Flexible Representation Analysis[4] のサーベイ

SHIMIZU, Toshihiro

9 October 1997

1 始めに

多相型の有用性については今さら言うまでもないとして、多相型のはいった型システムを利用した実装技術の話である。多相型の入った型システムを持つ言語は、静的に型付けされているにもかかわらず、型付言語という言葉から予想される実装の恩恵を受けずに、長いこと実装されてきた。実際多相型があると、型付とはいっても静的に型が定まらないところがあるので、安直に考えれば、全てのデータはポインタ[1]にしてやらくちゃいけないなということになってしまい、実際そうして実装されてきた。有名な SML の実装の SML/NJ0.93 までででは floating のはいった演算をすると、すごく遅くなるのは周知のことである。

1.1 背景

そこで、近年[2]、型システムを利用したデータの実行時表現の改善 (representation analysis) をする技術が考案されてきている。最初のものは [2] で、型推論において、具体化される型情報を使って、データの coercion を行うコードを挿入するというものである。具体的には、$\lambda x.(x, x)$ のようなコードでは、x にくる値というのは、1 word で表現されていると決めてしまう。しかしそれだと、浮動小数点数等はあいかわらず boxed representation になってしまう。そこで、普段は unboxed representation をしておき、実際にこの関数に値を適用するとき、値を boxed representation に変換 (coercion) する。この coercion を行うためのコードというのは、型推論における、型の instantiation する型をもとに静的に決定できる。例えば、

```
let val f = fn x => x
in f 10; f 2.3 end
```

のようなプログラムでは、`f` に適用される値の型は静的に求まるので、その型情報を利用した coercion のコードをはさむことができる。実際`f 2.3` という呼び出しは $\mathrm{unwrap}[\texttt{real}](f(\mathrm{wrap}[\texttt{real}](2.3)))$ というような形に変更する。ここで、wrap は値を boxed representation に変換する関数である。この方式の問題点としては、(1)ref のような変更されうるデータについて適用できないことと、(2)list のような再帰的データ構造を coercion するのはコストがかかりすぎるので、実際上行えないという点があげられる。

これらの問題を解決する手法として、[1] では type passing で実装する手法を提案している。この手法では実行時に値を coercion して data representation を変更することはせず、かわりに実行時に型をひきずりまわすことで、静的にはわからなかった型を知り、それによって、データの representation を知り、また決めるということをするものである。ここでは、変更可能なデータや再帰的構造のデータの問題は解決できる。しかし、(1) 型の場合わけが非常に繁雑になり、(2) コンパイル時間がかかるという問題点がある。

---

[1] boxed representation という

[2] といっても、最初の論文がでてから 5 年近くなる

1.2 この仕事

この仕事はなにをしたものかというと、 coercion の実装を type passing を利用してより効率的にしようというものである。具体的には coercion のコストを減らそうというものである。具体的にはこれまであった coercion では、 polymorphic な関数に適用するには、末端まで、 boxed にしてしまっていたのを、最初のとこだけ boxed にしてやるだけでできないか？というものである。これだけだと何がなんだかわからないかもしれないが。

1.3 結果を先に言ってしまうと...

[2] であったように naive な実装から、 coercion をはさんだ変換における効率の向上に比べると、すごくよくなったという感じを一見受けない表である。しかし、よくよく見ると、どうも、`real*real*real` のような浮動小数点数のデータ構造を含み、その演算が多いものはすごく速くなっている。 polymorphism の多いものもコストそんなに遅くならずに済んでいる。それにしても、私の低いプログラミング能力から比べると、 Shao さんのコンパイラ作りの速さには驚嘆する思いばかりである。[3] [4]

1.4 Coercion v.s. Type Passing

Coercion と Type Passing の利害得失についてまとめておく。比較として naive な実装技法である、 naive な実装いついても比較のため触れる。まとめておく。

- Naive な実装方法。

  全てのデータは 1 word(boxed representation) になっている。全てのデータが 1 word なので、実装は楽だが、 C のようなプログラムに比べると非効率なことは否めない。また、 C や assembly 言語といった低レベルな言語とのインターフェースを設けるのも困難。

