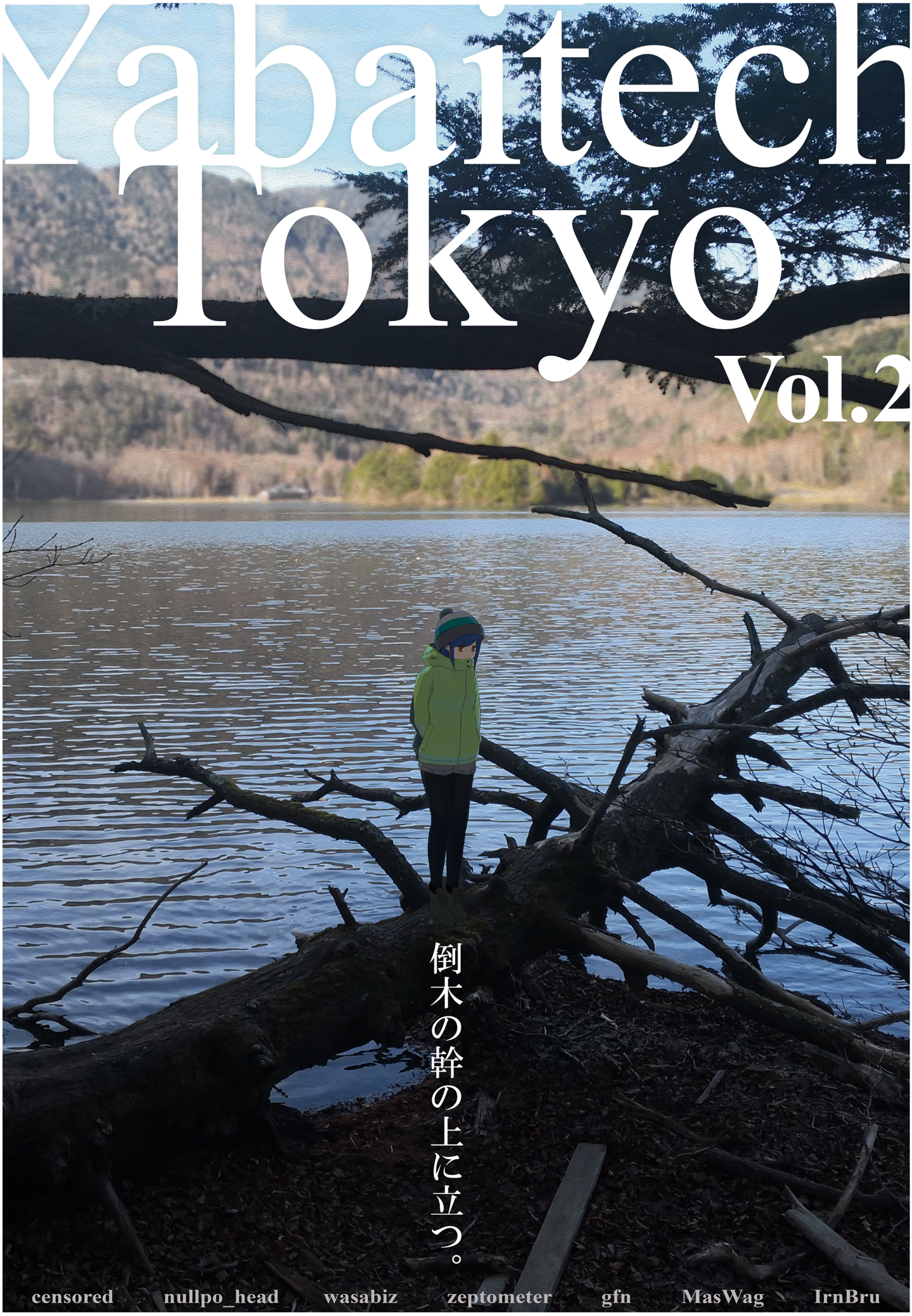
Yabaitech Tokyo Vol.2



censored　nullpo_head　wasabiz　zeptometer　gfn　MasWag　IrnBru

yabaitech.tokyo vol.2

2019.04.14@ 技術書典 6

目次

けもフレ bot を支える技術（導入編）

censored

本記事では、2017 年 2 月 6 日より稼働している「けものフレンズなんだね！ bot（@kemofre_bot）」（以下「けもフレ bot」）の AWS への実装時の作業録、および bot の解説を扱っていきます。AWS の 1 年間限定**ではない**無料枠を使って運用することを目標としていきます。本記事の扱う分野としては AWS lambda、MeCab、twitter bot の運用 tips あたりですが、解説というよりは作業録に近いのでだいぶ分かりづらいかもしれまん。因みに、「AWS lambda twitter bot」で Qxxta で検索すると 63 件ほどヒットするので、本記事で不明瞭な点があったらその辺を参照すれば大丈夫です！

1. なぜ今更けものフレンズ bot なのか

けものフレンズとはけものフレンズプロジェクトによるメディアミックス作品の名称 [1] です。2017 年 1 月より 3 月までテレビ東京ほかで第 1 期が放送され、2019 年 1 月より第 2 期けものフレンズ 2 がテレビ東京ほかで放送されました [2]。本記事ではけものフレンズ 1 期の頃のミームであった「すっごーい！キミは⁉ なフレンズなんだね！」と楽しくユーザーに話しかけてくれる twitter bot を AWS サービス上に実装していきます。なぜ今更こんなことをやるのかと言えば、実は 1 年ほど前まで twitter でけもフレ bot を運用していたので、手元に基本的なコードがあったからです。作成したのがもう 2 年前のことになるので最早当時のことは朧げにしか覚えていないですがざっくり経緯を書くと、内輪で楽しむために bot を作ったのが始まりでした。後々いろんな機能を付け加えていきましたが、bot の基本的なコンセプトは単純です。フォローの誰かが「世界一バグを生むのが得意」などと特定のキーワード（得意、好き等）を含むツイートしたときに、すかさず「すっごーい！あなたは世界一バグを生むのが得意なフレンズなんだね！」と返す、内輪に煽り

リプライを飛ばすための bot でした。界隈で有名な人に面白がられて以降フォロワーが増え、最大で 1 万フォロワー程度まで行きました。大学時代は大学に置きっぱなしだった個人用 PC を使って運用していましたが、 卒業とともに運用が終了しました。 このままアカウントも削除して良かったのですが、

- 停止した後もちょくちょく復活させてほしいとリプライや DM が来ていた
- AWS の勉強をしたいと思っていた
- 2 期をやっていた[*1]

の 3 つの理由からこの bot を AWS 上に移植することにしました。EC2 へ移すだけなら非常に簡単なのですが、 こんな bot のためにお金を払うのも馬鹿馬鹿しいので、AWS の無料枠に収まるように頑張っていきます。

2. AWS lambda について

AWS lambda とはサーバーのプロビジョニングや管理をすることなくコードを実行することができるサービス [3] です。いわゆるサーバーレスアプリケーションを運用するためのクラウドリソースで、 常に多くのトラフィックが来ることが期待されないアプリケーションの運用に向いています。 サーバーレスアプリケーションでは何かしらのイベント（今回は cron の様な一定時間）をトリガーとして何かしらの処理（今回は twitter のタイムラインの読み取りとツイート処理） が実行されます。AWS lambda には lambda function と lambda layer の 2 種類があります。Lambda function は処理を行うコード本体、lambda layer は様々な lambda function で使用できる zip アーカイブです。Lambda function は lambda layer をバージョン区切りで読み込むので lambda layer を更新した場合、それを使用する lambda function も更新しなければなりません。 よって、lambda layer では頻繁に開発を行わない既存のライブラリ等、lambda function では頻繁に開発を行うコードを使うように書き分けることで効率よく開発を行うことができます。 今回は既存の python ライブラリや MeCab バイナリ、MeCab 辞書などを layer として扱っています。AWS lambda では月あたり 1,000,000 件のリクエスト、400 GB- 秒のメモリ使用、3,200,000 秒の計算時間が**無料**で利用できます。 これらの内どれか一つでも超過するとメモリの使用量 × 計算時間に比例した金額がかかります。

---

1 色々「話題」となった 2 期もなんとかこの記事を書きながら視聴しています。執筆時現在 11 話まで見ましたが、3 話のイルカと 9 話のイエイヌが可愛いと思います。

3. 環境セットアップ

3.1. AWS コマンドラインインターフェース（AWS CLI）のセットアップ

コンソールからではローカルからは 10MB 以内のファイルしかアップロードできないため AWS CLI を利用します。但し、AWS CLI でも 70MB 以内のファイルしかアップロードできないので、MeCab は S3 経由で lambda にアップロードする必要があります。インストール方法は公式サイト [4] を参照してください。個人的には `pip install awscli` でよいと思います。

4. MeCab+ カスタム NEologd IPA 辞書の lambda layer の作成

4.1. Lambda 用の MeCab ipadic-neologd のビルド

事前準備

[8] を参考に行いました。AWS 公式の lambda のサポート情報のページ[*2]に行って、記載されている AWS lambda のイメージを使って EC2 インスタンス（2019 年 3 月現在 amzn-ami-hvm-2017.03.1.20170812-x86_64-gp2）を起動します。以下、ログイン後のコマンドです。

```
$ # 環境を最新にしておきます。
$ sudo yum update -y
$ # MeCab のビルドのため g++ をインストールします。
$ sudo yum install gcc-c++ -y
$ # NEologd のビルドのため patch、git、autodie をインストールします。
$ sudo yum install patch git perl-autodie.noarch -y
$ # g++ へのパスを通します。
$ sudo ln -s /usr/libexec/gcc/x86_64-amazon-linux/4.8.5/cc1plus \
$ /usr/local/bin/
```

2 https://docs.aws.amazon.com/ja_jp/lambda/latest/dg/current-supported-versions.html

これで準備は完了です。

MeCab のダウンロードとビルド

MeCab 公式サイト[*3]へ行って最新のものを確認しましょう。以降様々なパスを /opt 以下に設定しますが、これにより実際に動かした際に様々なリンクの解決でこの prefix が用いられる為、自前で PATH や LD_LIBRARY_PATH の設定をする必要がなくなります。

```
$ cd ~
$ curl -L \
$ "https://drive.google.com/uc"\
> "?export=download&id=0B4y35FiV1wh7cENtOXlicTFaRUE" \
> -o mecab-0.996.tar.gz
$ tar -zxvf mecab-0.996.tar.gz
$ cd mecab-0.996
$ # インストール先を /opt/ に設定します。
$ sudo mkdir -p /opt
$ sudo ./configure --prefix=/opt --with-charset=utf8
$ sudo make
$ sudo make install
```

IPA 辞書のダウンロードとビルド

```
$ cd ~
$ curl -L \
> "https://drive.google.com/uc"\
> "?export=download&id=0B4y35FiV1wh7MWVlSDBCSXZMTXM" \
$ -o mecab-ipadic-2.7.0-20070801.tar.gz
$ tar -zxvf mecab-ipadic-2.7.0-20070801.tar.gz
$ cd mecab-ipadic-2.7.0-20070801
$ sudo ./configure --prefix=/opt --with-charset=utf8 \
$ --with-mecab-config=/opt/bin/mecab-config
```

3 http://taku910.github.io/mecab

```
$ sudo make
$ sudo make install
```

mecab-python3 のダウンロードとビルド

```
$ cd ~
$ sudo yum install swig -y
$ wget –quiet \
$ https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh \
$ -O ~/miniconda3.sh
$ /bin/bash miniconda3.sh -b -p $HOME/miniconda3
$ echo 'export PATH=$HOME/miniconda3/bin:$PATH' >> ~/.bashrc
$ echo 'export PATH=$PATH:/opt/bin/' >> ~/.bashrc
$ echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/lib/' >> ~/.bashrc
$ source ~/.bashrc
$ sudo chown -R ec2-user /opt
$ mkdir -p /opt/python/
$ pip install mecab-python3 -t /opt/python/
```

mecab-ipadic-NEologd のダウンロードとビルド

IPA 辞書は高品質な辞書ですが、一方新語や流行語に弱いという欠点があります。それに対し考案されたのが mecab-ipadic-NEologd[9] です。

ここではまず [10] にしたがって、elimintate-redundant-entry オプションをつけることによって最小構成でインストールを試みます。

```
$ cd ~
$ git clone --depth 1 https://github.com/neologd/mecab-ipadic-neologd.git
$ cd mecab-ipadic-neologd/
$ ./bin/install-mecab-ipadic-neologd -y \
$ -p /opt/neologd-minimum -n -eliminate-redundant-enrty
```

この時点で $HOME/neologd の中身は 400MB 以上ありますが、zip 圧縮すると 90MB 程度になるので、S3 経由で lambda function/layer をアップロードすることができます。しかし、まだ lambda の /tmp ディレクトリのストレージ上限である 512MB までは 100MB の余裕があり、その分辞書の精度を上げられないか試します。ところが、他のオプションでは t2.micro instance のメモリが足りずにビルドが途中で落ちてしまいます。仕方ないので、t2.medium instance をお金を払って起動して ignore-all オプションのみでビルドしてみました。比較してみると、システム辞書の部分が大きく異なっていることが分かります。

```
neologd-ignore-all:
合計 702M
-rw-r--r-- 1 ec2-user ec2-user 257K  2月  7 16:07 char.bin
-rw-r--r-- 1 ec2-user ec2-user  693  2月  7 16:07 dicrc
-rw-r--r-- 1 ec2-user ec2-user  73K  2月  7 16:07 left-id.def
-rw-r--r-- 1 ec2-user ec2-user 3.4M  2月  7 16:07 matrix.bin
-rw-r--r-- 1 ec2-user ec2-user 1.9K  2月  7 16:07 pos-id.def
-rw-r--r-- 1 ec2-user ec2-user 7.3K  2月  7 16:07 rewrite.def
-rw-r--r-- 1 ec2-user ec2-user  73K  2月  7 16:07 right-id.def
-rw-r--r-- 1 ec2-user ec2-user 698M  2月  7 16:07 sys.dic
-rw-r--r-- 1 ec2-user ec2-user 5.6K  2月  7 16:07 unk.dic

neologd-minimum:
合計 406M
-rw-r--r-- 1 ec2-user ec2-user 257K  2月  7 16:21 char.bin
-rw-r--r-- 1 ec2-user ec2-user  693  2月  7 16:21 dicrc
-rw-r--r-- 1 ec2-user ec2-user  73K  2月  7 16:21 left-id.def
-rw-r--r-- 1 ec2-user ec2-user 3.4M  2月  7 16:21 matrix.bin
-rw-r--r-- 1 ec2-user ec2-user 1.9K  2月  7 16:21 pos-id.def
-rw-r--r-- 1 ec2-user ec2-user 7.3K  2月  7 16:21 rewrite.def
-rw-r--r-- 1 ec2-user ec2-user  73K  2月  7 16:21 right-id.def
-rw-r--r-- 1 ec2-user ec2-user 403M  2月  7 16:21 sys.dic
-rw-r--r-- 1 ec2-user ec2-user 5.6K  2月  7 16:21 unk.dic
```

この主な原因は seed/mecab-user-dict-seed.20190204.csv.xz なのですが、中を見てみると結構程頻度そうな単語も多く収録されていることがわかります。ここで、「語数の長い形態素は低頻出である」「NEologd 辞書に登録されている語数の長い形態素の大半はより短い形態素に分解可能である」という経験的な仮定をおき、性能への悪影響を最小限

にしつつ低頻出な形態素を取り除きます。/bin/install-mecab-ipadic-neologd に max_surface_length というオプションがあるので、これを用いて長い形態素を辞書から削除していきます。しかし、このままでは IPA の辞書にもこの長さ制限がかかってしまい、有用な長い形態素（国名など）も除かれてしまうので、`libexec/make-mecab-ipadic-neologd.sh` を修正し、`mecab-user-dict-seed.20190204.csv` にのみ選択的に反応するようにします。差分としては次のようになります。

```
diff --git a/libexec/make-mecab-ipadic-neologd.sh b/libexec/make-mecab-ipadi
c-neologd.sh
index 5b7ae14..179efe9 100755
--- a/libexec/make-mecab-ipadic-neologd.sh
+++ b/libexec/make-mecab-ipadic-neologd.sh
@@ -449,8 +449,10 @@ if [ ${MIN_SURFACE_LEN} -gt 0 -o ${MAX_SURFACE_LEN} -
gt 0 ]; then
              fi
              if [ ${MAX_SURFACE_LEN} -gt 0 ]; then
+               if [[ ${TMP_SEED_FILE_NAME} =~ mecab-user-dict-seed ]]; then
              echo "${ECHO_PREFIX} Delete the entries whose length of surfac
e
is longer than ${MAX_SURFACE_LEN} from seed file"
              cat ${NEOLOGD_DIC_DIR}/${TMP_SEED_FILE_NAME} | perl -ne "use
Encode;my \$l=\$_;my @a=split /,/,\$l;\$len=length Encode::decode_utf8(\$a
[0]);
print \$l if(\$len <= ${MAX_SURFACE_LEN});" >
${NEOLOGD_DIC_DIR}/${TMP_SEED_FILE_NAME}.tmp
              mv ${NEOLOGD_DIC_DIR}/${TMP_SEED_FILE_NAME}.tmp
${NEOLOGD_DIC_DIR}/${TMP_SEED_FILE_NAME}
+             fi
              fi
            fi
        done
```

その後、

```
$ ./bin/install-mecab-ipadic-neologd -n -y \
> -p /opt/neologd-custom \
> --ignore_adverb \
> --ignore_interject \
> --ignore_noun_ortho \
```

```
> --ignore_noun_sahen_conn_ortho \
> --ignore_adjective_std \
> --ignore_adjective_verb \
> --ignore_ill_formed_words \
> --max_surface_length 11 \
> --min_surface_length 1
```

を適用したところ、ファイルサイズは 501MB になりました。この 2 つの辞書を比較して見ます。けものフレンズの単語は今では多くの単語が登録されているため、作成当時のようなカスタム辞書を自分で作る手間は必要ありませんでした。すっごーい！

```
$ mecab -d neologd-minimum
うみゃー！やってみたーい！
う　感動詞,*,*,*,*,*,う,ウ,ウ
み　接頭詞,名詞接続,*,*,*,*,み,ミ,ミ
ゃ　名詞,一般,*,*,*,*,*
ー　名詞,一般,*,*,*,*,*
！　記号,一般,*,*,*,*,！,！,！
や　助詞,並立助詞,*,*,*,*,や,ヤ,ヤ
って　助詞,格助詞,連語,*,*,*,って,ッテ,ッテ
み　接頭詞,名詞接続,*,*,*,*,み,ミ,ミ
た　助動詞,*,*,*,特殊・タ,基本形,た,タ,タ
ー　名詞,一般,*,*,*,*,*
い　名詞,一般,*,*,*,*,い,イ,イ
！　記号,一般,*,*,*,*,！,！,！
EOS

$ mecab -d neologd-custom
うみゃー！やってみたーい！
うみゃ　動詞,自立,*,*,五段・マ行,仮定縮約１,うむ,ウミャ,ウミャ
ー　名詞,一般,*,*,*,*,*
！　記号,一般,*,*,*,*,！,！,！
やっ　動詞,自立,*,*,五段・ラ行,連用タ接続,やる,ヤッ,ヤッ
て　助詞,接続助詞,*,*,*,*,て,テ,テ
み　動詞,非自立,*,*,一段,連用形,みる,ミ,ミ
た　助動詞,*,*,*,特殊・タ,基本形,た,タ,タ
ー　名詞,一般,*,*,*,*,*
い　動詞,自立,*,*,一段,連用形,いる,イ,イ
！　記号,一般,*,*,*,*,！,！,！
EOS
```

```
$ mecab -d neologd-minimum
いいとこまできてるですね、やりますね
いい　形容詞 ,自立 ,*,*, 形容詞・イイ ,基本形 ,いい ,イイ ,イイ
とこ　名詞 ,一般 ,*,*,*,*, とこ ,トコ ,トコ
まで　助詞 ,副助詞 ,*,*,*,*, まで ,マデ ,マデ
き　助動詞 ,*,*,*, 文語・キ ,基本形 ,き ,キ ,キ
てる　名詞 ,固有名詞 ,人名 ,名 ,*,*, てる ,テル ,テル
です　助動詞 ,*,*,*, 特殊・デス ,基本形 ,です ,デス ,デス
ね　助詞 ,終助詞 ,*,*,*,*, ね ,ネ ,ネ
、　記号 ,読点 ,*,*,*,*,、,、,、
や　助詞 ,並立助詞 ,*,*,*,*, や ,ヤ ,ヤ
り　助動詞 ,*,*,*, 文語・リ ,基本形 ,り ,リ ,リ
ます　助動詞 ,*,*,*, 特殊・マス ,基本形 ,ます ,マス ,マス
ね　助詞 ,終助詞 ,*,*,*,*, ね ,ネ ,ネ
EOS

$ mecab -d neologd-custom
いいとこまできてるですね、やりますね
いい　形容詞 ,自立 ,*,*, 形容詞・イイ ,基本形 ,いい ,イイ ,イイ
とこ　名詞 ,一般 ,*,*,*,*, とこ ,トコ ,トコ
まで　助詞 ,副助詞 ,*,*,*,*, まで ,マデ ,マデ
き　動詞 ,自立 ,*,*, カ変・クル ,連用形 ,くる ,キ ,キ
てる　動詞 ,非自立 ,*,*, 一段 ,基本形 ,てる ,テル ,テル
です　助動詞 ,*,*,*, 特殊・デス ,基本形 ,です ,デス ,デス
ね　助詞 ,終助詞 ,*,*,*,*, ね ,ネ ,ネ
、　記号 ,読点 ,*,*,*,*,、,、,、
やり　動詞 ,自立 ,*,*, 五段・ラ行 ,連用形 ,やる ,ヤリ ,ヤリ
ます　助動詞 ,*,*,*, 特殊・マス ,基本形 ,ます ,マス ,マス
ね　助詞 ,終助詞 ,*,*,*,*, ね ,ネ ,ネ
EOS
```

eliminate-redundant-enrty オプションが MeCab の公式ドキュメントでは非推奨であった通り、やはり性能が大きく異なっています。eliminate-redundant-entry オプションは辞書の単語に非可逆な正規化を施しビルドするため、実際の入力データも正規化を施した後に mecab に入力するようなケース、すなわち検索や推薦など単語の正規形のみ得られれば十分な場合では有効ですが、今回のような鸚鵡返しを想定した状況では正規化無しの入力が

前提とされる為、使用するのは難しいことが改めて確認されました[*4]。これにて MeCab を lambda で動かすのに必要なファイルは揃ったので、/opt 以下の /opt/aws 以外を scp なり何なりでローカルに落としておきます。 尚、/opt/lib/mecab 以下にあるデフォルトの IPA 辞書は必要ない上無駄に容量を取るので除いた方が良いです。

Lambda layer として固める

先ほど作成したカスタム IPA-NEologd 辞書は neologd-minimum であれ、neologd-custom であれ zip ファイルとして圧縮しないと lambda で動かすことができません。 そのため、 まずは辞書を zip で圧縮し、 その zip ファイルを展開するための python ファイル `python/unzip_neologd.py` を記述します。

```
from zipfile import ZipFile
import os
import traceback
import re

def unzip_neologd():
  with ZipFile('/opt/neologd.zip', 'r') as zipObj:
     # Extract all the contents of zip file in different directory
     zipObj.extractall('/tmp/')
  return "/tmp/neologd"
```

そしてローカルに落とした必要なファイルを全て固めた zip ファイルを S3 にアップロードし、コンソールからソース元として指定します。

実際に動かす際には

```
import MeCab
from unzip_neologd import unzip_neologd
neologd_path = unzip_neologd()
tagger = MeCab.Tagger(" -d " + neologd_path)
print(tagger.parse("けもフレ 2 期のイエイヌがちょっと可愛そう。"))
```

4　ここまでに t2.micro t2.medium で $0.06 かかってしまい当初の無料で達成するという目標は潰えています。MeCab 辞書のビルドも無料でできる方法があったらどなたか教えて下さい。

の様にすることでカスタム辞書で MeCab を使用することができます。

5. Twitter bot を動かす

本節では実際に lambda を用いて twitter bot を動かしていきます。正直なところ今回の bot で技術的に一番難しいのは前節の MeCab の動かし方なので、これ以降は細かい技術的な工夫の話になります。

5.1. 準備

Twitter API 用のアクセスキーの取得

Twitter API を呼ぶのに必要なアクセスキーを Twitter Developer Platform[*5]で申し込みます。 現在では審査がだいぶ厳しくなってしまっていますが、 当時はもう少し楽だったので開発用アカウントと運営用アカウントそれぞれでキーを取得しています。

Tweepy の lambda layer の作成

Twitter bot を動かすのに今回は tweepy を用います。MeCab のビルド時に用いた環境（今回は t2.micro でも大丈夫です）を用いてインストールします。 今回は MeCab と別の lambda layer としてインストールしたいので、別のディレクトリにインストールします。

```
$ sudo rm -r /opt/python
$ pip install tweepy -t /opt/python/
```

/opt/python をローカルにダウンロードし、zip で固めて lambda layer を作成します。 今度は CLI でも作成できるはずです。

```
$ aws lambda publish-layer-version --layer-name tweepy \
> --description "tweepy python library" \
> --zip-file fileb://<path to file>/tweepy_layer.zip \
> --compatible-runtimes python3.7
```

5 https://developer.twitter.com/

5.2. 基本動作の実装

さて、こうして lambda 上で twitter bot を動かす準備はできました。けもフレ bot の機能の内、最も単純な機能である「一定時間毎にタイムラインからランダムにツイートを取得し、対応するツイートをする」の部分の実装に入ります。が、その前にいくつか気をつけておく部分があるので記しておきます。

Python での日本語処理

lambda はデフォルトだと日本語（多分マルチバイト文字）に対応していないので `# -*- coding: utf-8 -*-` を lambda function のソースコードに加えるようにします。

mecab-python3 の処理速度

[11] によれば、mecab-python3 の parseToNode メソッドは parse メソッドに比べ 2 倍以上遅いようです。4 年前の記事だったので一応手元で追試験を行ったところ、 確かに 2 倍程度の差があったので確かなようです。 機能をフルに使おうとしない限り、 自作で parseToNode を実装した方が良さそうです。

空白の処理

空白の含まれた文字列を MeCab に入力すると、通常空白は出力結果には含まれません。しかしこの設定では、 例えば 「サーバル タベチャダメダヨ」 という半角空白文字列を MeCab に入力すると形態素解析の判断には用いられますが出力には現れず、元の文字列を復元することが困難です。そこで、[12] で紹介されているオプションを MeCab の Tagger に与えます。

```
r' --node-format=%M\t%f[0],%f[1],%f[2],%f[3],%f[4],%f[5],%f[6],%f[7],%f[8]
\n'
r' --unk-format=%M\t%f[0],%f[1],%f[2],%f[3],%f[4],%f[5],%f[6]\n '
```

これにより「 タベチャダメダヨ」のように文中の半角スペースが次の単語と結合して見出し語として表示されるようになります。

無料枠内での許容計算時間

さて、Lambda の無料枠内で bot で動かすことは実際に可能かを考えてみます。MeCab を lambda で実際に動かすのに必要なメモリは 1024MB あれば大丈夫そうです。Lambda は月間 400,000GB 秒のメモリ計算時間が無料なので、400,000 * 1024 / 1024 / 31 / 24 / 60 = 8.96 回 / 分のペースで function を呼び出すことができます。また 3,200,000 秒の計算時間が無料なので、3,200,000 / 31 / 24 / 60 = 71.68 秒 / 分 function を呼び出すことができます。 けもフレ bot を一回稼働させるのにかかる時間は大体 20 秒以内なので 20 秒おき程度に呼び出すことが可能です。

6. 導入編まとめ

ここで締め切りが来てしまいました。表紙を描いたりサークルカットを描いたりしたので許してください。実は 3 月中旬頃に bot の最小限の機能である通常のツイートを再開しています。しかし、他の機能（リプライ、フォローなど）は凍結回避の為に様々な工夫を施す必要がある為まだ実装中です。

Google Cloud Platform では f1.micro インスタンスが 1 インスタンス年分無料なので、GCP を使えばこんな面倒なことはせずに簡単にできます。GCP 使いましょう。

参考文献

[1] フリー百科事典　ウィキペディア日本語版 . けものフレンズ . https://ja.wikipedia.org/wiki/ けものフレンズ , 2019.

[2] フリー百科事典　ウィキペディア日本語版 . けものフレンズ *(アニメ)*. https://ja.wikipedia.org/wiki/ けものフレンズ _(アニメ), 2019.

[3] Amazon Web Service. *AWS Lambda (*サーバーレスでコードを実行・自動管理*)* | *AWS*. https://aws.amazon.com/jp/lambda/, 2019.

[4] Amazon Web Service. *AWS CLI* のインストールと設定 - *Amazon Kinesis Data Streams*. https://docs.aws.amazon.com/ja_jp/streams/latest/dev/kinesis-tutorial-cli-installation.html, 2019.

[5] Satoru Kadowaki. *AWS APIGateway + Python Lambda + NEologd* で作るサーバレス日本語形態素解析 *API - Speaker Deck*. https://speakerdeck.com/satoruka-dowaki/aws-apigateway-plus-python-lambda-plus-neologddezuo-rusa-baresuri-ben-yu-xing-tai-su-jie-xi-api, 2017.

[6] Ichinose Shogo. *MeCab* を *AWS Lambda* で動かす *(2017* 年版*)*. https://shogo82148.-github.io/blog/2017/12/06/mecab-in-lambda/, 2017.

[7] marmarossa. *AWS* の *lambda* 上で *MeCab* を実行する *(*他のバイナリへも応用可*)!! - marmarossa's blog*. http://marmarossa.hatenablog.com/entry/2017/02/03/223423, 2017.

[8] @rtaguchi. *AWS Lambda* で *MeCab* を動かす（*2018* 年 *9* 月時点）. https://qiita.-com/rtaguchi/items/e6cb594196fa8186e9a2, 2018.

[9] Toshinori Sato. *mecab-ipadic-NEologd : Neologism dictionary for MeCab*. https://github.com/neologd/mecab-ipadic-neologd/, 2015.

[10] @carrotflakes. *AWS Lambda* で *MeCab + NEologd* 辞書を *Node.js* で動かす – *Qiita*. https://qiita.com/carrotflakes/items/d3bab5a81817c7e7edc0, 2018.

[11] @yukinoi. *Python* での *MeCab* を速くする *tips* – *Qiita*. https://qiita.com/yuki-noi/items/81a707c1317c97f5fdf9, 2015.

[12] nullnull. *mecab* で半角スペースを省略せず認識させる - *Into the Horizon*. https://nullnull.hatenablog.com/entry/20120629/1340990207, 2012.

Writing a (micro)kernel in Rust in 12 days

—The first 3 days—

nullpo_head

1. Introduction

ハロー！私の認識が正しければ、あなたにはこのページの上部にでかでかと書いてある記事のタイトルが見えているはずだ。見えるかい？ "Writing a (micro)kernel in Rust in 12 days"だ。 見えたならきっと、 この記事が一体何をする記事なのかはわかっているね？そう、ずばり、12 日間で Rust でマイクロカーネルを書き上げるのだ！

この記事は、 私が Rust でカーネルを書いてみたいと思ったことから始まった開発ログだ。Rust 言語は、2019 年の今最も注目されているプログラミング言語の一つで、 低レイヤーの分野においては、特にモダンな言語機能を一通りそろっていることによる快適性と、所有権という新しい概念による GC に頼らないメモリ安全性から注目されている。(C 言語でのメモリ関連のデバッグの苦労を思い出せ！) コミュニティ全体で、システムレイヤ ―での応用を考慮しながら開発が進められているのも魅力的なポイントだ。だから、2019 年にカーネルを書く際に Rust でやってみようと思うのは全く奇妙なことではないだろう。

一方カーネルは、 ご存知オペレーティングシステムの中核コンポーネントだ。CPU のもっとも権限の高い空間で動作し、コンピュータのすべてを支配しなければならない。それを 0 から自分で書くというのはとてもエキサイティングな体験だと思う。プログラマーの「3 大作ってみたいもの」によくあがる、 コンパイラ・ CPU・オペレーティングシステムのまさに 3 つ目にもあたるしね。 そして今回書いていくカーネルはただのカーネルではなくマイクロカーネルだ。

マイクロカーネルとは何か、という詳しい話はあとに回すとして、今回マイクロカーネ

ルを選んだ大きな理由は二つある。一つは、マイクロカーネルはサイズがとても小さいということ。実用レベルの OS である Redox でも、マイクロカーネル部分のコードは 1 万行未満だ。Linux カーネルのような巨大なモノリシックカーネルを自作してみようというのは途方もない話だが、マイクロカーネルならば短い期間でゼロからでも完成させることができるはずだ。実は私は去年ついに大学院を卒業して会社員になり、（あるいはなってしまい、）無事自由時間が今までの 7 分の 2 に減ってしまった！だから短い期間で完成させたいというのは実に切実な願いで、そのために作るカーネルとしてマイクロカーネルを選んだのは自然なことだったわけだ。

そして、もう一つの理由は、L4 マイクロカーネルファミリーという有名なマイクロカーネルの仕様が存在すること。実装するべき完成図がはっきりしているから学びながら開発していくのに都合がいいし、しかも無事仕様をみたせたときには自作したカーネルのうえで、既存のユーザーランドアプリケーションが動作することになる。OS を自作するとき、こんなに嬉しいことはないだろう！小さな労力でこれを達成したら、あとはサーバーなどの自分のユーザーランドをインクリメンタルに開発してゆくことも可能だ。L4 マイクロカーネルは実世界で使われた実績もあってまだ活発に開発・研究がされている。そして今回扱う L4 X.2 Standard のいいところは、何と言ったってシステムコールが 12 個しかないことだ。そう聞いたらなおさら短期間で作れそうな気がしてこないかい？仕様があるってことに加えて、しかもその仕様が小さいわけだ。これは一つ目の理由をさらに強いものにしてくれるね。

色々と説明したけれど、つまりはマイクロカーネルは自作をするにはピッタリだってこと！そんなわけで、私は今から Rust でマイクロカーネルを書き始める、というわけだ。

前提知識

さて、この記事を読むにあたって前提とする知識がわりとある。一つめ。オペレーティングシステムの開発に関するベーシックな理解。オペレーティングシステムの基本的な概念 - 例えばカーネルとかデバイスドライバ、プロセスとかいったもの - について、実際に自分でゼロから書いて見たことはなくていいから、だいたいの仕組みのイメージがわくことは前提としたい。つまりは学部の実験レベルの知識だ。本当は普通の Web プログラミング経験があれば十分、ということにしたかったんだけれど、残念ながら私の原稿の速度と締め切りがそれを許さなかった！本文中で、C 言語のコード、Makefile、それにリン

カスクリプトだとかが簡単な説明だけで出てくることになる。ただし詳細な説明こそはしないが、Wikipedia の一段落目に出てきそうな説明文くらいは書く元気はあるから、そういう大学時代の知識の記憶が怪しい人や、大学時代の記憶がたまたままだないって人も、ある程度安心してほしい。

そして二つめ。さらに、Rust 言語のチュートリアル程度の知識。残念ながら、この記事では Rust 言語の文法や所有権など基本的な概念の入門まではカバーしない予定だ。でもチュートリアルをやったことがあるというだけで十分。なぜかって？私がそうだから！私の Rust 経験はまだチュートリアルを終わらせたことと簡単な 2 つのプログラムを書いたことしかない。ただしこの記事では、必要になる知識をあらかじめ系統的に章立てして、天下り的に読者の皆さまに説明していくことは**しない**。そうではなくて、12 日間という期間でマイクロカーネルを実際に私が step by step で実装し、その過程を綴っていく。そして、開発に必要になることや私が経験したトラブルを読者の皆さんと共有する。だから、低レイヤ開発で必要になる Rust の知識、それに単に Rust 初心者がぶつかるよくある問題なんかもカバーしていけるだろう。(私が体を張ることによってね。)

基本的には、この記事はマイクロカーネルを作るための体系だった教科書であるよりかは、私のマイクロカーネル開発の日記により近いものとなる予定だ。私が実際に開発した際に（若干）読者の皆様を意識しながらそのログを残し、それにたいして抜け落ちている説明を後から書き加える形にする（予定）。だからきっと、このシリーズは読者の皆様が実際に Rust でマイクロカーネルを作り上げてみるのにはピッタリなものとなるに違いない。ぜひ読者の皆様も、私の 12 日間を文章で追体験するだけでなく、1 日ずつコードを書いていき、マイクロカーネルをゼロから作り上げるエキサイティングな 12 日間を実際に過ごしてほしい。それでは、早速私 (達の) 開発 1 日目を始めよう。

2. 1st day: L4 Microkernel

記念すべき第 1 日は、L4 マイクロカーネルに関する概念の整理だ。普通のカーネルと違って、マイクロカーネルは普段 Linux や macOS で開発を行っていては、あまりなじみがなくても仕方がない。[*6]そこで 1 日目では、具体的な目標である L4 の仕様についての

6 Windows に触れると話はややこしくなる。Windows はマイクロカーネルの要素が色濃く残っているのだ。だから少なくともしばらくは Windows の話はしない。

話を始める前に、マイクロカーネルの概念についても整理しようと思う。そして最後に、実際に L4 X.2 の実装である Pistachio カーネルをビルドして、Qemu 上で動かしてみる。そうすれば、明日以降私が何を作ればいいのかがはっきりするだろう。

2.1. What is Microkernel?

早速だが、マイクロカーネルとは何だろう？ひと段落で言えばだいたい次のようになる。

> マイクロカーネル（英：microkernel）とはオペレーティングシステムの設計思想、及びそのような OS のカーネル部の名称である。
> OS が担う各種機能のうち、必要最小限のみをカーネル空間に残し、残りをユーザーレベルに移すことで全体の設計が簡素化でき、結果的にカスタマイズ性が向上し、性能も向上できるという OS の設計手法のことである。
> カーネル本体が小規模な機能に限定されるので「マイクロカーネル」と呼ばれるが、必ずしも小さな OS を構成するとは限らない。

以上、Wikipedia 日本語版 [1] からの引用だ。Wikipedia の一段落目に出てきそうな説明文くらいは書くといったが、その約束を早速果たしてしまった。ちなみに、マイクロカーネルの対義語にあたる、普通のカーネルの呼び名はモノリシックカーネルという。Linux カーネルはこのモノリシックカーネルの典型的な例だ。とはいっても、これだけじゃ何が何だかという感じだろうし、重要な特徴をかいつまんで説明していこう。

極小性

まず、重要なことは、さっきの引用で説明されているように、カーネルには必要最小限の機能のみがあるということだ。 カーネル空間では、CPU の特権命令を発行する必要があるような、本当にプリミティブな機能だけが動く。そして、それ以上の機能は、ユーザー空間で動くプロセスが、そのプリミティブな機能をシステムコールとして使うことで実現していく。これらのユーザー空間のプロセスはマイクロカーネルの世界では、「サーバー」と呼ばれる。L4 カーネルを例にとれば、カーネル空間で動く機能はおおまかに次の 5 つだけとなる。

(1) スレッドプリミティブ

(2) スケジューリング

(3) メモリ空間の制御

(4) IPC

(5) 割り込み制御

もしかしたら、一見カーネル空間が担う機能としては普通のリストに見えるかもしれない。だから、普通の Linux や macOS のカーネル空間には存在しているけれど、このリストでは実は欠けているものをいくつか挙げてみよう。まず、真っ先に気づきやすいのはデバイスドライバがリストにないことだ。マイクロカーネルアーキテクチャでは、デバイスドライバはユーザー空間で実現される。最近の Linux で提供されている、User space I/O にあたるものがマイクロカーネルからデバイスドライバに提供され、デバイスドライバはユーザー空間にいながら特定の許可されたメモリ空間や IO ポートをたたくことでデバイス管理を実現する。これにより、マイクロカーネルではデバイスドライバがクラッシュしたとして、ほかのユーザー空間プロセスに影響を与えることはないわけだ。ファイルシステムもデバイスドライバと似たような方法でユーザー空間に移されている。

欠けているものその 2 として挙げたいのは、プロセス管理機構。一見すると、あれ、それは (1) で実現されているんじゃ？と思うかもね。でも実はそういうわけじゃない。L4 でいうスレッドとは、レジスタの値とメモリ空間の組を保存したもの、くらいの意味合いしかない。これもそれだけ聞くと普通そんなものなんじゃ？と思ってしまうだろうから、モノリシックカーネルではプロセス管理のためにもっとどんな機能があるか思い出そう。

分かりやすいのはプロセスの起動だろう。execve システムコールは、POSIX OS でプロセスを起動するためのシステムコールだとざっくり言ってしまってもいいと思うが、（おっと、あなたが顔をしかめるのが目に浮かぶようだ！でも fork や exec の話は今は本質じゃないから先にすすませてくれ。）このシステムコールの第一引数は起動したい実行ファイルのパスだ。例えば macOS では、ここに Mach-O フォーマットのファイルへのパスが渡される。すると、カーネルは、あたらしいスレッドプリミティブを作成し、Mach-O をパースしてそのスレッドのメモリ空間に展開してくれるわけだ。一方、L4 のシステムコールではそこまでやってくれない。そもそもファイルシステムはカーネルにないし、ファイルシステムを用意してあげたとして、Mach-O をパースするのはユーザー空間の役割だ。L4 のカーネルがやってくれるのは、単に空のメモリ空間と空のレジスタを持ったスケジュール可能なスレッドプリミティブを作ることだけ。それ以外のすべてはユーザー空間に実装されたサーバーの役割だ。例えばプロセスが開いているファイルハンドルの管理、プロセスの実行ユーザーや実行グループの管理といった機能もすべて同様に L4 カー

ネルには欠けている。L4 のスレッドプリミティブを呼ぶだけでは、いわゆるプロセスらしいことは何も実現できないってことだ。

このように色々な機能がユーザー空間のサーバーの役割とされて、カーネルは最小の機能だけを実現することになっているのが、「マイクロカーネル」と呼ばれる理由だろう。これにより、カーネルのサイズはとても小さいものになる。それに、それぞれのサーバーはメモリ空間が分離されているから、デバイスドライバについての説明で話したように、一つ一つがクラッシュしても、その影響を最小限に抑えることができ、堅牢性も向上する。

OS パーソナリティ

じゃあ、そんな最小限なカーネルを使ってしまったら、いったいどうやって全体として Linux みたいなオペレーティングシステムを実現するのかって？一言でいえば、そういうマイクロカーネルに足りない部分を管理するサーバーを一つ作る、というのが答えだ。普通のカーネルに必要なもろもろの機能を実現するこのサーバーは、OS パーソナリティと呼ばれることがある。OS パーソナリティサーバーが、ファイルシステムやデバイスドライバを IPC を通じて制御し、それをもとに POSIX の fork や read, write のようなユーザーアプリケーションから見えるシステムコールを実装して一つのオペレーティングシステムを実現する、というのがマイクロカーネルベースのオペレーティングシステムの一つの形だ。

マイクロカーネルの面白いところは、このパーソナリティサーバーを取り換えることで同じマイクロカーネルでも違うオペレーティングシステムを実現できることだ。L4 の上で Linux パーソナリティを走らせれば、Linux のアプリケーションが動くし、(仮にあるとすればだが)macOS パーソナリティを走らせれば macOS のアプリケーションが動く。おまけにパーソナリティはユーザー空間のサーバーだから、同時にいくつも並行して走らせることができる。[*7]マイクロカーネルとパーソナリティの試みは、面白い具体例が色々ある。Fiasco.OC L4 カーネル上で Linux が動く L4 Linux[2]。これは今も開発が活発に続いている。何しろ Fiasco.OC は商用 L4 マイクロカーネルだ。過去の例でいえば、Minix が L4Ka 上で動く Minix/L4[3]、Mach カーネル上で Linux が動く MkLinux[4]、それに同じく Mach 上で動く OS である GNU Hurd[7]. 他にも色々あるから、探してみるのもい

7 まるでハイパーバイザみたいな話だが、実際マイクロカーネルは現代の仮想化機構と共通点が多い。

いんじゃないだろうか。

この面白い特徴から、一度あるマイクロカーネルの互換カーネルを作ってしまえば、その上でそのマイクロカーネル用の OS パーソナリティやサーバー群が動くことになる！これが Introduction で説明した自作カーネルを既存のマイクロカーネルの仕様に準拠させるモチベーションの一つだ。

2.2. L4 Microkernel

さて、これでマイクロカーネルの一般的な概念の紹介はおしまいだ。まぁ覚えていてほしいことは、マイクロカーネルは小さな機能だけで構成されていること。それに、通常のオペレーティングシステムとしての機能は、マイクロカーネルの上で動く OS パーソナリティで実現されることだ。この節からは、L4 マイクロカーネルについての話を始めよう。

L4 は、先ほど説明したマイクロカーネルアーキテクチャに基づくマイクロカーネルの一つだ。L4 と呼ばれるマイクロカーネル実装は複数あるから、きちんと呼ぶなら「L4 マイクロカーネルファミリー」になる。このファミリーというのは、だいたい「Windows ファミリー」と同じくらいの感覚だ。Windows には 95 系と NT 系があるわけだけど、どちらも Windows カーネルと呼ばれるし、それぞれでもバージョン毎にだいぶ違いがある。つまり、ファミリーとは一つの実装を指しているわけではなく、共通の系譜を継いだお仲間たちだというわけだね。そして、L4 マイクロカーネルファミリーの系譜の起源は、Jochen Liedtke 先生による 1993 年の、ずばり「L4」だ。[8] [*8] L4 を起源とする L4 ファミリーの系譜の図を Wikipedia から図 1 に引用した。論文"From L3 to seL4 - What Have We Learnt in 20 Years of L4 Microkernels?"[14] は、L4 の重要な概念がよくまとまっているうえに、この系譜図上の L4 から seL4 にいたるまでに L4 ファミリーが解決してきた問題や、設計の変遷をまとめたとても良い読み物だ。L4 についてより詳しい資料を読みたくなったら初めにこれを参照してほしい。(http://sigops.org/s/conferences/sosp/2013/papers/p133-elphinstone.pdf) 他の L4 の重要な情報源といえばずばり公式サイトである http://l4hq.org だ。各種 L4 のドキュメントや仕様がまとめられているので、こちらも初めに参照してほしい。つまり、両方とも外せない情報源なのでぜひ目を通してくれ。

