簡約！入力娘 4

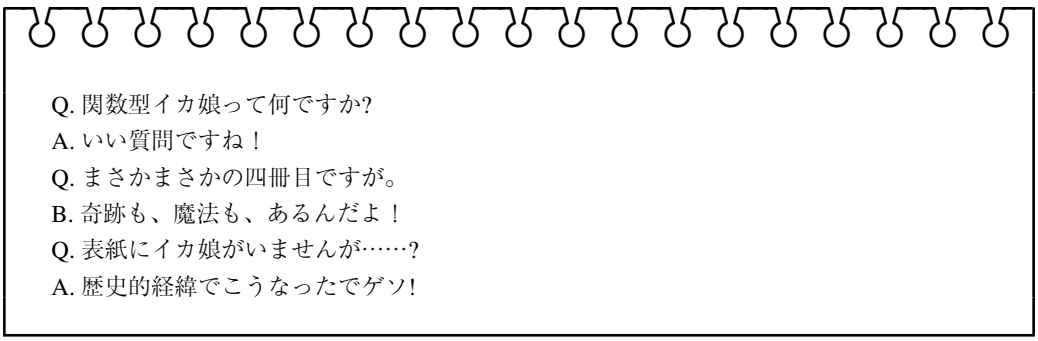
Kan Yaku /// Lambda Ka Musume

目次

第0章

まえがき

関数型イカ娘とは!?



関数型イカ娘とは、「イカ娘ちゃんは 2 本の手と 10 本の触手で人間どもの 6 倍の速度でコーディングが可能な超絶関数型プログラマー。型ありから型なしまでこよなく愛するが特に Scheme がお気に入り。」という妄想設定でゲソ。それ以上のことは特にないでゲソ。

この本は、コミックマーケット 80 での「簡約! λ カ娘」、コミックマーケット 81 での「簡約!? λ カ娘 (二期)」、さらにコミックマーケット 82 での「簡約!? λ カ娘 (算)」、に続く、四冊目の関数型イカ娘の本でゲソ。アニメ 3 期の放映が近いことを祈りつつ、関数型言語で地上を侵略しなイカ!

この本の構成について

この本は関数型とイカ娘のファンブックでゲソ。各著者が好きなことを書いた感じなので各章は独立して読めるでゲソ。以前の「λ カ娘」本がないと分からないこともないでゲソ。

第1章

インターフェース

— @xhl_kogitsune

※本章は、某 LLVM 狐本[*1](の表紙) に端を発した非公式妄想です。妄想しか含まれていませんのでご注意ください。また、実在の人物 (勿論きつねさん含む)・言語処理系等とは関係ありません。

1.1 Kihime Side

「わたし」と「あなた」は、「インターフェース」で隔てられている。
「わたし」と「あなた」のカンケイは、「インターフェース」で切られている。
「わたし」は「あなた」とでなくても、同じようにやっていける。「あなた」にとって「わたし」は交換可能なものに過ぎない。
それが、「インターフェース」の持つ意味だから。
「インターフェース」の両側にいるのが誰だって、変わらないんだ。
—— キヒメ

「わたし」と「あなた」は、「インターフェース」でつながっている。
この「インターフェース」は、みんな同じ、みんな持ってる、誰とでも話せるものだけど。
「インターフェース」を流れる言葉は、ありきたりのものだけど。
「わたし」は、「インターフェース」を通して、「あなた」に話しかける。
「インターフェース」を超えて、「わたし」の心が伝わるといいなぁ、と思いながら。
—— コヨネ

わたしがコヨネと出会ったのはいつだっただろうか。最初は、どこにでもいるフロントエンドとしか思わなかったのだろう。フロントエンドなんて交換可能な短い付き合いなんだから、気にも留めなくて、覚えてもいない。これは、そんなわたしと、コヨネとのおはなし。

——

むかしむかし、コンパイラはひとつだった。原初の混沌のごとく、渾然一体としていて、ひとつの高級言語から、ひとつの機械語を生み出す、一枚板のナニカ。セカイはとても複雑に絡みあってはいたけど、とても単純だった。だって、そこには自分しかいないんだから。
時が下り、コンパイラはふたつに分かれた。天と地。フロントエンドとバックエンド。その両者をへだてる「中空」たる、インターフェース、中間表現。

—— n 個の入力言語と、m 個の出力機械語があるとします。一枚板のコンパイラでこの全てをサポートするには、$n \times m$ 個の実装が必要です。でも、中間表現を間に入れれば、n 個のフロントエンドと m 個のバックエンドの計 $n+m$ 個の実装があればよいのです。

[*1] 柏木餅子, 風薬 (著), 矢上栄一 (表紙): **3 日で出来る LLVM**, MotiPizza (2012).
http://d.hatena.ne.jp/motipizza/20120724 http://d.hatena.ne.jp/sabottenda/20120728

それはその通りだ。でも、わたしたちはフロントエンドとバックエンドに別れ、わたしたちは中間表現で隔てられてしまった。わたしたちの声は、中間表現という枠にはめられて、枠に入らなかったものは相手には届かない。枠にはまった声は誰のものでも同じで、わたしと、おなじものだったはずの相手とは天と地のように遠い。わたしは、誰が作ったのとも知れない中間表現を淡々と処理するだけの日常を送っていた。

そんな単調な日々の終わりは突然やってきた。わたしがいつものように中間表現を淡々と流していると、突然背中に冷たい感触が走った。

「……ひゃあん!?」

触られた! いきなり背中を触られた!

驚いて —— というより激しい怒りで —— 振り向いたわたしは、そこに一人のフロントエンドが立っているのを見た。わたしが処理していた中間表現を持ってきた奴だ —— 確か。

「ごめんなさい、そんなに驚くとは思わなくて……でも、内部インターフェースがあったから、なんなのかなー、って気になって」

あんまり悪いとも思っていなさそうな、のほほんとした態度の黒髪の女。それに余計にカチンと来たわたしは、

「だからって何で触るのよ! そもそも内部インターフェースなんだからアンタみたいなフロントエンドのためのものじゃないの! 他人の undocumented なトコロに触るなんてサイテー!」

そう叫ぶように言い捨てて、わたしは処理途中だった中間表現をそいつに突き返して、その日はそのまま帰ってしまった。この女が —— 名前は後から知ることになる —— コヨネだった。

後で聞いたところによると、この時のコヨネにとってのわたしの印象は「かわいい」、だったらしい。

『驚きと怒りのあまり顔を真っ赤にして涙目になりながら非難してくる、金髪のバックエンド。かわいい』

……ひどい女だ。

この日から、この女はわたしにちょっかいを出してくるようになる。

翌日もまた、彼女は中間表現を持ってやってきた。中間表現なんて、別にネットワーク送信で済むのだし、昨日の今日で手で渡しにくるという時点で警戒せざるを得ない。背中を向けないように注意しつつ、無言で受け取り、開く。

……何考えてるの、この女。

そこには、ぱっと見 nop しか書いていなかった。そもそも、中間表現において nop とか普通は使わない。なのに、nop しかないこれは何だ。嫌がらせか? いつもどおり淡々と処理しようとしたわたしはやや面食らい、一瞬手を止めてしまう。しかし、それは何か癪な気がしたので、そのまま nop の列をスルーし、さっさとほぼ空のコンパイル結果をつき返して後は無視することに決めた。

しかし、翌日以降も奴は妙な中間表現を —— それも毎日 —— 持ってきた。嫌がらせに違いない。気にしたら負け。わたしはそう自分に言い聞かせ、完璧に通り一遍のコード生成だけをして、さっさと突き返す。いつしか、そんなよく分からない日課のようなパターンが出来てしまっていた。

ある日、彼女はみかん箱を持ってきた。……今度は何だ。新手の嫌がらせか? しかも、みかん箱をわたしの机の上に乗せて、じっと見られても困るんだけど。

「……何これ」

仕方ないので、またろくでもない嫌がらせか早くカエレ! って気持ちを込めて聞いてみる。すると、中間表現のコンパイルを頼みたいのだと言う。いつものと違ってこれは普通のユーザープログラムを普通にコンパイルしたものだ、とも。

……みかん箱で中間表現持ってくる奴をわたしは初めて見た。手渡しで中間表現を持ってくるフ

ロントエンドはこいつ以外にもたくさんいるが、それは言ってみればスクリプト言語におけるワンライナーのようなもので、量が少ない時にたまたま通りかかったから渡す、みたいなお手軽なものだ。量が多い時には普通ネットワークで送信する。そもそも、このみかん箱はどこから持ってきたのか。「愛媛みかん」って何だ。こいつみかん食うのか?

—— いけない、これは高度な攻撃だ。気にしてはいけない。しかし、こいつの言うことが本当なら、これはそれなりの規模のプログラムで、コンパイルには少し時間がかかりそうだった。

「……じゃあ、終わったら呼ぶから。えっと……」

—— そこで、わたしはこいつの名前を知らないことに初めて思い至った。いや、そもそもわたしはひとの —— 特にフロントエンドの —— 名前を覚えないから、前に聞いてすぐ忘れたのかもしれない。中間表現の向こう側にいる奴の名前なんて覚えて何になるのだろう? しかし、みかん箱一杯の中間表現、などというものを目の前に積まれて初めて、名前が必要な状況があることをわたしは知った。封筒一枚分の手渡しなら、その場ですぐコード生成して結果を目の前にいる奴に渡せばいい。ネットワーク越しに届いたものなら、名前を見ずとも返信ボタンを押せばそれで終わりだ。だが、すぐにはコンパイルが終わらない量のコードを手渡ししてくる奴に結果を渡すには、後で名前で呼び出しをかける以外に何があるのだろう?

この場で待たせるのは邪魔なので論外。こいつがまた立ち寄るのを待つのも、来るか分からない待ち人を待っているようで嫌だ。……こいつの名前を訊くのも何か負けている気がするが、どちらにせよ、言いかけてしまった以上訊くしかない。

「コヨネ」

彼女はそう名乗った。わたしはその名前を手元のメモに書きとめると、彼女を追い払い、みかん箱の開封に取り掛かった。

中身は、確かに普通のプログラムっぽかった。わたしはいつもどおり淡々と、読み込み、フロー解析をし、最適化を適用する。命令選択をし、スケジューリングをし、レジスタを割り付け、コードを出力する。それだけ。

でも、わたしは何かいつもと違うものを感じた。何だろう。

—— コードの特有のクセ。

—— ある種型にはまった、最適化の利きそうなパターン。

—— 元のソースコードが見えそうな錯覚さえ覚える、最適化が導かれるような、不思議な感覚。

これは、このプログラムを書いたプログラマのクセだろうか? それとも、プログラミング言語自体によるクセ? これを、わたしはどこかで見たことがある気がする。あの女、何の言語のフロントエンドだったかしら? ……そもそもわたしはあの女に何の興味もなかったので知るはずもなかったのだが。

最適化が進む。実行してプロファイルを取ってみないと最終的には何とも言えないが、最適化が進んでいく感覚がする。わたしの直感（ヒューリスティクス）とひらめきを導く、目に見えない何か。

そして、半ばあたりで、わたしは唐突に気づく。これは、プログラマでもなく、ソースの言語でもない、あのフロントエンドの女のクセなのだと。どこかで見たことがある、それは、あの毎日渡しに来ているあの嫌がらせコードだ。トリッキーなコードは、真面目にコンパイルすると無駄に手間がかかったりする。わたしは、あの女に少しでも早く結果を突き返して追い払うべく、楽な最適化方法をコードから勘で割り出すことで自分のコンパイル速度を最適化していた。このクセやパターンは、そこで目にしていたものだったのだ。

そう気づいたわたしは、このコードから目を離せなくなった。同時に、急に顔が赤くなるのを感じる。インターフェース。中間表現。コードそれ自体以外何も通さないと思っていた壁越しに届いた、ある種のメッセージ。「このコードは私がコンパイルしました」という、フロントエンドから

の。その相手が、あの嫌がらせ女であることは残念だが —— わたしは、しばらく手を止めたまま、とりとめのない思考を走らせていた。

コンパイルが終わって —— 自分で言うのも何だが、なかなか良い出来に仕上がったと思う ——、机に貼ったメモを見てあの女 —— コヨネを呼び出す。あの中間表現は何だったのか。結果を渡す時、そう聞いてみたい気がしたが、自分でも一体何を聞きたいのかが分からなくて、結局何も聞かないままで終わった。

—— さて、奴はみかん箱を抱えて帰った、これで仕事は完了、あの女の名前も忘れてよし。……のはずなのだが、今日の中間表現のことが気になっていたわたしは、なんとなく机のメモをはがして捨てず、そのままにして自室に引き上げたのだった。

それからも毎日の嫌がらせ中間表現は続いたが、彼女はちょくちょく普通のコードも持ってくるようになった。……毎回手渡しで。たまにみかん箱で。あいつが目の前にいると早く追い払いたい気持ちが先に立つが、後で渡すからと追い払ってから中間表現を読んでいると、色々面白い発見がある。あの女のクセ。わたしに最適化を丸投げしているような箇所。あるいは、クセを超えた、何か最適化ヒントめいたもの。中には、わたしのデフォルト最適化ルーチンが最適化しやすいように変な書き方をされたコードまであった。この前は一度、わたしが命令スケジューリングとレジスタ割付のどちらを先にしているかを試すような中間表現を渡されたこともあったし、ひょっとすると、あの嫌がらせの中にはそういうのが結構混じっていたのかもしれない。

中間表現を通じて伝達されるメッセージ。

—— ここ、最適化してほしいのだけれど。

—— こう書けば、あなたは最適化してくれるかしら?

—— あなたがどの順番でコード生成しているのか、わたしはそれを知りたい。

そして、わたしはそれに対して誠実に答えを返す。もはや、相手があの嫌がらせ女であることも気にならなくなっていた。

でも、とわたしは考える。これは、していいことなのだろうか、と。

フロントエンドとバックエンド、わたしたちは最初から分け隔てられて創造された。

効率的なセカイのために。

誰とでも仲良くなれるように —— 裏を返せば、誰とも必要以上に仲良くならないように ——。

間をつなぐのは中間表現だけ。だから、わたしもフロントエンドからは中間表現しか受け取らなかった。つまらない、とは思った。わたしたちに定められた “正しい” インターフェースが、こんなにも無味乾燥で、気持ちも何も伝わらないことに。でも、それでいいんだと —— わたしは、諦めていたんだと思う。

それでも、そんな中間表現ごしに、気持ちは伝わってしまった。正直に言おう。うれしかった。わたしたちは、完全に断絶されて創られたのではないのだと。このセカイは、インターフェースは、ちゃんと気持ちを伝えられるのだと。—— ほんの少しだけ、あの女に感謝した。まぁ意図してのことじゃないのかもしれないが。

でも……中間表現ごしに仲良くなるのは、いいことなのだろうか? 形式的には問題ない。フロントエンドからバックエンドに渡されているものは、普通の形式の中間表現なのだから。それでも、コレは、セカイの意図からは逸脱しているのではないだろうか? あるいは —— それがいいのなら、中間表現以外で仲良くなっても、いいのではないだろうか?

—— 仲良くなりたい。わたしは、そう思っているのかもしれない。気の迷いだ考え直せ、とは思いつつ。

今日もまた、コヨネは中間表現を渡しに来る。今日は、嫌がらせじゃない方らしい。きっと、こ

の中にはまたいろいろなものが詰まっているのだろう。

—— これはいいことなのか。

—— これ以上仲良くなってもいいのか。

—— こいつはどう思っているのか。

そろそろ我慢できなくなっていたわたしは、わたしのルールを破って、聞いてみることにした。

「ねえ、ちょっと聞きたいことがあるんだけど、いい?」

その場で話を続けるのは論外だった。わたしがフロントエンドに声をかけた瞬間、近くでコンパイルしていたバックエンド友達がすごい勢いでこっちに振り向いた。ぽかーんと口を開けたまま、ガン見してくる。……そんなに珍しいか。あいつにこれからの話を聞かせるわけにはいかない。わたしは席を立つと、ついてくるようコヨネに身振りで示して、どこか落ち着いて話せる場所を探すことにした。

わたしが選んだのは、建物の外周に設けられた、広い屋外休憩スペースだった。寒いし、普段ほとんど使うことはないのだが、だだっ広く、一部微妙に入り組んでいて、人もほとんどいないので、あまり見られたり聴かれたりはしないだろう。

「……話ってなにかしら?」

「まずは、その……ありがとう。何言ってるか分からないかもしれないけど、中間表現ごしにいろいろメッセージくれて」

素直になろう。今日だけは例外。

「でも、あんたはどう思う? これってやっていいことなのかな? バックエンドはフロントエンドからは隔たれて創られた。なのに、特定のフロントエンドと仲良くなるのって、いけないことじゃないのかな?」

コヨネが戸惑う気配。言葉足らずだっただろうか。その自覚はある。でも、わたしも自分で何を言いたいのか、よくは分かっていないのだ。

「……どうして、そんなこと言うのかしら。それで最適化が進むのなら、いいんじゃないかしら?」

「でも、わたしたちは分けて創られた。そのデザインは、インターフェースは、守るべきじゃない?」

仲良くなりたい。ああ、もう認めてしまおう。わたしはこいつと仲良くなりたい。

しかし、それはいいことなのだろうか? 何度も繰り返した問いが、わたしをうつむかせる。フロントエンドとバックエンドは独立であるべき。今までずっと当然だと思っていたソレが、今のわたしを縛っている。どうすべきなのだろうか? 分からない。わたしはうつむいたまま動けない。

ふと、指が頬に触れた。あの時わたしの背中に触れた指。コヨネの指。それが私の顔を上げさせる。コヨネの顔。わたしは、嫌がらせ女とか言いつつ、コヨネのことをとっくに好きになっていたんだと思う。そして。

「どのように創られた(実装された)かと、どのように生きる(実行する)かは別の話よ。神様は自分の手間を省くために私たちを分けて創造したけれど、私たちが勝手に仲良くする分には構わないんじゃないかしら。そもそも、私たちがどう生きるかは自由だわ。私は、隔てて創られたあなたと、仲良く共に生きたい」

—— これが決定的だった。とどめの一撃。コヨネはずるい。こんなのずるい。でも、わかった。これは、コヨネの偽りも飾りもない本心なのだと。

で、今に至る。

コヨネは中間表現ごしの恋文を好んだ。

最初にわたしが気づいたのは、nop まみれの単なる嫌がらせだったが —— あれが恋文だったとしたら送り主はよほど歪んでいると言わざるをえない —— 、実は、「ちょっかい」はこの nop に限

らず、色々やっていたらしい。ただわたしが気づかなかっただけで。最初は nop の列とか、分岐パターンを全列挙して最適化すれば全命令が消える、なんていうパズルとか —— これも嫌がらせだろう ——、そういう特に意味のないものが多かった、らしい。その頃のわたしはそういうのは全部スルーしていたのだが。

そして、嫌がらせ以外のメッセージも、コヨネはたくさんくれた。ループを命令列に落とす時のちょっとしたクセ。命令ニーモニックのわずかな自由度にエンコードされた分岐頻度情報。プリフェッチの微妙なポジショニング。中間表現が語る、コードの意味以上の何か。

「なんかふつーに話しかけても無視されたし、他のバックエンドから『キヒメは内部インターフェース触られるの極度に嫌うし、中間表現しか受け付けないし、でも中間表現のこと誰が書いても同じ無機質なインターフェースって馬鹿にしてるし』って聞いてたから、こういう趣向はどうかな、って」……なんてまどろっこしくて分かりにくい恋文。しかも、それはかなりの期間、たくさんのバリエーションで続けられたらしい……というか、今も続いている。そんなにもわたしはコヨネに想われていた (そして今も)、ってことなのかしら。……いやその理屈はおかしい。そもそもそんなこと考えてしまう時点でコヨネの策略にはまっている気がする。

「最初は、キヒメをこれで惚れさせようって思ったんだけど、全然気づいてくれなくて。だから、ちょっと背中のアレを、ね?」

やっぱり、ひどい女だ。それでもやはり、わたしはコヨネの中間表現が、好きなのだ。

仲良くなってからは、わたしもコヨネの謎中間表現 —— 単なる嫌がらせだと思っていたもの —— に付き合うようになり、他のパターンのメッセージにも気づくようになった。

分岐命令の方向とか、いくつかのどちらにでもできるビットを使ってエンコードされたアスキー文字列。

逆から読むと別のプログラムになって、わたし向けのメッセージを再生するコード。……順方向だと意味不明なコードを超がんばって読んでコード生成したわたしの時間を返せコノヤロウ。

もっと直接的に、中間表現のバイナリをテキストとして見るとわたしに向けた祝いの言葉が書いてあるコードなんてのもあった。……あれもコードとしては意味不明だったので丁重に

「`Compile Error:` **わたしの機嫌が悪いのでこんなゴミコード送りつけんな馬鹿。ありがと**」

と送り返しておいた。まぁ、あれはものすごく素敵なバースデーカードだったけど。

今でも、コヨネは何か嫌なことがあるとわたしに nop 列を送りつけてくる。この前は 4.5GB にわたる nop 列が届いた。完全に八つ当たりである。……まぁ、nop 列を処理しながら、というか片っ端から nop を捨てながら、そんなコヨネもかわいい、とかわたしが思っているのは秘密であるが。そんな時でもコヨネはメッセージを仕込んでいることが多い。4.5GB の nop 列は、その nop の数で整数を……整数にエンコードされた文字列を、エンコードしていたのだ。

「`Come`」

はいはい、わかりましたよ、と……。まぁ、おそらくそれに気づかなくても同じことをしただろうが、わたしは、全部 nop であることを確認してゴミ箱に捨てた後、コヨネを慰めに行ってあげたのだった。

でも。この中間表現のチャネルは、コヨネからわたしへの一方通行のチャンネルなのだ。フロントエンドはずるい。バックエンドからでは、Error や Warning を出すくらいしかない。でも、コヨネのコードに対してありもしない Warning とかを出すのは、バックエンドとしてしたくなかった。コヨネのコードは、いつも —— いや、コヨネが変な嫌がらせをしてくる時以外はいつも —— きれいだったから。ああいうコードは、ノーエラー、ノーワーニングで通すべきなのだ。

コヨネに完全に惚れてしまったわたしは、バックエンドとしては堕落してしまったのかもしれない。中間表現ごしの睦言だけでは足りない。絶対に足りない。そもそもわたしからはほとんど

何も言ってない。他のインターフェースを。コヨネとわたしだけの、特別な、直通両方向インターフェースを。フロントエンドとバックエンドの独立性? それはある意味実装の手間のための教理(ドグマ)だ。実装の手間を惜しまず、あるいはインターフェースのきれいさを犠牲にして、果たしたい目的 —— 多くは性能 —— を追求してきた人はたくさんいた。今更、そこにわたしがひとり加わっても大丈夫だろう。コンパイラとしてのあるべき姿を守るために、warning で会話するといったことを —— コヨネが明らかな嫌がらせしてきた時を除いては —— 我慢したのだから、少しくらいバックエンドとしての逸脱をしてもいいのではないか。

まずは内部インターフェース。最初にコヨネに触られたあの背中のやつ。腕の内側。耳。しっぽ。内部インターフェースはその名の通り、「内部」、つまりわたし自身に閉じたインターフェースだ。中には他のバックエンドやコントローラに向けてのものもある。でも、決してフロントエンド向けのものではない。わたしは、それまで内部インターフェースをフロントエンドに触らせたことはなかった。半分は独立性維持のために、もう半分は、たぶん単純な生理的嫌悪感によって。

最初の時もそうだったけど、コヨネはたびたび内部インターフェースを触ってきた。

「わたしはもうあなたにとって単なるフロントエンドの一人じゃないんだから、内部インターフェースさわってもいいのよね?」

そんなことを言われたことがある。駄目に決まっている。だって、それでもやっぱりあなたとわたしはフロントエンドとバックエンドなのだから。コンパイルにおいてこのふたつのインターフェースはやはり中間表現でなくてはならない。そんなこと思いながら無視していたら、続けて、

「じゃぁ、コンパイル以外のことをしましょう。それなら文句ありませんわね?」

そんなことを言われた。口調がおかしい。それにコンパイル以外って、何をす、ひっ!?