- coercion

  recursive data や mutable data は fully-boxed representation にならざるをえない。コンパイル時間は比較的短かく済み、 monomorphic code は高速に実行可能。

- type passing

  データの表現を実行時に変換することはないので、 ref 等の mutable data、 list 等の再帰的なデータ構造を扱うこともできる。しかし、場合分けが多くコストがかかる。これは大くのプログラムでは最適化によって除くこともできるが、モジュールをいれて分割コンパイルしたりする場合にはできないと思う。さらに以下のような問題点がある。

  - polymorphic 関数は非常に cost がかかってしまうこと。

    Type Passing では場合分けが非常に複雑多岐に渡ってしまい、そのための手続き呼び出しが頻繁に起こってしまうことが問題となる。もちろん、これらは各場合についてコードをさしこんでいけばなくすことができるが、当然コード爆発の問題を抱えることになる。

  - *vararg problem*

    という、より深刻な問題がある。要はコンパイル時に型がわからないと、どこに引数がわからん場合があるということだ。これを各場合毎にコードを作るなんてことをしてしまうと、 k 引数の関数は 2^k の場合があり、それぞれコードを作るなんてことしたら、自明にコードが exponential になってしまう。とんでもないということである。これを避けるのに、 TIL[5] では、引数は全部 boxed にしている。

[3] 一人で書いているのかは知らないが。

[4] Leroy にしてもあれだけコンパイラ書いて、論文書いてと、すごいという言葉しか浮かばない。

> ちなみに僕がコンパイラを作った時には C に落として、 SML2C の形でコンパイルをしたため、引数はメモリを使った受け渡しをしているため、受け渡しするのに、 2word をとっておくことで実装していた。

二つの本質的な違いは、 coercion を使う方法は実行時にデータを変換するのに対し、 type passing はデータを変更せず、各型に対して違うコードを走らせるという点にある。つまり、データの方を変更するか、コードの方を変えるかという話である。

2 ここではどうデータを表現するか？

というわけで、今日紹介する論文の話にやっと辿り着いた。この論文のお話というのは、先にも書いたが、キャッチフレーズとしては「coercion と type passing のいいところを合わせて幸せになろう」である。アイディアはしごく単純で、

- 多相型関数への適用は、 [2] のように、すべて coercion をして boxed representation にしてあげるんだけど、
- 先っちょまで全部 boxed にするのはコストがかかるから、それはせず、一番上だけ boxed にして、後困った時には runtime type information を使って、 boxing してあげよう

ということである。

2.1 表現方法について

Naive な実装方法では全ての data は 1 word で表現されていた。これを fully boxed representation と呼ぶ。ところで

```
fun move (x,y) = (x+1.0, y +1.0)
```

のようなコードのコンパイル結果として望まれるのは、`x,y` は floating pointer register にとって、 C でいう`double a[2]` を結果として返す関数を作ることである。このようにコンパイルするのを、 unboxed representation と呼ぶ。 [2] の coercion は、 unboxed representation のデータを

```
fun id x = x
```

のような関数に適用する時には、データを fully boxed representation に変換するのであった。ところで、よく考えると、`id` にこのデータを適用してきちんと動くためには、この段階で末端まで boxed になっている必要はない。つまり、 record の中身まで boxed になっている必要はないのである。このように一番上のところだけ boxed にするのを simply boxed representation と呼ぶ。これによって、 type passing で実行時に型をぐちゃぐちゃいじくるの変わりに、簡単な coercion だけで済ますことができる。大体、レコードにしろリストにしろクロージャにしろ、ほとんどのデータは boxed になっているから、実際にはなんのコストもいらない。しかしこの場合には coercion が実行時の型情報を必要とするようになるので、 type passing が必要になる。これは次のようなコードを考えればわかると思う。

```
let
 fun f x = ...
 fun g y = ... f (#1 y)
in
 g (1.0,2.0)
end
```

