COMFRK

VOL.1

目　次

夏休み子供λ相談室

著 : ranha

はじまり

凛　こんばんわ。せんせー、ちょっとお願いがあるんだけど。

有也 あれ？どうかした？

凛　うん。実は余りにも暇だったから、先生にもらった本を読んでたんだけど。

有也 ・・・覚えてない・・・。何の本？

凛　計算がどうこう？っていうことが書いてる本なんだけど。えーと、らむだ計算ていうのでちょっと分からない所があるんだ。

凛　それで、明日にでも教えて貰えないかなーと思って電話してみたんだけど。

有也 λ計算ねー。それで、何が分からないの？

凛　えーっとね、普通のλ計算じゃなくて、型がくっついたλ計算では、どんな式も停止するっていう話。これ証明が載ってないんだよ。

凛　自分で考えてみたんだけど、お手上げじゃー。

有也 強正規化定理の話かな？

凛　ソレ！先生分かる？

有也 まぁ、教えられないことは、ないと思う。けどめんどうくさいなぁ・・・。何かくれたらやる。

凛　家庭教師の癖に！！生徒の奴隷じゃなかったの！！

有也 酷過ぎる・・・。時間外労働なんだから、何かくれ。

凛　アイスクリームを複数個ならどう？

有也 それは乳固成分 15% 以上 (うち乳脂肪分 8% 以上) のもの？

凛　はい・・・。

有也 あとクーラーも付けて！！

凛　クーラーなんてあげられるわけないです！！

有也 いや・・・そういう事じゃなくて冷房の効いた部屋にしておいてくださいって事。

凛　それなら多分大丈夫。じゃあちゃんと質問出来るように勉強しておくね。

証明のはじまりは、定義から

有也 涼しい！！クーラーはすごいなぁ・・・。ところで、どう？自分で証明出来た？

凛　やっぱりダメ。良く分かんない。だいたい、どう証明していけばいいのか分かんない。

有也 なるほど何が分からないのか分からないし、取りあえず、順番に定義していってみるよ。

Def. 型付きλ計算の型を、次のように再帰的に定義します。

(1) τ が基底型であれば、τ は型

(2) τ1 と τ2 が型であれば、τ1 ⇒ τ2 は型

(3) 以上 (1) - (2) の繰り返しによって出来るものだけが型付きλ計算の項

有也 一応最初だから書いてみたけど、(3) のような規則は普通は省略されるよ。

Def. 型付きλ計算の項を、次のように再帰的に定義します。

(1) 変数 x は項

(2) t が項で、x が変数ならば、λx.t は項

(3) t1 と t2 が共に項であれば、t1t2 は項

有也 じゃあ練習、全ての項は 1 つ以上の変数を含む。さて、これはどうやって示すと良いでしょうか！

凛　えっと、構造に関する帰納法、っていうのを使えば良いんだよね。

有也 項の構造に関する帰納法を使うと、どういう風に証明出来るのかな？

凛　これは簡単だから出来そうだよ。

proof begin

項の形で場合分けして考える。

case 変数 {
　明らかに成り立つ。
}

case λx.t{
　t は項なので、帰納法の仮定から 1 つ以上の変数を含む。
　しかし、そもそも x があるから明らか。
}

case t1t2{
　t1,t2 は帰納法の仮定から、それぞれ 1 つ以上の変数を含む。
　よって明らか。
}

proof end

有也 うん。練習だからこんな感じで良いよ。ちなみに、同じようにして型に関しても構造に関する帰納法が使えるね。

凛　本にも書いてたから、とりあえず使ってるけど。でもなんでコレで大丈夫なのかは分かって無いよ。

有也 βリダクションと、型付けに関する帰納法のところでまとめて説明するから大丈夫。じゃあ定義を続けるよ。

Def. 型付きλ計算のβリダクションを、項の間で、次のように再帰的に定義します。

$$\frac{t \to_\beta t'}{\lambda x.t \to_\beta \lambda x.t'}\ (\mathrm{REDlam})$$

$$\frac{t1 \to_\beta t1'}{t1t2 \to_\beta t1't2}\ (\mathrm{REDapp1})$$

$$\frac{t2 \to_\beta t2'}{t1t2 \to_\beta t1t2'}\ (\mathrm{REDapp2})$$

$$\frac{}{(\lambda x.t1)t2 \to_\beta t1[x:=t2]}\ (\mathrm{REDapp3})$$

(t1[x:=t2] は、項 t1 に自由出現する変数 x を、全て項 t2 で置き換える事を表した記号です。REDlam などは規則に対して名前をつけているわけです。RED は reduction の略です。)

Def. 型付き λ 計算の文脈。

(1) 変数 x が型 τ であることを、x:τ として表すとした時、有限の集合 {x1:τ1, x2:τ2, ..., xn:τn} を 文脈 と呼びます。通常、文脈は Γ を用いて表します。

(2) Γ,x:τ = Γ ∪ {x:τ} (ただし x:τ ∉ Γ) とします。(つまり直和になります)

Def. 型付き λ 計算の型付け規則を、次のように再帰的に定義します。

$$\frac{x:\tau \in \Gamma}{\Gamma \vdash x:\tau} \quad \text{もしくは} \quad \frac{x:\tau \in \Gamma}{\Gamma,x:\tau \vdash x:\tau}\ (\mathrm{DERvar})$$

$$\frac{\Gamma,x:\tau1 \vdash t:\tau2}{\Gamma \vdash \lambda x.t:\tau1 \Rightarrow \tau2}\ (\mathrm{DERlam})$$

$$\frac{\Gamma \vdash t1:\tau1 \Rightarrow \tau2 \qquad \Gamma \vdash t2:\tau1}{\Gamma \vdash t1t2:\tau2}\ (\mathrm{DERapp})$$

(ここでも規則に対して名前をつけています。DER は derivation(導出)の略です。)

有也　取りあえず、上のように横棒を使って定義された規則を"導出規則"と呼びます。導出規則を使って"結論"を導く事を、"導出"と言います。"結論"ていうのは横棒の下の事だと思ってもらっていいよ。

凛　先生２つ質問がありますよ？まず、⊢ ってなんなの？

有也　そうだね。例えば、Γ ⊢ t1:τ1 ⇒ τ2 と書いて、Γ" の元で" 項 t1 を型 τ1 ⇒ τ2 であると "型付け出来る" 事を表す記号になるんだよ。

凛　なるほどー。それからそれから、この横棒は何を表してるんですか？

有也　例えば、規則 REDapp1 に注目してみよう。横棒の上には、t1 →β t1'。横棒の下には、t1t2 →β t1't2。

これは、t1 →β t1' が成り立つ "ならば" t1t2 →β t1't2 が成り立つ事を意味してるんだよ。

凛　じゃあ、横棒を使わずに、ならば、を使って定義すれば良いんじゃ無いですか？

有也　まぁ横棒を使っている事には理由があるんだ。ちょっと次の例を見て。

$$\frac{\dfrac{z:\tau1,\ x:\tau1 \vdash x:\tau1}{z:\tau1 \vdash (\lambda x.x):\tau1} \qquad z:\tau1 \vdash z:\tau1}{z:\tau1 \vdash (\lambda x.x)z:\tau1}$$

有也　これは環境として {z:τ1} を持つ時に、(λx.x)z が τ1 で型付け出来る事を表してる図なんだけど、こういう図を "導出木" と言います。

一番下 (" 結論 ") を根として上方向に成長しているという見方をすると木に見えなくもないです！

有也　もし、普通に " ならば (→)" を使って定義していると、こういう見やすい形にはちょっと出来ないよね。

凛　おー！！

有也　さて、β リダクションや型付け全体に関する性質を証明する時にも、やっぱり帰納法を使いたいわけ。

じゃあ、それぞれについてどういう形で帰納法が定義出来るかな。

凛　構造に関する帰納法があいう形だから・・・横棒の上で成り立つと仮定して、下で成り立つ事を証明出来れば良いのかな？

有也　結果としては、それで正しいよ。ちなみに、そのような帰納法を導出に関する帰納法と呼ぶ。

ところで、導出に関する帰納法は、より一般的な " 整礎帰納法 " のインスタンスになってる。

この整礎帰納法を利用して、何が言えれば帰納法であると言えるか、を考えるよ。

凛　清楚って私みたいな？

有也　集合上の関係 < で、

... $a_2 < a_1 < a_0$ といった無限下降列が存在しないものを整礎な関係 (well-founded relation) と言う。

この時、整礎帰納法を…

Def. 整礎帰納法。

< を集合 A 上の整礎な関係、P を A の要素に関する述語とした時に、

$\forall a \in A.((\forall b<a.P(b)) \to P(a)) \leftrightarrow \forall a \in A.P(a)$

有也　と定義します。

凛　無視しないでよ！！

有也　じゃあまずおなじみの数学的帰納法を整礎帰納法から言うには、どういう整礎な関係を言えば良いのかな？

凛　n < m ↔ suc(n) = m かな・・・。自然数に関する話だから、無限下降列も存在しないね。

有也　構造に関する帰納法は？具体的に、型の構造に関する帰納法

で考えてみよう。

凛　τ'<τ ↔ (∃τ''.(τ'⇒τ'') = τ または、
∃τ''.(τ''⇒τ') = τ) だとすると、基底型の場合が在るから無限下降列は存在しないね。

有也 そうそう。導出に関する帰納法も言えるかな？

凛　んー、

$$d' < d \leftrightarrow \frac{\ldots\ d'\ \ldots}{d}$$

…かな。

有也 "1つだけ小さい"部分導出を定義しているわけだね。本当に無限下降列が存在しないことを確認してみて。

凛　リダクションの時はREDapp3があるし、それから、型付けの時にはDERvarがあるからそこでストップ！されるのかな。

有也 そうだね。これで整礎帰納法という一般的な形から、導出に関する帰納法が得られた。

有也 でも、

$$d' <_1 d \leftrightarrow \frac{\ldots\ d'\ \ldots}{d}$$

…として、< を $<_1$ の推移的閉包として定義しても整礎な関係になるので、一般にはこちらを導出に関する帰納法として使うよ。こっちはd' < dを､d'がdの部分導出であること、としている。

簡単に言うと任意の導出dで性質Pが成り立つ事を言うには、dの任意の部分導出d'でPが成り立つと仮定した時に、dでPが成り立つ事を示せば良い。

凛　この導出に関する帰納法っていうのを使えば、私が昨日つまずいた所も上手く出来るかも！

有也 そういえば、どこでつまずいていたの？

強正規化定理って？

凛　つまずくっていうより、何も出来てなかったんだけど。強正規化定理を何の工夫もせずに証明しようと思ったの。

有也 まぁ、実はそれですんなり出来るかもしれないけどね。とりあえず、強正規化定理(Strongly Normalization Theorem)の定義を言ってください。

凛　うーんと。型付きλ計算で、ある項が型付け出来るならば、その項が強正規化可能である、かな。

有也 強正規化可能というのは？

凛　ある項が強正規化可能っていうのは、その項を始点にする無限のβリダクション列が存在しない、です。

有也 言い換えると、ある項を始点として、必ず有限のリダクションで正規な項が見つかる、ということだね。

凛　正規な項って何？

有也 それ以上βリダクション出来ないような項ってこと。項tが強正規化可能～と書くのがちょっと面倒だから、

Def. SN(t) ↔ 項tが強正規化可能である

有也 …と書くことにします。

凛　この定理ってもしかして、型付きλ計算は、型のないλ計算よりも弱いって事を言ってるのかなぁ。λ計算ではこの定理は成り立たないんだよね。

有也 弱いって・・・。何を言いたいのかはなんとなく分かるけど。それよりもだ・・・。いいぜ。型付きλ計算が何でも計算出来ると思ってるのなら

凛　まずはそのふざけた幻想をぶち殺す！

有也 殺すとかいっちゃダメですよ～。

凛　今のハメだよ！

有也 ところで、強正規化可能な項tについて、tを始点とする全てのβリダクション列のうち最も長い列の長さをν(t)と書く事にするよ。

tは強正規化可能だから、全てのβリダクション列の長さは高々有限になる事に注意。ν(t)に関する帰納法を使うからね。

Def. 強正規化可能な項tについて、ν(t)＝項tを始点とする全てのβリダクション列のうち最も長いものの長さ。

有也 項tを根として、βリダクションで成長していくような木を考えてもらえば、ν(t) = そうやって作られた木の高さ、とも言える。

凛　強正規化可能な項tは、高々ν(t)ステップあれば全ての正規な項を見付ける事が出来るわけね。

強正規化定理をそのまま証明しようとする

有也 強正規化定理を次の形で証明したい、わけだよね。

strong normalization theorem
Γ ⊢ t:τ → tが強正規化可能。

凛　導出に関する帰納法を手に入れたんだし、さっそく装備して使ってみようっと。

proof begin

Γ ⊢ t:τの導出に関して場合分けをします。
case DERvar{
　t = x
　変数に対するβリダクションは定義されていないので、

明らかに成り立つ。

}

case DERlam{

t = λx.y, τ = τ1 ⇒ τ2

$$\frac{\Gamma, x:\tau 1 \vdash y:\tau 2}{\Gamma \vdash \lambda x.y:\tau 1 \Rightarrow \tau 2}\ \mathrm{DERlam}$$

Γ ⊢ λx.y:τ1 ⇒ τ2 の導出に関する帰納法を使うとすると、
帰納法の仮定として、
Γ,x:τ1 ⊢ y:τ2 → y が強正規化可能が使えて、
今 (Γ ⊢ λx.y:τ1 ⇒ τ2) が言えているので、
棒線の上の (Γ,x:τ1 ⊢ y:τ2) も言えているから、
y が強正規化可能ということが言える。

凛　λx.y →β λx.y' →β λx.y'' ... になってて、しかも y は強正規化可能なんだから、いつかは止まるね。

y が強正規化可能なので、λx.y も明らかに強正規化可能。

}

case DERapp{

t = t1 t2, τ = τ2

$$\frac{\Gamma \vdash t1:\tau 1 \Rightarrow \tau 2 \qquad \Gamma \vdash t2:\tau 1}{\Gamma \vdash t1t2:\tau 2}\ \mathrm{DERapp}$$

Γ ⊢ t1t2:τ2 の導出に関する帰納法を使う。
帰納法の仮定から、t1 と t2 が強正規化可能。

凛　この時、t1t2 が強正規化可能になる事を示したいから、うーんと・・・つまりこれ以上 β リダクション出来ない事を言えば良いんだし、取りあえず項 t1 の形について場合分けしてみよう。

case t1 = x{

xt2 が強正規化可能になる事を示す。

凛　x が変数で β リダクション出来ないから、この項を β リダクションしようと思ったら、t2 →β t2' →β t2''... とするしかない。でも t2 は強正規化可能なんだから、いつかは止まる。

x は変数で、t2 は強正規化可能なので明らかに成り立つ。

}

case t1 = λx.t1'{

(λx.t1')t2 →β t1'[x:=t2] になる。

t1'[x:=t2] が強正規化可能であることが言えれば、
明らかに 1 ステップ前の (λx.t1')t2 も強正規化可能。

凛　導出に関する帰納法を使いたいけど、t1'[x:=t2] って、一般に (λx.t1')t2 の導出中に出てこない・・・。先生、これ導出に関する帰納法は使えない？

有也 (λx.t1')t2 の導出過程で、t1'[x:=t2] が現れているとは一般には言えない。じゃあ、他の帰納法はどうだろう？

凛　えっと、まず項の構造に関する帰納法だよね。帰納法を使うためには、t1'[x:=t2] が (λx.t1')t2 の部分構造になってるってことを言えないといけない。
でも、t1' 中に x が沢山現れていたら、その分だけ t1'[x:=t2] は複雑になっちゃうから、ダメな気がする。

有也 型の構造に関する帰納法は？

凛　型の構造に関する帰納法は、(λx.t1')t2 の型が τ2 だとすると、t1'[x:=t2] の型も τ2 になるけど、τ2 ⇒ τ2 にはなんないから型の構造に関する帰納法もダメ。

凛　やっぱりダメじゃん！！

proof fail

}

}

有也 もうダメとか言ってる間はずっとダメなんだよ！考えて！もっと考えて！！

凛　あかん?。あかんよ?。

有也 昔の偉い人は、急がば回れと言った。つまり休憩しよう。

休憩 1

有也 さて、直接突っ込むと途中でつまっちゃったね。

凛　(λx.y)z の形で帰納法が使えなくなったよ。

有也 そうだねー。じゃあどうするか。帰納法が使えないんだったら。

凛　帰納法を使えるようにすれば良いじゃない！

有也 そう！(λx.y)z の形で帰納法が使えるようにしたら良いんだよ。

凛　えっと、じゃあ・・・うんと・・・？・・・え？

有也 今定義しているものだとどうにもならないから、新しい性質を定義してあげよう。

凛　分かんないよ・・・そんなの。

有也 どういう定義だと良さそうなのか、順番に考えてみよう？
Γ ⊢ t:τ ⇒ t が強正規化可能の代わりに、Γ ⊢ t:τ ⇒ t が性質 P を満たすことを証明したいわけ。

凛　強正規化可能の代わりに使うんだから、性質 P を持つ事がいえれば、強正規化可能である事が言えれば良いのかなぁ。

凛　それから (λx.y)z ってなった時に帰納法が使えたら良いんだよね。

有也 もうちょっと詳しく、さっき失敗した時の状況を考えてみて。

凛　(λx.y)z が強正規化可能であることを証明しようと思って、
・(λx.y) が導出に関する帰納法の過程から、強正規化可能である。
・z も導出に関する帰納法の過程から、強正規化可能である。
って事が言えてた。この中で強正規化可能を、性質 P を持つって事で置き換えてみて・・・。

凛　(λx.y) と z が性質 P を持つなら、(λx.y)z が性質 P を持つ、で良いのかな？

有也 それだとすぐに全ての項が性質 P を持つって事は言える。だから、性質 P を持つならば強正規化可能である、の証明を考えてみるよ。
(λx.y)z が性質 P を持つならば、(λx.y) と z が定義から性質 P を持つって事が言える。
だから、帰納法を使って (λx.y) と z が共に強正規化可能である事が言えるんだけど、また y[x:=z] が強正規化可能であるって事を言わないといけなくなっちゃうよ。

凛　じゃあ、z が性質 P を持っていて (λx.y)z が性質 P を持ってるんだったら、λx.y が性質 P を持つ。なのかな。

有也 それだと、基底型と関数型の場合で分けて、型の構造に関して定義するのが自然な気がするね。

凛　これで合ってるの？

有也 実はこれで合ってる。先に定義をあたえておくと。

Def. 型 τ を持つ項 t の reducibility を型に関して帰納的に定義する。

(1)τ が基底型のとき
項 t が強正規化可能であれば、項 t は reducible。
(2)τ が τ1 ⇒ τ2 のとき
任意の型 τ1 を持つ reducible な項 t1 で、tt1 が型 τ2 で reducible になるならば、項 t は型 τ1 ⇒ τ2 で reducible。

Def. reducible(t,τ) は、t は型 τ を持つ reducible な項であることを表す。

凛　私はあてずっぽうしたけど、この定義で上手くいく理由が分かんない。

有也 ポイントは、強正規化可能な項というのは何も型付き λ 計算にしか存在しないわけじゃないということ。型無し λ 計算でも、いくつも強正規化可能な項は作れるよね？

凛　あ、そっか。それもそうだね。

有也 型付き λ 計算は、型を持っているから、その分だけ型無し λ 計算よりも縛りが強い。
その型を利用して reducibility は定義されている。だから強正規化可能性の定義に比べて、制約が強い。

有也 ところで、この後は強正規化定理の証明に入って行くんだけだけど、大きく 2 パートに分かれるんだ。どんな 2 つに分かれると思う？

凛　えっと、まず reducibility が言えたら強正規化可能である事が言えないとダメだよね。
それから、型付け出来る項が強正規化可能である事を証明したいんだから・・・。型付け出来る項が reducibility を満たす、かな？

有也 そう。その 2 つに分けられるんだ。

memo 証明の流れ

1. reducible な項は、強正規化可能である。
2. 型付けされる項は、reducible な項である。

有也 1 は reducibility の定義が強正規化可能の定義よりも強い事を言ってる。
でも、定義は単に強ければ良いってものでもない。あんまり縛りが強いと、型付け λ 計算の項でもそれを満たさないものが出て来てしまう。
ここで 2 が意味を持ってくるね。

有也 あと、reducibility で基底型の時に強正規化可能であることを仮定しているのは、reducible な項が強正規化可能である事を帰納法を使って証明したいからというのも分かるかな。
基底型は、型の構造に関する帰納法で基底に相当するからね。

凛　なるほど、reducibility は絶妙な定義になってる、って事なんだね！！ところで、突然 reducibility って名前がついちゃったけど、なんで？

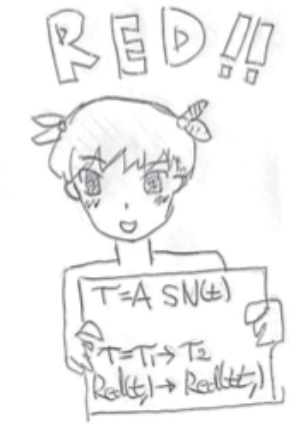
有也 それは以降の証明が、Girard らの Proofs and Types という本でなされた証明をもとにしてるからだよ。その本で、上の性質 P を reducibility と呼んでるんだ。

凛　なんと虎の巻がござったか。

有也 まぁ手っ取り早く言えばコピペね。コピペついでに、便宜上なんだけど、項が neutral である事を言う neutrality というのを定義しておくよ。

Def. 項が次の形である時、neutral であると言う。

項 x(x は変数) と、項 t1t2。つまり　λ 抽象されていない項の事を表す。



reducible な項は、強正規化可能である

有也　まず一段階目。
凛　　型に関する帰納法ですぐに出来るかな？やってみよう。
有也　それも良いんだけど、ちょっと待って。まず次の補題を証明してみて。

lemma
∀ t ∈項 . ∀ t' ∈項 . ∀ τ ∈型 .
(reducible(t,τ) ∧ (t →β* t')) → reducible(t',τ)

凛　　→β* ってなに？
有也　→β* は、β リダクションである →β の反射的推移閉包の事ね。
要するに項 t と項 t' で t →β* t' であるっていうのは、t から t' に何回か β リダクションすれば辿り着けるって事。
注意点は、0 回の β リダクションで辿り着ける時、つまり t' = t の時も成り立つよ。

Def. β リダクション →β の、反射的推移閉包を →β* と書く。

凛　　"β リダクションは reducibility を保存する " って事かな。これはすぐ出来そう。

proof begin
case τ = A(A は基底型) {

凛　　基底型の項が reducible であるということは、定義からその項は強正規化可能であるって事だから、その項を β リダクションして得られる項 t' も強正規化可能で良いよね。

項 t が強正規化可能なので、t' も強正規化可能。

凛　　それから、型 A の項 t を β リダクションして得られた項 t' も型 A をもってるはずだから。

項 t' は型 A を持ち、強正規化可能。
よって t' は型 A で reducible。
}

有也　ちょっと待って。すっ飛ばしてる所があるよ。
凛　　えっ。ナンノコトデスカ。
有也　強正規化可能な項を β リダクションして得られる項が、それまた強正規化可能であることの証明。
それから、型 τ を持つ項を β リダクションして得られる項が、それまた型 τ を持つことの証明。
凛　　先生そんなの自明だよ！！とらすとみ？だよみ？。
有也　ダメ。でもテキストに証明載ってるし、まぁ良い事にしておくかなぁ・・・。
凛　　えへへ？ありがと。
有也　凛はかわいいなぁ。こうやって 1 つ 1 つの命取りが決定的な命取り。
凛　　えッ・・・。や・・・やるよ？証明するよ？
有也　凛は頑張り屋さんだなぁ。
凛　　うー馬鹿にされてる。

課題
forwarding-lemma:
∀ t ∈項 . ∀ t' ∈項 .
SN(t) ∧ (t →β* t') → (SN(t') ∧ (ν(t') < ν(t)))

subject reduction theorem:
∀ Γ ∈文脈 . ∀ t ∈項 . ∀ τ ∈型 .
Γ ⊢t:τ ∧ (t →β* t') → Γ ⊢t':τ

有也　続けて関数型の場合もどうぞ。

case τ = τ1 ⇒ τ2{

凛　　関数型の項 t' が reducible であることをいうのは、定義がああなってるから、任意の型 τ1 の reducible な項 t1 で、tt1 で reducible になるって言えば良いんだよね。

型 τ1 の reducible な項 t1 を仮定する。
この時、t't1 が reducible になる事を言いたい。

凛　　t →β* t' って事は、tt1 →β* t't1 だ。えっと・・・。tt1 は型 τ2 を持つよね。だから、型の構造に関する帰納法が使えて、t't1 が reducible だ！

t →β* t' であるから、
帰納法の仮定より、t't1 は reducible。
t't1 が、任意な reducible な項 t1 で、
reducible であることが示せた。
従って、reducibility の定義から、t' は reducible。
}
proof end

