# 第 4 回コンパイラ実験の実装概説

組型の引数を展開する機能をモジュール Tuple_args として実装した．基本的な実装の方針をまず説明する．組型の引数を展開する為には,

1. 引数が組型であることを認識,
2. 関数の定義を組を展開した引数を取るように変更,
3. 関数引数の変更に合わせて関数適用の際に渡す引数のフォーマットも変更する,

という主に三つの機能を実現する必要がある．

また, このモジュールは `Syntax.t`, `KNormal.t` 同士の変換として実装するという二つの可能性が考えられると思うが, どちらにもメリットデメリットがある．~~いずれにしても, ある関数の引数が組であるという情報が最適化の間に増えることはないのでこの最適化は iter の度に適用するものではなく, 一度だけ適用すれば良いものであると考えられる.~~ 組型の情報は増えないが, インライン展開などとの組み合わせを考えると KNormal.t の方が良いだろうという結論に落ち着いた．

(* メリットデメリットを後で追記 *)

まず三つの機能についてそれぞれのロジックの説明をしたのち, 実装の説明に移りたいと思う．### ロジックの説明一つ目の組型引数の認識には, Typing により引数に付加された型情報を利用した．例えば関数 `f x` の引数 `x` が Typing の結果として, `val x : ((Int, Int), ((Int, Int), (Int, Int)))` という型を持つとわかったとしよう．このとき, `x = ((px11,px12),((px211,px212),(px221,px222)))` のように組を展開することができる．

二つ目の段階では, この `[px11;px12;px211;px212;px221;px222]` を引数に取れるように関数本体 `e1` を書き換える．しかし, `e1` 全体のコードを走査しながら逐一対応する変数に書きかえるのは大変である．そこでいったん無駄な lettuple(* 構文としては `let (x1, ..., xn) = e1 in e2` のようなもの*) を増やし, 他の最適化 (Beta によるエイリアス削除, Constfold による lettuple を要素ごとの let に展開) を利用して, iter を回せば上手く結果として対応する変数に置き換わるであろう一時的に膨れ上がったコードを出力することにした．

具体的には以下のコードを使って説明する．

```
let rec f x =
  let (x1,x2) = x in
    let (x11,x12) = x1 in
    let (x21,x22) = x2 in
    let (x211,x212) = x21 in
    let (x221,x222) = x22 in
    (x11 + x12 + x211 + x212 + x221 + x222)
```

```
in print_int (f ((2,3),((5,6),(7,8))))
```

重要な点として, Typing により組型を持つと判明するからには, その変数が lettuple の右辺として表れているはず (?) であるということがある. (関数の引数であるからには, 右辺が組の宣言により定義された変数である可能性はないので)

上記のコードで `f` の本体の一番最初に `let x = ((px11,px12),((px211,px212),(px221,px222))) in` というコードを挿入する. (KNormal.t の文法では let の右辺にネストした組を書けないので, 多少の工夫が必要. この点については後述) すると後の処理によって, 元の関数 `f` の本体部分に次のような変換がなされるはずである. 1. Constfold では, lettuple の右辺が既知の組である場合は要素ごとの let に展開する. これは例えば, `let (a1,a2) = x in e2` という式を見たとき, `x = (x1,x2)` がわかっていれば, `let a1 = x1 in let a2 = x2 in e2` に書きかえられるということである. 2. Beta により, `e2` から `e2[a1 -> x1, a2 -> x2]` に置き換わる 3. Elim により, 不要になった `a1,a2` の宣言が削除される

以上の Tuple_args の第 2 段階, 関数定義の書き換えとその最適化を上記の例で見てみよう. (* この過程を手動で全部書くのはだいぶ面倒ではある *) 以下は, 正確な出力ではなく, わかりやすさのために適宜計算途中の結果を入れる変数を省いている. 変換後の関数名は, `P_`という接頭辞が付いている (product(直積) の気持ち).

```
(*After Tuple_args*)
let rec f  px11 px12 px211 px212 px221 px222=
    let x = ((px11,px12),((px211,px212),(px221,px222))) in
        let (x1,x2) = x in
        let (x11,x12) = x1 in
        let (x21,x22) = x2 in
        let (x211,x212) = x21 in
        let (x221,x222) = x22 in
        (x11 + x12 + x211 + x212 + x221 + x222)

(*After KNormal*)
    (*面倒なのでいったん省略*)
(*After iter*)
let rec P_f  px11 px12 px211 px212 px221 px222=
    (px11 + px12 + px211 + px212 + px221 + px222)
```

この関数定義の書き換えにともなって, 関数適用のフォーマットも変更しなければ不整合が生じてしまう. 例えば, 上記の例でもともとは `f ((2,3),((5,6),(7,8)))` という関数適用だったのに対し, 引数を展開した後の `f'` では `f' 2 3 5 6 7 8` としなければならない. この変換に関して 1 点難しい問題があり, それは高階関数で `f` 自体を引数としてとっている場合である. 上記のような関数適用の書き換えは, すでにプログラム内に書かれている式 (陽な関数適用) については行うことができるが, 高階関数内で `f` に適用が起こる場合 (陰な関数適用) は書き換えができない.