このとき、`x,y` はともに型変数を型としているとする。`g` に渡される`(1.0,2.0)` は boxed なので、 simply boxed representation ではそのまま渡される。しかし、`g` の中で、`f (#1 y)` と呼ぶ時に、

この値を boxed に変換しなくてならないが、これをするにはy の型情報が必要である。これはコンパイル時には定まらない。そこで型情報を実行時に与えてあげる必要が生じる。

ちなみに [2] で、 fully boxed representation にしなくてはならない理由として、例えば、 list を考えると、 polymorphic function の中でまた polyrmorphic function に適用される場合に、 list の中身が boxed でないと困るということがあげられた。この system ではどうなるかというと、こういう場合には、実行時に分かる型を使って、 boxing を遅らせるということをするのである。この場合、 boxing の回数は変わらないことを注意しておく。

なお、 simply boxed representation では、 mutable data や recursive data も効率良い表現ができるのは明らかだろう。というのも、 polymorphic function に渡す時に、問題となる部分を boxed に変換する必要はないから。

また、 runtime type manipulation と boxing の trade off で、引数は必ず boxed にするという partially boxed representation をとれば、 vararg problem を解消することもできる。

2.2 この方法の利点

- recursive and mutable type を coercion に基く手法で扱うことに可能にした。
- type-passing に比べれば、まだ runtime type information のコストは軽い。
- 等々

3 直観的なお話

Boxed representation の場合には、このような議論をする場合に canonical boxed form を決めておかないと困る。これまでの coercion に基く手法では、 full boxing が canonical boxed form であった。しかし、ここでは、 simply boxed representation や、 partially boxed representation といったものをいれるわけだから、それを考えないといけない。 wrap[τ] という関数は τ 型の値を canonical boxed form に変換する関数である。これまでの coercion base の手法では、この関数というのは常に fully boxed representation に変換するものであった。 unwrap は wrap の逆をする。

3.1 Simply Boxed Representation

Simply boxed representation は top layer のみ boxed representation にする。 $\mathtt{real} \times \mathtt{real}$ のような monomorphic value を simply boxed representation にするのが容易なことはすぐわかる。しかし、 $\beta \times \mathtt{real}$ のような型はどうするのだろう。 β を single word データと考えていくのはあまりうまくいかない。次の例で考える。

```
fun foo (x,y) = (x,y+3.0)::nil
fun bar (x,y) = (x+3.0,y)::nil
```

$\mathtt{foo} : \forall\beta.(\beta\times\mathtt{real}) \to (\beta\times\mathtt{real})$ であり、 $\mathtt{bar} : \forall\beta.(\mathtt{real}\times\beta) \to (\mathtt{real}\times\beta)$ である。foo やbar の中では、 β 型のデータはリストの中で coercion されてしまう。そのため、結果はどちらも simply boxed list でなくなる。 β は 1 word で boxed representation になっているから。 wrap[$\beta \times \mathtt{real}$] の望まれる定義は β を unboxed representation にして、くっつけるということをしてほしい。これをするには実行時に β の型が必要になる。

ここで重要なことは、 boxing scheme は valid なのは、 boxing scheme が type instantiation の関係と順番をいれかえても結果が同じだということである。 α が $\beta \times \mathtt{real}$ または $\mathtt{real} \times \gamma$ に instantiate され、次に $\mathtt{real} \times \mathtt{real}$ に instantiate されたとする。このとき、どっちからきても、最後の boxed form が同じであることが当然要求される。

full boxing よりも simple boxing の方が coercion が多い場合もある。が、ここでは触れない。

3.2 Partially Boxed Representation

simple boxing の問題点として、 wrap$[\tau]$ は vararg problem を引き起こすことがあげられる。これは partially boxed representation を用いることで解決できる。