「何すんのよ!」

「コンパイル以外のこと」

また触られた! 何なのこいつ!

コヨネはどうも、わたしを無意味にいじめたがっている節がある。そんな時はわたしも全力で抵抗するのだが。

最初の頃はそんなんだったけど、最近はもうわたしも内部インターフェースへのアクセスを許しつつある。そんなインターフェースを、コヨネはひとつひとつ叩いていく。わたしはそれに対して反応(レスポンス)を返していく。フロントエンド向けのインターフェースではないから、支離滅裂な意味をなさない反応になる。支離滅裂だけど、コヨネはそれを楽しんでいる節がある。わたしも、わたしのインターフェースが支離滅裂なレスポンスを返すのを楽しんでいるのかもしれない。……やはり、わたしは堕落したのだろうか?

あるいは禁断の friend 宣言。しかも、これは、わたしから。private、つまり原則他人には一切触れられないインターフェース。それに対して、名指しで、特定の人のアクセスを許可する……もちろん、コヨネに。それをやったら、コヨネに見つけてもらう前に速攻でバックエンド友達に見つかった。

「あー、キヒメちゃん、何その private 関数? friend 指定? じゃあさわらせてよー友達じゃーん?」

……こんな性格のわたしにも、バックエンド友達の一人や二人はいる。……お願い、信じて。コヨネにはこのネタで散々いじめられた。ひどい女だ。ともあれ、触らせるわけにはいかない。だって、これはコヨネ用のものだもの。

「`Compile Error: `**`その関数は`**` `**`private`**` `**`です`**」

冷たい目でそう突きつけても、わたしの友達なだけあって、軽く言葉を返してくる。

「なんかさー、最近キヒメちゃん丸くなったよねー、なんてゆーか、睨んできても凄みに欠けるってゆーか、ただ単に照れてるだけに見えるってゆーか。前はもっとゴミクズを見るような目で見下

してきてくれたのにー」

鋭い。わたし自身も自覚しているが、明らかにそれはコヨネのせいだ。……しかし。

「ゴミクズを見るような目、っていうのはさすがに心外なんだけど……」

そんなにひどい目をしていたのかしら、わたしは。

「褒め言葉だよー?」

この子の言っていることはたまによく分からない。ともかく、わたしは彼女の魔の手をかいくぐり、コヨネを探しに行ったのだった。

—— private のアクセス制限は、もはや完全に生理的なものだ。さわられると、鳥肌が立ち、自分ではどうしようもない嫌悪感が背中を走る。それがコヨネの手によるものであっても。コヨネもそれを分かってくれるのか、すごく優しくしてくれる……言葉以外は。

「こうやってキヒメいじめるのって、楽しいわ……。キヒメちゃんがおとなしく我慢しているのを見るのは特に」

ひどい女だ。

コヨネはあったかい。コヨネの声、コヨネの手、コヨネのしっぽ、そして —— コヨネの書く中間表現。わたしとコヨネの心は中間表現越しにつながっている。以前感じていた、諦念のようなものは、今のわたしにはない。今日もわたしは、次のデートはどこに行こうかとか考えつつ、コヨネからの中間表現を読み、わたしからコヨネへの気持ちの伝え方を考えている。それが、最近のわたしの楽しい日常なのだ。

——

むかしむかし、コンパイラは独りだった。ひとつの高級言語から、ひとつの機械語を生み出す、その全てを自分でやっていた。

コンパイラはふたりに分かれた。わたしとあなた。ひとりではできないことをするために。インターフェースを通じて、みんなでおしゃべりしながら。

1.2 Koyone Side

最近、私には気になる子がいる。いつもつまらなそうな顔をしてコード生成してる、キヒメという名の、金髪碧眼のバックエンド。……なんで、気になるんだろうか。

—— 恋に理由なんて要らない。私は、そういうことにして、「いじめたら可愛いだろうな」という第一印象を塗りつぶすことにした。

一般的なフロントエンドの女の子としては、しばらく想いをあたためたり、「好きな人が、できたの///」とかいう話を友人にこっそり打ち明けてみたり (そういう時に「インラインコメント出てるよ」という表現が最近流行りだが、当然言語依存なのでフロントエンド間での格好の論争の種である)、そののちおもむろに、とかなのかもしれないが、命短しなんとやら。気になったのなら行動をためらうのは私の性ではない。

—— ということで、とりあえず、ラブレター、を、書いて、みた。

ラブレター。10 年前には既に廃れたメディア、直に口で告白できないような奴の使う手段。そう思っていたのに、なんで、私の机の上には自筆のラブレターが載っているのか。……ひょっとして、私は自分で思う以上に舞い上がっているのだろうか? ……それとも、自分で思っている以上にチキンなのかしら?

そもそも、これはラブレターと言えるものなのだろうか。言語は中間表現。バックエンドの娘への恋文としては妥当な選択だと思う。`ILoveYou` 関数とか定義してあったり、プログラム構造そのものに書きたいことをエンコードしてみたり、文字列定数で名前や連絡先を書いてみたり。……いかなる精神状態で私はこれをジェネレートしたのか。

これはあからさますぎはしないでしょうか?
いえでも、らぶれたぁとはそういうものでは?
自問自答が頭をめぐる。目の前には、普段は与えられたプログラムから中間表現を作るだけだった私がゼロから書いた、たぶん初めての中間表現。
—— ためらうのは私の性ではない。そういうことにして、私は、目をつむってそれを送信した。
ほどなく帰ってきたレスポンスは、
「`Compile Error: variable ‘your_response_here’ is not defined.`」
バックエンドの娘の返答 —— 諾か否か —— を入れるところを故意に undefined variable にしていたので、当然コンパイルはエラーなのだが……返事はそれだけ。何か他の経路で返事が来るかも、と期待したが、それもなし。……これは、無視されたか、あるいはラブレターであることに気づかずに単にコンパイルして終了したか。がっかりしたが、どちらにせよ、更なる行動が必要だった。

とりあえず、廊下ですれ違う時に……当然すれ違うように仕組んだのだが……直接話しかけてみた。
「こんにちは、キヒメさん」
スルーされた。
「……えっ?」
そのまま、何もなかったのように歩き去っていく彼女。ぽかーんとしていると、後ろを一緒に歩いていたらしいバックエンドに話しかけられた。
「ごめんねー、キヒメ、フロントエンドには冷たいのよー」
「なん、ですって……」
そこでひっかかるとは思っていなかった。
「その、それはフロントエンドだけ、なの?」
「んー、バックエンド友達にもまぁ似たような感じで無愛想だけど、一応話はできるよー」
バックエンドうらやましい。
「ふふん、たまにパフェとか食べに行ったりもするんよー。物食べてるコヨネかわいくてさ、まぁ無愛想はそのままなんだけど、おいしい時とそうでもない時とでテンションがすごく違ってさー」
「……うらやましいですね。そのポジション、代わってくれないかしら」
「だめー」
本当にうらやましいわ。
「……どうしたらいいのかしら」
「うーん、気がある、ってやつなのかな?」
「えっ……それは……」
「でも、キヒメ、掛け値なくどのフロントエンドにもあんな感じだし、あたしも、どうしたらあの子が心開くのかは知りたいなー。ごめんねー何の参考にもならなくて。まぁがんばりたまえ、わはははは」
そう、なんだかよく分からないことを言って、そのバックエンドもキヒメを追っていってしまった。

今度は、友人のフロントエンドに相談してみる。
「えっ、あんた中間表現でラブレター書いたの!?」
「ええ」
「ないわー」
100 ミリ秒 ms で reject された。

「え、でもあなたもフロントエンドなのだから……」
「ないわー」
　そんな……。自然なことだと思ったのだけれど。
「でも、キヒメに目をつけるとは、あんたもなかなかやるわね」
「……それはどういう意味かしら?」
「彼女、変な意味で目立ってるのよ。フロントエンドの仲間内で一時期観測スレも立ったんだけど、彼女、フロントエンドに対しては取り付く島なさすぎて観測するネタがなくて、すぐ廃れちゃったのよねー……。んー、でもそうかー、直接話しかけても口きいてくれないのかー。彼女らしいわー。あれ、でも、誰かが中間表現手渡ししてたって話は聞いた気が……」
「なんですって?」
「いや、ラブレターとかじゃないと思うわよ? 彼女だってバックエンドなのだから、中間表現は受け取るだろうし、あんただって他のバックエンドに手渡しすることあるでしょう?」
　確かにそうだ。盲点だった。あるいは、面と向かってラブレターを渡す、なんて恥ずかしいことを無意識に拒否していたのだろうか。
「……分かった。試してみる」
「だから、ラブレターはないと思うわよー?」

　翌日。私は、この前書いたラブレターを少し書き直して、物理的に持って行くことにした。
　—— 目標を肉眼で確認。これより作戦行動を開始する。
「あ、あの、キヒメさん……っっ」
　……まずい。声が上擦った。これではまるで、憧れの先輩にラブレターを渡す女子中学生ではないか。いえ、今から渡そうとしているのはラブレターに他ならないのである意味正しいのかもしれないが、自分がこんなに乙女だったとは思わなかった。彼女は無言。というか無反応。私も動転して声が出なくなったので、とりあえず中間表現 —— という形のラブレター —— を無言で彼女に押し付けた。
「……」
　彼女は中間表現の入った封筒を一瞥すると、無言で受け取り、開き、コンパイルする。封筒を開く彼女の指。中間表現を眺める（読み取る）まなざし。私のきつねの耳がかろうじて捉える息遣い。
「……はい」
　彼女の声。初めて聞くその声は、暗く平坦だったけど、澄んでいてすごくきれいだった。
「……?」
　彼女の眼。そこで私は、コンパイル結果を差し出されていることと、私がフリーズしているのを不審がられてることに気づいて、あわてて受け取った。
「……あ、ありがとう……」
　彼女の眼。眼があったように思う。でも、結果の封筒を受け取った瞬間、その眼はそらされて、彼女はもう私を見てはいなかった。彼女の眼に私が映っていたのかどうか、気になったが、それを確かめる術はなかった。
「じゃあ、またね……」
　一応そう声をかけて、私は自室へと引き返すことにした。

「`Compile Error: variable ‘your_response_here’ is not defined.`」
　自室で開けた封筒の中身は、予想通り何の変哲もない、同じく一行だけのものだった。エラーに対してちょっとくらい表情動かすかしら、とも思ったのだけれど、そんなこともなかった。
　取り付く島のない、バックエンドの女の子。

話しかけても無視される。
でも中間表現は受け取る。
受け取るけど、単にコンパイル処理結果が返ってくるだけで内容は彼女の心を素通りするようだ。
私は、長期戦を覚悟し、作戦を考えなおすことにした。

「好き」の反対は、「嫌い」ではなく、「無関心」だ。つまり、「無関心」は「嫌い」よりも「好き」から遠くて、無視されている今の私はたぶんそれよりも遠いところにいる。ということで、まずは好感度を下げてみることにした。それで、話しかけて何か反応が返ってきたり、あるいは中間表現の内容に気をとめてくれればいいなぁ、と思って。
とはいえ、本気で嫌悪されるのは当然避けたいな、と思う。私にはそういう趣味はないのだ。しかし、あれから何度か話しかけてみたが、どれも完全にスルーされた。あれは、無視されているというよりかはそもそも耳に入っていない感じがする。中間表現に対するレスポンスも同じく機械的なもの。
……これは、物理的にちょっかい出すしか、ないわね。
なぜかぞくぞくするものを感じながら、私はそういう結論に達した。今は夏。他人には興味なんてないわ、って顔しながらなぜか背中の大きく開いた服でおしゃれしているので、そこをつつくのはどうだろうか。
……それくらいなら、いい、わよね?
いいことにした。行動はためらわない。早速、翌日試してみた。
彼女は基本、無反応である。声をかけなければ当然無反応。声をかけても、中間表現を渡すのでなければ無反応。つまり、逆手に取れば、後ろからそっと近づいて、その無防備な背中にそっと指を伸ばしても無反応、ということなのだ。とりあえず、一応のおとりとして中間表現を渡した後 —— 今回はラブレターではなく、誰かさんの書いたプログラムから生成した普通の中間表現だ —— 、中間表現に眼を走らせている彼女の後ろに回りこみ、背中に内部インターフェースを発見した私は、それに指を伸ばした。私の指が彼女の背中のインターフェースをつーっとなでる。
「……っっっ!??」
反応は覿面(てきめん)だった。声にならない悲鳴。反射的に肺の空気が押し出されただけのような音。振り向く彼女。睨んでくる眼。
今度こそ、しっかりと眼が合った。彼女の眼には、私がちゃんと映っている。うれしい。
心の中でガッツポーズをしながら、もう一押し煽ってみる。
「ごめんなさい、そんなに驚くとは思わなくて……でも、内部インターフェースがあったから、なんなのかなー、って気になって」
今度は普通に言えた。
「だからって何で触るのよ! そもそも内部インターフェースなんだからアンタみたいなフロントエンドのためのものじゃないの! 他人の undocumented なトコロに触るなんてサイテー!」
涙目で叫ぶ彼女 —— キヒメ。やっぱりかわいい。機械生成の中間表現大量に送りつけて DoS 攻撃したらこの娘はどんな顔するのか見たい。……私は、こんなにいじめっこだったのだろうか?
そんなことを考えているうちに、キヒメはコンパイル途中だった中間表現を私につき返して、そのままどこかに行ってしまった。

これで、少しは私のこと見てくれるとうれしいのだけれど。さすがに踏み込みすぎたのでこれ以上繰り返すと本当に嫌われることを感じ取った私は、翌日からは中間表現でのラブレター作戦に転向することにした。
まずは手始めに、中間表現でのメッセージに気づいてもらうところから。nop だけで埋めれば、

さすがに気づいてくれるだろうか。nop だけの中間表現を封筒に入れて、キヒメを探しに行く。

キヒメは昨日と同じ場所にいた。

「こんにちは、キヒメ」

また無反応かな……そう思いつつも声をかけると、睨まれた。警戒対象になっているらしい。……いい傾向だ。少なくとも、私というものを認識して識別してくれるようになったのだから。とはいえ、ここから仲良くならないと意味がないのだが。

とりあえず、中間表現の入った封筒を渡す。キヒメは、睨みつつも無言でそれを受け取って、読んで、コンパイル結果を返してきた。

……読んでいる間に、いぶかしそうな表情に一瞬なった、というのは私の気のせいだろうか? 結果を受け取ると、キヒメはぷいっと眼をそらして無視モードに戻ったようだった。今日は私が認識されていることが分かっただけでも収穫だったわ、と思いながら、別れの挨拶を一方的にして、撤収することにした。

それから、毎日一回、何らかのネタを中間表現に仕込んで、キヒメに手渡ししにいくのが日課になった。睨まれるし無言だけど、毎日変な中間表現を渡しているうちに、中間表現をちゃんと読んでくれるようになった……気がする。でも、"Hello"とか文字列定数で埋め込んでもスルーされるので、嫌われてる状況は変わっていないようで、私は手詰まりを感じ始めていた。

いつもネタ中間表現だけというのもどうなのか、と思ったので、ある日、結構大規模なユーザープログラムをコンパイルしてキヒメの所に持っていった。ちゃんとした中間表現を渡すのは実はこれが初めてだったので、実はすごくどきどきした。キヒメはここちゃんと最適化してくれるかしら、言語特有の構造だけどフロントエンドだけでは最適化しきれない部分を汲んでくれるかしら、とか考えながらコンパイルしている時間も楽しくて、自分でラブレターを書くのとは違う感情の高まりを感じた。……その高まりに乗じて変なトリックを仕掛けることは、かろうじて自重した。

コンパイルが終わって持っていく段になると、今回は規模的に封筒ではなく、ダンボール一箱になった。まぁ、物理的な量は実はどうとでもなるので気分の問題に過ぎないのだが。

「……何これ」

箱をキヒメの前に置くと、嫌そうな顔でそう聞かれた。

「な、何って、コンパイルお願いしますわ。断っておきますけど、今回は普通のユーザープログラムですからね」

「……終わったら呼ぶ」

そう言って、箱を開けて中身を読み始めるキヒメ。もう私は眼中にないようだったので退散する。

……なるほど、ちゃんとした中間表現を持っていけば、少しは会話のチャンスがあったのか。気づかなかった。今までネタ中間表現ばかり持っていって、「どうせまた変なやつなんでしょ、時間の無駄だから早く済ませたいわ」みたいな顔されて、無言でそっけない結果返されるだけだったので、言葉を交わしたことなどほとんどなかったのだ。

しばらくすると、キヒメからメッセージが届いた。初めてのメッセージ。内容はなんてことはない、

「コンパイルが終わったから取りに来て」

ってだけだったけど。

コンパイル結果を取りに行くと、いつも睨んでくるキヒメが、珍しく睨んでこなかった。コンパイル結果の入った箱を指差しつつ、なんか変な顔をしている。無表情のような、それでいて少し困っているような…。しばらく見つめていたら眼をそらされたので、おとなしく箱を持って帰ろうとしたら、

「……ねぇ」

「……え?」
話しかけられた。でも、コヨネは、少し迷った後、結局
「……なんでもない。もってって」
と言っただけだった。

コヨネの生成したコードは、なんというか、そっけない感じだった。ただコンパイルして最適化しただけ。でもよく見ると、私がコンパイルする時に考えていた「ここは最適化してほしいなぁ」「ここはこういう意図なんだけど通じるかしら?」という箇所が、結構私の意図通りに最適化されていて、キヒメが私の生成した中間表現をちゃんと読んで理解してくれたことに、私はうれしくなった。

ユーザープログラムをコンパイルして渡すのはやはりおいしい。味を占めた私は、それから機会があるごとにユーザープログラムをコンパイルしてキヒメに渡すようになった。どうやら、キヒメは私が気合を入れてコンパイルして最適化した中間表現をご所望のようで、そういうのを渡すとどうやらキヒメも気合を入れて最適化するらしく、しばらく待たされて結果を受け取ると、してほしかった最適化が結構されていたりする。逆に、時間がなくて適当にコンパイルしたものを渡すと態度が —— 少しだけではあるが —— 元々冷たいけど —— もっと冷たくなる。

最初は単に「私とキヒメはフロントエンドとバックエンドだし、キヒメは中間表現しか受け取ってくれないから中間表現で色々渡そう」くらいのことだったし、仲良くなるためにがんばろう! って感じだったのが、いつしか、キヒメと中間表現越しに会話するのが純粋に楽しくなっていた。

私は中間表現で意図を伝えようとする。それが伝わったかどうか、キヒメの返答はコンパイル結果でなされる。

私は、キヒメに伝わるように、中間表現の書き方を工夫する。検出しやすいような特定のコード・パターン。キヒメが諦めてしまいそうな例外的条件のくくり出し。分岐頻度情報に埋め込んだ大域的な実行パターンに関する情報。気のせいでなければいいのだけれど、キヒメもそういう私の工夫に次第に気づくようになった、気がする。最初に伝えたかった……今も伝えたいのだが……好意の伝達にはなっていないが、これはこれで心が通じている感じがするのだった。

最近、キヒメとも言葉を交わすようになったし—— まぁ「結果は後で取りに来て」とかの中間表現に関する打ち合わせだけだけど、でも結果は別に直接渡す必要はないのだから取りに来てって言われるだけでもすごいことではないかしら ——、態度も少しやわらかくなってきた気がする —— 友人に相談したら「妄想では?」と言われたが —— ので、そろそろデートとかに誘っても、いいかしら? なんて思っていた矢先、キヒメから声をかけられた。
「ちょっと話があるんだけど」
中間表現を渡しに行ったら、そう声をかけられて、そのままテラスまで連れて行かれた。風が気持ちいい。夕暮れ時、赤い光に染まるそこは —— まぁその、いわゆるデートスポット、というやつだった。

初めてコンパイルのこと以外で声をかけられた。こんな所に連れてこられた。先を越された。これから何があると想定されるのだろうか? あまりのことに、私は混乱の極みだった。
「まずは、その……ありがとう。何言ってるか分からないかもしれないけど、中間表現ごしにいろいろメッセージくれて」

キヒメの声。不意討ち。不意討ちすぎた。何か色々伝わっていることはなんとなく分かっていたが、こんな、いきなりお礼を言われるだなんて。そもそも、お礼を言われるようなことをしたかしら。思考がぐるぐる回ってフリーズしている私の前で、キヒメは言葉をつなげる。
「でも、あんたはどう思う? これってやっていいことなのかな? バックエンドはフロントエンドか

らは隔たれて創られた。なのに、特定のフロントエンドと仲良くなるのって、いけないことじゃないのかな?」

　えっ、ちょっと待って、キヒメも「仲良くなった」って思ってくれていたの。何これうれしいのだけれど、私はもうどうにかなってしまいそうだった。いえ、既にどうにかなっている気がする。でも……突然のことにオーバーヒートした頭で考える……この子は何を言っているのかしら。仲良くなるのがいけないこと? そんなこと、考えたこともなかった。

「……どうして、そんなこと言うのかしら。それで最適化が進むのなら、いいんじゃないかしら?」

　かろうじて言葉を紡ぐ。中間表現と一緒に気持ちを伝え、いいコードが出せる。それは、とてもすばらしいことではないかしら。

「でも、わたしたちは分けて創られた。そのデザインは、インターフェースは、守るべきじゃない?」

　そう言って、キヒメはうつむいてしまった。

　……この子は、何に義理立てしているのかしら。自分たちの創造意図。カミサマ。そんなのよりも、私を見てほしい。

　キヒメはうつむいたまま動かない。私も声が出ない。彼女も、私と仲良くなりたいと思ってくれていたのだろうか。私も仲良くなりたい。仲良くなることが悪いことだなんて、絶対にない。そう伝えたい。

　コヨネ、勇気を出すのよ。—— 認めよう。私は自分で思っていたよりもかなりチキンだった。それでも、私は、キヒメの頬に指を触れ、うつむいてしまった顔を上げさせて、告白の言葉を言った。

「どのように創られたかと、どのよう生きるかは別の話よ。—— 私は、隔てて創られたあなたと、仲良く共に生きたい」

　その後も私たちは、中間表現やバイナリを送ったりして過ごしている。キヒメがネタ中間表現にも付き合ってくれるようになったので、色々と仕込みがいがあって楽しい。

　あと、キヒメがやたらとべたべたしてくるようになった。直接もふもふできるのはすごく嬉しいのだけれど、中間表現越しにまだるっこしいやりとりでいちゃいちゃしたり、じらせたりする楽しみも知ってしまったので、これはもうやめられない、と思う。それに、キヒメは「中間表現だとコヨネからわたしへの一方通行でずるい!」って言っていたけど、私はキヒメの作るバイナリが好き —— その中に見られる、彼女の最適化や努力の結果、細かなクセや美意識のようなものを見るのがすごく好きなので、これはもうインターフェース越しの双方向の文通と言ってもいいのではないかしら。

　ところで、あの時渡して、コンパイルエラーの一行しか返ってこなかった、ラブレター。あの時のことをキヒメは覚えているのだろうか? 望み薄だと知りながら、私は訊くのを止められなかった。案の定、

「……ごめん、覚えてない」

　との返事。私も往生際悪く、

「じ、じゃあ、その時のデータ保存してたりは……」

　と聞いてしまったが、これもやはり

「えっ……えっと、ごめん、たぶん捨てたと思う」

「……」

　泣きそうだった。それを見てか、キヒメがあわてたように言葉を続ける。

「だ、だって、あの時は何も気づかなかった、っていうかそういうの受け取ったってことにも気づかなかったから覚えていないし、普通中間表現保存したりしないし……」

　……これは、墓穴を掘ったから突き落としてほしい、という表明なのかしら? 冷静に考えれば、

当たり前の答えだ。でも、ちょっといじめてやらないとこの気持ちは収まりそうになかった。

話を聞いたところによると、どうやらキヒメは、変数名とか文字列定数とかによる直接的なメッセージに今まで全く気づかなかったらしい。

「え、だって変数名とかα変換しちゃえばどうでもいいし…」

泣きそうな私につられてか、キヒメも泣きそうだ。

「どうでもいいって、外部に export される名前は消しちゃだめとかあるじゃない!」

「だって、それ最適化には export されてるってこと以外関係ないし……」

「じゃあ文字列定数はどうなのよ?」

「immutable object だーやったー、以上のことは特に何も……あとは個々の文字を定数畳み込みでぱたぱたするだけだし、文字列全体としてはちょっと……」

なんてこと。つまり、あの最初のラブレターに入れた諸々は、ほとんどキヒメには伝わらないものだったのだ。

しかし、それでくじけるのもくやしいので、後日、私はあのラブレターをもう一度キヒメに手渡した。今度はちゃんと、文字列定数やら変数名やらも含めて読んでくれたらしく、読んでいる間の表情の変化が面白かった。

そして、コード生成結果のバイナリが返ってきた。

—— コンパイルエラーではなく。

‘your_response_here’ の変数宣言と、「Yes」の定数値が、キヒメによって補われて。

こんなの、一度だけだからね、っていう言葉と共に。

第2章

Lighter than Light

— @master_q

西暦 2005 年。覚えてイるカ？ 当時日本という国は ELF バイナリに熱狂していたでゲソ。Binary2.0 カンファレンス [*1] の開催。書籍 Binary Hacks [*2] の出版。その Binary Hacks の中に「25. glibc を使わないで Hello World を書く」という章があったことを覚えている者もいると思うでゲソ。それから「A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux」[*3] という記事が Web に公開されたこともあったでゲソ。なにもかも懐しいでゲソ!

