自然なデータ表現を持つ多相型言語の LLVM IR へのコンパイル方式†

上野 雄大

本論文では，SML# コンパイラのマシンコード生成方式の現状および新たな構想について述べる．SML# は，Standard ML を包摂しながら，マシンネイティブなデータ表現，正確なごみ集め（GC），および分割コンパイルを採用する関数型言語である．これらの特徴を実現する上での実装上の主要な課題は，多相関数を多相的な性質を持つマシンコードにコンパイルすることにある．型主導コンパイルに基づくビットマップコンパイルの理論はこの課題を解決する基礎を与えているが，実装技術についてはスタックフレームを巧みに制御することを示唆するに留まっており，その実装はコード生成器とコンパイラフロントエンドの強い結合を招き，LLVM など最先端のコンパイラ基盤の利用を難しくする．本論文では，SML# コンパイラ開発におけるこの課題へのこれまでの取り組みを報告した後，これまでのアプローチが抱える問題点を挙げ，それらを克服する新たなコンパイル方式について述べる．

1 はじめに

SML#[3] は，Standard ML を包摂しながら，C 言語との高度な連携機構，SQL データベースとの統合，マルチスレッドサポート，分割コンパイルとシステムリンカによるリンク，正確でオブジェクトを移動しない並行ごみ集め（GC）などの先進的機能を持つ関数型言語である．これら SML# の機能の大半は，マシンネイティブな（自然な）データ表現を基盤として構築されている．自然なデータ表現とは，例えば整数やポインタの一部のビットをタグ等の特別な用途に使用したり，処理系の都合でデータサイズを統一したりしない，C 言語と同様の，CPU が直接サポートするデータ表現のことである．自然なデータ表現の採用によって，SML# でない言語で書かれた様々なライブラリを SML# から直接利用し，システム共通のオブジェクトファイル形式を通じてそれらライブラリを SML# プログラムに直接リンクすることが可能となった．このシステムとの直接連携を核として，データベースサーバーやマルチスレッドライブラリなどと連携を取り，それらとのシームレスな統合を実現するために表層言語を拡張しランタイムライブラリを整備したものが，今日の SML# のおおよその構造である．

多相型関数型言語において自然なデータ表現を実現する鍵となった技術は，型注釈から実行コードを生成する型主導コンパイル [2] に基づくビットマップコンパイル [1] である．自然なデータ表現を採用する多相型言語では，プログラムが取り扱うデータのレイアウトは必ずしも静的に決まらない．例えば，関数

`fun` α `dup (x :` α`) = (x, x)`

が作る組のレイアウトは，束縛型変数 α のインスタンスに依存して動的に決まる．また，正確な GC のためには，`x` の値が GC でトレースすべきポインタであるかどうかを管理することも必要である．型主導コンパイルは，このようなインスタンスに依存する処理を，型抽象を関数抽象に，型適用を関数適用にコンパイルすることでコード化する．ビットマップコンパイルの結果，`dup` は以下のラムダ式にコンパイルされる．

$$\mathsf{dup} = \Lambda\alpha.\, \lambda S : \mathsf{size}(\alpha).\, \lambda T : \mathsf{tag}(\alpha).\, \lambda x : \alpha.$$
$$\langle\!\langle (x, x) \text{ をアロケートするコード} \rangle\!\rangle$$

ここで，S および T はインスタンスのサイズ（値の

Katsuhiro Ueno, 東北大学, Tohoku University.

† 本稿は，第 22 回プログラミングおよびプログラミング言語ワークショップ（PPL 2020）で発表された論文「自然なデータ表現を持つ多相型言語の LLVM IR へのコンパイル方式」の要約である．

バイト数）およびポインタかどうかを表すタグをそれぞれ受け取るための追加の引数である．関数本体は，S から組のバイト数と第 2 要素のオフセットを計算し，その大きさのメモリを確保し，そのオフセットに x の値をストアし，タグをメタ情報（ビットマップ）としてそのメモリに付与するコードである．