Partially boxing の simple boxing との違いは、 unknown function の type については、実行時の型情報に左右されないように、引数と結果は常に simple boxed にする。これによって、 vararg problem を避けている。 partial boxing では、 coercion は runtime type に依存しない。この boxing scheme では、 wrap$[\alpha \to \beta]$ は coercion をしなくて済む。というのも、 α と β は partially boxed になっているから。

3.3 他の boxing scheme

もあるけれど、今回は省略。これらを *flexible* に切り替えられるというのが、 title の基になったのだろう。ちなみにどのような boxing scheme が妥当なものかは後で述べるが、直観的には、どういう順番で canonical boxed form にしても、結果が同じになるものが妥当な boxing scheme である。

4 Formal な話

ターゲット (TGT) は [1] 中で λ_i^{ML} と呼ばれるものである。 2 階の λ 計算に、 `typerec` という型についての場合分けの construct を加えたものと思えばいい。 SRC は Mini-ML と呼ばれる体系 (λ-calculus+tuple) に、 coercible tycon と incoercible tycon を区別するための、 `pk` , `upk` という二つの型コンストラクトと、その型を加えたものである。特に中間言語として、 F_ω の predicative variant(IL) を使用しているが、これは別に直接変換もできるのだが、 ML-style 以外の一般の多相型のコンパイル技法についても言えることを示すためだ。

Predicative polymorphism とはどういうものかの解説は省略する。おおまかには、 system F と呼ばれる impredicative polymorphism の体系では、型の中から型全体を参照している型が許されるのに対して、 predicative polymorphism では許されないという点が違いである。もっと分かり易く言うと、型抽象された恒等関数に多相型 ($\Pi t : T.\tau$ のような型) を適用してあげることはできない。というのも多相型は T の要素ではないから。ちなみに、 $\mathrm{let}\, x : \sigma = N \,\mathrm{in}\, M \,\mathrm{end}$ を predicative polymorphism の体系に加えたものが、 core-ML を陽に型付けしたものと考えることができ、これが XML で、よく研究されている。これは、 impredicative polymorphism と predicative polymorphism の中間で、 let というシンタクスで、多相的関数の宣言を可能にしている。当然のことながら、 predicative polymorphism では、 $\lambda f : (\Pi t.t \to t). \cdots f\ nat\ 3 \cdots f\ bool\ true \cdots$ のような項は書けない。なお、 impredicative polymorphism(system F) の項の type inference が undecidable なことが知られている。

話が横道にそれたが、ここでの中間言語が predicative variant だというのは、項に適用できるのは、 type constructor(μ) のみで、任意の polymorphic type を適用できないという意味である。じゃー、 let 式はどう変換されるのかということだけ疑問だと思うがそれに答えておく。別にどうってことなくて、 λ_i^{ML} にあるように IL では type constructor から型へ変換するための T という関数が用意されていて、実行時には、それによって、 type constructor を型に対応付ける。 type constructor の中に、 $\Lambda t :: \kappa.\mu$ というものがあって、それが多相型に対応している。ちなみに IL の type check は可能である[5]。

[5] predicative variant だから

pk、 upk に関する typing rule, operational semantics はそれぞれ以下のようになる。

$$\frac{\Delta;\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash \mathtt{pk}\, e : \mathtt{Pack}(\tau)} \qquad \frac{\Delta;\Gamma \vdash e : \mathtt{Pack}(\tau)}{\Delta;\Gamma \vdash \mathtt{upk}\, e : \tau}$$

$$\frac{e \hookrightarrow v}{\mathtt{pk}\, e \hookrightarrow \mathtt{pk}\, v} \qquad \frac{e \hookrightarrow \mathtt{pk}\, v}{\mathtt{upk}\, e \hookrightarrow v}$$

SRC から IL への変換については省略する。 SRC は ML の polymorphism で、 IL はより表現力の高い polymorphism をサポートしているから、変換できることは、容易に想像できると思う。変換は type directed に行われる。 IL から TGT への変換も省略する。ここでの話の肝は、 wrap という関数が、 TGT の中でどう定義され、 canonical boxed form がどのような形で calculus の中で表現されているかと思うので、そこを中心に述べる。

