
簡約!?
ハカ娘
ROCK!

目次

第0章

まえがき

関数型イカ娘とは!?

Q. 関数型イカ娘って何ですか?
A. いい質問ですね！
Q. 六冊目なんて、こんなの絶対おかしいよ！
B. 僕達の役目はね。関数型言語とイカ娘の設計、実装、原理、応用に関する妄想を抜き取って、文章に変えることなのさ
Q. 今回の表紙はえらいロックですね！
A. 関数型プログラミングはロックでゲソ!

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、さらにさらにコミックマーケット 83 での「簡約!? λ カ娘 4」、に続く、もーさらにさらにコミックマーケット 84 での「簡約!? λ カ娘 Go!」、に続く、六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

λ力娘探索 2?

— @ark_golgo

1.1 プロローグ

「……私の 1 目勝ちじゃなイカ？」
「そのようじゃな。ヨセで細かくなったとは思っておったが、逆転には至らなかったようじゃ。」
「3 子で初勝利でゲソ！　はぁ。ふらふらでゲソ！」
「この半年で置石 2 つ分くらい強くなったようじゃの。いやはやたまげたわい。お主と打ち始めた当初は 4 子でも楽勝だったがの。」
　λカ娘は　C84 のころからある老人と囲碁を打つようになった。当初は 4 子のハンデ (段級位差にして 4 段級差に相当) をつけてもらってもまるで勝てなかったが、最近は 3 子でもいい勝負が出来るようになっていた。そして、今回 3 子で初勝利を収めた。
「やっぱり囲碁ソフトと打ちまくったのが良かったんじゃなイカ？」
「囲碁ソフト？　はて、儂が昔試してみた囲碁ソフトはあまりに弱すぎて話にならなかったが……。最近はお主の相手になるくらい強いソフトが出ているのかね? 」
「そうでゲソ。ここ数年で囲碁ソフトは急激に進歩したでゲソ。 最近は私より強くて、トッププロに 4 子で勝ったりしているでゲソ！」
「ほう、そうなのか。面白そうじゃの。儂も一つ買ってみるか。」
　さて、最近強くなったという囲碁ソフトについて、ここでλカ娘に解説してもらうことにしよう。旧来の囲碁ソフトとの比較を交えながら、強くなった理由を語ってもらおう。

1.2 囲碁のルールおさらい

「囲碁のルールを説明するでゲソ。囲碁のルールは細かいものまで説明していると結構大変なので、今回の記事に関係ある部分だけ説明するでゲソ！ 以下の 4 つを把握しておけば十分でゲソ。」

1. 黒白交互に盤の交点に打つ。1 手目は黒から (図 1.1)。[*1]

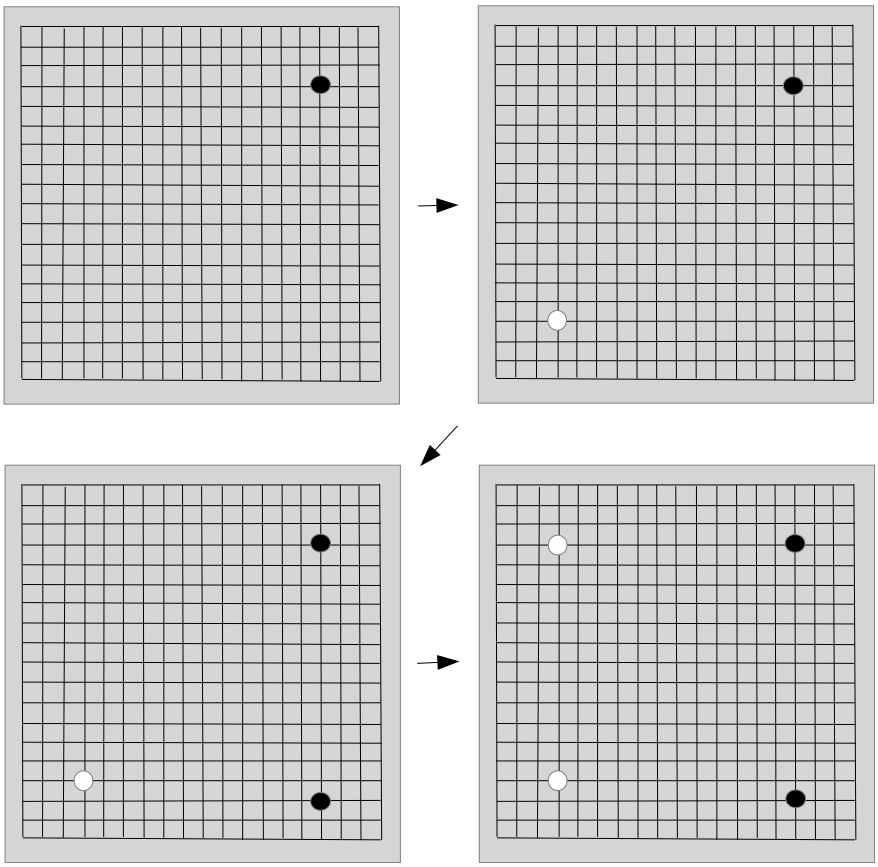

図 1.1: 交互に打つ

[*1] ただし置き碁 (ハンデ戦) の場合は黒があらかじめ石を何個か置き、その後白から打ち始める。λカ娘が勝った対局は、黒のλカ娘が先に 3 個石を置いていた。

2. 相手の石の四方を囲むと取れる (図 1.2)。

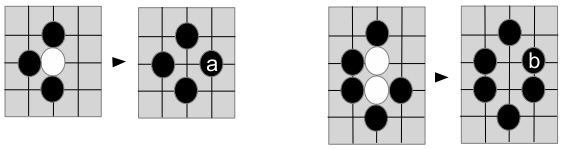

図 1.2: 左図の状態から、a で示した石を黒が打つと、真ん中の白石は上下左右囲まれるので取られる。b のように、複数石あっても同様である。

3. 自分から石を取られるような手は打てない (自殺手の禁止) が、それ以外は空いている場所のほぼどこでも打てる。そのため、自由度が非常に大きい。ただし一見自殺手に見えても、先に周りの相手の石を取れる場合は打てる (図 1.3)(『コウ』という概念があって、厳密には打てない場所もあるが、今回は説明しない)。

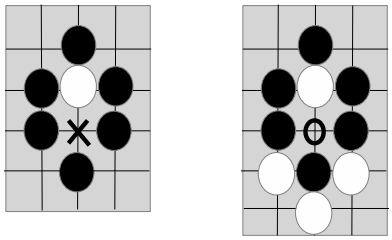

図 1.3: 白は×には打てないが○には打てる

4. 相手より大きな陣地を囲えば勝ち (図 1.4)。陣地とはそれぞれの石または盤端で四方を囲まれた石の無い領域のこと。図は数えやすいように 9 路盤 (練習用の小さな盤) で示している。黒と白の丸は実際の石で、四角はそれぞれの陣地を示す。四角の所には実際には何も置かれていない。この場合、黒の陣地は 36 目 (目(もく)とは地の単位)、白の陣地は 23 目で黒の 13 目勝ち。λカ娘が勝った対局は、1 目だけλカ娘の陣地が大きかったということ。

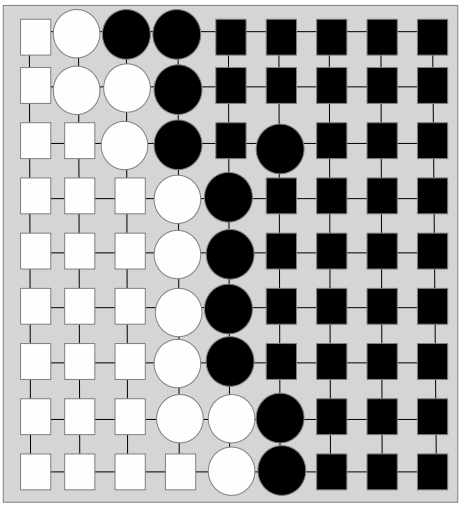

図 1.4: ■が黒の陣地で 36 目。□が白の陣地で 23 目。黒 13 目勝ち

1.3 AND-OR 探索

「ゲームの AI において人間の『読み』に相当する機能は基本的に『AND-OR 木』というものがベースになっているでゲソ。対戦型のゲーム (厳密に言うと『二人零和有限確定完全情報ゲーム』と呼ばれる種類のもの) ならばどんなものを扱ってもいいけど、ここでは盤面が小さな Tic-Tac-Toe で説明するでゲソ。」
「聞いたことないゲームじゃな。」
「日本では三目並べとか○×とか呼ばれているでゲソ。とりあえず図 1.5 を見るでゲソ。爺さんもやったことあるんじゃなイカ？」
「ああ、これか。これなら子供の頃やったことあるわい。」
「図 1.5 は最大あと 4 手で決着する、Tic-Tac-Toe のある局面 (a) からの変化を網羅したものでゲソ。(a) での手番は×でゲソ。×は次にどこに打てるでゲソか ?」
「上、左、左下、下の 4 マスじゃな。」
「その通りでゲソ！ その 4 つの可能性があるでゲソ。人間がこのゲームをする時には、それぞれの箇所に打ったらその後何が起こるか考えるんじゃなイカ？ 例えば、上のマスに打ったとするでゲソ。そうすると、局面は (b) のようになるでゲソ (上の局面から下の局面への線は、この一手打ったことによる変化を示しているでゲソ)。

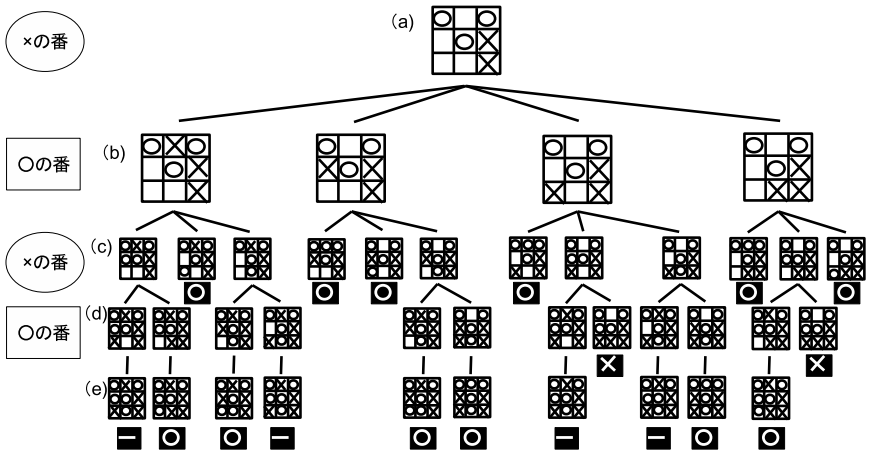

図 1.5: Tic-Tac-Toe の変化を網羅したゲーム木。ノードの下にあるマークは、どちらが勝ったかを表す。「-」は引き分けを表す。

局面 (b) では相手、○の手番でゲソ。この局面 (b) ではまだ決着がついていないのでゲームを続行するでゲソ。○は左、左下、下の三か所に打てるでゲソ。
○がもし左のマスに打ったとしたら……次は局面 (c) になるでゲソ。×がその次に左下に打つと局面 (d) に、そうすると次に○は下にしか打てなくて、下に打つと局面 (e) になって、引き分けになるでゲソ。これが一つの可能性じゃなイカ！
もちろん、可能性を一つ考えるだけではダメでゲソ。例えば、局面 (b) で○は左ではなく左下に打つこともできるでゲソ。そうすると……○の勝ちでゲソ!つまり私の負けじゃなイカ！　局面 (b) は○の手番だから○の好きな場所に打つことが出来るので、当然一番○にとって良い、つまり×にとって一番悪い手、『左下』を打つでゲソ！　この可能性を見逃して最初の局面 (a) で×を上に打ってしまうと、次に○に左下を打たれて負けてしまうでゲソ。
だから、局面 (a) では上、左、左下、下のすべての可能性を、その次の局面 (b) でも左、左下、下のすべての可能性を……という風に、すべての可能性を網羅したものが図 1.5 でゲソ！　」
「なるほどのう。でも、これでは全て列挙してみただけで、勝ちか負けか、どこに打てばいいのか分からんのう。」
「そこで『AND-OR 木』でゲソ！　」
「この網羅図と論理式の AND-OR が関係するのか。難しいのう。」
「実際はそんなに難しくないでゲソ！　図 1.6 を見るでゲソ！　これは、さっきの網羅図に対応する『AND-OR 木』でゲソ！

AND-OR 木を作る目的は、『自分が勝てる手』を探すことでゲソ (見つかったら当然それを打ちたいからでゲソ)。単純化すると、ゲームには『双方最善を尽くした時自分が勝てる局面』と『双方最善を尽くした時自分が負ける局面』しかないので、これを仮に『True(真)』と『False(偽)』で表すことにするでゲソ。引き分けがあるゲームについては、引き分けをどうするかは難しい問題でゲソが、今回は引き分けは自分の勝ち True とするでゲソ！
図 1.6 の場合は、自分が×だとして、×が勝てる局面　=True にたどり着けるような手を探すでゲソ。自分の手番の時 X は、次の一手で True になるような局面が一つでもあれば、他の手の結果がどう

なろうと関係なく、True になる手を選べるので、X も True になるでゲソ。どの手を選んでも False になる場合は X も False になるでゲソ。これは要するに次の局面の集合に対する『OR』演算と同じでゲソ！

相手の手番の時 Y は、逆に、次の一手で False になるような局面が一つでもあれば、他の手の結果がどうなろうと関係なく、False になる手を選んでくるので、Y も False になるでゲソ。どの手を選んでも True になる場合は Y も True になるでゲソ。これは要するに次の局面の集合に対する『AND』演算と同じでゲソ！

そうやって、図 1.7 のように、下から順に、つまり確定しているところから順に、各局面の勝敗を判定していくことが出来るでゲソ！　こうすると、今の局面で勝ちか負けか分かる上、勝ち (True) の場合は子ノードから True になっている奴選ぶのが最善手であることが分かるでゲソ！　自分が負け (False) の場合は子ノードはすべて False なのでどこ打っても負けになるでゲソ…」

「なるほどのう。ところで、全く同じ局面が何度も登場しているようじゃが、それを全て調べるのは無駄じゃないのかね？　例えば、引き分けになっている局面は全て同じじゃな。まあこれは AND-OR 木の末端じゃから、調べなおしても大した負荷にはならんかもしれんが、探索の途中にも同じ局面があるのう。これはかなりの負荷になる場合もあるのではないかね？」

「おっしゃる通りでゲソ！　実は、いくつかのゲームではゲーム木中で全く同じ局面に到達する場合があるでゲソ。そのような場合、何度も探索し直すのは無駄なコストがかかるので、『置換表』というものに一度探索した局面とその結果を記録しておくのが普通でゲソ。」

