ヤバイデッカ

トーキョー vol.4

zeptometer irnbru MasWag
mh wasabiz censored gfn

YABAITECH.TOKYO

vol.4

2020

目次

忙しくないプログラマのための
ゲーム紹介記事

mh

1. はじめに

みなさん、ゲームしてますか？人生はゲームみたいなものという意見もあるでしょうが、ここではコンピュータゲーム一般（据置型ハードやスマホゲームを含む）を指すことにしましょう。去年は何本くらいゲームをプレイしましたか？ストーリーはクリアしました？やり込み要素まで手をつけていますか？最近気づき始めたのですが、人々はあまりゲームをしないんですね。「有名タイトルをストーリークリアまで」というのはまだ良い方で、「スマホゲーを一つ」とか「プログラム書いてる方が楽しい」という人の方が余程多い。それともそれが大人になるってことなのかな。悲しいですね。もちろんゲームというのは据置機大御所ビッグタイトルから個人制作のブラウザゲームまで毎年山ほど制作されているわけで、しかもそれが自分にとって楽しいかどうかも多少なりプレイしてみなければわからないですから、有名な / 話題に上がったものをピックアップして遊ぶ、楽しくやり込めるものを一つ見つけるというのは理にかなった話ではあります。とはいえそれでは一ゲーマーとしてあまりに悲しい、みなさんにもっと非有名ゲームを遊んで欲しい、小規模制作会社の高品質なゲームを楽しんでほしい、あわよくばインディーズゲーム沼に嵌ってほしい、クソゲーを踏んで辛くなってほしい、むしろ自分から踏み抜きに行くようになってほしい。そんな気持ちで本記事はできています。

というわけで本記事ではゲームタイトルの紹介を行なっていきます。ヤバイ"テック"？関係ありません。技術的な内容は一切ありません。とはいえあまりにテーマから離れると私がサークル内で浮いてしまう恐れがありますから、想定読者をプログラマのみなさんと置きまして、多少でも興味を持ってもらえるように内容がプログラミングであるものを中心にタイトルを選択

していきます。ここで「内容が」プログラミングであると書きましたが、これは端的にはゲーム中で定義される独自の言語を用いて、与えられた入出力を満たすプログラムを記述することを目標としていることを指しています。基本的にはパズルゲームに分類されるでしょう。おかしな言語を指定してくる競技プログラミング、と形容すればわかりやすいでしょうか。もちろん何をどうすればいいかはゲーム内で説明されますから、前提知識なしで大丈夫、一般の方でも十分楽しめると思います。一つでも目に留まるタイトルがあればなによりです。

2. まえおき

本題に入る前にどうやればそれらのゲームを手に入れられるのかについて言及しておきます。本記事では基本的に PC でプレイするゲームを対象とするので、PC 上にそのための環境を構築することになります。とは言っても特に言及しておかねばならないのは Steam[*1]くらいでしょう。ご存知の方も多いだろうと思いますが、Steam は Valve co. によって運用される PC ゲームプラットフォームの最大手の一つです。アカウントを登録し公式アプリをインストールしてしまえば、そこを起点にゲームの検索・購入・インストールから実行まで行えるので迷うことはあまりないと思います。インディーズを含む数多くのゲームが配信されており、PC でゲームをするならアカウントを作成しておいて間違いないでしょう。より詳しい情報は公式ページなり Wikipedia なりを参照していただければと思います。

Steam 以外で紹介するのはいわゆるブラウザゲームになります。普段お使いのウェブブラウザからアクセスするだけでプレイできますから、一部 Flash が必要なものがあるくらいで特に準備は必要ありません。またこちらの方は無料プレイのものが多くなります。ありがたいですね。ジャンル自体が気にいるかわからない、とりあえず試しにプレイしてみたいという方はそちらから入るのも良いでしょう。

3. ほんだい

さあ本題に入りましょう。まずは（比較的）有名そうなところからです。

1 https://store.steampowered.com

Human Resource Machine

Human Resource Machine[*2]は、一会社員となり上司から与えられるお仕事を機械的にこなすためのルーチンを組むゲームです。少々前ですが AtCoder の Chokudai 氏が言及していたこともありご存知の方もいらっしゃるでしょうか。ゲーム内容は端的に言えば機械語でのプログラミングで、add, neg, succ 程度の簡素な数値演算に比較とジャンプ命令を組み合わせて、高級言語ならさらりと書けるであろうコードをチマチマと組み立てることになります。使用できる命令の幅は次第に増え、後半にはメモリアクセス的処理も登場します。全体的な難易度はプログラミングに慣れているなら簡単かなといったところですが、高難易度問題や各問題の追加目標となるステップ数最適化と行数最適化（コードゴルフ）も用意されており、そこまで手を付ければそれなりに大変です。 世界観は （企業的な意味でも） 少々ブラックな雰囲気になってはいますが主張は強くなく、各面の出題役である上司キャラとの会話と数ステージごとに挿入されるムービー程度です。開発元に特徴的なギョロ目二頭身キャラたちは（慣れれば）可愛らしくもありますが、ストーリーの方はなかなか難解なのでオマケ程度に考えておくのが良いかも。

このゲームのようにアセンブリを書かせるシステムは比較的多く、 この後でもいくつも紹介することになります。 普段と命令セットが変わることによるゲーム感の向上 （もしくは仕事感の減少） や低レイヤゆえの実装の簡便さが採用の理由でしょうか。 とは言えそこに少しのアレンジが加わることで大きくゲーム体験が変わってしまうのが醍醐味でしょう。Human Resource Machine はそういった意味ではかなり味付けが少ない部類なので、この類のゲームの入り口としてもお勧めです。Steam 以外にも Nintendo Switch からも配信されているので、手も出しやすいと思います。

このゲーム実はすでに続編である 7 Billion Humans [*3]も公開されています。こちらでは前作で一会社員だったところから少々階級が上がり、中間管理職としてチーム運営、i.e. 並列プログラミング、を行うことになります。基本的なインタフェースは大きく変化しないものの、プログラムの内容がマルチスレッド想定となることで難易度は体感かなり上昇しています。複数の部下たちが全員同じ仕事（コード）を実行していくのですが、必ずしも同期が取れないうえ社員間でのデータ受け渡しも頻発します。逐次実行に慣れた頭ではなかなかに歯応えがあり、前作はサクサク解けた人でも挑戦しがいがあると思います（筆者はまだ途上で苦しんでいます）。こち

2 https://tomorrowcorporation.com/humanresourcemachine

3 https://tomorrowcorporation.com/7billionhumans

らも前作同様各面に実行時間最適化・コードゴルフ目標が設定されているのでやり込みも可能です。存分に部下たちを働かせましょう。

TIS 100

TIS 100[*4]はタイトルに冠された並列計算機、TIS100 のプログラミングを行うゲームです。7 Billion Humans と同じく並列プログラミングを行うことになるものの、システムは大きく異なります。TIS100 のハードウェアは小さな計算ノードがグリッド状に接続されて構成されており、各ノードに個別にコードを記述できるようになっています。7 Billion Humans では全ノード（社員）に同じコードを与えていたところが別々になった分全体の挙動を把握するのは楽になりますが、一つのノードはレジスタを二つしか持たずまた記述できるコード量も小さく制限されているため、処理を小さく分割して各ノードに分散させていく必要があり、そこがゲームの肝となっています。本作にはチュートリアルが用意されておらず知人から譲り受けた仕様書という体の pdf（実際に印刷することが推奨されています）から命令セットやハードウェアを読み解きながら操作方法を学んでいく形をとっており、解像度の低いディスプレイを再現したゲーム画面もあり雰囲気も楽しめると思います。Human Resource Machine には最適化目標ありましたが、こちらでは世界中のプレイヤー達がどれだけのコード量・実行時間で各面をクリアしたかの情報が収集されており、自分の位置と共にヒストグラムで確認することができます。各面の回答を複数保存できるなど記録追求もやりやすい仕様になっているのでランキング上位を目指して最適化に励むのもいいですし、TIS100 組み込みの描画機能でお絵かきに挑戦するのもいいかもしれません。

