
ヤバイテックトーキョー
VOL.5
屋内しか勝たん

YABAITECH.TOKYO

vol.5

2020

目次

"Python" Cookbook

— Recipes for Eating Python —

IrnBru

こっちが深淵を覗いてる時、深淵が覗いてき返したらめっちゃ怖いじゃん – わさびず –

1. 序章 : Python を食べ尽くせ！

Python 2 の賞味期限が切れ、Python 3 に移らないと"不味い"、そんな時期がやってきた。

「Python は簡単」

巷ではそんな風に評されることも少なくない。しかし、私の知る Python とは実に噛み応えがあり、そして正しく扱うことでその味が強く引き出されるものである。読者諸君の中にも分かった気でいる方が多いのではないだろうか。誰もが Python のことを知っているが、本当に口で味わった人間はあまりに限られており、そしてお察しの通りその味をどうしても知りたいというプログラミング界隈における自然な欲求の高まりは過去類を見ない程高まっていた！(当サークル内調べ) そう、今回お届けするのは"Python Cookbook", 多くの人が口にしたことはないであろう Python、すなわちニシキヘビをおいしく調理して食すことで Python の深みを知ろうというお料理会である！これはまだ味わったことのない Python の真髄に迫るべく、その血肉まで食らわんとする 5 人の漢たちが Python へと挑んだ、戦いと悪ふざけの記録である。

なお、ご注意いただきたいが本記事は Python お食事会を出汁にヤバイテックメンバーの言動をまとめ、皆様に本誌を読む上で「ほう、生産者はこんなやつらなのか」と親しみを持ってもらうものであるという側面があります (vol.5 でやることか？)。このため、本筋とは全く

関係ないクソしょうもない会話が盛り込まれておりますがご了承ください。こちとらそっちがメインなんだよ。

1.1. 登場人物

- わざびず (びず): 人に悪ふざけをさせるのが好きな方。vol.2 で粘菌をかわいいかわいい言いながら育てていたが、最近は大分進化したため P 業にいそしみアイドルにかわいいかわいい言うのが仕事。学部時代、学生控室に悪ふざけで生ハムの原木を導入しようと主張し始めた狂人。
- censored (cen): 安全圏から悪ふざけを眺める方。筆者は彼を一般的なものとは微妙にズレた感性の持ち主だと思っている。その実リスクリターンを冷静に計算している一面もあり、外れくじを引かない計算高い狂人。
- MasWag (Mas): 意外と悪ふざけに悪ノリする方。でも完全に乗り切るわけではなく冷静な部分も。自他ともに認めるガチオートマトン野郎で非常に優秀。今回こんなところにいるのはちょっと足場を間違えてしまっただけなんだ ...。
- zeptometer (zpt): 人の悪ふざけに乗せられて真に受ける方。結果として狂った提案を実に真面目に実現させようとする。面白そうというだけでなく実利があると本気で考え学生控室に生ハムの原木を本気で導入しようとしていた実直な狂人。行動力の分だけ厄介と評判。
- IB: 本稿執筆者。執筆を引き受けたタイミングではノリと勢いで何とかなると思っていたが現実を知る。もし私が原木導入推進派だったら学生控室にはゴキブリが 30 倍発生していた (当社比)。わたしだけがまともなんだ。

2. Python 以外の全てに至る道

＊都内某所・某所にて＊

IB: というわけで、マジでやるんですねこれ (約束から 1 時間遅れで某所に来た)
cen: こちらが Python400.0 です
zpt: 超未来の Python じゃん

ニシキヘビ 400g の入った未開封箱だ！！

IB: いやー改めて意味がわからないし記事になる気がしない。誰が Go サインを出してしまったのか
びず：いや絶対需要あるでしょ。俺は読みたいよ
IB: 他人事だな
zpt: Python 食べた記事、バズらないわけがないじゃないですかやだなぁ
IB: 確かに！！(ヤケクソ)
cen: レシピはもう用意してあるから、あと足りない食材買ってくるとこから始めるんで

すっと差し出されたディスプレイに映し出されるはどこからか集めてきた、 ニシキヘビのスープと野菜炒めと二シキヘビとは特に関係の無い M＊k＊'s kitchen のレシピ。

IB: 何で M＊k＊'s kitchen?
Mas: M＊k＊'s kitchen のレシピとか参考にしてます。
IB: いやだから何で M＊k＊?
zpt: Y＊uTube チャンネル面白かったから ...

IB: 確かに！！
cen: じゃあもう IB 待ってて時間押してるから、とっとと買い出し行こうぜ
IB: ごめんて
zpt: アメ横の地下食品街で食材は大体集めるんで。本企画は COVID-19 流行前のものです)

かつて M＊k＊'s kitchen のレシピで鶏肉をニシキヘビに取り替えられ頭を抱えた人間がいただろうか？さながら Python2 から Python3 への移行を推奨されたが諸事情により移行をできない開発の気分だ ... だが読者諸君も気になるはずだ！果たして Python (ニシキヘビ) はおいしく調理して食べることができるのか！！これまでの Python クックブックの常識を覆す逸品たり得るのか！！！これは食材を買いに行くしか無いったら無い！！！！

3. Python 調理用食材購入

こうして食材を購入しに行く一行だったが、 その前には幾重もの試練が待ち受けているのだった ... その一部始終をお届けしよう！

cen: じゃあパンダ食べるか
IB: あ？
cen: こわい
IB: パンダ食べる方が怖くないか？
cen: パンダを食べるのにはちゃんと正当な理由がある
IB: ほう

誤解なきよう、 ここでパンダを食べようと言い始めたのは Python のライブラリに存在する pandas にあやかり、誰からともなく「pandas にあやかってパンダ食べましたっつったら面白くね」と言い出したことに端を発するのである！そのことを聞かされた筆者は畏敬の念からか「正気か？」と思うに至った。しかし、どうやら本気らしく堂々と語った彼らに敬意を表し、この言葉は胸にとどめたい。

IB: 正気か？
cen: 俺は記事書かないんで
IB: 人の心家に忘れてきてるよ
cen: パンダ見に言って pandas 攻略〜〜〜！みたいな話もあったよね
IB: きっつ ...
びず: 和歌山に行くといくらでも見られるよ
IB: そうなん？

びず：アドベンチャーワールドの人工飼育がめちゃくちゃ強くて 2 年に 1 頭くらい産まれるんだよね、 パンダ。 関西の人間はみんなそこ知ってるし、 テレビで上野動物園でたまにパンダとか言ってる度に、「愚かだな、こいつら」って思ってる

恐ろしいことにあのかわいらしいパンダで鬼の首を取ったような表情をするわさびず氏に震えが止まらない。ここで突然関西パンダ事情でイキり始めた彼の形相を撮影できていなかったのは痛恨のミスだ。

IB: まさかパンダでここまでイキられるとは思わなかったけど
びず：いやいやいや、関西の人間はみんなイキってるから。パンダに関してはもう、本当に、本当にもう東を見下してるから
cen: 上野動物園なんて目じゃない、何で騒いどるんみたいな感じなの？
びず：本当にそう思ってる、全然誇張してない
zpt: 俺も心の底からそう思ってる
びず：穏健派の zpt さんですらそう思ってる
zpt: 和歌山クソ遠いなあとか全然思ってない
cen: 絶対そう思ってるやつじゃん

4. 地下商店街 食材購入編

こうして話すうちに目的地である地下商店街に到着。 雑然としており、 ラッシュ時の駅のホームのような人混みにもかかわらず上手く行き交う人々。さらに不思議なことにその人々を超える量の商品がずらっと陳列されてあったり、 あちこちに水槽があったりと混沌としている。

左：地下商店街の入り口　右：珍妙な商品を売ってくれる地下の店員との交流

左：なんだかわからないがとにかく辛そうな品々　右：水が並々張ってある水槽には生きているか死んでいるかわからない魚がふよふよ

筆者は詳しくないが、恐らく中国やベトナムあたりのニシキヘビと比べ勝るとも劣らない珍

奇な食材がずらりと並んでいた。しかし恐ろしいことに、我々のレシピはこの地下街にすら存在しない食材を求めていた。ここで揃わない以上、妥協して代替可能な食材を見繕うしかないのであった ...。

5. Python のある場所へ至る道 (帰宅)

何かは分からないが間違いなく何らかの法律に違反していそうな構造の地下から這い出た五人組。そこに待ち受けるは次なる関門、代用品の入手であった。

cen: 残り必要な食材見てみるわ。...M ＊ k ＊'s キッチン系が足りない。よくわからんカタカナの野菜
zpt: 紫キャベツとカリフラワーか ...
IB: いやでもそれ見た目を整えるためだけに用意してるでしょ？
cen: え、食用花とか面白くない？
IB: 何を食べる会をやってるんですか
cen: なるべく、面白いものを食べたい
IB: 本稿の目的はニシキヘビをおいしくいただくことです
cen: でも結構足りないな ... 中国ハムとかは代用品が必要だね。中国ハムは見た感じ生ハム原木っぽい感) じだったから生ハム原木を買えば解決！！よかったね zpt さん念願叶うよ！
IB: 学生控室時代からの伏線ここで回収か〜〜〜
Mas: やるじゃん
びず : 導入しようって言ってたの誰だったっけ
IB: 喜んで主導してたのはびずと zpt さんだけだったね
びず : マジ？
IB: いや、あん中で本気で食べたいから買いたいって言ってたのは zpt さんだけだからね。
zpt: えぇ？そんなぁ、みんな地下に生ハム原木あったら食べたいでしょ
IB: びずは乗ってるけど愉快犯だからねこれ。zpt さんが異常行動を前のめりにやってるのを間近で見) られるからみたいだけだよ。実質 panda 扱い
びず : いやいやいやいや、それはね ... あるけど
cen: あるじゃん
びず : zpt さんだけは本気でやってくれるから ...
zpt: え？お前らそうだったの？え〜すごい傷ついた
cen: ごめんね zpt さん
zpt: そうやって俺を持ち上げてゲラゲラ笑ってたんだな

旧交をいい感じに温めることができ、手ごたえを感じた一行は代用品を求めスーパーへと向

かうのであった

Mas: お、pandas じゃないけどパンダ焼きあるじゃん
IB: パンダ焼き食べる？
びず : あれ食べて「pandas、攻略〜」みたいな
IB: Python 入れるか？
Mas: パンダ焼きの中に Python の白焼き入れて、「Python pandas」ってね！
cen: パンダ焼き、何か 20 個 500 円くらいの小さいサイズっぽい
zpt: まあそんなもんでしょ
Mas: 人形焼みたいな感じらしいな。... まあ並ぶか

pandas のコンプリートを何としても行う使命に駆られ吸い込まれるようにパンダ焼きの列に吸い込まれる Mas、zpt、cen の三人。時間の無駄でしかない会の時間を有効に活用し無駄に使える時間を増やすため、びずと IB はスーパーへと買い出しに向かい、二手に分かれることに。その道中、喉が渇いたからと立ち寄った自販機が 100 円玉を飲み込んだ瞬間に沈黙するというハプニングにも見舞われましたが私は元気です。令和の自販機難しいな、機嫌が悪かったのか？飲み物を買うつもりが飲み込まれる小銭、この世は喰うものと喰われるものがほんの一瞬の隙で入れ替わってしまうと告げているかのような出来事に震えることしかできなかった。果たしてこれから対峙する Python... 本当に喰う側はどちらなのか。そう問いかけられているようであったということにしたい。

100 円玉をごくごく飲み込みすやすや眠った

こんなハプニングに見舞われつつもスーパーにたどり着き、食材の購入は進んだ。パンダ焼き購入組とも合流し全ての食材を揃え、いざ帰路へ。

びず：二輪免許欲しくない？
IB: 唐突だな、俺はいらないけどびずは欲しいの？
びず：実はちょっと欲しい
IB: アクティブだなぁ
びず：アクティブというか、今は土日に荒川までチャリで行ってるんだけど、チャリだと限界があるんで、もーちょい遠くに行きたいなって
IB: 今よりもうちょっと遠くへって .... アウトドアだなぁ
びず：アウトドア ... だけどやってることほとんどひきこもりだからね。外に出てるだけで家でボーッと引きこもってるか外で誰とも喋らず引きこもってるかの違いだから。本当に

突如として妙なアウトドアの基準を持ち出すびず氏、突然ではあるがそのインタビューの一部始終をお伝えしたい。

IB: 何に対する言い訳かわからないけどそれは普通にアウトドアじゃん
びず：アウトドアじゃないんだって！昨日もここから 20 キロメートルくらいのところにある公園に自転車で行ったんだけど。

IB: 自転車で？
びず：うん。
IB: ア ウ ト ド ア や ん
びず：いやいやいや。
IB: 別に友達とワイワイするのがアウトドアじゃないからね。お外に出るためにお外に出てるならそれはアウトドアでしょ！
びず：お外に出てるんだけど、でも夜なんだよ。可能な限り人がいないタイミング狙ってんだよ。都会じゃない方がいいんだけどあの公園は高台になってるから、都会であることを忘れられる。空が広くなる。
IB: いやいや外で引きこもってるっていうのは、「一日中ボードゲームしたいねー。うーん、じゃあ八王子に行って安い宿でも借りてひたすらボードゲームやろっか」ってやつでしょ。
びず：それ完全に社会性じゃない？
IB: でもインドアじゃん、外籠りよ。
びず：いやいや、それは違うよ、それは中にいるだけで「アウトドア」
IB: 社会性をアウトドア・インドアの基準にしてるだけじゃん
びず：そうそうそう、俺はそっち側なの
IB: それ一般的じゃないですよ。土日に人が少ないからって言って高台に自転車こいで行くのはもうアウトドア好きやん。
びず：高台のすごい開けたところでベンチ座ってツイッターしてるだけだから。ひきこもり。結局、家の中とか外とか関係ないってこと
IB: じゃあ何、みんなでスマブラやろうぜはアウトドアってこと？
びず：みんなでスマブラは完全にアウトドアでしょ
IB: それじゃあツイッター社会性あるでしょ！高台行ったらツイッターしてるって言ってたじゃん！
びず：いや、 ツイッターは ... ツイッターは社会性ないよ。 あれは虚空に向かって語りかけて深淵を覗いてるだけ。向こうから干渉してこないから
IB: えぇ ... もう今度から「この時間帯びず外出てるなー」ってタイミングでリプ飛ばして社会性で轢いてやる ...
びず：違うじゃんそれは、深淵を覗くのは深淵だと思ってるからで ...。こっちが深淵を覗いてる時、深淵が覗いてき返したらめっちゃ怖いじゃん。
IB: 決まり文句を勝手に言い換えるなよ、深淵は見返せよ！
びず：「ツイッターはフィクションの世界だ」って思って見てるから。タイムラインはフィクションだよ。アニメ見るのと一緒。アニメこっちに干渉してこないから。
IB: 実はその怖いアニメがツイッターなんすよ。気づいてないかもしれないけど
びず：いやいや、知ってるんだ俺は
IB: 何を言おうとアウトドアはアウトドア。びずのやってるそれはもうドアウトドアだよドアウトドア

こうしてアウトドア観について議論を深めるうちに我々は Python お料理部屋へと戻ってきたのであった。

zpt: さて、まず最初に執り行いますは開封の儀です
cen: お、いよいよ。もう当初の予定時刻から大分遅れてるからね…
びず : じゃあそろそろスマブラする？
cen: Switch でできる、3 人まで用意あるから
IB: GC コン持って来れば良かった
cen: IB は 1 時間遅れたんだから持ってくる時間あったでしょ？
zpt: あ〜そうやって過ぎたことで人を責める
Mas: 過ぎたことで人を責めるのはアウトドア
zpt: 話をややこしくするのはやめてほしい

6. 料理を開始する者とそうでもない者

zpt: じゃあ開封する人〜。
Mas: zpt さんでしょ。
IB: お前だよ！お前の記事を書くんだよ！
zpt: ちょっと待って、趣旨変わってない？まあいいや、はい、というわけでこれが蛇の肉！

取り出されたるニシキヘビ 400g 7106 円

zpt: ちゃんと処理された肉だなぁという感じ。愛知から送られてきたのか。ていうか絶対もうこの入れてあった箱とかどっかで買ってきたお菓子の箱だもんな

cen: いいじゃん
IB: どこで買ったの？これ
cen: オンラインショップ、"ニシキヘビ 肉"で調べたら出てくる
IB: へぇ
cen: "ヘビ 肉"だと出てこないんだよ
IB: ニシキヘビなら出るの？
cen: うん、ここに NLP の罠があるって、にわかに盛り上がった
zpt: "ニシキヘビ"は"ニシキ"と"ヘビ"に分解されないから"ヘビ"で検索しても出てこないっていうね
IB: なるほど
zpt: じゃあこれを ... どうする？まず何から作るかですね
IB: 全然段取りを知らないんだけども
cen: 白焼き！
Mas: 白焼きだと思う
zpt: 白焼きかぁ
IB: 白焼きを食べた新鮮な反応が一番最初に欲しいか
びず: じゃあそれでいきましょう
zpt: 残りは ... スケジューリング面倒臭いな
IB: 自分料理スキルに関しては ... ゼロなので
cen: じゃあ自分は ... ここで料理が美味しくなる舞を ...
Mas: 俺も料理スキルゼロなので舞を ...

7. 調理からいざ、実食

cen: パソコンの画面にレシピ表示しよ
びず: スマブラの画面じゃなくて？
zpt: 別にスマブラの画面でもいいよ
Mas: え、いいの？
zpt: いや、やっぱゲームするなら"Untitled Goose Game"やって欲しい
IB: 何それ？
zpt: あれ、知らない？ガチョウになって街の人にちょっかいかけるやつ。少し話題になったんだけど
cen: あ〜〜！あのクッソしょうもないやつ？基本ゴートシミュレーターの綺麗番みたいな(筆者注
zpt: いやいや、でもあれはちゃんとパズルゲームだよ。いたずらを仕掛けるっていうのがちゃんと体系化されてて、あれをしたら街の人がこう反応して、これをしたらこう反応してっていうのが決まってて、うまくそれをつないでいくと街の人がうまくハマってくれるっていう ...
cen: でもゴートシミュレーターも似たようなとこあるじゃん。ジェットパックつけると空を

飛べて、街の人には嫌がらせをする
zpt: ちょっと料理してるだけだと暇になりそうだからゲームする担当やらない？
Mas: マジで言ってんの！？
zpt: 真面目に考えたら 5 人で料理って無理だからね？
cen: やるか～
びず：実際やることないんだよな
IB: ボドゲする？

スッと取り出されるボードゲーム「商売往来」

IB: 見かけより重いやつなんだけど
cen: あーだから重さで遅刻を
IB: そういう重さじゃない
zpt: よし場もあったまったところで、俺は料理をする

こうして我々は持ち前のチームワークを活かし、ガチョウさん (のゲームを遊ぶ) チーム、パンダさん (が外箱に大きく描かれているボードゲームをプレイする) チーム、パイソンさんチームに分かれ、分担作業を行うのであった。

zpt: よし、白焼きのパックを開くぞ。ひとまず開いたけど白焼きどうするかな、あらかじめ塩を塗りこんでおくか ... ？うわなんか触り心地が
IB: え、触り心地どうなの？なんだ表面の ... 鶏肉はこんな感じの表面してるっけ？変な感じ ...
Mas: よくわからん。鶏ではないが
IB: 手触りがなんか ... 変というか ... 気持ち悪いんだよ
びず：え、触りたい触りたい。... あー ... なんかカスカスしてる。脂がない
zpt: あー、それだね、脂がないのか

実に奇妙な触り心地。地を這うための筋力だろうか

zpt: じゃあ始めるか ...。普通の油がいい？オリーブオイルがいい？
びず : オリーブオイルで
zpt: オリーブオイルで
cen: M ＊ k ＊’s kitchen 始まっちゃったな
びず : .... よし、じゃあ俺たちはボードゲームやるか
IB: やらざるを得ない

湯煎解凍をしたら素焼きの時間だ！

こうして料理が始まるとともにそれぞれが分担された役割に戻る。"Untitled Goose Game"が起動しガチョウがガーガー言い始め、ボードゲームの説明が始まり、そしてヘビは焼けるのである。

～10 分後～

zpt: 肉です
cen: え、もうできたの？
zpt: いや焼くだけやねん、そりゃあ

これが素焼きだ！！

IB: じゃあ、zpt さんの実食からいきましょうか
びず : 面白い表情してね
zpt: 困る …。断面がやっぱ鶏肉なんだよなぁ
IB: 君のリアクション一つで記事の出来が決まるからな
zpt: うーん責任重大だ。では … ん、硬った、ナニコレ！
cen: マジ？
zpt: 前歯でサッとかもうと思ったら無理やった

二人ともこの表情にならざるを得ない硬さ

zpt: 噛めば噛むほど弾力あるなぁ。味は特に出てこない
びず : 硬いってのは噛み切れないってこと？
zpt: 多分 … 中に内臓の膜みたいなのが有って …
びず : 焼き方の問題とかではなくって？
zpt: いや多分いくら焼いてもゴムはゴム
IB: 名言だ
zpt: 肉の味がするチューインガム。

そして恐る恐る他のメンバーも Python の白焼きを口に運ぶ …

Mas: 硬い！
びず : 硬ぁったっ。
IB: 1cm の厚さに対して感じる分厚さが 5cm くらいある！
びず : 風味は確かに肉だね … 風味は鶏肉？
IB: 薄切りにするのが正解かもしれない
cen: 俺も食べるかー。(咀嚼).... えぇ？
IB: その厚みから疑問になるほど厚みを出してくるでしょ？
cen: マジ？なんか味もなんとも言えない味するね
びず : 例えるなら … 鶏？
cen: 一番近そうなのは鶏？でも油がなくてパサパサって感じでもないね。硬ァ！筋切らない

