目次

第 0 章 めたせぴ☆ふぁうんでーしょん `@master_q` **1**

0.1 はじまり 1
0.2 レストランにて 2
0.3 砂の中 4
0.4 最初のスケッチ: POSIX からの脱出 6
0.5 冷たい壁 12
0.6 二度目のスケッチ: 省メモリ化の追求 13
0.7 あこがれ 23
0.8 三度目のスケッチ: コンテキストを操る 26
0.9 突然の連絡 41
0.10 これからのこと 41
0.11 参考文献 42

第0章

めたせぴ☆ふぁうんでーしょん

— @master_q

0.1 はじまり

西暦 2013 年。凶の年。ソフトウェアエンジニアの多くは Web の世界に住んでいた。じゃゔぁすくりぷと、あんどろいどじゃゔぁ、おぶじぇくとしー、しーしゃーぷ。高機能な言語と環境をあやつり、彼等は栄華をきわめていた。きらびやかな UI、きらびやかな通信。またある人々はしーげんごとしーぷらすぷらすをあやつり、匠の技で複雑なハードウェアをあやつり、複雑多機能なうぇぶぶらうざを作り、そしてそれらを支えるオペレーティングシステムりなっくすを作っていた。

しかしぼくらは幸か不幸かそのどちらでもなかった。オープンソースのソフトウェアを使って組み込みデバイスを作っていたのだ。世間からは考えられないほど古いバージョンのオープンソースを使い、世間からは考えられないほど汚れたソースコードを弄り、そして世間からは考えらない奇妙なデバイスの集合を組み合わせていた。それだけならまだよいのだが悲しいことに作った製品への世間からの評価は低かった。ぼくらの中にはこの製品化の渦に飲み込まれて息絶える者もいた。

そうして生き延びたぼくたちは現実から逃避するようにインドネシアに逃げのびた。もう設計などしたくはなかったのだ。自然豊かなこの国で、ぼくたちは安らぎ、ソフトウェアのことなど忘れかけていた。それでも組み込みシステムへの想いは捨てきれず、何か改善する手はないかと失われたテクノロジ **型システム**を学び会得していた。しかし祖国を離れてまともな仕事がある訳もなく、先進国向けの Web サービスを作るアルバイトで食い繋いでいた。もはや組み込みの民はこのまま消えゆくしかないのか……そう思いかけたとき……

「ワシが **めたせぴ☆** でゲソ!」

どこからともなく静かだが力強い声が響いた。

「これからおぬし達は一本の糸をたどることになるでゲソ。この糸は古えのテクノロジー型システムによって綿密に計算/予測された運命なのでゲソ!」

なにかわからないが、安心とやはり不安がいりまじって混乱する。この娘はいったい何を言っているのだ？

「今おぬし達を囲んでいる状況はよくわかっているでゲソ。増えるカスタムシステムコール。多種のデバイス。崩壊しつつある旧来の CPU アーキティクチャ。POSIX API 上に構築された複雑なミドルウェア。絶え間ない製品リリース。猛進進化する型付けされていないオープンソースソフトウェアへの追従」

この娘は何者だろう？ しかし、そうだ。その通りだ……

「そしておぬし達は今、型システムを手にしている。それがおぬし達だけが持つ特性、もしくは力じゃなイカ？ それならこれから取るべき道はあまりにもわかりきっているでゲソ!!!」

そう言うと、めたせぴ☆と名乗る娘は消えてしまった。……あれは夢だったのか？ 今となってはぼくたちにもわからない。

0.2 レストランにて

ぼくたちはちょっと贅沢をして海辺のレストランに来ていた。いつも作っている Web サービスではない別の話をしたくなったのだ。さっきの娘にそそのかされたわけじゃない、ほんの気晴らし。それでもぼくたちの話題はいつも組み込みシステムに向いた。あの頃ぼくたちを苦しめていた問題の根っこは何だったのだろう？ 製品開発において最も重要なのは不具合の解消だ。不具合の解消方法には根本対策と暫定対策がある。根本的に対策できれば 100 点満点だが、工数の関係上たいていの対策は暫定策に終わる。暫定対策を施しても似た不具合が将来必ず起きることになる。不具合の根本原因を放置しておくと長い時間をかけて毒が体全体にまわり、そしてプロジェクトそのものが死ぬのだ。ぼくらは先輩からこの法則を何度も教えれられ、そして痛いほど体で味わっていた。だからぼくらが集まるといつも “組み込みシステムの持つ不具合の根本対策” に話題が向くのだった。

レストランでの話し合いは長時間にわたり、そしてぼくらは誰しもが落ちつく結論に辿りついた。
「やはり最も大きな不具合は実行時エラーの増加なんだ」
そうこの結論には誰だって辿りつく。実行時エラーを減らすために色々な手法が提案されているのがその証拠だ。UML による設計もそうだろう。モデル駆動型アーキテクチャもそうだろう。契約プログラミングもそうだろう。でもぼくたちがいた大規模組込開発のドメインで、それらが特効薬になったという話は聞いたことがなかった。
「努力目標は機能しない」
だれかがそう言った。

ぼくらは型推論を知っていた。具体的な理論はよくわからないが、実行時エラーの一部を魔法のようにコンパイル時に知ることができた。ぼくらはこの特性が気に入って、日々の Web サービスの開発にも使っていた。
「どうして kernel ドメインで型による設計が使えないんだろう？」
またれかがボソっとつぶやいた。ぼくらは UNIX ライク kernel をカスタマイズ/メンテするチームだった。その kernel を使うプロジェクトは 10 年以上も続いていた。(国を逃がれた今、そのプロジェクトがどうなったのかはわからない)
「型で kernel を書くプロジェクトはたくさんある [*1] みたいだよ。 OCaml で kernel を書くとか手堅い選択肢かもしれないよ？」
「うん、ぼくもソース読んでみたさ。でもこれはオモチャ kernel だよ。割り込みはポーリングで拾っているしバスドライバさえない。移植性は皆無だ。この kernel を拡張していっても、かつてぼくらがいたドメインで使えるようにはならないよ」
ちょっとビンタン [*2] を飲みすぎたみたいだ。ぼんやりする頭を海からの風がすぎる。ぼくらのレストランでの会合はいつもこんな感じだ。この国はいい。こうやってぼんやりと昔話をして過ぎる時間。もういいじゃないか。そんなばかでかい問題。もう、ぼくらの知ったことじゃないよ。

でも今日はいつもと少し違っていた。
「いきなりスクラッチから書くのが無理なんじゃないか？」
いつもの話題がもう一つ先に進んだのだ。
「UNIX ライク kernel はもう世界に浸透しきってしまった。いまさら新しいインターフェイスの kernel がすぐに受け入れられるとは思えない。それなら型による kernel 設計も自然に UNIX ライク kernel を目指すべきなんじゃないかな」
正直ぼくらは POSIX インターフェイスにもうお腹いっぱい [*3] だった。しかし kernel は所詮ただの

[*1] “Haskell/OCaml 製の OS って何があるんでゲソ？” http://metasepi.org/posts/2012-08-18-haskell-or-ocaml-os.html
[*2] インドネシア定番のビール http://www.multibintang.co.id/
[*3] 参考: “Copilot への希望と絶望の相転移” http://www.paraiso-lang.org/ikmsm/books/c81.html

kernel。使ってくれる人がいなければゴミ同然だ。
「kernel の品質維持のためにドッグフード [*4] が有効なことはもう 20 世紀に結論が出ている。ドッグフードを行なうにしても POSIX インターフェイスがなければ今使っているプログラムのほとんどが使えない」
ドッグフードというのは “自分達の開発したソフトウェアを使って自分達のソフトウェアを作る” という意味だ。この開発手法は単純で自社製品の品質維持に効果的だ。というのも品質が維持できなければ開発行為自体に大きく影響するからだ。ドッグフード開発されていないために品質が悪化する例としては、社内部門が作った使いにくい社内ツールがイメージしやすい。社内部門が自分達が使わないソフトウェアを開発して、それ以外の全社員が使うことがある。このような会社はソフトウェアに対する理解が浅いと言えるかもしれない。

「試しに POSIX インターフェイスでない独自 API の kernel を作ってドッグフード可能になるまでの道のりを考えてみようよ」
「そうだなぁ、ドッグフード可能にするためには kernel をコンパイルするコンパイラとその他日々使う最低限のソフトウェア全部を作り込まなければならないはめになるよ (図 1)。でも、もし POSIX インターフェイスをまずは採用すれば既存の GCC コンパイラや Firefox のような Web ブラウザも比較的容易に移植できる。だから新しい kernel を作り込むことに集中できると思う。もしその kernel が安定動作するようになれば、POSIX ではない新しいインターフェイスをドッグフードの開始後にゆっくり作り込むことに後から挑戦することだってできるじゃないか」

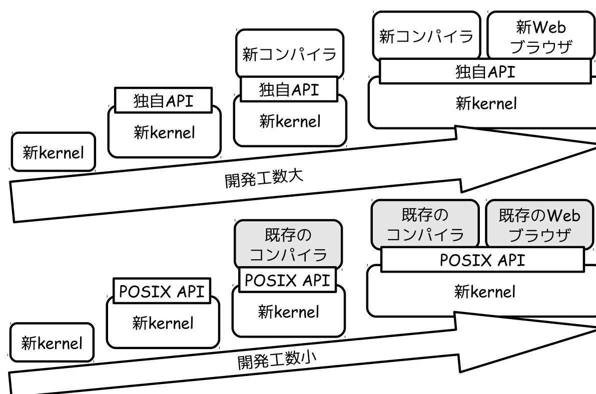

図 1: POSIX API を採用することで OS 開発の工数を削減

「それなら **スナッチ** [*5]すればいいんじゃない？」
耳慣れない用語だ。
「アイデアレベルだけど、既存の C 言語で設計された kernel を関数単位かモジュール単位で強い型を持つ言語で置換すればいいんじゃないかな」
砂の上に図を描きはじめた (図 2)。ぼくらの中には一人だけいつも非現実的なアイデアばかりを主張するやつがいる。

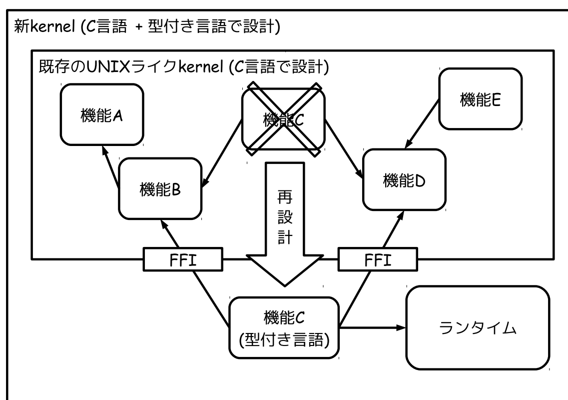

図 2: スナッチ設計: 既存 kernel を少しずつ設計置換

[*4] 闘うプログラマー http://www.amazon.co.jp/dp/4822247570
[*5] コナミのゲームであるスナッチャー http://en.wikipedia.org/wiki/Snatcher から

「そんな簡単に言うけど可能なの
かな？ この図のモデル……ああ、めんどうだから以後**スナッチ設計** と呼ぼう、このスナッチ設計を実現するためには “相応のコンパイラ” と “POSIX インターフェイスがなくても動くランタイム” が必要になるけれど、そんなコンパイラ実在するのかい？ OCaml を使うにしてもランタイムをずっとスリムにしなきゃいけない。それから割り込みベースで動作している UNIX ライク kernel をスナッチするのであれば、割り込みコンテキストを扱う方法も考えないと……」
また壁だ。いつもこの話題はどこかで大きな壁にはばまれる。
「ごめん、ちょっと酔ったみたいだ。散歩に行ってくるよ」
ぼくは一足早めにレストランを出ることにした。

0.3　砂の中

家に帰らずにぼくはまだ暗い海岸にいた。あれ？ 暗闇にさっきの娘が見えるような。名前は……なんだっけ。
「めたせぴ☆でゲソ! いいかげん覚えてほしいでゲソ」
元気の良い声が場違いに響き渡った。
「なにを悩んでいるんでゲソ？ さぁワシに話してみるが良いでゲソ」
ニヤニヤ笑いが癪に障る。
「おぬしの思っていることは全部お見通しでゲソ! どうやら自分の使いやすいコンパイラがどっかに落ちてないか砂の中を探しているようじゃなイカ？ すでに小さなバイナリを吐く型推論を持つコンパイラに出会っているのに、おぬしはそのコンパイラが最適解なのか悩んでいるようでゲソ。しかし最適解を探しているだけではいたずらに時間が過ぎるだけということも理解しているはずじゃなイカ？」
いや心配しているのはそこじゃないんだ。
「曳光弾とプロトタイプ [*6] でゲソ」
うん？
「自分が作っているものが製品化できるのか、それとも単に実証実験にのみ使っていずれ捨てるのか、作る前にあらかじめその決断をしておけば、どんなコードも無駄にはならないんでゲソ」
プロトタイプはわかるんだけど……曳光弾ってなんだろう？
「おぬしはあまり本を読まないようでゲソ。曳光弾は “射撃後飛んでいく間に発光することで軌跡がわかるようになっている弾丸” のことでゲソ。おぬしは今、暗闇のなかにいるんじゃなイカ？ 進む方角さえわからないでゲソ。もっと言えば本当に解が存在するかさえ不安なんじゃなイカ？ そんな時、おぬしが取り得る選択肢は二つしかないのでゲソ。プロトタイプは要求から製品に近いものをでっちあげる作業でゲソ。当然品質は劣悪なものになるはずでゲソ。これでは最終的な製品にはならない、けれど方角はわかるかもしれないでゲソ。曳光弾はプロトタイプとは違い、しっかりとした品質のソフトウェア部品を作る行為でゲソ。この曳光弾は運がよければ製品に搭載できるかもしれないでゲソ。品質が確保できているからじゃなイカ。ただしもちろん運わるく製品の方向性とは見当違いの部品かもしれないでゲソね。それでも曳光弾を作ることによって、その近傍にあるモノが暗闇の中で少しだけ見えるはずでゲソ。それもまた前進と言えるんじゃなイカ？」
ややこしいな……えっとこの場合は？
「かんたんじゃなイカ。コンパイラが曳光弾。その他のコード全てはプロトタイプにすぎないんでゲソ」
よく言いきれるな……

[*6] 達人プログラマー http://www.amazon.co.jp/dp/4894712741

「固く考えることはないでゲソ。曳光弾とは言っても好きなコンパイラをベースにすればいいんでゲソ。どーせどのコンパイラを選んだにしても不足した機能を追加しなければならないことには変わりがないのでゲソ。この間おぬしはコンパイラを物色していたんじゃなイカ？」

そう、あの時は jhc [*7] が組み込み開発にとても良い特性を持っていた。コンパイラが組み込みに向いているかどうかは三つの指標を比較すればだいたい見当がつく。

A. バイナリサイズ
B. 未定義シンボル数
C. 依存ライブラリ数

A に関しては自明だ。組み込み開発では使えるメモリ領域が限られていることがある。もちろんスワップなど使えない。プログラムのサイズは小さいにこしたことはないのだ。C に関しても比較的理解しやすい。プログラムが別のライブラリに依存している場合には POSIX API のない世界にそのライブラリもろとも移植しなくてはならなくなる。それでももしかするとプログラム自体を変更すればライブラリ依存を削除できる可能性はある。B は少しわかりにくい。プログラムバイナリがあるライブラリへの未定義シンボルを多く持つということは、そのライブラリへの依存度が高いということだ。ということはプログラム本体へ修正をほどこしても、その依存度が強いライブラリへの癒着はほどけない可能性が高くなる。つまり A,B,C の値それぞれが小さければ小さいほど POSIX の外の世界への移植度が高いことになるのだった。

いくつかの型推論を持つコンパイラについて、三つの指標の値を調査して表にまとめてみよう。

コンパイラ	バイナリサイズ (byte)	未定義シンボル数	依存ライブラリ数
GHC-7.4.1	797228	144	9
SML#-1.2.0	813460	134	7
OCaml-4.00.1	183348	84	5
MLton-20100608	170061	71	5
jhc-0.8.0	21248	20	3