TIS100 の開発元である ZACHTRONICS[*5]はこの種のゲームをメインに制作を行なっているチームとなっています。実のところこの記事用にリストアップしたゲームの半分ほどが同団体の制作でありどこまで紹介すべきか少々悩む部分もあるのですが、プレイした分については並べておこうと思います。どれも質が高い作品となっているので、気になるものがあればぜひプレイしてみてください。

4 http://www.zachtronics.com/tis-100

5 http://www.zachtronics.com

SHENZHEN I/O

SHENZHEN I/O[*6]はハードウェア込みでの製品開発をテーマにしています。TIS100 同様にゲーム付属のハードウェア仕様書 pdf を参照しつつ、指定の基板状にマイコンチップを配置・配線・コーディングして製品を完成させます。各チップに独自のアセンブリ言語でコードを書き込むことになるのですが、TIS100 同様書き込める行数には制限があるので、どう役割を分割するか考えることになる他、数値に加えてアナログ値（ゲームの都合上実際にはデジタルですが）の二種類の入出力配線を取り扱うことになります。TIS100 と比べるとノードの種類や情報のやりとりなどで自由度が上がり、その分要求仕様も増え考えることも増えたという印象です。順当な発展版と見てもいいのかもしれません。

開発元は違いますが Silicon Zeros[*7]も回路を接続するという意味で似たシステムになっています。こちらでは扱う部品がかなり低レイヤ寄りで、加算器やアダー、ラッチにアドレス指定で値を読み書きするメモリといった電子回路部品を組み合わせていくことになります。部品の配置と回路の配線だけでコードの記述はないためマウスのみでプレイでき、他に比べかなり軽いゲーム感でサクサクと進められるので、電子回路を学び始めた人にお勧めです（？）。

Space Chem

さてここからは一般的なプログラミングからは離れる方向に進んでいきましょう。
Space Chem[*8]は打って変わってコンピュータから離れ、化学をテーマにしています。とはいっても授業で学んだような化学合成が出てくるわけではありません。謎技術によって分子・原子を単体分離して扱える上結合数の増減も物理法則を無視して自由自在に行えるのであくまでフレーバーになります。ゲーム的には合成装置の盤面に原子を持ち運べるアームとそれが移動する軌道を設定し、軌道上の各ポイントにアームの掴み・離しや入力化合物の投入指示などを配置することで全体の動作を調整していくことになります。クレーンゲームのような機械をプログラムするイメージが近いでしょうか。配置の自由度は比較的高く、面によってはその自由度を生かして生成速度を調整し、複数の装置同士が協調してバランスよく動作するようにする必要も出てきます。なお、筆者の環境では OS のアップデートに伴って起動できなくなったため積みゲーと化しました。Mac をメインにお使いの方は注意してください。

---

6 http://www.zachtronics.com/shenzhen-io

7 http://pleasingfungus.com/Silicon%20Zeroes

8 http://www.zachtronics.com/spacechem

Opus Mugnam

Opus Mugnam[*9]も近いシステムを持ったゲームです。こちらは錬金術がテーマになっており、原子の代わりに地水火風の四元素のエレメントに加え、三原質（硫黄、水銀、塩）や金属元素を結合・分離・変換させて目的の物質を構成していくことになります。ヘクスの盤面にアーム、物質の結合・分離や変換を行う装置を配置し、掴み離しや回転・伸縮といった一連の動作を各アームにプログラムして、生成物を移動させながら合成を進める形になります。Space Chem がクレーンゲームのようなイメージだったのに対し、こちらは工場の製造ライン（ラインではなく二次元になりますが）でベルトコンベアの代わりにロボットアームを使うようなイメージです。生成物だけでなくアームも盤面状に実体を持つため、やりとりの間に衝突が起きないような配置と動作定義が必要になります。ラインを構成した後、盤面上でアームが回転し生成物が整然と流れ組み立てられていく様は、グラフィックの質の高さもありいつまでも眺められます。見た目にも大変美しいゲームなので、ぜひ公式ページから PV を見てみてください。

実はこのゲームには前身となる作品も存在しています。the codex of alchemists [*10]がそれで、盤面がヘクスではなくグリッドであることを除けばゲームシステムはほぼサブセット的内容になっています。こちらは無料でのプレイが可能なので試しに遊んでみてはいかがでしょうか。

KOHCTPYKTOP

KOHCTPYKTOP [*11]も the codex of alchemists 同様、Zachtronics 最初期の作品の一つです。冷戦時代の共産主義をフレーバーにしつつ、基板へ特殊な導体での配線を行うパズルになっています。二種の導体で実現される論理ゲートを組み合わせて回路を構成するのですが、配線上を流れる電気信号の物理的な遅延を考慮しうまく利用する必要があります。問題数は多くないものの難易度はかなりハードです。ブラウザゲームとして公開されている他、これらを含む初期作品や制作ノートをまとめた ZACH-LIKE [*12]が Steam で配布されています。

9 http://www.zachtronics.com/opus-magnum/

10 https://www.kongregate.com/games/krispykrem/the-codex-of-alchemical-engineering

11 http://www.kongregate.com/games/krispykrem/kohctpyktop-engineer-of-the-people?referrer=Jayisgames

12 http://www.zachtronics.com/zach-like/

Manufactoria

そろそろ Zachtronics から離れましょう。Manufactoria[*13]は工場の生産物の品質チェック装置を作るゲームで、具体的には製品に組み込まれた記号列が条件をみたすかをチェックすることになります。製品（記号列）をベルトコンベアで流しながら、先頭一文字による分岐や文字の追加削除を利用して指定の条件を満たすものを選別していきます。各操作で先頭の一文字を処理していくという意味ではオートマトンを平面上に構成していくような趣があります。記号列と言いましたが入力で使われる記号は赤丸青丸の二種類なので実質バイナリで、実際入力を数値とみなした問題も出てきます。中盤からはプレイヤーのみ利用できる色がさらに二色追加され、可能な操作が増える分さらに複雑な処理を求められるようになります。最適化目標等は設定されていませんが、コミュニティを探せば部品の最小配置数を競っていたりもするので気になったら覗いてみても良いかもしれません。[*14]

jahooma's logic box

jahooma's logic box[*15]も同様に文字列処理を行うブラウザゲームです。入力として与えられる文字列を処理部品を用いて盤面上を流しつつ変換していく点では同じなのですが、こちらでは入力文字種が 01 から英アルファベット + 数値にまで増えています。タイトル通り処理部品であるボックスを盤面に配置していくことになりますが、各面自体がまたボックスになっており、以降の面でも一つの部品として利用できるようになるという独特の構成になっています（とはいえ実際に再利用できるのは一部のみですが）。途中からは自分自身を部品として配置できる再帰的な構成も現れます。また値の受け渡し方法もベルトコンベアをいちいち引いていた Manufactoria からは少々変わり、各ボックスから一方向に一直線に投げ飛ばすようにして受け渡しを行います。各基本部品の処理能力も多少高く・幅広くなっているので、メインのプレイ感も部品の配置というよりは処理をどう行うかの方に変わっているように思います。

なお上記の引用先は改訂版で、前作となる作品も存在します[*16]。どちらもブラウザゲームですが、旧版は Flash、新版は HTML による実装になっているようです。正直なところ旧版の方がグラフィックや UI は上なのですが、旧版をプレイして気に入ったなら問題数がより充

13 http://pleasingfungus.com/Manufactoria/

14 たとえばこちら (http://blog.livedoor.jp/lkrejg/archives/65437773.html)

