修 士 論 文

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

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

大和谷 潔

2002 年 3 月

修 士 論 文

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

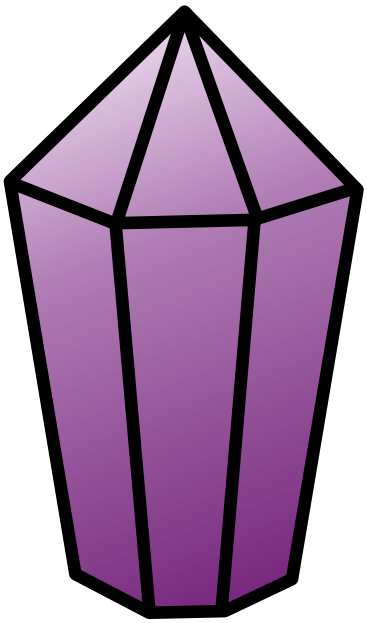
指導教官　大堀淳 教授

審査委員主査　大堀淳 教授
審査委員　田島敬史 助教授
審査委員　権藤克彦 助教授

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

010119 大和谷 潔

提出年月: 2002 年 2 月



目 次

第 1 章　序論 **5**
1.1　背景と目的 5
1.2　現状の問題点 6
1.2.1　有機体としてのオブジェクト 6
1.2.2　プロパティ 7
1.2.3　クラス階層にもとづく型システム 8
1.3　本研究の貢献 9
1.3.1　外部リソース操作のための型システムの構築 9
1.3.2　スクリプティング指向関数型言語 Amethyst の設計 9
1.3.3　実装 9

第 2 章　外部リソース操作のための型システム **11**
2.1　多相型レコード計算 11
2.1.1　概要 11
2.1.2　カインド付型システム 12
2.1.3　例 15
2.2　外部オブジェクトのための型システム 16
2.2.1　オブジェクト型 16
2.2.2　要件と解法 17
2.2.3　カインド付型システムの改良 18
2.3　カインド付型システムによるクラス階層の表現 20
2.3.1　クラスとカインド 21
2.3.2　仮想的オブジェクト指向言語 21
2.3.3　OL 型と ML 型の対応 22
2.3.4　安全性 22
2.3.5　例 24

第 3 章　スクリプティング指向関数型言語 Amethyst **25**
3.1　外部宣言 25
3.1.1　ドメイン宣言 25
3.1.2　外部型 25
3.1.3　レコード型の外部型 26
3.1.4　直和型の外部型 26
3.1.5　外部変数 27
3.2　外部型の仕様 27

3.2.1 レコード型の外部型 . . . 27
3.2.2 直和型の外部型 . . . 28
3.3 型式 . . . 28
3.3.1 カインドに制約される型変数 . . . 29
3.3.2 カインドに制約される quantification . . . 29
3.4 例 . . . 29

第 4 章 実装 **32**
4.1 実装の概要 . . . 32
4.1.1 問題点とその解決 . . . 32
4.1.2 システム構成 . . . 33
4.2 コンパイラ . . . 34
4.2.1 全体像 . . . 34
4.2.2 多相型レコード計算のコンパイル . . . 36
4.2.3 オブジェクト型への対応 . . . 42
4.2.4 直和型外部型のコンパイル . . . 45
4.3 ランタイム . . . 46
4.3.1 全体像 . . . 46
4.3.2 メモリ管理 . . . 48
4.3.3 仮想機械 . . . 50
4.4 ドメインモジュール . . . 52
4.4.1 記述言語の選択 . . . 52
4.4.2 ドメインモジュールのインターフェイス . . . 53
4.4.3 PostgreSQL ドメイン . . . 54
4.4.4 Java ドメイン . . . 64