凛　　出来た！！
有也　OK。ちなみに、この補題は Girard の本では CR2 と呼ばれています。

lemma CR2:
∀ t ∈項 . ∀ t' ∈項 . ∀ τ ∈型 .
(reducible(t,τ) ∧ (t →β* t')) → reducible(t',τ)

凛　　てことは CR1 もあるの？
有也　CR1 が、項 t が型 τ で reducible なら、t は強正規化可能である、だよ。

lemma CR1: ∀ t ∈項 . ∀ τ ∈型 .reducible(t,τ) → SN(t)

凛　じゃあ早く CR1 証明しちゃおうよ。
有也 CR2 だけでは CR1 を倒す事は出来んぞ。
凛　・・・なんですって。
有也 CR1 を攻め落とすには、同時に CR3 を攻める必要があるんだ。
凛　でもお高いんでしょう？
有也 ちょっとだけね。CR3 とは。

lemma CR3:
∀ t ∈項 . ∀ τ ∈型 .neutral(t) ∧
(∀ t' ∈項 .(t → βt') → reducible(t',τ)) →
reducible(t,τ)

凛　取りあえずやってみます。

proof begin
case τ = A(A は基底型){
　(CR1){
　　基底型の reducibility の定義から明らか。
　}
　(CR3){

凛　さっき証明した CR2 の逆向きっぽいから簡単に言えそう。

　　t →β t' となる任意の t' が強正規化可能である時に、
　　t が強正規化可能である事を言いたい。

　　t' が強正規化可能であるということは、
　　ν(t') が存在する。
　　"1 ステップの "β リダクションによって、
　　t から得られる項は高々有限。
　　高々有限であるから、
　　得られる項のうちで ν を最大にする項 t'' が存在して、
　　ν(t) = ν(t'')+1 になる。
　　ν(t) が定義されるのだから、
　　明らかに項 t は強正規化可能。
　}
}

凛　τ が基底型の時は言えたね。
有也 CR3 の時の形は、1 つの補題としておこう。

backward-lemma:
∀ t ∈項 .(∀ t' ∈項 .(t →β t') → SN(t')) → SN(t)

case τ = τ1 ⇒ τ2{
　(CR1){
　　型 τ1 ⇒ τ2 を持つ項 t が reducible である。
　　この項が強正規化可能であることを言いたい。

凛　CR3 を使えと言われてるから・・・。CR3 を使う為には neutral な項を作らないといけない。適当な変数を持って来て、(t x) を考えてみよう。

　　型 τ1 の適当な変数 x を持ってくる。

凛　えーと、CR3 に tx を渡すと、tx の β リダクション先が reducible になることを言わないといけないから大変・・・。あれ、でも適当な変数 x は、これ以上 β リダクション出来ないよね。
CR3 は、t から t' に β リダクション出来る時は、t' が絶対に reducible なると言えないといけない、と言ってるんだから、β リダクション出来ない変数 x は、型の構造に関する帰納法で CR3 に関する reducible であると言える！

　　変数 x は CR3 に関し帰納法の仮定から reducible である。
　　t が reducible であるという事は、
　　reducibility の定義から tx も reducible になる。
　　ここで tx の型は τ2 で、CR1 の帰納法の仮定から、
　　tx が強正規化可能であることが言える。

凛　t を変数 x に apply した tx が強正規化可能っていうことは言えた。でもここからどうすれば良いんだろう。えっと、t が強正規化可能って事を言えば良いんだよね・・・。
凛　先生、tx が強正規化可能であると言える時に、t が強正規化可能ってことは言えるのかな？
有也 ちょっと一般化して、

　　lemma1:
　　∀ t1 ∈項 . ∀ t2 ∈項 .SN(t1t2) → SN(t1)

有也 … の形で証明してみて？

　　proof begin
　　t1 →β t1' な t1' を仮定する。
　　すると、t1t2 →β t1't2。
　　forwarding-lemma を用いて、
　　t1't2 は強正規化可能で、
　　また、ν(t1't2) < ν(t1t2)。

　　ν(t1t2) に関する帰納法を使う。
　　ν(t1't2) < ν(t1t2) なので、
　　帰納法の仮定から t1' が強正規化可能。

　　t1 →β t1' を仮定して、
　　t1' が強正規化可能になる事が言えた。
　　よって backward-lemma から t1 が強正規化可能。
　　proof end

　　lemma1 を用いて、CR1 が証明出来た。

}
(CR3){

凛　えーと、項 t が reducible って事を言いたいんだから、型 τ1 で reducible な任意の項 t1 に関して、tt1 が reducible になるって事を言えばいいんだな。

型 τ1 で reducible な項 t1 を仮定して、
tt1 が reducible になることを証明する。
型 τ1 に関して CR1 の帰納法から t1 は強正規化可能。

凛　"t' は reducibility" が与えられてるって事は、多分これを使う証明になるんだよね・・・。ということは、こんな補題を証明出来たら良いのかな？

lemma
∀ t ∈項 . ∀ t' ∈項 . ∀ τ1 ∈型 . ∀ τ2 ∈型 .
neutral(t) ∧ reducible(t1,τ1) ∧ SN(t1) ∧
(∀t' ∈項.(t →β t')→reducible(t',τ1⇒τ2))→
reducible(tt1,τ2)

proof begin
まず、項 t1 は CR 1 より強正規化可能。

tt1 →β t2 で場合分けをする。
t が neutral なので、t = λx.t' の可能性はないから、
tt1 →β t't1 と tt1 →β tt1' だけを考える。

case tt1 →β t't1 のとき{
　つまり t →β t'。
　t' は命題の仮定で reducible だと言っている。
　t' の型は関数型なので、
　reducibility の定義から t't1 は reducible。
}
case tt1 →β tt1' のとき{
　つまり t1 →β t1'。
　t1 は reducible な項であるから、t1' は、
　型 τ1 で CR 2 の帰納法の仮定から reducible。
　forwarding-lemma より、
　SN(t1') で ν(t1') < ν(t1)。
　ν(t1) に関する帰納法を用いる。
　従って帰納法の仮定から tt1' は reducible。
}
tt1 →β t2 な t2 が reducible である事が言えた。
項 tt1 は neutral であるから、
CR3 の帰納法の仮定より tt1 は reducible。
proof end

lemma より、tt1 が reducible になるから、
項 t は reducible である。
}
}
proof end

凛　CR1 と CR3 を同時に証明出来た！！
有也　おめでとう。これで CR1,CR2,CR3 を証明した事になるね。特に CR1 として、reducible なら強正規化可能である、ということが言えたわけだ。
凛　ふー疲れたー。ちょっと休憩しようよー。
有也　そうなー。

休憩 2

凛　こうしてヒントを貰いながらなんとか証明出来てるけど、自分で思いつくのは難しいよ・・・。
有也　強正規化定理の証明は、やっぱり急所は reducibility の定義にあるからね。もちろん他にも証明のバリエーションはあるけど。
凛　オリジナルの証明っていつぐらいのものなの？
有也　少なくとも、Girard の証明のルーツのあたりにあるものは、W.W.Tait の Intensional Interpretations of Functionals of Finite Type I(1967) だと思う。
凛　そんなに昔の証明なんだ。
有也　そのまま型付き λ 計算での強正規化定理を証明しよう、という話では無いんだけれども、すごく近い事をやっている論文だよ。
この段階で既に computability predicate という reducibility と同じ形の述語が導入されてる。
有也　余談なんだけど、reducibility の定義の形をうまくペアノ算術のレベルに翻訳出来ると仮定すると矛盾が導けるので、少なくとも reducibility を使う形ではペアノ算術自身でペアノ算術の強正規化定理を証明する事が出来ないことも言えます。
凛　・・・つまり？
有也　強正規化定理が証明出来ると、まぁ、無矛盾性が証明出来るわけです。
でも出来無い。すると、ペアノ算術自身でペアノ算術の無矛盾性が証明出来ないってこと。つまり？
凛　えーっと、ゲーデルの第二不完全性定理に一応従っているということ？
有也　そゆこと。だったらどうした、って感じなんだけど。ただペアノ算術の無矛盾性が証明出来る事は事実ね。
凛　知ってるだけで全然中身は知らないよ・・・。有也じゃあ取りあえず、今回の証明をきちんと理解すること。そうすると、

Gödel の System T という体系でも同じように強正規化定理が証明出来るようになるから。
そこに更に、ダイアレクティカ解釈を絡めると System T の強正規化定理を持ってペアノ算術の無矛盾性が言えます。

凛　ちんぷんかんぷんー。

有也 あとで本貸してあげるから、すごく頑張って読んでみて。

凛　じゃあ頑張って残りも証明しちゃおう。

有也 残るは、型付け出来るならば reducible であるって事だね。
まずは、次の補題を証明しておこう。

型付け出来る項は、reducible である

abstraction lemma

∀ t1 ∈項 . ∀ τ1 ∈型 . ∀ τ2 ∈型 .
(∀ t ∈項 .reducible(t,τ1)→reducible(t1[x:=t],τ2))→
reducible(λx.t1,τ1 ⇒ τ2)

凛　λx.t1 が reducible って事を言うには、本当は任意の reducible な項 t で (λx.t1)t が reducible になることを言わないといけないんだよね？

有也 reducibility の定義そのものだと、そうなるね。

凛　その３つを言うのに本当に必要なのは t1[x:=t] で reducible になることだけ、ということを言ってるのかな。

有也 答えは証明の中にある！

proof begin

凛　まず補題の仮定から言える事を整理してみよう・・・。

CR1 により、t1[x:=t] が強正規化可能、t も強正規化可能。

凛　t1[x:=t] が強正規化可能ってことは、λx.t1 も強正規化可能なのかな？

有也 なーに証明してみれば分かる！！

凛　うぇーい。

lemma2 : SN(t1[x:=t]) → SN(λx.t1)
proof begin

凛　(λx.t1) をβリダクションして得られるのは λx.t1' しかありえないよね。その後で、backward-lemma を使えば出来るかも。

λx.t1 →β t を仮定する。βリダクションの定義から、
t = λx.t1' の場合だけを考えれば良い。
この時、t1 →β t1'。
ここで、t1[x:=t] →β t1'[x:=t] となる。
forwarding-lemma より、t1[x:=t] は強正規化可能で、
かつ ν(t1'[x:=t]) < ν(t1[x:=t])。
ν(t1[x:=t]) に関する帰納法より、
(λx.t1') は強正規化可能。
λx.t1 →β t な任意の t で t が強正規化可能だと言えたので、
backward-lemma より λx.t1 も強正規化可能。
proof end

lemma2 を用いて、λx.t1 が強正規化可能と言える。

凛　じゃあ続きに戻って・・・。うんと、(λx.t1)t が reducible になることを証明しないといけないかな。

lemma

∀ t1 ∈項 . ∀ t ∈項 . ∀ τ1 ∈型 . ∀ τ2 ∈型 .
reducible(t,τ1) ∧ SN(t) ∧ SN(λx.t1) ∧
(∀ t' ∈項 .reducible(t',τ1)→reducible(t1[x:=t'],τ2))→
reducible((λx.t1)t,τ2)

proof begin

凛　(λx.t1)t →β t2 な t2 はすべて reducible であることを言って、CR3 を使って (λx.t1)t が reducible であると言えそう。

まず項 (λx.t1)t のリダクションで場合分けをする。
case (λx.t1)t →β t1[x:=t] {
　これは補題の仮定そのものであるから明らか
}
case (λx.t1)t →β (λx.t1')t {
　λx.t1 →β λx.t1' なので、
　forwarding-lemma より λx.t1' は強正規化可能で、
　ν(λx.t1') < ν(λx.t1) となる。
　また、t1 →β t1' なので、任意の reducible な項 t' で、
　t1[x:=t'] →β t1'[x:=t'] となるので CR2 から、
　t1'[x:=t'] は型 τ2 で reducible。

　ν(λx.t1) + ν(t) に関する帰納法を使う。
　ν(λx.t1') + ν(t) < ν(λx.t1) + ν(t)
　より帰納法の仮定から (λx.t1')t は reducible。
}
case (λx.t1)t →β (λx.t1)t' {
　t →β t' なので、CR 2 から t' は reducible。
　また forwarding-lemma から、
　t' は強正規化可能で ν(t') < ν(t) となる。
　ν(λx.t1) + ν(t) に関する帰納法を使う。
　ν(λx.t1) + ν(t') < ν(λx.t1) + ν(t) なので、
　帰納法の仮定から、(λx.t1)t' は reducible。
}
さて (λx.t1)t →β t2 な t2 で reducible が言えた。
ここで (λx.t1)t は neutral な項なので、CR3 を用いて、
(λx.t1)t が reducible。
proof end

lemma より、任意の reducible な項 t で (λx.t1)t が reducible。

よって λx.t1 は reducible。
proof end

凛　じゃあこれを使えば、⊢t : τ ならば項 t は型 τ で reducible であるという事を証明出来るのかな？
有也 試しにやってみて。

reduce lemma(?)
∀ t ∈項 . ∀ τ ∈型 . ⊢t:τ → reducible(t,τ)
proof begin
case t = x(x は変数) {
　このような導出は存在しない。なぜならば、文脈が空だから。
}
case t = t1t2, τ = τ2 {
　帰納法の仮定より、項 t1 は型 τ1 ⇒ τ2 で reducible、
　項 t2 は型 τ1 で reducible。
　ところで reducibility の定義より、
　項 t1t2 は型 τ2 で reducible。
}
case t = λx.t', τ = τ1 ⇒ τ2 {
　x : τ1 ⊢t' : τ2
　―――――――――――
　⊢λx.t' : τ1 ⇒ τ2

凛　アレ！？横棒の上で文脈が空じゃないから、帰納法の仮定使えないじゃん！

}
proof fail

凛　うえー。関数型はどうもイヤらしいなぁ。Γ ⊢t:τ で置き換えてみよっと。

reduce lemma(?)
∀ t ∈項 . ∀ τ ∈型 . ∀ Γ ∈文脈 .Γ ⊢t:τ
　→ reducible(t,τ)

proof begin
case t = x(x は変数) {
　この導出が存在する為に、(x:τ) ∈ Γ でなければ成らない。

凛　文脈中の変数が reducible であることは制約付けられない！！どうしたら良いんだろう。

}
proof fail

凛　およよ！！
有也 勝負あったな・・・。じゃあ次の形で証明してみて。

reduce lemma
項 t が x1:τ1,...,xn:τn ⊢t:τ で型付けされるとする。
この時、型 τi を持つ reducible な項 ti(1 <= i <= n) で、
t[x1:=t1,...,xn:=tn] が型 τ を持つ reducible な項となる。
(便宜上、t[x1:=t1,...,xn:=tn] の事を t[X:=Tn] で書きます。)

proof begin
case t = xi(文脈に含まれる変数である時) {
　reducible な項 ti が代入されたのだから、
　当然 t[X:=Tn] は reducible
}
case t = t't'' {
　帰納法の仮定から、
　t'[X:=Tn] と t''[X:=Tn] は reducible。
　したがって、(t'[X:=Tn])(t''[X:=Tn]) は reducible。
　ところで、
　(t'[X:=Tn])(t''[X:=Tn]) = (t't'')[X:=Tn] なので、
　明らか。
}
case t = λx.t'(x は fresh な変数で
　x1...xn や t1...tn と衝突しない), τ = τ1 ⇒ τ2 {
　x1:τ1, ..., xn:τn, x:τ1 ⊢t':τ2
　―――――――――――――――――――
　x1:τ1, ..., xn:τn ⊢λx.t':τ1 ⇒ τ2

　τ1 で reducible な項 t'' を仮定する。
　帰納法の仮定より、項 t'[X:=Tn,x:=t''] は reducible。
　ここで abstraction lemma を用いて、
　λx.(t'[X:=Tn]) は reducible。
　x は fresh variable なので、
　λx.(t'[X:=Tn]) = (λx.t')[X:=Tn]。
　ここで、(λx.t')[X:=Tn] が reducible なので、
　((λx.t')[X:=Tn])t'' が reducible だと言える。

　任意の reducible な項 t'' で、
　((λx.t')[X:=Tn])t'' が reducible だと言えたので、
　(λx.t')[X:=Tn]。
}
proof end

有也 じゃあ次の補題も簡単に言えるね。
reduce theorem
∀ t ∈項 . ∀ τ ∈型 . ⊢t:τ → reducible(t,τ)

proof begin
reduce lemma の文脈が空の時に相当する。
proof end

凛　やった！！

有也 遂に強正規化定理の証明だ。

強正規化定理の証明

strong normalization theorem

∀ t ∈項 . ∀ τ ∈型 . ⊢t:τ → SN(t)

proof begin

reduce theorem より項 t は型 τ で reducible。

CR1 を用いて、項 t は強正規化可能。

proof end

課題： 実は今回、しれっと "∀ t ∈項 . ∀ τ ∈型 . ⊢t:τ → SN(t)" の形で証明しましたが、一般化した、"∀ t ∈項 . ∀ τ ∈型∀ Γ ∈文脈 . Γ ⊢t:τ → SN(t)" も証明出来ます。

この形で強正規化定理を証明してみてください。

（ヒント： reduce-lemma を、"Γ,x1:τ1,...,xn:τn ⊢t:τ で型付けされるとする ..." の形で証明しなおす。）

証明を終えて

凛　出来た！！先生やった！

有也 おめでとう。reduce lemma がちょっと難しかったかな。

凛　ちょっと遠回りすれば良かったんだね。

有也 ちなみに、この強正規化可能定理は、論理関係 (logical relation) を用いた証明の代表的なものでもあるんだよ。

凛　その論理関係っていうのは何なの？

有也 論理関係っていうのは、あるモデルの性質や、モデル間の関係を調べる為に使われるんだけど、モデルの話はまた今度にして、参考になる文献を紹介するだけにとどめておきます。

凛　休憩タイム？

有也 アイスクリーム食べながら TAS 動画を観る仕事があります。

休憩 3

有也 よくこのルートを考えついたよなぁ･･･。さすが TAS さんだ。人間の発想じゃない。

凛　ふと思ったんだけど。自分の証明が正しい物であるかどうかを調べてくれるツールとか無いんですか？

有也 なんでまた。

凛　うーん自分で定理を証明しようと思った時に、自明だーと思ってる事が本当は自明じゃなかったり、もっとみっちり定義して行きたいとか、間違った帰納法の使い方をしてないかーとか。

凛　そういうの、１人でやる時は気になるから、自習する時にも使える何かがあると良いなぁって。

有也 じゃあ定理証明支援系を使ってみると良いかも。ちょうど、型付きλ計算の強正規化定理を、そういう趣きの系のもとで証明したもので、割と読みやすい論文があるよ。

Kevin Donnelly and Hongwei Xi

A Formalization of Strong Normalization for Simply-Typed Lambda-Calculus and System F

有也 この論文は、ATS/LF という系のもとで、型付きλ計算や強正規化可能性の概念を形式化して、ATS/LF の上で強正規化定理の証明をするというものだよ。

凛　何が嬉しいの？

有也 そりゃ当然、証明全体の緻密さが格段に上がることかな。もちろん、その為には定義をちゃんとしなければいけないし、証明も事細かにすることになる。

凛　うーん・・・大変。定理証明支援系って、ATS/LF の他にどんなのがあるの？

有也 Isabelle/HOL とか、Coq とか、Twelf かな・・・。

でも今回は Agda2 を使って、証明までは行かないけど、型付けλ計算を形式化するところまではやってみます。ところで、型付けλ計算をどう証明支援系の中で扱えば良いと思う？

凛　ようするに、インタプリタを書く時の感じでやれば良いんでしょ？

```
data Term = Var String
          | App Term Term
          | Lam String Term
```

凛　って定義して、今回は eval 関数を書く必要はなくて、なんだろう、自由変数に対する代入が定義出来れば良いのかな？

有也 それだけ？

凛　いやえっと・・・。α同値変換とかも必要。

有也 うん。でも証明する時には、α同値変換に関わらず定義や定理が成り立つ、事も証明しないといけなくなっちゃう。

凛　あ・・・そっか・・・。ちゃんとやるって事は、そこまで考えないといけないんだね・・・。

有也 こういう形で表現する事を、FOAS(First Order Abstract Syntax) と呼びます。

有也 所で、インタプリタを実装する言語自身は、上に挙げた問題点をすでに解決しているわけ。

ということは、形式化の対象となる言語 (object-language) を埋め込む先の言語 (meta-language) の持つ機能を出来るだけ利用して表現したい。そういう表現の仕方を、HOAS(Higher Order Abstract Syntax) と呼びます。

有也 具体的には object-language の変数は、meta-language の変数に対応して、object-language の代入は、meta-language の apply で表現出来る、という感じ。その考えを元にすると、

```
data Term = App Term Term
          | Lam (Term -> Term)
```

有也 …と書けるよ。

凛　うーんと、object-language での代入 t[x:=t] は、meta-language での (λx.t)y になるのかな？

有也 そうそう。取りあえず定義をしてみよう。

有也 Komike.agda というファイルで、

```
{-# OPTIONS --no-positivity-check #-}
module Komike where

open import Data.Nat -- 自然数を使う為の呪文

data Term : Set0 where
    App : Term → Term → Term
    Lam : (Term → Term) → Term

data Tm : Set0 where
    TMcst : Tm
    TMlam : (Tm → Tm) → Tm
    TMapp : Tm → Tm → Tm

data RED : Tm → Tm → N → Set0 where
    REDlam :
        ∀{f}{g}{s} →
            (∀{x} → RED (f x) (g x) s) →
            RED (TMlam f) (TMlam g) (suc s)
    REDapp1 :
        ∀{t1}{t1'}{t2}{s} → RED t1 t1' s →
          RED (TMapp t1 t2) (TMapp t1' t2) (suc s)
    REDapp2 :
        ∀{t1}{t2}{t2'}{s} → RED t2 t2' s →
          RED (TMapp t1 t2) (TMapp t1 t2') (suc s)
    REDapp3 :
        ∀{t}{f} → RED (TMapp (TMlam f) t) (f t) 0
```

有也 としよう。REDapp1 と REDapp2 は、型付けλ計算のβリダクションの定義そのままの形なので問題無いね。だけれども、REDlam と REDapp3 はちょっと違うので詳しくみてみよう。

有也 まず、REDapp3 の元の定義は、

$$\frac{}{(\lambda x.t1)\ t2 \rightarrow_\beta t1[x:=t2]}$$

有也 …です。これを見ると、object-language での (λx.t1)t2 が meta-language の (TMapp (TMlam f) t) に対応。

object-language での t1[x:=t2] が meta-language の (f t) に対応していることが分かります。

object-language の代入を、meta-language の通常の apply で表す為に HOAS を入れたので、これで良さそうですね。

有也 次に、REDlam の元の定義は、

$$\frac{t \rightarrow_\beta t'}{\lambda x.t \rightarrow_\beta \lambda x.t'}$$

有也 object-language での λx.t は、meta-language での TMlam λx.t となります。

この λx.t の t の部分を取り出して考えたい所だけど、例えば、Agda2 での (TMlam λx.x) を考えると、λx.x の body 中の x だけ取り出す事は出来ない。

そこで、2つの関数 f,g を任意の項 t に apply した結果の (f t) と (g t) にリダクションの関係が成り立つということで言い換える。

どんな項を自由変数 x に代入してもリダクションの関係が成り立つということは、元々言いたかった項 t と項 t' でβリダクションが出来るという事と言い換えられます。

よってこの定義で妥当であると言えます。

有也 こんな感じで HOAS に沿った定義を進めていくと、ちゃんと証明出来るよ、という話が上の論文に書いてます。

凛　なるほど全然分からないです。

有也 どんな感じか紹介してみただけだから！

凛　私は Coq を使う。

有也 それが良いかも。ただそこで上に書いた呪いの no-positivity-check が効いて来るよ。

凛　え？

有也 Term の定義、特に TMlam を見てもらえば分かるけど、これは negative に Tm が出現してしまっているので、Coq で許される帰納的な定義にはなっていないです。

有也 negative に出現する (negative occurrence) についてはググってください。これが出来ちゃうと何がまずいかを示す方が良いね！例えば、

```
{-# OPTIONS --no-positivity-check #-}
module Term where

data Term : Set0 where
    Lam : (Term → Term) → Term

app : Term → Term → Term
app (Lam f) x = f x

omega : Term
omega = app (Lam (λ x → app x x))
            (Lam (λ x → app x x))
```

有也 omega は、型無しλ計算で良く知られてる (λx.xx) (λx.xx) と同じだよね。omega から omega が出て来て omegaomegaomegaomega...。これはオメガ大変。それから、

```
{-# OPTIONS --no-positivity-check #-}
```

```
module False where

data ⊥ : Set0 where

data F : Set0 where
    f : (F → ⊥) → F

p : F → ⊥
p (f b) = b (f b)

x : ⊥
x = p (f p)
```

