分割できながら等価：型レベル・エイリアス

Jacques Garrigue

OCaml のモジュールシステムは強力だが、名前空間の形成に使うと様々な問題点がある。まず、入れ子モジュールで階層化しようとすると、ファイル分割ができなくなるという問題。また、それを諦めて同じモジュールを複数の名前空間にコピーすると、一部の等価性が失われたり、型情報が異常に大きくなったりする。OCaml 4.02 に導入された型レベル・エイリアスはモジュールへのリンクを型情報に記録することでそれらの問題の解決に貢献している。

1 背景

ML のモジュールシステムが強力で、少なくとも理論的には言語に名前空間のための特別な機構などを加えなくても、ライブラリーが柔軟に構成できるはずである。入れ子モジュールが階層化を可能にし、ファンクターが柔軟なリンク機能を提供している。

しかし、これが言語の内側で正しくても、モジュールシステムのみでライブラリーの全ての問題が扱える訳ではない。処理系が解決しなければならない問題として、モジュール名をどうライブラリーファイルに結びつくか、そしてどうやって再コンパイルが少ない形で分割コンパイルをサポートできるかが上げられる。

OCaml [6] が採用したアプローチは至って単純である。ローカルに束縛されていないモジュール名がファイルシステム上に同名のファイルとして検索される。ファイルの配置を自由にするために、ライブラリーパスというフォルダーのリストが使われている。そういった外部モジュールは**コンパイル単元** (compilation unit) と呼ばれる。各コンパイル単元に対して、中味のシグネチャーだけを含んだインターフェースファイルを用意することで、コンパイル単元同士の直接な依存関係が回避され、最適な分割コンパイルが可能になっている。コンパイル単元とそのインターフェースの組を不透明なシグネチャーを備えたモジュールとして捉えることができるので、この仕組みを言語の意味論の中で解釈することもできる。どの時点でどのファイルを再コンパイルするかはユーザ（あるいはビルドシステム）に任されているが、コンパイルされたファイルの中に依存する単元のシグネチャーのハッシュ値を含むことで一貫性が保証される。

残念ながら、この単純さには欠点がある。ファイルシステム上にコンパイル単元を探すときは名前しか考慮されないので、入れ子のないモジュール空間ができてしまう。特に、あるモジュールの中に同名のサブモジュールを二つ以上定義できないのと同様に、一つのプログラムの中に同名のコンパイル単元を二つ以上組み込むことができない。複数のライブラリーを利用するプログラムに取って、大きな枷になる。

Objective Caml 3.05 以降、複数のコンパイル単元を集約単元 (packed unit) のサブモジュールとしてまとめることで、この問題を回避できるようになった。しかし、この方法では、集約単元が元のコンパイル単元に置き換わる形になり、全てのサブモジュールのコードが含まれる。一つのサブモジュールを使うために、集約単元全体をプログラムに組み込まなければならず、プログラムが大きくなる。この意味では、集約単元は分割コンパイルに逆行していると言える。

Type-level module aliases: independent and equal†1

Jacques Garrigue, 名古屋大学多元数理科学研究科, Graduate School of Mathematics, Nagoya University.

†1 この論文は [4] の日本語訳に基づいている。

ただし、分割できないのはコンパイルではなく、集約後の単元である。

Standard ML の実装の SML/NJ では、全く異なる方法が選ばれた。コンパイルの実行およびライブラリーの名前空間は**コンパイル・マネジャー** [2] [1] というツールに任されている。コンパイル・マネジャーは専用の構文で書かれた特殊なファイルによって、モジュール名とライブラリー内の実装のマッピングやコンパイルされたライブラリーのファイルシステム上の位置を知る。この情報によって、再コンパイルを最適化できるだけでなく、モジュール名をライブラリー内に隠蔽することで名前の衝突も避けられる。後から外に見える名前を変えることも可能である。この豊かな表現力のお陰で、コンパイル・マネジャーはファイルシステムに基づいた方法の諸問題を回避できる。しかし、そのために ML の意味論に含まれない新しい言語が導入されている。

この二つの方法を見ると疑問を抱かざるを得ない。ML のモジュールシステムの表現力が非常に高いと言われているのに、なぜ実際のプログラムの名前空間の問題を解決するために別の機構を必要とするのか。

型レベル・エイリアスはシグネチャーの中にモジュールのエイリアスを宣言できるようにするモジュールシステムの小さな拡張である。それを使うと、分割コンパイルを失わない形で名前空間は自由にモジュールを使って構成できる。適用的ファンクターの等価性を強化するという別のメリットもある。型レベル・エイリアスは OCaml 4.02 以降のバージョンで使える。