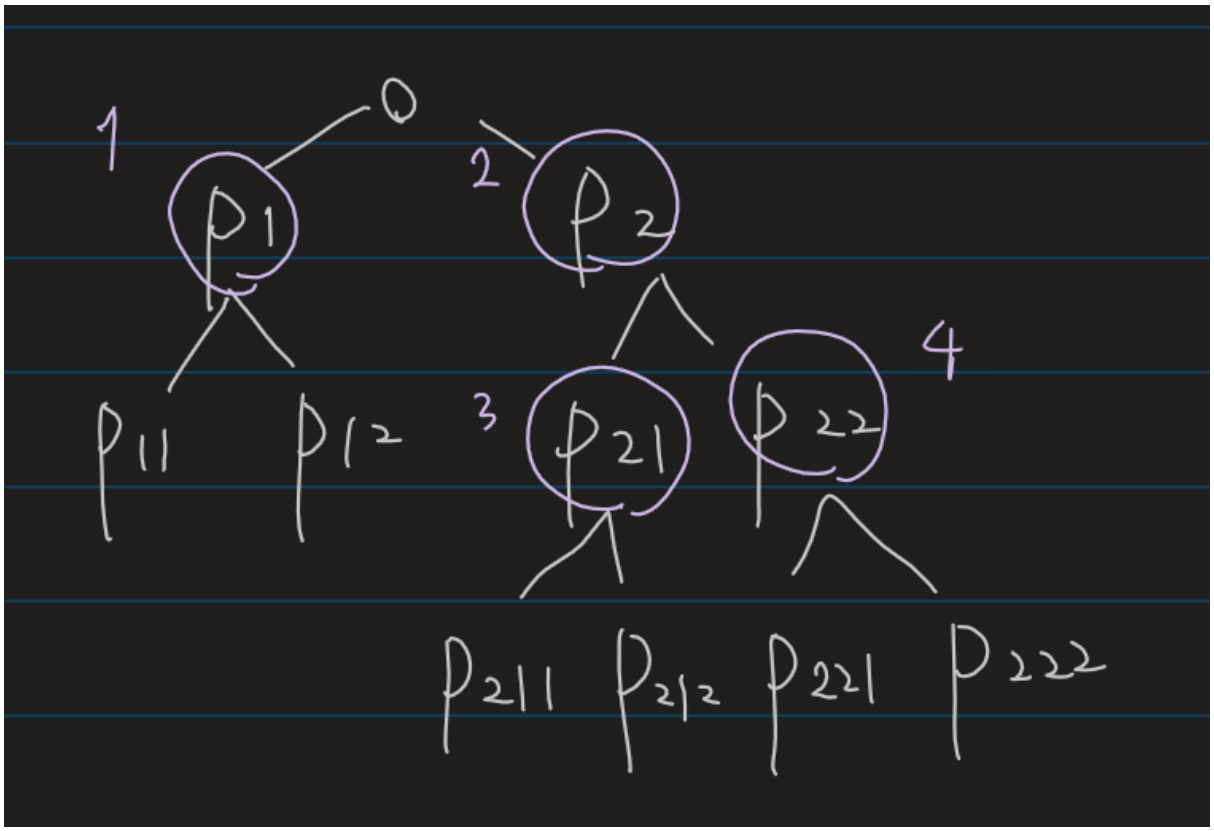
素朴な解決策として, 変換後の `f'` を陽な関数適用のために使う一方で, 陰な関数適用のために `f` の定義自体も残しておくという方法が挙げられる. これは関数定義が二倍に増えてあまりよくない方法に見えるかもしれないが, `f` が高階関数の引数にならない (陰な関数適用がない) 場合は後の最適化で不要定義削除により `f` の定義は消えるので現実的にはそんなに悪い方法ではないだろう. これはクロージャ変換で行っている処理にも似ていて, 必要な可能性が否定できないので念のために定義しておくが, 基本的には結果として必要ないので削除できるというわけである.

さて, 第 3 段階の本題に入り, 関数適用のフォーマット変換をどのように行うのかを考えよう. まずそもそも変換が必要な場合は, 組の展開がされている関数についてのみである. これを判定するために, 組展開された関数を入れた環境を用意し, ついでに引数の変換に使う関数などを返してもらうことにする.