凛　ひー。

有也 なんだけれども、上の形式化の段階ではそのように Term を使って定義しちゃってるんだ。
ところで、Twelf ではメタな定理 (object-language に対する、meta-language レベルでの定理) に対して、型検査以外にも証明に対する検査が走るので、Twelf でやるのもアリかも。

凛　一長一短な気がする・・・。何か他に方法はないの？

有也 FOAS や HOAS 以外の方法で abstract syntax を定義する方法も考えられているし、色々と都合の良いように仮定して FOAS で進めるっていう方法もあると思うけど、取りあえず夏休みを潰して挑戦してみるのが良いと思う！

凛　えー・・・。ゲームしたいよ・・・。DS で定理証明支援系って、出ないのかなぁ。

終りに

有也 ゲームって、何の？

凛　メモリーズオフゆびきりの記憶と、ファイアーエンブレム新・紋章の謎。

有也 DS とセットでファイアーエンブレム貸してよ。

凛　先生はゲームで現実を蔑ろにするからダメだよ・・・。でも今作は難易度にルナティックっていうのがあるらしくて、その攻略は見てみたいな。

有也 そんな面白そうなものが！！勝手に借りて行きます。

凛　だ・・・ダメッ！！もっと夏休みらしく現実を・・・。

有也 実はねー StarCraft2 っていうゲームがあってねー。

凛　それってどこが作ってるゲーム？

有也 Blizzard。

凛　ダメ！！大学も死んじゃうよ！！安心の Blizzard だよ！？

有也 Blizzard ならちかたないね。

凛　どうしたら良いの・・・。先生がダメに成ってしまう。

有也 じゃあファイアーエンブレム持って明日から旅行に行きます。一石二鳥で良かったー。

凛　旅行ってどこに行くの？

有也 関東の海岸線のあたりを歩いて攻めます。

凛　そのなんとか旅行には着いて行く事になりました。今決めました。ダーク♀監視員です。

有也 まぁいっか。1 人よりも楽しいし。その代わり歩きながら証明してもらいます。

凛　あ！今度は Church-Rosser の定理が良いな。それから・・・。

終りに 2

どうしてこのような記事を書いてしまったのかを勝手に説明したいと思います。もともとは、Thorsten Altenkirch and Andreas Abel による "A Predicative Strong Normalisation Proof for a λ-calculus with Interleaving Inductive Types" の論文の紹介、をする為の、Impredicative な証明をする為の、普通に System F の強正規化定理の証明あたりからはじめようと思った。というものがありました。

そのような日本語の文章をまとめあげるには、何か機会があった方が良いかと思っていたので、折角のお誘いもあり、今回はじめて同人誌の原稿を書こうとも思ったのでした、のですが、いざ書いてみると余りにも雲行きが怪しく成って来たので、更にその手前、単純型付け λ 計算の強正規化定理の証明にしかならなかった、という感じです。

この記事では、どこから出て来たのか分からない定義に少しでも納得の出来そうな説明を付ける事と、出来るだけ細かく証明は書こう、と思った事がありました。

その結果として証明の途中に、主に凛ちゃんの考えてる事が何故か口から漏れ出して、文章となって表れています。

ごちゃごちゃして分かりにくくなった気も勿論しますが、すっきりした証明は幾らでもあるのでそこはお許しください！という事で。

それから、作中で挙げた Agda2 版の強正規化定理の証明を、http://github.com/ranha に上げるかもしれないので興味のある人はのぞいてみてください。と同時に、Agda2 の適当な入門でも書ければ良いのですが・・・。でなければ、解読不可能ですし。

最後に参考文献リストを書いてさよならにしたいと思います。
さようなら！

参考文献

Proofs and Types
J.Y.Girard Y.Lafont P.Taylor
Published by Cambridge University Press 1989
http://www.paultaylor.eu/stable/Proofs+Types.html
本記事の証明はこれに倣っています。

A Formalization of Strong Normalization for Simply-Typed Lambda-Calculus and System F
Kevin Donnelly Hongwei Xi
LMFTP 2006
http://www.ats-lang.org/PAPER/paper.html
本記事の証明 (の一部) はこれに倣っています。

プログラム意味論
横内寛文
共立出版 情報数学講座 7 巻 1994
2 章が「ラムダ計算の基礎」となっており、その後は表示的意味論を展開している。

プログラミング言語の基礎理論
大堀淳
共立出版 情報数学講座 9 巻 1997
名前しか出せなかったモデルの話と論理関係について、日本語で説明している。

計算論 計算可能性とラムダ計算
高橋正子
近代科学社 コンピュータサイエンス大学講座 24 1991
型付きλ計算でなく、型無しのλ計算とλ計算のモデルを扱っている。現在でも入手可能な良書。

Type and Programming Languages
Benjamin C. Pierce
The MIT Press 2002
単純型付けλ計算にフィットした形で、強正規化定理を証明している。

Lambda-Calculus and Combinators: An Introduction
Cambridge University Press 2 版 2008
J. Roger Hindley Jonathan P. Seldin

Introduction to Combinators and λ -Calculus
(Cambridge University Press の London Mathematical Society Student Texts 1 1986 の 2 版)
読みやすいフォントになった。一通りの話題を取り扱っていて章立ても 2 版で読みやすい順番になった。
個人的に "Care of your pet combinator" が初音ミクさんで歌われないかなーというのがあるので、初音ミクさんマスターの方々は是非実現してください！

Foundations for Programming Languages
John C. Mitchell
The MIT Press 1996
論理関係について、1 章割かれて解説されている。その他様々な話題が丁寧に解説されている本。

ゲーデルと 20 世紀の論理学
田中一之 [編]
東京大学出版会 2007
ダイアレクティカ解釈についての詳しい説明がある。

証明論入門
竹内外史 八杉満利子
共立出版株式会社 2010
最近復刊された。こちらにもダイアレクティカ解釈についての説明がある。また、PA の無矛盾性証明にも 1 章割かれている。

Haskell コミュニティ探訪 - 処理系とライブラリを中心にして -

著 : shelarcy

Haskell コミュニティと聞いて，最初に思い浮かべるものは何でしょうか？ Haskell.org のような Haskell に関する情報を集約したサイトでしょうか？それとも Haskell-Cafe のようなメーリング・リストでしょうか？こうしたサイトやメーリング・リストは確かに Haskell コミュニティの中でも大きく，重要な部分です。ですが，Haskell についてより深く知ろうとするならこうした大きなコミュニティだけではなく，より小さく細かなコミュニティについても知っておく必要があるでしょう。

そこでこの記事では，主要な処理系とライブラリを中心とした開発コミュニティについて紹介していくことにします。この記事が読者のみなさんにとって，より多くの Haskell コミュニティに目を向けていくきっかけとなるなら，これ幸いです。

GHC

2010 年 8 月現在，Haskell コミュニティで広く使われている処理系は GHC です。現在の Haskell コミュニティの大部分は，この GHC に周りに形成されていると言っても過言ではないでしょう。そこでまずはこの GHC，及び GHC に付属のツールやライブラリを開発しているコミュニティから見ていくことにしましょう。

GHC の開発の中心人物は Simon Peyton Jones，Simon Marlow，Ian Lynagh の三人です。この内 Simon Peyton Jones と Simon Marlow は Microsoft Research の研究員です。Simon Peyton Jones は型システムや最適化機能など，主にコンパイラ周りの開発を行っています。Simon Marlow は実行時システム（RunTime System）やライブラリなど，主に実行時に行う処理回りの開発を行っています。

Ian Lynagh は，GHC の開発を行うという立場で Microsoft Research（正確には Simon Peyton Jones と Simon Marlow の二人）から仕事を請け負っている開発エンジニアです。Ian Lynagh は Simon Peyton Jones と Simon Marlow の二人を補佐するため，GHC 本体とライブラリのバグ修正，Makefile などのビルドスクリプトの整備，GHC 用の自動テストツールである Builder の開発，Trac やメーリング・リストでのユーザーのやり取りなど，開発に関わる作業を全般的に行っています。

この三人の他に，目的ごと，機能開発プロジェクトごとの，開発メンバーがいます。

GHC の開発は Haskell.org で行われています。Haskell.org の http://darcs.haskell.org に，GHC と関連ツール，そして http://darcs.haskell.org の packages ディレクトリ配下にライブラリの repository があり，そこで darcs というバージョン管理システムを使ってこれらの開発を行っています。また http://hackage.haskell.org/trac/ghc に設置された Trac で，GHC 開発や内部実装に関する情報を集積したり，GHC のバグや将来実装すべき機能などに関する情報を Ticket という形で管理しています。また Simon Peyton Jones は Trac 上に Status/SLPJ-Tickets というページをつくり，Ticket の中から今後の課題として気になるものを収集し，機能ごとの課題としてこのページにまとめています。

Haskell.org では他に GHC に関係するメーリング・リストも提供しています。Glasgow-haskell-users は文字通り，GHC のユーザーが使用上の疑問をぶつけたり，GHC の挙動について議論を行ったりするためのメーリング・リストです。Glasgow-haskell-bugs はバグ報告の目的に使われるメーリング・リストで，Trac に登録した Ticket や Ticket に対する変更が自動的に報告されるように設定されています。

cvs-ghc は，主に ghc の repository への patch のコミット・ログと，Builder などの自動テストツールを使ってのテスト結果を流すためのメーリング・リストです。偶にこれから入れようとする変更について議論を行うこともあります。また，開発者以外が GHC 本体に対する patch を送るための場でもあります。同様に cvs-libraries には GHC に関連するライブラリの repository へのコミット・ログが，cvs-others には GHC に関連するツールの repository へのコミット・ログが流れます。

hoopl

Simon Peyton Jones は Tufts 大学の Norman Ramsey と João Dias の二人と共に，Tufts 大学の git の repository を使って hoopl（Higher-Order OPtimization Library）という名前のライブラリを開発しています。また，darcs を使った git repository のミラーも存在します。残念ながらこれらの repository にはアクセス制限がかかっているため，開発者でない外部の人間が気軽に開発版を入手することはできません。ですが，開発が一区切りするたびに HackageDB に hoopl の新しいバージョンがアップロードされるので，hoopl の動向が気になる方はそちらをチェックすると良いでしょう。

hoopl は GHC のコード生成器の一部として開発されていた，（GHC の中間言語の一つとして使われている抽象アセンブリ言語である）C-- の制御フローグラフを最適化するためのデータフロー解析エン

ジンを分離独立させたものです。残念ながら現在の GHC のコード生成器では hoopl を利用していませんが，近い将来 hoopl を使った実装へと変更される予定です。（開発が順調に進めば，この記事を読む頃には既に hoopl を使った実装になっているかもしれません。）

hoopl の開発目的の一つは，泥臭い実装細部を抽象化し，解析の方針を示した簡単なコードだけで様々なデータフロー解析の実現を可能にすることです。hoopl によって簡単にデータフロー解析を実装できるようになれば，様々な種類のデータフロー解析を GHC に加えることが容易になるからです。また，簡単にデータフロー解析を実装できることで，大学の教材として使えるという副次的な効果もあったようです。

hoopl の開発者のうち Norman Ramsey と João Dias の二人は，コンパイラの研究，特に抽象アセンブリ言語である C-- を使ったマシンコードに比較的近い領域での最適化に関する研究をしています。なので，hoopl の開発には彼らの研究のための新しいツールの作成という側面もあるのでしょう。

hoopl の開発は GHC 本体とは独立して行われているため，最新の開発版を GHC の利用する packages/hoopl repository には直接ミラーを置かずに，実際に GHC のコード生成器の実装に使用するバージョンが packages/hoopl repository に置くようにしています。（残念ながら執筆時点では少しバージョンがずれているようですが，GHC の実装に hoopl が使われるようになり次第，packages/hoopl repository の hoopl は GHC に対応したものとなるでしょう。）

もちろん，hoopl が独立して開発されていることから，最新版の hoopl を GHC で使用するために，GHC の当該部分のみならず hoopl の側も弄らなければならないこともあるでしょう。そうした場合，コード生成器を弄る側の開発者が，一時的に hoopl の開発に加わって当該部分を変更する作業を行うことになっています。

データ並列 Haskell チーム（DPH: Data Parallel Haskell）

DPH チームは，GHC 上にデータ並列と呼ばれる形の並列処理を行う機能，およびライブラリを構築することを目的としたチームです。DPH チームの中心メンバーは New South Wales 大学に所属する Manuel Chakravarty，Roman Leshchinskiy，Gabriele Keller，Ben Lippmeier の四人です。

他に Simon Peyton Jones と Simon Marlow の二人が補助的な役割を果たします。二人が開発している GHC の最適化や並列処理関係の機能などの多くは，DPH チームにとっても欠かすことのできないものです。また，DPH チームの要望に応じて二人が GHC に機能を付け加えるなど，DPH チームにとってより直接的な支援を行うこともあります。二人はこのようにして，直接的あるいは間接的な形で DPH チームを支援します。

DPH チームは，データ並列処理を行うための専用の配列を提供する，複数のライブラリを作成しています。その一つが，チームの名前にもなっている DPH ライブラリです。DPH ライブラリの作成には DPH チーム全員が関わっていますが，この中で中心的な役割を果たしているのは Manuel Chakravarty と Roman Leshchinskiy の二人でしょう。

DPH ライブラリはただのライブラリではなく，GHC の機能の一として開発されています。その理由の一つとしては，DPH ライブラリでは，マルチコアでの並列処理を実現するライブラリ部分だけではなく，GHC の助けを借りてリスト風の構文を提供するフロントエンドを持つことが挙げられるでしょう。結果，ライブラリの開発はフロントエンドを提供する GHC の repository と，ライブラリを提供する packages/dph の darcs の repository，この両方を使って行われています。このように DPH ライブラリと GHC は密接な関係にあります。そのため，DPH についてやり取りするメーリング・リストとしては GHC のメーリング・リストを使いますし，DPH のバグなどについて報告する場合には GHC の Trac を使うことになります。

もう一つは，行列演算を行うことを目的とした repa（REgular PArallel arrays）です。repa は DPH から派生したライブラリで，DPH ライブラリのバックエンド部分を実装に利用しています。repa では今のところ GHC の助けを借りた構文提供は行なっていないため，repa の開発はライブラリ単体で行われます。repa は Gabriele Keller が主に開発し，Ben Lippmeier がこのライブラリのメンテナンス及びサンプルプログラムの作成を行っています。このメンテナンスと開発を行う darcs の repository は http://code.haskell.org にあります。repa には専用の Trac が用意されていて，repa のバグ報告はこの専用の Trac で行います。

そしてもう一つ，様々な手段で配列処理を高速化することを目的とした accelerate というライブラリがあります。accelerate では配列処理のための共通の API と，その API を使って配列処理を行うためのバックエンドを提供します。accelerate の API は，Gabriele Keller による repa の API デザインを参考に，Manuel Chakravarty が中心となって開発を行っています。（ただし，開発初期の古い repa の API を参考にしていたことや機能不足などを理由に，今の accelerate の API は repa のものとは多少異なったものとなっています。）

accelerate のバックエンドとしては，今のところ Haskell の配列を使った参考実装と，CUDA を使った GPGPU 用のバックエンドが提供されています。CUDA バックエンドの開発は，New South Wales 大学の大学院生である Sean Lee と Trevor McDonell の二人が中心となって行っています。また，LLVM を使ったバックエンドの開発も行われているようです。

accelerate の開発には http://code.haskell.org にある darcs の repository を使います。accelerate には専用のメーリング・リストも用意されていて，accelerate に関する質問や議論はこちらのメーリングリストで受け付けています。また専用の Trac も提供されていて，accelerate に対するバグ報告はそちらの Trac 上で行うようになっています。

型システムの開発

型システムの開発には，今のところ Simon Peyton Jones を除きいつも中心となる人物は存在しません。関連する部分で一緒に仕事をすることはありますが，基本的にはプロジェクトごとに研究者や学生が関わるという形になっています。現行のプロジェクトをいくつか例として挙げてみましょう。

Simon Peyton Jones は（現在は）同じ Microsoft Research に属する Dimitrios Vytiniotis と共に，GHC の型システムの拡張や，既存の型システムを置き換える新しい型推論の仕組みなどの提案を行っています。提案した型システムのうちいくつかは，GHC に既に実装されています。また，GHC に実装されていない型システムのうちいくつかについては，Dimitrios Vytiniotis が自分のページで論文に関連するファイルとして簡単な実装を公開しています。

Simon Peyton Jones と Dimitrios Vytiniotis の二人は，最近の論文で OutsideIn というアルゴリズムを使った新しい型推論の仕組みをいくつか提案していました。2010 年 8 月現在，二人はこの OutsideIn アルゴリズムを用い，これまで一枚岩に実装されていた型検査器をモジュール化された実装へと整理し直すことを目的とした，GHC の型システムの書き換え作業を行っています。新しい型システムの開発によって他の開発作業が妨げられないよう，この作業は現在 ghc-new-tc などの独立した branch で行われています。ですが，GHC6.14.1 のリリース前には新しい型システムを完成させ，ghc の開発 branch に変更を反映する予定となっています。

さて，この新しい型システムの実装には，二人の他に，もう一人 Pennsylvania 大学の大学院生である Brent Yorgey が関わっています。彼は新しい型システムの最初の実装に関わった経験を元に，種（kind）に対する拡張を GHC に提供しようとしています。この種に対する拡張は，she（The Strathclyde Haskell Enhancement ）などに着想を得たもので，型族（Type Families）を使った型レベル・プログラミングで重要な役割を果たす予定です。

もちろん，GHC の型システムに関わっているのは，この三人だけではありません。DPH チームの Manuel Chakravarty は，Simon Peyton Jones と共に型族の開発に関わっています。ただ現行の GHC6.12.x の型族の実装には，いくつものバグや他の言語機能と連携する上での大きな制限があることから，現在型族に対する新機能の追加を一時中断してバグや機能制限などの問題を解決に主軸を移した開発を行っています。先の段落で書いた OutsideIn を使った新しい型システムでは，こうした問題のうちいくつかが解決される予定です。また新しい型システムで解決するものの他にも，型族と他の言語機能を適切に連携させるためのアイデアが既に出てきていることから，そう遠くない将来，型族への機能追加に再び乗り出すことができるようになるでしょう。

GHC のライブラリ

GHC に付属するライブラリの多くは，GHC の Trac や libraries という名前のメーリング・リストを使って開発や議論などを行います。GHC に付属するライブラリにバグがあれば，Trac に Ticket として登録します。また，GHC に付属するライブラリに疑問があれば，libraries リストにメールを送って尋ねます。

GHC に付属するライブラリに変更を加えたい場合，darcs を使って patch を作成した後，Proposal として Trac の Ticket に登録し，libraries リストで変更を提案として呼びかけるのがマナーになっています。変更を提案してから二週間後，変更に対する議論が続いていなければ，これまでの議論の概要を Ticket にまとめます。そして変更に対して大体の合意が取れているなら，patch の修正すべき点を修正し，ライブラリのメンテナーにコミットして貰います。この説明はあくまで大雑把なものです。より詳しく知りたければ，HaskellWiki の Library submissions のページを参照してください。

これが GHC に付属するライブラリの開発の基本的なやり方ですが，例外もいくつかあります。例えば DPH ライブラリはまだ未完成なため，こうした手続きを経ずに独自に開発を行っています。また，Cabal のようにプロジェクト固有の Trac やメーリングリストがある場合には，そちらを使って開発や議論などを行います。

少し特殊な例としては，base パッケージが挙げられます。base パッケージの開発は他のライブラリとは違い，様々な問題に柔軟に対応できるようかなり臨機応変な開発体制が採られています。GHC の開発者が新機能を提供する際，base パッケージの内部的な実装な実装は議論を経ずに変更され，GHC.* モジュールに新しい関数や型などが追加されます。ですがこの場合，GHC.* 以外のユーザーが使用するべきモジュールで関数や型などを公開する時には，他のライブラリと同様の変更手続きに従って変更を行うようにしています。また，Trac やメーリング・リストで既存の機能に対する議論が行われ，現在の機能を変更するのに十分なものが出てきた場合には，通常の変更手続きを経ずに直接インターフェースを含めた base パッケージに対する変更を行うこともあります。

Hackage

2010 年 8 月現在，Haskell で書かれた様々なソフトウェアやライブラリはパッケージとして HackageDB に登録され，cabal コマンドを使って HackageDB にあるパッケージをダウンロードして使用するようになっています。今や HackageDB は，Haskell コミュニティの大きな部分を占めるものだと言っても過言ではないでしょう。

この HackageDB を中心とするコミュニティの形成を目指したのが，Hackage というプロジェクトです。Hackage ではコミュニティ形成のため，cabal コマンド（cabal-install パッケージ）のバックエンドである Cabal や cabal コマンドの開発，それと HackageDB の構築などを行ってきました。もちろん，これで全ての開発が終了したわけではありません。Cabal や cabal コマンドにはまだまだ不満と言える部分が残っているからです。この不満の解消が Hackage における現在の目的です。

cabal コマンドや Cabal の開発は，Well-Typed 社の Duncan Coutts が中心となって進めています。Well-Typed 社は Haskell を使って仕事をする会社で，Ian Lynagh と Duncan Coutts の二人が共同出資者となって経営しています。

話を戻しましょう。Hackage では http://hackage.haskell.org/trac/hackage/ に専用の Trac を用意しています。Cabal や cabal コマンドに将来実装するべき機能のアイデアをまとめたり，Cabal や cabal コマンドのバグを管理するのには，この Trac を使用します。

また Cabal や cabal コマンドの開発に関する議論を行う場として，cabal-devel というメーリング・リストが提供されています。cabal-devel リストでは開発に関する議論を行う他，Hackage の専用 Trac に登録された Ticket や Ticket に対する変更が送られてきます。また，Cabal や cabal コマンドに対する patch を送るのも cabal-devel リストになります。このように cabal-devel リストには Cabal に関する様々な情報が送られてきますが，それでも cabal-devel リストはあくまで開発者用のメーリング・リストという位置づけです。Cabal に対する質問は libraries リストの方に送る必要があります。

なお，Cabal や cabal コマンドは GHC のような特定の Haskell 処理系にだけ依存したものではありません。Cabal では GHC の他，Hugs，JHC，LHC，NHC（，開発版ではさらに UHC）に対応しています。また，オプションを利用することで，どの処理系に付属する Cabal か関係なく，Cabal の対応する好きな処理系をビルドやインストールに使用することができます。

Haskell Platform

Haskell Platform は，GHC にライブラリや cabal コマンドなどのツールを加え，Haskell を使ってプログラミングするのに必要な環境一式を提供するものです。Haskell Platform の現在のバージョンである 2010.2.0.0 では，Haskell Platform によって追加提供されるライブラリはごく狭い範囲に限られています。Haskell Platform 2010.2.0.0 では，かつて GHC と一緒に提供されていたライブラリと，cabal コマンドをビルドするのに必要なライブラリぐらいしか，Haskell Platform には付属していません。ですが，Haskell Platform の次のリリースでは，HackageDB に登録されている中で重要度の高いライブラリがいくつかが Haskell Platform に付属するライブラリとして追加される予定になっています。

Haskell Platform では処理系として GHC を提供しますが，Haskell Platform の主体は GHC とは別になっています。Haskell Platform の中心人物は Duncan Coutts と Galois 社の Don Stewart の二人です。Don Stewart の勤める Galois 社も Haskell などの関数型言語を使って仕事をする会社です。

話を戻しましょう。Haskell Platform では http://trac.haskell.org/haskell-platform/ に専用の Trac を用意しています。また，Haskell Platform に対する質問や議論を行うための haskell-platform というメーリング・リストも提供されています。

Trac の AddingPackages のページには，Haskell Platform に新しくライブラリを追加するための手続きが載っています。しかし，Haskell Platform の提供するライブラリがまだまだ少ないことや Haskell Platform の正式リリースが数ヶ月前に行われたばかりということもあって，ライブラリ追加の手続きに積極的に関わっていこうとする人間は少なく，この手続きはまだ機能していないようです。Haskell Platform の周りにあまり人がいない今暫くの間は，この状況が続くでしょう。ですが時間が経ち，GHC ではなく Haskell Platform を Haskell プログラミングのための環境だと考える人が増え，haskell-platfrom リストなどの Haskell Platform のコミュニティに参加する人が増えてくれば，やがてこの状況は変わっていくでしょう。

Google Summer of Code(GSoC)

Haskell コミュニティでは，毎年開催されている GSoC を GHC やライブラリ等の開発を進める機会としてうまく活用しています。GSoC で行われた Haskell.org のプロジェクトのうち，何割かは確実に成果として Haskell コミュニティに取り込まれていきます。

Haskell コミュニティが GSoC の成果を取り込めていける理由の一

つは，プロジェクトの選定基準にあります。Haskell コミュニティでは，新しくライブラリやソフトウェアを作るものよりも既存のライブラリやソフトウェアを改良するものを重視してプロジェクトを選びます。これにより，GSoC から何も目が出ずに終ることをある程度まで防いでいます。