8 L3 という L4 の前身のカーネルもある。そして正確にはそちらが起源なのだが、説明を簡単にするため L4 から始めたい。

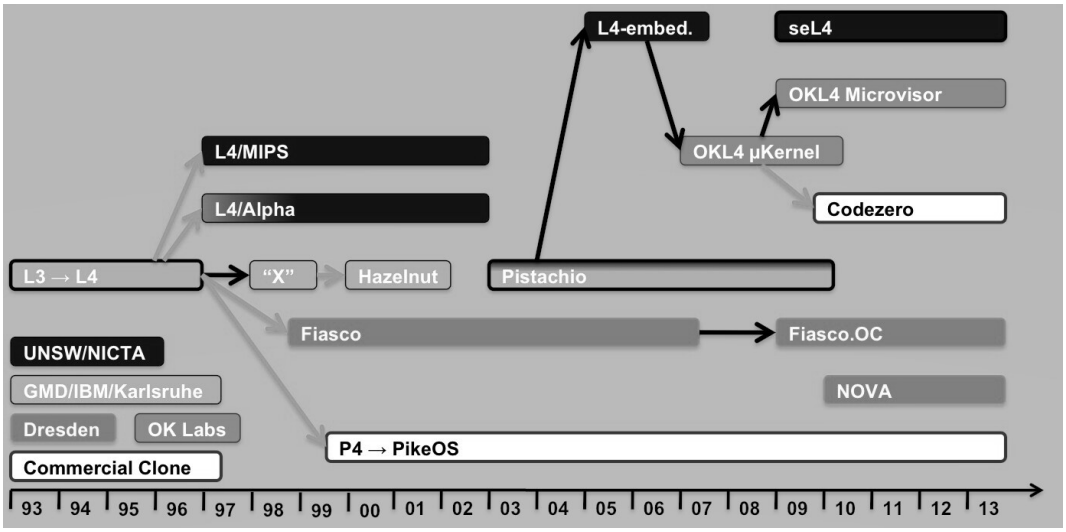


図 1　Family tree of the L4 microkernel with time line([9] より引用)

マイクロカーネルとパフォーマンス - L4 の革命

L4 の開発がはじめられたモチベーションは、マイクロカーネルが抱えていた大きな問題の解決にあった。その問題とは、パフォーマンスの悪さだ。マイクロカーネルは、その設計上 IPC がボトルネックになりやすいということがよく議論にあがる。[14] マイクロカーネルでは、システムコールがサーバー間の複数の IPC で実装され、そして IPC はコンテキストスイッチを伴う。一方、モノリシックカーネルのシステムコールは通常 1 回のコンテキストスイッチで済むから、比較するとマイクロカーネルではシステムコールのコストが割高になってしまう。

L4 は、Liedtke がこの問題を解決するために開発した、画期的なマイクロカーネルだった。先ほど名前だけ少し出た、Mach[10] というマイクロカーネルでは、IPC のコストが顕著だった。[*9]この Mach は L4 の文脈では、「第 1 世代マイクロカーネル」として言及される。もちろん、L4 が第 2 世代だという流れでだけどね ;) Liedtke は、L4 で極小性をキーコンセプトとしてマイクロカーネルをデザインしなおした。IPC の機構は Mach に比べて

9　Mach は Minix と並んでもっとも著名なマイクロカーネルの一つだろう。現在の macOS の XNU カーネルの源流でもある。著名であるにもかかわらず、不名誉な文脈でしか触れられないことに不満を持つファンもいるだろうから、ここでその栄誉を称えておく！

極端に単純になり、メッセージサイズも小さくなり CPU のキャッシュになるべくすべてのメッセージが載るように設計した。スケジューリングなども単純化され、さらにはカーネルすべてをアセンブリで書いてしまうというものすごい執念でありうる極小のマイクロカーネルを完成させた。結果はどうなったと思う？なんと IPC は Mach に比べて 20 倍も高速化したのだ！ [9] ちなみに、のちに L4 カーネルは C や C++ で書き直されたが、パフォーマンスは悪化しなかったので安心してほしい。さらに後続の研究では、Linux パーソナリティが L4 上で実装され、本物の Linux カーネルに比べても数パーセントのオーバーヘッドしか発生しないことが示された。[14] こうして L4 により、マイクロカーネルは実用的なオーバーヘッドで実現できるということが示されたのだ。

マイクロカーネルの逆襲

L4 の登場以降、マイクロカーネルは様々な方向に発展している。MINIX のタネンバウム教授が 2016 年に"Lessons learned from 30 years of Minix"で述べているように、L4 ベースのマイクロカーネルはほとんどあらゆる iPhone のセキュリティチップで動いているし、QNX は色々な組み込み機器で実用化されている。こいういう機器では、少々の性能のオーバーヘッドよりも信頼性が重要だからだ。[15] そのタネンバウム先生の MINIX 自身も、実は世界中の Intel x86 CPU で動いていたという事実が最近判明した。[12]

マイクロカーネルの発展は実用化方面だけではない。一番クールなプロダクトはやはり seL4 だろう。[5] seL4 は、マイクロカーネルのコードサイズが小さいことを利用して、その正当性を Isabella で証明したおそらく世界で唯一のカーネルだ。メモリ安全性といった一般的な性質や、ポートベースのポリシーシステムが決して違反されないことなどを数学的に証明している。[6] しかも seL4 はその仕様設計自体もかなりシンプルで良い感じだ。実際、今回実装する仕様の対象として seL4 はかなり有力な候補だった。しかし、仕様の大きさの問題でこの記事では seL4 を選ばなかった。また、seL4 の仕様とそれ以前の L4 ファミリーの共通点はほとんどないことも注意事項かもしれない。思想上の家族、といったところなのかもね。

普段 Linux の世界で生きていると、1992 年、つまり L4 発表の前年のリーナスとタネンバウム先生 [11] の論争からマイクロカーネルの時は止まっているかのように思えてしまうものだ。しかし L4 以降、マイクロカーネルのパフォーマンスは大幅に改善した。実用化という面では、QNX, MINIX, L4 の数を足してみれば、もしかすると実は Linux よりも

多くの数のマイクロカーネルが世界では動いている可能性すらある。証明済みカーネルというユニークな成果も、マイクロカーネルでなければ不可能だったものだ。あまり大きくは知られていないけれど、マイクロカーネルの世界はどんどん発展していて、いつの間にか「逆襲」を果たしていたというわけかもね。

2.3. L4 X.2 Standard

L4Ka::Pistachio

さて、前の節の話でなんとなく分かるように、L4 と一言で言ってみてもまぁ色々な実装があるわけだ。じゃあ今回何を作ろうかと考えて決めた結論は、図 1 のど真ん中あたりに陣取っている `Pistachio` だ！ Pistachio を選んだ理由は、一言でいって、短期間で自作するのにぴったりだから、だ。系譜図で紹介したように、L4 にはたくさん実装があるのだけれど、そのうちいくつかの実装は、「L4 マイクロカーネルの標準仕様」に準拠して実装されている。マイクロカーネルの仕様、といわれてもピンとこないかもしれないが、要は POSIX と似たようなもので、カーネルが実装しているシステムコール、そのシステムコールの呼び出し ABI などが定義されている。割り込みが起きたときにどのようなプロトコルで処理が走るべきか、なんかも定義されているのは少しカーネルらしいところかもしれない。そして、Pistachio は、きちんと定義された最後の仕様といっていい、L4 X.2 Standard というものに準拠している実装なのだ。L4 X.2 Standard はシステムコールが 12 個しかないというとても小さい仕様になっていて、準拠したマイクロカーネルを実装しやすい。ちなみにこの記事が 12 日でカーネルを作る、と言い張っているのはこの API が 12 個しかない、ということを根拠にしている。[*10] X.2 以降のおおよそ似たような L4 ファミリーといえば OKL4 と Fiasco.OC になるのだが、これらはどちらも実用的なマイクロカーネルとなっていて、アカデミアの世界で完結している X.2 に比べ実装されている API がとても多く、互換カーネルを作るのが困難だ。それにそもそもこれらのカーネルは、もはやきちんと定義された仕様が存在しない。一方 seL4 は、実用性も兼ねているにもかかわらずきちんと定義された仕様が存在するのだが、やはり X.2 に比べるとだいぶ大きな仕様になってしまっている。そんなわけで、最後の実質的な更新は 2010 年頃と少し古いけれど、互換カーネルを自作するのにピッタリの対象としては、X.2、それに Pistachio が選ばれたというわけ。

---

10 まだ実装はさっぱり終わってないから、正直これは私の希望でしかないんだけどね！

L4 X.2 Standard

さて、L4 の X.2 Standard について簡単にだけ触れておこう。 そもそも L4 の仕様というときは、だいたい API と ABI の仕様を決めていると思ってくれていい。L4 X.2 では 12 個のシステムコール (API) と、それらを呼び出す際の ABI が決められている。そのシステムコールとは、次のものだ。

(1) `KERNELINTERFACE`

(2) `EXCHANGEREGISTERS`

(3) `THREADCONTROL`

(4) `SYSTEMCLOCK`

(5) `THREADSWITCH`

(6) `SCHEDULE`

(7) `IPC`

(8) `LIPC`

(9) `UNMAP`

(10) `SPACECONTROL`

(11) `PROCESSORCONTROL`

(12) `MEMORYCONTROL`

冗談抜きでこれで全部だ。 実にシンプルで小さい仕様だ。 このシステムコールさえ実装すれば、おおよそ Pistachio と互換のカーネルができあがることになる。 ファイルフォーマットが ELF であること、みたいな仕様の範疇を越えることもあるから完全に互換ではないが、 気を付ければ同じユーザーアプリケーションを動かすことは実現できる。 それぞれのシステムコールが何をするものなのかはおおよそ名前から推測できると思う。 これらの詳しい仕様の資料は、http://www.l4ka.org/l4ka/l4-x2-r7.pdf から入手できる。

2.4. Let's boot L4 Pistachio in Qemu

Pistachio のビルド

少しおしゃべりが過ぎたようだ。 そろそろ作業の方に入った方がいいだろうね！あら

ゆることはまず `git clone` から始めよと昔から言われているように、まずは Pistachio を手元に clone しよう。公式レポジトリは `https://github.com/l4ka/pistachio/commits/master` だから、早速ここから clone してくれ・・・と言いたいところなのだけど、実は私の環境だとこのレポジトリのビルドが失敗した。私の環境は Ubuntu 18.04 で、普通にインストールして入ってくる gcc のメジャーバージョンは 7 だ。gcc は 5 以降あたりから、Position Independent Code をデフォルトでビルドするようにオプションが変更されていて、私はそれでなかなか Pistachio のビルドが通らなくて痛い目を見た。ちょっと古めのソースコードをビルドしてみようとすると大抵こうなるものだ。*sigh* 自分が経験した痛い目を、わざわざ文章で再現してもう一度痛みを確かめるような趣味は私にはないから、gcc 7 でもビルドが通るように私が修正したものを使うように歴史を修正しながら作業過程を書いていくことにする！ [*11]

```
$ git clone https://github.com/nullpo-head/pistachio.git
```

これで、私がした苦労をスキップして、Ubuntu 18.04 上でビルド可能な Pistachio が手に入る。どう Makefile を変更すれば良いのかが気になる人は、このレポジトリの最新 2 コミットを見てほしい。

それではビルドを始めよう。Pistachio に限らない話なのだが、L4 は資料があるのか無いのか微妙なラインを攻めてくる。Pistachio のまともに完遂できるビルド方法に関しても、英語でも日本語でも良いものがなかなか見つからなかったのだが、幸運なことに n_kane さんの記事 (http://d.hatena.ne.jp/kayn_koen/20100801/1280622473) がとても参考になる。これをもとにビルドを進めていこう。

まずはカーネルのビルドからだ。カーネル用のビルドディレクトリを作りつつ、カーネルをビルドしよう。

```
$ cd pistachio/kernel
$ make BUILDDIR="$(pwd)/../build-kernel"
```

11 ちなみにビルドが通るように修正したあとに、apt から最新の gcc とは別に gcc5 をインストールできることを知った。もしかしたらこれを使えば元のレポジトリでもビルドが通るのかもしれない。

n_kane さんいわく、BUILDDIR は絶対パスである必要があるとのことだ。

```
$ cd ../build-kernel
$ make menuconfig
```

ここで Linux や busybox のビルドでおなじみの menuconfig が表示される。お好みで設定を変えていくわけだけど、ここでは 64bit カーネルをビルドするよう設定する。もう平成も終わって令和になるわけだから、今から作るカーネルはもちろん 64bit カーネルだからね。

```
$ make -j4
```

そしてビルドする。`x86-kernel` バイナリができあがった。

次はユーザーランドのビルドだ。同じように進めていく。ただし今度は autoheader と autoconf を動かす必要がある。どうしてそんな構成になっているのかは分からないけど、まぁとにかく従ってみよう。

```
$ cd ../user
$ autoheader
$ autoconf
```

さらに今度はビルドディレクトリも自分で用意して、その中でビルドを進める。

```
$ cd ..
$ mkdir build-user
$ cd build-user
$ ../user/configure --prefix="$(pwd)/../install-user" --with-kerneldir="../b
uild-kernel/"
$ make -j4
$ make install
$ cp ../build-kernel/x86-kernel ../install-user/libexec/l4/
```

カーネルに比べて少し複雑だけど、これでビルドが完了して、install-user ディレクトリの中にもろもろの成果物が出力される。大事なものはだいたい l4 ディレクトリ以下にまとまるようだ。確認してみよう。

```
$ cd install-user/libexec/l4/
$ ls
grabmem*  kickstart*  l4test*  pingpong*  sigma0*  x86-kernel*
```

That's it! できた成果物を、それぞれ少し説明しよう。

- kickstart これは L4 のためのブートローダのようなものだ。GRUB などのブートローダは一旦この kickstart を起動し、その kickstart が x86-kernel を起動する。どうしてそんなことになっているかはあとで multiboot と一緒に説明しよう。
- x86-kernel これが Pistachio マイクロカーネルの本体だ。
- sigma0 これはルートページャと呼ばれる特殊なサーバーになる。
- pingpong これはユーザー空間のサーバーで、Pistachio 上で動くユーザーアプリケーションだと言ってもいい。ただし、sigma0 の次に動かすことを想定しているから、OS パーソナリティサーバーなしに動くユーザーアプリケーションだね。といっても、L4 は組み込みの世界でよく使われるから、汎用の OS パーソナリティなしにアプリケーションが動く状況は珍しくないはずだ。pingpong は、いくつかのプロセスを立ち上げ、互いに IPC を送り合うことで、マイクロカーネルの性能を測定する動作をする。
- l4test こちらもユーザーアプリケーションだ。pingpong が IPC をするものだったのに対し、l4test は L4 の API に対する広範なテストを動かすものだ。
- grabmem 気にしなくていい。

というわけで、これらが Pistachio レポジトリをビルドして得られたものだ。せっかくビルドしたんだから、じっと睨んでいても仕方がない。Qemu 上で起動してみよう。

Multiboot を使って Qemu でブートさせる

さっきビルドしたバイナリ群は、すべて"Multiboot"という仕様に準拠したバイナリになっている。読者の皆さまの中で、UNIX 系の自作 OS に挑戦したことがある人は知っているかもしれないね。CPU やプラットフォームによって、OS のブートプロセスというのは異なるものだ。だから、OS が色んなプラットフォームに対応しようと思えば、すべてのプラットフォーム上でブートローダーを書く必要がある。逆に OS 自体も色々なものがあ

るわけで、あるブートローダーが色々な OS をブートできるようにしようと思うと、OS の数だけブートローダーから OS に処理を渡すコードを書く必要がある。 つまり、 色々な OS が色々なプラットフォームで動くための色々なブートローダーが乱造されることになるわけだ。 これはとても非効率だし、 開発者もできればそんなことはやりたくない。 この問題を解決するために提案されたのが Multiboot 仕様だ。[13] これに準拠さえしていれば、この面倒くさい問題を回避できるようになる。あらゆる Multiboot 準拠のブートローダーは、 あらゆる Multiboot 準拠のオペレーティングシステムを起動できるのだ。 しかも Multiboot 仕様では、ブートローダーが CPU の初期化の面倒くさいところをやってくれることになっているので、Multiboot 準拠のカーネルを書けば x86 プロセッサの面倒くさいブートプロセスなんかをスキップすることができる。私がカーネルを書くときは、必ず Multiboot 準拠にしてこの恩恵にあずかることにしている。OS 側として Multiboot に準拠するのに必要なことは、Multiboot ヘッダと呼ばれるものをカーネルの先頭にくっつけ、カーネルバイナリのフォーマットを ELF にするだけなので実に簡単だ。

n_kane さんの記事では GRUB を使って Pistachio をブートしていたが、実は Qemu には Multiboot のバージョン 1 仕様のブートローダーが組み込まれている。 だから、Qemu で直接 Pisachio を起動することができる。やっていこう。

```
$ qemu-system-x86_64 -kernel kickstart -initrd x86-kernel,sigma0,pingpong
```

すると図 2 のような画面が表示される。 これは Pistachio がブートしている途中の画面だ！

qemu に渡したコマンドラインオプションを解説すると、`-kernel` がまず Multiboot ブートローダーが最初に起動すべきバイナリを渡すものだ。Pistachio の場合、 ここに一旦 `kickstart` を渡すことになる。qemu にはさらに、`-initrd` オプションで、`x86-kernel,sigma0,pingpong` を渡している。`kickstart` はこれの ELF フォーマットをパースして、x86-kernel に教えてくれる役割を果たしている。 その後、 本体の `x86-kernel` に処理が渡される。 ちなみに豆知識なんだけれど、そもそも Multiboot は Mach や L4 のようなマイクロカーネルを起動するためのブートローダーとしても想定されているらしい。`-initrd` オプションで渡しているものは「ブートモジュール」 と Multiboot では呼ばれる。 マイクロカーネルは普通、カーネル自身にファイルシステムや複雑な ELF ローダーがないことが多いので、ブートローダーがあらかじめ起動に必要なバイナリ群をメモリ上に展開してお

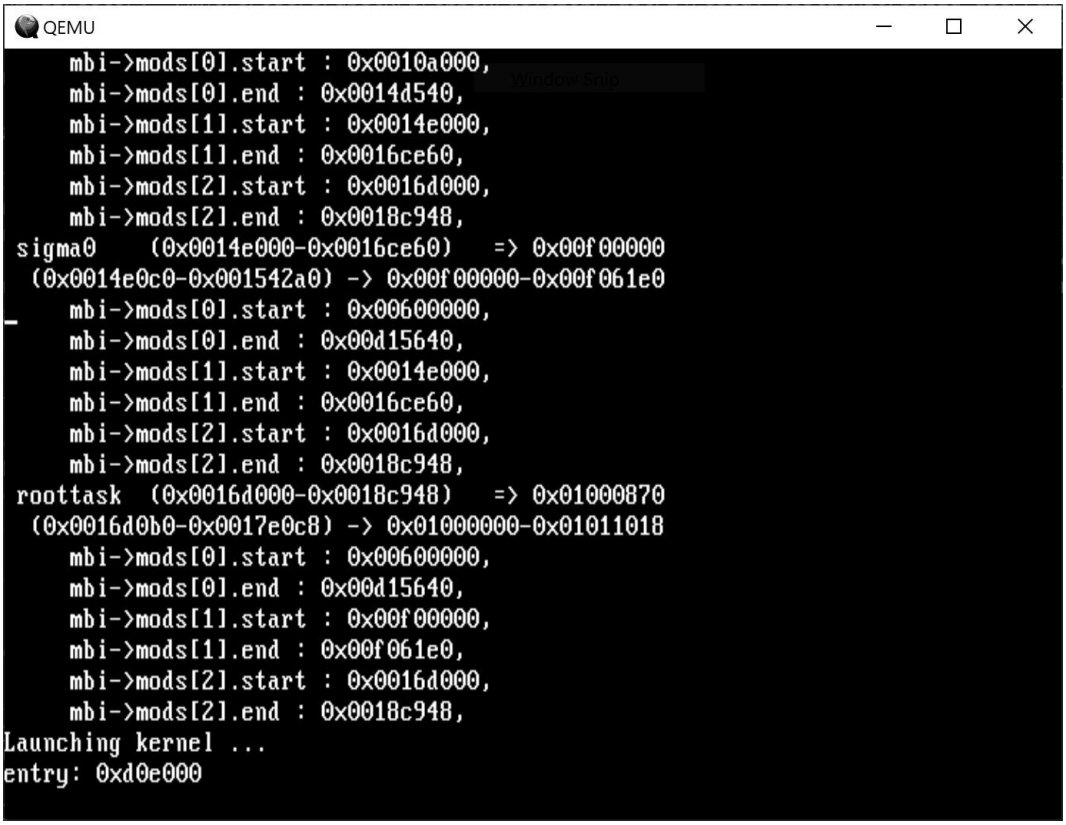


図 2　Pistachio がブートする様子

き、マイクロカーネルに処理を渡す、というのがモチベーションだ。その構造は、しっかり Pistachio でも活かされている。

さて、図 2 は確かにブート途中の画面なんだけれど、実は処理が進んでもここから動かない。今回アプリケーションとして起動している pingpong の出力は、VGA じゃなくてシリアル経由でしか表示されないからだ。オプションを変えて起動しなおしてみる。

```
$ qemu-system-x86_64 -nographic -kernel kickstart -initrd x86-kernel,sigma
0,pingpong
```

すると、下のように pingpong の出力がターミナルに表示された。

```
Launching kernel ...
```

```
entry: 0xd0e000

L4Ka::Pistachio - built on Apr  2 2019 01:51:37 by nullpo@Surface-EPJH0U2 us
ing gcc version 7.3.0
Pingpong started 1

Please select ipc type:

1: INTER-AS
2: INTRA-AS (IPC)
3: INTRA-AS (LIPC)
4: XCPU
a: architecture specific
s: print SQLite table
Benchmarking Intra-AS IPC...
IPC ( 0 MRs): 2131.71 cycles, 0.79us, 0.00 instrs
IPC ( 1 MRs): 1720.19 cycles, 0.68us, 0.00 instrs
...
```

これで無事、ビルドした Pistachio が起動したことになる！そしてユーザー空間のアプリケーションとして pingpong を動かしてみた。もっと派手な見た目が動く Genode みたいなものもあるんだけど、説明がややこしくなるから標準バンドルの pingpong だけに話をしぼらせてもらった。時間があれば、xv6 を Pistachio 上に移植したり、Minix OS パーソナリティを L4 上に実装したりして、マイクロカーネルの実装が終わったときにかっこいい見た目の OS が動くようにしたいものだ。でもまずはこの地味な pingpong が第 1 歩！これが偉大な一歩かは知らないが、少なくとも結構頑張った 1 日目にはなったんじゃないかな。

3. 2nd Day: Writing a Minimal Kernel in Rust

昨日の 1 日目では、マイクロカーネルの概念と L4 マイクロカーネルの話を学び、実際に Pistachio カーネルを Qemu 上で動作させてみた。今日からは、あり物のカーネルを動かすのではなく、自分の手で L4 マイクロカーネルの実装を始めることにする。その第一歩として今日は、Rust でベアメタルプログラミングに挑戦し、自作の最小のカーネルを Qemu 上で動かしてみよう。今日の目的は、一旦マイクロカーネルのことは忘れ、Rust

でのベアメタル開発の世界に飛び込んでみることだ。以降の章で、今日作るカーネルをもとにして少しずつマイクロカーネルを作りあげていくことになる。さて、ふだんは Web プログラムなどの開発を行っていて、低レイヤでのプログラミングにはなじみがないが、ここまで読み進めた方もいないとは限らない。この記事では基本的な低レイヤプログラミングの基礎知識を 1 ステップずつ網羅していくようなことは、できない。[*12]しかし、実際に手を動かして試してみて、低レイヤプログラミングの楽しさを知ってもらえたとしたらこれほど嬉しいことはない。

一方、ここまで読んでいるような読者の方は、おおよそ、この手のプログラミングにはもう慣れっこだろう。私自身、こういうプログラムを書くのは初めてではない。Xv6 という小さな UNIX ライクなオペレーティングシステムを MIPS アーキテクチャに移植したことがあるし、それを自分たちで設計・FPGA 上に実装したオリジナル CPU に移植したこともある。また、CPU の仮想化支援機構を利用して既存のオペレーティングシステムに Linux パーソナリティを実現する Noah というソフトウェアを作ったことがある。このマイクロカーネルと似たような話の開発では、かなりの時間を仮想環境内のベアメタルプログラミングに費やした。だけどこのような低レイヤプログラミングに慣れっこだというのは、あくまで「C 言語を使う限りにおいては」だ。だから、Rust でカーネルを書くと何が楽になり、何が独特の難しさかといった点が重要なポイントだろう。C 言語でならこの手のプログラミングに慣れているという方は、いつもの慣れた手順が Rust の世界ではどのように様変わりするのかということを楽しんでもらえると嬉しい。

3.1. Bare Metal Programming

さて、いよいよ Rust でのベアメタルプログラミングの世界に飛び込んでみる時がきた。実は Rust の世界ではすでに、"Writing an OS in Rust" (https://os.phil-opp.com/) という素晴らしい Web シリーズが存在する。このシリーズは、Rust でベアメタルプログラミングをするための準備から、x86 アーキテクチャの基礎知識までカバーしている素晴らしいシリーズだ。Rust でオペレーティングシステムを書くなら、これをチェックしない手はない。私も今日 Rust で小さなカーネルを書いてみるにあたって、このシリーズのお世話になることにした。実のところ、私がこの記事を、教科書のようなスタイルではなく私の個人的なカーネル開発のログを時系列で綴っていくスタイルにしようと思ったのも

12 締切の都合で

このシリーズの存在が理由だ。すでに素晴らしい教科書が存在するというのに、そこに不完全なものを新たに一つ加えたところで何になるだろう！(ただし"Writing an OS in Rust"は2019年1月時点ではまだ未完結で、ページングに関する解説が執筆されているところであることには注意してほしい。だが、活発に執筆が続けられているので、すぐに完結することだろう。)さて、早速"Writing an OS in Rust"の第1章、"A Freestanding Rust Binary"に従って、Rustでのベアメタルプログラミングに入門してみることにしよう。"Writing an OS in Rust"は主にPhilipp Oppermann氏によって執筆されている。これから毎度毎度"Writing an OS in Rust"とフルタイトルと呼ぶのは、書くのも読むのも大変だから、"Philipp's"と呼んでいくことにしよう。

まずは、Rustのツールチェインの準備からだ。Philipp'sで説明されているところによると、Rustでオペレーティングシステムを作るにあたっては、nightlyコンパイラがまだ必要らしい。私はRustに触れたのが比較的最近なので、Rustツールチェインのインストールにはきちんと Rustupツールを使っている。その場合は下記のコマンドでnightlyコンパイラを簡単にインストールすることができる。もし、Rustに触れてみたのがかなり昔で、Rustツールチェインのインストールに Rustupを使わなかったという読者の方は、Rustupを使ってツールチェインをインストールしなおすのがいいだろう。

```
$ rustup install nightly
```

これでRustツールチェインの準備は完了だ。シンプル！さて、ここでこれから、自分がカーネルを書いていくことになるRustプロジェクトを初期化しておく。これはこのプロジェクトに名前を付けるということでもある。これから作っていくカーネルは、最終的にはL4マイクロカーネルファミリーのマイクロカーネルになる。Rust製のL4カーネル実装はたぶん私のものが初めてになるだろうから、それにふさわしい名前として、私は自分のカーネルを `Rusty L4` と呼ぶことにした。だから、ここで私が走らせるコマンドはこうだ。

```
$ cargo new rusty_l4 --bin --edition 2018
```

cargo コマンドの使い方は覚えているだろうか？念のためおさらいしておくと cargo コマンドは Rust 標準のビルドツールだ。Rust のプロジェクトは基本的に cargo コマンドを使って管理していく。`--edition 2018` は見慣れないオプションだ。 これは 2018 年ギリギリにリリースされたばかりの Rust 2018 の機能を有効にするためのオプションである。Rust 2018 は多くの便利な新機能が導入されたエキサイティングなバージョンだ。特にこだわりがなければこれを使わない手はない。

さらに、Philipp's にしたがって、Rusty L4 のプロジェクトでは nightly コンパイラをデフォルトで使用するように設定する。

```
$ rustup override add nightly
```

no_std

C 言語では、以下のコードに対して、

```
int _start() {
  for(;;);
}
```

```
gcc loop.c -nostdlib
```

のようなコマンドをたたくと、OS に依存しない、起動後ひたすら無限ループをする ELF 実行ファイルを作成することができる。ELF が正しくロードされる環境なら、大抵の場所で動くだろう。`_start` は、ELF 実行ファイル、つまり Linux での標準的な実行ファイルが OS から初めて起動されるときに読み込まれるエントリーポイントだ。通常の C プログラムでは、_start は標準ライブラリによって実装されていて、初期化処理を経た後 `main` が呼ばれることになる。こういうレイヤの低いことを簡単にできるのが C の強みだ。

これを Rust で実現するにはどうすればいいだろうか？ Philipp's によれば、`#![no_std]` アトリビュートの出番だ。 これは、C 言語での `-nostdlib` オプションに相当する概念だ。ただし、単にこのアトリビュートをファイルの先頭に書くだけではうまくいかない。

Philipp's に従って開発を進めると、 先ほどの C プログラムにあたるコードは次のようになる。

```
// main.rs

#![no_std] // don't link the Rust standard library
#![no_main] // disable all Rust-level entry points

use core::panic::PanicInfo;

#[no_mangle] // don't mangle the name of this function
pub extern "C" fn _start() -> ! {
    loop {}
}

/// This function is called on panic.
#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
  loop {}
}
```

そして、`Cargo.toml` にはこう記述しよう。

```
[package]
name = "rusty_l4"
version = "0.1.0"
authors = ["FOO BAR <foo@bar.com>"]
edition = "2018"

[profile.dev]
panic = "abort"

[profile.release]
panic = "abort"
```

この状態から、Linux 上では以下のコマンドでビルドが成功する。

```
$ cargo rustc -- -C link-arg=-nostartfiles
```

macOS 上では別のコマンドが必要なので Philipp's を参照してほしい。もっとも macOS 上では static バイナリの実現は難しいのだけれど。C 言語に比べて増えたことは、だいたい次の二つだった。

- panic の処理を自分で書いたこと、そして panic のための stack unwinding を無効にしたこと。Rust では、ベアメタル環境でも配列の境界チェックなどで panic が発生する。そのためのハンドラを与えてやる必要がある。それに C++ の例外のように stack unwinding が存在するから、それを無効化する必要がある。でもこれは、toml に panic の設定を書くだけで無効化ができるんだから、かなり簡単な方だと思う。
- `#[no_mangle]` を与える必要があること。これまた C++ と同じで、Rust でも関数名がバイナリレベルではソースコード上と一致しない。同じ名前のシンボルでも、名前空間が違えば区別する必要があるから、その手の情報を名前に埋め込むのだ。これを mangling と呼ぶ。`#[no_mangle]` で `_start` に対してマングリングを無効化している。

3.2. Minimal Rust Kernel

ここまでで、Rust でベアメタルプログラミングに挑戦してみて、C 言語との違いについて述べてきた。いまのところ、いくつか違いはあるけれど、C でもできたことがよりイマドキな書き方でできるんだな、というくらいの感想を持っているかもしれない。むしろ、色々とベアメタルでは使えなくなってしまう言語機能の穴埋めに奔走させられた気すらしているかも。しかし、イマドキの言語でベアメタルプログラミングをする便利さを実感させられたのはここからだった。急に開発スピードが加速していくから注意してついてきてほしい。

ビルド設定

ただの no_std プログラムをビルドするのではなく、今からはオペレーティングシステムがなくても動くようなプログラムをビルドすることになる。そのために、Target Specification というものを書く必要がある。これは json ファイル形式のファイルで、cargo のビルドを細かく制御することができる。

```
{
  "llvm-target": "x86_64-unknown-none",
```

```
  "data-layout": "e-m:e-i64:64-f80:128-n8:16:32:64-S128",
  "arch": "x86_64",
  "target-endian": "little",
  "target-pointer-width": "64",
  "target-c-int-width": "32",
  "os": "none",
  "executables": true,
  "linker-flavor": "ld.lld",
  "linker": "rust-lld",
  "panic-strategy": "abort",
  "disable-redzone": true,
  "features": "-mmx,-sse,+soft-float"
}
```

これを `x86_64-rusty_l4.json` という名前で保存した。各フィールドの意味はなんとなく類推できるんじゃないだろうか。注目すべきは os が none になっていること。あと、若干脇にそれるけれど、`"disabel-redzone": true,` を指定していること。redzone は x64 で初めて登場した、32 bit 時代にはなかった概念だ。コンパイラの最適化のために、一定の大きさのスタックポインタの先をふれてもいいという規約のことを指す。これは割り込みハンドラの処理と相性が悪いから、カーネルのコード中ではこの redzone を無効化する必要がある。

cargo xbuild によるクロスコンパイル

さて、次に、このコマンドを走らせる。

```
$ cargo install cargo-xbuild
```

これは Rust でカーネルを書き始めての初めての cargo install だ。パッケージのエコシステムが整っているということは、必ずしも Rust の特徴だというわけではない。Ruby みたいな LL 言語しかし、Go のようなコンパイル言語しかり、最近の言語は程度の差こそあれパッケージの管理機構を言語ランタイム自体と結び付けて提供している。しかし、カーネル開発だという文脈だと、このエコシステムは間違いなく Rust の強みだ。C や C++ 言語ではこのエコシステムはきちんと育っていない。私は、時間の（特に締め切りの）都合で Rust の crate をどんどん使用していったのだが、結果としてカーネル開発がいかに

楽になるかということが体感できた気がする。

さて、cargo-xbuild は、Rust の標準ライブラリのサブセットである core ライブラリや compiler_builtins を Cargo で指定したターゲット向けにクロスコンパイルしてくれる crate だ。core ライブラリと compiler_builtins ライブラリには、Rust の標準ライブラリのうち、ベアメタル環境でも使える機能が含まれている。cargo-xbuild を使わない場合、自分でこれらのライブラリをクロスコンパイルしないと、Rust らしい色々な機能が使えなくなってしまう。これがないと、カーネル開発が非常に不便になるというくらい、必要性が高い crate だ。 そしてこの crate は Philipp 氏自身によって開発されている。cargo-xbuild 以外に多くの crate が出てくることになるが、そのうち大半の crate が Philipp 氏によるものであったりする。カーネル開発を便利にしてくれる crate を整備していった上で、ブログまで書いてるんだから、まるでブルドーザーみたいな人だ。cargo-xbuild を用いてビルドをするためには、次のコマンドでビルドを行おう。

```
$ cargo xbuild --target x86_64-rusty_l4.json
```

VGA への表示と Bootimage

Philipp's に従って、次は VGA を使って画面に文字を書く機能を main.rs に加えてみよう。_start 周辺を、下のコードのように書き換えよう。

```
static HELLO: &[u8] = b"Hello World!";

#[no_mangle]
pub extern "C" fn _start() -> ! {
    let vga_buffer = 0xb8000 as *mut u8;

    for (i, &byte) in HELLO.iter().enumerate() {
        unsafe {
            *vga_buffer.offset(i as isize * 2) = byte;
            *vga_buffer.offset(i as isize * 2 + 1) = 0xb;
        }
    }

    loop {}
```

```
}
```

今回、コード中に unsafe が出現している。適当な値のポインタに書込みを行うんだから当然だ。これは、VGA のテキストバッファに文字を書き込むことで、Qemu のエミュレータ画面に Hello World を表示させている。VGA の詳しい仕様については Philipp's の解説や Wikipedia を参考にしてほしい。

さて、これをいったいどうやって Qemu 上で起動するのだろう？ただ ELF ファイルを作るだけでは、エミュレータ上で OS としてブートさせることはできない。Pistachio がやっていたように Multiboot に対応するか、もしくは自前でブートローダーを作る必要があるのだ。そこで `bootloader create` と `bootimage` ツールの出番だ。この crate は後者、Phillip 氏によってピュア Rust(とインラインアセンブリ) で書かれたブートローダーで、`bootimage` と組み合わせて使用する。bootloader crate をインストールすると、私のカーネルが bootloader でブートできる形になり、bootimage ツールは cargo のラッパーツールとして働いて、実際にブート可能なディスクイメージを作成してくれる。bootimage をインストールしよう

```
$ cargo install bootimage --verion "^0.5.0"
```

このコマンドはセマンティックバージョニングでの、0.5.x の最新のバージョンの bootimage をインストールする。そして、Cargo.toml に bootloader crate への依存を追加する。

```
[dependencies]
bootloader = "0.4.0"
```

もっと新しいバージョンももう Publish されているのだけど、これで大丈夫だ。ではさっそく、bootimage コマンドを使ってカーネルをビルドしよう。

```
$ bootimage build --target x86_64-rusty_l4.json
```

bootimage は内部で cargo-xbuild を使ってビルドしてくれるから、core ライブラリの心配は引き続きしなくて大丈夫だ。

さて、いったい何が起きたのか確認してみよう。これだけで、ブート可能なディスク

イメージが生成されている。

```
$ cd target/x86_64-rusty_l4/debug/
$ ls
bootimage-rusty_l4.bin  build/  deps/  examples/  incremental/  native/  rus
ty_l4*  rusty_l4.d
$ file bootimage-rusty_l4.bin
bootimage-rusty_l4.bin: DOS/MBR boot sector
```

なんとブータブルイメージが出力されている！この `bootimage-rusty_l4.bin` がお目当てのものだ！ Philipp's でも説明されているように、bootloader + bootimage では、 ブートローダーのバイナリファイルの後ろに、rusty_l4 の ELF バイナリがくっついたものが、スタンドアローンバイナリとして生成される。それが BIOS によって起動され、ブートローダーは rusty_l4 バイナリを適切にメモリ上に配置してブートするようになる。 いくつかのコマンドをたたいて、crate をインストールしただけなのに！起動してみよう。

```
$ qemu-system-x86_64 -drive format=raw,file=bootimage-rusty_l4.bin
```

すると、 図 3 のように、Qemu の画面上に Hello, World! が表示された。 create により、カーネル開発の速度がすごいぐらいに加速されている。OS のブートから VGA 表示までがこんなに早いのは素晴らしいことだ。というか何かプログラムを書いた気がしないくらいだ。

どんどん Philipp's の実装を進めていこう。VGA プリントのモジュールを作ることが次の目標だ。 そのために必要な crate をそろえていく。`volatile` crate と `spin` crate を Cargo.toml に加えよう。

```
[dependencies]
spin = "0.4.9"
volatile = "0.2.3"
```

volatile は、C の volatile を知っている人にとっては話が早いだろう。これはたとえば、

```
struct Buffer {
  chars: [[Volatile<ScreenChar>; BUFFER_WIDTH]; BUFFER_HEIGHT],
```

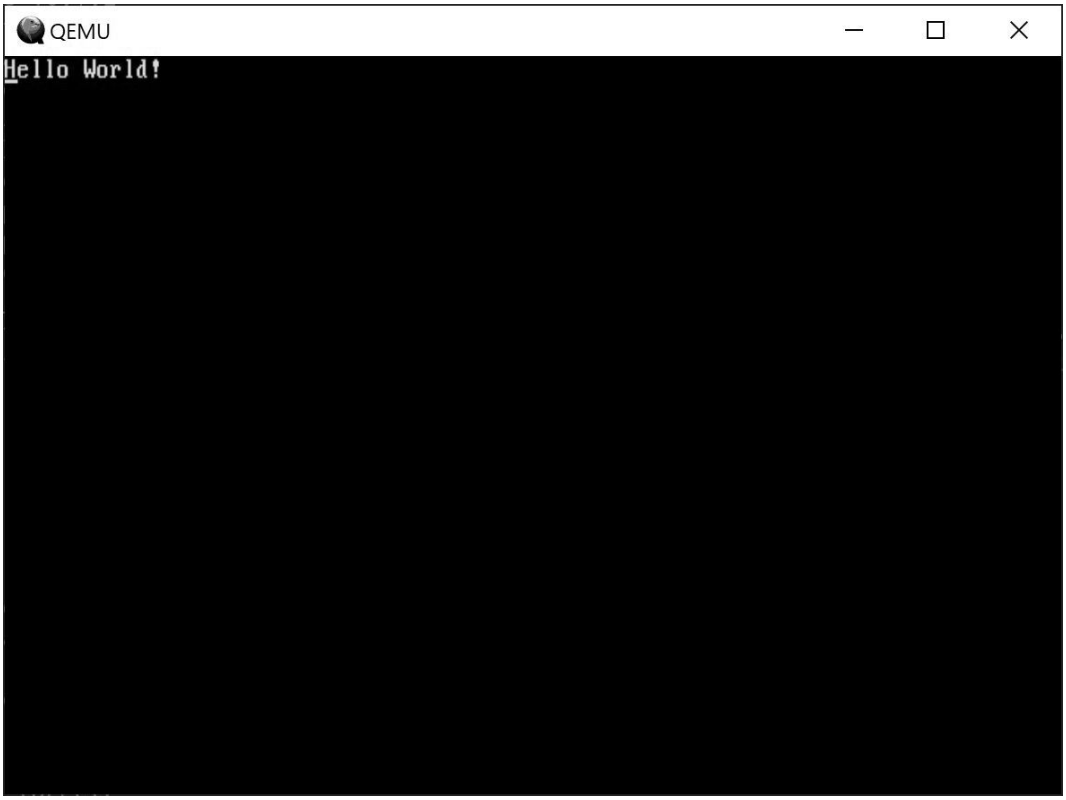


図 3　bootimage + bootloader で、何も書かなくてもカーネルが起動する

```
}
```

のように使用する。カーネル開発では、まさに VGA のような、メモリ上の特別なアドレスへの読み書きを行って何かハードウェアとやりとりすることがある。しかし、これはコンパイラから見れば、意味もなくメモリを読み書きしているようにみえるので、最適化の過程でコードが消されてしまう。それを防ぐために、このアドレスのメモリデータは、その読み書きに外部との相互副作用がありますよ、というのを明示するのが volatile だ。Rust にもこの volatie の仕組みがあり、そしてこの volatile crate は、それを使いやすいラッパーで隠ぺいした Philipp 氏の crate だ。一方の spin は、スピンロックによるロックを提供するための create だ。ベアメタル環境では、標準ライブラリの Mutex などが基本的に使えない。しかし、この spin crate は、スピンロックを同期プリミティブとして使用した Mutex を提供してくれる。ちなみにこちらは Philipp 氏の crate ではない。

次に lazy_static をインストールしよう。これはかなり重要な crate だ。Cargo.toml に次の記述を加える。

```
[dependencies.lazy_static]
version = "1.0"
features = ["spin_no_std"]
```

Rust では、C のようなグローバルにアクセスできるにもかかわらず、mutable であるような変数は基本的に使えない。static スコープで、かつ mutable であるようなものは borrow checker のルールを満たせない。このような変数は、mutable ではないためにしばしば初期化が難しくなる。lazy static は、そんな問題を解決する lazy_static! マクロを提供してくれる。lazy static を使うと、static な変数を、初めてアクセスがあったときに任意の非定数式で初期化できるようになる。IDT や GDT のようなプロセッサ構造体の定義に、lazy static は多用することになる。

さて、準備は完了した。VGA へのプリント機能を提供する、vga_buffer.rs をかきあげよう。

```
#[allow(dead_code)]
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(u8)]
pub enum Color {
    Black = 0,
    Blue = 1,
    Green = 2,
    Cyan = 3,
    Red = 4,
    Magenta = 5,
    Brown = 6,
    LightGray = 7,
    DarkGray = 8,
    LightBlue = 9,
    LightGreen = 10,
    LightCyan = 11,
    LightRed = 12,
    Pink = 13,
    Yellow = 14,
    White = 15,
```

```
}
```

Color enum を定義する。Debug を derive するだけで、のちのちデバッグプリントができるようになる。このまま一気に完成形まで進めてしまおう。

```
use volatile::Volatile;
use core::fmt;
use lazy_static::lazy_static;
use spin::Mutex;

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(transparent)]
struct ColorCode(u8);

impl ColorCode {
    fn new(foreground: Color, background: Color) -> ColorCode {
        ColorCode((background as u8) << 4 | (foreground as u8))
    }
}

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(C)]
struct ScreenChar {
    ascii_character: u8,
    color_code: ColorCode,
}

const BUFFER_HEIGHT: usize = 25;
const BUFFER_WIDTH: usize = 80;

#[repr(transparent)]
struct Buffer {
    chars: [[ScreenChar; BUFFER_WIDTH]; BUFFER_HEIGHT],
}

pub struct Writer {
  column_position: usize,
  color_code: ColorCode,
  buffer: &'static mut Buffer,
}
```

```
lazy_static! {
    pub static ref WRITER: Mutex<Writer> = Mutex::new(Writer {
        column_position: 0,
        color_code: ColorCode::new(Color::Yellow, Color::Black),
        buffer: unsafe { &mut *(0xb8000 as *mut Buffer) },
    });
}

impl Writer {
    pub fn write_string(&mut self, s: &str) {
        for byte in s.bytes() {
            match byte {
                0x20...0x7e | b'\n' => self.write_byte(byte),
                _ => self.write_byte(0xfe),
            }
        }
    }

    pub fn write_byte(&mut self, byte: u8) {
        match byte {
            b'\n' => self.new_line(),
            byte => {
                if self.column_position >= BUFFER_WIDTH {
                    self.new_line();
                }

                let row = BUFFER_HEIGHT - 1;
                let col = self.column_position;

                let color_code = self.color_code;
                self.buffer.chars[row][col].write(ScreenChar {
                    ascii_character: byte,
                    color_code,
                });
                self.column_position += 1;
            }
        }
    }

    fn new_line(&mut self) {
        for row in 1..BUFFER_HEIGHT {
```

```
            for col in 0..BUFFER_WIDTH {
                let character = self.buffer.chars[row][col].read();
                self.buffer.chars[row - 1][col].write(character);
            }
        }
        self.clear_row(BUFFER_HEIGHT - 1);
        self.column_position = 0;
    }

    fn clear_row(&mut self, row: usize) {
        let blank = ScreenChar {
            ascii_character: b' ',
            color_code: self.color_code,
        };
        for col in 0..BUFFER_WIDTH {
            self.buffer.chars[row][col].write(blank);
        }
    }
}

impl fmt::Write for Writer {
    fn write_str(&mut self, s: &str) -> fmt::Result {
        self.write_string(s);
        Ok(())
    }
}
```

これにさらに、`println!` マクロを、標準ライブラリからまねしながらコピーしてくる。

```
#[macro_export]
macro_rules! print {
    ($($arg:tt)*) => ($crate::vga_buffer::_print(format_args!($($arg)*)));
}

#[macro_export]
macro_rules! println {
    () => ($crate::print!("\n"));
    ($($arg:tt)*) => ($crate::print!("{}\n", format_args!($($arg)*)));
}
```

```
#[doc(hidden)]
pub fn _print(args: fmt::Arguments) {
    use core::fmt::Write;
    WRITER.lock().write_fmt(args).unwrap();
}
```

これで完成だ！すると main.rs で、次みたいなことができる。

```
pub mod vga_buffer;

#[no_mangle]
pub extern "C" fn _start() -> ! {
    println!("Hello Hello, World!\nsome numbers: {} {}", 42, 1.337);

    loop {}
}
```

注目してほしいのは、もうフォーマットが動いていることだ。printf にあたるものがこんなにスムーズに使えるのはすごい。これをビルドして起動すると、無事に図 4 のように、Qemu の画面上に新たな Hello, World! が表示された。