第 5 章 結論と今後の課題 **76**
5.1 結論 . . . 76
5.2 今後の課題 . . . 76

付 録 A Amethyst 文法定義 **77**

付 録 B ZINC 抽象機械 **81**
B.1 ZINC 抽象機械の構成 . . . 81
B.2 コンパイル . . . 81
B.2.1 定数 . . . 83
B.2.2 ローカル変数 . . . 83
B.2.3 関数適用 . . . 83
B.2.4 関数抽象 . . . 85
B.2.5 `let` 式 . . . 86
B.2.6 プリミティブ関数 . . . 88
B.3 コンパイル／実行例 . . . 89
B.3.1 関数適用 . . . 89
B.3.2 部分適用 . . . 90

B.3.3 関数を返す関数 . . . 91
B.3.4 末尾呼び出しの最適化 . . . 91
B.3.5 `let` 式 . . . 92
B.3.6 再帰的定義 . . . 93
B.3.7 再帰関数の適用 . . . 94

付 録 C ユーザーズガイド **96**
C.1 Amethyst の入手 . . . 96
C.2 インストール . . . 96
C.3 コマンド . . . 96
C.3.1 基本コマンド . . . 96
C.3.2 ラッパーコマンド . . . 97

第1章　序論

1.1　背景と目的

産業の発達は分業の進展をともなう。ソフトウェア産業もその例外ではない。複雑さを増すソフトウェアシステムに対処するため、個別の機能に特化したソフトウェア部品、すなわちコンポーネントを組み合わせてアプリケーションを構築する手法が広まっている。Web アプリケーションや GUI フロントエンドでそれは顕著である。

アプリケーション構成方法の変化は、当然、プログラミング言語にも影響を及ぼす。Perl, Python, Ruby などいわゆる「スクリプティング言語」が存在感を増しているのはその結果だろう [Ous98]。これらの言語は、他言語で記述されたライブラリを取り込むインターフェースを備え、「開放性」に優れている。そのため、コンポーネントを結び付けて一つのシステムに組み立てる「アプリケーション記述言語」として用いられている。

しかし、元来これらの言語は小規模なプログラムの記述を用途に想定して生みだされたものであり、適用範囲が広がるにつれて問題点が顕在しつつある。とくに、その優れた開放性の陰で、安全性が犠牲となっている。これらの言語は、ライブラリが提供する種々のデータモデルを区別無く扱うことができるが、それは、それらの差異をその型システムが判別し得ないことの副産物であるとも言える。このため、型に関する整合性をプログラマが自力で検証せざるを得ず、ソフトウェアが複雑化 · 大規模化するにしたがって、型のミスマッチを見逃し実行時のシステム障害を引き起こす可能性が増す。

「開放性」と「安全性」は両立し得ないだろうか。

本研究は、関数型言語をスクリプティング言語としてとらえることでその解答を探る。近代的な関数型言語は静的な型システムを備えており、プログラムの安全性を高めている。また、変数の型の明示を不要としている点や、polymorphic な関数や高階関数を動的に組み合わせることで少数の関数から目的の関数を生成できる点など、既存のスクリプティング言語に劣らない高い記述力を誇っている。

しかし、「開放性」に関して既存の関数型言語処理系は弱い。スクリプティング言語を含め「高級言語」と呼ばれる言語は、ライブラリの低レベルな実装上の詳細をプログラマから隠蔽し、概念上のモデルに一致したプログラミングインターフェイスを提示すべきだろう。関数型言語のその強力な型システムが、ここではむしろ障害となっている。型システムが課す厳密な制約に従ったうえで、種々の言語および形式で記述された多様なデータモデルを表現することは困難である。このため現在の関数型言語では、外部ライブラリのごく低レベルなインターフェイスをそのままプログラマに露出するのみにとどまっている。ここには、関数型言語としての特性が生かされていない。

本研究は、関数型言語の「安全性」とスクリプティング言語の「開放性」を両立する理論と実装技術の構築を目的とする。

1.2 現状の問題点

「開放性」に関して関数型言語が抱える問題点を具体的にまとめておく。

「関数」のレベルでは、既存の関数型言語処理系の多くが、すでに他のスクリプティング言語と同等の開放性を達成している。SML や Haskell の主な処理系では、外部で定義された関数を呼び出したり、反対に関数型言語で定義した関数を外部から呼び出すことができる。