さらに，GSoC で行うプロジェクトが GHC に対する改良である場合，GSoC 終了後に Microsoft Research のインターンに来ることを要求し，このインターンを GSoC とセットで利用します。GSoC の続きを Microsoft Research のインターンで行うことで，プロジェクトを一夏限りのものとせず，GSoC 後も継続してプロジェクトに取り組める体制を提供するというわけです。また，Microsoft Research にインターンとして呼び，直接会って話すことで，メールや IRC などでは埋め難いコミュニケーション上のギャップを埋めるという意図もあると思います。（もちろん，GSoC とセットになったインターンだけでなく，インターンだけで来た人も GHC の開発に利用しています。）

Utrecht 大学

Utrecht 大学では，他の Haskell コミュニティとは少し毛色の違う独自のコミュニティを形成しています。従って，Haskell コミュニティについて語るなら，Utrecht 大学の Haskell コミュニティについて言及することを避けることはできないでしょう。

Utrecht 大学では，GHC や Haskell Platform に付属しているものとは違うライブラリをいくつか使用します。Utrecht 大学で開発されたライブラリを集めた uulib というのがその一つで，uulib では独自のプリティプリンタやパーサー・コンビネータなどを提供します。また，uulib にあるパーサー・コンビネータの新バージョンとして，uu-parsinglib というライブラリが別途提供されています。

また，uuagc（Utrecht University Attribute Grammar Compiler）という名前の属性文法を扱うためのプリプロセッサを開発し，Haskell プログラミングのためのツールとして広く利用しています。

Utrecht 大学では，他に UHC（Utrecht Haskell Compiler）という Haskell 処理系を開発しています。UHC の開発チームは uu-parsinglib の作者である Doaitse Swierstra や uuagc の作者である Arie Middelkoop，他二人を中心とするメンバーで，属性文法を使った uuagc に大きく依存した実装となっています。UHC には uhc-developers という開発者用のメーリング・リストと，uhc-users というユーザー用のメーリング・リストが用意されています。今のところ uhc-users リストではほとんどメールは行き交っていませんが，開発計画にあるように UHC から Cabal を利用できるようになったり，cabal コマンドを使って直接 UHC をインストールできるようになれば，少しは流量が増えるでしょう。なぜなら，UHC にはいくつか魅力的な独自の拡張機能があるので，環境さえ整備されればそうした部分に魅力を感じて使い始める人もいると思われるからです。

なお，UHC に対するバグ報告には Google Code を使います。Utrecht 大学にシステムが用意されているわけでも，Haskell.org に Trac が用意されているわけでもないので注意してください。

差分のアルゴリズム

著 : cubicdaiya

はじめに、この文章は主に差分アルゴリズムのちょっとした応用について述べたものである。差分を計算するアルゴリズムについては昔から研究され、資料も豊富だが、それの応用について述べたものについてはあまり見かけることはない。そこで、筆者が以前実装した diff のライブラリ dtl での経験を元にそういった差分アルゴリズムの応用について書いてみるというのがこの文章主旨である。なので、ここでは差分アルゴリズムの簡単なおさらいは行うものの、差分アルゴリズムの詳細についてはあまり詳しく述べない (知りたい方は参考文献を参照してほしい)。

差分アルゴリズム。一度やってみるとわかるが、何も考えずに差分を求めるようなプログラムを書こうとしてもまずうまくいかないだろう。もし、動的計画法さえ全く知らないのにうまくいったのならあなたはとても頭がいい。そしてセンスも。うまくいかなかった人は先人の知恵を借りることにしよう。一応言うと私は自分でやろうてしてさっぱりうまくいかず、そうした。

まず、差分を計算するというのは次の 3 つを計算することと同義である (文字列「acbdeacbed 」と「acebdabbabed」での例を示す)。

- 編集距離 … 二つの要素列の違いを数値化したもの (6)。
- LCS … 最長共通部分列 (acbdabed)。
- SES … ある要素列を別の要素列に変換するための最短手順 (C a, C c, A e, C b, C d, D e, C a, D c, C b, A b, A a, A b, C e, C d)。

(※)「A」は追加、「D」は削除、「C」は共通を表す。

編集距離は「追加」と「削除」の合計値を指している。LCS はその名の通り、二つの要素列の連続する共通部分のうち最も長い列を指し、
この場合は「acbdabed」である (ただし、LCS は一つとは限らない)。これは SES の「C」で表される部分をつなぎ合わせたものである。

これらの値は動的計画法を使って二つの要素列を元にしたグラフ (エディットグラフ) から各々の部分列毎に算出していくことで計算することが可能だが、動的計画法は次の理由からあまり好ましくない。

- 計算量が O(MN) (M, N は各要素列の長さ)
- エディットグラフの表を作るのに大量のメモリが必要 (M 行 N 列または N 行 M 列)

M と N の取り得る値があらかじめ決まっている場合はともかく、バージョン管理システムなどではこれらの値は任意であるため、場合によっては深刻なメモリ不足に陥る危険性がある。そのためか、実用的に使われているような diff プログラムやバージョン管理システムでは動的計画法は使われていない。それらはもっと効率的なアルゴリズムを使っている。以下は有名なバージョン管理システム (Monotone は知らない人もいるかもしれない) で採用されている差分アルゴリズムである。ただし、そのまま採用されているわけではなく、最適化がなされている場合が多い。特に Git で使われている Myers のアルゴリズムを近似アルゴリズムとして改良したものは若干精度は落ちるもののとてつもなく速い。これは GNU diffutils でも採用されている手法である。逆に最適化がなされれていない Monotone はある条件下ではとてつもなく遅い。

- Subversion … Wu のアルゴリズム
- Git … Myers のアルゴリズム
- Mercurial … Gestalt Approach
- Monotone … Wu のアルゴリズム

ここではそのような効率的なアルゴリズムの一つである Wu のアルゴリズムについて軽く解説する。

Wu のアルゴリズムは平均計算量が O(N+PD)、最悪の場合でも O(NP) な差分アルゴリズムである。後発の Git や Mercurial、Bazzar に取って代わられようとしているが、十分に枯れた安定感や周辺ツールの充実もあって今でも人気がある Subversion や拙作の dtl で使われている。このアルゴリズムは (証明を理解しようとするのはともかく) 他の差分アルゴリズムと比べて実装が楽で当時、アルゴリズムの論文を読んでさっぱり理解できなかった私が擬似コードをそのまま書き写しただけでちゃんと動いたほどだ (単に擬似コードが秀逸だったのかもしれないが。あと今は一応理解しているつもりでいる)。

ひとまず、Wu のアルゴリズムを使って引数として与えられた二つの要素列間の編集距離を計算する簡単な Python コードを紹介しよう。LCS や SES の求め方については dtl や github にある実装を見て欲しい。というのも LCS と SES を求めるコードはちょっと複雑でこれを含めると途端に長くなってしまうのだ。

```
import sys
def editdistance(a, b):
  m = len(a)
  n = len(b)
  if m >= n:
    a, b = b, a
    m, n = n, m
  offset = m + 1
  delta  = n - m
  size   = m + n + 3
  fp = [ -1 for idx in range(size) ]
  p = -1
  while (True):
    p = p + 1
    for k in range(-p, delta, 1):
      fp[k+offset] =
        snake(a, b, m, n, k,
              fp[k-1+offset]+1, fp[k+1+offset])
    for k in range(delta+p, delta, -1):
      fp[k+offset] =
        snake(a, b, m, n, k,
              fp[k-1+offset]+1, fp[k+1+offset])
    fp[delta+offset] =
      snake(a, b, m, n, delta,
            fp[delta-1+offset]+1,
            fp[delta+1+offset])
    if fp[delta+offset] >= n: break
  return delta + 2 * p

def snake(a, b, m, n, k, p, pp):
  y = max(p, pp)
  x = y - k
  while x < m and y < n and a[x] == b[y]:
    x = x + 1
    y = y + 1
  return y

if __name__ == "__main__":
  print onp.editdistance(sys.argv[1], sys.argv[2])
```

このアルゴリズムは動的計画法を改良したような形になっている。動的計画法ではあらかめじめエディットグラフのための M * N の表を作り、さらに表をすべて埋めるだけの計算を行う必要があるが、これはかなり無駄な計算が多くなる。そして何よりものすごくメモリを喰う。Wu のアルゴリズムでは編集距離の削除部分 (P) を二つの要素列を比較していきながら、比較が最後まで終わらない場合はこの値をインクリメントしていく。編集距離は比較が完全に終わった時の P の値から以下の公式で求まる。

D = N - M + 2P

∵ D: 編集距離 , P: 編集距離の削除部分 , N>M(N, M は要素列の長さ)

LCS と SES については比較の過程で経路を記録していくことによって算出することができる。これを実現するコードについては dtl や github のコードを参照してほしい。

最悪の場合に対処する。Wu のアルゴリズムに限った話ではないが、多くの差分アルゴリズムは比較する要素列が似通っている場合に効率よく動作するように設計されていることが多く、要素列が全然違う場合、素朴な実装だととんでもなく遅くなるかメモリを使い果たしてしまうことがある。これは差分が大きすぎて SES を求めるために記録した経路が膨大になった結果、メモリ不足に陥るためである。dtl では SES を記録する際は以下のような手順を実行するようにしてこの問題を回避している。

1. SES の経路を動的配列に記録していく。
2. 動的配列が一定のサイズに達したらそこまでの SES を求める。
3. 動的配列をクリアする。
4. 要素列の比較が終わっていない場合は 1 に戻る。
5. 求めた複数の SES を連結して 1 つにする。

この方法は簡単ではあるが、どのみちメモリの動的割り当て・解放を繰り返す羽目になるので、あまり速くない。GNU diffutils などはもっといい方法を採用しており、Myers のアルゴリズムに近似アルゴリズム的な改良を加えて計算量を大幅に落としている。

差分を構築する。差分は SES を加工することによって構築する。ここでは Unified Format を生成する方法について記述する。

Unified Format はできるだけ少ない出力で人間に見やすいことを念頭に作られた差分の形式である。ほかにも Context Format という形式があり、プロジェクトによってはパッチの形式は Unified Format よりもこちらのフォーマットが好まれることがある。これはその昔、Unified Format をサポートしているツールが少なかったことに起因しているのだろうけど、正直言って Context Format はものすごく見づらい。

Unified Format の話に入ろう。Unified Format では二つの文字列「abc」と「abd」の差分は以下のように出力される。

```
@@ -1,3 +1,3 @@
 a
 b
-c
+d
```

まずは @@ で囲まれた数字だが、これを次のように形式化しよう。

```
@@ -a,b +c,d @@
```

a, b, c, d の値の意味は次のようになっている

- a, c … 変更箇所の開始位置。
- b, d … 開始位置の箇所から表示する要素数。

つまり、さきほどの差分に表示されている @@ -1,3 +1,3 @@ はそれぞれの要素列の1つ目から3つ目までを表示していることを表している。このような差分をハンクと呼び、Unified Format はこのハンクが複数集まったものと言える。dtl では Unified Format のハンクを表す構造体は以下のように定義されている。

```
/**
 * Structure of Unified Format Hunk
 */
template <typename sesElem>
struct uniHunk {
  int a, b, c, d; // @@ -a,b +c,d @@
  // anteroposterior commons
  vector< sesElem > common[2];
  vector< sesElem > change; // changes
  int inc_dec_count; // count of incr and decr
};
```

Unified Format のハンクには「A」あるいは「D」の要素を少なくとも1つ以上含んだ「C」の要素がある一定の回数 (dtl では3回) までしか連続しない部分があり、上記の構造体ではこれを change という変数に格納している。そして、この変数 change に格納された部分の前後に「C」の要素がある一定の回数連続した部分が存在し、この前後の部分をそれぞれ common[0] と common[1] に格納している。また、inc_dec_count は dtl がハンクを構成する際に使用するメタ情報である。

Unified Format のハンクを SES から構築するには SES の先頭から「A」あるいは「D」の要素を探索し、そこから前後の「C」の要素群を見つけ、これらを以下の順序で連結する。

1. common[0]
2. change
3. common[1]

あとは同じようにしてすべてのハンクを計算してそれらを連結すれば Unified Format の差分が出来上がる。a, b, c, d は SES の先頭からハンクになる条件を満たす部分を探索していくことで計算することができる。dtl では SES から Unified Format な差分を構築する関数として composeUniHunks がある。

パッチ。差分が求まればそれを元に要素列を変更したり、元の要素列を復元することが可能になる。一番簡単な方法としては対象の要素列と SES を比較して「A」の場合は要素を挿入、「D」の場合は要素を削除する。例えば、対象とする二つの要素列を A と B、A から B への SES を C とすると、A に C を適用すれば B が得られる。以下は変更元の要素列に SES を使って patch を適用するアルゴリズムの擬似コードである。要素列の変更時は要素の挿入が頻繁に発生するので、変更元の要素列はリストとして扱うのがよい。

```
Algorithm Patch
Begin
  sesVector sesVec;        // SES の配列
  elementList elemList;    // 変更元の要素列のリスト
  L := sesVec.length();
  j := 1
  For i := 1 to L do
    if sesVec[i].type = SES_ADD :
      elemList.insert(j, sesVec[i].elem)
    elif sesVec[i].type = SES_DELETE :
      j := elemList.erase(j);
    elif sesVec[i].type = COMMON :
      j := j + 1;
  End
End
```

また、SES は元の要素列に比べて長くなるので、できればより小さな差分 (例えば Unified Format の差分) を適用してパッチをあてるのが望ましい。そのため、dtl には Unified Format の差分をパッチとして扱い、それを適用するための関数 uniPatch がある。Unified Formmat の差分は断片的な SES の集まりと捉えることができるので、プログラム的にはさきほどのパッチプログラムとあまり変わらない。SES にインデックスがついたものとして扱えばよい (それが案外ややこしいことは確かだが)。

diff3 は3つの要素列をそれぞれ比較して変更点を抽出したり、マージするためのプログラムである。diff3 はバージョン管理システムようなソフトウェアにおいて複数のブランチ間の変更を統合したり、他人の変更を自分の作業コピーに取り込む際に使用される。

マージしよう。まず、要素列をそれぞれ A, B, C とし、B が変更前の要素列、A と C がそれぞれ別々のユーザによって B に異なる変更を加えた要素列とする。この場合、マージして最終的に出来上がる要素列は衝突が発生しないのであれば A と C を組み合わせたものとなる。実際のマージの手順は dtl だと以下のようになっている。

1. 要素列 B と A の差分を取る (SES を SES_BA とする)。
2. 要素列 B と C の差分を取る (SES を SES_BC とする)。
3. SES_BA と SES_BC を先頭から比較していき、A と C の変更点が組み合わされた要素列 S を生成する (S は最初は空)。

最後の手順では、比較する SES 同士の要素とその種類 (「C」,「A」,「D」) の組合せによって S に要素を追加するか、衝突が発生するかどうかはほぼ一意に定まるため、実装はそれほど難しくない。詳しい実装については dtl の src/Diff3.hpp にある merge_ という関数を見て欲しい。

衝突箇所の特定。三つの要素列 A, B, C をそれぞれ「adc」,「abc」,「aec」とし、B が変更前の要素列とする。この組合せでは、A で「b」が「d」に、C で「b」が「e」となっているため、それぞれ同じ箇所に対して別々の変更がなされており、衝突していると判断することができる。diff3 プログラムで衝突をどのように表現するかは一様に決まってるわけではないが、以下のように何らかのセパレータを挟むのが一般的である。

```
a<d|e>c
```

上記は dtl で A, B, C をマージして衝突が発生した結果、生成される要素列の一例である (dtl ではセパレータは変更可能)。各セパレータの意味は以下の通り。

1.「<」　　：衝突箇所の開始地点
2.「<」と「|」で囲まれた部分　：A の衝突部分
3.「|」と「>」で囲まれた部分　：C の衝突部分
4.「>」　　：衝突箇所の終了地点

さて、真面目に読んで頂いている読者であれば普通は以下のようになった方がいいんじゃないのと思うかもしれない。

```
a<d|b|e>c
```

一応言っておくと dtl でもこうなる予定だった。しかしなかなか動作が安定せず、ひとまず今の形になった。将来的には改良されるかもしれない。

diff や diff3 と違い、3 つの要素列間でどの部分が衝突しているかどうかをチェックするのは LCS や SES がわかってなくてもできる。・・・というよりは dtl で SES や LCS を使って算出しようと試みたが、うまくいかなかったので、3 つの要素列をループで比較しながらアドホックに求めるようにした。ただ、自分で実装した後でほかのプログラムを参考にしてみようとしたが、同じようなことをしていた。そのため、どうやったら綺麗に書けるのか筆者には今でもわからない。

最後に、この原稿の初期のタイトルは「Making VCS」といって筆者がバージョン管理システムもどきを作った時のことを書くはずだったのだが、リポジトリの構造あたりの掘り下げが不十分であまり納得のいく記事にならなかったので、差分の話に焦点を絞った内容となった。この記事を書くひとつのきっかけになった dtl だが、現在 (2010 年 7 月 4 日執筆時点で)、最新版のバージョンが 1.06 であり、近々いくつかのバグフィックスがなされた 1.07 が出る予定である。Google Code のダウンロードリストを見るとダウンロード総数は現在 (2010 年 7 月 4 日執筆時点) で 400 近くあり (Mercurial リポジトリからの clone 回数は含まれない)、たまに海外の開発者からバグレポートや感謝のメール、コメントをもらうこともある。

参考文献

``An O(NP) sequence comparison algorithm'', Sun Wu, Udi Manber, Gene Myers
各種言語による Wu のアルゴリズムの実装 , http://github.com/cubicdaiya/onp
dtl(C++ で書かれた拙作の diff ライブラリ), http://code.google.com/p/dtl-cpp/
Keynote による Diff の発表資料 , http://www.slideshare.net/cubicdaiya/diff-2660306

メインメモリアクセスマニュアル

著 : nish

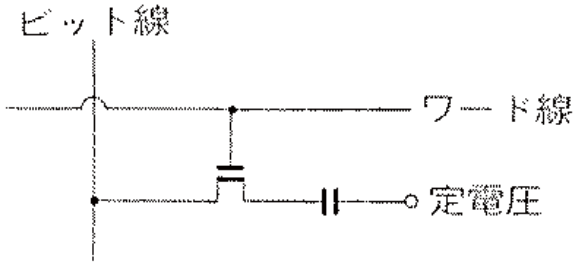
昔々、DRAM がまだただの DRAM だったころ、DRAM の速度は CPU とほとんど同じでした。

時は流れていつの間にか CPU にはキャッシュが不可欠になり、DRAM は FP-DRAM になり、EDO-DRAM になり、SDRAM になり、DDR-SDRAM になり、DDR2-SDRAM になり、DDR3-SDRAM になり、もうすぐ DDR4-SDRAM になります。

今までもこれからもメインメモリが DRAM であることは疑いようがないようですが、キャッシュミスの重さがよく知られているように、もっともプログラムの足を引っ張る部分でもあります。

不幸にして DRAM にはドライバもなければ、コンパイラも OS のサポートもありません。たまにはこの厄介なデバイスとの付き合い方を考えてみるのも悪くないでしょう。

DRAM とは何であって何でないか？まずはおさらいです。DRAM とはどういうデバイスだったでしょうか。DRAM は最小素子がトランジスタとコンデンサからなるメモリデバイスです。



< 図 1. DRAM 素子 >

特徴的なのは、アドレスバスがデバイスのアドレス空間に必要な幅の 1/2 しかなく、アドレスを Row と Col の 2 回に分けて受け取ること、また、定期的にアクセスを中断してリフレッシュ動作を行わないとデータを保持できないことが挙げられます。

ここで Row と Col を分離しているのはピン数をケチるためだったと思われるのですが、今に至るまでそれが維持されているのは以下の理由によります。

内部的には DRAM は Row アドレスでアクセスされる Col サイズの配列構成を取ります。高速化 DRAM は Row アドレスが入力されると、内部的に 1 行分をバッファにコピーします。コピー元となった行データは電荷を失い破壊されます。次に、バッファに対し Col アドレスで必要なワードを選択し、バッファに対してアクセスを行います。アクセスが終了すれば、バッファを行に書き戻して処理が終了します。

こうすることで直接 DRAM にアクセスするより高速にアクセスできるのですが、Row アドレスを入力してから Col を入力するまでに行の選択、バッファへのコピーで Row Address Setup Latency(RL) と呼ばれる待ち時間が、Col アドレスを入力してからデータにアクセスするまでに Col Address Setup Latency(CL) と呼ばれるバッファからの選択待ち時間がかかります。

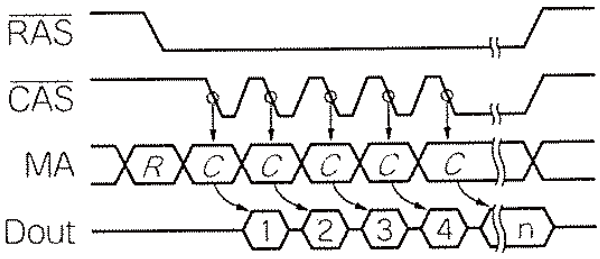
したがって、Row アドレス入力から RL 時間経過までに Col アドレスを入力すればよく、Row と Col を分離するのには一定の合理性があるというわけです。

PC における DRAM の歴史。内部的に行バッファを持つ高速化 DRAM の歴史は FP-DRAM、Fast Page DRAM から始まりました。

一般的にデータ局所性の観点から、DRAM に対して連続してアクセスされるアドレスが同じ Row アドレスになるということはかなり多くなるはずです。

そこで、プロセッサの速度についていけなくなってきた DRAM のアクセス時間を改善するために、DRAM 側で「高価で少量だが高速なメモリを間に挟む」というまっとうな解決策が図られたのでした。

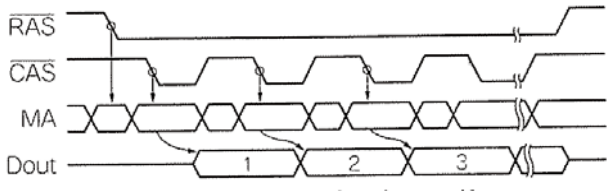
これを高速ページモードとして実装したのが FP-DRAM です。



< 図 2. 高速ページモード >

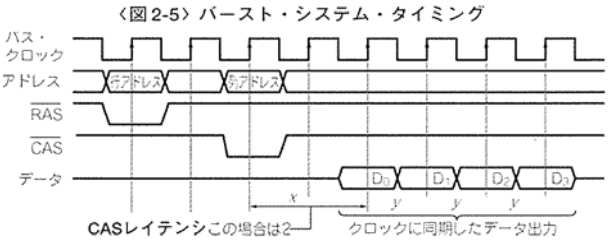
PC 用メモリモジュールとして有名な EDO-DRAM も、SDRAM も、もちろん DDR-SDRAM もこの延長線にあります。順を追って見ていきましょう。

EDO-DRAM は FP-DRAM の Data の Out 時間が Extend されたものです。今までアドレス指定からデータアクセスまでは割り込みのできない操作でした。EDO-DRAM ではデータアクセスと、次にアクセスしたい Col アドレスの指定をオーバラップさせ、データに連続的にアクセスできるようになりました。



< 図 3. Extended Data Out>

SDRAM はデータ入出力が Asynchronous から Synchronous になり、バースト転送がデフォルトのデータアクセス手段になりました。バースト転送は、一度 Col アドレスを指定すると、その Col アドレスが位置するバースト長分のブロックのワードを連続して転送する機能です。



< 図 4. SDRAM>

もちろん EDO-DRAM 同様に先行するデータアクセス中に別の Col アドレスを指定することで、連続的にデータアクセスを行うことができますし、Row アドレスを指定することで、ページ切り替えもオーバラップすることができます。ただし、オーバラップさせるためには、データバスが空く時間から逆算して Row/Col アドレスを発行する必要があります。

どういうことかと言いますと、Row アドレスを発行した後、RL クロック後には必ず Col アドレスを発行する必要がありましたし、Col アドレス発行後、CL クロック後には必ずデータアクセスを行う必要がありました。

先行するデータアクセスが次のデータアクセスまでに終了していない場合は、先行するデータアクセスはキャンセルされ、次のデータアクセスが有効になります。

また、リード直後のライトはデータバスを 1 クロック余計に確保し、データノイズを防ぐ構造になっています。

それ以外の場合では、ライトは Col アドレス発行と同時に実行できます。

DDR-SDRAM は、データアクセス速度が DoubleDataRate になりました。ベースクロックの立ち上がりと立ち下がりのたびにデータを入出力するため、同等ベースクロックの SDRAM の 2 倍の転送速度を持ちます。またライトが Col アドレス発行後常に 1 クロック消費するようになりました。

DDR2-SDRAM は、ベースクロックごとに 4 つのデータを入出力し、同等ベースクロックの SDRAM の 4 倍の転送速度を持ちます。一見すると QuadDataRate とでも呼びたくなりますが、実は DDR2-SDRAM 内部の I/O モジュールがベースクロックの 2 倍で動いており、その I/O モジュールが DDR で入出力しています。