3.3. No Magic with Multiboot Header

さて、先ほどまでは Philipp's の 2nd edition に基づいて Rust でカーネルを書いてきた。Philipp 氏が用意してきた Crate の手軽さもあいまって、あまりベアメタルプログラミングらしいことをせずともあれよあれよという間に Hello, World! が無事画面に躍り出ることになった。でも、正直に言おう。ちょっとこれじゃ高級すぎてよくわからない！込み入ったことをしてくると、リンカスクリプトを書いたりブートプロセスを工夫したりするシーンがすぐにやってくるだろうから、もう少し、地に足のついた方法でブートについては試したい。また、bootimage + bootloader は、マイクロカーネル方式と相性が悪いという現実的な問題もある。Multiboot の説明の際に話したように、マイクロカーネルではブートローダーに複数の独立したバイナリモジュールをメモリ空間に読み込んでもらう必要があるからだ。そういうわけで、古き良き Multiboot Header をスクラッチでアセンブリで書いていこう。その後、自分で書いたスタックやレジスタの初期化、それにロングモードへのジャンプなんかと合わせてと、Rust で書いたバイナリとリンクしていくことにする。

図 4　Hello World 2

…

…

…

…

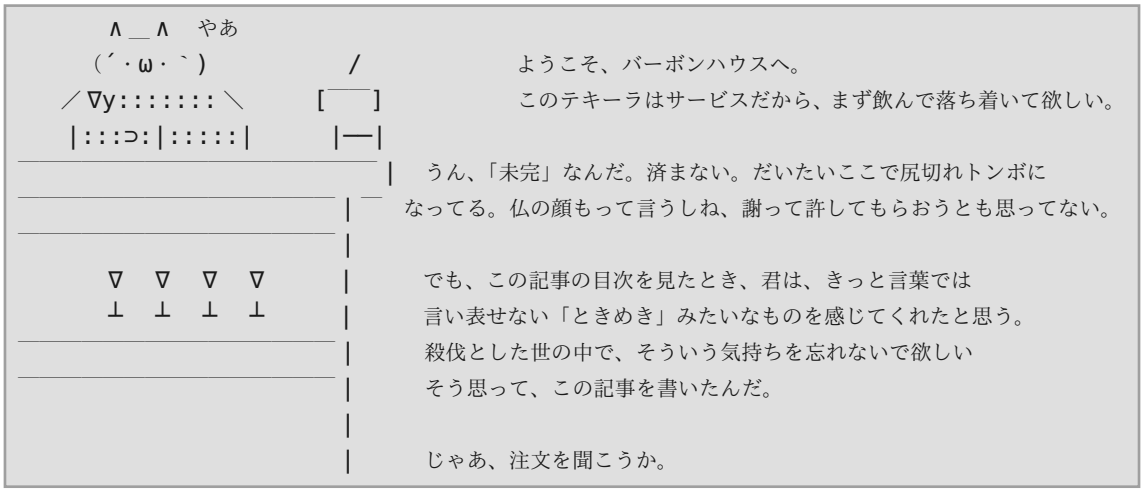
…

…

…

…

3.4. Welcome



nullpo_head はどうして締め切りに間に合わない？原因は？対策は？調べてみました！

はい、ここで締め切りです。いかがでしたか？いよいよ本番始まったな？ってところで尻切れトンボになったら「いかがでしたか？」もくそもないですね。とても反省しています。ちなみに原因と対策を考えると「仕事を辞める」が最善策になります。平日何もできなさすぎる。

一応、この記事では、導入部分と Philipp さんの crate を使いまくったうえでの Rust でのカーネル開発の導入までは完結しました。これ以降はヤバイテックトーキョーの次号連載をお楽しみにしてください。この後は、上にちらっと書いてあるように、Multiboot を使って、static library としてビルドしたカーネルと色々リンクさせて頑張っていく感じになります。記事が完成しなかったので先にいうとスタックは深く初期化しておけ、Rust はデバッグビルドだと lazy_static ともあわさってアホ程スタック食う。何かの役に立つかもしれないから、以降の続きの筋書も、予告編としてメモ書き程度に残しておきます。また、コード自体はもう少し先まで進んでいるので、解説は間に合っていませんが、参考にはできるかもしれません。https://github.com/nullpo-head/rusty-l4 なので、生暖かく見守りたい人は watch でもしといてください。ところでこの連載記事は、Web で公開するときは、勝手に英語になってくれてたらうれしいなという願望があります。Peddle OS (https://

pebble-os.github.io/) とかいうもろかぶりネタが進行中ですし。そんなことを考えていたら本文をなんとなく翻訳調の文章で書いてみたくなった感じです。このノリの源流ってどこなんでしょうね。

そんなわけで、2 日目の途中で今回での記事は終了ですが、次回の連載を楽しみにしてくれたら幸いです。いかがでしたか？以降、次号の予告編です。

3.5. 予告編

- multiboot ヘッダーを書こう
- まず、startup.S を書いて、Qemu のシリアル機能を使って無事'A'という文字を出力！無事ブートできたみたいだ。
- 急になるが環境を一気に用意する
- リンカスクリプトを書こう
- multiboot v1 に別れを告げる
- grub2 用の起動ディスクを用意
- WSL 上だと（というかおそらく実際に grub で起動していない環境だと）grub-mkrescue でうまくブートデバイスが作れない。`grub-pc-bin` をインストールするとうまくブータブル ISO が作られるようになる。See https://github.com/Microsoft/WSL/issues/807
- ロングモードまで入ってさっさと Rust の世界に帰ることにしよう。
- 個人的にはこの部分を書くのがいつも一番面倒だ！ 1GB ページと GDT をさっさと用意してしまおう
- Rust とは関係ない、Multiboot の部分で随分と苦労したが、Philipp の高級な crate を使ってブートした時と同じ Hello, World を無事表示できた。これで Rusty L4 の最小カーネルは完成だ！

4. 予告編 ; 3rd Day: Write a Microkernel Server for L4 Pistachio

さて、この挑戦も今日も 3 日目だ。もっとも、開発をし終わったあとにこの文章を

編集して、 こまぎれの記録を一つにまとめて皆さんにはお見せしているから、 私としては実際にはだいぶ日数が経っているのだけど。 ともあれ、 昨日で私は、Rust でのベアメタルプログラミングの世界を体験してきた。 最終的には、Qemu の画面上に、 無事に"Hello, World!"を表示できたわけだ。 ベアメタルプログラミングをしているにもかかわらず、C での開発のときと違い、crate を使って簡単にブートができてしまったことや、`println!` マクロが使えるまでの道のりの簡単さは本当に驚異的だった。 ベアメタルプログラミングをやっているにもかかわらず、Trait なんていう普段 Web 開発をしているときに使うような高級な概念を普通に使える感覚も新鮮だ。Rust はベアメタルプログラミングをもっと簡単に、そして楽しくしてくれそうだというのは間違いない。

おっと！ Rust に関する話に少し脱線してしまったようだ。そろそろ今日、3 日目にやることについて話そう。今日は一旦フルスクラッチのベアメタルプログラミングの世界から離れて、L4 マイクロカーネル上での世界に戻ってみる。 まずは、VGA のメモリ領域をたたいて、Hello, World! と表示させるだけのサーバーから始めよう。明日以降、これを自分で書いたカーネルの上で走らせることを目標とするためだ。 今回書こうとしているサーバーは、1 番目に起動することを想定する特殊なサーバーであることには留意してほしい。多分普通にサーバーを書く分にはもっと苦労しないはずだ。今回のこの作業を通して、一般に CPU 上で OS のカーネルがメモリ空間をどう管理するかについても触れるかもしれない。

参考文献

[1] ウィキペディアの執筆者 . マイクロカーネル . `http://bit.ly/2FS4h4M`, 2019.

[2] Operating Systems Group Dresden. *L4 Linux*. `http://bit.ly/2FS4g0I`, 2019.

[3] Kazuya Kodama. *L4/Minix WWW Page*. `http://bit.ly/2FQPNlN`, 2005.

[4] MkLinux authors. *Welcome to MkLinux.org*. `https://www.mklinux.org/`, 2008.

[5] Data61/CSIRO. *The seL4 Microkernel*. `https://sel4.systems/`, 2016.

[6] Data61/CSIRO. *The Proof*. `http://sel4.systems/Info/FAQ/proof.pml`, 2016.

[7] Free Software Foundation, Inc. *GNU Hurd*. https://www.gnu.org/software/hurd/, 2016.

[8] Jochen Liedtke. Improving IPC by kernel design. *14th ACM Symposium on Operating System Principles*, pages 175–188, 1993.

[9] ウィキペディアの執筆者 . *L4* マイクロカーネルファミリー . http://bit.ly/2FPwAAF, 2019.

[10] ウィキペディアの執筆者 . *Mach*. https://ja.wikipedia.org/wiki/Mach, 2019.

[11] ウィキペディアの執筆者 . アンドリュー・タネンバウムとリーヌス・トーヴァルズの議論 . http://bit.ly/2FPDTIJ, 2019.

[12] NETWORK WORLD. *What is MINIX? The most popular OS in the world, thanks to Intel*. http://bit.ly/2FPEbzj, 2017.

[13] Free Software Foundation, Inc.. *Multiboot Specification version 0.6.96*. http://bit.ly/2FPTQPg, 2010.

[14] Kevin Elphinstone and Gernot Heiser. From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels?. *SOSP'13*, pages 0–0, 1993.

[15] Tanenbaum Andrew S. Lessons Learned from 30 Years of MINIX. *Commun. ACM*, 59, pages 70–78, 2016.

定理証明支援系を作ろう！

wasabiz

1. 目標

定理証明支援系というと Coq[22] や Isabelle[21] のような巨大なソフトウェアを思い浮かべる人も多いかもしれません．しかしその実，大抵の証明支援系は核となる非常に小さい論理の上に証明を便利に記述するための殻が覆いかぶさっているにすぎません．そこで，今回は証明支援系をある程度の使いやすさを保証しつつもなるべく**省エネ・省コード**に実装し，定理証明支援系が思ったよりもとっつきやすいものであるということを確認したいと思います．

一言で定理証明支援系を作ると言っても，それがまともに使える状態であるためには核となる論理の部分を実装する以外にも以下のような機能が必要になると思われます．

- パーサー
- 束縛関係の管理
- 型検査
- 型推論
- 効率的な評価器
- タクティック言語

以上の機能は一つ一つを取り出してみてもそれなりに実装するのが大変なものばかりです．そこで，今回は，これらの機能を**全て実装せずに**定理証明支援系をつくります．と言っても，これらの機能が提供されないと言うわけではありません．既存の機能と理論をうまく活用してこれらを極めて小さいコードだけで実現しようというのです．ポイントは，**定理証明支援系の上で定理証明支援系と作る**ということです．たとえば Coq にはパー

サーも，束縛関係の管理も，型検査器も，型推論器も，効率的な評価器も，タクティック言語もすでに実装されています．Coq の上で定理証明支援系を作成すれば，(非常にうまくやれば)Coq 自体に実装されているこれらの機能をハックして利用することができます．具体的には以下のような方針で，様々な言語機能の実装を**サボります**．(ややこしいので，実装に用いる定理証明支援系を**メタ言語**，これから実装される言語を**オブジェクト言語**と呼びます．)

- パーサー : DSL
- 束縛関係の管理 , 型検査 , 型推論 : 高階抽象構文 (HOAS)
- 効率的な評価器 : 評価による正規化 (NbE)
- タクティック言語 : メタ言語のタクティック言語

具体的な方針は以下の通りです．まず，核となる論理として**高階論理**と呼ばれる論理を採用します．高階論理は簡単に言えば普通の論理にラムダ抽象がはいったようなもので，例えば以下のような命題を記述することができます．

$$\Big(\lambda P^{o\to(\iota\to o)\to o}.\forall f^{\iota\to o}.\neg P\big(\top, f\big)\Big)\big(\lambda\varphi^{o}.\lambda Q^{\iota\to o}.Q(c)\ \wedge\ \varphi\big)$$

高階論理は (数学の大部分を直接記述できる程度に) 非常に強力な表現力を持ちながら極めて簡潔な構文と演繹体系を持ちます．よって今回の目標にはぴったりです．また，実装に使用する言語は Lean と呼ばれる言語です．[20] Lean は 2014 年ごろから Microsoft Research で開発されている定理証明支援系で，Coq とよく似た型理論に基づく言語です．Lean は Coq よりも現代的な設計と強力なメタプログラミング機構を持つ言語で，個人的にとても気に入っている言語です．[*13]

記事の構成は以下の通りです．この記事では高階論理の実装を段階的に行なっていきます．高階論理は型付きラムダ計算に命題の型と項の等さを表す述語を加えた体系と見なせます．

---

13 実はこの記事を書き始めた段階では Haskell での実装を予定していました．しかし，Haskell の型システムには dependency がなく，論理の推論規則が満たすべき性質を証明するのが難しかったのと，型レベルのラムダ抽象がないせいでコードが不必要に複雑化してしまうため使用をやめてしまいました．

$$\text{高階論理} = \text{型付きラムダ計算} + \text{命題の型 } o + \text{項の等さを表す述語 } M_1 \doteq M_2$$

さらに型付きラムダ計算は型のないラムダ計算に型を入れたものだと考えることができます.

$$\text{型付きラムダ計算} = \text{(型なし) ラムダ計算} + \text{型}$$

そこで, 2 節ではまず, 型のないラムダ計算を定義し実装します. つぎに, 3 節では型のついたラムダ計算を定義し実装します. 最後に, 4 節で型のついたラムダ計算を拡張して高階論理を定義し実装します.

この記事の読み方について簡単に説明します. この記事では定理証明支援系を作成するのにあたって必要な理論的背景とその実装の解説を交互に挟んでいます. 理論的な話をする小節にはタイトルに ★ マークをつけておきました. もし理論方面に興味がある方は ★ マークのついた理論方面の節だけを読むのもいいでしょう. 特に, 高階論理について最短で知りたいという方は ★ マークのついた節だけを読んで 4 章に突入するのも良いと思います. また, 実装から読み始めたほうが理解が早いという人は実装の部分だけを読んでしまっても問題ないと思います. なんなら, 一番最後の節から読み始めてもよいと思います. あと, Lean は知名度がないので今回は Lean の構文を知らない読者にもわかるよう随所に Lean 自体の解説も挟みました. 具体的な目次は以下の通りです.

目次

- 3.6 節 Church 流の単純型付きラムダ計算 (その 2) ★
- 3.7 節 型判断の実装 (その 1)
- 3.8 節 型判断の実装 (その 2)

- 4 節 高階論理
  - 4.1 節 高階論理の意味論 ★
  - 4.2 節 高階論理の演繹体系 ★
  - 4.3 節 高階論理の実装
- 5 節 まとめ

2. ラムダ計算

2.1. 型無しラムダ計算 ★

この小節では**型無しラムダ計算** (untyped lambda calculus) について復習します．まず，変数の無限集合 Var が定まっているとします．Var の要素は以下で $x, y, z, \cdots$ で表すことにします．型無しラムダ計算の**項** (**ラムダ項** , lambda term) は以下の BNF で帰納的に定義されます．

$$M ::= x \mid \lambda x.M \mid MM$$

$\lambda x.M$ の形のラムダ項を**ラムダ抽象**と呼び，MN の形のラムダ項を**適用**と呼びます．適用は左結合とし，$\lambda x_1.\cdots\lambda x_n.M$ を略記して $\lambda x_1,\cdots,x_n.M$ と書きます．すべてのラムダ項からなる集合を Tm と書きます．型無しラムダ計算の項は型付きラムダ計算の項と対比して**前項** (preterm) と呼ばれることもあります．

型無しラムダ計算でもっとも重要な概念が変数の束縛です．まず，ラムダ項 M の**自由変数**の集合 $\mathrm{fv}(M) \subseteq \mathrm{Var}$ を以下のように構造に関する帰納法により定義します．

$$\begin{aligned} \mathrm{fv}(x) &:= \{x\} \\ \mathrm{fv}(\lambda x.M) &:= \mathrm{fv}(M) \setminus \{x\} \\ \mathrm{fv}(MN) &:= \mathrm{fv}(M) \cup \mathrm{fv}(N) \end{aligned}$$

変数 x が $\mathrm{fv}(M)$ に属するとき，x は M について**自由である** (free) と言います．変数 x が $\mathrm{fv}(M)$ に属さない時，x は M について**自由でない** (non-free) とか**新鮮である** (fresh) と言い，$x \# M$ と書きます．また，$\mathrm{fv}(M) = \varnothing$ であるとき，M を**閉じている** (closed) といい，閉じていない項を**開いている** (open) といいます．閉じた項は**閉項**とも呼ばれます．

つぎに集合 Tm の上に α **同値** (α-equivalence) と呼ばれる同値関係を導入します． 多くの標準的な教科書 (たとえば，[1]) では α 同値を厳密には定義せず，いくつかの例を示して定義に代えています． ここでは正確な記述のために，Nominal set theory[2, 3] で用いられている α 同値の定義を簡単に紹介します．[3] では圏論の言葉を用いて Nominal set theory が展開されていますが， ここではラムダ計算を定義するのに必要な部分だけを初等的な言葉のみで説明します．

ここで先に種明かしというかネタバラシをしてしまうと， 以下の内容は α 同値性の正確な記述の大変さを読者のみなさまに感じてもらうのが狙いです． コンピュータ上でラムダ計算を実装する際には当然定義のあやふやさは許されないので， 以下で解説する数学的大変さがそのままプログラミングの際にも問題になります． この後ラムダ計算の実装の話に移ったときには，以下で解説される α 同値性の複雑さをいかに回避しつつ α 同値性を実装するかを議論します． というわけで， ここからしばらくは「α 同値の概念が実はとても複雑である」ということさえわかってもらえれば十分です．(とはいえ， 形式的な議論に慣れている人には以下の話はそれだけでも十分面白いと思います．)

変数 $x,y \in \mathrm{Var}$ に対して写像 $\left(x\,y\right) : \mathrm{Var} \to \mathrm{Var}$ を，x を y に，y を x に写し，それ以外の場合は入力をそのまま返すものと定義します．このような写像を**互換** (transposition) と呼びます．互換 π について，Tm から Tm への写像 $\pi \cdot (-)$ を構造に関する帰納法により定義します．

$$\begin{aligned} \pi \cdot x &\coloneqq \pi(x) \\ \pi \cdot (\lambda z.M) &\coloneqq \lambda\pi(x).\pi \cdot M \\ \pi \cdot (MN) &\coloneqq (\pi \cdot M)(\pi \cdot N) \end{aligned}$$

α 同値性を定義するために新しい量化子 И を導入します．(ちなみに，И は「for fresh」と読みます．)

$$Иx \in X.\, \varphi(x) :\Longleftrightarrow X \setminus \left\{x \in X \;\middle|\; \varphi(x)\right\} \text{ is finite}$$

$Иx \in X.\, \varphi(x)$ は数学的には「有限個を除く全ての $x \in X$ について $\varphi(x)$ が成立する ($\varphi(x)$ holds for all but finitely many $x \in X$)」という意味になります．以下では特に X として Var を取り，φ としてラムダ項についての命題を取ります．その場合，実は $Иx.\, \varphi(x)$ は「新鮮な変数 x について $\varphi(x)$ が成り立つ」ことと同値になります．[*14] [*15]

最後に α 同値性を定義します．二項関係 $=_\alpha \subseteq \mathrm{Tm} \times \mathrm{Tm}$ を以下の規則に基づく帰納的定義により定義します．

$$\frac{}{x =_\alpha x} \qquad \frac{Иy.\, (x\, y) \cdot M =_\alpha (x'\, y) \cdot M'}{\lambda x.M =_\alpha \lambda x'.M'} \qquad \frac{M_1 =_\alpha M_2 \qquad N_1 =_\alpha N_2}{M_1 N_1 =_\alpha M_2 N_2}$$

実はこのような定義を採用することで $=_\alpha$ は同値関係になります．$M =_\alpha N$ が成り立つ時，M と N は α **同値**であると言います．直観的には，二つのラムダ項が α 同値であるとはそれらが同じ束縛構造を持つことを意味します．

α 同値なラムダ項はすべて同一視したほうが都合が良いので，以下ではそのようにします．そのために，まず Λ を商集合 $\mathrm{Tm}/{=_\alpha}$ として定義し，Λ の元 $[M]_{=_\alpha}$ を (記法を乱用して) 適当な代表元 M で表します．

さて，すこし話を遡って Tm の定義を思い出してみます．Tm は帰納法で定義した集合なので，Tm の元について何か性質を証明したい場合は帰納法の原理を用いることができました．一方で，Λ は帰納法で定義したわけではなく，商集合として定義しました．そのため，Λ の元について何か性質を証明しようと思っても帰納法が使えません．しかし実は，Nominal set theory の言葉を用いれば，集合 Λ をある種の帰納法によって定義することができます．[*16]その定義を経由することで Λ の元にかんする性質を帰納法によって証明することができます．Λ が Tm を α 同値で割ったものだったことを思い出せば，これは

---

14 より正確には「『φ に出現する全ての項に対して新鮮な』x について $\varphi(x)$ が成り立つ」ことと同値になります．

15 ここでは「『全ての』新鮮な変数 x について」なのか「『ある』新鮮な変数 x について」なのかを明示しませんでしたが，実はどちらと思って読んでも同値になることが知られています．詳しくは [3] の定理 3.9 を参照してください．

16 Nominal set の圏 $\mathbb{N}\mathfrak{om}$ の上の始代数になります．

Tm の上の帰納法に α 同値なものを同一視するような拡張を施したものと見なせます.

Λ の帰納法を説明するために**写像の自由変数**について説明します. 先ほど定義した互換 $(x\ x') : \mathrm{Tm} \to \mathrm{Tm}$ は α 同値性を保ちます.

補題 1　$M_1 =_\alpha M_2$ のとき $(x\ x') \cdot M_1 =_\alpha (x\ x') \cdot M_2$.

よって, 互換は Λ 上で well-defined な写像です. $f_1 : \mathrm{Var} \to \Lambda$ を写像とします. f_1 に**自由変数の集合が定義される**とは, 変数の有限集合 $F \subseteq \mathrm{Var}$ が存在して

$$\forall x, x' \in \mathrm{Var} \setminus F.\ \forall y.\ (x\ x') \cdot f_1(y) = f_1\big((x\ x') \cdot y\big)$$

が成り立つことであるとします. f_1 に自由変数の集合が定義されるとき, そのような F の中で最小なものを f_1 の**自由変数の集合** $\mathrm{fv}\left(f_1\right)$ と呼びます. これは交差を使って具体的に与えることができます.

$$\mathrm{fv}\left(f_1\right) \coloneqq \bigcap \left\{ F \subseteq \mathrm{Var} \;\middle|\; \forall x, x' \in \mathrm{Var} \setminus F.\ \forall y.\ (x\ x') \cdot f_1(y) = f_1\big((x\ x') \cdot y\big) \right\}$$

具体例を挙げましょう. たとえば, 項 M と変数 x を固定した上で, 以下のように関数 f_1 を定義します. すると, f_1 には自由変数の集合が定義されます. より具体的には, 自由変数の集合 $\mathrm{fv}\left(f_1\right)$ は $\mathrm{fv}(M) \cup \{x\}$ となります.

$$f_1(y) = \begin{cases} M & y = x \\ y & \text{otherwise} \end{cases}$$

上の定義は, 通常ラムダ項にのみ定義される「自由変数の集合」の概念をラムダ項を値にとる写像に拡張したものです. この例のように, 一般に写像は入力に対して条件分岐を行ったり, 何らかの定数ラムダ項を用いるなどして定義されます. そのようにして写像の定義中に現れるラムダ項 (や変数) の自由変数を全部かきあつめたものをその写像の「自由変数の集合」と呼んでいます. ただし, このままだと場合によっては「自由変数の集合」が無限集合になってしまうことがあります. どんなラムダ項もその自由変数の

集合は有限集合だったので，「自由変数の集合」の概念を写像に一般化する際にもこの性質を要求した方が都合がよいです．つまり，写像に自由変数の集合が定義されるとは，その自由変数が高々有限個に抑えられるということに他なりません．

ここまでは写像として $\mathrm{Var} \to \Lambda$ という形のもののみを考えてきました．しかし，その制限は本質ではなく，基本的にどんな形の写像に対しても同じように自由変数の概念を定義することができます．例えば，詳細は省きますが，f_1 と同様に，$f_2 : \mathrm{Var} \times \Lambda \to \Lambda$，$f_3 : \Lambda \times \Lambda \to \Lambda$ のような型の写像に対しても $\mathrm{fv}\left(f_2\right), \mathrm{fv}\left(f_3\right)$ を定義することができます．[*17]

定理 2　写像 $f_1 : \mathrm{Var} \to \Lambda$，$f_2 : \mathrm{Var} \times \Lambda \to \Lambda$，$f_3 : \Lambda \times \Lambda \to \Lambda$ をとる．f_1, f_2, f_3 それぞれに自由変数の集合が定義されているとする．さらに，以下の条件を満たすとする．

$$x \notin \mathrm{fv}\left(f_1\right), \mathrm{fv}\left(f_2\right), \mathrm{fv}\left(f_3\right) \Longrightarrow \forall M.\, x \# f_2(x, M)$$

すると写像 $f : \Lambda \to \Lambda$ であって，

- $f(x) = f_1(x)$
- $f(\lambda x.M) = f_2\left(x, f(M)\right)$ *(ただし，$x \notin \mathrm{fv}\left(f_1\right), \mathrm{fv}\left(f_2\right), \mathrm{fv}\left(f_3\right)$)*
- $f\left(M_1 M_2\right) = f_3\left(f\left(M_1\right), f\left(M_2\right)\right)$

を満たし，かつ自由変数の集合が定義されているようなものがただ一つ存在する．

この定理を用いることで，ラムダ項の**置換**を定義することができます．変数 x とラムダ項 M について，以下の写像 f_1, f_2, f_3 で定理により誘導される写像を $[M/x](-) : \Lambda \to \Lambda$ と書きます．$[M/x](-)$ を適用することを x を M で**置換する**と言います．

- $f_1(y) = \begin{cases} M & \text{y = x} \\ y & \text{otherwise} \end{cases}$

17 より正確な定義は教科書を参照してください．Nominal set theory では写像 f に自由変数の集合が定義されていることを f は finitely supported であると言います．また，$\mathrm{fv}(f)$ を $\mathrm{supp}(f)$ と書きます．

- $f_2(x, M) = \lambda x.M$
- $f_3(M_1, M_2) = M_1 M_2$

いずれも well-defined であり，定理の条件を満たすことが簡単に確認できます．定理 2 ではラムダ項からラムダ項を作るための帰納法を与えていますが，より一般にはあらゆる述語・構造について Λ の帰納法が存在します．

置換を用いてラムダ項の β **簡約**を定義します．二項関係 $\to_\beta \subseteq \Lambda \times \Lambda$ を以下のひとつの規則とみっつの合同性の規則で帰納的に定義します．

$$\frac{}{(\lambda x.M)N \to_\beta [N/x]M}$$

$$\frac{M \to_\beta M'}{\lambda x.M \to_\beta \lambda x.M'} \qquad \frac{M \to_\beta M'}{MN \to_\beta M'N} \qquad \frac{M \to_\beta M'}{NM \to_\beta NM'}$$

$M \to_\beta N$ であるとき，M が N に β **簡約される**と言います．単に簡約と言った場合，β 簡約のことを指します．一度簡約された項が再度簡約されることもあり得ます．ラムダ項の長さ 1 以上の加算列 $\left(M_i\right)_i$ が全ての (有効な)i について $M_i \to_\beta M_{i+1}$ を満たす時，その列を**簡約列**と言います．特に M_0 を明示して，M_0 **の簡約列**とも言います．ある簡約列が無限列であることを**発散する**と言います．M の長さ 2 以上の簡約列が存在しない時，M を β **正規形** (β-normal form) あるいは単に**正規形** (normal form) と言います．また，M の無限簡約列が存在しない時，M は**強正規化可能である**と言います．$\to_\beta$ の反射推移対称閉包を $=_\beta$ を書きます．定義より，$=_\beta$ は Λ 上の同値関係です．$M =_\beta N$ であるとき，M と N は β **同値である**と言います．

2.2. 高階抽象構文

2.1 節でラムダ計算の形式的定義を行う際に α 変換を定義するのに苦労しました．これはラムダ計算をコンピュータ上で実装する場合でも同様で，束縛の構造を正しく表現するためには (特に効率よく実装するためには) それなりの量と複雑さを持つコードを書かなければなりません．そこでこの記事では**高階抽象構文** (Higher-Order Abstract Syntax, HOAS) と呼ばれる手法を用いてラムダ計算を実装し，**束縛の構造に関するコードをほと**

んど記述しないまま実装します．直観的には，HOAS はメタ言語 (今回の場合は Lean(の処理系)) に束縛関係の管理を外注してしまうという手法です． この手法ではオブジェクト言語 (今回の場合はラムダ計算) は必然的にメタ言語の中で DSL として実装されます．そのため，ある意味ではパーサーもメタ言語の処理系に外注していると言えます．結果的にですが，HOAS を採用することによりオブジェクト言語のパーサーを実装する必要もなくなったことになります．

さて，Lean で HOAS を用いてラムダ計算を定義してみます．ラムダ項を表す型 `preterm` は以下のように定義できます．`inductive` は Lean で代数的データ型を定義するための構文です．Lean の代数的データ型は常に Haskell でいうところの GADT の流儀で定義されます．

```
inductive preterm : Type
| lam : (preterm → preterm) → preterm
| app : preterm → preterm → preterm
```

例えば， ラムダ項 $\lambda x.\lambda y.yx$ は `preterm` では `lam (λ x, lam (λ y, app y x))` として表現されます． つまり， 高階関数をうまく用いることで， メタ言語の束縛構造を利用してオブジェクト言語の束縛構造を定義しています． これにより， オブジェクト言語の α 同値な項はすべて同一視されます．(メタ言語の処理系が α 同値な (メタ言語の) 項を同一視する機能を実装しているからです．) しかし，実は上に挙げたコードは Lean に与えるとエラーが出ます．

```
hol.lean:1:0: error
arg #1 of 'preterm.lam' has a non positive occurrence of the datatypes bei
ng declared
```

これは，Lean では停止するコードにしか型がつかないという制約によるものです．`inductive` を用いてデータ型を定義すると帰納法を行うためのコードが自動で生成されます． しかし `preterm` に対しては停止する帰納法が定義されないのです． これは直観的には， 集合論

的モデルではこのような不動点 (解) であって非自明なものが存在しないためです．[*18] Lean の仕様では一般的にどのような帰納的データ型の定義が許されているかがもう少しわかりやすく定められています．[*19]一方で Haskell の型システムは Lean よりも寛容なので `preterm` のような型が実際に定義できます．そのため，この問題は Lean のような定理証明支援系固有の問題と言えます．

さて，ここまで HOAS を Lean で直接は定義できないという話をしてきましたが，実は仮に直接定義できたとしても，HOAS には実用上の問題があります．一つ目の問題は一旦構築した項の中身を読み出せないということです．もう一つの問題は `preterm` 型の値の中に対応するラムダ項が存在しないものがあるということです．まず，一つ目の問題は，例えば `preterm` 型の値を通常の表記に変換する `repr` に相当する関数が定義できないということを指します．[*20]この問題は古くから知られている問題で，[6] ではこのような構文木を指して「write-only syntax」と呼んでいます．Write-only syntax を readable にするためには自由変数を表すコンストラクタを `preterm` を追加する方法が知られています．しかし，この方法を用いると自由変数の束縛を自前で管理する必要があり，HOAS の良さであった束縛関係に関する処理を全く書く必要がないという点が失われてしまいます．二個目の問題も古くから知られている問題で，[7] などでも指摘されています．たとえば，isapp なる (メタ言語の) 関数を以下のように定義します

$$\mathrm{isapp}\big(\mathrm{app}(M,N)\big) \coloneqq \mathrm{true}$$
$$\mathrm{isapp}\big(\mathrm{lam}(f)\big) \coloneqq \mathrm{false}$$

ただし，true, false はそれぞれ真と偽を Church encoding で表したラムダ項とします．もし Lean で実装する場合は以下のようなコードになるでしょう．

```
def isapp : preterm → preterm
| (lam f) := false
```

---

18 集合 X に関する方程式 $X \cong \{f : X \to X\}$ の解は一点集合のみです．これの数学的基礎付けについては**領域理論** (Domain theory) の教科書が良い導入になるかも知れません．(たとえば [5] など)

19 たとえば [4] の 4.4 節．

20 なお，この問題はメタ言語が型付きの場合にのみ起こります．

```
| (app m₁ m₂) := true
```

このとき，$\mathrm{lam}(\mathrm{isapp})$ は `preterm` 型をもちかつ (メタレベルで) 閉じた項であるので，なんらかのラムダ項を表していることが期待されますが，実際には対応するラムダ項は存在しません．

2.3. 多相高階抽象構文

2.2 節で挙げた HOAS の欠点を克服する手法が**多相高階抽象構文** (Parametric HOAS, PHOAS) と呼ばれる手法です．PHOAS で前項を表す型を定義すると以下のようになります．

```
inductive preterm (α : Type) : Type
| var : α → preterm
| lam : (α → preterm) → preterm
| app : preterm → preterm → preterm
```

このように型 `preterm` が型変数 `α` によってパラメータ化されています． 各コンストラクタの型シグネチャに現れている `preterm` はゼロ引数に見えますが， 暗黙に `α` が適用されています．(これは Lean の仕様で，帰納的定義を行う際は：の前の引数は再帰の中で常に固定されていると解釈されます． この場合は，`inductive preterm (α : Type) : Type` という行の：の前に `(α : Type)` があるのでこの `α` は定義中で固定されています．) さて， このデータ型は通常 `α` が全称量化された状態で使われます．

```
def Preterm : Type 1 := Π (α : Type), preterm α
```

すると，`Preterm` はすべての閉じたラムダ項を表す型となります． [*21]直観的には以下のように説明されます． まず `var` の引数と `lam` の引数の引数が型変数 `α` になっていることに注目してください．`α` が全称量化されている場合，`var` の引数として取れる `α` 型の値は `lam` で束縛したものに限られます． よって閉項のみが表現されます． さらに，`α` が全称量化されていることにより，`preterm α` 型の値は `var,lam,app` のいずれかで生成されたもの

21 定義された変数の先頭文字が大文字になっていますが， これは Lean では特別な意味を持ちません．

に限られます．よって，Preterm はすべての閉じたラムダ項を忠実に表現する型と言えます．

ちなみに，細かい点ですが，Lean の型システムでは Type は predicative なので Preterm のような dependent product は通常の型の宇宙 (Type あるいは Type 0 で表される) には収まらない程度に大きくなります．そのため Preterm 自体の型を Type 1 というひとつ大きな宇宙にしています．

ではこの Preterm を用いて型無しラムダ計算が実現できるか確認します．Preterm は閉項しか表現できませんが，しばらくは開いた項を扱わないので問題ありません．Preterm は型が量化されているので，Preterm 型の項の中身を取り出す場合は，まずどのような型で実体化するかを考える必要があります．たとえば deBrujin level (有名な deBrujin index の変種で，束縛変数の数字が項の内側にいくにつれて増えていくようにしたもの) を使って項を文字列に変換する場合はこの α として自然数の型 ℕ を採用します．

```
def preterm.repr' : preterm ℕ → ℕ → string
| (var n) _ := repr n
| (lam f) lv := "(λ" ++ repr lv ++ "." ++ preterm.repr' (f lv) (lv + 1) +
+ ")"
| (app m₁ m₂) lv := "(" ++ preterm.repr' m₁ lv ++ " " ++ preterm.repr' m₂ l
v ++ ")"

def preterm.repr (m : Preterm) : string :=
preterm.repr' (m ℕ) 0
```

Lean には Haskell や Coq と同じく型クラスが用意されています． この定義を用いて has_repr 型クラスのインスタンスを宣言して，値の文字列化を行う汎用関数 repr が適用できるようにします．

```
instance Preterm_has_repr : has_repr Preterm :=
⟨preterm.repr⟩
```

Parametric でない HOAS では実現できなかった repr を記述することができました．

では，実際にラムダ項を作成してみます．最も簡単な例は引数をひとつとってそれをそのまま返すラムダ項 $\lambda x.x$ です．この項はよく i という名前で呼ばれます．i を Preterm の

値として記述すると以下のようになります．

```
def i : Preterm := λ α, lam (λ x, var x)
```

先ほどインスタンスを定義した `repr` を用いてこれを表示してみます．

```
#eval i
--> (λ0.0)
```

`--` は Lean の一行コメントです．`-->` 以降に書いた文字列 `(λ0.0)` が `repr i` の結果です．正しく表示できているようです．同様に s や k と呼ばれる以下の項も定義してみます．

$$s = \lambda x.\lambda y.\lambda z.(xz)\big(yz\big)$$
$$k = \lambda x.\lambda y.x$$

s と k はそれらの適用による組み合わせで任意のラムダ項 (に β 同値な項) を表現できるという意味で非常に重要な項なのですが，ここでは特に掘り下げないことにします．

```
def s : Preterm := λ α, lam (λ x, lam (λ y, lam (λ z, app (app (var x) (va
r z)) (app (var y) (var z)))))
def k : Preterm := λ α, lam (λ x, lam (λ y, var x))
```

やはり `repr` を使って表示してみます．

```
#eval s
--> (λ0.(λ1.(λ2.((0 2) (1 2)))))
#eval k
--> (λ0.(λ1.0))
```

うまく動作しているようです．

次に自然数の Church encoding を試してみます．Church encoding では自然数を以下のようなラムダ項として表現します．

$$
\begin{aligned}
0 &= \lambda f.\lambda x.x \\
1 &= \lambda f.\lambda x.fx \\
2 &= \lambda f.\lambda x.f\big(fx\big) \\
3 &= \lambda f.\lambda x.f\Big(f\big(fx\big)\Big)
\end{aligned}
$$

4 以上の自然数に対しても同様に定義されます． 以上のラムダ項を PHOAS で記述します．

```
def n₀ : Preterm := λ α, lam (λ f, lam (λ x, var x))
def n₁ : Preterm := λ α, lam (λ f, lam (λ x, app (var f) (var x)))
def n₂ : Preterm := λ α, lam (λ f, lam (λ x, app (var f) (app (var f) (va
r x))))
def n₃ : Preterm := λ α, lam (λ f, lam (λ x, app (var f) (app (var f) (app (
var f) (var x)))))
```

念のため正しく定義できているか確認します．

```
#eval n₀
--> (λ0.(λ1.1))
#eval n₁
--> (λ0.(λ1.(0 1)))
#eval n₂
--> (λ0.(λ1.(0 (0 1))))
#eval n₃
--> (λ0.(λ1.(0 (0 (0 1)))))
```

つぎに自然数を受け取って次の自然数を返す `succ` を定義します．

```
def succ : Preterm := λ α, lam (λ n, lam (λ f, lam (λ x, app (var f) (app (a
pp (var n) (var f)) (var x)))))
```

`succ` を使う時には 2 つ注意点があります．ひとつ目は，`Preterm` 型の二つの値を受け取って適用する場合は型変数を適切に分配してやる必要があるということです． そこで， 二つの `Preterm` を受け取ってそれらの適用を返す補助関数 `App` を定義しておきます．

```
def App : Preterm → Preterm → Preterm :=
λ m₁ m₂ α, app (m₁ α) (m₂ α)
```

二つ目は，(当然ですが) 単に適用した項を定義しただけでは項の簡約は行われないということです．そのため，succ_n₀ の結果は n₁ とは異なります．

```
def succ_n₀ : Preterm := App succ n₀

#eval succ_n₀
--> ((λ0.(λ1.(λ2.(1 ((0 1) 2))))) (λ0.(λ1.1)))
```

それでは項の簡約を定義してみます． ここでは簡単に定義できる並列最外簡約を実装します． ざっくりとした簡約ですが， これ以上細かく実装すると今回の記事の趣旨から逸脱してしまうのでこれで良しとします．

```
def subst' {α} : preterm (preterm α) → preterm α
| (var x) := x
| (lam f) := lam (λ x, subst' (f (var x))) -- terminates!
| (app m₁ m₂) := app (subst' m₁) (subst' m₂)

def pbeta' {α} : preterm (preterm α) → preterm α
| (var x) := x
| (lam f) := lam (λ x, pbeta' (f (var x)))
| (app (lam f) m) := subst' (f (subst' m))
| (app n m) := app (pbeta' n) (pbeta' m)

def pbeta : Preterm → Preterm :=     -- parallel outermost reduction
λ m α, pbeta' (m _)
```

ここで新しく現れた Lean の構文について説明しておきます．pbeta の定義の中に現れている _ はプレースホルダで，Lean の処理系が適切な項を推論して補完してくれます． ここでは preterm α という項が自動的に補完されます． また，subst' と pbeta' の定義の先頭に現れている {α} は {α : _} の略です．これは引数 α が暗黙の引数である (処理系により自動で適切な項が補完される) ことを指定しています．また，α 自体の型はプレースホルダ _ になっており，その後の α の使われ方から Type であることが自動的に推論されます．

pbeta の実装の解説は後回しにして，どのように動作するかを先に確認します．先ほどの succ_n₀ を簡約してみます．

```
#eval succ_n₀
--> ((λ0.(λ1.(λ2.(1 ((0 1) 2))))) (λ0.(λ1.1)))
#eval pbeta succ_n₀
--> (λ0.(λ1.(0 (((λ2.(λ3.3)) 0) 1))))
#eval pbeta (pbeta succ_n₀)
--> (λ0.(λ1.(0 ((λ2.2) 1))))
#eval pbeta (pbeta (pbeta succ_n₀))
--> (λ0.(λ1.(0 1)))
#eval pbeta (pbeta (pbeta (pbeta succ_n₀)))
--> (λ0.(λ1.(0 1)))
```

うまく動いていますね．これだけでも十分遊べますが，今回の趣旨から逸れてしまうのでほどほどにしておきます．

つぎに発散する項を作ってみます．

```
def ω : Preterm := λ α, lam (λ x, app (var x) (var x))
def Ω : Preterm := App ω ω
```

Ω は $(\lambda x.xx)(\lambda x.xx)$ という項を表しています．この項は簡約しても結果が自分自身となるクワインと呼ばれる種類の項の一つです．実際に簡約して結果が Ω となることを確認します．

```
#eval Ω
--> ((λ0.(0 0)) (λ0.(0 0)))
#eval pbeta Ω
--> ((λ0.(0 0)) (λ0.(0 0)))
```

たしかに Ω を簡約しても Ω になりました．

最後に，後回しにした pbeta や subst' のそれぞれの意味についてざっくりとだけ説明します．(ここは読み飛ばしてもよいです．) subst' は置換を完了させるための手続きです．(Parametric) HOAS では変数の置換は単なるメタレベルの適用で実現されます．(pbeta' の下から二行目を見るとわかる通り置換は Lean の関数適用です．) ただし適用

を行うには項の型が `preterm (preterm α)` という形をしている必要があります．`subst'` は適用を完了させ，型を `preterm α` に平坦化します．細かい点ですが，Lean は `subst'` が停止することを自動で検出してくれます． これは実は自明ではないので， 場合によっては手動で証明を書く必要があるかも知れません．`pbeta'` は $(\lambda x.M)N$ という形の項を探し，置換を行います． 先ほど説明した通り並列最外簡約です．`pbeta` は型を合わせるために `pbeta'` とは別に定義しました．

3. 単純型付きラムダ計算

2 節では型のないラムダ計算を扱いました． 一方で， ラムダ計算を論理学的に用いようとする場合，**型付きのラムダ計算**を考えることになります．特に， この記事の目標である高階論理では**単純型付きラムダ計算** (Simply-typed lambda calculus, STLC) が使われます． そのためにまず単純型付きラムダ計算について復習します．型付きラムダ計算の定義には Curry 流 (Curry-style) と Church 流 (Church-style) と呼ばれる二種類の流儀があります． [*22]どちらも基本的なアイデアや直観は同じですが技術的には異なるやり方で導入されます．Curry 流は 2 節で登場した型無しラムダ計算を元に定義されるので，まずは Curry 流での定義を紹介します． その後，Church 流での定義を紹介します． あとで詳しく話しますが，今回実装するのは Church 流の型付きラムダ計算です．

3.1. Curry 流の単純型付きラムダ計算 ★

Curry 流の単純型付きラムダ計算を定義します． まず， **基底型**の空でない集合 $\mathbb{B}$ を固定します．そして，**型**の集合 $\mathrm{Ty}(\mathbb{B})$ を以下の規則により帰納的に定義します．

$$\frac{\tau \in \mathbb{B}}{\tau \in \mathrm{Ty}(\mathbb{B})} \qquad \frac{\tau \in \mathrm{Ty}(\mathbb{B}) \qquad \sigma \in \mathrm{Ty}(\mathbb{B})}{\tau \to \sigma \in \mathrm{Ty}(\mathbb{B})}$$

$\tau \to \sigma$ の形の型を τ から σ への**関数型**と呼びます． **型環境** (type environment) は Var から $\mathrm{Ty}(\mathbb{B})$ への部分関数で定義域が有限であるものと定義します．型環境 Γ について， その定義域が $\{x_1, \cdots, x_n\}$ のとき，Γ を $x_1 : \Gamma(x_1), \cdots, x_n : \Gamma(x_n)$ と書きます． また，$\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \varnothing$ のとき，$\Gamma_1 \cup \Gamma_2$ を Γ_1, Γ_2 と書きます． すべての型環境

22 あるいは， **暗黙的型付け** (implicit typing) と**明示的型付け** (explicit typing) とも呼ばれます．[11] extrinsic と intrinsic という用語が使われることもあります．

からなる集合を $\mathscr{E}$ で表します．

三項関係 $\mathscr{W} \subseteq \mathscr{E} \times \Lambda \times \mathrm{Ty}(\mathbb{B})$ を定義します．$(\Gamma, M, \tau) \in \mathscr{W}$ であることを $\Gamma \vdash M : \tau$ と書きます． また， この形の命題を M についての**型判断** (type judgment) と呼びます．$\mathscr{W}$ は以下の規則により帰納的に定義されます．[*23]

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{x : \tau, \Gamma \vdash M : \sigma}{\Gamma \vdash \lambda x.M : \tau \to \sigma} \qquad \frac{\Gamma \vdash M : \tau \to \sigma \qquad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \sigma}$$

ラムダ項 M と型 τ について，適当な環境 Γ が存在して $\Gamma \vdash M : \tau$ であるとき，M は**型 τ を持つ** (has type τ) と言います． また，M が何らかの型を持つとき M は**型付け可能である** (typeable) と言います．このとき，以下の事実が成り立ちます．

定理 3

- $M =_\beta M'$ のとき，$\Gamma \vdash M : \tau \iff \Gamma \vdash M' : \tau$
- M が型付け可能ならば M は強正規化可能．

