ML モジュールの形式化と正確なコンパイル手法

El Pin Al

言語実装 Advent Calendar 2018 10 日目の記事です。

1. はじめに

Standard ML や OCaml などに代表される ML というプログラミング言語族はそれぞれが二つの言語からなる。一つはコア言語であり、もう一つはモジュール言語である。ML のモジュールシステムは、大きな ML プログラムを整理、管理する際に生じる様々な問題を解決するための手法である。

コア言語の型と項は共に**ストラクチャ**（structure）と呼ばれるものにひとまとめにされる。ストラクチャは、他のストラクチャもその要素として含むことができるという点で、階層的である。ストラクチャのコンポーネントは**ドット記法** (Cardelli and Leroy; 1990) でアクセスされる。ドット記法とは単に $M.t$ のように、モジュール M からコンポーネント t を取り出す方法を指す。

シグネチャ（signature）はモジュールのインターフェイスを指定するために用いられる。モジュールの型コンポーネントは transparent（明瞭、manifest）あるいは opaque（不明瞭、abstract）に指定される。例えば、次のコード例 (Standard ML の文法) では、ストラクチャ `M` の型コンポーネント `t` は (どのような型と等しいか不明であるという点で) 不明瞭である。

```
signature S = sig
  type t
end

structure M :> S = struct
  type t = int
end
```

ストラクチャ `M` の外からは、`M.t` が `int` と等しいということは分からない。一方シグネチャ `S` で型コンポーネント `t` が次のように明瞭に定義されていたのであれば、

```
signature S = sig
  type t = int
end
```

`M.t` が `int` と等しいということが外からも分かる。

また、ストラクチャはシグネチャの指定する全てのコンポーネントの実装を最低限提供していれば、そのシグネチャにマッチする。次の例では、`M` がシグネチャ `S` に対して余分なコンポーネント `y` を持っているが正しく型付けされる。

```
signature S = sig
  val f : int -> bool
end

structure M :> S = struct
  val y = 71
  fun f x = x < y
end
```

ファンクタ (functor) とは、モジュール上の関数である。一階ファンクタはストラクチャ上の関数であり、高階ファンクタは (ファンクタも含んだ) モジュール上の関数である。高階ファンクタは高階モジュールと呼ばれることも多い。Standard ML '97 では一階ファンクタのみがサポートされているが、SML/NJ や Moscow ML、Alice ML、OCaml などは高階ファンクタをサポートしている。

Standard ML におけるファンクタは generative である。これは「ファンクタを適用した結果できるモジュールの抽象型は、ファンクタが適用されるたびに新しく生成される」ことを意味する。

一方、OCaml におけるファンクタは applicative である[*1]。Applicative functor は同じ引数[*2]を与えられると同じ抽象型を返す。

---

1 昔はこの説明でよかったが、2014 年にリリースされた OCaml 4.02.0 で generative functor が追加された (ドキュメント)。

2「"同じ引数" って具体的に何？」という疑問に関しては、型安全性だけを実現するためには、静的部分が同じであればよい。しかし、「抽象安全」を実現したいのであれば動的部分も考慮する必要がある (Rossberg, Russo and Dreyer; 2014)。

モジュールは強力なデータ抽象の仕組みをもたらす。モジュールをシグネチャで隠蔽 (seal) することによって、実装側は内部のデータ表現を利用者側から隠す事ができる。
これを implementor-side データ抽象 (もしくは provider-side データ抽象) という。ML はさらに client-side データ抽象をファンクタにより導入している (Dreyer; 2005) (Crary; 2017)。

モジュール言語とコア言語は階層化されている。モジュールをコア言語の通常の値として操作することは**第一級モジュール**と呼ばれる概念へと繋がり、型検査を決定不能にしてしまうことが知られている (Lillibridge; 1997) [*3]。ただし、モジュールを明示的に pack することによってコア言語の第一級の項として扱うのであれば実現可能である (Russo; 2000) (Rossberg, Russo and Dreyer; 2014)。

2章ではモジュールの形式化を、3章では POPL 2019 で出る予定の Fully abstract module compilation (Crary; 2019) を取り扱う。4章は、より ML モジュールについて知りたい人のためのガイドである。

この文章は『型システム入門』あるいはそれに類する型理論の知識を要求する。より正確には、型理論の基礎とカインド、全称型、存在型を理解していれば、この文章を読むことは可能なはずである。

1.1. 記法

k はカインドを、c は型コンストラクタを、τ は型を、α, β は型変数を、e は項を、σ はシグネチャを、M はモジュールを、それぞれ表す。

組は $\langle e_1, e_1 \rangle$ で、積型は $\tau_1 \times \tau_2$ で表される。T を真の型を表すカインドとする。