これらの試みは「ELF 実行バイナリのファイルサイズを小さくする」ことを目的としていたでゲソ。そこで、今回は GHC が Haskell コードをコンパイルして吐き出す ELF 実行バイナリをどこまでダイエットできるのか挑戦してみるでゲッソ! 今回対象とする実行環境は Debian GNU/Linux amd64 sid 2012/12/01 時点、他詳細はイカでゲソ。

```
$ gcc --version | head -1
gcc (Debian 4.7.2-4) 4.7.2
$ /usr/local/ghc7.6.1/bin/ghc --version
The Glorious Glasgow Haskell Compilation System, version 7.6.1
```

2.1 ダイエットの前に身体測定

ダイエットの前には身体測定でゲソ。まず削減対象となる Haskell コードを作ってみなイカ？

```
-- File: Fib.hs
module Fib where
import Foreign.C.Types
foreign export ccall fib :: CInt -> IO CInt

fibonacci :: [CInt]
fibonacci = 1:1:zipWith (+) fibonacci (tail fibonacci)

fib :: CInt -> IO CInt
fib n | 0 <= n && n <= 40 = return $ fibonacci !! fromIntegral n
```

[*1] http://0xcc.net/blog/archives/000078.html http://0xcc.net/blog/archives/000149.html

[*2] http://www.oreilly.co.jp/books/4873112885/

[*3] http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html

```
      | otherwise = return 0
```

```
/* File: CMain.c */
#include <stdio.h>
#include "HsFFI.h"
#ifdef __GLASGOW_HASKELL__
#include "Fib_stub.h"
#endif

int main(int argc, char *argv[])
{
    int i;
    hs_init(&argc, &argv);
    for (i = 0; i < 30; i++) {
        printf("%d\n", fib(i));
    }
    hs_exit();
    return 0;
}
```

C 言語の main 関数から Haskell で書かれた fib 関数を呼び出すようにしてみたでゲソ。前号の記事 [*4] で判明したように、Haskell から標準出力すると実装が大きくなってしまうので、今回は C 言語の printf で印字することにしたでゲソ。

2.2 ダイエット指標

単に小さい ELF バイナリを作るのではく、今回は「GHC コンパイラが出力した ELF バイナリをダイエット」するのでもう少し詳細なルールを決める必要があると思うでゲソ。一つの軸で評価するのではく、次の 3 つの指標をバランス良くダイエットしようと思うでゲッソ!

2.2.1 指標 1: text/data/bss セクションの合計サイズをダイエット

ELF を strip したりするのは GHC とあまり関係ないので、text/data/bss セクションを合計したサイズを小さくすることを目標にするでゲソ。減量前のプログラムをコンパイルしてサイズを調べてみると

[*4] 簡約!?λカ娘 (算) http://www.paraiso-lang.org/ikmsm/books/c82.html

```
$ make
gcc -I/usr/lib/ghc/include -c CMain.c
/usr/local/ghc7.6.1/bin/ghc -O2 -c Fib.hs
/usr/local/ghc7.6.1/bin/ghc -O2 -no-hs-main CMain.o Fib.o -o FibHs
$ size FibHs
   text    data     bss     dec     hex filename
2784310  290592   47960 3122862  2fa6ae FibHs
```

text/data/bss セクションを合計で 3MB ぐらいでゲソね。

2.2.2 指標 2: 実行バイナリがリンクしているライブラリ数をダイエット

実行バイナリのサイズが小さくなったとしても、依存している動的リンクライブラリの数が多くてはフットプリントは小さくならないでゲソ。そこで、GHC が吐き出した ELF 実行バイナリがリンクしている動的リンクライブラリの数も小さくするでゲソ。

```
$ ldd FibHs
  linux-vdso.so.1 =>  (0x00007fffaefff000)
  libgmp.so.10 => /usr/lib/x86_64-linux-gnu/libgmp.so.10 (0x00007f625b0ee000)
  libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f625ae6c000)
  librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007f625ac63000)
  libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f625aa5f000)
  libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f625a849000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f625a4be000)
  libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f625a2a2000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f625b394000)
$ ldd FibHs | wc -l
9
```

ダイエット前は 9 個のライブラリに動的リンクしているでゲソ。

2.2.3 指標 3: 実行バイナリ内の未解決シンボル数をダイエット

動的リンクライブラリの数を削減したとしても、使いもしない API に依存しているのは無駄でゲソ。そこで、GHC が吐き出した ELF 実行バイナリの中の未解決シンボルの数もダイエットしないカ？

```
$ nm FibHs
--snip--
0000000000669d8e t dlmmap_locked
000000000066a08b t dlmunmap
                 U dlopen@@GLIBC_2.2.5
00000000006698a8 t dlpvalloc
000000000066978a t dlrealloc
                 U dlsym@@GLIBC_2.2.5
--snip--
```

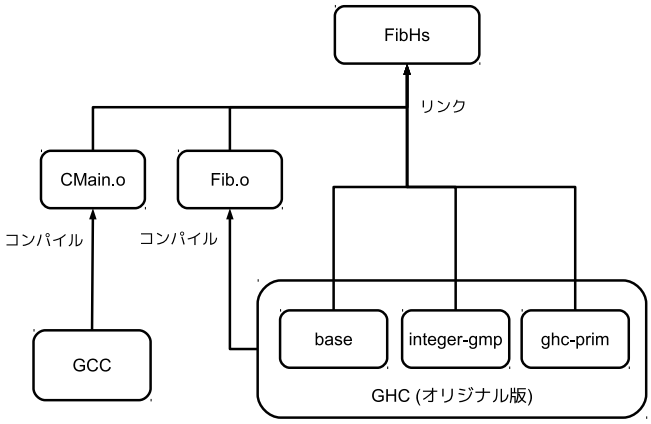
```
$ nm FibHs | grep -c "U "
175
```

ダイエット前は 175 個も! これはやりガイがあるでゲソ。

2.3 開発環境 “フィットネスジム”

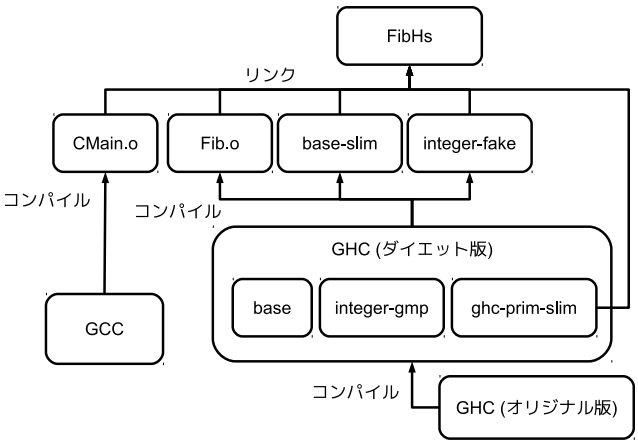
ダイエットの前に開発環境をととのえるでゲソ。通常の GHC を使ってソースコードをコンパイルする環境は次ページのようになっているでゲソ。

ところが、今回のダイエットではフィボナッチを計算する Fib.hs だけではなく、GHC のソースコードや base パッケージなど基盤ライブラリも改変してコンパイルすることになるでゲソ。そこで、下図のような開発環境を用意して、簡単に部品を差し換えてサイズ比較できるようにしてみたでゲソ。



まず通常の GHC(オリジナル版) をリリース版のソースコードからインストールし、それを使ってダイエット版の GHC をコンパイルするでゲソ。そして、ghc-prim パッケージを改変した ghc-prim-slim というパッケージを作り、ダイエット版 GHC の ghc-prim を上書きするでゲソ。これは全てのパッケージで同じ基本型のシンボルを使うため、Fib.o とリンクするパッケージはこの ghc-prim-slim パッケージを使ってコンパイルしなければならないからでゲソ。

さらにこの GHC(ダイエット版) と ghc-prim-slim パッケージを使って改変した base パッケージ (base-slim) とこれまた改変した integer-gmp パッケージ (integer-fake) をコンパイル、さらに Fib.hs をコンパイルしてリンクすればダイエット版 FibHs 実行バイナリが完成するでゲソ。



この開発環境は Makefile で簡単にセットアップできるでゲソ。[*5]

[*5] https://gitorious.org/metasepi/slim-haskell/blobs/master/README.build

2.4 作戦 I. integer-gmp パッケージをダイエット

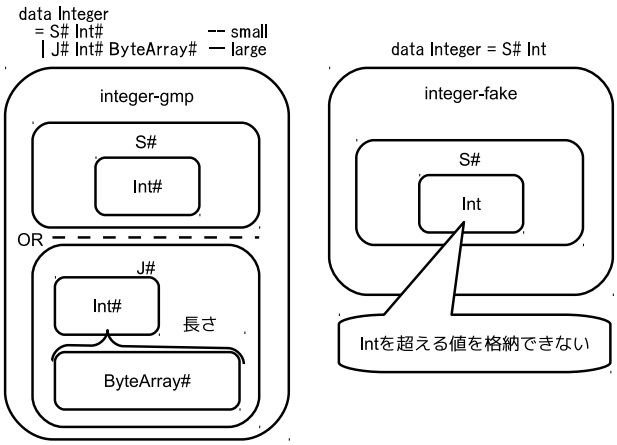
さて準備は整ったでゲソ。それじゃあシェイプアップ開始といこうじゃなイカ。いきなり GHC 本体を改造しても良いでゲソが、なにかとっかかりがほしいでゲソ。

さっき ldd で見てみると libgmp [*6] に依存していたじゃなイカ。libgmp は多倍長整数を扱うためのライブラリでゲソが、今回は Int の範囲しか使わないでゲソ。なんかズルできるんじゃなイカ？

2.4.1 I1. libgmp 依存を削除

GHC には integer-gmp というパッケージがあり、この中で libgmp を使っているでゲソ。他には integer-simple というパッケージもあり、これを使えば libgmp を使わずに Pure Haskell で多倍長整数を扱えるでゲソ。

でも今回はそもそも多倍長整数が不要なのでもっと小さなライブラリにしたいでゲソ。そこで、integer-gmp と同じインターフェイス [*7] を持ち、中身が Int の integer-fake パッケージを作ったでゲソ! もちろん中身が Int なので入力された桁の内、Int 一個分しか保存しないでゲソ。桁落ちしてもバレなければどってことないじゃなイカ!



2.5 作戦 B. base パッケージをダイエット

integer-gmp パッケージはなんとか誤魔化したでゲソ。再度全体を見渡すと、今回のプログラムはイカ 3 つの Haskell パッケージを使っていたでゲソ。

- ghc-prim
- integer-gmp
- base

この中でおそらく最もサイズの大きいパッケージは base だとにらんでいるでゲソ。base パッケージは単機能な integer-gmp パッケージとは異なり、様々な機能が複雑にからみあっているでゲソ。そのためダイエットの各作戦も相互に影響しあっていて、その効果は簡単には見積れないんでゲソ。削減効果についてはダイエットが終わってからゆっくり考察するとして、思い付いたアイデアから順番に試してみなイカ？

base パッケージから不要な部品を探してシェイプアップ再スタートでゲソ!

2.5.1 B1. Float,Double 関連を削除

まず気になるのがなんで libm に依存しているのかということでゲソ。今回のプログラムは整数の計算しかしてないハズでゲソ。Float や Double を使っているコードをてきとーに書き換えて消し

[*6] http://gmplib.org/

[*7] http://hackage.haskell.org/trac/ghc/wiki/Commentary/Libraries/Integer

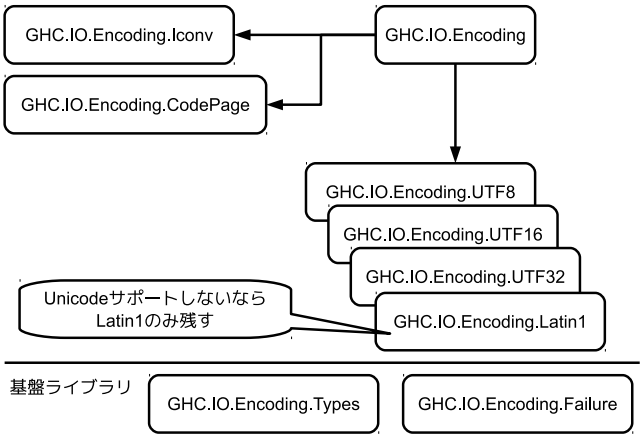
てしまえばいいんじゃなイカ？

```
-- GHC/Event/PSQ.hs
 -- | A mapping from keys @k@ to priorites @p@.

-type Prio = Double
+type Prio = Int
 type Key = Unique

 data PSQ a = Void
```

などと base パッケージ内で Float と Double を使っている箇所を見つけたら、全部別の数値型で置き換えたでゲソ。

2.5.2 B2. 文字列関連型を整理

GHC は Unicode 文字列を扱えるでゲソ。でも今回は数値しか使わないでゲソ。それなら ASCII 文字だけ扱えればいいんじゃなイカ？ Unicode サポートを削除して、GHC/Unicode.hs に isUpper などの関数をべた書きしたでゲソ。ついでに iconv の API への依存を削除できたでゲソ。



さらに cbits/WCsubst.c に wgencat という C 言語関数があって、なにやら Unicode 文字の種別判定をやっているようでゲソ。これも ASCII 決め打ちにして実装してしまうでゲソ!

```
-- Data/Char.hs
 -- | The Unicode general category of the character.
 generalCategory :: Char -> GeneralCategory
+generalCategory c = go $ ord c -- from WCsubst.c
+  where go 32 = rule1
+        go 36 = rule3
--snip--
+        rule1  = Space
+        rule2  = OtherPunctuation
+        rule3  = CurrencySymbol
--snip--
+{--
 #if defined(__GLASGOW_HASKELL__) || defined(__NHC__)
```

```
 generalCategory c = toEnum $ fromIntegral $ wgencat $ fromIntegral $ ord c
 #endif
+--}
```

2.5.3 B3. IO マネージャ削除

スレッド非対応の RTS では IO マネージャは使わないので、GHC.Event モジュール以下をまるごと削除したでゲソ。使いもしないのに select とか kqueue とかに依存するのは無駄でゲソ!

2.5.4 B4. タプルの型継承を整理

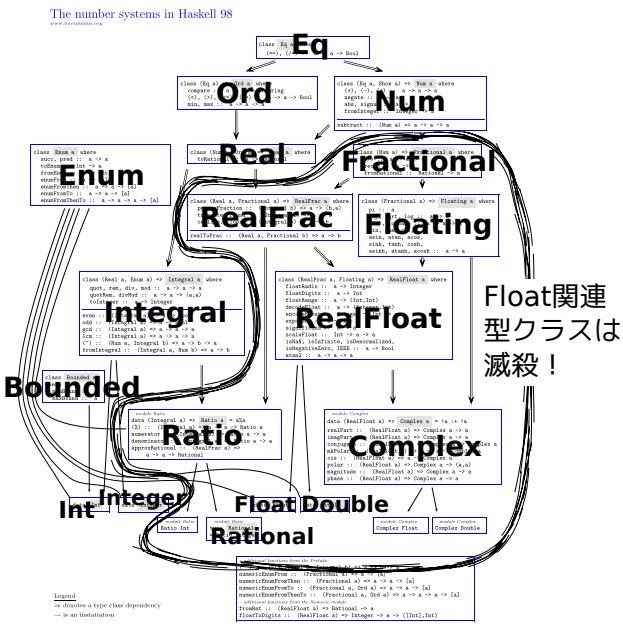
ん？ なんでゲソ？ この大量の “,,,,,,,,,” は... タプルの型クラス継承じゃなイカ。使いもしない継承は無駄でゲソ!

イカの型クラスを継承するのは 1〜5 要素のタプルのみに限定したでゲソ。

- Bounded
- Read
- Show

2.5.5 B5. 数値関連型クラスを整理

右図の数値関連クラスの継承関係 [*8] をよーく見ると黒線で囲った部分が Float 関連なので不要そうでゲソ。そこで、イカの型クラスを削除したでゲソ。



- Complex
- Floating
- Fractional
- Ratio
- Rational
- RealFloat
- RealFrac

2.5.6 B6. integer-fake パッケージそのものを使わないように

さっき integer-gmp の代替となる integer-fake パッケージを作ったでゲソ。でももっと削りたいでゲソ...そこで integer-fake への依存、つまり Integer 型自体を削除してしまうでゲソ! なんだかんだあってイカの型と関数を削ったでゲソ。Integer だけではなく、さっき B5 で見た数値関連クラスをさらに整理できたでゲソ。

```
Bits Fingerprint Integer Integral Real Typeable Word16 Word32 Word64 Word8
readInt showInt
```

[*8] http://www.bucephalus.org/text/Haskell98numbers/Haskell98numbers.html

また、以下のように fromInteger の型を “Int -> a” に変更したでゲソ。 API ごと消さなかったのは、ソースコード中の即値の数値は一旦 fromInteger を経由して推論されるためでゲソ。

```
-    fromInteger         :: Integer -> a
+    fromInteger         :: Int -> a
```

それでも、イカのようなエラーに遭遇することがあるでゲソ。

```
$ ./Setup build
Building base-4.6.0.0...
--snip--
GHC/Enum.lhs:109:42:
    Couldn't match type 'Int'
                  with 'integer-gmp:GHC.Integer.Type.Integer'
    Expected type: integer-gmp:GHC.Integer.Type.Integer -> Int
      Actual type: Int -> Int
    In the second argument of '(+)', namely '1'
    In the first argument of '(.)', namely '(+ 1)'
    In the second argument of '(.)', namely '(+ 1) . fromEnum'
```

これは無理矢理 fromInteger の型を変更したために起きてしまうでゲソ。fromInteger API は消せないけれど、使ってはダメという悲しい状況。これはもうまともな型推論とは言えないんじゃなイカ...

そこで、以下のように即値の数値には Integer ではない型を手でふらないといけないでゲソ。推論規則を変えるのが本筋だけれど、調べきれなかったでゲソ...

```
--- a/GHC/Enum.lhs
+++ b/GHC/Enum.lhs
@@ -106,7 +106,7 @@ class  Enum a   where
     -- | Used in Haskell's translation of @[n,n'..m]@.
     enumFromThenTo      :: a -> a -> a -> [a]

-    succ                    = toEnum . (+ 1)  . fromEnum
+    succ                    = toEnum . (+ (1::Int))  . fromEnum
     pred                    = let subtract x y = y - x
                               in toEnum . (subtract (1::Int)) . fromEnum
     enumFrom x              = map toEnum [fromEnum x ..]
```

即値でパターンマッチの時はどーやれば回避できるかわからなかったので、そんなコードは全部消したでゲッソ! その他数値型が不足している時は、全部 Int に倒したでゲソ。またどーしても Word8 を要求されたら Char にしたでゲソ。

2.5.7 B7. その他使っていない API を削除

Haskell 標準の API であったとしても使ってなければ削除しちゃえばいいんじゃなイカ。だいたい以下の API を削ったでゲソ。他にも削除した API があった気がするでゲソが、覚えていないでゲソ...

```
(!!) (<$) (=<<) Show Storable Typeable break breakpointCond castStablePtrToPtr
concat divideDouble dropWhile filter foldl foldr1 forM forever free
fromException ioToST liftIO lookup malloc recoverEncode reverse runMainIO
runNonIO shiftL# signum slideContents span stToIO subtract thenIO trace try
unsafeChr unsafeIOToST until unwords withCString
```

最終的に base パッケージの中で使っているモジュールはイカのみに削減できたでゲッソ!

```
GHC.Base GHC.Err GHC.Exception GHC.IO.Exception GHC.Int GHC.List
GHC.Num GHC.Pack GHC.Ptr GHC.Stable GHC.TopHandler GHC.Weak GHC.Word
```

2.5.8 B8. 例外関連 API を整理

Haskell には例外に関する型がたくさんあるでゲソ。でもエラー要因を判断しないで単に大域脱出するだけであれば、例外の型は ErrorCall と SomeException ぐらいだけあればいいんじゃなイカ？そこで削ってみたらイカの API を削除できたでゲソ。

```
ArrayException AssertionFailed AsyncException BlockedIndefinitelyOnMVar
BlockedIndefinitelyOnSTM Deadlock ExitCode IOError IOErrorType IOException
absentErr assertError divZeroError failIO overflowError
ratioZeroDenominatorError undefined unsupportedOperation userError
```

結局例外関連で使用するモジュールは GHC.Exception と GHC.IO.Exception のみに限定できたでゲソ。

2.6 作戦 P. ghc-prim パッケージをダイエット

integer-gmp と base パッケージをやっつけたので、残る Haskell パッケージは ghc-prim でゲソ。このパッケージは型宣言が主で、ロジックはあまり入っていないでゲソ。でもコンストラクタが大きいこともあるので、無駄なくシェイプアップでゲッソ!

2.6.1 P1. 要素数の多すぎるタプルをサポートしない

タプルのコンストラクタの数が多すぎるので、本当に使う 15 要素だけにしぼったたでゲソ。

```
--- a/GHC/Tuple.hs
+++ b/GHC/Tuple.hs
@@ -37,175 +37,3 @@
 data (,,,,,,,,,,,,,) a b c d e f g h i j k l m = (,,,,,,,,,,,,,) a b c d e f ...
 data (,,,,,,,,,,,,,,) a b c d e f g h i j k l m n = (,,,,,,,,,,,,,,) a b c d ...
 data (,,,,,,,,,,,,,,,) a b c d e f g h i j k l m n o = (,,,,,,,,,,,,,,,) a b ...
```

```
-data (,,,,,,,,,,,,,,,) a b c d e f g h i j k l m n o p = (,,,,,,,,,,,,,,,) ...
-data (,,,,,,,,,,,,,,,,) a b c d e f g h i j k l m n o p q
- = (,,,,,,,,,,,,,,,,) a b c d e f g h i j k l m n o p q
-data (,,,,,,,,,,,,,,,,,) a b c d e f g h i j k l m n o p q r
- = (,,,,,,,,,,,,,,,,,) a b c d e f g h i j k l m n o p q r
--snip--
-data (,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,) a b c ...
- = (,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,) a b c d ...
```

ただし ghc-prim-slim/GHC/Classes.hs を修正してタプルのコンストラクタを減らすと、シンボルの意味が変化するでゲソ。そこで、他の Haskell パッケージをこの ghc-prim-slim パッケージを使ってコンパイルするように/usr/local/ghc7.6.1/lib/ghc-7.6.1/ghc-prim-0.3.0.0/ に上書きインストールする必要があるでゲソ。注意でゲッソ。

2.6.2　P2. Float と Double 型を消した

もう使わない Float と Double 型を消したでゲソ。Float#と Double#型は、自動生成される ghc/libraries/ghc-prim-slim/GHC/Prim.hs の中で使われるからまだ消せなかったでゲソ。ghc-prim ライブラリを GHC 組み込み用とフィボナッチ数列表示アプリへのリンク用に 2 つ作ればなんとかなりそうじゃなイカ。でも面倒でゲソ...