また DDR2 からは、複雑であったオーバラップを解決するために Posted CAS が導入されています。これは、DDR2-SDRAM 内部で Col アドレスをバッファリングし、Col アドレス発行からデータアクセスまでの時間制約を和らげるものです。

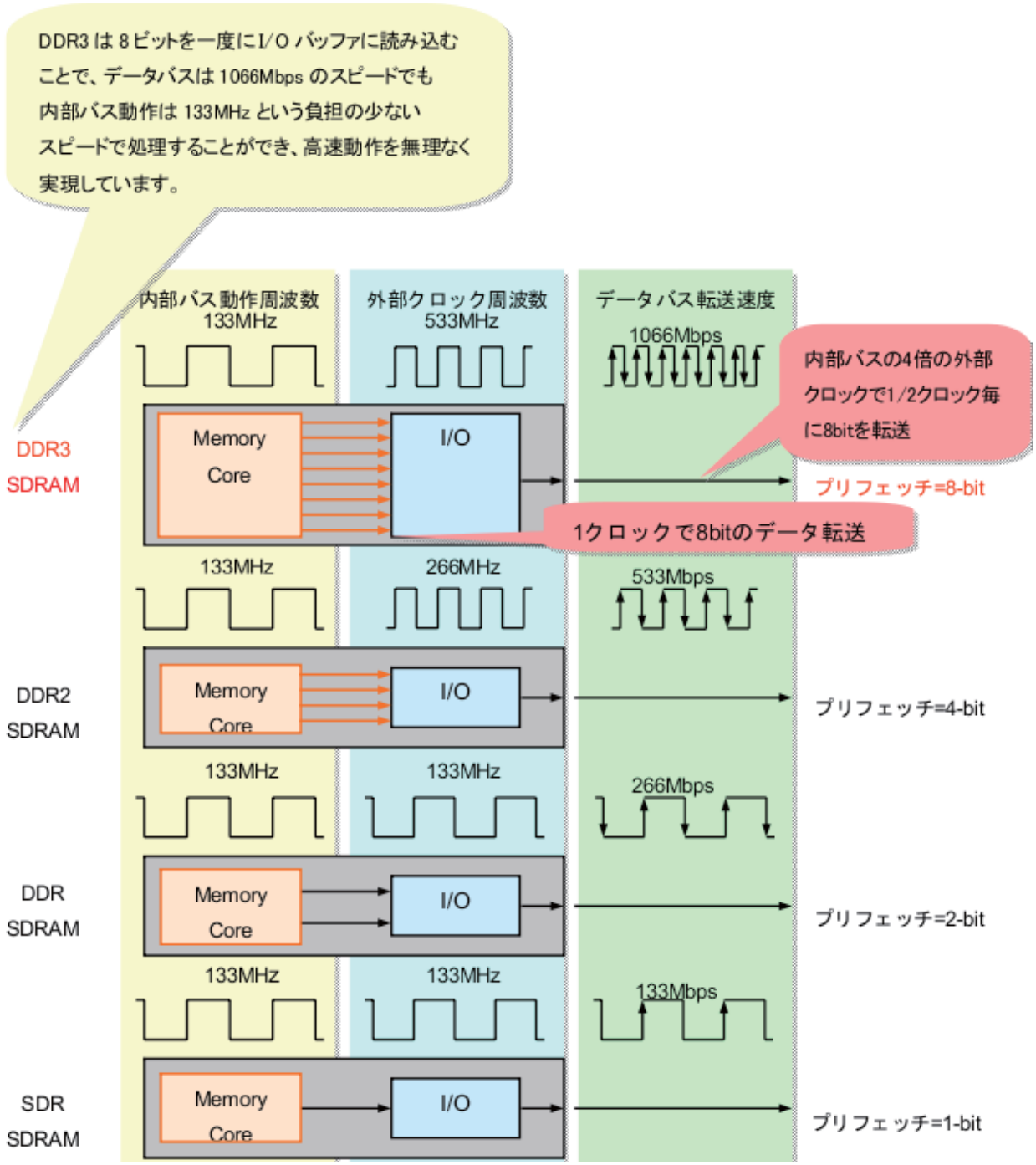
DDR3-SDRAM は、DDR2-SDRAM の I/O モジュールが倍の速度になったもので、特に目新しいことはありません。

ただし DDR3 の頃からほとんどのプロセッサにメモリコントローラが標準搭載されたことは注意する必要があるでしょう。

なお、DRAM の RL/CL はおおよそ次のような値域となっています。

種別	RL	CL	clock(MHz)	転送速度 (Mbps)
EDO DRAM	40ns-60ns	40ns-60ns	-	-
SDRAM	2-3	2-3	66-133	66-133
DDR SDRAM	2-3?	2-3	100-200	200-400
DDR2 SDRAM	2-6?	2-6	200-400	400-800
DDR3 SDRAM	5-11?	5-11	400-800	800-1600

SDRAM 以降は規格上用意されている値ですので、製品としては大きい方の値が一般的です。ほとんど高速化されていないことと、それでも少しずつ高速化されていることがわかるかと思います。



< 図 5. すべて DDR>

CPU と SDRAM。続いて、これらの DRAM が実際に今時の PC 上で x86 と組み合わさった時にどういった動作となるのかを考えてみましょう。SDRAM 以前のメモリは、あえて記載するほどのことはないでしょうから無視します。

電気的には CPU と SDRAM モジュールは 64 ビットバスでつながっています。デュアルチャネルやトリプルチャネルの場合は、そのバスが 2 本か 3 本あります。これを 128 ビットバスとして考えるか、64 ビットバスが 2 本と考えるかは世代によって異なるようです。

CPU とメインメモリの関係を考える上で外せないのはもちろんキャッシュメモリです。キャッシュメモリ上のキャッシュラインと SDRAM のバースト転送サイズは通常一致させます。キャッシュラインサイズは 32Byte または 64Byte といったところで、SDRAM のバースト転送サイズを設定してサイズを調整します。

キャッシュ上にデータが乗っている場合にはもちろんキャッシュから読みますが、キャッシュヒットしなかった場合には、SDRAM からデータ転送を行います。このときバーストリードが終了し、キャッシュラインがすべて埋まるまでプロセッサはストールするわけではなく、バーストリードの最初のデータが来た時点で実行されるのが

一般的です。プロセッサが動作しているのと平行してキャッシュフィル作業が続けられることになります。

また、キャッシュメモリはライトバック動作するものが一般的ですので、ダーティなキャッシュを破棄するときには、(ライトバッファを通して) キャッシュラインごとバーストライトされます。

なお、DDR 以降の SDRAM は 1 ワード単位でのデータライトは苦手で、バーストライトを行う際にデータをマスクすることで何とか実現しています。ライトスルー動作は現実的ではありません。

ただし、同じバーストブロックに位置する書き込みを結合するライトバッファがありますので、直線的なアドレスに対するライトの速度はひどいことにはなりません。

CPU と SDRAM との速度差にも触れておきましょう。2GHz のプロセッサであれば、DDR2-800 の CL=2 を待ってる間に、10clock 消費します。4 ワードのデータ転送が 2clock で終わるので、これもプロセッサ側では 10clock 消費することになります。

Random Access Memory からの脱却。このようなメモリと我々はどのように付き合えばよいのでしょうか。

少し簡単なプログラムを書いて考えてみたいと思います。

まず、もっとも単純で頻度の高いメモリコピー時のメモリアクセスを考えてみましょう。メモリコピーのコードを今更書く人は少ないでしょうが、画像操作や行列計算で似たようなことをする人は多いと思います (それも今更書かないという気はしますが…)。

```
static void
copy(const void* const vpSrc,
     void* const vpDst,
     const size_t byteLength)
{
  const int32_t* const src
    = (const int32_t*)vpSrc;
  int32_t* const dst = (int32_t*)vpDst;
  const size_t count = byteLength
                     / sizeof(int32_t);

  for (size_t i = 0; i < count; ++i)
    dst[i] = src[i];
}
```

このプログラムをメモリバスを意識しながら追ってみましょう。

まず src からのリードを開始します。このデータはキャッシュに乗っていないので、メインメモリへのアクセスです。まずメインメモリに Row アドレスを投げます。続いて Col アドレスを投げます。RL + CL 時間後に、メインメモリから最初のデータが届きます。

ここで OutOfOrder のプロセッサでは、次に dst を読むことは確定的に明らかなので、RL + CL 時間を待たずに dst のキャッシュラインの準備ができます。dst のキャッシュラインは存在しないので、読み込まなければなりません。(メモリクロックからすれば) ほとんど直後にメインメモリに dst のアドレスを投げられるので、RL + CL 時間をオーバラップできるでしょう。

最初の必要なデータが来た時点でプロセッサは先に進みます。もちろん裏側ではキャッシュフィルが動作しています。

プロセッサは次に dst にデータを書こうとします。

InOrder のプロセッサでは、この時点でメインメモリから dst のデータを読み込みます。不幸にして、この時点で先行するメモリアクセスはほとんど終わっているので、残すところバースト転送のみです。オーバラップはあまり期待できず、RL + CL 時間がまともに掛かってきます。

また、この時点で強力な OutOfOrder のプロセッサであれば、未来のイテレーションの src アドレスを知っているはずです。従って、次のキャッシュラインの RL + CL をかなりオーバラップ実行できます。

InOrder のプロセッサはやはりバースト転送が始まった直後まで何もできません。

こうしてイテレーションを重ねていくとダーティなキャッシュラインが溜まってきます。ダーティなキャッシュラインを書き戻す際には、次にアクセスするアドレスがわかっている状態ですので、おそらく InOrder なプロセッサでも大部分をオーバラップできると期待できます。

では、実際に測定してみましょう。

比較コードは、単純なメモリリード / ライトです。直線的なアクセスのため、ほとんどのアクセスで Row アドレスが不要で、かつ CL もオーバラップできるため、限界速度が出ると期待したいところです。

```
static void
read(const void* const vpSrc,
     void* const dummy,
     const size_t byteLength)
{
  const int32_t* const src
    = (const int32_t*)vpSrc;
  const size_t count = byteLength
```

```
                    / sizeof(int32_t);
  for (size_t i = 0; i < count; i += 16) {
    volatile register int32_t t = src[i];
  }
}
#define copy read

static void
write(const void* const dummy,
      void* const vpDst,
      const size_t byteLength)
{
  int* const dst = (int*)vpDst;
  const size_t count = byteLength / sizeof(size_t);

  for (int i = 0; i < count; ++i) dst[i] = -1;
}
#define  copy    write
```

測定コードはこんなのです。

```
#define SIZE (32 * 1024 * 1024)
#define CACHE_LENGTH \
       (COMPILE_PARAM_CACHE_K_SIZE * 1024)

int main(int argc, char* argv[])
{
  const int count = atoi(argv[1]);
  int* src = (int*)malloc(SIZE);
  int* dst = (int*)malloc(SIZE);
  std::fill(src, src + SIZE, -1);
  std::fill(dsr, dst + SIZE, 0);

  const clock_t start = clock();
  for (int i = 0; i < count; ++i)
    copy(src, dst, SIZE);
  double period = 1000.0 * (clock() - start)
                / CLOCKS_PER_SEC;
  printf("%f\n", period);
}
```

	copy (ms)	read (ms)	write (ms)	theory (ms)
Celeron (Pentium4 2.8GHz) DDR-400*2	48.6	16.1	32.8	9.76
Pentium D 935(Nocona 3.0GHz) DDR2-667*2	23.6	8.85	15.4	6.3
Core Duo T2500(Core 2.0GHz) DDR2-400	46.5	15.3	31.0	10.4
Core2 Quad Q9300(Core2 2.5GHz) DDR2-800*2	16.7	9.48	12.6	5.2
Atom Z520(Atom 1.33GHz) DDR2-667	56.6	30.7	40.4	6.3
AthlonII X4 605e(K10 2.3GHz) DDR2-800*2	16.9	8.0	6.6	5.2
Xeon E5520(Nehalem 2.27GHz) DDR3-1066*3	7.41	7.74	5.60	3.93

理論値は、同サイズを read/write したときの理論転送速度です。理想的には OutOfOrder であれば、コピー時間は、理論値の 3 倍の時間となるはずですが、そうなっているのは Core2/K10 のみだということが分かります。

Nehalem は理想を飛び抜けてますが、トリプルチャネルによるものでしょう。上手くデュアルチャネル / トリプルチャネルを並列に使って高速処理できるのは K10/Nehalem ぐらいのようです。

それ以外のプロセッサではアクセス要求を適切にメモリコントローラに流せておらず、メモリバスを空けてしまっているように見えます。

そこで次のようなコードが考えられます。

まず明らかにライト時のキャッシュフィルは不要なので、メインメモリに直接書き込む movnti を使います。

```
#include "emmintrin.h"
#define COPY_BLOCK_COUNT \
       (4 * 1024 / sizeof(int32_t))

static void
sse_copy(const void* const vpSrc,
         void* const vpDst,
         size_t byteLength)
{
  const int32_t* const src
    = (const int32_t*)vpSrc;
  int32_t* const dst = (int32_t*)vpDst;
  const size_t count = byteLength
                     / sizeof(int32_t);
```

```
  for (size_t i = 0;
       i < count;
       i += COPY_BLOCK_COUNT) {
    for (size_t j = 0; j < COPY_BLOCK_COUNT; ++j) {
      _mm_stream_si32(&dst[i + j], src[i + j]);
    }
  }
}
#define copy    sse_copy
```

リード要求は可能な限り早く、連続して出します。キャッシュラインの先頭のみを踏み、キャッシュラインをキャッシュに読み込ませます。これで Row アドレスのセット回数が減り、メモリバスは楽になるはずです。

```
#include "emmintrin.h"
#define COPY_BLOCK_COUNT \
       (4 * 1024 / sizeof(int32_t))

static void
prefetch_copy(const void* const vpSrc,
                    void* const vpDst,
                    size_t byteLength)
{
  const int32_t* const src
    = (const int32_t*)vpSrc;
  int32_t* const dst = (int32_t*)vpDst;
  const size_t count = byteLength
                     / sizeof(int32_t);

  for (size_t i = 0;
       i < count;
       i += COPY_BLOCK_COUNT) {
    for (size_t j = 0;
         j < COPY_BLOCK_COUNT;
         j += CACHE_LINE_LENGTH/sizeof(int32_t)) {
      volatile register int32_t t = src[i + j];
    }
    for (size_t j = 0; j < COPY_BLOCK_COUNT; ++j) {
      _mm_stream_si32(&dst[i + j], src[i + j]);
    }
  }
}
#define copy    prefetch_copy
```

測定してみます。

	previous (ms)	sse_copy (ms)	prefetch (ms)
Celeron (Pentium4 2.8GHz) DDR-400	48.6	32.5	29.3
Pentium D 935(Nocona 3.0GHz) DDR2-667*2	23.6	16.6	17.8
Core Duo T2500(Core 2.0GHz) DDR2-400	46.5	24.1	28.7
Core2 Quad(Core2 2.5GHz) DDR2-800*2	16.7	13.8	21.9
Atom Z520(Atom 1.33GHz) DDR2-667	56.6	39.6	45.0
AthlonII X4 605e(K10 2.3GHz) DDR2-800*2	16.9	18.8	17.9
Xeon E5520(Nehalem 2.27GHz) DDR3-1066*3	7.41	11.8	14.1

おっと !prefetch が有効なのは Pentium4 だけでした。正確にはこれは SDRAM/DDR SDRAM 時代に有効なテクニックです [5]。DDR2 以降では、Row アドレスの設定をうまくオーバラップでき、ページ遷移がゼロコストでできていることがよくわかります。

sse_copy であってもかえって遅くなるのは K10/Nehalem です。キャッシュメモリに溜め込めず、したがってチャネルが空くまで書き戻しを待つことができなくなったためと考えられます。

これでは最新のプロセッサとメモリの優秀性を確認しただけになってしまいました。明らかにチャネルやプリフェッチ、OutOfOrder が役に立たないケースを考えてみましょう。

```
struct StandardNode {
  struct StandardNode* next;
  uintptr_t value;
};

struct Cursor {
  struct StandardNode* current;
  Cursor(StandardNode* c) : current(c) {}
  void next() { current = current->next; }
};

static Cursor setup(void* vpSrc,
                    size_t byteLength) {
  StandardNode* const src = (StandardNode*)vpSrc;
  const size_t count = byteLength
                     / sizeof(StandardNode);

  size_t* order = new size_t[count];
  for (size_t i = 0; i < count; ++i) order[i] = i;
```

```
  std::random_shuffle(order, order + count);
  StandardNode* prev = NULL;
  for (size_t i = 0; i < count; i++) {
    StandardNode& node = src[order[i]];
    node.next = prev;
    node.value = i;
    prev = &node;
  }
  delete[] order;
  return Cursor(prev);
}
typedef StandardNode Node;
```

単純なリストです。リストはノードにアクセスするまで次のアドレスがわかりません。従ってどんなに高級なプロセッサであっても、バースト転送が始まり、next ポインタが入手できるまで先読みすることは出来ないのです。

そこで実用性はともかく、このようなリストを考えてみます。

```
struct FutureNode {
  struct FutureNode* future;
  uintptr_t value;
};

struct Cursor {
  struct FutureNode* current;
  struct FutureNode* nextPtr;

  Cursor(FutureNode* c, FutureNode* n)
    : current(c),nextPtr(n) {}
  void next() {
    FutureNode* future = current->future;
    current = nextPtr;
    nextPtr = future;
  }
};

static Cursor setup(void* vpSrc,
                    size_t byteLength) {
  FutureNode* const src = (FutureNode*)vpSrc;
  const size_t count = byteLength
                     / sizeof(FutureNode);

  size_t* order = new size_t[count];
  for (size_t i = 0; i < count; ++i) order[i] = i;
  std::random_shuffle(order, order + count);

  FutureNode* prev = NULL;
  FutureNode* prev_prev = NULL;
  for (size_t i = 0; i < count; ++i) {
    FutureNode& node = src[order[i]];
    node.future = prev_prev;
    node.value = i;

    prev_prev = prev;
    prev = &node;
  }
  return Cursor(prev, prev_prev);
}
typedef struct FutureNode Node;
```

次のノードのポインタ next ではなく次の次のノードのポインタ future を用います。これによって、次のノードをリードしてる間に次の次のノードのアドレスを知ることができ、OutOfOrder を活かすことができます。

なお、32 ビット環境下でも 64 ビット環境下でもノード構造が複数のキャッシュラインにまたがる別の不幸を発生させないように、ノードサイズがポインタサイズの 2 倍になるよう value を設定しています。

早速計測してみましょう。より実際の使用に近付けるため、value も何だかよくわからない計算に使用しておきます。

```
#define SIZE (1024*1024*4 * sizeof(Node))

static uintptr_t seek(Cursor cursor) {
  uintptr_t v = 0;
  while (cursor.current != NULL) {
    v = v << 2 ^ cursor.current->value;
    cursor.next();
  }
  return v;
}

int main(int argc, char* argv[])
{
  int count = atoi(argv[1]);
  void* src = malloc(SIZE);
  Cursor cursor = setup(src, SIZE);
  const clock_t start = clock();
  for (int i = 0; i < count; ++i) seek(cursor);
  double period = 1000.0 * (clock() - start)
                / CLOCKS_PER_SEC;
  printf("%f\n", period);
}
```

	normal (ms)	future (ms)
Celeron (Pentium4 2.8GHz) DDR-400	667	384
Pentium D 935(Nocona 3.0GHz) DDR2-667*2	543	285
Core Duo T2500(Core 2.0GHz) DDR2-400	490	249
Core2 Quad Q9300(Core2 2.5GHz) DDR2-800*2	416	215
Atom Z520(Atom 1.33GHz) DDR2-667	744	739
AthlonII X4 605e(K10 2.3GHz) DDR2-800*2	465	267
Xeon E5520(Nehalem 2.27GHz) DDR3-1066*3	361	190

このようにプロセッサのメモリバスの動作を考えることで、プログラムの速度がずいぶん違ってくることがお分かりいただけたかと思います。

最後に、この記事における SDRAM 関連用語の表記はあまり正しくありません。正確なところは参考文献を参照してください。

測定に使用したコンパイラは gcc4.2 ～ 4.3 ですが、いずれもメモリアクセスに関する最適化は行っておらず、キャッシュ制御命令を含まない、書いたとおりの素直なコードになります。

なおバスの挙動はすべて理論上の仮説であり、実際にバスの動作を確認してるわけではありません。実は全然違う動きをしてるかも…

[参考文献]
[1] ELPIDA「SDRAM の使い方 - ユーザーズマニュアル」(J0123N50.pdf)
[2] ELPIDA「DDR SDRAM の使い方 - ユーザーズマニュアル」(J0234E50.pdf)
[3] ELPIDA「DDR2 SDRAM の使い方 - ユーザーズマニュアル」(J0437E40.pdf)
[4] ELPIDA「DDR3 SDRAM の新機能の使い方 - ユーザーズマニュアル」(J1503E10.pdf)
[5] AMD memcpy_amd.zip (現存せず)
[図出典]
図 1 CQ 出版「トランジスタ技術 1997 年 4 月号特集」
図 2-4 CQ 出版「トランジスタ技術 1999 年 4 月号特集」
図 5 ELPIDA「DDR3 SDRAM の新機能の使い方 - ユーザーズマニュアル」(J1503E10.pdf)

C++0x の空、VariadicTemplates の夏

著 : lyrical logical

「C++ は難しい」

いつからか、C++ は「難解な罠だらけで使いにくい」というレッテルを貼られがちな言語になってしまいました。多くの場合、それらは悲しい誤解です。確かに C++ の言語機能には、強力さに由来する難解さを持つものもあります。ADL, virtual/multiple inheritance, operator overload, そして template ……しかしそれらの難解さは、そのまま言語の使いやすさを示すものではありません。そういった言語機能を直接使わずとも、その強力さの支援を受けることができるのです。よくいわれることですが、C++ を使うのに、C++ の全てを理解する必要はありません。

とはいえ、あなたの崇高な目的のために、そういった難解な言語機能を直接利用しなければならないこともあるかもしれません。この文書は、C++0x で C++ に新たに加わる言語機能 "Variadic templates" を使いこなしたい、ヴァーチャルな人のために書かれました。単なる紹介に留まらない、Variadic templates の実践的な解説になれるよう、書いたつもりです。対象読者は、ジェネリックプログラミングに「ある程度」の理解のある C++ 経験者を想定しています。C++0x の知識が可能な限り必要にならないような例を選ぶようにしたのですが、一部新しい言語機能を利用している箇所があります。

ごめんなさいのこと

用語を日本語訳すべきかどうか悩んだのですが、基本的に訳さないことにしました。原稿を書いている 2010 年 7 月現在、定番の日本語訳というのがまだないためです。

また、コードの中には未だコンパイラの実装が無いために、動作の確認が行えていないものが少しですがあります。例えば alias declaration は、まだどのコンパイラでも実装されていないため、確認を行っていません。
動かなかったりしたらごめんなさい ヽ(；▽；)ノ

0x 以前の「擬似的な」可変長引数テンプレート

C++ は template という、ジェネリックプログラミングを強力にサポートする言語機能を持っています。ところで、ジェネリックプログラミングの文脈では「任意の長さの型の並び」が要求される場面が頻繁にあります。0x 以前の C++ は、これを直接表現するための言語機能を持っていませんでした。そのため、例えばタプルを表すクラスの宣言は、以下のように行われていました。

```
template <
  class T0 = null_type,
  class T1 = null_type,
  class T2 = null_type,
  class T3 = null_type,
  class T4 = null_type,
  class T5 = null_type,
  class T6 = null_type,
  class T7 = null_type,
  class T8 = null_type,
  class T9 = null_type>
class tuple;
```

「空」を表す型を用いて、擬似的に任意個の引数を扱っています。

プリプロセッサを用いて、パラメータ数の異なる宣言や定義を、必要な数だけ生成することもありました。以下は、Boost のプリプロセッサライブラリを用いて記述された、関数のように呼び出し可能なオブジェクト（ファンクタ）クラスの定義の例です。

```
template<
  typename R BOOST_FUNCTION_COMMA BOOST_FUNCTION_
TEMPLATE_PARMS>
class function<BOOST_FUNCTION_PARTIAL_SPEC>
  : public BOOST_FUNCTION_FUNCTION<
      R BOOST_FUNCTION_COMMA BOOST_FUNCTION_
TEMPLATE_ARGS> { ... };
```

これらの手法は大体うまく機能しますが、どちらも記述は煩雑で、またパラメータ数は一定数に制限されてしまいます。

テンプレート引数も、関数の可変長引数ように、自然に扱いたい。それを実現するのが Variadic Template です。

Variadic Templates 入門

任意個のテンプレート引数を受け取ることのできるテンプレートを、Variadic Templates と呼びます。テンプレートパラメーターを、省略記号 "..." で修飾することで、そのパラメーターに任意個の引数を格納することができるようになります。典型的な Variadic Templates を用いたクラステンプレートは、以下のように宣言されます。

```
template <typename... Types>
class variadic_class;
```