15 https://logicbox.jahooma.com

16 http://www.kongregate.com/games/jahooma/jahoomas-logicbox

実している新版にも手を出して（そして作者へお布施をして）みてください。

Tile factory

Tile factory[*17]もまた製品を盤面上で流しながら処理を加えていくゲームです。こちらでは処理対象がタイトル通りタイルで、これに指定の模様を印刷することが目的になります。ベルトコンベアでの移動、スプレーでの色の吹き付け、上書きを防ぐためのマスク貼り付けや塗料の混合など必要操作は多岐にわたります。装置の起動方法も独特で、タイルが上を通ると反応する圧力センサでの起動をメインに、シグナルの論理合成や遅延部品まで使って配線を行うことができます。扱う対象が文字列ではなくカラフルなタイルであることや各装置のポップな動きなど他より（視覚的な）親しみやすさは高めですが、システムの複雑さは引けを取らずボリュームもなかなかの仕上がりになっています。

herbert online judge

忘れていましたがプログラミングゲームといえば herbert online judge[*18]にも言及しておくべきでしょう。盤面上を動き回るロボットの挙動を操り、指定のマスを正しく踏むように独自の言語でプログラムするパズルゲームです。かなり厳しく設定された文字数制限を潜り抜けるためのコードゴルフ要素がメインになっています。サイトの公開自体は十年近く前になりますがまだプレイしている方はおり、有志による Wiki も更新されているようです。問題作成やリーダーボードも用意され、問題数も 2000 を超えているのでやりがいは十分でしょう。

Screeps

最後に少々毛色が違うものを二つ紹介して終わりにしましょう。ここまではテーマやシステムがプログラミングであるゲームでしたが、Screeps[*19]は操作がプログラミングになっている MMO ゲームです。ジャンル的にはシミュレーションで、各プレイヤーは自拠点から自律動作するユニットを生産し、広大なマップの上でリソースを収集し戦闘や建築を行いながら領土を広げていくことになります。特徴的なのはユニットの生産戦略から各ユニットの挙動までの全てがプレイヤーのプログラムにかかっているというところです。すなわち戦闘・生産・機動力

17 http://www.kongregate.com/games/duerig/tile-factory?acomplete=tile+fa

18 http://herbert.tealang.info

19 https://screeps.com

といったユニットごとの性能はもちろん、何を目的として、どこを目指して動くのか、敵やリソースを見つけたらどうするのかといった行動パターンまで全てをプログラムする必要があるのです。プログラミングのための言語は JavaScript で、API ドキュメントが公式ページに完備されています。MMO ですからもちろん対戦相手は他の人間（によってプログラムされたユニット）です。あなたもゲーム AI プログラミングにチャレンジしてみませんか。

DUSKERS

最後に DUSKERS[*20]を紹介します。これをプログラミングがテーマのゲームというのは少々違うかもしれませんが、見慣れた CUI でのコマンド入力が基本操作になるので親近感はあると思います。このゲームではプレイヤーは宇宙船のパイロットで、他の生存者を探して宇宙をさまようことになります。各所にある廃棄された宇宙船から資源を収集することでより長く探索を続けていくことになるのですが、メインとなる作業は全てドローンが担い、プレイヤーは本船からそれらドローンの遠隔操作を行うことになります。CUI からコマンドを叩いてドローンを操作し、対象宇宙船のシステムに侵入して情報を得、ドローンのセンサ越しに状況を見ながらの探索は、本当に宇宙空間に一人でいるような没入感を体験させてくれます。独特のプレイ感をぜひ体験してみてください。

4. おわりに

以上、つらつらと並び立てましたが、気になるものはあったでしょうか。世界にゲーム含め娯楽は山ほどあり、何に有限の人生を使うかは難しい選択ではありますがその選択肢の一つにこれらのゲームを加えられることができていれば幸いです。

20 http://duskers.misfits-attic.com

複数マシンでの実験を Ansible とかでシュっとした話

— 実験編 —

MasWag

1. はじめに

皆さん実験ぶん回してますか？私は CS の博士課程学生で、物凄くざっくり言うとシステムのバグを探す様な研究をしています。研究柄、自分の設計した手法が「従来手法と比べて高速で動作するかどうか」であったり「より多くのバグを探すことができるか」といったことを実験的に調べることがよくあります。私の様にアルゴリズムや何らかの手法を設計する人は勿論ですが、そうでない人も実験的にパフォーマンスを調べることは多かれ少なかれあるのではないでしょうか。例えばウェブサーバを Apache (https://httpd.apache.org/) と NGINX (https://www.nginx.com/) のうちどちらにするとより効率が良いかであったり、キャッシュの設定等の諸々のチューニングをして高速化をするときは実験的にパフォーマンスを比較する必要があると思います。実験すると言うのは簡単ですが、色々な組み合わせで試そうとするとかなり長い時間がかかることもあるので、あまり気軽に実験を回せない人も少なくないと思います。

しかし今はクラウドコンピューティングの時代です！各マシンで完全に独立な実験を回す場合、N 台のマシンで実験を回せば 基本的には 1/N の時間で実験が終わることになります。特に Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP) 等 クラウドの場合、多くの場合料金計算が 時間 × 台数なので、台数を増やしてもその分時間が減れば、なんと料金的には変わらないです。

とは言ったものの複数台のマシンで大量に実験を回すのは一苦労です。実験するマシンの調

達、実験するマシンの環境構築、実験タスクの各マシンへの分配、実験結果の回収方法、等々考える必要のある作業は多数あります。この辺りの作業で手間取ると無駄にクラウドの料金が増えてしまうので悲しいです。一方で、よくあるクラウドでの複数台管理についての情報は、「同じ働きをする」マシンを複数台立ててロードバランサ等でタスクを分配する、Web アプリケーション向けのものが殆どなので、そのまま並列実験に適用できないことが多いです。

この文章では、複数マシンで実験を回す方法についての説明をします。内容としてはクラウドを意識はしていますが、大半の内容はオンプレミス環境などでも動作するはずです。また、今回の実験は AWS で行ないましたが、AWS に特化した話題は極力避けています。例えば AWS ではオンデマンドインスタンスで所謂「定価」で計算資源を買うよりも、スポットインスタンスでオークション形式で買った方が安いですが、そういった話はしません。また、この文章で説明している手法は単なる一例にすぎません。「僕の考えた最強の手法」ですらなく、今回の実験をそれなりの労力で「シュっとさせた」というだけなので改善の余地は 大いにあります。一方で他の人がこういう実験をどういう風に回しているのかを知りたい、というニーズに答えられていれば幸いです。

1.1. おことわり

今回の文章中に挙げているスクリプトですが、比較的やっつけで書いたものも少なくないのでポータビリティについてはあんまり自信がないです。特に Arch Linux や macOS + Homebrew などの、比較的最新版に近いコマンドラインツールがインストールされる環境で確動作確認を行なっているので、ややバージョンの古めの Ubuntu などでは上手く動作しないかもしれません。

関連した本やブログ記事などがあれば良いのですが、筆者の知る限りでは似たことをやっている文書は見つかりませんでした。御存じの方が居たら教えていただけると参考になります。また技術選定については完全に筆者の好みです。他の例えば全体的に python に統一する、など別の方法も可能ですし、そういった方法と比べて今回の方法が優れているかどうかは一概には言えないでしょう。

2. 基本方針

今回の実験ではおおよそ以下の様なことを基本方針としています。

- できるだけ自動化させるが、諸々のコストが大きい場合は手動作業も厭わない (KISS の原則)
- できるだけ実験を行なった環境・コマンドの情報を残しておく (再現性の担保)

2.1. できるだけ自動化させるが、諸々のコストが大きい場合は手動作業も厭わない

人類皆自動化が大好きですし、勿論僕も好きです。特に多数のマシンの管理を行なう際に手動の作業が多いと手間ですし、手動で色々やって時間がかかるとクラウドの従量課金額が増えてしまうのでシンプルに勿体ないです。この記事のタイトルにある Ansible はマシンの構成を自動で管理するツールですし、自動化は行ないます。

しかし、だからと言って全ての作業の自動化はこの記事でのゴールではありません。行き過ぎた自動化によって構築するシステムが複雑すぎるものになってしまうのは考えものです。構築するシステムが複雑になればなるほどデバッグは大変になりますし、本当にやりたい実験に到達するまでの時間が長くなってしまいます。今回の最終目的は自分のやりたい実験を簡単に回すことなのでその上で大きな障壁とならない部分はできるだけシンプルに、場合によっては手動作業で行ないます。

2.2. できるだけ実験を行なった環境・コマンドの情報を残しておく

比較的大規模な実験をしていると、ちゃんと正しい手順で実験を行えているかが気がかりになることがよくあります。複数の設定で実験をやっていると `git clone` 等でバージョンを変更するのを忘れる、コンパイルを忘れるなどのオペレーションミスが起きる (または起きていないか段々自信がなくなってくる) ことがよくあると思います。