型付け可能なラムダ項には β 簡約と並んで有名な η **展開**という変換規則が定義されます．[*24] η 展開は以下の規則に加えて β 簡約のときと同様の追加規則により定義されます．

$$\frac{M \text{ has } \tau \to \sigma \qquad x \# M}{M \to_\eta \lambda x.Mx}$$

$\to_\beta \cup \to_\eta$ の反射推移対称閉包を $=_{\beta\eta}$ と書きます．$M =_{\beta\eta} N$ であるとき，M と N は $\beta\eta$ **同値である**と言います．$\beta\eta$ 同値性について定理 3 の前半の一般化が成り立ちます．

定理 4

- $M =_{\beta\eta} M'$ のとき，$\Gamma \vdash M : \tau \iff \Gamma \vdash M' : \tau$

---

23 定理 2 での α 同値な項を無視した帰納法の一般的な場合を用いており，定理 2 と同様の仮定を満たしていることが簡単に確認できます．

24 η 展開は型無しのラムダ項にも定義できますが，今回は型付きの場合しか考えません．

しかし，定理 3 の後半の一般化は成り立ちません．なぜなら，例えば $\lambda x.M$ や MN のような項について (それらが適当な型を持つなら) 以下のような無限簡約列が存在するからです．[12]

$$\lambda x.M \to_\eta \lambda y.(\lambda x.M)y \to_\beta \lambda y.[y/x]M =_\alpha \lambda x.M \qquad MN \to_\eta (\lambda x.Mx)N \to_\beta MN$$

この問題を回避するため η 展開の規則をやや変更し，新たな展開規則 $\to_{\eta^*}$ を導入します．[31] $\to_{\eta^*}$ を定めるために**穴付き文脈** (contexts with a hole)(以下単に文脈という) を定義します．文脈は以下の BNF により定義されます．

$$C ::= [\cdot] \mid \lambda x.C \mid C\,M \mid M\,C$$

文脈 C の中の唯一の $[\cdot]$ を**穴** (hole) と言い，C の穴を項 M で置き換えて得られる項を $C[M]$ と書きます．文脈 C が**関数的** (functional) とは，穴が適用されていること，つまり，ある文脈 C' と項 M が存在して $C = C'[[\cdot]M]$ が成り立つことを言います．このとき，関係 $\to_{\eta^*}$ を

- 項 M，変数 x，文脈 C が存在して，
- M は関数型 $\tau \to \sigma$ を持ち，
- M はラムダ抽象ではなく，
- x は M にたいして自由ではなく，
- C は関数的ではない

ときかつその時に限り $C[M] \to_{\eta^*} C[\lambda x.Mx]$ であると定義します．つまり，ラムダ抽象でなくかつ適用されていない項のみ η 展開を行えるという規則が $\to_{\eta^*}$ です．$\to_{\eta^*}$ にも $\to_\beta$ 同様簡約列などの概念が定義できます．[*25]特に，η^* 正規形を**長 η 形** (η-long form) と呼びます．η^* 展開と β **展開** (i.e., β 簡約の逆向き) を用いることで η 展開の規則を再現できます．加えて，定義より $\to_{\eta^*} \subseteq \to_\eta$ です．よって $M =_{\beta\eta^*} N$ と $M =_{\beta\eta} N$ は同値です．さらに η^* 展開に対しては定理 3 の後半の一般化が成立します．

定理 5

- M が型付け可能ならば M は $\beta\eta^*$ 強正規化可能．

---

25 $\to_{\eta^*}$ は項書き換え系をなさないので一部の用語はここだけの定義としておきます．

長 $\eta\beta$ 正規形を**標準形** (canonical form) と言います．定理 5 より任意の項に対してその標準形が存在し[*26]，しかもそれが計算可能です．以降では単に**正規化**といった場合標準形を計算する操作を指します．よって，任意の項 M,N に対して $M =_{\beta\eta} N$ は決定可能です．(M と N を正規化して結果の α 同値性を比較すればよいです．)

3.2. Curry 流と Church 流

単純型付きラムダ計算を PHOAS を用いて実現する方法は主に二種類あります．

ひとつはラムダ項自体は型なしのまま PHOAS でエンコードして，それが typeable であるという証明を持ち回るという方法です．依存型がある言語ならではの方法ですが，ここまで定義してきた型無しラムダ計算用のデータ型や関数をそのまま使えるという利点があります．一方で，毎回証明を書かなければならない上に PHOAS の項が typeable かどうかを表す述語とその証明に関連する補助関数を一式用意しなければなりません．これは今回の記事の動機に反します．

もうひとつの方法は，型検査をメタ言語に外注してしまうことです．それぞれの型付きラムダ計算の項の型を Lean レベルの型に埋め込むことで，Lean の型検査器に PHOAS で作成したラムダ項のオブジェクトレベルの (i.e., 今回定義しようとしているラムダ計算の) 型検査を行わせてしまいます．実はこのテクニックは Haskell でも Phantom type などを用いてよく使われるものです．この方式であれば，Lean の型検査がパスした時点でオブジェクト言語の型検査も終わっており，型検査に関する余計な手間をかけずに済みます．さらにこの方式には型推論がタダで手に入るという大きな利点があります．Lean の型検査器に検査を任せるということは，Lean の型検査器が持つ型推論器も使えるということです．Lean の持つ higher-order unifier がタダで使えるので，型推論の再実装のコストやそれに伴うバグに悩まされることもありません．よって，今回はこちらの方法を採ります．

しかし実は 3.1 節で導入した Curry 流の型付きラムダ計算は二つ目の方法を実装するのには用いることができません．もし一つ目の方法を採用するのであれば型無しラムダ計

26 この性質を canonicity と言います

算を用いるので Curry 流の STLC の技法をそのまま用いることができるのですが，二つ目の方法を採用する場合は各変数に対して型が固定されている必要があります．そのような型付きラムダ計算の流儀は Church 流と呼ばれています．そこで，次節では Church 流に基づく単純型付きラムダ計算の定義を紹介します．

3.3. Church 流の単純型付きラムダ計算 (その 1) ★

Church 流の単純型付きラムダ計算を定義します．型の集合 $\mathrm{Ty}(\mathbb{B})$ は Curry 流と同じ定義を用います．型無しラムダ計算と同様に変数の無限集合 $\mathrm{Var} = \{x, y, z, \cdots\}$ を固定します．加えて，変数に対して型を割り当てる写像 $\mathscr{A} : \mathrm{Var} \to \mathrm{Ty}(\mathbb{B})$ を固定します．[*27]まず，**擬項** (pseudoterm) の集合 $\Lambda_{\mathrm{Ty}(\mathbb{B})}$ を以下の規則で帰納的に定義します．

$$\frac{\mathscr{A}(x) = \tau}{x^\tau \in \Lambda_{\mathrm{Ty}(\mathbb{B})}} \qquad \frac{M \in \Lambda_{\mathrm{Ty}(\mathbb{B})} \qquad \mathscr{A}(x) = \tau}{\lambda x^\tau . M \in \Lambda_{\mathrm{Ty}(\mathbb{B})}} \qquad \frac{M \in \Lambda_{\mathrm{Ty}(\mathbb{B})} \qquad N \in \Lambda_{\mathrm{Ty}(\mathbb{B})}}{MN \in \Lambda_{\mathrm{Ty}(\mathbb{B})}}$$

擬項の**自由変数**や**束縛**の概念は型無しラムダ計算と同様に定義されます．型無しラムダ計算では α 同値な項を同一視するためにかなり苦労をしましたが，今回は暗黙のうちに α 同値な項は全て同一視することとします．

すべての型 τ に対して擬項の集合 $\Lambda_\tau \subseteq \Lambda_{\mathrm{Ty}(\mathbb{B})}$ を以下の (擬項の) 構造に関する帰納法により定めます．

$$\frac{}{x^\tau \in \Lambda_\tau} \qquad \frac{M \in \Lambda_\sigma}{\lambda x^\tau . M \in \Lambda_{\tau \to \sigma}} \qquad \frac{M \in \Lambda_{\tau \to \sigma} \qquad N \in \Lambda_\tau}{MN \in \Lambda_\sigma}$$

擬項 M が Λ_τ に属するとき M は τ **型を持つ** (has type τ) と言います．また擬項 M がなんらかの型 τ を持つとき M は**正しく型付けされている** (well-typed) と言い，そのような擬項を**項** (term) と呼びます．実は (擬) 項 M が型を持つときにはその型は一意に決まります．つまり，M が型 τ と σ を持つとき，$\tau = \sigma$ です．[*28]項の中に現れる変数 x^τ はしばしば x と略して書かれます．また，関数型を持つ変数をしばしば $f, g, \cdots$ というメタ変数で表します．

---

27 Curry 流で登場した型環境とは違い全域で定義された写像です．

28 この性質を**型付けの一意性** (uniqueness of typing) と呼びます．

Church 流のラムダ計算にも置換や β 簡約，η 展開を定義することができます． また，Curry 流の場合と同様の性質が成り立ちます．Church 流では標準形を帰納法を用いたより簡潔な方法で定義できます．各型 τ ごとに集合 $\mathrm{NF}_\tau \subseteq \Lambda_\tau$ と $\mathrm{NE}_\tau \subseteq \Lambda_\tau$ を相互再帰で定義します．

$$\frac{M \in \mathrm{NE}_\tau}{M \in \mathrm{NF}_\tau} \qquad \frac{M \in \mathrm{NF}_\sigma}{\lambda x^\tau.M \in \mathrm{NF}_{\tau\to\sigma}} \qquad \frac{}{x^\tau \in \mathrm{NE}_\tau} \qquad \frac{M \in \mathrm{NE}_{\tau\to\sigma} \qquad N \in \mathrm{NF}_\tau}{MN \in \mathrm{NE}_\sigma}$$

すると，NF_τ は τ 型を持つ標準形の集合と一致します． また，NE_τ の元を**中立形** (neutral form) と呼びます．

3.4. 単純型付きラムダ計算の実装

単純型付きラムダ計算を PHOAS を用いて実装します．今回用いる手法をここでは Typeful PHOAS と呼ぶことにします．また，2.3 節での型無しの PHOAS を Typeless PHOAS と呼ぶことにします． この節以降では基底型は一つだけしかないとします． その唯一の基底型は (高階論理の流儀で)ι で表されることが通例になっています．[*29]

まず，オブジェクト言語の型を表す型 `type` を定義します．

```
inductive type : Type
| base : type
| arrow : type → type → type
```

さらに，`type` 型を用いて項を表す型 `term` を定義します．3.1 節の Church 流の STLC の形式的な定義とは違い，擬項を定義せず直接項を定義します．`variable ν : type → Type` という文が登場していますが，これは単に `term` の引数を外に書いただけで，`inductive term (ν : type → Type) : type → Type` の略記です． 以降定義する関数に `ν` が登場した場合暗黙のうちに `(ν : type → Type)` という引数が追加されます．

```
variable ν : type → Type

inductive term : type → Type
| var : Π {t}, ν t → term t
```

29 individual の頭文字です．

```
| lam : Π {t₁ t₂}, (ν t₁ → term t₂) → term (arrow t₁ t₂)
| app : Π {t₁ t₂}, term (arrow t₁ t₂) → term t₁ → term t₂
```

Typeless PHOAS では単なる型変数 α だった部分が type を受け取って型を返す関数 ν に変化しています．[*30]各コンストラクタに現れる term という型の引数がうまく設定されているので正しく型がついた項しか作成できないようになっています．[*31]

Typeful PHOAS の場合でも Typeless PHOAS と同様に ν を全称量化すると各 t について t 型を持つ全ての閉項を表す型が得られます．

```
def Term (t : type) : Type 1 :=
Π ν, term ν t
```

それでは型付きのラムダ項をいくつか作ってみます．型無しラムダ計算の時にも試した Church encoding を試します．まず，自然数を表すラムダ項の型を定義します．

```
def nat : type := arrow (arrow base base) (arrow base base)
```

型無しラムダ計算のときと同じ例 (i.e., zero，succ，succ_zero) を記述してみます．実は，Lean の型推論の機能により，型無しラムダ計算の場合と全く同じコードで型付きラムダ計算の項が記述できます．唯一の違いは Term に引数が増えていることのみです．

```
def zero : Term nat :=
λ ν, lam (λ f, lam (λ x, var x))

def succ : Term (arrow nat nat) :=
```

30 3.1 節の定義とここでの term の定義を見比べると ν が使用されているあたりに違いがあります．この違いがなぜ起こるのかを説明するのは容易ではないですが，そのヒントになるかもしれない研究として HOAS の圏論的解釈を紹介しておきます．HOAS の圏論的解釈ではそれぞれの型が**前層** (presheaf) で解釈されます．すると ν は米田埋め込みとして解釈できます．より詳しい議論は [7, 8] を参照してください．

31 素の HOAS に型だけ導入することも可能です．今回採用している方式は HOAS を Parametric と Typeful の両方の面で拡張したものです．より詳しい議論は [10] を参照してください．

```
λ ν, lam (λ n, lam (λ f, lam (λ x, app (var f) (app (app (var n) (var f)) (v
ar x)))))

def App : Term (arrow t₁ t₂) → Term t₁ → Term t₂ :=
λ m₁ m₂ ν, app (m₁ ν) (m₂ ν)

def succ_zero : Term nat :=
App succ zero
```

ただしく項が作成できていることを確認するために (詳細は掲載しませんが)`Term` に対して `has_repr` のインスタンスを定義しました.

```
#eval zero
--> (λ0.(λ1.1)) : ((ι → ι) → (ι → ι))
#eval succ
--> (λ0.(λ1.(λ2.(1 ((0 1) 2))))) : (((ι → ι) → (ι → ι)) → ((ι → ι) → (ι → ι)))
#eval succ_zero
--> ((λ0.(λ1.(λ2.(1 ((0 1) 2))))) (λ0.(λ1.1))) : ((ι → ι) → (ι → ι))
```

うまく動いているようです. 一方で, オブジェクト言語 (i.e., STLC) で型がつかない項は Typeful PHOAS では表現できません.

```
-- // type error!
-- def ω : Term _ := λ ν, lam (λ x, app (var x) (var x))
```

3.5. 評価による正規化

この節では項の正規化を実装します. 特に, **評価による正規化** (Normalization by Evaluation, NbE) というテクニックを使って実装することで非常にコンパクトなコードで正規化を実装します. NbE についてはこの本の zeptometer による別の記事 (第 4 章) に詳細が掲載されているので, テクニックの概要の説明はそちらに譲ることにします. 代わりに, この記事では NbE を Typeful PHOAS でどのように実装するかに焦点を合わせます.

NbE を実装するためにはまずオブジェクト言語の項をメタ言語の項に変換する `eval` 手続きが必要です. `eval` は基底型の値は基底型の値に, 関数型の値は (メタ言語の) 関数型

の値に写します．それを表現するためにまず eval の結果が動く範囲を domain 型として定義します．domain 型は term 型と同様に ν に依存しているのでそれを量化した Domain 型も用意しておきます．

```
def domain : type → Type
| base := term ν base
| (arrow t₁ t₂) := domain t₁ → domain t₂

def Domain (t : type) : Type 1 :=
Π ν, domain ν t
```

domain 型を使えば eval は簡単に定義できます．subst と同じく ν をうまく実体化して実装します．

```
def eval' : Π {t : type}, term (domain ν) t → domain ν t
| _ (var x) := x
| _ (lam f) := λ x, eval' (f x)
| _ (app m₁ m₂) := (eval' m₁) (eval' m₂)

def eval : Term t → Domain t :=
λ m ν, eval' ν (m _)
```

`reify` と `reflect` はほぼ黒板通りに記述できます．[*32] [*33]

```
mutual def reify, reflect
with reify : Π {t : type}, domain ν t → term ν t
| base m := m
| (arrow t₁ t₂) f := lam (λ x, reify (f (reflect (var x))))
with reflect : Π {t : type}, term ν t → domain ν t
| base m := m
| (arrow t₁ t₂) f := λ x, reflect (app f (reify x))

def reify : Domain t → Term t :=
λ x ν, reify ν (x ν)
```

`eval` と `reify` が定義できたので， 正規化を行う関数 `normalize` は直ちに定義できます．

---

32 文献によっては `reify` の終域を標準形を表す型に，`reflect` の定義域を中立形を表す型にしている場合もあります．[14] その場合，`reify` の結果が標準形であることが自明になるというメリットがあります．

33 実際には， この定義は Lean version 3.4.2 ではコンパイルが通りません．`reify` と `reflect` が停止性することをコンパイラが自動で証明できないからです．そこで代わりに `reify` と `reflect` を同時に一つの再帰で定義します． こうすれば Lean はこれらが停止することを自動で発見してくれます．現行の Lean はこの手の自動証明にまだ改善の余地が多々あり，将来のバージョンの Lean では改善されているかもしれません．

```
def reify_reflect : Π (t : type), (domain ν t → term ν t) × (term ν t → domai
n ν t)
| base := ⟨ id, id ⟩
| (arrow t₁ t₂) :=
  let r₁ := reify_reflect t₁ in
  let r₂ := reify_reflect t₂ in
et reify (f : domain ν t₁ → domain ν t₂) := lam (λ x, r₂.1 (f (r₁.2 (var x)))) in
  let reflect (f : term ν (arrow t₁ t₂)) := λ x, r₂.2 (app f (r₁.1 x)) in
  ⟨reify, reflect⟩

def reify : Domain t → Term t :=
λ x ν, (reify_reflect ν t).1 (x ν)
```

```
def normalize : Term t → Term t :=
reify ∘ eval
```

`normalize` が正しく動くことを確認します．`zero` を `succ` した結果が正しく正規化されて Church encoding の形で出力されています．

```
#eval normalize zero
--> (λ0.(λ1.1)) : ((ι → ι) → (ι → ι))
#eval normalize (App succ zero)
--> (λ0.(λ1.(0 1))) : ((ι → ι) → (ι → ι))
#eval normalize (App succ (App succ zero))
--> (λ0.(λ1.(0 (0 1)))) : ((ι → ι) → (ι → ι))
#eval normalize (App succ (App succ (App succ zero)))
--> (λ0.(λ1.(0 (0 (0 1))))) : ((ι → ι) → (ι → ι))
```

`normalize` が β 簡約だけでなく η 展開を行うことも確認します．

```
def i : Term (arrow (arrow base base) (arrow base base)) :=
λ ν, lam (λ f, var f)

#eval i
--> (λ0.0) : ((ι → ι) → (ι → ι))
#eval normalize i
--> (λ0.(λ1.(0 1))) : ((ι → ι) → (ι → ι))
```

二つの項が $\beta\eta$ 同値であることを Lean で記号 ≈ を用いて記述できると便利です．そのために，型クラス `setoid` のインスタンスを作成します．[*34]

```
instance : setoid (Term t) :=
⟨inv_image eq normalize,
 inv_image.equivalence eq normalize eq.equivalence⟩
```

34 正確には記法 ≈ は型クラス `has_equiv` のインスタンスを定義すると使用できます．一方で型クラス `setoid` を持つ型は自動的に `has_equiv` を持つように (標準ライブラリで) 定義されています．

これで，`m n : Term t` について `m ≈ n` と書くと `normalize m = normalize n` を意味するようになります．

Lean で `m ≈ n` を証明するときは常に定義を展開した上で反射性を使うことが (メタ的な議論により) わかるのでそのような証明を自動化するためにタクティックを定義します．

```
meta def canonicity : tactic unit :=
`[ try { unfold has_equiv.equiv setoid.r inv_image }, try { reflexivity } ]
```

実際にこのタクティックを使用してみます．

```
def one : Term (arrow (arrow base base) (arrow base base)) :=
λ ν, lam (λ f, lam (λ x, app (var f) (var x)))

lemma zero_eqv_zero : zero ≈ zero :=
by canonicity

lemma succ_zero_eqv_one : App succ zero ≈ one :=
by canonicity
```

うまく動いているようです．

3.6. Church 流の単純型付きラムダ計算 (その 2) ★

ここまでの実装ではいずれも閉じたラムダ項のみを扱ってきました． それを開いた項に一般化するために，Church 流の単純型付きラムダ計算の拡張を紹介します． これにより，例えば 3.5 節で開発した正規形を計算する手続きを一般化して，開いた項の間の $\beta\eta$ 同値性を計算することができるようになります．

もう少し具体的に説明すると，ここまでで出てきた項の型 `Term t` を `Judgment Γ t` という型に拡張するのがこの節を含めた 3 節の残り (3.6 節，3.7 節，3.8 節) の狙いです．

$$\texttt{Term t} \quad \mapsto \quad \texttt{Judgment Γ t}$$

ただし，`Γ` は型のリストです． また， この拡張による理論面でのご利益としては，Church 流と Curry 流との関連について議論できるようになること， また， 直観主義命題論理と

型付きラムダ計算の間の Curry-Howard 同型対応と呼ばれる対応について議論できるようになることです.

固定された写像 $\mathscr{A} : \mathrm{Var} \to \mathrm{Ty}(\mathbb{B})$ の有限部分集合を**型環境** (type environment) と呼びます. 型環境全体の集合を $\mathscr{E}$ とします. 定義より, Church 流の型環境は Curry 流の型環境です. Church 流の型環境にたいしても Curry 流の型環境と同様の記法を用います. 三項関係 $\mathscr{W} \subseteq \mathscr{E} \times \Lambda_{\mathrm{Ty}(\mathbb{B})} \times \mathrm{Ty}(\mathbb{B})$ を以下の規則で帰納的に定義します. $(\Gamma, M, \tau) \in \mathscr{W}$ を $\Gamma \vdash M : \tau$ と書き, この形の命題を**型判断** (type judgment) と呼びます.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x^{\tau} : \tau} \qquad \frac{x : \tau, \Gamma \vdash M : \sigma}{\Gamma \vdash \lambda x^{\tau}.M : \tau \to \sigma} \qquad \frac{\Gamma \vdash M : \tau \to \sigma \qquad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \sigma}$$

3.3 節での定義とここでの定義の関連はつぎの通りです. M を任意の擬項とし, Γ を $\mathscr{A}$ の $\mathrm{fv}(M)$ への制限とします. すると M が τ 型を持つとき, 型判断 $\Gamma \vdash M : \tau$ が成り立ちます. 逆に, なんらかの Γ について型判断 $\Gamma \vdash M : \tau$ が成り立つとき, M は τ 型を持ちます.

また, Church 流に Curry 流と似たような型環境の概念が入ったことで両者を比較することができるようになります. 擬項を前項に送る忘却写像 $|\cdot| : \Lambda_{\mathrm{Ty}(\mathbb{B})} \to \Lambda$ を構造に関する帰納法で以下のように定義します.

$$\begin{aligned} |x^{\tau}| &:= x \\ |\lambda x^{\tau}.M| &:= \lambda x.|M| \\ |MN| &:= |M||N| \end{aligned}$$

すると, Church 流の型判断 $\Gamma \vdash M : \tau$ が成り立つとき, Curry 流の型判断 $\Gamma \vdash |M| : \tau$ が成り立ちます. 逆に, Curry 流の型判断 $\Gamma \vdash N : \tau$ が成り立つとき, $|M| = N$ となるような Church 流の項 M と型判断 $\Gamma \vdash M : \tau$ が成り立ちます.

型判断に対する様々な操作を定義するために $\mathrm{IPC}^{\to}$ という直観主義命題論理の断片を紹介します. 型付きラムダ計算と $\mathrm{IPC}^{\to}$ の自然演繹には著しい類似があり, 型判断についての様々な概念は論理学の用語を用いることで簡潔に説明されます.

$\mathrm{PVar} = \{P, Q, \cdots\}$ を**命題変数** (propositional variable) の集合とします．$\mathrm{IPC}^{\to}$ における**論理式** (formula)，あるいは単に**式**は以下の BNF で定義されます．

$$\varphi ::= P \mid \varphi \to \varphi$$

通常であればここで論理式の意味を決定するために論理式の**解釈** (interpretation) を定義しますが，本筋から逸れてしまうのでここでは定義しないことにします．代わりに，演繹体系のみを与えます．

$\mathrm{IPC}^{\to}$ の演繹体系として $\mathrm{IPC}^{\to}$ の自然演繹を導入します．ここからは証明そのものを数学的対象として扱うことに注意してください．[*35]

論理式の有限集合 Φ と論理式 φ の組 (Φ, φ) を**仮説的判断** (hypothetical judgment) あるいは単に**判断** (judgment) と呼びます．判断 (Φ, φ) をしばしば $\Phi \vdash \varphi$ と書きます．また，$\{\varphi_1, \cdots, \varphi_n\} \vdash \varphi$ をしばしば $\varphi_1, \cdots, \varphi_n \vdash \varphi$ と書きます．判断の左辺に並ぶ論理式を**仮定** (hypothesis)，右辺の論理式を**結論** (conclusion) と呼びます．判断の有限集合 Ξ と判断 $\mathscr{H}$ の組 $(\Xi, \mathscr{H})$ を**推論規則** (inference rule) あるいは単に**規則** (rule) と呼びます．規則 $(\{\mathscr{H}_1, \cdots, \mathscr{H}_n\}, \mathscr{H})$ をしばしば

$$\frac{\mathscr{H}_1 \qquad \cdots \qquad \mathscr{H}_n}{\mathscr{H}}$$

と書きます．P をなんらかの述語とするとき，規則の集合

$$\left\{ (\{\mathscr{H}_1, \cdots, \mathscr{H}_n\}, \mathscr{H}) \;\middle|\; P(\mathscr{H}_1, \cdots, \mathscr{H}_n, \mathscr{H}) \right\}$$

を**規則図式** (rule schema) と呼び，以下のように書きます．

$$\frac{\mathscr{H}_1 \qquad \cdots \qquad \mathscr{H}_n}{\mathscr{H}} P(\mathscr{H}_1, \cdots, \mathscr{H}_n, \mathscr{H})$$

上の形の規則図式について，$P(\mathscr{H}_1, \cdots, \mathscr{H}_n, \mathscr{H})$ をその**付帯条件** (side condition) と呼びます．規則図式 Hyp, I$\to$, E$\to$ を以下のように定めます．

---

35 たとえば，ここまでで登場した型判断は命題でしたが，今から扱う判断は構造です．

$$\frac{}{\Phi \vdash \varphi}\,\varphi \in \Phi \qquad \frac{\varphi, \Phi \vdash \psi}{\Phi \vdash \varphi \to \psi} \qquad \frac{\Phi \vdash \varphi \to \psi \qquad \Phi \vdash \varphi}{\Phi \vdash \psi}$$

Hyp, I→, E→ の元を順に**仮定** (hypothesis), → **導入** (→-introduction), → **除去** (→-elimination) と呼びます．判断の有限集合 Ξ と判断 $\mathscr{H}$ について，述語 $\Xi \vdash \mathscr{H}$[36]を次の規則[*37]によって帰納的に定義します．

$$\frac{(\Xi, \mathscr{H}) \in \mathrm{Hyp} \cup \mathrm{I}\to \cup \mathrm{E}\to}{\Xi \vdash \mathscr{H}}$$

$$\frac{\Xi_1 \vdash \mathscr{H}_1 \qquad \cdots \qquad \Xi_n \vdash \mathscr{H}_n \qquad \mathscr{H}_1, \cdots, \mathscr{H}_n \vdash \mathscr{H}}{\Xi_1 \cup \cdots \cup \Xi_n \vdash \mathscr{H}}$$

$\Xi \vdash \mathscr{H}$ が成り立つとき，規則 $(\Xi, \mathscr{H})$ は**導出可能である** (derivable) と言います．導出可能な規則を**派生規則** (derived rule) とも呼びます．$(\varnothing, \mathscr{H})$ が導出可能であるとき，$\mathscr{H}$ を**定理** (theorem) と呼びます．$\mathscr{H}$ が定理であるとき，$\mathscr{H}$ が**成り立つ** (holds) と言います．判断 $\mathscr{H}_1, \cdots, \mathscr{H}_n, \mathscr{H}$ について，全ての i について $\mathscr{H}_i$ が定理ならば $\mathscr{H}$ も定理であるとき，規則 $\left(\{\mathscr{H}_1, \cdots, \mathscr{H}_n\}, \mathscr{H}\right)$ は**許容可能である** (admissible) と言います．導出可能な規則は許容可能です．

許容可能な規則の代表例をいくつか挙げます．まず，以下の二つの規則は許容可能です．

$$\frac{\Phi \vdash \varphi \qquad \varphi, \Phi \vdash \psi}{\Phi \vdash \psi} \qquad \frac{\Phi \vdash \varphi}{\psi, \Phi \vdash \varphi}$$

この二つは順に**置換** (substitution)，**弱化** (weakening) という名前で呼ばれる規則です．実は，ここまでの話は判断の左辺を仮定の集合ではなく仮定のリストとしても成り立ちます．その場合，以下の二つの規則が許容可能になります．

$$\frac{\Phi \vdash \varphi}{\pi(\Phi) \vdash \varphi} \qquad \frac{\psi, \psi, \Phi \vdash \varphi}{\psi, \Phi \vdash \varphi}$$

ここで，$\pi(\Phi)$ は Φ の中身を並べ替えたものです．これら二つは順に**交換** (exchange)，**縮約** (contraction) と呼ばれる規則です．交換規則と縮約規則のおかげで，仮説的判断の左

36 判断を表す $\vdash$ とは異なります．

37 ここでの規則は単なる命題です．

辺を実質的に集合と見做すことができるため，リストとして定義しても集合として定義しても証明可能性は実質的には変わりません．以上の四つの規則のように特定の論理結合子について言及しない規則を**構造規則** (structural rule) と呼びます．通常，単に構造規則といえば上記の 4 つを指します．

また，$\mathrm{IPC}^{\to}$ では以下の二つの規則が許容可能です．

$$\frac{\psi, \Phi \vdash \varphi}{\Phi \vdash \psi \to \varphi} \qquad \frac{\Phi \vdash \psi \to \varphi}{\psi, \Phi \vdash \varphi}$$

一つ目は単なる $\to$ 導入です．ヒルベルト流からの類推で，上の一つ目の規則を**演繹定理** (deduction theorem)，二つ目の規則を**帰納定理** (resolution theorem) と呼ぶことがあります．これら二つは論理結合子 $\to$ の特徴付けになっています．

型付きラムダ計算と $\mathrm{IPC}^{\to}$ の間の関係について説明します．ここでは $\mathbb{B} = \mathrm{PVar}$ を仮定します．型環境をその像に送る関数を $|\cdot|$ とします．型判断 $\Gamma \vdash M : \tau$ が成り立つとき，$|\Gamma| \vdash \tau$ が成り立ちます．逆に，$\Phi \vdash \tau$ が成り立つとき，適当な型環境 Γ と項 M であって $|\Gamma| = \Phi$ であるものが存在して，$\Gamma \vdash M : \tau$ が成り立ちます．この事実は $\mathrm{IPC}^{\to}$ の Curry-Howard **同型対応**と呼ばれています．また，この議論から型付きラムダ計算においても型環境をリストとしてみなすことができ，かつ置換，弱化，交換，縮約，演繹定理，帰納定理に相当するものが成り立つことがわかります．以降の文章では，型判断を表すのにそれに対応する仮説的判断を用いることがあります．

3.7. 型判断の実装 (その 1)

前節 3.6 節の冒頭でも言及したように，Typeful PHOAS には開いた項がうまく扱えないという問題があります．本節 3.7 節と次節 3.8 節では前節 3.6 節で定義した型判断を用いてその問題を解決することを試みます．具体的には，3.4 節で定義した項に加えて型判断を実装し，項と型判断との間の相互変換を定義することで，開いた項をうまく扱うことを試みます．[*38]この相互変換について，本節 3.7 節と次節 3.8 節では異なる方針で実装を試みています．本節 3.7 節の方針はより素直なのですが (最後の最後で) 失敗します．一方，次節 3.8 節では代わりにややトリッキーな方針を採用して，相互変換を適切に定義し

38 この問題に対する他のアプローチとして文脈様相型を持つプログラミング言語 (例えば Beluga [18]) を用いる手法があります．

ます．ここからの話はそれなりにテクニカル (な割に実りが少ない) ので，前節 3.6 節の内容を実装したんだということだけ頭にいれておけば，本節 3.7 節と次節 3.8 節は読み飛ばして 4 節へ進んで構いません．

型判断を表す judgment₁ 型を定義します．ここでは型環境を型のリストとして表現します．

```
inductive judgment₁ : list type → type → Type
| var : Π {Γ t}, t ∈' Γ → judgment₁ Γ t
| lam : Π {Γ t₁ t₂}, judgment₁ (t₁ :: Γ) t₂ → judgment₁ Γ (arrow t₁ t₂)
| app : Π {Γ t₁ t₂}, judgment₁ Γ (arrow t₁ t₂) → judgment₁ Γ t₁ → judgment
₁ Γ t₂
```

ただし，t ∈' Γ は以下で定義される型です．[*39]

```
inductive mem : α → list α → Type -- proof relevant version
| here : Π {x l}, mem x (x :: l)
| there : Π {x l y}, mem x l → mem x (y :: l)

local infix ` ∈' `:50 := mem
```

型付きラムダ計算の構造規則を実装してみます．最終的に以下の補題を証明することが目標です．[*40]それぞれの型が構造規則にちょうど対応していることがわかります．

```
def xchg  : (Γ₁ ~ Γ₂) → judgment₁ Γ₁ t → judgment₁ Γ₂ t
def weak  : judgment₁ Γ t₁ → judgment₁ (t₂ :: Γ) t₁
def subst : judgment₁ (t₁ :: Γ) t₂ → judgment₁ Γ t₁ → judgment₁ Γ t₂
def contr : judgment₁ (t₁ :: t₁ :: Γ) t₂ → judgment₁ (t₁ :: Γ) t₂
```

---

39 Lean の標準ライブラリには ∈' によく似た ∈ があるのですが，Prop 型として定義されていて今回の用途には使いづらいので Type 型として再実装しました．これらの違いは簡単に言えば，x ∈ l は l の中のどこかに x が入っていることだけを表しそれがどの位置に入っているかという情報は含まないのに対し，x ∈' l は l の中の何番目の位置に x が入っているかの情報まで含みます．

40 補題と書いていますが，後々の都合上 proof relevant にしています．なので，lemma ではなく def を用いています．

ただし，Γ₁ ~ Γ₂ は Γ₁ と Γ₂ がリストとして並べ替えになっていることを表しています．[*41]

```
inductive perm : list α → list α → Type
| nil   : perm [] []
| skip  : Π {x : α} {l₁ l₂ : list α}, perm l₁ l₂ → perm (x :: l₁) (x :: l₂)
| swap  : Π {x y : α} {l : list α}, perm (y :: x :: l) (x :: y :: l)
| trans : Π {l₁ l₂ l₃ : list α}, perm l₁ l₂ → perm l₂ l₃ → perm l₁ l₃

local infix ~ := perm
```

さきに簡単なものから実装します．もし置換規則が証明できていれば，縮約規則はそれを用いて直ちに定義できます．

```
def contr : judgment₁ (t₁ :: t₁ :: Γ) t₂ → judgment₁ (t₁ :: Γ) t₂ :=
λ m, subst m (var here)
```

同様に，交換規則があれば弱化規則は直ちに証明できます．

```
def weak : Π {Γ t₁ t₂}, judgment₁ Γ t₁ → judgment₁ (t₂ :: Γ) t₁
| _ _ _ (var h) := var (there h)
| _ _ _ (lam m) := lam (xchg perm.swap (weak m))
| _ _ _ (app m₁ m₂) := app (weak m₁) (weak m₂)
```

この補題は，証明木で考えると以下のような証明木の間の変換を定義しています．

$$\left(\frac{\varphi_1, \Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \to \varphi_2}\right) \mapsto \left(\frac{\dfrac{\psi, \varphi_1, \Gamma \vdash \varphi_2}{\varphi_1, \psi, \Gamma \vdash \varphi_2}}{\psi, \Gamma \vdash \varphi_1 \to \varphi_2}\right)$$

41 perm も Lean で既に実装されたものがあるのですが，やはり Prop 型なので Type として再実装しました．

交換規則についても以下の補題を示すことで簡単に定義できます．[*42]

```
def mem_perm : Γ₁ ~ Γ₂ → t ∈' Γ₁ → t ∈' Γ₂
```

この補題は「リストの並び替えが所属関係を保つ」ことを主張しています．これを用いた交換規則の具体的な証明は以下のようになります．

```
def xchg : Π {Γ₁ Γ₂ t}, (Γ₁ ~ Γ₂) → judgment₁ Γ₁ t → judgment₁ Γ₂ t
| _ _ _ p (var h) := var (mem_perm p h)
| _ _ _ p (lam m) := lam (xchg (perm.skip p) m)
| _ _ _ p (app m₁ m₂) := app (xchg p m₁) (xchg p m₂)
```

さて，残る最後の規則である置換規則についても，以下のように交換規則を用いることで簡単に証明できると予想されます．

```
def subst : Π {Γ t₁ t₂}, judgment₁ (t₁ :: Γ) t₂ → judgment₁ Γ t₁ → judgmen
t₁ Γ t₂
| _ _ _ (var here) m := m
| _ _ _ (var (there h)) m := var h
| _ _ _ (app m₁ m₂) m := app (subst m₁ m) (subst m₂ m)
| _ _ _ (lam m₁) m := lam (subst (xchg perm.swap m₁) (weak m))
```

しかし残念ながら，実際には現行の Lean は上の定義が停止することを自動で証明できず，エラーが出てしまいます．そこで，やや本質的でないのですが，`subst` が停止することを証明するために型判断の**高さ (height)** という概念を導入します．

---

42 ちなみに，補題の証明は以下の通りです．

```
def mem_perm {t : α} : Π {Γ₁ Γ₂}, Γ₁ ~ Γ₂ → t ∈' Γ₁ → t ∈' Γ₂
| _ _ perm.nil h := h
| _ _ (perm.skip _) here := here
| _ _ (perm.skip p) (there h) := there (mem_perm p h)
| _ _ perm.swap here := there here
| _ _ perm.swap (there here) := here
| _ _ perm.swap (there (there h)) := there (there h)
| _ _ (perm.trans p₁ p₂) h := mem_perm p₂ (mem_perm p₁ h)
```

```
def heightof : Π {Γ t}, judgment₁ Γ t → ℕ
| _ _ (var h) := 1
| _ _ (lam m) := 1 + heightof m
| _ _ (app m₁ m₂) := 1 + max (heightof m₁) (heightof m₂)
```

そして，交換規則を用いても型判断の高さが変わらないことを示します．

```
lemma height_xchg_eq_height {p : Γ₁ ~ Γ₂} {m : judgment₁ Γ₁ t} : heightof (x
chg p m) = heightof m
```

この補題を用いて subst が停止することを証明したものが以下になります．Lean は subst を well-founded recursion により定義しようとします．その際に引数が減少していることを have を用いて明示しています．subst が実装するアルゴリズム自体は上のものと全く同じです．繰り返しますが，この問題はあくまで今回用いた Lean 処理系固有の問題で，別のバージョンや別の処理系なら最初の subst だけでコンパイルが通る可能性もあります．

```
def subst : Π {Γ t₁ t₂}, judgment₁ (t₁ :: Γ) t₂ → judgment₁ Γ t₁ → judgment
₁ Γ t₂
| _ _ _ (var here) m := m
| _ _ _ (var (there h)) m := var h
| _ _ _ (app m₁ m₂) m :=
  have heightof m₁ < heightof (app m₁ m₂),
    by unfold heightof; rw add_comm;
    from lt_add_of_le_of_pos (le_max_left _ _) zero_lt_one,
  have heightof m₂ < heightof (app m₁ m₂),
    by unfold heightof; rw add_comm;
    from lt_add_of_le_of_pos (le_max_right _ _) zero_lt_one,
  app (subst m₁ m) (subst m₂ m)
| _ _ _ (lam m₁) m :=
  have heightof (xchg perm.swap m₁) < heightof (lam m₁),
    by unfold heightof; rw height_xchg_eq_height;
    from lt_add_of_pos_of_le zero_lt_one (le_refl _),
  lam (subst (xchg perm.swap m₁) (weak m))
using_well_founded
{ rel_tac := λ _ _, `[exact ⟨_, measure_wf (λ v, heightof v.snd.snd.snd.fs
t)⟩] }
```

論理結合子 $\to$ の特徴付けである演繹定理と帰納定理を定義します．といっても，ここまでの準備のおかげで非常に簡単に定義できます．まず，演繹定理は単に $\to$ の導入則と同じだったので，それをそのまま用いるだけです．

```
def abs : judgment₁ (t₁ :: Γ) t₂ → judgment₁ Γ (arrow t₁ t₂) :=
lam
```

帰納定理も， 型合わせのために weak を用いることにだけ注意すればすぐに定義できます．

```
def antiabs : judgment₁ Γ (arrow t₁ t₂) → judgment₁ (t₁ :: Γ) t₂ :=
λ m, app (weak m) (var here)
```

項 term と型判断 judgment₁ の間の相互変換を定義します． 型判断の仮定が空の場合，項と型判断の間には直接の対応関係があります．そうでない場合は演繹定理を用いることで，型判断の仮定が空の場合に帰着させます．例えば，$\iota \to \iota, \iota \vdash \iota$ という型判断を項に変換したい場合， まず型判断を演繹定理により $\vdash \iota \to (\iota \to \iota) \to \iota$ に変換して， それを $\iota \to (\iota \to \iota) \to \iota$ という型の項に変換します．実際の定義は以下のようになります．この実装では env という型を経由している点が非自明です．

```
inductive env : list type → Type
| nil {} : env []
| step : Π {t Γ}, ν t → env Γ →  env (t :: Γ)

def to_term_var : Π {Γ t}, t ∈' Γ → env ν Γ → term ν t
| (_ :: Γ) _ here (env.step x _) := term.var x
| (_ :: Γ) _ (there h) (env.step _ Δ) := to_term_var h Δ

def to_term' : Π {Γ t}, judgment₁ Γ t → env ν Γ → term ν t
| _ _ (var h) Δ := to_term_var ν h Δ
| _ _ (lam m) Δ := term.lam (λ x, to_term' m (env.step x Δ))
| _ _ (app m₁ m₂) Δ := term.app (to_term' m₁ Δ) (to_term' m₂ Δ)

def to_term : judgment₁ [] t → Term t :=
λ m ν, to_term' ν m env.nil
```

逆に項を型判断に変換する関数は以下のような型をもつはずです.

```
def to_judgment₁ : Term t → judgment₁ [] t
```

しかし，残念ながらこのような関数を Lean で定義することは不可能だと思われます．まず, WF という名前の述語[*43]を以下のように定義します.

```
inductive wf : list (Σ t, ν₁ t × ν₂ t) → Π {t}, term ν₁ t → term ν₂ t → Type
| var : Π {Γ t} {x₁ : ν₁ t} {x₂ : ν₂ t},
  ⟨t, x₁, x₂⟩ ∈' Γ → wf Γ (var x₁) (var x₂)
| lam : Π {Γ t₁ t₂} {f₁ : ν₁ t₁ → term ν₁ t₂} {f₂ : ν₂ t₁ → term ν₂ t₂},
  (Π x₁ x₂, wf (⟨t₁, x₁, x₂⟩ :: Γ) (f₁ x₁) (f₂ x₂)) → wf Γ (lam f₁) (lam f₂)
| app : Π {Γ t₁ t₂} {m₁ : term ν₁ (arrow t₁ t₂)} {m₂ : term ν₂ (arrow t₁ t
₂)} {n₁ : term ν₁ t₁} {n₂ : term ν₂ t₁},
  wf Γ m₁ m₂ → wf Γ n₁ n₂ → wf Γ (app m₁ n₁) (app m₂ n₂)

def WF (t : type) (m : Term t) : Type 1 :=
Π ν₁ ν₂, wf [] (m ν₁) (m ν₂)
```

wf は logical relation や parametricity という名前でも知られる述語です．WF t m は全ての m が満たしていてほしい性質です． 実際， 様々な具体的な m : Term t に対して WF t m を (Lean の中で) 証明をすることができます．[*44]もし，全ての項に対して WF t m が成り立つことを表す以下の項を Lean の中で証明できれば，それを用いて to_judgment₁ を定義できます.

```
def term_wf : Π t m, WF t m := ???
```

term_wf を仮定した場合の to_judgment₁ の定義は以下のようになるでしょう.

```
def to_judgment₁_var : Π {Γ t} {x₁ : ν₁ t} {x₂ : ν₂ t}, (sigma.mk t (prod.
mk x₁ x₂)) ∈' Γ → t ∈' (list.map (λ x, sigma.fst x) Γ)
| _ _ _ _ here := here
```

43 ここでは命題ではなく型にしています.