ビットマップコンパイルは，自然なデータ表現を実現する基礎をラムダ計算上の変換として与えていると言える．その一方で，ビットマップコンパイル後のラムダ式の操作的意味（自然意味論）をマシンコードで実装する方法については，十分な検討がなされていない．ラムダ計算の自然意味論とマシンコードの最大の相違点は，ラムダ計算の自然意味論では任意の型の値を式の評価結果や関数の引数および返り値にすることができるのに対し，マシンコードでは途中の計算結果や関数の引数，返り値の型が静的に決まっていなければならないことである．例えば，前述の dup 関数の引数 x は，ラムダ計算上では α 型を持ち，意味論上では，32 ビット整数，64 ビット浮動小数点数，ヒープにアロケートされたデータへのポインタなど，様々な大きさや表現を持つ任意の型の値に束縛され得る．コンパイラはこの意味をマシンコードで表現しなければならない．しかし，このように多様な値を取りつつ正確な GC もサポートする「α 型の変数」に相当するレジスタはマシンコードには存在しない．自然な表現を持ちながら「α 型の変数」に相当する多相性を有するストレージをマシンコードで効率の良い形で表現できなければ，たとえビットマップコンパイルを装備していたとしても，自然なデータ表現を装備した実用的なコンパイラを実現することはできない．

本論文では，LLVM IR を対象のマシンコードとし，SML# コンパイラの開発過程において試みた，あるいはこれから試みようとしている，「α 型の変数」を実現する方式を 3 つ報告する．第一の方式は，ビットマップコンパイルと共に提案されたフレームビットマップ方式 [1] である．第二の方式は，多相関数に渡すデータをヒープ上にアロケート（boxing）することで多相性を表現し，boxing されているかどうかの違いを動的な条件判定で吸収する呼び出し規約方式である．第三の方式は，性質の異なる複数のレジスタの組を「α 型の変数」に相当する汎用のストレージとみなし，型主導コンパイルで受け渡される情報を用いてレジスタの組からレジスタを選択するレジスタ選択方式である．このうち第三の方式は SML# コンパイラに実装されていない方式であり，本論文の執筆時点においてはあくまで構想の段階にある．その他の方式は過去に SML# コンパイラに実装され評価された方式である．

2 フレームビットマップ方式

文献 [1] では，この問題に対し，全ての一時変数をスタックフレームに割り当てる仮想機械に対して限定的な解決策を与えている．その概要は以下の通りである．まず，一時変数の型を，ポインタを表す boxed，ポインタでないデータを総称する unboxed，およびそのどちらかが型変数 $t_1, \ldots, t_n$ のインスタンスに応じて実行時に現れることを表す型 $t_1, \ldots, t_n$ に限定する．次に，一時変数の生存区間解析を型ごとに行い，コードの実行に必要な最小のフレームサイズを計算し，フレームレイアウトを決定する．最後に，組に対するメタ情報と同様に，決定されたフレームレイアウトに対してスタックフレームのビットマップを計算する．スタックフレーム中の $t_1, \ldots, t_n$ 型を持つ領域に対しては，動的に受け渡されたタグ $T_1, \ldots, T_n$ からビットマップを計算するコードを出力する．

この方式を実現する上での最大の困難はフレームビットマップの管理である．フレームビットマップは動的に計算されるメタ情報であり，その計算にはコンパイラフロントエンドが行う型主導コンパイルと，コード生成器が生成するフレームレイアウトの連携が不可欠である．このため，コンパイルフェーズ間の独立性が失われる．型主導コンパイルとコード生成器は共に高度に複雑な処理であり，それらを相互依存させさらに複雑化することは現実的に困難である．また，フレームビットマップのように実行時に計算されるメタ情報をスタックフレームに含める仕組みを，LLVM などのコンパイラ基盤は一般に提供していない．SML# コンパイラが自前のネイティブコード生成器を有していた 1.2.0 版までは SML# コンパイラはこの方式で実装されていたが，コンパイラコードの管理コストと生成コードの品質の観点から LLVM の利

用を検討したとき，フレームビットマップ方式は実現不可能と判断された．

3 呼び出し規約コンパイル方式

LLVM IR で多相関数を実装する方式を検討するにあたり最初に採用した想定は，最も汎用なデータストレージはメモリであり，従ってコードの多相性はポインタを介して表現するのが自然であろう，ということである．例えば，多相的な dup 関数の場合，