しかし、関数の単なる集合体としてライブラリをとらえる見方は、最近では適当ではない。とくにコンポーネントとして扱われることを想定したライブラリは、オブジェクト指向を基盤とする抽象度の高いモデルの上に構築されている。これらのライブラリを関数型言語に組み入れる際には、その背景にあるモデルを直感させるインターフェイスをプログラマに提示することが望ましい。けれども既存の関数型言語では、以下に述べる三点において適切なインタフェースを提示する手段がない。

1.2.1 有機体としてのオブジェクト

オブジェクト指向では、「オブジェクト」をフィールドの単純な集合を超えてそれ自体が identity をもつ存在としてとらえ、人間や組織のような有機体に比喩して論じることが多い。一方、既存の関数型言語では、外部データを opaque な「参照」の形でしか取り込めず、外部データの内部構造を外部関数を通して断片的に知ることしかできない。そのため、オブジェクト指向のもとで記述されたライブラリを関数型言語から扱う際に、有機体としてのオブジェクトの特質を反映したコードを記述することが難しい。たとえば Java Native Interface を通して Java オブジェクトを扱うことを考えてみる。つぎのような Java クラス `Person` を定義したとする。

```
class Person{
  public String Name;
  public int    Age;
  Person(String name,int age){ Name = name; Age = age; }
}
```

既存の関数型言語からクラス `Person` のインスタンスを扱う場合、JNI 関数を呼び出すライブラリが用意されていると仮定して、コードは次のようになるだろう。

```
val obj = JNI.NewObject ("Person", ("YAMATO", 36))
    :
val name = JNI.GetStringField ("Person:Name", obj)
val age = JNI.GetIntField ("Person:Age", obj)
val _ = print ("Name=" ^ name ^ ", Age=" ^ Int.toString age)
```

上記のコードからは、`obj` が指している実体が `Name` および `Age` というフィールドを持つオブジェクトであるという全体像が見えにくい。ML プログラマならば、次のように各フィールド値をレコードにコピーすることを考えるだろう。

```
fun PersonToRecord obj =
  in {Name=JNI.GetStringField ("Person:Name", obj),
      Age=JNI.GetIntField ("Person:Age", obj)}
  end
```

```
    :
val obj = JNI.NewObject ("Person", ("YAMATO", 36))
val objrec = PersonToRecord obj
    :
val _ = case objrec of
  {Name=name, Age=age} => print ("Name=" ^ name ^ ", Age=" ^ Int.toString age)
```

クラス Person のインスタンスが Name と Age の二つのフィールドを持つということが、objrec の構成に反映されている。

しかし、上記のコードで生成されるレコードは「死んだ」コピーに過ぎない。つまり、「参照」(obj) から得られた値によって生成されるレコード (objrec) は、生成された時点での実オブジェクトの「コピー」であり、それ以降は、実オブジェクトとの関連は絶たれている。そのため、コピー元である実オブジェクトの状態が変化しても、コピー先のレコードには最新状態が反映されない。関数型言語の長所のひとつは「状態」という概念を排除している点にある。したがって、このような問題は関数型言語の原理を汚すものとして考慮しない態度にも根拠はある。しかし、ML が ref を取り入れて汎用言語への一歩を踏み出したように、関数型言語と外部との interoperability を実現するためには原理と現実の要請との間で妥当な折り合いをつけてこの問題を解決する必要がある。

1.2.2 プロパティ

コンポーネント指向のプログラミング言語あるいはフレームワークでは、「プロパティ」あるいは「アクセサ」と呼ばれる機構を用いて、オブジェクトの実装詳細を隠蔽しながら、概念上のモデルに沿った形でオブジェクトの「状態」を公開する手法を採用している。

