

Functional Scripting

汎用的関数型言語における系統的な外部リソース操作の原理と実装

大和谷 潔:学生番号 010119

2002年 2月 18日

目的

- 関数型言語の信頼性
 - スクリプト型言語の開放性
- } を兼ね備えたプログラミング言語の開発

背景

- コンポーネント指向
- 現状のスク립ト型言語に欠ける信頼性
- 現状の関数型言語に欠ける開放性

コンピュータネットワーク指向

- ⇒ コンピュータネットワークを組み合わせたアプリケーション構成手法が普及。
- ⇒ コンピュータネットワークを結び付けるスク립ティング言語が登場。
- ⇒ 現状では、ad hocなスク립ティング言語が普及している。



コンピュータネットワークと言語の関連を科学的に検討する必要がある。

現状のスク립ティング言語

記述力 Syntax sugar、変数宣言不要

開放性 コンポーネントを取り込む機構

しかし、

⇒ 安全性に欠ける

現状の関数型言語

記述力 型宣言が不要、高階関数、多相型関数

安全性 コンパイル時の型チェック

しかし、

⇒ 開放性に欠ける

アプローチ

関数型言語と外部ライブラリとの連携を強化

1, 型システム

オブジェクト指向など他の型システムの表現

2, 言語設計

関数型スクリプトインク言語 Amethyst

3, 処理系の実装

外部ライブラリに対応して動的に調整可能な機構

1,型システム

要求：オブジェクト指向クラスライブラリとの連携が重要。

- オブジェクト指向はコンポーネントの基礎
- 豊富なクラスライブラリ

問題：

1. オブジェクトの性質 (状態、実装の隠蔽) を表す型がない
2. オブジェクト指向の polymorphic 型システム

解決：

1. 外部オブジェクトを表す「オブジェクト型」を導入
2. オブジェクト指向でのサブタイプ関係を多相型レコード計算とオブジェクト型で表現

オブジェクト型

外部ライブラリが公開するオブジェクトを表現する型

$$(\alpha_1, \dots, \alpha_k) \quad T[\{lbl_1 : \tau_1, \dots, lbl_n : \tau_n\}]$$

オブジェクト型の性質：

- 直接生成できない。

× $Person[\{Name = "YAMATO"\}]$

- プリミティブ (外部関数) により生成

$genPerson : string \rightarrow Person[\{Name : string\}]$

- レコード型と同様にフィールド lbl_1, \dots, lbl_n をもつ。

型システムへのオブジェクト型の組み込み

問題：レコード型と同様にオブジェクト型をパターンマッチしたい。

解法：フィールドの有無によって型を分類する。

多相型レコード計算 [Ohori]:

共通するフィールドをもつ型の集合 (カインド) を導入

$$\text{kind} ::= \{ \text{lbl}_1 : \tau_1, \dots, \text{lbl}_n : \tau_n \}$$

例：

```
fun getName x = case x of {Name = n, ...} ⇒ n
```

```
getName :  $\forall(\alpha :: \{ \text{Name} : \beta \}, \beta). \alpha \rightarrow \beta$ 
```

Nameフィールドを持つすべての型を引数にとることができる。

同じフィールドを持つオブジェクト型とレコード型を同じカインドに。

$$\begin{array}{l} \{Name : string\} \\ Person[\{Name : string\}] \end{array} \in \{\{Name : string\}\}$$

例 : `getName` がレコードと `Person` の両方に適用できる。

```
getName :  $\forall (\alpha :: \{\{Name : \beta\}\}, \beta). \alpha \rightarrow \beta$ 
getName {Age = 30, Name = "YAMADA"}
getName (genPerson("TANAKA"))
```

オブジェクト指向的 polymorphism

オブジェクト指向型システムの特徴：

- クラスの継承関係から導出される型のサブタイプ関係
- サブタイプ関係をもとにした、制約された polymorphism

問題：オブジェクト指向的 polymorphism を、関数型言語的型システムで表現

解法：クラスとカイン드의類似に着目

型とカイン드의関係 \iff 型とクラスの関係

カインド間の包含関係 \iff クラス間の継承関係



レコードとオブジェクト型によってサブタイプ関係を表現

オブジェクト型

α *Obj* []