Γ は型付け環境である。

代入 (substitution) は $[\tau'/\alpha]\tau$ (や $[e/x]\tau$) で表すものとする。変数捕縛は暗黙の α 変換によって回避されるものとする。

1.2. 準備

1.2.1. 依存型

もっとも一般的な意味では、依存型 (dependent type) は型を値とする関数である。この定

3 しかしながら、ある制約を課すことによって型検査の決定可能性を犠牲にせずにコア言語とモジュール言語の統合を達成した 1ML (Rossberg; 2015) というプロジェクトがある。

義はF_ωの型演算子などを含む。しかし普通はその中でも特に項でインデックス付けされた型のことを指す。

この文章で用いる依存型は依存和 (dependent sum) と依存積 (dependent product) である。

依存和型 $\Sigma x{:}\tau_1.\tau_2$ は積型 $\tau_1 \times \tau_2$ の一般形である。導入形式は以下の通り：

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : [e_1/x]\tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \Sigma x{:}\tau_1.\tau_2}\ \mathrm{PAIR}$$

依存積型 $\Pi x{:}\tau_1.\tau_2$ は関数型 $\tau_1 \to \tau_2$ の一般形である。導入形式は以下の通り：

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \Pi x{:}\tau_1.\tau_2}\ \mathrm{ABS}$$

依存和や依存積をカインドに持ち上げると、依存和カインド $\Sigma\tau{:}k_1.k_2$ と依存積カインド $\Pi\tau{:}k_1.k_2$ ができる。

依存型についてより詳しく知りたい読者は (Pierce; 2005) の 2 章などを読むとよい。

1.2.2. Singleton Kinds

Singleton kind $\mathrm{S}(\tau)$ は τ と等しい全ての型を表すカインドである (Stone and Harper; 1999)。例えば $\mathrm{int} : \mathrm{S}(\mathrm{int})$ である。導入形式は以下の通り：

$$\frac{\Gamma \vdash \tau : \mathrm{T}}{\Gamma \vdash \tau : \mathrm{S}(\tau)}\ \mathrm{SINGLETON}$$

2. モジュールの形式化

2.1. 型によるモジュールの解釈

モジュールをどのような型によって解釈するのかは様々な研究がなされてきたが、それらを俯瞰するのに丁度良い論文は (Jones; 1996) である。モジュールの型を表すために用いられるのは存在型や依存型、manifest type である。以下ではそれらを順に見ていく。

2.1.1. 存在型

1985 年 (POPL) と 1988 年 (TOPLAS) に

- John C. Mitchell and Gordon. D. Plotkin (1985). Abstract types have existential types.
- John C. Mitchell and Gordon. D. Plotkin (1988). Abstract types have existential type.

という論文が出た。

これはモジュールの型コンポーネントを存在型として扱う。例えば

```
structure M = struct
  type t = int
  val x : t = 0
  fun f (x : t) : int = x
end
```

というモジュール M に対して、$\exists\alpha.\big(\alpha \times (\alpha \to \text{int})\big)$ という型を与える (ここでは簡単のためレコードではなく組を用いてモジュールを表す)。

存在量化子は M.t がどんな型だったかを完全に隠してしまう。これはモジュールを第一級の値として扱うことを許す。

しかし、(MacQueen; 1986) は存在型を用いる方法の欠点を指摘した。 次のように、 モジュールを unpack すると

```
unpack [a, x] = M in N
```

型変数 a が得られるが、これは int と何の関係もない存在である。

さらに、複数回 unpack した場合、

```
unpack [a, x] = M in
  unpack [b, y] = M in N
```

型変数 a, b の間の関係性すら失われる : 型 a と型 b は全くの別物と見なされるのだ。

他にも、unpack はスコープが閉じているという点で問題である。つまり、unpack されたモジュール x の出現は全て予想されなければならず、十分早く unpack しなければならない。これは実用的なモジュラープログラミングの観点からは非常に不便である。

このような欠点を保有しているため、存在型はモジュールを表現するものとして使い物にならないと思われるかもしれないが、(Rossberg, Russo and Dreyer; 2014) の F-ing modules

は存在型を非常に上手く利用し洗練されたモジュールシステムを展開していくので一読をお勧めする。

2.1.2. 依存型

(MacQueen; 1986) が存在型を批判し、依存型の利用を促進して以来、多くのモジュール論文は依存型ベースである。

例えば、

```
structure M = struct
  type t = int
  val x : t = 0
end
```

は、$\langle \mathrm{int}, 0 \rangle : \Sigma\alpha{:}\mathrm{T}.\alpha$ と解釈される。つまりモジュールとは型と項の組で、その組は依存和を持つ。