`fun` α `dup (x :` α`) = (x, x)`

この x は α のインスタンスに依らず常にポインタとして実装する．この関数を整数 n に適用した場合，dup を呼ぶ前に n はヒープにアロケート（boxing）され，そのポインタが dup に渡されるとする．多相性はポインタを介して表現されるから，コードに現れる全ての一時変数の型は静的に定まる．以下，本節では，引数や返り値が boxing されていなければならないかどうかを関数ごとに定める規約を呼び出し規約と呼ぶ．

多相関数をこのように実装する一方で，単相関数は可能な限り boxing しない方式で実装したい．そのためには，呼び出し元から呼び出し先の関数が多相関数であるかどうかを何らかの方法で判断する必要がある．しかし，多相型言語の場合，多相関数に対する型適用が存在するため，この判断を静的に行うことは一般に難しい．

この問題をビットマップコンパイルとは独立に解決するため，SML# コンパイラでは，呼び出し先の関数が boxing を必要とする関数かどうかを実行時に判定し呼び出し規約を変換する方針を取った．この判定のために，関数クロージャに呼び出し規約を表すフラグを追加する．各呼び出し元において，このフラグを参照して呼び出し規約を判定し boxing や unboxing を行う条件分岐コードを生成する．このコード生成を行うコンパイルフェーズを呼び出し規約コンパイルと呼ぶ．

呼び出し規約が注釈として付加されたラムダ計算を考える．

$$\begin{aligned} M &::= c \mid x \mid \lambda^{kk} x{:}\tau.M \mid M^{kk}\ M \mid \cdots \\ k &::= \mathbf{b} \mid \mathbf{u} \mid \mathbf{r} \end{aligned}$$

k は呼び出し規約を表す注釈であり，$\mathbf{b}$ はヒープにアロケートされている（boxing されている）ことを，$\mathbf{u}$ および $\mathbf{r}$ は boxing されていない 32 ビット整数および 64 ビット浮動小数点数をそれぞれ表す．$\lambda^{k_1 k_2} x{:}\tau.M$ は，引数の受け方に関する規約を k_1，返り値の渡し方に関する規約を k_2 とする関数である．$M_1^{k_1 k_2} M_2$ は，引数の渡し方に関する規約を k_1，返り値の受け方に関する規約を k_2 とする関数呼び出し式である．

呼び出し規約注釈のないラムダ計算から呼び出し規約注釈付きラムダ計算への変換は，各部分式の型と整合するように規約を選択することで行われる．例えば，整数型には規約 $\mathbf{b}$（boxing された整数）または $\mathbf{u}$（boxing されていない整数）が整合し，浮動小数点型には $\mathbf{b}$ または $\mathbf{r}$ が整合する．関数型や組型などヒープにアロケートされるデータ型には $\mathbf{b}$ のみが整合する．

呼び出し規約コンパイルによって，$\lambda^{k_1 k_2} x{:}\tau.M$ は注釈付きのクロージャを生成するコードに，$M_1^{k_1 k_2} M_2$ は自身の注釈とクロージャの注釈を比較して分岐するコードにそれぞれコンパイルされる．この分岐には，全ての注釈の組み合わせを網羅する数の分岐先が必要である．すなわち，もし m 個の組み込み型と n 引数の関数が存在するならば，最大で $(m+1)^{n+1}$ 通りの分岐が必要となる．この指数オーダーの分岐コード生成を避けるため，SML# コンパイラでは，関数の型から定まる最も自然（specific）な規約と最も一般的（generic）な規約 $\mathbf{bb}$ にのみ着目した．関数クロージャは，最も自然な規約で呼び出される関数本体と，最も一般的な規約で呼び出されるラッパーの 2 つのコードを持つ．関数呼び出し側では，クロージャの持つ最も自然な規約が呼び出し側の規約と一致している場合，最も自然な規約を用いて関数を呼び出す．そうでなければ，最も一般的な規約を用いる．