表 1: “hoge” と印字するプログラムに見るコンパイラの特性

この表を見るかぎり jhc は小さなバイナリを吐き、さらには POSIX インターフェイスへの依存度も低い。そのため kernel ドメインにランタイムを移植する工数も少なく済みそうだ。ここまではわかっている。

「ベースにするコンパイラは決まったようでゲソね。あとはプロトタイピングを繰り返して、コンパイラに不足している機能を燻り出せばいいんじゃなイカ」

そうはいっても適切な題材が思い付かない……

「絵を描く前には白いキャンバスにスケッチをするものでゲソ」

そう言うとまたあの娘は消えてしまった。

[*7] Jhc Haskell Compiler `http://repetae.net/computer/jhc/`

0.4 最初のスケッチ: POSIX からの脱出

POSIX 上で動いているプログラムと POSIX の外で動くプログラムとの違いとはなんなのだろう (図 3)。

POSIX 上で動作する Haskell プログラムの内部は大きく二つの部分にわかれている。Haskell コード由来の部分とランタイム由来の部分だ。Haskell コードはどのような外部 API にも依存していないハズだ。FFI などを使わなければだけれど。Haskell コードはランタイムのみに依存している。さらにそのランタイムは外部のライブラリに依存しているはずだ。libc だけか、libpthread もか、とにかくなにかのライブラリ依存があるはずだ。そうでなれば副作用を実現できないからだ。そのライブラリ群は最終的には POSIX API にいきつく。その API を実現しているのは libc と kernel のセットだ。

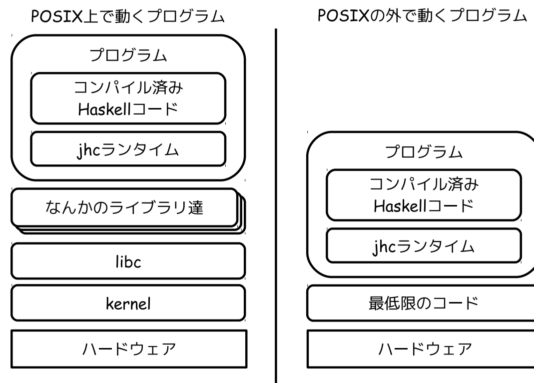

図 3: POSIX 内と外の Haskell プログラムの想像イメージ

さてこの Haskell プログラムを POSIX API のない生のハードウェアが見える世界に移植するにはどうしたら良いのだろう? ランタイムが依存する POSIX API のみをカバーする最低限のコードを用意すればいい。“最低限のコード”。それが Haskell プログラムを動作させるために必要な Haskell で書けない部分、プリミティブだということになる。

ぼくはそんなプログラムを作る練習になる題材を探してみることにした。

0.4.1 調査

ぼくは 9 セルバッテリを付けた ThinkPad [*8]を開いた。ディスプレイから無機質な xmonad のタイルが目に入る。ぼくらは以前 NetBSD kernel を製品開発に採用していた。その頃のクセがまだ残っていて、ぼくの PC には NetBSD のソースコードリポジトリを丸ごと rsync [*9] してあった。ネットワークが不通になっても開発できるようにするのが先進国の外での常識だ[*10]。

jhc で実験的なアプリケーションを作る方針を決めたは良いが、現時点では重大な制約がある。それは jhc の吐くバイナリが再入可能ではないということだ。jhc の吐くバイナリはコンテキストを一つしか持てない。もちろんスレッドも扱えない。つまり、jhc ではハードウェア割り込みを前提とした kernel を書くことは出来ないことになる。しかし何か kernel に近いところの題材が欲しい。NetBSD のソースコードツリーを探していたら格好の題材がすぐ見つかった。bootloader だ。

NetBSD bootloader は起動するとキーボード入力を待つ。この待ちがタイムアウトすると NetBSD kernel が自動的に起動されるのだが、タイムアウト前に何かキーを入力すると bootloader は独自のコマンドラインを起動する。NetBSD bootloader のコマンドラインを起動して、help コマンドを実行してみた結果が以下のログだ。

[*8] 今回対象とする実行環境は Debian GNU/Linux amd64 sid 2013/06/10 時点。 GHC のバージョンは 7.6.3。
[*9] “Linux 上での NetBSD バーチャル開発の方法” `http://d.masterq.net/?date=20110207#p01`
[*10] 筆者は東日本大震災後にこの開発スタイルを取るようになったようでゲソ。

```
>> NetBSD/x86 BIOS Boot, Revision 5.9 (from NetBSD 6.0.0_PATCH)
>> Memory: 639/130048 k
Press return to boot now, any other key for boot menu
booting cd0a:netbsd - starting in 0 seconds.
> help
commands are:
boot [xdNx:][filename] [-12acdqsvxz]
     (ex. "hd0a:netbsd.old -s"
--snip--
help|?
quit
>
```

このコマンドラインから boot コマンドを使えば、ファイル名を指定して NetBSD kernel を起動できる。このコマンドラインループは NetBSD kernel が起動するまでの間割り込み禁止のまま動作する (図 4)。さすがにこのコマンドラインと bootoader の全てのコマンドを Haskell で実装するのは大変だ。しかし help と boot コマンドのみを Haskell でスナッチすることはできるのではないか？ どちらのコマンドももちろん割り込み禁止で動作するからだ。

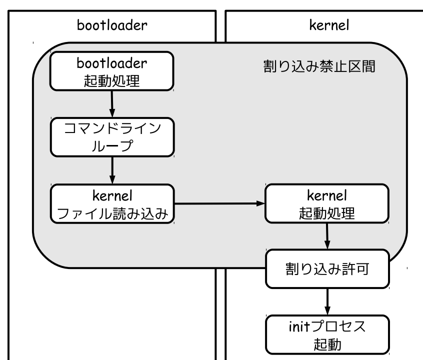

図 4: bootloader は割り込み禁止で実行される

ぼくはまず jhc のランタイム [*11] をよく観察してみることにした。

jhc は Haskell コードをコンパイルすると単一の C 言語ソースコードを吐き出す。この C 言語ソースコードの中には Haskell ソースコードに由来する全てのロジックが写し込まれている。その後 jhc は GCC を呼び出して左記の C 言語ソースコードとランタイムのソースコードを一緒にコンパイルする (図 5)。すると無事実行バイナリができる訳だ。この jhc のコンパイルフローは途中で停止させることもできる。それが “-C” オプション [*12] だ。また “–tdir” オプションを使うことで中間生成されたランタイムソースコードを jhc が削除するのを防止できる。例えば次のようなコマンドを使えば生成された C 言語ソースコードとラ

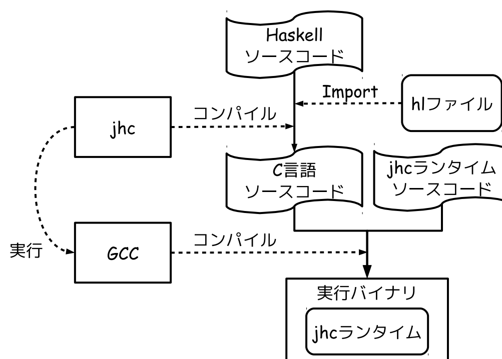

図 5: jhc のコンパイルフロー

[*11] https://github.com/ajhc/ajhc/tree/arafura/rts
[*12] http://ajhc.metasepi.org/manual_ja.html#オプション

ンタイムソースコードが手に入る。

```
$ echo 'main = print "hoge"' > Hoge.hs
$ jhc -C --tdir=rts Hoge.hs
$ ls
Hoge.hs  rts/
$ ls rts
HsFFI.h  jhc_rts_header.h  lib/  main_code.c  rts/  sys/
```

main_code.c が Haskell コードをコンパイルして得られた結果の C 言語ソースコード。その他のファイル群が jhc のランタイムソースコードだ。jhc はコンパイルを実行する毎にランタイムを再コンパイルする。ランタイムの規模が劇的に小さいからこそなせる技だ。

ここで生成された C 言語ソースコード main_code.c の中身を見てみる。

```
char jhc_c_compile[] = "gcc rts/rts/profile.c rts/rts/rts_support.c rts/rts/gc_
none.c rts/rts/jhc_rts.c rts/lib/lib_cbits.c rts/rts/gc_jgc.c rts/rts/stableptr
.c -Irts/cbits -Irts rts/main_code.c -o hs.out '-std=gnu99' -D_GNU_SOURCE '-fal
ign-functions=4' -ffast-math -Wextra -Wall -Wno-unused-parameter -fno-strict-al
iasing -DNDEBUG -O3 '-D_JHC_GC=_JHC_GC_JGC'";
char jhc_command[] = "jhc -C --tdir=rts Hoge.hs";
```

丁寧にも GCC に渡す予定だった引数が列挙されている。そこで指示通りに jhc が本来実行するはずだったコンパイルの続きを手動で実行してみた。

```
$ gcc rts/rts/profile.c rts/rts/rts_support.c rts/rts/gc_none.c rts/rts/jhc_rts
.c rts/lib/lib_cbits.c rts/rts/gc_jgc.c rts/rts/stableptr.c -Irts/cbits -Irts r
ts/main_code.c -o hs.out '-std=gnu99' -D_GNU_SOURCE '-falign-functions=4' -ffas
t-math -Wextra -Wall -Wno-unused-parameter -fno-strict-aliasing -DNDEBUG -O3 '-
D_JHC_GC=_JHC_GC_JGC'
$ ls
Hoge.hs  hs.out*  rts/
$ ./hs.out
"hoge"
```

なるほど。あっさり実行バイナリが手に入った。

0.4.2 設計

ということは下記のような設計が可能なのではないか？ (図 6)

1. bootloader の初期化が終わってから jhc が生成した C 言語ソースコードを呼び出す
2. Haskell を使って書かれたコマンドラインループが起動する
3. コマンドラインループが kernel 起動コマンドを検出したら既存の C 言語ソースコードに戻る

問題はどうやって実現するかだ。NetBSD bootloader は四つのライブラリを内包していて、比較的リッチな環境でプログラミングできる (図 7)。ひょっとすると jhc ランタイムの実行に必要な機能がすでにそろっている可能性もある。とにかくいきなり生のハードウェアの上で動くコードを jhc で作るよりもはるかに楽ができるはずだ。

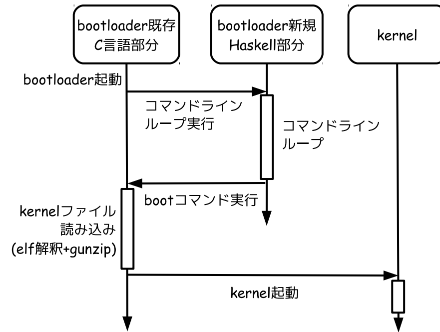

図 6: bootloader の一部を Haskell で再設計

残る疑問は jhc が生成した C 言語ソースコードはどのようにして呼び出されるのかだ。答は生成されたランタイムソースコード [*13] を調べてすぐに判明した。

jhc が生成する C 言語コードの main 関数は以下のような動作をする。

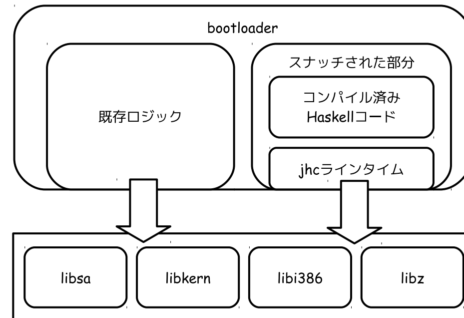

図 7: bootloader で使えるライブラリ

1. hs_init 関数 call
2. setjmp の設定
3. _amain 関数 call
4. hs_exit 関数 call

つまり GHC と同じように hs_init 関数を呼び出した後、_amain 関数という jhc 独自の関数を呼び出せば良いようだ。setjmp の設定は、例外を扱うためだけだ。例外が起きないように Haskell コードを書く上ではこれは不要と思っていい。_amain 関数は jhc が Haskell コードをコンパイルした結果の C 言語ソースコード内にある。

```
// File: rts/main_code.c (ミューテータ)
void
_amain(void)
{
        return (void)b__main(saved_gc);
}

static void A_STD
b__main(gc_t gc)
{
        return ftheMain(gc);
}
```

*13 https://github.com/ajhc/ajhc/blob/v0.8.0.1/rts/rts/jhc_rts.c#L164

ftheMain 関数というのが Haskell の Main.main 関数に相当するようだが、_amain 関数はそのラッパーになっているようだ。

これで材料を集めるのは終わっただろう。設計 [*14] に入ろう。まず既存の NetBSD bootloader のソースコードを観察すると、boot2 という関数がメイン関数のようだった。

```
/* file: sys/arch/i386/stand/boot/boot2.c */
void
boot2(int biosdev, uint64_t biossector)
{
/* --snip-- */
    print_banner();
#endif

    printf("Press return to boot now, any other key for boot menu\n");
```

ここで注意しなければならないことがある。bootloader の中では簡単に printf 関数が使えない。POSIX 上で動作するプログラムと異なり、bootloader の下には libc も kernel もいない。bootloader の中で printf 関数を使うには、ビデオやシリアルの初期化をしなければならない。

この boot2 関数の中ではじめて printf 関数を使う箇所が上のソースコードだった。ということはこの printf 関数の直前までの間にコンソールの初期化は終わっているはずで、当該箇所で Haskell コードをそのまま実行すれば文字を印字できるはずだ。おそらくキー入力も受け付け可能だと思われる。そこで以下のように printf 関数直前に Haskell コードの呼び出しを追記した。

```
/* file: sys/arch/i386/stand/boot/boot2_ara.c */
    print_banner();
#endif
    { /* Run Haskell code */
        int hsargc = 1;
        char *hsargv = "nbboot";
        char **hsargvp = &hsargv;

        hs_init(&hsargc, &hsargvp);
        if (jhc_setjmp(&jhc_uncaught)) {
            jhc_error("Uncaught Exception");
        } else {
            _amain();
        }
        /* hs_exit(); */
    }
    printf("Press return to boot now, any other key for boot menu\n");
```

[*14] 最初のスケッチのソースコード https://gitorious.org/metasepi/netbsd-arafura

これで Haskell コードが bootloader から呼び出されるはずだ。次の課題は hs_init 関数と _amain 関数の先にある jhc ランタイムが依存している関数の実装ということになる。ほとんどは単純なスタブ関数を実装するだけだが、唯一面倒だったのがメモリ管理まわりだった。jhc ランタイムは malloc 関数を要求していた。幸いにも NetBSD bootloader が内包するライブラリが alloc という関数を持っている。これを使えば malloc の API を実装することは可能だ。

これで C 言語側の設計は終わりだ。コマンドラインループを Haskell で記述しよう。 NetBSD bootloader の C 言語での設計を見るかぎり、以下のような Haskell コードと等価のようだった。

```haskell
-- file: metasepi-arafura/sys/arch/i386/stand/boot/Boot2Ara.hs
import Control.Monad
import Data.Maybe
import Data.Map (Map)
import qualified Data.Map as Map
import Foreign.C.Types
import Foreign.Ptr

foreign import ccall "glue_netbsdstand.h command_boot" c_boot :: Ptr a -> IO ()

commands :: Map String (IO ())
commands = Map.fromList [("help", command_help),
                         ("boot", c_boot nullPtr)]

command_help :: IO ()
command_help = putStr $ "\
\commands are:\n\
\boot [xdNx:][filename] [-12acdqsvxz]\n\
\help\n"

main :: IO ()
main = do
  putStrLn "Haskell bootmenu"
  forever $ do
    putStr "> "
    s <- getLine
    fromMaybe (putStr s) $ Map.lookup s commands
```