を用いる。

(値表現と無関係な α で、他の型システムの制約を表現。)

型が所属するすべてのクラスをレコードのフィールドで表現。

(OO) (ML)

$\text{class } A \iff \{A : \text{unit}\} \text{Obj}$

$\text{class } B \text{ inherits } A \iff \{A : \text{unit}, B : \text{unit}\} \text{Obj}$

サブタイプ関係で制約された OO-polymorphism を、カインドで制約された ML-polymorphism で表現。

(OO) $(\text{class } A) \rightarrow \text{int}$

(ML) $\forall \alpha :: \{A : \text{unit}, \dots\}. \alpha \text{Obj} \rightarrow \text{int}$

2,Amethyst 言語

Standard ML + external 宣言 + カイソド付多相型

ドメイン：ライブラリのカテゴリ

domain Java = imports "init" of "jni.so"

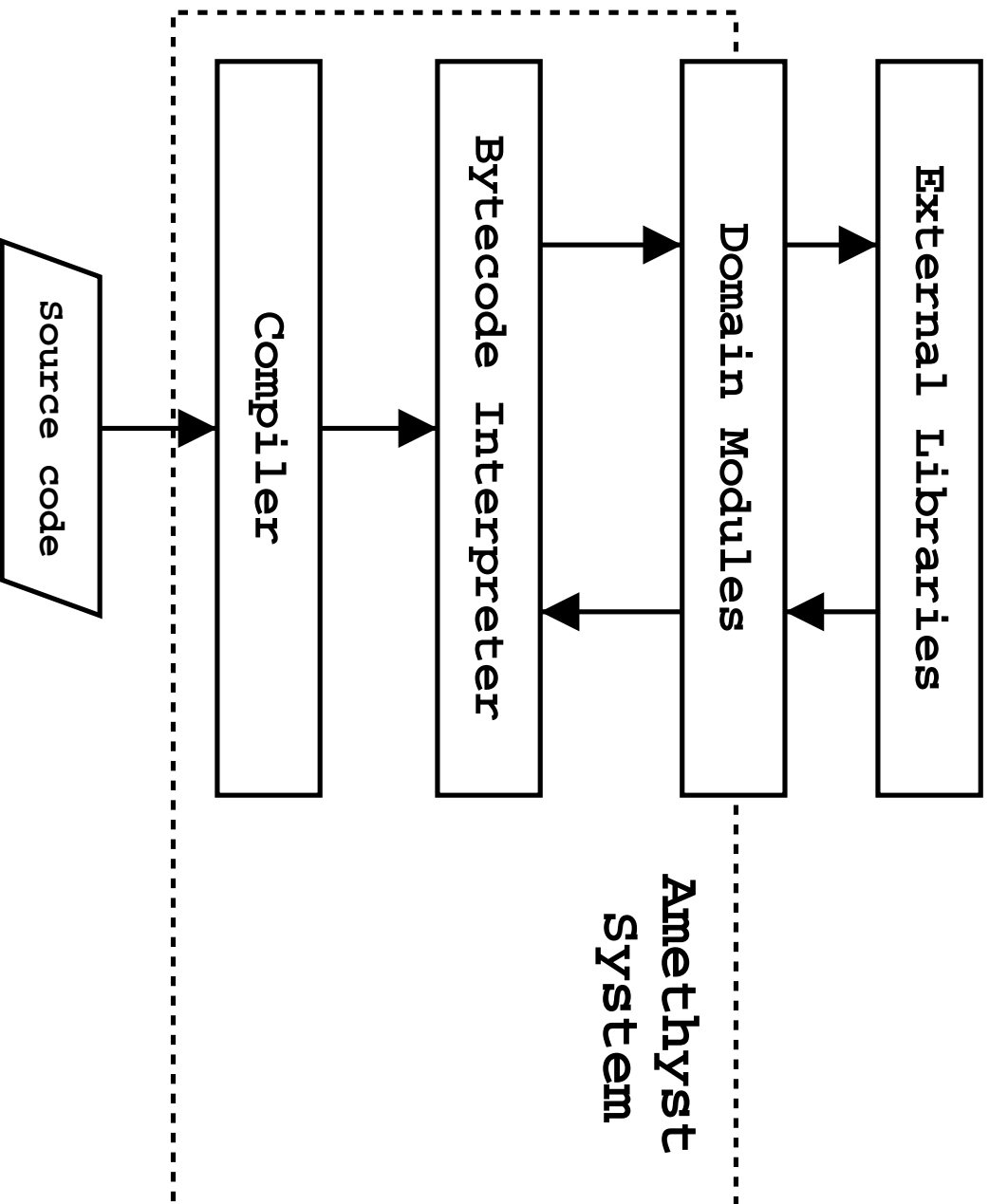
外部型：

```
external type Person = {Name:string, Age:int} imports "Person" of Java
```

