修 士 論 文

関数型言語におけるインライン展開に関する研究

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

佐藤　浩二

2005 年 3 月

修 士 論 文

関数型言語におけるインライン展開に関する研究

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　小川瑞史 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

310046 佐藤　浩二

提出年月: 2005 年 2 月

概 要

インライン展開は、多くのコンパイラにおいて実行される最適化の１つである。ML などの関数型言語において、インライン展開を行うためには、呼び出される関数の中に含まれる自由変数が、インライン展開後のコードでも正しく参照されている必要がある。しかしながら、一般に特別な命令を導入することなく、すべての自由変数が、インライン展開をする位置から参照できるとは限らない。そこで、本論文では、クロージャ変換の特性を利用することで、関数の中に含まれる自由変数を正しく参照する手法を述べる。これにより、自由変数の存在に影響されないインライン展開を行うアルゴリズムの構築が可能となり、より多くの関数適用式をインライン展開の候補とすることが可能となる。本論文では、そのアルゴリズムの構築をし、それによって変換されたコードが型を保存することについても示す。さらに、Standard ML の拡張コンパイラである IML コンパイラの上に、実装をしてその実用性を示す。

目 次

第 1 章　序論 **3**
1.1 最適化 3
1.2 研究の背景 3
1.3 目的 4
1.4 構成 5

第 2 章　インライン展開 **6**
2.1 インライン展開の定義 6
2.1.1 関数インライン展開 6
2.1.2 β 簡約 7
2.1.3 束縛変数の展開 7
2.2 インライン展開の利点 8
2.3 インライン展開の問題点 9
2.4 関連する最適化 9
2.4.1 定数の畳み込み 9
2.4.2 共通部分式の削除 10
2.4.3 無用命令の削除 10
2.4.4 フィールド要素取得命令の変換 11
2.4.5 条件文の変換 11
2.5 コンパイルの終了の保障 12
2.6 自由変数の参照 12
2.7 高階関数 13

第 3 章　クロージャ変換後でのインライン展開 **15**
3.1 クロージャ変換 15
3.2 クロージャ変換後でのインライン展開 16
3.3 インライン展開のアルゴリズム 18
3.4 型保存定理の証明 27
3.4.1 型システム 27
3.4.2 型保存定理の証明 29

第 4 章　実装と評価 **41**
4.1　コードサイズの抑制 41
4.2　その他の最適化 42
4.3　実装環境 44
4.4　評価 45
4.5　クロージャより環境を取得するインライン展開の効率 48

第 5 章　結論と今後の課題 **50**
5.1　結論 50
5.2　今後の課題 50

第1章　序論

1.1　最適化

現在、コンパイラの多くは、入力コードを実行可能な目的コードに変換する機能とともに、入力コードを変換して、より実行効率のよいコードを生成することを目的とした、最適化の機能を備えている。最適化の目的である実行効率の向上には、大きく以下の 2 つがある。

- 実行速度を速くする
- 実行プログラムの使用メモリを少なくする

最適化の中には、実行速度を速くするために、使用メモリが増してしまう、また、その逆のパターンを持つものも存在する。

一般に、コンパイラは、まず入力言語に対して構文解析、意味解析を行い構文木を生成する。そして、その構文木に対して、なんらかの変換を施して中間言語の形に変換する。さらに、その中間言語に対して、再び別の変換を施し、また別の中間言語に変換されるといったことが繰り返され、最終的に目的のコードが出力される。

最適化する対象の言語としては、コンパイル中の中間言語であるか、もしくは、最終的な目的コードに対して行われる。関数型言語であれば、最適化の対象となる中間言語の表現には、ラムダ式、CPS[1]、A-normal form[5] などがある。多くの最適化、特に大域的な解析が必要な最適化は、これらの中間言語を対象に行われることが多い。目的コードに対しての最適化は、一般には大域的な解析は難しいため、ある命令を同一の意味をもったより速い命令に置き換えるなどの、局所的な最適化となることが多い。

また、目的コードが抽象機械コードであれば、そのコードを実行するには抽象機械が必要となる。例えば、JAVA などがこれにあたる。

このようなものでは、実行コードを実行前にあらかじめ解析し、変換する静的な最適化のみならず、抽象機械にて、コードの実行時に動的に最適化を行うこともある。

本論文では、動的な最適化については扱わず、静的な最適化について述べる。

1.2　研究の背景

最適化の１つにインライン展開がある。これは、関数適用式において呼び出される関数の本体を、呼び出した場所に、プログラムの意味が変化しないように置き換える変換であ

る。インライン展開の主な利点として、以下の２つがあげられる。

- 関数呼び出しに伴うオーバヘッドの軽減
- 展開後のコードに対して、別の最適化が実行可能

前者は、関数呼び出しをする際の引数の受け渡し、レジスタ管理、スタック情報の更新を削除することによる実行効率の向上を指す。

後者は、展開後のコードに対して別の最適化を実行することで、一層の速度向上が可能となることを指す。特に、展開後のコードは、関数の仮引数に実引数が与えられるので、関数の中を直接インライン展開した場合には、不可能であったインライン展開が実行できる可能性がある。

ML をはじめとする多くの関数型言語では、関数内に自由変数が存在するため、インライン展開する場合は、自由変数が正しく参照されるように展開する必要がある。単純なインライン展開によって、自由変数が正しく参照されなくなる場合には、一般には、束縛変数名を付け替えることによって解決を図っている。しかしながら、それだけではすべての自由変数の参照を正しく解決できるとは限らない。

例えば、高階関数を持つ関数型言語では、関数の結果として関数を返すことが可能であり、そのような関数をインライン展開することを考える。`let val a = f M in a N end` は、関数 f に実引数 M を与えることによって、その結果としてある関数を返し、その関数を変数 a に束縛して、実引数 N に関数 a を適用するコードであるとする。ここで、変数 a が束縛している関数が判明したとしても、その関数の中に自由変数が存在して、かつ、その自由変数が関数 f の中で定義されていたとすると、`a N` の関数適用式の位置からはその自由変数を参照することは不可能である。

そのため、このような関数に対しては、展開すべきコードが判明したとしても、自由変数を正しく参照できないために、インライン展開を行うことができない。

このように、関数型言語では自由変数の関係で、インライン展開が不可能となることがあり、インライン展開の候補を減らしてしまうことがありえる。

1.3 目的

本研究の主な目的は、自由変数に影響されることなくインライン展開を行うアルゴリズムの構築である。すなわち、自由変数が正しく参照できないために、インライン展開が不可能であった関数に対して、インライン展開を可能とするアルゴリズムを構築することである。

具体的には、以下のことについて述べる。

- クロージャ変換後のインライン展開
  クロージャ変換後では、すべての関数は関数コードと環境の組から構成されるクロー

ジャへと変換される。この関係を利用して、自由変数の参照を正しく解決し、自由変数に影響されることなくインライン展開が可能なアルゴリズムを構築する。

- 提案するアルゴリズムの型保存定理の証明
  上記で提案するインライン展開のアルゴリズムによって変換されたプログラムが、変換前と変換後で同一の型を持つことを証明する。

- 提案するアルゴリズムの実装
  上記で提案するアルゴリズムを、Standard ML の拡張コンパイラである IML コンパイラに実装し、その実用性を検証する。

1.4 構成

本論文の構成を述べる。2 章では一般的なインライン展開について説明する。3 章では本論文で提案するインライン展開の手法とアルゴリズムについて述べる。また、そのアルゴリズムが型を保存することについても示す。4 章では 3 章で提案したアルゴリズムの実装、およびその結果について説明する。5 章では結論と今後の課題について述べる。

第2章　インライン展開

2.1　インライン展開の定義

インライン展開の定義として３つ述べる。本論文が対象としているインライン展開は、関数インライン展開と β 簡約である。

2.1.1　関数インライン展開

関数インライン展開とは、関数適用の式において呼び出される関数の本体を呼び出した場所にプログラムの意味が変化しないように展開することである。

Standard ML での基本的なインライン展開の方法は、関数適用の式を、let 式によって実引数を呼び出される関数の仮引数名で束縛し、その後、呼び出される関数の本体を展開したコードに置き換える。Standard ML では、関数呼び出しの時はまず実引数の式を評価して、その結果の値を関数に渡す。そのため、意味が変換しないようにインライン展開を行うには、let 式によって実引数の式をあらかじめ評価してその値を束縛する。同じく、Standard ML での複数引数の関数適用式は、左から順に引数が評価されるので、let 式での変数の束縛の順序も、左側の引数から順に束縛する必要がある。それゆえに、遅延評価などを行う言語や、右から実引数を評価していく言語ではその定義にあうように変換する必要がある。

本論文では、すべて対象とする言語は Standard ML とする。

```
let
  fun f x y = x + y
in
  f 1 2
end
```

$\Rightarrow$

```
let
  fun f x y = x + y
in
  let
    val x = 1
    val y = 2
  in
    x + y
  end
end
```

上記はインライン展開前後のそれぞれのコードであり、インライン展開前のコードに現

れる関数適用式 (f 1 2) の部分をインライン展開している。具体的には let 式によって、実引数 1, 2 を仮引数 x, y に束縛して、その後、本体部分を展開している。

上記のインライン展開は、プログラムの意味を変えておらず正しい変換であるが、展開する関数本体の中に自由変数が含まれる場合は、単純にインライン展開をするとプログラムの意味を変えてしまう恐れがある。そのため、そのような場合ではプログラムの意味が変わらないように注意してインライン展開を行う必要がある。仮に、正しくインライン展開ができない場合、すなわち、プログラムの意味が変わる場合には当然インライン展開をしてはならない。

2.1.2 β 簡約

関数型言語のモデルとなるラムダ式には、β 簡約と呼ばれる基本的な変換規則があり、これは仮引数を実引数で置き換える変換である。

以下の式を見てみる。

```
(fn x => fn y => x + y) 1 2
```

上記の式において、仮引数を実引数で置き換えると、$1 + 2$ となる。この例のように、実引数が副作用を含まないのであれば問題ないが、対象とする言語である Standard ML では、副作用を持つ式が実引数として評価される可能性があるため、単純に置き換えるとプログラムの意味を変えてしまう恐れがある。そのため以下のように、β 簡約においても let 式で実引数の値を束縛した後に本体部分を置き換える。

```
let
  val x = 1
  val y = 2
in
  x + y
end
```

2.1.3 束縛変数の展開

関数インライン展開以外のもう 1 つのインライン展開に、束縛変数をそれが束縛している式で置き換えるものがある。以下の例を見る。M は任意の式を表す。

```
let                                  let
  val x = M                            val x = M
in                          ⇒        in
  x * x * 3.14                         M * M * 3.14
end                                  end
```

上記は、束縛変数 x を x に束縛されていた式 M で置き換えている。このような展開は大きく 2 つの問題がある。

1 つはプログラムの意味の変化である。束縛している式 M が副作用のない式であれば、展開してもプログラムの意味に変化はないが、もし、副作用があれば、展開することによって副作用のある式を複数回実行するために、プログラムそのものの意味が変化してしまうことである。本論文が対象としている言語である Standard ML は副作用を持つ言語であるため、この変換を行う場合はこの点に注意する必要がある。

もう 1 つは同一処理の複数回実行である。以下の例を考えてみる。

```
let                                    let
  val x = 3 * 5                          val x = 3 * 5
in                          ⇒          in
  x * x                                  (3 * 5) * (3 * 5)
end                                    end
```

この変換は、(3 * 5) の実行は本来 1 回のみなのであるが、インライン展開されることで 2 回実行されてしまう。このような変換は実行効率を下げる可能性があり、展開することが必ずしも好ましいとはいえない。また、たとえインライン展開すべき束縛変数が 1 度しか現れなくとも、展開場所が関数の中やループの中であれば、同一の処理が複数回行われる可能性がある。そのため、束縛変数を展開する場合は以上のことを考慮して展開する必要がある。

2.2 インライン展開の利点

インライン展開を行うことの利点として以下の 2 つがある。

- **関数呼び出しに伴うオーバヘッドの削減**

インライン展開をすることで得られる直接の利点は、関数呼び出しに伴うオーバヘッドの削減である。関数が呼び出しに伴う処理として、引数の受け渡し、レジスタの管理、戻りアドレスの格納、関数コードへの先頭アドレスへのジャンプなどがある。また、高階関数を持つ言語では、複数引数の適用において高階関数が持つオーバヘッドを削除する最適化としての性質がある。さらに、多相型を持つ言語では、関数が持つ多相型を展開することにより多相型に関するコストを削除する性質も併せ持っている。

- **展開されたコード部分に対する各種最適化**

インライン展開をすることによって展開されたコードに対して、別の各種最適化を施すことにより、一層実行効率を向上させることが可能である。また、展開された部分は、関数の本体にある仮引数の値が明らかになるため、関数の本体を直接最適化することでは不可能である最適化を新たに行える可能性がある。これは、関数を適用するそれぞれの状況

に応じて、それぞれ個別にその関数を最適化することに対応する。

2.3 インライン展開の問題点

インライン展開を行うことの欠点として以下の３つを挙げる。

- **コードサイズの増加**

インライン展開することにより、一般にプログラムのコードサイズが増加することとなる。この問題は最も重大な問題である。多くの source-to-source の最適化は、その目的として２つに大別することができる。１つは実行速度の向上を目的とした最適化である。もう１つは使用メモリの減少である。通常、コードサイズの増加は使用メモリの増加につながる。インライン展開は、まさに使用メモリの増加と引き換えに実行速度を向上させる最適化である。

しかしながら、極端なコードサイズの増加は実用的でないため抑止する必要がある。よって、インライン展開をする場合は、コードサイズの増加と速度向上の２つのバランスを考えた上で、実際にインライン展開をするか否かを決定する必要がある。

- **キャッシュミスの増加**

インライン展開をすることにより、１つの関数の中でより多くのローカル変数が使われる。これは、一般にレジスタアロケーションなどを難しくし、キャッシュミスの増加を招く可能性がある。その場合、変数の参照により時間がかかるため実行効率が落ちる可能性がある。

- **コンパイル時間の増加**

インライン展開によって生成された新しいコード部分は、それ以前には解析されていない部分である。そのため、新しいコード部分に対して別の最適化を施そうとした場合は、それまでの解析結果が使用できないため新たに解析をやり直す必要が生じる可能性がある。

また、コードが大きくなることにより、より複雑な解析が必要となり、他の最適化についてもコンパイル時間が増加する可能性が生じる。

2.4 関連する最適化

2.4.1 定数の畳み込み

定数の畳み込みとは、定数計算をコンパイル時に実行して、その計算結果で式を置き換えるものである。

例えば、a = 1 + 2 という式の場合は、1 + 2 を 3 で置き換えて、a = 3 とすることができる。

ただし、定数計算の結果が、オーバフローを起こす場合は注意する必要がある。

2.4.2 共通部分式の削除

これは、共通の式が複数回表れる場合に、一番初めに計算した結果でその後の共通部分式を置き換える変換である。一番初めに計算した結果を後で使用するためには、あらかじめ共通式の結果を、変数に束縛しておく必要がある。

```
3 * 3 + 3 * 3        ⇒        let
                                 val x = 3 * 3
                               in
                                 x * x
                               end
```

上記の例だと、3 * 3 の共通部分式の結果を x に束縛して、3 * 3 の部分は変数 x の参照に置き換えている。この最適化は、共通部分式の数やその処理の重さに比例して最適化の効果も高くなる。

ただし、共通部分式に副作用がある場合は、実行される回数や位置が変わることにより、プログラムの意味を変化させる恐れがあるため注意する必要がある。

2.4.3 無用命令の削除

実行されることのない命令、すなわち、制御の流れがそこに届くことのない命令は削除することができる。

```
if true then 3 * 3 else 4 * 4    ⇒    3 * 3
```

上記の式では、else の時に実行される 4 * 4 は実行されることがないので削除することができる。すなわち、上記の式は必ず 3 * 3 が実行されるため、直接 3 * 3 で置き換えることが可能である。

a = a といった式も無用命令であり削除可能である。

また、a = 3 * 3 という式があっても、変数 a がプログラム中で 1 回も使用されていなければ削除可能である。さらに a = b という式の場合は、変数 a の参照を変数 b の参照で置き換えることにより、変数 a がプログラム中で 1 回も現れなくなれば、結果として削除が可能である。

ただし、a = M という式があり式 M が副作用を持っている場合は、a を消してしまうと式 M が実行されずプログラムの意味を変えてしまう恐れがある。そのため、副作用を持つ式については注意が必要である。

2.4.4 フィールド要素取得命令の変換

レコードのフィールドが取り出される場合に、その取り出されるレコードの要素が判明すれば直接その要素で置き換えることができる。

```
let                                    let
  val x = (M, N)                         val x = (M, N)
in                          ⇒          in
  #1 x                                   M
end                                    end
```