省略記号 "..." で修飾されたテンプレートパラメーター Types は、template parameter pack と呼ばれます。template parameter pack は、通常のテンプレートと同じように引数を格納することができます。

```
variadic_class<char> c_char;
variadic_class<int, float, double>
  c_int_float_double;
```

parameter pack に格納された引数を展開するには、宣言時同様、省略記号 "..." で修飾します。

```
variadic_class<Types...>
```

宣言時は前置修飾、利用時（展開時）は後置修飾、と覚えるのがいいでしょう。展開は、引数一つずつに対して行われます。

```
template <typename... Types>
struct function_pointer {
  auto (*fp)(Types...) -> void;
};
function_pointer<bool, int, float, double> f;
```

f は下のようなクラスのオブジェクトとして扱われます。

```
struct function_pointer<bool, int, float, double>
{
  auto (*fp)(bool, int, float, double) -> void;
};
```

Variadic Template を用いると、タプルクラスは次のように宣言することができます。

```
template <typename... Elements>
class tuple;
```

クラステンプレート function_pointer のメンバ変数 fp の宣言には、C++0x で導入される関数型表記の新しい形式が用いられています。新しい形式を使うと、関数宣言や、関数ポインタ変数の宣言と初期化、関数型は、以下のように記述できます。

```
auto fun(ArgTypes...) -> ReturnType;
auto (*fp)(ArgTypes...) -> ReturnType
  = &fun;
std::function<auto (ArgTypes...)
  -> ReturnType> functor=fp;
```

関数型言語に慣れ親しんでいる人にとっては、こちらの方が自然に見えるかもしれませんね。

template parameter pack への型の格納

前述の通り、template parameter pack への型の格納は、通常のテンプレートと同じように行うことができます。

```
tuple<int, double> int_double_tuple;
tuple<float, std::string, std::vector<int>>
float_string_vector_of_int_tuple;
```

0 個の引数を扱うこともできます。

```
tuple<> empty_tuple;
```

0 個の引数を許容したくない場合には、以下のような宣言で引数の数を制限することができます。

```
template <typename Head, typename... Rest>
class more_than_one;
more_than_one<int> one;
// error: wrong number of template arguments
// (0, should be 1 or more)
more_than_one<> empty;
```

デフォルト引数を与えることはできません。

```
// error: template parameter pack 'Types'
// cannnot have a default argument
template <typename... Types = int>
```

よって、template parameter pack を持つクラステンプレートの山括弧を省略することはできません。

```
// error: missing template arguments
// before 'invalid'
some_variadic_class_template invalid;
```

クラステンプレート tuple のパラメーター Types は、あくまで型を格納する template parameter pack でした。そのため、値を格納することはできません。

```
// error: type/value mismatch at argument 1
tuple<1> one_tuple;
```

ですが、通常のテンプレート引数が一部の型の値を扱えるように、template parameter pack でも値を扱うことができます。

```
template <int... values>
class int_list;
int_list<1> one;
int_list<1, 2, 3> one_two_three;
int_list<> empty;
```

例えば型レベルの文字列を表すクラスは、以下のように宣言することができます。

```
template <char... sequence>
struct compiletime_string;
```

また、クラステンプレートを扱うこともできます。

```
template <template <typename T, typename Alloc>
          class... Containers>
```

なかなか使い道の難しい機能ですね。

template parameter pack に格納された引数の展開

parameter pack に格納された引数は、省略記号 "..." で後置修飾することで、その場に展開することができます。

```
pattern_include_parameter_packs ...
```

parameter pack を含む部分式を省略記号 "..." で後置修飾した式を pack expansion と呼びます。修飾される部分式は pack expansion の「パターン」と呼びます。

pack expansion による template parameter pack の展開の利用例として、タプルクラスの定義には以下のようなものが考えられます。

```
// primary template declaration
template <typename... Elements>
class tuple;

template <class Head, typename... Tail>
class tuple<Head, Tail...>
  : public tuple<Tail...> {
  ...
};

template <>
class tuple<> {
  ...
};
```

プライマリテンプレートは、クラステンプレート tuple が任意長の型引数を取る、ということを表しています。更にテンプレートの特殊化により、型引数の個数が 1 以上の場合と、0 の場合の定義を別々に行っています。型引数が N 個のタプルは、型引数が N - 1 個のタプルを実装のために継承しています。継承するタプルクラスのテンプレート引数を指定する際に pack expansion が利用されています。

- 型引数が N - 1 個のタプルが実装されていれば、型引数が N 個のタプルが実装できる。
- 型引数が 0 個のタプルが実装されている。

上記の二点から、帰納的に 0 以上の全ての要素数のタプルが実装できていることが分かりますね。ちなみに、C++0x では std::tuple として標準で用意されているので、自分でタプルを実装する必要はありません。

pack expansion による値引数の展開例として、型レベルの文字列から文字列リテラルへの変換は、以下のように実装できます。

```
template <char... sequence>
struct compiletime_string {
  static char str[];
};

template <char... sequence>
char compiletime_string<sequence...>::str[]
  = { sequence... };
```

可変長引数関数テンプレート

C から存在する言語機能として、可変長引数関数があります。

```
void unsafe_variadic_fun(int arg, ...);
```

しかし、C の可変長引数関数の実装には、いくつか問題があります。

- 実装の都合上、最低でも 1 つの引数が必要。
- 型安全でない

C++0x では、Variadic templates を利用することで、型安全な可変長引数関数テンプレートを記述することができます。

```
template <typename... Args>
void variadic_fun(Args... args);
```

pack expansion を型に持つパラメーターを、function parameter pack と呼びます。上記の args は function parameter pack と呼ばれ、任意個の引数を格納することの出来るパラメーターとして扱うことができます。可変長引数関数テンプレートは、C 形式の可変長引数関数と同じように呼び出すことができます。

```
variadic_fun(1);
variadic_fun(1, 3.0, true);
```

0 個の引数を扱うこともできます。

```
variadic_fun();
```

function parameter pack も template parameter pack 同様、pack expansion による展開を行うことができます。

```
template <typename... Args>
void variadic_function(Args... args){
  another_variadic_fun(args...);
}
```

通常、可変長引数関数テンプレートは、再帰的に関数呼び出しを行うことで、引数それぞれに対する操作を行います。以下は、全ての引数を出力する関数テンプレート variadic_print の実装例です。

```
template <typename Arg>
void variadic_print(Arg arg){
  std::cout << arg << std::endl;
}

template <typename Head, typenaem... Tail>
void variadic_print(Head head, Tail... tail){
  variadic_print(head);
  variadic_print(tail...);
}
```

parameter pack の長さの取得

template parameter pack と function parameter pack の二つを称して parameter pack と呼びます。parameter pack の長さは sizeof... 演算子により取得することが出来ます。

```
size_t type_len = sizeof...(Types);
const size_t args_len = sizeof...(args);
```

sizeof... 演算子を知ったことで、variadic_print 関数を以下のように書きたくなるかもしれません。

```
template <typename Head, typenaem... Tail>
void variadic_print(Head head, Tail... tail){
  std::cout << head << std::endl;
  if (sizeof...(tail) > 0) return;
  // error: no matching function for call
  // to 'variadic_print()'
  variadic_print(tail...);
}
```

多くの人の直感に反して、このコードはコンパイルすることができません。実際には呼ばれなかったとしても、引数が 0 個の variadic_print 関数テンプレートが、実体化されようとするためです。引数が 0 個の variadic_print 関数は存在しませんから、コンパイルエラーになるわけですね。

parameter pack の制限

parameter pack は、どこでも宣言できるわけではありません。例えば、プライマリテンプレートの template parameter pack は、最後尾以外に書くことはできません。

```
// error: parameter pack 'Args' must be at
// the end of the template parameter list
template <typename... Args, typename Last>
```

この制限から、プライマリテンプレートは、複数の template parameter pack を含むことができないということが分かります。

```
// error: parameter pack 'Args0' must be at
// the end of the template parameter list
template <typename... Args0, typename... Args1>
```

関数テンプレートでは、同様の制限はありません。テンプレート引数の導出があるためです。

```
template
<typename... LhsElems, typename... RhsElems>
void fun(std::tuple<LhsElems...> lhs,
         tuple<std::RhsElems...> rhs);
```

また、クラステンプレートの部分特殊化においても同様です。

```
template <typename Lhs, typename Rhs>
struct concat_tuple;

template <typename... LhsTypes,
          typename... RhsTypes>
struct concat_tuple<
  std::tuple<LhsTypes...>,
  std::tuple<RhsTypes...>> { ... };
```

しかし function parameter pack は、プライマリテンプレートにおける template paramete pack 同様、最後尾以外に書くことはできません。

```
// error: parameter packs must be at the end of
// the parameter list
template <typename Last, typename... Args>
void fun(Args... args, Last last);
```

pack expansion を配置できる箇所

pack expansion も parameter pack 同様、記述できる場所は限られ

ています。

・関数呼び出しの際の引数部

```
comma_delimited_print(args...)
expand_left(args..., 1, 2, 3);
expand_right(1, 2, 3, args...);
expand_center(1, 2, 3, args..., 4, 5, 6);
expand_twice(args..., args...);
```

・initializer list

```
int is[] = { nums... };
struct { int i; float f; double d; }
  noname_struct = { mems... };
std::vector<int> v = { nums... };
std::vector<std::string> list_of_string
  = { "start", strings..., "end" };
```

・ラムダ（無名関数）のキャプチャリスト

```
[args...]() -> { ... };
[=, &args...]() -> { ... };
```

・テンプレートの引数部や関数型の引数部など、型の一部

```
tuple<Elems...> tuple;
class tuple<Head, Tail...>
  : public tuple<Tail...> { ... };
friend class C<Types...>;
manipulate_tuple_fun<Elems...>(t);
auto apply_handler(my_handler_t h, Args... args)
  -> my_handler_result_t;
using my_handler_t = auto (*)(Args...)
  -> my_handler_result_t;
```

・ベースクラスリストと、初期化子リストにおけるベースクラスの初期化子

```
class Deriv : public Classes... {
public:
  Deriv() : Classes()... { }
};
```

・attribute

```
char buf [[ align(Types...) ]] [N];
```

・例外仕様

```
void throwable() throw(Exceptions...);
```

上記の箇所以外で利用することはできません。

型の一部の最後の例では、C++0x で導入される alias declaration を利用しています。alias declaration は、typedef のように、型の別名を宣言するものです。

```
using dictionary
  = std::unordered_hash<std::string,
                        std::string>;
```

typedef との記法以外の違いとして、template により型引数を取ることができます。tempalte により型引数を取ることのできる alias declaration を、template alias と呼びます。

```
template <typename T> using id_dictionary
  = std::unordered_hash<T, T>;
```

特殊化することは出来ませんが、どうしても必要な場合には、クラステンプレートによる型関数を挟むことで解決できます。

```
template <typename... T>
using my_type_alias = impl<T...>::type;
```

ちなみに今回の記事では、typedef は使わず using による alias declaration を利用しています。読みやすいので……

複雑な pack expansion

もう一度 pack expansion の定義をおさらいしましょう。

```
pattern_include_parameter_packs ...
```

parameter pack を含む部分式を、省略記号 "..." で後置修飾した式が pack expansion でした。ところで、parameter pack を含む「部分式」というのは、どういうことでしょうか？これに関しては、実際のコードを見るのが早いでしょう。

```
fun_having_reference_arguments(&args...);
```

参照演算子が適用されているように見えるかも知れませんが、parameter pack は値や変数ではありませんから、アドレスを取ることはできません。この場合、パターンは "&args" になり、pack expansion は以下のように展開されます。

```
fun_having_reference_arguments(
  &arg0, &arg1, ... , &argN);
```

展開されているのは parameter pack ではなくパターンだということが分かりますね。パターンは柔軟な記述が可能です。例えば、関数を適用することができます。

```
print_nums(to_int(args)...);
```

to_int は関数テンプレートかもしれませんが、勿論関数テンプレートも問題なく適用できます。単なる関数呼び出し以外にも、cast したり、typeid とメンバ関数呼び出しを組み合わせたり、演算子を適用したり、何でもできます。

```
logical_and(static_cast<bool>(args)...);
print_types(typeid(args).name()...);
pool_objects(new Types()...);
print_nums(3 * nums + 1 ...);
```

最後の例は少し分かり辛いかもしれませんが、nums が parameter pack になっています。1 の後にスペースをいれないと浮動小数リテラルであるとパースされてしまいます。気をつけましょう。

```
// error: too many decimal points in number
print_nums(3 * nums + 1...);
```

関数適用時の引数部以外でも、同様のパターンの記述が可能です。

```
vector<list<int>> vector_of_list_of_int
  = {{nums}...};
using removing_cv_types_tuple
  = tuple<
      typename std::remove_cv<Types>::type...>;
auto get_rvalue_ref(&&Args... args) -> void;
```

定義が示すように、パターンは長さの等しい複数の parameter pack を含むことができます（pack が複数形になっています）。

```
template <typename... Args>
auto convert_and_print(Args... args) -> void {
  print_nums(to_int<Args>(args)...);
}
```

長さの異なる parameter を展開することはできません。

```
template <int... indexes, typename... Args>
auto choice(Args... args) -> std::vector<int> {
  return { *(args.begin() + indexes)... };
}

// error: mismatched argument pack lengths
// while expanding '*(args.begin() + indexes)'
choice<0, 1, 2>({0}, {0, 1});
```

パターンに別の pack expansion を含ませることもできます。例えば、複数の基底クラスを取る CRTP なクラステンプレートを記述することができます。

```
template <template <typename T> class... CRTPs>
class Deriv
  : public CRTPs<Deriv<CRTPs...>>... { ... };
```

少し難しいですね。このような書き方は C++ では出来ませんが、二行にわけてみましょう。

```
using self = Deriv<CRTPs...>;
class Deriv : public CRTPs<self>... { ... };
```

こうしてみると、元の見た目に比べてやっていること自体は簡単ですね。単にクラステンプレートに自身の型を渡しているだけでした。

parameter pack を含む関数テンプレートのオーバーロード解決

function parameter pack をパラメーターに含まない関数テンプレートは、パラメーターに含む関数テンプレートより特殊になります。

```
template <typename T> void f(T);
template <typename... Args> void f(Args...);

f();     // call f(Args...)
f(1);    // call f(T)
f(1, 2); // call f(Args...)
```

また、function parameter pack を含む関数テンプレート同士では、parameter pack に格納される引数が少ないほど特殊になります。

```
template <typename T> void f(T);
template <typename... Args> void f(Args...);
template <typename T, typename... Args>
  void f(T, Args...);

f();     // call f(Args...)
f(1);    // call f(T)
f(1, 2); // call f(T, Args...)
```

これは、部分的には特殊な部分を持つ関数テンプレートがある場合でも同様です。

```
template <typename... Args>
void f(std::string, Args...);
template <typename T,
          typename U,
```

```
          typename... Args>
  void f(T, U, Args...);

// call f(T, U, Args...),
// not f(std::string, Args...)
f("text", 0);
```

template parameter pack を含む型をパラメーターに持つ関数テンプレート同士でも、同様のルールが適用できます。

```
template <typename T> void f(std::tuple<T>);
template <typename... Types>
void f(std::tuple<Types...>);
template <typename T, typename... Args>
void f(std::tuple<T, Args...>);

// call f(std::tuple<Types...>)
f(std::make_tuple());
// call f(std::tuple<T>)
f(std::make_tuple(1));
// call f(std::tuple<T, Types...>)
f(std::make_tuple(1, 2));
```

パラメーター数が少ない程特殊なのは、template parameter pack でも変わりません。

```
template <typename... Types>
void f(std::tuple<Types...>,
       std::tuple<Types...>);
template <typename... Lhs, typename... Rhs>
void f(std::tuple<Lhs...>, std::tuple<Rhs...>);

// f(std::tuple<Types...>,
//   std::tuple<Types...>);
f(std::make_tuple(), std::make_tuple());

// f(std::tuple<Types...>,
//   std::tuple<Types...>);
f(std::make_tuple(1), std::make_tuple(1));

// call f(std::tuple<Lhs...>,
//        std::tuple<Rhs...>);
f(std::make_tuple(true), std::make_tuple(1));
```

template parameter pack が空の時は、より特殊な関数が選ばれます。

```
template <typename... Types>
void f(std::tuple<Types...>);
template <typename... Types>
void f(std::tuple<const Types...>);

// call f(std::tuple<const Types...>)
f(std::make_tuple());
```

> 実際には、関数テンプレートのオーバーロードは、解決の前にテンプレートパラメーターの deductin が行われます。そのため、引数を指定しない場合のオーバーロードを理解するには、まずその点を理解する必要があります。ですが、この記事では重箱の隅をつつくつもりはないので、今回は省略しました。細かい部分は、コンパイラの実装が正しいか分からないので確認が難しい、という事情もあります……

制約付きの可変長引数関数テンプレートを定義する

例えば引数の総和を求める可変長引数関数テンプレートを書くとき、non-type template parameter pack のように書きたいと思うかもしれません。

```
template <int head, int... tail>
struct sum { ... };
// error: expansion pattern ‘int’ contains
// no argument packs
int sum(int head, int... tail);
```

int は template parameter pack ではなく、ただの型です。そのため省略記号 "..." で修飾することはできません。では template parameter pack を使って実装してみましょう。

```
int sum(int head){
  return head;
}

template <typename... Tail>
int sum(int head, Tail... tail){
  return head + sum(tail...);
}
```

これは大体うまく機能しますが、引数の暗黙変換を許してしまいます。

```
int s = sum(1, 2.0, false);
```

暗黙変換を許可したくない場合、明示的に引数型に対する制約を書く必要があります。制約の記述には、C++0x で導入されるコンパイル時の assertion を実現する言語機能 static_assert が利用

できます。

```
template <typename... Tail>
int sum(int head, Tail... tail){
  static_assert(constraint<Tail...>::value,
                "all argument types need int");
  return head + sum(tail...);
}
```

例えば constraint は以下のように実装することができるでしょう。

```
template <typename... Types>
struct constraint {
  static const bool value = false;
};

template <>
struct constraint<> {
  static const bool value = true;
};

template <typename ...Rest>
struct constraint<int, Rest...> {
  static const bool value
    = constraint<Rest...>::value;
};
```

このような型レベルの述語を毎回書く必要があるのは、大変な苦痛です。また、読む側も耐えられないでしょう。template parameter pack に格納されている型が、任意の述語を満たしているかどうかを返す高階述語を定義してしまいましょう。

```
template <template <typename T> class Pred,
          typename... Types> struct all_of;
template <template <typename T> class Pred>
struct all_of {
  static const bool value = true;
};
template <template <typename T> class Pred,
          typename Head, typename... Tail>
struct all_of {
  static const bool value
    = Pred<Head>::value &&
      all_of<Pred, Tail...>::value;
};
```

all_of クラステンプレートを利用して、constraint クラステンプレートをもう一度実装してみます。

```
template <typename... Types>
struct constraint {
  template <typename T>
  struct is_int
  { static const bool value = false; };
  template <>
  struct is_int<int>
  { static const bool value = true; };
  static const bool value
    = all_of<is_int, Types...>::value;
};
```

最初の実装に比べると、読むのも書くのも易しくなりましたね。しかし、この all_of 述語の実装には少し問題があります。それは、論理演算子が short circuit として動作しないことです。例えば、以下のコードをコンパイルすることはできません。

```
template <typename T>
struct fail_pred
{ static_assert(sizeof(T) && false, "fail"); };
template <>
struct fail_pred<int>
{ static const bool value = false; };
// error: static assertion failed: "fail"
bool b = all_of<fail_pred, int, double>::value;
```

例え論理積演算子の左辺が false でも、述語が全てのテンプレート型引数に対して instantiation されてしまうためです。述語の instantiation を遅らせて、short circuit として機能する論理積演算子を実装してみましょう。

```
template <bool Lhs, typename Rhs>
struct logical_and_impl
{ static const bool value = false; };
template <typename Rhs>
struct logical_and_impl<true, Rhs>
{ static const bool value = Rhs::value; };

template <bool Lhs, typename Rhs>
struct logical_and {
  static const bool value
    = logical_and_impl<Lhs::value, Rhs>::value;
};

bool b = logical_and<fail_pred<int>,
                     fail_pred<double>>::value;
```

logical_and を利用すれば、述語の評価を遅延する all_of 述語が実装できます。

```
template <template <typename T> class Pred,
          typename Head, typename... Tail>
struct all_of {
  static const bool value
    = logical_and<Pred<Head>,
                  all_of<Pred, Tail...>>::value;
};
```

少し見栄えは悪くなってしまいましたが、余計な instantiation を避けるためにも、こちらの実装の方が良いでしょう。

> static_assert は非常に便利な機能ですが、利用に際して注意する点があります。常に失敗する assertion を次のように記述すると、テンプレートの instantiation されるされないに関わらず失敗するため、コンパイルできません。
>
> ```
> static_assert(false, "message");
> ```
>
> テンプレートパラメーターに依存していないため、instantiation されるよりも前に assertion が行われるためです。
> 常に失敗する、テンプレートパラメーターを含む式を記述することで、instantiation を遅延させてやる必要があります。fail_pred クラステンプレートでは sizeof 演算子を利用して、instantiation を遅延させていました。しかし sizeof 演算子は不完全型に対して適用することができません。これは場合によっては問題になります。常に false を返す述語を定義するか、標準で用意されている述語を用いるか、まだこれという定番はありません。また、パラメーターのない特殊化の場合、遅延することが出来ない問題もあります。中々難しいですね。

末尾再帰的な実装を持つ可変長引数関数テンプレートを定義

現実には sum 関数テンプレートは、二項演算 "+" が定義されている型に対して、ジェネリックに実装したくなるはずです。

```
template <typename T>
auto sum(T t) -> T { return t; }

template <typenaem Head, typename... Rest>
auto sum(Head head, Rest... rest) -> ReturnType {
  return head + sum(rest...);
}
```

さて、ReturnType はどのように記述すべきでしょうか？ C++0x では、新しい言語機能として、式の型を取得する "decltype" が導入されました。

```
decltype(1 + 2) i = 1 + 2;
decltype(fun(param)) ret = fun(param);
```

これをこれを利用すれば、比較的素直に書けるはずです。

```
auto sum(Head head, Rest... rest)
  -> decltype(head + sum(rest...));
sum(0);
sum(0, 2.2f);
```

ここまでは問題がないように見えます。残念なことに、このコードは引数が三つ以上になると、突然コンパイルが通らなくなります。

```
// error: no matching function for
// call to 'sum(int, float, double)'
sum(0, 2.2f, 4.0);
```

何故関数引数が三つになると、コンパイルが通らなくなるのでしょうか？関数は通常、自分自身の名前や宣言を、その定義の中から参照することができます。これは、関数テンプレートにおいても同様です。しかし、decltype は関数テンプレートのシグネチャとして記述されています。定義ではありません。そのため、decltype 内の "sum" からは 1 引数の関数テンプレートしか見つけられません。その結果が「int, float, double を取る関数 sum は見つからない」といったエラーメッセージにつながるわけです。

関数テンプレートの instantiation のタイミングを考えると、これは非常に解しがたい仕様です。実用的にも、末尾再帰やそれに近い実装をもつ関数テンプレートで decltype が使えないのは、惜しい話です。何にせよ、自身の名前をシグネチャに含めることができない以上、他の実装方法を考える必要があります。通常はクラステンプレートの static 関数として実装し、インターフェースとして橋渡しを行う関数テンプレートを用意することになるでしょう。

```
template <typename... Types>
struct sum_impl;

template <typename T>
struct sum_impl<T> {
  static auto apply(T t) -> T { return t; }
};

template <typename Head, typename... Rest>
struct sum_impl<Head, Rest...> {
  static auto apply(Head h, Rest... rest) ->
     decltype(h,
              sum_impl<Rest...>::apply(rest...))
  {
    return h + sum_impl<Rest..>::apply(rest...);
```

```
  }
};

template <typename... Args>
auto sum(Args... args) ->
  decltype(sum_impl<Args...>::apply(args...)) {
  return sum_impl<Args...>::apply(args...);
}
```

元のコードに比べると、随分冗長なコードになってしまいましたが、これで実装することが出来ました。返値型が意図した通りの結果になっているかどうか、以下のコードで確認することができます。

```
static_assert(
  std::is_same<
    double,
    decltype(sum(1, 4.0f, 2.3))
  >::value,
  "not double");
```

is_same は標準ライブラリで定義されている、型の比較を行う述語です。std::string も、文字列の連結として二項演算 "+" を実装していますから、適用できるはずです。

```
static_assert(
  std::is_same<std::string,
               decltype(
                 std::string("hello") +
                 "generic" +
                 "world")
  >::value,
  "not std::string");
// error: invalid operands of types
// ‘const char*’ and ‘const char*’ to
// binary ‘operator+’
static_assert(
  std::is_same<std::string,
               decltype(
                 sum(std::string("hello"),
                     "generic", "world"))
  >::value,
  "not std::string");
```

assertion が行われるでもなく、コンパイルエラーになってしまいました。これは sum 関数テンプレートが、右結合として実装されてしまっているためです。通常の二項演算 "+" は左結合です。明示的に括弧をつけると、以下のようになります。

```
(std::string("hello") + "generic") + "world"
```