本方式は SML# 2.0.0 版から採用されており，SML# コンパイラ自身を含む多くの動作実績を持つ．それら動作実績を通じて，動的な呼び出し規約の判定や boxing によって極端なオーバーヘッドは発生しないことは確認されている．その一方で，本方式の弱点もいくつか見つかっている．第一の弱点は，多相性を取り除く最適化を進めるたびに分岐の数が増えることである．例えば以下の関数を考える．

```
fun apply f x = f x
```

関数適用式 `f x` は，その最も自然な規約が最も一般的な規約であることから，最も一般的な規約で `f` を呼び出す分岐のないコードにコンパイルされる．一方，プログラムの最適化を進めた結果，`f` には int → int 型の関数しか来ないことがわかり，以下のように型注釈を付けて多相性を除去したとしよう．

```
fun apply (f : int -> int) x = f x
```

すると，関数適用式 `f x` の規約は **uu** となるため，規約を動的に判定する条件分岐コードが現れる．もし `f` に単相関数しか来ない場合，この分岐コードは無駄になり，一般的な規約で呼び出す側のコードはデッドコードとなる．ゆえに，多相性を除去することが分岐やデッドコードが増える要因となる．

第二の弱点は，多相性を伴わないはずの関数呼び出しでも boxing が発生し得ることである．例として以下の関数を考える．

```
fun fact 0 k = k 1
  | fact n k = fact (n-1) (fn z => k (z*n))
```

この関数の型は $\forall\alpha.\,\mathsf{int} \to (\mathsf{int} \to \alpha) \to \alpha$ である．継続 `k` の型は $\mathsf{int} \to \alpha$ であるから，`k 1` および `k (z*n)` は最も一般的な規約で `k` を呼ぶことになる．従って，継続が呼ばれるたびに整数の boxing と unboxing が行われる．計算の途中で作られる全ての継続の呼び出しで boxing を行うのは効率が悪く，またプログラマの直感にも反する．

4 レジスタ選択方式

次期 SML# コンパイラに向けて，上述のような弱点を持つ呼び出し規約方式に代わる，「α 型の変数」を構成する別のアプローチを検討している．本節ではその構想を述べる．

新たな実装方式を展開するにあたっての洞察は，全てのインスタンスを実装するために必要なレジスタ全てを組にしたものを汎用のストレージとして利用することである．多相関数の実装に必要な汎用のストレージは LLVM IR には存在しないが，その一方で，多相関数の個々のインスタンスを静的に決めたならば，多相関数のインスタンスの決定によって値を保持するレジスタの種類が限定され，その結果 LLVM IR での実装が可能となる．そこで，全ての種類のレジスタを使うように多相関数を実装しておき，そのうちのひとつを実行時に選択できるならば，多相関数の個々のインスタンスと同じ振る舞いをする単一のコードが実現できるはずである．この選択は，ビットマップコンパイルで受け渡される情報を利用することで実現できる．汎用のストレージをレジスタで実現できるならば，呼び出し規約方式のように boxing を行う必要はなく，従って boxing に由来する一切のオーバーヘッドは原理的に生じない．

この方針を素直に実装した場合，関数呼び出し時は，関数の型に依らず，レジスタの組を用いて引数と返り値を受け渡すことになる．例えば，整数に 1 を足す関数

```
fun inc x = x + 1
```

は整数しか操作しないが，以下のようなコードにコンパイルすることになる．

```
define fastcc {i32, double, i8*}
       @inc(i32 %a, double %b, i8* %c) {
  %d = add i32 %a, 1
  %r = insertvalue {i32, double, i8*} undef,
                   i32 %d, 0
  ret {i32, double, i8*} %r
}
```

LLVM IR コードは複雑になるものの，引数および返り値が caller-save レジスタを介して受け渡しされるならば，`undef` に関する最適化により，不要なレジスタ操作を含まない最適なマシンコードがこの LLVM IR コードから生成されるはずである．