上記の式では x に (M, N) が束縛されているので、その 1 番目の要素を取り出す`#1 x` は M の結果を取り出すこととなる。ここで、M が定数や変数であれば直接 M で`#1 x` を置き換えても問題ない。しかしながら、M が実行を伴う式であった場合は同一の処理を 2 回行うこととなるので、置き換えることにより逆に効率の低下を招く恐れがある。ただし、置き換えることにより変数 x の定義が削除可能になる場合には問題ない。

また、副作用のある式の場合は、置き換えることでプログラムの意味を変化させる可能性があるため注意が必要である。

2.4.5 条件文の変換

```
let
  val x = if y then e2 else raise e3
in
  e4
end
```

上記の式のように、条件文の一方が例外を発生するコードの場合は、以下のように変換ができる。

```
if y then
  let
    val x = e2
  in
    e4
  end
else raise e3
```

この変換は直接実行効率に影響はない。しかしながら、このように変換することで e4 の中で e2 の結果を束縛した x を使用することが可能となり、より効率のよい最適化が可能となる。

2.5 コンパイルの終了の保障

インライン展開はコンパイルの中間で行う最適化の一種である。そのため、ある程度コンパイル時間が増加してしまうのは許されるが、極端な増加は好ましくない。特に、コンパイルが終了しないという自体は必ず防がなければならない。

コンパイルの終了を保障するために最も気をつけなければならない点として、再帰関数のインライン展開がある。再帰関数は、呼び出される関数をインライン展開しても、その展開したコードの中に再びその関数が呼ばれる式が現れる。そのため、その展開されたコードに対して、再びインライン展開を実行することを繰り返すと無限ループに陥り、コンパイルが終了しない事態となってしまう。

```
let
  fun f x = x +  (f (x - 1))
in
  f 100
end
```

$\Rightarrow$

```
let
  fun f x = x + (f (x - 1))
in
  let val x = 100
  in x + (f (x - 1))
  end
end
```

上記のように、再帰関数 f に 100 が与えられる式をインライン展開すると、展開した式に再び再帰関数 f の呼び出しが現れ、この式をさらにインライン展開し、それを繰り返すとコンパイルが終了しない。これを防ぐために、一般に再帰関数の呼び出し式はインライン展開の対象から外すこととなる。

2.6 自由変数の参照

関数適用式をインライン展開するためには、呼び出される関数が実行するコードが判明している必要がある。しかしながら、判明したコードを単純に展開することはプログラムの意味を変えてしまう可能性があり注意する必要がある。

以下の例を見てみる。

```
let
  val x = 1
  fun f y = x + y
  val x = 10
in
  f 0
end
```

$\Rightarrow$

```
let
  val x = 1
  fun f y = x + y
  val x = 10
in
  let val y = 0
  in  x + y
  end
end
```

この変換は関数 f をインライン展開してるのであるが、関数 f の中に存在している自由変数 x の参照が、インライン展開後のコードでは異なっている。よって、この関数 f のインライン展開は、プログラムの意味が変化しており正しい変換ではない。上記の f 0 をインライン展開する場合は、展開前後で同一の自由変数 x を参照していなければならない。

このように、ある変数が宣言された後に、同様の変数名で再び変数が宣言された場合、最初に宣言した変数が参照できなくなることでインライン展開をすることが不可能となってしまう。

このような問題の解決には、一般に束縛変数の名前を変更することで対応する。束縛変数は仮の名前であり、その名前事態には意味がないため変更してもプログラムの意味を変えることはない。先ほどの式であれば、以下のように束縛変数名の付け替えを行うことで正しくインライン展開が行える。

```
let
  val z = 1
  fun f y = z + y
  val x = 10
in
  let val y = 0
  in  z + y
  end
end
```

上記の式では、変数 x で重ならないように、初めの束縛変数 x を z に変更している。

名前の付け替えをどのように行うかは様々な手法が存在するが、よく知られた方法にはインライン展開を行う前に、あらかじめすべての束縛変数の名前に関して重複がないようプログラム全体で唯一の名前に変更することで、上記のような問題自体起こらないようにする手法がある。

2.7 高階関数

高階関数を扱える関数型言語では、関数の引数に関数を渡すことや、関数の結果として関数を返すことが可能である。

`fun f x = x 1` という関数があった場合、引数 x は関数が渡されることは明らかである。しかし、引数 x が束縛する関数は不明なため、x 1 をインライン展開することは通常できない。flow-directed inlining[6] では、これを次のように状況を限定してインライン展開をしている。まず、プログラム全体を解析して引数 x が束縛する可能性のある関数を把握する。解析の結果、引数 x が束縛する可能性のある関数がただ 1 つしかない場合にのみインライン展開を行うことが可能となる。ただ 1 つしかないとは、すなわち、その関数こそが関数適用が起こって関数本体が実行される時に、引数 x が束縛している関数というこ

とである。よって、仮引数 x はその関数が束縛されているとみなしてインライン展開を行うことが可能である。

次に、関数の返り値として返された関数に対する関数適用をインライン展開することを考える。例として以下のコードを見る。

```
let
  fun f x = let val y = h x
            in  fn z => y + z
            end
  val g = f M
in
  g 1
end
```

$\Rightarrow$

```
let
  fun f x = let val y = h x
            in fn z => y + z
            end
  val g = f M
in
  let val z = 1
  in y + z
  end
end
```

M は任意の式を表し、h は任意の関数を表す。上記は g 1 をインライン展開しているわけであるが明らかに間違っている。g は関数 f を式 M に適用することで返される関数を束縛しており、その関数のコードは `fn z = y + z` であることは判明できる。よって、g を 1 に適用した場合、z に 1 が渡されて y + z が実行されることは確かであるが、自由変数 y の値が不明である。自由変数 y は関数 f の中で定義されているため、関数 f の外から参照することはできない。関数 f のコードを見ると、y は関数適用式 h x の値である。さらに、x は式 M の値を束縛していることがわかるなら、h M を実行することで自由変数 y が束縛している値を知ることも可能である。しかしながら、これは式 M もしくは、関数 h に副作用がある場合は常に同一の値を返すとは限らないため、自由変数 y が束縛している値と同一である保障はない。仮に、どちらとも副作用がないとすれば自由変数 y と同一の値となるが、本来必要のない h M の実行を行う必要があり、逆に処理速度が落ちてしまうことが予想される。

flow-direted inlining では関数を束縛している変数から、その関数の中に存在する自由変数を取得するプリミティブ命令 cl-ref を導入することで解決を図っている。例えば、上記の例だと `val y = cl-ref(g, y)` とすることで g の中の自由変数 y を取得することになる。そのような命令があれば、y の値が取得できるためインライン展開が可能となる。しかしながら、プリミティブ命令の導入なしでは、このような関数に対してインライン展開をすることは難しい。

第3章　クロージャ変換後でのインライン展開

本論文では、自由変数の参照を正しく効率的に解決するために、クロージャ変換後の中間言語でインライン展開を行うことを提案する。提案するアルゴリズムでは、特別なプリミティブ命令を導入することなく、正しく自由変数の参照を解決することが可能であるため、自由変数に影響されることなくインライン展開を行うことができる。

3.1　クロージャ変換

まず、クロージャ変換について説明する。

ML などの言語では関数をネストして宣言することを許している。ネストされた関数の中では、外側で定義された変数を参照することが可能である。しかしながら、ネストされた関数のまま機械語にコンパイルするのは一般に複雑であり難しい。

クロージャ変換の主な目的は、機械語へのコンパイルを容易にするために、関数のネスト構造を解消し平坦な構造に変換することにある。すなわち、すべての関数をトップレベルで定義することである。しかしながら、ネストされた関数をそのままトップレベルへと移動すると、その関数に含まれている自由変数が参照できなくなる可能性があるため単純には移動できない。ネストされた関数をトップレベルに持っていくためには、その関数の外側で定義された変数、すなわち、ネストされた関数の中に存在する自由変数の参照をなくす必要がある。そのために、クロージャ変換では関数宣言を、関数コードと環境の２つの組から構成されるクロージャへの生成命令に変換する。環境とは対象となる関数内に存在する自由変数の値を保存しているデータである。関数コードは対象となる関数の中に存在していたすべての自由変数の参照を、環境からのアクセスに置き換えたものである。

クロージャ生成命令は関数コードと環境の２つを取る。関数コードがアクセスする環境はこの時に渡される環境である。関数宣言をクロージャ生成命令に置き換えるために、クロージャ変換は環境の生成、関数内にある自由変数を環境からのアクセスに置き換える処理を行い、関数宣言は関数コードと環境の 2 つを取るクロージャの生成命令へと置き換えられる。クロージャに実引数が適用されることで、関数コードが実行される。

関数コードに存在したいた自由変数は、すべて環境からのアクセスに置き換わっているため、関数コードには自由変数が含まれない。よって、すべての関数コードはトップレベルに移動することが可能であり、関数のネスト構造を解消することが可能となる。

以下はクロージャ変換の例である。

```
let
  fun f x =
    let
      fun g y = x + y
    in
      g x
    end
in
 f 0
end
```

$\Rightarrow$

```
let
  code g_code env y = (#x env) + y
  code f_code env x =
    let
      val g_env = {x = x}
      val g = Closure(g_code, g_env)
    in
      g x
    end
  val f_env = {}
  val f = Closure(f_code, f_env)
in
  f 0
end
```

ここで、Closure は関数コードと環境の組からクロージャを生成する命令である。環境はすべてレコード型であり自由変数の名前と同一のラベルを持っている。よって、すべての自由変数は、環境からその変数名と同一のラベル名の値を取り出す命令に置き換えられている。

関数 g の関数コードである g_code では、自由変数 x はレコード型である環境 env のラベル x の参照に置き換えられており、クロージャ生成命令でラベル x を持ったレコードが渡されている。結果、関数コードの中に自由変数は無くなり、関数 f の中にネストされて宣言されていたのがトップレベルへと移動されている。関数 f についても同様にクロージャ生成命令に置き換えられている。ただし、関数 f の中には自由変数が存在していないので、クロージャ生成命令では空のレコードを環境として渡している。

このようにしてすべての関数宣言は、関数コードと環境の組から構成されるクロージャの生成命令に置き換えられ、さらに、関数のネスト構造が解消されている。

3.2 クロージャ変換後でのインライン展開

クロージャ変換後の中間言語では、すべての関数宣言は関数コードと環境の組から構成されるクロージャに置き換えられ、関数内に存在していた自由変数は環境への参照に置き換えられることで関数コードに自由変数は存在しなくなる。すべての自由変数が環境への参照に置き換わるということは、環境への参照は自由変数への参照と同じ意味をもつ。よって、環境をアクセスすることができるのであれば、それは元の関数内に存在していた自由変数へのアクセスが可能であるといえる。

```
let
  fun f x =
        let val y = h x
        in  fn z => y + z
        end
  val g = f M
in
  g 1
end
```

$\Rightarrow$

```
let
  code f_code2 env z =
                (#y env) + z
  code f_code1 env x =
   let
     val y = h x
     val f_env2 = {y = y}
   in
     Closure(f_code2, f_env2)
   end
  val f_env1 = {}
  val f = Closure(f_code1, f_env1)
  val g = f M
in
  g 1
end
```

上記はクロージャ変換前後のそれぞれのプログラムである。

クロージャ変換前のコードを見る。束縛変数 g は、式 M の結果に関数 f を適用することで返される関数を束縛している。関数を束縛した変数 g を 1 に適用する式をインライン展開するためには、変数 g が束縛している関数のコード、つまり `y + z` に存在する自由変数 y を正しく参照しなければならない。しかしながら、自由変数 y は関数 f の中で定義されており関数 f の中からしか参照することができない。よって、関数 f の外では自由変数 y の参照が正しくできないために、インライン展開をすることもできない。

次にクロージャ変換後のコードを見る。クロージャ変換後においても、クロージャg に 1 が適用されると f_code2 が実行されることは判明できる。f_code2 のコードは `(#y env) + z` であり環境 env への参照が現れる。この際に参照される環境 env は f_code1 で定義されているために、f_code1 の外から直接参照することは不可能であり、クロージャ変換前のコードと同様にインライン展開をすることができない。

そこで、クロージャのデータ構造を利用して環境を参照することを考える。クロージャとは関数コードと環境の組で構成されている。すなわち、実行時には第一要素と第二要素を持つ組と同様のデータ構造に、その関数コード (関数コードを指すポインタ) と環境が格納されており、特別なデータ構造に格納されているわけではない。従ってクロージャを組とみなして、クロージャの第二要素を取り出す命令は環境を取り出すことに一致する。すなわち、クロージャg に 1 が適用され f_code2 が実行された際に参照される環境は、クロージャg の第二要素を取り出すことにより取得することができる。よって、直接に環境を参照できない場合は、実行時にクロージャの第二要素を取得する命令コードを挿入し、その値を参照することにより、実行時には正しく環境を参照することができるため、先ほどのクロージャ変換後のプログラムは以下のようにインライン展開をすることが可能と

なる。

```
let
  code f_code2 env z = (#y env) + z
  code f_code1 env x =
   let
     val f_env1 = h x
     val f_env2 = {y = y}
   in
     Closure(f_code2, f_env2)
   end
  val f_env1 = {}
  val f = Closure(f_code1, f_env1)
  val g = f M
in
  let
    val env = #2 g
    val z = 1
  in
    (#x env) + y
  end
end
```

クロージャg の 2 番目の要素を let 式により、f_code2 の環境の仮引数名 env で名前付けしている。これにより、実行時に env は f_code2 がアクセスすべき環境を束縛することとなり、結果的に f_code2 の環境 env の参照が正しく解決されるためインライン展開が可能となる。

このように、クロージャ変換後の中間言語であれば環境を動的に取得する命令を挿入することで、すべての自由変数を正しく参照することにつながり、自由変数に影響されることなくインライン展開が可能となる。これは、関数コードの判明のみでインライン展開の候補とすることができるため、クロージャ変換前の中間言語では不可能だったインライン展開が可能となり、より多くの関数適用をインライン展開の候補とすることが可能となる。

3.3 インライン展開のアルゴリズム

ここでは、クロージャ変換後の中間言語を対象としたインライン展開のアルゴリズムを示す。

まず、図 3.1 にクロージャ変換後の中間言語が持つ式の定義を示す。

$$
\begin{aligned}
M \quad := \quad & c(c \in Const) \\
| \quad & x \\
| \quad & (x_1, x_2) \\
| \quad & x[1] \\
| \quad & x[2] \\
| \quad & \lambda x_{env}.\lambda x_{arg}.M \\
| \quad & \lambda x_f.\lambda x_{env}.\lambda x_{arg}.M \\
| \quad & Closure(x_1, x_2) \\
| \quad & Apply(x_1, x_2) \\
| \quad & if\ x\ then\ M_1\ else\ M_2 \\
| \quad & let\ x = M_1\ in\ M_2 \quad (M_1 \neq let\ x' = M_1'\ in\ M_2')
\end{aligned}
$$

図 3.1: クロージャ変換後の中間言語の定義

各、式の意味について述べる。

- c は定数を表す。
- x は変数への参照を表す。
- (x_1, x_2) は第一要素に x_1、第二要素に x_2 を持った組を生成する。
- $x[1]$ は x が束縛している組の第一要素を取り出す。
- $x[2]$ は x が束縛している組の第二要素を取り出す。
- $\lambda x_{env}.\lambda x_{arg}.M$ は関数コードを表す。関数コードはクロージャ生成命令によって環境を受け取ることによりクロージャを生成し、それに実引数が適用されることで式 M が実行される。x_{env} には環境が束縛され、x_{arg} には実引数が束縛される。
- $\lambda x_f.\lambda x_{env}.\lambda x_{arg}.M$ は再帰関数のコードを表す。$\lambda x_{env}.\lambda x_{arg}.M$ と同じくクロージャ生成命令によって、環境を受け取ることによりクロージャを生成し、それに実引数が適用されることで式 M が実行される。$\lambda x_{env}.\lambda x_{arg}.M$ との違いは、関数が実行される際に、仮引数 x_f は自分自身である $\lambda f.\lambda env.\lambda arg.M$ を束縛して、その元で式 M を実行することである。また、x_{env} には環境が束縛され、x_{arg} には実引数が束縛される。

- $Closure(x_1, x_2)$ は再帰関数を含む関数コードを束縛した変数 x_1 と環境を束縛した変数 x_2 を取ってクロージャを生成する。
- $Apply(x_1, x_2)$ はクロージャを束縛した x_1 に実引数 x_2 を適用し、クロージャが持つ関数コードを実行する。
- $if\ x\ then\ M_1\ else\ M_2$ は x が true であれば M_1 を、false であれば M_2 を実行する。$(true, false \in Const)$
- $let\ x = M_1\ in\ M_2$ は式 M_1 の表す値に x の名前を付けて M_2 を実行する。ただし、式 M_1 は let から始まる構文ではないと仮定する。