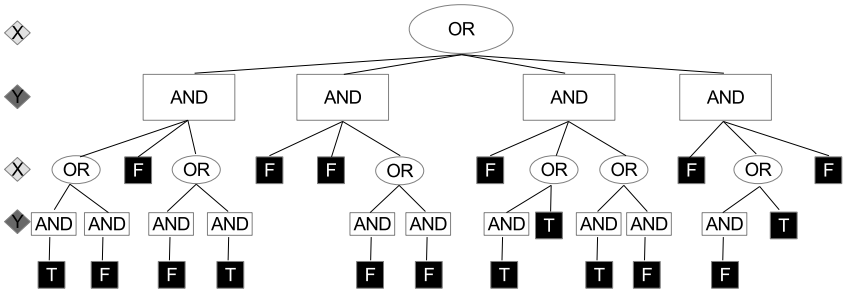

図 1.6: 図 1.5 のゲーム木を「AND-OR 木」で表したもの

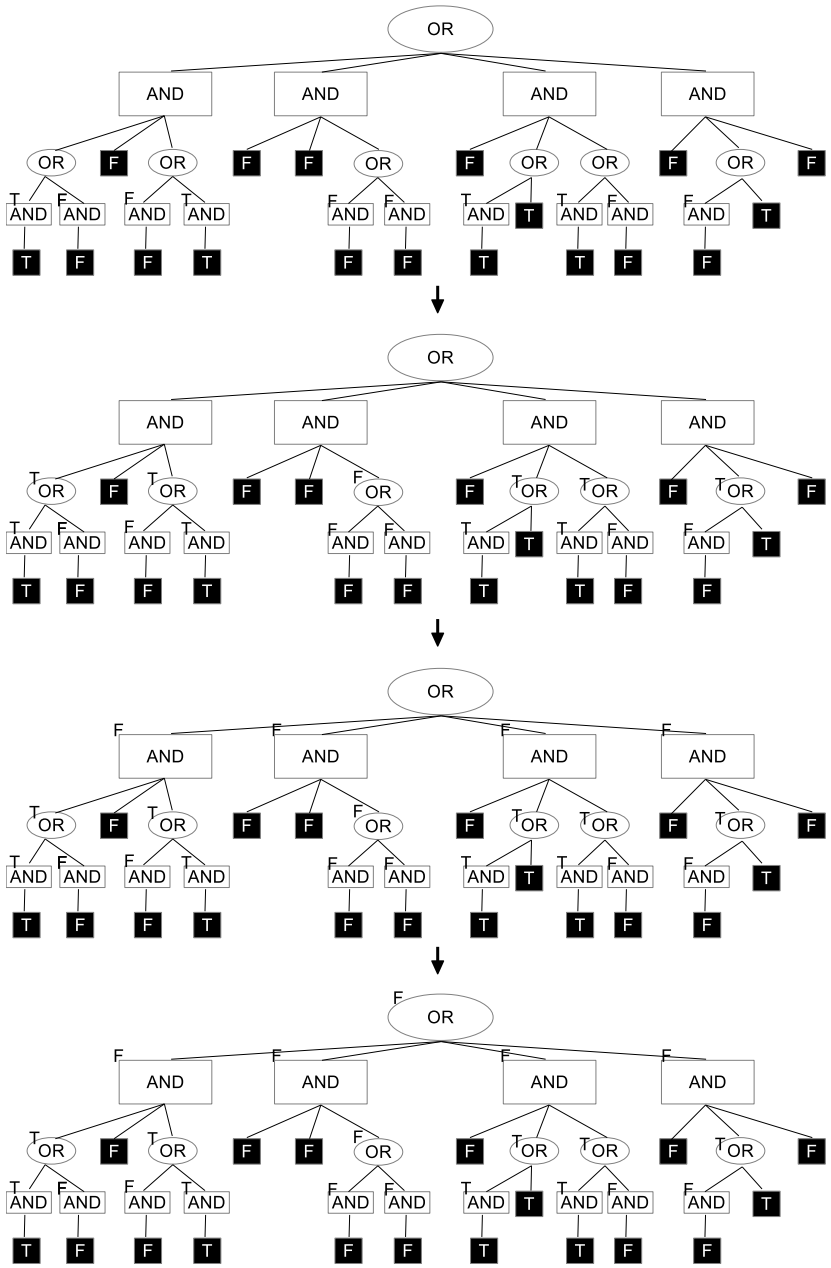

図 1.7: AND-OR 木のルート局面の勝敗が決まる様子

1.4 Min-Max 探索

「上記の AND-OR 探索では、ゲームの決着がつくまで探索できることを想定していたでゲソ。でも、現実に楽しまれているゲームは、選択肢の幅が広すぎて、とてもゲームの決着がつくまで探索は出来ないことが多いでゲソ。あるいは、例えばリバーシならスコアは-64〜64 の整数になるでゲソが、勝つか負けるかだけではなく決着時に出来るだけ大差で勝ちたい/負けるにしても小差にしたいという場合もあるんじゃなイカ？ そのような場合は、『評価関数』と『Min-Max 探索』を組み合わせて使うでゲソ。順に説明するでゲソ！

評価関数はゲームの途中でどちらがどれだけ有利かを計算する関数で、自分の手番が有利なほど大きな値 (通常は正の値) を、相手の手番が有利なほど小さな値 (通常は負の値) を返すでゲソよ。例えば、絶対負け = −10000、絶対勝ち=10000、完全に互角=0、やや有利=1000 のようになるでゲソ。何せゲームの途中で判断するので、評価関数を作るのはなかなかに難しく、それだけで本何冊も書けそうでゲソ。でもここでは、人間のプレイヤーがゲームの途中で『勝ちそう！ 』とか『負けそう… 』とかを判断しているのをプログラムにやらせるというイメージだけあればいいでゲソ！ これを、AND-OR 木で最後まで読み切った時の True/False だった部分の代わりに使うでゲソ。

そして、さっきは True と False だったので、AND-OR で良かったでゲソが、評価関数を使うともう AND-OR では計算できないでゲソ。そこで Min-Max でゲソ！ Min-Max 探索は、先ほどの『OR ノード』が『Max ノード』に、『AND ノード』が『Min ノード』になったものでゲソ。

自分の手番の時 X は、次の一手で好きな手を選べるので、次の局面の中から出来るだけ大きな評価値を選んだ時の評価値になるでゲソ。これは要するに次の局面の集合に対する『Max』演算と同じでゲソ！

相手の手番の時 Y は、逆に、次の一手で相手が好きな手を選べるので、次の局面の中から出来るだけ小さな評価値を選んだ時の評価値になるでゲソ。これは要するに次の局面の集合に対する『Min』演算と同じでゲソ！

図 1.8 が、そうやって下から順に評価値が決まっていく様子を示したものでゲソ！ これで、現在の局面の評価値、つまりどれくらい有利/不利そうか、が分かる上、Max ノードなので子ノードのうち一番評価値が大きかった子ノードに対応する手を打つのが最善手であるというのも分かるでゲソ！ 」

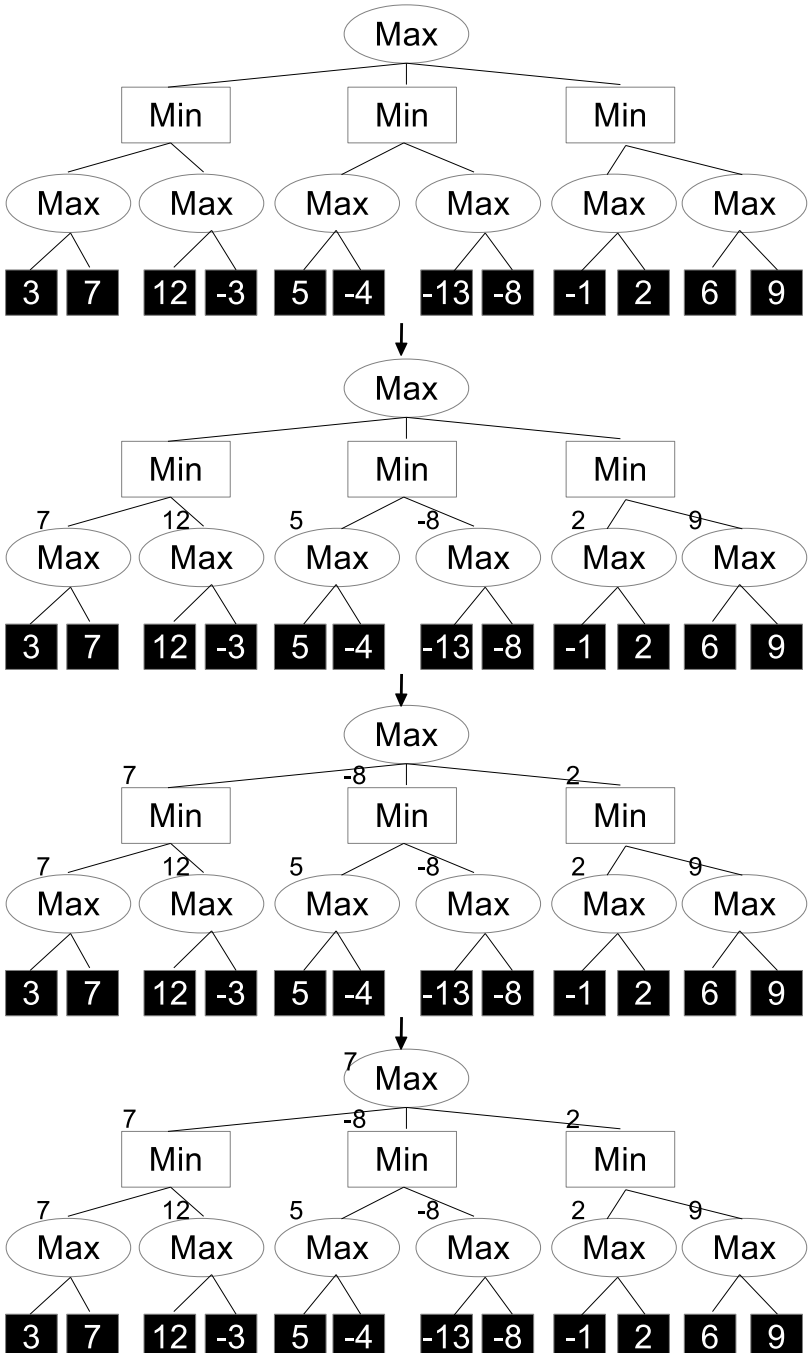

図 1.8: Min-Max でルート局面の評価値が決まる様子

1.5 α-β 探索

「実は、Min-Max 探索と同じ結果を保証しつつ、Min-Max 探索よりも大幅に少ない探索量で済む探索手法があるでゲソ。それが α-β 探索でゲソ！　どれくらい探索量が減るかというと、元の探索量の平方根くらいになるでゲソ。図 1.9 を使って説明するでゲソ。
(現在は並列化などのために厳密にそうとは言えないが) 通常は図の左にあるノードから評価関数が呼ばれるでゲソ。

- (a)start でゲソ。
- (b) 一番左の Min ノード (i) までの探索が終了した所でゲソ。ここで『よし、この一番左の手を選べば最低でも +7 で勝てる。でも、もっといい手はないのかな? 他の二つの可能性も調べてみて、+7 よりも私に有利な手ならそっちにしよう』というわけで次の探索に行くでゲソ。
- (c) というわけで二番目の手の探索を開始し、Max ノード (j) までの探索が終了した所でゲソ。
- (d)『ふむ、二番目の手を打つと、相手には評価値 +5 という手が発生する。相手はなるべく評価値の低い手を打ってくる (Min) から、二番目の手を打ったときのノード (k) の評価値が +5 より大きいということはありえない。さらに悪いことにもっと評価値が低い手、マイナスの手がある可能性もある。しかし、冷静に考えると、(b) で +7 より有利な手を探していたんだから、(k) が +5 以下であることが分かっただけで十分だな。これ以上 (k) について調べる必要はない。一部のノードは見てすらいないのだけど、三番目の手の探索に移れるな。』

という感じでゲソ！　」
「ふむ、なるほどのう。少し分かりづらいが、考えなくて良い局面があるわけじゃな。では、囲碁プログラムもこの『あるふぁべーた探索』を使って作られているのかね？」
「昔の囲碁プログラムはそうだったでゲソ。でも、すごく弱いのしか作れなかったでゲソ！　その理由を説明するでゲソ。」

1.6 α-β 探索が囲碁ではうまく機能しなかった理由

「α-β 探索が囲碁でうまく機能しなかった理由は大きく二つあるでゲソ！」「一つは囲碁では良い評価関数が作れないからでゲソ。すごくざっくり言うと、リバーシでは石のパターンを学習することで良い評価関数を作ることが出来、チェスや将棋では駒や駒組みの評価値を学習することで良い評価関数を作ることが出来たでゲソ。しかし、囲碁は、一つ一つは個性の無い石が無限とも言える組み合わさり方で様々な価値を生み出すので、良い評価関数を作ることが非常に難しいでゲソ！　人間の強い人がどのようにして形勢を判断しているかを言葉で説明しても、最低でもアマチュア初段くらいの力のある人でないと理解できないくらい、囲碁の形勢評価は難しいでゲソ!」
「確かにそうじゃのう。儂も良くアマチュアのあまり強くない人を指導するのじゃが、形勢判断の仕方を説明するのはすごく難しく感じるわい。」
「もう一つは、囲碁では打てる手の数が多すぎて、深く読めないからでゲソ。囲碁はルール上打てる手が、9 路盤 (練習用の小さい盤) の最初の状態で 81 通り、19 路盤 (公式の盤) の最初の状態で 361 通りあり、中盤付近でも 9 路盤で 60 通りくらい、19 路盤で 250 通りくらいもあるでゲソ！　ちなみに、リバーシだと石のひっくり返せる所しか打てないので、打てる場所は 10 通りくらいのことが多いでゲソ。もちろん手を絞り込む研究もされているでゲソが、少々絞り込んだだけでは焼け石に水でゲソ！　また、絞り込むと重要な手を見落とすでゲソ！」
「囲碁は、有段者になると、手を瞬間的に数か所に絞り込めると思うのじゃが、コンピュータにとって、それは難しいということじゃな。では、囲碁プログラムは一体どうやって強くなったのかね？　」

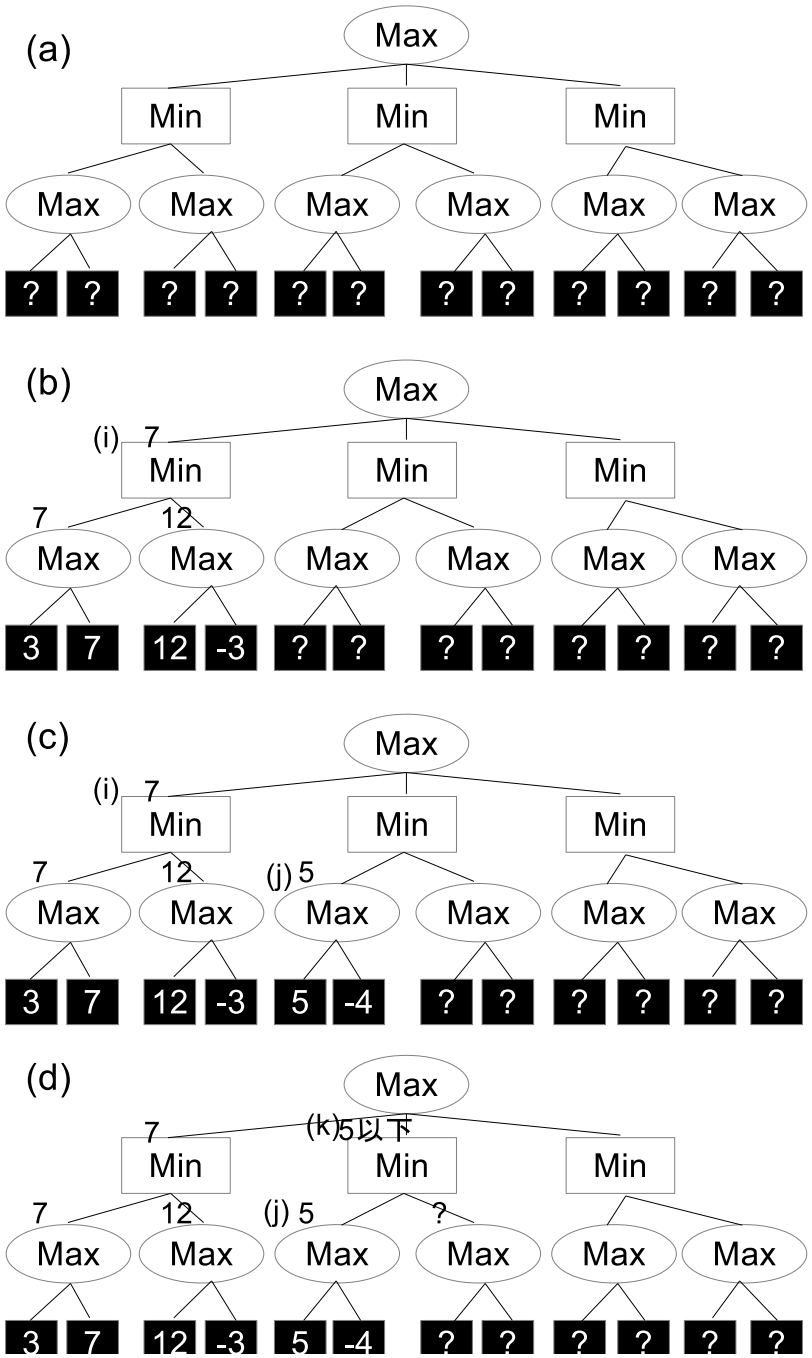

図 1.9: α-β 探索の様子

「それを今から説明するでゲソ！ 」

1.7 原始モンテカルロ法

「『モンテカルロ法』という名前くらいは聞いたことがあるでゲソか？ 乱数を使って近似値を求める方法でゲソ！ 囲碁におけるモンテカルロ法、厳密に言うと原始モンテカルロ法とは、『次の一手として有力な手がどれかを探すために、まず候補手を打ってみて、その後無作為に最後まで、自殺手以外打つ場所が無くなるまで、打つということを何回も繰り返し、結果の平均、通常は何目勝ったかではなく何回勝ったか、すなわち勝率をその候補手の点数として、点数が最大の候補手を次の一手として選ぶ』手法でゲソ。なお、『無作為に最後まで打つ』シミュレーションのことを『プレイアウト』と呼ぶでゲソ。図 1.11 を見ると分かると思うでゲソ。この図の場合は、一個の候補手あたりのプレイアウトが 100 回ということでゲソね。単純に考えると候補手が 74 箇所あるでゲソから合計で 7400 回になるでゲソ。実際には、プログラムにもよるでゲソが、合計 1 万〜100 万回くらいプレイアウトするでゲソ！ 一見すごく多そうでゲソが、αβで全部の手を深いところまで調べるのに比べたらはるかに少ないでゲソ！ 」