このフォーマット変換についても上記と同じ例を用いて, どのようなことをしたいのかをまず説明しようと思う. 目標としては, 元のフォーマット `[x] = ((px11,px12),((px211,px212),(px221,px222)))` に対して `[((2,3),((5,6),(7,8)))]` を渡していたところを, 新しいフォーマット `[px11;px12;px211;px212;px221;px222]` に準じた `[2;3;5;6;7;8]` という形に変換したい. これも第 2 段階のときと同じで, いったん冗長なコードを追加して, 最適化によって効率的な変換になることを想定している. 上記の例については, 以下のような変換をすることを考えた.　元の引数は 1 行目の lettuple の右辺に, 変換後の引数は最終行に位置している.

```
let (p1,p2) = ((2,3),((5,6),(7,8))) in （*変換前の引数*）
    let (p11,p12) = p1 in
    let (p21,p22) = p2 in
    let (p211,p212) = p21 in
    let (p221,p222) = p22 in
    (p11 p12 p211 p212 p221 p222) （*変換後の引数*）
```

lettuple の左辺にネストした組を書けないので, 以下のようにタプルを木構造状に書いたとき, 変換後の引数に使われるのは leaf の変数だけであるが, 展開のためには中間ノードも置いてやる必要がある. また, lettuple で leaf まで変数を展開するために, 変数同士の依存関係を考えて順番を調整しなければならない. これは root から行きがけ順に出てきたノードに書かれた変数を lettuple の左辺とする式を作ればいい (はず).




これらの let を順に挿入するコードは, KNormal の `insert_let` を参考に, `let x = e1 in e2` の `e2` 部分を受け取り, 適切に let を追加した後の `let y = ... in let x = ... in e2` にような式を返す関数を fold していくことにより作る実装となっている. 詳細については次のセクションで述べる.

また, 関数適用ごとに同じ変数名で中間ノードと適用に使う変数 `[px11;px12;px211;px212;px221;px222]` を変えなければ, 変数名が衝突する恐れがあると思ったので, 関数適用毎にカウンタを更新して変数名の末尾につけるようにした.

KNormal.t で関数適用は `KNormal.App(f,args)` として表され, `f,args` はそれぞれ変数名に限定されている. しかし, `f` が直接的に定義済みの関数名ではなく, 例えば `id f` のような関数適用の結果として帰ってくる関数を置いた中間変数かもしれない. ひとまずは, `f` が組型変数の展開がされた関数名そのものなのかをチェックすることになっている. エイリアス解析すらしていないので, ちょっと関数名が隠されているともう最適化ができない貧弱な仕様になっている.

当初に組型変数の情報は増えないので iter の中で繰り返し tuple_args をやる意味はないと思っていたが, inline 展開によって置き換えられる関数適用が増える可能性があるのでそのような事情も踏まえて KNormal.t 同士の変換をするモジュールに書き換えた.

### 実装の説明

基本的には新規に作成したファイル tuple_args.ml 内に変更は限られている. 多少の例外として, type.ml に tuple_depth, tuple_size_with_depth という関数が増えている. これらは第 1 段階で組型変数の構造を分析

するのに用いるが, 基本的に型を見てやる作業なので Type モジュールの方に入れた.

以降はモジュール Tuple_args で `Tuple_args.f` が呼ばれた後の処理の流れに沿って説明する. mincaml の (悪しき) 慣習に従って, 外に見せる関数は `f`, そこから呼ばれて実質的な処理をする関数は `g` という名前になっている.

g では KNormal.t に関するパターンマッチをしているが, 本質的に重要なのは LetRec(関数定義), App(関数適用) の場合のみである. まずは簡単な App のケースの処理を説明しよう. 以下で括弧内に書かれている変数名は tuple_args.ml における対応を表している.

**■KNormal.App(f,args) の場合**　ロジックの説明のときに, 組型引数の展開がされた関数を登録した環境 (g の引数 env に対応) が, 引数の変換をする関数などを返すと言っていた. 実際には, make_newargs と change_format という関数が返っている.

- make_newargs : カウンタの値 (count) -> 変換後の引数のリスト
- change_format : 変換前の変数リスト (args), count, 変換後の関数適用 (e2') -> 適切に引数の変換のための let を挿入した後の e2'

処理内容としては, make_newargs を用いて変換後の引数 (newargs) を作り, `e2' = App("P_"^f,newargs)` として change_format の適用結果を返す.

**■KNormal.LetRec({name;args;body = e1},e2) の場合**

**組型引数展開のパラメータ**　全ての組型引数を展開する必要はない, ということがスライドに書いてあったので展開の程度を調整できるように 2 つのパラメータ max_size, max_depth を用意した. これらは順に (入れ子を展開した) 全要素数, 入れ子の深さに対応している.