また、この式は A-normal form の形を取ると仮定する。A-normal form では計算の途中過程はすべて名前付けされるため、$let\ x = M_1\ in\ M_2$ の場合、M_1 は let から始まる構文ではないことにつながる。また、この式で使用されるすべての束縛変数名は、プログラム中で唯一のものであり変数名の重複はないものとする。束縛変数名とは関数コードの環境名と引数名、そして let 式で名前付けされる変数名のことである。BV(M) は式 M の中に含まれる束縛変数名の集合を表す。図 3.2 のように定義される。

$$
\begin{aligned}
BV(c) &= \emptyset \\
BV(x) &= \emptyset \\
BV((x_1, x_2)) &= \emptyset \\
BV(x[1]) &= \emptyset \\
BV(x[2]) &= \emptyset \\
BV(\lambda x_{env}.\lambda x_{arg}.M) &= \{x_{env}, x_{arg}\} \cup BV(M) \\
BV(\lambda x_f.\lambda x_{env}.\lambda x_{arg}.M) &= \{x_f, x_{env}, x_{arg}\} \cup BV(M) \\
BV(Closure(x_1, x_2)) &= \emptyset \\
BV(Apply(x_1, x_2)) &= \emptyset \\
BV(if\ x\ then\ M_1\ else\ M_2) &= BV(M_1) \cup BV(M_2) \\
BV(let\ x = M_1\ in\ M_2) &= \{x\} \cup BV(M_1) \cup BV(M_2)
\end{aligned}
$$

図 3.2: 束縛変数名の集合の定義

図 3.3 にインライン展開のアルゴリズム I を示す。
$e[x_1/x_2]$ は、式 e の中の変数 x_1 への参照をすべて x_2 への参照へ置き換えた式を表す。
インラインアルゴリズム I は 3 つの値 F, A, e を取る。それぞれについて説明する。

$$I(F,A,c) = \begin{cases} let\ p = c\ in\ I(F,A',e) & \textbf{if } A = (p,e) :: A' \\ \\ c & \textbf{if } A = nil \end{cases}$$

$$I(F,A,x) = \begin{cases} let\ p = x\ in\ I(F,A',e) & \textbf{if } A = (p,e) :: A' \\ \\ x & \textbf{if } A = nil \end{cases}$$

$$I(F,A,(x_1,x_2)) = \begin{cases} let\ p = (x_1,x_2)\ in\ I(F,A',e) & \textbf{if } A = (p,e) :: A' \\ \\ (x_1,x_2) & \textbf{if } A = nil \end{cases}$$

$$I(F,A,x[i]) \quad (i = 1\ or\ i = 2)$$
$$= \begin{cases} let\ p = x[i]\ in\ I(F,A',e) & \textbf{if } A = (p,e) :: A' \\ \\ x[i] & \textbf{if } A = nil \end{cases}$$

$$I(F,A,\lambda env.\lambda arg.e)$$
$$= \begin{cases} \textbf{let } e_1 = \lambda env.\lambda arg.I(\emptyset,nil,e) \\ \textbf{in } let\ p = e_1\ in\ I(F\{p:(e_1,\varepsilon)\},A',e_2) & \textbf{if } A = (p,e_2) :: A' \\ \\ \lambda env.\lambda arg.I(\emptyset,nil,e) & \textbf{if } A = nil \end{cases}$$

$$I(F,A,\lambda f.\lambda env.\lambda arg.e) = \begin{cases} let\ p = \lambda f.\lambda env.\lambda arg.I(\emptyset,nil,e) \\ in\ I(F,A',e_1) & \textbf{if } A = (p,e_1) :: A' \\ \\ \lambda f.\lambda env.\lambda arg.I(\emptyset,nil,e) & \textbf{if } A = nil \end{cases}$$

$$
\begin{array}{l}
I(F, A, Closure(x_1, x_2)) \\
\quad \begin{cases}
let\ p = Closure(x_1, x_2) & \textbf{if } A = (p, e_1) :: A' \\
in\ I(F\{p : (e_2, x_2)\}, A', e_1) & \textbf{and } F(x_1) = (e_2, env) \\
 & \\
let\ p = Closure(x_1, x_2) & \textbf{if } A = (p, e_1) :: A' \\
in\ I(F, A', e_1) & \textbf{and } x_1 \notin dom(F) \\
 & \\
Closure(x_1, x_2) & \textbf{if } A = nil
\end{cases} \\
 \\
I(F, A, Apply(x_1, x_2)) \\
= \begin{cases}
\textbf{if } env = \varepsilon & \textbf{if } isIinline(Apply(x_1, x_2)) = true \\
\textbf{then } let\ env_2 = x_1[2] & \textbf{and } F(x_1) = (\lambda env_1.\lambda arg_1.e_1, env) \\
\qquad in\ I(F, A, e_2[x_2/arg_2]) & \textbf{where } \lambda evn_2.\lambda arg_2.e_2 = \\
\textbf{else } I(F, A, e_2[env/env_2, x_2/arg_2]) & \quad replaceVarName(\lambda env_1.\lambda arg_1.e_1) \\
 & \\
let\ p = Apply(x_1, x_2)\ in\ I(F, A', e_1) & \textbf{if } isInline(Apply(x_1, x_2)) = false \\
 & \textbf{and } A = (p, e_1) :: A' \\
 & \textbf{and } GetNextCode(x_1, x_2, F) = \varepsilon \\
 & \\
let\ p = Apply(x_1, x_2) & \textbf{if } isInline(Apply(x_1, x_2)) = false \\
in\ I(F\{p : (e_2, \varepsilon)\}, A', e_1) & \textbf{and } A = (p, e_1) :: A' \\
 & \textbf{and } GetNextCode(x_1, x_2, F) = e_2 \\
 & \\
Apply(x_1, x_2) & \textbf{if } isInline(Apply(x_1, x_2)) = false \\
 & \textbf{and } A = nil
\end{cases}
\end{array}
$$

$$
I(F, A, if\ x\ then\ e_1\ else\ e_2)
$$

$$
= \begin{cases} let\ p = if\ x\ then\ I(F, nil, e_1)\ else I(F, nil, e_2) & \\ in\ I(F, A', e) & \textbf{if } A = (p, e) :: A' \\ \\ if\ x\ then\ I(F, nil, e_1)\ else\ I(F, nil, e_2) & \textbf{if } A = nil \end{cases}
$$

$$
I(F, A, let\ x = e_1\ in\ e_2) = I(F, (x, e_2) :: A, e_1)
$$

図 3.3: インライン展開アルゴリズム

- e は直後にインライン展開を行う式である。
- F は、変数の集合から関数コード $(\lambda env.\lambda arg.e)$ と環境を束縛している変数名の組への関数である。ただし、環境については不明、もしくは必要ない場合は ε とする。

  また、$dom(F)$ は F に含まれる変数の集合を表し、$F\{x : (e, env)\}$ は、F を $x : (e, env)$ で拡張して得られる関数を表す。

  この関数 F は、インライン展開をする際に適用されるクロージャが持つ関数コードと環境を取得するために使用する。また、クロージャが生成される時に、クロージャ生成命令に渡される変数が束縛している関数コードを知る必要もあるので、そのためにも使用される。
- A は変数名と式の組のリストである。最後は nil 要素であり要素が 1 つもない場合は nil のみである。変数名は let 式で定義される変数名、式はその変数に値が束縛された後に実行される式であり、それが実行される順に並んでいる。
  $letx = e_1 ine_2$ では変数名 x、式 e_2 の組がリストの 1 要素となる。

  例えば、

  $$
  M = I(F, (x_1, e_1) :: (x_2, e_2) :: \cdots :: (x_n, e_n) :: nil, e)
  $$

  とすると M は以下の式と等しい意味を持つ。

  $$
  \begin{array}{l} let\ x_1 = e\ in \\ let\ x_2 = e_1\ in \\ \cdots \\ let\ x_n = e_{n-1}\ in \\ e_n \end{array}
  $$

リストの先頭が (p, e) であり残りのリストを A' とすると、全リストを (p, e)::A' と表記する。

また、ABV(A) は以下の意味を持つ集合である。

$$\begin{aligned} ABV(nil) &= \emptyset \\ ABV((x_1, e_1) :: A') &= \{x_1\} \cup BV(e_1) \cup ABV(A') \end{aligned}$$

このリストは A-normal form の表現を保持するために使用される。

以下の式を考える。

$$\begin{aligned} &let\ x_1 = \lambda env.\lambda arg.let\ y = 1\ in\ y\ in \\ &let\ x_2 = (0, 0)\ in \\ &let\ x_3 = Closure(x_1, x_2)\ in \\ &let\ x_4 = Apply(x_3, 0)\ in \\ &x_4 \end{aligned}$$

ここで $Apply(x_3, 0)$ が、インライン展開されるとする。すると、$let\ x_4 = let\ y = 1\ in\ y\ in\ x_4$ となってしまい、この式で直接置き換えると A-normal form の表現が壊れてしまう。これを防ぐためにリスト A を導入している。

プログラム全体の式を e とし、それをインライン展開する場合は $I(\emptyset, nil, e)$ を実行する。続いて、アルゴリズムの中で使用されている 3 つの関数について説明する。

- replaceVarName($\lambda env.\lambda arg.e$) は env, arg、そして e の中で束縛されるすべての束縛変数の名前をプログラム全体において唯一な名前に付け替える関数である。

  インライン展開をした場合に展開する関数コードをそのまま置くと、同一の変数名が複数回現れてしまい変数名の重複はないという仮定を満たさない。これは、すべての変数名はプログラムの中で唯一であることを保存するためにある。

- GetNextCode(x_1, x_2, F) は $Apply(x_1, x_2)$ がもしクロージャを返し、かつ、そのクロージャが保存している関数コードを判明することができるならばそのコードを返す。$Apply(x_1, x_2)$ がクロージャを返さない、もしくは、判明できないなら ε を返す。

- isInline は、関数適用の式をインライン展開するか否かを判別する関数である。これにはインライン展開が可能かどうかの判別と共に、コードサイズとの関係から実際にインライン展開をするかどうかの最終判定が含まれる。$isInline(Apply(x_1, x_2)) = true$ の場合は、インライン可能なので $x_1 \in dom(F)$ が保障される。

各変換規則について説明する。

- $e = c$ の場合
  $A = (p_1, e_1) :: A'$ の場合は、let 式によって c を p_1 に束縛して、その元で $I(F, A', e_1)$ を実行するコードに変換する。$A = nil$ の時は、そのまま c を出力する。

- $e = x$ の場合
  $A = (p_1, e_1) :: A'$ の場合は、let 式によって x を p_1 に束縛して、その元で $I(F, A', e_1)$ を実行するコードに変換する。$A = nil$ の時は、そのまま x を出力する。

- $e = (x_1, x_2)$ の場合
  $A = (p_1, e_1) :: A'$ の場合は、let 式によって (x_1, x_2) を p_1 に束縛して、その元で $I(F, A', e_1)$ を実行するコードに変換する。$A = nil$ の時は、そのまま (x_1, x_2) を出力する。

- $e = x[i]$ の場合 (i = 1 or i = 2)
  $A = (p_1, e_1) :: A'$ の場合は、let 式によって $x[i]$ を p_1 に束縛して、その元で $I(F, A', e_1)$ を実行するコードに変換する。$A = nil$ の時は、そのまま $x[i]$ を出力する。

- $e = \lambda env.\lambda arg.e$ の場合
  $A = (p_1, e_1) :: A'$ の場合は、まず関数の本体についてインライン展開アルゴリズム I を実行しそれを e_2 とする。関数本体についてはそこを 1 つのコードのトップとみなせるので、$F = \emptyset$、$A = nil$ として実行する。その後、let 式で e_2 を p_1 に束縛してその元で $I(F\{p_1 : (e_2, \varepsilon)\}, A', e_1)$ を実行するコードに変換する。ここで e_1 の中では、変数 p_1 は e_2 を束縛しているので F を $p_1 : (e_2, \varepsilon)$ で拡張している。環境は必要ないので ε である。$A = nil$ の場合は、関数の本体をインライン展開アルゴリズム I で変換したコードを出力する。

- $e = \lambda f.\lambda env.\lambda arg.e$ の場合
  $A = (p_1, e_1) :: A'$ の場合は、基本的には $\lambda env.\lambda arg.e$ と同様の変換である。違いは、e_1 をインライン展開する際に F を拡張していないことである。これは、p_1 は再帰関数の関数コードを束縛することとなるためである。再帰関数のインライン展開は、コンパイルが終了しな事態を導く可能性があるのでインライン展開の対象から外している。$A = nil$ の場合は、関数本体についてインライン展開アルゴリズム I で変換したコードを出力する。

- $e = Closure(x_1, x_2)$ の場合
  $A = (p_1, e_1) :: A'$ かつ $x_1 \in dom(F)$ の場合は、let 式によって $Closure(x_1, x_2)$ を p_1 に束縛して、その元で $I(F\{p_1 : (e_2, x_2)\}, A', e_1)$ を実行するコードに変換している。ここで、F を $p_1 : (e_2, x_2)$ で拡張する。p_1 が束縛するクロージャは、x_1 が束縛する関数コードをもつので e_2 は F から取得する。もし、$x_1 \notin dom(F)$ の場合は、上記の変換との唯一の違いは F を拡張しないことである。これは、クロージャがもつ関数コー

ドを知ることができないためである。$A = nil$ の場合は、そのまま $Closure(x_1, x_2)$ を出力する。

- $e = Apply(x_1, x_2)$ の場合
  これは、4 つの場合に分けられている。その中で、実際にインライン展開を行うのは 1 番目の場合である。

  インライン展開をするためには $Apply(x_1, x_2)$ が実行するコードを知る必要がある。そのため、まず F の集合の中から x_1 が束縛するクロージャが持つ関数コードと環境の組を取得する。関数コードについては変数名の重複が起こらないように束縛変数名の置き換えを行う。ここで、F から環境が取得できた場合と取得できない場合とで変換方法が異なる。環境が取得できた場合は、関数コードの中の仮引数を実引数 x_2 で、環境名を集合 F から取得した環境で置き換える。実引数 x_2 と環境はどちらも変数なので、直接仮引数と置き換えても問題はない。そして、その置き換えたコードを再び、同一の F, A の元でインライン展開アルゴリズム I により変換する。F から環境が取得できなかった場合は、クロージャ x_1 の第 2 要素を取得した値を let 式を使用して環境名で束縛する。これにより、実行時に環境は正しく参照される。そして、その元で関数コードの仮引数を実引数 x_2 で置き換えたコードを、再び同一の F, A の元で、インライン展開アルゴリズム I により変換する。どちらの場合も変換したコードに対して、同一の F, A で再びインライン展開アルゴリズム I を実行している。これは、展開した関数コードに対して A-normal form を保持する目的の他に、展開したコードの中に新しくインライン展開可能なコードが生成された場合に、それについてもインライン展開を行うためでもある。

  2 番目、3 番目については、$A = (p_1, e_1) :: A'$ であり、かつ、$Apply(x_1, x_2)$ について、$isInline$ 関数により展開しないと判断された場合である。この場合は、どちらとも let 式によって $Apply(x_1, x_2)$ を p_1 に束縛して、その元で $I(F', A', e_1)$ を実行するコードを出力する。ここで、F' は、$Apply(x_1, x_2)$ の返す値が、クロージャであり、かつ、そのクロージャが保持する関数コードが判明するのであれば、F を p_1 と p_1 が持つ関数コードで拡張する。それ以外は $F' = F$ である。

  最後はインライン展開をせず、かつ、$A = nil$ の場合である。これは $Apply(x_1, x_2)$ をそのまま出力する。

- $e = if\ x\ then\ e_1\ else\ e_2$ の場合
  $A = (p, e) :: A'$ の場合は、e の中の e_1, e_2 に関してインライン展開を行ったコードを let 式によって p に束縛して、その元で $I(F, A', e)$ を実行するコードに変換する。$A = nil$ の時は、e の中の e_1, e_2 に関してインライン展開を行ったコードを出力する。

- $e = let\ x = e_1\ in\ e_2$ の場合
  $I(F, (x, e_2) :: A, e_1)$ を変換後のコードとする。すなわち、A の先頭に (x, e_2) を格納しておき、まず e_1 についてインライン展開を試みる。

アルゴリズムIでは、環境をクロージャから取得する命令コードを挿入することにより、自由変数に影響されずにインライン展開を行うことができる。ただし、アルゴリズムIは非常にシンプルであり、関数Fだけではすべてのクロージャに対して、それが持つ関数コードと環境を把握することはできない。例えば、以下の式を考える。

$$
\begin{aligned}
&let\ x_1 = \lambda env.\lambda arg.e_1 \\
&in\ x_2 = (x_1, M_1) \\
&in\ x_3 = x_2[1] \\
&in\ x_4 = Closure(x_3, M_2) \\
&Apply(x_4, M_3)
\end{aligned}
$$

Mは任意の式を表す。この場合、$Apply(x_4, M_3)$をインライン展開するにはx_4がFに含まれていなければならないが、アルゴリズムIでは$x_4 \notin dom(F)$となってしまう。しかし、これは事前に別の最適化を行い、より単純な構造に変換することでインライン展開が可能となる。もしくは、アルゴリズムIを拡張し、F以外に変数の値を解析するためのパラメータを追加することは容易であり、それにより詳細な解析が可能となり、上記のような式に対しても応することが可能となる。