2 等価性のためのエイリアス

型レベル・エイリアスは分割コンパイルと全く違う方面から現れた概念である。元々、中田と Garrigue の Traviata [8] において、再帰モジュールの型推論を可能にするための機構として導入された。後から、この機構が OCaml 流の適用型ファンクターの等価性の問題をいくつか解決できることが確認された [3]。次の例では、`Set.Make` を要素の比較関数をもらい、集合を表す抽象データ型 `t` とそれを扱うための関数を返すファンクターとする。

```
module StringSet1 = Set.Make(String)
module StringSet2 = Set.Make(String)
module S = String
module StringSet3 = Set.Make(S)
```

OCaml のファンクターは適用型なので、ファンクターの結果の中に現れる個性を持った (nominal な) 型の等価性はファンクターの引数の等価性によって決まる。具体的には、ここで `StringSet1` と `StringSet2` で同じ引数 `String` が適用されるので、`StringSet1.t` と `StringSet2.t` は等しくなる。しかし、`StringSet3` では引数が `S` なので、`StringSet1.t` と `StringSet3.t` が等しくならない。従来の OCaml では `S` と `String` が等しいを型レベルの情報として残すことができなかった。

型レベル・エイリアスがあると、モジュール定義 `module S = String` のシグネチャーは

```
module S = String
```

あるいは、単集合カインドを意識した記法で、

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

になり、`S` と `String` が型レベルでも等価になり、`StringSet1.t` と `StringSet3.t` が等しくなる。

それを可能にするために、型の等価性は型定義の展開だけでなく、型レベル・エイリアスを考慮したモジュール・パスの正規化も使う、より強い方法に変わる。

3 分割のためのエイリアス

この強化された等価性は役に立つものの、それをフルに活用するために再帰モジュールを含む型システムの大規模な変更が必要なので、OCaml への導入が見送られて来た。しかし、エイリアスにはもう一つのメリットがある。型によって与えられた情報はコードには要らないということに着目し、コンパイルされたコードの一部が削られる。それをすると、依存関係が少なくなり、分割コンパイルが強化される。この応用を主な目的とすれば、型システムの小さな変更でも十分対応できる。

次のコードをコンパイル単元 `Mylib` の中味とする。

```
module A = MylibA
module B = MylibB
```

A と B のエイリアスはシグネチャーに現れるので、Mylib のコンパイルした実装は MylibA と MylibB を参照しなくても良い。すなわち、Mylib 自身は MylibA と MylibB に依存しない。結果的には、Mylib.A を使って、Mylib.B を使わないようなプログラムは MylibA のみを組み込めばいい[†2]。集約単元が分割できない問題がこれで回避できる。さらに、ここでは、Mylib のコンパイルされた実装が空なので、こちらも組み込まれない。

シグネチャー間の依存関係を減らすこともできる。Mylib のシグネチャーの型をチェックするとき、MylibA と MylibB のシグネチャーを参照する必要はない。結果的には、Mylib のコンパイルされたシグネチャーは MylibA と MylibB と完全に独立している。後者が変わったときに Mylib を改めてチェックする必要がなく、Mylib を先にコンパイルすることさえ可能である。ライブラリーを階層化するときにこの特徴が役に立つ。

4 名前空間への応用

上記の説明から、既にモジュール名の衝突を避ける方法が分かる。ライブラリーを作るときに各コンパイル単元の名前にそのライブラリー固有の名前を含めておけばいい。利用をスムーズにするために、上の Mylib のように、各コンパイル単元の略称を与えるコンパイル単元を用意すればいい。そうすると、利用するソースの先頭に `open Mylib` という一行を加えるだけで、各モジュールが短い名前で参照できるようになる[†3]。

これで、利用側の問題が解決される。しかし、定義中のライブラリーの中でも、他の単元を短い名前で参照したい。ありがたいことに、Mylib のシグネチャーがエイリアス先の中味に依存しないので、MylibA と MylibB の中で `open Mylib` と書いてもいい。これによって依存関係に巡回が生じない。

まとめると、以下のレシピーを使えば集約単元をエイリアスを使った分割可能な構造に変えられる。

1. 各単元の名前をライブラリー名を含む長いものに変える。
2. 全てのコンパイル単元に対して短い名前を与えるインターフェイス用のコンパイル単元を作る。
3. このコンパイル単元を他の全ての単元で `open` する。
4. 見せたい単元のエイリアスだけを含めたエキスポート用のインターフェイス単元も作る。(隠すものがなければ、(2) と同じもので良い)