もう一つの懸念事項は実験環境を忘れてしまうということです。一つ一つの実験について、例えば gcc のコンパイルオプションが `-O3` だったか `-O2` だったかを人間が覚えていくのはなかなか難しいですし、自分が書いたソフトウェアの細かいバージョンと実験結果の対応関係は、ちゃんと記録を残しておかないとまずわからなくなってしまうでしょう。

これらの理由で過去に行なった実験自体の正しさに確信が持てない場合、つまり「そういえばこの実験ってどういう環境でやったんだっけ ...?」という状況になってしまった場合、最終

的には再実験を行なって「正しい」実験を再度行なうことになると思います。大規模な実験となると時間もかかりますし、クラウドで行なうとなるとお金もかかってしまいます。逆に言うと、時間を置いて新しい実験設定と比較実験を行ないたい場合でも、過去の実験環境の詳細がわかっていれば以前試した部分については実験結果を使い回せるという利点があります。

上記に様に無駄な不安を減らしたり実験の回数を減らすためにも、できるだけ実験を行なった環境・コマンドの情報を残しておく、つまり同じ実験を再現させられるようにしておく、ということを今回の実験では重視します。

3. イカれたメンバーを紹介するぜ！！

- **マシンの作成**手動！！
- **マシンの起動** vmctl ！！
- **構成管理** Ansible ！！
- **実験スクリプト、実験結果の管理** Git ！！
- **対話的な諸々の自動化** expect ！！
- **実験開始・終了のお知らせ** slack ！！

以上！！

ここから各登場ツールについて説明をしていきます。なお、最終的なワークフローは図 1 の様になります。

3.1. vmctl

vmctl (https://github.com/MasWag/vmctl) は様々な仮想マシンに対して同様のインターフェースで起動、停止などの基本的な操作を行なえる様にした、手製の shell script です。「様々な仮想マシン」と言いつつ現状 Amazon AWS の EC2 と VirtualBox にしか対応していないですが、原理的にはコマンドラインインターフェースが用意されている仮想マシンであればそれなりの工数で追加できることになっています。

今回は特に複数種類の仮想マシンを扱うこともないと思いますが、マシンの id を毎回打ちたくないという理由で使いました。例えば ec2 だと各マシンに対して `i-1234567890abcdef0` の

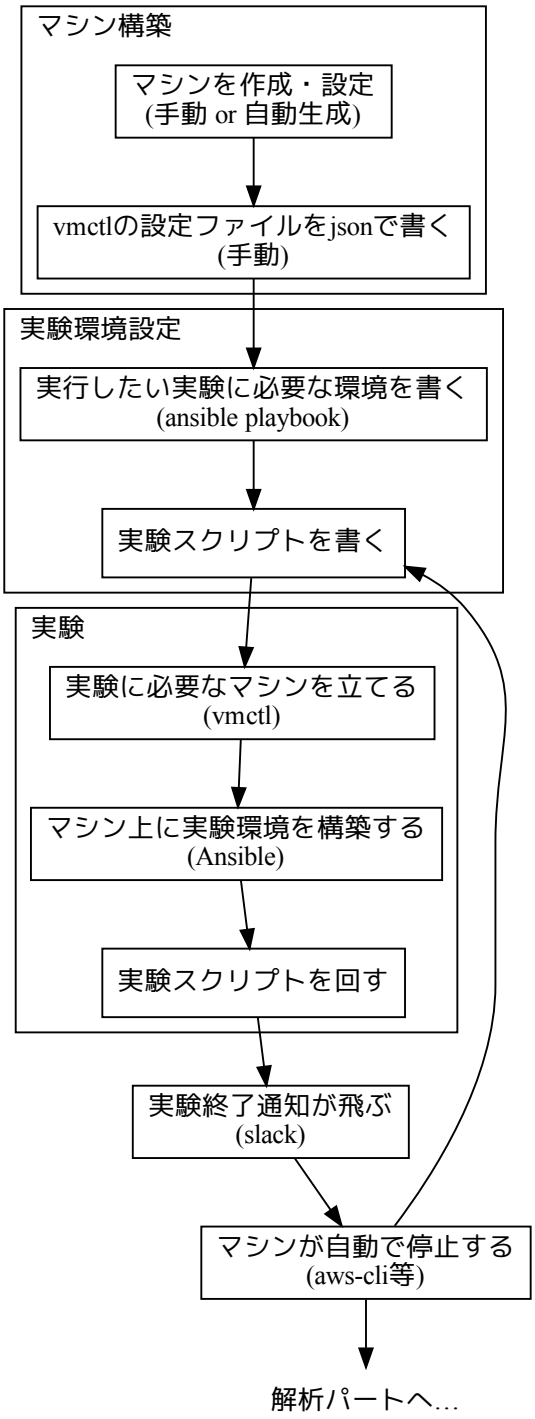

図 1　実験全体のワークフロー

様な id が割り振られます。 率直に言うとこれは人間が覚えて直接扱うべきものではありま

せん。 できるものなら各マシンに役割の分かり易い名前を付けたり、 例えば `my_instance1` `my_instance2` `my_instance3` ... の様に連番のマシン名を付けたりしたいです。今回の vmctl の用途はまさにこの名前付けです。例えばマシン名が `my_great_instance` のマシンを起動するのであれば、

```
$ vmctl start my_great_instance
```

で 起動できます。特にマシン名が連番である場合は bash の連番展開を使って、

```
$ vmctl stop my_instance{1..10}
```

の様に使うこともできます。

代替品 – Terraform

クラウドのマシンを一括で起動・停止するという意味では Terraform (https://www.terraform.io/) を使うこともできます。Terraform は 必要なインスタンス等のインフラストラクチャをテキストで定義して、自動で作成 (apply) ・破棄 (destroy) するツールです。Terraform を使う場合は予め準備しておいたインスタンスを起動・停止するのではなく、毎回新しいインスタンスを作成・破棄することになるので、より Immutable Infrastructure 的なワークフローに向いていると思います。なお、筆者が Terraform を使うときは Packer と Ansible で大元のイメージを作成しています。

3.2. Ansible

Ansible (https://www.ansible.com/) は言わずと知れた超有名構成管理ツールで、YAML でマシンの設定を記述することで、自動で環境構築やデプロイを行えるツールです。今回は必要なソフトウェアのインストールの他に、実験スクリプトや実験用のコードの入っている git レポジトリの clone/pull 等に使っています。実験対象のプログラムのコンパイル等の定型処理を忘れない、というのも一つの利点ですが、Ansible は複数マシンに対して実行することができるので、特に多くのマシンを扱いたい場合には便利になります。

代替品 – 構成管理ツール

Ansible 以外にも例えば以下に挙げた様に、構成管理ツールはかなり多数あるので、好きなも

のを使うと良いと思います。個人的にも Ansible を使っている深い理由はないですし、色々な比較をしている web ページもかなりあるので、ここでは名前を挙げるのみとします。

- itamae (https://itamae.kitchen/)
- Chef (https://www.chef.io/)
- Puppet (https://puppet.com/)

3.3. Git

Git (https://git-scm.com/) は有名、というより 2020 年現在ほぼデファクトスタンダードとなっている分散バージョン管理システムです。今回は Git を実験内容を記述したスクリプトの管理だけではなく、実験結果のテキストファイルの管理にも使っています。実験スクリプトはソフトウェアなので当然 Git での版管理は便利です。複数マシンで実験を行なう場合、同じ様な実験結果のファイルが生成されると思いますが、場合によっては同名の実験結果ファイルが生成されてしまうと思います。実験結果も Git で管理することにより、この様に実験結果のファイルが衝突したときにちゃんとマージできるからです。この、「複数マシンで同時に実験した結果を上手い具合にマージできる」という点が並列実験システムにおいてかなり重要であると考えています。