「なんと！ そんな無茶苦茶な方法で強いソフトが作れるのかね！？ 」

「モンテカルロ法は 1990 年代には提案されていたでゲソが、当初は弱いソフトしか作れなくて、皆もこんな方法で強いソフトが作れるわけがないと思っていたでゲソ！ でも、2006 年頃に再び注目され、多くの工夫が加えられたことで、強くなったでゲソ！ それの一部を次の節で説明するでゲソ。」

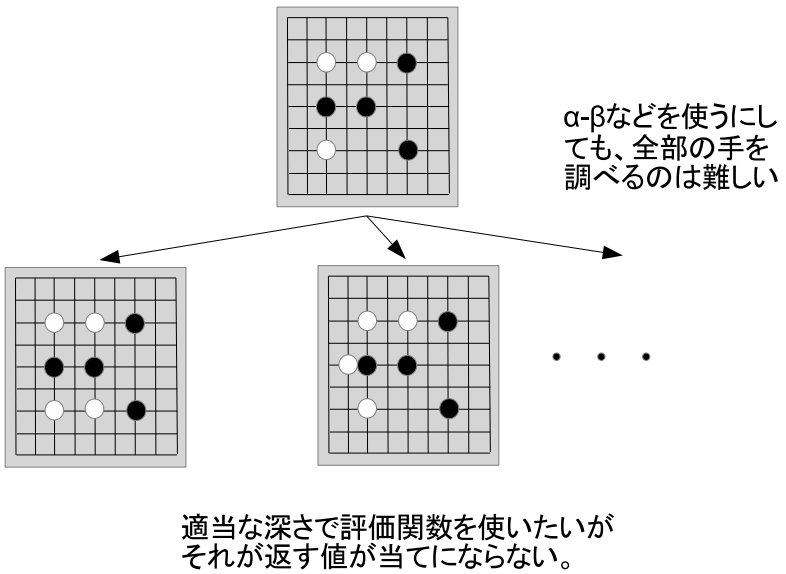

図 1.10: 囲碁で α-β は上手く機能しない

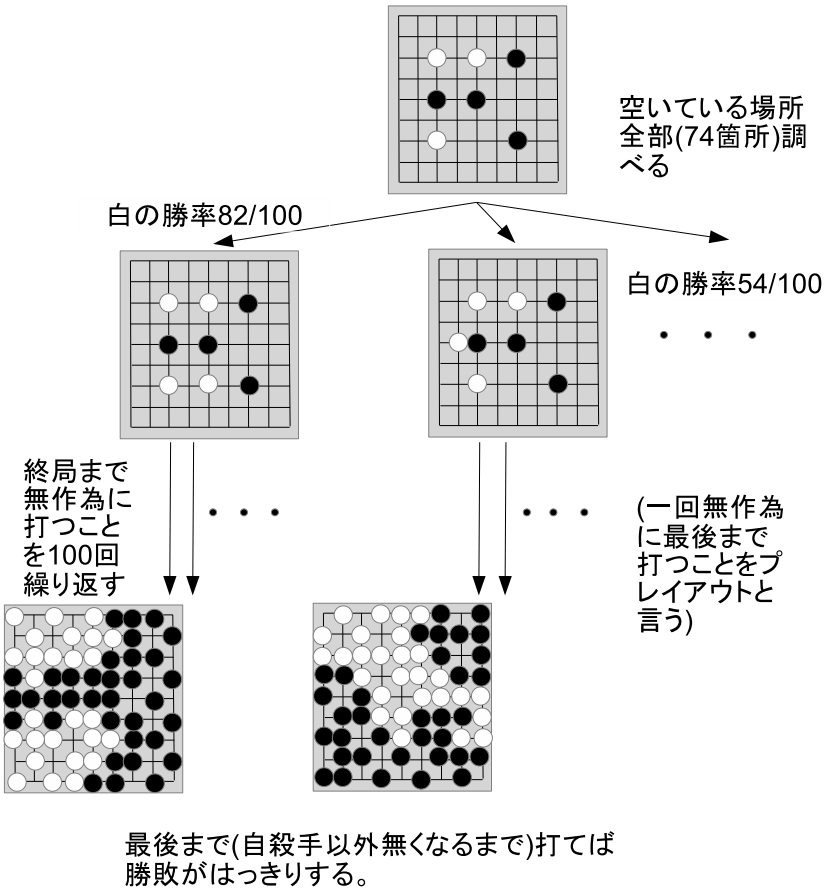

図 1.11: 囲碁における原始モンテカルロ法。

1.8 モンテカルロ木探索

「原始モンテカルロ法だけでも弱い囲碁プログラムが出来るでゲソが、原始モンテカルロ法に木探索を組み合わせるともっと強くなるんじゃなイカ？ 図 1.12 を使って、モンテカルロ法における木探索を説明するでゲソ。まずは普通に、各候補手に対して何度かプレイアウトを行って結果を集計するでゲソ。それから、ある候補手に対してのプレイアウト数が閾値 (1〜数百のことが多い) を超えたら、そこにノードを作るでゲソ (それまではプレイアウト数を集計するための仮のノードを

作っておく)。そうすると、深さ 1 のところにノードがたくさん出来るでゲソ。そうなったら、以降のプレイアウトを行う際に、そのノードの勝率とプレイアウト数に注目するでゲソ。そして、『勝率が高く、まだプレイアウト数が少ない』ノード (候補手) を選んで、そこからまたプレイアウトをするでゲソ。これを何段も繰り返すと、モンテカルロとして有望なところは深いところまでノードが作られるでゲソ！ そして、上記の $\alpha-\beta$ 探索と同様の効果が得られるようになるでゲソ！ これがモンテカルロ木探索でゲソ。」

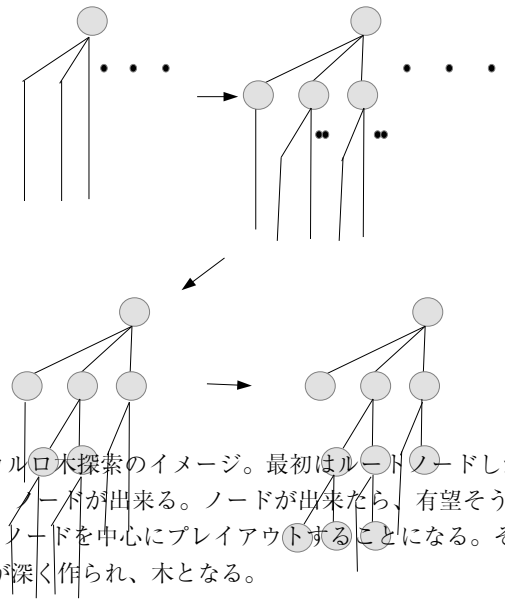

図 1.12: モンテカルロ木探索のイメージ。最初はルートノードしかないが、プレイアウトを繰り返すにしたがって、ノードが出来る。ノードが出来たら、有望そうな (勝率が高く、プレイアウト回数がまだ少ない) ノードを中心にプレイアウトすることになる。その結果、有望そうなところだけ、どんどんノードが深く作られ、木となる。

1.9 モンテカルロ木探索を用いた囲碁ソフトの現状

「モンテカルロ木探索を用いることで、囲碁ソフトは飛躍的に強くなったでゲソ！ 当初は 9 路盤でしか有効でないと思っていた人もいるようでゲソが、たくさんの工夫によって、19 路盤でもかなり強くなったでゲソ。現在最強の囲碁ソフトは『Zen』と『CrazyStone』だと言われているでゲソが、どちらもネット碁の KGS というところで 5d〜6d[*2] くらいにランクされているでゲソ。これは日本基準だとアマチュアの 7〜8 段くらいでゲソ。Zen は 4 子置いて (Zen 側が 4 段級差有利) トッププロの武宮プロに勝ったことがあるでゲソ。CrazyStone も 4 子置いてトッププロの石田プロに勝ったことがあるでゲソ！ また、Zen は 3 子置いて高校チャンピオンの大表さんに勝ったこともあるでゲソ。」

[*2] KGS では対戦成績によって段級位がつく。5d というのは 5 段ということ。ただし、日本のアマチュアの 5 段よりも評価が厳しいと言われている。なお、日本と同様に級もあり、例えば 1 級なら 1k となる。

「そんなに強いのかね！　儂もトッププロと雑誌の企画で何度か打ったことがあるが、3 子置いただけでは勝ったことないし、4 子置いてようやく五分五分という感じじゃったな。」
「でも、アマチュアトップレベルの多賀さんとハンデなしで打ってみたこともあるでゲソが、その時はボコボコにされていたでゲソ……。図 1.13 がそのボコボコにされた棋譜でゲソ。黒が CrazyStone でゲソが、なんと上辺の攻め合いを CrazyStone が理解出来ていなくて、図の局面でも CrazyStone は自分が優勢と思っていたでゲソ！　でも、実際には 100 目以上の大差で多賀さんが優勢でゲソ！　ここから碁の神様が CrazyStone の代わりに打ったとしても、逆転は不可能でゲソ！　仕方なくオペレーターが投了したでゲソ……。このように、現在の囲碁ソフトは強いでゲソが、攻め合いのような細く長い読みを必要とする局面が弱点なんじゃなイカ？　これを解決するために世界中の研究者が研究しているでゲソが、未だ有効な解決策は見つかっていないでゲソ。もちろんこの文章の著者も研究しているでゲソ！」
「なるほどのう。確かに『適当に最後まで打って』形勢判断するという仕組みだと、一本道で長い手順は見落としてしまうわけじゃな。」
「ちなみに、モンテカルロ将棋も研究されているでゲソ。でも、まだまだ α-β ベースのプログラムには勝てないようでゲソ。なぜ囲碁でモンテカルロ法が上手く働くのか、理由は大きく二つあるでゲソ。まず一つは、囲碁は一手の手の価値の差が小さいから、ランダムシミュレーションの結果の平均が形勢の良い近似になりやすいからでゲソ。もう一つは、囲碁はランダムに打ってもほぼ間違いなく終局するゲームだからでゲソ。」
「言われてみれば確かにそうじゃな。将棋だとそもそも適当に打っていたら詰まないし、詰みを見落とすか見落とさないかだけで結果が逆転してしまうからの。」

1.10　エピローグ

「いやはや囲碁ソフト試してみたぞい。最初は試しにお主と同様 3 子置かせて (3 段級差ソフト有利) みたのじゃが、コロッと負かされてしまったワイ。ただ、何局か打っているうちに弱点が分かってしまって、最近は 5 子 (5 段級差ソフト有利) でも勝てるようになったワイ。」
「5 子でゲソか……。相変わらずすごいでゲソね……。爺さんプロより強いんじゃなイカ？」
「いやいや、プロが試さないような変な手を試しているうちに弱点が分かったというだけじゃよ。しかし、やはりお主のように、人間と打っている方が楽しいの。」
「(私は人間ではなくλカ娘でゲソが、つっこまないでおくでゲソ)。じゃあ今日も打つでゲソ!」
　λカ娘の挑戦は終わらない……。

1.11　参考文献

- コンピュータ囲碁-モンテカルロ法の理論と実践-(松原仁　編集、美添一樹、山下宏共著 共立出版)
- ゲーム計算メカニズム (小谷善行 編著、岸本章宏、柴原一友、鈴木豪 共著　コロナ社)
- Combining Online and Offline Knowledge in UCT(Sylvain Gelly, David Silver, Proceedings of the 24th International Conference on Machine Learning)
- Computing Elo Ratings of Move Patterns in the Game of Go(Rémi Coulom, ICGA Computer Games Workshop, Amsterdam, The Netherlands, June 2007)

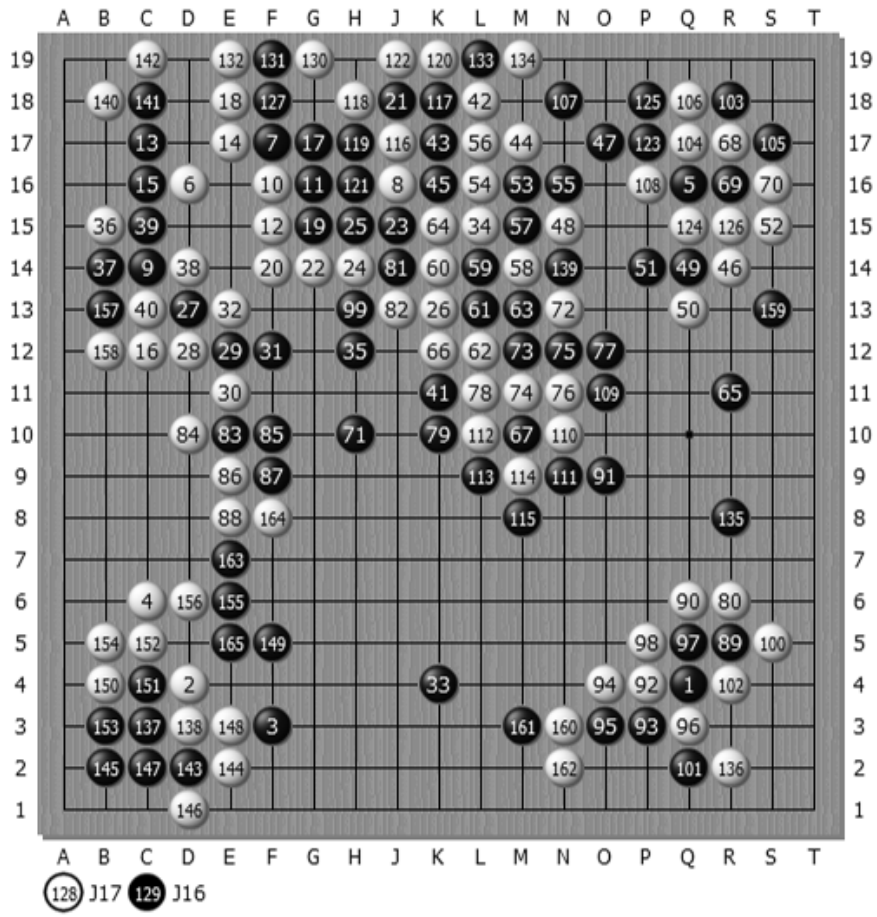

図 1.13: 囲碁ソフト、トップアマに完敗!