44 メタ的な議論を用いれば ∀m : Term t.WF t m is inhabited が証明できると予想されています．[9]

```
| _ _ _ _ (there h) := there (to_judgment₁_var h)

def to_judgment₁' : Π {Γ t} {m₁ : term (λ x, unit) t} {m₂ : term (λ x, uni
t) t}, wf Γ m₁ m₂ → judgment₁ (list.map (λ x, sigma.fst x) Γ) t
| _ _ _ _ (wf.var h) := var (to_judgment₁_var h)
| _ _ _ _ (wf.lam f) := lam (to_judgment₁' (f () ()))
| _ _ _ _ (wf.app m₁ m₂) := app (to_judgment₁' m₁) (to_judgment₁' m₂)

def to_judgment₁ : Term t → judgment₁ [] t :=
λ m, to_judgment₁' (term_wf _ m _ _)
```

しかし，term_wf は定義できないはずです． ちゃんと確認したわけではないのですが，[15] の手法を適応すれば，term_wf が定義できないことが示せるはずです．

3.8. 型判断の実装 (その 2)

3.7 節では型判断を実装するのに項との変換がうまく定義できませんでした．この節では別のエンコーディングを用いて型判断を実装して， 項との変換が正しく動くことを確認します．

ν を全称量化することで自由変数を持たない項を表せたように，関数型をうまく使うことで自由変数を 1 つだけもつ項の型も定義することができます．

```
-- // term without free variables
def Judgment0 (t : type) := Π ν, term ν t
-- // term with one free variable
def Judgment1 (t₁ t₂ : type) := Π ν, ν t₁ → term ν t₂
```

このアイデアをうまく用いて型判断を表す型 judgment₂ を以下のように定義できます．

```
def judgment₂ : list type → type → Type :=
λ Γ t, list.foldr (λ t α, ν t → α) (term ν t) Γ

def Judgment₂ (Γ : list type) (t : type) : Type 1 := -- Type 1
Π ν, judgment₂ ν Γ t
```

たとえば，ι と $\iota \to \iota$ 型の二つの自由変数を持つ ι 型の項は以下のような型で表されま

す.

```
#reduce Judgment₂ [base, arrow base base] base
--> Π ν, ν base → ν (arrow base base) → term ν base
```

また，型判断 $x_1 : \iota \to \iota, x_2 : \iota \vdash \lambda y^{\iota}.x_2 : \iota \to \iota$ は以下のように term のコンストラクタを用いて定義されます.

```
def ex2 : Judgment₂ [arrow base base, base] (arrow base base) :=
λ ν, λ x₁ x₂, lam (λ y, var x₂)
```

型判断の規則を定義する前に to_term と to_judgment₂ が定義できることを確認します. じつはこれは非常に簡単です. Judgment₂ の定義より，Judgment₂ [] t は Term t と全く同じ型です. よって，to_term も to_judgment₂ も単なる恒等関数として定義すればよいです.

```
def to_term : Judgment₂ [] t → Term t :=
id

def to_judgment₂ : Term t → Judgment₂ [] t :=
id
```

型判断の規則を定義します. 最も特徴的なのは弱化です. 簡潔に定義できます.

```
def weak : Judgment₂ Γ t₂ → Judgment₂ (t₁ :: Γ) t₂ :=
λ m ν x, m ν
```

この定義を見ると，オブジェクトレベルの弱化を行うためにメタレベルの弱化を行なっていることがわかります. 置換の定義も素直です. ここで 2 節で定義した subst' をそのまま用いています.

```
def subst'' : Π {Γ}, judgment₂ (term ν) (t₁ :: Γ) t₂ → judgment₂ ν Γ t₁ → ju
dgment₂ ν Γ t₂
| [] m₁ m₂ := subst' ν (m₁ m₂)
```

```
| (t :: Γ) f m := λ x, subst'' (λ x', f x' (var x)) (m x)

def subst : Judgment₂ (t₁ :: Γ) t₂ → Judgment₂ Γ t₁ → Judgment₂ Γ t₂ :=
λ m₁ m₂ ν, subst'' ν (m₁ _) (m₂ ν)
```

型判断の定義に現れる最も基本的な三つの規則も素直に定義されます. [*45]

```
def var : t ∈' Γ → Judgment₂ Γ t
def lam : Judgment₂ (t₁ :: Γ) t₂ → Judgment₂ Γ (arrow t₁ t₂)
def app : Judgment₂ Γ (arrow t₁ t₂) → Judgment₂ Γ t₁ → Judgment₂ Γ t₂
```

縮約は 3.7 節と全く同じコードで定義できます.

```
def contr : Judgment₂ (t₁ :: t₁ :: Γ) t₂ → Judgment₂ (t₁ :: Γ) t₂ :=
λ m, subst m (var here)
```

交換も (ここでは詳細は省略しますが) 素直に実装できます.

```
def xchg : Π {Γ₁ Γ₂ t}, (Γ₁ ~ Γ₂) → judgment₂ Γ₁ t → judgment₂ Γ₂ t
```

演繹定理と帰納定理も同様に適切に定義することで実装できます.

```
def abs : judgment₂ (t₁ :: Γ) t₂ → judgment₂ Γ (arrow t₁ t₂)
def antiabs : judgment₂ Γ (arrow t₁ t₂) → judgment₂ (t₁ :: Γ) t₂
```

abs が定義できたことで以下のように型判断の間 (つまり開いた項の間) の $\beta\eta$ 同値性が計算できるようになります.

---

45 たとえば app は以下のようなコードになります.

```
def app' : Π {Γ}, judgment₂ ν Γ (arrow t₁ t₂) → judgment₂ ν Γ t₁ → judgment₂ ν Γ t
₂
| [] m₁ m₂ := app m₁ m₂
| (t :: Γ) f m := λ x, app' (f x) (m x)

def app : Judgment₂ Γ (arrow t₁ t₂) → Judgment₂ Γ t₁ → Judgment₂ Γ t₂ :=
λ m₁ m₂ ν, app' ν (m₁ ν) (m₂ ν)
```

```
instance : setoid (Judgment₂ Γ t) :=
(inv_image eq (normalize ∘ to_term ∘ judgment₂.abs'),
 inv_image.equivalence eq (normalize ∘ to_term ∘ judgment₂.abs') eq.equivale
nce⟩
```

ただし，`abs'` は型環境が空になるまで `abs` を繰り返すような関数です．

4. 高階論理

4.1. 高階論理の意味論 ★

高階論理 (Higher-order logic, HOL) を導入します．高階論理と呼ばれるものには様々な変種があり確立された唯一の定義があるわけではありません．例えば，Church の Simple Type Theory (STT)[26] や Coquand と Huet の Calculus of Constructions (CoC)[25] が有名です．この節では高階論理のなかでも特に小さくわかりやすい，等号に基づく定義を紹介します．

高階論理の構文論を定義します．高階論理は Church 流の型付きラムダ計算を拡張したような構文を持ちます．ただし，基底型や関数型に加えて o で表される型を持ちます．つまり，高階論理の型の集合は以下の BNF で表せます．

$$\tau ::= \iota \mid o \mid \tau \to \tau$$

ただし，ι は $\mathbb{B}$ の元を表すとします．一階述語論理とのアナロジーで言えば，ι が項が動く領域で，o が命題が動く領域です．o 型を持つ変数をしばしば $p, q, \cdots$ というメタ変数で表します．高階論理の**言語** (language) あるいは**シグニチャ** (signature) とは型で添字づけられた集合 $\left(\Sigma_\tau\right)_{\tau \in \mathrm{Ty}(\mathbb{B})}$ のこととします．以下，言語 Σ を固定します．擬項の定義 $\Lambda_{\mathrm{Ty}(\mathbb{B})}$ を以下の規則で拡張します．

$$\frac{c \in \Sigma_\tau}{c^\tau \in \Lambda_{\mathrm{Ty}(\mathbb{B})}} \qquad \frac{M \in \Lambda_{\mathrm{Ty}(\mathbb{B})} \qquad N \in \Lambda_{\mathrm{Ty}(\mathbb{B})}}{M \doteq N \in \Lambda_{\mathrm{Ty}(\mathbb{B})}}$$

つまり，高階論理の擬項の集合は以下の BNF で表せます．

$$M ::= x^\tau \mid c^\tau \mid \lambda x^\tau . M \mid MM \mid M \doteq M$$

項の定義 Λ_τ は以下の規則で拡張されます.

$$\frac{c \in \Sigma_\tau}{c^\tau \in \Lambda_\tau} \qquad\qquad \frac{M \in \Lambda_\tau \qquad N \in \Lambda_\tau}{M \doteq N \in \Lambda_o}$$

Σ_τ の要素であるような項を**定数** (constant) と呼びます. β 簡約や η 展開は定数や $\doteq$ に対して congruent に振る舞うとします. o 型の項を**論理式** (formula) と呼びます. 閉じた論理式を**閉論理式** (closed formula) あるいは**文** (sentense) と呼びます. 言語 Σ と Σ 上の文の集合 E の組 (Σ, E) を**理論** (theory) と呼びます. E の要素を**公理** (axiom) と呼ぶことがあります.

高階論理の意味論を定義します. 高階論理には**集合論的意味論** (set-theoretic semantics), Henkin **意味論** (Henkin semantics), **トポス意味論** (topos semantics) など複数の意味論があります. ここで紹介するのはもっとも標準的な意味論である集合論的意味論です. [*46] Ω を 0 と 1 からなる二点集合とします. この記事では 0 を真, 1 を偽として扱います. 言語 Σ の**構造** (structure) とは集合と写像の組の族 $\mathscr{M} = \left(D_\tau, \mathscr{M}_\tau : \Sigma_\tau \to D_\tau\right)_{\tau \in \mathrm{Ty}(\mathbb{B})}$ であって,

$$D_o = \Omega$$
$$D_{\tau\to\sigma} = \left\{f : D_\tau \to D_\sigma\right\}$$

を満たすものとします. [*47]構造 $\mathscr{M}$ についての**付値** (valuation) とは各変数 $x \in \mathrm{Var}$ に対して $D_{\mathscr{A}(x)}$ の要素を割り当てる写像です. 構造 $\mathscr{M}$ による項の**解釈** (interpretation) を型 τ と付値 θ を添字にとる写像 $[\![\cdot]\!]^{\mathscr{M}}_{\tau,\theta} : \Lambda_\tau \to D_\tau$ として以下の帰納法で定めます.

---

46 集合論的意味論は素朴で直観的なのですが, 意味論として期待される性質を満たさないことが知られています. 現代的にはトポス意味論やその一般化を用いることが良いとされているようです. 例えば, Bauer による投稿 [16] を参照してください. トポス意味論への入門記事としては [29] が大変わかりやすいです.

47 全ての基底型 ι に対して D_ι を決めれば残りの D_τ は自動的に決まります.

$$
\begin{aligned}
[\![x^\tau]\!]^{\mathscr{M}}_{\tau,\theta} &:= \theta(x) \\
[\![c^\tau]\!]^{\mathscr{M}}_{\tau,\theta} &:= \mathscr{M}_\tau(c) \\
[\![\lambda x^\tau.M]\!]^{\mathscr{M}}_{\tau\to\sigma,\theta}(v) &:= [\![M]\!]^{\mathscr{M}}_{\sigma,[v/x]\theta} \\
[\![MN]\!]^{\mathscr{M}}_{\sigma,\theta} &:= [\![M]\!]^{\mathscr{M}}_{\tau\to\sigma,\theta}\Big([\![N]\!]^{\mathscr{M}}_{\tau,\theta}\Big) \\
[\![M \doteq N]\!]^{\mathscr{M}}_{o,\theta} &:= \begin{cases} 0 & ([\![M]\!]^{\mathscr{M}}_{\tau,\theta} = [\![N]\!]^{\mathscr{M}}_{\tau,\theta}) \\ 1 & \text{otherwise} \end{cases}
\end{aligned}
$$

論理式 φ について $[\![\varphi]\!]^{\mathscr{M}}_{o,\theta} = 0$ が成り立つとき，$\mathscr{M},\theta \vDash \varphi$ と書きます．また，任意の付値 θ について $\mathscr{M},\theta \vDash \varphi$ であるとき，$\mathscr{M} \vDash \varphi$ と書きます．任意の構造 $\mathscr{M}$ について $\mathscr{M} \vDash \varphi$ であるとき，$\vDash \varphi$ と書き，φ は**恒真である** (valid) と言います． ある構造 $\mathscr{M}$ と付値 θ が存在して $\mathscr{M},\theta \vDash \varphi$ となるとき，φ は**充足可能である** (satisfiable) と言います．φ_1 と φ_2 を論理式とします．任意の構造 $\mathscr{M}$ と任意の付値 θ について $[\![\varphi_1]\!]^{\mathscr{M}}_{o,\theta} = [\![\varphi_2]\!]^{\mathscr{M}}_{o,\theta}$ となるとき，φ_1 と φ_2 は**論理同値である** (logically equivalent) と言い，$\varphi_1 \simeq \varphi_2$ と書きます． 文の集合 E について，任意の E の要素 A が $\mathscr{M} \vDash A$ を満たすとき，$\mathscr{M}$ を E の**モデル** (model) と呼びます． また， 理論 $T = (\Sigma, E)$ の**モデル**とは E のモデルのことです． 論理式 φ と文の集合 E について， 任意の E のモデル $\mathscr{M}$ が $\mathscr{M} \vDash \varphi$ を満たすとき，$E \vDash \varphi$ と書き，φ を E の**論理的帰結** (logical consequence) と呼びます．

高階論理の表現能力について簡単に触れます． 高階論理が持つ論理式の構成子は $\doteq$ だけです．これでは一見，高階論理が非常に表現力の弱い論理に見えます．[*48]しかし，実はラムダ抽象をうまく用いることで通常用いられる種々の論理結合子を高階論理の中で定義することができます．たとえば，論理式 φ_1 と φ_2 に対して以下のような項を考えます．

$$
\Big(\lambda f^{o\to o\to o}.f\big(f \doteq f\big)\big(f \doteq f\big)\Big) \doteq \Big(\lambda f^{o\to o\to o}.f\,\varphi_1\varphi_2\Big)
$$

この項は o 型なので， 論理式です． この論理式を $\varphi_1 \spadesuit \varphi_2$ と書くことにします． する

48 たとえば，論理結合子が $\wedge$ しかない命題論理を考えると，それはかなり表現力の弱い論理であると言えるでしょう．

と，以下が成り立ちます．

補題 6　任意の構造 $\mathscr{M}$ と付値 θ について，$\mathscr{M},\theta \vDash \varphi_1 \spadesuit \varphi_2$ であるとき，かつそのときに限り $\mathscr{M},\theta \vDash \varphi_1$ かつ $\mathscr{M},\theta \vDash \varphi_2$

よって，上で定義した $\spadesuit$ は通常の論理で言う論理積 (conjunction) と見なせます．これと同じことがより一般に直観主義論理の基本的な論理結合子[*49]について成り立ちます．[28] 論理結合子 $\top$，$\wedge$，$\to$，$\forall$ を以下のように定義します．

$$\begin{aligned}
\top &\coloneqq \left(\lambda p^o.p\right) \doteq \left(\lambda p^o.p\right) \\
\varphi_1 \wedge \varphi_2 &\coloneqq \left(\lambda f^{o\to o\to o}.f\,\top\top\right) \doteq \left(\lambda f^{o\to o\to o}.f\,\varphi_1\varphi_2\right) \\
\varphi_1 \to \varphi_2 &\coloneqq \left(\varphi_1 \wedge \varphi_2\right) \doteq \varphi_1 \\
\forall x^\tau.\varphi &\coloneqq \left(\lambda x^\tau.\varphi\right) \doteq \left(\lambda x^\tau.\top\right)
\end{aligned}$$

すると，以下の補題が成り立ちます．

補題 7　任意の構造 $\mathscr{M}$ と付値 θ について

- $\mathscr{M},\theta \vDash \top$
- $\mathscr{M},\theta \vDash \varphi_1 \wedge \varphi_2$ であるとき，かつそのときに限り，$\mathscr{M},\theta \vDash \varphi_1$ かつ $\mathscr{M},\theta \vDash \varphi_2$
- $\mathscr{M},\theta \vDash \varphi_1 \to \varphi_2$ であるとき，かつそのときに限り，$\mathscr{M},\theta \vDash \varphi_1$ ならば $\mathscr{M},\theta \vDash \varphi_2$
- $\mathscr{M},\theta \vDash \forall x^\tau.\varphi$ であるとき，かつそのときに限り，任意の元 $a \in D_\tau$ について $\mathscr{M},[a/x]\theta \vDash \varphi$

が成り立つ．

以上の論理結合子を用いることで，他の様々な論理結合子も定義することができます．

---

49 線形論理でいうところの negative な結合子．あるいは，圏論的意味論で右随伴によって意味が定められる結合子．

$$
\begin{aligned}
\bot &:= \forall p^o.p \\
\neg\varphi &:= \varphi \to \bot \\
\varphi_1 \leftrightarrow \varphi_2 &:= \Big(\varphi_1 \to \varphi_2\Big) \wedge \Big(\varphi_2 \to \varphi_1\Big) \\
\varphi_1 \vee \varphi_2 &:= \forall p^o.\Big(\Big(\varphi_1 \to p\Big) \wedge \Big(\varphi_2 \to p\Big)\Big) \to p \\
\exists x^\tau.\varphi &:= \forall p^o.\big(\forall x^\tau.\varphi \to p\big) \to p \\
\exists! x^\tau.\varphi &:= \exists x^\tau.\varphi \wedge \forall y^\tau.\big[y/x\big]\varphi \to x \doteq y
\end{aligned}
$$

ここで命題の上の量化 $\forall p^o.\varphi$ を多用していますが， このテクニックは Girard の System F(二階命題論理) で論理結合子を定義する際に使われるテクニックと同じものです． 一連の議論により，高階論理が論理として十分強力な表現力を持つことがわかります．[*50]

さて，最後に蛇足として，高階論理の他の形式化について述べておきます．(なので，気になる方以外は読み飛ばして大丈夫です．) 今回の定義では組み込みの述語は $\doteq$ だけであり， 他の論理結合子はすべて $\doteq$ を用いて定義される派生物であるという立場をとりました． しかし，これが唯一のやり方というわけではありません．最も単純なやり方として，初めから全ての論理結合子とそれに関連する推論規則を組み込みとして定義するというのも考えることができます． その場合は組み込みの論理結合子とここまで見てきたような $\doteq$ で定義した論理結合子が論理的に等価になるので， 体系としては冗長性を含んでいるということになります．別のやり方として，$\doteq$ を組み込みとして持たない高階論理の上で $\doteq$ を派生物として定義することもできます．中でも，$\to$ と $\forall$ だけを組み込みとして持ち，他の論理結合子をそれらを用いて定義する方法がよく知られています．[30] すでに $\doteq$ が単体でも十分な表現力を持っていることがわかっているので，$\to$ と $\forall$ を用いて $\doteq$ を表現できれば全ての論理結合子が表現できることになります． 具体的には以下のように $\doteq$ を定義します．

$$
M_1 \doteq M_2 := \forall P^{\tau\to o}.PM_1 \to PM_2
$$

---

50 個人的には，等号 ($\doteq$) で全ての論理結合子が表現できることは本質的ではなく，たまたまそうなっているだけな気がしています． 等号だけあればよいという事実は技術的におもしろいですし実際今回の記事のように実装を目的としたケースでも役に立つのは間違いないですが，私自身はこの事実に特に深遠な理由があるとは考えていません．

この $\doteq$ は Leibniz equality と呼ばれる等号の定義です．ややびっくりするような定義ですが，これが実際に等号としての機能を果たすことはすぐにわかります．

補題 8　上で定義した $\doteq$ について，$\mathscr{M}, \theta \vDash M_1 \doteq M_2$ であるとき，かつそのときに限り，$[\![M_1]\!]^{\mathscr{M}}_{\tau,\theta} = [\![M_2]\!]^{\mathscr{M}}_{\tau,\theta}$.

証明　$\mathscr{M}, \theta \vDash M_1 \doteq M_2$ ならば $[\![M_1]\!]^{\mathscr{M}}_{\tau,\theta} = [\![M_2]\!]^{\mathscr{M}}_{\tau,\theta}$ を示す．(逆は明らか．) 定義より $\mathscr{M}, \theta \vDash \forall P^{\tau\to o}. P\, M_1 \to P\, M_2$ なので，任意の関数 $P : D_\tau \to \Omega$ について $P\left([\![M_1]\!]^{\mathscr{M}}_{\tau,\theta}\right) = 0$ ならば $P\left([\![M_2]\!]^{\mathscr{M}}_{\tau,\theta}\right) = 0$ である．ここで，P として，

$$P(x) = \begin{cases} 0 & x = [\![M_1]\!]^{\mathscr{M}}_{\tau,\theta} \text{のとき} \\ 1 & \text{そうでないとき} \end{cases}$$

をとると，$P\left([\![M_1]\!]^{\mathscr{M}}_{\tau,\theta}\right) = 0$ なので，$P\left([\![M_2]\!]^{\mathscr{M}}_{\tau,\theta}\right) = 0$ である．$P(x)$ が 0 を値にとるのは $x = [\![M_1]\!]^{\mathscr{M}}_{\tau,\theta}$ のときだけなので，$[\![M_1]\!]^{\mathscr{M}}_{\tau,\theta} = [\![M_2]\!]^{\mathscr{M}}_{\tau,\theta}$ である．■

このように $\doteq$ を基本とせずとも等価な表現能力を持つ高階論理を定義することができます．[*51]

4.2. 高階論理の演繹体系 ★

本節では高階論理の証明体系を与えます．証明体系の与え方には様々な流儀がありますが，この記事では (私の個人的な好みによって) 自然演繹を採用します．

本題に移る前に，ひとつ注意を述べます．前節 4.1 節で導入した高階論理は**古典高階論理** (Classical HOL) と呼ばれる種類の高階論理です．一方で，本節 4.2 節で定義する高階論理の証明体系は**直観主義高階論理** (Intuitionistic HOL) に対応する証明体系です．古典公開論理と直観主義高階論理はアイデアはほぼ同じですが直観主義高階論理の方が意味論の定義が大変なので先に古典論理を説明しました．証明論的には両者の違いはそこまでや

51 ただし，高階論理の演繹体系を考える場合 (少なくともこの記事の範囲では)$\doteq$ を基本とする形式のほうが都合がよいです．

やこしいものではなく，古典高階論理の演繹体系は直観主義高階論理の演繹体系を通常のやり方で古典にする (例えば，排中律を仮定する) だけで得られます．[17] 本節で導入する演繹体系は HOL Light[19] のコア言語から多相型を除いたもの，あるいは**初等トポスの内部言語** [27] を冪対象でなく指数対象で形式化したものと考えても良いです．

それでは，高階論理の自然演繹を導入します．高階論理の自然演繹の判断は $\Phi \vdash_\Gamma \varphi$ という形です．ただしこのとき，Γ は型環境，φ は論理式，$\Phi = \left\{\varphi_1, \cdots, \varphi_n\right\}$ は論理式の有限集合で，各論理式について以下の型判断が成り立つとします．

$$\Gamma \vdash \varphi : o \qquad \Gamma \vdash \varphi_1 : o \qquad \cdots \qquad \Gamma \vdash \varphi_n : o$$

高階論理の導出は以下の三つの規則図式を基に定義されます．

$$\frac{}{\Phi \vdash_\Gamma \varphi}\, \varphi \in \Phi$$

$$\frac{}{\Phi \vdash_\Gamma M \doteq N}\, M =_{\beta\eta} N \qquad \frac{\Phi \vdash_\Gamma M_1 \doteq M_2 \qquad \Phi \vdash_\Gamma \left[M_1/x\right]N}{\Phi \vdash_\Gamma \left[M_2/x\right]N}$$

これらの図式に当てはまる規則を順に**仮定** (hypothesis)，$\doteq$ **導入** ($\doteq$-introduction)，$\doteq$ **除去** ($\doteq$-elimination) と呼びます．通常，上記の三つの規則図式に加えて以下の二つの規則図式を加えます．

$$\frac{\varphi_1, \Phi \vdash_\Gamma \varphi_2 \qquad \varphi_2, \Phi \vdash_\Gamma \varphi_1}{\Phi \vdash_\Gamma \varphi_1 \doteq \varphi_2} \qquad \frac{\Phi \vdash_{\{x:\tau\} \cup \Gamma} M \doteq N}{\Phi \vdash_\Gamma (\lambda x^\tau.M) \doteq (\lambda x^\tau.N)}\, x \# \Phi$$

これら二つの規則図式はそれぞれ**命題外延性** (propositional extensionality)，**関数外延性** (functional extensionality) と呼ばれます．関数外延性の規則は ξ **規則** (ξ rule) とも呼ばれます．

この体系では，自然演繹に通常要請される性質が問題なく成立します．たとえば，証明の構造規則として弱化と置換の規則が許容可能になります．

補題 9　以下の規則は許容可能である．

$$\frac{\Phi \vdash_\Gamma \varphi}{\psi, \Phi \vdash_\Gamma \varphi} \qquad \frac{\Phi \vdash_\Gamma \psi \qquad \psi, \Phi \vdash_\Gamma \varphi}{\Phi \vdash_\Gamma \varphi}$$

また，仮定についてのみならず型環境についても期待される性質が成り立ちます．

補題 10　以下の規則は許容可能である．

$$\frac{\Phi \vdash_\Gamma \varphi}{\Phi \vdash_{\{x:\mathscr{A}(x)\}\cup\Gamma} \varphi} \qquad \frac{\Gamma \vdash M : \tau \qquad \Phi \vdash_{\{x:\tau\}\cup\Gamma} \varphi}{[M/x]\Phi \vdash_\Gamma [M/x]\varphi}$$

$\mathrm{IPC}^{\to}$ の場合と同様の理由で高階論理の判断の左辺は集合ではなく (有限の) リストとして定義しても証明可能性は変わりません．その場合，交換規則と縮約規則が許容可能になります．同様に，型環境をリストとして定義することも可能です．

この演繹体系は集合論的意味論に対して (強い) 健全性を持ちます．

定理 11　$E \vdash_\Gamma \varphi$ のとき $E \vDash \varphi$ が成り立つ．

この演繹体系での基本的な推論の例をいくつかあげてみます．例えば，4.1 節で導入した基本的な論理結合子について，その導入則と除去則が許容可能になります．

$$\frac{}{\Phi \vdash_\Gamma \top} \qquad \frac{\Phi \vdash_\Gamma \varphi_1 \qquad \Phi \vdash_\Gamma \varphi_2}{\Phi \vdash_\Gamma \varphi_1 \wedge \varphi_2} \qquad \frac{\Phi \vdash_\Gamma \varphi_1 \wedge \varphi_2}{\Phi \vdash_\Gamma \varphi_1}$$

$$\frac{\Phi \vdash_\Gamma \varphi_1 \wedge \varphi_2}{\Phi \vdash_\Gamma \varphi_2} \qquad \frac{\varphi_1, \Phi \vdash_\Gamma \varphi_2}{\Phi \vdash_\Gamma \varphi_1 \to \varphi_2} \qquad \frac{\Phi \vdash_\Gamma \varphi_1 \to \varphi_2 \qquad \Phi \vdash_\Gamma \varphi_1}{\Phi \vdash_\Gamma \varphi_2}$$

$$\frac{\Phi \vdash_{\{x:\tau\}\cup\Gamma} \varphi}{\Phi \vdash_\Gamma \forall x^\tau.\varphi}\, x \# \Phi \qquad \frac{\Phi \vdash_\Gamma \forall x^\tau.\varphi}{\Phi \vdash_\Gamma [M/x]\varphi}\, \Gamma \vdash M : \tau$$

ただし $\wedge$, $\to$, $\forall$ の導入則が許容可能であるためには命題外延性と関数外延性が必要です．いずれも紙の上では 2, 3 行で証明できるので，興味がある方は証明してみてください．

さて，論理結合子を用いることで命題外延性と関数外延性の規則図式がなぜそう呼ばれるかがわかります．命題外延性の規則図式を追加することは以下の公理図式を仮定することと同値です．

$$\left(\varphi_1 \leftrightarrow \varphi_2\right) \leftrightarrow \left(\varphi_1 \doteq \varphi_2\right)$$

これは「(解釈で) 区別できないもの (命題) は同じである」というまさに (命題についての) 外延性そのものです．同様に，関数外延性の規則図式を追加することは以下の公理図式を仮定することと同値です．

$$\left(\forall x^{\tau}.\, f_1 x \doteq f_2 x\right) \leftrightarrow \left(f_1 \doteq f_2\right)$$

これもまさに「(適用によって) 区別できないもの (関数) は同じである」という (関数についての) 外延性に他なりません．

4.3. 高階論理の実装

ついに高階論理を実装します．特に，対話的証明が行えることを確認します．高階論理の項は単純型付きラムダ計算に o 型と $\doteq$ という項が増えただけです．[*52]実装は 3 節の各種の定義をほんの少し変更するだけで済みます．[*53]

```
inductive type : Type
| base : type
| prop : type
| arrow : type → type → type

inductive term : type → Type
| var : Π {t}, ν t → term t
| lam : Π {t₁ t₂}, (ν t₁ → term t₂) → term (arrow t₁ t₂)
| app : Π {t₁ t₂}, term (arrow t₁ t₂) → term t₁ → term t₂
| eq : Π {t}, term t → term t → term prop
```

4.1 節で定義したように様々な論理結合子を定義します．今回は論理結合子をメタレベ

52 4.1 節と違い定数は考えませんが表現能力は変わりません．

53 実はこれはたまたまです．PHOAS によるエンコーディングは実装が綺麗に行く場合とそうでない場合がかなりはっきり分かれているのですが，今回実装した範囲ではたまたまほんの少しの変更で済みました．

ルの関数ではなくオブジェクトレベルの関数として実装しました．[*54]

```
def top : Term prop :=
λ ν, eq (lam (λ x : ν prop, var x)) (lam (λ x, var x))

def and : Term (arrow prop (arrow prop prop)) :=
λ ν, lam (λ p₁, lam (λ p₂, eq (lam (λ f : ν (arrow _ (arrow _ prop)), app (
app (var f) (top ν)) (top ν))) (lam (λ f, app (app (var f) (var p₁)) (var p
₂)))))

def Forall : Term (arrow (arrow t prop) prop) :=
λ ν, lam (λ f, eq (var f) (lam (λ x, (top ν))))

def bot : Term prop :=
λ ν, app (Forall ν) (lam (λ p, var p))

def implies : Term (arrow prop (arrow prop prop)) :=
λ ν, lam (λ p₁, lam (λ p₂, eq (app (app (and ν) (var p₁)) (var p₂)) (var p
₁)))

def not : Term (arrow prop prop) :=
λ ν, lam (λ p, app (app (implies ν) (var p)) (bot ν))

def iff : Term (arrow prop (arrow prop prop)) :=
λ ν, lam (λ p₁, lam (λ p₂, app (app (and ν) (app (app (implies ν) (var p
₁)) (var p₂))) (app (app (implies ν) (var p₂)) (var p₁))))

def or : Term (arrow prop (arrow prop prop)) :=
λ ν, lam (λ p₁, lam (λ p₂, app (Forall ν) (lam (λ r, app (app (implies ν) (
app (app (and ν) (app (app (implies ν) (var p₁)) (var r))) (app (app (impli
es ν) (var p₂)) (var r)))) (var r)))))

def Exists : Term (arrow (arrow t prop) prop) :=
λ ν, lam (λ f, app (Forall ν) (lam (λ r, app (app (implies ν) (app (Foral
l ν) (lam (λ x, app (app (implies ν) (app (var f) (var x))) (var r))))) (va
r r))))
```

54 `Forall` と `Exists` は先頭を小文字にすると Lean の予約語と名前が衝突するため大文字から始めています．

最後に高階論理の証明を定義して，全ての定義が終わりです．[*55]

```
inductive Theorem : Π {Γ}, list (Judgment Γ prop) → Judgment Γ prop → Prop
| hyp : Π {Γ Φ} {φ : Judgment Γ prop},
  φ ∈ Φ → Theorem Φ φ
| eq_intro : Π {Γ Φ t} {m₁ m₂ : Judgment Γ t},
  m₁ ≈ m₂ → Theorem Φ (eq m₁ m₂)
| eq_elim : Π {Γ Φ t} (m : Judgment (t :: Γ) prop) (m₂ m₁ : Judgment Γ t),
  Theorem Φ (eq m₁ m₂) → Theorem Φ (subst m m₁) → Theorem Φ (subst m m₂)
| prop_ext : Π {Γ Φ} {φ₁ φ₂ : Judgment Γ prop},
  Theorem (φ₁ :: Φ) φ₂ → Theorem (φ₂ :: Φ) φ₁ → Theorem Φ (eq φ₁ φ₂)
| fun_ext : Π {Γ Φ t₁ t₂} (m₁ m₂ : Judgment (t₁ :: Γ) t₂),
  Theorem (list.map weak Φ) (eq m₁ m₂) → Theorem Φ (eq (lam m₁) (lam m₂))
```

これで定理証明支援系が完成しました．お疲れ様でした．

と言ってもこれだけだとよくわからないので，実際に証明を書いてみます．たとえば，$p \vdash_{\{p:o\}} \top \doteq p$ を証明してみます．日本語にすると，「命題 p が仮定されているとき，p は真である．」という意味です．これは当然証明できてほしいでしょう．答えを先に述べると，以下の証明木を構成し，Lean の型検査が通ればその証明が正しい証明であることがわかるという寸法です．

$$\dfrac{\dfrac{}{\top, p \vdash_{\{p:o\}} p}\text{仮定} \qquad \dfrac{}{p, p \vdash_{\{p:o\}} \top}\doteq\text{導入}}{p \vdash_{\{p:o\}} \top \doteq p}\text{命題外延性}$$

判断 $p \vdash_{\{p:o\}} \top \doteq p$ に対応する Lean の命題は `@Theorem [prop] [var here] (eq (weak top) (var here))` です．`@Theorem` は命題 `Theorem` の暗黙の引数を明示的に与えるという構文です．この場合，`@Theorem [prop] ...` で判断の型環境 Γ が o 型の一つの変数からなるという意味になります．よって，Lean の中で以下の `...` を埋めれば証明が完了します．

```
def theorem1 : @Theorem [prop] [var here] (eq (weak top) (var here)) := ...
```

ここで，普通にプログラミングして Lean の項を作るのではなく，Lean のタクティック言語を用います．

---

55 `theorem` は Lean の予約語なので，代わりに `Theorem` と名付けます．

```
def theorem1 : @Theorem [prop] [var here] (eq (weak top) (var here)) :=
begin
  -- ここを埋める
end
```

すると，Lean は以下のようなメッセージを出力して証明を埋めるように指示してきます．

```
Tactic State:
⊢ Theorem [var here] (eq (weak top) (var here))
```

出来上がるべき証明木は根が命題外延性なので `prop_ext` を用います．もし普通に Lean の証明を書くのであればここで単に `prop_ext` と書くのですが，今回は Lean の上で実装された演繹体系の証明を書いているので `apply Theorem.prop_ext` と書くのが正解です．

```
def theorem1 : @Theorem [prop] [var here] (eq (weak top) (var here)) :=
begin
  apply Theorem.prop_ext,
  -- ここを埋める
end
```

すると，Lean は証明すべき命題があと二つ残っているという旨のメッセージを出力します．

```
Tactic State:
2 goals
⊢ Theorem [weak top, var here] (var here)

⊢ Theorem [var here, var here] (weak top)
```

このように，Lean 自体のタクティック言語を用いることで今回作成した論理の証明を対話的に記述することができます．この証明を最後まで書くと以下のようになります．

```
def theorem1 : @Theorem [prop] [var here] (eq (weak top) (var here)) :=
begin
  apply Theorem.prop_ext,
  { apply Theorem.hyp,
```

```
    simp },
  { apply Theorem.eq_intro,
    canonicity }
end
```

ここで apply Theorem.??? という形以外のタクティック (i.e., simp と canonicity) が用いられています． これらはそれぞれ $p \in \left[\top, p\right]$ と $\left(\lambda p^o.p\right) =_{\beta\eta} \left(\lambda p^o.p\right)$ を証明するために用いられています． このように， どの規則を使用して証明木を構成するかを apply Theorem.??? で指定し， その際に必要となる付帯条件を Lean 自体のタクティックで証明するのが，この証明支援系での証明の基本的な流れです．

最後にもう少し実用的な例として，$\Phi \vdash_{\varnothing} \varphi_1 \wedge \varphi_2$ ならば $\Phi \vdash_{\varnothing} \varphi_1$ を証明します．詳細は省きますが，この証明支援系での証明がどのような雰囲気かがわかると思います．

```
def theorem2 {φ₁ φ₂} {Φ : list (Judgment [] prop)} : Theorem Φ (app (app an
d φ₁) φ₂) → Theorem Φ φ₁ :=
begin
  intro p,
  apply Theorem.eq_elim
    (var here)
    φ₁
    (λ ν, app (lam (λ f, app (app (var f) (φ₁ ν)) (φ₂ ν))) (lam (λ p₁, lam (
λ p₂, var p₁)))),
  { apply Theorem.eq_intro,
    canonicity, },
  { apply Theorem.eq_elim
      (@id (Judgment [arrow _ prop] prop) $ λ ν f, app (var f) (lam (λ p₁, l
am (λ p₂, var p₁))))
      (λ ν, lam (λ f, app (app (var f) (φ₁ ν)) (φ₂ ν)))
      (@id (Judgment [] (arrow _ prop)) $ λ ν, lam (λ f, app (app (var f) (t
op ν)) (top ν))),
    { apply Theorem.eq_elim
        (var here)
        (eq
          (@id (Judgment [] (arrow _ prop)) $ λ ν, lam (λ f, app (app (va
r f) (top ν)) (top ν)))
          (λ ν, lam (λ f, app (app (var f) (φ₁ ν)) (φ₂ ν))))
        (λ ν, app (app (lam (λ p₁, lam (λ p₂, eq (lam (λ f : ν (arrow _ (a
rrow _ prop)), app (app (var f) (top ν)) (top ν))) (lam (λ f, app (app (va
```

```
r f) (var p₁)) (var p₂)))))) (φ₁ ν)) (φ₂ ν)),
      { apply Theorem.eq_intro,
        canonicity },
      { from p } },
    { apply Theorem.eq_elim
        (var here)
        (@id (Judgment [] _) $ λ ν, app (lam (λ f, app (app (var f) (top ν)) (
top ν))) (lam (λ p₁, lam (λ p₂, var p₁))))
        top,
      { apply Theorem.eq_intro,
        canonicity },
      { apply Theorem.eq_intro,
        canonicity } } }
end
```

5. まとめ

いかがでしたか？

この記事では定理証明支援系である Lean の上で高階論理に基づく定理証明支援系を作成しました．最終的なソースコードは github 上に公開してあります．[24] 紙面上は説明のためにかなり多くのコードを書いていますが，実際に最後の証明の部分にたどり着くだけなら 200 行ちょっと程度のコードだけで十分です．この数字が**極めて**小さいと言えるかどうかは微妙ですが，かなり小さくコンパクトに実装できていると思います．この記事の企画の類似物として HOL Light[19] や HaskHOL[23] がありますが，HOL Light ではパーサーや型推論器などの機能を一切提供しないコア部分が 670 行あります．当然ですが，今回の企画と HOL Light では定理証明支援系を小さく実装するというアイデアが共通しているのみで，参考程度の値にしかなりませんが，概観はつかめるかと思います．

ちなみに，今回は実装言語として定理証明支援系を選んだので作成した定理証明支援系自体を検証したいと思うわけですが，残念ながらそれはできません．理由は 3.7 節で説明したのと同じで，今回は色々と省コードにするためにかなり飛び道具を使っており，Lean の中では証明できない Lean の性質を用いてオブジェクト言語の健全性を保証しているからです．結局，PHOAS を使うのをやめて普通に頑張って実装したほうが後々の拡張のためには良いかもしれません．

さて，完走した感想ですが，今回の企画の趣旨である「**省エネ・省コード**で定理証明支援系を実装」というのは自分が楽して省エネで記事を完成させるという意味もありました．実際，Lean のコードを書き始めてから最後の節の最後の大きな証明を書くまでには 1 週間かかっていなかったはずです．しかし，記事を書くにあたって理論的背景を本気で書き始めてしまったことと，説明をわかりやすくしようと色々工夫するうちにコード量と執筆料が膨大になったことで，まったく省エネではありませんでした．また，この記事は $\mathrm{SAT}\Upsilon\mathrm{SF}_\mathrm{I}$ で書いているのですが，今回は初めてそれなりの量の文章を $\mathrm{SAT}\Upsilon\mathrm{SF}_\mathrm{I}$ で記述するとあって，その学習にもそれなりに時間が取られました．$\mathrm{SAT}\Upsilon\mathrm{SF}_\mathrm{I}$ は設計や意味論が美しく概ね不満はないのですが，ドキュメントが足りなかったり，処理系のバグにそれなりの頻度で遭遇するなど，まだまだハマった時の対処法が大変な印象でした．自分の場合は $\mathrm{SAT}\Upsilon\mathrm{SF}_\mathrm{I}$ の神であるところの gfn 氏に直接質問をしたり相談ができる環境だったのでまだよかったですが，独学で本格的に $\mathrm{SAT}\Upsilon\mathrm{SF}_\mathrm{I}$ を使うにはまだもう少し苦労が必要なようです．とはいえ，組版の品質は TEX に匹敵しますし，基本的な機能に絞ればすでに実用にも耐えられると思うので，あとは時間の問題かと思います．今回の執筆は自分にとっても非常に勉強になるとてもよい機会でした．高階論理，ラムダ計算，Lean，その他諸々，これらを初めて知った人でもそうでない人でも，この記事の内容が少しでも心に残れば幸いです．

参考文献

[1] Henk P. Barendregt. Lambda calculi with types. *Handbook of logic in computer science*, 2, pages 117–309, 1993.

[2] Murdoch Gabbay and Andrew M. Pitts. A New Approach to Abstract Syntax with Variable Binding. *Formal Asp. Comput.*, 13, pages 341–363, 2002.

[3] Andrew M. Pitts. *Nominal sets*. Cambridge University Press, 2013.

[4] Jeremy Avigad, Gabriel Ebner, and Sebastian Ullrich. *The Lean Reference Manual (Release 3.3.0)*. `https://leanprover.github.io/reference/lean_reference.pdf`, 2018.

[5] 横内 寛文 . プログラム意味論 . 共立出版 , 1994.

[6] Olivier Danvy, Morten Rhiger, and Kristoffer H. Rose. Normalization by evaluation with typed abstract syntax. *J. Funct. Program.*, 11(6), pages 673–680, 2001.

[7] Martin Hofmann. Semantical Analysis of Higher-Order Abstract Syntax. In *14th Annual IEEE Symposium on Logic in Computer Science*, pages 204–213, 1999.

[8] Jonathan Sterling and Bas Spitters. *Normalization by gluing for free λ-theories*. `https://www.jonmsterling.com/pdfs/gluing-note.pdf`, 2018.

[9] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

[10] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008*, pages 143–156, 2008.

[11] 萩谷 昌己 and 西崎 真也 . 論理と計算のしくみ . 岩波書店 , 2007.

[12] C. Barry Jay and Neil Ghani. The Virtues of Eta-Expansion. *J. Funct. Program.*, 5(2), pages 135–154, 1995.

[13] Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly Typed Term Representations in Coq. *J. Autom. Reasoning*, 49(2), pages 141–159, 2012.

[14] Olivier Danvy, Chantal Keller, and Matthias Puech. Typeful Normalization by Evaluation. In *20th International Conference on Types for Proofs and Programs, TYPES 2014*, pages 72–88, 2014.

[15] Daniel Schepler. *Parametricity*. `https://coq-club.inria.narkive.com/C0bDEhA1/parametricity#post4`, 2011.

[16] Andrej Bauer. *Is there a nice characterisation of topoi with nice meta-logical properties?*. `https://mathoverflow.net/q/132639`, 2013.

[17] Christoph Benzmüller and Dale Miller. Automation of Higher-Order Logic. *Compu-*

tational Logic, pages 215–254, 2014.

[18] Brigitte Pientka. *The Beluga language*. http://complogic.cs.mcgill.ca/beluga/index.html, 2019.

[19] John Harrison. *The HOL Light theorem prover*. https://www.cl.cam.ac.uk/~jrh13/hol-light/, 2019.

[20] Leonardo de Moura. *Lean theorem prover*. https://leanprover.github.io, 2019.

[21] Makarius Wenze. *The Isabelle proof assistant*. https://isabelle.in.tum.de/index.html, 2019.

[22] The Coq Consortium. *The Coq proof assistant*. https://coq.inria.fr, 2019.

[23] Evan Austin. *HaskHOL*. http://ecaustin.github.io/haskhol/, 2019.

[24] Yuichi Nishiwaki. *LeanHOL*. https://github.com/nyuichi/LeanHOL, 2019.

[25] Thierry Coquand and Gérard P. Huet. The Calculus of Constructions. *Inf. Comput.*, 76(2/3), pages 95–120, 1988.

[26] Alonzo Church. A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, 5(2), pages 56–68, 1940.

[27] Joachim Lambek and Philip J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986.

[28] William M. Farmer. The seven virtues of simple type theory. *J. Applied Logic*, 6(3), pages 267–286, 2008.

[29] Taichi Uemura. トポスと高階論理 . https://staff.fnwi.uva.nl/t.uemura/files/topos-and-hol.pdf, 2018.

[30] Bart Jacobs. *Categorical Logic and Type Theory*. North Holland, 1999.

[31] Yohji Akama. On Mints' Reduction for ccc-Calculus. In *International Conference on Typed Lambda Calculi and Applications, TLCA '93*, pages 1–12, 1993.

プログラミング言語を形式化する
もう一つの方法について

zeptometer

1. はじめに

意味ありげなタイトルですがこの記事の主旨は Beluga[*56]というプログラミング言語あるいは定理証明系[*57]を紹介することです。Beluga はプログラミング言語を形式化してその性質を証明することを目的としている変わり種の定理証明系で、現在 McGill 大学の Brigitte Pientka らにより活発に開発されています。 公式サイトによる紹介は以下の通りです。

> Beluga is a functional programming language designed for reasoning about formal systems. It features direct support for object-level binding constructs using higher order abstract syntax and treats contexts as first class objects.

