分割できながら等価：型レベル・エイリアス

Jacques Garrigue　名古屋大学
Leo P. White　ケンブリッジ大学

型レベル・エイリアス

- とても**単純な機能**なのに、
- **適用型ファンクター**を補完するし、
- **名前空間**としてのモジュールと**分割コンパイル**の矛盾も解決。
- 先週発表された **OCaml 4.02** から利用可能！

定義

シグネチャーでもエイリアスを許すだけ。シグネチャーで

```
module S = String
```

と書くと、S が String のエイリアスであるという宣言になる。
具体的には、S が使われる度に String に展開される。

単元型として理解することもできる

```
module S : (module String)
```

型推論規則

— 型レベル・エイリアスの推論

```
# module M = P ;;
module M = P
```

$$\frac{\Gamma \vdash P \text{ well formed}}{\Gamma \vdash P : (\texttt{module}\ P)}(*)$$

— 部分型関係の拡張

$$\frac{\Gamma(P) = S}{\Gamma \vdash (\texttt{module}\ P) <: S}$$

ここで P はモジュール・パスを指す。

(*) P がモジュール定義の右辺であるときのみ。

起源

– モジュールの型レベル・エイリアスが機能として初めて Traviatta [Nakata&G, ICFP 2006] で現れた

– 再帰モジュールの型推論のために導入された

```
module rec Tree = struct
  module F = Forest
  ...
end
and Forest  = struct ... Tree ... end
```

F の型を推論するときには Forest の型がまだ分からないので、エイリアスとして扱う必要がある

発見

しばらくしてから、型レベル・エイリアスが OCaml 流の適用型ファンクターと相性がいい事に気づいた。

```
module S = String
module SSet = Set.Make(S)
module StringSet = Set.Make(String)
let f (x : SSet.t) = (x : StringSet.t)
This expression has type SSet.t = Set.Make(S).t
but expected type StringSet.t = Set.Make(String).t
```

S と String の同値性を推論できない 4.01 以前の OCaml では最後の行がエラーになる、型レベル・エイリアスがあると成功する

適所

- 適用型ファンクターを補完できても、優先度が低い
- しかし、本来の役割はプログラム解析の**簡単化**
- モジュール・エイリアスの型付けを軽量にすることで、
  型レベル・エイリアスは**依存関係を減らす**
- そのおかげで**柔軟で階層的な名前空間**が構築できるようになる

モジュールによる名前空間

- ML のモジュール・システムは非常に**強力**である
- 入れ子モジュールにより、**階層的なライブラリー**が構築できる
- 型の共有のおかげで、あるモジュールを名前空間の別の場所に**コピー**できる
- MLのモジュールは**最高の名前空間**ではないか？

OCaml: 崩れた階層

この理想は**理論**にとどまる

- **分割コンパイル**を可能にするために、モジュールがファイルにマップされる
- ライブラリーは木ではなく、**モジュールの森**になる。同時に使うライブラリーのモジュールが混ざる

→**名前の衝突**が起こりうる (リンク不能になる).

- **-pack** コマンドにより、ライブラリーを一つのモジュールにまとめられるが、**不可分**になってしまう

→モジュールを名前空間として使うと、インターフェースと実行ファイルが**巨大**になる (例: Jane Street 社の Core)

SML/NJ: ツール任せの階層

- SML/NJ では前述の問題が回避できる、
- しかし、コンパイル・マネジャー（CM）という外部ツールを使わなければならない
- 名前空間はCMの特殊なファイルで表現され、専用の構文を使う
- 言語の重要な部分が仕様に含まれない

パックされたライブラリー

このライブラリーは二つのモジュール・ファイル `mylibA.ml` と `mylibB.ml`，そして名前空間ファイル `mylib.ml` からなる

```
Mylib                          Interface
 module A = MylibA      :       module A : sig ... end
 module B = MylibB              module B : sig ... end
```