外部変数：

```
external val genPerson : string -> Person imports "newPerson" of Java
```

3, 実装



コンパイラ

問題：

```
fun getName x = #Name x
```

- x の型が呼び出しごとに異なる。
- x からファイルを取得する方法が x の型に依存。
(レコード)

```
getName {Name="TANAKA", Salary=100}  
(Java オブジェクト)
```

```
getName genPerson("YAMATO")
```

- ファイルを取得す個所で、ファイルの取得し方法を知りたい。
- レコード/オブジェクトを生成する個所では、取得し方法が分かる。

解決：(多相型レコード計算を応用)
レコード/オブジェクトを生成する個所からフィールドを取出す個所へ
フィールド取出しを実行する関数(セクタ関数)を受け渡す。

例：

フィールドを取出す側：

```
fun getName x = #Name x  
⇒ fun getName S x = S(x)
```

レコード/オブジェクトを生成する側：

```
getName (genPerson("YAMATO"))  
⇒ getName Sel(Person, Name) (genPerson("YAMATO"))
```

ドメインモジュール

ドメインモジュールの役割： 外部ライブラリの物理的詳細を隠蔽

例：

- Java はオブジェクト指向。
- その外部インターフェイスである JNI は C 関数で構成。

↓

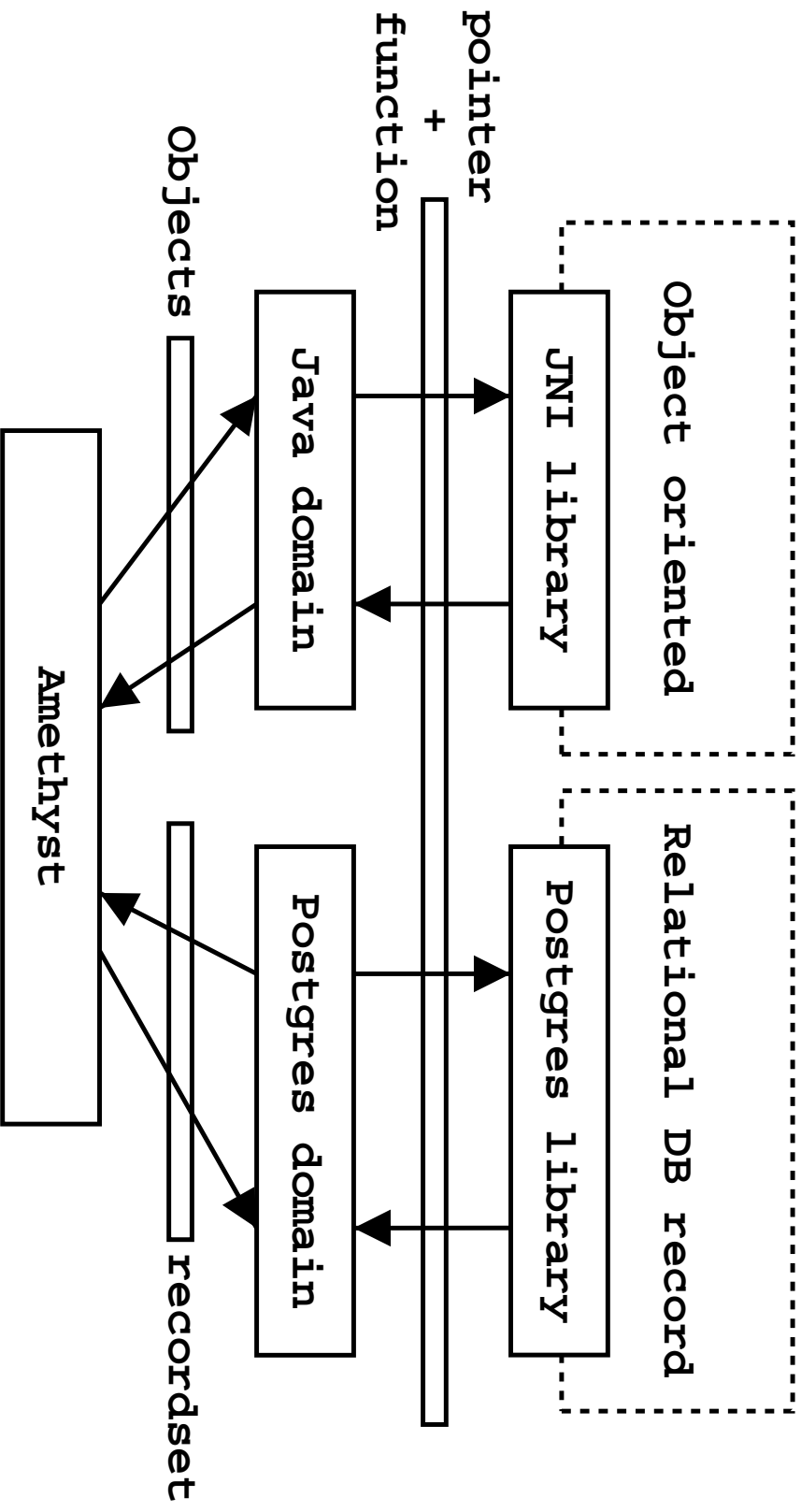
Java ドメインモジュールが

- 物理インターフェイス (C 関数) を隠蔽
- 本来の抽象モデル (オブジェクト指向) を再現

↓

Amethyst 上で

「オブジェクト指向」的に Java オブジェクトを操作



複数のドメインジョイントが共存可能

```
domain Java = imports "init" of "jni.so"
external Person = {Name:string, Age:int} imports "Person" of Java
external val genPerson : string -> Person imports "newPerson" of Java

domain postgres = imports "init" of "postgres.so"