とダメなんじゃない？
zpt: 確かにそうかも。レシピだとスープの方はひたっすら煮込んで、ニシキヘビも細く裂いてるんだよね
IB: ぶっちゃけ舐めてたけどもうこっからは趣旨が変わったね、いかにこいつをうまく食べてやるかの勝負だね
びず：うん、そうだね
cen: 想像以上に厳しかった！
びず：これ…でも、いや、これすごいな、上手く食えそうな気がする
cen: ちゃんと調理しないといけないタイプのヤツ (調理しない)
IB: ね、挑戦って感じがしてきたわ、これ。これ…普通に (食事企画として) 面白いぞ (料理しない)
びず：正直冷やかしだったが、これは"実力"を試されてる (料理しない)

Mas: あ、パンダ焼きに白焼き入れて食べるの忘れてた (料理を手伝っていて偉い)
cen: あ、そういえば。じゃあ"Python pandas"失敗です

ヤバイテックトーキョーはパンダと共に歩みます

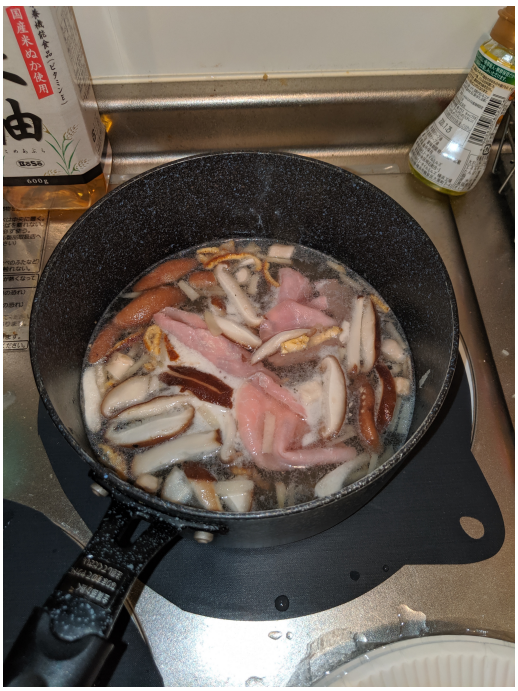
こうしてただの冷やかしから、Python という強靭な食材を相手取った真剣勝負が始まるのであった

zpt: じゃあ MasWag さん、やるぜぇ。ところで料理をしたくなってしまった人は是非手伝っ

て
びず：俺はゲームしたい
cen: 俺もなんだよ
IB: それはそれとしてボドゲがしたい
Mas: 何だお前ら

8. 本当の勝負が始まったけど頼れる仲間はみんな目が死んでる

Mas: これが蛇が入るスープ



みんなの努力の結晶ったら結晶だ

Mas: よく見ると結構お湯が少ない！ (300 cc)
zpt: そうなんすよこのレシピ、よく見るとお湯の量が書いてないんすよ面白いですよね
IB: 300cc で足りるの？レシピの完成図だともうちょい多そうじゃない？

Mas: 俺ら肉が 100g しかないから。 レシピは 600g って書いてあるでしょ？俺らその 1/6 しかこれに使えないんだよね
IB: 確かに。あれ、レシピに鶏肉って書いてあるけど、こっちは鶏肉入れてるの？
Mas: 鶏がらスープ入れた
IB: あー、レシピだと出汁とってるだけなのか
zpt: そうそう、だから鶏がらスープで代用
zpt: ちなみに、蛇肉をどう柔らかくするか問題なんですけど。スープのレシピだと「ひたすら煮る」、炒め物の方だと「包丁でバンバン叩いてスジを切る。」
IB: 文明開化の音が聞こえる
zpt: ただ安い肉に通用する方法が蛇に通用するのか僕は不安ですが ...
Mas: やらないよりマシだしいいんじゃない
IB: 実験だよ、実験！
zpt: やってみるかぁ。じゃあ白菜炒めやってく

zpt: ちなみに、蛇肉をどう柔らかくするか問題なんですけど。スープのレシピだと「ひたすら煮る」、炒め物の方だと「包丁でバンバン叩いてスジを切る。」
IB: 文明開化の音が聞こえる
zpt: ただ安い肉に通用する方法が蛇に通用するのか僕は不安ですが ...
Mas: やらないよりマシだしいいんじゃない
IB: 実験だよ、実験！
zpt: やってみるかぁ。じゃあ白菜炒めやってく
Mas: よし！白菜炒めのヘビの方は叩くぞ！
zpt: じゃあこいつ (Python) を親の仇の如く、めっためったにします

— 響き渡る包丁をニシキヘビに叩きつける音 —

IB: 果たしてこれがどのくらい効くかだな
zpt: うーん、何とも言えない。.... 結構楽しくなってきた (ダンダンダンダンダンダンダン)
IB: そういうとこやぞ
zpt: 単純作業楽しくない？いいか、 これは親の仇だが単純作業なんだ。実際責任重大だし。(ダンダンダンダンダン ...) こんなもんでまぁいっかあ
Mas: でも大分厚みが変わってきたかも

分かり辛い厚みの比較をした図

zpt: 親の仇図 1 と not 親の仇図 2
IB: 死んだ仇だけがいい仇
zpt: そうだよ

9. イカした食材ニシキヘビ

　こうして調理をしたり冷やかしたりすることで作業は進み、 ついに一品目が完成することに。

zpt: はい、こちら白菜の漬物と蛇肉とキクラゲの炒め物です
IB: おお〜！
Mas: やった"Pickle on Python"だ！すげぇちゃんとしてる
びず：ただ、蛇の見た目が … なくない？
cen: 蛇要素がない

諸事情により工程は省くが左をこうこうこうしてこうしたら右になる！

一同：じゃあいただきます
びず：あ、美味しい！ヘビ食ったけど美味しいよ
Mas: マジ！？
びず：八宝菜っぽいから、ヘビがイカっぽい
cen: ああ〜。あー、うまっ！イカの入った八宝菜みたいな
zpt: ヘビはイカだった
cen: かなり近い。細く切ったのが完全に正解
zpt: 確かにこれはイカだわ
IB: おいおい料理チョイスが完璧か？
zpt: 誰がこのレシピ見つけたんだっけ
cen: MasWag さん。Pickle を作りたいってことで見つけてきてた
IB: そこを何とか俺がやったってことにしてくんねぇか
cen: それは無理だな
Mas: これ、いいな
びず：これはイカ料理だな。
cen: なんだろうな … 独特の … というかヘビ自体には味がない
zpt: いや、むしろこの食感のために食べてるという
IB: そうそう、食感を提供するだけ
cen: これはありだよね、普通に中華屋で出てきても美味しく食べられる
びず：うん、これはいいわ
IB: むしろこの、ピクルスの味を邪魔しない肉っていう立ち位置

cen: そうそうそう、これいいね
Mas: ちゃんと"Python Cookbook"じゃん
びず : 肉の加工はなんか変えたの？
zpt: えっとね、叩いた。親の仇の如くバンバン。であと下味をつけた
Mas: これクックパッド書けるよ
cen: これ「前菜」がまずかったせいでね
zpt: 期待度合いがね
cen: ていうかこれ、肉抜いても美味しいんだよね
zpt: その説はある
cen: ちゃんと味もついてるおかげで全然変な臭みとかも感じないし ...
IB: ... これ Python のおかげじゃねぇな？これ勝利か？大分怪しいぞ。これさっきのレシピが肉なくても美味しいことが証明されただけじゃ？
cen: でもね、ほら、ヘビの肉が食感のアクセントになってる
zpt: 別にベースの味が美味しいからって何加えても美味しいわけじゃ無いし。あー日本酒欲しい。ちょっとコップくれよコップ。おい、誰だよストロングゼロ買ってきたの
cen: そういえばストロングゼロ買ったんだった
IB: ホント、倫理観を溶かしにきてる
cen: いやいや、人権飲みたくない？

IB: なんで一品目でもう美味しくなっちゃってんだよ！！
zpt: あぁー申し訳ない！
Mas: ベトナム行ったらこれ普通に出てくるんじゃない？
IB: とにかく記事にできるかどうか、戦々恐々としてるよ ...

さらに次の料理、 ヘビのスープに取り掛かる zpt。 我々がボードゲームに死力を尽くす中、zpt は zpt で苦しんでいた。どういうわけか手助けがなく、孤立無援の中、時折 MasWag の力も借り調理をする。そして鳴り響く親の仇を叩く音、果たして Python の行く末は ...

zpt: スープです ...

見た目おいしそうなスープ

IB: なんかこれ ... もう見た目いい感じじゃ
びず : うん、これもう中華で普通に出てきそう。この前出てきた
IB: みんながこぞって Python 食べ始める時代くるぞ
びず : 意外に中華スープがいい感じに
zpt: さて、 裏で別に作っていた一皿ですが ... これは M ＊ k ＊’s Kitchen のレシピに従った品なんだけども、盛り付けのセンスが圧倒的に ...

なんだかよくわからない一皿だ！

Mas: 何かレシピの写真と違わないか？
びず：使ってる野菜の種類が違うからな … でもほら、味とは関係ない
IB: 俺たちが食べたかったのは野菜の盛り付け部分じゃなくてここ (Python) だからね
Mas: そうそう、実質レシピ通り
びず：アスパラは添えるだけ

zpt: まあスープから順番にいきましょうか
びず：じゃあちょっと … いただきまーす。…. んー！おお！おいしい！！

わさびずもご満悦

cen: マジか
びず : ちょっと味付けが微妙 … ？
cen: え、何だったの最初の一言
びず : スープ全体よりもヘビだけの方がおいしいなこれ
Mas: なるほど
zpt: じゃあ残りの人がなくならないように分けるのよ
cen: お、うまっ。肉って煮ると大体ぐずぐずになるけど、これはしっかり食感が残るのがいい。スープの方は確かに味が微妙だなぁ、中華スープの元がダメだったのかな。もっと合う味がありそうではある
Mas: あーでも、陳皮と一緒に食べるとよりうまい。味が爽やかで、なんか今までで味が一番しっかりしてる気がする
cen: そもそもヘビに最適化された味をしている
zpt: 細長いの大事だね
cen: 細長くなって噛んだ時よりほろほろになっていい感じ。ちゃんとヘビに最適化された味になってる。中華料理やっぱすげぇよ
zpt: 正しい食べ方がわかったな

Mas: 最初からの成長が目覚ましい

cen: こっちの方 (M ＊ k ＊'s Kitchen のレシピ) も食べていいっすか
zpt: どうぞ
cen: じゃあ、いただきます。ただこれ、ちょっと不安だぞ。肉がでかすぎる感じがある。(咀嚼) うーん。まあ、不味くは無い
Mas: 味はしっかりついてる
IB: うん、神妙な顔になるね
びず : やっぱこっちみたいにぶつ切りだと食感があんま良く無いな
cen: 味は別にいいんだけどね
びず : 白焼きよりはまし。下から 2 番目くらい

10. Win over Python...

こうして、我々の味への飽くなき探求は終わった。そう、勝利したのだ。適切な処理 (親の仇のごとく叩く、手で引き千切る) を施すことで程よい食感を提供する存在となった Python、その真髄とは概ねイカと同等の役割を果たすことと分かったので、皆様におかれましては「あ！今晩ニシキヘビ食べようとおもったのに〜〜」ということがあるかと思いますが落ち着いてイカで代用しましょう。

結論 : Python の代わりに Squid を使え (?)

Squid は Python の Proxy (代理) になるということでここは一つ、へへ。そして読者諸氏を騙すようで申し訳ないが、実はこの記事の裏テーマは zpt 氏の本性を暴露することであったのだ。それだけをモチベーションに本記事を執筆した筆者は、遂に zpt 氏が料理を黙々と行う中一切の手伝いを行わずボードゲームに興じ続けた。果たして筆者は本当に彼を晒し上げることなど許されるのであろうか？それは誰にもわからない。我々が深淵を覗く時、深淵は「え、プログラミング言語食べたら絶対面白いやん！」と言っているのだから ...

P.S. "Untitiled Goose Game"は周囲の状況と音楽がリンクするプレイ感覚、シンプルながらもユニークで面白みのあるグラフィクス、そして何よりいたずら心を刺激するパズルが一体となり、童心に帰る時間を届けてくれました。最後にピシッと"オチ"まで用意する周到さも見事です。 こんな記事読んでる暇があったら是非このゲームをお楽しみいただければと思います。これだけは伝えたかった。それでは。

SATɣSF$_{I}$ クラスファイルの開発の「コードを書く」以外の Tips

zeptometer

1. はじめに

yabaitech.tokyo もおかげさまで vol.5 を発行することができました。yabaitech.tokyo は発足当初から SATɣSF$_{I}$ を組版に使用していますが、今回は vol.5 に使うクラスファイルを独立したパッケージとして分離するという試みを行っています。この記事ではその過程で得られた、クラスファイルを開発するにあたっての「コードを書く」以外の知見、具体的には「Satyrographos パッケージの公開」と「Regression test を書く」部分について共有していきます。

しかしなんですね、同人誌を書くことで得た知見を同人誌の記事にするというのはなかなかマッチポンプ感があります。こういうことやってもいいのかねという気持ちにならんこともないのですが、こういう Tips を共有するのは有益だと思われるのでぐだぐだ言わずに書いていきます。こういうネタで記事が書けるのは「SATɣSF$_{I}$ で同人誌を書いているということをサークルの特色にできる」程度に SATɣSF$_{I}$ が浸透してない（が知名度はある）からなのですが ..... こういった知見の共有によってより SATɣSF$_{I}$ が普及していけばいいなと思っています。

そのような趣旨の記事なので、この記事は「SATySFi はインストールしたことがあって多少書いたこともあるけど、そのコードをライブラリとして公開したことはない」くらいの SATySFi ユーザを想定しています。SATɣSF$_{I}$ が何かを知りたいという方は The SATɣSF$_{I}$book [1][*1]の第一章がよい導入になると思います。SATɣSF$_{I}$ インストールしてみたいんだけどねえ、という方は「*nix 向け SATɣSF$_{I}$ インストールバトル手引き 2020 年 8 月版」[2] をおすすめします。

1 ここからダウンロードできるよ https://booth.pm/ja/items/1046747

1.1. class-yabaitech について

class-yabaitech は yabaitech.tokyo vol.5 で使用している SATYSF$_{\mathrm{I}}$ クラスファイルです。Github で公開されています[*2]。前身は gfn 氏の cs-thesis [*3]で、これを改変しつつ yabaitech.-tokyo vol.1 から利用していたものを今回独立した Satyrographos パッケージとして公開したものとなります。

2. Satyrographos パッケージを公開する

さて、熱心な SATYSF$_{\mathrm{I}}$ ユーザであるあなたはクラスファイルをいい感じに書き上げました。他の人にも自分のイカした成果を使って欲しいと思うのはごく自然な成り行きでしょう。今どきのプログラマであれば Github かその他のにプラットフォームにソースコードを公開することになるでしょうね。

これで十分か？というとそうでもありません。今の状況だとあなたの公開したクラスファイルを他のユーザが使う上で二つの障壁があります。

- クラスファイルをダウンロードして .satysfi 以下に配置する必要がある
- クラスファイルの依存ライブラリについても同様にインストールする必要がある

これらは結構な手間なのでできれば自動で済ませたいところです。一般的にこういった問題を解決するのが**パッケージマネージャ**です。Node.js であれば npm/yarn, Ruby であれば RubyGem ですね。 幸いなことに SATYSF$_{\mathrm{I}}$ にも有志によって開発されている Satyrographos [*4]というパッケージマネージャがあります。Satyrographos のリポジトリにクラスファイルを登録することであなたのクラスファイルを利用しやすくしましょう！

2.1. Satyrographos パッケージを定義する

リポジトリに登録するのに先立ってパッケージを定義していきましょう。 以下では `class-yabaitech` の実例を示しながら説明していきます。

---

2 https://github.com/yabaitechtokyo/satysfi-class-yabaitech

3 https://github.com/gfngfn/cs-thesis

4 https://github.com/na4zagin3/satyrographos

現在 Satyrographos ではクラスファイルのパッケージ名には `class-` を前置することが推奨されています。 そのため我々 yabaitech.tokyo のクラスファイルのパッケージ名は `class-yabaitech` となっているわけです。

Satyrographos は OCaml のパッケージ管理システムであるところの opam の上に乗っかる設計になっています。 依存関係などの解決を opam のレイヤで行ってインストールに必要な処理を Satyrographos で行う感じですね。 そのため opam と Satyrographos それぞれの設定ファイルが必要となります。

- opam 用のファイル
  - satysfi-class-yabaitech.opam
  - satysfi-class-yabaitech-doc.opam
- Satyrographos 用のファイル
  - Satyristes

ここにおける `class-yabaitech` は Satyrographos パッケージの名前ですが、opam パッケージとしては SATySFi 用のパッケージであることを明示するために `satysfi-` を前置しています。そのため opam のパッケージ名が `satysfi-class-yabaitech` となり、 これがファイル名にも反映されているわけです。 また、`class-yabaitech` はドキュメントも提供していますが、opam のレベルでは別のパッケージとして提供する必要があるので、本体と分離して `satysfi-class-yabaitech-doc.opam` を定義する必要があります。Satyrographos はドキュメントの設定もまとめて扱えるので本体の設定と一緒に Satyristes に記述します。

それでは最初に satysfi-class-yabaitech.opam を見ていきましょう。

```
opam-version: "2.0"
name: "satysfi-class-yabaitech"
version: "0.0.2"
synopsis: "The yabaitech.tokyo SATySFi class file"
description: """
The yabaitech.tokyo SATySFi class file.

This requires Satyrographos to install.
```

```
See https://github.com/na4zagin3/satyrographos.
"""
maintainer: "Yuito Murase <yuito@acupof.coffee>"
authors: [
    "gfngfn"
    "Masaki Waga"
    "Yuichi Nishiwaki <yuichi.nishiwaki@icloud.com>"
    "Yuito Murase <yuito@acupof.coffee>"
]
license: "MIT"
homepage: "https://github.com/yabaitechtokyo/satysfi-class-yabaitech"
bug-reports: "https://github.com/yabaitechtokyo/satysfi-class-yabaitech/issues"
dev-repo: "git+https://github.com/yabaitechtokyo/satysfi-class-yabaitech.git"
depends: [
  "satysfi" {>= "0.0.4" & < "0.0.6"}
  "satyrographos" {>= "0.0.2" & < "0.0.3"}
  "satysfi-base" {>= "1.2.1" & < "2.0.0"}
  "satysfi-fonts-noto-sans" {>= "2.001+1+satysfi0.0.4"}
  "satysfi-fonts-noto-serif" {>= "2.001+1+satysfi0.0.4"}
  "satysfi-fonts-noto-sans-cjk-jp" {>= "2.001+1+satysfi0.0.4"}
  "satysfi-fonts-noto-serif-cjk-jp" {>= "2.001+1+satysfi0.0.4"}
  "satysfi-fonts-asana-math" {>= "000.958+1+satysfi0.0.4"}
]
build: [ ]
install: [
  ["satyrographos" "opam" "install"
   "-name" "class-yabaitech"
   "-prefix" "%{prefix}%"
   "-script" "%{build}%/Satyristes"]
]
remove: [
```

```
  ["satyrographos" "opam" "uninstall"
   "-name" "class-yabaitech"
   "-prefix" "%{prefix}%"
   "-script" "%{build}%/Satyristes"]
]
```

まあこういうファイルは雰囲気で書けば大体問題ない[*5]ので `version` から `depends` の範囲をいい感じに改変すれば大体問題ないです。`name` のパッケージ名に `satysfi-` を前置することと、`depends` に依存パッケージ (SATySFi と Satyrographos を含む) をちゃんと書くことだけ気をつけてください。

satysfi-class-yabaitech-doc-opam の方はこんな感じです。

```
opam-version: "2.0"
name: "satysfi-class-yabaitech-doc"
version: "0.0.2"
synopsis: "Document: The yabaitech.tokyo SATySFi class file"
description: """
Document: The yabaitech.tokyo SATySFi class file.

This requires Satyrographos to install.
See https://github.com/na4zagin3/satyrographos.
"""
maintainer: "Yuito Murase <yuito@acupof.coffee>"
authors: "Yuito Murase <yuito@acupof.coffee>"
license: "MIT"
homepage: "https://github.com/yabaitechtokyo/satysfi-class-yabaitech"
bug-reports: "https://github.com/yabaitechtokyo/satysfi-class-yabaitech/issues"
dev-repo: "git+https://github.com/yabaitechtokyo/satysfi-class-yabaitech.git"
```

5 とは言ってもなんかぶっ壊れてしまうと辛いですよね ..... そういう場合は Satyrographos の作者による「素敵なライブラリを Satyrographos で配布しよう！」[3] を参照するか、それでも解決しない場合は SATySFi Slack (https://satysfi.slack.com) の satyrographos チャンネルで質問することをお勧めします。

```
depends: [
  "satysfi" {>= "0.0.5" & < "0.0.6"}
  "satyrographos" {>= "0.0.2.6" & < "0.0.3"}
  "satysfi-class-yabaitech" {= "0.0.2"}
]
build: [
  ["satyrographos" "opam" "build"
   "--name" "class-yabaitech-doc"
   "--prefix" "%{prefix}%"
   "--script" "%{build}%/Satyristes"]
]
install: [
  ["satyrographos" "opam" "install"
   "--name" "class-yabaitech-doc"
   "--prefix" "%{prefix}%"
   "--script" "%{build}%/Satyristes"]
]
remove: [
  ["satyrographos" "opam" "uninstall"
   "--name" "class-yabaitech-doc"
   "--prefix" "%{prefix}%"
   "--script" "%{build}%/Satyristes"]
]
```

class-yabaitech のドキュメントの実態は class-yabaitech を用いたデモ文書で、 そのため依存関係に satysfi-class-yabaitech が依存関係に入っております。正直これでドキュメントだと言っていいのかはかなり怪しい部分がありますが .....