上記の Haskell コードはコマンドラインから “boot” という入力を検出すると C 言語で書かれた command_boot 関数を呼び出す。この command_boot 関数が kernel をファイルシステムから取り出し起動する。つまりこれでコマンドラインループのみを Haskell コードでスナッチできたわけだ。さて qemu で bootloader を起動してみよう……あれ？ Haskell コードが実行されない。動くはずなのに……

デバッグの結果どうやら malloc が 1MB のメモリを確保しようとしているために、NetBSD bootloader の alloc ヒープが狭すぎて abort しているようだった。bootloader はコンベンショナルメモリ内の 640kB で動作するため 1MB もの大きな alloc ヒープは確保できない。別の方法としては

1MB 以降の潤沢な拡張メモリを alloc ヒープとして使う手もあるが、今度は BIOS から読み書きすることができなくなってしまう [*15]。しかもリソースの制御は設計において重要な課題の一つだ。実際の製品ではリソースは無限にあるわけではなく制約がつくのが通常だ。たかだかコマンドラインループを実現するために最低 1MB ものヒープが必要になるのではこの先 kernel を書く上で致命的な欠陥を生じかねない。ここは踏み止まって省メモリ化を検討すべきだ。そこでぼくは jhc ランタイム中での malloc の使用箇所を洗い出してみることにした。

```
$ grep -n malloc rts/rts/gc_jgc.c
193:    saved_gc = gc_stack_base = malloc((1UL << 18)*sizeof(gc_stack_base[0]));
298:    base = _aligned_malloc(MEGABLOCK_SIZE, BLOCK_SIZE);
320:    struct s_megablock *mb = malloc(sizeof(*mb));
512:    struct s_cache *sc = malloc(sizeof(*sc));
588:    struct s_arena *arena = malloc(sizeof(struct s_arena));
618:gc_malloc_foreignptr(unsigned alignment, unsigned size, bool finalizer) {
```

193 行目と 298 行目がどちらも 1MByte ものメモリを確保している。このサイズを単に小さくしてしまえば逃げられないのだろうか？

- gc_stack_base: 1MByte → 128kByte
- MEGABLOCK_SIZE: 1MByte → 64kByte
- BLOCK_SIZE: 4kByte → 256Byte

再コンパイル……実行……動いた! だましだましコンベンショナルメモリに押し込んだが、jhc は POSIX レイヤーがなくとも動作可能なバイナリを吐き出せることを実証できた。またスナッチモデルが少なくとも割り込み禁止状態で動作するソフトウェアに対しては有効であることもわかった。

ぼくは意気揚々としてみんなにこの成果を報せた。あの苦しみを経験したぼくらならこの jhc コンパイラを製品化可能なレベルにまで、みんなで成長させることができるのではないかと思ったのだ。ところがみんなの反応はぼんやりしていた。

「bootloader を書き換えても何の証左にもなっていないと思う」

「デバイスドライバ書くならメモリマップド IO をさわらなきゃだけど Haskell だと無理じゃない？」

「そもそも C 言語で記述された kernel と同等の表現を得られるのかな？」

「もし製品化するならコンパイラの中身を弄らなきゃならない。素人には無理だよ」

みんなから湧き出る不安と疑問。

ぼくは意気消沈した。

0.5　冷たい壁

海が見える。この国では外洋に接する海岸が多く、しかも地震が多い。例年のように津波による被害が出る [*16]。いつものおだやかな海岸も時に本来の巨大な力を剥き出しにする。

技術探索も同じだ。一見うまく出荷できそうな基本設計も詳細な実装をする段階になって根本的な欠陥が見つかることも多い。出荷できない技術など存在しないも同然だ。人はみな視界を持って

[*15] リアルモードでも拡張メモリをアクセスする方法は存在するらしいです。残念ながら筆者は具体的な手順を知りません……

[*16] http://ja.wikipedia.org/wiki/スマトラ島沖地震

いる。どこまで見通せるかは人それぞれだが、その範囲はたかだか有限なのだ。ぼくの道は光に通じるのか、それともやはり冷たい高い壁が待ち構えるのみなのか……
「モチベーションを複製することは困難でゲソ」
いつの間にかあの娘が隣にいた。
「おぬしが出会う問題と他人が出会う問題は同じようでも微妙に異なるじゃなイカ。それらが蓄積されると問題の集合同士は大きく異なることになるでゲソ」
なにを当然のことを言ってるんだよ……
「モチベーションは問題意識から生まれることが多いでゲソ。するとモチベーションを他人の中に完全に複製するためには、おぬしがこれまで出会ってきた問題を全部説明しなければならないことになるんじゃなイカ？」
そんなことは有限時間では不可能だ。
「いまは初速が必要なのでゲソ」
つまり？
「モチベーションを共有していなくてもプロジェクトに参加したくなる、そんな魅力的な機能と性能と品質とそして明確な用途を示す必要があるのでゲソ。そして究極的にはその魔法の瞬間まではおぬし一人で辿りつくしかないんじゃなイカ？」
しかしぼくの中にあるエネルギーは有限だ。その初速を得るに足りない場合はどうすればいいんだ……
「その時は」
風がきこえる。
「未到達ということになるんじゃなイカ。おぬしはいつも自分のことしか見ていないようでゲソ。いま足場にしている jhc コンパイラという一つの到達点に誰がいったいどうやって辿りついたのか、そのことを考えてみるべき時じゃなイカ？」
…
　…
…　…

決めた。jhc コンパイラをぼく一人の手で成長させる。この純朴なコンパイラが秘めた光をはなつまでは。 jhc コンパイラの作者である **J** *17 に何度か patch を送ってみたが、merge はしてくれどもあまり活発な応答はなかった。おそらくとてつもなく忙しいのかもしれない。もしかしたら……いや、どんな天才でも気力が底をつくことだってある。全てのリソースは有限なのだ。

jhc のランタイムとコンパイルパイプラインの最終段については理解が進んでいたので、修正のアイデアがいくつもあった。そこでぼくは jhc をフォークして **Ajhc** *18 というプロジェクトを立ち上げることにした。オープンソースプロジェクトにおいてプロジェクトのフォークはあまり良いイメージがない。しかしリポジトリがない状況でこれ以上 jhc の機能開発をすることは困難だった。もちろんぼくが実装する機能の宣伝をする場所が欲しかったこともある。

J には悪いと思ったが、正直に何が問題であるのかをまとめてメールを送った。*19 さぁ準備は整った。あとは進むだけだ。ぼくのリソースのあるかぎり。

0.6 二度目のスケッチ: 省メモリ化の追求

プロジェクトを起こしたからにはわかりやすい応用例があった方がいい。最初のスケッチでは bootloader をスナッチしたが、今ふりかえると見た目が地味だったと思う。もっとわかりやすい

*17 John Meacham `http://repetae.net/`
*18 Ajhc - Haskell everywhere `http://ajhc.metasepi.org/`
*19 ANNOUNCE: Start Ajhc project with forking jhc. `http://www.haskell.org/pipermail/jhc/2013-March/001007.html`

アピールが必要だ。最近 Maker ムーブメントの影響でマイコンプログラミングが流行っている。Arduino のような C 言語ではない言語を使う開発環境もある。jhc を使ったマイコン開発という切り口はわかりやすいのではないか？

0.6.1 ハードウェア選定

bootloader のスナッチでは alloc ヒープを 320kByte 確保していた。この alloc ヒープは C 言語の malloc と Haskell ヒープの両方を管理している。kernel のファイル読み込みで alloc ヒープは酷使されるので、マイコンでは Haskell ヒープのみを使い、さらに jhc の GC をよく観察すれば、ヒープ全体の容量は桁を一つ落とすことが不可能でないかもしれない。つまり数十 kByte のメモリ (RAM) で動作する Haskell コードだ。やるからには限界に挑戦したくなっていた。

そんなマイコンを探してみよう。この国には秋葉原のような便利な場所がない。それでもインターネットはやはり便利でネット通販で開発ボードを買える。今回のスケッチにぴったりのボードが見つかった。

- ボード名: STM32F3DISCOVERY `http://www.st.com/stm32f3discovery`
- CPU: ARM Cortex-M4
- ROM: 256kByte
- RAM: 48kByte
- 備考: ST-LINK/V2 チップ付属 - gdb によるデバッグが可能 *20
- 価格: 950 円

すごい時代だ。評価ボードがこんな価格で入手できる。さっそく注文した。

手元に届くまでの間に開発環境を整えよう。STM32F3-Discovery Application Template *21 に STM32F3DISCOVERY 用の LED チカチカのソースコード一式が落ちていた。ありがたくスケッチの足場として使わせてもらうことにした *22 。

数日の後ボードが届いたので、クロス gdb で C 言語のデモを動かしてみたところ問題なく動作した*23 。これでスタート地点に立てたことになる。

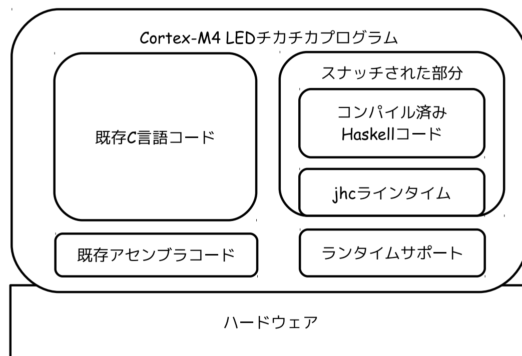

図 8: Corntex-M4 上のプログラムのスナッチ

0.6.2 作戦

さっき見つけた C 言語で書かれている LED チカチカデモをスナッチしてみることにした。

おそらくこのスケッチの全体像は図 8 のようになるだろう。このデモは C 言語とアセンブラでできているはずだった。これを Haskell コードを使ってスナッチする。この時、jhc ランタイムが依存するプリミティブが足りないこともあるかもしれない。その場合は適宜ランタイムサポートのため

*20 `https://github.com/texane/stlink`
*21 `https://github.com/mblythe86/stm32f3-discovery-basic-template`
*22 二度目のスケッチのソースコード `https://github.com/ajhc/demo-cortex-m3`
*23 `http://www.slideshare.net/master_q/20130629-deb-cortexm3`

のコードを書くしかない。このランタイムサポートのコードを小さく抑えれば、jhc ランタイムの依存する機能プリミティブが見えてくるはずだ。

ぼくはこのスナッチのゴールについて考えてみた。アセンブラのコードはおそらく C 言語でさえ記述できないしろものなので、置換することは不可能だ。ということはランタイムサポートと既存 C 言語コードの最小値を得た地点がゴールということになる。

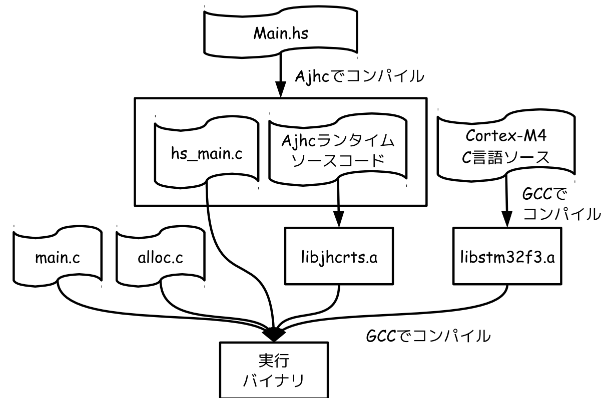

図 9: Corntex-M4 クロスコンパイル環境

さて、先の bootloader のスナッチではあまりちゃんとしたビルドプロセスを作らなかった。まだ jhc についての理解が浅かったためだ。今回は x86 の開発マシンの上で ARM のバイナリをコンパイルするため、しっかりとしたクロスコンパイル環境を整えた方がいい。また、前回のスナッチでは jhc のランタイムソースコードを開発対象のディレクトリに含めてしまっていた。これは開発対象のソースコードが jhc の特定バージョンに紐づいてしまう原因になる。

そこで図 9 のようなクロスコンパイル環境を作った。開発環境に Ajhc ランタイムソースコードを保持するのではなく、Haskell ソースコードをコンパイルする際に生成される Ajhc ランタイムソースコードをライブラリ化してリンクするようにした。これでコンパイル時に使用しているバージョンの Ajhc に対応するランタイムを使うことができる。もし Ajhc ランタイムに修正を入れたくなったら Ajhc 本体に修正を加えなければならない。libstm32f3.a は先の Template に付属していたユーティリティが入っている。また Template には malloc API がなかったので、bootloader の alloc コードを移植した。

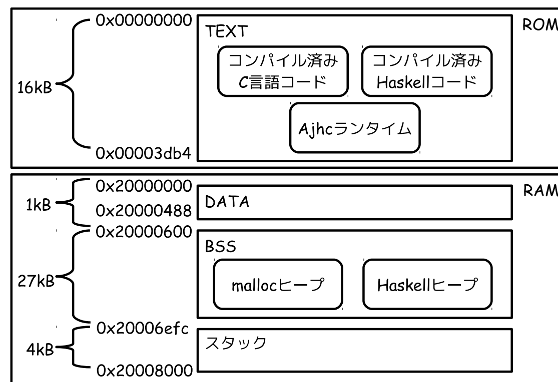

図 10: Cortex-M4 上で動作する実行バイナリのメモリマップ例

いろいろな要素が出てきて混乱してきた。クロスコンパイラによって生成された実行バイナリの中身を一度整理してみることにした。図 10 は実行バイナリのメモリマップだ。

TEXT 領域には手書きした C 言語コードと Ajhc が吐き出した C 言語コードがまざっている。Haskell で動的に確保するサンクは Haskell ヒープの中に配置される。Haskell コードがこの領域より多くのサンクを同時に確保しようとした場合には abort になる。さらに Haskell ヒープの管理を目的として malloc ヒープが用意されている。この malloc ヒープがあふれた場合も即 abort だ。Ajhc の吐き出すコードは再帰の実行のためにスタックを多く消費することがある。スタックは若いアドレスに向かって成長するため、このメモリマップだとスタックが BSS を破壊することがある。製品

化の際にはスタック領域を RAM の先頭に配置した方が良いかもしれない。

クロスコンパイラによって生成された実行バイナリは次のように起動する (図 11)。

1. CPU はリセットベクタから実行を開始する
2. リセットベクタは C 言語の main 関数を呼び出す
3. C 言語の main 関数はハードウェアの初期化を行なう
4. C 言語の main 関数は Ajhc ランタイムの hs_init を呼び出す
5. hs_init 関数は Ajhc ランタイムの初期化を行なう
6. C 言語の main 関数は_amain 関数を呼び出し Haskell コードが走る

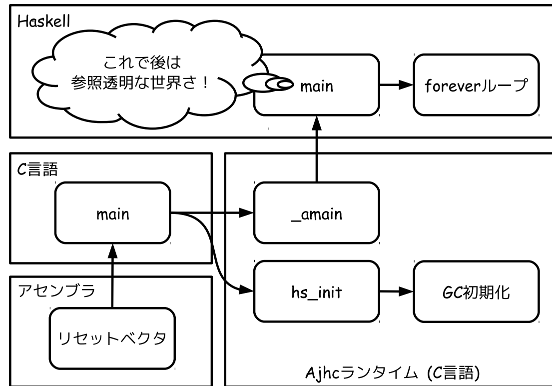

図 11: Cortex-M4 上での Haskell コードの起動

さぁ Haskell コードを書いてみよう。マイコンの Hello World といえば LED チカチカだろう。このボードには LED が 8 つ搭載されているが、その内 LED3 だけを blink させてみた。

```
import Data.Word
import Control.Monad
import Foreign.Ptr
import Foreign.Storable

foreign import ccall "c_extern.h Delay" c_delay :: Word32 -> IO ()
foreign import ccall "c_extern.h &jhc_zeroAddress" c_zeroAddress16 :: Ptr Word16

gpioPin9, led3 :: Word16
gpioPin9 = 0x0200
led3     = gpioPin9

brrPtr, bsrrPtr :: Ptr Word16
brrPtr  = c_zeroAddress16 `plusPtr` 0x48001028
bsrrPtr = c_zeroAddress16 `plusPtr` 0x48001018

ledOff, ledOn :: Word16 -> IO ()
ledOff = poke brrPtr
ledOn  = poke bsrrPtr

main :: IO ()
main = forever $ do
  ledOn led3
  c_delay 10
```

```
  ledOff led3
  c_delay 10
```