3.4 型保存定理の証明

この節では図 3.3 のアルゴリズムが、変換前後で型を保存することを証明する。

3.4.1 型システム

入力言語に対して型システムの定義をする。
まず、型τを以下に定義する。

$$\tau := b \mid \tau \to \tau \mid \tau \times \tau \mid Code(\tau, \tau) \mid \bot$$

bは基底型であり、その型を持つ定数$c : b \in Const$があるとする。$\tau_1 \to \tau_2$は、τ_1の型を受け取り、τ_2の型を返す型を表す。$\tau_1 \times \tau_2$は、τ_1とτ_2の型を持つ組を表す。$Code(\tau_1, \tau_2)$は、環境の型τ_2をクロージャ生成時に受け取って、$\tau_1 \times \tau_2$の型をもつクロージャを返す関数コードの型を表す。$\bot$は、if文の型判定をするために導入した特別な型である。

入力式eが型環境Γの元でτを持つことを、

$$\Gamma \triangleright e : \tau$$

と書く。Γは型環境であり、変数の有限集合から型への関数である。$\mathrm{dom}(\Gamma)$は型環境に含まれる変数の集合を表す。

型システムとは型を導出する証明システムである。以下の形の推論規則を持つ。

$$\frac{\Gamma \triangleright e_1 : \tau_1 \quad \Gamma \triangleright e_2 : \tau_2 \quad \ldots \quad \Gamma \triangleright e_n : \tau_n}{\Gamma \triangleright e : \tau}$$

これは、$\Gamma \triangleright e_1 : \tau_1, \quad \ldots \quad \Gamma \triangleright e_n : \tau_n$ が成り立つなら、$\Gamma \triangleright e : \tau$ が成り立つことを示している。すわなち、$\Gamma \triangleright e : \tau$ であるならば型システムよりその型が導出可能である。

入力言語に対する型システムを図 3.4 に示す。

$$(const) \qquad \Gamma \triangleright c : \tau \quad (c : \tau \in Const)$$

$$(var) \qquad \Gamma\{x : \tau\} \triangleright x : \tau$$

$$(prod) \qquad \frac{\Gamma \triangleright x_1 : \tau_1 \quad \Gamma \triangleright x_2 : \tau_2}{\Gamma \triangleright (x_1, x_2) : \tau_1 \times \tau_2}$$

$$(proj) \qquad \frac{\Gamma \triangleright x : \tau_1 \times \tau_2}{\Gamma \triangleright x[i] : \tau_i} \ (i = 1 \ or \ i = 2)$$

$$(code) \qquad \frac{\{env : \tau_{env}, arg : \tau_1\} \triangleright e : \tau_2}{\Gamma \triangleright \lambda env.\lambda arg.e : Code(\tau_1 \to \tau_2, \tau_{env})}$$

$$(reccode) \qquad \frac{\{f : Code(\tau_1 \to \tau_2, \tau_{env}), env : \tau_{env}, arg : \tau_1\} \triangleright e : \tau_2}{\Gamma \triangleright \lambda f.\lambda env.\lambda arg.e : Code(\tau_1 \to \tau_2, \tau_{env})}$$

$$(closure) \qquad \frac{\Gamma \triangleright x_1 : Code(\tau_1 \to \tau_2, \tau_{env}) \quad \Gamma \triangleright x_2 : \tau_{env}}{\Gamma \triangleright Closure(x_1, x_2) : (\tau_1 \to \tau_2) \times \tau_{env}}$$

$$(app) \qquad \frac{\Gamma \triangleright x_1 : (\tau_1 \to \tau_2) \times \tau_{env} \quad \Gamma \triangleright x_2 : \tau_1}{\Gamma \triangleright Apply(x_1, x_2) : \tau_2}$$

$$(env) \qquad \frac{\Gamma \triangleright e : (\tau_1 \to \tau_2) \times \tau_{env}}{\Gamma \triangleright e : (\tau_1 \to \tau_2) \times \bot}$$

$$(if) \qquad \frac{\Gamma \triangleright x : bool \quad \Gamma \triangleright e_1 : \tau \quad \Gamma \triangleright e_2 : \tau}{\Gamma \triangleright if \ x \ then \ e_1 \ else \ e_2 : \tau} \qquad (true : bool \in Const \quad false : bool \in Const)$$

$$(let) \qquad \frac{\Gamma \triangleright e_1 : \tau_1 \quad \Gamma\{x : \tau_1\} \triangleright e_2 : \tau_2}{\Gamma \triangleright let \ x = e_1 \ in \ e_2 \ : \tau_2}$$

図 3.4: 入力言語の型システム

規則 (code)、(reccode) は、関数コードに自由変数は含まれないことを表している。クロージャを表す型は組を表す型である。ただし、その第一要素は τ_1 を取って τ_2 を返す関

数型であり、第二要素は環境の型を表している。これはクロージャはコードと環境の組から構成されるためである。ただし、そのようにクロージャの型に環境の型を含めると規則 (if) に問題が生じる。以下の型が成り立つとする。

$$\Gamma \triangleright x : bool \quad \Gamma \triangleright e_1 : (\tau_1 \to \tau_2) \times \tau_3 \quad \Gamma \triangleright e_2 : (\tau_1 \to \tau_2) \times \tau_4$$

e_1、e_2 はクロージャの型であるが、e_1、e_2 の環境の型が異なるため規則 (if) より、$\Gamma \triangleright if\ x\ then\ e_1\ else\ e_2 : \tau$ は成り立たない。そのため、環境の型は $Apply$ の時には不必要であるのに環境の型が違うために、if 文によって返されたクロージャに $Apply$ をすることができなくなる。これを防ぐために規則 (env) を導入している。規則 (env) によって、クロージャの環境の型は $\perp$ であっても成り立つこととなる。これにより、e_1, e_2 は、以下でも成り立ち、環境の型を同一にすることが可能である。

$$\Gamma \triangleright e_1 : (\tau_1 \to \tau_2) \times \perp \quad \Gamma \triangleright e_2 : (\tau_1 \to \tau_2) \times \perp$$

すなわち $\perp$ は、クロージャが持つ環境の型は不明であることを表しているといえる。

3.4.2 型保存定理の証明

図 3.3 で示したアルゴリズムが型を保存することを、図 3.4 の型システムを用いて証明する。

型を保存することを示すには以下の定理を証明すればよい。

定理 3.1 もし、$\emptyset \triangleright e : \tau$ ならば、$\emptyset \triangleright I(\emptyset, nil, e) : \tau$ である。

これを証明するために、いくつかの補題を証明する。

補題 3.1 もし、$\Gamma \triangleright e : \tau$ かつ、$x \notin dom(\Gamma)$ なら、$\Gamma\{x : \tau'\} \triangleright e : \tau$ である。
同様に、$\Gamma \triangleright e : \tau$ かつ、$x \in dom(\Gamma)$ なるすべての x に対して、$x \notin dom(\Gamma')$ であるなら、$\Gamma \cup \Gamma' \triangleright e : \tau$ である。$\Gamma \cup \Gamma'$ は、Γ を Γ' に含まれるすべての要素で拡張した型環境を表す。

補題 3.2 もし、$\Gamma \triangleright e_1 : \tau$ かつ、式 e_1, e_2がα 同値であれば $\Gamma \triangleright e_2 : \tau$ である。

補題 3.3 もし、$\Gamma\{x_1 : \tau_1\} \triangleright e_1 : \tau_2$ かつ、$\Gamma \triangleright x_2 : \tau_1$ ならば、$\Gamma \triangleright e_1[x_2/x_1] : \tau_2$ である。

上記 3 つの補題については、帰納法によって証明することができる。ここでは省略する。さらに、以下の 2 つの補題が成り立つ。

補題 3.4 $M = I(F, A, e_0)$ とし、任意の環境 Γ について以下の *3* つの条件が成り立つならば、$\Gamma \triangleright M : \tau_n$ である。

条件 *1* $\Gamma \triangleright e_0 : \tau_0$ である。

条件 *2* $x \in dom(F)$ なるすべての x に対して、$F(x) = (e_c, env)$ とすると以下が成り立つ。
・$env = \varepsilon$ の時
ある $\tau_{env}, \tau_{1'}, \tau_{2'}$ があり、$\Gamma \triangleright e_c : \tau_c$、$\Gamma \triangleright x : \tau_c$、かつ $\tau_c = Code(\tau_{1'} \to \tau_{2'}, \tau_{env})$ であるか、もしくは、$\Gamma \triangleright e_c : Code(\tau_{1'} \to \tau_{2'}, \tau_{env})$ かつ、$\Gamma \triangleright x : (\tau_{1'} \to \tau_{2'}) \times \tau_{env}$ である。また、$e_c = \lambda env'.\lambda arg'.e'$ の形をしている。
・$env \neq \varepsilon$ の時
ある $\tau_{env}, \tau_{1'}, \tau_{2'}$ があり、$\Gamma \triangleright e_c : Code(\tau_{1'} \to \tau_{2'}, \tau_{env})$、$\Gamma \triangleright x : (\tau_{1'} \to \tau_{2'}) \times \tau_{env}$ かつ、$\Gamma \triangleright env : \tau_{env}$ である。また、$e_c = \lambda env'.\lambda arg'.e'$ の形をしており、かつ env は変数である。

条件 *3* $A = (p_1, e_1) :: (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil \quad (n \geq 1)$ であり、$V_1 = ABV(A)$、$V_2 = BV(e_0)$、$V = V_1 \cup V_2$とした時、集合 V_1と集合 V_2それぞれの中に重複した要素はなく、V_1と V_2は互いに素である。また、$x \in V$ なるすべての x に対して、$x \notin dom(\Gamma)$ であり、かつ $\Gamma\{p_1 : \tau_0\} \triangleright e_1 : \tau_1,\ \Gamma\{p_1 : \tau_0, p_2 : \tau_1\} \triangleright e_2 : \tau_2,\ \ldots ,\Gamma\{p_1 : \tau_1,\ \ldots ,p_n : \tau_{n-1}\} \triangleright e_n : \tau_n$ が成り立つ。

補題 3.5 $M = I(F, A, e_0)$ とし、任意の環境Γ について、以下の *3* つの条件が成り立つならば、$\Gamma \triangleright M : \tau_0$ である。

条件 *4* $\Gamma \triangleright e_0 : \tau_0$ である。

条件 *5* 補題 *3.4* の条件 *2* と同じ。

条件 *6* $A = nil$ であり、$V = BV(e_0)$ とすると、集合 V の中に重複した要素はなく、かつ、$x \in V$ なるすべての x に対して $x \notin dom(\Gamma)$ である。

[証明]
補題 3.4、補題 3.5 を同時にアルゴリズム I の呼び出し回数に関する帰納法で証明を行う。

任意の環境としてΓ が与えられ、$M = I(F, A, e_0)$ の場合において図 3.3 で使用されている replaceVarName 関数、getNextCode 関数については、次のように定義する。

・$replaceVarName$ について
ある式 e があり、$e' = replaceVarName(e)$ とすると次の関係が成り立つ。この関数は、e の中で定義される束縛変数名の付け替えをするのみなので e と e' は α 同値である。また、プログラム全体で唯一の名前に置き換えとは、$V_1 = ABV(A) \cup BV(e_0)$、$V_2 = BV(e')$ とすると、V_2 の中に重複した要素はなく、かつ V_1と V_2 は互いに素であり、さらに、$x \in V_2$ なるすべての x に対して $x \notin dom(\Gamma)$ となる e' であると定義する。

・$getNextCode$ について

$getNextCode$ 関数が使用される場合は、アルゴリズムの定義より、$e_0 = Apply(x_1, x_2)$ のみである。$getNextCode(x_1, x_2, F) = e(e \neq \varepsilon)$ とすると次の関係が成り立つ。この関数は、$Apply(x_1, x_2)$ がクロージャを返し、それが格納している関数コードが判明すれば返すというものである。$e \neq \varepsilon$ より $Apply(x_1, x_2)$ は、クロージャを返すので、$\Gamma \triangleright Apply(x_1, x_2) : \tau$ が成り立つならば、型システムの定義より、ある $\tau_0, \tau_1, \tau_{env}$ があり、$\tau = (\tau_0 \to \tau_1) \times \tau_{env}$ である。型システムの定義より、$(\tau_0 \to \tau_1) \times \tau_{env}$ は、Closure 構文が $Code(\tau_0 \to \tau_1, \tau_{env})$ の型の関数コードを束縛した変数と τ_{env} の型を束縛した変数を取ることでのみ生成される。e はクロージャの保存している関数コードなので、$\Gamma \triangleright e : Code(\tau_0 \to \tau_1, \tau_{env})$ が成り立つ。また、関数コードを返すので、$e = \lambda env'.\lambda arg'.e'$ の形を取る。

• 補題 3.4 の証明。

$M = I(F, A, e_0)$ とし、Γ を任意の環境と仮定する。e_0 の構造においてそれぞれ場合わけする。

$e_0 = c$ の場合

A $= (p_1, e_1) :: (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil \quad (n \geq 1)$、$\Gamma \triangleright c : \tau_0$ かつ、$\Gamma\{p_1 : \tau_0\} \triangleright e_1 : \tau_1$, $\Gamma\{p_1 : \tau_0, p_2 : \tau_1\} \triangleright e_2 : \tau_2, \ \ldots\ , \Gamma\{p_1 : \tau_1, \ \ldots\ , p_n : \tau_{n-1}\} \triangleright e_n : \tau_n$ であり、$A = (p_1, e_1) :: A'$ とすると、アルゴリズムの定義より、$I(F, A, c) = let\ p_1 = c\ in\ I(F, A', e_1)$ である。ここで、A の大きさ、すなわち n の値によって場合わけする。

n = 1 の場合は A' = nil である。仮定と補題 3.1 より、補題 3.5 の任意の環境を $\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F、nil、e_1)$ とすると、条件 4、条件 5、条件 6 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F, nil, e_1) : \tau_1$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = c\ in\ I(F, nil, e_1) : \tau_1$ であるので、$\Gamma \triangleright I(F, A, c) : \tau_1$ が成り立つ。

$n \geq 2$ の場合は A' $= (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil$ である。仮定と補題 3.1 より、補題 3.4 の任意の環境を $\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F、A'、e_1)$ とすると、条件 1、条件 2、条件 3 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F, A', e_1) : \tau_n$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = c\ in\ I(F, A', e_1) : \tau_n$ であるので、$\Gamma \triangleright I(F, A, c) : \tau_n$ が成り立つ。

・$e_0 = x$ の場合

A $= (p_1, e_1) :: (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil \quad (n \geq 1)$、$\Gamma \triangleright x : \tau_0$ かつ、$\Gamma\{p_1 : \tau_0\} \triangleright e_1 : \tau_1$, $\Gamma\{p_1 : \tau_0, p_2 : \tau_1\} \triangleright e_2 : \tau_2, \ \ldots\ , \Gamma\{p_1 : \tau_1, \ \ldots\ , p_n : \tau_{n-1}\} \triangleright e_n : \tau_n$ であり、$A = (p_1, e_1) :: A'$ とすると、アルゴリズムの定義より、$I(F, A, x) = let\ p_1 = x\ in\ I(F, A', e_1)$ である。ここで、A の大きさ、すなわち n の値によって場合わけする。

n = 1 の場合は A' = nil である。仮定と補題 3.1 より、補題 3.5 の任意の環境を $\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F、nil、e_1)$ とすると、条件 4、条件 5、条件 6 が成り立つ。 よっ

て、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F, nil, e_1) : \tau_1$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = x\ in\ I(F, nil, e_1) : \tau_1$ であるので、$\Gamma \triangleright I(F, A, x) : \tau_1$ が成り立つ。

$n \geq 2$ の場合は A' $= (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil$ である。仮定と補題 3.1 より、補題 3.4 の任意の環境を $\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F、A'、e_1)$ とすると、条件 1、条件 2、条件 3 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F, A', e_1) : \tau_n$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = x\ in\ I(F, A', e_1) : \tau_n$ であるので、$\Gamma \triangleright I(F, A, x) : \tau_n$ が成り立つ。

・$\boldsymbol{e_0 = (x_1, x_2)}$ の場合

A $= (p_1, e_1) :: (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil \quad (n \geq 1)$、$\Gamma \triangleright (x_1, x_2) : \tau_0$ かつ、$\Gamma\{p_1 : \tau_0\} \triangleright e_1 : \tau_1,\ \ \Gamma\{p_1 : \tau_0, p_2 : \tau_1\} \triangleright e_2 : \tau_2,\ \ldots\ , \Gamma\{p_1 : \tau_1,\ \ldots\ , p_n : \tau_{n-1}\} \triangleright e_n : \tau_n$ であり、$A = (p_1, e_1) :: A'$ とすると、アルゴリズムの定義より、$I(F, A, (x_1, x_2)) = let\ p_1 = (x_1, x_2)\ in\ I(F, A', e_1)$ である。ここで、A の大きさ、すなわち n の値によって場合わけする。

n $= 1$ の場合は A' $=$ nil である。仮定と補題 3.1 より、補題 3.5 の任意の環境を $\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F、nil、e_1)$ とすると、条件 4、条件 5、条件 6 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F, nil, e_1) : \tau_1$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = (x_1, x_2)\ in\ I(F, nil, e_1) : \tau_1$ であるので、$\Gamma \triangleright I(F, A, (x_1, x_2)) : \tau_1$ が成り立つ。