ここで先ほどの二つの opam ファイルの `build install remove` を見ていただきたいのですが、`satyrographos opam ...` のような形で Satyrographos に処理を投げていますよね。このあたりの Satyrographos 特有のインストール等の処理を記述するのが Satyristes ファイルです。

```
(library
  (name "class-yabaitech")
  (version "0.0.2")
  (sources
    ((packageDir "src")))
  (opam "satysfi-class-yabaitech.opam")
  (dependencies (
      (base ())
      (fonts-noto-sans ())
      (fonts-noto-serif ())
      (fonts-noto-sans-cjk-jp ())
      (fonts-noto-serif-cjk-jp ())
      (fonts-asana-math ()))))
(libraryDoc
  (name "class-yabaitech-doc")
  (version "0.0.2")
  (build
    ((satysfi "__test__/integration/integration.test.saty"
      "-o" "yabaitech-demo.pdf")))
  (sources
    ((doc "yabaitech-demo.pdf" "./yabaitech-demo.pdf")))
  (opam "satysfi-class-yabaitech-doc.opam")
  (dependencies ((class-yabaitech ()))))
```

`(library ...)` の部分がクラスファイル本体を、`(libraryDoc ...)` がドキュメントに関する記述になっています。これらも概ね雰囲気で書いてもらえれば大体問題ないと思います (雑すぎる .....)。一つ気をつける点があるとすれば `(sources ((packageDir "src")))` でしょうか。これによって `src` 以下のファイルが全て .satysfi 下の class-satysfi ディレクトリの直下に移動されます。結果として例えば `src/yabaitech.satyh` を `@require class-yabaitech/yabaitech` とすることでインポートできるようになるわけです。インストールするファイルを一つずつ指定したい場合は `(package <src> <dst>)` と書くこともできますが、基本的に `packageDir` を使っておけば問題はないと思われます。

2.2. Satyrographos パッケージを公開する

ここまで来たらあとはもうパッケージを Satyrographos のリポジトリに公開するだけです。Satyrographos のリポジトリは na4zagin3/satyrographos-repo [*6]で管理されているのでここに PR を出せばいいのですが、opam-publsih を使うことでこの作業を半自動化することができます。最初に以下のコマンドを実行して PR の雛形を作ります。

```
$ opam publish --repo=na4zagin3/satyrographos-repo
```

PR を作る過程でいくつか入力を求められますが、 適当に済ませていくと satyrographos-repo への PR が作られます。そうすると bot からそのパッケージを package snapshot に追加するように言われるので、言われた通りに PR に変更を追加しましょう。

satyrographos-repo の CI が通れば特に問題なくマージされるでしょう。これで class-yabaitech の新しい版がどこからでも使えるようになります。

2.3. おまけ：パッケージの定義の健全性を CI で確認する

先ほど「こういうファイルは大体雰囲気で書けばいい」とは言ったものの、それもちゃんと動作してこそです。動作を確認するお手軽な方法は実際にそれらの opam/Satyristes ファイルを使ってパッケージをインストールしてみることです。具体的には以下のコードを実行することで手元の環境にパッケージをインストールすることができます。

```
$ opam pin add .
$ satyrographos install -l class-satysfi-yabaitech \
    -l class-satysfi-yabaitech-doc
```

とりあえずこれらのコマンドが失敗せずに終了すれば、opam/Satyristes がある程度ちゃんと書けているということができるわけです。

さらにこれらのコマンドを CI で実行することによってうっかりパッケージ定義のチェックを忘れるなんて事態も防ぐことができます。Github Actions でこれを行うには `.github/workflows/build.yml` に以下のジョブを定義します。

6 https://github.com/na4zagin3/satyrographos-repo

```
on: [push, pull_request]


jobs:
  check-opam-sanity:
    name: Check sanity of opam/Satyristes
    runs-on: ubuntu-latest
    container:
      image: amutake/satysfi:0.0.5
    steps:
      - uses: actions/checkout@v1
      - name: Try install class-yabaitech and class-yabaitech-doc
        run: |
          export HOME=/root
          eval $(opam env)
          opam update
          opam pin add "file://${PWD}"
          satyrographos install -l class-yabaitech -l class-yabaitech-doc
```

こうすることで push / PR がある度にサンドボックス環境でパッケージのインストールを行なってくれて結構便利です。ちなみに `container` の部分に書いてあるように SATYSF$_{\mathrm{I}}$ / Satyrographos の使える docker image として amutake さんのものを使用しています。

3. Regression Test で保守性を高める

Regression Test とはソフトウェアの変更を行なった際にその変更が予想外の変更を起こしていないかを確認するためのテストを指します。`class-yabaitech` にもある種の regression test があり、これによって最新のクラスファイルが壊れていないことを容易に確認することができるようになっています。

クラスファイルの Regression Test を書くする具体的なメリットはいくつかありますが、もっとも重要なのはコードのリファクタリングがしやすくなることです。基本的にリファクタリングはソフトウェアの動作を何も変更しないことを要求するので、Regression Test が問題なく通

るかどうかさえ確認すればコードを壊すかどうか心配する必要もなくがんがんリファクタしていけるんですね。yabaitech.tokyo の場合だともともと 1 ファイル 1181 行 に全てが書かれていたクラスファイルを壊すことなく 30 個くらいのモジュールに分割できました。

3.1. class-yabaitech の Regression Test

class-yabaitech では `__test__` ディレクトリ下にテスト関連のコードを置いています。具体的な構成は以下のようになっています。

```
__test__
  __pdf_snapshots__  -- スナップショット
  integration        -- 全体のテスト
      ...
  toc                -- 目次ページ単体のテスト
      ...
  fakedoc.satyh      -- テスト用補助ファイル
  regression.test.js -- テスト定義
```

class-yabaitech の Regression test はテスト用の SATySFI ファイルをコンパイルして `__pdf_snapshots__` 下の pdf と比較することで行われます。コンパイル結果がスナップショットと一致すれば OK、 一致しない場合には `__diff_output__` というディレクトリが作られてそこに差分のファイルが出力されます。 これを見て意図した差分が得られていればスナップショットを更新し、そうでない場合には実装のコードを修正することで開発を進めます。

3.2. Regression Test のセットアップ

このような Regression Test を実際にセットアップする手順を記しておきます。 まず最初に必要な依存関係をインストールしておきましょう。

- Node.js + Yarn
  - テストの記述と実行に必要
  - https://classic.yarnpkg.com/en/docs/install/
- diff-pdf

- pdf の同一性を確認するために必要
- https://github.com/vslavik/diff-pdf/blob/master/README.md

これらのインストールが完了したら yarn で javascript プロジェクトを新たに作成します。

```
$ yarn init
```

純粋にテストするためだけのプロジェクトで publish するものではないので質問には雑に答えていっても問題ないと思われます（private は yes にしといてもいいかもしれないですね）。答え終わったら `package.json` が作成されているはずです。 そうしたら次にテスト用の依存関係を追加していきます。

```
$ yarn add jest jest-pdf-snapshot shelljs tmp --dev
```

jest はテスティングフレームワーク、jest-pdf-snapshot は pdf のテストを補助してくれるツールで、shelljs と tmp はテスト中で SATYSF$_{\mathrm{I}}$ を実行するために必要な依存関係です。

依存関係をセットアップできたらテストを書いていきましょう。 最初に必要なのはテスト用の SATYSF$_{\mathrm{I}}$ ファイルです。 ここでは `__test__/test.saty` と名前をつけておきましょう。このファイルには描画結果を確認したいコマンドなどを書いておきます （実際のコードは省略します）。SATYSF$_{\mathrm{I}}$ のコードが用意できたら次は js で描画結果をテストするコードを書きます。 こちらは `regression.test.js` とでも名付けておきましょう（注： jest のテストコードは `.test.js` か `.spec.js` で終わる必要があります [4]）。

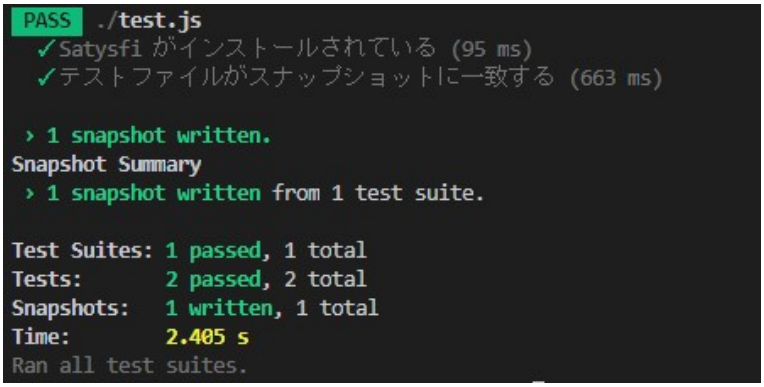
```
const shell = require("shelljs");
const tmp = require("tmp");

const { toMatchPdfSnapshot } = require("jest-pdf-snapshot");

expect.extend({ toMatchPdfSnapshot });

shell.cd("__test__");
```

```
// src の SATySFi ファイルをコンパイルする
function compileSatysfi(src) {
  const tmpFile = tmp.fileSync();

  const { code: exitCode } = shell.exec(`satysfi ${src} -o ${tmpFile.name}`, {
    silent: true,
  });

  const pdfBuffer = fs.readFileSync(tmpFile.name);
  tmpFile.removeCallback();

  return {
    exitCode,
    pdfBuffer
  };
}

// 各テスト後のクリーンアップ処理
afterAll(() => {
  shell.rm("*test.satysfi-aux");
});

test("Satysfi がインストールされている", () => {
  expect(shell.exec("satysfi -v").code).toBe(0);
});

test("テストファイルがスナップショットに一致する", () => {
  const result = compileSatysfi("test.saty");

  // コンパイルが成功する
  expect(result.exitCode).toBe(0);
```

```
  // コンパイル結果がスナップショットに一致する
  expect(result.pdfBuffer).toMatchPdfSnapshot();
});
```

ここまで書けたら大体テストは完成です。この時点ではスナップショットがないので、初回のテスト実行でスナップショットが作成されます。

```
$ jest
```



3.3. CI で Regression Test を実行する

テストができたらそれを CI で回さない手はありません。ここでは Github Actions で push / PR 毎に Regression test を実行するように設定してみましょう。

```
on: [push, pull_request]


jobs:
  regression-test:
    name: Run regression test
    runs-on: ubuntu-latest
    container:
      image: zeptometer/satysfi-yarn-diff-pdf:satysfi0.0.5
    steps:
```

```
      - uses: actions/checkout@v1
      - name: Install Yarn dependencies
        run: yarn install
      - name: Install Satyrographos dependencies
        run: |
          export HOME=/root
          eval $(opam env)
          opam update
          opam pin add satysfi-class-yabaitech.opam "file://${PWD}"
          satyrographos install -l class-yabaitech
      - name: Run regression tests
        run: |
          export HOME=/root
          eval $(opam env)
          jest --ci
```

Regression test の実行のためには SATYSFI だけでなく Yarn と diff-pdf が必要なので、Docker イメージには自前の zeptometer/satysfi-yarn-diff-pdf を使っています。 これで壊れたコードを push した際には自動で検出できるので便利です。

4. まとめ

この記事では SATYSFI クラスファイルを作る上で「コードを書く」以外の部分の yabaitech.tokyo での経験を共有しました。 これがベストな方法かというとかなり怪しい部分はありますが、それでもどういう問題があってどういう風にすればそれなりに解決するか、というノウハウが同様の問題に取り組んでいる方のお役に立てれば ..... いいなあ。

ところで：初稿の段階で mh と na4zagin3 さんに事前にレビューしてもらい改善のアドバイスをいただきました。お二人の協力に感謝します。

参考文献

[1] Takashi Suwa. *The SATYSFIbook*. 同人誌 , 2018.

[2] na4zagin3. **nix* 向け *SATySFi* インストールバトル手引き *2020* 年 *8* 月版. https://qiita.com/na4zagin3/items/a6e025c17ef991a4c923, 2020.

[3] na4zagin3. 素敵なライブラリを *Satyrographos* で配布しよう！. https://qiita.com/na4zagin3/items/b392f5d522f9bcc0493b, 2020.

[4] Facebook Inc. *Configuring Jest*. https://jestjs.io/docs/en/configuration.html#testmatch-arraystring, 2020.

アスペクト指向プログラミングで C++ プログラム用モニタを実装した話

MasWag

1. 自動でログ取りたくない？

プログラムの実行に対して色々な場面でログを取りたくなると思います。例えばデバッグの際にプログラムの流れやプログラム中での変数の値を捕捉するために printf デバッグを行うこともありますし、ユーザの入出力を捕捉するために関数の入出力をログに取ることもあります。デバッグの場合は gdb や各種 IDE に付属のデバッガを使って対話的にプログラムの流れや変数の値を確認することができますが、稀にしか発生しないバグを調べる際にはより非対話的にプログラムの様子を「モニタリング」したい場合もあると思います。

本章では特に C++ のプログラムに対して「モニタリング」用のコードを自動生成する簡易的なツール、**asm.c++ (A Simple Monitor for C++)** を作ってみます。asm.c++ は GitHub(https://github.com/MasWag/asmcpp) にて公開しています。asm.c++ はアスペクト指向プログラミングによって「モニタリング」用のコードを自動で挿入します。本章で行う「モニタリング」は単に `printf` や `std::cout` を適宜挿入するだけなので、手動で行うことも可能です。但し、モニタリングについてはビジネスロジックと無関係ですし、色々な関数をモニタリングしたい場合は多くの変更が必要となるので管理のコストが小さくありません。本章の目標はこれらの手間を自動化することにあります。

なお、システムのモニタリングという話題では**実行時検証 (runtime verification)** と呼ばれる分野もあります。実行時検証ではシステムの動作を表わすログとシステムの充たすべき仕様を受け取り、ログ中で仕様を違反する挙動があるかどうかを判定します。今回のモニタリングではこの様な自動判定は行ないませんが、今回示す様なアスペクト指向プログラミングの手法は

例えば JavaMOP[1] 等実行時検証の実装に良く使われています。

2. asm.c++ の概要

この節では asm.c++ の概要や構成技術について説明していきます。

2.1. アスペクト指向プログラミング

asm.c++ では**アスペクト指向プログラミング (Aspect Oriented Programming、AOP)** [2] の技術を使ってモニタを C++ のソースコードに挿入しています。アスペクト指向プログラミングでは複数の関数やクラスを横断する様な機能を**アスペクト**と呼び、各関数やクラスの実装から分離する試みのことです。具体的には、例えばある条件を充たす関数の最初や最後に特定の処理を挿入することができます。なお、この様に挿入される各処理を**アドバイス**と呼びます。

様々な言語に対してアスペクト指向プログラミングが実装されていますが、C++ 向けのアスペクト指向プログラミング処理系として **AspectC++**[3] があります。asm.c++ では AspectC++ を使うことで、既存の C++ のプログラム中にモニタリング用のコードを挿入します。

2.2. モニタ生成のアーキテクチャ

asm.c++ は図 1 にある様なワークフローで既存の C++ コードにモニタを挿入します。まず始めに asm.c++ は内部の AWK コードを使うことでモニタリング対象の関数を記載した設定ファイルを受け取り、AspectC++ のコードを生成します。次にモニタリング用の AspectC++ コードから、AspectC++ を使うことでモニタ付きの C++ コードを生成します。asm.c++ が生成したモニタ付き C++ コードは通常の C++ コードなので、後は通常のコンパイルと同様に適宜 gcc や clang 等の C++ コンパイラを使ってコンパイルすることができます。

2.3. asm.c++ の設定ファイル

asm.c++ を使う際はモニタリング対象の関数を指定するために設定ファイルを書く必要があります。設定ファイルのフォーマットは以下の様になります。

```
([RETURN_TYPE]) [FUNCTION_NAME], [TYPE] [ARG_NAME], [TYPE] [ARG_NAME], ...
([RETURN_TYPE]) [FUNCTION_NAME], [TYPE] [ARG_NAME], [TYPE] [ARG_NAME], ...
...
```

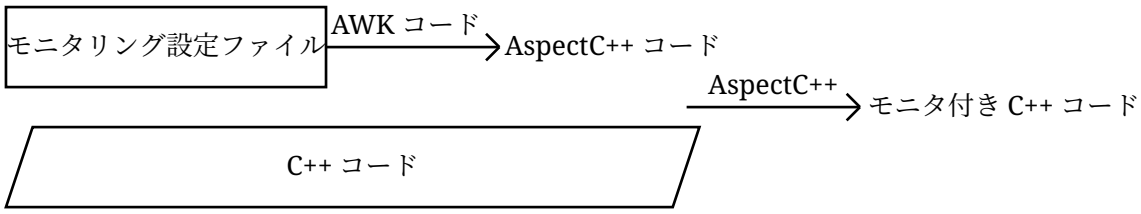


図 1　asm.c++ によるモニタ生成のワークフロー

設定ファイルの各行が一つの**モニタリングルール**に対応します。ざっくり言うとモニタリングしたい関数一つに対して一つのモニタリングルールを書く必要がありますが、AspectC++ の Function Matching を使うことで複数の関数に対応するモニタリングルールを記述することもできます。各行はカンマ区切りの複数フィールドから成ります。 カンマで区切られていますがクオーテーションに対応していない点が CSV とは異なります。

各行の第一フィールドは、 モニタリング対象の関数の返り値の型とモニタリング対象の関数名を示し、それ以降のフィールドが関数の引数の型と名前を示します。なお、関数の返り値の型である `[RETURN_TYPE]` を省略した際は返り値がモニタリングされません。返り値をモニタリングしないで関数の終了のみをモニタリングしたい場合は `[RETURN_TYPE]` に `void` を記述します。第二フィールド以降が省略された際は関数の引数はモニタリングせず、呼び出しのみがモニタリングされます。

例えば C/C++ で `int fibonacci(int i);` の様な型の関数に対するモニタリングルールは以下の様になります。

```
int fibonacci, int i
```

2.4. asm.c++ の使用方法

asm.c++ の使用方法は以下の様になります。

```
asmc++ [MONITORED_FILE] -c [CONF_FILE] -o [OUTPUT_FILE]
```

例えば設定ファイルが `fibonacci.conf`、モニタリング対象のファイルが `fibonacci.cc`、モニタ挿入後のファイルが `fibonacci_monitored.cc` の場合、以下の様になります。なお現在のところ引数の解析周りの実装が雑なので引数の順序を変更することはできません。

```
$ asmc++ fibonacci.cc -c fibonacci.conf -o fibonacci_monitored.cc
```

例えば fibonacci.cc、fibonacci.conf が以下の場合、モニタ挿入後のファイル fibonacci_monitored.cc をコンパイルして実行すると以下の様な実行になります。「ASM」 から始まる行が挿入されたモニタによる出力になります。

```
/* fibonacci.cc */
#include <cstdio>
#include <cstdlib>

int fibonacci_impl(int a, int b, int c) {
  if (c == 0) {
    return b;
  } else {
    return fibonacci_impl(b, a + b, c - 1);
  }
}

int fibonacci(int i) {
  return fibonacci_impl(0, 1, i);
}

int main(int argc, char *argv[])
{
  int n;
  if (argc > 1) {
    n = atoi(argv[1]);
  }
  printf("%d\n", fibonacci(n));
  return 0;
}
```

```
/* fibonacci.conf（コメントには未対応なので、この行は削除する必要があります）*/
fibonacci_impl, int a, int b, int c
```

```
$ ./fibonacci_monitored 10
ASM: fibonacci_impl, a: 0, b: 1, c: 10
ASM: fibonacci_impl, a: 1, b: 1, c: 9
ASM: fibonacci_impl, a: 1, b: 2, c: 8
ASM: fibonacci_impl, a: 2, b: 3, c: 7
ASM: fibonacci_impl, a: 3, b: 5, c: 6
ASM: fibonacci_impl, a: 5, b: 8, c: 5
ASM: fibonacci_impl, a: 8, b: 13, c: 4
ASM: fibonacci_impl, a: 13, b: 21, c: 3
ASM: fibonacci_impl, a: 21, b: 34, c: 2
ASM: fibonacci_impl, a: 34, b: 55, c: 1
ASM: fibonacci_impl, a: 55, b: 89, c: 0
89
```

3. asm.c++ の実装

この節では asm.c++ の実装について説明していきます。asm.c++ はシェルスクリプトです。asm.c++ の全体像は以下の様になります。

```
<<CHECK_ASPECT_C>>

<<CHECK_ARGS>>

# Generate the Aspect C++ code
ASPECT_TMP=$(mktemp "$(dirname "$input_file")"/"$input_file".XXXXXX.ah)
<<GENERATE_HEADER>>
<<GENERATE_BODY>>
<<GENERATE_FOOTER>>
```

```
# weave monitor
<<WEAVE_MONITOR>>
```

3.1. Aspect C++ の確認 (<<CHECK_ASPECT_C>>)

まず始めに AspectC++ がインストールされているかどうか確認する必要があります。今回はパスの通っている場所に ag++ が存在するかどうかを調べます。

```
if ! command -v ag++ > /dev/null; then
    echo "asm.C++ Error: ag++ is not found." > /dev/stderr
    echo "asm.C++ Error: asm.C++ requires that ag++ is installed under a pathed
directory." > /dev/stderr
    exit 1
fi
AG=$(command -v ag++)
```

3.2. 引数の確認 (<<CHECK_ARGS>>)

次に引数を確認します。前述の通りとても雑な実装になっています。

まず、引数のフォーマットは [MONITORED_FILE] -c [CONF_FILE] -o [OUTPUT_FILE] なので、引数は 5 個必要になります。なお <<USAGE_TEXT>> の箇所には適切な使い方を書いたメッセージが入ります。

```
if [ $# -lt 5 ]; then
    echo "Error: not sufficient arguments" > /dev/stderr
    cat <<EOF > /dev/stderr
<<USAGE_TEXT>>
EOF
    exit 1
```

更に、第二引数と第四引数はそれぞれ -c と -o である必要があります。

```
elif [ "$2" != "-c" ] || [ "$4" != "-o" ]; then
    echo "Error: invalid usage" > /dev/stderr
    cat <<EOF > /dev/stderr
<<USAGE_TEXT>>
EOF
    exit 1
fi
```