代替品 – バージョン管理システム

Git 以外にも Mercurial や Bazaar、darcs など様々な分散バージョン管理システムがあるので、これも好きなものを選ぶと良いと思います。一方バージョン管理システムと言っても、RCS の様にローカルで完結しているものや、Subversion の様に集中型で気軽にマージができないものは向かないと思います。

複数マシン間でデータを同期させたいだけであれば、例えば rsync でも良い様に思えますが、過去の変更の情報をちゃんと追えなかったり merge がちゃんとできないので、しっかりとワークフローを練らないと代替は難しいと思います。

速度面での課題

Git はかなり大きなレポジトリを扱えますが、実験ログが膨大になったり、特に巨大なバイナリファイルを多数扱う必要が出てくると Git が思う様に動かなくなっていきます。具体的には例えば `git merge` などにかなりの時間がかかる様になります。これについては、(筆者は試した

ことがないですが) 例えば Git-LFS を使うことで全体のワークフローとの噛み合わせを保ちつつ巨大なファイルを扱える様になると思われます。

3.4. expect

expect (https://core.tcl-lang.org/expect/index) は Tcl 製の対話的な CUI プログラムの自動化ツールです。普通の CUI の処理の自動化なら例えばシェルスクリプトで行なえますが、ssh や ftp 等で必要となる対話的な操作の自動化をシェルスクリプトで行なうのは至難の技です[*21]。こういった場面で expect を使うことで比較的容易に自動化することができます。 今回は実験スクリプトをリモートで実行する際に expect を使いました。Ansible でも似たことができるかもしれないですが、知らないうちにオーバーヘッドが載って実験結果に影響があると良くないので、できるだけ簡素な方法ということで expect を使いました。

代替品

今回はオリジナルの Tcl 版の expect を使いましたが、python や ruby などでも expect の移植版や類似のものが出ているので、そちらを使っても良いでしょう。

3.5. slack

slack (https://slack.com) は有名チャットツールで、http 経由で簡単に外部から通知を送ることができます。今回は実験終了の通知を送るために使いました。実験の進め方とは関係ないですが、実験が終わるのかを逐一見に行くのは精神衛生上よろしくないですし、今回のワークフローの中では重要だと考えます。

代替品

今回は slack を単に通知を送るためだけに使っているので、 特に slack である必要もないですし、 普段 slack を使っていない人がわざわざ使う必要はないでしょう。他のチャットツールを使っても良いですし、それこそメールで通知を送っても大丈夫です。

21 シェルスクリプトでも例えば bash 4.0 で追加されたコプロセスを使えば実現できそうな気がしますが、それはまた別の話ということで

4. マシン構築パート

ここから、図 1 のマシン構築パートの流れについて説明していきます。ここではそれぞれの実験に共通した実験用マシンの設定を行ないます。

4.1. マシンの作成・初期設定 (手動)

まず始めに実験に使うマシンを作成して、ユーザ設定や最低限必要な設定などを行ないます。「マシンの初期設定」というとかなり広い範囲のことが含まれそうですが、環境構築スクリプトをもう一度回せば同じ環境を構築できる様にしたいので、できる限り手動操作を減らすと良いでしょう。例えば今後必要となるユーザや SSH 鍵の配置といったアクセス周りの設定や、今後の実験で必要なソフトウェアのうち、インストールに手動の作業が必要なものだけをインストールすると良いでしょう。

アレンジ例

今回はメインの実験で MATLAB が必要で、ライセンス管理の自動化が厄介なので、MATLAB のインストールまでを手動でやりました。必要なソフトウェアにライセンス等の問題が全くないのであればこの工程は自動化しても良いと思います。この場合マシンイメージの作成過程をスクリプトとして残すことができるので、最低限必要な環境設定に加えて、インストールに時間がかかるソフトウェアのインストールもこの段階で済ませてしまうと良いと思います。例えば、Packer (https://packer.io/) で予め必要なマシンイメージを自動で作っておいて、必要な時に必要なだけ Terraform とかでマシンを作って、不要になったらすぐ壊す、ということが可能です。

また、初期設定に手動部分がどうしても必要な場合でも、自動設定の部分と手動設定の部分を分離することで、Packer などを部分的に用いるワークフローにすることも良いと思います。

4.2. vmctl の設定ファイルを書く

次に vmctl を設定します。vmctl の設定は json で書かれた設定ファイル (`~/.vmctl.json`) で行ないます。基本的には 1) インスタンス名、2) インスタンスの種類 (ec2 等)、3) インスタンス ID、が書かれた json ファイルで、例えば以下の様になります。

```
[
    {
        "name": "marisa",
        "type": "ec2",
        "instance_id": "i-0fsdfd13c7bf3d6b6",
        "profile": "sample"
    },
    {
        "name": "reimu",
        "type": "virtual_box",
        "instance_id": "95a2dsfdb-0dfbf-40bb-bf15-92df8d07c7dc"
    }
]
```

インスタンス数が少ない場合は手書きしても大丈夫ですが、多数のインスタンスを扱う場合は設定ファイルを自動で生成したくなると思います。Amazon EC2 については 例えば次のコマンドで生成することができます。

```
$ aws ec2 describe-instances --query 'Reservations[*].Instances[*]
.[InstanceId,Tags[?Key==`Name`].Value|[0]]|[]' |
  jq --arg profile "$PROFILE" 'map({"type": "ec2", "instance_id":.
[0], "name": .[1]})' > ~/.vmctl.json
```

5. 実験環境設定パート

次は図 1 の実験環境設定パートについて説明していきます。ここではそれぞれの実験に特有の実験内容についての設定をしていきます。

5.1. 実験環境構築用の Ansible playbook を書く

まず始めに実験環境構築用の Ansible playbook を書きます。ここでは、データセットの準備や実験対象のプログラムのコンパイル、実験用 Git レポジトリの準備等に加えて、後で必要となる aws-cli や slack に通知を送るためのスクリプトの設定等も行ないます。 細かい内容につい

てはさておき、例えば以下の様な YAML ファイル で設定を行なうことができます。

```
- hosts: aws
  user: ubuntu
  tasks:
    - name: Install required packages
      apt:
        pkg:
          - awscli
          - unzip
          - ...
      become: yes

    - name: configure aws-cli
      file:
        dest: ~/.aws/
        state: directory

    - name: configure aws-cli
      copy:
        src: ~/.aws/config
        dest: ~/.aws/config

    - name: configure aws-cli
      copy:
        src: ~/.aws/credentials
        dest: ~/.aws/credentials

    - name: Download and extract the dataset
      unarchive:
        dest: /tmp
        src: http://example.com/dataset.zip
```

```
        remote_src: yes

    - name: clone bar-experiments
      git:
        repo: "git@example.com:foo/bar-experiments.git"
        dest: /home/ubuntu/bar-experiments

    - name: setup notif_my_slack
      file:
        dest: ~/bin/
        state: directory

    - name: setup notif_my_slack
      shell: m4 -DHOST=$(/usr/bin/aws ec2 describe-instances --ins
tance-ids "$(cat /var/lib/cloud/data/instance-id)" --query 'Reserv
ations[*].Instances[*].Tags[?Key==`Name`].Value' | tr -d [] | xarg
s) /home/ubuntu/bar-experiments/utils/notif_my_slack.m4 > ~/bin/no
tif_my_slack

    - name: setup notif_my_slack
      file:
        path: ~/bin/notif_my_slack
        mode: '0755'
```

notif_my_slack.m4

環境設定用の Ansible notebook の"setup notf_my_slack"の部分で m4 及び notif_my_slack-.m4 が出てきたので説明をします。notif_my_slack.m4 は m4 のコードです。今回 m4 は notif_my_slack というシェルスクリプトを生成するためのプリプロセッサとして使っています。m4 自体もチューリング完全なプログラミング言語ですが、今回は単に文字列 HOST を EC2 でのインスタンス名に置換するためだけに使っています。 なお最後の https://hooks.slack-.com/services/<The ID> は slack で外部から通知を飛ばす用の URL (Incoming Webhooks の