$n \geq 2$ の場合は A' $= (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil$ である。仮定と補題 3.1 より、補題 3.4 の任意の環境を $\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F、A'、e_1)$ とすると、条件 1、条件 2、条件 3 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F, A', e_1) : \tau_n$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = (x_1, x_2)\ in\ I(F, A', e_1) : \tau_n$ であるので、$\Gamma \triangleright I(F, A, (x_1, x_2)) : \tau_n$ が成り立つ。

・$\boldsymbol{e_0 = x[i]}$ の場合 (i $= 1$ or i $= 2$)

A $= (p_1, e_1) :: (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil \quad (n \geq 1)$、$\Gamma \triangleright x[i] : \tau_0$ かつ、$\Gamma\{p_1 : \tau_0\} \triangleright e_1 : \tau_1,\ \Gamma\{p_1 : \tau_0, p_2 : \tau_1\} \triangleright e_2 : \tau_2,\ \ldots\ , \Gamma\{p_1 : \tau_1,\ \ldots\ , p_n : \tau_{n-1}\} \triangleright e_n : \tau_n$ であり、$A = (p_1, e_1) :: A'$ とすると、アルゴリズムの定義より、$I(F, A, x[i]) = let\ p_1 = x[i]\ in\ I(F, A', e_1)$ である。ここで、A の大きさ、すなわち n の値によって場合わけする。

n $= 1$ の場合は A' $=$ nil である。仮定と補題 3.1 より、補題 3.5 の任意の環境を $\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F、nil、e_1)$ とすると、条件 4、条件 5、条件 6 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F, nil, e_1) : \tau_1$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = x[i]\ in\ I(F, nil, e_1) : \tau_1$ であるので、$\Gamma \triangleright I(F, A, x[i]) : \tau_1$ が成り立つ。

$n \geq 2$ の場合は A' $= (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil$ である。仮定と補題 3.1 より、補題 3.4 の任意の環境を $\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F、A'、e_1)$ とすると、条件 1、条件 2、条件 3 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F, A', e_1) : \tau_n$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = x[i]\ in\ I(F, A', e_1) : \tau_n$ であるので、$\Gamma \triangleright I(F, A, x[i]) : \tau_n$ が成り立つ。

・$\boldsymbol{e_0 = \lambda env.\lambda arg.e}$ の場合

ある τ_{env}、τ_a、τ_b があり、A $= (p_1, e_1) :: (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil \quad (n \geq 1)$、$\Gamma \triangleright \lambda env.\lambda arg.e : \tau_0$、$\tau_0 = Code(\tau_a \to \tau_b, \tau_{env})$ かつ、$\Gamma\{p_1 : \tau_0\} \triangleright e_1 : \tau_1, \quad \Gamma\{p_1 : \tau_0, p_2 : \tau_1\} \triangleright e_2 : \tau_2, \ \ldots, \Gamma\{p_1 : \tau_1, \ \ldots, p_n : \tau_{n-1}\} \triangleright e_n : \tau_n$ である。型システムの定義より、$\{env : \tau_{env}, arg : \tau_a\} \triangleright e : \tau_b$ である。さらに、仮定より、補題 3.5 の任意の環境を $\{env : \tau_{env}, arg : \tau_a\}$ として与え、$M = I(\emptyset, nil, e)$ とすると、条件 4、条件 5、条件 6 が成り立つ。 よって、帰納法の仮定より、$\{env : \tau_{env}, arg : \tau_a\} \triangleright I(\emptyset, nil, e) : \tau_b$ が成り立つ。型システムの定義より、$\Gamma \triangleright \lambda env.\lambda arg.I(\emptyset, nil, e) : Code(\tau_a \to \tau_b, \tau_{env})$ である。$e_a = \lambda env.\lambda arg.I(\emptyset, nil, e)$、$A = (p_1, e_1) :: A'$ とする。ここで、A の大きさ、すなわち n の値によって場合わけする。

n $= 1$ の場合は A' = nil である。$p_1 \notin dom(\Gamma)$ なので、補題 3.1 より、$\Gamma\{p_1 : \tau_0\} \triangleright p_1 : \tau_0$ かつ、$\Gamma\{p_1 : \tau_0\} \triangleright e_a : \tau_0$ であり、仮定と補題 3.1 より、補題 3.5 の任意の環境を $\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F\{p_1 : (e_a, \varepsilon)\}, nil, e_1)$ とすると、条件 4、条件 5、条件 6 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F\{p_1 : (e_a, \varepsilon)\}, nil, e_1) : \tau_1$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = e_a\ in\ I(F\{p_1 : (e_a, \varepsilon)\}, nil, e_1) : \tau_1$ が成り立つ。一方、アルゴリズムの定義より、$I(F, A, \lambda env.\lambda arg.e) = let\ p_1 = e_a\ in\ I(F\{p_1 : (e_a, \varepsilon)\}, nil, e_1)$ であるので、$\Gamma \triangleright I(F, A, \lambda env.\lambda arg.e) : \tau_1$ が成り立つ。

$n \geq 2$ の場合は A' $= (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil$ である。$p_1 \notin dom(\Gamma)$ なので、補題 3.1 より、$\Gamma\{p_1 : \tau_0\} \triangleright p_1 : \tau_0$ かつ、$\Gamma\{p_1 : \tau_0\} \triangleright e_a : \tau_0$ であり、仮定と補題 3.1 より、補題 3.4 の任意の環境を $\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F\{p_1 : (e_a, \varepsilon)\}, A', e_1)$ とすると、条件 1、条件 2、条件 3 が成り立つ。よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F\{p_1 : (e_a, \varepsilon)\}, A', e_1) : \tau_n$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = e_a\ in\ I(F\{p_1 : (e_a, \varepsilon)\}, A', e_1) : \tau_n$ が成り立つ。一方、アルゴリズムの定義より、$I(F, A, \lambda env.\lambda arg.e) = let\ p_1 = e_a\ in\ I(F\{p_1 : (e_a, \varepsilon)\}, A', e_1)$ であるので、$\Gamma \triangleright I(F, A, \lambda env.\lambda arg.e) : \tau_n$ が成り立つ。

・$\boldsymbol{e_0 = \lambda f.\lambda env.\lambda arg.e}$ の場合

ある τ_{env}、τ_a、τ_b があり、A $= (p_1, e_1) :: (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil \quad (n \geq 1)$、$\Gamma \triangleright \lambda f.\lambda env.\lambda arg.e : \tau_0$、$\tau_0 = Code(\tau_a \to \tau_b, \tau_{env})$ かつ、$\Gamma\{p_1 : \tau_0\} \triangleright e_1 : \tau_1, \quad \Gamma\{p_1 : \tau_0, p_2 : \tau_1\} \triangleright e_2 : \tau_2, \ \ldots, \Gamma\{p_1 : \tau_1, \ \ldots, p_n : \tau_{n-1}\} \triangleright e_n : \tau_n$ であり、$A = (p_1, e_1) :: A'$ とするとアルゴリズムの定義より、$I(F, A, \lambda f.\lambda env.\lambda arg.e) = let\ p_1 = \lambda f.\lambda env.\lambda arg.I(\emptyset, nil, e)\ in\ I(F, A', e_1)$ である。型システムの定義より、$\{f : \tau_0, env : \tau_{env}, arg : \tau_a\} \triangleright e : \tau_b$ である。さらに、仮定より、補題 3.5 の任意の環境を $\{f : \tau_0, env : \tau_{env}, arg : \tau_a\}$ として与え、$M = I(\emptyset, nil, e)$ とすると、条件 4、条件 5、条件 6 が成り立つ。 よって、帰納法の仮定より、$\{f : \tau_0, env : \tau_{env}, arg : \tau_a\} \triangleright I(\emptyset, nil, e) : \tau_b$ が成り立つ。型システムの定義より、$\Gamma \triangleright \lambda f.\lambda env.\lambda arg.I(\emptyset, nil, e) : \tau_0$ である。ここで、A の大きさ、すなわち n の値によって場合わけする。

n $= 1$ の場合は A' = nil である。仮定と補題 3.1 より、補題 3.5 の任意の環境を $\Gamma\{p_1 : \tau_0\}$

として与え、$M = I(F、nil、e_1)$とすると、条件 4、条件 5、条件 6 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F, nil, e_1) : \tau_1$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = \lambda f.\lambda env.\lambda arg.I(\emptyset, nil, e)\ in\ I(F, nil, e_1) : \tau_1$ であるので、$\Gamma \triangleright I(F, A, \lambda f.\lambda env.\lambda arg.e) : \tau_1$ が成り立つ。

$n \geq 2$ の場合は A' $= (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil$ である。仮定と補題 3.1 より、補題 3.4 の任意の環境を$\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F、A'、e_1)$ とすると、条件 1、条件 2、条件 3 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F, A', e_1) : \tau_n$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = \lambda f.\lambda env.\lambda arg.I(\emptyset, nil, e)\ in\ I(F, A', e_1) : \tau_n$ であるので、$\Gamma \triangleright I(F, A, \lambda f.\lambda env.\lambda arg.e) : \tau_n$ が成り立つ。

・$\boldsymbol{e_0 = Closure(x_1, x_2)}$ の場合

ある τ_{env}、τ_a、τ_b があり、A $= (p_1, e_1) :: (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil \quad (n \geq 1)$、$\Gamma \triangleright Closure(x_1, x_2) : \tau_0$、$\tau_0 = (\tau_a \to \tau_b) \times \tau_{env}$、$\Gamma\{p_1 : \tau_0\} \triangleright e_1 : \tau_1$, $\Gamma\{p_1 : \tau_0, p_2 : \tau_1\} \triangleright e_2 : \tau_2$, ... ,$\Gamma\{p_1 : \tau_1, \ \ldots\ , p_n : \tau_{n-1}\} \triangleright e_n : \tau_n$ である。型システムの定義より、$\Gamma \triangleright x_1 : Code(\tau_a \to \tau_b, \tau_{env})$、$\Gamma \triangleright x_2 : \tau_{env}$ である。

ここで、$dom(F)$ の状態と A の大きさ、すなわち n の値によって場合わけをする。

$x_1 \in dom(F)$ かつ、n $= 1$ の場合は、$F(x_1) = (e_a, e_{env})$、$A = (p_1, e_1) :: A'$ とすると、A' $=$ nil でありアルゴリズムの定義より、$I(F, A, Closure(x_1, x_2)) = let\ p_1 = Closure(x_1, x_2)\ in\ I(F\{p_1 : (e_a, x_2)\}, nil, e_1)$ である。仮定より、$\Gamma \triangleright e_a : Code(\tau_a \to \tau_b, \tau_{env})$ である。$p_1 \notin dom(\Gamma)$ なので、補題 3.1 より、$\Gamma\{p_1 : \tau_0\} \triangleright p_1 : \tau_0$、$\Gamma\{p_1 : \tau_0\} \triangleright e_a : Code(\tau_a \to \tau_b, \tau_{env})$ かつ、$\Gamma\{p_1 : \tau_0\} \triangleright x_2 : \tau_{env}$ が成り立つ。さらに、x_2 は変数であり、仮定と補題 3.1 より、補題 3.5 の任意の環境を $\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F\{p_1 : (e_a, x_2)\}$、$nil$、$e_1)$ とすると、条件 4、条件 5、条件 6 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F\{p_1 : (e_a, x_2)\}, nil, e_1) : \tau_1$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = Closure(x_1, x_2)\ in\ I(F\{p_1 : (e_a, x_2)\}, nil, e_1) : \tau_1$ であるので、$\Gamma \triangleright I(F, A, Closure(x_1, x_2)) : \tau_1$ が成り立つ。

$x_1 \in dom(F)$ かつ、$n \geq 2$ の場合は、$F(x_1) = (e_a, e_{env})$、$A = (p_1, e_1) :: A'$ とすると A' $= (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil$ であり、アルゴリズムの定義より、$I(F, A, Closure(x_1, x_2)) = let\ p_1 = Closure(x_1, x_2)\ in\ I(F\{p_1 : (e_a, x_2)\}, A', e_1)$ である。仮定より、$\Gamma \triangleright e_a : Code(\tau_a \to \tau_b, \tau_{env})$ である。$p_1 \notin (\Gamma)$ なので、補題 3.1 より、$\Gamma\{p_1 : \tau_0\} \triangleright p_1 : \tau_0$、$\Gamma\{p_1 : \tau_0\} \triangleright e_a : Code(\tau_a \to \tau_b, \tau_{env})$ かつ、$\Gamma\{p_1 : \tau_0\} \triangleright x_2 : \tau_{env}$ が成り立つ。さらに、x_2 は変数であり、仮定と補題 3.1 より、補題 3.4 の任意の環境を $\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F\{p_1 : (e_a, x_2)\}$、$A'$、$e_1)$ とすると、条件 1、条件 2、条件 3 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F\{p_1 : (e_a, x_2)\}, A', e_1) : \tau_n$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = Closure(x_1, x_2)\ in\ I(F\{p_1 : (e_a, x_2)\}, A', e_1) : \tau_n$ であるので、$\Gamma \triangleright I(F, A, Closure(x_1, x_2)) : \tau_n$ が成り立つ。

$x_1 \notin dom(F)$ かつ、n $= 1$ の場合は、$A = (p_1, e_1) :: A'$ とすると、A' $=$ nil であり、アルゴリズムの定義より、$I(F, A, Closure(x_1, x_2)) = let\ p_1 = Closure(x_1, x_2)\ in\ I(F, nil, e_1)$

である。仮定と補題 3.1 より、補題 3.5 の任意の環境を $\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F$、nil、$e_1)$ とすると、条件 4、条件 5、条件 6 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F, nil, e_1) : \tau_1$ である。さらに型システムの定義より、$\Gamma \triangleright let\ p_1 = Closure(x_1, x_2)\ in\ I(F, nil, e_1) : \tau_1$ であるので、$\Gamma \triangleright I(F, A, Closure(x_1, x_2)) : \tau_1$ が成り立つ。

$x_1 \notin dom(F)$ かつ、$n \geq 2$ の場合は、$A = (p_1, e_1) :: A'$ とすると、A' $= (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil$ であり、アルゴリズムの定義より、$I(F, A, Closure(x_1, x_2)) = let\ p_1 = Closure(x_1, x_2)\ in\ I(F, A', e_1)$ である。仮定と補題 3.1 より、補題 3.4 の任意の環境を $\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F$、A'、$e_1)$ とすると、条件 1、条件 2、条件 3 が成り立つ。よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F, A', e_1) : \tau_n$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = Closure(x_1, x_2)\ in\ I(F, A', e_1) : \tau_n$ であるので、$\Gamma \triangleright I(F, A, Closure(x_1, x_2)) : \tau_n$ が成り立つ。

・$e_0 = Apply(x_1, x_2)$ の場合

$isInline(Apply(x_1, x_2))$ の結果によって場合わけをする。

i) $isInline(Apply(x_1, x_2)) = true$ の場合

A $= (p_1, e_1) :: (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil \quad (n \geq 1)$、$\Gamma \triangleright Apply(x_1, x_2) : \tau_0$ かつ、$\Gamma\{p_1 : \tau_0\} \triangleright e_1 : \tau_1,\ \ \Gamma\{p_1 : \tau_0, p_2 : \tau_1\} \triangleright e_2 : \tau_2,\ \ldots\ , \Gamma\{p_1 : \tau_1,\ \ldots\ , p_n : \tau_{n-1}\} \triangleright e_n : \tau_n$ である。型システムの定義より、ある τ_{env}、τ_a があり、$\Gamma \triangleright x_1 : (\tau_a \rightarrow \tau_0) \times \tau_{env}$ かつ、$\Gamma \triangleright x_2 : \tau_a$ である。isInline の定義より、$x_1 \in dom(F)$ であり、$F(x_1) = (e_a, env)$ とする。仮定より、$\Gamma \triangleright e_a : Code(\tau_a \rightarrow \tau_0, \tau_{env})$ であり、かつ $e_a = \lambda env_1.\lambda arg_1.e$ の形をしている。$replaceVarName(\lambda env_1.\lambda arg_1.e)$ の返す式を、$\lambda env_2.\lambda arg_2.e'$ とすると、$replaceVarName$ の定義と補題 3.2 より、$\Gamma \triangleright \lambda env_2.\lambda arg_2.e' : Code(\tau_a \rightarrow \tau_0, \tau_{env})$ である。型システムの定義より、$\{env_2 : \tau_{env}, arg_2 : \tau_a\} \triangleright e' : \tau_0$ が成り立つ。さらに、$env_2 \notin dom(\Gamma)$ かつ、$arg_2 \notin dom(\Gamma)$ であるので、補題 3.1 より、$\Gamma\{env_2 : \tau_{env}, arg_2 : \tau_a\} \triangleright e' : \tau_0$、$\Gamma\{env_2 : \tau_{env}\} \triangleright x_2 : \tau_a$ についても成り立つ。よって、補題 3.3 より、$\Gamma\{env_2 : \tau_{env}\} \triangleright e'[x_2/arg_2] : \tau_0$ である。