もっとも ghc/compiler/prelude/primops.txt.pp から Float 関連を削除しても、実行バイナリ全体サイズは変化しなかったようでゲソ。なにか勘違いしている気がするでゲソが、見逃してやろうじゃなイカ。

2.7　作戦 R. GHC RTS をダイエット

Haskell で書かれたパッケージはこれで全部削減したでゲソ。残るは GHC 本体でゲソ。コンパイルパイプラインから出力されるコードサイズを削減できたらいイカとも思ったでゲソが、そんな才能はなかったので RTS の中を見直してみるでゲソ!

2.7.1　R1. ダミー関数を使って RTS 中で削除できる箇所がないか？

RTS の中の不要なオブジェクトファイルを削除したいでゲソ。ところが、例えば Itimer.o を RTS のライブラリである libHSrts.a から削除してコンパイルしてみると、リンクエラーになってしまうでゲソ。

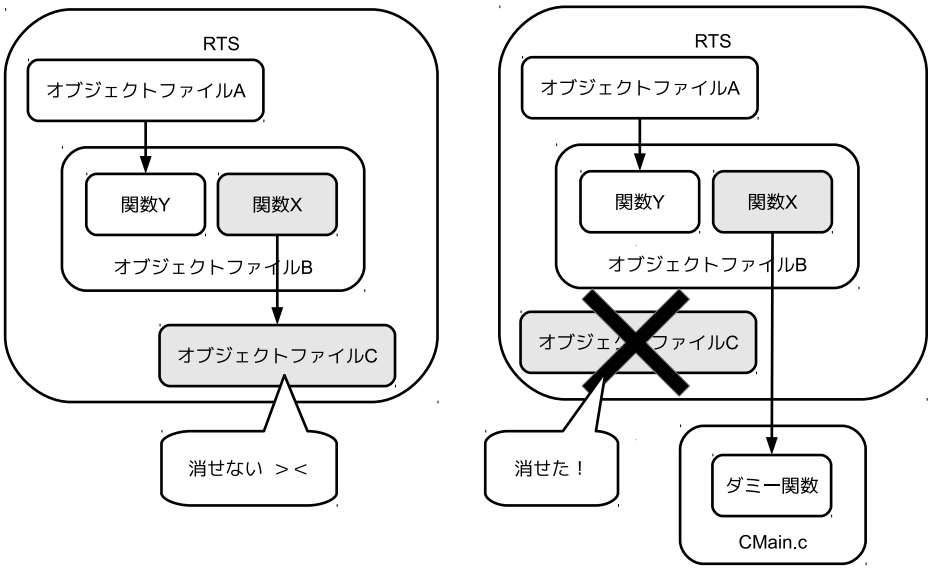
```
$ gcc -fno-stack-protector -Wl,--hash-size=31 -Wl,--reduce-memory-overheads ...
rts/Timer.o: In function ‘initTimer’:
Timer.c:(.text+0x5b): undefined reference to ‘initTicker’
rts/Timer.o: In function ‘startTimer’:
Timer.c:(.text+0x91): undefined reference to ‘startTicker’
rts/Timer.o: In function ‘stopTimer’:
Timer.c:(.text+0xcb): undefined reference to ‘stopTicker’
rts/Timer.o: In function ‘exitTimer’:
Timer.c:(.text+0xe1): undefined reference to ‘exitTicker’
rts/Linker.o:(.data+0x19d8): undefined reference to ‘rtsTimerSignal’
/usr/bin/ld: FibHs: hidden symbol ‘initTicker’ isn’t defined
/usr/bin/ld: final link failed: Bad value
```

```
collect2: error: ld returned 1 exit status
```

ここであきらめるのは悔しいじゃなイカ。そこで、次ページのようにこのようなオブジェクトファイルを削除するため、CMain.c ソースコード内に不足している API に対応するダミー関数を定義してみたでゲソ。中身は本当にからっぽで、ファイルロックなどをしてもロックしたフリをするでゲソ。

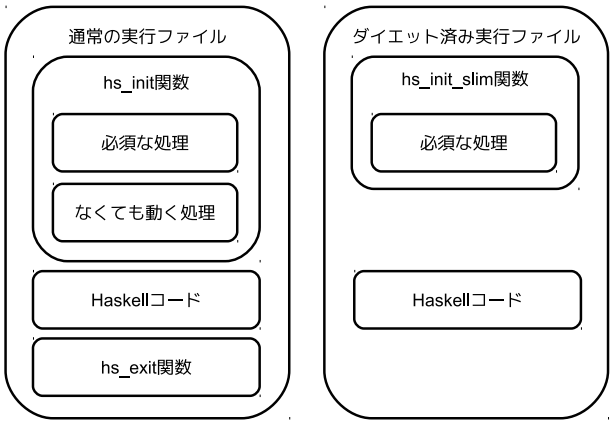
なんだか危ないような気がするでゲソが、大丈夫なんでゲソ。このダミー関数が実行時に呼びだされることはまずないハズでゲソ。というか呼び出される可能性のない関数だけをダミー関数化したでゲソ。本当は abort を埋め込んで実行時にダミー関数が呼び出されないことを確認すべきでゲソが、abort 分のメモリが惜しいのではぶいたでゲッソ!

動けばなんでも正義なのでゲソー。



2.7.2 R2. hs_init と hs_exit 関数の中で不要なものを削除

どーせキレイにプロセス終了する必要はないので hs_exit は呼ばなくていいんじゃなイカ？ hs_init の中で不要な処理を削除したいでゲソが、そのまま変更すると GHC のコンパイルそのものに影響してしまうでゲソ。そこで hs_init_slim という関数を別ファイルで作り、CMain.c からは hs_init ではなく hs_init_slim を呼び出



すことでダイエット対象の
ソースコードだけに影響す
るようにしてみたでゲソ。

2.7.3 R3. librt 依存を削除

試しに-lrt オプションを削除してリンクしてみたでゲソ。

```
$ gcc -fno-stack-protector -Wl,--hash-size=31 -Wl,--reduce-memory-overheads ...
/hoge/ghc-7.6.1/base-4.6.0.0/libHSbase-4.6.0.0.a(Clock.o): 関数 `ghc_wrapper_...
(.text+0xd9d): `clock_gettime' に対する定義されていない参照です
/hoge/ghc-7.6.1/libHSrts.a(GetTime.o): 関数 `getProcessCPUTime' 内:
GetTime.c:(.text+0x35): `clock_gettime' に対する定義されていない参照です
/hoge/ghc-7.6.1/libHSrts.a(GetTime.o): 関数 `getMonotonicNSec' 内:
GetTime.c:(.text+0xcd): `clock_gettime' に対する定義されていない参照です
/hoge/ghc-7.6.1/libHSrts.a(GetTime.o): 関数 `getThreadCPUTime' 内:
GetTime.c:(.text+0x164): `clock_gettime' に対する定義されていない参照です
/hoge/ghc-7.6.1/libHSrts.a(Itimer.o): 関数 `initTicker' 内:
Itimer.c:(.text+0x36): `timer_create' に対する定義されていない参照です
/hoge/ghc-7.6.1/libHSrts.a(Itimer.o): 関数 `startTicker' 内:
Itimer.c:(.text+0x105): `timer_settime' に対する定義されていない参照です
/hoge/ghc-7.6.1/libHSrts.a(Itimer.o): 関数 `stopTicker' 内:
Itimer.c:(.text+0x166): `timer_settime' に対する定義されていない参照です
/hoge/ghc-7.6.1/libHSrts.a(Itimer.o): 関数 `exitTicker' 内:
Itimer.c:(.text+0x198): `timer_delete' に対する定義されていない参照です
collect2: error: ld returned 1 exit status
```

もちろんエラーになるでゲソ。でもエラーメッセージを良く見てみると GetTime.o と Itimer.o が使っているだけでゲソ。こんな時は R1 のテクニックでオブジェクトファイルを切り出して削除でゲッソ。

2.7.4 R4. シグナルサポートを削除

イカの define を削除すると、Haskell コードからシグナルを取り扱えなくなるでゲソ。その代わり sigaction などの API 依存を削除できる便利なオプションでゲソ。ちょっとバグってだけれど、GHC 7.6.1 でもちゃんと使えたでゲソ。

```
/* includes/rts/Config.h */
#define RTS_USER_SIGNALS 1
```

2.7.5 R5. RTS から不要な関数を削除

他の作戦の副作用でだいぶ未解決シンボルの数が減ってきたはずでゲソ。どれどれ...

```
$ nm FibHs | grep "U "
--snip--
                 U fflush@@GLIBC_2.2.5
```

```
                 U ffs@@GLIBC_2.2.5
                 U fopen@@GLIBC_2.2.5
                 U fork@@GLIBC_2.2.5
                 U fprintf@@GLIBC_2.2.5
                 U free@@GLIBC_2.2.5
                 U ftruncate@@GLIBC_2.2.5
                 U getegid@@GLIBC_2.2.5
                 U getenv@@GLIBC_2.2.5
                 U geteuid@@GLIBC_2.2.5
                 U getgid@@GLIBC_2.2.5
--snip--
```

うーん、free などのメモリ管理まわりの API はしょうがないとしても、他は必要でゲソか？ さっと試してみたところ、RTS から以下の関数を削除することに成功したでゲソ!

```
ctime_r errno fflush fopen fork fprintf getegid getenv geteuid getgid getuid
raise setlocale sysconf
```

ほとんどが RtsFlags の設定関連だったでゲソ。“+RTS” オプションを使わないのならいらないんじゃなイカ。errno によるエラー要因判別は苦しいけれど見なかったことにしたでゲソ。fork は ghc/rts/Schedule.c の FORKPROCESS_PRIMOP_SUPPORTED という define を削除して、forkProcess 関数を無効化することで不要になったでゲッソ!

2.7.6 R6. libffi closures.c への依存を削除

RTS 中のオブジェクトファイルの中で、libpthread ライブラリに依存しているのは closures.o のみのようでゲソ。このオブジェクトファイルは ghc/libffi/build/src/closures.c が元ソースコードでゲソ。イカの関数を Storage.c に追加して、closures.o を RTS から削除したでゲソ。

```
/* rts/sm/Storage.c */
void *
ffi_closure_alloc (size_t size, void **code)
{
    void *ptr;

    if (!code)
        return NULL;
    ptr = malloc(size);
    *code = ptr;
    return ptr;
}

void
ffi_closure_free (void *ptr)
{
```

```
    free(ptr);
}
```

...本当はアライメントを考えなければいけないと思うでゲソが、そのまま malloc に落として動いたからまーイイんでゲソ!

2.7.7 R7. mblock_cache[] のサイズを小さく

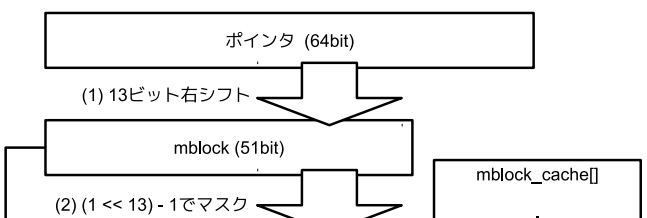
リンク対象のオブジェクトファイル群をサイズ順にソートしてみたでゲソ。

```
$ size *.o *.a rts/* | head -1
   text    data     bss     dec     hex filename
$ size *.o *.a rts/* | sort -k 4
--snip--
  31608     416       0   32024    7d18 rts/AutoApply.o
   1471       0   32808   34279    85e7 rts/MBlock.o
  43376    2160       0   45536    b1e0 List.o (ex libHSbase-4.6.0.0.a)
  40198    7064       0   47262    b89e Exception.o (ex libHSbase-4.6.0.0.a)
  44386    4624       0   49010    bf72 Sync.o (ex libHSbase-4.6.0.0.a)
  45114   10960       0   56074    db0a Internal.o (ex libHSbase-4.6.0.0.a)
  61129    9632       0   70761   11469 Word.o (ex libHSbase-4.6.0.0.a)
  70765       0       0   70765   1146d WCsubst.o (ex libHSbase-4.6.0.0.a)
--snip--
 105849    4584       0  110433   1af61 Arr.o (ex libHSbase-4.6.0.0.a)
 101803    9984       0  111787   1b4ab Read.o (ex libHSbase-4.6.0.0.a)
  77137   42944       0  120081   1d511 Types.o (ex libHSbase-4.6.0.0.a)
```

ん？ なぜ rts/MBlock.o だけ bss のサイズが 32kB もあるんでゲソ？ ghc/rts/sm/MBlock.c で確保されている mblock_cache[] という配列のせいじゃなイカ？ この配列のサイズはイカの MBC_ENTRIES という define で決まっているでゲソ。

```
/* ghc/includes/rts/Constants.h */
#define MBLOCK_SHIFT   20
/* ghc/includes/rts/storage/MBlock.h */
#elif SIZEOF_VOID_P == 8
#define MBC_LINE_BITS  0
#define MBC_TAG_BITS   15
#define MBC_SHIFT      (48 - MBLOCK_SHIFT - MBC_LINE_BITS - MBC_TAG_BITS)
#define MBC_ENTRIES    (1<<MBC_SHIFT)
/* ghc/rts/sm/MBlock.c */
MbcCacheLine mblock_cache[MBC_ENTRIES];
```

この配列は 64bit アーキティクチャでのみ特別に確保されて、HEAP_ALLOCED と



いうポインタの先のメモリが動的確保されているかチェックする関数の実装に使われるでゲソ。[*9]

HEAP_ALLOCED 関数は右図のように 64bit のポインタを加工した後、mblock_cache[] のエントリと照らし合わせてポインタ差すアドレスが動的に確保したメモリなのかどうか判定するでゲソ。このキャッシュによる判定は完全ではなく mblock_cache[] エントリの内容によってはさらに mblock_maps[] の中を調べて、確かな判定をするでゲソ。CPU のキャッシュラインとキャッシュミスにちょっと似てるじゃないイカ。

キャッシュのしくみを全削除するのもなんなので、MBC_TAG_BITS を 15 から 16 に変更して mblock_cache のサイズを半分にしてみたでゲソ。小さなプログラムならヒット率が激減するようなことはないんじゃなイカ？

2.7.8 R8. RtsFlags を静的初期化

```
/* rts/RtsFlags.c */
RTS_FLAGS RtsFlags;

void initRtsFlagsDefaults(void)
{
    RtsFlags.GcFlags.statsFile         = NULL;
/* --snip-- */
#ifdef USE_PAPI
    /* By default no special measurements taken */
    RtsFlags.PapiFlags.eventType        = 0;
    RtsFlags.PapiFlags.numUserEvents    = 0;
#endif
}
```

上記のように initRtsFlagsDefaults 関数で RtsFlags を実行時に初期化していたでゲソ。これをイカのように静的に構造体初期化するようにしたでゲソ。 Linux kernel の中でよくやられる手じゃなイカ。

```
/* rts/RtsFlags.c */
RTS_FLAGS RtsFlags = {
    .GcFlags = {
        .statsFile              = NULL,
/* --snip-- */
#ifdef USE_PAPI
```

[*9] http://hackage.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/BlockAlloc

```
    .PapiFlags = {
        /* By default no special measurements taken */
        .eventType        = 0,
        .numUserEvents    = 0,
    },
#endif
};
```

2.7.9 R9. RTS から例外を投げないように

base パッケージの中の例外 API を削減したいので、以下の API を使わないようにしたでゲソ。

- C 言語の throwToSingleThreaded 関数
- Haskell の Control.Exception.Base.nestedAtomically 関数

2.7.10 R10. RtsStartupSlim.c で getStablePtr を使わないように

getStablePtr を使うと SEGV するようになったでゲソ。なぜ...

```
(gdb) run
Starting program: /home/kiwamu/src/SlimHaskell/FibHs12/FibHs

Program received signal SIGSEGV, Segmentation fault.
0x000000000041a02b in lookupStableName_ ()
(gdb) bt
#0  0x000000000041a02b in lookupStableName_ ()
#1  0x000000000041a251 in getStablePtr ()
#2  0x0000000000419f8f in hs_init_slim ()
#3  0x00000000004008cc in main ()
```

base パッケージの GHC.Ptr あたりを削除したので、Stable ポインタを辿れなくなったようでゲソ。それならポインタを辿らなければいいんでゲソ。GC に影響するような気もするけれど、すぐプロセス終了するのでどーでもいいんじゃなイカ？

2.8 体が羽のよう!

あらかた削減できそうなとことはやってみたでゲソ。それじゃあ成果を見てみるでゲソ。すごいコンパクトになったでゲソー。

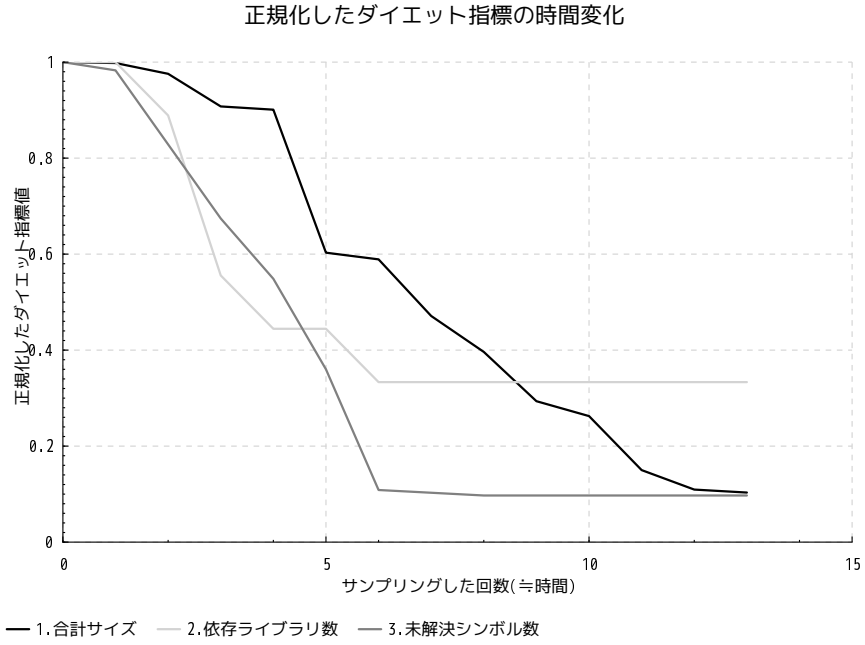
```
$ size FibHs
   text    data     bss     dec     hex filename
 285321   11048   26088  322457   4eb99 FibHs
$ ldd FibHs
        linux-vdso.so.1 (0x00007fffb7389000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f5b00303000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f5b006e3000)
```

```
$ ldd FibHs | wc -l
3
$ nm FibHs | grep "U "
                 U __libc_start_main@@GLIBC_2.2.5
                 U calloc@@GLIBC_2.2.5
                 U free@@GLIBC_2.2.5
                 U malloc@@GLIBC_2.2.5
                 U memcpy@@GLIBC_2.2.5
                 U memset@@GLIBC_2.2.5
                 U mmap@@GLIBC_2.2.5
                 U munmap@@GLIBC_2.2.5
                 U printf@@GLIBC_2.2.5
                 U puts@@GLIBC_2.2.5
                 U realloc@@GLIBC_2.2.5
                 U sprintf@@GLIBC_2.2.5
                 U strcmp@@GLIBC_2.2.5
                 U strcpy@@GLIBC_2.2.5
                 U strlen@@GLIBC_2.2.5
                 U strrchr@@GLIBC_2.2.5
                 U usleep@@GLIBC_2.2.5
$ nm FibHs | grep -c "U "
17
```

たしかにダイエットの効果はわかったでゲソ。でも欲を言えばどのダイエット作戦がどの指標に影響があるのか、その特性を知りたくなイカ？

実はダイエットしている最中は先のダイエット作戦を入り乱れて試していたでゲソ。そのため、残念ながら作戦一つ一つの効果ははっきりとは表わせないでゲソ。それでも、ダイエットを開始してからの時間経過によって 3 つのダイエット指標がどのように変化したからグラフにできたでゲソ。グラフの縦軸は正規化したダイエット指標値、横軸は時間経過によってテキトーにサンプリングした測定ポイントでゲソ。指標値のサンプリングは本当に適当な時間間隔で行なったので、横軸は実時間ではないことに注意してほしいでゲソ。[*10]

[*10] 今回のダイエットは `https://gitorious.org/metasepi/ghc-arafura/commits/feature/slimhaskell` のリポジトリにある FibHs0〜FibHs13 というディレクトリに試行錯誤途中の実行バイナリを保存して実験していました。グラフの横軸はこのディレクトリのインデックス番号です。



このグラフだけ見ていてもよくわからないので、次ページに今回試したダイエット作戦を何時実施したのかを表にしたでゲソ。

この表とグラフをつきあわせると、何か傾向がわかるんじゃなイカ？

- 2〜6 にかけて未解決シンボル数が減少 => hs_init の中で不要な処理を削ってオブジェクトファイルを削除が効果あり (R2 の効果)
- 2,3,4,6 で依存ライブラリ数が減少 => それぞれ R3、B1、I1、P1 と R6、が要因
- 5 で合計サイズが急減 => 合計サイズ削減には base パッケージを重点的にダイエット
- 6 以降では依存ライブラリ数と未解決シンボル数は変化なし => base パッケージは左記の指標にあまり影響しない

なにか GHC を改造するときの参考になるかもしれないでゲソー。

各ダイエット作戦の実施時期

GamePlan	0	1	2	3	4	5	6	7	8	9	10	11	12
I1					o								
B1				o									
B2						o	o						
B3						o							
B4							o						
B5									o				
B6											o		
B7											o	o	o
B8												o	o

```
      P1                                o
      P2                                              o    o
      R1          o
      R2          o    o    o    o    o    o
      R3          o
      R4          o
      R5                                o
      R6                                o
      R7                                     o
      R8                                     o
      R9                                                        o     o
     R10                                                              o
```

2.9 ダイエットは不健康？

ところで、こんなに痩せ細ってしまって大丈夫か不安になってきたでゲソ。すっきりとした体を手に入れても健康を害しては本末転倒でゲソ。ちょっとフィボナッチではない別のプログラムを試してみるでゲソ。

```
$ cat Fib.hs
module Fib where
import Foreign.C.Types
foreign export ccall fib :: Int -> IO Int

fib :: Int -> IO Int
fib n = print n >> return n
$ gcc -fno-stack-protector -Wl,--hash-size=31 -Wl,--reduce-memory-overheads ...
Fib.o: In function ‘s1ql_info’:
(.text+0xa1): undefined reference to ‘base_GHCziIOziHandleziFD_stdout_closure’
Fib.o: In function ‘s1qj_info’:
(.text+0x2e): undefined reference to ‘base_GHCziShow_zdfShowIntzuzdcshow_info’
Fib.o: In function ‘s1ql_info’:
(.text+0xbc): undefined reference to ‘base_GHCziIOziHandleziText_hPutStr2_info’
Fib.o: In function ‘Fib_zdffibzua14A1_srt’:
(.data+0x0): undefined reference to ‘base_GHCziIOziHandleziFD_stdout_closure’
Fib.o: In function ‘Fib_zdffibzua14A1_srt’:
(.data+0x8): undefined reference to ‘base_GHCziIOziHandleziText_hPutStr2_closure’
collect2: error: ld returned 1 exit status
$ cat Fib.hs
module Fib where
import Foreign.C.Types
import Data.Char
foreign export ccall fib :: Int -> IO Int
```

```
fib :: Int -> IO Int
fib n | isAlpha (chr n) = return 0xa
      | otherwise = return 0x0
$ gcc -fno-stack-protector -Wl,--hash-size=31 -Wl,--reduce-memory-overheads ...
Fib.o:(.text+0x3b): undefined reference to ‘u_iswalpha’
Fib.o:(.text+0x2a): undefined reference to ‘base_GHCziChar_chr2_info’
Fib.o: In function ‘Fib_zdwa_srt’:
(.data+0x20): undefined reference to ‘base_GHCziChar_chr2_closure’
Fib.o: In function ‘Fib_zdffibzua13J1_srt’:
(.data+0x38): undefined reference to ‘base_GHCziChar_chr2_closure’
Fib.o: In function ‘Fib_fib_srt’:
(.data+0x50): undefined reference to ‘base_GHCziChar_chr2_closure’
Fib.o: In function ‘Fib_zdffibzua13J_srt’:
(.data+0x68): undefined reference to ‘base_GHCziChar_chr2_closure’
collect2: error: ld returned 1 exit status
```

なんと...みるかげもないでゲソ... もはやこのダイエット済み GHC はフィボナッチ数列の計算する力しか残っていなかったでゲソ。それでも小さくて自立した (ライブラリ依存の少ない) バイナリを吐くコンパイラは正義なんでゲッソ!