一方で、現在の sum 関数テンプレートの実装は、以下のように処理してしまっています。

```
std::string("hello") + ("generic" + "world")
```

これではコンパイルは通りませんね。左結合として実装するには、少し手を加えてやる必要があります。

```
template <typename T>
struct sum_impl<T> {
  static auto apply(T t) -> T { return t; }
};
template <typename Lhs,
          typename Rhs, typename... Rest>
struct sum_impl<Lhs, Rhs, Rest...> {
  static auto apply(Lhs lhs, Rhs rhs,
                    Rest... rest) ->
    decltype(sum_impl<decltype(lhs + rhs),
                      Rest...>
             ::apply(lhs + rhs, rest...))
  {
    return sum_impl<decltype(lhs + rhs),
                    Rest...>::apply(lhs + rhs,
                                    rest...);
  }
};
```

これで、左結合の二項演算 "+" と同等のセマンティクスを持つ関数テンプレート sum が実装できました。最初の実装と修正後の実装は、関数型言語でいうところの foldr, foldl に対応しています。多層関数をクラステンプレートとして実現することで、ジェネリックな fold が実装できます。是非実装してみてください。

tuple_apply を実装する

タプルに格納した値を、関数に適用したい場合があります。std::get 関数テンプレートを使うことで、タプルの N 番目の値を取得することができます。

```
fun(std::get<0>(t),
    std::get<1>(t), std::get<2>(t));
```

このような冗長な記述を毎回するのは面倒ですね。関数とタプルを取り、タプルに格納された値を引数として関数を呼び出す関数テンプレートを定義してしまいましょう。要素数 N のタプルに格納された値の展開は、以下のように記述できそうです。

```
std::get<0>(t),
std::get<1>(t) ... std::get<N-1>(t)
```

indexes が 0 から N-1 までの値を持つ、要素数 N の template parameter pack であるとすると、これは以下のように記述することができます。

```
std::get<indexes>(t)...
```

std::tuple<Types...> から、このような parameter pack を作ることができるでしょうか？まずは、型レベルで整数を格納するための型を定義しましょう。

```
template <int... indexes>
struct indexes_list {};
```

次に、整数 N を取り、0 から N-1 までの値をパラメーターとして持つ indexes_list を返す型レベルの関数を定義する必要があります。仮に関数の名前が range であるとすると、range の振る舞いはどのようになるでしょうか？擬似コードで記述すると、range の振る舞いは以下のようになります。

```
range(0)   = { }
range(1)   = { 0 }
range(2)   = { 0, 1 }
range(N-1) = { 0, 1, ... N-2 }
range(N)   = { 0, 1, ... N-2, N-1 }
```

こうしてみると、range(N) は range(N-1) に N-1 を連結したもの、と見ることが出来そうですね。

```
range(0)   = { }
range(1)   = range(0) ~ { 0 }
range(2)   = range(1) ~ { 1 }
range(N-1) = range(N-2) ~ { N-2 }
range(N)   = range(N-1) ~ { N-1 }
```

これをそのまま実装してみましょう。

```
template <typename T, int N>
struct push;

template <int... indexes, int N>
struct push<indexes_list<indexes...>, N> {
  using type = indexes_list<indexes..., N>;
};

template <int N>
struct range_impl {
  using prev = range_impl<N-1>::type;
  using type = push<prev, N-1>::type;
};

template <>
struct range_impl<0> {
  using type = indexes_list<>;
};

template <typename T> using range =
  typename range_impl<T>::type;
```

range を利用することで、0 から始まる長さ N の範囲を表す indexes_list 型を得ることができます。

```
static_assert(
  std::is_same<
    indexes_list<0, 1, 2>,
    range<3>>::value,
  "same");
```

これで tuple_apply が実装できそうですね。

```
template <typename Fun, typename... Args,
          int... indexes>
auto tuple_apply(Fun fun,
                 std::tuple<Args...> t) ->
  decltype(fun(std::get<indexes>(t)...)) {
  return fun(std::get<indexes>(t)...);
}
```

いきなりこう書きたいところですが、このままでは、indexes を推論することができません。まずは引数に indexes_list を取るようにしましょう。値である必要はないので、ポインタ型にします。

```
template <typename Fun, typename... Args,
          int... indexes>
auto tuple_apply_impl(
  Fun fun,
  std::tuple<Args...> t,
  indexes_list<indexes...>* dummy) ->
    decltype(fun(std::get<indexes>(t)...)) {
  return fun(std::get<indexes>(t)...);
}
```

あとはこの関数テンプレートにダミーの値を渡す関数テンプレートを記述すれば完成です。decltype の中が少し汚くなってしまいますが、以下のように実装することができます。

```
template <typename Fun, typename... Args>
auto tuple_apply(Fun fun,
                 std::tuple<Args...> t) ->
  decltype(
    tuple_apply_impl(
      fun, t,
      static_cast<
        range<sizeof...(Args)>*>(nullptr)))
{
  range<sizeof...(Args)>* dummy = nullptr;
  return tuple_apply_impl(fun, t, dummy);
}
```

これで実装できました。実際に使ってみましょう。

```
// hello C++0x world!
tuple_apply(printf,
            std::make_tuple("hello %s world!\n",
                            "C++0x"));
```

実装の「肝」は tuple_apply_impl の次の一行です。

```
return fun(std::get<indexes>(t)...);
```

たった一行のコードですが Variadic Templates の強力さが十分に表れています。

> tuple_apply 関数テンプレートの返値型の記述が、あまり綺麗ではなくなってしまっていました。こんなときは C++0x で新しく標準ライブラリとして提供される、std::result_of クラステンプレートが利用できます。
>
> ```
> auto tuple_apply(
>   Fun fun,
>   std::tuple<Args...> t) ->
>   typename
>   std::result_of<Fun(Args...)>::type;
> ```
>
> result_of クラステンプレートは、関数のように振る舞う型と引数の型から、返値の型を求めることができます。

おわりに

説明を簡潔に簡潔に、と思いながら書いてたらコードばかりの記事になってしまいました。人間の言葉は難しいので、それでよかったのかもしれません。そのかわりそんなに実践的じゃなくなってしまった感も。

参考文献は大体 n3092 の Final Commitee Draft です。他にも参照した paper があるのですが、どの番号がどれか分からないので困りました……

Prolog で Scheme の操作的意味論を実装

著 : zick

本記事では、Scheme の操作的意味論を Prolog の述語として定義することで、インタプリタを 実装する方法を解説します。これが役に立つのかは私自身もよく分かっていませんが、「こんな実装方法もある」と頭の片隅に置いていただければ幸いです。

Scheme と言えば、仕様書に形式的意味論が載っているということで（ごく一部で）有名ですが、R5RS[1] では形式的意味論は 3 ページ程しか載っていません。しかし、R6RS[2] ではなんと 14 ページもあります。R6RS では形式的意味論として操作的意味論を用い、複雑な制御の流れを実現する call-with-current-continuation や dynamic-wind、R6RS の新機能である例外（の一部）まで扱っています。そのため、これをそのまま実装するだけで結構遊べます。

はじめは、2 年ほど前に書いた、R6RS の操作的意味論のすべてを実装した Prolog のプログラムを説明しようかと思ったのですが、行数を見たら 1000 行を超えていたので、それをすべて載せるのはあきらめました。代わりに、仕様を必要最低限（よりも更に小さい）ものに削った言語を Prolog で実装し、その後、それをふまえて R6RS で定義される本物の操作的意味論を実装する方法を述べます。対象となる読者は Lisp 系の言語と Prolog の両方の知識がある程度ある方なので、Scheme や Prolog 自身の説明は一切ありません。知らない人は気合いで読んで下さい。また、Prolog 処理系にはリストの連結を行う append/3、リストの長さを取得する length/2、リストに特定の要素が含まれるか確認する member/2 が定義されているものとします。

ひんそーな言語 YukihoScheme

。本節では、R6RS の操作的意味論から、多くの機能を削りすぎて、ひんそーになった言語 YukihoScheme を定義しながら、それを Prolog で実装していきます。

構文

。YukihoScheme の構文を定義します。値は数と真偽値と手続きの 3 種類。組み込みの手続きは **+** と **eqv?** の 2 種類。構文は **begin**、**if**、**lambda** の 3 種類を用意します (実のところ、副作用がないので、begin は意味を持ちません)。

P ::= e
A ::= v
R ::= v
e ::= (begin e ...) | (if e e e) | (e e ...) | proc | sqv
sqv ::= n | #t | #f
proc ::= + | eqv? | (lambda f e e ...)
f ::= (x ...)
v ::= sqv | proc
x ::= [variables except keywords]
n ::= [numbers]
keyword ::= begin | if | lambda

P は入力として受け付けるプログラム、**A** はプログラムの実行結果、**R** は観測できる（外に見せる）結果です。YukihoScheme では **A** と **R** は同じものですが、R6RS の定義では両者は異なります。パーサを作るのは面倒なので、式は Prolog のリストとして受け取ることにします。例えば、

```
((lambda (x) (if (eqv? x 0) 1 0)) 1)
```

は

```
[[lambda,[x],[if,['eqv?',x,0],1,0]],1]
```

のように表現します (? はそのままでは Prolog のアトムにできないので引用符で囲みます。`#t` と `#f` も同様です)。

ではこの構文を Prolog で実装してみましょう。何も考えずに述語を作るだけです。なお、**e** の 0 回以上の出現を意味する **e ...** のような記法は、**e_s** のように **_s** を付加した名前の述語にしています。

```
p(X) :- e(X).
a(X) :- v(X).
r(X) :- sqv(X), !.
r(X) :- proc(X).
e_s([]) :- !.
e_s([H|T]) :- e(H), e_s(T).
e([begin|T]) :- e_s(T), !.
e([if|T]) :- e_s(T), !.
e(X) :- e_s(X), !.
e(X) :- proc(X), !.
e(X) :- sqv(X), !.
e(X) :- x(X), !.
e([H|T]) :- e(H), e_s(T).
sqv(X) :- n(X), !.
sqv('#t') :- !.
sqv('#f').
proc('+') :- !.
proc('eqv?') :- !.
proc([lambda,F,E|Es]) :- f(F), e(E), e_s(Es).
f(X) :- x_s(X).
x(X) :- not(keyword(X)), atom(X).
x_s([]) :- !.
x_s([H|T]) :- x(H), x_s(T).
keyword(begin) :- !.
keyword(if) :- !.
keyword(lambda).
v(X) :- sqv(X), !.
```

```
v(X) :- proc(X).
v_s([]) :- !.
v_s([H|T]) :- v(H), v_s(T).
n(X) :- number(X).
n_s([]) :- !.
n_s([H|T]) :- n(H), n_s(T).
```

評価文脈。YukihoScheme の評価文脈の文法を定義します。文脈とは穴 [] を 1 つだけ含む式のことで、例えば (+ 1 []) のような式は文脈です。(+ 1 []) を C とおくと、C の穴を E で置き換えたもの、すなわち (+ 1 E) を C[E] で表します。評価文脈とは、次に評価する部分式を穴で示した文脈です。訳の分からない説明ですが、とりあえず定義を見て下さい。

P ::= [] | (if P e e) | (begin P e e ...) | (v ... P v ...)

ここから分かるのは、次に評価する部分式とは、

- 式全体
- **if** の第 1 引数
- **begin** の第 1 引数
- ある引数以外すべて評価済みである手続き呼び出しの式の、評価されていない引数

の 4 種類だということです（keyword は **v** に含まれないことに注意して下さい）。なんだか分かるような分からないような定義ですが、では、これを Prolog で実装しましょう。

どうやって？

この時点では評価文脈をどう定義したらいいのか、ほとんどの方はすぐには思いつかないでしょう（すぐに思いついた方は、リリカル Lisp を購入する権利を差し上げます）。という訳で、評価文脈はひとまず置いておき、簡約規則の定義をします。

簡約規則。では、YukihoScheme の簡約規則を定義します。ここで、{x → y}E は式 E に出現する x をすべて y で置き換えた式を表します。また、x fresh は x が使用されていない変数であることを表し、#x ... は x の出現回数を表します。

$P_1[(+)] \rightarrow P_1[0]$ [y+0]

$P_1[(+\ n_1\ n_2\ \cdots)] \rightarrow P_1[\ \Sigma\ \{n_1,\ n_2,\ \cdots\}]$ [y+]

$P_1[(\mathrm{eqv?}\ v_1\ v_1)] \rightarrow P_1[\#t]$ [yeqt]

$P_1[(\mathrm{eqv?}\ v_1\ v_2)] \rightarrow P_1[\#f]$ [yeqf]

$P_1[(\mathrm{if}\ v_1\ e_1\ e_2)] \rightarrow P_1[e_1] (v_1 \neq \#f)$ [yif3t]

$P_1[(\mathrm{if}\ \#f\ e_1\ e_2)] \rightarrow P_1[e_2]$ [yif3f]

$P_1[(\mathrm{begin}\ v_1\ e_1\ e_2\ \cdots)] \rightarrow P_1[(\mathrm{begin}\ e_1\ e_2\ \cdots)]$ [ybeginc]

$P_1[(\mathrm{begin}\ e_1)] \rightarrow P_1[e_1]$ [ybegind]

$P_1[(e_1\ \cdots\ e_i\ e_{i+1}\ \cdots)] \rightarrow$
$P_1[((\mathrm{lambda}\ (x)\ (e_1\ \cdots\ x\ e_{i+1}\ \cdots))\ e_i)]$
s.t. x fresh, $e_i \notin v$, $\exists\ e \in e_1\ \cdots\ e_{i+1}\ \cdots$, $e \notin v$ [ymark]

$P_1[((\mathrm{lambda}\ (x_1\ x_2\ \cdots)\ e_1\ e_2\ \cdots)\ v_1\ v_2\ \cdots)] \rightarrow$
$P_1[(\{x_1 \rightarrow v_1\}(\mathrm{lambda}\ (x_2\ \cdots)\ e_1\ e_2\ \cdots)\ v_2\ \cdots)]$ [yappN]

$P_1[((\mathrm{lambda}\ ()\ e_1\ e_2\ \cdots))] \rightarrow P_1[(\mathrm{begin}\ e_1\ e_2\ \cdots)]$ [yapp0]

例えば、規則 [y+0] は「次に評価すべき部分式が (+) であれば、それを 0 で置き換える」と読みます。これを Prolog で実装しようとすると、矢印の左側を第 1 引数、右側を第 2 引数にすればよいので、次のようになりそうですね。

reduce(P1, Next) :-
P1 は評価文脈 P の穴を (+) で置き換えたもので、
Next は P1 の穴を 0 で置き換えたもの .

ここまで考えると、評価文脈を Prolog でどう実装すればいいのかアイディアが思い浮かんでくるはずです。評価文脈を表す述語の引数は以下の 4 つにすると都合が良さそうです。

- 簡約前の式
- 簡約前の穴の中身（評価される部分式）
- 簡約後の式
- 簡約後の穴の中身（簡約された部分式）

評価文脈を表す述語が、簡約後の情報を引数にとるのは不自然かもしれませんが、評価文脈が簡約の中でしか使わないことを考えると、合理的な定義かと思います。もっといい方法を知っている方は、こっそりと私に教えて下さい。それでは、実際に評価文脈を Prolog で実装してみましょう。

```
ctx_p(X, X, Y, Y).
ctx_p([if,P,E1,E2],
      Hole, [if,NP,E1,E2], NextHole) :-
  e(E1), e(E2),
  ctx_p(P, Hole, NP, NextHole).
```

```
ctx_p([begin,P|Es], Hole,
      [begin,NP|Es], NextHole) :-
  e_s(Es),
  ctx_p(P, Hole, NP, NextHole).
ctx_p(Form, Hole, Next, NextHole) :-
  append(Vs1, [P|Vs2], Form),
  v_s(Vs1), v_s(Vs2),
  ctx_p(P, Hole, NP, NextHole),
  append(Vs1, [NP|Vs2], Next).
```

最初の事実は、式全体が穴の場合を表します。式全体が簡約前の穴と一致すれば、簡約後の式は簡約後の穴と一致するという定義です。残りの規則は見ての通り、if、begin、手続き呼び出しの場合です。この定義がいまいち分からなくても、簡約規則でこれらが使われるのを見ると、自然と意味が分かってくると思います。

簡約の実装。それでは簡約規則を実装していきます。まずは、+ に関する規則から。

```
% [y+0]
reduce(P1, Next) :-
  ctx_p(P1, [+],
        Next, 0).
% [y+]
reduce(P1, Next) :-
  ctx_p(P1, [+,N|Ns],
        Next, Sum),
  n(N),
  n_s(Ns),
  sum([N|Ns], Sum).
```

[y+0] に関しては、ほぼ定義のままですね。[y+] では、P1 が定義の形を満たしているか確認した後に、補助述語 sum を使って和を求めます。sum は次のように定義します。

```
sum([], 0).
sum([H|T], Sum) :- sum(T, S1), Sum is H + S1.
```

次に、eqv? に関する規則を実装します。

```
% [yeqt]
reduce(P1, Next) :-
  ctx_p(P1, ['eqv?',V1,V1], Next, '#t'),  v(V1).
% [yeqf]
reduce(P1, Next) :-
  ctx_p(P1, ['eqv?',V1,V2], Next, '#f'),
  not(V1==V2), v(V1), v(V2).
```

これらは、ほぼ定義のままなので何も言うことは無いでしょう。次は、if と begin に関する規則。これらも定義のままです。

```
%% if
% [yif3t]
reduce(P1, Next) :-
  ctx_p(P1, [if,V,E1,E2], Next, E1),
  v(V), e(E1), e(E2), not(V=='#f').
% [yif3f]
reduce(P1, Next) :-
  ctx_p(P1, [if,'#f',E1,E2], Next, E2),
  e(E1), e(E2).

%% begin
% [ybeginc]
reduce(P1, Next) :-
  ctx_p(P1, [begin,V,E|Es], Next, [begin,E|Es]),
  v(V), e(E), e_s(Es).
% [ybegind]
reduce(P1, Next) :-
  ctx_p(P1, [begin,E], Next, E),
  e(E).
```

最後に、手続き呼び出しに関する規則を実装します。これはちょっと複雑です。簡約規則と見比べながら読んで下さい。

```
% [ymark]
reduce(P1, Next) :-
  ctx_p(P1, Es1_E_Es2, Next,
        [[lambda,[X],Es1_X_Es2],E]),
  append(Es1, [E|Es2], Es1_E_Es2),
  e_s(Es1), e(E), e_s(Es2),
  not(v(E)), append(Es1, Es2, Es), exist_nv(Es),
  gen_atom(X), append(Es1, [X|Es2], Es1_X_Es2).
% [yappN]
reduce(P1, Next) :-
  ctx_p(P1, [[lambda,[X|Xs],E|Es],V|Vs], Next,
        [[lambda,Xs,NE|NEs]|Vs]),
  x(X), x_s(Xs), e(E), e_s(Es), v(V), v_s(Vs),
  length(Xs, Len), length(Vs, Len),
  subst(V, X, [E|Es], [NE|NEs]).
% [yapp0]
reduce(P1, Next) :-
  ctx_p(P1, [[lambda,[],E|Es]],
        Next, [begin,E|Es]),
  e(E), e_s(Es).
```

[ymark] では新しい変数を作り出す必要があります。これは、gen_atom という述語に任せています。gen_atom は assert を使って実装できますが、SWI-Prolog であれば、組み込み述語である gensym を使うと何も考えずに簡単に実装できます。また、v に

含まれず、e に含まれる要素を探す述語 exist_nv も定義してやる必要があります。

```
gen_atom(X) :- gensym('#:G', X).

exist_nv([]) :- fail.
exist_nv([H|_]) :- not(v(H)), !.
exist_nv([_|T]) :- exist_nv(T).
```

[yappN] では変数を対応する値で置き換えてやる必要があります。この仕事を行う述語 subst はリスト操作として定義できます。

```
subst(X, Y, Y, X) :- !.
subst(X, Y, [H|T], [Z|Zs]) :-
  subst(X, Y, H, Z),
  subst(X, Y, T, Zs), !.
subst(_, _, Z, Z) :- !.
```

評価。式の簡約を繰り返し、それ以上簡約できなくなれば、それが式の値になります。よって、式を評価する述語は次のように定義できます。

```
eval(X, Y) :-
  reduce(X, Z), not(X==Z), !, eval(Z, Y).
eval(X, X) :- a(X).
```

これで完成です。早速遊んでみましょう。

```
?- eval([+,1,2], X).
X = 3.

?- eval([if,['eqv?',[+,1,2],[+,2,1]],99,88],X).
X = 99.

?- eval([[lambda,[x,y],y],99,88],X).
X = 88.
```

あたりまえですがちゃんと動きます。しかし、これだけだとちょっと不満です。複雑な [ymark] を実装した割には、本当にそれが活かされてるかいまいち分かりません。そこで、カットを消し去りバックトラック可能にした eval2 を作ります。簡約の途中結果を出力すると分かりやすいので次のように定義します。

```
eval2(X, Y) :-
  reduce(X, Z), not(X==Z),
  write(Z), nl, eval2(Z, Y).
eval2(X, X) :- a(X).
```

これを使って (+ (+ 1 2) (+ 3 4)) を評価してみます。

```
?- eval2([+,[+,1,2],[+,3,4]],X).
[[lambda, [#:G2], [+, #:G2, [+, 3, 4]]], [+, 1, 2]]
[[lambda, [#:G2], [+, #:G2, [+, 3, 4]]], 3]
[[lambda, [], [+, 3, [+, 3, 4]]]]
[begin, [+, 3, [+, 3, 4]]]
[begin, [+, 3, 7]]
[begin, 10]
10
X = 10 ;
[[lambda, [#:G3], [+, [+, 1, 2], #:G3]], [+, 3, 4]]
[[lambda, [#:G3], [+, [+, 1, 2], #:G3]], 7]
[[lambda, [], [+, [+, 1, 2], 7]]]
[begin, [+, [+, 1, 2], 7]]
[begin, [+, 3, 7]]
[begin, 10]
10
X = 10 ;
false.
```

最初の試行では (+ 1 2) を先に計算していますが、次の試行では (+ 3 4) を先に計算しています。「引数の評価順序は規定されていない」という Scheme の仕様を見事に表す、興味深いものが見れました。実際にはもっと色々起こりますが、面白いところだけを載せました。本当の結果は自分で実装して確かめて下さい。

本物の R6RS の仕様に合わせる。ここまでが分かれば R6RS の操作的意味論も怖くありません。YukihoScheme の仕様は基本的に R6RS の定義から色々削ってひんそーにしたものであり、逆に言えば、YukihoScheme に色々付け足せば R6RS の定義になります。しかし、YukihoScheme にはない概念がいくつかあるので、本節はそこを中心に解説します。R6RS の操作的意味論そのものについては [3] を読んでおくと、理解が深まったり、かえって混乱してしまったり、良いことや悪いことが起こる、もしくは何も起こらないこと間違いなしです。

ストア。R6RS の定義では、ストアというものが式全体を囲みます（例 : (store ((#:g1 3)(#:g2 4)) (+ #:g1 #:g2))）。ストアとは、変数の値や、コンスセルの内容を格納する場所です。YukihoScheme では手続き呼び出しのとき、仮引数を実引数で単純に置換しました。R6RS でも基本的には同じですが、破壊的代入が使われる（可能性のある）変数は特別扱いされます。実引数の値はストアの中に置かれ、仮引数はストア中の場所を表すものに置換されます。とまあ、なんだか凄そうですが、作るのは難しくありません。R6RS ではストアの内容も式の一部と見なされるので、一部の簡約規則が少し長くなるだけです。

クォート。当たり前のことですが、R6RS の定義ではクォートを扱えます。文法として、クォートが含まれない式 e とクォートを含む式 es を別々に定義し、es に関する簡約規則がクォートを取り除いてくれるので、何も考えずに実装すれば終わりです。注意すべきは、'sym (クォート付きのシンボル) が e に含まれるということです。どう見てもクォートを含んでますが、これは e です。というのも、簡約している途中の段階では、式の中にシンボル（のようなもの）を見つけたときに、それが変数なのか、シンボルなのか区別できないため、シンボルには常にクォートを付けるんです。クォートを外す規則の中に「シンボルについているクォートを外す」という規則はありません。なんだか変な気もしますが、そういう仕様だから仕方ありません。

3種の穴。R6RS の評価文脈の穴は [] []* []o という 3 種類があります。評価文脈は 3 種類の中のいずれか 1 つのみを含みます。このように、穴が 3 種類あるのは多値を扱うためです。[]* は多値のみを扱う場面で使用する穴で、[]o は単一値のみを扱う場面で使用する穴です。次の 2 つの規則が使い方を端的に示しているかと思います。

```
P1[v1]* -> P1 [(value v1)]
P1[(values v1)]o -> P1 [v1]
```