この Haskell コードの動作をおおざっぱに図示してみる (図 12)。Haskell は実は Ptr 型を通して生のメモリへ直接読み書きを行なうことができる。書き込み関数が poke 、読み込み関数が peek だ。このボードの場合 0x48001028 アドレスに 0x200 を書くと LED3 が点灯し、0x48001018 アドレスに同じく 0x200 を書くことで LED3 が消灯する。c_delay 関数は C 言語の Delay 関数を呼び出して、時間待ち合わせを行なう。

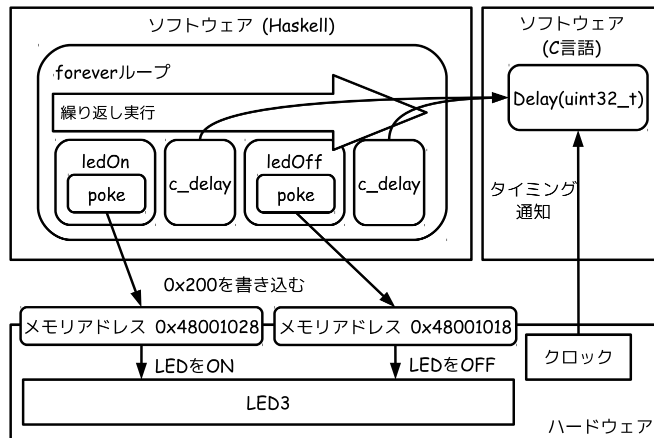

図 12: LED チカチカプログラムの動作

ここで jhc_zeroAddress という C 言語側で定義されたグローバルポインタ経由で Ptr 型を作っていることに疑問を持つかもしれない。これには理由がある。仮に nullPtr [*24] を使って通常の Haskell API のみで実装してみる。

```
--- a/stm32f3-discovery/hs_src/MainBlinkLedSimple.hs
+++ b/stm32f3-discovery/hs_src/MainBlinkLedSimple.hs
@@ -11,8 +11,8 @@ gpioPin9 = 0x0200
 led3     = gpioPin9

 brrPtr, bsrrPtr :: Ptr Word16
-brrPtr  = c_zeroAddress16 `plusPtr` 0x48001028
-bsrrPtr = c_zeroAddress16 `plusPtr` 0x48001018
+brrPtr  = nullPtr `plusPtr` 0x48001028
+bsrrPtr = nullPtr `plusPtr` 0x48001018

 ledOff, ledOn :: Word16 -> IO ()
 ledOff = poke brrPtr
```

この Haskell コードを Ajhc でコンパイルすると Haskell の main 関数は以下のような C 言語ソースに変換されることになる。

```
static void A_STD
ftheMain(gc_t gc)
```

[*24] http://hackage.haskell.org/packages/archive/base/latest/doc/html/Foreign-Ptr.html#v:nullPtr

```
{
        fR$__fControl_Monad_forever__2:;
        {   *((uint16_t *)(1207963672)) = 512;
            saved_gc = gc;
            (void)Delay((uint32_t)10);
            *((uint16_t *)(1207963688)) = 512;
            saved_gc = gc;
            (void)Delay((uint32_t)10);
            goto fR$__fControl_Monad_forever__2;
        }
        return;
}
```

一見 0x48001028 アドレスに 0x200 を正常に書き込んでいるように見える。しかし上記の C 言語ソースコードをよく見ると、ポインタアクセスに volatile が付いていない。これでは GCC 側では最適化で消されてしまうことになる。副作用と認められないのだ。

```
static void A_STD
ftheMain(gc_t gc)
{
        fR$__fControl_Monad_forever__2:;
        {   *((uint16_t *)(1207963672 + ((uintptr_t)&jhc_zeroAddress))) = 512;
            saved_gc = gc;
            (void)Delay((uint32_t)10);
            *((uint16_t *)(1207963688 + ((uintptr_t)&jhc_zeroAddress))) = 512;
            saved_gc = gc;
            (void)Delay((uint32_t)10);
            goto fR$__fControl_Monad_forever__2;
        }
        return;
}
```

元の jhc_zeroAddress を使った Haskell コードを C 言語に変換すると上記のような jhc_zeroAddress グローバルポインタに書き込みアドレスを加えたポインタに 0x200 を書き込むコードが生成される。Haskell 側で Ptr 型に volatile を付けるプラグマがあればよかったのだが、その方法はないようだった。

jhc_zeroAddress は 0 アドレスへのポインタとして C 言語側で宣言されている。そのためこの方式であれば C 言語コンパイラは愚直にメモリ書き込みを行なうアセンブラを吐くようになる。どうやら Ajhc を使ったハードウェア制御コードではこのイディオムがセオリー[*25] になりそうだった。

[*25] https://github.com/ajhc/demo-cortex-m3#write-haskell-code

0.6.3　デバッグ

さて作成したコードを実行してみた。なんとあっさり LED がチカチカ光るじゃないか。型の力はすばらしい。ポインタアクセスをしている界面のみ気をつけて設計すれば残りは型に守られた設計ができるのだ。

ところが先の簡単な forever ループを拡張してちょっと純粋な関数を加えるとコードが動かない。これは何が起きているのだろうか？ gdb で追ってみると例外ベクタに飛んでいる。なぜ例外が起きるのか？ もっと丁寧に追うとコードのかなり深くで例外が起きる。ヒープの制御がおかしいのだろうか。スタックがあふれたのだろうか。

数日の間このことが頭をはなれず、アルバイトでもぼーっとしていた。

「なにを悩んでるの？」

ふりむくと同僚がいた。彼はプログラミング言語にとても詳しく、なにかあるとサポートしてもらっていた。ぼくは思い切って Ajhc の計画について打ち明けてみた。

「うーん、その不具合の原因はよくわからないけれど、その計画は明らかに GC が鍵になるね」

jhc の GC は C 言語のみで記述されていて、行数も 800 行程度と短かい。しかしその実装はトリッキーでこれまで理解することを避けてきていた。それが今のデバッグの困難に結びついていたのだ。

「ちょっとコードを読んできてあげるよ。なにかわかったらメールするね」

数日後、ぼくは彼から jhc の GC (**jgc** と呼ばれる) のしくみを教えてもらった [*26]。とてもわかりやすい資料だった。ぼくはこれを元にして自分なりに jgc のコードを読んでみた。

まず jgc は Haskell ヒープ全体を arena グローバル変数で管理する (図 13)。次に Haskell のサンクなどのデータを格納するための器がある。その最も大きい単位が megablock だ。Haskell ヒープの容量が不足するたびに megablock は追加で確保される。megablock の中には固定サイズの block という構造がある。この block をさらに固定サイズのチャンクに分割して、そのチャンクに Haskell で使用するデータを格納する。

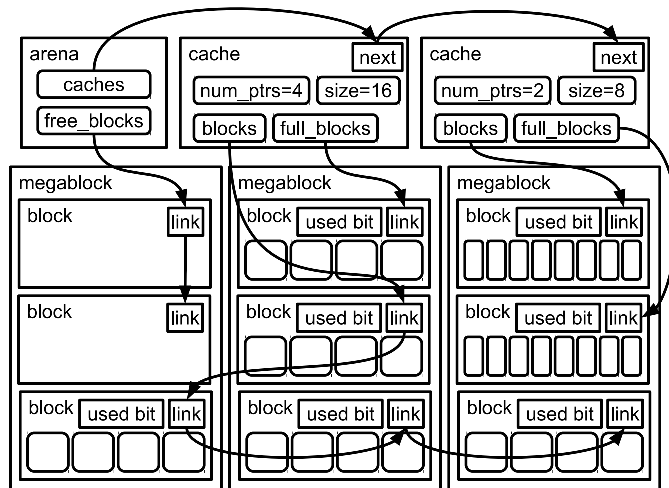

図 13: jgc での Haskell ヒープの管理

block には used bit の配列があり、使用中 [*27]のチャンクに対応するビットには 1 を。未使用のチャンクに対応するビットには 0 を書くのが決まりだ。さらに block には link というメンバーがあり、当該の block が Haskell ヒープ全体から見てどこに繋っているのかリスト管理できるようになっている。例えば、s_arena 構造体には free_blocks というメンバーがあり、このメンバーに繋がってい

[*26] 小二病でも GC やりたい `http://www.slideshare.net/dec9ue/gc-16298437`

[*27] ここでは GC ルートから辿れることを使用中を呼んでいる

る block は内包する全てのチャンクが未使用であることを表わす。

ここから少し見慣れない構造がでてくる。**cache** だ。s_arena 構造体の caches メンバーには cache という構造がぶらさがっている。この cache の先には block がぶらさがっているのだが、その block の中のチャンクのデータ構造を cache によって定義している (図 14)。すなわち cache の num_ptrs メンバーはチャンクの中のポインタの個数を表わし、size メンバーはチャンクの全体サイズを表わす。

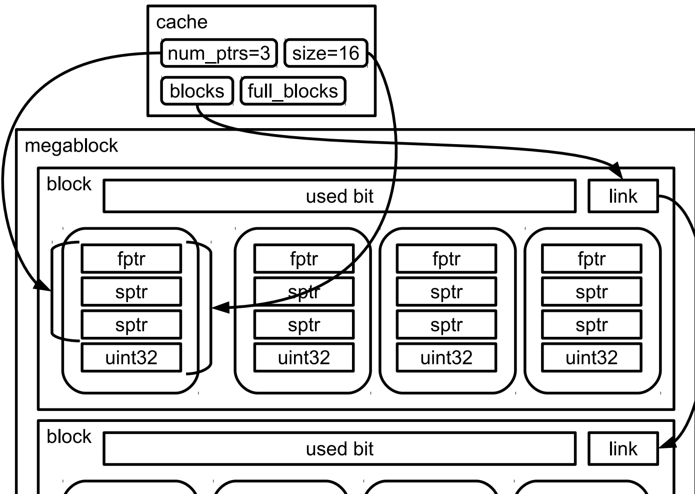

図 14: jgc でのチャンクの構造は cache で定義される

この時チャンク内のポインタは先頭から配置されるのが約束になっている。ポインタはスマートポインタになっていて格納される値は即値である可能性もある[*28]。この cache のリストは Ajhc ランタイムの初期化の中で find_cache 関数によって動的に確保される。

このチャンクは Haskell 由来のデータを格納しているが当然 GC する必要がある。jgc の GC はミューテータがコンストラクタによって Haskell ヒープから新しいメモリ領域を確保しようとする時にのみ呼び出される。また GC はマーキングだけを行ないスイープをしない。というのも cache の blocks メンバーの used bit を検索すれば未使用チャンクが見つかるし、仮にそこで見つからなかった場合には s_arena 構造体の free_blocks から新しい block を補充するれば良いからだ。jgc の GC は以下のような手順を取る (図 15)。

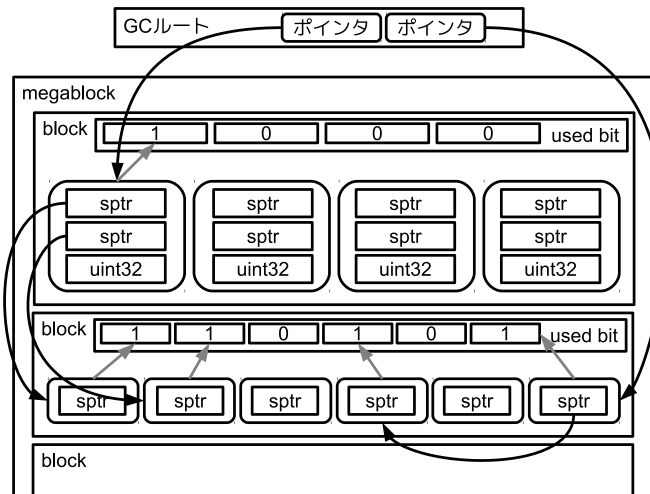

図 15: jgc の GC マーキング

1. Haskell ヒープのアロケート関数 s_alloc がミューテータによって呼び出される

*28 `http://ajhc.metasepi.org/manual_ja.html#ランタイムシステム`

2. s_alloc 関数は要求されたコンストラクタに対応する未使用チャンクを探す
3. 2 で見つからなかった場合 megablock を新たに確保すべきか GC すべきかを判定する
4. 3 で GC すべきと判断した場合 gc_perform_gc 関数を呼び出す
5. gc_perform_gc 関数は Haskell ヒープ全ての used bit を 0 クリアする
6. gc_perform_gc 関数は GC ルートから参照を辿り、当該チャンクに対応した used bit に 1 を立てる
7. 2 と同じように未使用チャンクを探してミューテータに渡す

スマートポインタにはいくつか種類があり、fptr_t もその一種だ。この fptr_t には最初はクロージャの評価関数へのポインタが格納されている。つまり未評価サンクだ (図 16)。このサンクを評価するにはサンクのスマートポインタを eval の第二引数に渡す。すると eval はサンクの遅延ビットをチェックし、未評価サンクだと判定すれば fptr_t の先にあるクロージャ評価関数に node_t を渡す。クロージャの評価が終わったら update 関数を呼び出してチャンクに入っている fptr_t をクロージャの評価結果で上書きする。この時 fptr_t の遅延ビットは 0 にする。これで評価済みサンクのできあがりだ。

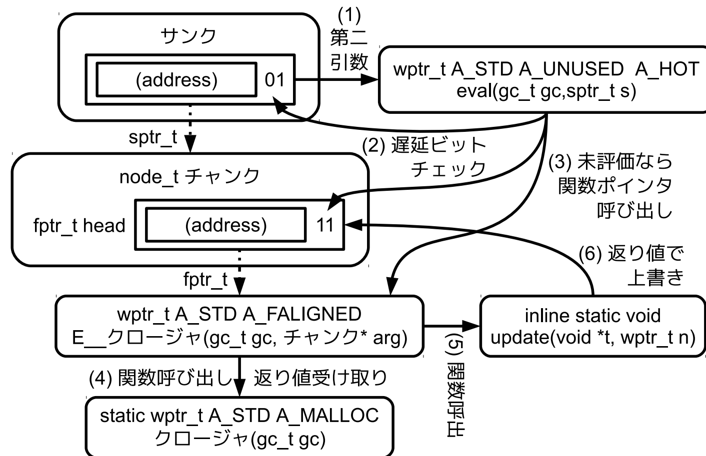

図 16: サンク評価の一例

かなり GC のコードを読み込んだおかげで Ajhc のランタイムの動作が想像できるようになった気がする。やはり同僚のアドバイスは正しかったようだ。ありがたいことだった。

ぼくはデバッグを再開した。やはり例外が起きている。どうも単純な IO のみの forever ループを逸脱すると Haskell ヒープを使うようになるようだ。この時サンクの評価のために関数ポインタへのジャンプ命令が実行されるのだが、その近辺で例外ベクターに飛んでしまう。関数ポインタの先も text 領域の範囲内だ。これはソフトウェアの問題ではなく ARM プラットフォーム側の問題ではないか？

ぼくは objdump で eval 関数のアセンブラを見てみた。関数ポインタジャンプは ARM では blx という命令のようだ。なんとなく嫌な予感がする。 ARM の命令リファレンスから blx 命令の章[*29] を見てみよう。

```
The BLX instruction can change the instruction set.
BLX label always changes the instruction set. It changes a processor in ARM
state to Thumb state, or a processor in Thumb state to ARM state.
BLX Rm derives the target instruction set from bit[0] of Rm:
* if bit[0] of Rm is 0, the processor changes to, or remains in, ARM state
```

[*29] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489i/CIHBJCDC.html

```
* if bit[0] of Rm is 1, the processor changes to, or remains in, Thumb state.
```

な ん だ と ？

Cortex-M4 は ARM state はサポートせず Thumb state のみしか使えないはずだ。ということは関数ポインタアクセスの場合は 0 ビット目に必ず 1 を立てないと例外が起きるということなのか？