π_i を組の i 番目の射影だとすると、$\pi_1\langle \mathrm{int}, 0 \rangle$ はモジュール `M` の型コンポーネントである。これは int と等しい。

依存型による形式化の元で静的に決定可能な型検査を実現するためには、phase distinction という概念が重要になってくる。

2.1.3. Manifest Types (Translucent Sums)

(Harper and Lillibridge; 1994) と (Leroy; 1994) は独立に translucent sums を導入した :

- Robert Harper and Mark Lillibridge (1994). A type-theoretic approach to higher-order modules with sharing.
- Xavier Leroy (1994). Manifest types, modules, and separate compilation.

Translucent (半透過的) という言葉が表すように、opaque (非透過的) と transparent (透過的) の両方の機能を取り入れている。 これは抽象的な型宣言と、 具体的な型宣言の両方をシグネチャに含む事ができる仕組みである。Standard ML '97 や OCaml などがサポートしている。

以下のストラクチャ `M` は、

```
signature S = sig
  type s
  type t = s * s
  val x : t
end

structure M :> S = struct
  type s = int
  type t = s * s
  val x = (1, 2)
end
```

(Jones; 1996) の記法を用いると、$\exists\alpha.\exists\beta = \alpha \times \alpha.\beta$ と表される。

(Leroy; 1994) は `type s` の形の宣言を abstract type declaration (抽象型宣言)、`type t = s * s` の形の宣言を manifest type declaration (顕在型宣言) と呼んだ。

実際には、型レベルで manifest type declaration と abstract type declaration を区別するより、singleton kind を用いて区別した方が扱いが簡単である (Leroy; 1994) (Lillibridge; 1997)。Singleton kind と依存和を組み合わせると、モジュール `M` は $\Sigma\alpha{:}\mathrm{T}.\big(\Sigma\beta{:}\mathrm{S}(\alpha \times \alpha).\beta\big)$ のように表される。

2.2. モジュールの形式化における課題

モジュールを形式化する際には phase distinction や avoidance problem に注意しなければならない。

2.2.1. Phase Distinction

(Cardelli; 1988) は phase distinction という用語を導入した。Phase distinction はコンパイル時と実行時の区別を意味する。

依存型を用いてモジュールを表現する場合、高階ファンクタまで拡張すると、phase distinction を侵害してしまう。(Harper, Mitchell and Moggi; 1990) は高階モジュール計算がストラクチャのみからなる言語の定義的拡張であることを用いて、phase distinction を遵守しつつ高階ファンクタをサポートし、コンパイル時に決定可能な型検査を実現した。

Advanced topics in types and programming languages (Pierce; 2005) の 8 章 4 節にも phase distinc-

tion の記述がある。 型検査が実行時式 (run-time expression) の同値性判定を行わないならば、phase distinction が遵守されていて、かつその言語は**静的型付き言語**であると言われる。

もし型検査が実行時式の同値性を検査するのであれば (これはしばしば symbolic execution と呼ばれる)、phase distinction は侵害され、 その言語は**依存型付き言語**であると言われる。(動的型付き言語とは言わないことに注意。 またこの場における依存型というのは意味的なもので、「構文的に項が型に現れるかどうか」とは別の話である。)

$M.t$ という型は表面上モジュール M に依存しているように見える。 モジュールというのは型も項もそのコンポーネントとして含むので、$M.t$ を別の型と比較することはモジュールの比較を要求し、phase distinction を侵害する恐れがある。それゆえに phase distinction を遵守するためには phase separation などの手法を用いることが必要である。

より詳しく知りたい場合は Phase distinction and separation for modules を見ること。

2.2.2. Avoidance Problem

次のモジュールレベルの let 式を考える。

```
let structure M :> sig
    type t
    val x : t
  end = struct
    type t = int
    val x = 3
  end
in
  struct
    val y = M.x
  end
end
```

このモジュールの (主要) シグネチャは何であろうか。`sig val y : M.t end` と言いたいところだが、`M` のスコープは let 式で閉じているため、このモジュールのシグネチャとして含むことができない。

この例だと `sig end` という、より情報量の少ないシグネチャを与えることができるが、一般的には「同値ではない無数の選択肢が存在し、 最良のシグネチャを選択できない」 こと

がある。この問題を avoidance problem と呼ぶ。これは元々 (Ghelli and Pierce; 1992) が System $F_{\leq}$ の文脈で発見したものである。この問題の影響で型検査が不完全になったり、余分なシグネチャ注釈を必要としたりする。