ここで、env に関して場合分けをする。$env = \varepsilon$ とする。$env_2 \notin dom(\Gamma)$ なので、仮定と補題 3.1 より、補題 3.4 の任意の環境を $\Gamma\{env_2 : \tau_{env}\}$ として与え、$M = I(F$、A、$e'[x_2/arg_2])$ とすると、条件 1、条件 2、条件 3 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{env_2 : \tau_{env}\} \triangleright I(F, A, e'[x_2/arg_2]) : \tau_n$ である。型システムの定義より、$\Gamma \triangleright x_1[2] : \tau_{env}$ であるので、$\Gamma \triangleright let\ env_2 = x_1[2]\ in\ I(F, A, e'[x_2/arg_2]) : \tau_n$ が成り立つ。一方、アルゴリズムの定義より、$I(F, A, Apply(x_1, x_2)) = let\ env_2 = x_1[2]\ in\ I(F, A, e'[x_2/arg_2])$ であるので、$\Gamma \triangleright I(F, A, Apply(x_1, x_2)) : \tau_n$ が成り立つ。

次に、$env \neq \varepsilon$ と仮定する。仮定より、env は変数であり、かつ、$\Gamma \triangleright env : \tau_{env}$ であるので、補題 3.3 より、$\Gamma \triangleright e'[x_2/arg_2, env/env_2] : \tau_0$ が成り立ち、仮定より、補題 3.4 の任意の環境を Γ として与え、$M = I(F$、A、$e'[x_2/arg_2, env/env_2])$ とすると、条件 1、条件 2、条件 3 が成り立つ。 よって、帰納法の仮定より、$\Gamma \triangleright I(F, A, e'[x_2/arg_2, env/env_2]) : \tau_n$ となる。

一方、アルゴリズムの定義より、$I(F, A, Apply(x_1, x_2)) = I(F, A, e'[x_2/arg_2, env/env_2])$ であるので、$\Gamma \triangleright I(F, A, Apply(x_1, x_2)) : \tau_n$ が成り立つ。

ii) $isInline(Apply(x_1, x_2)) = false$ の場合

A $= (p_1, e_1) :: (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil \quad (n \geq 1)$、$\Gamma \triangleright Apply(x_1, x_2) : \tau_0$、$\Gamma\{p_1 : \tau_0\} \triangleright e_1 : \tau_1$, $\Gamma\{p_1 : \tau_0, p_2 : \tau_1\} \triangleright e_2 : \tau_2$, ... ,$\Gamma\{p_1 : \tau_1, \ \ldots \ , p_n : \tau_{n-1}\} \triangleright e_n : \tau_n$ である。型システムの定義より、ある τ_{env}、τ_a があり、$\Gamma \triangleright x_1 : Code(\tau_a \to \tau_0, \tau_{env})$、$\Gamma \triangleright x_2 : \tau_a$ である。

ここで $GetNextCode(x_1, x_2, F)$ の結果と A の大きさ、すなわち n の値によって場合わけをする。

$GetNextCode(x_1, x_2, F) = \varepsilon$ かつ、n $= 1$ の場合は、$A = (p_1, e_1) :: A'$ とすると、A' = nil であり、アルゴリズムの定義より、$I(F, A, Apply(x_1, x_2)) = let\ p_1 = Apply(x_1, x_2)\ in\ I(F, nil, e_1)$ である。仮定と補題 3.1 より、補題 3.5 の任意の環境を $\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F$、nil、$e_1)$ とすると、条件 4、条件 5、条件 6 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F, nil, e_1) : \tau_1$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = Apply(x_1, x_2)\ in\ I(F, nil, e_1) : \tau_1$ であるので、$\Gamma \triangleright I(F, A, Apply(x_1, x_2)) : \tau_1$ が成り立つ。

$GetNextCode(x_1, x_2, F) = \varepsilon$ かつ、$n \geq 2$ の場合は、$A = (p_1, e_1) :: A'$ とすると A' $= (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil$ であり、アルゴリズムの定義より、$I(F, A, Apply(x_1, x_2)) = let\ p_1 = Apply(x_1, x_2)\ in\ I(F, A', e_1)$ である。仮定と補題 3.1 より、補題 3.4 の任意の環境を $\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F$、A'、$e_1)$ とすると、条件 1、条件 2、条件 3 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F, A', e_1) : \tau_n$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = Apply(x_1, x_2)\ in\ I(F, A', e_1) : \tau_n$ であるので、$\Gamma \triangleright I(F, A, Apply(x_1, x_2)) : \tau_n$ が成り立つ。

$GetNextCode(x_1, x_2, F) \neq \varepsilon$ かつ、n $= 1$ の場合は、$GetNextCode(x_1, x_2, F) = e_a$、$A = (p_1, e_1) :: A'$ とすると A' = nil であり、アルゴリズムの定義より、$I(F, A, Apply(x_1, x_2)) = let\ p_1 = Apply(x_1, x_2)\ in\ I(F\{p_1 : (e_a, \varepsilon)\}, nil, e_1)$ である。GetNextCode の定義より、ある τ_a, τ_b, τ_c があり、$\tau_0 = (\tau_a \to \tau_b) \times \tau_c$ かつ、$\Gamma \triangleright e_a : Code(\tau_a \to \tau_b, \tau_c)$ となる。$p_1 \notin dom(\Gamma)$ なので、補題 3.1 より、$\Gamma\{p_1 : \tau_0\} \triangleright p_1 : \tau_0$ かつ、$\Gamma\{p_1 : \tau_0\} \triangleright e_a : Code(\tau_a \to \tau_b, \tau_c)$ が成り立つ。GetNextCode の定義より、$e_a = \lambda env'.\lambda arg'.e'$ の形を取る。さらに、仮定と補題 3.1 より、補題 3.5 の任意の環境を $\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F\{p_1 : (e_a, \varepsilon)\}$、$nil$、$e_1)$ とすると、条件 4、条件 5、条件 6 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F\{p_1 : (e_a, \varepsilon)\}, nil, e_1) : \tau_1$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = Apply(x_1, x_2)\ in\ I(F\{p_1 : (e_a, \varepsilon)\}, nil, e_1) : \tau_1$ であるので、$\Gamma \triangleright I(F, A, Apply(x_1, x_2)) : \tau_1$ が成り立つ。

$GetNextCode(x_1, x_2, F) \neq \varepsilon$ かつ、$n \geq 2$ の場合は、$GetNextCode(x_1, x_2, F) = e_a$、$A = (p_1, e_1) :: A'$ とすると、A' $= (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil$ であり、アルゴリズムの定義より、$I(F, A, Apply(x_1, x_2)) = let\ p_1 = Apply(x_1, x_2)\ in\ I(F\{p_1 : (e_a, \varepsilon)\}, A', e_1)$ である。GetNextCode の定義より、ある τ_a, τ_b, τ_c があり、$\tau_0 = (\tau_a \to \tau_b) \times \tau_c$ かつ、$\Gamma \triangleright e_a :$

$Code(\tau_a \rightarrow \tau_b, \tau_c)$ となる。$p_1 \notin dom(\Gamma)$ なので、補題 3.1 より、$\Gamma\{p_1 : \tau_0\} \triangleright p_1 : \tau_0$ かつ、$\Gamma\{p_1 : \tau_0\} \triangleright e_a : Code(\tau_a \rightarrow \tau_b, \tau_c)$ が成り立つ。GetNextCode の定義より、$e_a = \lambda env'.\lambda arg'.e'$ の形を取る。さらに、仮定と補題 3.1 より、補題 3.4 の任意の環境を $\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F\{p_1 : (e_a, \varepsilon)\}$、$A'$、$e_1)$ とすると、条件 1、条件 2、条件 3 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F\{p_1 : (e_a, \varepsilon)\}, A', e_1) : \tau_n$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = Apply(x_1, x_2)\ in\ I(F\{p_1 : (e_a, \varepsilon)\}, A', e_1) : \tau_n$ であるので、$\Gamma \triangleright I(F, A, Apply(x_1, x_2)) : \tau_n$ が成り立つ。

・$\boldsymbol{e_0 = if\ x\ then\ e_a\ else\ e_b}$ の場合

A $= (p_1, e_1) :: (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil \quad (n \geq 1)$、$\Gamma \triangleright if\ x\ then\ e_a\ else\ e_b : \tau_0$ かつ、$\Gamma\{p_1 : \tau_0\} \triangleright e_1 : \tau_1,\ \Gamma\{p_1 : \tau_0, p_2 : \tau_1\} \triangleright e_2 : \tau_2,\ \ldots,\Gamma\{p_1 : \tau_1,\ \ldots,p_n : \tau_{n-1}\} \triangleright e_n : \tau_n$ である。型システムの定義より、$\Gamma \triangleright x : bool$、$\Gamma \triangleright e_a : \tau_0$、$\Gamma \triangleright e_b : \tau_0$ である。仮定より、補題 3.5 の任意の環境を Γ として与え、$M = I(F$、nil、$e_a)$ とすると、条件 4、条件 5、条件 6 が成り立つ。よって、帰納法の仮定より、$\Gamma \triangleright I(F, nil, e_a) : \tau_0$ である。同様にして、$\Gamma \triangleright I(F, nil, e_b) : \tau_0$ についても成り立つ。型システムの定義より、$\Gamma \triangleright if\ x\ then\ I(F, nil, e_a)\ else\ I(F, nil, e_b) : \tau_0$ である。ここで A の大きさ、すなわち n の値によって場合わけする。

n $= 1$ の場合は、$A = (p_1, e_1) :: A'$ とすると A' = nil である。仮定と補題 3.1 より、補題 3.5 の任意の環境を $\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F$、nil、$e_1)$ とすると、条件 4、条件 5、条件 6 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F, nil, e_1) : \tau_1$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = if\ x\ then\ I(F, nil, e_a)\ else\ I(F, nil, e_b)\ in\ I(F, nil, e_1) : \tau_1$ である。一方、アルゴリズムの定義より、$I(F, A, if\ x\ then\ e_a\ else\ e_b) = let\ p_1 = if\ x\ then\ I(F, nil, e_a)\ else\ I(F, nil, e_b)\ in\ I(F, nil, e_1)$ なので、$\Gamma \triangleright I(F, A, if\ x\ then\ e_a\ else\ e_b) : \tau_1$ が成り立つ。

$n \geq 2$ の場合は A' $= (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil$ である。仮定と補題 3.1 より、補題 3.4 の任意の環境を $\Gamma\{p_1 : \tau_0\}$ として与え、$M = I(F$、A'、$e_1)$ とすると、条件 1、条件 2、条件 3 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{p_1 : \tau_0\} \triangleright I(F, A', e_1) : \tau_n$ である。型システムの定義より、$\Gamma \triangleright let\ p_1 = if\ x\ then\ I(F, nil, e_a)\ else\ I(F, nil, e_b)\ in\ I(F, A', e_1) : \tau_n$ である。一方、アルゴリズムの定義より、$I(F, A, if\ x\ then\ e_a\ else\ e_b) = let\ p_1 = if\ x\ then\ I(F, nil, e_a)\ else\ I(F, nil, e_b)\ in\ I(F, A', e_1)$ なので、$\Gamma \triangleright I(F, A, if\ x\ then\ e_a\ else\ e_b) : \tau_n$ が成り立つ。

・$\boldsymbol{e_0 = let\ x = e_a\ in\ e_b}$ の場合

A $= (p_1, e_1) :: (p_2, e_2) :: \cdots :: (p_n, e_n) :: nil \quad (n \geq 1)$、$\Gamma \triangleright let\ x = e_a\ in\ e_b : \tau_0$ かつ、$\Gamma\{p_1 : \tau_0\} \triangleright e_1 : \tau_1,\ \ \Gamma\{p_1 : \tau_0, p_2 : \tau_1\} \triangleright e_2 : \tau_2,\ \ldots,\Gamma\{p_1 : \tau_1,\ \ldots,p_n : \tau_{n-1}\} \triangleright e_n : \tau_n$ であり、アルゴリズムの定義より、$I(F, A, let\ x = e_a\ in\ e_b) = I(F, (x, e_b) :: A, e_a)$ である。型システムの定義より、ある τ_a があり、$\Gamma \triangleright e_a : \tau_a$ かつ、$\Gamma\{x : \tau_a\} \triangleright e_b : \tau_0$ である。$x \notin dom(\Gamma)$ なので、補題 3.1 より、$\Gamma\{x : \tau_a\} \triangleright e_b : \tau_0,\ \ \Gamma\{x : \tau_a, p_1 : \tau_0\} \triangleright e_1 : \tau_1,\ \ \Gamma\{x :$

$\tau_a, p_1 : \tau_0, p_2 : \tau_1\} \triangleright e_2 : \tau_2, \ \ldots \ , \Gamma\{x : \tau_a, p_1 : \tau_1, \ \ldots \ , p_n : \tau_{n-1}\} \triangleright e_n : \tau_n$ が成り立つので、補題 3.4 の任意の環境を Γ として与え、$M = I(F$、$(x, e_b) :: A$、$e_a)$ とすると、条件 1、条件 2、条件 3 が成り立つ。 よって、帰納法の仮定より、$\Gamma \triangleright I(F, (x, e_b) :: A, e_a) : \tau_n$ であるので、$\Gamma \triangleright I(F, A, let\ x = e_a\ in\ e_b) : \tau_n$ が成り立つ。

• 補題 3.5 の証明。
$M = I(F, A, e_0)$ とし、Γ を任意の環境と仮定する。e_0 の構造において、それぞれ場合わけする。

・$\boldsymbol{e_0 = c}$ の場合
$\Gamma \triangleright c : \tau_0$ である。アルゴリズムの定義より、$I(F, nil, c) = c$ であるので、$\Gamma \triangleright I(F, nil, c) : \tau_0$ が成り立つ。

・$\boldsymbol{e_0 = x}$ の場合
$\Gamma \triangleright x : \tau_0$ である。アルゴリズムの定義より、$I(F, nil, x) = x$ であるので、$\Gamma \triangleright I(F, nil, x) : \tau_0$ が成り立つ。

・$\boldsymbol{e_0 = (x_1, x_2)}$ の場合
$\Gamma \triangleright (x_1, x_2) : \tau_0$ である。アルゴリズムの定義より、$I(F, nil, (x_1, x_2)) = (x_1, x_2)$ であるので、$\Gamma \triangleright I(F, nil, (x_1, x_2)) : \tau_0$ が成り立つ。

・$\boldsymbol{e_0 = x[i]}$ の場合 (i = 1 or i = 2)
$\Gamma \triangleright x[i] : \tau_0$ である。アルゴリズムの定義より、$I(F, nil, x[i]) = x[i]$ であるので、$\Gamma \triangleright I(F, nil, x[i]) : \tau_0$ が成り立つ。

・$\boldsymbol{e_0 = \lambda env.\lambda arg.e}$ の場合
ある $\tau_a, \tau_b, \tau_{env}$ があり、$\Gamma \triangleright \lambda env.\lambda arg.e : \tau_0$ かつ、$\tau_0 = Code(\tau_a \to \tau_b, \tau_{env})$ である。型システムの定義より、$\{env : \tau_{env}, arg : \tau_a\} \triangleright e : \tau_b$ である。仮定より、補題 3.5 の任意の環境を $\{env : \tau_{env}, arg : \tau_a\}$ として与え、$M = I(\emptyset$、nil、$e)$ とすると、条件 4、条件 5、条件 6 が成り立つ。 よって、帰納法の仮定より、$\{env : \tau_{env}, arg : \tau_a\} \triangleright I(\emptyset, nil, e) : \tau_b$ である。型システムより、$\Gamma \triangleright \lambda env.\lambda arg.I(\emptyset, nil, e) : \tau_0$ であり、アルゴリズムの定義より、$I(F, nil, \lambda env.\lambda arg.e) = \lambda env.\lambda arg.I(\emptyset, nil, e)$ であるので、$\Gamma \triangleright I(F, nil, \lambda env.\lambda arg.e) : \tau_0$ が成り立つ。

・$\boldsymbol{e_0 = \lambda f.\lambda env.\lambda arg.e}$ の場合
ある $\tau_a, \tau_b, \tau_{env}$ があり、$\Gamma \triangleright \lambda f.\lambda env.\lambda arg.e : \tau_0$ かつ、$\tau_0 = Code(\tau_a \to \tau_b, \tau_{env})$ である。型システムの定義より、$\{f : \tau_0, env : \tau_{env}, arg : \tau_a\} \triangleright e : \tau_b$ である。仮定より、補題 3.5

