1. はじめに

この文書は、https://github.com/takoeight0821/kagami/pull/24 がマージされた後の Malgo について解説します。必要なら適時、「Malgo」を「Griff」に、「.mlg」を「.grf」に読み替えてください。

2. プログラミング言語 Malgo の解説

Malgo は、多相型、型推論、カリー化、パターンマッチなどの機能を持つ静的型付き関数プログラミング言語である。以下に示すのは、標準出力に `Hello, world!` と出力するプログラムだ。

```
module Hello = {
  import Builtin;

  foreign import print_string :: String# -> ();
  foreign import newline :: () -> ();

  putStrLn :: String -> ();
  putStrLn = { (String# str) ->
    print_string str;
    newline ()
  };

  main = {
    putStrLn (String# "Hello, world!"#)
  };
}
```

Malgo プログラムは、モジュールと呼ばれる翻訳単位から構成される。モジュールは `module モジュール名 = { ... }` で宣言される。現在の Malgo 処理系には、1 ファイルにつき 1 つしかモジュールを宣言できない制約が存在するが、改良を予定している。

他のモジュールで定義された型や関数を利用するには、`import` 宣言を用いる。`import モジュール名 ;` と書くと、そのモジュールで宣言された型や関数が参照できるようになる。上記の例では、`Builtin` モジュールで定義される `String` 型を参照している。

Malgo は、C などの他の言語で書かれた関数を利用する FFI 機能も備えている。外部のオブジェクトファイルで定義された関数を利用するには、`foreign import 外部関数名 :: 型 ;` と書く。上記の例では、`Unit* print_string(char*)` と `Unit* newline(Unit*)` をインポートしている。

値には、boxed と unboxed の区別がある。 型の末尾に `#` がつく値は unboxed な値であり、32bit 符号付き整数や 64bit 浮動小数点数などの即値を表す。 末尾に `#` がつく文字列リテラルや数値リテラルは、 それぞれ対応する unboxed な型の値である。boxed な値は， 内部的にはポインタとして実装されており、オーバーヘッドを含む。 代数的データ型の値などはすべて boxed な値である。

多相関数の引数に適用できるのは boxed な値のみである。例えば、`id = { x -> x };` がある時、`id 1` は `1` を返す。（正確には `int32# 1#`）しかし、`id 1#` は以下のようなエラーを返す。

```
malgoc: error on ErrorKind.mlg:3:12:
  Kind mismatch: ''14':Boxed
                 Int32#:Int32Rep
3   main = { id 42# };
```

これは、`id :: a -> a` の `a` に相当する型変数 `'14` のカインドは Boxed だが、`42# :: Int32#` のカインドは Int32Rep である、というカインドエラーだ。カインドは「型の型」だと思っていい。

関数は `{引数 1 引数 2 ... -> 式}` と記述する。Malgo の関数は、ML 系言語や Haskell で見られる「パターンマッチ」 の構文をかねており， 以下のような文法で「引数の形に基づいて処理を分岐する」関数を記述できる。

```
map = { f Nil -> Nil
      | f (Cons x xs) -> Cons (f x) (map f xs)
      };
```

Malgo はデフォルトで関数をカリー化する。つまり、関数 `f = { x y -> e }` が定義されてい

る時、`f a` を実行すると関数 `{ y -> e[a/x] }` が返されるように振る舞う。（`e[a/x]` は、`e` 内の自由変数 `x` の出現を、式 `a` を評価した値で置き換えるという意味）

Malgo はデフォルトで関数をカリー化する。つまり、関数 `f = { x y -> e }` が定義されている時、`f a` を実行すると関数 `{ y -> e[a/x] }` が返されるように振る舞う。（`e[a/x]` は、`e` 内の自由変数 `x` の出現を、式 `a` を評価した値で置き換えるという意味）

引数がない関数を定義することもできる。 例えば、`{ print_string "hello"# }` と書くと、`{ () }` 型を持つ 0 引数関数が定義される。0 引数関数は lazy value とも呼び、名前の通り遅延評価される式を書きたい時に用いる。評価するには前置演算子 `!` を用いる。条件分岐を表す `if` はこの機能を使って実装されている。

```
if :: Bool -> {a} -> {a} -> a;
if = { True t _ -> !t
     | False _ f -> !f };
```

関数定義の波括弧 `{}` の中で、複数の式を `;` で区切って書くと、それらの式は順番に実行され、関数の戻り値は最後に実行された式の値になる。また、`let x = f 10#;` のように書くと、ローカル変数 `x` を定義できる。

```
main :: {()}
main = {
  let x = Int32# (add_Int32# 1# 2#);
  let printInt32 = {(Int32# x) -> print_int32 x};
  printInt32 x;
}
```