レジスタ数を超える引数の数を持つ関数はアセンブリ生成でエラーを吐くので, max_size はレジスタの数をデフォルトとしている.

**組型引数展開の処理**　まず, 元の関数の引数 (args) を関数 generate に渡して, 組型引数を展開しようとする. generate は補助関数の generate_g と合わせて, 上記のパラメータに合わせた展開を探した後, 見つかったのならば引数の変換などに必要な道具を返す.

**generate 関数**　generate では, min(最大深さ, 深さ上限) からデクリメントしつつ, max_size に収まる展開の深さを探す. なお, max_size と比較するのは, タプルを木構造と見たとき, maxdepth より下のノードを縮退させたときの全要素数 (:= 深さ付き全要素数) としてここでは考えている. 以上を踏まえた, 深さを考慮した展開というのが少しわかりにくいかもしれないので, 具体例を交えて補足を行う.

再び `x = ((px11,px12),((px211,px212),(px221,px222)))` の例を使って考えてみよう. 構造は上の木構造の図を見た方がわかりやすいだろう.　この組は全要素数 6, 最大深さ 3 である. 深さ上限が 2 なら

ばこの組は, `x = ((px11,px12),(px21,px22))`, `px21 = (px211,px212)`,`px22 = (px221,px222)` という展開になる. 全要素数上限が 2 なら, 深さ 3,2 だと深さ付き全要素数がそれぞれ 6,4 となって制約を満たさない. 深さ 1 のときは深さ付き全要素数が 2 となって `x = (px1,px2)`, `px1 = (px11,px12)`, `px2 = ((px211,px212),(px221,px222))` と展開される.

generate_g で展開できる深さが見つかったら, それを maxdepth として gen_tuple_args を呼ぶ. これは全ての引数について組型引数展開に必要な道具を生成する関数となっており, 一つ一つの引数について同じ目的を果たす gen_tuple_arg と合わせてこのモジュールにおける処理の心臓部を担う.

**型エイリアス** 以降の関数を考える上で, 多少わかりやすさを上げるために次のような型エイリアスが tuple_args で定義されている.

```
type variable = Id.t * Type.t
type vlist = variable list

type expand_tuple = (vlist * variable) list
type construct_tuple = (variable * vlist) list
```

**gen_tuple_arg 関数** `f x = e1` という関数で, `x = ((px11,px12),(px21,px22))`, `px21 = (px211,px212)`,`px22 = (px221,px222)` という展開になる場合を考えよう. ここでの組展開の変数名は適当においたものになっているが, gen_tuple_arg で求める変数名は機械的に生成するので決まったルールに従ったものになる. タプルの要素の変数名は, 展開前の変数名 x に対して, 直接の子を `x = (x_1, ..., x_n)` とするルールを再帰的に適用する. もし, その変数が leaf ならば”P_“を接頭辞として付ける. さて, 関数の入出力についても説明しよう. 関数の実装としては変数 x に対して, その型 t のパターンマッチをして, tuple 型を root か中間ノード, それ以外を leaf として再帰的な処理をするようになっている.

**入力**

1. suffix = leaf の変数名, つまり変換後の関数の引数 (newargs) にの見つけられる接頭辞
2. (x,t) = 変数名 x とその型 t
3. maxdepth = generate で求めた展開すべき深さ

**出力**

次の要素からなる 4 つ組になっている.

1. 変数リスト : vlist = 変換後の関数の引数 (newargs) に, このノード以下で新たに追加すべき変数リスト
2. このノードを表す変数 : variable
3. 第 2 段階の組構成に必要な帰りがけ (postorder) リスト: construct_tuple
4. 第 3 段階の組展開に必要な行きがけ (preorder) リスト : expand_tuple

ここでの 3 は組構成で `let x = ((px11,px12),((px211,px212),(px221,px222))) in` に対して, KNormal.t では let の右辺にネストしたタプルが書けないので多少の工夫が必要だと述べた部分に対応している. この工夫とは, 第 3 段階で展開に中間ノードをおく必要があったのと同様に, 中間ノードをおけばいい. 第 3 段階では組を展開するのに下の変数が上の変数に依存するので, 行きがけ順で lettuple を挿入した. これに対し, 組を構成する方では上の変数が下の変数に依存するので, 帰りがけ順で右辺が tuple の let 式を挿入すればいい.

この挿入すべき宣言は `let x = e1 in e2` の `e2` 部分を受け取り, 新しい式を返す関数として環境に登録するが, gen_tuple_arg ではわかりやすさのために宣言すべき左辺と右辺の組のリストを返し, それを gen_tuple_args を畳み込んで返すというようにした.