なんというアーキティクチャだ……信じられない。

まだ納得できないので GCC が良きにはからってくれないか Web 検索してみた。ところがどうも最新の GCC でもこの問題には決着がついていない [*30] ようだった。さすがにこれは Ajhc のランタイム側に修正を加えるしかない。ぼくは "JHC_ARM_STAY_IN_THUMB_MODE" というコンパイルフラグ[*31] を用意することにした。ソースコード [*32] [*33] を読めばわかる通りだが、このフラグを有効にすると関数ポインタを使う前に 0 ビット目を立てる。

やっと Haskell ヒープを使うプログラムが動作するようになった。ぼくは安堵して夕食を取ることにした。少しビンタンも飲もう。ところが机に戻ってみるとさっき実行させたままだったデモアプリケーションが停止している。おかしい……abort で停止している。gdb でバックトレースを取ってみるとどうやら s_alloc でメモリ領域を確保しようとして失敗しているようだった。メモリリークをしているのか？

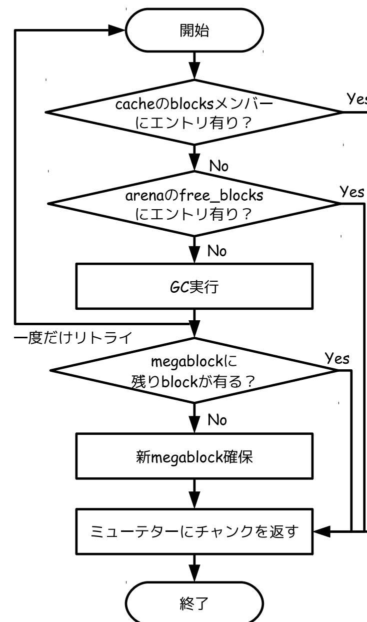

図 17: s_alloc 関数フローチャートと GC 後のリトライ追加

もう一度 s_alloc のソースコードを良く読んでみることにした。すると原因が判明した。jgc は megablock と block の確保を富豪的に楽観的に行なう。図 17 のフローチャートは s_alloc 関数の実行フローだ。よく見ると GC を実行した後に megablock を確保してしまうことがわかる。このままではヒープが足りているにもかかわらず、不要な megablock を投機的に確保してしまう。メモリが豊富にあるのであれば、時間効率をかせぐために投機的な megablock 確保は有効かもしれない。しかし数十 kB しかメモリがないマイコンではこの富豪的な megablock 確保方式は致命傷だ。

これもまた Ajhc ランタイムを修正する他にない。"JHC_JGC_NAIVEGC" というコンパイルフラグを作った。このフラグを有効にすると GC 実行の後、一度だけ s_alloc 関数を最初からリトライする。GC をしているということは blocks もしくは free_blocks にエントリが追加されたかもしれないからだ。

これでやっとデモプログラムが安定動作するようになった。ちょっと凝ったプログラムを作ろう。任意の文字列を LED でモールス信号表示するプログラムだ。まずモールス信号の型を決める。

[*30] http://communities.mentor.com/community/cs/archives/arm-gnu/msg01904.html2
[*31] http://ajhc.metasepi.org/manual_ja.html#cflags で指定できる特殊な define
[*32] https://github.com/ajhc/ajhc/blob/v0.8.0.4/rts/rts/jhc_rts.h#L99
[*33] https://github.com/ajhc/ajhc/blob/v0.8.0.4/rts/rts/jhc_rts.c#L147

```
data Morse = S | L | LetterSpace | WordSpace
```

S がトン、L がツーだ。制御コードは長いので、ここではその構造を図にしてみることにする (図 18)。containers ライブラリの Map を使って morseCode という文字→モールス信号列の変換表を作る。さらにモールス信号を LED の blink アクション列に変換する morseToIO 関数を作る。この二つを組み合わせることで任意の文字列を LED の blink アクション列に変換する morseEncodeIO 関数を作ることができる。最後にまた forever ループに食わせれば任意の文字列を繰り返しモールス信号表示するプログラムが完成する。

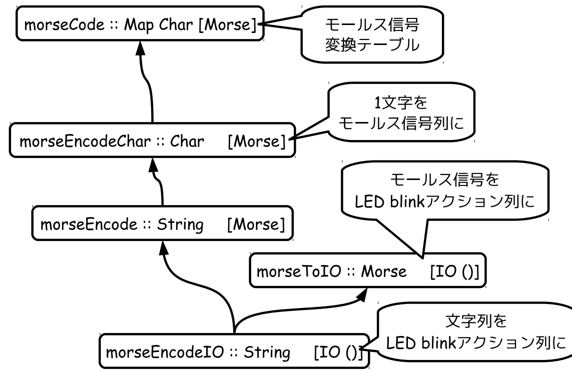

図 18: モールス信号 LED 表示デモの関数

ぼくはこのモールス信号プログラムを作っている時、一度もデバッガを使わなかった。また Linux 上でのテストもしなかった。ただ書き下してコンパイルが通ればその通りに動いた。型安全の威力はどのドメインにおいても有効なのだと思い知った。

このデモをみんなに見せてみた。こんどはみんなびっくりしてくれた。しかしツテを頼っても Ajhc を収益化する見込みは得られなかった。アルバイトをしながら Ajhc を開発していてもなかなか先に進めない。どうやらもう一頑張り必要なようだった。

0.7　あこがれ

ぬるくなったビンタンの瓶が見える。最近は暇されあれば jhc のソースコードを読んでいた。jhc のソースコードはおせいじにもキレイとは言いがたかった。それでも時間をかければ全体像が見えてきた。“An informal graph of the internal code motion in jhc” [*34] にも解説があるが、もっと関数レベルの関係図を頭に入れておきたい。描いてみると図 19 ができあがった。

まだ少しイメージがつかみにくいので型の変換に注目してみよう。jhc のコンパイルパイプラインは Haskell ソース文字列を以下の型を通して C 言語ソース文字列に変換する。この型の変換そのものがコンパイルという行為だというわけだ。

1. [FilePath] 型 (Haskell ソースコードのファイルパス)
2. CollectedHo 型∋ IdMap Comb 型∋ E 型 (コンパイル対象単体での最適化)
3. Program 型∋ Comb 型∋ E 型 (プログラム全体の最適化)
4. Grin 型∋ [FuncDef] 型∋ Lam 型 (λ 抽象での最適化)
5. ByteString 型 (C 言語ソースコード)

[*34] http://repetae.net/computer/jhc/big-picture.pdf

Haskell コードは jhc コンパイルパイプラインの中で大きく二つの型で表現されているようだった。E 型と Grin 型だ。E 型は二つの最適化フェーズで使われる。一つ目はコンパイル対象単体での最適化だ。実行プログラムをコンパイルする際にもこの最適化が行なわれるが、より注目すべきは hl ファイルだ。この hl ファイルは jhc におけるコンパイル済み Haskell ライブラリのフォーマットで、第一段階の E 型最適化が完了した時点で凍結された Haskell コードが格納されている。二つ目はプログラム全体での最適化だ。この第二段階の E 型最適化ではコンパイル対象と hl ファイルを一緒にして最適化する。そのため hl ファイルの中でも未到達な関数はこの第二段階で削除されることになる。Grin 型は C 言語ソースコードに変換される直前の変形に用いられる。

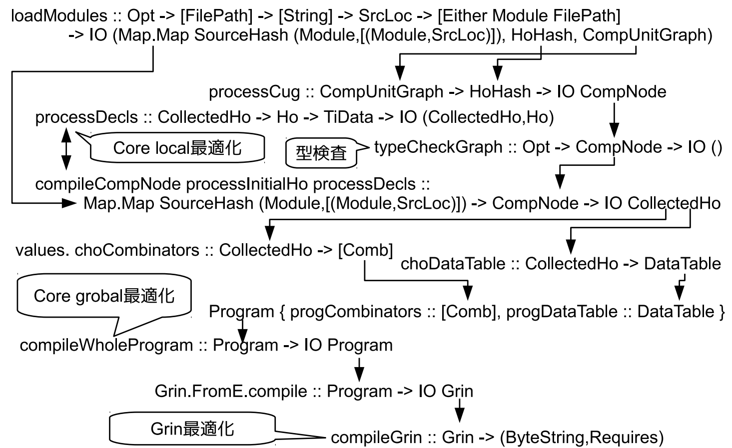

図 19: jhc コンパイルパイプラインの全体像

他にまとまったドキュメントには “Jhc User's Manual” [*35] がある。このドキュメントは英語で、ぼくには読みにくかった。そこで英語が得意な友達 [*36] に協力してもらって日本語訳 [*37] をしてみた。このマニュアルを読むことで jhc の設計ポリシーがわかってきた。

- grin は未到達のコードを削除する
- hl ファイルは動的/静的リンクなどはできず、grin にかけてバイナリ化しないと実行できない
- Haskell のプリミティブ型は C 言語の型そのまま
- コンパイルパイプライン途中の AST をダンプできるオプションが多数ある
- m4 プリプロセッサのような変態的な機能
- スマートポインタのために Int 型は 30 ビット幅
- Integer 型はまっとうに実装されておらず IntMax 型と同じ精度しか保持できない
- pure type system という型システムを採用していて、これはラムダキューブに対応する

特に重要なアイデアがプリミティブ型についてだ。

```
-- File: ajhc/lib/jhc-prim/Jhc/Prim/Bits.hs
data Bits32_  :: #

-- File: ajhc/lib/jhc/Jhc/Type/Word.hs
data {-# CTYPE "int"      #-} Int = Int Bits32_
```

[*35] Jhc User's Manual(英語) http://repetae.net/computer/jhc/manual.html
[*36] @dif_engine さんに手伝ってもらったでゲソ!
[*37] Ajhc ユーザーズマニュアル (日本語) http://ajhc.metasepi.org/manual_ja.html

この CTYPE というのは型プラグマ [38]で FFI の先である C 言語から見た型を指示する。つまり Haskell の Int 型は C 言語側から見ると C 言語の int 型に見えるということだ。Bits32_型は "#" という見慣れない型に落ちているが、これは "プリミティブ型である" という意味しか持っておらず、単に Haskell 側の型推論のつじつま合わせとして使われる。これらのプリミティブ型は当然アンボックス化された型で、この型をボックス化するのは Haskell で書かれたライブラリの責務だ。

この型定義を見て、ぼくは (+) 演算子の実装はどうなっているのか気になった。jhc の Haskell ライブラリ側では以下のような実装になっており、Int 同士の和は "Add" というプリミティブ型に落とされているようだった。この Add プリミティブ型は jhc コンパイラでの grin=>C への変換で単に C 言語の "+" 演算子に変換される。Int は完全にアンボックス化されて直接 C 言語コードに落ちることになるわけだ。

1. Haskell の (+) 演算子
2. ↓変換 — ajhc/lib/jhc/Jhc/Num.m4 (Num クラスの (+) 関数定義)
3. jhc の "Add" プリミティブ
4. ↓変換 — ajhc/src/Cmm/Op.hs (プリミティブの定義)
5. ↓変換 — ajhc/src/C/Generate.hs (オペレータの定義)
6. ↓変換 — ajhc/src/C/FromGrin2.hs (Grin=>C 変換)
7. C 言語 "+" 演算子

今度は型の変換を調べてみよう。例えば fromIntegral 関数を使った Int から Float への変換だ。

1. Haskell の fromIntegral :: Int -> Float 関数
2. ↓変換 — ajhc/lib/jhc/Jhc/Num.hs (fromIntegral の定義)
3. fromInteger . toInteger
4. ↓変換 — ajhc/lib/jhc/Jhc/Num.m4 (toInteger の定義)
5. fromInteger . I2I
6. ↓変換 — ajhc/lib/jhc/Jhc/Float.hs (fromInteger の定義)
7. I2F . I2I
8. ↓変換 — ajhc/src/Cmm/Op.hs (I2I のようなキャスト型定義)
9. ↓変換 — ajhc/src/C/Generate.hs (キャスト操作の定義)
10. ↓変換 — ajhc/src/C/FromGrin2.hs (Grin=>C 変換)
11. C 言語の二回キャスト (float)(int32_t)

なにやらややこしいが "I2I" と "I2F" というプリミティブを合成したものが Int から Float への変換のようだ。この二つのプリミティブはやはり grin=>C への変換で解釈される。そうしてなんのことはない単に C 言語のキャストに変換されるだけだ。

もっと簡単に言ってしまえば、jhc において Haskell のデータ型は全て C 言語と紐づけられたプリミティブ型のみで構成されている。もちろん Haskell 関数の評価については C 言語とは別物だ。C 言語の演算子や関数が部分適用された場合には相応の変換をかける必要がある。しかしその評価順序さえ C 言語と等価なもの (C 言語への変換直前の grin コード) に落としてしまえば、後はプリミティブ型を C 言語型に写像すればそのまま C 言語コードができあがる。この一見不可能に思えるアイデアこそが jhc の秘密のようだった。jhc は Haskell コードを C 言語に変換することを最初から強く意図して設計されていたのだ。

jhc のソースコードを読めば読むほどぼくはコンパイラという世界の魅力にとりつかれた。こん

[38] `http://ajhc.metasepi.org/manual_ja.html#型プラグマ`

なアイデアを思いつきそして具現化したJの才能にあこがれた。そしてもちろん嫉妬した。なぜこれほどの頭脳がオープンソースの世界で発揮されず、開発が停滞してしまっているのか。なぜ世界はこの偉業を引き継ごうとしないのか。焦りと失望が頭をめぐる。
「なにを考えているでゲソ？」
元気なあの声がする。
「コンパイラの開発仲間が増えないことに焦っているんじゃなイカ？」
そうだ。ぼく一人でできることは限られている。このままのスピードではいつになったら kernel をスナッチできるのか予想ができなかった。
「そもそも今の Ajhc コンパイラをはじめて見た者はどう感じると思うでゲソ？」
小さなマイコンにも適用できる組み込み用途の Haskell コンパイラ。
「じゃあ Haskell 言語をはじめて使ってみようと思った者が使うコンパイラはなんでゲソ？」
……GHC 以外にありえない……
「ということはこのままでは小規模組み込みのエンジニアでなおかつ型推論を持つ言語に興味のあるユーザーしか獲得できないでゲソ。人はいきなりコンパイラの開発者になったりしないものでゲソ。まずはそのコンパイラのユーザーになって、そしてコンパイラそのものに興味が出た者が開発者に昇格するのでゲソ。世の中の人間が、おぬしのように自分が使いもしないコンパイラをいきなりいじりはじめる変態ばかりだったら大変でゲソ」
「おぬしは今まで Ajhc に適したドメインにしかコンパイラの応用例を示してこなかったのでゲソ。それはたしかに有用だったでゲソ。しかしこれ以上に開発者を増やしたいのであれば Ajhc コンパイラの適用可能なドメインを広げる必要があるんじゃなイカ？ これまではプロトタイプに必要な機能しか Ajhc コンパイラに追加してこなかったじゃなイカ。つまり曳光弾を使うべきときが来たのでゲソ」
しかし、そもそも Ajhc に欠けている機能はあまりにも多い。まだ通っていない退行テストがたくさんある。GHC とプリミティブ型が違うため多くのライブラリはそのままでは移植できない。Template Haskell も使えない。どこから手をつければいいんだ？
「とぼけても無駄でゲソ! Ajhc に決定的に欠けている機能が、おぬしにはすでに解っているじゃなイカ。もちろんその機能は kernel を設計する際にも必須でゲソ。実装できないのであれば、Ajhc の先には kernel をスナッチ可能なコンパイラとしての未来はない、ということになるんじゃなイカ？」

0.8 三度目のスケッチ: コンテキストを操る

そう、Ajhc の決定的な欠陥。それは **コンテキスト**が一つしか持てないことだ。これは最初のスケッチの時からわかっていたことだ。このままではスレッドを実現できないし、割り込みコンテキストも扱えない。UNIX ライク kernel の中はイベントドリブンの処理が主であることを考えると、今の Ajhc では kernel の初期化処理の一部しかスナッチできないように思えた。

今は kernel をスナッチしているわけではない。そこで、二度目のスケッチをよく観察してみた。す

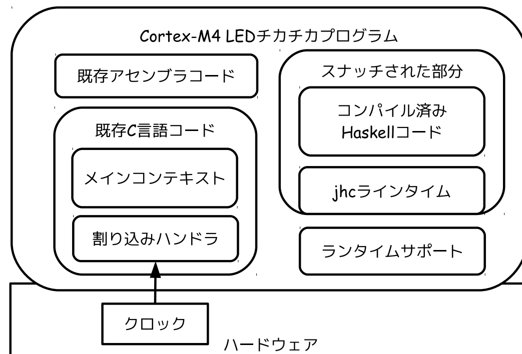

図 20: 二度目のスケッチでスナッチできなかった部分

ると、図 20 のように既存の C 言語コードは二つの部分にわかれていることがわかった。

二度目のスケッチでスナッチ対象にしていたのはメインコンテキストだ。このコンテキストが通常の動作では動いている。しかしクロック例外が起きた場合、即座にメインコンテキストは停止して割り込みハンドラに遷移する。つまり **受動的コンテキストスイッチ** が起きる。この割り込みハンドラをスナッチするにはコンパイル済み Haskell コードはもちろん、jhc ランタイムさえ **再入可能** でなければばならない。なにせランタイムの関数を実行途中のどの段階においても、受動的コンテキストスイッチが起きて割り込みハンドラが起動し、さらには再度実行途中の同じランタイムの関数が再実行されてしまう可能性があるのだ!

