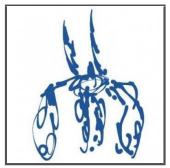
Dhall: Haskell の新たなキラーアプリ



@syocy

2018-11-10

設定ファイルとは

- コンパイルなどの前処理をせずにプログラムのパラメータを変えたい
- → パラメータをソースコードの外に切り離したい
- → **設定ファイル**

既存の設定ファイル言語への不満

- 現在主流の設定ファイル言語は機能が少ない
  - 同じ値を繰り返し書きたくない (Don't Repeat Yourself)
  - 入ってはいけない変な値が入っていたら教えてほしい (静的検査, 型)
  - 大きい設定ファイルを分割したい (インポート)
- ただし、設定ファイルとしての領分は守ってほしい
  - 副作用を持っていて動作が不定だったり、
  - 無限ループを起こしてハングしたりはしないでほしい

なにかいいものはないのか

Dhall

Dhall とは

- 今まで挙げた特徴を備えた設定ファイル言語
  - 型・関数・インポートの機能がある
  - 無限ループは起こらない（チューリング完全ではない）
  - 副作用は（標準では）ない
- 読みは dɔl (US) もしくは dɔːl (UK)
  - カタカナでは「ダール」もしくは「ドール」?
  - このスライドでは「ダール」と発音する

Dhall の開発体制

- 言語仕様は独立して管理されている
  - https://github.com/dhall-lang/dhall-lang
- 主要ツール、および言語仕様の参照実装は Haskell で書かれている
  - https://github.com/dhall-lang/dhall-haskell

Dhall の型: プリミティブな型

型	意味	リテラル (例)
`Bool`	真偽値	`True`, `False`
`Natural`	0 以上の整数	`0`, `1`
`Integer`	符号付き整数	`-1`, `+1`
`Double`	小数点付き数	`0.1`, `-2e10`
`Text`	文字列	`""`, `"abc"`

- プリミティブ型には一般的なものが揃っている
- 数値の型はできるだけ `Natural` を使うのがおすすめ
  - 使える標準関数が多いため

Dhall の型: 複合型

型	意味	値の例
`List`	0 個以上の値の集まり	`[1,2,3]`
`Optional`	0〜1 個の値の集まり	`Some 1`
Record	キーと値のペアの集まり	`a=1, b=2`
Union	「どれかひとつ」を表す	`<A={=}|B:{}>`

- Record は JSON のオブジェクトに相当
- Union の書き方が特徴的
  - 上記の例では「A と B のラベルがある Union 型で A を選んだ値」となる
  - 記述をサポートする標準関数がある

インポート

ローカルパスと URL からのインポートができる

- ローカルパスからのインポート
  - `let config = ./my/local/config.dhall`
- URL からのインポート
  - `let config = https://example.com/config.dhall`
  - インポートにあたってハッシュ値のチェックをすることもできる
- Dhall ファイルではない raw text のインポートも可能
  - 長い自然言語文章などに

Dhall の導入は難しい?

- なるほど Dhall は便利かも
- だけど自分の使っている言語にバインディングがない[1]
- もう YAML や JSON で設定作ってしまっているし……

本当に導入は難しいのか

[1] 今のところ最新仕様に準拠したバインディングは Haskell のみ

Dhall の導入は簡単

- dhall-to-yaml/dhall-to-json [2] がある
  - Dhall ファイルを YAML や JSON に変換するコマンドラインツール
  - プログラムで Dhall を読み込むことを考えなくて良い
  - バインディングがない言語でも Dhall を使える
  - 導入は Mac, Linux はコマンド一発、Windows は The Haskell Tool Stack を入れる必要がある

[2] https://github.com/dhall-lang/dhall-lang/wiki/Getting-started%3A-Generate-JSON-or-YAML

例題: Kubernetes の YAML を書いてみる

- 最近話題の OSS, Kubernetes は YAML を大量に用いることで有名
  - Wall of YAML などとも呼ばれる
  - 実際さまざまな YAML 管理のソリューションが存在する
- 今回は `dhall-to-yaml` を用いて安全・便利に Kubernetes の YAML を生成してみる
  - 実はすでに dhall-kubernetes[3] というのがあり、本当に Kubernetes YAML を作るならそれを使う方がよい。今回はあくまで例題として DIY する。

[3] https://github.com/dhall-lang/dhall-kubernetes

目的とする YAML

service.yaml[4]

```yaml
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```

[4] https://kubernetes.io/docs/concepts/services-networking/service/

愚直な Dhall

k_notype.dhall

まず型とかは考えずに愚直に書いてみる