引数が適切に与えられた場合、以下の様に適当な変数に代入します。

```
input_file=$1
conf_file=$3
output_file=$5
```

3.3. AspectC++ コードの生成

次に、AspectC++ コードを生成していきます。ここがメインパートになります。

ヘッダとフッタの生成 (<<GENERATE_HEADER>> 及び <<GENERATE_FOOTER>>)

生成される AspectC++ コードのヘッダとフッタについては固定されているので、単に固定された文字列を出力します。

```
cat <<EOF > "$ASPECT_TMP"
#include <iostream>

aspect ASM {
EOF

cat <<EOF >> "$ASPECT_TMP"
};
EOF
```

本体の生成 (<<GENERATE_BODY>>)

次に AspectC++ のコードの本体を生成します。別途 AWK のコードを生成して、そのコードを呼び出して本体を生成します。

```
TMP=$(mktemp /tmp/asmc++.XXXXXX.awk)
cat << 'EOF' >> "$TMP"
<<GENERATE_AWK>>
EOF

chmod +x "$TMP"
awk -f "$TMP" < "$conf_file" >> "$ASPECT_TMP"
rm "$TMP"
```

AspectC++ コード生成用の AWK コード (<<GENERATE_AWK>>)

ここから先が AspectC++ コード生成用の AWK コードになります。

まず asm.c++ の設定ファイルはカンマ区切りのため、AWK のフィールド区切りをカンマ (,) に設定します。

```
BEGIN {
  FS = ",";
}
```

この AWK コードでは return_type、name、args の三つの変数を使います。それぞれの使われかたは以下の様になっています。

- return_type: 返り値の型を表す文字列。返り値をモニタリングするときのみ設定される。
- name: モニタリング対象の関数を表す文字列。 単に関数名を書くのみではなく、AspectC++ の Function Matching を使うことができる。
- args: モニタリング対象の関数の引数名を表す文字列の配列

以下では asm.c++ の設定ファイルをパースして上記変数に適切な値を格納していきます。

```
{
    match($1, /[^ ]*$/);
    return_type = substr($1, 1, RSTART-1);
    name = substr($1, RSTART, RLENGTH);
    for (i = 2; i <= NF; i++) {
        match($i, /[^ ]*$/);
        args[i] = substr($i, RSTART, RLENGTH);
    }
}
```

ここからが AspectC++ のコードを生成する部分です。

まず始めに return_type が空でなく void 以外の型が入っている場合に関数の結果をモニタリングするアドバイスを生成します。 なお結果の出力には std::cout を使うため、 適切に operator<< がオーバーロードされていれば自作のクラスに対してもモニタリングが可能です。

```
length(return_type) && !match(return_type, /void/) {
    printf "  advice execution(\"%% %s(...)\") && result(res) : after (%s res){
\n", name, return_type;
    printf "    std::cout << \"ASM: %s\" << \", result: \" << res << std::endl;
\n", name;
    print "  }"
}
```

return_type が void の場合は関数の終了のみをモニタリングするアドバイスを生成します。

```
match(return_type, /void/) {
    printf "  advice execution(\"%% %s(...)\") : after () {\n", name;
    printf "    std::cout << \"ASM: %s\" << std::endl;\n", name;
    print "  }"
}
```

次に、引数及び関数の呼び出しをモニタリングするアドバイスを生成します。

まず始めにアドバイス定義の最初の部分を生成します。

```
{
    printf "  advice execution(\"%% %s(...)\")", name;
}
```

次に、引数をモニタリングする場合は引数のマッチングを行います。

```
NF > 1 {
    printf " && args(";
    for (i = 2; i <= NF; i++) {
        printf "%s",args[i];
        if (i < NF) {
            printf ", "
        }
    }
    printf ")"
}
```

更に、引数をキャプチャします。

```
{
    printf " : before (";
    for (i = 2; i <= NF; i++) {
        printf "%s",$i;
        if (i < NF) {
            printf ", "
        }
    }
    printf ") {\n"
}
```

最後に引数を出力します。

```
{
    printf "    std::cout << \"ASM: %s\"", name;
    for (i = 2; i <= NF; i++) {
        printf " << \", %s: \" << %s", args[i], args[i];
    }
    printf " << std::endl;\n  }\n\n"
}
```

3.4. AspectC++ を用いたモニタ付き C++ コードの生成

最後に、生成した AspectC++ のコードを使うことでモニタ付きの C++ コードを生成します。ここでコンパイルまで行ってしまうことも可能ですが、 ユーザが自由にコンパイル時のオプションを指定できる様にするために、今回はプリプロセス後に C++ のコードを生成するところまでを行います。なお ag++ は指定されたディレクトリ以下の .ah ファイルを自動で使うので、ASPECT_TMP を明示的に与える必要はありません。*7

```
$AG "$input_file" -p "$(dirname "$input_file")" --keep_woven --weave_only\
 -o "$output_file"
rm "$ASPECT_TMP"
```

4. まとめ

いかがだったでしょうか !!

本章ではアスペクト指向プログラミングによる簡単なモニタリングツールの実装 asm.c++ について説明しました。asm.c++ は実用的なモニタリングツールというより、アスペクト指向プログラミングを使うと簡単にモニタの注入ができるということを示すための簡易的な実装になっています。本格的なモニタリングツールとして使うためには、例えば以下の様な改善が必要だと思われます。

7 「asm.c++ でコンパイルオプションを上手く指定できる様にすれば良いじゃないか」と思うかもしれませんが、割と真面目に引数の解析が必要になって実装が複雑になると思うので今回はやりません。

- 関数の終了時にも引数を出力できる様にする (C/C++ ではポインタや参照渡しで結果を受け取ることも良くある)
- ポインタを受け取った場合、適切な内容を出力できる様にする
- 標準出力以外にログを吐ける様にする

参考文献

[1] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, Grigore Rosu. JavaMOP: Efficient parametric runtime monitoring framework.. In *ICSE*, pages 1427–1430, 2012.

[2] Wikipedia. アスペクト指向プログラミング . `https://ja.wikipedia.org/wiki/`アスペクト指向プログラミング , 2020.

[3] Olaf Spinczyk, Daniel Lohmann, Matthias Urban. Advances in AOP with AspectC++. In *SoMeT*, pages 33–53, 2005.

F-ing modules の型検査とコンパイル手法

gfn

1. はじめに

学習・趣味・実用など目的を問わず，プログラミング言語を設計・実装するとなったとき，マトモな人なら型検査器を用意するはずです[要出典]．そして多くの場合は既によく知られ実績のある **let 多相 (let-polymorphism)** [7, 8] を型システムの土台に採用し，ついでに型推論器まで作り込んだりすることかと思います[要出典]．モノ好きなら，この let 多相をベースとして，好みの言語機能とその型システムを追加していくことかと思います．このように，ちょっとした静的型つき言語をつくるとなれば，let 多相という多くの人がひとまず土台にしようと考える定番があるわけです．

さて，趣味や実用を目的として制作した言語でも，使用規模が大きくなってくると避けられない問題があります．それはいわゆるカプセル化，名前空間の分離，分割コンパイル云々といった，大規模なプログラムの構築を人間にも堪えられるようにする工夫の必要が出てくることです．さもなければ，当然ながら以下のような事態に見舞われるためです：

- 密結合なコード群が生まれ，機能追加や修正の際にどこをいじれば事足りるのか到底把握できない
- 新たに追加した定義の名前がどこかで既に定義されていた名前と重複してしまい，その定義がスコープに入っている箇所が壊れる
- ちょっとした変更でも毎回全てのコードをコンパイルし直さねばならず時間がかかる

こうした問題に対処する工夫は古くから様々な観点で考えられてきており，その手法も計算機科学らしいものからソフトウェア工学らしいものまで色々あるはずですが，特に本記事で注目

するのは Standard ML や OCaml などの ML 系言語で採用された**モジュールシステム**と呼ばれる機構です．ML 系言語でプログラムを書いた経験のある方はよくご存知かと思いますが，これはプログラムを**モジュール (module)** という単位で分割して扱うことで上記の問題を大きく緩和できるような，型システムとして定式化された仕組みです．特にカプセル化や名前空間に関する保証が型検査のフェーズでできるようになっている上，**ファンクタ (functor)** と呼ばれる “モジュールをモジュールに写す函数” のような機能まで定式化されており[*8]，これにより抽象化能力ひいてはコードの再利用性も高められているのが特徴です．

このように実用上の利便性に大きく貢献するモジュールシステムですが，その型システムの設計・実装の方法については核言語のそれに比べればおそらくあまり知られていないかと思います．特に「型検査器つくるならまあとりあえず let 多相が土台かな」という具合のスイートスポット的存在がモジュールシステムについては認知されていないだろうと思われます．

本記事では，そのようなモジュールシステムの定番となりうるものとして独断と偏見により **F-ing modules [11]** を紹介したいと思います．著者が実際に F-ing modules を土台としてモジュールシステムを備えた言語を設計・実装した経験から，（複雑多岐に混み入った定式化になりがちなモジュールシステムにありながら）筋の良い定式化が施されたものと直感したためです．モジュールシステムの提案自体は歴史的にも様々に為され種々の異なる型システムが存在し，一応それらについても簡単に触れたいと思いますが，それら従来の提案と精緻に比較検討した上でこの F-ing modules を選択したというわけではないことは念のためご了承ください．

2. モジュールシステムの基礎

この節ではモジュールという機構がどんなものであるかを簡単におさらいします．既に OCaml や Standard ML などの言語で既にモジュールシステムを使いこなしている方は読み飛ばして構いません．

2.1. モジュールとは

まずは永続的なスタックを扱うモジュールをリストで実装する例を見てみます．

スタックは（おそらくご存知かと思いますが）“要素を積んで置いておくことができ，一番上からしか取り出せない機構” であり，LIFO (= last-in first-out) を実現するデータ構造です．し

8　モナド等の文脈で登場するファンクタとは関係のない概念であることに注意．

たがって，スタックは

(1) 「空のスタックを返す empty」「要素を新たに積む push」「（空でなければ）一番上の要素を取り出す pop」「（空でなければ）一番上の要素を取り出さずに見る top」といった操作が API として提供されていて，かつ
(2) 一番上以外の要素を（それより上の要素を取り出さないまま）閲覧・変更することができないようになっていなければなりません．

ひとまずこれらの操作を簡単にリストで実装してみたのが以下です（実際に動かせる OCaml コードです）：

```
module Stack = struct
  type 'a t = 'a list

  let empty = []

  let push elem stack = elem :: stack

  let pop stack =
    match stack with
    | []            -> None
    | elem :: stack -> Some (elem, stack)

  let top stack =
    match stack with
    | []        -> None
    | elem :: _ -> Some elem
end
```

永続的なスタックの API なので，push は新しい要素が追加されたスタックを戻り値とし，pop は取り出した要素と取り出された残りのスタックとの組を（Some でくるんで）返します．これで (1) の要件は満たしているわけです．

一方，(2) の要件はまだ満たせていません．単にモジュールを定義しただけだと，スタックが

リストで実装されているという事実が隠蔽できていないのです．そのため，次のような計算が書けてしまい，スタックの途中の要素がそれより上の要素を取り出さずに見えてしまったりします：

```
let get_second (stack : 'a Stack.t) =
  match stack with
  | _ :: elem :: _ -> Some elem
  | _              -> None
```

“API から必然的に定まるわけではない内部実装の形式” が漏れていると，上記の `get_second` のような内部実装に依存した処理が “API の外側” で不用意に書けてしまい，内部表現を将来的により効率的なものに置き換えたいとなったときに支障をきたしたりします．

そこで (2) を達成するべく，“API から必然的に定まるわけではない内部実装上の形式” がモジュールの外部には漏れないことを型システム的に保証するための言語機能を導入したくなります．それが**シグネチャ (signature)** と呼ばれる機構です．シグネチャとはモジュールにつく型のようなもので，平たく言えば “API の過不足ない記述” であり，どんな型や函数などが備わっているのかの情報を外部に伝えます．裏を返せばそれより詳しくは伝えません．百聞は一見に如かずということで，スタックのモジュール `Stack` の API としてふさわしいシグネチャを掲げます：

```
sig
  type 'a t
  val empty : 'a t
  val push : 'a -> 'a t -> 'a t
  val pop : 'a t -> ('a  * 'a t) option
  val top : 'a t -> 'a option
end
```

`type 'a t` が「実態が何であるかは明かせないが，スタックを表現するためにこういう型を用意しているよ」という主張で，また他の `val empty :…` などがその型コンストラクタ `t`（外部から見ると `Stack.t`）を使って抽象化された型がつけられた函数たちの宣言です．OCaml では以下のように：`sig…end` とシグネチャをつけてモジュールを宣言することでそのシグネチャに書かれ

た宣言の内容のみが “外部から見えるモジュールの役割” になります：

```
module Stack : sig
  type 'a t
  val empty : 'a t
  …
end = struct
  type 'a t = 'a list
  let empty = []
  …
end
```

こう宣言しておくと，上で掲げた get_second を定義しようとしても，Stack の外部には t の実態がリストであることは漏れないので 「'a Stack.t と 'a list で型が合わない」 という旨の型エラーが出て弾かれるという算段です．こうした疎結合性を保証した実装にしておくことで，後々別の内部表現（例えばスタックを連続的に確保されたメモリ領域でスタックポインタを用いて扱うような低級なもの）に切り替えたくなっても，特にモジュールの外部は変更せず，或いは変更が必要だとしてもどのように変更すればよいかが明瞭で，大きくは困らずに変更できるわけです．こうした疎結合性の静的な担保がモジュールとシグネチャの強みであり，プログラムが或る程度の規模を超えてくると開発の継続に際してほとんど必須になってきます．

2.2. ファンクタとは

実際には，前節で扱った struct…end の形のモジュール（これを**ストラクチャ (structure)** と呼びます）やそれに対応するシグネチャ sig…end だけでは抽象化の機能として足りないことがあります．簡単な例としてマップを抽象化して扱いたい場合を考えましょう．マップは keyval とか辞書などとも呼ばれ，要するに有限個のキーをそれぞれ対応する値に紐づける機構です．このマップをキーと値の型に依らず汎用的に扱うモジュールを定義することを考えてみます．ひとまずシグネチャは以下のようになりそうな気がします：

```
sig
  type ('k, 'v) t
  val empty : ('k, 'v) t
```

```
  val add : 'k -> 'v -> ('k, 'v) t -> ('k, 'v) t
  val lookup : 'k -> ('k, 'v) t -> 'v option
  …
end
```

これに合わせてストラクチャを実装してみましょう ..... と言いたいところですが，ここでマップのキーとなる型には制約があることに気づきます．キーとなる型は，何らかの方法でキー同士の等価性が判定できなくてはならないのです．さもなければ，紐づけをマップに追加する函数 `add` やキーによってマップから値を引く `lookup` の実装に於いて，キーに対応する値が既に入っているのか否かを判別する手段がありません．大抵の基本的な型（整数，文字列，整数と文字列のペア, etc.）は等価性が判定できますが，例えばキーの型が `int -> bool` である場合はどうでしょうか？ 函数同士の等価性とはなんなのか，そしてそれは決定可能なのか，など種々の問題が現れます．こうした問題を解決してマップのモジュールを定義するにはどのようにすればよいでしょうか？

ひとつの選択肢は「キーの型は任意のものを許すが，等価性の判定できない型が使われた場合は評価中にエラーとする」という方針です．実際, OCaml でも等号 `=` は `'a -> 'a -> bool` と特に制限のない多相型がついており，函数のような比較が計算できないデータが与えられた場合は型検査では弾かれず実行時に失敗します．しかし，できれば実行時の失敗は避けられる言語仕様にしたいものですし，さらに言うならば等価性の判定方法は常にあらかじめ決まったものだけに限定せず自分で指定したいこともあります（例えば文字列上の等価性は NFC などの正規化処理で割ったものにしたいかもしれません）.

さらなる解決を図る選択肢として，ここでは**ファンクタ (functor)** を考えます[*9]．ファンクタは大雑把に言えば “モジュールを受け取ってモジュールに写すデカい函数” で，モジュールを受け取れることを利用して型とその型の上の一揃いの演算をパラメータにすることができます.

マップをファンクタで抽象化することを考えます．キーとして “等価性が判定できる型” のみを扱いたいので，キーに関するデータを「型とその上の等価性判定の函数を含むモジュール」で受け取るファンクタにします．したがって，キーに関するモジュールに課すシグネチャは以

9 **型クラス (type class)** の導入なども選択肢に上り得ます．ファンクタで定式化するか型クラスで定式化するか（或いは両方とも入れるのか）には様々な側面でトレードオフがあり，言語設計上の好みの問題という感じではあります．また，繰り返しになりますが，ここでいうファンクタはモナドの文脈で登場するファンクタ（Haskell の `Functor` 型クラスなど）とは特に関係がない概念であることに注意.

下のような具合になります（OCaml では `module type` というキーワード列でシグネチャの束縛・宣言をします）：

```
module type EQ = sig
  type t
  val equal : t -> t -> bool
end
```

このシグネチャをもつモジュールを受け取ってその型 `t` をキーの型とし，等価性判定 `equal` をもとにした挿入・検索等を実装したモジュールを返すファンクタが以下のように連想配列によって超ナイーヴに実装できます：

```
module Map = functor(Key : EQ) -> struct
  type key = Key.t
  type 'a t = (key * 'a) list
  let empty = []
  let add k0 v0 assoc =
    let rec aux acc assoc =
      match assoc with
      | [] ->
          List.rev ((k0, v0) :: acc)

      | (k, v) :: tail ->
          if Key.equal k k0 then
            List.rev_append ((k, v0) :: acc) tail
          else
            aux ((k, v) :: acc) tail
    in
    aux [] assoc
end
```

`functor(Key : EQ) -> struct…end` が“ファンクタ抽象”であり，`Key` が受け取るモジュールに束縛された識別子です．挿入用の函数 `add` 中で `Key.equal` を用いてキーの等価性を判定してい

るのがファンクタを用いている根本的理由です．ここでは挿入処理 add しか実装していませんが，検索処理 lookup なども同様に定義できます．実際に utop で動かすとたしかに動作します（# から始まる行が入力として与えたものです）[10]：

```
# module IntMap = Map(struct type t = int let equal = Int.equal end);;
module IntMap :
  sig
    type key = int
    type 'a t = (key * 'a) list
    val empty : 'a list
    val add : key -> 'a -> (key * 'a) list -> (key * 'a) list
  end
# let map = IntMap.add 42 "universe" IntMap.empty;;
val map : (int * string) list = [(42, "universe")]
# let map = map |> IntMap.add 57 "prime";;
val map : (int * string) list = [(42, "universe"); (57, "prime")]
# let map = map |> IntMap.add 42 "just a forty two";;
val map : (int * string) list = [(42, "just a forty two"); (57, "prime")]
# let map = map |> IntMap.add 57 "not a prime";;
val map : (int * string) list = [(42, "just a forty two"); (57, "not a prime")]
```

Map(…) がファンクタの適用であり，IntMap という整数をキーとするマップを作って使用しています．ただし Int.equal は OCaml の標準ライブラリであらかじめ定義されている整数上の

10 utop [1] は OCaml の REPL（対話環境）のひとつです．ここで貼った実行例は実際にコンソールに出力される文字列から多少加工しています．

等価性判定函数です[*11].

さて，動作することはわかりましたが，やはり実装がリストであることが漏れてしまっています．そこでファンクタにもシグネチャをつけます：

```
module Map : functor(Key : EQ) -> sig
  type key = Key.t
  type 'a t
  val empty : 'a t
  val add : key -> 'a -> 'a t -> 'a t
end = functor(Key : EQ) -> struct
  （実装は上記のものと同じ）
end
```

`functor(Key : EQ) -> sig…end` がファンクタにつくシグネチャです． マップの型 `'a t` は抽象化されて外からは実装が見えない一方で，キーの型 `key` は `Key.t` と同一であるとして実装が公開されている点に注意してください．さもなければキーの定義まで隠蔽されてしまい，上記の `IntMap` などを作っても `IntMap.key` と `int` が異なる型として扱われてしまいます．上記のコードも実際に `utop` で動かしてみます：

```
# module IntMap = Map(struct type t = int let equal = Int.equal end);;
module IntMap :
```

11 もっと言えば, `Int` モジュールが `Int.t` として `int` 型を提供しているので 1 行目は以下でもかまいません：

```
# module IntMap = Map(Int);;
module IntMap :
  sig
    type key = int
    type 'a t = 'a Map(Int).t
    val empty : 'a t
    val add : key -> 'a -> 'a t -> 'a t
  end
```

ただし， シグネチャの表記で `'a t` の宣言が変化していることからもわかるように， これは少し意味が変わります．OCaml のファンクタは applicative functor といって，“ファンクタによる作られ方が同じモジュールは互いに同一” ということを一部の場合に判定できる機能があります （この “applicative” もモナドの文脈のそれとは無関係）．`IntMap` 以外にも `Map(Int)` で作られたモジュール， 例えば `NumberMap` というモジュールがあったら，`NumberMap` と `IntMap` は同一のモジュールであると扱われ，例えば `string NumberMap.t` と `string IntMap.t` は全く同一の型と看なされます．F-ing modules によるモジュールシステムの元論文 [11] ではこの applicative functor も扱っているのですが，本稿では “適用ごとに異なるモジュールになる” ファンクタである generative functor のみを扱います．

```
  sig
    type key = int
    type 'a t
    val empty : 'a t
    val add : key -> 'a -> 'a t -> 'a t
  end
# let map = IntMap.empty;;
val map : 'a IntMap.t = <abstr>
# let map = map |> IntMap.add 42 "universe";;
val map : string IntMap.t = <abstr>
```

たしかにマップそのものは抽象化されて実装が見えなくなっています.