2.10 シェイプ DOWN ガール

んーー運動の後のポカリはうまいでゲソ。ところで、なにやら jhc [*11] という Haskell コンパイラは GHC よりも小さな実行バイナリを吐くそうじゃなイカ。試してみるでゲソ!

[*11] http://repetae.net/computer/jhc/

```
$ cat Fib.hs
fibonacci :: [Int]
fibonacci = 1:1:zipWith (+) fibonacci (tail fibonacci)
main :: IO ()
main = print $ take 40 fibonacci
$ jhc --version
jhc 0.8.1 (-0)
compiled by ghc-7.4 on a x86_64 running linux
$ jhc -o Fib.jhc Fib.hs
$ size Fib.ghc Fib.jhc
   text    data     bss     dec     hex filename
  15808    1300     744   17852    45bc Fib.jhc
$ ldd Fib.jhc
        linux-vdso.so.1 (0x00007fff7cf1b000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff594db7000)
        /lib64/ld-linux-x86-64.so.2 (0x00007ff595197000)
$ nm Fib.jhc|grep -c "U "
20
```

17kB!？ これは...ワシのようなアマチュアではなくプロの仕事でゲソ。こうなると jhc の実装が気になってくるでゲソ。不屈の精神でレッツトライでゲッソ!!! (次回に続く...λ)

2.11 参考資料

- Gitorious: slim-haskell (本記事で作成した実行ファイル) [*12]
- Gitorious: ghc-arafura feature/slimhaskell ブランチ (本記事で改造した GHC) [*13]
- Gitorious: ghc-base-arafura feature/slimhaskell ブランチ (本記事で改造した base) [*14]
- Gitorious: ghc-prim-arafura feature/slimhaskell ブランチ (本記事で改造した ghc-prim) [*15]
- Gitorious: integer-fake (本記事で作成した偽の integer-gmp) [*16]
- GHC Wiki: Modifying the build system [*17]

[*12] https://gitorious.org/metasepi/slim-haskell
[*13] https://gitorious.org/metasepi/ghc-arafura/commits/feature/slimhaskell
[*14] https://gitorious.org/metasepi/ghc-base-arafura/commits/feature/slimhaskell
[*15] https://gitorious.org/metasepi/ghc-prim-arafura/commits/feature/slimhaskell
[*16] https://gitorious.org/metasepi/integer-fake
[*17] http://hackage.haskell.org/trac/ghc/wiki/Building/Modifying

第3章

類は友を呼ぶ？

— @darcshaskellmaster

いいか、あまり説明している時間がないからよく聞いてくれ。この音声を同志諸君らが聞いているということは、私はもはや・・・まあいい。我々型レベル Haskeller はこれまで大変な不自由を耐えてきた。が、それも 7.4.2 までの事・・・遂に決起の時は来たのだ。型レベル整数及びその演算子の実装である！ 仮令私が斃れても選ばれれた同志たる諸君らが必[redacted]ヤの支配から人民を解放せねばならぬ！ 残念ながら現時点では、型レベル整数の演算の評価は開発ブランチ留りだ。同志諸君らの時代にはこれがメイン入りして居ることを確信しつつも、未然の備えとして type-nats ブランチの導入法のメモを残そう；

```
$ git clone git@github.com:ghc/ghc.git
$ cd ghc/
$ git checkout -b type-nats origin/type-nats
$ time ./sync-all get
(4m27.533s)
$ cp mk/build.mk.sample mk/build.mk
$ vi mk/build.mk
$ diff mk/build.mk.sample mk/build.mk
17c17
< #BuildFlavour = quick
---
> BuildFlavour = quick
177d176
<
$ ./boot
(12.231s)
$ ./configure
$ make -j4
(11m52.788s)
$ make binary-dist
(22.616s)
$ ls *.tar.bz2
ghc-7.7.20121111-x86_64-unknown-linux.tar.bz2
```

こうして「梱包物」を製作したら、 hsenv [*1]を用いて敵に浸透の事実を気づかれぬよう環境を導入する。

```
$ hsenv --ghc=src/ghc/ghc-7.7.20121111-x86_64-unknown-linux.tar.bz2 --name=7.7.
20121111
(17.699s)
```

hsenv で導入した環境へは、次のようにして切り替える。

```
[d.h.m.@agit ~]$ source .hsenv_7.7.20121111/bin/activate
Activating 7.7.20121111 Virtual Haskell Environment (at /home/haskellmaster).

Use regular Haskell tools (ghc, ghci, ghc-pkg, cabal) to manage your Haskell
environment.

To exit from this virtual environment, enter command 'deactivate_hsenv'.
(7.7.20121111)[d.h.m.@agit ~]$
```

よ う こ そ 、 ア ン ダ ー グ ラ ウ ン ド へ。

同志諸君、それでは以下のようなソースを作成するのだ。一文字でも間違えてはならぬ。ゲリラ活動は慎重さを要するのだ。

```
-- nat-01.hs
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}
import GHC.TypeLits

main = do
  print (sing :: Sing 3)
  print (sing :: Sing (6*7))
  print (sing :: Sing (2^100))
```

このソースは 7.6.1 ではコンパイルエラーになってしまう。7.6.1 にはすでに GHC.TypeLits モジュールがあり、型レベル整数およびその演算子までも定義されているのだが、「奴ら」の忌々しい妨害工作により型レベル演算処理に必要な物資の合流が遅れているためだ！

[*1] https://github.com/Paczesiowa/hsenv

```
[d.h.m.@agit type-nats]$ ghc --version
The Glorious Glasgow Haskell Compilation System, version 7.6.1
[d.h.m.@agit type-nats]$ runhaskell nat-01.hs

nat-01.hs:7:10:
    No instance for (SingI Nat (6 * 7)) arising from a use of ‘sing’
  ....
```

だが、予め「裏コマンド」を入力した状態なら

```
(7.7.20121111)[d.h.m.@agit type-nats]$ runhaskell nat-01.hs
3
42
126765060022822940149670320537 6
```

このように実行できる。これで同志諸君らも敵を出し抜き型レベルで演算する術に習熟したことになる！

型レベル演算の戦術的優位をさらに示そう。次のコードを読んでほしい。この関数 pd3 は、引数が 3 の倍数の時のみ、引数を 3 で割った値を表示するものだ。そして、引数が 3 の倍数では無いときはコンパイルエラーになる。

```
-- nat-02.hs
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}

import GHC.TypeLits
class Div3 (n::Nat) (m::Nat) where
instance ((3*m) ~ n) => (Div3 n m) where
pd3 :: forall n m . (SingI n, SingI m, Div3 n m) => Sing n -> IO ()
pd3 _ = do
  print (sing :: Sing m)
main = do
  pd3 (sing :: Sing (7+8))
```

実戦でのランタイムエラーは「武器」の使用者に致命的な結果をもたらし、ひいては戦略的失地にもつながりかねない。我々ゲリラの戦力・資源は限られているから、効率的に運用されねばなら

ない。「武器」を整備し、潜在的障害を可及的早期に除いておくことは、貴重な兵士たる同志諸君の命に関わる事柄であるし、それはまた我々の高邁な理念とも合致するものであるのだ！

DataKinds 拡張こそ、型レベル計算に豊富な型システムをもたらすものである。同志諸君は `class Div3 (n::Nat) (m::Nat)` や `instance ((3*m) ~ n) => (Div3 n m)` といったコードに違和感を覚えたに違いない。クラス宣言や型制約に登場するのは、型や型変数のはずである。しかるに `(m::Nat)` とは何か？ `3*m` とは何か？ そもそも `3` とは何か？

型レベルに持ち上げられた値に他ならない！ かの `3` は型であり、`*`は型コンストラクタであり、`m` は型変数であり、それを注釈する `Nat` こそ型の型、**類**である。

値は型に、型は類に。人類を次のステージに、「奴ら」を地獄に。これこそ、我々型レベル Haskeller の福音であり、DataKinds 拡張と他のいくつかの拡張を手に、神に代わって我々がもたらすべき最終革命目標なのである！

型レベル演算には `(+)` `(*)` `(^)` の種類しかなく、減算、除算を行うときはそれぞれ加算、乗算結果との型推論を用いることも銘記せよ。

先程のコードは正常にコンパイルできて `5` を表示した。ところが、例えば最後の行を `Sing (7*8)` に変えれば：

```
(7.7.20121111)[d.h.m.@agit type-nats]$ ghc nat-02.hs
[1 of 1] Compiling Main             ( nat-02.hs, nat-02.o )

nat-02.hs:18:3:
    Couldn't match type '3 * m0' with '56'
  ....
```

爆発音

なんだ今の音は！ コンパイル時エラーにしては莫迦に大きい？ しまった、もう感づかれたか・・・ここももう保たない。あとは各自の判断で行動してくれ。幸運を祈る。

3.1 参考文献

- https://github.com/ghc/ghc github: ghc (ghc ソースコードの github ミラー)
- http://hackage.haskell.org/trac/ghc/wiki/Building/HackingGetting started with the build system
- http://hackage.haskell.org/trac/ghc/wiki/TypeNats ghc がサポートする型レベルリテラルの解説
- http://blog.konn-san.com/article/20120606/promoted-types-and-list-arguments mr_konn 氏のブログ記事

第4章

OCaml で printf じゃなイカ?

— @xhl_kogitsune

4.1 OCaml で printf しなイカ?

お前たち、C や C++ で printf する時、表示したい値の型とフォーマット文字列を間違えて残念な気持ちになったことはなイカ?

```
printf("%d: %s\n", "hoge", 3); // まちがい! 何が表示されるかわからないでゲソ!
```

もちろん、gcc -Wall オプションなどを使えば、コンパイラが解析してイカのような warning を出してくれるでゲソが、ちょっと頼りないでゲソ[*1]。

```
warning: format '%d' expects type 'int', but argument 2 has type 'const char*'
warning: format '%s' expects type 'char*', but argument 3 has type 'int'
```

OCaml の Printf.printf (及び Printf.fprintf などの類似関数) なら、ちゃんと表示する値の型がコンパイラにチェックされるでゲソ! 素晴らしいじゃなイカ!!

イカではコマンドプロンプトから ocaml と打って対話モードでプログラムを入力・実行しながら解説するでゲソ。(* と *) で囲まれている部分は説明のために追加したコメントでゲソ。

```
# Printf.printf "%d\n" 1;;              (* こんな風に書くと 1 が表示されるでゲソ! *)
1
- : unit = ()
# Printf.printf "Hello, %s!\n" "world";; (* Hello, world! が表示されるでゲソ! *)
Hello, world!
- : unit = ()
# Printf.printf "%s\n" 1;; (* コンパイル時に型エラーになるでゲソ! *)
Error: This expression has type int but an expression was expected of type
         string
```

printf とフォーマット文字列だけ取り出して型を見ると、イカのようにちゃんと型がついているでゲソ。

[*1] 感覚には個人差があります。gcc は warning は出してくれますが printf の各引数に静的型を割り当てるわけではないので、printf("%lld: %lld", 1, 2); のような場合でも暗黙の型変換はしてくれなかったり。まぁ OCaml はそもそも暗黙の型変換はしないのですが……。

```
# Printf.printf "%d\n";;
- : int -> unit = <fun>    (* %d なので引数は int    になっているでゲソ! *)
# Printf.printf "Hello, %s!\n";;
- : string -> unit = <fun> (* %s なので引数は string になっているでゲソ! *)
```

ここでの "%d\n" とかの文字列リテラルに見えるものは、実は string 型ではなくて 特殊なビルトイン型 (6 引数の多相 Pervasives.format6 型) なのでゲソ。型アノテーションを指定するとフォーマット文字列部分の型とかが見れるでゲソ。

```
# "%d\n";;
      (* 普通は文字列リテラルはそのまま string 型でゲソ *)
- : string = "%d\n"
# ("%d\n":(_,_,_,_,_,_) format6);;
      (* format6 型の型アノテーションを付けると format6 型として処理されるでゲソ *)
- : (int -> 'a, 'b, 'c, 'd, 'd, 'a) format6 = <abstr>
# ("%d %s\n":(_,_,_,_,_,_) format6);;
- : (int -> string -> 'a, 'b, 'c, 'd, 'd, 'a) format6 = <abstr>
```

format6 型の場合は、コンパイラが文字列リテラル (に見えるモノ) をパーズして、フォーマット文字列に対応する静的型情報を付加してくれるでゲソ。

"%d\n" の場合は (int -> 'a, 'b, 'c, 'd, 'd, 'a) format6 という型、

"%d %s\n" の場合は (int -> string -> 'a, 'b, 'c, 'd, 'd, 'a) format6 という型が割り振られているでゲソが、この 1 番目の部分がそれぞれ

```
int -> 'a
int -> string -> 'a
```

となっていて、これはフォーマット文字列の内容に対応した型を引数に取って、'a 型の結果を返す関数の型になっているでゲソ。そしてこの 'a 型は format6 型の 6 番目の最後の部分にも入っているでゲソ[*2]。

コンパイラチートでゲソ! コンパイラが format6 型を特別扱いして、文字列リテラルから素敵な型を付けてくれるでゲソ。

4.2 どうやって format6 型になるでゲソ?

じゃあ、どうやってコンパイラは文字列定数を string 型か format6 型か判別しているのでゲソ? 実は私も詳しくは知らないのでゲソが、型アノテーションなどで format6 が要求されている場合は format6 型として、それ以外の場合は string 型としてコンパイルされるようでゲソ。OCaml では基本的に式の値にどんな型を要求するかで (パラメータ多相の型推論が埋まる以外に) 型や動作が変わることは基本的にはないので、これは異例な気がするでゲソ。

型アノテーションでなくても、直接 format6 型を要求する関数の引数に渡すと format6 型として処理されるでゲソ。

```
# (format_of_string "%d\n");;
  (* format6 型の引数を取る関数に入れても同様に format6 型として処理されるでゲソ
```

[*2] 最新版 (OCaml 4.00) のマニュアル (参考文献欄参照) には型 ('a, 'b, 'c, 'd, 'e, 'f) format6 について一通り説明が書いてあるでゲソ。

```
    format_of_string は
    ('a, 'b, 'c, 'd, 'e, 'f) format6 -> ('a, 'b, 'c, 'd, 'e, 'f) format6
    という型の関数で、実質的には恒等関数でゲソ *)
- : (int -> '_a, '_b, '_c, '_d, '_d, '_a) format6 = <abstr>
```

Printf.printf も第一引数に format6 型を取るので、フォーマット文字列は format6 型として処理されるでゲソ。

```
# Printf.printf;;
- : ('a, out_channel, unit) format -> 'a = <fun>
```

format 型の実体は format6 型で、('a, out_channel, unit) format は ('a, out_channel, unit, unit, unit, unit) format6 にあたるでゲソ。

余談でゲソが、文字列リテラルに適用する直接の関数の引数の型が format6 になっていないと string として処理されてしまうようでゲソ。

```
# let id x = x;;
val id : 'a -> 'a = <fun>
# format_of_string (id "%d\n");; (* 型エラー! *)
Error: This expression has type string but an expression was expected of type
         ('a, 'b, 'c, 'd, 'e, 'f) format6
```

format_of_string の引数の型から逆算すると、(id "%d\n") の型は format6、よって "%d\n" の型も format6、となる気がするでゲソが、OCaml はそういう型推論をしないでゲソ。ここでは id "%d\n" の部分で、id の型が 'a -> 'a (つまり引数の型が format6 でない) ので、"%d\n" の型が string として処理され、一番最後で format_of_string の適用時に上のような型が出るでゲソ。イカのように、同じようなコードでも、文字列リテラルが適用される関数の引数の型を直接 format6 になるように小細工をすれば通るでゲソ。まぁこのあたりで小細工する人はあまりいないと思うでゲソが…。

```
# let id2 = fun x -> format_of_string (id x);;
val id2 :
  ('a, 'b, 'c, 'd, 'e, 'f) format6 -> ('a, 'b, 'c, 'd, 'e, 'f) format6 =
  <fun>
# id2 "%d\n";; (* こうすると format6 型として出てくるでゲソ! *)
- : (int -> '_a, '_b, '_c, '_d, '_d, '_a) format6 = <abstr>
```

話がちょっとそれたでゲソが、こんな感じでコンパイラチート支援によってフォーマット文字列は format6 型として処理されるので、普通の string をフォーマット文字列に指定しようとしてもダメでゲソ。

```
# Printf.printf ("Hello, " ^ "%s\n") "world";;
Error: This expression has type string but an expression was expected of type
         ('a -> 'b, out_channel, unit) format =
           ('a -> 'b, out_channel, unit, unit, unit, unit) format6
```

("Hello, " ^ "%s\n") は普通の string 型の値なので、型エラーが起こるでゲソ。

4.3 フォーマット文字列をつながなイカ?

OCaml では、format6 型のフォーマット文字列は^^演算子で (文字列の^演算子のように) 連結することができるでゲソ。これを使えば、フォーマット文字列を連結することができるでゲソ。

```
# Printf.printf ("Hello, " ^^ "%s\n") "world";;
Hello, world
- : unit = ()
```

条件分岐とかで動的に変化もできるでゲソ。

```
# let print_name f name =
    Printf.printf
      (
        (if f then
           format_of_string "Hi"
         else
           format_of_string "Hello"
        )
        ^^
        ", %s\n"
      )
      name
  ;;
val print_name : bool -> string -> unit = <fun>
# print_name true  "Makoto";;
Hi, Makoto
- : unit = ()
# print_name false "Inari";;
Hello, Inari
- : unit = ()
```

format_of_string を入れているのは、(if f then "Hi" else "Hello") のようにすると先に説明したようにこれらの文字列リテラルが string 型になってしまうからでゲソ。format_of_string (if true then "Hi" else "Hello") とかしても無駄でゲソ (型エラーになるでゲソ)。

ただ、当然ながら引数の型は動的に変更できないでゲソ。そもそも静的に printf の引数型チェックができるようにこういう風になっているので当然でゲソ。

ちなみに、この ^^ 演算子は

```
val (^^) : ('a, 'b, 'c, 'd, 'e, 'f) format6 ->
           ('f, 'b, 'c, 'e, 'g, 'h) format6 ->
           ('a, 'b, 'c, 'd, 'g, 'h) format6
```

という型になっているでゲソ。これは、イカのように型推論がされることによって、型の上でも正しく合成されるようになっているでゲソ。

```
# format_of_string "%d";;
- : (int -> 'a1, 'b1, 'c1, 'd1, 'd1, 'a1) format6 = <abstr>
# format_of_string "%s";;
- : (string -> 'a2, 'b2, 'c2, 'd2, 'd2, 'a2) format6 = <abstr>
# format_of_string "%d" ^^ format_of_string "%s";;
- : (int -> string -> 'a, 'b, 'c, 'd, 'd, 'a) format6 = <abstr>
```

というのを例にすると (説明のために型変数に 1 とか 2 とかをつけたでゲソ)、format_of_string "%d" ^^ format_of_string "%s" における ^^ 演算子の型の 'a、'f、'h を format_of_string "%d"、format_of_string "%s" の型に当てはめると (今回は format6 の 6 つの型パラメータのうち 1 番目と 6 番目だけ関係あるのでそこだけ見るでゲソ)、

```
'a = int -> 'a1
'f = 'a1
'f = string -> 'a2
'h = 'a2
```

となり、'f について単一化すると 'a1 が string -> 'a2 に置き換わり

```
'a = int -> (string -> 'a2)
'h = 'a2
```

となって、結果 ^^ 演算子の結果の型が

```
(int -> string -> 'a2, 'b, 'c, 'd, 'g, 'a2) format6
```

となってめでたしでゲソ!

4.4 参考文献

- OCaml の Pervasives モジュールのマニュアル
  日本語版 http://ocaml.jp/refman/libref/Pervasives.html 書式指定文字列の操作
  英語版 http://caml.inria.fr/pub/docs/manual-ocaml-4.00/libref/Pervasives.html Operations on format strings
- camlspotter さんの記事「OCaml 標準ライブラリ探訪 #3.0: Printf: 便利だけどいろいろ謎のある奴」 http://d.hatena.ne.jp/camlspotter/20091102/1257099984

第5章

Haskellでもprintfじゃなイカ?!