たとえば COM では、propget および propput という属性を設定されたメソッドは、オブジェクトを構成する「仮想的」なフィールドに対する値の取得および値の設定をおこなうものと認識される。「仮想的」とは、プロパティメソッドに対応するフィールドをオブジェクトが実際には持たない可能性もあることを指している。同様に Java のコンポーネントフレームワークである JavaBeans でも、名前が set で始まるメソッドと get で始まるメソッドとの組を、「仮想的」なフィールドへのアクセサメソッドとして扱う。

これらは物理的には関数であり、プロパティへのアクセスはメソッドに対する呼び出しとして実装される。しかし、概念上はオブジェクトのフィールドに対する直接のアクセスのようにとらえられる。

このような規約に準拠したプログラミング言語、たとえば C#や Visual Basic では、通常のフィールドと同様の構文でこれらのプロパティにアクセスできる。プログラマは、それがオブジェクトのフィールドを直接アクセスするのか、メソッド呼び出しを通じてアクセスするのかを意識する必要がない。たとえばつぎの C#で記述されたクラス Person では Age フィールドを直接公開する一方で、Name プロパティを通して name フィールドを公開している。

```
class Person {
  public Person(string name, int age){
    this.name = name;
    this.Age = age;
  }
  private string name;
  public int Age;
```

```
  public string Name{
    get { return this.name; }
  }
}
```

この Person クラスを使用する側は、以下のコードのように、Age と Name の値を同様の構文で取得することができる。

```
  Person p = new Person("YAMATODANI", 36);
  Console.WriteLine("Name=" + p.Name + ",Age=" + p.Age);
```

しかし、既存の関数型言語処理系では、プロパティをオブジェクトのフィールドであるかのように扱う手段がない。したがって、次のようにフィールドとプロパティの違いを意識しなければならない。さきほどの JNI と同様に、.NET コンポーネントへのアクセス手段を提供するライブラリが用意されていると仮定する。

```
val obj = NET.NewObject ("Person", ("YAMATO", 36))
  :
val name = NET.GetStringProperty("Person:Name",obj)
val age = NET.GetIntField ("Person:Age", obj)
val _ = print ("Name=" ^ name ^ ", Age=" ^ Int.toString (age))
```

1.2.3 クラス階層にもとづく型システム

関数型言語のひとつの特徴は、その強力な型システムである。一方、異なる言語で記述された外部ライブラリは、それぞれの記述言語の型システムに従った形でインターフェイスを公開し、クライアントがこの型システムから導き出される制約を守ってライブラリを使用することを前提としている。したがって、関数型言語からそれらのライブラリを使用する際には、ライブラリ記述言語の型システムが前提とする条件が守られることを関数型言語の型システムによって保証しなければならない。逆に、ライブラリに対して本来可能である操作を、関数型言語の型システムが排除してしまうことは望ましくない。

ライブラリが C 言語のような単純な型のみをもつ言語で記述されている場合、問題は比較的容易である。しかし、クラスに基礎を置くオブジェクト指向的型システムにしたがって記述されたクラスライブラリを対象とする場合、それは簡単ではない。

オブジェクト指向と関数型言語との統合を目指した既存の研究では、つぎのいずれかの手法をとっている。

第一の方法は、クラスやメンバーアクセスを表現する構文および OO 型システムを関数型言語に加える。例として、[BK99] がクラスを基礎に置く型システムをそのまま ML に導入して Java との interoperability を実現している。しかしこの方法では、ラムダ計算とオブジェクト指向の二つの体系を対等に擦り合わせるために複雑な理論的後付けが必要であり、関数型言語としての不自然さは否めない。

第二の方法は、既存の構文および型システムを用いて、OO 型システムを模倣する。例として、[FLMJ99a] が COM インターフェイスの継承関係をファントムタイプを用いて表現している。ここで提示された手法は ML を含め他の関数型言語でもそのまま採用できるが、multi-inheritance に対応できない。この手法を発展させ、[MF01] では Java のクラスおよびインターフェイスの継承関係を Haskell の型システムで表現している。Java では multi-inheritance が認められているが、[MF01] は Haskell 特有の type class を用いて multi-inheritance に対処している。しかし、type class はそれ自体複雑な機構であり、他の言語処理系に同様に導入するのは難しい。ML を含めた関数型言語に広く適用できる汎用的な方法は、未だ見いだされていない。