の任意の環境を $\{f:\tau_0, env:\tau_{env}, arg:\tau_a\}$ として与え、$M = I(\emptyset、nil、e)$ とすると、条件 4、条件 5、条件 6 が成り立つ。 よって、帰納法の仮定より、$\{f:\tau_0, env:\tau_{env}, arg:\tau_a\} \triangleright I(\emptyset, nil, e):\tau_b$ である。型システムより、$\Gamma \triangleright \lambda f.\lambda env.\lambda arg.I(\emptyset, nil, e):\tau_0$ であり、アルゴリズムの定義より、$I(F, nil, \lambda f.\lambda env.\lambda arg.e) = \lambda f.\lambda env.\lambda arg.I(\emptyset, nil, e)$ であるので、$\Gamma \triangleright I(F, nil, \lambda f.\lambda env.\lambda arg.e):\tau_0$ が成り立つ。

・$\boldsymbol{e_0 = Closure(x_1, x_2)}$ の場合

$\Gamma \triangleright Closure(x_1, x_2):\tau_0$ である。アルゴリズムの定義より、$I(F, nil, Closure(x_1, x_2)) = Closure(x_1, x_2)$ であるので、$\Gamma \triangleright I(F, nil, Closure(x_1, x_2)):\tau_0$ が成り立つ。

・$\boldsymbol{e_0 = Apply(x_1, x_2)}$ の場合

$isInline(Apply(x_1, x_2))$ の結果によって、場合わけをする。

i) $isInline(Apply(x_1, x_2)) = true$ の場合

$\Gamma \triangleright Apply(x_1, x_2):\tau_0$ である。型システムの定義より、ある τ_{env}、τ_a があり、$\Gamma \triangleright x_1:(\tau_a \to \tau_0) \times \tau_{env}$ かつ、$\Gamma \triangleright x_2:\tau_a$ である。isInline の定義より、$x_1 \in dom(F)$ であり、$F(x_1) = (e_a, env)$ とする。仮定より、$\Gamma \triangleright e_a : Code(\tau_a \to \tau_0, \tau_{env})$ であり、かつ $e_a = \lambda env_1.\lambda arg_1.e$ の形をしている。$replaceVarName(\lambda env_1.\lambda arg_1.e)$ の返す式を、$\lambda env_2.\lambda arg_2.e'$ とすると、$replaceVarName$ の定義と補題 3.2 より、$\Gamma \triangleright \lambda env_2.\lambda arg_2.e' : Code(\tau_a \to \tau_0, \tau_{env})$ である。型システムの定義より、$\{env_2 : \tau_{env}, arg_2 : \tau_a\} \triangleright e' : \tau_0$ が成り立つ。さらに、$env_2 \notin dom(\Gamma)$ かつ、$arg_2 \notin dom(\Gamma)$ であるので、補題 3.1 より、$\Gamma\{env_2 : \tau_{env}, arg_2 : \tau_a\} \triangleright e':\tau_0$、$\Gamma\{env_2:\tau_{env}\} \triangleright x_2:\tau_a$ についても成り立つ。よって、補題 3.3 より、$\Gamma\{env_2:\tau_{env}\} \triangleright e'[x_2/arg_2]:\tau_0$ である。

ここで、env に関して、場合分けをする。$env = \varepsilon$ とする。$env_2 \notin dom(\Gamma)$ なので、仮定と補題 3.1 より補題 3.5 の任意の環境を $\Gamma\{env_2:\tau_{env}\}$ として与え、$M = I(F、nil、e'[x_2/arg_2])$ とすると、条件 4、条件 5、条件 6 が成り立つ。 よって、帰納法の仮定より、$\Gamma\{env_2 : \tau_{env}\} \triangleright I(F, nil, e'[x_2/arg_2]):\tau_0$ である。型システムの定義より、$\Gamma \triangleright x_1[2]:\tau_{env}$ であるので、$\Gamma \triangleright let\ env_2 = x_1[2]\ in\ I(F, nil, e'[x_2/arg_2]):\tau_0$ が成り立つ。一方、アルゴリズムの定義より、$I(F, A, Apply(x_1, x_2)) = let\ env_2 = x_1[2]\ in\ I(F, nil, e'[x_2/arg_2])$ なので、$\Gamma \triangleright I(F, nil, Apply(x_1, x_2)):\tau_0$ が成り立つ。

次に、$env \neq \varepsilon$ と仮定する。仮定より、env は変数であり、$\Gamma \triangleright env:\tau_{env}$ であるので、補題 3.3 より、$\Gamma \triangleright e'[x_2/arg_2, env/env_2]:\tau_0$ が成り立ち、仮定より、補題 3.5 の任意の環境を Γ として与え、$M = I(F、nil、e'[x_2/arg_2, env/env_2])$ とすると、条件 4、条件 5、条件 6 が成り立つ。 よって、帰納法の仮定より、$\Gamma \triangleright I(F, nil, e'[x_2/arg_2, env/env_2]):\tau_0$ となる。一方、アルゴリズムの定義より、$I(F, A, Apply(x_1, x_2)) = I(F, nil, e'[x_2/arg_2, env/env_2])$ なので、$\Gamma \triangleright I(F, nil, Apply(x_1, x_2)):\tau_0$ が成り立つ。

ii) $isInline(Apply(x_1, x_2)) = false$ の場合

$\Gamma \triangleright Apply(x_1, x_2):\tau_0$ である。アルゴリズムの定義より、$I(F, nil, Apply(x_1, x_2)) = Apply(x_1, x_2)$

であるので、$\Gamma \triangleright I(F, nil, Apply(x_1, x_2)) : \tau_0$ が成り立つ。

・$\boldsymbol{e_0 = if\ x\ then\ e_a\ else\ e_b}$ の場合

$\Gamma \triangleright if\ x\ then\ e_a\ else\ e_b : \tau_0$ である。型システムの定義より、$\Gamma \triangleright x : bool$、$\Gamma \triangleright e_a : \tau_0$、$\Gamma \triangleright e_b : \tau_0$ である。仮定より、補題 3.5 の任意の環境を Γ として与え、$M = I(F、nil、e_a)$ とすると、条件 4、条件 5、条件 6 が成り立つ。 よって、帰納法の仮定より、$\Gamma \triangleright I(F, nil, e_a) : \tau_0$ である。同様にして、$\Gamma \triangleright I(F, nil, e_b) : \tau_0$ についても成り立つ。型システムの定義より、$\Gamma \triangleright if\ x\ then\ I(F, nil, e_a)\ else\ I(F, nil, e_b) : \tau_0$ である。一方、アルゴリズムの定義より、$I(F, nil, if\ x\ then\ e_a\ else\ e_b) = if\ x\ then\ I(F, nil, e_a)\ else\ I(F, nil, e_b)$ であるので、$\Gamma \triangleright I(F, nil, if\ x\ then\ e_a\ else\ e_b) : \tau_0$ は成り立つ。

・$\boldsymbol{e_0 = let\ x = e_a\ in\ e_b}$ の場合

$\Gamma \triangleright let\ x = e_a\ in\ e_b : \tau_0$ である。型システムの定義より、ある τ があり、$\Gamma \triangleright e_a : \tau$ かつ、$\Gamma\{x : \tau\} \triangleright e_b : \tau_0$ である。さらに、仮定より、補題 3.4 の任意の環境を Γ として与え、$M = I(F、(x, e_b) :: nil、e_a)$ とすると、条件 1、条件 2、条件 3 が成り立つ。 よって、帰納法の仮定より、$\Gamma \triangleright I(F, (x, e_b) :: nil, e_a) : \tau_0$ である。一方、アルゴリズムの定義より、$I(F, nil, let\ x = e_a\ in\ e_b) = I(F, (x, e_b) :: nil, e_a)$ であるので、$\Gamma \triangleright I(F, nil, let\ x = e_a\ in\ e_b) : \tau_0$ が成り立つ。

以上によって、補題 3.4、 補題 3.5 が示せた。

定理 3.1 は、補題 3.4 より成り立つ。

第4章　実装と評価

これまでの理論に基づきインライン展開とその他いつくかの最適化を実装した。本章では、その実装の内容と評価結果について述べる。

4.1　コードサイズの抑制

図 3.3 のアルゴリズムで使用されていた isInline 関数は、極端にコードサイズが増大する関数についてはインライン展開を禁止する判断を行う。

実装するにあたっても、プログラム全体のコードサイズが極端に増加することを防ぐための判断をする必要がある。

ヒューリスティックな手法として、インライン展開する関数本体のコードサイズがある閾値以下の場合にのみインライン展開を実行する手法が多く取られている。この手法は、以下の 2 つの点から有効である。

- 関数の実行時間が小さいものほど、その実行時間に占めている関数呼び出しに伴うオーバヘッドの時間の割合が高くなる。インライン展開は関数オーバヘッドを軽減する効果があるため、実行時間が小さい関数ほどインライン展開による速度向上の効果は高いといえる。そして、一般には実行時間が小さな関数はコードサイズも小さくなることが多い。

- 小さな関数はインライン展開したとしても、プログラム全体に対するコードサイズの増加幅は小さい。

しかしながら、これだけでは必ずしもプログラム全体のコードサイズを期待した数値以下に抑制できるとは限らない。本論文では、プログラム全体のコードサイズをある閾値以下に抑えることを保障するために、入力されたプログラムのコードサイズと展開後のプログラムのコードサイズを比較した上でインライン展開をするかどうかを判断する。

プログラム全体のコードサイズを s 以下に抑えたいとして、インライン展開されていない初期のプログラム全体のコードサイズを psize とする。その基で入力プログラムに対してインライン展開アルゴリズムを実行していき、実際にインライン展開が可能な式 $Apply(x_1, x_2)$ が現れた際、以下の条件に従ってインライン展開を行うか否かを最終的に決定する。

1. $Apply(x_1, x_2)$ をインライン展開することにより増加するコードサイズと psize を加算した結果を tsize とする。

2. tsize $\leq$ s であれば、psize の値を tsize で更新して $Apply(x_1, x_2)$ をインライン展開する。s $<$ tsize であればインライン展開を行わない。

psize にはインライン展開されるごとにその増加分が加算される。psize の初期値は入力されたプログラムサイズなので、アルゴリズム実行中のある時点での psize は、その時点でのプログラム全体のコードサイズを表しており、仮に、それ以降一切インライン展開が行われないとすれば、最終的に出力されるプログラムサイズは psize に一致する。よって、この psize が閾値 s を超えない範囲でインライン展開を施せば、プログラム全体のサイズを一定値以下に抑えることができる。

反面、この手法だと 1 つの大きな関数をインライン展開してしまうことで簡単にプログラムサイズが増大してしまい、他の関数適用式に対するインライン展開が行われなくなる恐れがある。さらに、プログラムコードの上の部分だけインライン展開されて、下の部分は一切インライン展開されないといった事態が予想される。これは可能な限り防ぐ必要があり、インライン展開する関数も可能な限り小さいものを優先することが望ましい。そのため、プログラム全体のコードサイズだけではなく、プログラム全体のコードサイズと関数本体のコードサイズの両方から判断する。すなわち、以下の条件をどちらも満たした時にインライン展開をする。

- インライン展開する関数本体のコードサイズがある閾値よりも小さい
- インライン展開した結果のプログラム全体のコードサイズがある閾値よりも小さい

これにより、1 つのインライン展開が極端なプログラムサイズの増加を招くことはなくなるので、よりプログラム全体に渡ってインライン展開を行うことが可能となる。ただし、関数コードサイズの許容サイズを定めた閾値は、そのプログラムに応じて適切に設定する必要がある。例えば、閾値が大きすぎるとすぐにプログラム全体のコードサイズが極端に増加するために、インライン展開される場所に偏りがでてしまう可能性が高くなる。逆に、閾値が小さすぎるとほとんど関数適用式においてインライン展開がなされないこととなってしまう。

4.2 その他の最適化

インライン展開以外の最適化として、2.4 節で述べた以下の最適化を実行している。

- 定数の畳み込み
- 共通部分式の削除

- 無用命令の削除
- 条件文の置き換え
- フィールド要素取得命令の置き換え

インライン展開によってこれらの最適化が可能なコードが生成される可能性があるため、これらの最適化を施すことでインライン展開後のコードサイズを小さくすることが可能である。前節で述べたように、インライン展開を行うかどうかの条件にプログラム全体のコードサイズがある閾値以下であるという条件が存在する。仮に、この条件によってインライン展開できなかった場合は、その他の最適化を施すことによってプログラム全体のコードサイズが小さくなり、再びインライン展開が可能になることが予測される。

そこで、インライン展開アルゴリズムの実行中に、プログラム全体のコードサイズによってインライン展開できない関数適用式があった場合は、インライン展開アルゴリズムの実行終了後にその他の最適化を施し、その結果、コードサイズが十分に小さくなった場合は、再びインライン展開アルゴリズムを実行する。

図 4.1 は最適化部分の流れ図である。ただし、実際には何度も繰り返し最適化を行うとコンパイル時間が大幅に増加するので、たとえ繰り返しの条件を満たしてもある回数以上の繰り返しは禁止している。

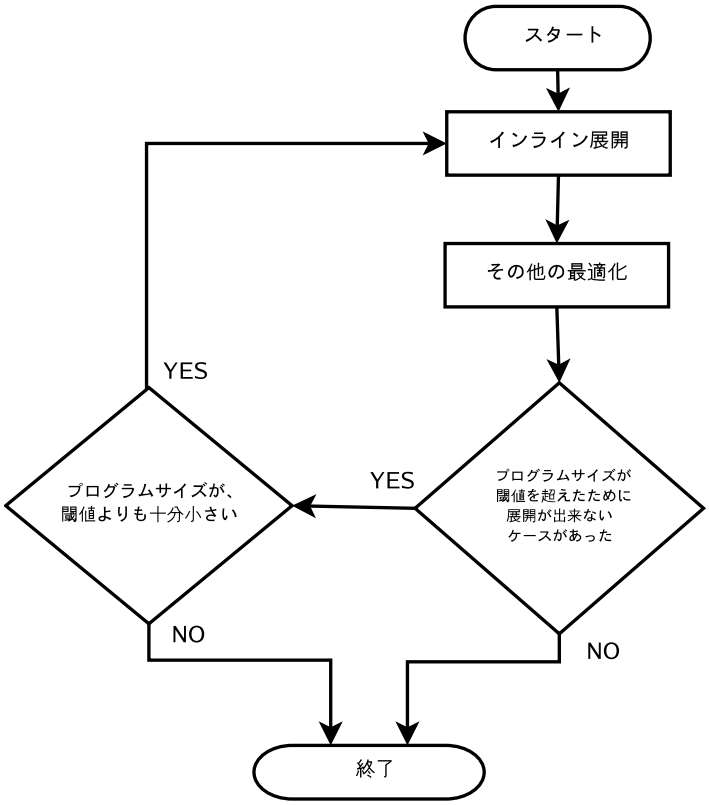

図 4.1: 最適化の流れ

4.3 実装環境

実装は、現在開発中である IML コンパイラ (仮称)[1] に行った。IML コンパライラは IML 言語を入力言語として取る。IML 言語は Standard ML を拡張した文法で定義されており、基本的に Standard ML の文法はほぼサポートしている。IML コンパイラ自身は Standard ML で記述されている。IML の主要な特徴として柔軟な型付け、多相型レコードの演算、他の言語との相互運用性などが上げられる。

IML コンパイラの処理の大まかな流れを、図 4.2 に示す。それぞれの処理の詳細な説明は省略する。

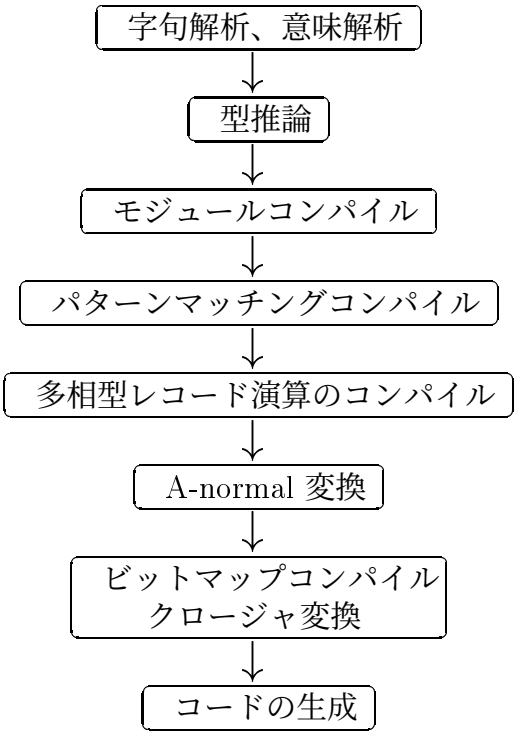

図 4.2: IML コンパイラの処理の流れ

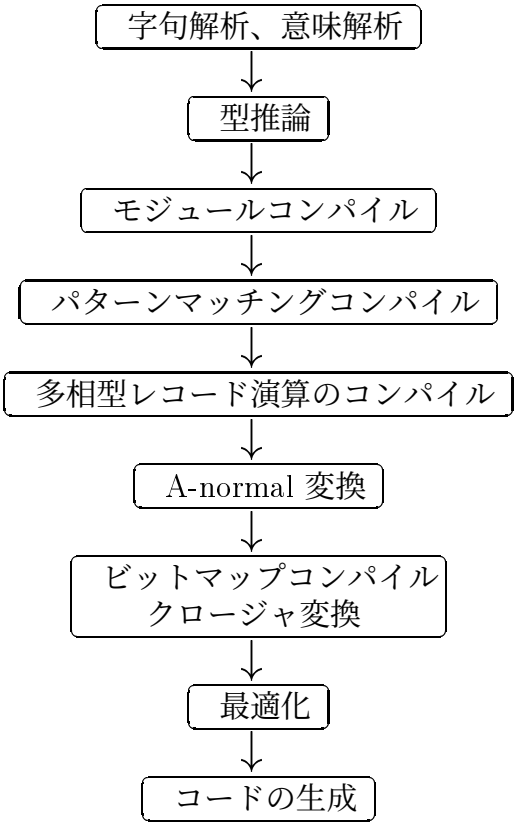

図 4.3: 最適化を追加した IML コンパイラの処理の流れ

多相型レコード演算が終わった段階での中間コードの表現は、ラムダ式を拡張した形 (拡張ラムダ式) となっている。さらに、それを A-normal 変換することで A-normal form の中間表現となる。一般的な最適化では、これら拡張ラムダ式、A-normal form 式を入力コードとすることが多い。