ちなみに，ここで紹介したマップは実装が連想配列なのでサイズ n に対して挿入や検索に $O(n)$ かかりますが，これでは実用上到底耐えないので，実際には以下のシグネチャで “全順序がつけられる型” を受け取って平衡二分木として実装したりします（`compare v1 v2` は `v1` の方が `v2` より大きければ正値を，小さければ負値を，等しければ `0` を返す函数[*12]）：

```
module type ORD = sig
  type t
  val compare : t -> t -> int
end
```

実際, OCaml の標準ライブラリの `Stdlib.Map.Make` はそのような API と実装になっています.

3. 構文

さて，導入も終わったところでここからは F-ing modules の定式化を見ていきましょう．図 2 に構文定義を掲げます．$x \in \mathrm{ValIdent}$, $t \in \mathrm{TypeIdent}$, $m \in \mathrm{ModIdent}$, $s \in \mathrm{SigIdent}$ がそれぞれ値，型，モジュール，シグネチャを表す識別子を動くメタ変数である下で[*13], M と S がそれぞ

12 本質的には 3 値でよく，実際 Haskell ではこれと同等な演算は `GT`, `LT`, `EQ` の 3 値をもつ ADT を戻り値として表現されています.

13 F-ing modules の元論文 [11] では主に証明上の簡便さのためこれらが X という共通のメタ変数で書かれていますが，本記事は明瞭さのためにこれらの識別子を互いに別の区分として扱うことにします.

$$
\begin{array}{rcl}
M & ::= & \{\boldsymbol{B}\} \\
& \mid & m \\
& \mid & M.m \\
& \mid & \mathbf{fun}(m : S) \to M \\
& \mid & m'(m') \\
& \mid & m :> S \\
\boldsymbol{B} & ::= & [B]^* \\
B & ::= & \mathbf{val}\ x = E \\
& \mid & \mathbf{type}\ t = T \\
& \mid & \mathbf{module}\ m = M \\
& \mid & \mathbf{signature}\ s = S \\
& \mid & \mathbf{include}\ M
\end{array}
\qquad
\begin{array}{rcl}
S & ::= & \{\boldsymbol{D}\} \\
& \mid & s' \\
& \mid & (m : S) \to S \\
& \mid & S\ \mathbf{with\ type}\ t' = T \\
\boldsymbol{D} & ::= & [D]^* \\
D & ::= & \mathbf{val}\ x : T \\
& \mid & \mathbf{type}\ t :: K \\
& \mid & \mathbf{module}\ m : S \\
& \mid & \mathbf{signature}\ s = S \\
& \mid & \mathbf{include}\ S
\end{array}
\qquad
\begin{array}{rcl}
x' & ::= & [m.]^* x \\
t' & ::= & [m.]^* t \\
m' & ::= & [m.]^* m \\
s' & ::= & [m.]^* s \\
E & ::= & \cdots \quad \text{（通常の式）} \\
& \mid & x' \\
T & ::= & \cdots \quad \text{（通常の型）} \\
& \mid & t' \\
K & ::= & \cdots \quad \text{（通常の種）} \\
& \mid & \mathrm{o}
\end{array}
$$

図 2　構文の定義

$$
\begin{aligned}
M_1(M_2) &\coloneqq \{\mathbf{module}\ m_1 = M_1; \mathbf{module}\ m_2 = M_2; \mathbf{module}\ m = m_1(m_2)\}.m \\
(M :> S) &\coloneqq \{\mathbf{module}\ m_0 = M; \mathbf{module}\ m = m_0 :> S\}.m \\
(M : S) &\coloneqq (\mathbf{fun}(m : S) \to m)(M) \\
(\mathbf{type}\ t = T) &\coloneqq \mathbf{include}\ \{\mathbf{type}\ t :: \mathrm{o}\}\ \mathbf{with\ type}\ t = T
\end{aligned}
$$

図 3　主な糖衣構文

れモジュールとシグネチャの構文を規定します．ただし，$[Z]^*$ は Z で表される構文が 0 個以上有限個連接されたものを表すメタ記法です．

E, T, K がそれぞれ式，型，種（カインド）の構文で，これらは“一定の整合性を満たしている”限り基本的には決め打ちされておらず，モジュールから値と型の要素にアクセスする構文 x' と t'，および“基本種”の o^*[14]のみが要請されます．この E, T, K のなす（通常のプログラミング言語の理論で中心的に扱われるような）言語は**コア言語 (core language)** などと呼ばれ，M と S のなす**モジュール言語 (module language)** と分離して定式化されているため，言語を実装する観点で言えば目的に応じて或る程度自由にコア言語を与えられるようになっています．

$\{\boldsymbol{B}\}$ の形の構文が**ストラクチャ (structure)** `struct…end` であり，B はモジュールに属する値，

14 慣習的には Ω と書かれることもあります．実際 F-ing modules の元論文 [11] でもこの記法を採っています．

型，入れ子のモジュールなどを**束縛**する構文です．$M.m$ はモジュール M の要素である入れ子のモジュール m を取り出す構文であり，このようなアクセス方法は前述の式と型に備わっているものと同様です．$\mathbf{fun}(m : S) \to M$ と $m'_1(m'_2)$ はそれぞれファンクタの抽象と適用であり，これらがモジュールシステムの抽象化能力を（ストラクチャのみの場合と比べて特に）向上させる立役者であると同時に，定式化を非自明にする元凶となる機構です．m' は M を一部の形に制限したもので，定義を見るとわかるように“モジュールの識別子に対して何度かの取り出しを連ねて得られるモジュール”を表します．このため，ファンクタの適用は $M_1(M_2)$ といった形ではなく一部の形に制約されていますが，この制約はあくまで定式化上“見通しをよくする”ためのものであり，一般の $M_1(M_2)$ の形は図 3 中に掲げるように糖衣構文として導入することが可能なため，使う側から見ると事実上制約にはなりません．同様に，モジュールに対するシグネチャの強制 $m :> S$ も一般のモジュールの式ではなくモジュールに識別子に対するものに制限された定式化になっていますが，やはり図 3 に示すように糖衣構文として $M :> S$ を扱えます．

シグネチャも $\{\boldsymbol{D}\}$ が `sig…end` に対応する基本となる構文であり，D は値，型，入れ子のモジュールなどを**宣言**します．$(m : S) \to S$ がファンクタにつくシグネチャの構文であり，また S **with type** $t' = T$ はマップの例で見たように一部の型の定義を限定する機構です．前章の例で見たように，シグネチャ中で型の宣言を **type** $t = T$ のように定義を露出して書きたい状況もありますが，これは図 3 に示すように **with** と **include** を組み合わせて糖衣構文として用意できるため，独立に考える必要はありません[*15]．

4. 型システム

4.1. 型や型環境の構文的定義

まず図 4 に型や型環境などの構文を掲げます．ここで $[Y \rightharpoonup Z]$ は Y の動く範囲の集合から Z の動く範囲の集合への有限部分写像全体を動くメタ記法です．τ は基本的には通常の式につくような型ですが，型パラメータをとる型抽象 $\Lambda\alpha :: \kappa.\tau$ なども含めた，種 κ がつく“型の項”です．構文を決め打ちする必要はなく，F-ing modules の元論文 [11] に基づけば少なくとも System Fω に埋め込める強さの範囲まで拡張できますが，ここではひとまず函数型 $\tau_1 \to \tau_2$，型変数 α，型抽象，型適用程度が入っているものとします．型の定義が型パラメータを取らないような単純なサブセットで構わないのであれば，種を o のみとし，型の構文も α や $\tau \to \tau$ など

15 F-ing modules の元論文 [11] では独立に用意されていますが，後述する static interpretation を扱った論文 [4] では糖衣構文として扱う旨の記述があります．

$$
\begin{aligned}
\tau &::= \alpha \mid \tau \to \tau \mid \Lambda\alpha :: \kappa.\, \tau \mid \tau\, \tau \mid \cdots & \xi &::= \exists A.\, \Sigma \\
\kappa &::= \mathrm{o} \mid \kappa \to \kappa & \Sigma &::= R \mid \forall A.\, \Sigma \to \xi \\
R &::= \left([x \rightharpoonup \tau], [t \rightharpoonup \llbracket = \tau :: \kappa \rrbracket], [m \rightharpoonup \Sigma], [s \rightharpoonup \llbracket = \xi \rrbracket]\right) & A &::= [\alpha \rightharpoonup \kappa] \\
\Gamma &::= (R, A)
\end{aligned}
$$

図 4　型や型環境の構文

必ず o がつくものだけに限定するとよいかと思います.

ストラクチャにつく “レコード型の” シグネチャ R は

- 値の識別子に型を紐づける R_{val}
- 型の識別子をエイリアス元となる実体の型項と種に写す R_{type}
- モジュールの識別子を（後述の）具体シグネチャに紐づける R_{mod}
- シグネチャの識別子をエイリアス元となる（後述の）抽象シグネチャに紐づける R_{sig}

の 4 つからなる $\left(R_{\mathrm{val}}, R_{\mathrm{type}}, R_{\mathrm{mod}}, R_{\mathrm{sig}}\right)$ という部分写像の組ですが，各識別子 x, t, m, s が動く可算集合がどの 2 つも排反であるという自然な前提の下では $\left(R_{\mathrm{val}} \uplus R_{\mathrm{type}} \uplus R_{\mathrm{mod}} \uplus R_{\mathrm{sig}}\right)$ という 1 つの有限部分写像と明らかに同等（つまり相互に 1 対 1 対応して変換可能）なので，適宜後者のように 1 つの有限部分写像として扱うことにします．同様に，型環境 Γ はこの “レコード” の構造に型変数の部分写像も加えた $\left(\left(\Gamma_{\mathrm{val}}, \Gamma_{\mathrm{type}}, \Gamma_{\mathrm{mod}}, \Gamma_{\mathrm{sig}}\right), \Gamma_{\mathrm{tyvar}}\right)$ という組ですが，やはりこれも全て結合した 1 つの有限部分写像として扱うことにします．型環境とは呼んでいるものの，直観的には “通常の型環境に相当する $\left([x \rightharpoonup \tau], [t \rightharpoonup \kappa], [\alpha \rightharpoonup \kappa], [m \rightharpoonup \Sigma]\right)$ と，解釈 (interpretation) 或いは代入に相当する $\left([t \rightharpoonup \tau], [s \rightharpoonup \xi]\right)$ とが融合したもの” と捉えることもできるかもしれません[*16].

Σ と ξ は**意味論的シグネチャ (semantic signature)** と呼ばれ，“シグネチャの内部表現” に相当するものです[*17]. Σ は**具体シグネチャ (concrete signature)** というもので，ストラクチャにつくシグネチャは前述の通り Γ という型環境と同一の形で，ファンクタにつくシグネチャは $\forall A.\, \Sigma \to \xi$ という形で表します．一方 ξ は**抽象シグネチャ (abstract signature)** というもの

16 解釈に相当する部分も，t に $\llbracket = \tau \rrbracket$, s に $\llbracket = \xi \rrbracket$ という singleton kind がそれぞれついていると捉えれば，型環境であると直観的に理解することも可能ではあります.

17 「意味論的シグネチャ」という名称はメタ変数 S で表されるシグネチャが “実際に書かれる構文的な対象” であることとの対比によるものだろうと思います.

で，これは具体シグネチャに対して型変数を存在量化したものです．定番の *Types and Programming Languages* [9] およびその訳書『型システム入門』[10] の 24 章「存在型」にも記載がありますが，モジュールシステムに於いて型変数の存在量化はデータ抽象，すなわち型の実装の隠蔽に相当する機構を成します．

4.2. 型システム

それでは実際に型システムを見てみます．図 5，図 6，図 7，図 8 に掲げる通り相互に依存した複数種類の型つけ規則からなっています．図 5 がモジュールに対する型つけ[*18]，図 6 がシグネチャに対する **elaboration**[*19]，図 7 がコア言語に対する型つけの追加規則，図 8 が意味論的シグネチャ上の部分型つけの規則であり，以下で順を追って見ていきます．

記法

“ストラクチャに対応するシグネチャ R 中から型を取り出す処理を型のパス t' へと拡張した処理” である $\mathrm{Access}(R, t')$ を以下のように定めます：

$$
\begin{aligned}
\mathrm{Access}(R, t) &\coloneqq R(t) \\
\mathrm{Access}(R, m.t') &\coloneqq \mathrm{Access}(R_0, t') && \text{if } R(m) = R_0 \\
\mathrm{Access}(R, m.t') &\coloneqq \mathrm{undef} && \text{if } R(m) = \forall A.\, \Sigma \to \xi
\end{aligned}
$$

“値のパスに対する取り出し” $\mathrm{Access}(R, x')$ についても同様に定めるとします．

部分写像の結合 $f_1 + f_2$ は以下で定義されます：

$$
(f_1 + f_2)(a) \coloneqq \begin{cases} f_2(a) & \text{if } a \in \mathrm{dom}\, f_2 \\ f_1(a) & \text{if } a \in \mathrm{dom}\, f_1 \setminus \mathrm{dom}\, f_2 \end{cases}
$$

すなわち “重複した場合は後者を優先する結合” です．

---

18 元論文 [11, 4] では図 5 の $\Gamma \vdash \boldsymbol{B} : \xi$ と $\Gamma \vdash B : \xi$ に相当するものが単一の型判定で扱われていますが，ここではより実装上便利な形として 2 つに分けて記述しています．図 6 の $\Gamma \vdash \boldsymbol{D} \hookrightarrow \xi$ と $\Gamma \vdash D \hookrightarrow \xi$ についても同様．

19 日本語に訳されることがほぼない用語ですが，おおよそ「綿密な仕上げ」を意味し，“入力を構造にしたがって再帰的に辿ってより洗練された形にして出力する規則” を指して使われます．

モジュールの型つけ

図 5 に示すように，モジュールに対する型つけは $\Gamma \vdash M : \xi$ という形の型判定，すなわち「型環境 Γ の下でモジュール M に抽象シグネチャ ξ がつく」という命題に関する導出規則で定式化されています．ファンクタの抽象と適用に対する規則 (M-Fun)，(M-App)，およびシグネチャによる**強制 (coercion)** の規則 (M-Coerce) が特に本質的に重要な規則ですが，これらは（各規則の上段にあるように）シグネチャの elaboration や意味論的シグネチャ上の部分型つけに依存するのでそれらが登場してから触れることにし，まずはそれ以外の (M-Binds)，(M-Ident)，(M-Proj) について見ていきます．

(M-Binds) は通常のストラクチャに対する型つけで，これは束縛の列に対する型つけ $\Gamma \vdash \boldsymbol{B} : \xi$ にほぼそのままフォールバックするものです．$\Gamma \vdash \boldsymbol{B} : \xi$ は単に各要素となる束縛 B に対して $\Gamma \vdash B : \xi$ で型つけし，それらを (Bs-Cons) にあるように適切に結合してひとつの抽象シグネチャにします．ただし，型つけ規則中の ε は空列を，$Z \cdot \boldsymbol{Z}$ は先頭 Z と後続 $\boldsymbol{Z}$ の連接（いわゆる cons）をそれぞれ表す記法とします．ここで部分写像の結合 $R_1 + R_2$ を用いているのは，ML 系のモジュールの定式化に於いて “同名の函数がモジュール中で複数ある場合は後方のものほど優先されて残る” ことに対応した定義になっています．同一モジュール中で同名の函数の定義を認めないようなより厳しい定式化にしたい場合は規則中の下段の $R_1 + R_2$ を $R_1 \uplus R_2$ にするとよいですが，同名の函数を定義して古い函数定義を覆い隠すことは特に **include** を用いる際に便利だったりすることもあって，後者優先の $R_1 + R_2$ の形で紹介しています．また，規則 (Bs-Cons) 中では型変数に関して “既に型環境 Γ が持っている型変数が同名で存在量化されているなら結合してはいけない” という暗黙の side condition がありますが，これは暗に α 変換することによって充足されるものです．ただし，型検査器の実装上は “シグネチャを走査している際に遭遇した，抽象化された型の宣言に対応する型変数を導入する際に `gensym` のような機構でフレッシュな ID を取る” ことで自ずと達成されるため，それほど気にかけることはありません．なお，実際に型変数が導入される箇所は図 5 のモジュールに対する型つけの範囲にはなく，後出の図 6 に掲げるシグネチャの elaboration の規則にあります．

個々の束縛に関する規則を見ていくと，まず (B-Val) と (B-Type) は単純で，これはコア言語の型つけ $\Gamma \vdash E : \tau$ および種づけ $\Gamma \vdash T :: \kappa \hookrightarrow \tau$ に基づいてそれぞれ値と型の単一の束縛に対応するシグネチャを返すものです（コア言語の規則は図 7 とともに後述します）．(B-Sig) も（後述するシグネチャの elaboration に依存するものの）やっていることは単純で，単にシグネチャの識別子を elaboration で得られた抽象シグネチャに紐づけるシグネチャを返すだけです．こ

$\boxed{\Gamma \vdash M : \xi}$

$$\frac{\Gamma \vdash \boldsymbol{B} : \xi}{\Gamma \vdash \{\boldsymbol{B}\} : \xi}\ \text{(M-Binds)} \qquad \frac{\Gamma(m) = \Sigma}{\Gamma \vdash m : \exists\varnothing.\, \Sigma}\ \text{(M-Ident)} \qquad \frac{\Gamma \vdash M : \exists A.\, R \qquad R(m) = \Sigma}{\Gamma \vdash M.m : \exists A.\, \Sigma}\ \text{(M-Proj)}$$

$$\frac{\begin{array}{c}\Gamma \vdash S_1 \hookrightarrow \exists A_1.\, \Sigma_1 \qquad \mathrm{dom}\Gamma \cap \mathrm{dom}\, A_1 = \varnothing \\ \big(\Gamma \uplus A_1\big) + \{m \mapsto \Sigma_1\} \vdash M_2 : \exists A_2.\, \Sigma_2 \qquad \mathrm{dom}\, A_1 \cap \mathrm{dom}\, A_2 = \varnothing\end{array}}{\Gamma \vdash \mathbf{fun}\big(m : S_1\big) \to M_2 : \exists\varnothing.\, \big(\forall A_1.\, \Sigma_1 \to \exists A_2.\, \Sigma_2\big)}\ \text{(M-Fun)}$$

$$\frac{\begin{array}{c}\mathrm{Access}\big(\Gamma, m'_1\big) = \forall A_1.\, \Sigma_1 \to \xi_1 \qquad \mathrm{dom}\Gamma \cap \mathrm{dom}\, A_1 = \varnothing \\ \mathrm{Access}\big(\Gamma, m'_2\big) = \Sigma_2 \qquad \Gamma \vdash \Sigma_2 \leqslant \exists A_1.\, \Sigma_1 \uparrow \theta\end{array}}{\Gamma \vdash m'_1\big(m'_2\big) : \theta\xi_1}\ \text{(M-App)}$$

$$\frac{\Gamma(m) = \Sigma \qquad \Gamma \vdash S \hookrightarrow \xi \qquad \Gamma \vdash \Sigma \leqslant \xi \uparrow \theta}{\Gamma \vdash (m :> S) : \xi}\ \text{(M-Coerce)}$$

$\boxed{\Gamma \vdash \boldsymbol{B} : \exists A.\, R}$

$$\frac{}{\Gamma \vdash \varepsilon : \exists\varnothing.\, \varnothing}\ \text{(Bs-Nil)} \qquad \frac{\begin{array}{c}\Gamma \vdash B_1 : \exists A_1.\, R_1 \qquad \mathrm{dom}\Gamma \cap \mathrm{dom}\, A_1 = \varnothing \\ \big(\Gamma \uplus A_1\big) + R_1 \vdash \boldsymbol{B}_2 : \exists A_2.\, R_2 \qquad \mathrm{dom}\, A_1 \cap \mathrm{dom}\, A_2 = \varnothing\end{array}}{\Gamma \vdash B_1 \cdot \boldsymbol{B}_2 : \exists\big(A_1 \uplus A_2\big).\, \big(R_1 + R_2\big)}\ \text{(Bs-Cons)}$$

$\boxed{\Gamma \vdash B : \exists A.\, R}$

$$\frac{\Gamma \vdash E : \tau}{\Gamma \vdash \big(\mathbf{val}\, x = E\big) : \exists\varnothing.\, \{x \mapsto \tau\}}\ \text{(B-Val)} \qquad \frac{\Gamma \vdash T :: \kappa \hookrightarrow \tau}{\Gamma \vdash \big(\mathbf{type}\, t = T\big) : \exists\varnothing.\, \big\{t \mapsto \lfloor = \tau :: \kappa \rfloor\big\}}\ \text{(B-Type)}$$

$$\frac{\Gamma \vdash M : \exists A.\, \Sigma}{\Gamma \vdash \big(\mathbf{module}\, m = M\big) : \exists A.\, \{m \mapsto \Sigma\}}\ \text{(B-Mod)}$$

$$\frac{\Gamma \vdash S \hookrightarrow \xi}{\Gamma \vdash \big(\mathbf{signature}\, s = S\big) : \exists\varnothing.\, \big\{s \mapsto \lfloor = \xi \rfloor\big\}}\ \text{(B-Sig)} \qquad \frac{\Gamma \vdash M : \exists A.\, R}{\Gamma \vdash \mathbf{include}\, M : \exists A.\, R}\ \text{(B-Include)}$$

図 5　モジュールに対する型つけ規則

れら 3 つはいずれも抽象型に対応する型変数を導入しない規則です．一方で入れ子のモジュールの束縛に対する型つけ規則である (B-Mod) は特徴的で，この規則をアルゴリズム的に読むならば “入れ子のモジュールのもつ抽象化された型に対応する型変数の存在量化 $\exists A.\, (-)$ が帰りがけに外に押し出されて伝搬” します．これと対応するように，ストラクチャにつくシグネチャ（＝型環境）の中ではモジュールの識別子に紐づけられるのは抽象シグネチャではなく具体シ