エンドポイント) です。取得法などについては slack, Incoming Webhooks, 等で検索すると詳しい説明が出てくるので省略します。

```
#!/bin/sh

if [ $# -gt 0 ]; then
  curl -X POST -H 'Content-type: application/json' --data '{"text"
:"'"$*"' from HOST"}' https://hooks.slack.com/services/<The ID>
else
  curl -X POST -H 'Content-type: application/json'  --data  `"{\"t
ext\":\"###From' HOST###\n$(cat)\"}" https://hooks.slack.com/servi
ces/<The ID>
fi
```

5.2. 実験スクリプトを書く

「実験スクリプトを書く」というとただ一言で終わってしまうので、個人的に採用している実験用 Git レポジトリの構成についても説明します。実験用に Git レポジトリを作っている理由は前述の様に複数マシンで同時に実験した結果を上手い具合にマージできるからですが、できるだけマージ時に衝突しないように図の様なディレクトリ構成を採用しています。ざっくり言うと気をつけている点は以下の点です。

- 各実験に個別のスクリプト等は各実験用のディレクトリに入れる。逆に共通のスクリプト等は `/utils` 以下に格納する
- 各実験に timestamp 付きの ID を割り振って衝突しない様にする
- 実験用スクリプト名は常に `run.sh`
- 各ディレクトリには実験の説明を書いた `README.md` を書く

```
  <20200102-1234-experiment1>
      README.md
      run.sh
      launch.sh
```

```
 ...
<20200203-2345-experiment2>
    README.md
    run.sh
    launch.sh
 ...
utils
    setup.sh
    teardown.sh
    notif_my_slack.m4
 ...
```

それではそれぞれのファイルについて説明していきます。なお、notif_my_slack.m4 の説明は 5.1 節を参照してください。

run.sh

run.sh は実験用のメインになるスクリプトです。ファイル名を run.sh に固定するのは自動化を容易にするためです。複数のスクリプトが欲しい場合はディレクトリを切る運用にしました。run.sh での処理はざっくり書くと以下の様になります。

- 準備用のスクリプトである setup.sh (後述) を呼ぶ
- 実験本体の処理
- 終了処理用のスクリプトである teardown.sh (後述) を呼ぶ

setup.sh

さて、setup.sh は準備用のスクリプトです。とはいえやっている内容は以下の二つだけになります。

- 実験対象のプログラム等、外部で使っている git レポジトリののバージョンをファイルに保存
- slack に実験開始のお知らせをする

```
#!/bin/sh

git --git-dir ~/<Program>/.git rev-parse HEAD > git-hash

readonly experiment=$(pwd | sed 's:.*/::')

notif_my_slack <<EOF
experiment ${experiment} started.
The arguments: $@
EOF
mkdir -p results
```

teardown.sh

次に実験終了時のスクリプト、teardown.sh です。ざっくり言うとやっている内容は以下の三点になります。

- 実験内容を git commit する
- slack に実験終了のお知らせをする
- 実験に使ったインスタンスを停止する

```
#!/bin/sh

readonly experiment=$(pwd | sed 's:.*/::')

# Synchronize the filesystem and wait for 60 sec.
sync
sleep 60
git add .
git commit -m "experiment ${experiment} $* finished"
notif_my_slack "experiment_${experiment}_finished"
/usr/bin/aws ec2 stop-instances --instance-ids "$(cat /var/lib/clo
ud/data/instance-id)"
```

6. 実験パート

次は図 1 の実験パートについて説明していきます。ここが実験のメインパートですが、これまでにしっかりと準備しているので、やっていることは極々シンプルです。

6.1. 実験に必要なマシンを立てる

まず始めに実験に必要なマシンを立てます。今回既に実験に必要なマシンは構築されているので、vmctl を使ってマシンを立ち上げるだけです。例えば以下の様なコマンドでマシンを立ち上げられます。なお、マシン名が連番だったり共通部分がある場合はシェルのブレース展開 (Brace Expansion) を使うと便利です。

```
$ vmctl start マシン 1 マシン 2 ...
```

6.2. マシン上に実験環境を構築する

次にマシン上に実験環境を構築します。具体的には実験用レポジトリやデータセットの準備などを行ないます。とはいえ既にこれらの準備をする Ansible playbook を準備していると思うので、単に Ansible playbook を実行するだけになります。

Ansible を使っていく上で問題になるのがインベントリの管理です。というのも少なくとも Amazon EC2 のインスタンスは起動する度にグローバル IP アドレスが変わるのでインベントリファイルを事前に作っておく訳には行きません。こんなときのために (?) Dynamic Inventory という仕組みが Ansible にはありますが、今回は難しいことは考えずに静的なインベントリファイルの IP アドレスの部分を実行前に生成することにしてみます。例えば次のシェルスクリプトと m4 のコードを元にするとインベントリを作れます。やっていることは単に下の m4 ファイルの"IP"と書かれた部分を vmctl で得られた IP アドレスに置き換えているだけです。

```
m4 -DIP="$(vmctl ip "$@")" aws_host.m4 > aws_host
```

```
[aws]
IP
```

```
[aws:vars]
ansible_ssh_user=ubuntu
ansible_ssh_private_key_file=~/.ssh/id_ecdsa

[all:vars]
ansible_ssh_port=22
```

インベントリが生成されたら Ansible playbook を実行しましょう。

```
$ ansible-playbook -i aws_host setup.yaml
```

6.3. 実験スクリプトを回す

次に実験スクリプトを回します。前述の様に今回は expect を使って、SSH 越しにリモートインスタンスで実験スクリプトを動かしていきます。 実験スクリプトをリモートで動かす際は nohup を使っても良いのですが、後で実験スクリプトの様子を確認したくなる場合もあるので、個人的には GNU screen を愛用しています。

```
#!/usr/bin/expect
#****h* utils/run_remote
# NAME
#  run_remote
# DESCRIPTION
#  execute run.sh in a remote machine
#
# USAGE
#  ./run_remote.tcl <ssh args> <experiment_id> <run.sh args>
#
# EXAMPLE
#  ./run_remote.tcl 127.0.0.1 20190102-1234-test_experiment ARGS
# PORTABILITY
#  We need expect <https://core.tcl-lang.org/expect/index> at /usr
```

```
/bin/expect.
#******

#****h* run_remote/lshift
# NAME
#  lshift
# DESCRIPTION
#  An implementation of the shift of perl. This implementation is
taken from Tcler's Wiki <https://wiki.tcl-lang.org/page/lshift>
#
# USAGE
#  lshift listVar
#
# EXAMPLE
#  lshift argv
#******
proc lshift listVar {
    upvar 1 $listVar l
    set r [lindex $l 0]
    set l [lreplace $l [set l 0] 0]
    return $r
}

if {[llength $argv] < 2} then {
    puts "Error: <ssh arguments> and <experiment_id> must be given
"
    puts "Usage: ./run_remote.tcl <ssh args> <experiment_id> <args
>"
    exit 1
}
```

```
set host [lshift argv]
set experiment_id [lshift argv]

set timeout 30

eval spawn ssh $host
expect {
    "(yes/no*)?" {
        send "yes\n"
        exp_continue
    }
    "*\\\$" {
        send "screen\n\n"
        expect {
            "*\\\$" {
                send "cd ./foo-experiments/${experiment_id}\n"
                expect "*\\\$"
                send "./run.sh [join $argv]\n"
                expect "ok*"
            }
        }
    }
}
exit 0
```

7. 実験後

最後に実験後の流れについて説明をします。

7.1. teardown.sh

まず始めに実験後に自動で行なわれる処理は teardown.sh (前述) に書かれている以下の内容

ものです。具体的なスクリプトの内容については 5.2 節を参照してください。

- 実験内容を `git commit` する
- slack に実験終了のお知らせをする
- 実験に使ったインスタンスを停止する

7.2. Git レポジトリの同期

次に Git レポジトリを同期させる必要があります。Git レポジトリの運用方針ですが以下の様にしていきます。

- 基本的に `master` ブランチを使う
- 但し実験後、マージ前の一時的な場合のみ他のブランチを使う

Git レポジトリの push

まず始めに Git レポジトリを push しますが、ここは例によって Ansible で行ないます。ここで重要なことですが、push 先のブランチは `tmp` から始まる一時的なもので、hostname が付いていて各インスタンスについて固有なものとなっているということです。なおインベントリは前述のコマンド・スクリプトで生成できます。Git レポジトリの push が終わったらインスタンスは不要なので停止させてください (無駄に課金されて勿体ないので)。

```
- hosts: aws
  tasks:
    - name: Install required packages
      apt:
        pkg:
          - git
      become: yes


    - name: push to the remote
      command:
        cmd: git push origin master:tmp_{{ ansible_hostname }}
        chdir: /home/ubuntu/bar-experiments
```