第2章

僕のカノジョはスナッチャー

— @master_q

2.1 川のほとり

『下におろして、垂直にまっすぐ持ち上げて敵を打ち抜くんでゲソ。ここ、両脇が大事なんでゲソ。そーして、打つべし、打つべし、打つべし、ぐるーんでゲソー!』
「なんなの、それ？」
『スナッチを制する者は、世界を制すでゲソ。最初のターゲットを決めるのが難しいんでゲソ』
「またサバゲーの話？」
『ん？ 何やってんでゲソ？』
「型推論の宿題」
『なんで大学でやんないんでゲソ？』
「かっこう悪いから」
『ワシのもやってくれなイカー』
「Dvorak 配列のままだよ」
『ぬ？ おぬしもスナッチやればいいじゃなイカ。なんでいつも ThinkPad 持ってるんでゲソ？』
「自分こそ、なんでいつもここに来るんだよ。」
『それはでゲソね、…あれ、なんででゲソ？』
「お気楽だね。イカ墨くさい」
『出してないでゲソ』
「あのさ、なんでいつもこんなこと…」
『人間の指はおもしろいでゲソー』
「ん？」
『ワシはこうしてないと溢れちゃうじゃなイカ』
「溢れちゃうって、どうなっちゃうんだ？」
『……きっと…すごいことになるでゲソ……』

すごいことなんてない。ただ当たり前のことしか起こらない。僕達の大学から見えるオンボロなビル、ソフトハウス–メタセピのオフィスができたとき、プログラマたちは大騒ぎだった。毎日決まった時刻に鳴り出すスナッチのメロディーが、僕にはなんだか不吉なサイレンのように聞こえる。それはうっすらと広がり、世界を覆っていく。

2.2 カノジョのこと

僕のカノジョはスナッチャーだ。スナッチャーというのは他の生物の一部を置換して自分の体内に取り込むエイリアンの名前だ。いつから彼等がこの世界に現われたのかはわかっていない。とにかく現在では僕たち人間とスナッチャーが同居して暮していた。

僕が大学で授業を受けていると、カノジョが突然声をかけてきた。どうも授業の宿題が解けないらしい。僕は他人と会話するのが苦手なのでおっくうだったが、簡単に鍵となるアイデアを伝えた。次の日もその次の日もカノジョは僕に声をかけてきた。そのうち、僕達は毎週いっしょにハイキングにいくようになった。毎週どこにハイキングに行くのか、実のところそのネタを考えるのも僕にはおっくうだった。紙と鉛筆がそれまで僕の友達だったから。

カノジョはスナッチャーなので、ときおり僕の一部をスナッチしたくなるようだ。どこが好かれているのかわからないが、カノジョは僕のことをお気に入りだ。もちろん僕だってスナッチされるのは御免だ。カノジョだって僕を完全にスナッチしてカノジョ自身の中に飲み込んでしまったら、僕という存在がなくなってしまう。一緒にハイキングに行く相手がいなくなる。そんなのつまらないと思っているんだろう。今のところ、カノジョは僕のことを本気でスナッチしてこようとは思わないようだ。

カノジョのお気に入りのタブレットは Nexus 7 だ。このタブレットは Android OS で動いていて、Android NDK を使えば C 言語だけでアプリケーションを書くこともできる。カノジョはスピード狂だったから、Android のアプリはいつも C 言語で書いていた。でも僕は C 言語よりも Haskell の方が気に入っている。型の強い言語というのは設計時の安心感が段違いだからだ。せっかくなので、Haskell で面白いアプリをカノジョのために書いてあげたいと最近は思うようになった。

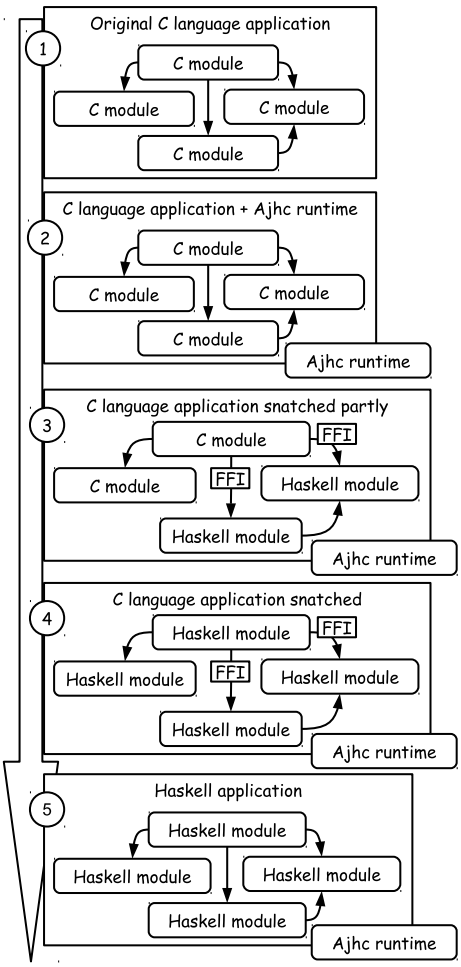

図 2.1: Snatch-driven development の概要

2.3 Snatch-driven development

ところが Android NDK を使っても Haskell で Android アプリケーションを書くのは簡単ではない。たしかに Android NDK を使えば POSIX API を使った C 言語プログラミングができる。そのため GHC を使ったクロス開発が可能だ。しかし GHC だとバイナリサイズが巨大になる傾向がある。簡単なロジックでも数十 MB ものコードサイズになってしまうこともある。また、Android はアクティビティという少しかわったしくみでプログラムの状態管理をしている。このようなドメイン固有の作法を守りつつゼロからアプリケーションを設計することは簡単ではないのだ。もし C 言語のまっとうな実装があれば、そのコードの設計を推測して Haskell で書き直すこともできる。しかしプログラムの規模が大きくなるにつれてこの作業は難しくなる。というのも”C 言語のコードから設計を推測する” ことが完全に完了しないと” 設計を Haskell で書き下す” ことができないからだ。当然この作業が完全に完了しないと動作確認はできない。小さなユニットテストはできるが、そのユニットテストが設計から正しく意図されたものであるかも自信が持てない。この流れを少しずつ進行させる方法はないだ

ろうか？

ところで、ソフトウェアには “Snatch-driven development” という開発手法がある。Ajhc という Haskell コンパイラを組み合わせて使えば、C 言語の設計を動作可能な状態をたもったまま Haskell で書き直すことが可能だ。また Ajhc には大変小さなコードサイズのバイナリを吐くという特性もある。以前からこの手法に興味があった。カノジョへプレゼントする Android アプリを作るのに、C 言語で書かれた簡単なアプリケーションを Haskell でスナッチした後、その Haskell の設計を成長させるような方法を取ることはできないだろうか？ Android アプリを C 言語から Haskell で書き直すことは、スナッチ設計を身に付ける上で良いトレーニングになるだろう。

スナッチのやり方を簡単に解説しておこう (図 2.1)。スナッチは多くの場合以下のステップを取る。下のステップの番号はそのまま図 2.1 の番号にそのまま対応している。

Step 1 まず既存の動作可能な C 言語のソースコードを用意する。この C 言語ソースコードは実行可能、もしくはテスト可能であればなんでも良い。

Step 2 その C 言語ソースコードにとりあえず Ajhc コンパイラのランタイム部分だけを埋め込む。この状態でまず実行/テストしてみて問題ないことを確認する。

Step 3 その後、プログラム内のモジュールの内 Haskell 化できそうな箇所を選んで設計置換をする。この時、Haskell と C 言語の間は FFI を使って接続する。つまり Haskell の型を使わず C 言語の ABI でモジュール間を接続する。

Step 4 この設計置換をくりかえしてプログラム全体を Haskell 化する。

Step 5 全体を Haskell 化しおわった状態で動くことを確認できたら、最後に FFI を取り去って Haskell module 同士を型によるインターフェイスで接続する。

これで元の C 言語の設計がそのままの形をたもったまま Haskell 化できたことになる。

2.4 Android NDK とそのサンプルアプリケーション

さっそく Android アプリスナッチの準備をはじめよう。何を用意すればいいだろうか？

- Android NDK 開発環境とその概要理解
- スナッチする Android NDK サンプルソースコード

こんなところだろうか。順番に手をつけていこう。

2.4.1 Android NDK 開発環境

Android NDK [*1] というのは先にも言った通り Android のアプリケーションを C 言語で書くための開発環境だ。詳細ははぶくが Android は通常の UNIX ライク OS と異なり、プロセスではなくプロセスの上に構築されたアクティビティというしくみでアプリケーションを管理している。通常はこのアクティビティは Java 言語で設計することになっている。特殊なライフサイクルがあるので生の C 言語では扱いにくいのだ。ところが NativeActivity というしくみを使う

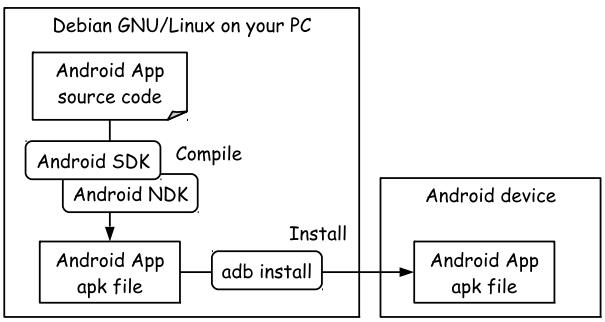

図 2.2: Android NDK アプリのクロス開発環境

[*1] http://developer.android.com/tools/sdk/ndk/index.html

と、このアクティビティを C 言語のみで設計することが可能になる。しかし Android フレームワークから渡されるイベントにすばやく応答するために、Android NDK プログラミングでは POSIX スレッドを使う必要がある。もちろん誰もが知っている通り、POSIX スレッドを使ったプログラミングは大変難しい。デッドロックを防止するために非常に注意深い設計が求められることなどがその理由だ。

そこで、Android NDK ではこの POSIX スレッドを使ったアクティビティ制御のラッパーとして android_native_app_glue を提供している。android_native_app_glue はアクティビティのライフサイクルを C 言語でも書きやすくしてくれるミドルウェアのようなものだ。android_native_app_glue を使えば設計者は POSIX スレッドを意識することなく NativeActivity アプリケーションを設計できる。

Android NDK 開発環境の設定に入る前に、開発環境のおおざっぱなイメージをつかんでおこう (図 2.2)。通常のアプリケーションとは違い、Android 上のアプリケーションは Android OS 上ではなく別の PC 上でコンパイルする。ぼくは Debian GNU/Linux を使っているので、Intel アーキティクチャ上の Debian に Android SDK と Android NDK をインストールして、ARM アーキティクチャの Android OS 向けのアプリケーションパッケージである apk ファイルをビルドできる環境を整える必要がある。apk ファイルができあがったら Android SDK に付属している adb というコマンドを使って apk ファイルを Android デバイスにインストールする。これで Android デバイスのランチャー画面から自作の Android アプリケーションのアイコンが見えるようになるはずだ。

僕はさっそく Android NDK 開発環境をととのえるために愛用の ThinkPad を開いた。[*2] まず必要なファイルをそろえよう。以下のファイルをダウンロードする。

- Android SDK (adt-bundle-linux-x86_64-20130917.zip) [*3]
- Android NDK (android-ndk-r9-linux-x86_64.tar.bz2) [*4]

そして Linux 用の JVM をインストールした後、Android SDK/NDK のインストールを行なう。

```
$ cd $HOME
$ unzip -x adt-bundle-linux-x86_64-20130917.zip
$ mv adt-bundle-linux-x86_64-20130917/sdk $HOME/android-sdk
$ rm -f adt-bundle-linux-x86_64-20130917
$ export PATH=$PATH:$HOME/android-sdk/tools:$HOME/android-sdk/platform-tools
$ tar xf android-ndk-r9-linux-x86_64.tar.bz2
$ mv android-ndk-r9 android-ndk
$ export PATH=$PATH:$HOME/android-ndk
$ sudo apt-get install openjdk-7-jdk ant lib32z1-dev lib32stdc++6
$ android update sdk --no-ui --force
```

Android SDK/NDK のセットアップが終わったので、なにかサンプルアプリケーションをビルドしてみよう。さっき展開した Android NDK の中に native-activity という NativeActivity を使ったサンプルアプリケーションが含まれている。どうやら非常に規模の小さいアプリケーションのようだ。今回はこの native-activity をスナッチしよう。さっそくスナッチの前にこのサンプルをビルドして apk ファイルを作ってみよう。

[*2] Debian GNU/Linux amd64 sid 2013/11/01 時点。GHC のバージョンは 7.6.3。Ajhc のバージョンは 0.8.0.9。

[*3] http://developer.android.com/sdk/index.html

[*4] http://developer.android.com/tools/sdk/ndk/index.html

```
$ cd $HOME/android-ndk/samples/native-activity
$ android update project -p ./ -n native-activity -t android-10
$ ndk-build NDK_DEBUG=1
$ ant debug
$ file bin/native-activity-debug.apk
bin/native-activity-debug.apk: Java Jar file data (zip)
```

Android NDK アプリケーションのコンパイルは非常に簡単にできるようだ。このビルドで生成された apk ファイルに Android にインストールするアプリケーションが入っている。最後にこの apk ファイルを実機にインストールしてみよう。まず Android デバイスと PC を USB ケーブルで繋ぎ、adb コマンドで接続確認をする。

```
$ sudo adb devices
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
List of devices attached
0X5X4X0X0X3X1X0X        device
```

“List of devices attached” の後にデバイスが見えれば接続に成功している。デバイスが見えたことを確認できたら続いて adb コマンドで apk ファイルをインストールしよう。

```
$ sudo adb install -r bin/native-activity-debug.apk
1221 KB/s (156823 bytes in 0.125s)
        pkg: /data/local/tmp/native-activity-debug.apk
Success
```

インストールが終わったら、Android デバイスのランチャーから”NativeActivity” というアプリを立ち上げてみよう。アプリの黒い画面を指でなぞると色が変化するはずだ。この”NativeActivity” アプリはタッチセンサーの入力に応じて画面に一枚だけ表示しているポリゴンの色を変化させる OpenGL ES のデモアプリケーションだ。

2.4.2 Android NDK サンプルソースコード

さっきコンパイルした Android NDK サンプルソースコード”$HOME/android-ndk/samples/native-activity” の中身をざっくり調べておこう。

```
native-activity
|-- AndroidManifest.xml
|-- default.properties
|-- jni
|   |-- Android.mk
|   |-- Application.mk
|   `-- main.c
`-- res
    `-- values
        `-- strings.xml
```

設定ファイル群については置いておくとして、なんと C 言語のソースコードは 1 ファイルし

か入っていないようだ。詳しいソースコード解析はスナッチしながらするとして、おおざっぱに main.c ファイルの中をのぞいてみよう。[*5]

```
// -- snip -- Some #include lines
struct saved_state {
// -- snip --
struct engine {
// -- snip --
static int engine_init_display(struct engine* engine) {
// -- snip --
static void engine_draw_frame(struct engine* engine) {
// -- snip --
static void engine_term_display(struct engine* engine) {
// -- snip --
static int32_t engine_handle_input(struct android_app* app,AInputEvent* event){
// -- snip --
static void engine_handle_cmd(struct android_app* app, int32_t cmd) {
// -- snip --
void android_main(struct android_app* state) {
    struct engine engine;

    memset(&engine, 0, sizeof(engine));
    state->userData = &engine;
    state->onAppCmd = engine_handle_cmd;
    state->onInputEvent = engine_handle_input;
// -- snip --
```

構造体の定義が 2 つ。static 関数が 5 つ。エントリ関数が 1 つ。というシンプルな構成だ。android_native_app_glue.h を include することで生の Android NDK ではなく android_native_app_glue ラッパーを使っていることがわかる。static 関数はエントリ関数の頭でコールバックとして設定する。Android フレームワーク側からのイベントをこのコールバックで処理する。つまり android_native_app_glue を使った Android NDK アプリケーションはイベントドリブンだということになる (図 2.3)。ここまでで図 2.1 の Step 1 が完了し、Step 2 への準備が整った。