少し脱線するが **能動的コンテキストスイッチ**というものも存在する。その例としては kernel のプロセス切り換えや、ユーザ空間スレッドライブラリのスレッドスイッチ、それからコルーチンの yield などがあるだろう。今回の Cortex-M4 のデモの中にはこの能動的コンテキストスイッチは見つからなかった。ChibiOS/RT [*39] のようなスレッドを使う OS を Cortex-M4 に搭載する場合にはスナッチ対象になる可能性はある。しかし受動的コンテキストスイッチよりは難易度が低いだろう。

しかし当然だが複数コンテキストの管理は難しい問題だ。既に jgc の実装が単一コンテキストを前提にして作り込まれてしまっている。しかも UNIX ライク kernel を実装することを考えると、スレッド対応だけではなく再入可能も必須だ。多くの言語処理系は受動的なコンテキストスイッチを受け付けないが、Ajhc に関してはその逃げは許されない。さらに割り込み応答のジッターをなくすためにはメインコンテキストが GC を実行している最中でも割り込みコンテキストに受動的に切り換え可能でなくてはならない (図 21)。GC の実行最中で単純に割り込み禁止にすればいいというものではない。GC の処理時間が長いかもしれないからだ。もちろん GC を小間切れにする方法もあるが、いずれにせよなんらかの対策が必要ということだ。

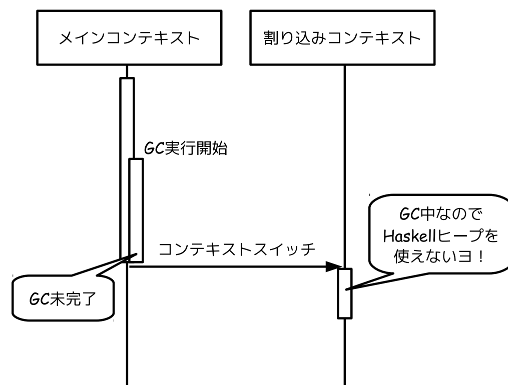

図 21: GC 実行中の割り込みコンテキストへのスイッチ

もう少しだけ整理して考えてみよう。そもそもコンテキストとはいったいなんなのだろう？ Ajhc の文脈でのコンテキストとは C 言語のコンテキスト、もっと言えば CPU のコンテキストのことだ。CPU は **レジスタ**を持っており、そのレジスタを変化させることでプログラムを実行している。近代の CPU ではさらにそのレジスタの中にはスタックポインタがあり、メモリ上の一点を指し示している。その先のメモリには**スタック領域** があり、その中に計算途中のデータが納められている。なんらかのイベントをトリガーにして、このレジスタとスタックを違うものに入れ代える行為がコンテキストスイッチだ。能動的コンテキストスイッチではこのトリガーのタイミングはソフトウェア自身で決定できる。ところが受動的コンテキストスイッチではこのトリガーはソフトウェアではなくハードウェアによって任意のタイミングで起きてしまう。この “いつトリガーが引かれるのかソフトウェア側からわからない” という点が難題なのだった。

[*39] ChibiOS/RT free embedded RTOS `http://www.chibios.org/dokuwiki/doku.php`

0.8.1 jgc 詳細

再入可能を実現するためにはクリティカルリージョンを見つけだして管理することが必要になる。Ajhc の中では GC がその主要因であることは間違いない。もう少し jgc のしくみを詳細に追ってみよう。jgc の GC ルートはいったい何なのだろうか。GC ルートの種類によっては排他制御が必要になるはずだ。ランタイムの gc_perform_gc 関数の動作を見てみよう。

1. GC マーク用のバッファ確保
2. すべての cache の下にある block の used bit を 0 クリア
3. 全 StablePtr をマークしてバッファにコピー
4. GC スタック内のスマートポインタをマークしてバッファにコピー
5. バッファの後ろからスマートポインタを取り出す
6. スマートポインタの先にあるスマートポインタをマークしてバッファにコピー
7. バッファが空でなければ 5 に戻る
8. バッファを解放

ということは、主に GC ルートとなるのは以下の二つのようだ。

- 確保した StablePtr
- GC スタック (gc 変数と gc_stack_base グローバル変数の間にある要素)

StablePtr の方は簡単だ。 newStablePtr 関数によって、GC が回収しないグローバルなポインタを作った場合、当該ポインタは GC ルートになるべきだ。一方 GC スタックというのは何者だろう？ saved_gc と gc_stack_base という二つのグローバル変数は Ajhc のランタイムが起動する際に GC スタック領域の先頭を指すように初期化される[*40]。つまり最初は saved_gc と gc_stack_base は同じ場所を指していたわけだ。この saved_gc は Ajhc ランタイムから呼び出される Haskell コードのエントリポイントである _amain 関数の先では gc という引数によって取り回される。

```
// File: main_code.c (ミューテータ)
void
_amain(void)
{
        return (void)b__main(saved_gc);
}

static void A_STD
b__main(gc_t gc)
{
        return ftheMain(gc);
}
```

この引数 gc をミューテータがどのように使うかというところが少しややこしい[*41]。

[*40] https://github.com/ajhc/ajhc/blob/v0.8.0.4/rts/rts/gc_jgc.c#L198
[*41] https://github.com/ajhc/ajhc/blob/v0.8.0.4/rts/rts/gc_jgc.h#L53

```
// File: main_code.c (ミューテータ)
static wptr_t A_STD A_MALLOC
fR$__fJhc_Basics_$pp(gc_t gc,sptr_t v80776080,sptr_t v58800110)
{
    {   gc_frame0(gc,1,v58800110); // <= GC ルートに v58800110 を登録
        wptr_t v100444 = eval(gc,v80776080);
        if (SET_RAW_TAG(CJhc_Prim_Prim_$BE) == v100444) {
            return eval(gc,v58800110);
        } else {
            sptr_t v106;
            sptr_t v108;
            /* ("CJhc.Prim.Prim.:" ni106 ni108) */
            v106 = ((struct sCJhc_Prim_Prim_$x3a*)v100444)->a1;
            v108 = ((struct sCJhc_Prim_Prim_$x3a*)v100444)->a2;
            {   gc_frame0(gc,2,v106,v108); // <= GC ルートに v106 と v108 を登録
                sptr_t x7 = s_alloc(gc,cFR$__fJhc_Basics_$pp);
```

このソースコードでミューテータは s_alloc 関数か eval 関数を呼び出す手前で GC ルートにすべきスマートポインタを gc 変数の手前に登録している。つまり GC スタックにスマートポインタを積んでいるわけだ。この GC スタックへの登録マクロは gc_frame0 という名前だが、呼び出し前に必ずスコープを新たに作ることが約束事だ。こうすることで、スコープを抜けると自動的に元の GC スタックが復元される (図 22)。

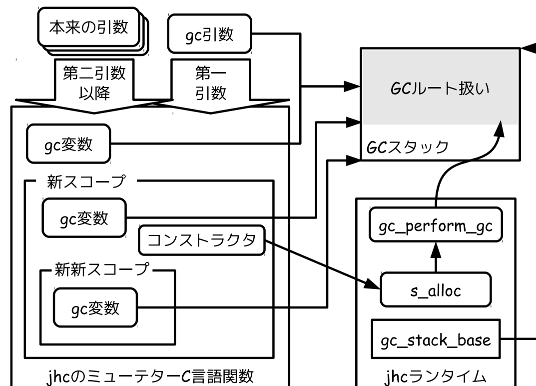

図 22: GC スタックとミューテータ

こんな簡単なしくみで GC ルートを制御できるということが信じがたいが、とにかく jhc ではうまくいっていたようだ。おそらくコンパイルパイプラインの中段あたりに秘密があるに違いなかった。

0.8.2 クリティカルリージョンの排他方法

jgc の GC ルートのしくみがわかったので、Ajhc のクリティカルリージョンをあらいだしてみよう。その前にクリティカルリージョンを保護するためのロックについてリストにしておこう。Ajhc は POSIX の上だけではなく様々なドメインで使えるようでなければならない。

- pthread_mutex_lock [*42]

[*42] http://netbsd.gw.com/cgi-bin/man-cgi?pthread_mutex_lock++NetBSD-current

- NetBSD kernel の mutex_enter [*43]
- CAS を使ったシンプルなロック
- 割り込み禁止
- ロックなし

POSIX の上で動かすときは pthread_mutex_lock を使えばいい。Cortex-M4 のようなマイコンでのロックは割り込み禁止で十分だ。遠い将来 NetBSD kernel をスナッチすることになったら mutex_enter に対応させる必要がある。これまでの jhc と互換性のある動作、つまり pthread を使わない場合にはロックなしで良いかもしれない。mutex_enter は API が複雑なので対応は後回しにすることにして、現状では pthread_mutex_lock に合わせた Ajhc ランタイムの API を定義して、他のロック機構を使う場合には切り換えられるようにした。

排他とは言ったが、再入可能についても考えた場合にはただ単にロックすれば良いというものではない。例えばコンテキスト A が処理 X 中にロック Y を獲得していたとしよう (図 23)。その最中に割り込みが入り、今度は割り込みコンテキスト B が処理 X を再度実行してロック Y の獲得を試みたとする。この時、当然ロック Y への二回目の獲得であるために割り込みコンテキスト B はロック Y の解放待ちになる。ところがロック Y を獲得しているコンテキスト A は割り込みコンテキスト B が完了しないかぎりロック Y を手放さない。みごとなデッドロックの完成だ。このとき処理 X は “再入可能ではない” とされる。スレッドセーフであっても再入可能でなくなってしまうこのような要因をコンパイラで防止することはできないものか？ Haskell コードやランタイムに mutex を使わせることによっていらぬ危険が増えるのは避けたかった。

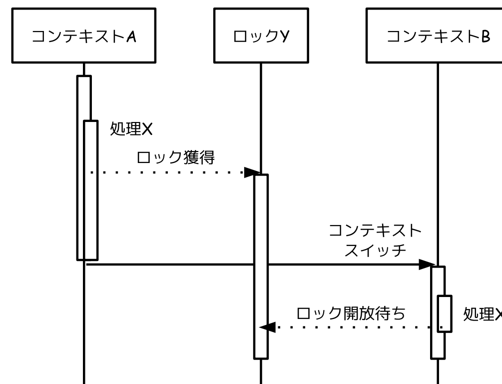

図 23: 割り込みと自コンテキストによる二重 lock

ぼくは長い時間この問題について考えてみた。しかし NetBSD kernel mutex_enter について調べているうちに結論が得られた。結論として、この再入時の二重ロックの問題はコンパイラによって解決されるべき問題ではない。mutex の実装によって防御するか、プログラマが適切に再入時の排他制御を設計する必要がある。

NetBSD kernel の mutex_enter 関数は、以下のような API で制御される。mutex_enter 関数は pthread_mutex_lock 関数と等価、mutex_exit 関数は pthread_mutex_unlock 関数と等価だ。注目すべきは pthread_mutex_init 関数と同じ機能を持つ mutex_init 関数だ。属性を指定する type 引数があるところまでは似ているが ipl という見慣れない引数が追加されている。この mutex をロックしている最中は ipl 引数で指定したレベルまで割り込みが禁止される。

```
// File: netbsd/src/sys/sys/mutex.h
* void mutex_init(kmutex_t *mtx, kmutex_type_t type, int ipl);
```

[*43] http://netbsd.gw.com/cgi-bin/man-cgi?mutex_enter++NetBSD-current

```
* void mutex_enter(kmutex_t *mtx);
* void mutex_exit(kmutex_t *mtx);
```

ここで割り込みレベルという聞き慣れない用語が出てくるが、これはハードウェア割り込みを応答性を求められる順番に優先度分けするものだ。より応答性が必要な優先度の高い割り込みを禁止している最中はそれより低い優先度の割り込みも禁止される。ややこしいが、ここでは優先度をつけた割り込み禁止の機構だと理解して問題ないだろう。

例えば、Intel PRO/Wireless 3945ABG の割り込みルーチンを見てみよう。wpi_attach 関数がデバイスドライバの初期化だ。この中で wpi_intr 関数が割り込みレベル IPL_NET で励起されるように設定されている。

```
// File: netbsd/src/sys/dev/pci/if_wpi.c
static void
wpi_attach(device_t parent __unused, device_t self, void *aux)
{
// --snip--
    sc->sc_ih = pci_intr_establish(sc->sc_pct, ih, IPL_NET, wpi_intr, sc);
// --snip--
    if ((error = wpi_alloc_rpool(sc)) != 0) {
// --snip--
}

static int
wpi_alloc_rpool(struct wpi_softc *sc)
{
// --snip--
    mutex_init(&ring->freelist_mtx, MUTEX_DEFAULT, IPL_NET);
    SLIST_INIT(&ring->freelist);
```

この wpi_intr 関数は割り込みコンテキストで実行されるにもかかわらず、中では以下のように mutex を使う。しかし先の wpi_alloc_rpool 関数の中でこの mutex は IPL_NET 割り込みレベルで初期化されているため、mutex_enter(&sc->rxq.freelist_mtx) と mutex_exit(&sc->rxq.freelist_mtx) の区間はアーキティクチャ側で IPL_NET に対応する割り込み禁止をかけた状態になるはずだ。そのためこの mutex を保持している最中は IPL_NET 割り込みによる受動的コンテキストスイッチが起きないことになる (図 24)。

```
// File: netbsd/src/sys/dev/pci/if_wpi.c
static int
wpi_intr(void *arg)
{
// --snip--
    if (r & WPI_RX_INTR)
        wpi_notif_intr(sc); // 中で wpi_alloc_rbuf 関数を呼ぶ
```

```
// --snip--
}

static struct wpi_rbuf *
wpi_alloc_rbuf(struct wpi_softc *sc)
{
    struct wpi_rbuf *rbuf;

    mutex_enter(&sc->rxq.freelist_mtx);
    rbuf = SLIST_FIRST(&sc->rxq.freelist);
    if (rbuf != NULL) {
        SLIST_REMOVE_HEAD(&sc->rxq.freelist, next);
        sc->rxq.nb_free_entries --;
    }
    mutex_exit(&sc->rxq.freelist_mtx);

    return rbuf;
}
```

POSIX の上でシグナルハンドラのようなものを使う際にもこのような特別な排他制御をコンパイラの利用者やライブラリ設計者が適切に設定するべきだ。例えばシグナルハンドラ中で実行可能な関数は制限されるべきだ。もちろん POSIX にその制限は明記されている。もしシグナルを受けた際にその限定された API 以外の関数を実行する場合には sigwait [*44] でシグナルを受け取る専用のスレッドをプロセス内で一つ作り、適切なスレッドにシグナル受信メッセージを配送すべきだ。