2.3. モジュールのコンパイル

(Harper, Mitchell and Moggi; 1990) はストラクチャとファンクタをコンパイル時コンポーネントと実行時コンポーネントに分解する方法を示した。(MacQueen; 1986) から始まった依存型によるモジュールの研究によると、ストラクチャとは $\Sigma\alpha{:}\mathrm{T}.\sigma(\alpha)$ という型を持つ組 $[\tau, e]$ であり、ファンクタは $\Pi s{:}\big(\Sigma\alpha{:}\mathrm{T}.\sigma(\alpha)\big).\big(\Sigma\alpha{:}\mathrm{T}.\sigma'(\alpha)\big)$ という型を持つ関数である。したがってストラクチャは型 τ と項 e に分けられ、ファンクタは型レベル関数と多相関数に分解される。これを phase separation または phase splitting と呼ぶ。

3. Fully Abstract Module Compilation

この章では、POPL 2019 の conditionally accepted paper である Fully abstract module compilation (Crary; 2019) を紹介する。

この論文は dynamic correctness theorem を満たす、初のモジュールのコンパイル手法を提案する。Dynamic correctness theorem とはモジュールのコンパイルにおける translation が、その source term と contextually equivalent な target term を生成することを保証する定理である。

モジュールを含んだプログラムを、モジュールの無いプログラムに変換することを考える。このような操作をこの文章ではモジュールのコンパイル、または translation と呼ぶ。translation の入力を source language、出力を target language と呼ぶ。

Target language に型がなければ、モジュールのコンパイルは簡単である。単にストラクチャをレコードに、ファンクタを関数に変換すればよい。しかし型を保存しつつコンパイルすることには、代え難い魅力がある。型保存コンパイルには利点がいくつかあるが、その中でも大きなものは正しいコンパイラの作成を型が補助してくれることである。詳細は型付き中間言語 (TIL) や型付きアセンブリ言語 (TAL) の論文を参照すること (Tarditi, Morrisett, Cheng, Stone, Harper and Lee; 2004) (Morrisett, Walker, Crary and Glew; 1999)。

コンパイラの最も望ましい正当性 (correctness) は contextual equivalence である。Contextual equivalence は source term とその translate 結果がどんな正当な方法でも区別できないことをいう。これは最も強力な正当性である。なぜなら translate されたコードがどのように

使われるか、さらに translate されるかどうか、他のコードとリンクされるのか、もしくはもっと他の方法で用いられるか、などとは全く関係がないからである。Contextual equivalence を示すには型が重要である。

Full abstraction は「二つのモジュールが等しいとき、かつそのときに限り、それらの translate 結果は等しい」という性質である。Abstraction preservation はどんな適法な target code も source code にあった抽象を破壊することができないことをいう。

Full abstraction は「他の言語からリンクされたコードでさえも、プログラマの書いた抽象を侵害しない」ことを保証する。これは複数の言語からなるプログラムの開発において特に価値がある。

(Harper, Mitchell and Moggi; 1990) は初めてモジュールのコンパイルを数学的に研究したが、コンパイル自体ではなく、高階ファンクタの equational theory (特に phase distinction 2.2.1を尊重すること) に主眼を置いていた。

Phase distinction が尊重されているかどうかは演繹的には明らかではない。モジュールは型と項を含み、ファンクタはモジュール上の関数であるので、ファンクタを適用した結果のモジュールの型コンポーネントは一見項に依存しているように見える。実際にはそのようなことはなく、モジュール言語というのは型が項に依存しないように構成されている。(Harper, Mitchell and Moggi; 1990) はモジュールが静的なフェーズと動的なフェーズに分離できることを示した。

この論文は phase separation アルゴリズムそのものに興味がある。Phase separation を行うとモジュール M は二つの要素 $[c, e]$ に分けられる。c が型コンストラクタ、e が項である。このとき、モジュール M は完全にコンパイルされ、c と e は module language へ一切言及しない。

(Harper, Mitchell and Moggi; 1990) は sealing と generativity を省いている。抽象化の手段は λ 抽象、つまり client-side データ抽象だけが提供され、provider-side データ抽象は提供されなかった。provider-side データ抽象は client-side データ抽象と比べてより堅牢である (Crary; 2017)。

(Shao; 1998) は (Harper, Mitchell and Moggi; 1990) に似たアルゴリズムを用いた。(Harper, Mitchell and Moggi; 1990) と違って sealing と generativity をサポートした。しかし translation はそれらを単に除去した。つまり抽象を保っていない。これは Shao の cross-module inlining をサポートするという目的にはマッチしていた。