4.1 TGT Language

Target language のうち、 boxing,unboxing に関連するところだけ話をする。 TGT には次の primitives がある。ここで $\mathtt{Wrap}[\mu]$ は μ の canonical boxed from を表す type constructor と考えればいい。具体的には型の場合分けをする Typerec というコンストラクトを用いて実現されるのだが、具体的な Wrap[] の実現に依存しない話をするので、これは省略。見通しをよくしておくと、先に述べた simply boxed representation や、 partially boxed representation を表す $WRAPtycon$ は後で定義する。

$$\begin{aligned} \mathrm{wrap} &: \forall t :: \Omega. T(t) \to T(\mathtt{Wrap}[t]) \\ \mathrm{unwrap} &: \forall t :: \Omega. T(\mathtt{Wrap}[t]) \to T(t) \end{aligned}$$

4.2 coercion function の挿入

最初に Leroy が提案した方法での coercion function の挿入で、肝になるのは変数のところで、次のような形であった。

$$\frac{\Gamma(x) = \forall \alpha_1 \ldots \alpha_n.\tau \quad \mathrm{dom}(\rho) \subset \{\alpha_1, \ldots, \alpha_n\}}{\Gamma \vdash x : \rho(\tau) \Rightarrow \mathrm{S}_\rho(x, \tau)}$$

$$\begin{aligned} \mathrm{S}_\rho(a' : \alpha) &= \mathrm{unwrap}[\rho(\alpha)](a') \\ \mathrm{S}_\rho(a' : b) &= a' \\ \mathrm{S}_\rho(a' : \tau_1 \times \tau_2) &= \mathrm{let}\, x = a' \,\mathrm{in} \\ &\qquad (\mathrm{S}_\rho(\mathrm{fst}(x) : \tau_1), \mathrm{S}_\rho(\mathrm{snd}(x) : \tau_2)) \\ &\quad \mathrm{end} \\ \mathrm{S}_\rho(a' : \tau_1 \to \tau_2) &= \lambda x.\, \mathrm{S}_\rho(a'(\mathrm{G}_\rho(x : \tau_1)) : \tau_2) \end{aligned}$$

$$\begin{aligned} \mathrm{G}_\rho(a' : \alpha) &= \mathrm{wrap}[\rho(\alpha)](a') \\ \mathrm{G}_\rho(a' : b) &= a' \\ \mathrm{G}_\rho(a' : \tau_1 \times \tau_2) &= \mathrm{let}\, x = a' \,\mathrm{in} \\ &\qquad (\mathrm{G}_\rho(\mathrm{fst}(x) : \tau_1), \mathrm{G}_\rho(\mathrm{snd}(x) : \tau_2)) \\ &\quad \mathrm{end} \\ \mathrm{G}_\rho(a' : \tau_1 \to \tau_2) &= \lambda x.\, \mathrm{G}_\rho(a'(\mathrm{S}_\rho(x : \tau_1)) : \tau_2) \end{aligned}$$

S_ρ は boxed な値を unboxed representation に値を変換する関数を加え、 G_ρ は unboxed な値を boxed representation に値を変換する関数を加える。

さて、この論文では F_ω に coercion をいれるので、 Leroy の手法における coercion function の挿入は type instantiation のところで行われる。本質的に同じであるのは、 ML の場合はある意味で polymorphic type が型の位置に出現するのは変数の所だけなことから自明だろう。その IL からの TGT への変換規則は次のようになる。

$$\frac{\Delta;\Gamma \vdash e : \forall t :: \kappa.\sigma \Rightarrow e' \quad e'_c = \mathcal{C}_{\mathcal{C}}([\mu^\omega/t]\sigma^u, [\mu^u/t]\sigma^u)}{\Delta;\Gamma \vdash e[\mu] : [\mu/t]\sigma \Rightarrow @e'_c(e'[\mu^\omega])}$$