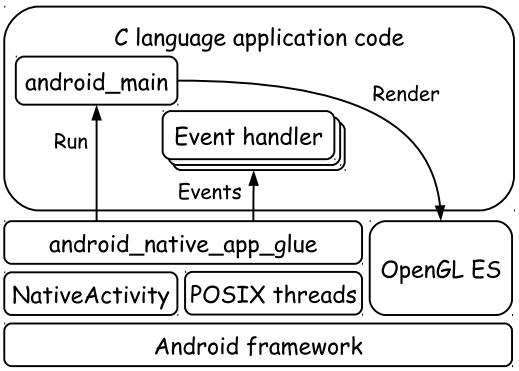

図 2.3: Android NDK アプリの構造

[*5] native-activity サンプルソースコードとこの記事に記載されているその他のソースコードのライセンスは Apache License 2.0 http://www.apache.org/licenses/LICENSE-2.0。

2.5 スナッチ 1: 何もしない Haskell プログラム

さぁ最初のスナッチを開始しよう。この章は図 2.1 の Step 2 に対応する。実はスナッチ設計においてもっとも難しいのはこの最初のスナッチだ。C 言語のコードと Haskell のコードを連携させるだけであれば楽なのだが、最初のスナッチではその連携をさせるためのビルドプロセスを先に作らなければならないからだ。なぜ独自のビルドプロセスが必要になるのだろうか？ 今の Android NDK アプリケーションのビルドシステムは C 言語のビルドしか考えていない。このビルドシステムに Haskell ソースコードのビルドさせる必要がある。Haskell ソースコードのビルドに必要な要素とは何なのだろうか？ おおざっぱに以下に列挙してみよう。

- スナッチした Haskell ソースコード
- Haskell ソースコードを C 言語化するための ajhc コマンド呼び出し
- ajhc コマンドが自動生成するランタイムの実行に必要なスタブ関数群
- 元の C 言語ソースコードである main.c と、ajhc コマンドが自動生成したランタイム、Haskell から変換された C 言語ソース[*6]、スタブ関数群の 4 つをコンパイルするための gcc コマンド呼び出し

今回作成するビルドプロセスはややこしいので図 2.4 にまとめてみた。ここで説明するビルドプロセスはこのスナッチをする中で Makefile にまとめることにする。このビルドシステムでは元の C 言語のソースコードは jni ディレクトリに、Haskell のソースコードは hs_src ディレクトリに配置する。Haskell ソースはまずはじめに Ajhc によってコンパイルされて hs_build ディレクトリに C 言語のファイルとして出力される。この時、Ajhc は Haskell ソースコード由来の C 言語ファイル hs_main.c だけではなく、ランタイムのソースコード hs_build/lib/*.c と hs_build/rts/*.c も吐き出す。その後、サンプルコードに付属していた ant のビルドシステムが走り、これまで登場した C 言語のソースコードを全てまとめてコンパイルし、Android アプリケーションである apk ファイルを作成する。

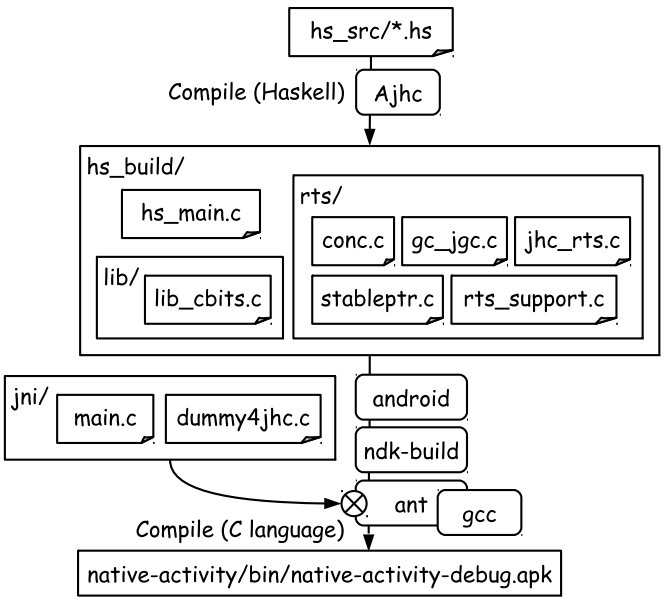

図 2.4: Android NDK Haskell アプリのビルドプロセス

スナッチを開始する前に Ajhc Haskell コンパイラをインストールしよう。

[*6] Ajhc コンパイラは Haskell ソースコードを C 言語ソースコードに変換した後、その C 言語ソースコードと Ajhc ランタイムソースコードを GCC にコンパイルさせることで実行バイナリを得る

```
$ sudo apt-get install haskell-platform libncurses5-dev gcc m4
$ export PATH=$HOME/.cabal/bin:$PATH
$ cabal update
$ cabal install ajhc
$ ajhc --version
ajhc 0.8.0.10 (ef04922345d3503235620c86405af838f0271347)
compiled by ghc-7.6 on a x86_64 running linux
```

まず、これまで調べたサンプルアプリケーションのビルド手順を Makefile にまとめておこう。これで make コマンドを実行するだけで C 言語のサンプルアプリケーションがビルドできるようになった。

```
APPNAME = native-activity
ANDROID_TARGET = android-10
CDIR = jni
CSRC = $(wildcard $(CDIR)/*.c $(CDIR)/*.h)

all: bin/$(APPNAME)-debug.apk

bin/$(APPNAME)-debug.apk: $(CSRC)
        android update project -p ./ -n $(APPNAME) -t $(ANDROID_TARGET)
        ndk-build NDK_DEBUG=1
        ant debug
```

Makefile の文法に慣れていなければ、この Makefile の記述についての理解を飛ばしてしまっても良い。ここでは図 2.4 のビルドプロセスについて理解できていれば十分だからだ。

図 2.1 の Step 2 によると最初のスナッチではまず Ajhc のランタイム部分だけを既存 C 言語プログラムに埋め込むのだった。これは具体的には何をすべきだろうか？ Ajhc ランタイムだけを動作させることはできない。そのランタイムの上で動作させる Haskell プログラムが必要になる。もっとも簡単な Haskell プログラムとはなんだろう。hs_src/Main.hs というファイルに何もしない Haskell プログラムを書こう。

```
main = return ()
```

この何もしない Haskell プログラムを C 言語のエントリ関数から呼び出してみる。jni/main.c ファイルを修正しよう。

```
$ git diff jni/main.c
diff --git a/jni/main.c b/jni/main.c
index 2e19635..4767594 100644
--- a/jni/main.c
+++ b/jni/main.c
@@ -234,6 +234,17 @@ void android_main(struct android_app* state) {
     // Make sure glue isn't stripped.
     app_dummy();
```

```
+    {
+        // Init & run Haskell code.
+        int hsargc = 1;
+        char *hsargv = "q";
+        char **hsargvp = &hsargv;
+
+        hs_init(&hsargc, &hsargvp);
+        _amain();
+        // hs_exit(); /* FALLTHROUGH */
+    }
+
     memset(&engine, 0, sizeof(engine));
     state->userData = &engine;
     state->onAppCmd = engine_handle_cmd;
```

jni/main.c ファイルの_amain() 関数によって Haskell の main 関数が呼び出される。当然、これだけでコンパイルが通るわけはない。Ajhc のランタイムはいくつかの POSIX の関数に依存しているためだ。しかし Android アプリでは標準入出力などは使わないため、それらの関数のいくつかはダミーの空関数にすることができる。そこで、その関数群のダミー実装を提供する jni/dummy4jhc.c ファイルを作る。

```
#include "jhc_rts_header.h"
int jhc_printf_stderr(const char *format, ...) { return 0; }
int jhc_fputs_stderr(const char *s) { return 0; }
int jhc_fflush_stdout() { return 0; }
void jhc_print_profile(void) { return 0; }
int jhc_utf8_getchar(void) { return 0; }
int jhc_utf8_getc(FILE *f)  { return 0; }
int jhc_utf8_putchar(int ch)  { return 0; }
int jhc_utf8_putc(int ch, FILE *f)  { return 0; }
```

Ajhc では JHC_USE_OWN_STDIO フラグをコンパイル時に渡した場合には上記の関数を独自実装に置き換えることができる。JHC_USE_OWN_STDIO フラグを使わなかった場合には上記の関数群は POSIX を使った標準実装になる。Ajhc のランタイムのファイル群とさっき作った dummy4jhc.c ファイルをコンパイル対象に追加するために、jni/Android.mk ファイルを修正する。

```
$ git diff jni/Android.mk
diff --git a/jni/Android.mk b/jni/Android.mk
index 9e64d80..9727688 100644
--- a/jni/Android.mk
+++ b/jni/Android.mk
@@ -17,8 +17,16 @@ LOCAL_PATH := $(call my-dir)
 include $(CLEAR_VARS)

 LOCAL_MODULE    := native-activity
```

```
-LOCAL_SRC_FILES := main.c
+LOCAL_SRC_FILES := main.c dummy4jhc.c  ../hs_build/hs_main.c \
+       ../hs_build/rts/rts_support.c ../hs_build/rts/jhc_rts.c \
+       ../hs_build/lib/lib_cbits.c ../hs_build/rts/gc_jgc.c \
+       ../hs_build/rts/stableptr.c ../hs_build/rts/conc.c
+LOCAL_C_INCLUDES := hs_build/cbits hs_build
+LOCAL_CFLAGS     := -std=gnu99 -D_GNU_SOURCE -falign-functions=4 \
+       -ffast-math -fno-strict-aliasing -DNDEBUG -D_JHC_GC=_JHC_GC_JGC \
+       -D_JHC_CONC=_JHC_CONC_PTHREAD -D_JHC_USE_OWN_STDIO
 LOCAL_LDLIBS    := -llog -landroid -lEGL -lGLESv1_CM
+LOCAL_ARM_MODE  := arm
 LOCAL_STATIC_LIBRARIES := android_native_app_glue

 include $(BUILD_SHARED_LIBRARY)
```

最後に Ajhc を使った Haskell から C 言語への変換を Makefile で自動化しよう。

```
$ git diff Makefile
diff --git a/Makefile b/Makefile
index 423e04a..dd6bd7a 100644
--- a/Makefile
+++ b/Makefile
@@ -3,13 +3,23 @@ ANDROID_TARGET = android-10
 CDIR = jni
 CSRC = $(wildcard $(CDIR)/*.c $(CDIR)/*.h)

+HSBUILD = hs_build
+HSDIR = hs_src
+HSSRC = $(wildcard $(HSDIR)/*.hs)
+
 all: bin/$(APPNAME)-debug.apk

-bin/$(APPNAME)-debug.apk: $(CSRC)
+bin/$(APPNAME)-debug.apk: $(HSBUILD)/hs_main.c $(CSRC)
        android update project -p ./ -n $(APPNAME) -t $(ANDROID_TARGET)
        ndk-build NDK_DEBUG=1
        ant debug

+$(HSBUILD)/hs_main.c: $(HSSRC)
+       mkdir -p $(HSBUILD)
+       ajhc -fffi -fpthread --include=hs_src \
+            --tdir=$(HSBUILD) -C -o $@ $(HSDIR)/Main.hs
```

これでサンプルアプリケーションは” 一度何もしない Haskell コードを実行してから起動する” ようになった。一見、今回のスナッチには何の意味もないように見える。しかしこの最初のスナッチ

は実は無から有を作り出す大事な作業だ。今回のスナッチをする前はアプリケーションのソースコードは全部 C 言語で書かれていた。この C 言語コードを Haskell 化しようとしても、Haskell コードを配置する場所がなかった。なぜなら Haskell コードが一行も書かれていなかったからだ。また Ajhc のランタイムもなかった。最初のスナッチによってたった一行だけの Haskell コードと Ajhc ランタイムをアプリケーションが内在するようになった (図 2.5)。

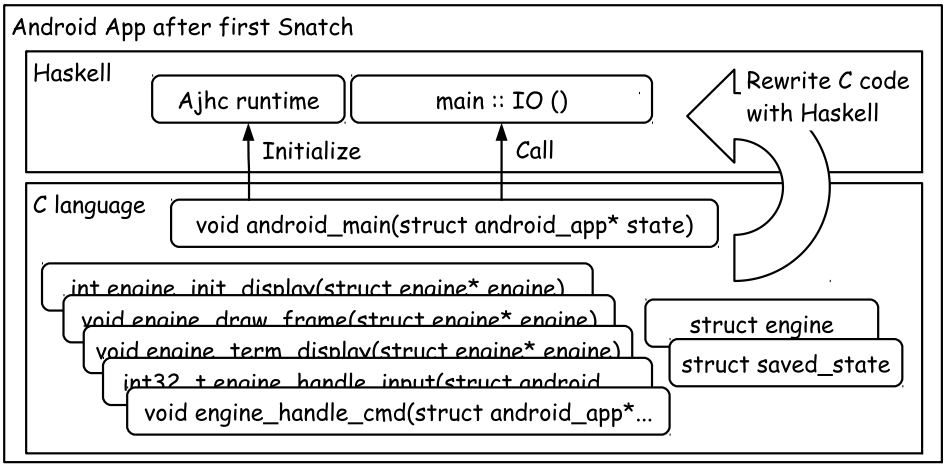

図 2.5: 最初のスナッチで C コードを Haskell 化できるようになった

スナッチが終わったらビルドが通ることと、元のアプリケーションと同様の動作を確認する必要がある。今回は何もスナッチせず、単に何もしない Haskell 関数を呼び出しているだけだ。それでも Ajhc ランタイムの初期化でエラーが発生する可能性は否定できない。幸運にもこの時点でコンパイルの異常と動作の異常は見られなかった。

これで Haskell コードを置く場所ができ、またその Haskell コードが実行可能であることが確かめられた。少なくとも Ajhc ランタイムの初期化には成功している。この後は、Haskell コードを含むアプリケーションがコンパイル可能/実行可能な状態を維持しているか確かめながらスナッチを進めることになる。

2.6 スナッチ 2: 最初のターゲット

これで Android 上で Haskell コードを動かす準備ができた。つまり C 言語のコードを Haskell でスナッチする準備が整ったわけだ。図 2.1 の Step 3 に進むことにしよう。さて、まずどこから手をつけるべきだろうか？ 最初に調べた時に内部の構造は関数と構造体定義だけの簡単なものだった。強い型を好む僕としては C 言語の構造体定義に Haskell の型を付けたいと思うところだが、おそらく構造体定義はほうぼうの関数で共用しているはずなので、ある程度スナッチが進行した後でないと Haskell の型を付けても影響範囲が大きいだろう。まずは一番簡単な関数を Haskell 化してみよう。行数と構造から見て一番簡単な C 言語の関数は engine_handle_input() のようだ。

```
static int32_t engine_handle_input(struct android_app* app,AInputEvent* event){
    struct engine* engine = (struct engine*)app->userData;
    if (AInputEvent_getType(event) == AINPUT_EVENT_TYPE_MOTION) {
        engine->animating = 1;
        engine->state.x = AMotionEvent_getX(event, 0);
```

```
        engine->state.y = AMotionEvent_getY(event, 0);
        return 1;
    }
    return 0;
}
```

この関数はどのような処理を行なっているのだろうか？

1. struct android_app から struct engine へのポインタを取り出す
2. AInputEvent_getType() 関数で AInputEvent からイベント種別を取得
3. AMotionEvent_getX() と AMotionEvent_getY() で AInputEvent から xy 座標を取得
4. struct engine のメンバーを書き換える

どうやら関数の処理の中心には struct engine がいてデータを仲介しているようだ。さっき構造体よりも関数をスナッチ対象にすると決めたが、この関数を見るかぎり struct engine の方を先にスナッチするべきに見える。

Haskell の FFI の使い方を復習しておこう。構造体のスナッチには Foreign.Storable.Storable クラス[*7] [*8] を使う。Storable クラスのインスタンスは poke と peek という 2 つのアクションを定義する。これらのアクションの中で、C 言語の構造体と Haskell の型との相互変換を記述する。すると、当該の C 言語の構造体へのポインタを peek すると Haskell ヒープに確保された Haskell の型が得られ、また C 言語の構造体へのポインタへ Haskell の型を poke すると Haskell の型に内在していたデータが C 言語の構造体への書き込まれる。