(Shan; 2006) と (Rossberg, Russo and Dreyer; 2014) は sealing と generativity をサポートしたモジュールの、異なるコンパイル手法を提案した。それは sealing を存在型を導入するものとして扱うものであった。これは抽象の非常に自然な扱い方である。

しかし (Shan; 2006) と (Rossberg, Russo and Dreyer; 2014) は dynamic correctness result や full abstraction に言及しなかった。双方とも source language の dynamic semantics を定義しなかったのだから当然ではある。この論文では、(Shan; 2006) と (Rossberg, Russo and Dreyer; 2014) の translation は「もし target language が関数の停止を観測する方法 (call-by-value や Haskell の `seq`) を持っている場合は fully abstract ではない」ということを明らかにする。

この論文のコンパイルのアルゴリズムは、phase-separation translation として与えられる。それは (Harper, Mitchell and Moggi; 1990) が行ったように、モジュールを型コンストラクタと項に分解する。

Dynamic correctness は source と target の間の contexual equivalence を利用する。したがって、source expression と target expression が異なる言語に属するということは問題である。この問題の解決策として、Source language からモジュール関係のものを全て取り除いたものを target language となるようにする。それにより target language は source language の部分集合になる。Target term を source term とみなせば、source language とその translation 結果の比較は容易である。

3.1. モジュール計算

この論文の目的は表現力のあるモジュールシステムを研究することではなく、データ抽象の観点から重要であるモジュールシステムの根本的な要素を説明することである。

カインド 1 は unit 型コンストラクタ $\star$ を唯一の元とするカインドである。型 unit は unit 項 $\star$ を唯一の元とする型である。

モジュールの基本要素として、static atom $\langle\!| c |\!\rangle$ と dynamic atom $\langle\!| e |\!\rangle$ がある。$\langle\!| c |\!\rangle$ は型コンストラクタ c のみをコンポーネントとして持つモジュールである。$\langle\!| e |\!\rangle$ は項 e のみをコンポーネントとして持つモジュールである。

Applicative functor シグネチャは $\Pi^{\mathrm{ap}}\alpha{:}\sigma_1.\sigma_2$ のように書き、generative functor シグネチャは $\Pi^{\mathrm{gn}}\alpha{:}\sigma_1.\sigma_2$ のように書く。

型情報の伝搬には singleton kind を用いる。Singleton kind が型の同値性に影響を与えるの

で、型の同値性は文脈依存である。型変数 α と型 τ が等しいと判断されるのは、$\alpha : \mathrm{S}(\tau)$ であるときだけである。また、どのカインドにおいて型の同値性を判定するかも重要である。$\lambda\alpha{:}\mathrm{T}.\alpha$ と $\lambda\alpha{:}\mathrm{T}.\tau$ は、$\mathrm{S}(\tau) \to \mathrm{T}$ においては等しいが、$\mathrm{T} \to \mathrm{T}$ においては等しくない。

Sealing (:>) は抽象化をもたらす。例えば、型コンポーネントにシグネチャでカインド T を指定すると、その型は抽象的になる。型の抽象化を適切に強制するためには sealing を形式上は計算的作用として見る必要があり (Dreyer, Crary and Harper; 2003)、従って抽象的な型を不純な (つまり sealing を用いた) モジュールから取り出すことはできないようにしなければならない。

Generative functor は sealing をその本体に含むことができるが、applicative functor はそうではない。

Dynamic atom $\langle\!| e |\!\rangle$ から項 e をとりだすときは $\mathrm{Ext}M$ を使う。純粋なモジュールから型コンストラクタ c を取り出すときは —Ext とは違って — 構文的なものではなく次のような判断を使う。

$$\Gamma \vdash \mathrm{Fst}(M) \gg c$$

この方法は (Dreyer; 2005) が発明した。判断を使うと、静的なコンポーネントを項やモジュールに対して構文的に依存させずに済む。Γ が空のときや、文脈から推測できる場合は単に $\mathrm{Fst}(M) \gg c$ と書く。

モジュール変数 m から型コンストラクタを取り出すにはどうすればよいだろうか。そこで型変数 α を m の静的部分を表す変数として用意しておく。文脈などに $\alpha/m{:}\sigma$ という風に記録される (σ はシグネチャのメタ変数)。

3.2. Phase Separation

型コンストラクタとカインドはモジュールに関する形式を何一つ持たない。これにより source language と target language は、型コンストラクタとカインドに関しては全く同じものを共有する。これは $\mathrm{Fst}(M)$ が判断として定義されているお陰である。