引数および返り値の受け渡し方の更なる最適化として，レジスタの組を大きなレジスタ（例えば `i64` 型のレジスタ）1 つに集約する方法が考えられる．引数を渡す際は，引数の値を持つレジスタから引数の受け渡しのために一時的に用いるレジスタに値をコピーする．返り値を受け取る際もレジスタ間コピーを行う．この方法の利点は，LLVM による最適化を頼らなくても最も少ない数のレジスタだけを使用した関数呼び出しコードを実現できることである．例えば前述の `inc` 関数は以下のコードにコンパイルされる．

```
define fastcc i64 @inc(i64 %x) {
  %a = bitcast i64 %x to i32
  %b = add i32 %a, 1
  %c = bitcast i32 %b to i64
  ret i64 %c
}
```

この方式には，呼び出し規約コンパイル方式にあったような弱点は無い．分岐コードはレジスタ選択コードでしか生成されず，またそれらのコードが生成されるのは多相関数内に限られる．そのため，多相性を除去する最適化によりコードが増えることはない．また，引数と返り値は多相関数においてもレジスタを介して受け渡され，関数呼び出し時に boxing が行われることはなく，メモリアクセスと GC の起動回数の両方が減ることが期待できる．

この方式には 2 つの欠点が考えられる．ひとつは，LLVM IR レベルでの引数および返り値の型が全て `i64` に固定されるため，浮動小数点数を扱う関数の呼び出しのたびに浮動小数点数レジスタと整数レジスタの間でのムーブが発生することである．そのため，浮動小数点レジスタを用いて浮動小数点引数を受け渡す C などに比べて，浮動小数点演算を多用するプログラムが遅くなる可能性がある．ただし，少なくとも呼び出し規約コンパイル方式よりは実行効率の良いコードが生成されるはずである．

もうひとつの欠点は，表層言語における関数の型とは無関係に関数の実装型が決定されることである．このことは，将来の言語拡張による互換性の喪失を引き起こす恐れがある．例えば，SIMD 命令で用いるベクトルレジスタの値など，64 ビットよりも大きな値を SML# がサポートした場合，`i64` に集約する最適化が行えなくなる．本論文の執筆時点において x86_64 アーキテクチャで最大のレジスタは ZMM レジスタであり，その大きさは 512 ビット（64 バイト）である．ZMM レジスタをサポートする場合，わずか 32 ビットの整数を受け渡す時も ZMM レジスタを使うようにするか，あるいは `i64` レジスタと ZMM レジスタの組を引数の受け渡しに用いるかのどちらかを選択することになると思われる．どちらの場合でも LLVM IR での関数の型を変える変更となり，しかもこの変更は ZMM レジスタを用いない関数に対しても適用されなければならない．これは，オブジェクトファイルの互換性を失う変更になると考えられる．

5 まとめ

型主導コンパイル後のラムダ式を LLVM IR コードにコンパイルする方式として，フレームビットマップ方式，呼び出し規約コンパイル方式，およびレジスタ選択方式の 3 つを紹介し，SML# コンパイラに実装した上での評価を述べた．マシンコードへのコンパイルする上での課題は，多彩な値を取りつつ正確な GC をサポートする「α 型の変数」に相当するレジスタがマシンコードに存在しないことである．フレームビットマップ方式は全ての一時変数をスタックフレームに割り付けることでこの課題を解決したが，型主導コンパイルフェーズとコード生成器の分離を難しくするため，LLVM IR コード生成での採用を諦めた．呼び出し規約コンパイル方式は boxing 操作を行うことでこの課題を解決し，LLVM IR へのコード生成を達成したものの，プログラマの直感に反するコードの増加や boxing のコストがかかり，最適とは言い難い．レジスタ選択方式は構想段階にあり未実装であるが，前方式の欠点を克服する最適な方式であると期待している．

謝辞 本研究の一部は JSPS 科研費 19K11893 の助成を受けたものです．

参 考 文 献

[1] Nguyen, H.-D. and Ohori, A.: Compiling ML Polymorphism with Explicit Layout Bitmap, *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06, New York, NY, USA, Association for Computing Machinery, 2006, pp. 237–248.

[2] Ohori, A.: A Polymorphic Record Calculus and Its Compilation, *ACM Trans. Program. Lang. Syst.*, Vol. 17, No. 6(1995), pp. 844–895.

[3] : SML# Project, `http://www.riec.tohoku.ac.jp/smlsharp/`.