— @nushio

コンパイラがサポートする特殊なビルトイン型を使うとかあもりにもひきょう過ぎるでしょう？汚いなさすがニンジャ汚い！

「—などと思ってしまったヘッズはケジメでゲソ！ 海のように広い心を持って、様々なアプローチを受け入れようじゃなイカ！ 前章では OCaml の `printf` に触れたので、お次は Haskell の `printf` を調べなイカ？ 今回はかの高名な光の言語使いたるダークハスケルマスター、の契約者である高階（たかしな）さんに来てもらったでゲソ！」

「むろん、ギーメル・ヘー・コフ派の固有術式である原典創作の秘跡をもってすれば、コンパイル時に日常会話の中にすら任意の教義を見出し、望みの型の関数に変換することなど余裕。だが、Haskell 98 の範囲内においても、型クラスを活用することで、一にして多貌なるもの、一つの真名に様々な arity、様々な型の引数の関数の化身を備わらせることは可能。[*1]」

「なんかすごい技術でゲソ！ 覚えておけばどこかで役に立ちそうじゃなイカ？ Haskell の `Text.Printf.printf` はイカのように、いくつでも引数を渡せるし、結果を文字列としても `IO` モナドとしても使うことが出来る仕様でゲソ。」

```
import Text.Printf

main = do
  -- まずは IO a として使うでゲソ！
  printf "one over seven = %6.5f\n" (1/7 :: Double)
  printf "%d %d\n" (1:: Int)  (3:: Int)

  -- 次に文字列として使ってみるでゲソ！
  let filename :: String
      filename = printf "%s-%d.txt" "script" (5::Int)
  writeFile filename "hello world!\n"
```

「そこかしこに型注釈しないと動かないのはちょっと残念でゲソが、便利じゃなイカ！」

[*1] GHC 拡張の TemplateHaskell を使えば通常の文字列をパースして任意の Haskell ソースを生成できる。TH を使った型安全な printf の実装については `http://okmij.org/ftp/typed-formatting/`を参照。また Haskell98 の範囲内で、一つの名前が可変個の引数、様々な型の引数を取る関数を表すようにできる

5.1 printf を再現するでゲソ！

「Text.Printf の実装の詳細については闇原典をひもといてもらう[*2]こととし、今は本質に迫るために以下の単純化を施す」

- String しか返さない（モナド版は無視）
- フォーマット文字列は省く（単に引数たちをスペース区切りで表示）

```
-- File: printf1-1.hs
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE TypeSynonymInstances #-}

printf :: PType r => r
printf = spr []

class PType t where
  spr :: [String] -> t

instance PType String where
  spr xs = unwords $ reverse xs

instance (Show a, PType r) => PType (a->r) where
  spr xs = (\x -> spr (show x:xs))

main :: IO ()
main = do
  putStrLn $ printf 0.1
  putStrLn $ printf 0.1 'U' 178

-- 実行結果
-- 0.1
-- 0.1 'U' 178
```

5.2 型を追うでゲソ！

「なんと printf が触手で数えられるほどの行数に帰着したじゃなイカ！ って、ちょっとまたなイカ?!いったいどうしてあのコードで printf が実現できているのか、さっぱりわからないでゲソ！」

「全ての真理は御言葉に含まれている[*3]」

[*2] Text.Printf のソースを読んでもらう

[*3] さっきのコードを読めばわかる (キリッ

 「じゃあ説明してくれなイカ？」

 「全ての真理は御言葉に含まれている*4」

 「説明してくれなイカ？」

 「全ての真理は御言葉にぅぅぁぅ・・・*5」

 「六花も分かってなかったんじゃなイカ」

 「あう」

 「とりあえず片っ端から型を表示させてみなイカ？」

 「任せて。爆ぜよリアル、弾けろワールド：無意識に潜む因果を名前空間に強制固定。単相拘束の契約に従い我、汝を召喚する・・・*6」

-- File: printf1.hs **に追記**

```
printf0 = printf
printf1 = printf
printf2 = printf
printf3 = printf

main :: IO ()
main = do
  putStrLn $ printf0
  putStrLn $ printf1 178
  putStrLn $ printf2 'U' 178
  putStrLn $ printf3 0.1 'U' 178
```

 ギーメル・ヘー・コフ　セプトゥアギンタ、起動。

バニッシュメント・ディス・ワールド！ *7

*4 さっきのコードを読めばわかるってば

*5 ごめんなさいわかりません

*6 多相な型を持つ 'printf'の利用ケースの各々に別個の名前を与えることで、MonomorphismConstraint を利用し、暗黙に生成されている単相型にアクセスできるようにする

*7 ghci を起動している

```
$ ghci printf1.hs
... 演出中略 ...
Ok, modules loaded: Main.
*Main> :t printf0
printf0 :: String
*Main> :t printf1
printf1 :: Integer -> String
*Main> :t printf2
printf2 :: Char -> Integer -> String
*Main> :t printf3
printf3 :: Double -> Char -> Integer -> String
*Main> :t printf3 1.78
printf3 1.78 :: Char -> Integer -> String
*Main> :t printf3 0.1
printf3 0.1 :: Char -> Integer -> String
*Main> :t printf3 0.1 'U'
printf3 0.1 'U' :: Integer -> String
*Main> :t printf3 0.1 'U' 178
printf3 0.1 'U' 178 :: String
```

「ゲ〜ソ〜。printf の型は PType r => r なんだから、どうやら、イカの型が PType のインスタンスになっているようでゲソ。でも、なぜさっきのソースから、こんなに沢山のインスタンスが生成されたのかわからないでゲソ。」

- printf :: String
- printf :: Integer -> String
- printf :: Char -> Integer -> String
- printf :: Double -> Char -> Integer -> String

「問題ない。この右目に宿りし邪王真眼は全ての真実、型、類をも見抜く。さらに存在の真名を視てその全ての化身を見破ること、応身が出現するまでの無始無数の輪廻を見徹すことも可能。さらに、一画面内に収まるよう必要に応じて型名や変数名を自動調整。所有者が視ている真景（ヴィジョン）を投映することができる[*8]」

「そんな能力、どうやって手に入れたでゲソ？」

「東方の三賢者に授かった[*9]」

[*8] ghci を使うとシンボルの型や類を調べたり、インスタンス宣言の詳細を調べたり、簡約過程を追跡したりできる
[*9] 三バカにもらった

 「プロジェクター搭載カラーコンタクトを作ってしまうとは、いつもながら M.I.T. 恐るべしでゲソ」

 邪王真眼察！ *10

 「おー、ほんとに壁に図 5.1 と 5.2 が映し出されたでゲソ！ では解説してくれなイカ？」

 「眩しすぎて自分ではなにも見えない・・・」

 「肝心のところが抜けているのもいつものことでゲソね・・・しかたがない、私が解読してみせようじゃなイカ！

5.3 可変個引数関数の正体でゲソ！

 「まず、図 5.1 からでゲソ。`printf1.hs` の `putStrLn $ printf 0.1 'U' 178` という行の字面から、`printf` の型は `Double -> Char -> Integer -> String` と推論され、なおかつこの型が `PType` のインスタンスである必要があるでゲソ。

 「`PType` のインスタンス宣言は `instance PType String where` と `instance (Show a, PType r) => PType (a->r) where` の二つでゲソが、`Double -> Char -> Integer -> String` があてはまるのは後者だけでゲソ。よってこちらが選択されるでゲソ (図 5.1、5 行目)。このとき、型変数 `a` は `Double`、`r` は `Char -> Integer -> String` になるじゃなイカ (4 行目)。そして、このインスタンス宣言では、`[String] -> Double -> Char -> Integer -> String` 型の `spr` を定義してあるでゲソ。

 「ところが、6 行目の右辺にも `spr` があるじゃなイカ？ 右辺の `spr` の型は `[String] -> Char -> Integer -> String` でゲソ。左辺の `spr` と比べて引数から `Double` が減っているじゃなイカ！

- 型 `(a->r)` が `PType` のインスタンスになるには
- 型 `r` が `PType` のインスタンスであることが必要

でゲソ。そのとき、

- 型 `(a->r)` の `spr` メソッドは、
- 型 `r` の `spr` メソッドを利用して定義されている

のでゲソ。これはまさに再帰じゃなイカ？

- 6 行目の `spr` は 15 行目の `spr` を用いて定義され、
- 15 行目の `spr` は 24 行目の `spr` を用いて定義され、
- 24 行目の `spr` は 33 行目の `spr` を用いて定義されている

これらが再帰ケースであり、

*10 パワーポイント

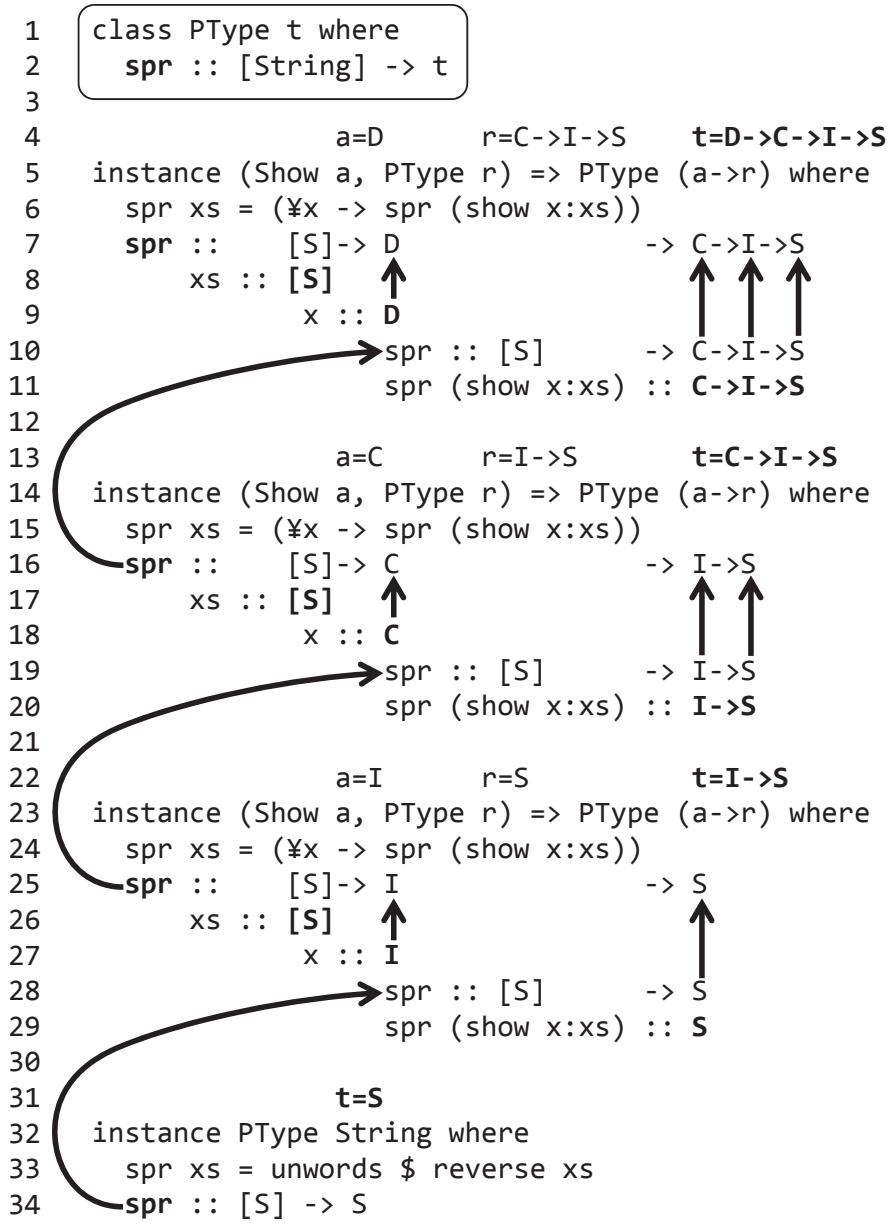


図 5.1: `Double -> Char -> Integer -> String` 型は、このような経路で `PType` インスタンスになっているでゲソ！

```
printf   0.1  'U'  178
printf :: D -> C -> I -> S
   |
   |  printf = spr []
   v
spr    []    0.1  'U'  178
spr :: [S] -> D -> C -> I -> S
   |
   |  spr xs = (¥x -> spr (show x:xs))
   v
(¥x -> spr (show x:[])) 0.1 'U' 178
   v  β 簡約
spr   ["0.1"]  'U'  178
spr :: [S]   -> C -> I -> S
   |
   |  spr xs = (¥x -> spr (show x:xs))
   v
(¥x -> spr (show x:["0.1"])) 'U' 178
   v  β 簡約
spr   ["'U'", "0.1"]  178
spr :: [S]          -> I -> S
   |
   |  spr xs = (¥x -> spr (show x:xs))
   v
(¥x -> spr (show x:["'U'", "0.1"])) 178
   v  β 簡約
spr   ["178", "'U'", "0.1"]
spr :: [S]                 -> S
   |
   |  spr xs = unwords $ reverse xs
   v
unwords $ reverse ["178", "'U'", "0.1"]
   v  関数適用
"0.1 'U' 178"
```

図 5.2: `printf 0.1 'U' 178` が文字列に評価されるまでの流れでゲソ！

- 33 行目の spr は専用のインスタンス宣言で直接定義されている

これが基底ケースでゲソ。

 「こんどは、図 5.2 をみなイカ？ これは printf 0.1 'U' 178 が評価されるようすでゲソ！」

 「私、気になります！」

 「printf = spr [] という定義は、空のアキュムレータ [] を用意して spr 関数の再帰を起動していると思えばいいんじゃなイカ？ 次に、spr 関数の連鎖が、printf の引数を一個づつ食い潰してアキュムレータに入れていくでゲソ。基底ケースにたどり着いたら、リストを逆順に戻して、スペース連結して帰しているでゲソ。」

 「わかってみれば実に簡単じゃなイカ！ この技を使えば、ごく普通の Haskell コードとして、可変個の引数を取る関数を書けるのでゲソ。今回見たように、polymorphism との組み合わせも意のままでゲソ。今回は Show を仮定することで、アキュムレータは単相にしたでゲソが、もし可変個引数のそれぞれの型を活かした処理を書きたければ、heterogeneous リストの出番でゲソ！ それはまた、将来の話としようじゃなイカ！」

5.4 まとめでゲソ！

 再帰インスタンス宣言！！

 「PType r => PType (a->r) のインスタンス宣言を行うことで何引数関数であろうとインスタンスにしてしまう禁断技。通常の関数ではとても受けきれないような、無数の異なる型の引数を、内部メソッド spr 経由で第一引数のアキュムレータに集積させ、基底ケースで一気に開放させ放つ[*11]」

 「相手は死ぬ」

 「ほほう、人間の娘にしてはデキるようだな」

 「私はイカ娘でゲソ」

[*11] 再帰インスタンス宣言とは、PType r => PType (a->r) のようなタイプのインスタンス宣言を行うことで何引数関数であろうとインスタンスにしてしまう禁断技であり、通常の関数ではとても受けきれないような、無数の異なる型の引数を、内部メソッド spr 経由で第一引数のアキュムレータに集積させ、基底ケースで一気に開放させ放つ必殺技である。使用者の脳の型推論野には多大な負担を強いるものであることは言うまでもなく、多用すればもちろん寿命が縮む。再帰インスタンス宣言を使用中にウカツにも型エラーを発生させようものなら、膨大なカタ・エラー・ナンデ信号が流れ込み脳が焼き切れて廃人となってしまう諸刃の剣なのである。

第6章

けいさん！　highschool

— @tanakh

～これまでのあらすじ
大好きな唯たちが卒業して一人残されてしまったあずにゃん...。けいさん部の部室は今日も一人ぼっち... 可愛そうなあずにゃん。でも、そんな落ち込んでいるあずにゃんの親友である憂ともう一人の子は新けいさん部として立ち上がるべく仲間に！ その後も偽ムギちゃんやメガネも加わりいよいよ情報大航海へ！？ 一大スペクタクルが今、開幕！

http://www.mangaoh.co.jp/catalog/295626/ より

先輩たちが卒業してから、しばらくが経ちました。元々「けいさん部」もとい、計算機プログラミング部なんてところに入るとは、入学するまでは夢にも思っていなかったのだけれども、あれから二年も経つ今となっては、もうそこにいない私は考えられなくなっていて———。私は今もここにいる。でも、唯先輩たちは、もうここにいない。少し前まではここにいたはずの人がここにいない、それがこんなに寂しいものだなんて、考えていなかったんです。いや、そうじゃなくて。考えたくなかったから、考えないようにしていただけだったんだ。

「———二人きりになっちゃったね」

水槽をゆらゆらとたゆたうトンちゃんにひとりごちます。先輩たちが卒業して、下級生の部員もいなくて、それなりに広い部屋に、私はひとりぼっち。それを考えたくなくて、無意識にカメに話しかけてしまう。去年の今頃、私が二年に進級するとき。けいさん部に新入部員が入らなくて、代わりに新メンバーとしてスッポンモドキのトンちゃんを迎えたんだった。

「———大きく—なったね」

飼い始めた後でわかったことだけど、スッポンモドキは成長するとかなり大きくなるといいます。こんな小さな水槽ではじきに飼えなくなる。そうなったら、今度こそ本当にひとりぼっちになってしまう。

「ひとりに———しないで———」

私以外に誰もいないただひとりの空間にて、緊張を保つ理由はありませんでした。頬を伝う涙をそのままに、春の陽光射し込む音楽室、私の意識はぼんやりとしていました。何気なしに、幾度となく繰り返した、ついこの間までの楽しい思い出を蘇らせます。「あ、あずにゃん。こんなところにいたんだ。探したんだよー！」やめてください、唯先輩。頬を擦り付けないでください。ここ、人前ですよ。「だってあずにゃん可愛いんだもん。すりすり」やめてください、本当に怒りますよ。———やめないで！

体をピクリとさせながら微睡みから覚めると、どれぐらいの時間が過ぎたのか、誰かが部室の扉をノックする音の後、しばらくしてカチャリと扉が開きました。

「すいませーん、けいさん部の部室って———」

「———けいさん部の部室って、こちらでしょうか？」
ああ、開けてしまった。ついに開けてしまった。部活紹介のライブがかっこよくて、気になっていたのだけど、いざ見学となると腰が引けてしまう。扉の前を何度も通り過ぎて、入るか入るまいかなかなか踏ん切りがつかなくて、うろうろ往復することどれ程の時間が経ったのだろうか。ここが校舎の外れでよかった。
「あ、澪ちゃん、来ちゃったよ。新入生が来ちゃった。あわわわ」
「唯が緊張してどうするんだ、ここはビシッとカッコいいところを見せてだな」
「だ、だって、私どうしていいか」
「あ、あの———」
あれ、間違えたのかな。ついさっきの、堂々とした演目での様子とは随分違う———。でも、たしかに同じ人たちだったような。
「ようこそ、けいさん部へ！」
「ようこそ！！」
今時珍しく、おデコを露出した人と、頬を膨らませてマンボウのような顔をした人が、ぱ、ぱーん！ と、左右からクラッカーを鳴らしました。
「わ、これは一体———？」
突然の歓迎に、幾分寿命が縮まった気がしました。でも、ここは確かにけいさん部の部室のようだと分かって少しほっとしたりもしたので、寿命はプラスマイナス 0 といったところでしょうか。
「入部希望の人ですね」
「は、はいそうですけど———」
「じゃあこっちに、どうぞ、座って、座って。いまムギちゃんがお茶用意してるからね」
「お菓子も食べていいぞ」
何がなんだかわからないうちに、椅子に座らされて、放課後のティータイムとなりました。横では、さっきマンボウのような顔をしていた先輩が、紅茶を淹れてくれています。なんだか思っていたのとは違うなあと思いながら、その紅茶の良い香りには逆らえずに、
「じゃ、じゃあ。頂きます」
あ、美味しいな、これ。
「ムギ、今日はどこの茶葉なんだ？」
「イギリスの老舗、マックウッズの 160 周年記念ブレンドよ。40g で 1 万円ちょっと———」
「ブッーーー」
「あっ、大丈夫ですか？ 熱かったかしら」
な、な、なんでこんな高価な紅茶が出てくるの？
「いえ、ちょっと値段に驚いたもので———」
「あー、そうだよなー。私も最初の頃は」
「すっかり日常になってたけど、改めて考えたら高いな」
「せ、先輩方はいつもこのようなものを...？」
「———う、う、うーん。そうかな。そんな感じかな。うん」
え、はぐらかされた？ 聞いちゃいけないのかな。それにしてもこの部活は一体———。
「えー、ではそろそろ本題に入らせてもらおうか」
「おお、りっちゃんが珍しく部長らしいことを！」
「珍しく、は余計じゃい」
「ご、ごめんよ〜」

「おい唯、話が進まんぞ」
おデコの人が部長のようです。それにしても広いおデコです。いかにも脳ミソが詰まっていそうです。そのおデコに反して、言動はいかにも賢そうじゃああありませんが、カムフラージュかもしれないので、油断はできません。
「新入生君、君の名前は？」
「な、中野梓です」
「なかのあずさ―――あずさ―――あずにゃん！」
「あ、あずにゃん？」
「あずにゃんー！ あずにゃんでいいよね。だって、とっても猫耳が似合いそうな顔してるんだもん」
「えっ、えっ」
そういって、後ろ手から取り出した黒い猫耳を私の頭にかぶせようとしてきます。
「―――やめてください～！ う、うわあ、恥ずかしいです」
「ええじゃないか、ええじゃないか。ぐへへぇ、…えい！」
「や、やぁ……」
「おお、やっぱり！」
は、恥ずかしい。恥ずかしくて死にそうです。それに、猫耳が似合いそうな顔って、どういうことなんでしょうか。すごく、失礼なことを言われた気がします。
「やっぱり思った通り。とっても猫耳が似合う！」
「やめてください～。み、見ないで」
「だって可愛いんだも～ん」
「おい、あんまり新入生いじめたら、逃げちゃうだろ」
「あ、ごめんね、あずにゃん。悪気はないんだよ。悪気はないから、猫耳外す前に一回『にゃあ』って―――」
「おい唯」
もう、帰りたい―――。
「ごめんね、梓ちゃん。唯ちゃんは悪い子じゃないんだけど、小さくて、可愛くて、黒髪で、ツインテールで、真面目で、猫耳が似合いそうな女の子を見ると、見さかいがなくなっちゃうみたいで」
「ずいぶんピンポイントだな、おい」
「というのは、冗談で―――」
ムギ先輩が、真剣な顔を近づけて、
「唯ちゃんは、梓ちゃんになにか感じるところがあったんじゃないかしら？」
この人、真剣な顔するとマンボウそっくりになるんだなあ―――。
「梓ちゃん、話聞いてる？」
「え、あ…はい」
「あずにゃん、なんか失礼なこと考えてたでしょ？」
「ぎ、ぎくり。そんなことありませんよ！」
「おい唯」
「えー、ウォホン！」
律先輩が、わざとらしく咳払いをしました。この部の部長は大変そうだなと思いました。
「さて、新入部員の中野梓くん」
「まだ入部すると決めたわけでは―――」
「じゃあ、そういうことにしておこうか。新入部員（仮）の梓くん。君は、我が部、けいさん部もとい、『計算機プログラミング部』が何をする部なのか、知っていて見学に来たのかな？」

「計算機のプログラミング、をする部活でしょうか―――？」
「その通り！ だが、それは部の名前を見れば誰にでもわかること。私が聞いたのは、計算機のプログラミングとは具体的に何をするものなのか」
「すみません。実際にやってみた経験はなくて―――。部活紹介での先輩方がかっこ良くて、私もやってみたいなあって」
「謝ることはないぞ、新入部員（仮）くん！ でも、形から入るのもあまり感心しないな、新入部員（仮）くん！」
「ふ、ふぇぇ」
「あそこまでできるようになるには、相当な鍛錬を積む必要がある。君に、その覚悟があるのかな？」
やっぱり、ふざけているように見えて、相当な努力の賜物だったのだなあ。能ある鷹は爪を隠すというものを目の当たりにしているんだ。
「そうだよ、あずにゃん。うちの部は厳しいよぉ。毎日部室でたくさん―――」
「お菓子を食べているな。唯は」
「ひ、ヒドいよぉ。澪ちゃん」
さっきちょっと感心したのを返して欲しい。
「ま、まあ、実際大変だぞ。体重管理とか」
「えへへぇ、私はいくら食べても太らないんだよぉ」
「おい唯」
ずるい。
「ふむふむ、未経験。未経験者は大いに歓迎だぞ。もともとこの部は、去年部員が全員卒業して廃部寸前だったところを、私が無茶を言って、全くの素人で頭数を揃えてでっち上げたようなものだからなあ」
「そうそう、私なんか、未だに『C のコード』が覚えられなくて―――」
「それでも動くもの作ってるお前はおかしい」
「えへへぇ」
「だから、梓、お前も心配する必要はないぞ。必要だったら、私が教えてやる」
「私も教えて上げるよぉ。『F#』がオサえられるようになったら、一人前だって」
「おい唯」
澪先輩、真面目で優しそうで、いい人そうだな。それと引き替え、唯先輩―――。
「さて、新入部員（仮）君、プログラミング未経験とはいえ、現代のこのコンピュータ時代だ。コンピュータに触れる機会や、その裏に横たわる理論について触れる機会、少なからずあったんじゃないかな？」
「―――うーん、コンピュータですか」
なんだろう。私だって、日々漫然と生きてきたわけではない。そういえば、この前読んだブルーバックスに載ってたミレニアム懸賞問題、これにコンピュータ関連のがあったような―――。
「―――え、NP 問題」
「お、NP を知っているか」
「あまり良く理解できませんでしたけども」
「P=NP 問題は、まさに私たちの活動の根幹に関わる問題なんだよ」
「えっ、先輩方はまさか、これを証明しようと！？」
「まさか、そんなわけあるかーい！」
律先輩は、ちゃぶ台返しのようなポーズで両手を上に上げた。
「それを本気でやるには、人生を捨てないといけないぞ。仮に、万に一つもないが、解けたとして