Target language が source language の subset であることには三つの利点がある。

(1) Target language のメタ理論を考える必要がない (source language のメタ理論に包含される)。

(2) 型の translation は恒等関数である。

(3) Full abstraction が dynamic correctness theorem の直接の結果になる。

項の translation はかなり単純であり、source term と target term が全く同じ型を持つ。

純粋なモジュールの translation は $\Gamma \vdash_{\mathrm{P}} M : \sigma \rightsquigarrow [c, e]$ と書かれる。これはモジュール M が型コンストラクタ c と項 e に分けられることを意味する。Turnstile ($\vdash$) 右下の P は M が純粋 (pure) であることを表す。

不純なモジュールの translation は $\Gamma \vdash_{\mathrm{I}} M : \sigma \rightsquigarrow e$ と書かれる。抽象型は概念上実行時に決定されるので、不純なモジュール (つまり seal されたモジュール) は静的部を持たない。項 e はこのとき存在型を持つ。この e はモジュールが最終的に値となったときの静的部と動的部を動的に計算する項である。Turnstile 右下の I は M が不純 (impure) であることを表す。

一方、シグネチャの translation は $\sigma \rightsquigarrow [\alpha{:}k.\,\tau]$ と書かれる。τ は α に言及することができる。このとき型変数 α はカインド k を持つ。

例えば、static atom は $\langle\!| k |\!\rangle \rightsquigarrow [\_{:}k.\,\mathrm{unit}]$ となり、dynamic atom は $\langle\!| \tau |\!\rangle \rightsquigarrow [\_{:}1.\,\tau]$ となる。

Applicative functor シグネチャの静的部は引数の静的部を結果の静的部に写す。つまり $\sigma_1 \rightsquigarrow [\alpha_1{:}k_1.\,\tau_1]$, $\sigma_2 \rightsquigarrow [\alpha_2{:}k_2.\,\tau_2]$ であるとすると、$\Pi^{\mathrm{ap}}\alpha{:}\sigma_1.\sigma_2$ の静的部は $\Pi\alpha{:}k_1.k_2$ となる。

Generative functor シグネチャは静的部を持たず、動的部のみを持つ。
$\sigma_1 \rightsquigarrow [\alpha_1{:}k_1.\,\tau_1]$, $\sigma_2 \rightsquigarrow [\alpha_2{:}k_2.\,\tau_2]$ であるとすると、
$\Pi^{\mathrm{gn}}\alpha{:}\sigma_1.\sigma_2 \rightsquigarrow [\_{:}1.\,\forall\alpha{:}k_1.[\alpha/\alpha_1]\tau_1 \to \exists\alpha_2{:}k_2.\tau_2]$ となる。

純粋なモジュールを不純なものとして扱うための規則：

$$\frac{\Gamma \vdash_{\mathrm{P}} M : \sigma \rightsquigarrow [c, e] \qquad \sigma \rightsquigarrow [\alpha{:}k.\,\tau]}{\Gamma \vdash_{\mathrm{I}} M : \sigma \rightsquigarrow \mathrm{pack}\ [c, e]\ \mathrm{as}\ \exists\alpha{:}k.\tau}\ \mathrm{FORGET}$$

Sealing の規則：

$$\frac{\Gamma \vdash_{\mathrm{I}} M : \sigma \rightsquigarrow e}{\Gamma \vdash_{\mathrm{I}} (M :> \sigma) : \sigma \rightsquigarrow e}\ \mathrm{SEAL}$$

Sealing が行なっていることは M が不純であることを要求することだけである。

3.3. Contextual Equivalence

Contextual equivalence は次のように表される：

$$\Gamma \vdash e \approx e' : \tau$$

これは、二つの項 e, e' がより大きなプログラムの hole に埋め込まれたときは必ず同じ**観測可能な結果** (observable result) を生成することを意味する。

モジュールに対しては次のようになる：

$$\Gamma \vdash M \approx M' : \sigma$$

(Crary; 2017) によると次の三つの項は全て contextually equivalent である：

$$\text{pack } \Big[\text{bool}, \big\langle \text{true}, \lambda x.x \big\rangle\Big] \text{ as } \exists\alpha.\alpha \times (\alpha \to \text{bool})$$

$$\text{pack } \Big[\text{int}, \big\langle 0, \text{isEven?} \big\rangle\Big] \text{ as } \exists\alpha.\alpha \times (\alpha \to \text{bool})$$

$$\text{pack } \Big[\text{int}, \big\langle 0, \text{isZero?} \big\rangle\Big] \text{ as } \exists\alpha.\alpha \times (\alpha \to \text{bool})$$

Contextual equivalence の詳しい定義は (Pierce; 2005) の 7 章や (Crary; 2017)、あるいは (Crary; 2019) を参照せよ。