Git レポジトリの merge

次に Git レポジトリの merge を手元のマシンで行ないますが、早い話単に `git merge` するだけです。一つずつやっても良いですが、`tmp` から始まるリモートレポジトリを全部 merge させたいのであれば例えば次のコマンドで行なえます。

```
$ git branch -a | grep remotes/origin | grep tmp | sed 's:remotes/
origin/::' | xargs -I{} git merge {}
```

Git レポジトリの merge を終えて不要になったブランチは例えば次のコマンドで消せます。こちらは並列実行しても問題ないので、`xargs` の `-P` オプション (POSIX には入っていないですが `GNU xargs` を始めとして多くの `xargs` が `-P` に対応しています) を使っても大丈夫です。

```
$ git branch -a | grep remotes/origin | grep tmp | sed 's:remotes/
origin/::' | xargs -I{} git push origin :{}
```

8. ◊ (Eventually) 回予告

いかがだったでしょうか !!

今回は複数マシンでの実験を Ansible とかでシュっとした話のうち、実験を回して結果を回収するところまで説明をしました。今回の方法が最適解であるかはさておき、自分で並列実験システムを構築するときのたたき台になると幸いです。

さて、実験を回したら当然結果を解析する必要があります。解析パートは実験が並列であるかどうかとはあまり関係ないですが、自動解析も結構工夫のしがいがあるところです。ということで ◊ (Eventually) 回 は解析パートと題して、シェルスクリプトでの実験結果解析を説明する予定です。

おまけページ : C プログラムの形式検証について

wasabiz

みなさんこんにちは．この章ではおまけページとして C プログラムの検証についての記事の冒頭数ページをチラ見せしたいと思います．よく映画とかである「本編の劇場公開に合わせて今回特別に冒頭 10 分を先行公開！」みたいなやつですね．この yabaitech vol4 を刊行するにあたって社会情勢的な部分で非常に困難が多かったのですが，この記事も本来ならしっかり完成して頒布される予定でした．ここでチラ見せする記事はまだまだ作っている途中という雰囲気がつよいので，表紙ページにどうどうとタイトルを載せてお金を取るというのが非常に憚られました．そのためここではあくまでおまけとして掲載させていただきます．

Let's verify your C program!

wasabiz

1. はじめに

この記事では C 言語で書かれたプログラムの正しさを証明する手法について解説します. C のプログラムの性質を証明すると言っても世の中にはいろいろな手法があります. とりあえず挙げてみるなら大まかに以下の三つの方針がありえるでしょうか.

- すでに存在する証明をもとに絶対に正しい C プログラムを生成する. (プログラム抽出)
- C プログラムを入れると欲しい性質が満たされているか自動で検査してくれるツールを使う（自動検証）
- C プログラムを別の言語に変換して変換後のプログラムに証明をつける（自動 / 半自動検証）

一つ目の手法でよく使われるのは Coq とかでしょうか. Coq それ自体は依存型付のプログラミング言語でなんか証明も書けるようなものです. Coq で書ける (プログラマー的に興味のある) 証明はこの Coq に内蔵された依存型付言語に対するものなのですが, Coq にはそのような Coq プログラムを別の言語に (挙動を保ったまま) 変換する extraction という機能が備わっています. そのため欲しい C プログラムと同じ挙動をするプログラムを Coq 上で記述して Coq 上でそれについての証明を書き C プログラムに変換するということが出来ます. （たぶんデフォルトでは C への抽出に対応していないのでいろいろと頑張る必要がある. ）この方法は Coq というとても整備された環境で証明が書けるという利点はあるのですが既に出来上がっている C プログラムの検証をするには向いていません.

二つ目の手法はよく研究されている手法です. こちらは一つ目の手法と異なりすでに出来上がっている C プログラムの検証が可能です. 有名なところだと VeriFast や Facebook Infer などがあります. C 言語に限らなければ Dafny や Java PathFinder が有名でしょうか. とはいえ上にあげた 3 つはそれぞれ理論的 / 実装的に異なるものなので一概には比較できません. VeriFast の場合は Z3 を用いた Inductive Predicate 付きの Separation Logic を使った仕様記述と Z3 をベースにした Symbolic Execution による検証を実装しています. VeriFast はすでにある C プログラムに対して自動で検証ができしかも速い (らしい) という意味でとても良さそうなのですが, 一方で仕様記述言語が独自のため表現能力に限界があるという点と自動証明がうまく行かなかった場合につらい点がイマイチです. Infer も同じく Separation Logic ベースでメモリのバグなどを自動で検出してくれますが入力に仕様を与えられないのと健全

性がない (偽陽性がでる) 点はイマイチです.

三つ目の手法はややトリッキーですがおそらく最も現実的な手法です. 最も有名なのは Frama-C です. Frama-C は典型的には C プログラムを WhyML と呼ばれる ML ライクな言語に変換します. WhyML は Why3 というプロジェクトの成果で, Why3 では WhyML を検証するための様々なツール (例えば Coq で証明するためのライブラリや自動検証ツールなど) を提供しています.

本記事で扱う内容は三つ目の手法です. ただし使うツールは Frama-C + Why3 ではなく c-parser と呼ばれるツールです. c-parser は seL4 という証明付きマイクロカーネルを開発する上で作成されたツールです. seL4 はそれ自体が非常によくできておりカーネル開発者的視点からもプログラム検証的視点からもとても興味深いプロジェクトなのですが, それは一旦さておき c-parser は C で書かれたマイクロカーネルを Simpl という言語に翻訳するためのツールです. Simpl は While 言語に毛が生えたようなもので, WhyML とは異なり手続き型の側面を強く残しています. Why3 と Simpl の違いはソフトウェアスタックの大きさです. Why3(+Frama-C) は巨大かつレイヤーが多重になった構成ですがあ Simpl(+c-parser) はすべてが Isabelle/HOL 前提で作られており Isabelle/HOL の中だけで完結するように作られています. そのため盲目的に信頼しなければならないコンポーネントが少ないという特徴があります. また, Isabelle の既存の資産を最大限に活用することができます. Why3 の方がさまざまなプラグインを持っており機能自体は多いのですが, Simpl(+c-parser) の思想と seL4 の証明に使われたという実績を考慮してこちらを採用することにしました.

2. Isabelle とかの概要

Isabelle/HOL は高階論理に基づく定理証明支援系です. 使用感はざっくりいうと Coq みたいなものですが, Coq では証明も項も全て Coq の言語で記述するのに対して Isabelle はコアとなる論理の上でさまざまな「ライブラリ理論」が提供されています. よく使われるライブラリ理論（というかみんなこれしか使っていない）が「HOL」という理論です. 項として OCaml みたいな言語があり，それに対する証明を書く論理が別途提供されています. 依存型はないのでそこはうまいこと項 + 述語とかで対応します. ちなみに証明記述のための Isar という言語があってこれがとても便利です.