もう少しだけ説明しておこう。Haskell では関数とデータの境界はあいまいだ。しかし C 言語では手続とデータ、つまり関数と構造体は明確に分かれている。つまり C 言語のスナッチにおいては、手続のスナッチは foreign import を使って C 言語の関数を Haskell から呼び出し可能にすればいい。構造体のスナッチは Storable クラスを使って C 言語の構造体と Haskell の型の相互変換のルールを記述すればいい。スナッチがあまり進行していない時点では、この論理をあまり崩さない方がよい。既存の C 言語の要素をスナッチするときに邪魔になることが多い。もし完全にスナッチが完了して C 言語側の構造体を使わなくなったら、Haskell の型に Storable クラスを継承させる必要はない。また C 言語の関数を完全に Haskell の関数に置き換えてしまったら foreign import は不要となり、直接 Haskell 関数同士を接続できることになる。

手を動かす前に自分がどんなコードを設計したいのか想像してみよう (図 2.6)。通常、可能な場面ではボトムアップ設計よりもトップダウン設計でやった方が効率が良いはずだ。この設計に必要なテクニックは 3 つに大別される。

1. スナッチ対象になっている C 言語関数の Haskell 化
2. Storable クラスで C 言語の構造体に Haskell の型を付ける
3. スナッチ対象が使用している C 言語の関数群を foreign import

一度のスナッチでどこから手をつけるべきかは場合による。トップダウンに設計したい場合には 1,2,3 の順番。ボトムアップに設計したい場合には 3,2,1 の順番だろう。今回のスナッチではスナッチ対象になっている C 言語の関数の規模も小さいのでトップダウン設計を採用することにする。C 言語の関数の規模が大きい場合には、まず既存の C 言語関数を foreign import してしてから C 言語の関数の中身を一度にスナッチするのではなく少しずつ Haskell 関数化している手もある。

[*7] Foreign.Storable.Storable `http://hackage.haskell.org/package/base/docs/Foreign-Storable.html`

[*8] 参考: `http://itpro.nikkeibp.co.jp/article/COLUMN/20080805/312151/?ST=develop&P=3`

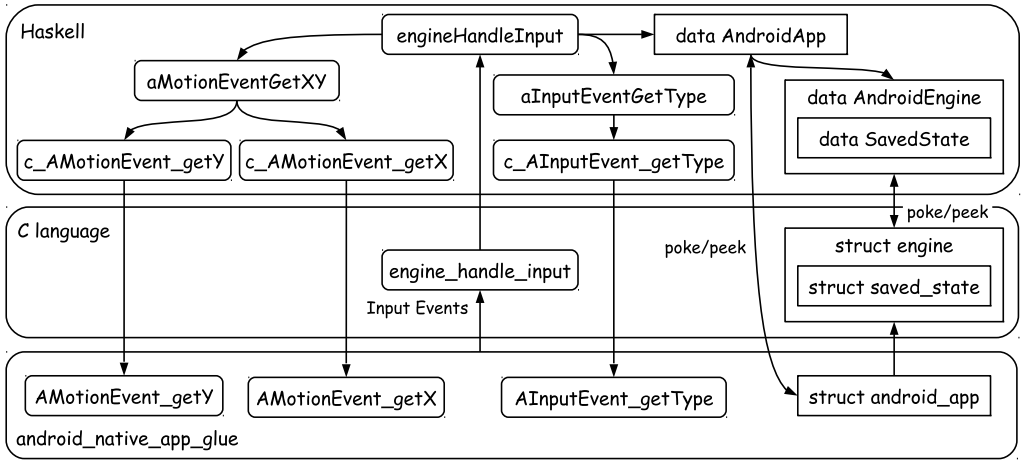

図 2.6: スナッチ後の engine_handle_input() 関数のイメージ

2.6.1 スナッチ対象になっている C 言語関数の Haskell 化

先の図 2.6 を参考にしてソースコードをトップダウンに書いてみよう。例えば hs_src/Main.hs はこんな感じだろうか。

```
{-# LANGUAGE ForeignFunctionInterface #-}
import Foreign.Ptr
import Foreign.Storable
import AndroidNdk

main = return ()

type FuncHandleInput =
  Ptr AndroidApp -> Ptr AInputEvent -> IO Int
foreign export ccall "engineHandleInput"
  engineHandleInput :: FuncHandleInput
engineHandleInput :: FuncHandleInput
engineHandleInput appP evP = aInputEventGetType evP >>= go
  where
    go AInputEventTypeMotion = do
      engP <- appUserData `fmap` peek appP
      peek engP >>= updateEngine >>= poke engP
      return 1
    go _ = return 0
    updateEngine eng = do
      (x, y) <- aMotionEventGetXY evP
      let stat = engState eng
      return $ eng { engAnimating = 1
                   , engState = stat { sStateX = truncate x,
```

```
                                        sStateY = truncate y }}
```

どー見ても C 言語の engine_handle_input() 関数をそのまま写しただけだ。スナッチ設計では C 言語から Haskell に変換している段階ではこれ以上抽象化を行なわない方が良い。いくらこの段階で抽象化したところで、まだこの関数の外部と C 言語のインターフェイスで繋ってしまっているため局所最適化になってしまうのだ。Haskell を使った抽象を楽しむのは完全にスナッチが終わった後に取っておくことにしよう。

この Haskell で書かれた engineHandleInput 関数は foreign export で C 言語から呼び出せるようになっている。main.c から呼び出すとしたらこんな感じだろうか。つまり engine_handle_input() 関数の中身がごっそり Haskell 化できることになる。

```
static int32_t engine_handle_input(struct android_app* app,AInputEvent* event){
    return engineHandleInput(app, event);
}
```

2.6.2 Storable クラスで C 言語の構造体に Haskell の型を付ける

さて、こんな実装に移行するためにはどうやって Storable を使えば良いだろうか。なんとなくそこそこ大きな実装になる予感がする。ちゃんとしたインターフェイスを切った方がいいだろう。まず hs_src/AndroidNdk.hs ファイルに AndroidNdk というモジュールを作ろう。

```
module AndroidNdk ( module AndroidNdk
                  , module AndroidNdk.Storable
                  , module AndroidNdk.Import ) where
import AndroidNdk.Storable
import AndroidNdk.Import
```

図 2.3 によると今回のサンプルアプリの下のレイヤーは android_native_app_glue と OpenGL ES のようだ。この AndroidNdk モジュールには android_native_app_glue 由来の要素を格納することにする。また AndroidNdk モジュールは NativeActivity を使った全てのアプリケーションを抽象化できるようには設計しない。現時点では NativeActivity のしくみとアプリケーション群からの要求が見えないからだ。AndroidNdk モジュールの中身は Storable と Import の二種類に分かれている。C 言語の構造体を Storable でラップするのが AndroidNdk.Storable モジュール。C 言語の関数を foreign import するのが AndroidNdk.Import モジュールだ。

まずなにはともあれデータ型、つまり構造体を Storable でラップしないことには関数を作れない。今回のサンプルアプリケーションで出てくる構造体は 3 つあるが、その包含関係は以下のようなっている。

```
struct android_app ∋ struct engine ∋ struct saved_state
```

ということは最初に Storable インスタンスにするべきなのは struct saved_state だろう。struct engine を Strorable インスタンスにしようとしても先に struct saved_state が Haskell から取り扱えねばならないからだ。hs_src/AndroidNdk/Storable.hs ファイルを作成して、SavedState 型を作ろう。

```
{-# LANGUAGE ForeignFunctionInterface #-}
module AndroidNdk.Storable (
  AndroidApp(..),
```

```
  AndroidEngine(..),
  SavedState(..)
  ) where
import Foreign.Storable
import Foreign.Ptr

-- struct saved_state
foreign import primitive "const.sizeof(struct saved_state)"
  sizeOf_SavedState :: Int
foreign import primitive "const.offsetof(struct saved_state, x)"
  offsetOf_SavedState_x :: Int
foreign import primitive "const.offsetof(struct saved_state, y)"
  offsetOf_SavedState_y :: Int
data SavedState =
  SavedState { sStateX :: Int
             , sStateY :: Int }
instance Storable SavedState where
  sizeOf    = const sizeOf_SavedState
  alignment = sizeOf
  poke p sstat = do
    pokeByteOff p offsetOf_SavedState_x $ sStateX sstat
    pokeByteOff p offsetOf_SavedState_y $ sStateY sstat
  peek p = do
    x <- peekByteOff p offsetOf_SavedState_x
    y <- peekByteOff p offsetOf_SavedState_y
    return $ SavedState { sStateX = x, sStateY = y }
```

ここで少し気になることがある。struct saved_state は以下のように C 言語側で定義されていた。

```
struct saved_state {
    float angle;
    int32_t x;
    int32_t y;
};
```

しかし今回作った SavedState 型ではこれら構造体メンバーの内で x と y のみ Haskell 側から使えるようにしている。angle メンバーは SavedState 型に含まれない。angle メンバーに有意な値が入っている struct saved_state へのポインタに SavedState 型を poke した場合、angle メンバーの内容が消えてしまうようなことはないのだろうか？ 結論から言うと使い方に気をつければ angle メンバーが消えることはない。図 2.7 を見てほしい。この図では stateC という C 言語構造体の実体に Haskell 側の SavedState 型のインスタンスである stateHs を poke している。SavedState 型では x と y メンバーに対応する sStateX/sStateY レコードしか定義されていない。そのため、この poke では stateC の x と y メンバーのみ更新されることになる。angle メンバーはそのまま残る。この Storable インスタンスの使い方で注意しなければいけないのは、struct saved_state のメンバー全てが SavedState 型に写されている訳ではないので一旦 C 言語型から渡ってきたポインタを peek したら、必ず同じ

ポインタに対して poke しないと Haskell 側で定義した SavedState 型において欠損しているメンバーは不定値になってしまうことだ。

ところでこのような問題があるのに構造体の一部しか Haskell の型をつけないのだろうか。いくつか理由があるな大きな理由は C 言語の関数を少しずつスナッチするのと同じように、C 言語の構造体メンバーの少しずつスナッチした方が安全だからだ。構造体の規模が大きい場合にはその全てのメンバーをスナッチしようとすると誤りが入ってしまう可能性が大きくなる。構造体メンバーの内の一部だけスナッチし、残ったメンバーは C 言語側でのみ取り扱うような設計をしたい場合今回のようなテクニックを使うことができる。本来であれば Ajhc に GHC の vector ライブラリ[*9] を移植して使うべきだろう。

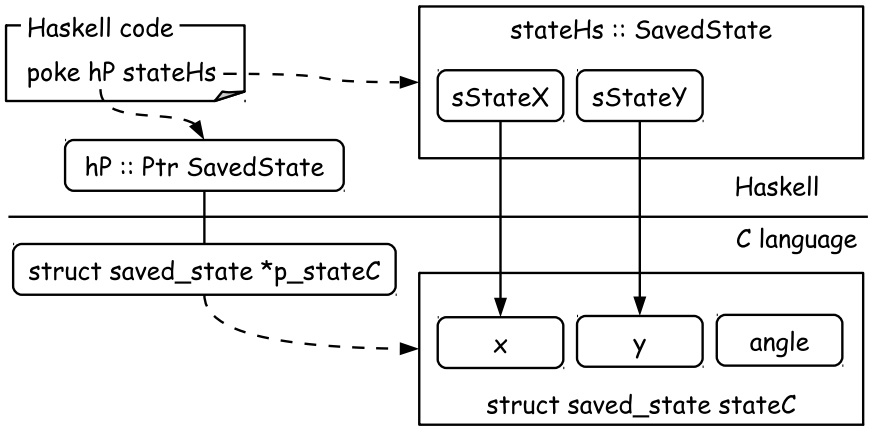

図 2.7: ポインタに対する部分的な poke

また foreign import primitive “const.HOGEHOGE” [*10]という見慣れないキーワードが見える。このキーワードは jhc 独自のもので、HOGEHOGE の位置にあるテキストをコンパイル結果の C 言語ソースコードに直接埋め込むキーワードだ。これは少し危険だが場面によっては大変重宝する。今回のスナッチでは struct saved_state のサイズと構造体先頭からメンバー変数へのオフセットを知るのに使っている。

[*9] http://hackage.haskell.org/package/vector

[*10] http://ajhc.metasepi.org/manual_ja.html#foreign プリミティブ

続いて struct android_app と struct engine も Storable インスタンスにしよう。hs_src/AndroidNdk/Storable.hs に追記する。

```
-- struct engine
foreign import primitive "const.sizeof(struct engine)"
  sizeOf_Engine :: Int
foreign import primitive "const.offsetof(struct engine, animating)"
  offsetOf_Engine_animating :: Int
foreign import primitive "const.offsetof(struct engine, state)"
  offsetOf_Engine_state :: Int
data AndroidEngine =
  AndroidEngine { engAnimating :: Int
                , engState     :: SavedState }
instance Storable AndroidEngine where
  sizeOf    = const sizeOf_Engine
  alignment = sizeOf
  poke p eng = do
    pokeByteOff p offsetOf_Engine_animating $ engAnimating eng
    pokeByteOff p offsetOf_Engine_state     $ engState eng
  peek p = do
    animating <- peekByteOff p offsetOf_Engine_animating
    state     <- peekByteOff p offsetOf_Engine_state
    return $ AndroidEngine { engAnimating = animating
                           , engState     = state }
-- struct android_app
foreign import primitive "const.sizeof(struct android_app)"
  sizeOf_AndroidApp :: Int
foreign import primitive "const.offsetof(struct android_app, userData)"
  offsetOf_AndroidApp_appUserData :: Int
data AndroidApp = AndroidApp { appUserData :: Ptr AndroidEngine }
instance Storable AndroidApp where
  sizeOf    = const sizeOf_AndroidApp
  alignment = sizeOf
  poke p app = do
    pokeByteOff p offsetOf_AndroidApp_appUserData $ appUserData app
  peek p = do
    userData <- peekByteOff p offsetOf_AndroidApp_appUserData
    return $ AndroidApp { appUserData = userData }
```

2.6.3 スナッチ対象が使用している C 言語の関数群を foreign import

C 言語の構造体に対する Storable インスタンスが完成したので、今度は C 言語の関数群を hs_src/AndroidNdk/Import.hs に foreign import しよう。

```
{-# LANGUAGE ForeignFunctionInterface #-}
module AndroidNdk.Import where
```

```
import Foreign.Ptr
import Foreign.C.Types

data AInputEventType =
  AInputEventTypeKey | AInputEventTypeMotion | AInputEventTypeFail
  deriving(Eq)
newtype {-# CTYPE "AInputEvent" #-} AInputEvent = AInputEvent ()

foreign import primitive "const.AINPUT_EVENT_TYPE_KEY"
  c_AINPUT_EVENT_TYPE_KEY :: Int
foreign import primitive "const.AINPUT_EVENT_TYPE_MOTION"
  c_AINPUT_EVENT_TYPE_MOTION :: Int
foreign import ccall "c_extern.h AInputEvent_getType"
  c_AInputEvent_getType :: Ptr AInputEvent -> IO Int
foreign import ccall "c_extern.h AMotionEvent_getX"
  c_AMotionEvent_getX :: Ptr AInputEvent -> CSize -> IO Float
foreign import ccall "c_extern.h AMotionEvent_getY"
  c_AMotionEvent_getY :: Ptr AInputEvent -> CSize -> IO Float
```

CTYPE ”AInputEvent”という型プラグマ[*11]を使った。このプラグマは data や newtype で定義した型に対して C 言語側から見た型を付与する。今回のサンプルアプリケーションの中では AInputEvent という C 言語の型の中身をさわらず、AInputEvent へのポインタをとりまわすだけだ。そんな時はいちいち Storable 型を定義する必要はなく上記の AInputEvent 型のように中身のない型を定義して CTYPE 型プラグマを使えばいい。