本研究ではそれらの表現ではなく、これまでの理論に基づきクロージャ変換後の中間表

[1] JAIST 大堀により提案され、現在開発中である。

現でインライン展開とそれに関連するいくつかの最適化を実装している。図 4.3 に最適化を追加後の流れを示す。

IML では A-normal form の表現をした中間言語に対してクロージャ変換処理を行うが、同時に、ビットマップ変換と呼ばれる変換も行っている。ここでは、ビットマップ変換の詳細について述べることは省略するが、その主な目的は相互運用を可能とするガーベッジコレクタの実現にある。また、クロージャ変換後の中間言語も A-nomal form の表現で構成されている。

現在のところ、IML コンパイラが生成するコードは IML 抽象コードである。そのためコンパイルされたコードを実行するためには IML 抽象機械が必要となる。IML 抽象コードを実行する IML 抽象機械については、Standard ML で記述されたものがあり、それを使用する。

4.4 評価

実装システムに対して行ったいくつかのベンチマークテストについて述べる。評価は Pentium4 2.8GHz の Linux Fedora Core3 の上で行った。表 4.1 に評価したプログラムを示す。ソースコード行は IML 言語で記述されたソースコードの行数を示す。さらにクロージャ変換後のサイズを matrix を 1.0 とした際の比率を示す。の行数を示す。サイズはクロージャ変換後のコードサイズの比率を matrix を基準に示してあり、各プログラムのクロージャ変換後のコードサイズを matrix のクロージャ変換後のコードサイズで割った値である。

プログラム	ソースコード行	サイズ
matrix	80	1.00
sort	83	0.72
mandel	85	0.43
gauss	240	3.92
huffman	314	8.22

表 4.1: ベンチマークプログラム

それぞれのプログラムの内容について簡単に述べる。matrix は、m × n と n × o の行列の積を計算するプログラムである。sort は、ランダムに並べられた n 個の数字をクイックソートにより昇順に並べ替えるプログラムである。mandel は、マンデルブロー集合を求めるプログラムである。gauss は、ガウスの消去法を用いて n 元連立一次方程式を解くプログラムである。huffman は、文字列 s に対して s に含まれるそれぞれの文字をハフマン符号化し、それを用いて文字列 s を圧縮する。さらに、圧縮した文字列を復号化して元

の文字列 s を取り出すプログラムである。

インライン展開には 2 つの閾値を設定する必要がある。1 つはインライン展開後の出力プログラムサイズの許容最大値である。これは入力コードサイズの 2 倍とした。よって、インライン展開後のコードサイズが入力コードサイズの 2 倍を超えることはない。もう 1 つの条件であるインライン展開可能な関数コードサイズの閾値については、入力コードサイズに対するある割合とし、本節では入力コードサイズの 0%、3%、5%、10%、20%、50%、100%を閾値としたそれぞれの結果を示す。ただし、0%はの場合は実質インライン展開がされることがないため、入力コードとインライン展開後の出力コードは同一である。また、ここでいうサイズとはすべてクロージャ変換後のコードサイズを指す。

各プログラムの入力コードに対してインライン展開のみを実行したコードのコードサイズと実行時間の比率を表 4.2、表 4.3 に示す。各値は、インライン展開後の値をインライン展開前の値で割った値である。入力コードは閾値 0%の出力コードと同一であるため、閾値 0%の値は必ず 1.0 となる。

基本的には閾値が大きいほどコードサイズが大きく増加している。しかし、閾値が 100%の場合はどのプログラムも 50%の場合と違いはなく、閾値 50%の時も、閾値 20%と違いが出たのは mandel のみであった。すなわち、閾値を極端に大きくしても、関数がインライン展開された回数に変化はないこと示している。ただし、huffman については閾値 3%以上はすべて入力コードの 2 倍を超えるため、途中のコードからインライン展開がなされておらず、すべて 2.0 に極めて近い値で終わっている。

実行時間については、どの閾値においても平均で 4%の速度向上が見られた。最も速度向上が大きかったのは mandel であり、閾値 50%の時では 11%速度が向上した。逆に、huffman では速度が低下している。この要因として以下のことがあげられる。huffman は入力コードサイズ自体が他のプログラムよりも非常に大きいのであるが、さらにインライン展開によって、そのコードサイズが 2 倍近くになっている。それにより抽象機械が使用するメモリが非常に大きくなり、メモリ確保のため Standard ML の GC が頻繁に起動することで速度が低下していると考えられる。mandel の場合においても閾値 50%の時には 2 倍近くコードサイズが増加しているが、入力コードのサイズ自体が huffman の 1/20 程度であるため、コードサイズが 2 倍になっても GC に大きく影響を与えるほどではない。この影響は、抽象機械を C/C++などの GC のない言語に変更することで少なくなるものと思われる。

次に、インライン展開後にその他のすべての最適化を実行したコードに対するコードサイズと実行時間の比率を表 4.4、表 4.5 に示す。この比率はすべて入力コードである表 4.2、表 4.3 の閾値 0%を基準に示している。

閾値 0%の場合は、インライン展開事態はされないため入力コードに対して直接その他の最適化を実行したことと同じである。コードサイズは閾値 3%以上ではすべてにおいて大きく減っており、平均で 40%前後コードが減少している。閾値 0%ではほとんど変化がないので、入力コードそのものには最適化ができるコードは少なく、最適化がなされたコードのほとんどはインライン展開によって生成されているといえる。huffman は、閾値

プログラム	各閾値でのコードサイズの比率 (出力コードサイズ / 入力コードサイズ)						
	0%	3%	5%	10%	20%	50%	100%
matrix	1.00	1.37	1.42	1.56	1.56	1.56	1.56
sort	1.00	1.21	1.28	1.54	1.72	1.72	1.72
mandel	1.00	1.11	1.11	1.52	1.59	1.83	1.83
gauss	1.00	1.57	1.79	1.84	1.84	1.84	1.84
huffman	1.00	1.97	1.98	1.95	1.95	1.95	1.95
平均	1.00	1.45	1.52	1.69	1.73	1.78	1.78

表 4.2: 各閾値でのインライン展開後のコードサイズの比率

プログラム	各閾値での実行時間の比率 (展開後の実行時間 / 展開前の実行時間)						
	0%	3%	5%	10%	20%	50%	100%
matrix	1.00	0.94	0.94	0.94	0.94	0.94	0.94
sort	1.00	0.95	0.95	0.95	0.95	0.95	0.95
mandel	1.00	0.95	0.95	0.92	0.91	0.89	0.89
gauss	1.00	0.96	0.96	0.96	0.96	0.96	0.96
huffman	1.00	1.02	1.02	1.05	1.05	1.05	1.05
平均	1.00	0.96	0.96	0.96	0.96	0.96	0.96

表 4.3: 各閾値でのインライン展開後の実行時間の比率

3%以上では 1 回目のインライン展開でコードサイズが 2 倍になるため、その後に、その他の最適化が実行されコードサイズが小さくなることで、再び 2 回目のインライン展開とその他の最適化が行われた。その結果、コードサイズは 60%以上減少した。全体に閾値 20%以上はコードサイズに大きな違いはなく、閾値 100%は 50%とまったく同一であった。

実行時間についても、閾値 3%以上はすべてにおいて 40%以上と大幅に速度が向上している。特に huffman では 70%以上速度が向上している。ただし、これはコードサイズが増加したときと逆で、コードサイズが減少したことにより、GC の起動が減ったことも、より一層の速度向上につながった要因と考えられる。ここにおいても閾値 100%の場合は、閾値 50%の場合と変化はなかった。すべての評価プログラムにおいて大幅な速度向上を示しており、クロージャ変換後においても十分なインライン展開の効果があるといえる。

インライン展開後に他の最適化を実行すると実行速度の向上と共に、コードサイズについても大幅に減少する結果となった。しかしながら、閾値が大きいほど全体に渡ってインライン展開がされない危険性は確実に高くなるため、不必要に大きな閾値を設定するべきではない。評価した結果から判断すると閾値 50%、100%は、閾値の大きさに比較して大

プログラム	各閾値でのコードサイズの比率 (出力コードサイズ / 入力コードサイズ)						
	0%	3%	5%	10%	20%	50%	100%
matrix	1.00	0.64	0.61	0.59	0.59	0.59	0.59
sort	1.00	0.76	0.73	0.66	0.64	0.64	0.64
mandel	1.00	0.83	0.83	0.63	0.63	0.60	0.60
gauss	1.00	0.63	0.60	0.60	0.60	0.60	0.60
huffman	0.92	0.34	0.36	0.36	0.36	0.36	0.36
平均	0.98	0.64	0.63	0.57	0.56	0.56	0.56

表 4.4: 各閾値でのインライン展開と他の最適化を行ったコードサイズの比率

プログラム	各閾値での実行時間の比率 (最適化後の実行時間 / 最適化前の実行時間)						
	0%	3%	5%	10%	20%	50%	100%
matrix	1.00	0.75	0.74	0.74	0.74	0.74	0.74
sort	1.00	0.75	0.75	0.69	0.63	0.63	0.63
mandel	1.00	0.73	0.73	0.49	0.50	0.40	0.40
gauss	1.00	0.42	0.38	0.36	0.36	0.36	0.36
huffman	0.99	0.26	0.26	0.26	0.26	0.26	0.26
平均	1.00	0.58	0.57	0.51	0.50	0.48	0.48

表 4.5: 各閾値でのインライン展開と他の最適化を行ったコードの実行時間の比率

きな速度向上は見られない。逆に、閾値が 3%ではより大きな閾値に比べ平均で 5%以上実行速度が遅い結果となっている。これらより判断すると閾値は入力コードの 10%前後が適切といえる。ただし、この最終結論にはさらに多くのプログラムによる検証が必要である。

4.5 クロージャより環境を取得するインライン展開の効率

提案したアルゴリズムではクロージャから環境を取得することで、自由変数の参照を正しく解決できるとことが最大の利点である。しかしながら、4.4 節で行った評価プログラムでは、実際にクロージャから環境を動的に取得しなければインライン展開できないケースは、huffman と gauss においてそれぞれ 3 回のみであった。この回数が、それらのプログラム中で実行された全てのインライン展開の回数に占める割合は 3%以下である。今回の実装では、インライン展開をプログラムの上から順に試みていき、さらに、実際に行う

か否かの判断もコードサイズからのみ判断している。一般に関数コードのサイズは、最後の引数を与えることにより実行されるコードが最も大きくなることが多い。すなわち、`fn x => ... fn y => ... fn z => ...` のコードであれば `fn z => ...` が最もコードサイズが大きくなる傾向がある。そのため、`fn z => ...` への関数適用がインライン展開されるなら、その前の `fn x => ...`、`fn y => ...` への適用式もインライン展開されていることが多くなり、それにより `fn x => ...`、`fn y=> ...` の中で定義されている変数が参照可能となり、結果として、クロージャから動的に環境を取得する必要性はなくなる。以上の理由などにより、今回の実装では、提案したアルゴリズムの利点が効率に大きな影響を与えることはなかった。しかしながら、これはインライン展開をするか否かの最終決定アルゴリズムをコードサイズからのみではなく、他の要因も考慮するアルゴリズムに変更することで、`fn z =>...` の展開を優先的に行いたい事は十分考えられ、そのような場合には提案したアルゴリズムの利点がより一層有効に生かされると考えられる。

そこで、クロージャから環境を動的に取得するインライン展開が速度向上につながるかを確認するために、クロージャから環境を取得する必要のない展開であっても、すべてクロージャから環境を取得して、展開後のコードに存在する環境へのアクセスは、その結果を参照するようなインライン展開を行うアルゴリズムの実装をした。すなわち、この実装では環境の値が判明できるとしても、すべて判明できないものとして扱う。

評価プログラムとして、4.4 節の評価プログラムで比較的インライン展開後の速度向上が明らかであった matrix と mandel を使用する。インライン展開の条件はすべて先ほどと同一である。閾値を入力コードの 10%とした場合において、実行速度が matrix は 4%、mandel は 6%向上した。先ほどのケースでは実行速度が matrix は 6%、mandel は 8%向上しており、それと比較すると若干効率が下がることが確認された。これは、クロージャから環境を取得する余分な命令が挿入されるためであると考えられる。しかし、若干効率は下がるものの、このようなインライン展開においても速度向上を確かに示す結果となった。

第5章　結論と今後の課題

5.1　結論

本論文では、自由変数に影響されないインライン展開を実現するために、クロージャ変換後の中間言語を対象にアルゴリズムの構築を行い、そのアルゴリズムが型を保存することを示した。さらに、構築したアルゴリズム、並びに関連する最適化を実装し、いくつかのプログラムを対象に評価を行った。その結果、クロージャ変換後のコードサイズを平均で40%前後減少させることができ、実行速度についても40%以上向上する結果となり、クロージャ変換後においてもインライン展開が有効に働くことが確認できた。

ただし実際に、クロージャから環境を取得することにより自由変数の参照を解決し、インライン展開を行う変換は、数的にはごくわずかであったため、実行速度の向上に大きく影響することはなかった。しかしながら、クロージャから環境を取得するインライン展開については、個別に評価を行った結果、環境を直接参照可能な場合に比べて若干効率は落ちるものの、確かに速度が向上し、有効であることを確認した。この結果より、クロージャから動的に環境を取得するインライン展開により、従来不可能であった関数適用式をインライン展開の候補とすることは有効であると考えられる。

5.2　今後の課題

今後の課題としては、インライン展開する式のより効率的な決定方法が挙げられる。

本論文で構築したアルゴリズムでは、インライン展開を行うか否かの判断を関数のコードサイズとインライン展開後のプログラムサイズから判断している。インライン展開後のプログラムサイズから判断するため、必ずある範囲内のサイズにプログラムは抑えることが可能である。

しかしながら、この方法ではアルゴリズムの実行中にプログラムサイズが許容値を超えた場合には、それ以降は一切インライン展開が行われくなる可能性を否定することはできない。許容値の範囲内では全ての関数適用式のインライン展開を行うことが不可能な場合は、順にインライン展開を行うのではなく、インライン展開することによる効果が高いものを優先して行うべきである。さらに、インライン展開をすることによる効果は、展開するコードサイズだけに依存するわけではない。例えば、プログラム実行時に数度しか呼ばれない関数よりも、数多く呼ばれる関数を展開するほうが、インライン展開を行う効果は一般に高い。すなわち、より効率的なインライン展開を実現するためには、インライン展

開による効果を推測し、その結果に基づいてインライン展開を行うことが望まれる。

謝辞

本研究を進めるにあたり、ご指導賜りました大堀淳教授に深く感謝いたします。また，本研究を行うにあたり、日頃よりお世話になった計算機言語学講座の皆様に深く感謝致します。

参考文献

[1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[2] Andrew W. Appel and Trevor Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, Vol. 7, No. 5, pp. 515–540, 1997.

[3] J. Michael Ashley. The effectiveness of flow analysis for inlining. In *International Conference on Functional Programming*, pp. 99–111, 1997.

[4] Arne de Bruijn. Implementation of inlining in stratego.

[5] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993*, Vol. 28(6), pp. 237–247. ACM Press, New York, 1993.

[6] Suresh Jagannathan and Andrew K. Wright. Flow-directed inlining. In *SIGPLAN Conference on Programming Language Design and Implementation*, pp. 193–205, 1996.

[7] S. Jones and S. Marlow. Secrets of the glasgow haskell compiler inliner, 1999.

[8] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. Typed closure conversion. In *Symposium on Principles of Programming Languages*, pp. 271–283, 1996.

[9] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pp. 181–192, 1996.

[10] 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.