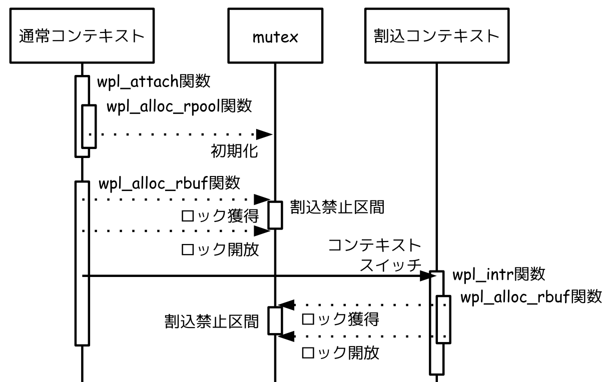

図 24: mutex_enter の ipl について

一般化すると、Haskell で書くコードであったとしても、そのコードが受動的コンテキストスイッチを受ける側なのか、それともハードウェアトリガーによって起動される側なのかを意識して設計する必要があるのだ。

コンパイラ利用者がこの制約を受け入れないかぎり、そのコンパイラは C 言語をスナッチするに足る能力を得ることはできない、というのが今の時点でのぼくの結論だった。

[*44] sigwait(2) http://netbsd.gw.com/cgi-bin/man-cgi?sigwait++NetBSD-current

0.8.3 再入可能の実現

コンパイラへの要求がぼんやりと見えてきた気がする。さぁ Ajhc のクリティカルリージョンを調べてみよう。ざっと調べる限りでは以下に大別されるようだった。これらをコンテキストローカルな配置にすることができないか、もしそれが不可能であれば排他してやる必要がありそうだ。

A. ランタイム
  1. グローバル変数として確保される gc_t saved_gc
  2. グローバル変数として確保される struct s_arena *arena
  3. GC スタック
  4. megablock
  5. struct s_arena の下に繋がれる struct s_cache

B. ミューテータ
  1. グローバル変数として確保される struct s_cache
  2. GC スタックを管理する gc 引数

グローバル変数に関してはコンテキスト毎の管理にできないか検討して、無理であれば mutex で排他するしかないだろう。megablock に関しても複数の利用者がいる場合であれば alloc/free 時にロックするしかない。残る問題は GC スタックだ。

思い出してみよう。GC スタックはミューテータによって管理されていて、GC ルートになるのだった。この GC スタックをコンテキスト間で共用することを考えよう。GC スタックはミューテータの gc 変数のスコープによって管理されていた。変数スコープはコンテキストの外からは知ることが難しい。そこで退避されているコンテキストの中からスタックポインタを取り出して、当該スタックの中でスマートポインタに該当しそうなものを GC ルートと見做すしかない。でもこれは Boehm GC [*45] とほとんど同じ処理をすることになる。もしくは別の案として、GC スタックという領域を C 言語のスタックとは別に確保して、GC スタックへのスマートポインタの追加と削除をグローバルロックで排他する……ナンセンスだ。

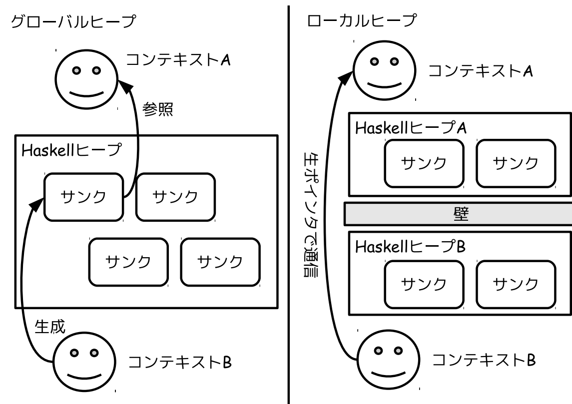

図 25: グローバルヒープとローカルヒープ

もっと高い場所からこの GC ルートと Haskell ヒープの問題を見渡してみよう。そこには二つの道が見える。 **グローバルヒープ** と**ローカルヒープ** だ (図 25)。グローバルヒープはこれまで議論してきた GC ルートの管理方式だ。すなわちコンテキストが多数あっても Haskell ヒープは唯一一つだけ用意する。そのため Haskell ヒープへの操作は場合によって排他制御する必要が生じる。もう一つ考えられるのが Haskell ヒープをコンテキスト毎に一つずつ持つ案だ。

[*45] A garbage collector for C and C++ `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

グローバルヒープのメリットは明確だ。コンテキスト間で同一のヒープを使っているため、排他処理さえ気をつければコンテキスト間のサンクの受け渡しがゼロコピーで実現できる。ほとんどの言語処理系はグローバルヒープを選択している。しかし、先に見たとおり jgc との相性は悪いようだ。Ajhc でグローバルヒープを使う場合には GC をほぼまるまる再設計する必要がありそうだ。

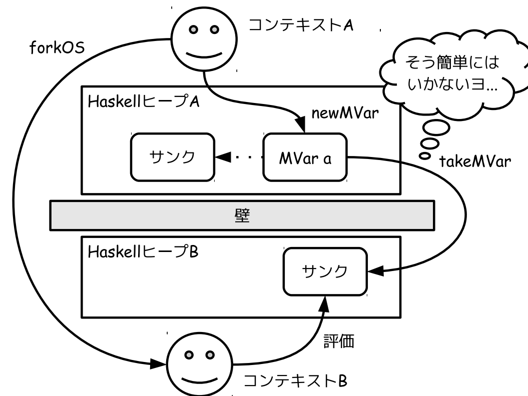

図 26: MVar に繋いだサンクは誰が評価すべきなのか？

ローカルヒープを採用した場合にはグローバルヒープとは異なりコンテキスト間でサンクを受け渡すのは難しくなる (図 26)。サンクを作る時点ではコンテキストの中のみで消費するのか、MVar のようなしくみを使って別のコンテキストに渡すために作るのか判別できない。MVar に繋いだ後にサンクを作ったコンテキストではない別のコンテキストから評価しようとするとややこしい事態になる。誰がこのサンクを評価すべきなのだろうか？

ローカルヒープにはさまざまな問題が考えられるが、ぼくは Ajhc の再入可能を実現するにあたってローカルヒープを採用することにした。理由はいくつかあるが最も大きな理由は Haskell 言語が状態の共有を嫌う言語だということだ。グローバルヒープを採用した場合、状態の共有は実現しやすいけれど並列実行に気をつかう必要がある。

ローカルヒープを採用した場合、ロックなしに並列実行を実現できる。並列 GC もおまけでついてくる。メインコンテキストが GC 実行中でも割り込みを受けられる。さらにコンテキストとゴミが紐づいているために、自コンテキストで出したゴミは自コンテキストの GC ペナルティになる。他人に迷惑をかけないわけだ。もっと考えると、kernel の中はイベントドリブンが主だ。ということは GC する前にコンテキストそのものが消滅する可能性も高いかもしれない。コンテキストが消滅したら紐づいていた Haskell ヒープをまるごと回収してしまえば良いのだ。ヒープ回収の時点になってしまえばマーキングする必要もない。

最初のコンテキストの定義を思い出してみよう。コンテキストとはレジスタとスタックのセットなのだった。ローカルヒープはこのコンテキストのセットに二つの要素 **GC スタック**と **Haskell ヒープ** を追加することに他ならない。いわば Haskell 文脈の中においてだけ C 言語の ABI を拡張していることになる。こう考えればなんともシンプルな発想だ。

方針は決まった。GC スタックと Haskell ヒープをコンテキスト毎に持つように変更しよう。まず考えるべきなのが管理構造体だ。Haskell ヒープは arena グローバル変数で管理されていた。この arena グローバル変数をコンテキストローカルに持つには gc 変数を同じように関数の引数で持ちまわるのが一番簡単だ。POSIX の外ではスレッドローカルストレージなど使えないからだ。一見して gc 変数と arena 変数の二つをミューテータ内の全関数の引数に追加するのは無駄に思えた。arena 変数の 1 メンバーとして gc 変数を持てば良いのではないか？ しかし調査をすすめてみるとそれほど簡単な問題ではないことがわかった。arena 変数の方は簡単だ。常にコンテキストに対して一つだけ存在し、そのアドレスは変化しない。しかし gc 変数はミューテータ内のローカルスコープによってその具体的なアドレスは変動している。スコープや関数を抜ければ元の状態が復元されなければならない。このような復元をコンパイラパイプライン側で実現するのは不可能ではないが、煩雑だ。また、当然この gc 変数の復元にはコストがかかる。x86 のようにレジスタが少ないアーキ

ティクチャはこれから淘汰されることを祈って、ぼくは gc と arena 両方の引数をミューテータに追加することにした。

arena 変数の構造体を変更しよう (図 27)。いままでグローバル変数で定義されていた管理構造体はできるだけ arena 変数の中に埋め込むべきだ。link は arena 変数を管理するリスト用メンバーだ。arena 変数は used_arenas か free_arenas どちらかのグローバルリストに繋がれる。全コンテキストにメッセージを送りたければ used_arenas を辿れば良いし、新しくコンテキストに arena 変数を割り当てたければ free_arenas から取り出せば良いわけだ。gc_stack_base メンバーはそのままこれまでの gc_stack_base グローバル変数の意味そのままだ。GC スタックの初期ポインタを保存している。array_caches と array_caches_atomic メンバーもこれまでのグローバル変数をそのまま arena 変数に埋め込んだものだ。force_gc_next_s_alloc というメンバーは "次回の s_alloc 時に強制的に GC をかける" というフラグだ。なぜこんなものが必要になるのかというと、hs_perform_gc という C 言語関数が Haskell 標準で決まっているためだ。この関数は Ajhc 側のコンテキストを全く知ることなく強制 GC を指示する。そのため、今回の実装ではランタイムをグローバルロックした後で、このフラグを立てる必要がある[*46] 。そして Haskell ヒープから領域を確保する s_alloc 関数の頭でこのフラグの判定して強制 GC を実行すればいい[*47] 。

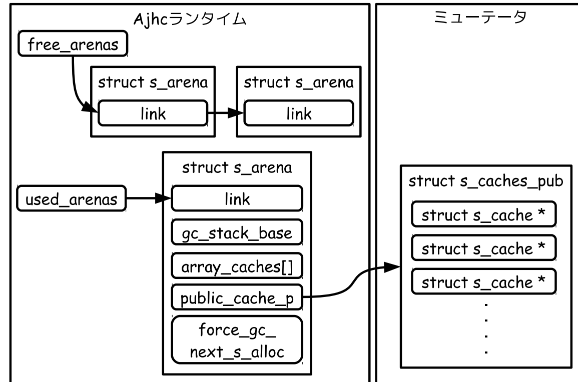

図 27: s_arena 構造体と GC スタックの構造と配置

arena 変数にはもう一つ public_caches_p というメンバーが追加されている。このメンバーはミューテータ側で宣言される cache 管理構造体へのポインタだ。s_arena 構造体の中身はミューテータ側から隠されているが、このミューテータが宣言した cache だけは見えてほしい。そのためにこんなややこしいしくみがどうしても必要だった。ミューテータは以下のようなコードで、コンテキスト起動時に public_caches_p メンバーを初期化する。

```
// File: main_code.c (ミューテータの例)
struct s_caches_pub {
    struct s_cache *cCJhc_Prim_Prim_$x3a;
    struct s_cache *cFR$__fJhc_Basics_$pp;
    struct s_cache *cCJhc_Type_Ptr_Ptr;
};
// --snip--
void
jhc_hs_init(gc_t gc,arena_t arena)
{
```

*46 https://github.com/ajhc/ajhc/blob/v0.8.0.7/rts/rts/gc_jgc.c#L826
*47 https://github.com/ajhc/ajhc/blob/v0.8.0.7/rts/rts/gc_jgc.c#L573

```
        alloc_public_caches(arena,sizeof(struct s_caches_pub));
        find_cache(&public_caches(arena)->cCJhc_Prim_Prim_$x3a,arena,
                   TO_BLOCKS(sizeof(struct sCJhc_Prim_Prim_$x3a)),2);
        find_cache(&public_caches(arena)->cFR$__fJhc_Basics_$pp,arena,
                   TO_BLOCKS(sizeof(struct sFR$__fJhc_Basics_$pp)),3);
        find_cache(&public_caches(arena)->cCJhc_Type_Ptr_Ptr,arena,
                   TO_BLOCKS(sizeof(struct sCJhc_Type_Ptr_Ptr)),0);
}

// File: ajhc/rts/rts/gc_jgc.c
void
alloc_public_caches(arena_t arena, size_t size) {
        if (arena->public_caches_p == NULL) {
                arena->public_caches_p = malloc(size);
        }
}
```

ちょっとややこしくなってきた。気分をかえて、ローカルヒープを採用した Ajhc でのコンテキストの一生を見てみよう。

Ajhc におけるコンテキストは C 言語からの関数呼び出しによって始まる。C 言語から Haskell コードの main 関数を呼び出す際には_amain 関数を C 言語から呼び出すのだった。

```
// File: main_code.c (ミューテータ)
void
_amain(void)
{
        arena_t arena;
        gc_t gc;
        gc = NULL;
        arena = NULL;
        jhc_alloc_init(&gc,&arena);
        jhc_hs_init(gc,arena);
        b__main(gc,arena);
        jhc_alloc_fini(gc,arena);
}
```

C 言語文脈にはなかった gc と arena という引数を二つの言語の界面である export 関数で初期化するように変更した (図 28)。jhc_alloc_init 関数では gc と arena をプールから確保して初期化する。その後 jhc_hs_init 関数を使ってミューテータで定義されている cache を初期化する。export 関数から呼び出される b__main 関数から先は gc と arena を引数によって引き回す Haskell コンテキストの世界だ。b__main 関数が完了したら、この gc と arena は解放される。もちろん free してしまうのでなく、ランタイム側内部でプールして次回使用時に使いまわす。

このような arena 引数を持ち回るしくみをミューテータに追加するには、もちろん Ajhc のコンパイルパイプラインに修正を加える必要がある。この arena 引数を追加する変更は grin=>C への変換に細工をすれば実現できる (図 29)。

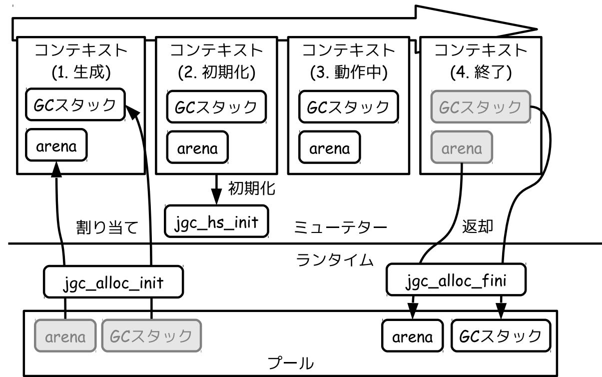

図 28: gc と arena 変数とコンテキストの一生

compileGrin 関数は Grin 型を C 言語のソースコードが入った ByteString 型に変換する。この関数の頭で C 言語ソースコードの構造が ans で示されている。Grin 型で定義されている内容を ans につめかえるのがこの変換の役目だ。ans の個々の要素に関して、修正箇所を見てみよう [*48] 。

最初に変更するべきは jgcs エントリだ。GC タイプとして jgc が選択された場合にのみ s_caches_pub 構造体にミューテータ側で定義するべきキャッシュを列挙することにしよう[*49] 。さらに jhc_hs_init 関数の宣言に arena 引数を加える。この header と body は ans のエントリだった。つまり C 言語の関数定義を generateC に通すとヘッダと本体が得られるのだ[*50] 。Haskell 由来の C 言語関数全てに第一引数に gc を第二引数に arena を取ることにしよう。jhc_hs_init 関数の中身は icaches で定義されている [*51] 。ミューテータ側で定義された cache は arena の下の s_caches_pub メンバーに配置することにしたので、そのように書き換えた。もし当該の Haskell 関数が FFI で export されていたら gc と arena の新規割り当てを行なうようにした[*52] 。