1.3 本研究の貢献

以上で挙げた具体的な問題点に対し、本研究は以下に述べる解答を与える。

1.3.1 外部リソース操作のための型システムの構築

Standard ML の型システムに「自然」な拡張を加えることにより、外部ライブラリとの連携に関して前節で挙げた型理論的な問題をつぎのように解決した (2 章)。

データの物理的な構造を隠蔽しながら、それに対するパターンマッチを可能とする「オブジェクト型」を導入した。そして、[Oho95] による多相型レコード計算をもとに、オブジェクト型を導入した型システムを構築した。外部ライブラリが提供するオブジェクトあるいはコンポーネントをオブジェクト型の値として表現することで、オブジェクトあるいはコンポーネントとしての特質を損なうことなく関数型言語上でそれらを扱えるようになる。ソースコード上での「外観」、すなわちその値に対してどのようなパターンマッチが可能であるかということと、値の物理表現とを分離するアイデアは、[Wad87], [Oka98] などにも見える。

つぎに、オブジェクト型と多相型レコード計算とを応用して、オブジェクト指向言語にみられるクラス間の階層関係を関数型言語の型システムで表現する手法を開発した。過去の研究は、オブジェクト指向と関数型言語との「対等」な統合を試みている。この方法では、ユーザはオブジェクト指向的型システムと関数型言語的型システムの両方を常に意識させられる。一方、本研究による手法は関数型言語としての自然な拡張により実現するもので、オブジェクト指向的な概念は関数型言語の体系に自然に吸収される。

1.3.2 スクリプティング指向関数型言語 Amethyst の設計

上記の理論的基礎を現実のプログラミング言語として具体化するため、ML-like な言語 Amethyst[1] を設計した (3 章)。

Amethyst は、Standard ML の Core syntax をベースとし、オブジェクト型宣言文を始め外部リソースの使用を宣言する構文をいくつか追加している。これらの構文が導入する外部リソースに対する操作は、2 章で示す型システムによりチェックされその安全性が保証される。

1.3.3 実装

最後に、Amethyst の処理系を実装し、以上の研究成果の実用性を実証した (4 章)。

従来の関数型言語の処理系は、外部との interoperability を重視せず閉じた系を前提としている。対照的に本研究では、外部ライブラリとの連携実現を主要な目的として Amethyst 処理系を設計した。とくに、外部ライブラリが意図する概念的なモデルを損なうことなく関数型言語に取り入れることを目指している。既存の関数型言語処理系の FFI(Foreign Function Interface) は、外部ライブラリの物理的なインターフェイスをそのままプログラマに提示する。しかし、その物理的インターフェイスは実装上の制約を受けて設計されたもので、必ずしもライブラリが本来意図するモデルを反映していない。たとえば、JNI(Java native interface) ライブラリを利用すると、Java クラスライブラリを外部から操作できる。物理的には、これは一連の C 関数の呼び出しによって実現される。既存の処理系は、これらの C 関数をほぼそのままの形でプログラマに提示し、プログラマがこれらの関数を直接使用することを求める。つまり、Java プログラミング

[1] Perl(真珠),Ruby(ルビー) に続く第三のスクリプティング言語を目指して、Amethyst(紫水晶) と命名した。Pearl は 6 月、Ruby は 7 月、Amethyst は 2 月のそれぞれ誕生石である。

ではなく、C プログラミングをプログラマに強要する。しかしプログラマの意識の上にあるのは、Java オブジェクトのメソッド呼び出しや、フィールドへのアクセスである。Amethyst では、バイトコードインタプリタと外部ライブラリとの間に、ドメインモジュールと呼ぶプラグイン可能なレイヤを設けている。このレイヤは、ライブラリの物理的詳細を隠蔽し、ライブラリが本来意図する抽象的インターフェイスを再構築する役割を果たす。このようなシステム構成と、上述の型理論上の成果を組み合わせることによって、Amethyst はライブラリが意図するモデルに沿ったインターフェイスをプログラマに提示することを可能とする。