グネチャだったのです．裏を返せば“存在量化の束縛子は意味論的シグネチャ中で可能な限り外側に押し出されていくようになっており，一方で束縛子がそれ以上押し出されないように堰き止めねばならないのが (B-Sig) と後述の (M-Fun) の終域側” とも言えます．残る (B-Include) は **include** M の“M の定義を抽象化されたまま現在のストラクチャ直下にぶち撒ける” という直観をそのまま反映して“M につく抽象シグネチャを現在のストラクチャの一部分とするためにそのまま返す” という規則になっています．$\Gamma \vdash B : \xi$ を導出する規則のうち，この (B-Include) だけは“要素が 1 個とは限らない意味論的シグネチャ” を返します．

(M-Ident) についても，やはりモジュールの識別子に紐づけられているのは具体シグネチャであるという特筆すべき定式化が表れています．$(m \mapsto \Sigma) \in \Gamma$ の Σ には一般に抽象型に対応する型変数が自由出現し得ますが，“これに対応する束縛子は (B-Mod) と (Bs-Cons) によって外側に押し出されるか後述する (M-Fun) によって全称量化されるかしており，大域的に見て自由出現なわけではないことが保証されている” といえます．

シグネチャの elaboration

シグネチャの elaboration の規則は図 6 に示す通りです．これは $\Gamma \vdash S \hookrightarrow \xi$ という型判定，すなわち「型環境 Γ の下，（実際に書かれた）シグネチャ S は内部的表現である意味論的シグネチャ ξ へと洗練されて扱われる」という命題に対する導出規則で定義されています．陽にストラクチャに対応するシグネチャに対する規則 (S-Decls) は宣言の列に対する elaboration にフォールバックされており，宣言の列や宣言に対してもシグネチャに対するものと同様に $\Gamma \vdash \boldsymbol{D} \hookrightarrow \xi$ および $\Gamma \vdash D \hookrightarrow \xi$ という形で elaboration が定式化されています．

宣言の列に対する規則 (Ds-Nil), (Ds-Cons) はモジュールの束縛の列に対する規則と同様に“要素ごとに規則を適用して適切に畳み込む” ものになっています．ただし，こちらは side condition に $\mathrm{dom}\, R_1 \cap \mathrm{dom}\, R_2 = \varnothing$ が追加されていることに注意してください．これにより宣言列中で同名の識別子を多重に定義して後発のものによって手前で宣言されたものを隠すことはできないようになっています．

個々の宣言に対する規則を見ると，まず (D-Val) は素直な規則で，（実際に書かれた）型 T に対して内部的表現 τ への elaboration を施して識別子 x に紐づけます．モジュールの宣言に対する規則 (D-Mod) は束縛とよく似ており，やはり型変数を持ち上げて外に送り出します．シグネチャの宣言や埋め込みに対する規則 (D-Sig), (D-Include) もほぼ束縛の型つけ規則と対応しており，シグネチャの場合はやはり型変数を外へ送り出しません．重要なのは型の宣言に対する規則

$\boxed{\Gamma \vdash S \hookrightarrow \xi}$

$$\frac{\Gamma \vdash \boldsymbol{D} \hookrightarrow \xi}{\Gamma \vdash \{\boldsymbol{D}\} \hookrightarrow \xi}\ \text{(S-Decls)} \qquad \frac{\mathrm{Access}(\Gamma, s') = \lfloor = \xi \rfloor}{\Gamma \vdash s' \hookrightarrow \xi}\ \text{(S-Ident)}$$

$$\frac{\Gamma \vdash S_1 \hookrightarrow \exists A_1.\, \Sigma_1 \qquad \mathrm{dom}\Gamma \cap \mathrm{dom}\, A_1 = \varnothing \qquad (\Gamma \uplus A_1) + \{m \mapsto \Sigma_1\} \vdash S_2 \hookrightarrow \exists A_2.\, \Sigma_2 \qquad \mathrm{dom}\, A_1 \cap \mathrm{dom}\, A_2 = \varnothing}{\Gamma \vdash (m : S_1) \to S_2 \hookrightarrow \exists \varnothing.\, (\forall A_1.\, \Sigma_1 \to \exists A_2.\, \Sigma_2)}\ \text{(S-Fun)}$$

$$\frac{\Gamma \vdash S \hookrightarrow \exists A.\, R \qquad \mathrm{Access}(R, t') = \lfloor = \alpha :: \kappa \rfloor \qquad A(\alpha) = \kappa \qquad \Gamma \vdash T :: \kappa \hookrightarrow \tau}{\Gamma \vdash (S\ \textbf{with type}\ t' = T) \hookrightarrow \exists (A \setminus \{\alpha\}).\, [\tau/\alpha]R}\ \text{(S-With)}$$

$\boxed{\Gamma \vdash \boldsymbol{D} \hookrightarrow \exists A.\, R}$

$$\frac{}{\Gamma \vdash \varepsilon \hookrightarrow \exists \varnothing.\, \varnothing}\ \text{(Ds-Nil)} \qquad \frac{\Gamma \vdash D_1 \hookrightarrow \exists A_1.\, R_1 \qquad \mathrm{dom}\Gamma \cap \mathrm{dom}\, A_1 = \varnothing \qquad (\Gamma \uplus A_1) + \Gamma_1 \vdash \boldsymbol{D}_2 \hookrightarrow \exists A_2.\, R_2 \qquad \mathrm{dom}\, A_1 \cap \mathrm{dom}\, A_2 = \varnothing \qquad \mathrm{dom}\, R_1 \cap \mathrm{dom}\, R_2 = \varnothing}{\Gamma \vdash D_1 \cdot \boldsymbol{D}_2 \hookrightarrow \exists (A_1 \uplus A_2).\, (R_1 \uplus R_2)}\ \text{(Ds-Cons)}$$

$\boxed{\Gamma \vdash D \hookrightarrow \exists A.\, R}$

$$\frac{\Gamma \vdash T :: \mathrm{o} \hookrightarrow \tau}{\Gamma \vdash (\textbf{val}\ x : T) \hookrightarrow \exists \varnothing.\, \{x \mapsto \tau\}}\ \text{(D-Val)}$$

$$\frac{\Gamma \vdash K \hookrightarrow \kappa \qquad \alpha \notin \mathrm{dom}\Gamma}{\Gamma \vdash (\textbf{type}\ t :: K) \hookrightarrow \exists \{\alpha \mapsto \kappa\}.\, \{t \mapsto \lfloor = \alpha :: \kappa \rfloor\}}\ \text{(D-Type)}$$

$$\frac{\Gamma \vdash S \hookrightarrow \exists A.\, \Sigma}{\Gamma \vdash (\textbf{module}\ m : S) \hookrightarrow \exists A.\, \{m \mapsto \Sigma\}}\ \text{(D-Mod)}$$

$$\frac{\Gamma \vdash S \hookrightarrow \xi}{\Gamma \vdash (\textbf{signature}\ s = S) \hookrightarrow \exists \varnothing.\, \{s \mapsto \lfloor = \xi \rfloor\}}\ \text{(D-Sig)} \qquad \frac{\Gamma \vdash S \hookrightarrow \exists A.\, R}{\Gamma \vdash \textbf{include}\ S \hookrightarrow \exists A.\, R}\ \text{(D-Include)}$$

図 6　シグネチャの elaboration に関する型つけ規則

(D-Type) で，ここで抽象化された型に対応する型変数 α がフレッシュに取られています．逆にここ以外で存在量化される型変数が導入されることはありません．その意味では，この (D-Type) が型の抽象化の機構を最も直接的に反映した規則です．ただし，それはあくまで “数学的な規則上の話” であって，実装上は型変数をフレッシュに取らねばならないことに注意すべき箇所は他にもあります．これについてはファンクタに対する型つけ規則に触れる際に述べます．

シグネチャの elaboration に戻り，既に見た (S-Decls) と次々節で触れる (S-Fun) 以外の規則を見て elaboration について終えましょう．(S-Ident) は単に型環境から識別子に紐づけられた実態の抽象シグネチャを取り出すだけです．(S-With) は S **with type** $t' = T$ という構文の役割そのものが表れている規則で，S 中で定義されている抽象化された型への “パス” t' に対し，これを型 T に固定し抽象化されていない型へと具体化する機能を反映しています．

elaboration の具体例

elaboration の簡単な例を見てみましょう（[11] に載っている例を少し変えたものです）．以下の（構文的に書かれた）シグネチャからは，

$$\left\{\textbf{module } \mathsf{A} : \left\{\textbf{type } \mathsf{t} :: \mathsf{o}; \textbf{ val } \mathsf{x} : \mathsf{t}\right\}; \textbf{ signature } \mathsf{S} = \left\{\textbf{type } \mathsf{u} :: \mathsf{o}; \textbf{ val } \mathsf{f} : \mathsf{A.t} \to \mathsf{u}\right\}\right\}$$

elaboration により以下のような抽象シグネチャが得られます：

$$\exists\{\alpha \mapsto \mathsf{o}\}.\left\{\mathsf{A} \mapsto \left\{\mathsf{t} \mapsto \left\lfloor = \alpha :: \mathsf{o}\right\rfloor, \mathsf{x} \mapsto \alpha\right\}, \mathsf{S} \mapsto \left\lfloor = \exists\{\beta \mapsto \mathsf{o}\}.\left\{\mathsf{u} \mapsto \left\lfloor = \beta :: \mathsf{o}\right\rfloor, \mathsf{f} \mapsto (\alpha \to \beta)\right\}\right\rfloor\right\}$$

(D-Mod) の “存在量化の束縛をできるだけ外に押し出す” ような規則により，α の存在量化は（A のマップ先だけをスコープにしたりせず）モジュール全体を囲うまでに押し出されています．一方で (D-Sig) は抽象シグネチャをそのままシグネチャの識別子に紐づけるので “存在量化を外へ押し出す” ようなことはせず，したがって β は S のマップ先だけをスコープとします．

本稿では詳しく触れませんが，elaboration により上記の例でいう α の束縛が “外へ押し出される” ことで A.t の形で表れていた S から A への “依存関係” が或る種 “除去される” ことが F-ing modules に於いて依存型を用いずに定式化できることに本質的に効いているようです．

ファンクタ抽象とファンクタのシグネチャ

シグネチャの elaboration を (S-Fun) 以外把握したところで，後回しにしていたファンクタ抽象の型つけ規則 (M-Fun) と (S-Fun) を見てみます．それぞれ再掲します：

$$\frac{\Gamma \vdash S_1 \hookrightarrow \exists A_1.\, \Sigma_1 \qquad \mathrm{dom}\,\Gamma \cap \mathrm{dom}\,A_1 = \varnothing \qquad \left(\Gamma \uplus A_1\right) + \left\{m \mapsto \Sigma_1\right\} \vdash M_2 : \exists A_2.\, \Sigma_2 \qquad \mathrm{dom}\,A_1 \cap \mathrm{dom}\,A_2 = \varnothing}{\Gamma \vdash \textbf{fun}\left(m : S_1\right) \to M_2 : \exists\varnothing.\, \left(\forall A_1.\, \Sigma_1 \to \exists A_2.\, \Sigma_2\right)}\ \text{(M-Fun)}$$

$$\frac{\Gamma \vdash S_1 \hookrightarrow \exists A_1.\, \Sigma_1 \qquad \mathrm{dom}\,\Gamma \cap \mathrm{dom}\, A_1 = \varnothing \qquad \left(\Gamma \uplus A_1\right) + \left\{m \mapsto \Sigma_1\right\} \vdash S_2 \hookrightarrow \exists A_2.\, \Sigma_2 \qquad \mathrm{dom}\, A_1 \cap \mathrm{dom}\, A_2 = \varnothing}{\Gamma \vdash \left(m : S_1\right) \to S_2 \hookrightarrow \exists \varnothing.\, \left(\forall A_1.\, \Sigma_1 \to \exists A_2.\, \Sigma_2\right)}\ \text{(S-Fun)}$$

これらの規則はファンクタのシグネチャ $\forall A.\, \Sigma \to \xi$ が直観的に表すものを把握すると読みやすくなります．すなわち，このシグネチャは「A に含まれる各型変数に該当するところはどんな型が来てもよいから，とにかく Σ の形に適合するモジュールを受け取る．そうしたら，そのときに各型変数に対応している型で ξ 中の型変数の箇所をそれぞれ置き換えたようなシグネチャのつくモジュールを返す」という具合のファンクタにつけるものと理解するとわかりやすいです．実際 (M-Fun) はこの直観とおおよそ対応しており，モジュールの識別子に紐づけられるのが具体シグネチャであることに注意する以外自然に読めると思います．(S-Fun) によって elaboration でこのファンクタの形のシグネチャを得る規則も，型変数がダブらないようにする side condition があることを除いて素直な規則に見えてくるはずです．

ただし，ここでの side condition $\mathrm{dom}\,\Gamma \cap \mathrm{dom}\, A_1 = \varnothing$ は型検査器の実装に際して（単に **type** の宣言ごとにフレッシュに型変数をとるだけでは）自然には達成されないものであることに注意しなければなりません．すなわち，“$\Gamma \vdash S_1 \hookrightarrow \exists A_1.\, \Sigma_1$ で取り出された A_1 を適切に α 変換してから型環境に追加” しなければ正確な実装になりません．この α 変換に際しては，A_1 の各型変数ごとにそれを置き換えるフレッシュな型変数をとる実装にすれば十分です．

こうした α 変換相当の処理を行なわない実装を (M-Fun) として与えてしまうと，例えば以下のようなまずいプログラムに型がつく型検査器の実装になってしまいます：

$$\Big\{\textbf{signature}\ \mathsf{S} = \big\{\textbf{type}\ \mathsf{t} :: \mathsf{o}\big\};\ \textbf{module}\ \mathsf{F} = \textbf{fun}(\mathsf{M1} : \mathsf{S}) \to \textbf{fun}(\mathsf{M2} : \mathsf{S}) \to \big(\big\{\textbf{val}\ \mathsf{f} = \lambda \mathsf{x}.\, \mathsf{x}\big\} :> \big\{\textbf{val}\ \mathsf{f} : \mathsf{M1.t} \to \mathsf{M2.t}\big\}\big)\Big\}$$

要するに $\lambda \mathsf{x}.\, \mathsf{x}$ が任意の型から任意の型への変換を担えてしまうことになり，明らかに型検査器の実装が健全性を満たしていません．これは，このプログラムの :> の強制が型検査に通ってしまうのが要因です．強制の型つけ規則の詳細については後で触れますが，ここでは単に f につくそれぞれの型に関するコア言語での部分型つけのみを考えればよいです．簡単に言えば，毎回フレッシュに型変数を取り替えずそのまま抽象シグネチャを返す間違った (M-Fun) の実装と正常な (S-Ident) が組み合わさることで，抽象化された 2 つの型 M1.t, M2.t が（実際には全く無関係な実体をもつ型がファンクタ適用によって渡されうるにもかかわらず）同じ型変数 α で表されてしまうために，$\lambda \mathsf{x}.\, \mathsf{x}$ に自然につく型 $\forall \gamma :: \mathsf{o}.\, \gamma \to \gamma$ よりも（M1.t → M2.t の内部表現であ

る）型 α → α の方が一般性が高い，すなわち ∀γ :: o. γ → γ ⩽ α → α である[*20]とコア言語の部分型つけに基づいて判定されてしまうからです．このように，(M-Fun) は目立たないながら実装にあたって少々注意を要する規則です．

或いは「$\Gamma \vdash M : \exists A.\Sigma$ という型判定に相当する処理で，入力 Γ と M を与えた戻り値として $\exists A.\Sigma$ が返ってきた時点で必ず $\mathrm{dom}\,\Gamma \cap \mathrm{dom}\,A = \varnothing$ が成り立つような実装にする」という方針で実装することも可能です．それは (S-Ident) に相当する処理に α 変換にあたる型変数の取り替えを加えるという方法です．要するに，シグネチャの識別子 s' の出現を検査するごとに，型環境から取り出した抽象シグネチャ $\exists A.\Sigma$ に対し，A に含まれる各型変数に対応する新しい型変数をフレッシュに取って陽に操作的に α 変換するのです．いわば，let 多相でいうところの，変数の出現を見るごとにそれにつく型を多相型から型スキームへと instantiate して得る処理をするのとよく似ています．こうすれば (M-Fun) で取り替えずともよくなります[*21]．

いずれとしても，“陽に束縛を外す” 規則である (M-Fun) には理論だけでなく実装に際しても注意が必要です．

コア言語に対する追加規則

コア言語に対する型つけ規則と elaboration 規則の拡張は図 7 に示す通りです．モジュール言語の存在がコア言語に介入するのは，基本的にはモジュールからの型や値の取り出しだけです[*22]．これ以外の規則ならびに構文そのものについては，モジュール言語全体としての健全性を支えるための “一定の整合性” を満たしていれば自由に設定することができます．

ファンクタ適用と強制の部分型つけによる型検査

さて，ファンクタと強制に対する型つけ (M-Fun)，(M-Coerce) を担う根幹である，いわゆる signature matching の機構について見ていきます．これは意味論的シグネチャ上の部分型つ

---

20 一瞬違和感があるかもしれませんが，コア言語の部分型つけでは “より多相な型の方が低い側” です．これは次のような直観的理解が可能です： 型 ∀γ :: o. γ → γ がつけられる変数に対しては，型 int → int もつけられる．したがって int → int は “集合的に捉えて ∀γ :: o. γ → γ を包む” 型である．

21 これについては，もともと著者が Sesterl [12] という自作言語の処理系の実装に F-ing modules を組み込んだ際に `@elpin1al` さんが実装を読んで不具合を指摘して頂いた [2] ことで把握しました．ご指摘感謝致します．

22 F-ing modules の元論文 [11] では $M.x$ といったもっと一般性の高い形を含む取り出し方が扱えるようになっていますが，ここではモジュールを識別子からの射影の形に限定しました．これは主に後述の static interpretation の定式化が簡潔に済むことによるもので，実際 static interpretation を提案する [4] でもこのような定式化になっています．

$\boxed{\Gamma \vdash E : \tau}$ $\boxed{\Gamma \vdash T :: \kappa \hookrightarrow \tau}$ $\boxed{\Gamma \vdash K \hookrightarrow \kappa}$

$$\frac{\mathrm{Access}(\Gamma, x') = \tau}{\Gamma \vdash x' : \tau} \qquad \frac{\mathrm{Access}(\Gamma, t') = \left[= \tau :: \kappa\right]}{\Gamma \vdash t' :: \kappa \hookrightarrow \tau} \qquad \frac{}{\Gamma \vdash \mathrm{o} \hookrightarrow \mathrm{o}}$$

図 7　コア言語に対する追加規則（その他の形の式については通常どおり適切に定める）

$\boxed{\Gamma \vdash \Sigma \leqslant \xi \uparrow \theta}$

$$\frac{\mathrm{dom}\,\theta = \mathrm{dom}\,A_2 \qquad \forall \alpha \in \mathrm{dom}\,\theta.\ \left(\Gamma \vdash \theta(\alpha) :: A_2(\alpha)\right) \qquad \Gamma \vdash_{\mathrm{conc}} \Sigma_1 \leqslant \theta\Sigma_2}{\Gamma \vdash \Sigma_1 \leqslant \exists A_2.\, \Sigma_2 \uparrow \theta} \text{(Sub-CA)}$$

$\boxed{\Gamma \vdash_{\mathrm{conc}} \Sigma_1 \leqslant \Sigma_2}$

$$\frac{\begin{array}{c}\mathrm{dom}\,R_1 \supseteq \mathrm{dom}\,R_2 \qquad \forall x \in \mathrm{dom}\,R_2 \cap \mathrm{ValIdent}.\ \left(\Gamma \vdash_{\mathrm{type}} R_1(x) \leqslant R_2(x)\right) \\ \forall t \in \mathrm{dom}\,R_2 \cap \mathrm{TypeIdent}.\ \left(\Gamma \vdash_{\mathrm{type}} \tau_1 = \tau_2 \wedge \kappa_1 = \kappa_2 \quad \text{where } R_i(t) = \left[= \tau_i :: \kappa_i\right]\right) \\ \forall m \in \mathrm{dom}\,R_2 \cap \mathrm{ModIdent}.\ \left(\Gamma \vdash_{\mathrm{conc}} R_1(m) \leqslant R_2(m)\right) \\ \forall s \in \mathrm{dom}\,R_2 \cap \mathrm{SigIdent}.\ \left(\Gamma \vdash_{\mathrm{abs}} \xi_1 \leqslant \xi_2 \wedge \Gamma \vdash_{\mathrm{abs}} \xi_2 \leqslant \xi_1 \quad \text{where } R_i(s) = \left[= \xi_i\right]\right)\end{array}}{\Gamma \vdash_{\mathrm{conc}} R_1 \leqslant R_2} \text{(Sub-CC-Struct)}$$

$$\frac{\mathrm{dom}\,\Gamma \cap \mathrm{dom}\,A_2 = \varnothing \qquad \Gamma \uplus A_2 \vdash \Sigma_2 \leqslant \exists A_1.\, \Sigma_1 \uparrow \theta \qquad \Gamma \uplus A_2 \vdash_{\mathrm{abs}} \theta\xi_1 \leqslant \xi_2}{\Gamma \vdash_{\mathrm{conc}} \forall A_1.\, \Sigma_1 \to \xi_1 \leqslant \forall A_2.\, \Sigma_2 \to \xi_2} \text{(Sub-CC-Fun)}$$

$\boxed{\Gamma \vdash_{\mathrm{abs}} \xi_1 \leqslant \xi_2}$

$$\frac{\mathrm{dom}\,\Gamma \cap \mathrm{dom}\,A_1 = \varnothing \qquad \Gamma \uplus A_1 \vdash \Sigma_1 \leqslant \xi_2 \uparrow \theta}{\Gamma \vdash_{\mathrm{abs}} \exists A_1.\, \Sigma_1 \leqslant \xi_2} \text{(Sub-AA)}$$