なんだかわかるようなわからないようなことが書いてありますね。これの意味するところがなんとなくわかるところまで持っていくのがこの記事の目標となっております。がんばるぞー。記事の流れは以下のようになっております：

- 背景として単純型付きλ計算とその$\beta\eta$正規形について解説する
- Normalization by Evaluation というテクニックについて解説をしながら Reason で実装し、その実装の問題点について考える

56 http://complogic.cs.mcgill.ca/beluga/

57 Curry-Howard 同型が理論的背景にあるから境目が曖昧になるんですわ、しゃーない

- Beluga の言語機能について解説しつつ Normalization by Evaluation を再実装し、先の問題点が解消されていることを確認する

2. インストール

今のところ Beluga をインストールするためにはソースコードから直接コンパイルするのが確実そうです。GitHub のリポジトリ (https://github.com/Beluga-lang/Beluga) の INSTALL に各環境へのインストール方法が書いてあります。筆者は Linux での動作を確認しています。

3. 背景の背景：単純型付きλ計算の正規形

この記事では最終的に Normalization by Evaluation を形式化しますが、その前提知識として Church 流の単純型付きλ計算 (Simply Typed Lambda Calculus, STLC)[1] について説明しましょう。STLC は静的な型システムを持つプログラミング言語のエッセンスを抽出した体系のうち最もシンプルなものの一つです。

STLC の**型**は**基底型** ι と**関数型** $A \to B$ からなります。$\vdash A : \mathit{Type}$ という記法は「A は STLC の型である」ことを表わします。このように何らかの知識を表す記法をここでは**判断**と呼ぶことにします (便宜上この記事での判断は全て $\vdash$ を伴う記法を用います)。STLC の型に関する判断は以下の規則によって導出されます。

$$\frac{}{\vdash \iota : \mathit{Type}} \qquad \frac{\vdash A : \mathit{Type} \qquad \vdash B : \mathit{Type}}{\vdash A \to B : \mathit{Type}}$$

以降 STLC の型は暗黙にこれらの導出規則によって導かれたものと考えることにしましょう。

一方で STLC の**項**は**変数** x、**関数** $\lambda x^A.M$、**関数適用** MN からなります。これらはプログラムに相当するものです。さて、**環境** Γ を変数と型の対の集合とするときに、型判断 $\Gamma \vdash M : A$ を定義できます。これは「環境 Γ の下で項 M が型 A を持つ」ことを表す判断で、以下の規則に従って導出されます。

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ (\mathrm{Var})$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A.M : A \to B}\ (\text{Abs}) \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ (\text{App})$$

λ計算で重要な概念としてβ簡約があります。これは関数適用を実行することでプログラムが進んでいく過程を抽象化した概念です。項の集合を *Term* をするとき、β簡約 $\to_\beta$ は *Term* 上の二項関係で以下のような性質を満たすものと定義します。

$$\begin{aligned}
\left(\lambda x^A.M\right)N &\to_\beta M[x := N] \\
\lambda x^A.M &\to_\beta \lambda x^A.N \qquad \text{if } M \to_\beta N \\
ML &\to_\beta NL \qquad \text{if } M \to_\beta N \\
LM &\to_\beta LN \qquad \text{if } M \to_\beta N
\end{aligned}$$

また、二項関係 $\to_\beta^*$ を $\to_\beta$ の反射推移閉包としましょう。上の定義で $M[x := N]$ は「項 M 中における変数 x の使用を項 N に置き換えて得られる項」を表していますが、厳密な定義は結構面倒くさいのでここでは省略します。一つ目のルールが実際に関数適用を実行している部分で、他は部分項が実行されている場合のルールです。さて、STLC に関する重要な性質の一つとして、STLC で導出可能な項[*58]がβ簡約に関して以下に定義する**合流性**と**強正規化性**を満たすということが知られています。

定義 1　項 M がβ正規形である $\Leftrightarrow M \to_\beta N$ を満たす項 N が存在しない

定義 2　$\to_\beta$ が**合流性**を満たす $\Leftrightarrow M \to_\beta^* N_1$ かつ $M \to_\beta^* N_2$ ならば $N_1 \to_\beta^* L$ かつ $N_2 \to_\beta^* L$ であるような L が存在する

定義 3　$\to_\beta$ が**強正規化性**を満たす $\Leftrightarrow$ 無限簡約列 $M_1 \to_\beta M_2 \to_\beta \ldots$ が存在しない

β簡約の合流性と強正規化性の帰結として、「STLC 上で導出可能な任意の項はβ正規形をただ一つもつ」ということが言えます。

もう一つ重要な概念として**η正規形**があります。ふんわりとした説明になってしまいますが、これは「関数型を持つ項はλ項の形をしていてほしい」という気持ちに基いた概

58 厳密には「STLC の導出規則によって導出された型判断を伴う項」ですかね

念です。これは形式的には**η展開**よって記述されますが、素朴なη展開 $\to_\eta$ は以下のような $Term$ 上の二項関係として定義できます。

$$\begin{aligned} M &\to_\eta \lambda x^A.Mx && \text{when } M \text{ has type } A \to B \\ \lambda x^A.M &\to_\eta \lambda x^A.N && \text{if } M \to_\eta N \\ ML &\to_\eta NL && \text{if } M \to_\eta N \\ LM &\to_\eta LN && \text{if } M \to_\eta N \end{aligned}$$

一つ目のルールで関数型を持つ項をλ項の形に展開しており、先ほど述べた気持ちを表しています。ここでわざわざ「素朴な」η展開と呼んでいるのは、この定義だと強正規化性が満たされないためです。例えば以下のような自明な無限展開列が存在します。

$$M \to_\eta \lambda x^A.Mx \to_\eta \lambda x^A.\big(\lambda x^A.Mx\big)x \to_\eta \cdots$$

この例の場合、二つ目の項で欲しいものは得られているためここでη展開を止めるように定義を制限してやる必要があります。若干ややこしい話になってしまうためここでは詳細は省きますが[*59]、うまく制限したη展開はβ簡約と同様の合流性と強正規化性を満たすことが知られています [2]。

ある項がβ正規形かつη正規形である時、その項を**βη正規形**と呼びます。STLC における任意のβη正規形の項は以下の導出規則によって導かれることが知られています。

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A \in Neut}\ (\mathrm{NVar})$$

$$\frac{\Gamma, x : A \vdash M : B \in Norm}{\Gamma \vdash \lambda x^A.M : A \to B \in Norm}\ (\mathrm{NLam}) \qquad \frac{\Gamma \vdash R : \iota \in Neut}{\Gamma \vdash R : \iota \in Norm}\ (\mathrm{Embed})$$

$$\frac{\Gamma \vdash R : A \to B \in Neut \qquad \Gamma \vdash M : A \in Norm}{\Gamma \vdash RM : B \in Neut}\ (\mathrm{RApp})$$

この導出規則においては中立項 $Neut$ と正規項 $Norm$ を相互再帰的に定義しています。結果として得られる $Norm$ がβη正規形の項の集合になります。この定義がβ簡約とη展開の定義に基づく正規形と一致していることはあまり自明ではない気がしますが、とりあえずここではそういうものだと思ってください。

---

59 wasabiz の記事で言及があります。興味のある方はそちらを参照してください。

さて、これまでの話をふまえるとSTLCの項が与えられた時にそれに対応するβη正規形の項を得たいというのはごく自然な考えであると思われます。

- プログラム
  - 入力：STLCの項
  - 出力：入力の項に対応するβη正規形
  - 実装：⁇

この問題に対する実装方針としてまず最初に思いつくのは「正規形を得るまで項にβ簡約とη展開を繰り返し適用する」という素朴なアルゴリズムでしょう。しかしながらこれにはいくつか困難な点が存在します。

一つ目の問題点は変数束縛の管理です。例えば以下のような型判断を考えてみましょう。

$$y : A \to B \vdash \left(\lambda x^{A \to B}.\lambda y^{A}.xy\right)y : A \to B$$

定義によるとこの項は以下のようにβ簡約できます。

$$\left(\lambda x^{A \to B}.\lambda y^{A}.xy\right)y \to_{\beta} \left(\lambda y^{A}.xy\right)\left[x := y\right]$$

ここで $M[x := N]$ は M 中の x を N に置き換えた項だったわけですが、$\left(\lambda y^{A}.xy\right)\left[x := y\right]$ の場合、λ抽象によって導入された y と x を置き換えた後に入る y で変数名が衝突してしまいます。

$$\left(\lambda y^{A}.xy\right)\left[x := y\right] \neq \lambda y^{A}.yy \qquad \text{// ダメ！}$$

こういった事態は**意図しない変数補足**と呼ばれます。これを避けるために、変数名の衝突が起きる場合にはλ抽象の束縛変数を衝突しないものに置き換える必要があります。

$$\begin{aligned}\left(\lambda y^{A}.xy\right)\left[x := y\right] &= \left(\lambda z^{A}.xz\right)\left[x := y\right] && \text{// 束縛変数の y を z に置き換えると}\\ &= \lambda z^{A}.yz && \text{// オーケー}\end{aligned}$$

このようにちゃんとした置換を実装するためには意図しない変数補足が起きないように確認しつつ必要であれば束縛変数の名前を書き換えていく必要があるわけです[*60]。これを

60 この束縛変数の名前の書き換えのことを**α変換**と呼びます。詳細は [1] をご参照ください。

きちんと実装するのがなかなか面倒です。β簡約に関してあまり本質的でない部分に実装を割かないといけないのはあまりよろしくありません。

もう一つの問題点はη展開を適切に定義するのが面倒くさいという点です。η展開の説明でも述べたように、素朴な規則を用いると無限展開列が生じていつまでたっても正規形を得ることができません。それを防ぐためにはη展開の規則を制限してやる必要がありますが、これはこの記事にわざわざ書くのが面倒くさい程度には複雑です。そういった性質のものをきちんと定義に基いてプログラムに書き起こす作業がいかに大変かは想像していただける通りかと思われます。

このようにβ簡約、η展開の定義に基いて正規形を得るという方針は素直に見えて実は様々な困難があり、あまり実装したいものではありません。

4. 背景 : Normalization by Evaluation / 評価による正規化

このような前置きをしたからには簡約を用いた正規化以外にも STLC の項のβη正規形を得る方法はあるわけです。それがこれから紹介する Normalization by Evaluation(NbE, 評価による正規化)[3] です。基本的なアイデアは名前の通りで、**λ項に対する eval 関数を用いて正規形を得る**というものです。

一般的に eval(評価) 関数とはプログラムのコードを受け取ってそれを実行した結果を返す関数のことです[*61]。もう少し抽象的な解釈として、ここでは eval 関数を「λ項の集合からそれに対応する**意味**[*62]の集合への関数」と考えることにしましょう。以降λ項の集合を *Term*、意味の集合を *Sem* と呼ぶことにします。

$$eval : Term \to Sem$$

eval 関数に対する妥当な仮定として以下のものを考えます。

仮定 4　M と N が同じ正規形を持つなら $eval(M) = eval(N)$ である。

仮定 5　M と N が同じ正規形を持たないなら $eval(M) \neq eval(N)$ である。

---

61 Ruby や Python、Javascript などのインタープリタを前提とした言語がサポートしていることが多い印象があります。あと Common Lisp とか Scheme とか Clojure とか。

62 要は semantics のことです。

これらの仮定が成り立つ時、**同じ正規形を持つ項はその場合に限って同じ意味を持つ**ということが言えるはずです。とすると、ある意味 s が与えられた時に $eval(M) = s$ となるような M のうちただ一つの正規形を選んでくることができるはずです。これを実現する関数を NbE の文脈では reify(具象化) と呼んでいます。以降、正規形からなる集合を *Norm* と呼びましょう。reify の型は以下のようになるはずです。

$$reify : Sem \to Norm$$

ここまでくれば後は簡単で、この eval と reify を組み合わせることで項の正規形を得る関数を定義できます。

$$normalize = reify \circ eval : Term \to Norm$$

これが NbE の基本的なアイデアです ([4] にこれを端的に表した図があったので図 1 として引用しておきます)。

NbE のアイデアのキモは reify 関数を如何に実装するかという点にあります。これを実際に Reason で NbE を実装しながら見ていくことにしましょう[*63]。最初にすることは STLC の型と項を Reason のデータ型として定義することです。

```
type typ =
  | Base
  | Arr(typ, typ);

type term =
  | Var(string)
  | Abs(string, typ, term)
  | App(term, term);
```

変数名は string で表すことにしています。term は型判断とは独立して定義されているので導出できない項も表現できてしまいますが、今のところは妥当な型判断を伴う項だけを考えることにしましょう。

次に *Sem* に対応するものとして `semantics` という型を定義します。

63 Reason を選んだのは代数的データ型をサポートした静的型付け言語で綺麗なコードが書けそうだったからで特に深い意味はないです :)

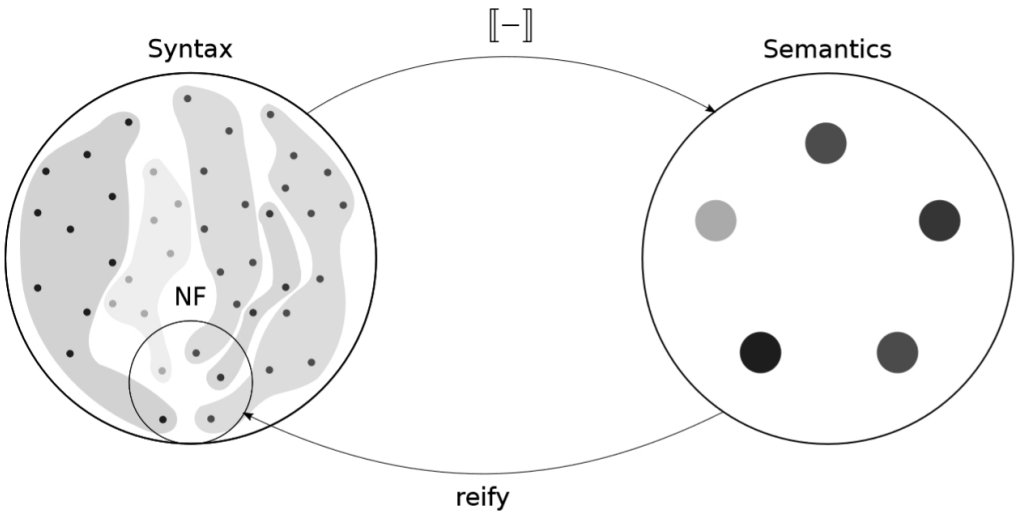


図 1　NbE の基本的なアイデア ([4] より引用)

```
type semantics =
  | MBase(term)
  | MAbs(semantics => semantics);
```

基底型の項に対応する意味が `MBase` で、 これは正規形の項をそのまま持ちます。 型に `term` を使っているのは Reason の型システムでは正規形を表現できないためです。 また、関数型の項に対応する意味が `MAbs` で、これは意味から意味への関数から成ります。

STLC の項を評価するためには評価される項の他に自由変数に対応する意味を持つ**環境**が必要です。 これを `env` という型として定義しましょう。 これは変数名を与えた時に対応する `semantics` を返す関数として実装できます。空の環境 `envEmpty` は必ず `Undefined_variable` を例外として投げます。また `envExt` は変数名と `semantics` のペアを受け取って既存の環境を拡張します。

```
type env = string => semantics;

type exn +=
  | Undefined_variable;
```

```
let envEmpty: env = _ => raise(Undefined_variable);

let envExt: (string, semantics, env) => env =
  (name, sem, env) => (name2) =>
    if (name === name2) {
      sem;
    } else {
      env(name2);
    };

let envEmpty: env = _ => raise(Undefined_variable);

let envExt: (string, semantics, env) => env =
  (name, sem, env) => (name2) =>
    if (name === name2) {
      sem;
    } else {
      env(name2);
    };
```

然る後に `eval` の本体を書くことができます。項の構造に関する再帰を用いて素直に実装できます。変数の場合は環境から変数名に対応する意味を返す。λ抽象の場合は意味を受け取ってそれを環境に追加した上でλ抽象の本体を評価する関数を返す。適用の場合はそれぞれを評価した上で一つ目の意味が `MAbs` になっているはずなのでこの関数に二つ目の意味を渡す。

```
type exn +=
  | Wrong_application;

let rec eval: (term, environment) => semantics =
  (tm, env) => {
    switch (tm) {
    | Var(name) => env(name)
    | Abs(name, _, n) => MAbs(sem => evaluate(n, envExt(name, sem, env)))
    | App(n1, n2) =>
      let sem1 = evaluate(n1, env);
      let sem2 = evaluate(n2, env);

      switch (sem1) {
```

```
      | MAbs(f) => f(sem2)
      | MNat(_) => raise(Wrong_application)
      };
    };
  };
```

というわけですんなりと eval を実装することができました。いいですね、人生もこうすんなりといってほしいものです。

最後に本題の reify 関数を実装します。まず補助関数として新しい変数名を生成する関数を定義しておきます。

```
let varCount = ref(0)
let gensym: unit => string =
  () => {
    let num = varCount^;
    varCount := num + 1;
    "x" ++ Js.Int.toString(num)
  }
```

reify 本体は以下のように定義できます。これはもう一つの関数 reflect との相互再帰によって実装されています。

```
type exn +=
  | Illegal_type;

let rec reify: (semantics, typ) => term =
  (sem) => (ty) => {
    switch (sem) {
    | MBase(tm) => tm
    | MAbs(fn) =>
      switch (ty) {
      | Base => raise(Illegal_type)
      | Arr(ty1, ty2) =>
        let x = gensym();
        Abs(x, ty1, reify(fn(reflect(Var(x), ty1)), ty2))
      }
    }
  }
```

```
and reflect: (term, typ) => semantics =
  (tm, ty) => {
    switch(ty) {
    | Base => MBase(tm)
    | Arr(ty1, ty2) =>
      MAbs((sem) => reflect(App(tm, reify(sem, ty1)), ty2))
    }
  }
```

基底型の意味についてはそれが持っている正規形を取り出すだけで大丈夫です。関数型の場合には、とりあえずλ抽象の項を作って、body の部分にはλ抽象によって導入された変数 x に対応する意味を関数 fn に渡して帰ってきた意味に対して reify を再帰的に適用します。この「変数 x に対応する意味」を担当するのが reflect で、これは *Neut* に属する項についてそれに対応する意味を返します。

煙に巻かれたような気持ちになるかもしれませんが、実際にこれでβη正規形を得ることができます。以上が Reason による NbE の実装です。

5. Beluga による Normalization by Evaluation の定式化

さて、先ほど示した Reason による実装にはいくつかうれしくない点があります[*64]。

- 束縛の管理に本質的でないコードが必要
- 項が型判断を伴わない場合がある
- 正規形の項の型がない
- 上記の要因によって実行時エラーが発生しうる

しかし Beluga を使えばこれらの問題に悩まされることはありません。実際に見ていきましょう。

64 Reason の型システムでもこれらの問題点をある程度解決することは可能です。Beluga がどのような問題を解決したいかを説明するためあえて改善の余地のある実装にしています、ご了承ください。

5.1. Beluga の概要

冒頭でも説明したように、Beluga は McGill 大の Brigitte Pientka らによって開発されている定理証明支援系あるいはプログラミング言語です。 定理証明支援系というと Coq や Agda, Isabelle などが有名ですが、Beluga がそれらと異なるのは「プログラミング言語[*65]の形式化とその性質の証明」に特化した言語設計となっている点です。

Beluga では「プログラミング言語を形式化する部分」と「形式化されたプログラミング言語についての性質を記述する部分」を異なるレイヤに分離しています。 前者のレイヤは LF と呼ばれる言語によって記述され、一方で後者のレイヤは LF に対する**メタプログラミング**として記述されます (図 2)。これらの概念がどういうものかを実際にコードを書きながら示すことがこの章の目標です。

5.2. LF による STLC の形式化

NbE を Beluga で記述する上で最初にすることは STLC の形式化です。Beluga ではこれを LF[5, 6] によって実現します。LF は依存型理論をベースとした言語で、論理体系やプログラミング言語を形式化するために考案されたものです。 この章では実際にコードを書きながら LF のエッセンスを理解していくことにしましょう。

まずは型の定義をしましょう。STLC の型は基底型と関数型から成るのでした。 型に関する判断を LF にエンコードすると、以下のようなコードになります。

```
LF tp : type =
| base : tp
| arr  : tp → tp → tp
;
```

このコードでは 3 つの新しい定数 `tp`, `base`, `arr` を定義しています。`LF` は新しい LF 定数を定義するためのキーワードです。まず最初の行で `tp` という STLC の型を表す型を定義しています。その後に `tp` に関係する定数として `base` と `arr` を定義します。これらはそれぞれ STLC の基底型と関数型に対応します。`arr` の型は `tp → tp → tp` となっていますが、こ

65 もう少し抽象的に言うと「束縛構造を持つような構造」一般のことを指しますが、プログラミング言語はこれに含まれると考えて差し支えないでしょう。

図 2　Beluga の二つのレイヤ

れは関数型に関する導出規則が判断を二つとって判断を一つ返すことに対応しています。例えば $A \to B$ が `arr A B` に対応するわけです。 なお、`LF` で定義する定数名は**全て小文字である必要があります**。 これは Beluga が多くの種類の変数を扱っているが故の制限です。 ちなみに `LF` は糖衣構文で、 以下のように 1 行ずつ LF 定数の定義を行うこともできます。

```
tp   : type.
base : tp.
arr  : tp → tp → tp.
```

次に STLC の項を定義します。LF の依存型は「STLC において正しい型判断を伴う項」を表現することを可能にします。見てみましょう、定義はたったの 4 行です。

```
LF term : tp → type =
| app : term (arr a b) → term a → term b
| lam : (term a → term b) → term (arr a b)
;
```

1 行目で `term` という型を定義しています。 これは `tp` の項を一つとります。 たとえば `tp` の型を持つ項 `A` があったときに、`term A` で「STLC において A という型を持つλ項」を表します。 このように型の中に項の出現を許すのが依存型の特色です。`app` は関数適用の項に対応する定数で、 型 `arr a b` の項と型 `a` の項から型 `b` の項をつくります。 以下の STLC の導出規則に素直に対応していることがわかります。

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ (\text{App})$$

ところで上の定義の中に変数に対応するものがないことに気付かれたかもしれません。LF では**エンコードする対象の言語の変数を LF の変数で直接表わします**。例えば STLC の項とその LF におけるエンコードは以下のようになります。

STLC の項	LF による表現
x	`x`
xy	`app x y`

関数の項に対応する定数 `lam` はこのアイデアの延長線上にあります。(Abs) の規則を思い出してみましょう。

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A.M : A \to B}\ (\text{Abs})$$

環境の中に型 A の変数 x がある時にそれを項の方へ持っていってλ抽象にすることができるというルールです。 先ほどの LF の変数で STLC の変数を表すというアイデアに基づくと、 上の導出規則における前者の状況は `term a → term b` の型を持つ LF 項で表せるはずです。 この項から `term \arr{a}{b}` の LF 項をつくるのが `lam` の役割で、 これは丁度 (Abs) の規則に一致します。`lam` を使うと以下のようにエンコードができます。

STLC の項	LF による表現
$\lambda x^A.x$	`lam (\x.x)`
$\lambda x^{A \to B}.\lambda y^A.xy$	`lam (\x. (lam \y. (app x y)))`

このようにエンコードの対象の言語 (ここでは STLC) の変数をエンコードする側の言語 (ここでは LF) の変数で表わすテクニックを Higher Order Abstract Syntax(HOAS) と呼びます。変数名を文字列で管理する方法に比べて、HOAS は変数の取り扱いの面倒な部分をエンコードする側の言語に任せることができ実装が大幅に楽になる利点があります。

LF の依存型は大変強力なので、STLC のβη正規形を表現することも可能です。 以前に定義したβη正規形の導出規則を Beluga のコードに落としこむと以下のようになります。

```
neut  : tp → type.
norm  : tp → type.
```

```
nlam  : (neut a → norm b) → norm (arr a b).
rapp  : neut (arr a b) → norm a → neut b.
embed : neut base → norm base.
```

正規形の定義は *Neut* と *Norm* の相互再帰的なっているため LF 文では定義できません。変わりに一行ずつ新しい LF 定数を定義しています。

5.3. 計算の階層から Normalization by Evaluation を定義する

先ほど LF で定義した STLC の項と正規項を用いて NbE を実装していきましょう。ただここで一つ問題がでてきます。例えば Coq であれば STLC の項を代数的データ構造として定義して、それに対するプログラムないし証明を書くことで STLC の性質を書くことができます。しかしながら今回我々が STLC を表現するために定義した LF 定数はあくまでただの定数でパターンマッチができるようなデータ構造ではありません。この問題を解決するために Beluga ではメタプログラミングの階層を提供して、そこから **LF のコードをオブジェクトとして扱って**、それに対して計算することができるようになっています。Beluga ではこれを**計算の階層**と呼んでいます。名前の通り Beluga で実際に計算できるのはこちらの階層になるので、こちらが Beluga のメインの言語だと言って差し支えないでしょう。

計算の階層がどのように LF のコードを扱うのか実際に NbE を定義しながら見てみましょう。まず意味を表すデータ型として `Sem` を定義します。

```
schema ctx = some [a:tp] block x:neut a;

stratified Sem : {g : ctx} [ ⊢ tp] -> ctype =
| Base : [g ⊢ norm base] → Sem [g] [ ⊢ base]
| Arr  : {g : ctx} ({h : ctx} {#S: [h ⊢ g]} Sem [h] [ ⊢ A] → Sem [h] [ ⊢ B])
                  → Sem [g] [ ⊢ arr A B]
;
```

いきなりこんなの見せられてもなんのこっちゃわかりませんな[*66]。大丈夫です、これから

66 ここらへんの文章を書く段階になって Beluga を説明する上で NbE はあまり適切ではないんじゃないのかと気付いた次第です。許してほしい。

これが何を意味しているのかを説明していきますから。Beluga では LF の階層のオブジェクトを [] で囲んで表記します。 これらのオブジェクトの中で一番重要なのは LF のコードでしょう。g が LF の環境、T が LF の型を表す時に [g ⊢ T] は「環境 g の下で型 T を持つような LF のコード」の型を表します。 先に定義した STLC のエンコードを用いると以下のような型を考えることができます。

Beluga の型	意味
[⊢ tp]	空の環境における STLC の型
[x:term base ⊢ term base]	環境 x:term baseの下で基底型を持つ STLC の項
[g ⊢ norm (arr a b)]	何らかの環境 gの下で型 arr a bの型を持つ STLC の項

最後の例にあるように、Beluga では環境を表す変数を使うことができます。これを**文脈変数** (context variable) と呼びます。 環境変数を使う際にはその環境がどのような型を伴うかについての情報が必要です。これを実際に与えるのが schema で、schema 文によって定義されます。先ほどのコードの一行目を見てみましょう。

```
schema ctx = some [a:tp] block x:neut a;
```

ここでは ctx という schema を定義しています。block 以降が環境がどのような型を持つかを表わしていて、この例だと ctx の schema を持つ環境に出てくる変数は STLC の中立項であると定義しています。ただし neut a の a の部分が STLC の型であることを明示してやる必要があるので、それを some [a:tp] の部分で与えています。

続く行で実際に意味を表わすデータ型 Sem を定義しています。

```
stratified Sem : {g : ctx} [ ⊢ tp] -> ctype =
```

stratified は代数的データ構造を定義するための文です[*67]。なお、stratified で定義するデータ型の名前は全て**大文字から始まる必要がある**のでご注意ください。Sem は ctx の文脈変数 g と STLC の型 (のコード) をパラメータとして持ち、g の環境の下で第二引数で

67 厳密には代数的データ構造を定義する構文は別に inductive というのがあります。 ただ通常の代数的データ構造では negative な位置に再帰することができないため、 ここではより強力な stratified を用いています。詳しいことは [7] に書いてありそう。

与えられた STLC の型を持つ項に対応する意味を表わします。なお、ctype は計算の階層における型を表わす型です。

```
| Base : [g ⊢ norm base] → Sem [g] [ ⊢ base]
```

基底型の意味 Base は Reason の場合と同じくその正規項をそのまま使って与えます。[g ⊢ norm base] が何らかの環境 g の下で基底型を持つ STLC の正規項の型で、それを第一引数にとって対応する型 Sem [g] [⊢ base] の意味としています。

```
| Arr  : {g : ctx} ({h : ctx} {#S: [h ⊢ g]} Sem [h] [ ⊢ A] → Sem [h] [ ⊢ B])
                  → Sem [g] [ ⊢ arr A B]
```

関数型の意味 Arr も Reason の場合と同じく意味から意味への関数によって定義します。ただし Beluga では Reason の場合よりもかなり厳密に記述することができます。第一引数に ctx の環境 g をとるのは Base の場合と同じです。次の引数にとるのが意味から意味への関数ですがこの部分の型だけに注目してみましょう。

```
{h : ctx} {#S: [h ⊢ g]} Sem [h] [ ⊢ A] → Sem [h] [ ⊢ B]
```

ここで新たに二つの概念、**置換変数**と**メタ変数**が出てきます。それぞれ見ていきましょう。

置換は計算の階層の概念で、これは**ある環境の LF 項をより弱い**[*68]**他の環境の LF 項に変換するものです**。置換は Beluga において特別扱いされるオブジェクトで、それゆえ置換変数という独立した変数を用意されているわけです。ちなみに #S のように**先頭が # で残りが大文字になっている必要があります**。第二引数の {#S: [h ⊢ g]} がまさにそれですね。この置換変数の型 [h ⊢ g] は環境 g から環境 h へ変換する置換を表しています。この置換変数を与えることで、新しく導入された文脈変数 h ともとの文脈変数 g との関係を明示していています。

メタ変数も計算の階層の概念で、これは **LF のコードを他の LF のコードに埋め込む際に使われます**。また、メタ変数は**名前が全て大文字である必要があります**。ここでは [⊢

68 ここで言う弱いとは環境の weakening の意味です。

A] のような形で「まだ型推論で解決されていない LF 項のコード」を表わしているだけですが、後に置換と一緒に用いる例をお見せします。

若干説明が長くなってしまいましたが、これらを踏まえると先ほどの型は「もとの環境 g よりも弱い任意の環境 h について、その環境における型 A の意味から型 B の意味への関数を与えるもの」ということになります。これを Arr に与えることで関数型の項の意味を与えることができます。

次に補助関数として sem_wkn を定義しましょう。これはある環境 g における意味があった時に、それをより弱い環境 h の意味へ変換する関数です。

```
rec sem_wkn : {h : ctx} {g : ctx} {#S : [h ⊢ g]} Sem [g] [ ⊢ A] → Sem [
h] [ ⊢ A]  =
  mlam h ⇒ mlam g ⇒ mlam S ⇒ fn e ⇒ case e of
  | Base [g ⊢ R] ⇒ Base [h ⊢ R[#S]]
  | Arr [g] f ⇒ Arr [h] (mlam h' ⇒ mlam S' ⇒ f [h'] [h' ⊢ #S[#S']])
;
```

rec という名前が示すようにこれは再帰関数となっていて、Sem の構造に沿って定義しています。Beluga では LF のコードに対するパターンマッチが可能で、上のコードだと case e of ... の部分が Sem とその引数に対するパターンマッチをやっています。

与えられた意味が Base の場合、それは [g ⊢ norm base] の型のコードを伴うはずです。そのため Base [g⊢R] というように LF 項の部分をメタ変数 R にマッチさせています。R は LF のコードを表すメタ変数なので、[g ⊢ norm base] の型のコードが代入されます。然る後にこれを使って [h ⊢ norm base] の型のコードを作りますが、ここで g から h への置換 #S を使います。具体的には R[#S] と書くことで [h ⊢ norm base] を得ることができます。これを環境 h のコードに埋め込むことで弱化を実現させています。Arr の場合も似たような感じで、こちらは #S[#S'] のように置換に対して置換を行うことで新しい置換を作り出しています。

意味が定義できたのでお次は eval 関数を定義しましょう。まずは環境を表すデータ型を定義します。

```
schema tctx = some [t : tp] block x : term t;
```

```
typedef Env : {g : tctx} {h : ctx} ctype =
  {T : [ ⊢ tp]} {#p : [g ⊢ term T[]]} Sem [h] [ ⊢ T]
;
```

一行目で新しい schema の `tctx` を定義していて、これは STLC の項からなる環境を表しています。この定義には新しい概念として**パラメータ変数**が出てきます。これも計算の階層の概念で、ある環境の中の特定の変数を表します。上のコードの例だと `{#p : [g ⊢ term T[]]}` は環境 `g` 中における `term T` 型の変数を表しています。つまるところある環境の型 `Env g h` は `g` の中の型 `T` 変数 `#p` が与えられた時に、それに対応するような `Sem [h] [ ⊢ T]` の意味を返す関数になっています。

eval 関数は以下のように定義できます。いくつかの補助関数が必要ですが、STLC の項の構造に沿って再帰していくのは Reason の場合とあまり変わりません。[*69]

```
rec env_ext : Env [g] [h] → Sem [h] [ ⊢ S] → Env [g, x : term S[]] [h]  =
  fn env ⇒ let env : Env [g] [h] = env
    in fn sem ⇒ mlam T ⇒ mlam p ⇒ case [g, x : term _ ⊢ #p] of
       | [ g, x : term S ⊢ x ] ⇒ sem
       | [ g, x : term S ⊢ #q[..] ] ⇒ env [ ⊢ T ] [ g ⊢ #q ]
;

rec env_wkn : {h' : ctx} {h : ctx} {#W : [h' ⊢ h]} Env [g] [h] → Env [g] [
h']  =
mlam h' ⇒ mlam h ⇒ mlam W ⇒ fn env ⇒
  let env : Env [g] [h] = env
  in mlam T ⇒ mlam p ⇒ sem_wkn [h'] [h] [h' ⊢ #W] (env [ ⊢ T] [g ⊢ #p])
;

rec eval : [g ⊢ term S[]] → Env [g] [h] → Sem [h] [ ⊢ S]  =
fn tm ⇒ fn env ⇒
  let env : Env [g] [h] = env
  in case tm of
     | [g ⊢ #p] ⇒ env [ ⊢ _] [g ⊢ #p]
```

69 説明が雑ではないかと思われたかもしれません。締切が 30 分前なので端折られたのです。申し訳ありません ... 一応必要な概念の説明はしたつもりなので練習問題のつもりで読んでいただければと思います。

```
    | [g ⊢ lam (\x. E)] ⇒
      Arr [h] (mlam h' ⇒ mlam W ⇒ fn sem ⇒
                  let extEnv = (env_ext (env_wkn [h'] [h] [h' ⊢ #W] env) se
m)
                  in eval [g, x : term _ ⊢ E] extEnv)
    | [g ⊢ app E1 E2] ⇒ let Arr [h] f = eval [g ⊢ E1] env
                           in f [h] [h ⊢ ..] (eval [g ⊢ E2] env)
;
```

また reify/reflect についても同様に、Reason のコードを精緻化した形で書くことができます。

```
rec app' : (g:ctx) {R : [g ⊢ neut (arr T[] S[])]}
           [g ⊢ norm T[]] → [g ⊢ neut S[]]  =
mlam R ⇒ fn n ⇒ let [g ⊢ N] = n in [g ⊢ rapp R N];

rec reify : Sem [h] [ ⊢ A] → [h ⊢ norm A[]] =
fn sem ⇒ case sem of
    | Base [h ⊢ R] ⇒ [h ⊢ R]
    | Arr [g] f ⇒ let  [g, x : neut _ ⊢ N] =
                    reify (f [g, x : neut _]
                             [g, x ⊢ ..]
                             (reflect [g, x : neut _ ⊢ x]))
                    in [g ⊢ nlam (\x. N)]
and reflect : [h ⊢ neut A[]] → Sem [h] [ ⊢ A] =
fn n ⇒ let [h ⊢ R] : [h ⊢ neut A[]] = n
        in case [ ⊢ A] of
            | [ ⊢ base ] ⇒ Base [h ⊢ embed R]
            | [ ⊢ arr B C ]
              ⇒ Arr [h] (mlam h' ⇒ mlam W ⇒ fn tm ⇒
                           reflect (app' [h' ⊢ R[#W]] (reify tm)))
;
```

仕上げに正規化関数 `normalize` を定義します。

```
rec normalize : Env [g] [h] → [g ⊢ term A[]] → [h ⊢ norm A[]] =
fn env ⇒ fn tm ⇒ reify (eval tm env)
```

```
;
```

この Beluga によって記述された正規化関数は型レベルで期待される動作が実現されていることを確認できます。これは Reason で書かれた正規化関数に対する大きな利点であると言えます。

最後にこのコードが Beluga の型検査を通ることを確認してこの章の締めとしましょう。今回のコードは http://bit.ly/2I36EmP からダウンロードできます。

```
$ beluga nbe.bel

## Type Reconstruction: nbe.bel ##

## Type Reconstruction done: nbe.bel  ##
```

お疲れさまです。

6. まとめとあとがき

ここまで読んでいただいてありがとうございます。最後の方は大分駆け足になってしまいましたが、Beluga という言語の雰囲気だけでも伝えることができれば幸いです。Beluga の言語設計はかなり異色を放っており、初見だと面喰らう方も多いかと思われます。しかしそれらはきちんとした理論的な背景を持つ言語設計で、一度飲み込むことができればあなたの心強い味方になってくれるはずです。

Beluga の特殊な言語設計によって得られる最大の恩恵は「エンコードされたプログラム言語に対する操作をかなり直感的に書き下すことができる」という点にあります。Coq や他の汎用の証明系でもプログラミング言語を形式化してその性質を証明することはもちろん可能ですが、それを実現する上では様々な困難とそれを解決するためのテクニックが存在します。Beluga では証明の対象をプログラミング言語に限定してそれに特化した言語設計にすることで、そういったあまり本質的でない証明テクニックの必要性を排除しようとしているわけです。

Beluga についてより詳しく知ってみたいという方は Beluga のホームページ (http://complogic.cs.mcgill.ca/beluga/) を訪れてみることをお勧めします。 特に Mechanizing Types and Programming Languages: A Companion[8] はプログラミング言語の様々な性質を証明しながら Beluga の機能を学習していくという体裁をとっておりよい教材になると思われます。Beluga の理論的な側面を知りたいという人は [9, 7, 10] あたりを見るといいのではないかと思います。

Beluga を実際に触ってみた個人的な感想としては「まだ開発途上の言語である」の感が強いです。 大学で研究の対象として開発されている状況を考えると当然といえば当然ですが。 ツールとしての機能やドキュメントの整備についてはまだまだ改善の余地があるように感じました。一方でコアの部分のアイデアはかなり魅力的に感じるのでこのままいい感じに発展していくことを期待しています。

この記事を書いたことは筆者にとって得難い経験となりました。 以前から Beluga について勉強したいと考えていたもののなかなか進捗がなく、技術書典 6 に寄稿するというモチベーションの下にようやくとりかかることができました。記事を書こうとすると自分が思っていたほど物事を理解していないことに気付くということを何度も繰り返しており、実際にアウトプットを出すことの大切さを痛感しています。 おかげで Beluga についてそれなりに深い知識を得ることができたという実感があります。

こんなところですかね。それではまた次号でお会いしましょう。

参考文献

[1] H. P. Barendregt. Lambda calculi with types. *Handbook of logic in computer science*, 2, pages 117–309, 1993.

[2] C. Barry Jay and Neil Ghani. The Virtues of Eta-Expansion. *J. Funct. Program.*, 5(2), pages 135–154, 1995.

[3] Sam Lindley. *Normalisation by evaluation in the compilation of typed functional programming languages*. PhD Thesis, 2005.

[4] Sam Lindley. *Normalisation by evaluation*. `http://homepages.inf.ed.ac.uk/`

slindley/nbe/nbe-cambridge2016.pdf, 2016.

[5] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework For Defining Logics. *Journal of the Association for Computing Machinery*, 40(1), pages 143–184, 1993.

[6] Frank Pfenning. Logical Framework : A Brief Introduction. *Proof and System-Reliability*, pages 137–166, 2002.

[7] Brigitte Pientka and Andrew Cave. Inductive Beluga: Programming Proofs. In *Automated Deduction - CADE-25*, pages 272–281, 2015.

[8] Brigitte Pientka. *Mechanizing Types and Programming Languages: A Companion*. https://github.com/Beluga-lang/Meta, 2018.

[9] Andrew Cave and Brigitte Pientka. Programming with Binders and Indexed Data-types. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 413–424, 2012.

[10] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual Modal Type Theory. *ACM Trans. Comput. Logic*, 9(3), pages 1–49, 2008.

多段階計算と可変参照のための型システム

gfn

この記事では，一種のメタプログラミング的手法として使われる**多段階計算** (*multi-stage programming*) と， 手続き的プログラミングを実現する構成要素である**可変参照** (*mutable reference*) とを共存させることについて扱います． これらを共存させたときに生じる問題と， その問題が実行時に生じないように静的な検査によって回避するための型システムについていくつかの既存研究を紹介し， またそれらの依然として弱点があると思われる点とそのおおまかな克服の方針について述べたいと思います．『型システム入門』[7, 8] を或る程度読めるくらいの前提知識を想定して書いています．

1. 多段階計算の基礎

多段階計算は，概して言えば単純型つき λ 計算のような一般的な計算体系に**ステージ** (*stage*) という概念が備わっていて[*70]， ステージがより低い部分にある式から順に計算が進んでいく，という仕組みをもつ計算体系の総称です．典型的には，各ステージは自然数[*71]に対応づけられており，第 n ステージから見て $n < m$ なる各第 m ステージは“生成されるコードの世界”に相当します．式の途中で「ここはステージを上げてコードにする」とか「ここはステージを下げて穴を開け，事前に計算しコードを生成して埋め込む」といった指示が書けるようになっており， これを用いてどの段階でどこの計算が進んでほしいかをプログラマが制御します．

---

70 細かいことを言うと，多段階計算の比較的早期の研究である MetaML [11, 9] では stage と level という概念が区別されていて，ここで言う stage とは MetaML での level に相当するのですが，stage という語の方が現在では定着しているように見受けられるので，ここでは stage の方を採用します．

71 0 を含みます．

多段階の計算体系は，歴史的には与えられたプログラムのうち事前に（卑近な言い方をすればコンパイル時に）計算してよい部分を見つけて計算し実行時にできるだけ無駄な計算をせずにパフォーマンスを上げたいといった動機で研究されている**部分評価** (*partial evaluation*) の文脈で登場し，与えられたプログラムの部分式で事前に計算できる部分をより低いステージに，事前には計算できない部分をより高いステージに割り当てる**束縛時解析** (*binding-time analysis*) という処理の結果出てくる中間表現として用いられたようですが，プログラマが直接手書きするのにも有用であるとして，それ自体をソース言語或いはソース言語に近い中間表現とする研究が発展してきて今に至るようです．

多段階計算には形式化の方法として様々な可能性があり，多段階プログラムの安全性を保証するための研究としては，既存の言語に適用できる何らかの検証手法を提案する（例えばより強力な型システムを与える）というよりも，保証したい性質に合わせて適切に制限された（それでいてできるだけ柔軟にプログラムが書ける）言語を設計し提案する傾向があります．そのため，研究の数だけ少しずつ異なる（構文と意味論さえ違いのある）言語が用意されているとさえ言えるのですが，構文・意味論・型システムを総合しておおよそ次の 3 種類のパターンに大別できると筆者は捉えています：

- コードコンビネータ方式
- $\lambda^{\bigcirc}$ [2] や MetaML [11] に準ずる方式
- $\lambda^{\Box}$ や MetaML$_e^{\Box}$ [3] に準ずる方式

以降でこれらの定式化をそれぞれ簡単に紹介したいと思います．

1.1. コードコンビネータ方式

コードコンビネータ方式は最も簡素なもので，これまで全く多段階計算について親しみのなかった方にとっても最初の理解のステージに立つのに有用と思われるので真っ先に紹介します．大雑把な定式化を採用するなら，コードを以下で定義されるような代数的データ型 code で扱うという方法をとります[*72]．ここでの定式化は（一般の多段階ではなく）2 段階のプログラムを書く用途に特化したものとなっています．実際には λU [1] のよう

72 要するに code 型は式をデータとして表すための型です．いわゆる函数型言語の言語処理系を実装したことのある方ならものすごく見慣れた感じの定義かと思います．

にこの方式の延長で多段階プログラムを扱うこともできますが，簡単のため省きます：

```
type code =
  | Var   of symbol
  | Abs   of symbol * code
  | App   of code * code
  | Int   of int
  | Plus  of code * code
  | Times of code * code
  …
```

ここで `symbol` は生成されるコード中での変数である**シンボル** (*symbol*) につける型で，番号や文字列など何らかの識別子が実体であると考えてください．具体的に 2 段階プログラムを書くにあたっては

```
val gensym : unit -> symbol
```

を用いてシンボルを生成します．これは呼び出しのたびにフレッシュなシンボルを生成してくれるプリミティヴです[*73]．上に掲げた `code` 型の定義の例には，通常の λ 項に対応する `Var`，`Abs`，`App` に加えて，`Int` や `Plus` などの整数に関する算術プリミティヴを表すコンストラクタが備わっていますが，勿論必要に応じて真偽値や文字列など種々のデータ型とそれらに関する演算に拡張することもできますし，形式的に扱う上では簡単のため除去してもかまいません．

このような定式化で実際に 2 段階プログラムを書いた具体例として，非負整数 n を受け取って「n 乗函数のコード」を返す函数 `genpower` を挙げます：

```
let rec aux n x =
  if n <= 0 then Int(1) else
    Times(Var(x), aux (n - 1) x)

let genpower n =
  let x = gensym () in
  Abs(x, aux n x)
```

73 Lisp 系言語の素養があってマクロを実装する読者の方ならまさに `(gensym)` という無引数の函数呼び出しでシンボルを生成しているでしょうから馴染みがあるかと思います．

この genpower は，例えば 3 を渡すと

```
Abs(X, Times(Var(X), Times(Var(X), Times(Var(X), Int(1)))))
```

という，$\lambda m.\, m * m * m * 1$ に相当するコードが生成される，という具合に振舞います．

生成されたコードを用いる方法のひとつとしてラン・プリミティヴ (*Run primitive*) という機構があります．これは与えられたコード断片を使うために下のステージへと“下ろしてくる”仕組みで，例えば

```
let f = run (genpower n) in
…
```

で（n が束縛されている非負整数値を n とすると）f に n 乗函数を束縛して in 以降を評価します．重要なのは，上記の実装が，多段階になっていない power 函数：

```
let rec power n x =
  if n <= 0 then 1 else x * power (n - 1) x
```

を用いて

```
let f = power n in
…
```

と書いた場合と比べると（どんな計算結果を得るかに関して振舞いは同一でも）パフォーマンスの改善が見込めるということです．後者だと f が適用されるたびに n 周再帰が回りますが，前者ではコードが生成されるときに再帰が回るのみであり，そのコードが run で下ろされてくるので f は

```
(fun x -> x * x * … * x * 1)  (* x が n 個並ぶ *)
```

という函数に束縛されることになり，したがって以降 f が適用されるときに逐一再帰が回ったりはしないのです．時間計算量に差が現れるというわけではありませんが，前者は後者と違い再帰を終えるかどうかの条件分岐を判定する必要がないため，f の適用が何度も起こるようなプログラムではその分処理が高速になることが期待できるというわけで

す．勿論後者のようなステージングしない実装でも最適化が効いてナイーヴで処理の重い再帰にならずに済むことはよくありますが，それでも前者のようなステージングを施した実装の方が概して速く動作することは実験的にも確かめられているようです [1]．メタプログラミングの動機が必ずしもパフォーマンスにあるとは限りませんが，少なくともパフォーマンスはひとつのわかりやすい動機ではあるということです．

1.2. ナイーヴなコードコンビネータ方式の問題点

多段階計算のおおまかな仕組みは前節で導入したとおりですが，前節のようなナイーヴなコードコンビネータ方式にははっきりと 2 つの弱点があります：

- 生成されるコードが型のつくコードとは限らない
- 生成されるコードが閉じたコードとは限らない

まず前者に関しては単純で，次のような例が問題になります：

```
Plus(Int(1), True)
```

後者に関しても，次のような例が問題になります：

```
let x = gensym () in run (Var(x))
```

要するに束縛されていないシンボルが出現するコードを走らせてしまえるのです．いずれも，一応言語設計を適切に改めることで防ぐことができます．前者の解決は簡単で，“生成するコード片の型もその整合性を静的に検査” すればよいです．要するに

```
type 'a code
type 'a symbol
```

という 1 つの型を引数にとる型コンストラクタ `code` と `symbol` でそれぞれコードとシンボルを扱うことにし，`Var`，`App`，`Plus`，`Int` `True` などの代わりに

```
val genVar : 'a symbol -> 'a code
val genApp : ('a -> 'b) code -> 'a code -> 'b code
val genPlus : int code -> int code -> int code
```

```
val genInt : int -> int code
val genTrue : bool code
...
```

といったインターフェイスでコード片を生成するための各種プリミティヴをプログラマに提供します．これにより，型のつかないコードを生成してしまう上記のプログラムに対応する以下のようなプログラムを型検査時に弾けます：

```
genPlus (genInt 1) genTrue
```

後者も（可変参照や限定継続といった複雑な言語機能と共存させない限りは）そんなに難しい話ではなく，“そもそもラムダ抽象のコードと同時にしかシンボルを生成しない”ことによって解決します．要するにプログラマには“生の `gensym` と `Abs`”は書けないようにし，代わりに

```
let genAbs k =
  let x = gensym () in Abs(x, k x)
```

なる函数 `genAbs` に相当する機構だけ言語機能として

```
val genAbs : ('a symbol -> 'b code) -> ('a -> 'b) code
```

というインターフェイスで提供すればシンボルが束縛されずに使われる心配はありません．こうした `genApp` や `genAbs` のようなコード生成に関する（コードコンストラクタが型などに関して適切にラップされた）函数を**コードコンビネータ** (*code combinator*) と呼び，これを用いたインターフェイスをもつ多段階計算の定式化を本記事では**コードコンビネータ方式**と呼んでいます．