external emprec = {Name:string "NAME", Age:int "AGE" } imports "" of postgres
external val getEmp : string -> emprec =
    imports "select NAME, AGE from EMP where @1" of postgres

fun getName r = #Name r

(getName {Name="YAMATO"},
 getName (genPerson("TANAKA")),
 getName (getEMP("SALARY > 1000")))
```

Amethyst システム

```
bash-2.04$ amethysti
>:load "pglib.ams"
domain postgres = imports "init" of pglib
:
>:load "pgtest.ams"
external type  emprec = {Name:string "S:NAME",Rank:int "I:RANK"}
                    imports "RECORD" of postgres
external fun  queryEmployee:connection -> string -> emprec =
:
    imports "query:select NAME, RANK from EMPLOYEE where @1" of postgres
:
val getNames = fn : forall ('a,'b:{Name:'a,...}) => 'b dbrec -> 'a list
>val c = open "localhost" "testdb" "kiyoshiy" "kiyoshiy"
        handle e as (PError msg) => (print msg;raise e);
val c = ??? : connection
>val emp = queryEmployee c "RANK = 1";
val emp = ??? : emprec
>empnames = getNames emp;
val empnames = ["Carlos Ghosn"] : string list
```

今後の課題

型システム

XMLなど非レコード的なデータモデルへの拡張

コンポーネント記述

コールバックのために必要(GUIのイベントハンドリングなど)