これを実装するためには、評価文脈を表す述語の引数を 1 つ増やし、穴の種類を受け取るようにする必要があります。この記事を読んで R6RS の操作的意味論を Prolog で実装する場合、おそらくこの穴の扱いが厄介になるかと思います。作りたい方は気合で頑張って下さい。

おわりに。「Prolog で操作的意味論を実装」というネタを私が実際にやったのは、もう 2 年以上前のことで、正直ほぼすべてを忘れていました。そのため、[3] を眺めながら、なんとか内容を思い出し、ここまで記事を書くことができました。「どう考えても誰の役にも立たないだろう」と思って書いたものが、数年後に自分自身の役に立つとは思ってもいませんでした。この記事を書くために新たに作った YukihoScheme は、加算があるのに減算がなかったり、そもそもコンスセルを扱えなかったり突っ込みどころ満載です。しかし、lambda があればコンスセルはシミュレートできます（だし、変数の捕捉を避ける必要があります（実装としては subst に規則を 1 つ追加するだけで `subst(_, Y, [lambda,F|T], [lambda,F|T]) :- member(Y,F).` となります）。
x = a + b と x + b = a が等価であることを使えば、引き算も作れます（ただし、素直に再帰が書けないので、不動点演算子を作る必要があります）。実際のところ、雪歩がひんそーではないのと同様に、YukihoScheme も（それほど）ひんそーではないんです。暇で死んでしまいそうな方はぜひお試しください。

最後になりましたが、ネタを提供してくれた後輩、挿絵を描いてくれた兄、この記事を書く機会を与えてくれた m0h1can さんに感謝。

参考文献

[1] R. Kelsey, W. Clinger, J. Rees (eds.): Revised5 Report on the Algorithmic Language Scheme,
Higher-Order and Symbolic Computation, Vol. 11, No. 1, August, 1998
[2] M. Sperber, R. K. Dybvig, M. Flatt, A. V. Straaten, R. Findler and J. Matthews.: Revised6
Report on the Algorithmic Language Scheme, Journal of Functional Programming, Vol. 19,
Supplement S1, August 2009, pp 1-301
[3] はてなようせいとまなぶ Scheme の形式的意味論 , http://lambda.bugyo.tk/hatena/

ゲームオーバーのすゝめ

著 : mascalade

最近みなさんはハマるゲームに出会っていますか？私はファミコン時代から約 20 年ほどゲームを楽しんでいるが、ここ 2，3 年でハマった！と胸を張って言えるタイトルがずいぶん減ったように思える。学生で時間があったからだと言われるかもしれないが、スーパファミコン、プレイステーション (無印) で遊んでいたころは、毎日ゲームをすることはもちろん、2 周目 3 周目といった周回プレイは当たり前だった。それこそクロノトリガーはマルチエンディングだったとはいえ、何回周回プレイを続けたか分からないほどだ。

人によってハマるの定義は様々だと思うので、最初に定義しておこうと思う。
・ほぼ毎日のように遊ぶ。
・1 回のプレイ時間が 2 ～ 3 時間以上。
・遊びたいという能動的な意思のもとプレイする。
(誰かと競っていたり、ソフトを返さないといけない、といった外部要因でプレイしたものは省く)
断りをいれる必要もないと思うが、この定義はあくまでも私のハマる定義であり、一般的な定義ではない。

おそらくこの数年で一番ハマったソフトは『STEINS;GATE』で二番手は『うみねこのなく頃に』だろう。この 20 年のゲーム人生で、しかも JRPG 大好きな私がハマったゲームのトップ 2 がどちらもサウンドノベルズゲームだとは…私自身衝撃を受けた。もちろん、これは私が空想技術ネタ大好き (空想科学読本愛読者) でライトミステリー大好き (金田一少年の事件簿や名探偵コナン愛読者) だからこそかもしれないが、それでもサウンドノベルズゲームが一番に来るのはなんとなく寂しい。もちろんサウンドノベルズゲームもゲームの一種ではあるが、その面白さは文章の内容やその構成 (内容にリンクした絵と音楽) が大部分でデジタル文庫を読んでいる感覚で、ゲームをしている気があまりしない。そこで最近遊んだタイトルの中から、サウンドノベルズを除いてハマったゲームを挙げてみる。

1．デモンズソウル
2．世界中の迷宮Ⅲ
3．戦場のヴァルキュリア 2
4．ゼノブレイド
5．ファイナルファンタジー 13

やはり JRPG が大部分を占めているが、デモンズソウルのようなアクション RPG、戦場のヴァルキュリアのような戦略シュミレーションもランクインし、レベルという育成概念こそ共通にあるがジャンルはバラバラだ。では、なぜこれらのゲームがハマるほど面白かったのかと言うと、“ゲームオーバーになりやすい”からだ。

おそらく多くの人はゲームオーバーになることを嫌い、いっそういらない要素だと考えているのではないだろうか？実際最近の多くのゲームは難易度を下げてゲームオーバーになりにくくしたり、ゲームオーバーの概念がないもの作られている。例えばゲームオーバーゲームの代名詞的であったプリンス・オブ・ペルシャが 2008 年に発売されたものになると、操作ミスで死にそうになっても相棒がすぐに助けてくれてゲームオーバーにならないゲームに変貌している。一見するとゲームオーバーにならない、常に WIN な状態でいられるというのは楽しそうに思える。しかしこれらのゲームを遊んでみると、メリハリが少なく淡々とゲームが進んで作業ゲーム化し、30 分も遊べばもう満足してしまう。

プレイ時間が 30 分でも満足できればいいじゃないかという意見もあると思うが、私が欲しいのは“止め時が分からない w”や“これは他の人にも薦めたい！”という『ゲームを続けたいという欲求』や『この満足を他の人にも味わってほしいと自ずから行動を起こしてしまう』ような面白さなのだ。そういう意味では先に挙げた“ゲームオーバーになりやすい”ゲームはよくあてはまる。実際私はデモンズソウルのプレイ動画を USTREAM で配信し、友人のその面白さを伝える活動を精力的に行っていた。ならば、この“ゲームオーバーになりやすい”ことが、ゲームをより面白くし、持続的にプレイすることを助け、質の高い満足を得るための重要な要素なのではないだろうか？

ゲームオーバーで思い出すゲームと言えば、やはり“スペランカー”だろう。

今でも“迷”作として名高いが、これはあまりにも弱すぎる主人公が人気なのであって、肝心のゲーム部分について語られることはほとんどない。スペランカーの死ぬ部分はよく知っているが、カギを集めて扉を開けながらゴールを目指すゲームシステムや 2 周目以降はアイテムが透明化していき見えなくなるという高難易度ゲームになることを知ってる人は少ないのではないだろうか？つまり単純にゲームオーバーになりやすければ楽しいというわけではない。

もうひとつ別の例を挙げるてみよう。永遠の“名”作である“スーパーマリオブラザース”も実は非常にゲームオーバーになりやすいゲームである。しかしながら、こちらはキャラクターの認知度はもちろん、ゲームシステムや裏ワザにいたるまで多くの人が知っているだろう。この差はゲームオーバーのデザインの差と言っても過言ではないと私は考える。スーパーマリオブラザーズのゲームオーバー必ず「マリオはなぜやられたのか？」ということが明確に表現されていて、さらにそれが自分の操作ミスに起因するものだと分かるように作られている。ゲームオーバーになると一旦プレイの流れが中断され、一瞬冷静になっているときに、ゲームオーバーの原因は自分のせいだと意識づけられるため強烈に印象付けられるのだ。そのため、ユーザは自分の操作ミスを悔いて、次はもっと上手にやろう！と言う向上心が持続的なプレイにつながり、ミスしたところを越えたことで楽しさと大きな満足を得られ、結果多くの人の記憶に残っているのだ。

ゲームオーバーの設計を行う上で、もう一つ重要な要素はデスペナルティだろう。いくら自分のミスでゲームオーバーになったとはいえ、ペナルティが重ければそこでプレイを止めるきっかけになり、最悪の場合はゲームの評価もそこで決まってしまうだろう。とはいえ、デスペナルティはゲームジャンルによって、内容もその効果もバラバラで単純に比較、評価できるものなのだろうか？いくつか例を挙げてみよう。

・ドラゴンクエスト (RPG) … 所持金が半分になり、直前のセーブポイント (教会) まで戻される。
・スーパーマリオ (ACT) … パワーアップがリセットされ、ステージの最初に戻される。
・リッジレーサー (RCG) … 挑戦しているグランプリの最初まで戻される。
・ときめきメモリアル (SLG) … 最初からやり直し。
・テトリス (PZL) … 最初からやり直し。

一見すると比較、評価が難しそうに思えるが、いずれも“とある地点まで戻される”という共通の要素が含まれている。この戻される量がデスペナルティの重さと言えないだろうか？ゲームジャンルによって量の概念が異なるため、ここでは共通の評価項目として巻き戻される時間を評価軸として見ていくこととしよう。

こんな経験はないだろうか？通勤 (通学) 電車の中で PSP や DS で遊んでいると、電車を降りるタイミングとゲームのキリがいいタイミングが一致せず、ゲーム機のスリープ機能で一時中断するのだが､中断明けのキリがいいタイミングでセーブすることを忘れ、そのままプレイを続行し、不意なゲームオーバーで大昔のセーブポイントまで戻され、愕然としたことが。このことより、巻き戻される時間量がデスペナルティの重さの一つの評価項目として考えてよいと私は考える。

では、最初に挙げた 4 つのゲームではどのくらいの時間が巻き戻されているのか見てみよう。

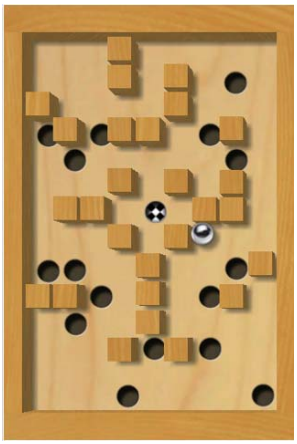
・デモンズソウル … 10 ～ 15 分
・世界樹の迷宮Ⅲ … 10 ～ 15 分
・戦場のヴァルキュリア 2 … 10 ～ 15 分
・セノブレイド … 5 ～ 10 分
・ファイナルファンタジー 13 … 5 ～ 10 分

この時間量は初見プレイでの時間であり、また難易度やステージ構成で巻き戻し時間は異なるため、おおよその時間である。デモンズソウル、世界樹の迷宮は広大なステージ内を探索するタイプのゲームでゲームオーバーになるとスタート地点に戻されるが、ある程度進むごとにショートカットが開通するため、意外と巻き戻される時間が少なく、どんなに奥まで探索しても巻き戻される時間量が劇的に増えることない。戦場のヴァルキュリアはミッションクリア型であり、1 ミッションのクリア時間が 15 ～ 20 分程度なので、たとえミッションの最初に戻されたとしても巻き戻される時間は少ない。ゼノブレイド、ファイナルファンタジー 13 は戦闘の (ほぼ) 開始直前に戻されるだけで、ラスボスのような特定の高 HP(ヒットポイント) ボスでない限り、巻き戻される時間は 5 分程度だ。こうやって見ていくと、普通に考えれば巻き戻し時間が多そうに思えるタイプのゲームであっても巻き戻される時間が 10 分前後と意外と短いことが分かる。

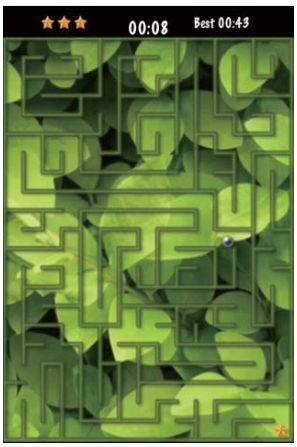
ここで 1 回まとめると、ハマるゲームというのは
・ゲームオーバーになりやすい。
・ゲームオーバーで巻き戻される時間は短い (10 分前後)。
この二つが満たされている必要があるのではないだろうか？
では、この条件が本当に正しいのか順番にみていこう。

まずはゲームオーバーの必要性について改めて考えてみよう。ゲームオーバーが必要か否かをみるためには、同じゲームルールでゲームオーバーの有無があるものを遊んでみるのが手っ取り早い。そこで今回は“迷路”ゲームを選択してみた。迷路ゲームはスタートからゴールまでのルートを探索するという共通のルールのもと、落とし穴に落ちるとスタート地点に戻されるというルールを加えるか否かで、ゲームオーバーの有無による面白さの違いが分かるはずだ。今回は手元に iPhone(と iPad) があるので豊富な iPhone アプリからゲームオーバーがあるゲーム、ないゲームをそれぞれダウンロードしてきて実際に遊んでみた。ちなみに遊んだのはゲームオーバーがあるゲームは『Labyrinth』『Labyrinth 2』で､ないゲームは『無限の迷路』の 3 種類だ。ゲームの傾向は以下の通り。

・Labyrinth シリーズ … ゴールまでの道のりが非常に明確な代わりに落とし穴や各種トラップがちりばめられている。ユーザは経路探索よりもトラップ回避することに専念する。



・無限の迷路 … 迷路の複雑さ、行き止まり具合が段階的にあがっていく。ユーザは純粋にゴールまでの道のりを探索することに専念する。



遊んでみた結果だが、個人的にはゲームオーバーがある Labyrinth シリーズのほうが面白いと感じた。無限の迷路もよくできた迷路ゲームなのはたしかなのだが、いくら複雑さが増したところでユーザに求められることは探索の一点のみであり、3 面も遊べばあとは作業化してしまい、面白さが薄れてしまうのに対し、Labyrinth シリーズは落とし穴の配置パターン、落とし穴に誘導するようなステージ構成やトラップの配置などでユーザの考えること、iPhone を操作する手さばきに様々なパターンを作ることができており、10 面以上を続けてプレイしても飽きがこなかったのだ。

ここに面白いデータがある。今回 iPhone アプリを用いたが、このアプリを購入する iTunesStore 内の AppStore では購入者の評価が 5 段階評価で提示されているのだ。その評価結果を以下に示す。

	☆5	☆4	☆3	☆2	☆1	平均
Labyrinth Lite Edition	80	49	51	24	20	☆3.6
Labyrinth	4	3	0	0	1	☆4.1
Labyrinth 2 Lite	56	28	32	11	5	☆3.9
Labyrinth 2	4	1	1	1	1	☆3.8
無限の迷路 Lite	656	294	763	800	1303	☆2.5
無限の迷路	4	2	5	4	2	☆3.1

購入者の評価でもゲームオーバーがある Labyrinth シリーズの方が面白いと言っているのだ。もちろん母数が大きく異なっていたり、AppStore の評価はゲームの面白さだけではなくアプリとしてのデキの部分も含まれるため、正確ではないかもしれないが、傾向としては十分に信用して良いだろう。

以上より、ゲームオーバーは必要であり、それがゲームを面白くする一つの要素であると私は考える。

次にゲームオーバーによる巻き戻し時間について考えてみよう。ここではゲームオーバーによる巻き戻し時間の量とユーザがゲームを続ける意欲の関係について見ていくのだが、意図的にゲームオーバーになるタイミングを操作するのはあまりにも不自然で、意欲にも大きく影響するだろう。そこで今回はトランプのババ抜きのようにゲームを進めていけばある程度同じ時間で必ず終わりがやってきて、さらに自然と周回プレイができて、その周回数で巻き戻し時間が調整できるものを探し、“麻雀”を選択してみた。麻雀は基本的に 1 回の局で勝負が決まらず、一種のラウンド制となっている。あまり詳しくはないのだが、現在の麻雀の対局数は以下のような種類がある。

・一荘戦：東場、南場、西場、北場それぞれ 4 局ずつの計 16 局
・半荘戦：東場、南場それぞれ 4 局ずつの計 8 局
・東風戦：東場の 4 局のみ
・一局清算：東一局のみ

<table>
<tr><th colspan="4">東場</th><th colspan="4">南場</th><th colspan="4">西場</th><th colspan="4">北場</th></tr>
<tr><td>東1局</td><td>東2局</td><td>東3局</td><td>東4局</td><td>南1局</td><td>南2局</td><td>南3局</td><td>南4局</td><td>西1局</td><td>西2局</td><td>西3局</td><td>西4局</td><td>北1局</td><td>北2局</td><td>北3局</td><td>北4局</td></tr>
<tr><td colspan="4">東風戦</td><td colspan="12"></td></tr>
<tr><td colspan="8">半荘戦</td><td colspan="8"></td></tr>
<tr><td colspan="16">一荘戦</td></tr>
</table>

それぞれのおよそのプレイ時間は上から 1 時間、30 分、15 分、5 分くらいとなっている。そこで今回はこの 4 つの対局方法で負けることをゲームオーバーに、巻き戻し時間＝プレイ時間とし、それぞれの対局後さらに対局を続けたいと思うかをみてみた。結果は次の通りだ。

1. 一局清算 (5 分) : 続ける意欲○　駆け引きほぼなしの運ゲームとなり、負けても自分が悪いという感覚が薄いため、続けてできる。
2. 東風戦 (15 分) : 続ける意欲○　オンライン麻雀や携帯ゲームの基本ルールだけあってお手軽で、負けてはいるのだが数回続けてプレイできる。
3. 半荘戦 (30 分) : 続ける意欲△　雀荘など対面打ちでの基本ルールではあるが、やはり 30 分頭を使い続けると疲れてしまい、その状態で負けたのだから、やる気はでにくい。
4. 一荘戦 (1 時間) : 続ける意欲×　国際標準ルールとはいえ、素人には長すぎて続けてもう一ゲームしようという気は起きない。

ここで面白いのは、東風戦を 2 回行うと半荘戦と同じ時間だけゲームを行ったこととなり、続ける意欲も当然同じだけ落ちると思われるが、意外と 3 回戦、4 回戦とモチベーションを維持したまま続けることができたのだ。人間が集中し続けられる時間が最長でも 2 時間とも言われていることを考慮すると、短い時間に息抜きができるタイミングを与えることで、集中する負荷が軽減され、結果的にゲームを持続的に遊ぶことができるのだろう。そしてその息抜きがゲームオーバーであり、そのサイクル時間は麻雀では 15 分が適当であるということだ。これは仮説で挙げたハマるゲームの条件である“巻き戻し時間が 10 分前後”とも当てはまる。

これまでハマるゲームの条件を個々に正しいのか考えてきたが、最後に条件をすべて満たしたゲームが本当にハマるゲームなのかを考えてみたいと思う。食べ物の食い合わせではないが、個々に良い要素であったとしても、組み合わさるとお互いが足を引っ張り合ってダメになるケースはよくあることだ。ゲームオーバーになりやすく、それによって巻き戻される時間が 10 分前後のゲームであり、今回はさらに読者がすぐに遊べるものという条件を追加して、私は“マインスイーパ”を選択してみた。マインスイーパはマス目に書かれた数字から隣接する 8 つのマスのどこに爆弾があるかを予測し、爆弾があるマス以外のマスをすべて開けるゲームだ。マインスイーパは難易度にもよるがゲームオーバーになりやすく、多少の運も入ってくるが、判断ミスやクリックミスなどのユーザ起因で非常に分かりやすいまたゲームオーバー時は初期状態に戻るため『巻き戻し時間＝プレイ時間』と分かりやすく、Windows パソコンを持っていたらすぐにでも遊べるゲームだ。

試しにプレイしたが、初級 (地雷数：10　マス目の数：9 × 9) ではプレイ時間 1 ～ 3 分程度で、よほどのポカをしなければゲームオーバーになりそうもなかった。そこで今回は中級 (地雷数：40　マス目の数：16 × 16) と上級 (地雷数：90　マス目の数：16 × 30) をそれぞれプレイし、ゲームオーバー時のゲームを続ける意欲をみてみた。ちなみに中級はうまくいけば 5 分ぐらいでクリアできたが、ゲームオーバーも多く、上級についてはクリアすることができなかったことをあらかじめ断わっておく…。

結果は次の通りだ。

	終了まで (回数)	平均巻戻し (時間)	60 秒以下 (回数)	600 秒以下 (回数)
中級	30 回	107.7 秒	14 回	16 回
上級	20 回	28.8 秒	18 回	2 回

なんとなく想像はついたが、マインスイーパは最初の 4，5 点までのファーストタッチで爆弾を引き当てるか、ある程度マス目が開けられたが、最後で 2 択を迫られ爆弾を引き当てるかの 2 極化し、平均値は中級が約 2 分となっているが、あまり意味のある数字ではなさそうだ。ちなみにやる気をなくすまでの回数がキリがいいのは、若干邪な気持ちが入ったためではあるが、もういいかな、と思ったのは間違いない。今回の施行で思ったのは、10 分以内であればそれは数秒であろうと数分であろうとゲームを続ける意欲には大きな影響を与えることはなく、むしろ中級プレイではクリア目前だったのに判断ミスでゲームオーバーになった、上級プレイ時はあまりにも短時間ゲームオーバーを量産したことの方が大きく影響を受けた。

若干別の要素の影が見え隠れしたが、同じルールのゲームを連続して 30 回、時間にして約 1 時間ほど連続してプレイできるゲームというのは十分ハマるゲームと言っていいのではないだろうか？よって、私のゲームオーバーがあるからこそゲームは面白くし、持続して遊べることの一因として見てもいいのではないだろうか？

大事なことなので 2 回言います。ゲームオーバーがあることでゲームは面白さを増し、さらには持続的にプレイを続けるための原動力になると考えられる。もちろん、ただゲームオーバーという条件を作れば良いわけではなく、なぜゲームオーバーになったのかが明確であり、さらにそれがユーザ起因であること、またゲームオーバーによって巻き戻されるプレイ時間は 10 分前後以内となるようにゲームデザインができていなければ、うまく機能しないだろう。

私も簡単ながら新しいゲームの企画、デザインを考えているが、いつもどうやったらうまくできるかばかりを考えていて、ゲームオーバーのデザインまでしっかりと考えていなかった。おそらく今回の記事を書くことなくゲームを作っていれば、それはきっと緊張感が足りず、淡々と進める薄っぺらいゲームになっていただろう。これから冬に向けてゲームの企画を考えていきたいが、もしこの記事を読んで賛同してくれる人がいれば、サウンドノベルズゲームに負けない、止め時が分からないｗと言わせて、みんなに勧めさせるようなゲームを一緒に作りましょう！

執筆者リスト

ranha (twitter-id: ranha)

日本語の実装に多数の不備が確認されている。

住所が Tsukuba というガイコクである点を考慮すると流暢に話すとも言える。

shelarcy (twitter-id: shelarcy)

"Haskell の人 " であることは誰もが知るところ。

ITpro で「本物のプログラマは Haskell を使う」連載中…、ってそれこそ誰もが知るところ。

http://itpro.nikkeibp.co.jp/article/COLUMN/20060915/248215/

cubicdaiya (hatena-id: cubicdaiya)

世間には知られていないが、類い稀なる変な人。

" いわゆる変な人 " 達が、彼と一度でも食事をするなどすると、変な奴だなーと認識する。

この前まではウノウの人。

nish

謎の人。

若い頃は x86 アーキテクチャ大好き小僧だった風があるのだけど、最近はそうでも。

(いやこれは編集者の勝手な想像です…)

lyrical logical (twitter-id: lyrical_logical)

天才魔法少女見習い。

高尾山で死んだ人の甦りである、とか旧約聖書 (ISO/IEC 14882:2003) には記されている。

zick (twitter-id: zick_minoh)

" リリカル Lisp の人 " と言えばその筋ではすぐさま理解が得られる稀有な人物。

" かんさいの人 " でもある。

mascalade

先頃吸収合併された…ちょ、待てな q あ w せ drftgy ふじこ lp

編集後記

COMFRK 編集長 m0h1can です。執筆者の方々には無理を押してご執筆頂き、本当にありがとうございました。おかげさまで、一癖も二癖もある記事が集りました。そしてまさにこれが、本誌を創刊しようと思い至った動機付けでもあります。言ってしまえば、変な記事が脈絡なく集った雑誌が欲しかったわけです。もっともそれは、昨今であれば web を見て解決する欲求かもしれません。ただ私が考えたのは、同期無き他者の行動の集積として自然とそうなってしまった web の多様なスクラップ、ではなくて、編集の意図が介在して変に統合されてしまった雑誌、という提案です。読者が週刊誌を拾い読む感覚で本誌を開き、日常会話のネタを知る感覚で特異な技術を覚える…そんな計画された知識の攪拌を実験していきたいなと思います。次回 COMFRK はおそらく冬頃になると思います。今後ともご贔屓に！

執筆者募集

拘りをもって何か技術記事を書きたい方、ぜひご連絡下さい！

twitter: m0h1can

email: m0h1can@comfrk.info

奥付

COMFRK VOL. 1

2010 年 8 月 13 日　　初版一刷

編集長　m0h1can
発行者　m0h1can
発行所　COMFRK
　　http://comfrk.info/
　　a@comfrk.info
装丁　　中村さん
印刷　　最寄りのキンコーズ