このままでもスナッチは継続できるだろう。しかし foreign import しただけの関数は生の C 言語 API のままなので使いにくく、また型安全からは遠い。そこで簡単なラッパーを hs_src/AndroidNdk/Import.hs に追加して強い型を付けよう。

```
aInputEventGetType :: Ptr AInputEvent -> IO AInputEventType
aInputEventGetType p = do
  i <- c_AInputEvent_getType p
  return $ inputEventType i

inputEventType :: Int -> AInputEventType
inputEventType e | e == c_AINPUT_EVENT_TYPE_KEY    = AInputEventTypeKey
                 | e == c_AINPUT_EVENT_TYPE_MOTION = AInputEventTypeMotion
                 | otherwise = AInputEventTypeFail

aMotionEventGetXY :: Ptr AInputEvent -> IO (Float, Float)
aMotionEventGetXY ep = do
  x <- c_AMotionEvent_getX ep 0
  y <- c_AMotionEvent_getY ep 0
  return (x, y)
```

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

最後にさっきから出てきていた c_extern.h へッダに main.c の中にあった include 行と構造体定義を移動させ、main.c から include する。つまり構造体と関数インターフェイス定義を main.c と Haskell コードで共有するのだ (図 2.8)。これで、OpenGL ES の API も Haskell 側からたたけるようになったはずだ。まだこの段階ではその要求はないけれど。

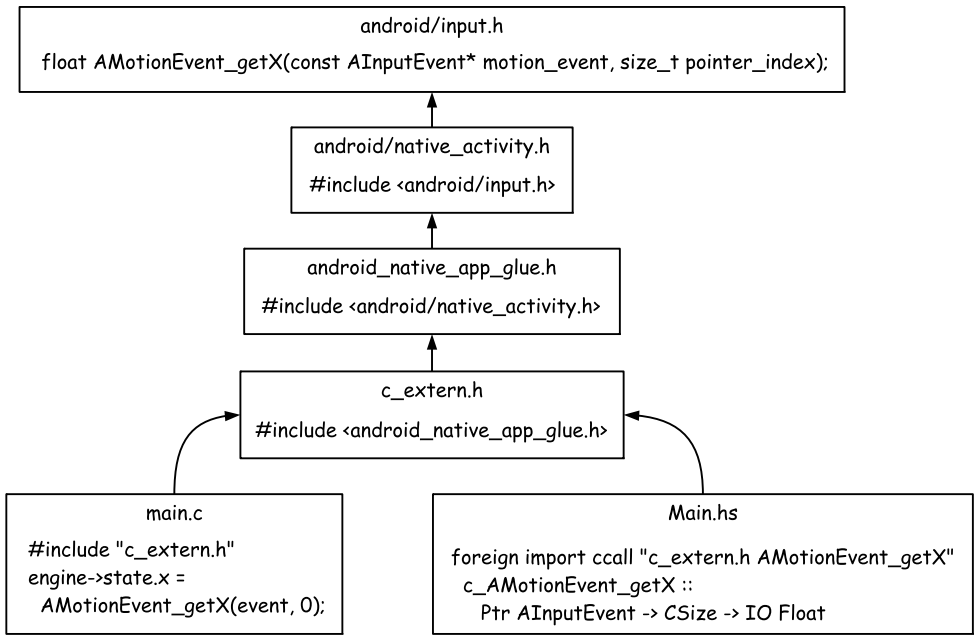

図 2.8: c_extern.h へッダで API 定義を C と Haskell で共有

今回のスナッチはこれで一区切りだ。今回のスナッチの途中でもビルド確認はしてきたが、スナッチの終わりビルドと動作確認をしておこう。やはり Android デバイス上で正常に動作しているように見える。次のスナッチに進もう。

2.7　スナッチ 3: Haskell コードの近傍をさらにスナッチ

さっきは C 言語の engine_handle_input() 関数の中身をスナッチした。しかし C 言語のエントリ関数から Haskell のコードを呼び出したままだと、main.c はスナッチが完了してもエントリ関数が残ったままになってしまう。せっかくなら engine_handle_input() 関数そのものを消してしまいたい。どうしたら良いだろう？ この関数を使っているコードはどこだろうか。

```
void android_main(struct android_app* state) {
// -- snip --
    memset(&engine, 0, sizeof(engine));
    state->userData = &engine;
    state->onAppCmd = engine_handle_cmd;
    state->onInputEvent = engine_handle_input;
    engine.app = state;
```

なるほど。android_main() 関数にて engine_handle_input() の関数ポインタを struct android_app メ

ンバーに代入している。この部分を Haskell 化することはできないだろうか。ここでもトップダウン設計をしてみよう。単純に考えると struct android_app へのポインタを Haskell に渡して onInputEvent メンバーを更新してもらえばいいことになる。

```
$ git diff jni/main.c
diff --git a/jni/main.c b/jni/main.c
index d3a037d..711f3bc 100644
--- a/jni/main.c
+++ b/jni/main.c
@@ -205,7 +198,7 @@ void android_main(struct android_app* state) {
     memset(&engine, 0, sizeof(engine));
     state->userData = &engine;
     state->onAppCmd = engine_handle_cmd;
-    state->onInputEvent = engine_handle_input;
+    initAndroidAppFunc(state);
     engine.app = state;

     // Prepare to monitor accelerometer
```

こんな initAndroidAppFunc 関数はどうなるだろうか？ hs_src/Main.hs ファイルに定義を追加しよう。

```
foreign export ccall "initAndroidAppFunc"
  initAndroidAppFunc :: Ptr AndroidApp -> IO ()
initAndroidAppFunc :: Ptr AndroidApp -> IO ()
initAndroidAppFunc appP = peek appP >>= updateApp >>= poke appP
  where
    updateApp app = return $ app { appOnInputEvent = p_engineHandleInput }
```

問題は p_engineHandleInput というモノだ。これは FunPtr 型にすべきだがどうやって作ればいいだろうか？ GHC であれば foreign import ccall ”wrapper”を使うだろう[*12]。ところが現状の Ajhc では wrapper にはバグがあり使えないようだ[*13] [*14]。しょうがない。手で関数ポインタを作ろう。hs_src/Main.hs を編集する。

```
$ git diff hs_src/Main.hs
diff --git a/hs_src/Main.hs b/hs_src/Main.hs
index beb330a..75e0af1 100644
--- a/hs_src/Main.hs
+++ b/hs_src/Main.hs
@@ -5,10 +5,10 @@ import AndroidNdk

 main = return ()
```

[*12] http://hackage.haskell.org/package/base/docs/Foreign-Ptr.html#t:FunPtr

[*13] FFI functions declared ”wrapper” are not supported - Issue #64 https://github.com/ajhc/ajhc/issues/64

[*14] みんながこの記事を読んでいるころまでに対策できていたら素敵でゲソ! (げふぅ!)

```
-type FuncHandleInput =
-  Ptr AndroidApp -> Ptr AInputEvent -> IO Int
 foreign export ccall "engineHandleInput"
   engineHandleInput :: FuncHandleInput
+foreign import ccall "&engineHandleInput"
+  p_engineHandleInput :: FunPtr FuncHandleInput
 engineHandleInput :: FuncHandleInput
 engineHandleInput appP evP = aInputEventGetType evP >>= go
   where
```

一度 export することで”engineHandleInput” という名前のシンボルを作り、再度関数ポインタを import する。なにか自作自演っぽいが、wrapper が使えないのでしょうがない。

次に onInputEvent メンバーを Haskell から取り扱えるようにするために Storable インスタンスを変更しよう。関数の型定義である FuncHandleInput エイリアスは Main.hs でも Storable.hs でも使うので、Main.hs から Storable.hs に移動させる。

```
$ git diff hs_src/AndroidNdk/Storable.hs
diff --git a/hs_src/AndroidNdk/Storable.hs b/hs_src/AndroidNdk/Storable.hs
index 58d5db0..8f259cb 100644
--- a/hs_src/AndroidNdk/Storable.hs
+++ b/hs_src/AndroidNdk/Storable.hs
@@ -2,10 +2,15 @@
 module AndroidNdk.Storable (
   AndroidApp(..),
   AndroidEngine(..),
-  SavedState(..)
+  SavedState(..),
+  FuncHandleInput
   ) where
 import Foreign.Storable
 import Foreign.Ptr
+import AndroidNdk.Import
+
+type FuncHandleInput =
+  Ptr AndroidApp -> Ptr AInputEvent -> IO Int

 -- struct saved_state
 foreign import primitive "const.sizeof(struct saved_state)"
```

最後に AndroidApp 型に onInputEvent メンバーに対応する appOnInputEvent レコードを追加しよう。

```
$ git diff hs_src/AndroidNdk/Storable.hs
diff --git a/hs_src/AndroidNdk/Storable.hs b/hs_src/AndroidNdk/Storable.hs
index 58d5db0..8f259cb 100644
```

```
--- a/hs_src/AndroidNdk/Storable.hs
+++ b/hs_src/AndroidNdk/Storable.hs
@@ -55,12 +60,18 @@ foreign import primitive "const.sizeof(struct android_app)"
   sizeOf_AndroidApp :: Int
 foreign import primitive "const.offsetof(struct android_app, userData)"
   offsetOf_AndroidApp_appUserData :: Int
-data AndroidApp = AndroidApp { appUserData :: Ptr AndroidEngine }
+foreign import primitive "const.offsetof(struct android_app, onInputEvent)"
+  offsetOf_AndroidApp_appOnInputEvent :: Int
+data AndroidApp = AndroidApp { appUserData :: Ptr AndroidEngine
+                             , appOnInputEvent :: FunPtr FuncHandleInput }
 instance Storable AndroidApp where
   sizeOf    = const sizeOf_AndroidApp
   alignment = sizeOf
   poke p app = do
     pokeByteOff p offsetOf_AndroidApp_appUserData $ appUserData app
+    pokeByteOff p offsetOf_AndroidApp_appOnInputEvent $ appOnInputEvent app
   peek p = do
     userData <- peekByteOff p offsetOf_AndroidApp_appUserData
-    return $ AndroidApp { appUserData = userData }
+    onInput <- peekByteOff p offsetOf_AndroidApp_appOnInputEvent
+    return $ AndroidApp { appUserData = userData
+                        , appOnInputEvent = onInput }
```

今回のスナッチは簡単な修正だった。それでも C 言語の関数が一つ消えたのは気持ちがいい。やはり図にまとめて今回のスナッチをふりかえってみよう (図 2.9)。中身だけスナッチ済みだった engine_handle_input() 関数全体を Haskell 化によって C 言語側から消してしまうために当該関数の使用者である android_main() 関数に変更を加えた。結果として android_native_app_glue から直接 Haskell のコールバック関数が呼び出されるようになった。

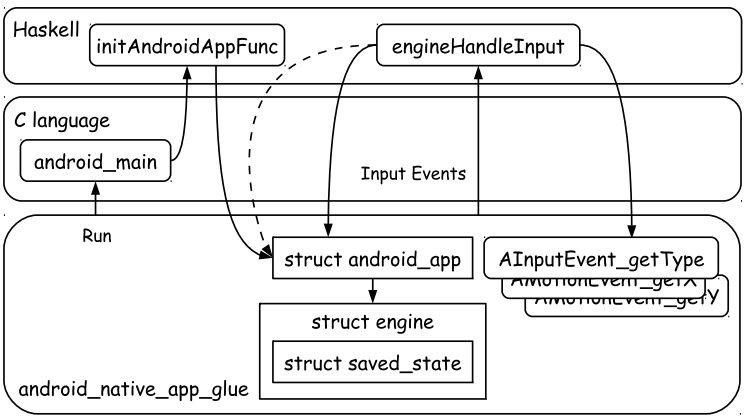

図 2.9: onInputEvent 関数ポインタの初期化をスナッチした結果

簡単な修正だったとはいえ、スナッチの終わりにはビルド確認と動作確認をするべきだ。もしここでエラーが起きていたら今回のスナッチによって不具合が入りこんだことになる。念の為の動作確認だったが、もちろん Android デバイス上での動作は正常に見える。これで今回のスナッチは完了だ。

2.8 こっそりと驚きを届けたい

『さいきん家でゴソゴソなにをやってるんでゲソ？』
「ん。別に」
『めずらしく ThinkPad 開いてるじゃなイカ。Coq の宿題でも出たんでゲソ？』
「そんなんじゃないよ」
『あやしいでげそ……おもしろいことを独り占めするの良くないでゲソ!』
「後でちゃんと説明するよ」

ここでプレゼントを作っているのがバレてしまっては面白さが半減だ。もうちょっと開発速度を上げた方がいいのかもしれない。

2.9 スナッチ 4: OpenGL ES をたたく

さて次はどこをスナッチしようか。engine_handle_input() 関数の近傍のスナッチは終わった。initAndroidAppFunc 関数の近傍のスナッチに進んでも良いが、ここは C 言語側で最も長い android_main() 関数だ。いきなりここを攻めるのは無謀だろう。他に短い C 言語関数を探してみると engine_draw_frame() が見つかった。

```
static void engine_draw_frame(struct engine* engine) {
    if (engine->display == NULL) {
        // No display.
        return;
    }

    // Just fill the screen with a color.
    glClearColor(((float)engine->state.x)/engine->width, engine->state.angle,
            ((float)engine->state.y)/engine->height, 1);
    glClear(GL_COLOR_BUFFER_BIT);

    eglSwapBuffers(engine->display, engine->surface);
}
```

この関数はフレーム毎の描画を OpenGL ES を使って行なう。glClearColor()、glClear()、eglSwapBuffers() という 3 つの関数が見えるが、これらは OpenGL ES の API だ。OpenGL ES のスナッチをするわけでないので、これらの関数は完全にスナッチされた後も C 言語の関数のまま残るだろう。これらの関数を hs_src/OpenGLES.hs に import しよう。

```
{-# LANGUAGE ForeignFunctionInterface #-}
module OpenGLES where
import Data.Word
import Data.Bits
import Foreign.Ptr

type C_GLbitfield = Word32
newtype EGLDisplayT = EGLDisplayT ()
```

```
newtype EGLSurfaceT = EGLSurfaceT ()
type EGLDisplay = Ptr EGLDisplayT
type EGLSurface = Ptr EGLSurfaceT

foreign import primitive "const.GL_COLOR_BUFFER_BIT"
  c_GL_COLOR_BUFFER_BIT :: C_GLbitfield
foreign import primitive "const.GL_DEPTH_BUFFER_BIT"
  c_GL_DEPTH_BUFFER_BIT :: C_GLbitfield
foreign import primitive "const.GL_STENCIL_BUFFER_BIT"
  c_GL_STENCIL_BUFFER_BIT :: C_GLbitfield
foreign import ccall "GLES/gl.h glClearColor"
  c_glClearColor :: Float -> Float -> Float -> Float -> IO ()
foreign import ccall "GLES/gl.h glClear"
  c_glClear :: C_GLbitfield -> IO ()
foreign import ccall "EGL/egl.h eglSwapBuffers"
  c_eglSwapBuffers :: EGLDisplay -> EGLSurface -> IO Word32
```

このままでも Haskell でコードを書けるが、C_GLbitfield 型が生の 32bit 値であるので型が弱い。そこで GLBitField 型を作り、[GLBitField] 型でビット指定できるようにしておこう。こうすればビットの意味をあやまって理解してしまうことを防げるはずだ。もっとも今回のスナッチでは gLColorBufferBit しか使わないので combineGLBitField は使うことはない。それでも明らかに型が弱い部分はその都度抽象化をしてやった方がいいだろう。hs_src/OpenGLES.hs に定義を追加する。

```
newtype GLBitField = GLBitField { unGLBitField :: Word32 }
gLColorBufferBit, gLDepthBufferBit, glStenciBufferBit :: GLBitField
gLColorBufferBit  = GLBitField c_GL_COLOR_BUFFER_BIT
gLDepthBufferBit  = GLBitField c_GL_DEPTH_BUFFER_BIT
glStenciBufferBit = GLBitField c_GL_STENCIL_BUFFER_BIT
combineGLBitField :: [GLBitField] -> GLBitField
combineGLBitField = GLBitField . foldr ((.|.) . unGLBitField) 0

glClear :: GLBitField -> IO ()
glClear = c_glClear . unGLBitField

glClearColor :: Float -> Float -> Float -> Float -> IO ()
glClearColor = c_glClearColor

eglSwapBuffers :: EGLDisplay -> EGLSurface -> IO Word32
eglSwapBuffers = c_eglSwapBuffers
```