3.4. Correctness

モジュールとその translation 結果は異なる型、構文的クラス (syntactic class) に属しているが、これらを関連付ける方法が必要である。そこで、モジュールと項の相互変換を行う Snd_σ と Join_σ という二つの関数の族を定義する。Snd はモジュールから項を取り出し、Join は型コンストラクタと項からモジュールを復元する。Snd と Join は**この言語内で表現される**。Snd はコンパイル時ではなく実行時に phase separation を行うので、dynamic phase separation とも呼ばれる。

Snd と Join が言語内で定義可能ということは「モジュールがコア言語の**定義的拡張**である」ことを示す。言いかえると、モジュールはコア言語が持つ基本的表現力の域を逸脱しない。

Snd と Join を用いることによって dynamic correctness を証明することができる。Dynamic correctness は「モジュールを phase separate した結果を (不純なら unpack してから)

Join したものは元のモジュールと contextually equivalent である」ということを言う。

Dynamic correctness が成り立つことによって、いくつかの系 (corollary) を示すことができる。

Abstraction preservation は「contextually equivalent な二つのモジュールを phase separate した結果である二つの項は contextually equivalent である」ことを言う。

Full abstraction は「二つのモジュールが contextually equivalent ($M_1 \approx M_2$) であるのは、それらの phase separate された結果である型と項がそれぞれ contextually equivalent ($\tau_1 \approx \tau_2$, $e_1 \approx e_2$) であるとき、かつそのときに限る」ことを言う。

3.5. 定義的拡張

抽象を計算的作用として扱ったので、第一級モジュールを定義的拡張として自然に扱うことができる。新しい項 $\mathrm{md}_\sigma(M)$ はモジュール M を pack し、型 $\mathrm{md}(\sigma)$ を持つ。モジュール $\mathrm{tm}_\sigma(e)$ は、pack されたモジュールを普通のモジュールに戻す。これにより、抽象データ型 (cf. *Types and programming languages* 第 24 章) の実装の動的な選択が可能になる。

$\mathrm{tm}_\sigma(e)$ の型コンポーネントは任意の項に依存するので、静的に型コンポーネントを決定することはできない。したがって $\mathrm{tm}_\sigma(e)$ は不純でなくてはならない。

$\sigma \rightsquigarrow [\alpha{:}k.\tau]$ とすると、

$$\begin{aligned}
\mathrm{md}(\sigma) &\stackrel{\text{def}}{=} \exists\alpha{:}k.\tau \\
\mathrm{md}_\sigma(M) &\stackrel{\text{def}}{=} \text{let } \alpha/m = M \text{ in pack } \left[\alpha, \mathrm{Snd}_\sigma m\right] \text{ as } \exists\alpha{:}k.\tau \\
\mathrm{tm}_\sigma(e) &\stackrel{\text{def}}{=} \text{unpack } [\alpha, x] = e \text{ in } \left(\mathrm{Join}_\sigma[\alpha, x] : \sigma\right)
\end{aligned}$$

このように定義することによって、第一級モジュールが phase distinction を遵守していることは明らかである。

第一級モジュールとは別の便利な拡張は $\mathrm{purify}_{\mathrm{S}(c:\sigma)}(M)$ (Dreyer; 2005) である。これは singleton signature を持つ不純なモジュールを純粋にするものである。Generative functor を用いて applicative functor を実装するのに便利である (らしい)。

$$\mathrm{purify}_{\mathrm{S}(c:\sigma)}(M) \stackrel{\text{def}}{=} \mathrm{Join}_{\mathrm{S}(c:\sigma)}\Big[c, \text{let } m = M \text{ in } \mathrm{Snd}_{\mathrm{S}(c:\sigma)} m\Big]$$

4. Further Reading

ML モジュールの研究は 30 年以上続いていて、 論文もたくさん出ている。github.com/elpinal/modules にモジュール論文のリストがある。

4.1. F-ing Modules

個人的には F-ing modules (Rossberg, Russo and Dreyer; 2014) はおすすめのモジュール論文である。しかし、最初に読む論文としては、難易度が高いかもしれない[*4]。これは再帰モジュール以外のほとんど全ての機能を持つモジュールシステムである。これを読めば高階ファンクタ、generative functor、半透過的シグネチャ (translucent signature)、(明示的な pack による) 第一級モジュールまでなら「比較的」簡単に実装できるようになる。Applicative functor は形式化、実装共に複雑である。