1.3. $\lambda^{\bigcirc}$・MetaML 方式

（すみません，執筆時間が足りなかったため元論文を参照されたいです ..... 一応，以降の話には理解しておく必要はありません）

1.4. $\lambda^{\square}$・ MetaML$_e^{\square}$ 方式

（すみません，執筆時間が足りなかったため元論文を参照されたいです ..... 一応，以降の話には理解しておく必要はありません）

2. 多段階計算＋可変参照

2.1. 可変参照による問題点の再来

ここまでは多段階の計算体系について紹介するとともに，その過程で如何にして不適格なコードの生成を静的に防げるように構文や型システムが設計されているかについて見てきました．しかし，これらの多段階の計算体系に可変参照をナイーヴに追加すると，型安全性が再び崩れ去ってしまいます．多段階の計算体系に単純に

- **ref** e
- $!e$
- $e := e'$

を追加すると，例えば次のような安全でないコードを生成するプログラムが型検査を通ってしまいます：

```
let r = ref <0> in
let _ = <fun x -> ~(r := <x>; <1>)> in
!r
```

このプログラムは型検査を通りますが，評価すると `<x>`，つまり束縛されていないシンボルの出現だけからなるコードが生成されてしまいます． 要するにシンボル `x` が可変参照 `r` を通じてそのシンボルのスコープではないところに“漏れ出て”しまうのです．この現象を**スコープ脱出** (*scope extrusion*) と呼びます． スコープ脱出をきたす上のようなプログラムを型検査で弾けないと，後のステージの評価中に実際にこのコードが使われた時に自由変数の出現で計算がスタックし実行時エラーを生じてしまいます[*74]．こういったスコープ脱出を起こすプログラムを型検査によって静的に防ぎたいというのが本記事の動機なので

74 実際，OCaml を MetaML と同様の多段階計算で拡張した MetaOCaml [5] の型検査器はここに挙げた問題のプログラムも受理してしまい，自由変数の出現は動的にエラーを吐きます．

した．次節以降ではこの動機に関する既存研究で提案された型システムのうち 2 つを紹介します．さらに，それらのもつ若干の弱点を挙げ，形式化できているわけではないですがその弱点を克服しうる個人的着想を簡単に述べたいと思います．

2.2. 既存研究 1：〈NJ〉

〈NJ〉[6] はコードコンビネータ方式をベースとし可変参照を含む 2 段階計算に対する単相の型システムで，前節で挙げたようなシンボルがそのスコープを漏れ出てしまうプログラムを弾くための機構が備わっています．具体的には**環境分類子** (*environment classifier*) [10] によく似た仕組みを応用することによってスコープ脱出を防ぎます．各コード部分につく型には“（シンボルの出現に関して）それが有効なスコープ”を示す分類子がつけられ，スコープに関して不整合があるプログラム，つまり何らかのコードが本来出現してよいスコープよりも広いスコープで使われようとしているプログラムは，型検査器が弾いてくれるのです．

構文

まず，式，値，型の構文は以下で定義されます：

$$
\begin{aligned}
e &::= c \mid p(e, \ldots, e) \mid l \mid x \mid \lambda x.\, e \mid e\, e \mid \lambda^{*} x.\, e \mid \lambda^{**} x.\, e \mid \langle\!| e |\!\rangle \\
v &::= c \mid l \mid \lambda x.\, e \mid \langle\!| e |\!\rangle \\
\tau &::= b \mid \tau \to \tau \mid \tau \ \mathrm{ref} \mid \langle \tau \rangle^{\gamma}
\end{aligned}
$$

型の構文に含まれるコードの型 $\langle \tau \rangle^{\gamma}$ の γ が重要であり，これが環境分類子に相当するのですが，型については一旦置いておき，先に式の構文と操作的意味論について触れましょう．l は可変参照を扱うための所謂**ロケーション** (*location*) です．$p(e_1, \ldots, e_m)$ はアリティ $m(\geq 1)$ のプリミティヴの適用であり，簡単のため部分適用はできないものとします[*75]．基本型のデータに対する算術演算や，可変参照に対する操作 **ref** e, $!\,e$, $e := e'$ はいずれ

75 部分適用したい場合は **let** $p' = \lambda x_1. \ldots \lambda x_n.\, p(x_1, \ldots, x_n)$ の要領で別の変数に束縛しなおせばよいです．

もプリミティヴ $p(e,\ldots,e)$ の構文に吸収して扱います[*76]．コードコンビネータも原則としてプリミティヴ p として与えられ，適切な型をもちます（後述）．例外はラムダ抽象のコードを生成するコードコンビネータで，シンボルのスコープを適切に制御するために特殊な仕組みが取られています．$\boldsymbol{\lambda}^{*}x.\,e$ がラムダ抽象のための特殊なコードコンビネータで，おおよそ `genAbs (fun x -> e)` に相当し，プログラマがステージ 1 でのラムダ抽象として書くのはこれです．$\boldsymbol{\lambda}^{**}x.\,e$ は評価中にのみ現れるラムダ抽象のコード生成に関する中間生成物であり，こちらはプログラマが直接書くものではありません．$\langle\!|e|\!\rangle$ は“既に完成しているコード片”であり，値であることに注意してください[*77]．なお，元論文 [6] では計算体系に含めている条件分岐は簡単のため省略していますが，自明な拡張で入れることができます．このほか，再帰のための不動点はプリミティヴのひとつとして扱えます．

操作的意味論

さて，それでは操作的意味論を見てみます．評価文脈を使った小ステップ意味論で形式化されています．評価文脈は次で与えられます：

$$E ::= E\,e \mid v\,E \mid p(v,\ldots,v,E,e,\ldots,e) \mid \boldsymbol{\lambda}^{**}x.\,E$$

評価規則は $N \mid \mu \mid e \to N' \mid \mu' \mid e'$ の形で与えられ，N が既に使用したシンボル名を重複防止のために保持しておく集合，μ がいわゆる可変参照のための**ストア** (*store*)（つまりロケーションから値への有限部分写像）です：

$$\frac{N \mid \mu \mid e \to N' \mid \mu' \mid e'}{N \mid \mu \mid E[e] \to N' \mid \mu' \mid E[e']}$$

---

76 記事中の記述では $+(e_1,e_2)$ を $e_1 + e_2$ に，$\mathbf{ref}(e)$ を $\mathbf{ref}\,e$ に，$:=(e,e')$ を $e := e'$ に，といった具合で適宜読みやすい形に書き換えます．

77 感覚的には $\langle\!|e|\!\rangle$ は MetaML の $\langle e \rangle$ と同じですが，e には MetaML の場合と違いエスケープが含まれておらず，それゆえにもう内側が評価されない“既に完成しているコード片”の値なのです．元論文 [6] では MetaML と同様の $\langle e \rangle$ で書かれていますが，誤解をきたさないように記法上区別することにしました．

$$
\begin{aligned}
&N \mid \mu \mid (\lambda x.\, e)\, v \to N \mid \mu \mid [v/x]e \\
&N \mid \mu \mid \mathbf{ref}\, v \to N \mid \mu[l \mapsto v] \mid l && (l \notin \mathrm{dom}\,\mu) \\
&N \mid \mu \mid !\, l \to N \mid \mu \mid v && (\mu(l) \equiv v) \\
&N \mid \mu \mid l := v \to N \mid \mu[l \mapsto v] \mid v && (l \in \mathrm{dom}\,\mu) \\
&N \mid \mu \mid \lambda^{*} x.\, e \to N \uplus \{Y\} \mid \mu \mid \lambda^{**} Y.\, [Y/x]e && (Y \notin N) \\
&N \mid \mu \mid \lambda^{**} Y.\, \langle\!| e |\!\rangle \to N \mid \mu \mid \langle\!| \lambda Y.\, e |\!\rangle
\end{aligned}
$$

ラムダ抽象に関する評価について簡単に述べると，$\lambda^{*} x.\, e$ はステージ 0 の式で，評価されるとフレッシュなシンボル Y を生成して変数 x をそのシンボルに束縛します．すなわち元の式を $\lambda^{**} Y.\, [Y/x]e$ という形に置き換え，本体の $[Y/x]e$ を評価します．評価結果が $\langle\!| e' |\!\rangle$ という形で返ってきたら，ステージ 1 のラムダ抽象 $\langle\!| \lambda Y.\, e' |\!\rangle$ として完成させる，という算段になっています．簡単のため，シンボル（ステージ 1 の変数）もステージ 0 の変数と同一の名前集合を動くものとして扱っています．

ラムダ抽象以外のコードコンビネータはプリミティヴの一部として扱うことにしていましたが，操作的意味論は以下のような要領で定義します（必要になれば同様の方法で定義していくらでも足せます）：

$$
\begin{aligned}
&N \mid \mu \mid \mathit{genInt}(n) \to N \mid \mu \mid \langle\!| n |\!\rangle \\
&N \mid \mu \mid \mathit{genPlus}\big(\langle\!| e_1 |\!\rangle, \langle\!| e_2 |\!\rangle\big) \to N \mid \mu \mid \langle\!| e_1 + e_2 |\!\rangle \\
&N \mid \mu \mid \mathit{genApp}\big(\langle\!| e_1 |\!\rangle, \langle\!| e_2 |\!\rangle\big) \to N \mid \mu \mid \langle\!| e_1\; e_2 |\!\rangle
\end{aligned}
$$

型システム

一番重要なのが型システムです．型判定はステージ 0 が $H \mid \Sigma \mid \Gamma \vdash^{0} e : \tau$，ステージ 1 が $H \mid \Gamma \vdash^{1\gamma} e : \tau$ の形をとります．この 2 つの 5 項関係に於いて，Γ が型環境，Σ が**ストア型つけ** (*store typing*) なのは可変参照をもつ体系の一般的な定式化で馴染みがありますが，ここでは新たに H という形で**名前ヒープ型つけ** (*name heap typing*) という概念が追加されています．名前ヒープ型つけと型環境は互いによく似ており，名前ヒープ型つけは以下の (1) か (2) の形を，型環境は以下の (1)–(3) のいずれかの形を各要素

とする有限列です[*78]：

(1) $\gamma \succ \gamma'$

(2) $(Y : \tau)^{1\gamma}$

(3) $(x : \tau)^{0}$

$\gamma \succ \gamma'$ は直観的には「すでにあるスコープ γ' の “すぐ上” に新しいスコープ γ を設ける」を指します．(2) はステージ 1 での変数の型つけに相当し，γ はその変数が有効なスコープを表します．(3) はステージ 0 での変数の型つけに相当します． 型環境と名前ヒープ型つけには，次のような規則で well-formedness を要請します．簡単に言えば，同一のスコープ名は多重に $\succ$ で新設されたりしないことを表します[*79]：

$$\frac{}{\vdash \varepsilon} \qquad \frac{\vdash H \qquad \mathrm{In}(\gamma', H) \qquad \neg\mathrm{In}(\gamma, H)}{\vdash H \cdot (\gamma \succ \gamma')}$$

$$\frac{\vdash H \qquad \mathrm{In}(\gamma, H) \qquad \neg\mathrm{In}(Y, H)}{\vdash H \cdot (Y : \tau)^{1\gamma}} \qquad \frac{}{H \vdash \varepsilon}$$

$$\frac{H \vdash \Gamma \qquad \mathrm{In}(\gamma', \Gamma, H) \qquad \neg\mathrm{In}(\gamma, H) \qquad \neg\mathrm{In}(\gamma, \Gamma)}{H \vdash \Gamma \cdot (\gamma \succ \gamma')}$$

$$\frac{H \vdash \Gamma \qquad \tau\text{中の各 } \gamma\text{に対して } \mathrm{In}(\gamma, \Gamma, H)}{H \vdash \Gamma \cdot (x : \tau)^{0}} \qquad \frac{H \vdash \Gamma \qquad \mathrm{In}(\gamma, \Gamma, H)}{H \vdash \Gamma \cdot (Y : \tau)^{1\gamma}}$$

$\mathrm{In}(\gamma, \Gamma)$ は γ が Γ で新設されていることを，$\mathrm{In}(\gamma, \Gamma, H)$ は γ が Γ 或いは H で新設されていることを表します． また， ストア型つけは特に変哲なくロケーションに型を紐づける写像です．

名前ヒープ型つけの直観を獲得するのはやや難しいですが，ひとことで言うなら “既にグローバルに確定しているスコープの上下関係を保持している機構” であり， 主簡約定理

78 元論文 [6] とは少しだけ形式化を変えています．ここでの $\gamma \succ \gamma'$ は元論文での $\gamma, \gamma \succ \gamma'$ という 2 要素に相当します．

79 well-formedness の定義は元論文 [6] でもやや記述が不明瞭で，正確か検証できていませんがひとまず掲載しています．

(subject reduction) を証明する上で本質的な役割を果たします[*80]．なお，“最も外側のスコープ” は γ_0 という名前で固定され，必ず名前ヒープ型つけに入っている（ε でも γ_0 は設けられて入っている）という特別な扱いをします．

ともあれ，型つけ規則を見てみましょう（一部省略）：

$$\frac{\mathrm{ConstType}(c) \equiv \tau}{H \mid \Sigma \mid \Gamma \vdash^0 c : \tau} \qquad \frac{\mathrm{ConstType}(c) \equiv \tau}{H \mid \Gamma \vdash^{1\gamma} c : \tau} \qquad \frac{(x : \tau)^0 \text{ in } \Gamma}{H \mid \Sigma \mid \Gamma \vdash^0 x : \tau}$$

$$\frac{(Y : \tau)^{1\gamma} \text{ in } \Gamma, H}{H \mid \Gamma \vdash^{1\gamma} Y : \tau} \qquad \frac{H \mid \Gamma \vdash^{1\gamma} e : \tau}{H \mid \Sigma \mid \Gamma \vdash^0 \langle\!|e|\!\rangle : \langle\tau\rangle^\gamma} \qquad \frac{H \mid \Gamma \vdash^{1\gamma'} e : \tau \qquad H \vDash \gamma \geqslant \gamma'}{H \mid \Gamma \vdash^{1\gamma} e : \tau}$$

$$\frac{H \mid \Sigma \mid \Gamma \vdash^0 e : \langle\tau\rangle^{\gamma'} \qquad H \vDash \gamma \geqslant \gamma'}{H \mid \Gamma \mid \gamma \vdash^0 e : \langle\tau\rangle^\gamma} \qquad \frac{H \mid \Sigma \mid \Gamma \cdot (x : \tau')^0 \vdash^0 e : \tau'}{H \mid \Sigma \mid \Gamma \vdash^0 \lambda x.\, e : \tau' \to \tau}$$

$$\frac{H \mid \Gamma \cdot (Y : \tau')^{1\gamma} \vdash^{1\gamma} e : \tau'}{H \mid \Gamma \vdash^{1\gamma} \lambda Y.\, e : \tau' \to \tau} \qquad \frac{H \mid \Sigma \mid \Gamma \cdot (\gamma \succ \gamma') \cdot (x : \langle\tau'\rangle^\gamma)^0 \vdash^0 e : \langle\tau\rangle^\gamma}{H \mid \Sigma \mid \Gamma \vdash^0 \lambda^* Y.\, e : \langle\tau' \to \tau\rangle^{\gamma'}}$$

$$\frac{\begin{array}{c} H \equiv H_1 \cdot (\gamma \succ \gamma') \cdot (Y : \langle\tau'\rangle^\gamma)^0 \cdot H_2 \\ \forall \gamma''.\, (H \vDash \gamma \geqslant \gamma'' \text{ and } \gamma \not\equiv \gamma'' \Rightarrow H \vDash \gamma' \geqslant \gamma'') \\ H \mid \Sigma \mid \varepsilon \vdash^0 e : \langle\tau\rangle^\gamma \end{array}}{H \mid \Sigma \mid \varepsilon \vdash^0 \lambda^{**} Y.\, e : \langle\tau' \to \tau\rangle^{\gamma'}}$$

スコープの “弱化” に相当する規則（要するに “狭いスコープで有効なシンボルはより広いスコープでも有効” という規則）が両ステージにあるほか，やはりラムダ抽象とそのコードコンビネータが重要です．$\lambda^* x.\, e$ の型検査では x に束縛されるシンボルのために新しいスコープ γ が外のスコープ γ' のすぐ上に新設されて e が走査されます．$\lambda^{**} Y.\, e$ は中間表現なのでプログラマにはあまり関係がなく，主簡約定理を証明するためにある規則です．H はどの規則でも上下で変わっていませんが，これも主簡約定理で簡約に従って適切に大きく取れるようにするための機構です．

---

80 型保存の観点で言うと，プログラムの評価が進むにつれてそのプログラムの型判定に使われる名前ヒープ型つけが大きくなっていきます．詳細に記述するとかなり長くなるので，気になる読者は実際に元論文 [6] の主簡約定理の証明を追うとよいかもしれません．

定理 1 (主簡約)　$H \mid \Sigma \mid \varepsilon \vdash^{0} e : \tau$, $H \vdash \Gamma$, およびストア型つけ Σ が適切に H と Γ に整合しているとき[*81], 或る H', Γ', $\Sigma' \supseteq \Sigma$ が存在して $H \cdot H' \mid \Sigma' \vdash^{1\Gamma \cdot \Gamma'} e : \tau$ かつ $H \vdash \Gamma$ を満たし, Σ' が $H \cdot H'$ と $\Gamma \cdot \Gamma'$ に整合する.

系 2　$\varepsilon \mid \varnothing \mid \varepsilon \vdash^{0} e : \langle \tau \rangle^{\gamma_0}$ および $\varnothing \mid \varnothing \mid e \to^{*} N \mid \mu \mid v$ ならば, v は $\langle\!| e' |\!\rangle$ の形であり, $\varepsilon \mid \varnothing \vdash^{1\varnothing} e' : \tau$ が成り立つ.

以上の定理により型検査を通ったプログラムはスコープ脱出を起こさないことが保証されています. 実際に最初に掲げたスコープ脱出をしてしまう例に相当するプログラム：

$$\left(\lambda r.\ (\lambda a.\ !\, r)\ \left(\lambda^{*} x.\ r := x\right)\right)\ \left(\mathbf{ref}\ \left(genInt(0)\right)\right)$$

を型検査すると, $r := x$ の下りの時点でスコープに不整合が起きて弾かれます. $!\, r$ で使っている箇所が原因となって型エラーが出るわけではないことも重要なのですが, 詳しく触れる執筆時間がなかったので元論文を参照されたいです, すみません.

2.3. 既存研究 2： λ^{poly}_{open}

λ^{poly}_{open} [4] は, 一転して MetaML$^{\square}_{e}$ をベースにした可変参照を含む多段階計算に対する型システムです. 執筆時間の都合であまり詳しく触れることはできませんが, Lisp 諸方言のようにシンボルを陽に名前で扱うことに特徴があり, 環境分類子のかわりに「どんな名前でどんな型をもつシンボルの集合が出現しうるか」の情報をレコード型のような形式でコード型にくっつけて持ち回ります. 詳しくは論文を参照されたいのですが, 特に次のような望ましい性質を満たしていることが大きな有用性です：

- 安全な局所的 run ができる
- 主要型が存在し, 型推論可能

型推論に関しては, **列多相** (*row polymorphism*) の知見を利用することで実現しています. 弱点となりうる点としては, シンボル名を変えると型も変わるので複雑なプログラムの読み書きをしていると変数名を変更してよいのかがすぐにはわからないであろうこと, 型エ

81 ストア型つけの整合性はわりと一般的な定義どおりです.

ラーがかなり煩雑になるであろうことが挙げられます[*82].

2.4. 改善の可能性

前節までに紹介したように，〈NJ〉の型システムはスコープ脱出を静的に防げるのですが，やや厳しすぎる（つまり妥当なプログラムも弾きすぎる）側面があります．例えば，次のプログラムは特にスコープ脱出を起こさないにもかかわらず型検査に通りません：

$$\left(\lambda k.\, \lambda^* x.\, k\, x\right)\left(\lambda y.\, \lambda^* z.\, y\right)$$

要するに $\lambda k.\, \lambda^* x.\, k\, x$ しかし，これが型検査に通らないのも或る程度うなずけることではあります．というのは，次のようなよく似た例がスコープ脱出を引き起こしてしまうためです：

$$\left(\lambda r.\, \left(\lambda k.\, \lambda^* x.\, k\, x\right)\left(\lambda y.\, r := y\right)\right)\left(genInt(0)\right)$$

〈NJ〉の厳しすぎる型検査を或る程度まともに緩和するには，上に掲げた前者のプログラムは通すが後者のプログラムは弾かないといけなさそうなわけです．これを実現するには，単なる函数の型ではなく，所謂“副作用”を引き起こす函数であるかどうかまで型レヴェルで区別する必要がありそうです．したがって

- 可変参照をモナディックに扱うことにする
- type and effect system のようにエフェクト註釈を型につける

などの方法をとることで“純粋な函数か非純粋な函数か”の区別を型レヴェルでしなければならないようです．上記項目に馴染みのない読者は [12] などを参照するとよいかと思います．

また，〈NJ〉には型推論が（少なくともそのままでは）できないという弱点があり，これもできれば克服したいのですが，型推論を実現するように変更を施すにも若干の困難があります．スコープの情報をトラックしなければならないのですが，ラムダ抽象のコードコンビネータを（導出木で下から上に）走査して新しいスコープを設けるときに，つ

82 あくまで可能性です．定性的な話なので，実際に型推論器を実装して使ってみないと実感としてはわかりません．

くられるラムダ抽象全体のコードのスコープは一般には推論中なので，その場では新しいスコープを決め打ちできず，型変数の要領でスコープ変数を導入して制約を生成することになります．このとき，型環境中のすべての型に新たにとられたスコープ変数が含まれてはならないという制約を表現するのにカインドのような機構が必要になりそうです．型推論の途中での“上りがけ”の型環境中ではまだコードの型であると確定していない部分に型変数 α が含まれており，この型変数 α はスコープ γ をもつ型で（制約解消の結果）置き換えられてはならない，という制約を保持する必要があるためです．

3. まとめ

執筆時間の確保不足のためかなり端折ってしまうことになりましたが，本記事では多段階計算の基礎を導入するとともに可変参照と組み合わせたときにどのような問題が生じるかに触れ，それを静的に弾く型システムについて既存研究を紹介しました．既存の型システムについて依然として存在する弱点についても述べ，その大雑把な解決への個人的着想についてごく簡単に触れました．

参考文献

[1] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *International Conference on Generative Programming and Component Engineering*, pages 57–76, 2003.

[2] Rowan Davies. A temporal-logic approach to binding-time analysis. In *IEEE Symposium on Logic in Computer Science (LICS'96)*, pages 184–195, 1996.

[3] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *JACM*, 48 (3), pages 555–604, 2001.

[4] Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for Lisp-like multi-staged languages. In *Proc. of POPL (POPL'06)*, pages 257–269, 2006.

[5] Oleg Kiselyov. The design and implementation of BER MetaOCaml – system description. *LNCS*, pages 86–102, 2014.

[6] Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. Refined environment classifiers: type- and scope-safe code generation with mutable cells. In *APLAS'16*, pages 271–291, 2016.

[7] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[8] Benjamin C. Pierce（住井 英二郎 監訳，遠藤 侑介 訳，酒井 政裕 訳，今井 敬吾 訳，黒木 裕介 訳，今井 宜洋 訳，才川 隆文 訳，今井 健男 訳）. 型システム入門 . オーム社 , 2013.

[9] Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. In *Proceedings of Workshop on Partial Evaluation and Program manipulation (PEPM2000)*, pages 34–43, 2000.

[10] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *Proc. of POPL (POPL'03)*, pages 26–37, 2003.

[11] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of Workshop on Partial Evaluation and Program manipulation (PEPM2000)*, pages 203–217, 1997.

[12] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4(1), pages 1–32, 2003.

モニタリングのための時間オートマトン入門

— 所属性判定 —

MasWag

1. はじめに

本章では、モニタリングのための時間オートマトン入門と題して、時間オートマトンの所属性判定問題について解説します。モニタリングは、広い意味では様々なシステムなどの様子を外部から観察することで、システムの異常などを検知することを指しますが、ここでは**実行時検証** (runtime verification) [6] と同義として扱います。実行時検証とは、システムに要求される**仕様**や、逆に**異常な挙動**の定義を数学の言葉で**形式的に**与え、それをシステムの動作ログと照し合わせることでシステムの異常を検知する手法のことです。実行時検証はモデル検査などの網羅的な形式手法と比べて、網羅性は欠いてしまうものの、複雑すぎてモデル検査がスケールしないシステムや、ブラックボックスなシステムに対しても適応可能であるという特徴があります。実行時検証は数学的には例えば以下の様に定義されます。システムの仕様 φ を記述する論理としては例えば線形時相論理 (Linear Temporal Logic, LTL) や有限オートマトンなどを用いることができます。

実行時検証

入力 : システムの動作ログ w とシステムの仕様 φ

出力 : システムの動作ログ w が仕様 φ を充たすかどうかを判定する。つまり $w \vDash \varphi$ や $w \in \mathscr{L}(\varphi)$ であるかどうかを判定する。

本章ではシステムの仕様 φ を記述する論理として**時間オートマトン**を考えます。時間

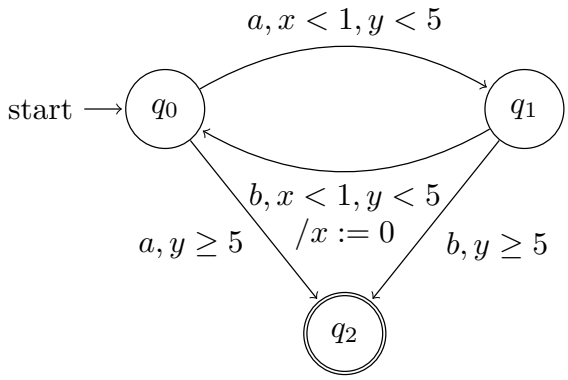


図 1　時間オートマトンの例。5 秒以内の *abab...* の繰り返しで、直近の b からの時間経過が 1 秒以内であるような言語を表現する。

オートマトンは、例えば図 1 にある様に、非決定的有限オートマトン (NFA) に時間についての条件を追加したものです。 入力のログとして、 実数値のタイムスタンプ付きのイベントの列 (**時刻付きワード**、timed word) を考えます。 時間についての制約を用いることで、 例えば「リクエストを送ってから 30 秒以内にレスポンスが返って来ないといけない」 や 「自動車は 10 分以内に 3 回以上車線変更をしてはいけない」 などの仕様を記述することができます。[*83]

モニタリングの文脈では、時刻付きワードを見て「異常」を検出したいのですから、**所属性判定**が重要となります。 本章では、 時間オートマトンの所属性判定問題を解くアルゴリズムと、その計算量を紹介します。

所属性判定問題 (membership problem)

入力 : 文字集合 Σ 上の時間オートマトン $\mathscr{A}$ と時刻付きワード w

出力 : $w \in \mathscr{L}(\mathscr{A})$ であるかどうかを判定する。

83 NFA 等を用いても、例えば「各イベントが 30 秒周期で発生する」などの仮定を用いると上記の仕様を記述することができますが、時間オートマトンなどを用いて時間を陽に扱うことで、より柔軟に時間についての制約を扱うことができます。

例えば決定的オートマトン (DFA) についての所属性判定であれば素朴にオートマトンの上の状態を辿る方法や、遷移行列を考えて行列積を求めていく方法などによって、入力ワード長について線形時間で解くことができます。一方で時間オートマトンの場合は、素朴にオートマトンの上の「状態」を辿る方法だと非決定的分岐によってどのくらいの状態を辿る必要があるのかすぐにはわからないですし、時間制約があるので遷移行列を使って解く方法は、少なくともそのまま適用することはできません。

1.1. 関連する論文・トピック

時間オートマトンの所属性判定問題の計算量解析は [1] において証明されています。所属性判定問題が NP 完全であるという結果は本章で述べているものと同じですが、時間オートマトンの定義が本章のものと異なるため、用いている構成にも若干の違いがあります。また、本章ではモニタリングの文脈を想定しているので、時間オートマトンの大きさについての時間計算量のみではなく、時刻付きワードの長さについての時間計算量も紹介しています。

時間オートマトンなど、時間を陽に扱う仕様を用いたモニタリングは、物理情報システム (Cyber-physical system, CPS) の文脈などで広く研究されています [5]。線形時相論理に対して、時間オートマトンの様に、陽に時間についての制約を加えた論理として**計量時相論理** (Metric temporal logic, MTL)[14] や**シグナル時相論理** (Signal temporal logic, STL)[17] などが知られています。正規表現に対して、時間オートマトンの様に、陽に時間についての制約を加えた論理としては、**時間正規表現** (Timed regular expression, TRE)[4] が知られています。また、シグナル時相論理については、充足するかしないかのブール値を返すのではなく、充足の度合いを実数値で返す、ロバスト意味論も知られており [10]、物理情報システムの反例生成問題 (falsification) などで広く使われています [9, 3]。

これらの論理式を用いる利点としては、オートマトンよりも記述しやすいという点が挙げられます。一方で時間オートマトンを用いることで、NFA や DFA において用いられた技術を流用しやすいということや、そもそも MITL や TRE は時間オートマトンへの変換が可能であるため、共通の基盤言語として時間オートマトンを用いることに大きな意義があると考えられます。

また、時間オートマトンはモデル検査の文脈でも広く用いられており、有名なツールと

しては例えば Uppaal[7] などがあります。

1.2. 本章の構成

本章の構成は以下のようになります。 まず第二節で時間オートマトンの定義を与えます。第三節では所属性判定の定義と、所属性判定の計算量について説明します。

1.3. 記法と仮定

本章では実数の集合を $\mathbb{R}$、 整数の集合を $\mathbb{Z}$ を、 自然数の集合を $\mathbb{N}$ を用いて表わします。 また、 例えば非負実数の集合を $\mathbb{R}_{\geq 0}$ と表記します。 集合 X について、X の冪集合を $\mathscr{P}(X)$ と表記します。 集合 X について、X の元からなる有限列の集合を X^* と表記します。

本章では、「普通の」オートマトンの話 (例えば [12] にあるような内容) は特に説明しません。 オートマトンの理論的な前提知識はあまり必要ないですが、 例えば非決定的分岐の気持ちなどは予めわかっておくと良いかもしれません。また NP 完全問題の定義がわかる程度の計算量の知識を前提とします。 その他よく使われる論理や集合の記号は特に断りなく用います。 また、 本章ではできるだけ用語を日本語に訳す様にしてあります。 時間オートマトンについての用語は定訳がないものも多く、 独自で訳したものもあるのでご了承ください。

2. 時間オートマトン

2.1. 時刻付きワード

先ず、時間オートマトンに与える入力として、**時刻付きワード** (timed word) を定義します。 時刻付きワードで考える文字集合 Σ は、 通常のオートマトンにおける文字列と同様に有限集合ですが、 時刻付きワードでは各文字 $\sigma_i \in \Sigma$ に正実数値のタイムスタンプ $\tau_i \in \mathbb{R}_{>0}$ が付いています。 これは「イベント σ_i が時刻 τ_i で起こった」と読むことができます。

定義 1 (時刻付きワード)　Σ を有限の文字集合とする。各 $i \in \{1, 2, \ldots, n-1\}$ について $\tau_i < \tau_{i+1}$ を満たす様な有限列 $w = (\sigma_1, \tau_1), (\sigma_2, \tau_2), \ldots, (\sigma_n, \tau_n) \in (\Sigma \times \mathbb{R}_{>0})^*$ を**時刻付き**

ワード (timed word) と言う。また、$\tau_0 = 0$ と定める。

例 2　文字集合 Σ を $\Sigma = \{\mathsf{a}, \mathsf{b}\}$ とする。このとき、$w = (\mathsf{a}, 0.1), (\mathsf{b}, 0.2), (\mathsf{a}, 0.8)$ は時刻付きワードである。これは例えば「イベント a が時刻 0.1 で起こり、イベント b が時刻 0.2 で起こり、イベント a が時刻 0.8 で起こった」と読むことができる。一方で $w' = (\mathsf{a}, 0.2), (\mathsf{b}, 0.1), (\mathsf{a}, 0.8)$ や $w'' = (\mathsf{a}, 0), (\mathsf{b}, 0.1), (\mathsf{a}, 0.8)$ は 時刻付きワードではない。[*84]

2.2. 時間オートマトン (構文論)

時間オートマトンでは、有限個の**クロック変数** (clock variable) を用いることで時刻付きワードのタイムスタンプ間の制約を表現します。各クロック変数は時刻付きワードのタイムスタンプの増加と同じだけ増えます。また、遷移の際にはクロック変数の値を 0 にリセットすることもできるので、遷移が起こった際のイベントからの時間経過を表すこともできます。遷移の際にガード制約を用いることで、クロック変数の値が特定の条件を充たすときだけ遷移を許可することができます。クロック変数の集合 C について、以下の BNF で定義される制約の集合を $\Phi(C)$ で表します。但し $c \in C$、$\bowtie \in \{\geq, >, =, <, \leq\}$、$k \in \mathbb{Z}$ とします。

$$g ::= c \bowtie k \mid g \wedge g$$

定義 3 (時間オートマトン)　有限の文字集合 Σ について、**時間オートマトン** $\mathscr{A}$ は 6 つ組 $\left(\Sigma, Q, Q_0, Q_F, C, \Delta\right)$ である。但し、Σ は有限の文字集合、Q は有限の状態集合、$Q_0 \subseteq Q$ は初期状態の集合、$Q_F \subseteq Q$ は受理状態の集合、C は有限のクロック変数の集合、$\Delta \subseteq Q \times \Phi(C) \times \Sigma \times \mathscr{P}(C) \times Q$ は状態遷移集合である。

例 4　図 1 は時間オートマトンの例 $\mathscr{A} = \left(\Sigma, Q, Q_0, Q_F, C, \Delta\right)$ である。但し、$\Sigma = \{\mathsf{a}, \mathsf{b}\}$、$Q = \left\{q_0, q_1, q_2\right\}$、$Q_0 = \left\{q_0\right\}$、$Q_F = \left\{q_2\right\}$、$C = \{x, y\}$、$\Delta = \left\{\left(q_0, x < 1 \wedge y < 5, \mathsf{a}, \varnothing, q_1\right), \left(q_1, x < 1 \wedge y < 5, \mathsf{b}, \{x\}, q_0\right), \left(q_0, y \geq 5, \mathsf{a}, \varnothing, q_2\right), \left(q_1, y \geq 5, \mathsf{b}, \varnothing, q_2\right)\right\}$ である。以

84 定義によっては w'' の様にタイムスタンプが 0 のものや、タイムスタンプが広義単調増加のものもを時刻付きワードに含める場合もあります。

後、$\delta_1 = \left(q_0, x < 1 \land y < 5, \mathsf{a}, \varnothing, q_1\right)$、$\delta_2 = \left(q_1, x < 1 \land y < 5, \mathsf{b}, \{x\}, q_0\right)$、$\delta_3 = \left(q_0, y \geq 5, \mathsf{a}, \varnothing, q_2\right)$、$\delta_4 = \left(q_1, y \geq 5, \mathsf{b}, \varnothing, q_2\right)$ と表記する。

2.3. 時間オートマトン (意味論)

クロック変数の集合 C について、関数 $\nu : C \to \mathbb{R}_{\geq 0}$ を**クロック割り当て** (clock valuation) と呼びます。クロック割り当て ν と $t \in \mathbb{R}_{\geq 0}$ について、$\nu + t$ を用いて、ν を t 時間だけ経過させたクロック割り当てを表記します。つまり、$\nu + t$ は各 $c \in C$ について、$(\nu + t)(c) = \nu(c) + t$ を充たすクロック割り当てです。クロック割り当て ν とクロック変数の部分集合 $R \subseteq C$ について、各 $c \in R$ がリセットされ、それ以外のクロック変数はそのままである様なクロック割り当てを $\langle \nu \rangle_R$ と表記します。つまり、$\langle \nu \rangle_R$ は以下を充たすクロック割り当てです。

$$\langle \nu \rangle_R(c) = \begin{cases} 0 & c \in R \text{のとき} \\ \nu(c) & c \notin R \text{のとき} \end{cases}$$

クロック割り当て ν とクロック制約 g について、ν が g を充たすとき $\nu \vDash g$ と表記し、充たさないとき $\nu \nvDash g$ と表記します。全てのクロック変数 $c \in C$ について 0 を返すクロック割り当てを 0_C と表記します。

時間オートマトンの意味論は**時間状態遷移システム** (timed transition system, TTS) と呼ばれる状態遷移システムを用いて定式化されます。時間状態遷移システムは時間オートマトンの**状況** (configuration) $s \in S$ の移り変りを表現します。時間オートマトンの状況は状態 $q \in Q$ とクロック割り当て $\nu \in \left(\mathbb{R}_{\geq 0}\right)^C$ の対です。直観的には状況 $\left(q, \nu\right)$ は、「状態 q にいて、クロック変数 $c \in C$ の値が $\nu(c)$ である」という状況を表しています。時間状態遷移システムの状況 $\left(q, \nu\right)$ の、クロック割り当て $\nu \in \left(\mathbb{R}_{\geq 0}\right)^C$ は時間 t が経過すると $\nu + t$ へと変化し、ガード条件 g を充たす、つまり $(\nu + t) \vDash g$ でありイベント σ が起こったときに遷移を行うことができ、変数 $c \in R$ がリセットされます。一連の状況の移り変わりを**動作** (run) と呼びます。

定義 5 (時間状態遷移システム)　時間オートマトン $\mathscr{A} = \left(\Sigma, Q, Q_0, Q_F, C, \Delta\right)$ について、**時間状態遷移システム** (timed transition system, TTS) は 4 つ組 $\left(S, S_0, S_F, \to\right)$ であ

る。但し、$S = Q \times \left(\mathbb{R}_{\geq 0}\right)^C$ は**状況** (configuration) の集合、$S_0 = Q_0 \times \left\{0_C\right\} (\subseteq S)$ は**初期状況** (initial configuration) の集合、$S_F = Q_F \times \left(\mathbb{R}_{\geq 0}\right)^C (\subseteq S)$ は**受理状況** (accepting configuration) の集合、$\rightarrow \subseteq S \times S$ は以下で定義される状況遷移関係である。

$$\left(\left(q,\nu\right),\left(q',\nu'\right)\right) \in \rightarrow \Leftrightarrow \exists\left(q,g,\sigma,R,q'\right) \in \Delta, t \in \mathbb{R}_{\geq 0}.(\nu + t) \vDash g, \nu' = \left\langle \nu + t \right\rangle_R$$

例 6　図 1 の時間オートマトン $\mathscr{A} = \left(\Sigma, Q, Q_0, Q_F, C, \Delta\right)$ について、例えば $\left(\left(q_0,\nu\right),\left(q_1,\nu\right)\right)$ や、$\left(\left(q_1,\nu\right),\left(q_0,\nu'\right)\right)$ は $\rightarrow$ に含まれるが、$\left(\left(q_1,\nu\right),\left(q_2,\nu'\right)\right)$ は $\rightarrow$ に含まれない。 但し ν, ν' はそれぞれ、$\nu(x) = 0.2, \nu\left(y\right) = 3.1$ $\nu'(x) = 0, \nu'\left(y\right) = 3.1$ を充たすクロック割り当てである。

定義 7 (動作)　時間オートマトン $\mathscr{A} = \left(\Sigma, Q, Q_0, Q_F, C, \Delta\right)$ の時間状態遷移システム $\left(S, S_0, S_F, \rightarrow\right)$ について、状況 $\left(q,\nu\right) \in S$ と状態遷移 $\left(q,g,\sigma,R,q'\right) \in \Delta$ が交互に表れる列、

$$\rho = \left(q_0,\nu_0\right),\left(q_0,g_1,\sigma_1,R_1,q_1\right),\left(q_1,\nu_1\right),\ldots,\left(q_{n-1},g_n,\sigma_n,R_n,q_n\right),\left(q_n,\nu_n\right)$$

が $\left(q_0,\nu_0\right) \in S_0$、及び各 $i \in \{1,2,\ldots,n\}$ について $\left(\left(q_{i-1},\nu_{i-1}\right),\left(q_i,\nu_i\right)\right) \in \rightarrow$ を充たすとき、ρ を $\mathscr{S}$ 上の**動作** (run) と定義する。 上記の動作 ρ と、Σ 上の時刻付きワード $w = \left(\sigma_1,\tau_1\right),\left(\sigma_2,\tau_2\right),\ldots,\left(\sigma_n,\tau_n\right)$ が、各 $i \in \{1,2,\ldots,n\}$ について、$\left(\nu_{i-1} + \tau_i - \tau_{i-1}\right) \vDash g_i$ と $\nu_i = \left\langle \nu_{i-1} + \tau_i - \tau_{i-1} \right\rangle_{R_i}$ を充たすとき、w が ρ に**対応する**と言う。時間状態遷移システム $\mathscr{S} = \left(S, S_0, S_F, \rightarrow\right)$ 上の動作 $\rho = s_0, \delta_1, s_1, \ldots, \delta_n, s_n$ が $s_n \in S_F$ を充たすとき、ρ を**受理動作** (accepting run) という。

例 8　図 1 の時間オートマトン $\mathscr{A} = \left(\Sigma, Q, Q_0, Q_F, C, \Delta\right)$ について、

$$\rho = \left(q_0,\nu_0\right),\delta_1,\left(q_1,\nu_1\right),\delta_2,\left(q_0,\nu_2\right),\delta_3,\left(q_2,\nu_3\right)$$

は $\mathscr{A}$ の受理動作である。 但し $\nu_0, \nu_1, \nu_2, \nu_3$ は $\nu_0(x) = 0, \nu_0\left(y\right) = 0$ $\nu_1(x) = 0.3$, $\nu_1\left(y\right) = 0.3$ $\nu_2(x) = 0, \nu_2\left(y\right) = 4.2$ $\nu_3(x) = 0.5, \nu_3\left(y\right) = 5.4$ を充たすクロック割り当てで

ある。時刻付きワード $(\mathsf{a},0.3),(\mathsf{b},1.2),(\mathsf{a},5.4)$ は、動作 ρ に対応する。

定義 9 (言語)　時間オートマトン $\mathscr{A}$ の時間状態遷移システムを $\mathscr{S}$ とする。時間オートマトン $\mathscr{A}$ の言語 $\mathscr{L}(\mathscr{A})$ を以下で定義される時刻付きワードの集合と定義する。

$$\mathscr{L}(\mathscr{A}) = \left\{ w \;\middle|\; w \text{ に対応する } \mathscr{S} \text{の受理動作 } \rho \text{ が存在する} \right\}$$

例 10　図 1 の時間オートマトン $\mathscr{A}$ の言語 $\mathscr{L}(\mathscr{A})$ は、

- ababab... という文字の列で、
- 最後の文字の時刻は 5 以上であり、
- それ以外の文字の時刻は 5 未満であり、
- 最後の文字以外の文字について、直近の b (直近の b が存在しない場合は時刻 0) からの時間経過が 1 未満である

ような時刻付きワードの集合である。

3. 所属性判定

所属性判定問題は、 時間オートマトン $\mathscr{A}$ と時刻付きワード w について、w が $\mathscr{A}$ の表わす言語に所属しているかどうかを判定する問題です。 所属判定問題は、 図 2 にあるように、 時刻付きワードの各イベント σ_i とそのタイムスタンプ τ_i について、 状況 $(s,\nu) \in \mathit{CurrConf}$ の移り変わりを追っていくことで解くことができます。NFA について同様のアルゴリズムを用いて所属性判定問題を解く場合、$\mathit{CurrConf}$ は Q の部分集合なので $\mathit{CurrConf}$ の大きさが時間オートマトンの大きさで抑えられます。 一方で時間オートマトンの場合は $\mathit{CurrConf}$ は $Q \times \left(\mathbb{R}_{\geq 0}\right)^C$ の部分集合であり、$\left(\mathbb{R}_{\geq 0}\right)^C$ は一般には無限集合なので $\mathit{CurrConf}$ の大きさを定数で抑えることができません。

3.1. 所属性判定問題の計算量

時間オートマトンの所属性判定問題は NP 完全であることが知られています [1] が、本節ではその証明をします。まずは所属性判定問題が NP に含まれることを示します。

Input: 時間オートマトン $\mathcal{A} = (\Sigma, Q, Q_0, Q_F, C, \Delta)$, 時刻付きワード $w = (\sigma_1, \tau_1), (\sigma_2, \tau_2), \ldots, (\sigma_n, \tau_n)$
Output: $w \in \mathcal{A}$ であるかどうかを返す

```
CurrConf ← {(q_0, 0_C) | q_0 ∈ Q_0}; NextConf ← ∅;
for i ∈ {1, 2, ..., n} do
    for (q, ν) ∈ CurrConf do
        for (q, g, σ, R, q') ∈ Δ do
            if ν + τ_i − τ_{i−1} ⊨ g then
                push (q, ⟨ν + τ_i − τ_{i−1}⟩_R) to NextConf
    CurrConf ← NextConf; NextConf ← ∅;
if ∃(q, ν) ∈ CurrConf. q ∈ Q_F then
    return w ∈ A である
else
    return w ∈ A ではない
```

図 2　所属性判定問題を解くアルゴリズムの例

定理 11　時間オートマトンの所属性判定問題は *NP* に属する。

証明　所属性判定問題の入力の時間オートマトンを $\mathscr{A} = \left(\Sigma, Q, Q_0, Q_F, C, \Delta\right)$、時刻付きワードを $w = \left(\sigma_1, \tau_1\right), \left(\sigma_2, \tau_2\right), \ldots, \left(\sigma_n, \tau_n\right)$ とする。まず始めに、時間オートマトン $\mathscr{A}$ の遷移の列 $\left(q_0, g_1, \sigma_1, R_1, q_1\right), \left(q_1, g_2, \sigma_2, R_2, q_2\right), \ldots, \left(q_{n-1}, g_n, \sigma_n, R_n, q_n\right) \in \Delta^*$ で $q_0 \in Q_0$、$q_n \in Q_F$ を充たすものを非決定的に選択する。次に、制約

$$\mathscr{C} = \left(\nu_0 = 0_C\right) \wedge \bigwedge_{i \in \{1,2,\ldots,n\}} \left(\left(\nu_{i-1} + \tau_i - \tau_{i-1} \vDash g_i\right) \wedge \left(\nu_i = \left\langle \nu_{i-1} + \tau_i - \tau_{i-1} \right\rangle_{R_i}\right)\right)$$

を考える。制約 $\mathscr{C}$ が充足可能であるとき $w \in \mathscr{L}\left(\mathscr{A}\right)$ が成立し、充足不能であるとき $w \notin \mathscr{L}\left(\mathscr{A}\right)$ となる。制約 $\mathscr{C}$ は線形計画問題であるので、例えばカーマーカーのアルゴリズム [13] を用いることで制約の大きさについて多項式時間で解くことができる。[*85]今回制約の大きさは時刻付きワード w の長さと各状態遷移 $\delta \in \Delta$ に含まれる制約 g の大きさ、及びクロック変数 C の個数についての多項式で表わすことができる。従って時間オートマトンの所属性判定問題は NP に属する。 ■

