の次世代 ML コンパイラの一部となる予定である．

8.2　今後の課題

本研究が構築したパターンマッチングコンパイルアルゴリズムは，他の研究で使用されている最適化手法を使用していない．したがって以下で示す既存の最適化手法を取り入れることでより効率のよいコンパイルおよびコード生成をできると期待される．

ヒューリスティックな最適化手法の導入 既存の決定木を用いたコンパイル手法が導入している，ヒューリスティックな最適化手法は，本研究で構築したコンパイル手法にも導入することができると考えられる．本論文で用いる決定木は，その構造が型によって明らかにされているため，型情報を利用した新たな手法も適用できることが期待される．

バックトラックオートマトンによるコンパイル手法の導入 決定木を用いるコンパイル手法とは異なり，バックトラックオートマトンによるコンパイル手法は，コンパイル時間と生成コードのサイズが指数関数的に増加することはない．この手法を本研究のコンパイル手法に取り入れることを検討することによって，コンパイル時間と生成コードサイズを縮小させる可能性がある．

謝 辞

本研究で報告した成果には，大堀淳教授との共同研究 [OO04] に基づくものが含まれている．これを含め，本研究全般に渡り御指導下さった大堀淳教授に感謝致します．

また，本研究及び本論文の作成にあたって，有益なご助言を頂きました大堀研究室の皆様に感謝致します．

参考文献

[Ait92] W. Aitken. SML/NJ match compiler notes. Distributed with Standard ML of New Jersey, 1992.

[AM87] A. W. Appel and D. B. MacQueen. A Standard ML Compiler. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, volume 274, pages 301–324, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.

[Aug85] L. Augustsson. Compiling pattern matching. In *Functional Programming and Computer Architechture*. ACM, 1985.

[BM85] M. Baudinet and D. MacQueen. Tree pattern matching for ML. Draft, 1985.

[Car84] L. Cardelli. Compiling a functional language. In *LISP and Functional Programming*, pages 208–217, 1984.

[eaeH92] P. et. al. editors Hudak. *Report on programming language Haskel a non-strict, purely functional language version 1.2.* SIGPLAN Notices, 1992.

[FM01] F. Le Fessant and L. Maranget. Optimizing pattern matching. In *International Conference on Functional Programming*, pages 26 – 37, 2001.

[HO01] M. Hashimoto and A. Ohori. A typed context calculus. *Theoretical Computer Science*, 266(1–2):249–272, 2001.

[Ler92] X. Leroy. The zinc experiment: an economical implementation of the ml language. Technical report, INRIA, 1992.

[Ler97] X. Leroy. *The Objective Caml User's Manual*. INRIA Rocquencourt, 1997.

[OO04] A. Ohori and S. Osaka. Pattern matching compilation by type tree expantion. to be submitted for publication, 2004.

[RMH90] M. Tofte R. Milner and R. Harper. The definition of standard ml. The MIT Press, 1990.

[SR00] Kevin Scott and Norman Ramsey. When do match-compilation heuristics matter? Technical Report CS-2000-13, Dept. of Computer Science, University of Virginia, 2000.

[SRR92] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. In *Automata, Languages and Programming*, pages 247–260, 1992.

[Wad87] P. Wadler. Compilation of pattern matching. *The Implementation of Functional Programming*, 1987.