$\mathcal{C}_{\mathcal{C}}(\mu', \mu)$ は、 TGT の項で、 $T(\mu')$ 型の項を $T(\mu)$ 型の項に coercion する。 $\mathcal{C}_{\mathcal{T}}(\sigma', \sigma)$ は σ' 型の項を σ 型の項に変換する。

このの $\mathcal{C}_{\mathcal{C}}$ という関数が上の S_ρ という関数に相当する。 polymorphic value は IL では、使う時には型を適用するが[6]、その適用される型を使って、 coercion function を挿入する。 Polymorphic function に適用される値は canonical boxed form である必要がある。それでこれらの引数を coerce する関数をいれる。結果は polymorphic value なので、 canonical boxed form であるから、復帰する関数をいれる。 $\mathtt{Wrap}[\mu]$ というのは tycon μ で表される型の canonical boxed form になっている値を表す型だった。

簡単に言うと、多相型の関数は canonical boxed form の値として受け取って、 canonical boxed form の値を返すようになっているので、 $\mathcal{C}_{\mathcal{C}}$ は、その関数をもらい、その前後に coercion のコードを挿入するコードをはくとよい。規則の $\mathcal{C}_{\mathcal{C}}([\mu^w/t]\sigma^u, [\mu^u/t]\sigma^u)$ は、 canonical boxed form の値を、もとの型に変換する関数と思える。これは直観的には、 σ を関数型と思っていうとわかりやすい。上の規則中の e' は polymorphic function なので、 canonical boxed value をもらって、 canonical boxed value を返す。それで、このままでは適用したときに変なことになる。そこで、その関数をその文脈で定まる型になるように coercion function をくっつける。その型は $[\mu^u/t]\sigma$ である。

$\mathcal{C}_{\mathcal{C}}, \mathcal{C}_{\mathcal{T}}$ と $\mathrm{S}_\rho, \mathrm{G}_\rho$ の対応は次を見るとよくわかる。 σ_1, σ_2 はそれぞれ canonical boxed form を表す型である。

$$\mathcal{C}_{\mathcal{T}}(\sigma'_1 \to \sigma'_2, \sigma_1 \to \sigma_2) = \lambda f : \sigma'_1 \to \sigma'_2.\lambda x : \sigma_1.(@(\mathcal{C}_{\mathcal{T}}(\sigma'_2, \sigma_2))(@(f)(@\mathcal{C}_{\mathcal{T}}(\sigma_1, \sigma'_1))(x)))$$

Polymorphic function に適用する場合にはその値を canonical boxed form に変換するというのが、 coercion を利用した実装技法の主眼であった。それで、その関数自身は canonical boxed form をもらって、 canonical boxed form を返すようになっていなくてはならず、従ってその型は $\sigma'_1 \to \sigma'_2$ である。で、引数は、 non-canonical boxed form でありうる値を canonical boxed form にしないといけないので、 $\mathcal{C}_{\mathcal{T}}(\sigma_1, \sigma'_1)$ に x を適用して、 f に渡す。一方計算結果をこの文脈で使うには、 canonical boxed form から、文脈から定まる表現に戻してやる必要があるので、 $\mathcal{C}_{\mathcal{T}}(\sigma'_2, \sigma_2)$ をはさむ。そういうこと。

4.2.1 もちょっとフォーマルに

μ^u は IL の type constructorμ を TGT の type constructor に変換するものである。殆ど自明な変換だが、違うのが一点だけある。 $\mathtt{Pack}()$ のような incoercibale tycon は、 wrapped なそれとする。つまり、

$$(\mathtt{Pack}(\mu))^u = \mathtt{Pack}(\mathtt{Wrap}[\mu])$$

となる。 incoercible tycon は monomorphic context と polymorphic context で表現を変えない。型の変換も同様に行う。次に、 wrapped な変換を示す。

$$\begin{aligned} \mu^w &= \mathtt{Wrap}[\mu^u] \qquad \text{if } \Delta \triangleright \mu :: \Omega \\ \mu^w &= (\Lambda t :: \kappa.\mu[t])^w \text{if } \Delta \triangleright \mu :: \kappa \to \kappa' \\ (\Lambda t :: \kappa.\mu)^w &= \Lambda t :: \kappa.\mu^w \end{aligned}$$