次に所属性判定問題が NP 困難であることを示します。

85 線形計画問題の充足可能性問題は線形計画法で解くことができます。例えば $Ax \geq b$ の充足可能性問題は制約 $Ax + \left(yy \cdots y\right)^T \geq b$ の元で y を最小化することで線形計画法に帰着することができます。具体的には y の最小値が 0 以下のときに $Ax \geq b$ が充足可能となります。

定理 12　任意の *NP* 問題は時間オートマトンの所属性判定問題に帰着できる。

所属性判定問題が NP 困難であることを、**ハミルトニアン路問題**を所属性判定問題に多項式時間帰着させることで示します。ハミルトニアン路とは、有向グラフの路 (path) で、全ての頂点を一度だけ通るようなもののことで、ハミルトニアン路問題は、与えられた有向グラフがハミルトニアン路を持つかどうかを判定する問題です。ハミルトニアン路問題は NP 困難であることが知られているので、この問題を時間オートマトンの所属性判定問題に多項式時間帰着させることで、所属性判定問題も NP 困難であることが示せます。

ハミルトニアン路問題

入力 : 有向グラフ $G = (V, E)$

出力 : 有向グラフ G がハミルトニアン路を持つかどうかを判定する。つまり G 上の路 $v_0, e_1, v_1, \ldots, e_n, v_n$ で全ての頂点を一度だけ通るものが存在するかどうかを判定する。

定理 12 の証明　ハミルトニアン路問題を時間オートマトンの所属性判定問題に多項式時間帰着させることで定理 11 を示す。ハミルトニアン路問題の入力の有向グラフを $G = (V, E)$ とする。時間オートマトン $\mathscr{A} = \left(\Sigma, Q, Q_0, Q_F, C, \Delta\right)$ を以下の様に定義する。

- $\Sigma = \{\sigma\}$
- $Q = V \sqcup \left\{q_f\right\}$
- $Q_0 = V$
- $Q_F = \left\{q_f\right\}$
- $C = \left\{c_v \mid v \in V\right\} \sqcup \{y, z\}$
- $\Delta = \Delta_E \sqcup \Delta_f$
- $\Delta_E = \left\{\left(v, \{y = 1\}, \sigma, \{y, c_v\}, v'\right) \;\middle|\; (v, v') \in E\right\}$

- $\Delta_f = \left\{ \left(v, \left\{ y = 1, c_v = z = |V|, \bigwedge_{v' \in V \setminus \{v\}} c_{v'} < |V| \right\}, \sigma, \varnothing, q_f \right) \middle| v \in V \right\}$

有向グラフ $G = (V, E)$ がハミルトニアン路を持つことは、時刻付きワード

$$w = (\sigma, 1), (\sigma, 2), \ldots, (\sigma, |V|)$$

が $w \in \mathscr{L}(\mathscr{A})$ を充たすことの必要十分条件である。また、$\mathscr{A}$ 及び w の大きさは G の多項式で抑えることができる。従って、ハミルトニアン路問題を時間オートマトンの所属性判定問題に多項式時間帰着させることができる。 ■

例 13　定理 12 の証明で用いたクロック変数の直観は、y を用いて全ての遷移を時間差 1 で起こる様にし、z がこれまでに通過した頂点の個数、c_i が頂点 q_i を通過してからの経過時間を表わすというものである。従って、$\mathscr{A}$ の受理動作における各制約は以下のハミルトニアン路の特徴付けと対応する。

- グラフのどの頂点から始めても良い ($Q_0 = V$)
- グラフのどの頂点で終わっても良い (各状態 $v \in V$ から受理状態 v_f への遷移がある)
- 頂点を $|V|$ 回通過している ($z = |V|$)
- 最後に通過した頂点以外は過去に一度以上通過している ($\bigwedge_{v' \in V \setminus \{v\}} c_{v'} < |V|$)
- 最後に通過した頂点はこれまで一度も通過していない ($c_v = |V|$)

図 3 に本構成の例がある。

定理 11 及び定理 12 より、以下が成り立ちます。

系 14　時間オートマトンの所属性判定問題は *NP* 完全問題である。

3.2. オートマトンを固定したときの所属性判定問題

モニタリングの文脈では、モニタリングする仕様を表わす時間オートマトン $\mathscr{A}$ は固定されていて、モニタリング対象のログを表わす時刻付きワード w が長くなっていくよう

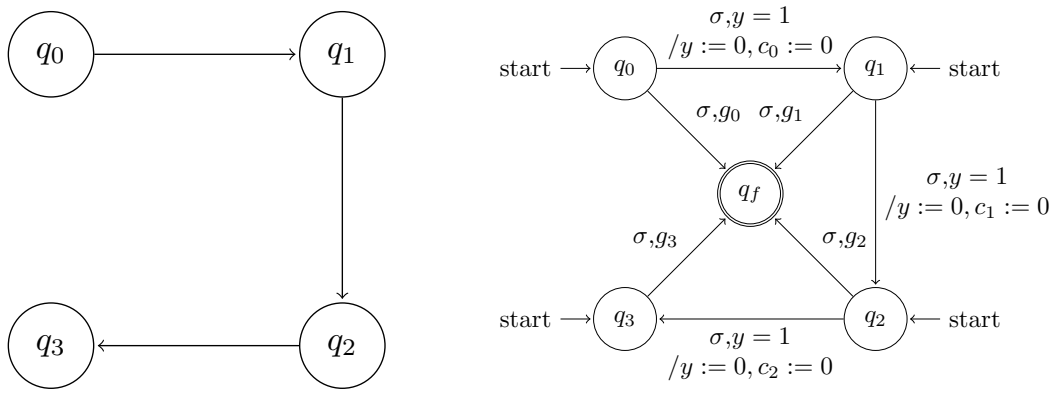


図 3　有向グラフ G (左) と G がハミルトニアン路を持つことを調べるための時間オートマトン $\mathscr{A}$(右)。但し g_i は制約 $g_i = \big(y = 1\big) \wedge \big(c_i = z = 4\big) \wedge \bigwedge_{j \in \{0,1,2,3\}, i \neq j} \big(c_j < 4\big)$。

な問題設定を考えることが自然です。時間オートマトンの所属性判定問題は、時間オートマトン $\mathscr{A}$ を固定した際には**多項式時間**で解くことができることがわかります。これは直観としては以下の理由によるものです。各イベントとタイムスタンプ $\big(\sigma_i, \tau_i\big)$ を読んだとき、各クロック変数 $c \in C$ について、値 $\nu(c)$ は直近で c がリセットされた時刻 τ を用いて $\nu(c) = \tau_i - \tau$ と表わすことができますが、変数のリセットは過去に読んだイベントと同時にしか起こらないため、各クロック変数のの値の取り方は入力の時刻付きワード w の長さ $|w|$ 通りで抑えることができます。従って、図 2 の疑似コード中の $\mathit{CurrConf}$ の大きさは $\big|Q\big| \times |w|^{|C|}$ で抑えられ、現在クロック変数 C は固定されているので、$\mathit{CurrConf}$ の大きさは時刻付きワード w の長さの多項式で抑えることができます。全体としては、$\mathit{CurrConf}$ の更新は $|w|$ 回行われますが、結局 $|w|$ についての多項式時間で所属性判定問題を解くことができます。

定理 15　入力の時間オートマトンを固定したとき、時間オートマトンの所属性は時刻付きワード w の長さについて多項式時間で判定することができる。

証明　図 2 のアルゴリズムが時刻付きワードの長さについて多項式時間で停止することを示す。

まず、疑似コードの 2 行目から 7 行目のループの先頭において、以下の式が不変条件となっていることを示す。

$$\forall\left(q,\nu\right) \in Conf, c \in C, \exists j.\nu(c) = \tau_{i-1} - \tau_j$$

$i = 1$ のとき、$CurrConf = \left\{\left(q_0, 0_C\right) \;\middle|\; q_0 \in Q_0\right\}$ であるので、任意の $c \in C$ について $0_C(c) = 0 = \tau_{1-1} - \tau_0$ が成り立つ。$i > 1$ のとき、$CurrConf$ が更新されるのは図 2 の 7 行目であるが、ここでは $NextConf$ の内容に更新されているので、6 行目で $NextConf$ に追加される $\left(q, \left\langle \nu + \tau_i - \tau_{i-1} \right\rangle_R\right)$ について、$\forall c \in C, \exists j.\left\langle \nu + \tau_i - \tau_{i-1} \right\rangle_R(c) = \tau_i - \tau_j$ が成り立つことを示す。各 $c \in R$ について、$\left\langle \nu + \tau_i - \tau_{i-1} \right\rangle_R(c) = 0 = \tau_i - \tau_i$ となる。各 $c \notin R$ について、$\left\langle \nu + \tau_i - \tau_{i-1} \right\rangle_R(c) = \nu + \tau_i - \tau_{i-1}(c) = \nu(c) + \tau_i - \tau_{i-1}$ となる。帰納法の仮定より、$\forall c \in C, \exists j.\nu(c) = \tau_{i-1} - \tau_j$ が成り立つので、各 $c \notin R$ について $\exists j.\left\langle \nu + \tau_i - \tau_{i-1} \right\rangle_R(c) = \tau_{i-1} - \tau_j + \tau_i - \tau_{i-1} = \tau_i - \tau_j$ となる。

このとき、$CurrConf$ の要素数は、$\left|Q\right| \times \left(\left|w\right| + 1\right)^{\left|C\right|}$ で抑えられる。従って 3 行目から 6 行目のループの回数は $\left(\left|Q\right| \times \left(\left|w\right| + 1\right)^{\left|C\right|}\right) \times \left|w\right|$ で抑えられる。それ以外の集合演算や $\nu \vDash g$ の計算時間、及び Δ についてのループの回数は w の長さに依存しないので、図 2 のアルゴリズムは時刻付きワードの長さについて多項式時間で判定することができる。 ■

3.3. 決定的時間オートマトンの所属性判定問題

3.2 節で示した様に、時間オートマトンの所属性判定問題は、入力の時間オートマトン $\mathscr{A}$ を固定した場合、時刻付きワード w の長さについて多項式時間で解くことができます。一方でこれは実際にモニタリングを行なう場合、単位時間当たりにモニタリングできる時刻付きワードの長さが徐々に短かくなるということになるので好ましくありません。時間オートマトンが**決定的**である場合には所属性判定問題を線形時間で解くことができる、つまり単位時間当たりにモニタリングできる時刻付きワードの長さが一定であるということがわかります。

定理 16　時間オートマトン $\mathscr{A} = \left(\Sigma, Q, Q_0, Q_F, C, \Delta\right)$ が決定的であるとき、つまり

- Q_0 が単元集合で、

- 各状況 $(q, \nu) \in Q \times (\mathbb{R}_{\geq 0})^C$ と $t \in \mathbb{R}_{>0}$ について、状態遷移 $(q, g, \sigma, R, q') \in \Delta$ で $(\nu + t) \vDash g$ を充たすようなものが高々一つしか存在しないとき、

時間オートマトンの所属性は時刻付きワード w の長さについて線形時間で判定することができる。

証明　図 2 のアルゴリズムが時刻付きワードの長さについて線形時間で停止することを示す。

まず、疑似コードの 3 行目において $CurrConf$ の要素数が高々 1 であることを帰納法で示す。$i = 1$ のとき、$CurrConf = \left\{ \left(q_0, 0_C\right) \middle| q_0 \in Q_0 \right\}$ であり、Q_0 は単元集合であるので、$CurrConf$ も単元集合である。また、$NextConf$ は空集合である。$i > 1$ のとき、$CurrConf$ が更新されるのは図 2 の 7 行目であるが、ここでは $NextConf$ の内容に更新されている。各状況 $(q, \nu) \in CurrConf$ について、状態遷移 $(q, g, \sigma, R, q') \in \Delta$ で $(\nu + \tau_i - \tau_{i-1}) \vDash g$ を充たすようなものは高々一つしか存在しないので、6 行目で $NextConf$ に追加される $\left(q, \langle \nu + \tau_i - \tau_{i-1} \rangle_R\right)$ も高々一つしか存在しない。帰納法の仮定より状況 $(q, \nu) \in CurrConf$ は高々一つしか存在しないので、$CurrConf$ の要素数も高々 1 となる。

従って 2 行目から 7 行目のループの計算量は、5 行目の線形制約の充足判定の $|\Delta| \times |w|$ 倍で抑えられる。それ以外の集合演算や $\nu \vDash g$ は w の長さに依存しないので、図 2 のアルゴリズムは時刻付きワードの長さについて線形時間で判定することができる。 ■

時間オートマトンが決定的であるというのは厳しい制約に見えるかもしれません。実際、(非決定的) 時間オートマトンが表現する言語のクラスは決定的時間オートマトンが表現する言語のクラスより真に大きいことが知られています [2]。一方で、例えば図 1 の時間オートマトンや、それ以外にも様々なオートマトンが決定的であることが知られています。

4. まとめ

本章ではモニタリングのための時間オートマトン入門ということで所属性判定問題の計算量を紹介しました。所属性判定問題は NP 完全ですが、モニタリングのためという観点で入力の時間オートマトンを固定した場合では、所属性判定問題は入力の時刻付きワードの長さについて多項式時間で解け、特に時間オートマトンが決定的である場合には時刻

付きワードの長さについて線形時間で解くことができます。従って、決定的時間オートマトンによって記述された仕様についてのモニタリングは効率的に解けることが期待できます。[*86]

(非決定的) 時間オートマトンが表現する言語のクラスは決定的時間オートマトンが表現する言語のクラスより真に大きいため [2]、NFA を DFA に変換するように、(非決定的) 時間オートマトンを決定的時間オートマトンに変換することは一般にはできませんし、そもそも (非決定的) 時間オートマトンを決定化可能であるかを判定することすら決定不能です [11]。一方で、(非決定的) 時間オートマトンを近似的に決定化する方法として、one-clock determinization[15]、や入力の時刻付きワード w の長さを制限する bounded determinization [16] などの手法が知られています。これらの手法を使うことで、誤検出が許される場合や入力の時刻付きワードの長さに制限がある状況においては、一般の時間オートマトンについてある程度効率良くモニタリングを行うことができることが期待できます。

また、NFA や DFA において培われてきた様々なアルゴリズムを時間オートマトンに適用することで、モニタリング問題に限らず時間オートマトンについての諸問題をより効率的に解くことは非常に興味深いと思われます。

参考文献

[1] Rajeev Alur, Robert P. Kurshan, Mahesh Viswanathan. Membership Questions for Timed and Hybrid Automata.. In *RTSS*, pages 254–263, 1998.

[2] Rajeev Alur, David L. Dill. A Theory of Timed Automata.. *Theor. Comput. Sci.*, 126(2), pages 183–235, 1994.

[3] Yashwanth Annpureddy, Che Liu, Georgios E. Fainekos, Sriram Sankaranarayanan. S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems.. In *TACAS*, pages

86 時刻付きワード w の長さについて線形時間で解くことができると言っても、定数部分が大きい場合など現実的にあまり効率的に所属性判定ができない場合もあります。また、時間パターンマッチング問題 [18, 19] など、モニタリングによって解きたい問題によっては決定的であることより強い制約がないと、時刻付きワード w の長さについて線形時間でモニタリングすることができない場合もあります。

254–257, 2011.

[4] Eugene Asarin, Paul Caspi, Oded Maler. Timed regular expressions.. *J. ACM*, 49(2), pages 172–206, 2002.

[5] Ezio Bartocci, Jyotirmoy V. Deshmukh, Alexandre Donzé, Georgios E. Fainekos, Oded Maler, Dejan Nickovic, Sriram Sankaranarayanan. Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications.. In *Lectures on Runtime Verification*, pages 135–175, 2018.

[6] Ezio Bartocci, Yliès Falcone. *Lectures on Runtime Verification - Introductory and Advanced Topics*. Springer, 2018.

[7] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, Martijn Hendriks. UPPAAL 4.0.. In *QEST*, pages 125–126, 2006.

[8] Johan Bengtsson, Wang Yi. Timed Automata: Semantics, Algorithms and Tools.. In *Lectures on Concurrency and Petri Nets*, pages 87–124, 2003.

[9] Alexandre Donzé. Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems.. In *CAV*, pages 167–170, 2010.

[10] Alexandre Donzé, Oded Maler. Robust Satisfaction of Temporal Logic over Real-Valued Signals.. In *FORMATS*, pages 92–106, 2010.

[11] Olivier Finkel. Undecidable Problems About Timed Automata.. In *FORMATS*, pages 187–199, 2006.

[12] J. ホップクロフト , R. モトワニ , J. ウルマン . オートマトン言語理論計算論 *I*. サイエンス社 , 2003.

[13] Narendra Karmarkar. A new polynomial-time algorithm for linear programming.. *Combinatorica*, 4(4), pages 373–396, 1984.

[14] Ron Koymans. Specifying Real-Time Properties with Metric Temporal Logic.. *Real-*

Time Systems, 2(4), pages 255–299, 1990.

[15] Moez Krichen, Stavros Tripakis. Conformance testing for real-time systems.. *Formal Methods in System Design*, 34(3), pages 238–304, 2009.

[16] Florian Lorber, Amnon Rosenmann, Dejan Nickovic, Bernhard K. Aichernig. Bounded determinization of timed automata with silent transitions.. *Real-Time Systems*, 53(3), pages 291–326, 2017.

[17] Oded Maler, Dejan Nickovic. Monitoring Temporal Properties of Continuous Signals.. In *FORMATS/FTRTFT*, pages 152–166, 2004.

[18] Dogan Ulus, Thomas Ferrère, Eugene Asarin, Oded Maler. Timed Pattern Matching.. In *FORMATS*, pages 222–236, 2014.

[19] Masaki Waga, Ichiro Hasuo, Kohei Suenaga. Efficient Online Timed Pattern Matching by Automata-Based Skipping.. In *FORMATS*, pages 224–243, 2017.

忙しいエンジニアのための Twitch ゲーム配信観戦ガイド

IrnBru

1. はじめに

忙しいエンジニアの皆様こんにちは。この文章を読んでいるということはこの本もそろそろ一通り読み終わり、充実した時間を噛み締めながら本記事をお読みになっていることだろう。執筆陣による渾身の記事の数々はいかがだっただろうか、さぞかし身にしみたものと思う。いやいや、もしくは何となく気になって手に取ってみたものの、意味が分からなくてとりあえず読めそうなところまで読み飛ばしここへ辿り着いたのかもしれない、私だ。本稿ではここまで読み進めた (あるいは読み飛ばした) 諸氏が忙しい現代人であり、ストレスに晒されながら日々を過ごしていることを考慮し、「エンジニアのために金も体力も使わない娯楽」を紹介したいと思う。

「オイオイオイお前この記事のラインナップで突然そんな話し始めるなんて正気か？」とお思いのことだろう、私もだ。ガチのテック記事を期待してた諸氏にとっては目を疑う事態だろうが、信じられないのは私の方なんだよ。半ば冗談のはずが割と本気で掲載許可が下りたのだから、ヤバイテックトーキョーは懐の深い同人誌とご理解いただきたい。だがしかし、ヤバイテックトーキョーの掲げる「コンピューターサイエンスのあらゆるジャンル」のうち「その他」を担う者として、忙しい現代エンジニアの諸氏の休憩時間や作業中の一時の癒しとなりうる娯楽をお伝えするという建前で、私の趣味全開のただのオタクトークを始めさせていただこう。

2. Twitch とは

さて、先ほどから言っている「娯楽」とは何なのか。そう、Twitch である。

「Twitch って何だよ！」と思っている諸氏のために簡単に説明をすると、Twitch とは「ゲーム専用の個人向けストリーミングプラットフォーム」である。要は友達が家でゲームしているのを中継しているようなものだ。是非"Twitch"でググってトップ画面から"コンテンツを探す"をクリックしてみて欲しい。すると数々のゲームが羅列されているではないか！後は簡単、「お、面白そうじゃーん」と思ったゲームをクリックし、目に入ったテキトーな配信者のページに飛ぶだけ。これだけで「こっちはこっちのしたいことしてるけどなんか友達はゲームしてるからたまにチラッと見れる空間」が出来上がるのである！簡単！

なお、基本的に配信カテゴリはゲームが中心であるが、別にゲームに限られるわけではない。雑談するだけの"Just Chatting"といったカテゴリがあったり、"Music & Performing Arts"というカテゴリでいわゆる DJ を個人でやっているチャンネルもあったりする。このように、Twitch はゲームだけに止まらず幅広いクリエイティブ、エンターテイメントの配信を支援しているサービスであるとおわかりいただけただろう。

ゲーム配信の話に戻ると、基本的に人気のある配信は「世界的なオンラインゲーム・大きな大会が開催されている際の対戦ゲーム・最新のビッグタイトル」である。

「世界的なオンラインゲーム」とは、皆様も一度くらいは耳にしたことがあるだろう"League of Legends (LoL)"や"PLAYERUNKOWN'S BATTLEGROUNDS (PUBG)", "Hearthstone"といったゲームである。そう、お気付きの通り、Twitch というプラットフォームがグローバルなサービスであるがために、日本と海外とで人気度合いに極端な開きがあるゲームの配信が多い。(それでも各ゲーム一定のプレイヤーが日本にも存在しているので不人気というわけではない、念のため) しかしまあ視聴者として観るぶんには別に有名かどうかなんて関係ないので、面白そうだなーと思ったら雰囲気で観始めれば良いと思う、私はそうしている。

「大きな大会が開催されている際の対戦ゲーム」とは上述のゲームも含まれるが、それに加え"ストリートファイター V"や"StarCraft II"といった対戦型ゲームを含む。普段はあまり視聴者が多くないが、大規模な大会が Twitch で配信される場合に数万人単位で視聴者が集まり非常に賑わう。"コンテンツを探す"に並んでいるゲームのソート順は各ゲームの総視聴者数順になっているが、いつもであれば見かけない対戦ゲームに急に数万人の視聴者がいる場合は大抵大会が開かれていると考えてよい。こうした経路で筆者が出会った

ゲームが"Rocket League"だ。「地を駆け空を飛ぶスーパーカーでアクロバティックなサッカー」をするという、頭のネジを車のパーツにしてるとしか思えない発想のゲームである。いや、実を言うとゲーム自体は知っていたのだが、大会シーンが存在していてトッププレイヤーの動きが芸術と言える域に達しているとは微塵ほども考えたことはなかった。言葉での説明は不可能なので、気になった諸氏は是非"Rocket League esports Twitch"で検索してみよう。(Twitch を入れないと大抵 YouTube が上位の動画に来るがこの記事は Twitch を応援しております)

更に、あの"大乱闘スマッシュブラザーズ DX" (GC のスマブラ) は今でも大規模な世界大会が開かれており、主に北米で根強く熱狂的な人気を誇っている。大きい大会では参加プレイヤーが 500 - 1000 人規模になることもあるので見応えも十分だ。過去にスマッシュブラザーズ DX (海外版のタイトルは"Super Smash Bros Melee"なことに注意) をプレイしたことのある人が前知識なく大会の試合を観ると、その動きの異次元さに驚愕すること間違いなしである。気になる方は是非"evo2018 smash melee"などでググってみて欲しい。ほぼ間違いなく YouTube に上がっている動画が引っかかるので「Tiwtch じゃないじゃん！」となるだろうが細かいことを気にする記事ではない。話が横道に逸れたが、各ゲームのカジュアルプレイヤー諸氏にとって大会は千載一遇のチャンスであり、一級のエンターテイメントなのでこぞって視聴してみて欲しい。

というわけで、大規模な対戦ゲームの大会に関しては大会運営が公式に Twitch にて配信を行うことが多い。是非タイムゾーンの違いに負けず生活を壊すことで Live で大会視聴をして観て欲しい。壊したくない各員に関しては大会配信をしていたチャンネルの"ビデオ"から過去の大会映像をチェックしよう！

最後に最新のビッグタイトルだが、本稿執筆時点 (2019/3/31) で言うと"ソウルシリーズ"の流れを汲んだフロムゲー最新作、"Sekiro: Shadows Die Twice"だろう。発売時点では多くの配信者が配信をし、総計 10 万人を超える視聴者が存在していた。発売から 2 週間ほど経った現在も常時数万人程度の視聴者が存在すると言う盛況ぶりである。

ここ数ヶ月以内の最新のゲームで人気があるものは"Apex Legends"という、いわゆるバトルロワイヤル形式の FPS だろう。プレイヤーの操作キャラクターが複数いる (現在は 9 名) 中から選択可能なのだが、各キャラクターがそれぞれ異なるアビリティを持っているのが特徴。3 人 1 組のチームを組んで合計 60 人の戦場に放り込まれることになるが、いわ

ゆる引きこもり的な戦法より積極的に戦いに行く戦法が強かったり、チームのメンバーが死亡しても条件を満たせば復活させることが可能であることなどがプレイヤーの中で好評である様子だ。過去 PUBG 等のバトロワゲーをプレイして来た人々の間でも非常に評価の高い作品で、今後もこの人気が続きそうだ。

以上の通り、配信される作品は偏っているように見えて多岐に渡る上、上位陣には及ばないものの視聴者数が 1000 ~ 10000 のゲームも根強い人気のあるものが多い。きっと諸氏の気に入るゲームの配信者が見つかるはずなので一度眺めてみてはいかがだろうか。

3. Twitch Prime

本稿において、読者諸氏にもっともインパクトがあるのはこの章だろう。何故ならそこの Amazon Prime に加入している君！なんと Amazon Prime に加入していると無料で Twitch Prime にも加入できちまうんだ！

(Twitch は Amazon に買収をされたサービスであり、過去 Twitch Prime に日本人は加入できないいわゆるおま国状態にあったが、Amazon Prime 経由で Twitch Prime に加入するという回り道が用意された)

Twitch Prime に加入していると何ができるかって？なんと無料で配信者の配信をサブスクライブできちまうんだ！サブスクライブをすると配信者を金銭的に支援できることはもちろん、配信者ごとに用意している「サブスクライバー特典」が付与されたり「配信中の広告を非表示」にしたりできるようになっちまうぜ！配信者の応援、しような！

おっと、「俺に直接メリットはないのか」って？もちろんある！ Twitch Prime 特典として「各種オンラインゲームのゲーム内コンテンツ」や「インディーズゲームを丸っと」なんてもんが貰えちまうんだ！

さらにビッグな特典が最近追加された ... そう！ "Nintendo Switch Online 会員権 最大 12 ヶ月無料"だ！！おいおいマジかよ Amazon Prime の月額会員権が 400 円で Nintendo Switch Online 会員権が月額 300 円だから ... 差し引きで ...Amazon Prime 会員権がほぼ無料！！すげぇぜ！ Nintendo に魂捧げたお友達はこのチャンスを逃す手はない！ Twitch 公式サイトにすぐアクセスだ！ 2019 年 9 月 24 日までがキャンペーン期間である様子、うかうかして登録してなかったなんてことのないようにな！

と言うわけで、無料サブスクライブ権だけでも実質 Amazon Prime の月額料金がほぼ全額返ってくるようなものなのになんとインディーズゲームも無料になったりその他アイテムが無料になったりといいことづくめなので、とりあえず Twitch Prime に登録だけでもしてみるのはいかがだろうか。

4. オススメ配信の紹介

さて、ここからの紹介はさらに危険である。何故なら個別の配信者に対して無許可で言及をするためである。私が普段から気分で見ているチャンネルの数々でありだからこそオススメしたいというわけだが、いかんせん配信者には無許可である。この記事を読む人間の数や実際に観に行く人間の数を考えるとこんな注意書きをするのはおこがましいが、万が一諸氏が本稿を読んだ後に配信に興味を持った場合には、間違っても「ヤバイテックトーキョーで紹介されてたから来ました！！！」というようなヤバイコメントを残さないようにお願いしたい。

一方で、雰囲気が気になるなーなんて魔が差したら是非ちらりとでも覗いてみて欲しい。配信者も現実に生きる人間であるため観に行こうと思った時間には配信をしていないこともあるが、配信チャンネルの個別ページにある"ビデオ"という欄をクリックすると過去の配信を確認することができる。配信がされていなかったらそちらを確認してみてはいかがだろうか。

4.1. rainbrain

https://www.twitch.tv/rainbrain

主にプレイするゲーム : LoL, 人気の FPS (最近は主に Apex), 話題になったゲーム

本稿を書く意義・モチベーションの 3 割は彼の存在を紹介しておきたいというところから来ている。主にプレイするのは LoL で、数年前の日本 LoL プロシーン黎明期にプロチームに所属しプレイをしていた。現在はプロシーンからは引退をし、Twitch でストリーマーをしている。ロールは主にジャングラーをやっているが、トップレーンを始めとし時期と気分によって割と他のロールもプレイする。FPS もよくプレイをしており、ここ数年のバトロワ系ゲームの人気が来た時には PUBG をプレイ、最近ではもっぱら Apex を配信している。

彼の配信は比較的落ち着いており、筆者は作業のお供にすることが多い。盛り上がりすぎず、盛り下がりすぎず、たまにあるいい感じに上手いプレイをおお！とか言いながら観るのが心地よい。

が、彼の配信の真骨頂はその配信時間の長さにある。彼の配信は一度始まるとほぼ丸一日配信が続く。朝に「お、配信してるな？」と思い外へ出た後、夕飯時に家路につくとなんと「お、配信してるな？」となる。王様のブランチも真っ青だ。LoL のプレイ時には顕著であり、平時から 20 時間近く配信を行い、しかも多い場合にはそれを 1 日おき程度にする。人呼んで配信モンスター。本人曰く「一度配信を始めたら短く切りたくないんだよねー」とのこと。

Tiwtch の配信者はコメント欄にチャットボットを設置しておくことも多く、Nightbot はよく使用されるボットなのだが、そのコマンドの中に"!uptime"という配信を始めてからの配信時間を表示するコマンドが存在する。過去、彼があまりに長時間の配信をすることに多くの視聴者が配信時間を気に掛け"!uptime"をすることが増え、当然配信開始から 10 時間だとか 20 時間だとか何度も表示されるに至った。あまりに長い配信時間から彼の体調を心配するコメント等もその度に流れるわけだが、あまりに頻繁に心配をされることに疲れ切った彼は"!uptime"コマンドの実装を変更するに至った。現在、彼の配信で"!uptime"と打ち込むと"Twitch Prime ならサブスクライブが無料！"と Twitch Prime 加入へのリンクと共に表示されるようになっている。その表示に合わせて「無料でサブスクライブできちまうんだ！」というようなコメントが流れるのがお約束のようなものになっている。

と、このようなエピソードが存在するほど長い配信をするので、一度彼が配信を始めると安心して「あ、あとで暇になったら観よ」と思いながら外に出かけることも可能である。時折味方に悪態をつくのもご愛嬌、配信モンスターも実は人間だ。いつ観に行っても変わらない居心地の良い配信で、さらにポイントを挙げると声が良い。また、新しいゲームをプレイし始める時もべらぼうに長い時間プレイするので、気になるゲームがどんな雰囲気かを観るには絶好の配信と言える。

筆者一押しの配信者なので是非気分が向いたら一度覗いてみて欲しい。

4.2. AKAsurvive

https://www.twitch.tv/akasurvive

主にプレイするゲーム : PUBG, Apex, CSGO, その他 FPS 全般

FPS のガチ勢。ニコニコ動画および YouTube に投稿されている動画シリーズ、「仲間任せの茜ちゃんサバイボゥ」にてフォローがとても上手なプレイヤーとして紹介され「フォロマキ」の名で親しまれる。(なお、この「マキ」の中の人は"Rumad"というプレイヤーであり、動画の音声を当てる際にボイスロイドの弦巻マキの声が当てられたことからマキちゃんと呼ばれる) AKASurvive というチャンネルで配信を行っているが、過去には上記動画の流れを汲んで「マキちゃん」として地声で配信を行っていたが、現在は主に"Rumad"というハンドルネームで男性声にボイチェンをして配信を行っている。バトロワ系のゲームで絶対的な強さを見せつけるため、FPS はエアプ勢の筆者でもその上手さ・強さが際立ってわかる。配信を適当に眺めているだけでスーパープレイ動画が楽しめるので、筆者はなんとなくボーッとしている時にふと観ることが多い。

相当の実力者であることは上記の動画シリーズからもギュッと凝縮して垣間見ることができるので一見の価値あり。最近その実力を見せつけるような記録を Apex のストリーマー大会中に見せている。バトルロワイヤルであるという性質上、参加者に対する優勝者数が少なく (Apex では同時に 20 チームが戦う)、つまり一戦あたりの優勝確率はそう高くはならないものである。そんな中、バトロワゲー Apex のストリーマー大会において、時間制限の中可能なかぎりポイントを稼ぐというルールで 14 連戦 14 連勝という偉業を達成している。もちろん彼と組んでいたチームメイトも実力者であり、"Rumad"氏一人の偉業ではないが、ほとんど急造と言えるようなチームで連携をし、見事に連勝を決めたプレイングは圧巻の一言。時間制限まで野良試合に潜りまくり点数を稼ぐというレギュレーションのため参加者同士の直接対決をするわけではないが、バトロワゲーにおいてここまで安定してトップを取り続けるというのは当然容易に達成できることではない。なお、1 ゲームにおける優勝が 10 点、1 キルあたり 1 点というルールであったため、他の多くのチームは「ゲームに参加 -> 可能なかぎり殲滅 -> ゲームから全滅離脱」という作戦をとり、キルによって点数を稼いでいた。また、14 ゲーム中 1 ゲームの記録がノーカウントになるなどのハプニングがあったことも影響し、最終的なポイントでの順位は 21 チーム中 6 位とまずまずの位置に収まったが、バトロワゲーにおける 14 連勝という圧倒的な記録の前

にはそんな話も霞んでしまう。

そんな大会の様子を観たい諸氏におかれては、是非とも AKAsurvive の配信ページ "Videos"の"Recent highlights and uploads"より「誰にも破れない大会無敗記録： 14 戦 14 勝」という配信記録を確認していただきたい。

4.3. 蒼汁

主にプレイするゲーム：Hearthstone

カカカカ、カモーーーン！でお馴染みの Hearthstone プレイヤー。え、知らない？観に行きましょう。

Hearthstone の日本で開催される公式大会での実況・解説を勤める実力がある。相手の手札に握られている重要なカードをピンポイントで当てられる程度の実力の持ち主。

色々なデッキを使いこなすため勉強になるし、プレイングに関して簡単に考えていることや展開の解説を入れてくれるため自身のプレイに取り込みやすい。トークも軽快であり、割とラジオ的な聞き方がしやすい配信という印象である。ミスプレイを視聴者に指摘されそうになると、世界大会で上位入賞した限られた者のみに与えられる限レアカードバック「ゴールデン・セレブレーション」を 2019 年アジア太平洋冬季選手権でベスト 6 入賞した際に入手し使用していることを背景に、「ゴールデンバックなんだよなー！」と豪快に視聴者にマウンティングするなど、お茶目なところもチャームポイント。

また落ち着いた声色をしていて声が良いのも視聴しやすいポイントか。是非一度配信を観に行き、そしてあわよくば Hearthstone のプレイをしてみてほしい。新拡張の「爆誕！悪党同盟」が 4 月 10 日にリリース予定、今ならお得なバンドルを予約購入も可能なので始めるなら今しかない！！！

さて、ここでタイムリーな情報が入ってきた。蒼汁氏、なんと 4/1 日を以って RED ONE 社の契約ストリーマーとなることが公表された。本人曰く配信のスタイル等に特別な変更を加えるわけではなく、契約ストリーマーとして配信頻度が増えるという話で、リスナーにとっては嬉しいニュースであった。(4/1 の配信で視聴者からは「エイプリルフールだと思ってた」と喜びの声が上がっている) すべてのタイミングが噛み合った今、蒼汁氏の配信から Hearthstone の世界を知ってみるのが通、かもしれない。

4.4. スタンミ

https://www.twitch.tv/sutanmi

主にプレイするゲーム：LoL

人呼んで LoL 芸人。決して平均的なプレイヤーから見て下手ということはないはずで、それなりの実力者なはずなのだがどういうわけかとにかく視聴者に舐められている。

メインのロールは Mid レーンで得意チャンピオンはアニビアという氷の鳥。配信中のテンションは上がり下がりが激しく、割と一人で賑やかになるタイプ。テンションが下がるといっても雰囲気が悪くなったり悪態をつくわけではなく、あくまで大げさな沈み込み方をする方向なのでやっぱり画面は忙しない。ことあるごとに大きいリアクションを取るのも芸人と呼ばれる所以か。どれだけプレイに失敗しようと、味方に迷惑をかけたりかけられたりしようと、視聴者に煽られたりしようと、ほとんどの場面でポジティブである点が特筆に値する。本当に楽しくてゲーム配信してるというのが伝わってくるし、ボロボロになって精神が壊れてもおかしくなさそうな状況に対しても前向きに実況を続けるそのメンタリティにはただただ感心するばかりである。 そして自分の貢献によって勝利した時には惜しみなく全力で喜ぶ、よかったなぁ ... あまりの連敗に本当にメンタルが折れることもあるのはご愛嬌。

もう一つの特徴は視聴者が割と容赦ないことであり、冗談と悪意の瀬戸際くらいを攻めるコメントをしてくるのだが、それらを真正面から軽く、しかし確かに受け止めてうまく雰囲気を保っているところも目を見張る。治安が悪い方向に良いという感じで、悪いと見せかけつつある一定のラインは踏み越えないよう配信者も視聴者も気をつけているように感じられ、良い意味で悪友関係のようなものなのだろう。治安が悪かったのを上手く「内輪ノリ」に昇華したとも言える為、そのあたりを飲み込める人にはオススメをしたい。

LoL というゲームのことがわからなくても配信者というものが自分の配信をうまく回していくという見本として一度は見ても損はしない、 是非とも覗いてみてほしい。(他の視聴者につられて辛辣なコメント等をしないでください、お兄さんとの約束だ！)

4.5. ZeRo (英語)

https://www.twitch.tv/ZeRo

主にプレイするゲーム：大乱闘スマッシュブラザーズ SP (英：Super Sumash Bros. Ultimate)

大乱闘スマッシュブラザーズ for WiiU (通称スマ 4) で数々の大会に出場しては優勝してきた猛者であり、 スマブラシリーズ最新作の SP でもその実力を配信で見せつけている。

陽気な性格でとにかく楽しそうに対戦をする。負けが込むと無口になったりするが、基本的にずっと笑いながらゲームをしている。スマ SP の実力はやはり一流で、対戦相手も決して弱いわけではないはずなのに軽々と倒してみせる、しかもほとんど全てのキャラクターで。 本作ではシステムへの理解という部分がゲームの上手さのウェイトを大きく占めているが、それを加味しても 70 キャラ近いプレイアブルキャラクターを使いこなしているのは流石と言う他ない。

配信頻度はかなり高く、一回の配信時間も 7-8 時間とまとまって長い。スマ SP のプレイヤーであれば参考になるプレイングを見ることもできるため、視聴してみてはいかがだろうか。

4.6. Games Done Quick (英語)

https://www.twitch.tv/gamesdonequick

主に配信されるゲーム：あらゆるジャンル

最後の紹介となるのは Games Done Quick, 通称 GDQ だ。 年に 2 度アメリカにて行われる世界最大規模のゲームのチャリティイベントであり、 例年 1 月に Awesome Games Done Quick が、6 月に Summer Games Done Quick がそれぞれ開催される。Games Done Quick の名の通り、 あらゆるジャンルのゲームのスピードラン (英：Speed Run, 日本語では"RTA"と呼ばれることが多い) が開催期間中に配信され続けるイベントだ。 この開催期間だが、 なんと各イベントで 1 週間にも及び、 数分程度で終わるゲームからクリアまで 5-6 時間程度かかる RPG に至るまで、ぶっ続けで配信が行われる。もちろんゲームご

とに走者は交代するのだが、1 週間の間一切配信が切られることはない。

レトロゲームも最新のゲームもとにかく走者さえいれば何でも走られるため、筆者は期間中とにかく暇があれば垂れ流しにしておき、気になるゲームをじっくり観るという楽しみ方をしている。また、一般的なゲーマーは一切知らないがどういうわけかスピードラン界隈だけでお約束になっているゲームやミームが存在しているのも注目ポイントだ。Hype! スピードランである、つまり可能な限りクリアまでの時間を短縮する術を尽くすイベントであるという性質上、全てのゲームがするすると小気味よくテンポよく、そして時にはテクニカルに攻略されていくことになる。こうした「難関があの手この手で次々と突破されていく様子」に惹きつけられる諸氏におかれては、このイベントを見逃す手は無い。

GDQ 最大の特徴はチャリティイベントであるという点だろう。なんと最新のイベントでは開催期間中に総計 200 万ドルを超える募金 (Donation) があり、世界中のゲーマー達の力が総結集された時の力を見せつけられるイベントだと言える。(出典：wikipedia "Games Done Quick" https://en.wikipedia.org/wiki/Games_Done_Quick) 募金は主に"Prevent Cancer Foundation"や"国境なき医師団"へと送られている。募金は Web 上から行えるが、スピードラン外やスピードラン中の走者がちょうどいいと思った見所の無い区間などに募金時に投稿されたメッセージが読み上げられるのはお約束。募金によってボーナススピードランが生えたり、スピードラン時の選択肢やキャラクター名の投票が行われたりするなど、募金行為そのものがエンターテイメント性を含んだものになっているところもお祭り性を高める要因だろう。

今年の Summer Games Done Quick は 6 月 23 日 - 30 日と決まっているため、スピードランの祭典に興味を持った諸氏については丸っと有給などを取りじっくり腰を据えて視聴してみるのも乙だろう。

4.7. 大会の公式放送

以上は私の普段視聴するチャンネルの中でオススメのものを紹介した者であるが、ゲームを観るなら競技シーンを観たい！という諸氏は是非大会公式の放送を視聴してほしい。各大会の公式放送を探す必要はあるが、"< ゲームタイトル > 大会 Twitch"とかでテキトーにググれば公式の放送情報が手に入るはず。多くの場合は大会公式チャンネルでは過去

の映像を"ビデオ"から確認可能なので、タイムゾーンの差などにも負けず観戦をしてみてほしい。海外大会であれば当然言語は英語であることが多いため、実況・解説で英語のトレーニングなんかをしながら楽しめるだろう。筆者はこの方法でひたすら海外の大きな大会を視聴しまくり、実況・解説を通してリスニング力を、大会の間英語でひたすら独り言を話すことでスピーキング力を高めたことがある。(生活は崩壊した) 無論日本国内の大会も多くの場合は Twitch で配信されるため、ガチ勢のプレイを観たい諸氏は Twitch をチェックしてみて欲しい。

大会の配信には通常の配信とは異なる趣があることを体験されたし。

本当はこれ以外にも取り上げたい配信者は数名いるのだが、今回のところはこのくらいで締めさせていただきたい。他にも魅力的な配信者、大会の放送、ゲームなどはたくさん存在している、お気に入りのゲーム配信を見つけるのもそう難しくはないはずだ。

5. おわりに

本稿は以上となるが、いかがだっただろうか？正直に言うと今も「オイオイ本当にこの怪文書をガチのテック記事と並べて置いて大丈夫か」と言った心境だがその評価は読者諸氏にお任せしたい。「まあなくてもいいけど料金とか変わんないならあってもいいんじゃない？」くらいの感想だとダメージ少なめで嬉しいです。グダグダ言ってはいるが、この記事との出会いも事故だと思って諦め、何となくラジオ的な賑わいが欲しいなと思った時には Twitch の視聴をしてみてはいかがだろうか。是非自分だけの友達の家で自分は作業してるけど友達はずっとゲームしてるライフを手に入れて欲しい。

P.S. こんなことを言いつつも、他の面々を見ていて「やっぱテック記事書きてぇ ...」と思ったので、次回は何かしらテックっぽいことできるといいなあなどと思っている。予防線を張っておくと、他のメンバーと比べ技術的なレベルは数段劣るので、無よりは非自明な何かができたら褒めてほしい。

あとがき

このたびは yabaitech.tokyo vol.2 をご購入いただきありがとうございます。yabaitech.tokyo は大学院の (元) 同期で構成されるゆるい集まりで，技術書展 5 から活動を始めた若いサークルです．ちなみに表紙の写真も大学院の同期のひとり (ただしこのサークルには無関係) なのですが，不思議な力によって美少女化されました．かわいいですね．

yabaitech.tokyo の記事は (読めばわかる通り) なんとなくみんなが書きたいことを書くというちゃんぽんなものです． これは vol.1 から変わってないです． ただし今回の vol.2 では著者数が vol.1 から比較してなんと 3 倍になっています．こんなにたくさん書きたがるなんてみんな暇ですね！すごい！もとい，アウトプットをしようという行動力とネタを探して実装する好奇心・やる気に満ち溢れる寄稿者の皆様方には感謝と尊敬の念を禁じえません．

基本的にはガチめの記事が多いですが蓋を開けてみたら勝手にそうなってただけです．当然ですが， この同人誌のおかげで一番勉強になっているのは読者ではなく書いた本人だったりするので， 読み手にとってのわかりやすさが最大限尊重されているとは言いがたいかもしれませんが， その場合は各々の著者に読者のみなさまが自分で文句を言ってください．たくさん文句を言って，著者の人間性を試しましょう．

本誌の内容はそれなりにニッチなところを攻めているのでなかなか大きな反響というのは得られにくいことが想定されます． ほんの小さなものであってもなんらかのフィードバックが得られれば著者らは大変よろこぶでしょう． 基本的に全員 Twitter を監視していると思うので，本誌に関する感想はぜひ Twitter やその他インターネット上にあげていただければとても励みになります．

1. SATYSFI について

本誌は vol.1 に引き続き SATYSFI を用いて製作されました．SATYSFI は LaTeX に変わる組版処理システムとして将来を期待されている新進気鋭のソフトウェアです．LaTeX は C 言語以前の石器時代の言語である TeX 上で無理やりプログラミングしなければならない点がめっさつらいのですが，一方の SATYSFI は静的型システムと簡潔な意味論を持つシステムとして設計されておりいろんな部分が現代的でほとんどつらみがないです．SATYSFI は前身となる組版処理システムを経て 2017 年に未踏事業で採択されたのち今なお改善が続けられ，ユーザーを増やし続けています．なお，第 5 章を執筆している gfn こそが SATYSFI の生みの親です．

本誌はおそらく SATYSFI で組版された文章の中でも The SATYSFIbook 以来最大規模の文章だと思います．特に，複数人での作業フローや多様な種類の記事を前提としたプロジェクトは少なくとも今回の規模ではこれまでなかったと思います．そのおかげで制作中にはそこそこの数の SATYSFI のバグを発見することができました．その意味ではヤバイテックトーキョーも SATYSFI の改善に貢献していると言えます．本誌の (見かけ上の) 出来上がりを参考に，SATYSFI を使ってみようという方が増えれば幸いです．その際はぜひ本誌を複数冊ご購入いただき，SATYSFI の布教活動に使用していただければと思います．

(文責 : wasabiz)

yabaitech.tokyo vol.2

発行日	2019.04.14
サークル	yabaitech.tokyo
Web サイト	http://yabaitech.tokyo
連絡先	admin@yabaitech.tokyo
印刷所	有限会社 ねこのしっぽ

---

yabaitech.tokyo vol.2