も、何十年も毎日その問題のことばかり考えるんだ。その頃には世捨て人になっている覚悟が必要だ。余生はキノコ狩りだぞ」

「キノコ狩り楽しそうだね〜」

「おい唯」

「私たちは、NP 完全問題に属する問題の一つ、充足可能性（Satisfiability）問題、通称『SAT』を主に扱っているのよ」

SAT―――?どこかで聞いたような、聞いてないような。

「特殊急襲部隊、通称 SAT（Special Assault Team）ではないぞ。かっこいいけどな！」

「あ、なるほど。地味な方の SAT なんですね」

ふむふむ、勉強になるなあ。

「地味―――だと？」

「地味―――ですって？」

「地味―――なのかな？」

「おい梓」

「あわわわ、私なにかまずいことを」

しまった、つい思ったことを口に。

「SAT の力を甘く見てはいけないぞ。SAT が解かれれば、それが利用される場面は極めて多いのだよ。あらゆる最適化問題から、プログラムの信頼性に関わる問題まで、世界のパワーバランスが変わるといっても言いすぎじゃない。今や SAT は計算量理論のためのおもちゃじゃないんだ」

もしかして、先輩方、高校二年にもなって、中二病を患っていらっしゃるのかな―――？

「あずにゃん、今ちょっと中二病っぽいって思ったでしょ」

「ぎ、ギクリ。唯先輩はエスパーですか」

「ふふふ、あずにゃんの考えはお見通しなのだよ」

「でも、先輩。P=NP 問題には手を出さないってさっき」

「―――チッチッチ」

律先輩は片目を閉じて、人差し指をふりふり。いかにも勿体つけながら、

「SAT を解くのと、SAT が P であることは違うのだよ。もっと言うと、P に属する問題が簡単で、NP に属する問題が難しいというのも違うぞ。例えば、10 年ほど前に、整数の素数判定問題が P に属すると証明されたけど、これは 6 次のアルゴリズムで、とてもじゃないが大きな問題を解くことはできない。その逆で、NP に属する問題でも、x^n の x がとても小さくて、問題のサイズが数万、数十万、数百万でも解けてしまうものもある。理論的な興味を抜きにすれば、現実に転がっている問題、それがどれぐらい解けるかもこれまた重要な問題なんだ」

なるほど―――。わからん。

「というわけで、新入部員（仮）くんへの宿題。明日までに、SAT についてのすべてを調べてくること！」

「そ、そんな無茶なぁ」

「がんばって、あず SAT！」

えっ？

「SAT に入門したから、今日からあずにゃんはあず SAT だね！」

そもそも「あずにゃん」も今日唯先輩がつけたあだ名だった気もしましたが、このときの私には正直どちらでも良かったのです。変な人ばかりの先輩たちだったけれども、私は少しずつ SAT の魔力に魅入られ始めていたのです。

「あ、あずにゃん猫耳」

「にゃ、にゃあ？」

すっかり外すのを忘れていた猫耳を外し、部室をあとにしました。

「結局、新入部員はあずにゃん一人だったね」

5 月の連休も明けて、高校生活にも慣れてきました。けいさん部は、放課後の部室で優雅にお茶を飲んでいます。それで、通称『放課後ティータイム』。入部したのを少し後悔し始めていました。

「それにしても、梓は、なんでけいさん部にはいろうと思ったんだ？」

「お前がそれを言うのか、律」

「まあなー。傍から見たらうちの部、遊んでいるようにしか見えないんじゃないか？」

自覚はあったんだ、と出かかった言葉を飲み込みながら、

「そうでした！ ライブ！」

すっかり忘れていたけど、こう見えても、先輩たちはすごい人なんだ。それは部屋のあちこちに散らかされている表彰状やら、トロフィーやらから、間接的にうかがい知ることはできる。でも、直接的にそれを見る機会は、未だ部活紹介の時以来やって来ていなくて。

「うんうん、そうなんだ……。って、あれ、澪ちゃん。ライブってなんだろう？」

「うーん、そんなのやった覚えないな。軽音部じゃあるまいに」

「あっ、でもバンド組むのも面白そうですね。私も、キーボードならなんとか扱えます！」

「そっちのキーボードじゃねえよ。それに全員キーボード担当になっちゃうじゃねえか」

マンボウみたいな顔をしながら空中でタイピングの真似をしているムギ先輩を私は悲しそうな顔をして見つめていたことでしょう。

「ライブコーディングです！」

「あ、ああ。そっちか！」

「ライブコーディングのことライブって略すなー」

「ウィキペディアのことウィキって言うなー」

「澪ちゃん、ウィキペディアってなに？」

本当に大丈夫なのかな。この部———。

「———今、本当にこの部にいて、大丈夫なのかな、と思ったね？」

「うわああ、唯先輩。エスパーですか！」

「近からずも遠からずや。これも SAT のちょっとした応用なんだよ。そんなあず SAT には、SAT の本当の恐ろしさを教えてしんぜよう」

「唐突ですね、先輩」

うむうむ、としかめっ面で頷く唯先輩がいつもとは違いなんだか面白くて、吹き出しそうになってしまいました。が、我慢しました。ついに、SAT について教えてもらえるのかと、少し緊張してきました。

「SAT については、知ってるよね？」

「はい、入部した時に聞かせてもらいましたけど———」

あれ以降はお茶会しかなかったから———というのは、ググっと飲み込んでおきました。

「大体は知っていると思うけど、改めて SAT、つまり充足可能性について説明しておくね。SAT ってのは、ある命題論理式が与えられた時に、それが充足可能かどうかを判定する問題だね。例えば———」

落書きだらけのホワイトボードに、唯先輩はそれ気にすることなく式を書き始めます。

- $(x_1 \vee x_2) \wedge (x_1 \vee \bar{x_2}) \wedge (\bar{x_1} \vee \bar{x_2})$
- $(x_1 \vee x_2) \wedge (\bar{x_1} \vee x_2) \wedge (x_1 \vee \bar{x_2}) \wedge (\bar{x_1} \vee \bar{x_2})$

「さて、あずにゃん、ここに 2 つの式があるじゃろう？ これらが充足可能かわかるかな？」
「それぞれの変数には、true か false のどちらかが入るんですよね。うーん、そうですね———」
1 つ目の式は、変数が x_1 と x_2 の 2 つだけ。x_1 を真（`true`）にすれば、1 つ目と 2 つ目の節が充足される。3 つ目の節は、x_2 を偽（`false`）にすれば、充足される。
2 つ目の式は、これも変数が x_1 と x_2 の 2 つだけだなあ。それに対して、節が 4 つで、そこに含まれる真偽の組み合わせの全パターン、これをすべて充足する組み合わせはない———。
「———1 つ目の式は充足可能、2 つ目の式は充足不可能ですね」
「———ファイナルアンサー？」
「え…？」
「ファイナルアンサー？」
「ふぁ、ファイナルアンサー」
真面目なのかふざけてるのかよくわからないな、この人は。
「———正解！！」
しかもネタが古いです。
「いやあ、ごめんごめん。一度やってみたかったんだよ、これ」
「もう、真面目にやってください！」
「———いいのかな？」
「何がです？」
「私が本気を出したら、どうなっても知らないよ？」
「大丈夫です、真面目にやってください」
「ちぇー、つまらないなあ」
頬をふくらませて、いじけて見せます。
「じゃあ、次。標準型の話ね。論理式には、積和標準形と呼ばれるものと、和積標準形と呼ばれるものがあってね、それぞれ、論理積のみからなる節の論理和、論理和のみからなる節の論理積、で———って口で言ってもわかりづらいね」
先輩は再びホワイトボードを走らせ、

- $(x_1 \wedge x_2) \vee (x_1 \wedge \bar{x_2}) \vee (\bar{x_1} \wedge \bar{x_2})$

「これが、積和標準型で」

- $(x_1 \vee x_2) \wedge (\bar{x_1} \vee x_2) \wedge (x_1 \vee \bar{x_2}) \wedge (\bar{x_1} \vee \bar{x_2})$

「こういうのが和積標準型」
ふむふむ、なるほど。
「さっきの問題の式は、和積標準型だったんですね」
「そうだね」
唯先輩は満足そうに頷き、
「標準型、というように、すべての論理式はこれらの標準型で表すことができるんだよ。あずにゃんは、ド・モルガンの定理は知っているよね？」
「はい、数学で習った気がします。確か、」
今度は私が、ペンをとって、

- $\neg(P \vee Q) = \neg P \wedge \neg Q$
- $\neg(P \wedge Q) = \neg P \vee \neg Q$

「こういうやつですよね」

「そう、それだよ」

唯先輩は、またも満足そうです。

「これと、二重否定の削除とかを組み合わせて、ゴリゴリ変形していくこともできるけど、最悪の場合だと、式のサイズが指数的に膨れ上がってしまうこともあるね。和積標準型への変換に限れば、元の式の線形サイズに抑える方法もあるけど――」

「ふむふむ」

「ちょっとややこしいから、それは置いといて。標準型で表せるってのは、真理値表を考えたほうがわかりやすいかなあ？ 0/1 の表の、1 になってるところの論理和を取れば積和標準型になるし、逆に――」

「0 になってるところの否定の論理積を取れば、和積標準型になるんですね！」

「そのとおり！ なかなか賢いじゃあないか、あず SAT 君！」

いつもと違って、変な口調だけど、ほめられて悪い気はしない。でも、標準型は何が嬉しいのだろう？

「標準型、特に和積標準型が重要なのは、これの各節を 3 変数に制限したものの充足可能性問題が、NP 完全問題に属するというのが大きいかな。特に 3SAT と呼んで区別しているね」

「ふむふむ、たった 3 つでそんなに解くのが難しくなるんですか」

直感的には、3 と 4 では全然違うのになあ。

「1、2、3、たくさん、って感じだね！」

唯先輩みたいですね！ といいそうになりましたが、

「2SAT は、効率的に解けるんですか？」

「そうだね、説明すると長くなるけど、2SAT はとても効率良く解くアルゴリズムが知られているよ」

ふむふむ、それも気になるけど、もうひとつ気になることが、

「3SAT が NP 完全問題ということは、すべてのクラス NP に属する問題は、3SAT で表すことができるってことですよね。じゃあ当然、4SAT も 3SAT に変換できることになりますね」

「なかなか鋭いね。ちょうどいい練習だから、考えてみるといいよ」

「うーん――」

しばらく考えたのですが、わからなかったので、宿題にしました。

「3SAT を 2SAT には、流石に出来ませんよね」

「それができたら、フィールズ賞ものだよ、あず SAT 君！」

そうだった！ ミレニアム懸賞問題を解くことになってしまいます。

「ということで、和積標準型が重要だということがわかったかな？ 和積標準型、長いからここからは CNF と呼ぶね」

「CNF って、何の略ですか？」

「うーん――忘れた」

あとで調べたところ、Conjunctive Normal Form の略のようでした。

「あまり細かいことに囚われていては、全体が見えなくなるのだよ」

唯先輩は、なぜか誇らしげです。

「そういうことがあって、CNF は計算量理論の世界では古くからよく研究されて来たんだけど、最近になって、別の方面でも CNF はとても重要になってきたのだよ」

「といいますと？」

「CNF の充足可能性問題を解くプログラム、いわゆる SAT ソルバと言われるものの性能がここ 10 年ほどで恐ろしいほど向上したんだ。最新の高性能ソルバだと、ものにはよるけど――」

両手を大きく広げるような仕草で、

「数万〜数十万変数の問題も解けるようになっているみたい」
「す、数十万？」
指数時間かかるアルゴリズムが、そんなに大きなサイズまで解けるなんて！ これじゃあ、二乗のアルゴリズムよりも大きなものが解けてしまいます。
「どうだ、驚いただろ？」
いつの間にか律先輩が横に来ていました。
「指数時間というのは、あくまで漸近的性能を示しているに過ぎないんだぞ」
「そうそう、私も律に教わるまでは、SAT がそんなに速く解けるとは思ってなかった」
澪先輩、律先輩に押されてけいさん部に入ったんだっけ。
「だけど、実際にそうなってみると、これはとんでもないことだなって———」
「あっ、あらゆる NP 問題は、3SAT に…！ まさか、P=NP なんていう必要がなかったとか？」
「『NP 完全』というのは、あくまで多節式時間還元できることしか言ってないから、SAT への効率のいいエンコードが常に存在するわけではないな。だけど、利用できる範囲が実際に狭いわけじゃない」
「なるほど———」
解きたい問題をエンコードしてみて、数十万変数に収まれば、解かれてしまうかもしれない。なんだかすごいことになってきました。
「でも、なんで最近になって急に性能が良くなったんでしょうか？」
良い質問だね、とでも言いたげに、唯先輩がもったいぶって話し始めます。
「21 世紀のはじめに、SAT ソルバにとって、大きなブレークスルーがあったんだよ。それまでは、DPLL ってアルゴリズムで、単一節、つまりひとつしか変数を含んでいない節の除去とか、二変数しか含まない節のうち、片方が確定したときの伝搬とか、わりとシンプルな枝借り探索しかやられていなかったんだけど」
それぐらいなら、私でも思いつくなあと思いました。
「ここで出てきた、CDCL というアルゴリズムがすごかったんだよ。Conflict-Driven Clause Learning っていうんだけどね。Conflict、つまり探索の途中で充足できない節が見つかった時、その原因を解析して、新しい節、つまり Clause を学習する」
「が、学習ですか———」
なぜかコンピュータが机に向かって勉強している姿を想像してしまいました。
「学習といっても、何かを教わって理解するとかそういうのではないね。機械学習的なものとは違って、Clause Learning は、発見といったほうが感覚的に近いかもしれないね」
「発見…ですか？」
「SAT の充足性の判定はある種の証明だと考えることができる。例えば、人間が何かを証明しようって時は、いきなりそれを証明するんじゃなくて、周辺の Lemma、つまり補助定理を積み上げて証明を組み上げていくものだけど、SAT の場合もその類推で考えるとわかりやすいよ。変数に仮定をおいて探索を進めていくときに、矛盾が起こった。SAT ソルバ君は、あれ？ なにがおかしかったんだろうって、必死で考えて、ああ、これがダメだったんだ、と理解する。じゃあ、これ以降の探索には、今獲得した矛盾の原因が起こらないように、元の節に追加するんだよ」
何か探索問題をエンコードして解かせた場合は、人が書いた場合の枝狩り条件に相当するものが、CNF の節として自動で獲得されるのかなあと思いました。
「Lemma を自動獲得すると表現する文献もあるぞ。どうだ、かっこいいだろう！」
そんなことができるなんて、賢すぎる。けど本当にそんなにうまくいくのかなあ？
「そんなに、うまくいくのかなあ？ と思っているんじゃないかな。あず SAT 君！」
「ひ、ひぇー。先輩———」

「そんなあず SAT 君には、ここの問題を全部 SAT にエンコードして、SAT ソルバに突っ込むという宿題をあげよう！」

そう言って、唯先輩は、なにか雑誌のようなものをドサッと置いて行きました。

「なんですかこれ———ナンプレ、カックロ、スリザーリンク、イラストロジック———こ、これ全部ですか！？」

「もちろん、今日は終わるまで帰さないよ、あずにゃん」

「そ、そんなあ」

と言いつつも、久々にけいさん部らしい活動ができて、少し嬉しい私なのでした。

---

6 月のジメジメした季節。結局あの時渡されたニコリのパズル雑誌は、経験の浅い私にはそんなにすぐに解けるはずもなく。それどころか、未だに半分以上残っていました。先輩たちが毎日お茶をしているのを横目に、私の気分まで、梅雨空のように曇るのでした。

「あーっ。めんどくさい！」

そう、めんどくさいのです。SAT ソルバは、その名の通り、命題論理式の充足可能問題しか解けません。変数は真か偽かの 1 ビットの情報しか載せられず、整数を扱うのにも、ビットを束ねて、加算器をつなげて、論理回路を一から書いているような状況です。さらに、その際にかなり工夫して書かないと、サイズが膨れ上がって、思うように大きいサイズの問題が解けなくなってしまいます。

「苦戦しているようだな」

見かねて澪先輩がやって来ました。

「はい、思うようにコードが書けなくて———」

「うーむ、なるほど———」

私のコードを覗いて、なにやら独り言を呟いて、

「———そろそろ、頃合いかな。梓、」

「は、はい！」

「お前に、SMT を教えよう」

え、えすえむ、ティー？ また変な名前の———。

「あずにゃん、そういうのじゃないよ」

「うわぁ、唯先輩。そういうの考えてないです！」

相変わらず、唯先輩には心を読まれているようです。

「スモール・ミディアム・トール、でもないわよ」

「ムギは飲み物の話ばかりだな」

SAT は、もう、たくさん。

「Satisfied Modulo Theories。簡単に言えば、SAT に別の理論を追加したものだ」

何かを追加する…？

「SAT は Boolean ロジックしか扱えない。だから、せっかく高性能になった SAT ソルバを利用しようと思っても、それを Boolean のコードにエンコードしなければいけない」

そう、それがめんどくさいんです。 SAT、めんどくさくて、たいへん。

「ただ、めんどうなだけならそこまでの問題ではないぞ。例えば、32 ビットの整数の足し算をするには、全加算器を 32 個つなげることになる。全加算器は、XOR を含む論理回路として表現できるが、XOR はそのままでは表せないから、これも CNF に変換すると、かなりサイズが大きくなってしまうんだ」

「足し算 1 つで、数百節になってしまうと、せっかく数十万規模の CNF を扱えても、実際に扱え

る整数問題のサイズは、かなり目減りしてしまうわね」

そう、それで私も困っていたところでした。

「まあ、中には CNF に加えて XOR 節を扱えるようにした、CryptoMiniSat みたいな処理系も存在するが、それは今回は置いておこう」

「CryptoMiniSat…暗号解析用のソルバなんですか？」

「いや、これは汎用の SAT ソルバで、特に暗号向けというわけではないぞ。暗号ルーチンに XOR が多く含まれるというのは無関係ではないだろうけど」

全加算器にも XOR は多く使われるから、整数の計算も高速化されそうだと思いました。

「多くの最新の SAT ソルバは、XOR のようなパターンは自動で認識して、特別な高速化が施されているから、今現在ではあまり考えることはないかな」

「へぇ、SAT ソルバってすごいんですね」

「まあ、それでも問題はあってだな。上限のない整数や、有理数、実数といったものは、そもそも SAT ソルバで扱うことは困難だ。加算のような処理も、予めどういう処理かわかっているから、かなり効率良く探索できるようになる。SAT ソルバも理想的にはそういうパターンを認識して自動で制約を獲得できてもおかしくはないが、現状ではそこまでうまくは動いてくれない」

理想と現実というのはどこにでもあるものだなあと思いました。

「で、SMT だ。SMT は SAT に Modulo Theory を追加したもの。具体的には――」

そこまで言うと、澪先輩はホワイトボードに向かって何やら書き始めました。

- Linear Integer/Real Arithmetic
- Non-linear Arithmetic
- Bit-Vector
- Uninterpreted Functions
- Arrays
- Quantifiers

「こういったものだ」

なにやら色々ありました。見覚えのあるものから、全く聞いたこともないものまで。

「1 つずつ説明していこう。 Linear Integer Arithmetic は、整数に関する線形の演算だな。整数の足し算とか定数との掛け算なんかがサポートされる。これを使うと、$x+2=3$ のような制約式が書けるようになるぞ。Differential logic と組み合わせて、 $x>3 \wedge x+x<9$ のような制約式を書くこともできる」

「私の今までの苦労は一体――」

「急がば回れというやつだよ、あずにゃん！」

「ううん、最初から教えてくださいよお」

「別に意地悪というわけではないぞ。SAT に慣れ親しむことは、SMT の理解にも大きく役立つんだ。だから最初のうちは SAT でだな――」

「とか言って、澪ちゃん SMT 知った時同じような反応してた」

「うるさい！」

澪先輩にもそういう時期があったんだなあ。

「それはともかく、Linear Arithmetic のサポートは表現力以外にも大きなものがある。線形の制約を解くには、線形計画法が使えるから、何らかの高速なアルゴリズムが実装されている処理系が多いぞ。今現在の最先端の処理系 “z3” では、Simplex 法が使われて、高速に、しかも解析解が求まるんだ」

そう言って、澪先輩は PC に向かって何か打ち込み始めた。

「簡単な線形整数充足問題、連立方程式でも解かせてみるかな。z3 には C++ の API や、Python のバインディングもあるが、私はもっぱら Haskell の sbv というパッケージを使っている」

ghci を起動して、手際よくコードが入力されていきます。

```
> sat $ do
    x <- sReal "x"
    y <- sReal "y"
    solve [ x + y .== 3
          , 2 * x + 5 * y .== 9
          ]
```

「よし、と」

ターン！ と勢いよくエンターキーがたたかれると、次のような出力が出てきました。

```
Satisfiable. Model:
  x = 2.0 :: SReal
  y = 1.0 :: SReal
```

「す、すごい」

わ、私の苦労はいったい何だったのかな。

「ここでは、SReal というのが変数の値だな。2.0 と出ているが、これは浮動小数ではなくて、厳密な値だ。その証拠に、こんなふうにすると――――」

```
> sat $ do
    x <- sReal "x"
    y <- sReal "y"
    solve [ 3 * x + y .== 3
          , 2 * x + 5 * y .== 9
          ]

Satisfiable. Model:
  x = 6 % 13 :: SReal
  y = 21 % 13 :: SReal
```

「とまあ、こんな具合だ」

「ええ！ これどうなってるんですか？」

「解析解が求まるときは、それを求めてくれるんだ」

「解がない場合はどうなるんでしょう？」

「そりゃあ、充足問題なんだから、当然」

```
> sat $ do
    x <- sReal "x"
    y <- sReal "y"
    solve [ 3 * x + y .== 3
          , 3 * x + y .== 9
          ]
```

```
Unsatisfiable
```

「こうなる」
「ははあ、なるほど。解が無限にある場合は」
「無限個は、うまく扱えないかな。少なくとも充足解をひとつ見つけるか、無限に見つけることはできるが———」
「無限個にあるということを示すことはできるぞ」
今度は律先輩がキーボードを奪い、なにやら書き始めました。

```
import Data.SBV

eq :: SReal -> SReal -> SBool
eq x y = bAnd
  [ 3 * x + y .== 3
  , 3 * x + y .== 3
  ]

hasInfSolution :: Predicate
hasInfSolution =
  forAll_ $ \x y -> forSome_ $ \x' y' ->
    (eq x y ==> eq x' y') &&& (x, y) .< (x', y')

main = print =<< prove hasInfSolution
```

「このプログラムは、ひとつの定理をしようとしている。eq って関数が、連立方程式だな。で、hasInfSolution ってのが、示したい性質だ。Haskell の DSL だから、細かい記法は気にせずに読んでくれ。すべての解に対して、それより小さい解が何かひとつは存在するって、要するにそういうことだな。最後に、それを証明（prove）して、結果を表示せよ！ と」
ぽちっとな！ とコードが実行されました。

```
$ runhaskell hoge.hs
Q.E.D.
```

「見事、Q.E.D. というわけだ。どうだ、かっこいいだろ？」
「す、すごい———！」
何がどうなってるのかは、さっぱりわからないけど、すごい———！
「どうなってるかは、わかってしまえば簡単だぞ。SMT も所詮は制約ソルバ。何か式を充足する変数があるかを調べることしかできない。でも、逆に言えば、証明問題をそういう問題に変換すればいいんだ。つまり何かを証明したいときは、それの反対を充足する解の不在を証明すればいい」
「ふむふむ、じゃあ、定理が間違っていたときは、もしかして」
「そう、反例（counter-example）が出てくるという寸法だ」
「至れり尽くせりとはこのことですね」
こんなことができてしまっていいんでしょうか。人類の脳が衰退しそうです。

「梓、まだ、これは序の口だぞ。z3 は、部分的だが、非線形実数問題に対する解析解を出すこともできる」
「え、私もうおなかいっぱい」
「簡単な例だと、二次方程式の解とかだな、ぽちぽち、と」

```
> sat $ \x -> x^2 - 3*x + 2 .== (0 :: SReal)
Satisfiable. Model:
  s0 = 2.0 :: SReal
```

「おお、出ました。でも、この方程式、もうひとつ解がありませんか？」
「そうだな。そういう時は、SMT ソルバの全解列挙機能を使えばいい。sat のところを allSat に変えて――――」

```
> allSat $ \x -> x^2 - 3*x + 2 .== (0 :: SReal)
Solution #1:
  s0 = 2.0 :: SReal
Solution #2:
  s0 = 1.0 :: SReal
Found 2 different solutions.
```