図 8　意味論的シグネチャ上の部分型つけ

けの形で定式化されており，図 8 に示す規則で表されます．型つけ規則から使われているのは $\Gamma \vdash \Sigma \leqslant \xi \uparrow \theta$ のみで，これが具体シグネチャ間の部分型つけ $\Gamma \vdash_{\mathrm{conc}} \Sigma_1 \leqslant \Sigma_2$ と抽象シグネチャ間の部分型つけ $\Gamma \vdash_{\mathrm{abs}} \xi_1 \leqslant \xi_2$ に依存している形です．また，型上の部分型つけ $\Gamma \vdash_{\mathrm{type}} \tau_1 \leqslant \tau_2$ および型の等価性判定 $\Gamma \vdash_{\mathrm{type}} \tau_1 = \tau_2$ はコア言語が提供する規則によります．このほか，$\Gamma \vdash \tau :: \kappa$ は System Fω の種づけ規則（のサブセット）を用います．

$\Gamma \vdash \Sigma_1 \leqslant \exists A_2.\, \Sigma_2 \uparrow \theta$ は「型環境 Γ の下で Σ_1 は（A_2 に含まれる各型変数を伴う）Σ_2 の

$\boxed{\mathrm{Lookup}_{(-)}(-,-)}$

$$\mathrm{Lookup}_{A_2}(\Sigma_1,\Sigma_2) \coloneqq \bigcup\left\{\mathrm{Lookup}_{\alpha}(\Sigma_1,\Sigma_2) \;\middle|\; \alpha \in \mathrm{dom}\, A_2\right\}$$

$$\mathrm{Lookup}_{\alpha}(R_1,R_2) \coloneqq \mathrm{Lookup}_{\alpha}\big(R_1(l),R_2(l)\big) \quad (\text{for some } l \in \mathrm{dom}\, R_1 \cap \mathrm{dom}\, R_2)$$

$$\mathrm{Lookup}_{\alpha}\big(\llbracket = \tau_1 :: \kappa_1 \rrbracket,\llbracket = \tau_2 :: \kappa_2 \rrbracket\big) \coloneqq \{\alpha \mapsto \tau_1\} \quad (\text{if } \kappa_1 = \kappa_2 \wedge \tau_2 = \alpha)$$

図 9　(Sub-CA) に相当するアルゴリズム

部分型となることができ，その際の各型変数に対する制約の解消結果は代入 θ である」ということを表す型判定です．ただし，規則 (Sub-CA) を読むとわかるように，この代入 θ は構文主導には決まりません（逆に，モジュール言語をなす他の規則は全て構文主導です）．したがってここまで紹介してきた型システムが即座に決定可能かはこの規則だけを原因として非自明ということになります[*23]が，実際には Σ_1 と Σ_2 を走査することでこの代入を決定することが可能です．F-ing modules の元論文 [11] にはこの代入を決定できるシグネチャの十分条件として explicit な意味論的シグネチャ，および valid な意味論的シグネチャの定義が与えられており，これに適合する全てのシグネチャに対して代入を決定できるアルゴリズムが具体的に構成されています．また，型環境中で紐づけられた具体シグネチャがすべて valid ならばその型環境を用いたモジュールに対する型つけの結果出てくるシグネチャは全て valid であること，同様にシグネチャに対する elaboration の結果出てくるシグネチャは全て explicit であることが示されています．正当性については本稿の範疇ではないので元論文 [11] を参照してもらうこととし，Γ, Σ_1, $\exists A_2.\,\Sigma_2$ を入力として $\Gamma \vdash \Sigma_1 \leqslant \exists A_2.\,\Sigma_2 \uparrow \theta$ なる代入 θ を決定するアルゴリズム $\mathrm{Lookup}_{A_2}(\Sigma_1,\Sigma_2)$ だけ図 9 に掲げておきます．

最も signature matching らしさが表れている規則は (Sub-CC-Struct) でしょう．これは 2 つのストラクチャに対応するシグネチャの各フィールドについて “さらなる入れ子構造が部分型つけを満たすことを再帰的に判定する” 規則です．値とモジュールの識別子については共変で再帰しますが，型とシグネチャについては等価性を判定することに注意してください[*24]．

---

23 種が一般の高階のものをとる以上，既に決定不能であることが知られている高階単一化 [6] に帰着できてしまいそうに見える程度には複雑なので，決定可能性は実際かなり非自明です．

24 これは F-ing modules に限らずモジュールシステム一般の言語設計上言えますが，もし型とシグネチャの識別子の定義間に共変な部分型つけしか要請しなかった場合，反変の向きの部分型つけを満たさずに定義された型の識別子が同じシグネチャの別の値につけられる型の中で反変な位置に出現する例が明らかに健全性を壊します．

(Sub-CC-Fun) は通常の函数型の部分型判定のように，始域側については反変，終域側については共変な判定をします．ただし，型変数がある分少し複雑になっており，まず始域側で “始域側が反変になるとしたら A_1 に含まれる型変数はどのように具体化されねばならないか” の制約を解消して代入 θ を得て，それを終域側の共変性の判定に用いています．

さて，いよいよ後回しにしていたファンクタ適用と強制に関するモジュール式の型つけ規則 (M-App), (M-Coerce) を読む準備ができました．図 5 まで戻るのも遠いので以下に再掲します：

$$\frac{\mathrm{Access}(\Gamma, m'_1) = \forall A_1.\, \Sigma_1 \to \xi_1 \qquad \mathrm{dom}\Gamma \cap \mathrm{dom}\, A_1 = \varnothing \qquad \mathrm{Access}(\Gamma, m'_2) = \Sigma_2 \qquad \Gamma \vdash \Sigma_2 \leqslant \exists A_1.\, \Sigma_1 \uparrow \theta}{\Gamma \vdash m'_1(m'_2) : \theta\xi_1}\ \text{(M-App)}$$

$$\frac{\Gamma(m) = \Sigma \qquad \Gamma \vdash S \hookrightarrow \xi \qquad \Gamma \vdash \Sigma \leqslant \xi \uparrow \theta}{\Gamma \vdash (m :> S) : \xi}\ \text{(M-Coerce)}$$

まず (M-Coerce) は簡単です． これは単にモジュールの識別子から得られた具体シグネチャ Σ がシグネチャの註釈から elaboration で得られた抽象シグネチャ ξ に対して実際に部分型になっているかを判定し，なっているならば（代入は特に使わずに捨てて）ξ を強制した結果として返しているだけです．こうしてモジュール m が “実装によらず外延的に見て S の内容のみを API としてもつように扱える” ことが保証でき，そのように扱われるようになるわけです．

ファンクタの適用の規則 (M-App) も，$\forall A.\, \Sigma_1 \to \xi_1$ がもつ直観的な意味に思い起こせば特に複雑な規則には見えないはずです． すなわち， ここで引数となるモジュールのシグネチャ Σ_1 が「A に含まれる各型変数に該当するところはどんな型が来てもよいからとりあえず Σ_2 の形に適合するか」を部分型つけによって判定しているのです．そして，その適合の結果得られた各型変数に型を紐づける代入 θ をファンクタの終域の抽象シグネチャに対して適用し，“より引数に合わせて具体化された” 抽象シグネチャを結果として返しているわけです．このようにして，ファンクタの適用が妥当であるかの判定に部分型つけの判定が使われています．

ファンクタのシグネチャの具体例

2.2 節で扱ったマップにつく以下のシグネチャは，

$$
\begin{aligned}
&\left(\mathsf{Key} : \left\{\textbf{type}\ \mathsf{t} :: \mathrm{o};\ \textbf{val}\ \mathsf{equal} : \mathsf{t} \to \mathsf{t} \to \mathsf{bool}\right\}\right) \to \{ \\
&\quad \textbf{type}\ \mathsf{key} = \mathsf{Key.t}; \\
&\quad \textbf{type}\ \mathsf{t} :: \mathrm{o} \to \mathrm{o}; \\
&\quad \textbf{val}\ \mathsf{empty} : \forall \mathsf{'a} :: \mathrm{o}.\ \mathsf{t\ 'a}; \\
&\quad \textbf{val}\ \mathsf{add} : \forall \mathsf{'a} :: \mathrm{o}.\ \mathsf{key} \to \mathsf{'a} \to \mathsf{t\ 'a} \to \mathsf{t\ 'a};\ \ldots\}
\end{aligned}
$$

elaboration により以下のような意味論的シグネチャになります：

$$
\begin{aligned}
&\forall\{\beta \mapsto \mathrm{o}\}.\ \{\mathsf{t} \mapsto [= \beta :: \mathrm{o}], \mathsf{equal} \mapsto (\beta \to \beta \to \mathsf{bool})\} \to \\
&\quad \exists\{\gamma \mapsto (\mathrm{o} \to \mathrm{o})\}.\ \{ \\
&\qquad \mathsf{key} \mapsto [= \beta :: \mathrm{o}], \\
&\qquad \mathsf{t} \mapsto [= \gamma :: \mathrm{o} \to \mathrm{o}], \\
&\qquad \mathsf{empty} \mapsto \forall\alpha :: \mathrm{o}.\ \gamma\ \alpha, \\
&\qquad \mathsf{add} \mapsto \forall\alpha :: \mathrm{o}.\ \beta \to \alpha \to \gamma\ \alpha \to \gamma\ \alpha,\ \ldots\}
\end{aligned}
$$

型システムの妥当性

ここまでに眺めてきた F-ing modules の型システムですが，型システムの恩恵である「型のついたモジュールは評価中に或る種の失敗をしない」といった正当性はどのように示されているのでしょうか．本稿は主にモジュールシステムの実装面の観点で記述しているのであまり深くは正当性について立ち入りませんが，簡単に触れておきます．

型システムの正当性といえば，典型的には型つけの対象となるプログラムに対して評価規則の形で意味論が与えられ，それに対して型システムが保存定理や進行定理を満たすことを示す形で証明しますが，F-ing modules によるモジュール言語はそうではなく，それ自体には意味論は定められていません．かわりに，元論文 [11] ではモジュール言語の項から System Fω の項へのコンパイル方法が型つけ規則に備わっており，F-ing modules で型のつく項からコンパイルされた結果の System Fω の項に必ず System Fω で型がつくことを示すことで正当性を保証しています[*25]．System Fω 自体の型安全性はよく知られているので，「元々の体系で型がついていた項のコンパイル結果は評価時に失敗しない」ということが言えているわけです．この正

25 ソース言語には意味論を与えず，既に正当性が知られているターゲット言語へのコンパイルが型つけを保存することを証明して正当性を主張するこうした方法は，言うほど珍しいわけでもありません．他に例を挙げるなら，MacroML [5] は MetaML [13] へのコンパイルが型つけを保存することを通じて正当性が保証されています．

当性の保証に於いて，意味論的シグネチャ Σ, ξ や内部的表現の型 τ はそのままで以下のように System Fω の型へと自然に埋め込むことができ，その意味で F-ing modules の体系は「モジュールシステムは System Fω の特殊な利用形態のひとつとして定式化できる」という主張にもなっています：

$$\begin{aligned}
\left\lfloor \exists \left\{\alpha_i \mapsto \kappa_i\right\}_{i=1}^{n}. \Sigma \right\rfloor &\coloneqq \exists \alpha_1 :: \kappa_1. \cdots \exists \alpha_n :: \kappa_n. \lfloor \Sigma \rfloor \\
\left\lfloor \left\{l_i \mapsto U_i\right\}_{i=1}^{n} \right\rfloor &\coloneqq \left\{l_i : \lfloor U_i \rfloor\right\}_{i=1}^{n} \\
\lfloor \tau \rfloor &\coloneqq \{\mathsf{val} : \tau\} \\
\lfloor 【= \tau :: \kappa】 \rfloor &\coloneqq \{\mathsf{typ} : \forall \alpha :: (\kappa \to \mathrm{o}).\ \alpha\ \tau \to \alpha\ \tau\} \\
\lfloor 【= \xi】 \rfloor &\coloneqq \{\mathsf{sig} : \lfloor \xi \rfloor \to \lfloor \xi \rfloor\}
\end{aligned}$$

ただし val, typ, sig は相異なりかつどの識別子とも異なる特殊なラベルであり，また l と U の動く範囲は（Σ の形式から推測できるとは思いますが）以下の通りです：

$$\begin{aligned}
l &::= x \mid t \mid m \mid s \\
U &::= \tau \mid 【= \tau :: \kappa】 \mid \Sigma \mid 【= \xi】
\end{aligned}$$

こうした点から，F-ing modules は実用目的に端を発しつつ理論的にも整然とした体系に辿り着いているといえると思います．

5. コンパイル手法

ところで，ここまで F-ing modules の型システムを眺めてきましたが，この F-ing modules を言語処理系に搭載することを念頭に置いた紹介をしている以上，コンパイル方法について触れないわけにはいきません．そこで，正当性の証明は（実装上必要とならないので元論文に任せて）扱わないこととし，“型つけに必要な装飾” などを剥がした型なしの λ 項などをターゲット言語とするコンパイル方法をこの章で紹介します．

5.1. ナイーヴな方法

最もナイーヴな方法は，レコードをもつ型なし λ 項へと落とし込むものです．実際には型情報に基づいて型つきの中間表現に落とし込み，その後最適化したりより低級な表現へと更に落とし込んだりできるはずですが，ひとまず型なし λ 項への落とし込み方を掲げます．これは F-ing modules の元論文 [11] に掲載されていた System Fω の項へのコンパイル方法から存在量

$\boxed{\Gamma \vdash M : \xi \Rightarrow e}$

$$\frac{\Gamma \vdash \boldsymbol{B} : \xi \Rightarrow e}{\Gamma \vdash \{\boldsymbol{B}\} : \xi \Rightarrow e} \qquad \frac{\Gamma(m) = \Sigma}{\Gamma \vdash m : \exists \varnothing.\, \Sigma \Rightarrow m} \qquad \frac{\Gamma \vdash M : \exists A.\, R \Rightarrow e \qquad R(m) = \Sigma}{\Gamma \vdash M_0.m : \exists A.\, \Sigma \Rightarrow e.l_m}$$

$$\frac{\begin{array}{c}\Gamma \vdash S_1 \hookrightarrow \exists A_1.\, \Sigma_1 \qquad \operatorname{dom}\Gamma \cap \operatorname{dom} A_1 = \varnothing \\ \Gamma + \{m \mapsto \Sigma\} \vdash M_2 : \exists A_2.\, \Sigma_2 \Rightarrow e_2 \qquad \operatorname{dom} A_1 \cap \operatorname{dom} A_2 = \varnothing\end{array}}{\Gamma \vdash \mathbf{fun}\big(m : S_1\big) \to M_2 : \exists \varnothing.\, (\forall A.\, \Sigma \to \xi) \Rightarrow \lambda m.\, e_2}$$

$$\frac{\begin{array}{c}\operatorname{Access}\big(\Gamma, m'_1\big) = \forall A_1.\, \Sigma_1 \to \xi_1 \qquad \operatorname{dom}\Gamma \cap \operatorname{dom} A_1 = \varnothing \qquad \operatorname{Access}\big(\Gamma, m'_2\big) = \Sigma_2 \\ \Gamma \vdash \Sigma_2 \leqslant \exists A_1.\, \Sigma_1 \uparrow \theta \qquad m'_1 = m_{10}.m_{11}....m_{1a} \qquad m'_2 = m_{20}.m_{21}....m_{2b}\end{array}}{\Gamma \vdash m'_1\big(m'_2\big) : \theta\xi_1 \Rightarrow \Big(m_{10}.l_{m_{11}}....l_{m_{1a}}\Big)\Big(m_{20}.l_{m_{21}}....l_{m_{2b}}\Big)}$$

$$\frac{\Gamma(m) = \Sigma \qquad \Gamma \vdash S \hookrightarrow \xi \qquad \Gamma \vdash \Sigma \leqslant \xi \uparrow \theta}{\Gamma \vdash (m :> S) : \xi \Rightarrow m}$$

$\boxed{\Gamma \vdash \boldsymbol{B} : \exists A.\, R \Rightarrow e}$

$$\frac{\begin{array}{c}\Gamma \vdash B_1 : \exists A_1.\, R_1 \Rightarrow e_1 \qquad \operatorname{dom}\Gamma \cap \operatorname{dom} A_1 = \varnothing \\ \big(\Gamma \uplus A_1\big) + R_1 \vdash \boldsymbol{B}_2 : \exists A_2.\, R_2 \Rightarrow e_2 \qquad \operatorname{dom} A_1 \cap \operatorname{dom} A_2 = \varnothing\end{array}}{\Gamma \vdash B_1 \cdot \boldsymbol{B}_2 : \exists\big(A_1 \uplus A_2\big).\, \big(R_1 + R_2\big) \Rightarrow \begin{array}{l}\mathbf{let}\ X_1 = e_1\ \mathbf{in} \\ \quad \mathbf{let}\ X_2 = \\ \quad\ \mathbf{let}\ \big(x = X_1.l_x\big)_{x \in \operatorname{dom}\Gamma_1} \cdot \big(m = X_1.l_m\big)_{m \in \operatorname{dom}\Gamma_1}\ \mathbf{in}\ e_2 \\ \quad \mathbf{in} \\ \quad \Big\{\big(l_x = X_1.l_x\big)_{x \in \operatorname{dom}\Gamma_1 \setminus \operatorname{dom}\Gamma_2} \cdot \big(l_m = X_1.l_m\big)_{m \in \operatorname{dom}\Gamma_1 \setminus \operatorname{dom}\Gamma_2} \cdot \\ \quad \big(l_x = X_2.l_x\big)_{x \in \operatorname{dom}\Gamma_2} \cdot \big(l_m = X_2.l_m\big)_{m \in \operatorname{dom}\Gamma_2}\Big\}\end{array}}$$

$$\frac{}{\Gamma \vdash \varepsilon : \exists \varnothing.\, \varnothing \Rightarrow \{\}}$$

$\boxed{\Gamma \vdash B : \exists A.\, R \Rightarrow e}$

$$\frac{\Gamma \vdash E : \tau \Rightarrow e}{\Gamma \vdash \big(\mathbf{val}\ x = E\big) : \exists \varnothing.\, \{x \mapsto \tau\} \Rightarrow \big\{l_x = e\big\}}$$

$$\frac{\Gamma \vdash T :: \kappa \hookrightarrow \tau}{\Gamma \vdash \big(\mathbf{type}\ t = T\big) : \exists \varnothing.\, \big\{t \mapsto \llbracket = \tau :: \kappa \rrbracket\big\} \Rightarrow \{\}}$$

$$\frac{\Gamma \vdash M : \exists A.\, \Sigma \Rightarrow e}{\Gamma \vdash \big(\mathbf{module}\ m = M\big) : \exists A.\, \{m \mapsto \Sigma\} \Rightarrow \big\{l_m = e\big\}}$$

$$\frac{\Gamma \vdash S \hookrightarrow \xi}{\Gamma \vdash \big(\mathbf{signature}\ s = S\big) : \exists \varnothing.\, \big\{s \mapsto \llbracket = \xi \rrbracket\big\} \Rightarrow \{\}} \qquad \frac{\Gamma \vdash M : \exists A.\, R \Rightarrow e}{\Gamma \vdash \mathbf{include}\ M : \exists A.\, R \Rightarrow e}$$

図 10　モジュールに対する型つけ規則＋コンパイル方法

化など型つけのための機構を除去して “操作に関わる構文だけ丸裸にした” ものです．元論文で

は System Fω の項へのコンパイルが型つけを保存することを証明するために pack/unpack といった存在量化のための機構や部分型つけの witness となる函数を噛ませたりする変換規則になっていますが，これらは（型情報を大々的に使うのでない限り）操作的意味論の観点ではなくても困らないものなので，根幹の理解を促すためここでは省いた形で紹介します．

図 10 に型主導のコンパイル規則を示します．$\Gamma \vdash M : \xi \Rightarrow e$ という形で「型環境 Γ の下，モジュール M にはシグネチャ ξ がつき，かつ e というターゲット言語の項にコンパイルされる」という判定をもちますが，灰色のついている箇所を除けば図 5 の $\Gamma \vdash M : \xi$ と完全に一致します．ターゲット言語は基本的には型なし λ 項ですが，レコード $\{l_1 = e_1, \ldots, l_n = e_n\}$ とレコードからの射影 $e.l$ が備わっているとします．また，**let** $X = e_1$ **in** e_2 は単なる $(\lambda X.\, e_2)\, e_1$ の糖衣構文とします．値とモジュールの識別子 x, m はそのままターゲット言語の変数として使えるほか，レコードのラベル l_x, l_m にも埋め込めるとします．

ここに掲げたコンパイル規則はファンクタが函数抽象に，ストラクチャが基本的にはレコード式にコンパイルされる定式化で，直観的にも素直な規則として捉えられるのではないかと思います．この素朴な規則に従ってコンパイルする処理系を実装することで，ひとまず動くものは創れるわけです．

5.2. ファンクタをコンパイル時に除去する手法

前節でひとまず F-ing modules のプログラムは型なし λ 項にコンパイルできることを見たわけですが，勘の良い方なら「ファンクタを函数にコンパイルしているが，モジュールは “動的に値が決定する” ような概念ではないし，コンパイル時にファンクタの適用を部分評価的に除去してオーバーヘッドを減らせるのではないか？」という洞察を抱いたかもしれません．実際，このような動機によってファンクタを静的に除去する手法が提案されており，そのひとつが**静的解釈 (static interpretation) [4]** と呼ばれるものです． 以下では元論文の定式化を真に拡張して static interpretation を紹介するので[*26]正当性に自信がないのですが，ひとまず直観的説明には事足りるかと思うので，気休め程度にご笑覧ください[*27]．

---

26 元論文ではファンクタが “1 階のもの” に制限されているほか，型の識別子はパラメータを取れず種 o がつくもののみに限定されています．もっとも，通常の型とは違ってファンクタが “1 階” に限定されていることは事実上制約にはなりません．唯一の要素としてファンクタをもつようなストラクチャを受け取るファンクタによって実質的に “高階な” シグネチャを扱えるからです．