substitutionρ の ρ^u の意味は $(\rho(t))^u$。

[6] 厳密には type constructor

4.1 Definition [valid canonical boxed form]: $\mathtt{Wrap}[]$ の定義が与えられた時に、 substitution と可換 ($\rho(\mathtt{Wrap}[\mu]) = \mathtt{Wrap}[\rho(\mu)]$) かつ、次の等式が導ける。

$$\begin{aligned}
\mathtt{Wrap}[\mu] &\equiv \mathtt{Wrap}[\mathtt{Wrap}[\mu]] \\
\mathtt{Wrap}[\times(\mu_1,\mu_2)] &\equiv \mathtt{Wrap}[\times(\mathtt{Wrap}[\mu_1],\mathtt{Wrap}[\mu_2])] \\
\mathtt{Wrap}[\rightarrow(\mu_1,\mu_2)] &\equiv \mathtt{Wrap}[\rightarrow(\mathtt{Wrap}[\mu_1],\mathtt{Wrap}[\mu_2])]
\end{aligned}$$

このように定義された valid canonical boxed form については、 μ^u, σ^u, μ^w が substitution と可換なことがいえる。

4.3 そいじゃー、最初に言ってた話はどう表現されるの？

というわけで、最初に言っていた、 partial boxing、 simply boxing や full boxing はどうなったんだということになるが、これらは全て、 canonoical boxed form であることが言える。

4.3.1 $\mathtt{Wrap}[]$ の定義

例えば、 simple boxed form はつぎ のようになる。ここでは pattern match の形で書いてあるが、実際に constructor としては、 `Typerec` を使って書く。

$$\mathtt{Wrap}[\mu] = \mathtt{Boxed}(\mathtt{Uncv}[\mu])$$

$$\begin{aligned}
\mathtt{Uncv}[\mathtt{int}] &= \mathtt{int} \\
\mathtt{Uncv}[\mathtt{real}] &= \mathtt{real} \\
\mathtt{Uncv}[\mathtt{Pack}(\mu)] &= \mathtt{Pack}(\mu) \\
\mathtt{Uncv}[\mathtt{Boxed}(\mu)] &= \mu \\
\mathtt{Uncv}[\times(\mu_1,\mu_2)] &= \times(\mathtt{Uncv}[\mu_1],\mathtt{Uncv}[\mu_2]) \\
\mathtt{Uncv}[\rightarrow(\mu_1,\mu_2)] &= \rightarrow(\mathtt{Uncv}[\mu_1],\mathtt{Uncv}[\mu_2])
\end{aligned}$$

僕も大山様や細谷様の真似して、問題を出してみます。

Problem 4.1 *Partial boxing*、 *full boxing* を上にならって定義すると、どう書けばいいか？

4.3.2 wrap, unwrap の定義

いよいよ、 wrap, unwrap の定義を述べる。 Simple boxing では次のように定義する。

$$\begin{aligned}
\mathrm{wrap}[\mu] &= \mathtt{box}[\mathtt{Uncv}[\mu]] o \mathtt{uncv}[\mu] \\
\mathrm{unwrap}[\mu] &= \mathtt{cover}[\mu] o \mathtt{unbox}[\mathtt{Uncv}[\mu]]
\end{aligned}$$

$$\begin{aligned}
\mathtt{uncv}[\mathtt{Boxed}(\mu)] &= \mathtt{unbox}[\mu] \\
\mathtt{uncv}[\times(\mu_1,\mu_2)] &= \mathtt{uncv}[\mu_1] \times \mathtt{uncv}[\mu_2] \\
\mathtt{uncv}[\rightarrow(\mu_1,\mu_2)] &= \mathtt{cover}[\mu_1] \rightarrow \mathtt{uncv}[\mu_2] \\
\mathtt{uncv}[\mu] &= \mathtt{identity}[\mu]
\end{aligned}$$