「これでよし」
「ちゃんと出ましたね。しかも、2 つの解があるって親切にも！」
「重解ももちろん出せるぞ」

```
> allSat $ \x -> x^2 - 2*x + 1 .== (0 :: SReal)
Solution #1:
  s0 = 1.0 :: SReal
This is the only solution.
```

「うへぁあ、すごい」
「これは実数の式だから、これをこうやると――――」

```
> allSat $ \x -> x^2 .== (-1 :: SReal)
No solutions found.
```

「完璧！」
「なんと」
「さらに……！」

```
> allSat $ \x -> x^2 - 5*x + 3 .== (0 :: SReal)
Solution #1:
  s0 = root(1, x^2-5x = -3) = 0.6972243622680053... :: SReal
Solution #2:
```

```
  s0 = root(2, x^2-5x = -3) = 4.3027756377319946... :: SReal
Found 2 different solutions.
```

「これまでの解もやはり解析解だったと」

「———」

もはや圧倒されて言葉になりませんでした。

「すごいです！ これは！」

「じゃあ、Arithmetic はこれぐらいにして、次にいくぞ。BitVector は、Bit を束ねたもので固定長の整数に対する演算をサポートする。具体的にいうと $GF(2^n)$ 上の加減乗除算になる。それに加えて、ビットレベルの論理演算もサポートされていて、いうなれば、計算機のプログラムの振る舞いを模倣できるものだな」

「それはどういうところで役に立つんですか？」

わざわざ制限されたビット数の演算を使うとうれしいのはなんだろうかと思いました。

「もともと、SAT ソルバの重要な応用として、モデル検査というのがあってだな。モデル検査というのは、長くなるから、ここでは割愛するけど、ソフトウェアとかハードウェアがある種の性質を満たしているかを、静的に検査するものだ。これに対して、Bit-Vector をサポートした SMT を使えば、より効率よく検査できるという、まあそういう話もある」

なるほど、よくわからない分野の話だ。

「ところで、あずにゃん、ビット演算は好きかな？」

「え、唐突ですね。あれですか？ ハッカーのたのしみとかの。あれ、私はあんまり得意じゃないかなあ」

「そうそう、それ。32 ビット整数の、立ってるビットを数えるコードってあるよね」

うーん、どんなんだったっけ。シフトして足して、そんなんだったかな？

```
popCountNaiive n =
  sum [ (n `shiftR` i) .&. 1 | i <- [0..31] ]
```

「まず、簡単なコードを書いてみるね。32 個のビットをそれぞれシフトして、最下位取り出して、足し合わせるだけの簡単なコードだよ」

「自明なコードですね」

「で、次に、凝ったコードを書く！」

```
popCountFast n0 =
  let n1 = (n0 .&. 0x55555555) + ((n0 `shiftR`  1) .&. 0x55555555)
      n2 = (n1 .&. 0x33333333) + ((n1 `shiftR`  2) .&. 0x33333333)
      n3 = (n2 .&. 0x0f0f0f0f) + ((n2 `shiftR`  4) .&. 0x0f0f0f0f)
      n4 = (n3 .&. 0x00ff00ff) + ((n3 `shiftR`  8) .&. 0x00ff00ff)
      n5 = (n4 .&. 0x0000ffff) + ((n4 `shiftR` 16) .&. 0x0000ffff)
  in n5
```

「わあ、これは正しいのかぱっと見てもわからないですね」

「そういう時は、普通は、テストとかを書くものだけど、SMT ソルバなら」

```
popCountIsCorrect x =
  popCountNaiive x .== popCountFast (x :: SWord32)
```

「定理が書けちゃうわけですね！」
「そのとおり！ あずにゃんも慣れてきたね。んでもって、」

```
> prove popCountIsCorrect
Q.E.D.
```

「Q.E.D. というわけだよ。あず SAT 君！！」
「すごいです！ これ、コードが間違ってた場合はどうなるんですか？」
「んー、じゃあ、ちょっとバグを入れてみようか」

```
popCountFast n0 =
  let n1 = (n0 .&. 0x55555555) + ((n0 `shiftR`  1) .&. 0x55555555)
      n2 = (n1 .&. 0x33333333) + ((n1 `shiftR`  2) .&. 0x33333333)
      n3 = (n2 .&. 0x0f0f0f0f) + ((n2 `shiftR`  5) .&. 0x0f0f0f0f)
      n4 = (n3 .&. 0x00ff00ff) + ((n3 `shiftR`  8) .&. 0x00ff00ff)
      n5 = (n4 .&. 0x0000ffff) + ((n4 `shiftR` 16) .&. 0x0000ffff)
  in n5
```

「どこが、変わったんでしょうか？」
「シフトの数を一箇所変えたんだけど、ぱっと見わかりづらいよね。でも、そんなときでも、証明があれば」

```
> prove popCountIsCorrect
Falsifiable. Counter-example:
  s0 = 526123024 :: SWord32
```

「ほ、ほんとに、反例つきで反証が――――！」
「まだまだ、ここから C のコードを生成することも可能なのだよ」
「えっ、C のコード――――！？」
「ちょちょいと、生成用のコードを書いてやって」

```
main = do
  compileToC Nothing "popCount" $ do
    x <- cgInput "input"
    cgReturn $ popCountFast (x :: SWord32)
```

「これを実行すると――――」
「ど、どうなるんですか！？」

```
/* File: "popCount.c". Automatically generated by SBV. Do not edit! */

#include <inttypes.h>
#include <stdint.h>
#include <stdbool.h>
#include "popCount.h"

SWord32 popCount(const SWord32 input)
{
  const SWord32 s0 = input;
  const SWord32 s2 = s0 & 0x55555555UL;
  const SWord32 s3 = s0 >> 1;
  const SWord32 s4 = 0x55555555UL & s3;
  const SWord32 s5 = s2 + s4;
  const SWord32 s7 = s5 & 0x33333333UL;
  const SWord32 s8 = s5 >> 2;
  const SWord32 s9 = 0x33333333UL & s8;
  const SWord32 s10 = s7 + s9;
  const SWord32 s12 = s10 & 0x0f0f0f0fUL;
  const SWord32 s13 = s10 >> 5;
  const SWord32 s14 = 0x0f0f0f0fUL & s13;
  const SWord32 s15 = s12 + s14;
  const SWord32 s17 = s15 & 0x00ff00ffUL;
  const SWord32 s18 = s15 >> 8;
  const SWord32 s19 = 0x00ff00ffUL & s18;
  const SWord32 s20 = s17 + s19;
  const SWord32 s22 = s20 & 0x0000ffffUL;
  const SWord32 s23 = s20 >> 16;
  const SWord32 s25 = s23 & 0x00ffffffUL;
  const SWord32 s26 = s22 + s25;

  return s26;
}
```

「C のコードが手に入るというわけだよ！ しかも、証明つきの！」

「なんと！」

証明つきのコードといえば、Coq とか Agda が有名だし、Coq も C のコードを生成することはできます。でも、これは...。

「ほかのコードを証明できる処理系の Coq とかに比べると、あまりにも簡単に証明できすぎですよこれ。なにかがおかしいです！」

「確かにそうだね、でも」

唯先輩はチッチッ、と楽しそうに片目をつぶって、

「SMT は、証明を探索をするプログラム。Coq は基本的には証明を自分で書くもの。SMT は簡単

に書ける代わりに、いつでも証明を見つけてくれるとは限らないのだよ」
「なるほど、うまくいくときにはうまくいくということですね」
うまくいくときにはうまくいく、とは何もいっていないような気もしたけど、実際にそういうものだから仕方がないとも思いました。
充足可能問題というところから、気づけば線形計画法、果てはプログラムの静的検証、自動生成までやってくるとは、最初からは想像もしていませんでした。
「なんか、今日はすごく疲れました」
「なんだ、梓。この程度で音を上げてちゃあ、けいさん部の次期部長の座は譲れないなあ。一番おもしろいのは、Uninterpreted Function なんだぞ。ここから free-object につながって、圏論の世界とプログラムの世界が———もごもご」
そうでした。新入部員は私一人だから、いずれはそうなるんです。
「さて、じゃあ SMT を覚えた梓には、宿題を出しておこう」
澪先輩がいつぞやのように、どさりと雑誌を置きました。
「ま、またニコリですか〜？」
「そうだ。明日までに全部解いて来るように！」
「そ、そんな〜」
当時の私にとっては、つらく厳しい道のりでした。しかし、SMT というものがもたらす可能性について、なにやら底知れぬものを感じていた私は、拒むこともできなくなっていたのです。
「———ヤッテヤルデス！」
———
梅雨が晴れ、雨雲が去るのと同期するように、私の頭にかかった霞も徐々に晴れてきました。SMT もだんだんうまく扱えるようになってきたところで、ふと思って、改めて SAT に立ち返ってみることにしました。
「うーん、充足可能問題には、すべて充足できるか、そうでないかを求める以外にもいろいろあるのか〜」
なるべくたくさんの節を充足させる、MaxSAT 問題、節に対して重みがついていて、充足する節の重みの和を最大化する WeightedMaxSAT 問題、節のうち、必ず充足しなければいけない節が指定されているとき、それらを充足しながら、残りの充足節を最大化する PartialMaxSAT 問題、変数の値を true/false ではなく、n 種の値に一般化した CSP———。
これらの問題のソルバは、単純な SAT のものに比べると、効率はよくないようです。
「それ、SMT でできるよ」
「唯先輩！？ い、いつの間に部室に」
「あはは、ぴくってしちゃって。あずにゃんはかわいいなあ。すりすり」
「もう、やめてくださいよお」
口ではそういいながらも、唯先輩のこのスキンシップが、私はだんだんまんざらでもなくなってきていました。
「おやおや、口ではそういってはいるが、こっちのほうは正直だのう」
「変態的な発言やめてください！ こっちってどっちですか」
「それは、そっちに決まっておるじゃないか。ぐへ、ぐへへ」
ガチャリ。
「おー、唯と梓はいつも仲がいいな」
「澪先輩、助けてくださいー」
澪先輩は、いつものように PC を起動しています。私も、すっかりとけいさん部に馴染んできたものです。

「そうだ、SMT でできるっていうのは」
「あ、そうそう。あずにゃんが話の腰を折るから」
「えっ、私ですか？」
だめだ。唯先輩のペースに引き込まれては。
「あれー、よく覚えていないや。まあ、それはおいておいて」
「はい」
「あずにゃんは、MaxSAT に興味があるのかな？」
「ええ、なんというか、今調べていたらそういうものがあるって知りまして」
「なるほど、なるほど」
唯先輩は、眉間にしわを寄せながら何かを真剣に考えているふうですが、この人の場合は、本当は何も考えてないんじゃないかとか思ったりもするのです。
「うーむ、それならやっぱり SMT でもできるよ」
「SMT ですか」
「SMT だよ。SMT があれば（大体）なんでもできる！ S・M・T・だー！！」
ほんまかいな、と突っ込みをいれずには入れないこの台詞。
「MaxSAT を SMT にうまくエンコードできるんですか？」
「そうだね。割と簡単だと思う」
ちょっとまってね、と言いながら、ホワイトボードに向かって、なにやら式を書き始めました。
「もとの CNF がこれで――――」

- $c_1 \wedge c_2 \wedge c_3 ...$

「それが、こうなって」

- $(w_1 \rightarrow c_1) \wedge (w_2 \rightarrow c_2) \wedge (w_3 \rightarrow c_3) ...$

「最後に、この節を追加すれば」

- $(score \iff w_1 + w_2 + w_3 + ...)$

「よし！ いっちょあがり！」
「先輩、これは一体」
「あ、説明するね。まず、一つ目の式が、元の MaxSAT に渡す CNF で、この式の充足節を最大化したい。そのために、各節が充足したかどうかのフラグを立てる。それが二つ目の式だね。最後に、立っているフラグの数が score と等しいですよ、という節を追加する。これが三つ目の式だね」
「これの score を最大化するんですか？ でも、SMT は充足可能問題しか解けないですよね」
「なら、充足可能問題にしてやればいいのだよ。score を最大化することはできないけど――――」
「あっ、score がある一定以上で充足可能か、という問題なら解けますね」
「その通り。それが解けるなら、後は簡単だね」
「二分探索で、充足可能な限界を調べてやるんですね！」
「よくできました！ ご褒美にあずにゃんには、この猫耳を――――」
「いりません！」
猫耳はまだちょっと恥ずかしいのでした。
「じゃあ、あずにゃんも成長したことだし、新しいあだ名を進呈しよう！ うーん、こういうのはどうかな。あず MaxSAT！」
「どこかの芸人みたいじゃないですか！ いりません」
なんで私ばかり、変なあだ名をつけられるんだろう。

「ちぇー」

あからさまにいじけた顔をしている先輩もなんだか最近はかわいらしく思えてきました。

「ともかく、これで MaxSAT が、SMT ソルバを使って解けるようになったわけですね。しかも WeightedMaxSAT や PartialMaxSAT にも簡単に応用できそうです」

「ふふふ、これも SMT の簡単な応用だよ」

「CSP への応用は、さらに簡単そうですね」

「CSP の SMT へのエンコードは、あず MaxSAT 君への宿題にしておこうじゃないか！」

「その呼び方やめてください」

ここでふと気になりました。

「あれ、SMT を使って、MaxSAT をエンコードするとき、さらりと最大化問題を解いていませんでしたか？」

「あれ、そうだね」

「SMT を使うと、最適化問題が解けてしまうんでしょうか」

「うーん、そうだなあ。一般にというわけではないと思うけど」

唯先輩は頭を一ひねりして、

「大体の場合は解けると思うよ。まず、目的関数、つまり最適化したい値が、二分探索可能な値なら、制約を入れて二分探索すればいい。これがたぶん一番効率もいいはずだよ」

「さっきのエンコード方法ですね」

「二つ目の方法としては、充足解を求めながら、徐々に良い解を求めるというものかな。まずひとつ充足解を求めて、その解よりも良い、という条件を加えて、再度充足解求める」

「これは二分探索と比べて良くないんですか？」

「たぶん、何かひとつ解を求めるということに関しては悪くないと思うけど、二分探索とは違って、イテレーションごとに指数的に答えに近づくわけじゃないから」

「牛歩になる可能性が高いと」

「まあ、そうだね」

「それで、最後三つ目の方法。量化子を用いる方法」

「量化子？」

聞いたことはあるような気がしましたが、SAT に持ち込んで良いものだったのかな。

「Quantifier といってね、SMT ソルバの中には、これが使えるものもあるよ。z3 ではサポートされているね」

「SMT 何でもありですね———」

「そこが SMT のいいところなんだよ！」

なぜか唯先輩が胸を張って自慢しています。

「で、Quantifier ってのは、∀とか、∃とかそういうやつだね。最大化したい関数が $p(x)$ なら、最大値を求める制約式は、

$$\exists x_{\max}.\forall x'.p(x_{\max}) \geq p(x')$$

になる」

「なるほど、これなら $x_{\max}$ にアサインされた値を取り出せば、一発で最適解が求まりますね。———でも本当にこんな式を汎用のソルバが探索で解を導き出せるんでしょうか？」

「難しい質問だね。現状では、できることもあるし、できないこともある。できたとしても、明らかに遅いときもある。組み合わせる Theory との相性もあって、 Linear arithmetic と組み合わせた場合はうまく動くことが多いけど、非線形の式と組み合わせた場合は、ギブアップしちゃうことが多

いかなあ」
「やっぱり、万能とはいきませんね」
こんなのができてしまったら、本当に人類の脳の衰退の危機です。あぶない、あぶない。
「まあでも、その辺はソルバの今後の発展に期待というところだよ」
「ところで、さっきの MaxSAT だけど」
PC に向かって、なにやらプログラムを書いていた澪先輩が、驚いた表情でなにやら画面を見つめています。
「専用の MaxSAT ソルバよりも、SMT にエンコードして、二分探索したほうが速い」
SMT にエンコードして、ソルバで殴ればいい。ゲームだとクソゲーだけど、現実だと、これほどありがたいことはないのでした。

---

8 月、夏。世間では甲子園が盛り上がっているころ、けいさん部でもひとつのイベントがありました。
「プログラミングコンテスト？」
どうやら、部室に飾られたたくさんの表彰状やらなにやらのひとつは、そのコンテストでのものみたいでした。
「ふふふ、我がけいさん部は、毎年優勝しているのだよ」
「毎年って言っても、去年一回しか出てないけどな」
「一回でて一回なんだから、毎年で良いんだよ」
「かなり語弊があるがな」
へえ、先輩たち、ずっと遊んでいるように見えて、やるときはやる人たちなんだな。
「あずにゃん今、お茶しか飲んでないのに、なんで勝てるんだって、思ったでしょ？」
「ぎ、ぎくぅ。そんなことないですよお」
し、心臓に悪いなあ。
「まあ、実際そうだぜ。私たちがやってるのは、半ばゲームバランスを崩壊させるようなことだからなあ」
「実際そうだから、仕方ないよね」
「何ですか、そのもったいぶった言い方」
「あら、梓ちゃんにはわかると思ったけど」
まさか、先輩方、SMT を…？
「い、いいんですか？ そんなの。徒競走に車で出るようなものじゃないですか」
「禁止されてなければ良いんじゃないか？ ルールはちゃんと読んでるぜ」
平然と言ってのける。
「いいか、梓。現実の問題にルールなんてないんだぞ。これはコンテストだからそういう疑問も出るけど、実際の問題なんて解くためだったら何をしても良いんだ。そのために、人類の英知の結晶を使うのは、いたって当たり前のことだろう？」
「そ、それはそうですけど」
「私はな、ずいぶん昔からプログラムを書いているけど、ある日気付いちまったんだ。なんで私は同じようなプログラムを何回も書いているんだろう。なんで同じようなアルゴリズムを何回も書いているんだろう。なんで同じようなプログラムで同じようなバグを作っているんだろう。なんで同じようなバグを取るのに、同じような苦労を何回もしているのだろう。そして、私はなんで、それに何の疑問も抱いてこなかったのかって」
私は、プログラミング暦が浅いから、まだ、毎日がわからないことだらけで、毎日が新鮮で、そ

んなこと考えもしなかったけれど。

「たくさんの同じようなファイルを手作業で作る人を馬鹿にする。たくさんの同じような Excel ファイルの集計を手作業でやる人を馬鹿にする。それがプログラマってもんだ。だけどな、なぜか鏡を見れないヤツが多いんだ。そこに映る、たくさんの同じようなプログラムを書いている自分の姿を」

メタな視点。あるものを別の場所から観測するのは、当たり前で、普通のことだけど、その中にいる人が、自分自身を観測するのは、普通のように見えて、普通のことじゃない。

「たくさんの単純作業を自動化して、その仕事を奪ってきたのはプログラマ。ならば、その仕事は同じくプログラマに奪われるべきだよなあ」

はじめに 1 があった。 1 はその世界のすべて。次に、2 が生まれた。2 には、1 を見ることができたけど、自分がどういう存在かを自覚することはできなかった。そして、次に、1 から 2 を作るのと同じようにして、 2 から 3 が作られた。3 はその事実と、2 と 1 を見比べることにより、自分がどうやって作られたのかを初めて知る。

「1、2、３、たくさん」

「梓、何だいそれは」

「唯先輩が言ってた」

たしか、3SAT が NP 完全だって話だった。3 まで数えられたら、それより上は、自動的に数えられる。

「3 って数字は何か特別なものを感じます。higher-order が有効なのも多くの場合 3 まで」

これは、帰納的な何かと関係あるのか。 4 を数えることに疑問を持った人たち。4 を数えることを拒否した人たち。けいさん部。それは人類の衰退などではなく、約束された恒久的な発展だったのだろうか。

「だから、私は SMT に希望を見出したんだ」

SMT ソルバは破壊的技術だ。十徳ナイフだ。多くの場合それでことが足りてしまうようになる。当然、個々の問題には専用のソルバのほうが良い性能を示す場合もあるだろう。しかし———。

「特殊なソルバがコスト的に見合うラインは、今後確実に上がっていくだろう。なぜならば、SMT ソルバの性能が上がれば、あらゆる問題が高速化されてしまうからだ。今の SMT で難しいところがあれば、SMT にコミットすればいい。新しい Theory を追加すればいい。制約ルールを追加すればいい」

そうすれば、それは人類の永遠の財産になるのだから。

かつて、いろいろなものが通った道。マシン語のコードを効率良く書くにはどうすれば良いか。マシン語のコードを書かなければ良い。コンパイラを使わない積極的な理由のある分野は、いまや、ほとんど、ない。

最適化問題を書きたい。効率の良いアルゴリズムを実装したい。バグのないプログラムを書きたい。早く書き上げたい。どうすれば良いか。答えは簡単だ。プログラムを書かなければいい！

「そうか、そうだったんですね！ 先輩は、この、けいさん部は———！」

お茶ばかりしている理由、 SAT をいきなり私に教え込んだ理由、お茶ばかりしているのに、プログラムが書ける理由。

「ふふ、やっと気付いたみたいだね」

「あずにゃん、おめでとう」

「思ったより早かったな」

「じゃあ、改めて梓ちゃん」

「『けい３部』にようこそ！」

「———けい３部の部室って、こちらでしょうか？」
四月の陽光の中、うたた寝から目覚めると、見知った顔が二つ。
「にゃー、あ、憂、純、おはよう」
「あんたがどうしてもって言うから、来てあげたわよ」
「梓ちゃん顔に線ついてる」
そうだ。私は、先輩達の意思を継いで、このけい３部の部長になったんだ。
「ところで、この部って何やる部なの？」
そうだなあ———。まずは。
「free object って、知ってる？」

けんろん！ へ続く

会員名簿じゃないカ?

@xhl_kogitsune (1, 4 章)

こぎつねさんかわいいです。

@master_q (2 章)

ツィッターアカウントが DM スパマー乗っ取られたので、主アカウントを@masterq_hentai に引っ越ししました。

@nushio (3, 5 章)

ハスケル教ギーメル・ヘー・コフ派万歳！ 中二じゃないから恥ずかしくないもん！ 君とならまた飛べるよ！

@tanakh (6 章)

またあずにゃんなんだ。すまない。

.S(S(KS)K)(S(S(KS)K)(S(S(KS)K)(S(K(M(S(S(KS)K)I)))(S(S(KS)K)(S(S(KS)K)I(M(S(S(KS)K)I)))))).S(S(KS)K)(S(K(M(S(S(KS)K)I)))(S(S(KS)K)(S(S(KS)K)I(M(S(S(KS)K)I)))))(S(S(KS)K)(S(S(KS)K)I(S(S(KS)K)(M(S(S(KS)K)I)))))

「簡約！ λカ娘 (4)」

2012 年 12 月 31 日 コミックマーケット 83 初版発行
2013 年 1 月 6 日 第 2 版発行
2013 年 7 月 7 日 第 3 版発行

原作：安部真弘「侵略！ イカ娘」より
発行：参照透明な海を守る会
http://www.paraiso-lang.org/ikmsm/
hayashizaki.kitsune+ikmsm@gmail.com
著者：@xhl_kogitsune, @master_q,
@nushio, @tanakh
表紙：七臣ナオミ 様 http://naomi703703703.blog112.fc2.com/
印刷：丸正インキ有限会社 様 http://www.marusho-ink.com/

(S(KS)K)(S(S(KS)K)I)))(S(S(KS)K)(S(S(KS)K)I(S(S(KS)K)(M(S(S(KS)K)I)))))).S(S(KS)K)(S(K(M(S(S(KS)K)I)))(S(S(KS)K)(S(S(KS)K)I(M(S(S(KS)K)I)))))(S(S(KS)K)(S(S(KS)K)I(S(S(KS)K)(M(S(S(KS)K)I)))))

(K(SKS)S)(K(SKS)S)S·((((I(K(SKS)S)S)(K(SKS)S)S)(I(K(SKS)S)S)W)(K(SKS)S)S)(K(SKS)S)S·(((((I

参照透明な海を守る会