4.2. *Advanced topics in types and programming languages*

(Pierce; 2005) の 8 章 (*Design considerations for ML-style module systems*) は ML のモジュールシステムに関する様々な概念の説明をしている。その中には sharing by construction と sharing by specification というあまり他では見ない用語もある。

4.3. 再帰モジュール

再起モジュールに関しては (Dreyer; 2007) などを読むとよい。他には MixML (Rossberg and Dreyer; 2013) という ML モジュールと mixin モジュールを統合する研究がある。(Im, Nakata, Garrigue and Park; 2011) は再帰モジュールの実用的な利用方法 (ファンクタの不動点など) を知るのに適している。

4.4. Open Existential Types

Open existential type (Montagu and Rémy; 2009) は、存在型の導入形式と除去形式をより細かい構成子に分解することによって、モジュールシステムが提供するような、modular type abstraction をモデル化する。

Open existential type を用いると、型の抽象化を計算的作用によって説明する必要がなくなる。

---

4 このような感想があるので、もしかしたら ATTAPL 8 章などを先に読んだ方がいいのかもしれない。ただ、ATTAPL 8 章は 53 ページあるので、もう少し短くて初心者向けの文章を捜索中。

詳細は元論文を読んでもらうこととして、すぐに分かりやすい利点としては次のようなものがある。フィールドがいくつもあるレコードの 1 つのフィールドのみを隠蔽したい場合、存在型では次のように書く。

$$\text{pack}\left[\text{int}, \{a = 1, b = 2, c = 3\}\right] \text{ as } \exists\alpha.\{a : \text{int}, b : \text{int}, c : \alpha\}$$

ここで、フィールド a や b の型について、機械的にわかる主要型をわざわざ明示しなければならない。

一方、open existential type だとより簡潔に書ける。

$$\exists\beta.\Sigma\langle\beta\rangle(\alpha = \text{int})\{a = 1, b = 2, c = (3 : \alpha)\}$$

フィールドが増えたときに後者の方がより扱いやすいのは明らかである。

5. さいごに

この文章について、誤謬・誤植に気づいた場合や、ご意見・その他諸々ありましたら遠慮なさらずに @elpin1al までご連絡ください。

6. 参考文献

- John C. Mitchell and Gordon. D. Plotkin (1985). Abstract types have existential types.
- David B. MacQueen (1986). Using dependent types to express modular structure.
- Luca Cardelli (1988). Phase distinctions in type theory.
- John C. Mitchell and Gordon. D. Plotkin (1988). Abstract types have existential type.
- Luca Cardelli and Xavier Leroy (1990). Abstract types and the dot notation.
- Robert Harper, John C. Mitchell and Eugenio Moggi (1990). Higher-order modules and the phase distinction.
- Giorgio Ghelli and Benjamin C. Pierce (1992). Bounded existentials and minimal typing.
- Robert Harper and Mark Lillibridge (1994). A type-theoretic approach to higher-order modules with sharing.
- Xavier Leroy (1994). Manifest types, modules, and separate compilation.
- Mark P. Jones (1996). Using parameterized signatures to express modular structure.

- Mark Lillibridge (1997). Translucent sums: A foundation for higher-order module systems.
- Zhong Shao (1998). Typed cross-module compilation.
- Greg Morrisett, David Walker, Karl Crary and Neal Glew (1999). From System F to typed assembly language.
- Chris Stone and Robert Harper (1999). Decidable type equivalence in a language with singleton kinds.
- Claudio V. Russo (2000). First-class structures for Standard ML.
- Benjamin C. Pierce (2002). *Types and programming languages*.
- Derek Dreyer, Karl Crary and Robert Harper (2003). A type system for higher-order modules.
- David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper and Peter Lee (2004). TIL: A type-directed, optimizing compiler for ML.
- Derek Dreyer (2005). Understanding and evolving the ML module system.
- Benjamin C. Pierce (2005). *Advanced topics in types and programming languages*.
- Chung-chieh Shan (2006). Higher-order modules in System F_ω and Haskell.
- Derek Dreyer (2007). A type system for recursive modules.
- Benoît Montagu and Didier Rémy (2009). Modeling abstract types in modules with open existential types.
- Hyonseung Im, Keiko Nakata, Jacques Garrigue and Sungwoo Park (2011). A syntactic type system for recursive modules.
- Andreas Rossberg and Derek Dreyer (2013). Mixin' up the ML module system.
- Andreas Rossberg, Claudio V. Russo and Derek Dreyer (2014). F-ing modules.
- Andreas Rossberg (2015). 1ML — Core and modules united (F-ing first-class modules).
- Karl Crary (2017). Modules, abstraction, and parametric polymorphism.
- Karl Crary (2019). Fully abstract module compilation.