これで OpenGL ES の API を Haskell からたたけるようになった。engine_draw_frame() のスナッチしてみよう。

```
$ git diff hs_src/Main.hs
diff --git a/hs_src/Main.hs b/hs_src/Main.hs
```

```
index 75e0af1..9527e98 100644
--- a/hs_src/Main.hs
+++ b/hs_src/Main.hs
@@ -1,7 +1,9 @@
 {-# LANGUAGE ForeignFunctionInterface #-}
+import Control.Monad
 import Foreign.Ptr
 import Foreign.Storable
 import AndroidNdk
+import OpenGLES

 main = return ()

@@ -30,3 +32,19 @@ initAndroidAppFunc :: Ptr AndroidApp -> IO ()
 initAndroidAppFunc appP = peek appP >>= updateApp >>= poke appP
   where
     updateApp app = return $ app { appOnInputEvent = p_engineHandleInput }
+
+foreign export ccall "engineDrawFrame"
+  engineDrawFrame :: Ptr AndroidEngine -> IO ()
+engineDrawFrame :: Ptr AndroidEngine -> IO ()
+engineDrawFrame engP = do
+  eng <- peek engP
+  when ((engEglDisplay eng) /= nullPtr) $ do
+    let w     = fromIntegral $ engWidth eng
+        h     = fromIntegral $ engHeight eng
+        s     = engState eng
+        x     = fromIntegral $ sStateX s
+        y     = fromIntegral $ sStateY s
+        angle = sStateAngle s
+    glClearColor (x/w) angle (y/h) 1.0
+    glClear $ gLColorBufferBit
+    void $ eglSwapBuffers (engEglDisplay eng) (engEglSurface eng)
```

SavedState に angle を AndroidEngine に width,height,display,surface を追加しよう。

```
$ git diff hs_src/AndroidNdk/Storable.hs
diff --git a/hs_src/AndroidNdk/Storable.hs b/hs_src/AndroidNdk/Storable.hs
index 8f259cb..cfe24c9 100644
--- a/hs_src/AndroidNdk/Storable.hs
+++ b/hs_src/AndroidNdk/Storable.hs
@@ -8,6 +8,7 @@ module AndroidNdk.Storable (
 import Foreign.Storable
 import Foreign.Ptr
 import AndroidNdk.Import
+import OpenGLES

 type FuncHandleInput =
   Ptr AndroidApp -> Ptr AInputEvent -> IO Int
@@ -19,19 +20,24 @@ foreign import primitive "const.offsetof(struct saved_s...
   offsetOf_SavedState_x :: Int
 foreign import primitive "const.offsetof(struct saved_state, y)"
   offsetOf_SavedState_y :: Int
+foreign import primitive "const.offsetof(struct saved_state, angle)"
+  offsetOf_SavedState_angle :: Int
 data SavedState =
   SavedState { sStateX :: Int
-             , sStateY :: Int }
+             , sStateY :: Int
+             , sStateAngle :: Float }
 instance Storable SavedState where
   sizeOf    = const sizeOf_SavedState
   alignment = sizeOf
   poke p sstat = do
     pokeByteOff p offsetOf_SavedState_x $ sStateX sstat
     pokeByteOff p offsetOf_SavedState_y $ sStateY sstat
+    pokeByteOff p offsetOf_SavedState_angle $ sStateAngle sstat
   peek p = do
     x <- peekByteOff p offsetOf_SavedState_x
     y <- peekByteOff p offsetOf_SavedState_y
-    return $ SavedState { sStateX = x, sStateY = y }
+    angle <- peekByteOff p offsetOf_SavedState_angle
+    return $ SavedState { sStateX = x, sStateY = y, sStateAngle = angle }

 -- struct engine
 foreign import primitive "const.sizeof(struct engine)"
@@ -40,20 +46,44 @@ foreign import primitive "const.offsetof(struct engine...
   offsetOf_Engine_animating :: Int
 foreign import primitive "const.offsetof(struct engine, state)"
```

```
   offsetOf_Engine_state :: Int
+foreign import primitive "const.offsetof(struct engine, width)"
+  offsetOf_Engine_width :: Int
+foreign import primitive "const.offsetof(struct engine, height)"
+  offsetOf_Engine_height :: Int
+foreign import primitive "const.offsetof(struct engine, display)"
+  offsetOf_Engine_display :: Int
+foreign import primitive "const.offsetof(struct engine, surface)"
+  offsetOf_Engine_surface :: Int
 data AndroidEngine =
   AndroidEngine { engAnimating :: Int
-                , engState     :: SavedState }
+                , engState     :: SavedState
+                , engWidth      :: Int
+                , engHeight     :: Int
+                , engEglDisplay :: EGLDisplay
+                , engEglSurface :: EGLSurface }
 instance Storable AndroidEngine where
   sizeOf    = const sizeOf_Engine
   alignment = sizeOf
   poke p eng = do
     pokeByteOff p offsetOf_Engine_animating $ engAnimating eng
     pokeByteOff p offsetOf_Engine_state     $ engState eng
+    pokeByteOff p offsetOf_Engine_width     $ engWidth eng
+    pokeByteOff p offsetOf_Engine_height    $ engHeight eng
+    pokeByteOff p offsetOf_Engine_display   $ engEglDisplay eng
+    pokeByteOff p offsetOf_Engine_surface   $ engEglSurface eng
   peek p = do
     animating <- peekByteOff p offsetOf_Engine_animating
     state     <- peekByteOff p offsetOf_Engine_state
+    width               <- peekByteOff p offsetOf_Engine_width
+    height              <- peekByteOff p offsetOf_Engine_height
+    eglDisp             <- peekByteOff p offsetOf_Engine_display
+    eglSurf             <- peekByteOff p offsetOf_Engine_surface
     return $ AndroidEngine { engAnimating = animating
-                           , engState     = state }
+                           , engState     = state
+                           , engWidth      = width
+                           , engHeight     = height
+                           , engEglDisplay = eglDisp
+                           , engEglSurface = eglSurf }

 -- struct android_app
 foreign import primitive "const.sizeof(struct android_app)"
```

ここまででわかる通り、Storable の管理は単純作業だ。設計対象によっては自動化できるかもしれない。

最後に C 言語の engine_draw_frame() 関数から Haskell の engineDrawFrame 関数を呼び出せばスナッチは終わりだ。

```
static void engine_draw_frame(struct engine* engine) {
    engineDrawFrame(engine);
}
```

簡単なスナッチだったが図にまとめておこう (図 2.10)。今回はサンプルアプリケーションのスナッチなので、OpenGL ES の中まではスナッチしない。しかし C 言語で書かれた OpenGL ES と Haskell のコードの共存は可能なようだ。

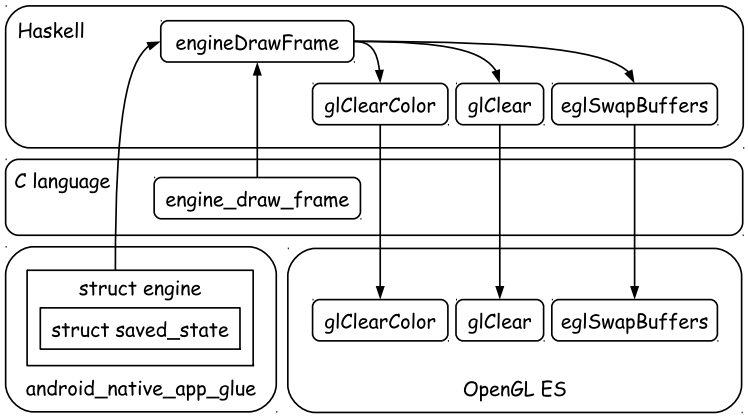

図 2.10: engine_draw_frame() 関数をスナッチした結果

今回のスナッチでは新しい部品である OpenGL ES の呼び出しを Haskell 化した。スナッチに失敗していた場合には画面描画がおかしくなるはずだ。スナッチの完了前にビルドと動作確認をしてみよう。Android デバイスにインストールしての動作確認は正常。画面描画にも問題は見られない。これで次のスナッチに安心して進むことができる。

2.10　スナッチ完了

その後数日かけて僕はサンプルアプリケーションのスナッチを続けてみた。たしかに作業に時間はかかったが、スナッチに使ったテクニック自体はこれまで使ってきたものを応用すればなんとかなったようだ。最終的に main.c をこれ以上 Haskell 化できないところまでスナッチを進行させることができた。その結果 main.c には何が残ったのだろうか？

```
void android_main(struct android_app* state) {
    app_dummy(); // Make sure glue isn't stripped.

    // Init & run Haskell code.
    int hsargc = 1;
    char *hsargv = "q";
    char **hsargvp = &hsargv;
```

```
    hs_init(&hsargc, &hsargvp);
    androidMain(state);
    hs_exit();
}
```

もはやこの C 言語のエントリコードは有意なロジックとは呼べない。以下 3 つのアクションを順番に呼び出しているにすぎないからだ。Android NDK サンプルソースコードは完全にスナッチされたと考えていいだろう。つまり図 2.1 の Step 5 まで到達できたことになる。

- hs_init(): Ajhc ランタイムの初期化
- androidMain(): Haskell エントリ関数の呼び出し
- hs_exit(): Ajhc ランタイムの終了

2.11 プレゼントを作ろう

せっかくスナッチしたので Android NDK フレームワークのようなものを作ってみよう (図 2.11)。結局 C 言語と Haskell の界面であるコールバックが煩雑な処理になりやすいので、その部分を NDK framework として隠した。この抽象化が最善だとは思えないが、そこは型の強い言語、後で再設計するのはそれほど大変ではない。

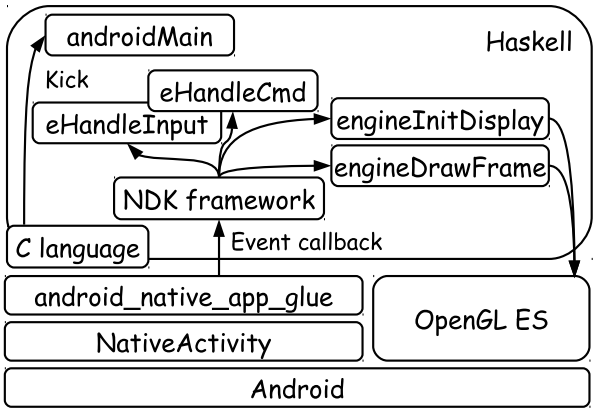

図 2.11: Haskell を使った Android NDK フレームワーク

このスナッチ済みサンプルアプリケーションを改造して何か面白いアプリは作れないだろうか。結局、先のサンプルはイベントをコールバックで受け取って、メインコンテキストで OpenGL ES を使って画面描画をするだけだ。タッチパネルで立方体を回転させるぐらいであれば、それほど苦労しないはずだ。既にスナッチは完了しているので、今回の改造は純粋な Haskell プログラミングで済んだ (図 2.12)。

単純なアプリだがタッチ UI で立方体で動かすのは少し不思議な気分がする。カノジョはこのアプリ[*15] を喜んでくれるだろうか。その様子を想像して、僕は今度の週末が楽しみになった。

2.12 お互いを理解することは小さく傷付け合うこと

ほんとうに簡単なサンプルアプリケーションをスナッチしただけだが、C 言語によるイベントドリブンな設計について良い知見が得られた。イベントドリブンのような Haskell で設計しにくい対象にも FFI を使って少しずつスナッチ設計をすることは可能だった。またスナッチのどの段階においても元のアプリケーションの挙動を確認することができた。完全にスナッチが終わった後は単なる Haskell プログラムとしての再設計することができた。今回はアプリケーションを再設計することで Android NDK を簡単なフレームワークで管理するようにして、新しい立方体アプリケーションにまで成長させることができた。つまり完全なスナッチが終わった後は Haskell 言語の設計効率を

[*15] https://play.google.com/store/apps/details?id=org.metasepi.ajhc.android.cube

活用できるわけだ。もちろんスナッチ後のアプリケーションコードには強い型が付いているため、再設計コストも見積りやすい。

しかしコンピュータアーキティクチャの全てを Haskell でスナッチすることがうれしいかは疑問だ。例えばバーチャルメモリのページデーモンではコンストラクタや GC を使ってはならないことは明白だ。ということは全てを Haskell で書き直せるか？ という問題については既に反例が見つかっているわけだ。NetBSD kernel のような巨大なアプリケーションでも C 言語で記述すべき部分と Haskell で楽に設計できる部分とが同居することだろう。

……ひょっとすると僕たち生物も同じなのかもしれない。スナッチとは相手を理解するために相手を少し傷付けて、相手の一部を自分の内に取り込むということでもある。お互いを小さく傷付け合いながら、お互いを少しずつ理解する。それが……興味を持つとか、愛するということなのかもしれない。

僕は ThinkPad の画面を閉じた。

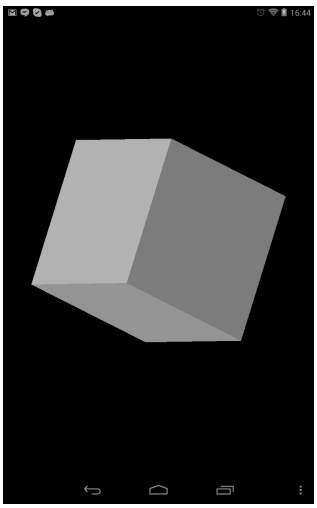

図 2.12: 立方体回転アプリ

2.13　それでも明日はやって来る

「どうして僕の左足をスナッチしちゃったの!？」
『ごめんでゲソ! ごめんでゲソ! ついあんまりにも面白くて……』
「これじゃ明日大学行けないじゃないか」
『明日の朝までに絶対元に戻すでゲソ。だから心配は不要じゃなイカ？』
「そういう問題じゃぁ……もういいよ。先に寝るね。おやすみ」

もうちょっとで眠れそうだったのだが、カノジョが布団に入ってきた。僕の背中にカノジョの温もりが伝わってくる。少し震えているようだ。僕はカノジョのてのひらに小さく口づけしてあげた。

「おやすみ。また明日ね」

2.14　ソースコードライセンス

引用した Android NDK の native-activity サンプルソースコードは以下のランセンスに服します。

```
* Copyright (C) 2010 The Android Open Source Project
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*      http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
```

2.15 参考文献

- 『フリクリ』（FLCL） - ガイナックス・Production I.G
- 『スナッチャー』（SNATCHER） - コナミ（現・コナミデジタルエンタテインメント）
- 『ONE LIFE』（ワン・ライフ） - the pillows
- 今回の記事で途中までスナッチした Android NDK サンプルアプリケーションソースコード[*16]
- タッチパネルで立方体を回転させる Android NDK アプリケーションソースコード[*17]
- Real World Haskell – Chapter 17. Interfacing with C: the FFI [*18]
- Metasepi team meeting #6: ”Snatch-driven development” [*19]
- Metasepi team meeting #8’: Haskell apps on Android NDK [*20]

[*16] https://github.com/master-q/trysnatch_androidndk
[*17] https://github.com/ajhc/demo-android-ndk
[*18] http://book.realworldhaskell.org/read/interfacing-with-c-the-ffi.html
[*19] http://www.slideshare.net/master_q/20131020-osc-tokyoajhc
[*20] http://www.slideshare.net/master_q/20131116-osc-fukuokaajhcdroid

会員名簿じゃなイカ?

@ark_golgo　（1 章）

　調子に乗って今回も原稿を出させていただきました。論文もこの調子で書けたらいいんですけどねえ…。

@master_q　(2 章)

　「mbed マイコンではじめる Haskell 講習会」[21] やりませんか？

@dif_engine　(表紙)

·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)(S(S(KS)K)M(S(S(KS)K)I)(S(S(KS)K)(S(S(KS)K)I))))·S(S(KS)K)(S(K(S(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))))))·

「簡約！？ λ カ娘 Rock!」

2013 年 12 月 31 日 コミックマーケット 85 初版発行

2014 年 5 月 7 日 第 2 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：@ark_golgo, @master_q

表紙：@dif_engine

印刷：ちょ古っ都製本工房 様 http://www.chokotto.jp/

---

[21] http://partake.in/events/ab56454b-c305-4f3b-b8ce-872871ab7da9