```
1  { kind =
2      "Service"
3  , apiVersion =
4      "v1"
5  , metadata =
6      { name = "my-service" }
7  , spec =
8      { selector =
9          { app = "MyApp" }
10     , ports =
11         [ { protocol = "TCP", port = 80, targetPort = 93
12     }
13 }
```

愚直な Dhall を YAML に変換

目的とする YAML ができた！

```
# cat k_notype.dhall | dhall-to-yaml
apiVersion: v1
kind: Service
spec:
  selector:
    app: MyApp
  ports:
  - targetPort: 9376
    protocol: TCP
    port: 80
metadata:
  name: my-service
```

より Dhall らしい方法

- 型などを意識しなくても目的とする YAML はできた
  - ユースケースによってはこれくらいでもまあ便利
- さらに Dhall の能力を引き出すなら、型やデフォルト値などを用意することができる
  - 基本的なアイデアとしては、
    - Union 型を用いて記述できる値を制限する
    - Record 型でデフォルト値を用意する

Kubernetes YAML の型を定義 (1/2)

k_types.dhall

Kubernetes YAML の型を定義するファイルを作る

```
1      let Kind_ = < Service : {} | Pod : {} | Deployment :
2
3  in  let ApiVersion = < v1 : {} >
4
5  in  let Metadata = { name : Text }
6
7  in  let Selector = { app : Text }
8
9  in  let Protocol = < TCP : {} | UDP : {} >
10
11 in  let Port = { protocol : Protocol, port : Natural, ta
12
13 in  let Spec = { selector : Selector, ports : List Port
```

Kubernetes YAML の型を定義 (2/2)

k_types.dhall

Union で表現した ApiVersion などを YAML の String に戻す処理も必要（実装は省略）

```
in  let makeYaml =
              λ ( args
                : { kind :
                      Kind_
                  , apiVersion :
                      ApiVersion
                  , metadata :
                      Metadata
                  , spec :
                      Spec
                  }
                )
```

定義した型を利用して書き直す

k_service.dhall

```
let k = ./k_types.dhall

in  k.makeYaml
    { kind =
        k.Kinds.Service {=}
    , apiVersion =
        k.ApiVersions.v1 {=}
    , metadata =
        { name = "my-service" }
    , spec =
        { selector =
            { app = "MyApp" }
        , ports =
            [ { protocol = k.Protocols.TCP {=}, port = 8
        }
    }
```

型付き版を YAML に変換

より安全に Kubernetes YAML が表現できた

```
# cat k_service.dhall | dhall-to-yaml
apiVersion: v1
kind: Service
spec:
  selector:
    app: MyApp
  ports:
  - targetPort: 9376
    protocol: TCP
    port: 80
metadata:
  name: my-service
```

デフォルト値 (1/2)

k_service_default.dhall

Kubernetes YAML のデフォルト値を作ってみる

```
1       let k = ./k_types.dhall
2
3   in  let defaultService =
4             { kind = k.Kinds.Service {=}, apiVersion
                 = k.ApiVersions.v1 {=} }
5
6   in  let tcp = { protocol = k.Protocols.TCP {=} }
```

デフォルト値を用意すると、同じような値をたくさん作るときに間違いをしにくくなる（今回例示するのは１つ）

デフォルト値 (2/2)

k_service_default.dhall

Record は演算子 ∧ で組み合わせることができる

```
in  k.makeYaml
    (   defaultService
      ∧ { metadata =
            { name = "my-service-1" }
        , spec =
            { selector =
                { app = "MyApp1" }
            , ports =
                [ tcp ∧ { port = 80, targetPort = 9376
            }
        }
    )
```

デフォルト値版を YAML に変換

もちろんこれまでと同じく YAML に変換できる

```
# cat k_service_default.dhall | dhall-to-yaml
apiVersion: v1
kind: Service
spec:
  selector:
    app: MyApp1
  ports:
  - targetPort: 9376
    protocol: TCP
    port: 80
metadata:
  name: my-service-1
```

まとめ

- Dhall は**設定ファイルとしての限界**を突き詰めた言語
  - インポート・型・関数などの機能を持ちながら、
  - 副作用や無限ループの危険がない
- YAML/JSON **に変換できる**ため、既存資産に組み入れやすい
  - Mac, Linux にはバイナリ配布があり導入もしやすい
- **ユースケースに合わせたレベル**で利用できる
  - インポートとデフォルト値があればいい
  - ばりばり厳密に型を定義したい、など

補遺: より高度な使い方