`open` で快適に使える

```
open Mylib
let x = A.f 3
```

しかし、分割コンパイルはモジュールの型しかみないので、このプログラムは`Mylib`、`MylibA`と`MylibB`をリンクする

型レベル・エイリアスを使えば

型レベル・エイリアスのおかげで型が変わる

```
Mylib                    Interface
 module A = MylibA  :     module A = MylibA
 module B = MylibB        module B = MylibB
```

結果的には、`Mylib.A`への参照がクライアント側で`MylibA`に変換できる

```
open Mylib
let x = A.f 3
```

このプログラムが`MylibA`のみをリンクする。`A.f` の代わりに `MylibA.f` を書くのと同じ。

依存関係の推論

互換性のために、依存関係を減らすには-no-alias-depsというフラグを使わなければならない

そのときの依存関係は以下のようになる

リンク・コンパイル時	Mylib	MylibA	MylibB
Mylib (デフォルト)	×	○	○
Mylib (-no-alias-deps)	×	×	×
Client (デフォルト)	○	○	○/×*
Client (-no-alias-deps)	コ	○	×

(*) Mylib が -no-alias-deps でコンパイルされたかどうかによる
(コ) コンパイル時のみ

ライブラリー作成への応用

パックを分割可能にするには、以下のレシピーを使う

1. 全てのモジュールの短い名前をユニークな長い名前にマップする名前変換ファイルを作る
2. このファイルを全てのモジュールで `open` する。これにより常に短い名前が使える
3. インターフェース用に輸出したい名前だけを変換する別の名前変換ファイルを作ってもいい

この方法を使えば、`MylibB` の中でも `MylibA` を `A` と書ける

ライブラリーの例

```
Mylib
 module A = MylibA      (* does not require MylibA *)
 module B = MylibB      (* does not require MylibB *)
MylibA
 open Mylib             (* compile-time dependency *)
 let f x = x+1
MylibB
 open Mylib             (* compile-time dependency *)
 let g x = (A.f x) * 2          (* requires MylibA *)
```

Mylib を先にコンパイルしなければならないが、そこからMylibA と MylibB への依存関係がないので、依存の循環が生じていない

便利なフラグ

元のファイルを完全に隠す `-pack` コマンドに比べて、この手段では以下のステップが要る

- ソースファイルを衝突のない長い名前に**改名**
- 全てのファイルの先頭に **open** 宣言を**追加**

この作業を軽減するために、以下のフラグが使える

```
ocamlopt -no-alias-deps -open Mylib -o mylibA.cmx a.ml
```

効果

完全なベンチマークは行っていないが、Core/Async のコンパイルで以下のことが分かった

- 最新のコンパイラーを使うだけで、コンパイルされた型ファイルのサイズが1/3に減った。そのおかげでコンパイル時間も短くなる
- 上のレシピーを応用すると実行ファイルのサイズが半分になった（場合によって1/10も）

制限と今後の課題

現時点では、型レベル・エイリアスはモジュール・パスの一部にしか対応していない

以下は型レベル・エイリアスにならない

- ファンクターの適用
- 型の制約
- ファンクターの引数
- 再帰モジュール

名前空間への応用に関して、これでは特に困らないが、将来的には適用型ファンクターの改善のためにその場合も扱いたい

PR#4049

依存関係の改善以外に、適用型ファンクターの本質的なバグが解決される

```
module A = struct
  module B = struct type t let compare x y = 0 end
  module S = Set.Make(B)
  let empty = S.empty
end
module A1 = A;;
A1.empty = A.empty;;
```

このプログラムでは最後の行がエラーになる。型レベル・エイリアスがあると治る

PR#3476

```
module FF(X : sig end) = struct type t end
module M = struct
  module X = struct end
  module Y = FF (X)                              (* XXX *)
  type t = Y.t
end
module F (Y : sig type t end)
         (M : sig type t = Y.t end) = struct end
module N = F (M.Y) (M);;
```

このプログラムでも最後の行がエラーになるが、必要な等式がファンクターの適用を含むので、まだ治らない