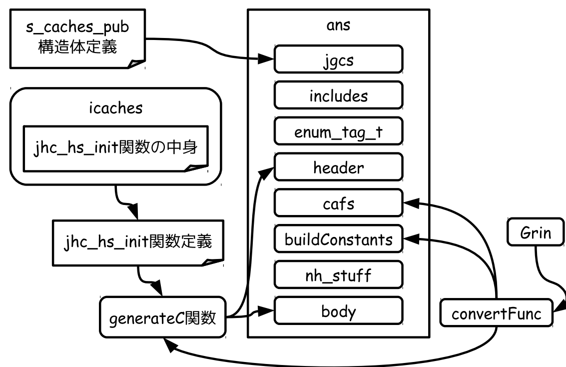

図 29: grin → C 変換

また場合によっては Haskell コードからランタイムの関数を import して使いたい時がある。この時にも gc と arena を渡す必要がある。そこで jhc_context という特殊な import 種別を作った [*53] [*54] [*55] 。

[*48] https://github.com/ajhc/ajhc/blob/v0.8.0.7/src/C/FromGrin2.hs#L129
[*49] https://github.com/ajhc/ajhc/blob/v0.8.0.7/src/C/FromGrin2.hs#L150
[*50] https://github.com/ajhc/ajhc/blob/v0.8.0.7/src/C/FromGrin2.hs#L157
[*51] https://github.com/ajhc/ajhc/blob/v0.8.0.7/src/C/FromGrin2.hs#L158
[*52] https://github.com/ajhc/ajhc/blob/v0.8.0.7/src/C/FromGrin2.hs#L214
[*53] https://github.com/ajhc/ajhc/blob/v0.8.0.7/src/C/Prims.hs#L22
[*54] https://github.com/ajhc/ajhc/blob/v0.8.0.7/src/FrontEnd/ParseUtils.hs#L410
[*55] https://github.com/ajhc/ajhc/blob/v0.8.0.7/src/C/FromGrin2.hs#L596

jhc_context 付きの import のおかげで、gc と arena 引数付きで C 言語の関数を呼び出せるようになった (図 30)。export にも同様のしくみが必要になりそうな気もしたが、いまのところは明確な用途が見あたらないので import 側のみの修正にとどめた。

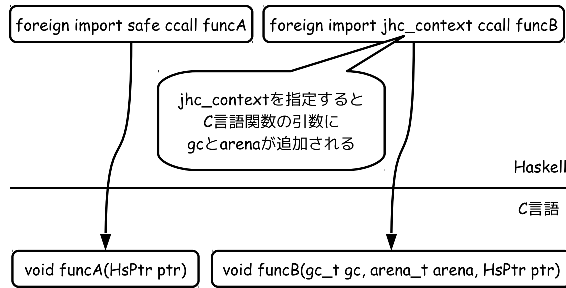

図 30: C 言語の関数呼び出しに gc と arena を追加する

例えばランタイムの gc_new_foreignptr 関数は、これまでは Haskell 側から gc 引数がわたってこないので、saved_gc というグローバル変数経由で GC スタックへのポインタを受け取っていた。jhc_context を使えば、Haskell 側から直に gc と arena 引数を任意の C 言語関数に渡すことができるようになった[56] [57] 。

やれやれ。これであらかた GC スタックと arena 変数をコンテキストローカルにできたはずだ。あとは残ったクリティカルリージョンを mutex で保護するだけだ。様々なロック機構に対応するため、図 31 のように 3 種類の並列実行に大別してオプションを作った。将来はもっと増えるかもしれない。

コンパイルオプション	-fnothread	-fpthread	-fcustomthread
ロックの初期化	ダミー実装	pthread実装	利用者が実装
ロックの獲得	ダミー実装	pthread実装	利用者が実装
ロックの開放	ダミー実装	pthread実装	利用者が実装
forkOS	単純関数実行	pthread実装	利用者が実装

図 31: コンパイルフラグによる mutex の実装差異

また、ランタイムのグローバルロックのために mutex をグローバル変数で一つだけ定義した。ロックを細分化した方がいいかもしれないが、今は簡単な実装にしておきたい。

0.8.4 スレッドの実装

これでコンテキストとコンテキストの分離が実現できた。ということは割り込みコンテキストのような受動的なコンテキスト切り換えが可能になったわけだ。もちろん “jhc_mutex_t を適切に設計した場合は” というただし書きはついてしまうがこれは設計開始当初に受け入れた制約だ。

このままでもシングルコアのマイコンを制御するには当面十分だ。しかしどうせ再入可能にできたのであれば、POSIX

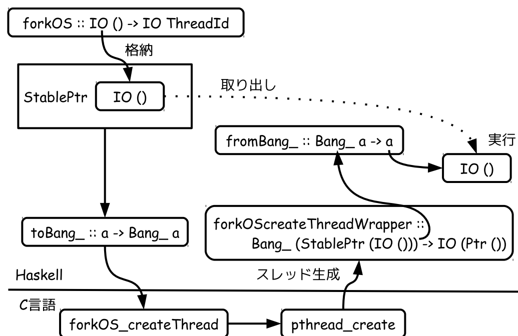

図 32: forkOS を pthread で実装

[56] https://github.com/ajhc/ajhc/blob/v0.8.0.7/lib/jhc/Jhc/ForeignPtr.hs#L64
[57] https://github.com/ajhc/ajhc/blob/v0.8.0.7/rts/rts/gc_jgc.c#L781

上でのスレッドもサポートしておきたい。継続的インテグレーション [58] することを考えれば、マイコンよりも POSIX 上でテスト可能にした方がいい。またマルチコア環境でしか起きない不具合も検出できるかもしれない。

既にランタイムの mutex は pthread を使って実装済みなので、状態共有をしない並列実行であれば forkOS を pthread_create を使って作れば十分だろう。まずランタイム側で pthread_create を使ったスレッド生成の関数を作る。Haskell 側は GHC の forkOS 実装を真似て StablePtr 経由で IO () を渡すようにした (図 32)。どうも jhc では StablePtr は直接 FFI に渡すことはできず、Bang_という型で包む必要があるようだった。

この実装で動きそうに思えるが、なぜかコンパイルエラーだ……

```
ajhc: Grin.FromE.compile'.ce in function: FE@.CCall.forkOScreateThreadWrapper
can't grok expression: <fromBang_ x108042543::IO ()> x62470112
```

エラーメッセージを読むかぎりではコンパイルパイプラインの中の Grin.FromE.compile'.ce 関数は fromBang_型を受け取ることを予期していないようだ。FromE での評価には ce 関数と cc 関数の二つの形式がある。今回のエラーが起きている ce 関数は正格評価で、cc 関数は遅延評価だ。fromBang_型を受け取るのは cc 関数のみで、ce 関数には当該のパターンマッチが存在しない。cc 関数でのパターンマッチの前に fromBang_ プリミティブを削除する関数をかませてしまうことにした [59] 。おそらく Bang_型は正格にするためのプリミティブだと思われるので ce 関数では無視してよさそうだが、多少不安だ。fromBang_プリミティブについては後付けでもう少し調査が必要そうだ。

さて、いくつかテストを書いてみよう。最初のテストは三つのスレッドから文字を吐き出すプログラムだ。これはさすがにうまくいった。

```
import Control.Monad
import Control.Concurrent

main :: IO ()
main = do
  l <- putStrLn "Type some string and enter." >> getLine
  forkOS $ (forever $ putChar '*')
  forkOS $ (forever $ putStr l)
  forever $ putChar '.'
```

次にマイコンと同じような動作をするテストを書いてみた (図 33)。つまり割り込みコンテキストの動作を pthread を使ってエミュレーションするのだ。まず C 言語側で Haskell コードに隠れてスレッドを生成しておく。この減算スレッドは Haskell で書かれた timingDelayDecrement 関数を定期的に呼び出して、TimingDelay グローバル変数が 0 になるまで減算する。Haskell 側の myDelay 関数は待ち時間が経過するのをビジーループで待ちあわせる。

このテスト実行すると、減算スレッドの sleep が解除されると同時に myDelay 関数の待ち合わせが解除された。これでマイコンの割り込みをスナッチする自信がついた。

[58] 執筆時点では Travis CI を使っている https://travis-ci.org/ajhc/ajhc
[59] https://github.com/ajhc/ajhc/blob/v0.8.0.7/src/Grin/FromE.hs#L365

0.8.5 割り込みコンテキストのスナッチ

今回のスケッチで作った再入可能の機能を使って、Cortex-M4 マイコンの割り込みハンドラを Haskell でスナッチしてみよう。

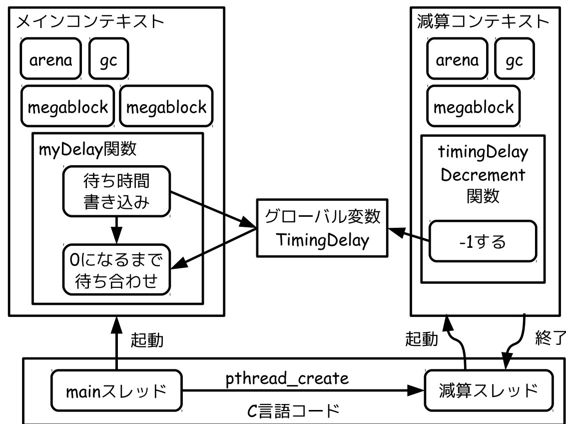

図 33: 割り込みハンドラの動作を pthread でエミュレーション

図 34 は Haskell でスナッチしたクロック割り込みによる delay の実装だ。まずマイコンの電源が入ると C 言語の main 関数が Haskell のメインコンテキストを起動する。これまで出てきたように_amain 関数がメインコンテキストの入口だった。この時、Ajhc ランタイムは arena と gc をプールからメインコンテキストに割り当てる。メインコンテキストは何度かコンストラクタを呼び出すうちに、おそらくは、二つの megablock を手に入れるだろう。この megablock 達の中にメインコンテキストが作ったサンクが納めれれる。メインコンテキストは電源 OFF までの間終了せず、ずっと滞留することになる。

一方 C 言語の main 関数がクロックの初期化をすると、メインコンテキストとは全く関係のないタイミングでクロック割り込みが発生するようになる。このクロック割り込みが発生するとクロック割り込みベクタに実行が移り、クロック割り込みベクタは Haskell で記述されている timingDelayDecrement 関数を起動する。この時、メインコンテキストとは別の Haskell コンテキストを起動したので、メインコンテキストとは別に arena と gc がランタイムによって割り当てられる。クロック割り込みコンテキストが終了すると、arena,gc,megablock はランタイムによって回収されて次回誰かが使うまでプールに保管する。もちろん次回このプールから arena 達を確保するすのは次のクロック割り込みだろうけれど。

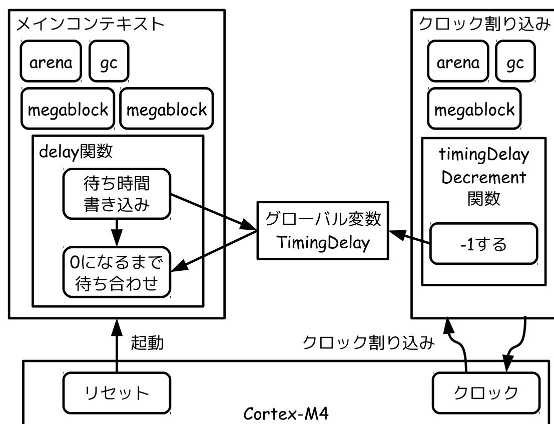

図 34: 割り込みハンドラを Haskell でスナッチ

こうしてコンテキストの分離が済んでしまえば、Haskell での割り込み実装は簡単だ。C 言語側でグローバル変数 TimingDelay を用意して、このグローバル変数を Ptr 型によってアクセスすればコンテキスト間通信ができる。

さて、長い今回のスケッチで結局 Cortex-M4 マイコン上で動作するプログラム*60 のどこまで Haskell でスナッチすることができたのだろうか？ 実行バイナリ内に含まれるシンボルの中で、

*60 https://github.com/ajhc/demo-cortex-m3/tree/master/stm32f3-discovery

Ajhc ランタイム、ランタイム用ダミー関数、malloc、Haskell コード由来ではないものを列挙してみよう。

- 例外/割り込みハンドラ
- data/bss 初期化コード (アセンブラ)
- Haskell スレッド間通信用グローバル変数
- クロックの初期化コード
- GPIO の初期化コード
- LED の GPIO 初期化コード

割り込みハンドラは使っていないものは全部 abort だ。data と bss の初期化は C 言語を使う前にアセンブラで行なう必要があるため Haskell 化は不可能。Haskell スレッド間通信用グローバル変数については MVar のようなスレッド間状態共有のしくみを構築すれば解決するだろう。もちろんそれは非常に困難な課題だけれど。

上記以外の初期化コードは全てメモリへの読み書きによって作られている。ということは Ptr 型を使えば Haskell でスナッチできるということだ。つまり Cortex-M4 マイコン上で動作するソフトウェアに関しては、限界まで Haskell でスナッチする見込みがたったのだ!

0.9 突然の連絡

何度目かの Ajhc リリースノートを書いていると、携帯のアラームが鳴った。

```
From J, at 2013年05月25日(土):
会社をやめることにしたよ。きっと jhc の開発に戻れると思う。
いままでメーリングリストで返事ができなくてすまなかった。
それもみんな会社が従業員に個人プロジェクトを持つことを許さなかったからだ。
オレはもう一度 jhc のようなオープンソース活動をしてみたい。だからやめた。
```

J、今まで疑っててごめん。本当はずっと気にかけていてくれたんだね。ぼくは最初から一人ぼっちじゃなかったんだ。

ふいにあの娘の声が聞こえたような気がする。

「ああ、わかっているさ。次のスケッチはもう見えている」

ぼくは思わず独り言をつぶやいた。

0.10 これからのこと

結局この一連のスケッチで勝ち取ったものはなんだったのだろう？ 一見なにも変化していないように思える。なにせシングルコアの小さなマイコンで動くソフトウェアを型付けできたにすぎない。しかしとにかく実際に使われているプログラムを型によって少しずつ再設計できる、ということは確かめられたようだ。これからの課題も山積みだが、何が必要なのかは今なら理解できる。

- MVar のような型を経由したスレッド間状態共有の実現
- ユーザー空間スレッドの実装
- Haskell Platform 相当のライブラリを移植
- cabal のようなパッケージ管理システム
- 大規模な Haskell プログラムのコンパイル実績
- さらなる応用例の提案

おぼろげながら見える。スケッチの糸が遠くデザインへと繋がっている。ふと目をあげると、まだ見慣れぬ優しい **アラフラの海** [*61] が広がっていた。

図 35: ”ビンタンビール”

0.11　参考文献

- 図 35: Sunset with Bintang - Copyright (C) 2010 Matteo Catanese All Rights Reserved. [*62] [*63]
- The NetBSD Project `http://www.netbsd.org/`
- Metasepi Project `http://metasepi.org/`
- Metasepi Project の近傍 `http://metasepi.org/map.html`
- Five-Year Plan for the Metasepi Union `http://metasepi.org/doc/20130508_5year_plan.pdf`
- Jhc Haskell Compiler `http://repetae.net/computer/jhc/`
- Ajhc - Haskell everywhere `http://ajhc.metasepi.org/`
- 「ファウンデーション」、アイザック・アシモフ著、岡部宏之訳、ハヤカワ文庫

[*61] デザイン Arafura `http://metasepi.org/posts/2013-01-09-design_arafura.html`

[*62] `http://www.flickr.com/photos/matteocatanese/5166690955/`

[*63] Licensed under a Creative Commons Attribution 2.0 Generic License. `http://creativecommons.org/licenses/by/2.0/`