この基本設計のもとに、コンパイラ、バイトコードインタプリタ、ドメインモジュールの各サブシステムを実装した。コンパイラは、外部型を組み込んだ型推論機構と、外部型の値を操作するソースコードを外部関数呼び出しをおこなうバイトコードへ変換するコンパイルとを、[Oho95] で示された手法を応用して実装した。バイトコードインタプリタは ZINC 抽象機械 [Ler90] をベースとし、外部データおよび外部関数に対応して抽象機械命令の追加、変更を加えた。また、ガーベジコレクション機構に外部データへの対応を組み込んだヒープ管理方式を実装した。

最後に、本研究の成果が現実のアプリケーション構築に対して適用可能であることを示すため、PostgreSQL データベースとのインターフェイスを提供するドメインモジュールと、Java クラスライブラリを操作可能とするドメインモジュールを実装した。

第2章　外部リソース操作のための型システム

本章では、オブジェクト指向的なデータモデルおよび型システムのもとで記述された外部ライブラリを関数型言語プログラムで扱うための型理論的な基盤を構築する。本章で述べる内容は [Oho95] が示す多相型レコード計算の応用である。したがって、まず多相型レコード計算の型システムについて概要を述べる。つぎに、外部データを表現する「オブジェクト型」を導入する。オブジェクト型によって、1.2.1 節、1.2.2 節で述べた「オブジェクト」としての特質を維持したまま外部オブジェクトを関数型言語から操作することが可能となる。最後に、クラスベースの型システムが課す制約を、オブジェクト型と多相型レコード計算を組み合わせることによって関数型言語の型システム上で表現する方法を述べる。

2.1　多相型レコード計算

2.1.1　概要

多相型レコード計算を導入するそもそもの動機は、つぎのようなプログラムを適切に型づけすることにある。

```
fun getName r = case r of {Name=n,...} => n
getName{Age=30, Name="YAMADA"}
getName{Name="TANAKA", ID=19}
```

現在の SML はこのコードを受け付けない。一行目の `getName` の定義を SML はエラーとする。

```
- fun getName r = case r of {Name=n,...} => n;
stdIn:9.1-9.24 Error: unresolved flex record
   (can't tell what fields there are besides #Name)
```

ところが、つぎのように `r` の型を指定すると SML は `getName` の定義を受け付ける。

```
- fun getName (r:{Name:string, Age:int}) = case r of {Name=n,...} => n;
val getName = fn : {Age:int, Name:string} -> string
- getName ({Age=30, Name="YAMADA"});
val it = "YAMADA" : string
```

けれども当然、このように定義された `getName` は {`Name="TANAKA", ID=19`} に適用することはできない。

SML では、つぎのように定義される関数には polymorphic な型が与えられる。

```
fun id x = x
```

関数 `id` は、その本体中で引数 `x` に対していかなる操作もおこなっていない。したがって、この関数はあらゆる型の式を引数にとりうる。これを反映し、ML は `id` に型 $\forall\alpha.\alpha \to \alpha$ を与える。ところが `getName` の

場合、引数に対し Name フィールドを取り出すという操作を加えている。したがって、すべての型の式に適用できるものではない。たとえば getName true や getName {ID=10119, Salary=1000} などの式はコンパイル時にエラーとすべきである。つまり、getName には polymorphic な型を与えられるべきだが、その polymorphism は Name フィールドをもつ型のみを引数にとり得るように限定されるべきである。しかし、現在の SML は限定された polymorphism という概念を持たない。よって引数に関して monomorphic な型しか getName に与えることができない。

polymorphism の源泉は、型変数のもつ、**いずれの**型も代入することができるという性質である。したがって、限定された polymorphism を getName に対して与えるためには、型変数に代入可能な型を何らかの形で制約する手段を導入する必要がある。getName の場合、適用されるべき引数に共通するのは、Name フィールドを持つという性質である。そこで、フィールドの有無を基準に、型変数に代入できる型を限定することを考える。つまり、getName の型を