Simpl は 2005 年ごろに Norbert Schirmer の博士論文で提案されたプログラミング言語で

す．Simpl は C 言語や Java のような手続き的なプログラミング言語からそのエッセンスを抽出したような言語です．エッセンスを抽出した言語，というと普通は理論的にはよく振る舞うけど実用的ではない言語を指しがちなのですが，Simpl の場合は C 言語にある「あやしい機能」をほぼそのままサポートしています．たとえば，相互再帰，ポインタ，ミュータブルな局所変数は当然として，関数ポインタや配列の添字外参照も直接モデルできます．Simpl は Isabelle/HOL の中で AST が定義されており，専用の Hoare Logic も提供されています．

seL4 のリポジトリには c-parser というツールが含まれています． これは Michael Norrish によって開発された，C 言語のプログラムを Simpl へと変換するプログラムです． 正確には StrictC と呼ばれる C 言語のサブセットに対応しています．（StrictC は簡単にいうと C 言語から goto や union などを抜いたものです．）Isar の拡張もバンドルされておりこのツールだけでも十分使えはするのですが， 基本的には次に説明する AutoCorres と組み合わせて使います．

AutoCorres は StrictC で書かれたプログラムを意味を壊さずに Isabelle/HOL 向けに変換するソフトウェアです．こういうと c-parser との違いがよくわからないと思うのでもう少し解説すると， 実は c-parser は実際に使ってみると出力結果の Simpl コードが結構キビしい感じのものになりがちです（まあ実はというかありがちな話ですが）．そこで AutoCorres は c-parser の出力結果をさらに抽象化して人間に読みやすく検証しやすいプログラムに変換してくれます．さらにその際にもとのキビしい感じの Simpl のプログラムと出力後のキレイなプログラムの間で正しく対応が取れていること（i.e. 意味が壊れていないこと）の証明も出力してくれます（これが AutoCorres という名前の由来です）． この仕組みのおかげで人間にとっての読みやすさ・検証しやすさと信頼性を両立しています．

さて先立つ節でこの記事では c-parser を使いますと宣言したところですが，実際には AutoCorres を使って間接的に c-parser を呼び出します． また，内容的には AutoCorres のチュートリアルに大まかに従ったものになります．

3. はじめての AutoCorres

まずは一番簡単な例としてピュアな C プログラムの証明を行ってみたいと思います．実際に証明するコード `minmax.c` はこちら：

あとがき

— yabaitech.tokyo を支える技術 —

(大仰なタイトルですがそんな大したことは書かないです、悪しからず)

この度は yabaitech.tokyo vol.4 をお買い上げいただきありがとうございます。yabaitech.-tokyo は大学院時代の同期が集まってできたサークルで、本誌のような合同誌を出すことを主な活動内容にしています。毎度のことながら、忙しいなか時間を記事やイラスト、その他の作業に充ててくれるメンバーの皆には頭があがりません。

合同誌 yabaitech.tokyo は vol.1 から今回に至るまで SATYSFI で組版を行っています。SATYSFI は静的な型システムを持つフル機能のプログラミング言語を供えた組版処理システムで、LaTeX の代替を目指す新進気鋭のソフトウェアです。ちなみに今回の表紙を描いてくれた gfn によって開発されています。

フル機能のプログラミング言語で原稿を書けるというのは結構快適なものです。ただそうなると、ただ原稿を書くだけでなくソフトウェア開発の手法を取り入れていきたい、と思うのがエンジニアの性でして。実際今回の合同紙作るにあたっては色々な試みを行いました。そこらへんの話を「yabaitech.tokyo を支える技術」として紹介させてください。

汎用ライブラリの開発と公開　satysfi-base(https://github.com/nyuichi/satysfi-base) は SATYSFI の標準ライブラリの代替を目指した SATYSFI ライブラリで、「Let's verify your C program!」 の寄稿者であるところのわさびずが owner です。 もともとは彼が yabaitech.tokyo vol.3 用に書いたライブラリを切り出してオープンソース化したものがもとになっています。今

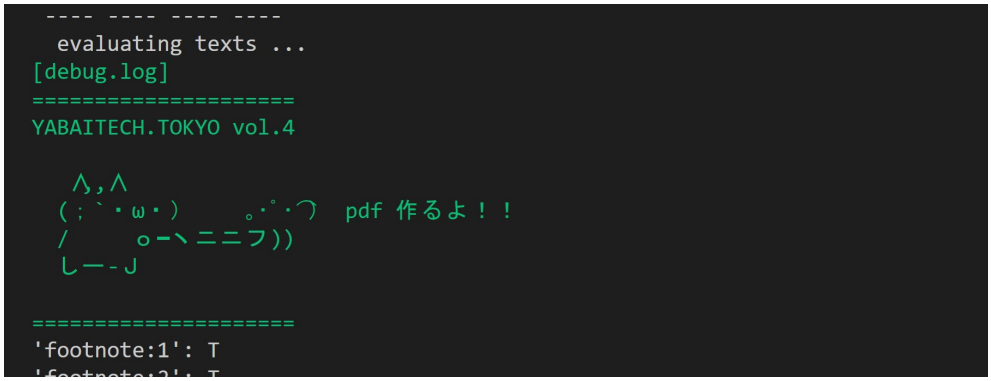

図 2　コンパイルをするとこんな感じのログがでます。かわいいですね。

回の vol.4 でも satysfi-base を多用しています。詳しくは「satysfi-base を使って関数型プログラミングで組版する」(https://qiita.com/wasabiz/items/ea55c65f23a5e187caf2) をご覧ください。

クラスファイルの回帰テスト　ソフトウェア開発をしているのであればテスト駆動開発をしたいですよね。しかしながらクラスファイルを書いている場合出力はもっぱら pdf ファイルになってしまい、テストを書くのはのはなかなか難しいところです。そういう事情で今回は代わりに回帰テストの導入を行いました。回帰テストは「既存の機能が壊れていないことを確認するためのテスト」で、やっていることは単純です。

- クラスファイルに新しいデザインやコマンドを追加したら、テスト用のコードとその時の pdf の出力を保存しておく
- コードに変更があったときにはテスト用のコードを再実行して出力が変わっていないかを確認する
- 出力の変更が期待したものであれば上書きする

回帰テストが特に便利なのはコードをリファクタする時です。つまり、リファクタするときに確認したいのは「そのコード変更が出力に影響を与えないこと」なので回帰テストがぴったりなわけですね。今回は vol.3 の際に使用したクラスファイルに回帰テストを導入してがっつりリファクタするということをやっていました。関連する記事として「SATySFi の組版結果をテストする」(https://qiita.com/zptmtr/items/59b485c47edfbca67c60) をご覧ください。

Github Actions による CI　Github Actions を使って原稿の pdf を作成したり回帰テストを実行したりします。標準的な環境の下で行うので現在のコードが特定個人の開発環境に依存していないことを確認できて良いです。セットアップにあたっては「GitHub Actions を使った文

書やパッケージの CI」(http://amutake.hatenablog.com/entry/2019/12/03/082528) を参考にさせてもらいました。

Satyrographos フォントライブラリの作成　vol.3 までは必要なフォントを自前でダウンロードして適切な設定をするということをやっていました。ところで SATySFi には有志のパッケージ管理システム Satyrographos があり、 これはフォントライブラリも配布することができます。折角なので必要なフォントを全部ライブラリとして登録してしまって satyrographos 経由でインストールできるようにできたらかっちょいいよね、という話になったので実際にやってみました。 以下が今回作成したフォントライブラリで対応したフォントの一覧です (括弧内はパッケージ名です)。

- DejaVu fonts (satysfi-fonts-dejavu)
- Asana-Math (satysfi-fonts-asana-math)
- Computer Modern Unicode (satysfi-fonts-computer-modern-unicode)
- Noto Sans (satysfi-fonts-noto-sans)
- Noto Serif (satysfi-fonts-noto-serif)
- Noto Sans CJK Jp (satysfi-fonts-noto-sans-cjk-jp)
- Noto Serif CJK Jp (satysfi-fonts-noto-serif-cjk-jp)

こうしてみると結構たくさん作っていますね。結構がんばったなあ僕。なおフォントライブラリを作成するにあたって Satyrographos の作者である na4zagin3 さんに多くの助言をいただきました。この場を借りてお礼申し上げます。

　というわけで合同誌の記事を書くのはもちろんのことながら、 このような形で SATYSF$_{\text{I}}$ で色々遊ぶというのもこのサークルの活動の一つになっています。それでは皆さま、また次回でお会いしましょう！

(文責 : zeptometer)

YABAITECH.TOKYO vol.4

2020 年 2 月 29 日 技術書典 8 版 (電子版)

2021 年 7 月 10 日 技術書典 11 版 (書籍版)

発行者　yabaitech.tokyo
Web サイト　http://yabaitech.tokyo
連絡先　admin@yabaitech.tokyo
印刷所　株式会社ポプルス