27 ここに掲げる定式化の正当性が証明できたらちょっとした論文にはなったりするかもしれません．

構文

本稿の形式化では，以下のような型なし λ 項の let 束縛がフラットに並んだ列をターゲット言語とします：

$$e ::= z \mid \lambda z.\, e \mid e\, e \qquad\qquad c ::= \mathbf{let}\, z = e \qquad\qquad \boldsymbol{c} ::= [c]^*$$

ただし z はターゲット言語の変数で，これは static interpretation に基づくコンパイルの最中にフレッシュに生成し出力に使われるものです．以降では z のことを**名前**と呼ぶことにします．型環境なども，元のモジュール中の **let** で束縛される変数に紐づく名前や “既に使用した” 名前を持ち回るために拡張します：

$$\begin{aligned}
\Psi &::= \exists \mathscr{A}.\, (\mathscr{M}, \boldsymbol{c}) \\
\mathscr{M} &::= \mathscr{R} \mid \left\langle \mathscr{G} \;\middle|\; \forall A.\, \Sigma \to \xi \;\middle|\; \lambda m.\, M \right\rangle \\
\mathscr{R} &::= \left([x \rightharpoonup \tau/z], [t \rightharpoonup \llbracket = \tau :: \kappa \rrbracket], [m \rightharpoonup \mathscr{M}], [s \rightharpoonup \llbracket = \Sigma \rrbracket]\right) \\
\mathscr{A} &::= (A, N) \\
\mathscr{G} &::= (\mathscr{R}, \mathscr{A})
\end{aligned}$$

ただし，N は名前の有限集合を動くものとします． いよいよ文字が多くなってきて大変ですが，根幹となる変更はストラクチャにつけるシグネチャ $\mathscr{R}$ で，変更前の R と比べて値の識別子に型だけでなく新たに名前も紐づけています． この $\mathscr{R}$ に加え，ファンクタにつけるシグネチャに相当する**ファンクタクロージャ (functor closure)** $\left\langle \mathscr{G} \;\middle|\; \forall A.\, \Sigma \to \xi \;\middle|\; \lambda m.\, M \right\rangle$ からなるのが拡張された具体シグネチャ $\mathscr{M}$ です． ファンクタクロージャは，中央に書かれた従来のファンクタのシグネチャに加え，“環境” である $\mathscr{G}$ と λ 抽象 $\lambda m.\, M$ を伴っているあたり， いかにもクロージャです． 型と同じ水準でクロージャという値の水準らしき構造が持ち回されているのはなんだか奇妙に見えますが， これこそ static interpretation の根幹をなす工夫であり，型検査時に “ファンクタをクロージャの形で型環境に保持して持ち回り，ファンクタ適用が来たらそこでクロージャを使って評価のようにモジュールを生成する” という或る種インタプリタに近いことをやります． それゆえに “static interpretation” という名前を冠しているといえます．$\mathscr{G}$ は先走って出てきましたが名前によって拡張された型環境で，従来 Γ だったものです．$\mathscr{A}$ は型変数だけでなく名前集合も保持するように拡張された束縛用の機構で，$\mathrm{dom}\, \mathscr{A}$ は $\mathrm{dom}(A, N) \coloneqq \mathrm{dom}\, A \uplus N$ で定義するとします．Ψ が拡張された抽象シグネチャであるとともに，生成されるコード $\boldsymbol{c}$ を戻り値として持ち回る機構です．Ψ はその束縛子 $\mathscr{A}$ に関して（型だけでなく名前についても）α 同値性で割られているとします．

$\boxed{\mathcal{G} \vdash M :\Rightarrow \Psi}$

$$\frac{\mathcal{G} \vdash \boldsymbol{B} :\Rightarrow \Psi}{\mathcal{G} \vdash \{\boldsymbol{B}\} :\Rightarrow \Psi} \qquad \frac{\mathcal{G}(m) = \mathscr{M}}{\mathcal{G} \vdash m :\Rightarrow \exists\varnothing.\,(\mathscr{M}, \varepsilon)} \qquad \frac{\mathcal{G} \vdash M :\Rightarrow \exists\mathscr{A}.\,(\mathscr{R}, \boldsymbol{c}) \qquad \mathscr{R}(m) = \mathscr{M}}{\mathcal{G} \vdash M.m :\Rightarrow \exists\mathscr{A}.\,(\mathscr{M}, \boldsymbol{c})}$$

$$\frac{\begin{array}{c}\overline{\mathcal{G}} \vdash S_1 \hookrightarrow \exists A_1.\,\Sigma_1 \qquad \operatorname{dom}\mathcal{G} \cap \operatorname{dom} A_1 = \varnothing \\ \left(\overline{\mathcal{G}} \uplus A_1\right) + \{m \mapsto \Sigma_1\} \vdash M_2 : \exists A_2.\,\Sigma_2 \qquad \operatorname{dom} A_1 \cap \operatorname{dom} A_2 = \varnothing\end{array}}{\mathcal{G} \vdash \mathbf{fun}(m : S_1) \to M_2 :\Rightarrow \exists\varnothing.\left(\left\langle \mathcal{G} \;\middle|\; \forall A_1.\,\Sigma_1 \to \exists A_2.\,\Sigma_2 \;\middle|\; \lambda m.\,M_2 \right\rangle, \varepsilon\right)}\ \text{(SI-M-Fun)}$$

$$\frac{\begin{array}{c}\operatorname{Access}(\mathcal{G}, m'_1) = \left\langle \mathcal{G}_1 \;\middle|\; \forall A_1.\,\Sigma_1 \to \xi_1 \;\middle|\; \lambda m.\,M \right\rangle \qquad \operatorname{dom}\mathcal{G} \cap \operatorname{dom} A_1 = \varnothing \\ \mathcal{G}(m'_2) = \mathscr{M}_2 \qquad \overline{\mathcal{G}} \vdash \mathscr{M}_2 \leqslant \exists A_1.\,\Sigma_1 \uparrow \theta :\Rightarrow \mathscr{M} \qquad \mathcal{G}_1 + \{m \mapsto \mathscr{M}\} \vdash M : \Psi\end{array}}{\mathcal{G} \vdash m'_1(m'_2) : \Psi}\ \text{(SI-M-App)}$$

$\boxed{\mathcal{G} \vdash \boldsymbol{B} :\Rightarrow \exists\mathscr{A}.\,\mathscr{R}}$

$$\frac{}{\mathcal{G} \vdash \varepsilon :\Rightarrow \exists\varnothing.\,(\varnothing, \varepsilon)}$$

$$\frac{\begin{array}{c}\mathcal{G} \vdash B_1 :\Rightarrow \exists\mathscr{A}_1.\,(\mathscr{R}_1, \boldsymbol{c}_1) \qquad \operatorname{dom}\mathcal{G} \cap \operatorname{dom}\mathscr{A}_1 = \varnothing \\ (\mathcal{G} \uplus \mathscr{A}_1) + \mathscr{R}_1 \vdash \boldsymbol{B}_2 :\Rightarrow \exists\mathscr{A}_2.\,(\mathscr{R}_2, \boldsymbol{c}_2) \qquad \operatorname{dom}\mathscr{A}_1 \cap \operatorname{dom}\mathscr{A}_2 = \varnothing\end{array}}{\mathcal{G} \vdash B_1 \cdot \boldsymbol{B}_2 :\Rightarrow \exists(\mathscr{A}_1 \uplus \mathscr{A}_2).\,(\mathscr{R}_1 + \mathscr{R}_2, \boldsymbol{c}_1 \cdot \boldsymbol{c}_2)}$$

$\boxed{\mathcal{G} \vdash B :\Rightarrow \exists\mathscr{A}.\,\mathscr{R}}$

$$\frac{\mathcal{G} \vdash E : \tau \Rightarrow e}{\mathcal{G} \vdash (\mathbf{val}\, x = E) :\Rightarrow \exists\{z\}.\,(\{x \mapsto \tau/z\}, \mathbf{let}\, z = e)}\ \text{(SI-B-Val)}$$

$$\frac{\overline{\mathcal{G}} \vdash T :: \kappa \hookrightarrow \tau}{\mathcal{G} \vdash (\mathbf{type}\, t = T) :\Rightarrow \exists\varnothing.\,(\{t \mapsto [\![= \tau :: \kappa]\!]\}, \varepsilon)}$$

$$\frac{\mathcal{G} \vdash M :\Rightarrow \exists\mathscr{A}.\,(\mathscr{M}, \boldsymbol{c})}{\mathcal{G} \vdash (\mathbf{module}\, m = M) :\Rightarrow \exists\mathscr{A}.\,(\{m \mapsto \mathscr{M}\}, \boldsymbol{c})}$$

$$\frac{\overline{\mathcal{G}} \vdash S \hookrightarrow \xi}{\mathcal{G} \vdash (\mathbf{signature}\, s = S) :\Rightarrow \exists\varnothing.\,(\{s \mapsto [\![= \xi]\!]\}, \varepsilon)} \qquad \frac{\mathcal{G} \vdash M :\Rightarrow \exists\mathscr{A}.\,(\mathscr{R}, \boldsymbol{c})}{\mathcal{G} \vdash \mathbf{include}\, M :\Rightarrow \exists\mathscr{A}.\,(\mathscr{R}, \boldsymbol{c})}$$

図 11　static interpretation による型つけ規則

型システム

図 11 に static interpretation に基づく型つけ（およびコンパイル）の規則を掲げます．これは $\mathcal{G} \vdash M :\Rightarrow \exists(A, N).\,(\mathscr{M}, \boldsymbol{c})$ という形の型判定に対する規則で，「型環境 $\mathcal{G}$ の下で M には

$\mathscr{M}$ というシグネチャがつき，$\boldsymbol{c}$ というターゲット言語の束縛列へとコンパイルされる．$\mathscr{M}$ に対しては A で型変数が存在量化されており，$\boldsymbol{c}$ には N に属する名前が使われている」と読むことができます．ただし，規則中で使われている $\overline{\mathscr{G}}$ は“型環境 $\mathscr{G}$ 中の名前を忘却して通常の Γ の形式の型環境にしたもの”であり，これによって一部の箇所で最初に掲げた通常の型つけ規則へとフォールバックされています．“名前を忘却する”函数は形式的には次のように定義されます：

$$\overline{\left(\mathscr{R},(A,N)\right)} \coloneqq \left(\overline{\mathscr{R}},A\right) \qquad \overline{\lbrack\!\lbrack = \tau :: \kappa \rbrack\!\rbrack} \coloneqq \lbrack\!\lbrack = \tau :: \kappa \rbrack\!\rbrack$$

$$\overline{\left\{l_i \mapsto \mathscr{U}_i\right\}_{i=1}^{n}} \coloneqq \left\{l_i \mapsto \overline{\mathscr{U}_i}\right\}_{i=1}^{n} \qquad \overline{\lbrack\!\lbrack = \xi \rbrack\!\rbrack} \coloneqq \lbrack\!\lbrack = \xi \rbrack\!\rbrack$$

$$\overline{\left\langle \mathscr{G} \;\middle|\; \forall A.\,\Sigma \to \xi \;\middle|\; \lambda m.\,M \right\rangle} \coloneqq \forall A.\,\Sigma \to \xi \qquad \overline{\tau/z} \coloneqq \tau$$

それでは規則全体を眺めてみます． 大枠は前節までに紹介した型システムと同様ですが，(SI-B-Val)，(SI-M-Fun)，(SI-M-App) に顕著な違いがあります． まず， ターゲット言語の束縛を生成する中心となる規則は (SI-B-Val) のみです． すなわち， 最も細かい単位としては入力の **val** による束縛に基づいてしかコードが生成されません． また，“出力コードが真に増えている”のが (SI-B-Val) と (SI-M-App) だけで，他の規則はいずれもコードを出力しないかその部分構造で生じたコードを繋げているだけであることから，ファンクタの適用以外ではコードが複製されることがないということも言えます．(SI-B-Val) の前提側で使われているコア言語の型つけ規則はやはりコア言語に基づいて適切に用意する必要がありますが，図 12 に示すように，この規則中でまさにターゲット言語の名前が取り出して使われています．

さて， ファンクタの静的除去を担う重要な規則が (SI-M-Fun) と (SI-M-App) です．(SI-M-Fun) はファンクタを走査した際にファンクタクロージャをつくって“戻り値”とし，(SI-M-App) ではそのファンクタクロージャから型環境と λ 抽象のようにして保持されたモジュール式を取り出し，そのモジュール式を走査してコード生成しています． 以下のような， 通常の λ 計算に対してクロージャを用いて定式化した big step semantics の値呼び評価規則と比べると，(SI-M-Fun) と (BS-Abs) が， また (SI-M-App) と (BS-App) が， それぞれおおよそアナロジーになっていることが感じられるかと思います：

$$\frac{\eta(x) = v}{\eta \vdash x \downarrow v}\ \text{(BS-Var)} \qquad \frac{}{\eta \vdash \lambda x.\,e \downarrow \left\langle \lambda x.\,e \;\middle|\; \eta \right\rangle}\ \text{(BS-Abs)}$$

$$\frac{\eta \vdash e_1 \downarrow \left\langle \lambda x.\,e \;\middle|\; \eta_1 \right\rangle \qquad \eta \vdash e_2 \downarrow v_2 \qquad \eta_1 + \left\{x \mapsto v_2\right\} \vdash e \downarrow v}{\eta \vdash e_1\,e_2 \downarrow v}\ \text{(BS-App)}$$

$\boxed{\mathscr{G} \vdash E : \tau \Rightarrow e}$

$$\frac{\mathrm{Access}(\mathscr{G}, x') = \tau / z}{\mathscr{G} \vdash x' : \tau \Rightarrow z}$$

図 12　static interpretation でのコア言語の型つけの拡張

ただし，(SI-M-App) では “受け取ったモジュールを m に束縛する” 際に多少工夫が必要になります．m はファンクタのボディ部分 M では “A_1 の型変数について抽象化された下で Σ_1 のシグネチャがつくモジュール” の情報しかもたないので，m に紐づけるシグネチャとしては $\mathscr{M}_2$ は “強すぎる”，すなわち “多くの情報を持ちすぎている” のです．そこで，$\mathscr{M}_2$ に含まれる名前などの情報は保持しつつも $\exists A_1.\, \Sigma_1$ のシグネチャがつくモジュールへと “弱めて” $\mathscr{M}$ とし，これを m に紐づけて M を走査する，という形式化をしています．この “弱める処理” を伴った部分型つけの規則が図 13 です．

..... と，なんだか自信満々なように書いてきましたが，節の最初でもことわったようにこれは著者が独自に拡張して定義を試みたものであって，正当性の保証はなく，あくまでもこういう直観で定式化できる手法があるよという程度の紹介でした．

正当性の厳密な証明は結構大変だろうと思います．というのも，特に (SI-M-App) は帰納法が回りにくい形をしているためです．おそらく（強正規化定理の証明に用いられるように）logical relation のような道具が必要となるかと思います．

大々的に説明したわりには最後の節は随分と歯切れの悪い解説になってしまいましたが，とりあえずファンクタをコンパイル時に除去することがこの節で触れたような要領で可能であるということだけ頭の片隅に入れていただければ幸いです．

6. まとめ

本稿ではモジュールシステムの定式化のひとつである F-ing modules について，主に型検査器として実装する観点を主軸として紹介しました．let 多相程度の型推論器・型検査器を実装したことのある方なら F-ing modules も劇的に大きな支障はなく実装できるかと思いますので，自作言語にモジュールシステムを組み込むのに本稿を参考としてもらえれば幸いです．型システムの正当性や関連研究との比較等については （著者自身がまだそれほど明るくないこともあって）殆ど言及しませんでしたが，理論自体に関心を持たれた読者におかれては是非元論文 [11] を参照されたいと思います．このほか，モジュールシステムの和文のサーヴェイについて

$\boxed{\Gamma \vdash \mathscr{M}_1 \leqslant \xi_2 \uparrow \theta :\Rightarrow \mathscr{M}_2}$

$$\frac{\mathrm{dom}\,\theta = \mathrm{dom}\,A_2 \qquad \forall \alpha \in \mathrm{dom}\,\theta.\ \big(\Gamma \vdash \theta(\alpha) :: A_2(\alpha)\big) \qquad \Gamma \vdash \mathscr{M}_1 \leqslant \theta\Sigma_2 :\Rightarrow \mathscr{M}_2}{\Gamma \vdash \mathscr{M}_1 \leqslant \exists A_2.\ \Sigma_2 \uparrow \theta :\Rightarrow \mathscr{M}_2}$$

$\boxed{\Gamma \vdash \mathscr{M}_1 \leqslant \Sigma_2 :\Rightarrow \mathscr{M}_2}$

$$\frac{\begin{array}{c}\mathrm{dom}\,\mathscr{R}_1 \supseteq \mathrm{dom}\,R_2 \\ \{x_1, \ldots, x_a\} = \mathrm{dom}\,R_2 \cap \mathrm{ValIdent} \qquad \{m_1, \ldots, m_b\} = \mathrm{dom}\,R_2 \cap \mathrm{ModIdent} \\ \forall i \in \{1, \ldots, a\}.\ \big(\Gamma \vdash_{\mathrm{type}} \tau_i \leqslant R_2(x_i) \quad \text{where } \tau_i/z_i = \mathscr{R}_1(x_i)\big) \\ \forall t \in \mathrm{dom}\,R_2 \cap \mathrm{TypeIdent}.\ \big(\Gamma \vdash_{\mathrm{type}} \tau_1 = \tau_2 \quad \text{where } \mathscr{R}_1(t) = \lfloor = \tau_1 :: \kappa_1 \rfloor \wedge R_2(t) = \lfloor = \tau_2 :: \kappa_2 \rfloor\big) \\ \forall j \in \{1, \ldots, b\}.\ \big(\Gamma \vdash \mathscr{R}_1(m_j) \leqslant R_2(m_j) :\Rightarrow \mathscr{M}_j\big) \\ \forall s \in \mathrm{dom}\,R_2 \cap \mathrm{SigIdent}.\ \big(\Gamma \vdash_{\mathrm{abs}} \xi_1 \leqslant \xi_2 \wedge \Gamma \vdash_{\mathrm{abs}} \xi_2 \leqslant \xi_1 \quad \text{where } \mathscr{R}_1(s) = \lfloor = \xi_1 \rfloor \wedge R_2(s) = \lfloor = \xi_2 \rfloor\big)\end{array}}{\begin{array}{r}\Gamma \vdash \mathscr{R}_1 \leqslant R_2 :\Rightarrow \big\{x_i \mapsto R_2(x_i)/z_i\big\}_{i=1}^{a} \uplus \big\{t \mapsto R_2(t) \bigm| t \in \mathrm{dom}\,R_2 \cap \mathrm{TypeIdent}\big\} \\ \uplus \big\{m_j \mapsto \mathscr{M}_j\big\}_{j=1}^{b} \uplus \big\{s \mapsto R_2(s) \bigm| s \in \mathrm{dom}\,R_2 \cap \mathrm{SigIdent}\big\}\end{array}}$$

$$\frac{\Gamma \vdash_{\mathrm{conc}} \forall A_1.\ \Sigma_1 \to \xi_1 \leqslant \forall A_2.\ \Sigma_2 \to \xi_2}{\Gamma \vdash \big\langle \mathscr{G} \bigm| \forall A_1.\ \Sigma_1 \to \xi_1 \bigm| \lambda m.\ M \big\rangle \leqslant \forall A_2.\ \Sigma_2 \to \xi_2 :\Rightarrow \big\langle \mathscr{G} \bigm| \forall A_2.\ \Sigma_2 \to \xi_2 \bigm| \lambda m.\ M \big\rangle}$$

図 13　static interpretation のための部分型つけ

は `@elpin1al` さんによるページ [3] が大変充実しています.

7. 謝辞

本稿は同じくヤバイテックトーキョーのメンバである wasabiz 氏のレビューを受け，数多くの有益なコメントを貰いました[*28]．また，特に普段よりサークルの運営に尽力し締切等のスケジュールを管理してくれている zeptometer 氏には大変お世話になりました．両氏をはじめとしてサークルメンバには感謝を表したいと思います.

真面目か？　いや, 思ったより長い記事になってレビューしてもらうのもそこそこ手間だったであろうし，サークルとしての手続きなどもかなり任せっきりになってしまっているし，謝辞入れないのもなと思って書きました．読者の皆様におかれましても読んで頂きありがとうござ

28 それでも依然として本稿の内容に瑕疵のある場合，その原因は全て著者に帰されるものです（念のため）.

いました．良き自作言語開発を！

参考文献

[1] Jeremie Dimino. *utop — a universal toplevel (i.e., REPL) for OCaml*. `https://github.com/ocaml-community/utop`, 2011.

[2] `@elpin1al`. *tweet*. `https://twitter.com/elpin1al/status/1269196178391887872?s=20`, 2020.

[3] `@elpin1al`. *Publications by El Pin Al*. `https://elpinal.gitlab.io/publications/`, 2018.

[4] Martin Elsman, Troels Henriksen, Danil Annenkov, and Cosmin E. Oancea. Static interpretation of higher-order modules in Futhark: functional GPU programming in the large. In *Proceedings of the ACM on Programming Languages 2, ICFP, Article 97*, pages 1–30, 2018.

[5] Steve Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *Proc. of ICFP'01*, pages 74–85, 2001.

[6] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13 , pages 225–230, 1981.

[7] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146 , pages 29–60, 1969.

[8] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science (JCSS)*, 17 , pages 348–374, 1978.

[9] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[10] 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 宣洋 訳, 才川 隆文 訳, 今井 健男 訳）. 型システム入門 . オーム社 , 2013.

[11] Andreas Rossberg, Claudio Russo, and Derek Dreyer. F-ing modules. *Journal of Functional Programming*, 24 (5) , pages 529–607, 2014.

[12] Takashi Suwa. *Sesterl*. `https://github.com/gfngfn/Sesterl`, 2020.

[13] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248 (1-2) , pages 211–242, 2000.

YABAITECH.TOKYO vol.5

2020 年 12 月 1 日 技術書典 9 版 (電子版)

2021 年 7 月 10 日 技術書典 11 版 (書籍版)

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