集約単元の作り方に近づけるために、OCaml 4.02 ではコンパイラが新しい `-open` オプションを受け付けるようになった。それを使えば、(3) が不要になる。

インターフェイス用のコンパイル単元は通常のコンパイル単元である。マッピングを与える以上のことをしても良い。例えば、エキスポートされたインターフェイスに入れ子のサブモジュールを使い、より深い階層化を行う。または、同じエイリアスを何度も使い、複数のビューを与えることもできる。型レベル・エイリアスは単なる論理的なポインターでしかないので、コストが生じない。

以前から、Jane Street 社が開発している汎用ライブラリー Core/Async [5] ではモジュールによる名前空間の構成法が使われて来た。元々、集約単元で名前の衝突を防ぎ、階層化で機能を整理していた。しかし、この方法では多くのコンパイル単元のシグネチャーと実行コードが一つの集約単元にまとめられるため、リンクされたプログラムが大きくなる上、コンパイル時間も問題になるほど長くなっていた。コードを全く変えずに、OCaml 4.02 に切り替えるだけで、各ライブラリーのシグネチャーのサイズの合計を 1/3 に減らすことができ、その分コンパイル時間も早くなる。さらに、上に書いたような手法がコードベース全体に適用された。そうすると、リンクされたプログラムの大きさは合計で半分になった [7] [9]。(短いプログラムでは 1/10 の場合もあった)

†2 厳密には、依存関係をなくすと MylibB が副作用を含んだ場合にプログラムの動作が変わる。そのせいで、依存関係をなくす機能は `-no-alias-deps` オプションを指定したときにのみ有効である。

†3 Standard ML と異なり、OCaml では `open` がモジュールの中味をローカルな環境に加えるだけで、定義中のモジュールのシグネチャーには加えない。

5 制限と今後の課題

型レベル・エイリアスの導入は型と実行コードの両方にまたがるため、かなり大きな拡張である。それを最低限の規模に抑えるために、OCaml 4.02 における型レベル・エイリアスの実装は型レベルに反影されるモジュール・エイリアスに多くの制限を加えている。具体的には、以下のモジュール式へのエイリアスは型レベルに反影されない。

- モジュールパス以外の式 (ストラクチャーおよびファンクター)
- ファンクターの適用を含むパス
- 不透明な型注釈
- ファンクターの引数として導入されたモジュール変数を頭とするパス
- 定義中の再帰モジュールを頭とするパス

一つ目の制限は本質的である。それを許せば言語全体をシグネチャーの中に書くことになる。しかし、他の制限は型検証アルゴリズムの強化で解除できるはずである。それを行えば、この機能はよりエレガントなものになるだろう。

謝辞

分割コンパイルへの応用について、ケンブリッジ大学の Leo White に多くのアイデアをいただいた。また、実験とデバッグに協力してくれた Jane Street 社の社員達に感謝する。

参 考 文 献

[1] Blume, M.: *CM: The SML/NJ Compilation and Library Manager*, Lucent Technologies, Bell Labs, May 2002. `http://www.smlnj.org/doc/CM/new.pdf`.

[2] Blume, M. and Appel, A. W.: Hierarchical modularity, *ACM Transactions on Programming Languages and Systems*, Vol. 21, No. 4(1999), pp. 813–847.

[3] Garrigue, J. and Nakata, K.: Path resolution for recursive nested modules, *Higher-Order and Symbolic Computation*, Vol. 24(2012), pp. 207–237.

[4] Garrigue, J. and White, L.: Type-level module aliases: independent and equal, *ACM SIGPLAN ML Family Workshop*, September 2014.

[5] Jane Street: Open Source software. `http://janestreet.github.io/`.

[6] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., and Vouillon, J.: *The OCaml system release 4.01, Documentation and user's manual*, Projet Gallium, INRIA, September 2013.

[7] Minsky, Y.: Better namespaces through module aliases, May 2014. `https://blogs.janestreet.com/better-namespaces-through-module-aliases/`.

[8] Nakata, K. and Garrigue, J.: Recursive modules for programming, *Proc. International Conference on Functional Programming*, Portland, Oregon, 2006.

[9] Shinwell, M. and Chapman, N.: Personal communication, May 2014.