$$\begin{aligned}
\mathtt{cover}[\mathtt{Boxed}(\mu)] &= \mathtt{box}[\mu] \\
\mathtt{cover}[\times(\mu_1,\mu_2)] &= \mathtt{cover}[\mu_1] \times \mathtt{cover}[\mu_2] \\
\mathtt{cover}[\rightarrow(\mu_1,\mu_2)] &= \mathtt{cover}[\mu_1] \rightarrow \mathtt{uncv}[\mu_2] \\
\mathtt{cover}[\mu] &= \mathtt{identity}[\mu]
\end{aligned}$$

Partial boxing では、 uncv[], cover[] は $\rightarrow$ (Boxed(μ_1), Boxed(μ_2)) のような constructor については、 `identity` と同じになることを注意しておく。

5 Module への拡張について

省略

6 結果について

ベンチマークの結果は図 9 にあるようになっている。数字がぱらぱらしていて見にくいが、全部がすごく速くなったというわけではない。 Leroy の手法によるコンパイラと比べている。

mutable or recursive tycon は `pack` のように扱う。 list は simple boxed representation である (要素は boxed)。配列は real、 int32 の時は flat array。他は boxed array。

nucleic は `real * real * real` の巨大なリストを含む。こいつが、 full boxed から simply boxed になったおかげで高速になった。 **simple**、 **fft** も同じ理由で高速化された。一方、多相型がすごく多い **seive**、 **mlyacc**、 **vliw**、 **kb-comp** はほぼ同じ結果。これらは、 monomorphic data に直接適用されるか、 incoercible object に適用されている。 5% から 10% の性能低下は type passing のコストと思われる。 **unzipr** は coercion が余分にかかっていて遅くなっていると考えられる。これらの speed-up は TIL の speed-up と同等だそうである。しかし、 polymorphic code について、コストがかからないでいるのがいいと著者は述べている。

7 感想

いずれにせよ、近年の多相型の実装技術の研究は、型を実行時に利用することや、中間コードを型付けして、効率のよいコードをはくといった手法をもたらした。この研究もその一つである。

型情報を実行時に利用するために、型のデータを実行時に構成するのは、 [6] にあるように lazy type instantiation をすれば、そんなにコストがかからないとはいえ、この論文の実験のように、type passing をすると実行時間が、 10 この手法では、 [6] のような手法はとられていない。また、 [6] の手法では copy gc を通常の方法で行えないという欠点があった。この点も含めて [3] では、型の実行時における生成のコストをより軽減させる手法が提案されている。僕も、このコストを軽減させようとして、研究をしようとしていたのだが、 [6] の手法をよく理解していなかったせいか、 tolmach の手法より効率の悪い手法でやっていた。とりあえずやっていたことはコンパイラを作って、実際に大くのプログラムでコストがかかることを確認して、自分の言ってる手法だとコストが減って嬉しいねというストーリーのつもりだったがだめだった。

これからどうするか。輪島から曽々木へと向かう道の侘しい海を見たり、柳田村の宿で一人考えた。能登の北は侘しいところだった。

References

[1] Robert Harper and Greg Morrisett. Compiling Polymorphism Using Intensional Type Analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130 – 141, 1995.

[2] Xavier Leroy. Unboxed objects and polymorphic typing. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177 – 188, 1992.

[3] Yasuhiko Minamide. Full Lifting of Type Parameters, 1996. An earlier version of this paper appeared in the proceedings of Second Fuji International Workshop on Functional and Logic Programming as Compilation Based on a Calculus for Explicit Type Passing.

[4] Zhong Shao. Flexible Representation Analysis. In *International Conference on Functional Programming*, 1997.

[5] D. Tarditi, G. Morriset, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *Conference on Programming Language Design and Implementation*, 1996.

[6] A. Tolmach. Tag-free garbage collection using explicit type parameters. In *ACM Conference on LISP and Funcional Programming*, pages 1 – 11, 1994.