$$\forall(\alpha :: \{\!\{\mathtt{Name} : \beta\}\!\}, \beta).\alpha \to \beta$$

と表現する。これは、getName が、ラベルを Name とするフィールドをもついずれの型をも引数にとりうることを意味している。型変数 α を修飾している $\{\!\{\mathtt{Name} : \beta\}\!\}$ は、α に代入されうる型を限定する役割を果たしている。「型」という概念が、変数に束縛しうる値の集合を定義しているのと同様に、これは型変数に代入しうる型の集合を定義している。つまり、型の上位レベルの概念である。これを**カインド**と呼ぶ。

2.1.2 カインド付型システム

カインド付型システムの定義を以下に述べる。

議論の対象としてつぎのように式およびパターンを定義しておく。

$$\begin{array}{rcl}
e & ::= & c^b \\
& | & x \\
& | & \lambda x.e \\
& | & e\ e \\
& | & \{lbl = e, \cdots, lbl = e\} \\
& | & \mathtt{case}\ e\ \mathtt{of}\ p\ \Rightarrow\ e \mid \cdots \mid p\ \Rightarrow\ e \\
& | & \mathtt{let\ val}\ x = e\ \mathtt{in}\ e\ \mathtt{end}
\end{array}$$

$$\begin{array}{rcl}
p & ::= & c^b \\
& | & x \\
& | & \{l : p, \cdots, l : p\} \\
& | & \{l : p, \cdots, l : p, \ldots\}
\end{array}$$

簡単のため、ML の datatype で宣言されるタグつき直和型は含めていない。

型およびカインドはつぎのように定義される。

$$\begin{array}{rcl}
\sigma & ::= & \tau \\
& | & \forall(\alpha :: k, \cdots, \alpha :: k).\tau
\end{array}$$

$$\begin{array}{rcl}\tau & ::= & b \\ & | & \alpha \\ & | & \{lbl : \tau, \cdots, lbl : \tau\}\end{array}$$

$$k \quad ::= \quad \{\!\{l : \tau, \cdots, l : \tau\}\!\}$$

カインド $\{\!\{l_1 : \tau_1, \cdots, l_n : \tau_n\}\!\}$ は、**少なくとも** l_1 から l_n までの n 個のフィールドを持つ型の集合を指す。とくに $\{\!\{\}\!\}$ はすべての型を要素とするカインドを表す。ただし、$\alpha :: \{\!\{\}\!\}$ を α と省略する場合がある。また、ラベルから型への部分関数を用いて、フィールドの有限集合を表す。たとえば F を $dom(F) = \{l_1, \cdots, l_n\}$ であり $F(l_1) = \tau_1, \cdots, F(l_n) = \tau_n$ であるようなラベルから型への関数とするとき、$\{l_1 : \tau_1, \cdots, l_n : \tau_n\}$ を $\{F\}$ と表記する。

変数を型に対応づける**型環境**を $\mathcal{T}$ で表記する。

$$\mathcal{T} \quad ::= \quad \{x : \tau, \cdots, x : \tau\}$$

また、$\mathcal{T}\{x : \tau\}$ を $\mathcal{T} \cup \{x : \tau\}$ の略記とする。

$\mathcal{T}_n \cdots \mathcal{T}_1$ は次のように定義される n 個の型環境の合成を指す。

$$\begin{array}{rcl}\mathcal{T}_n \cdots \mathcal{T}_1(x) & = & \texttt{if } x \in dom(\mathcal{T}_1) \texttt{ then } \mathcal{T}_1(x) \\ & & \texttt{else } \mathcal{T}_n \cdots \mathcal{T}_2(x)\end{array}$$

$\mathcal{K}$ で表記される**カインド環境**は、型変数をカインドに対応づける。

$$\begin{array}{rcl}\mathcal{K} & ::= & \emptyset \\ & | & \mathcal{K}\{\alpha : k\}\end{array}$$

$$\mathcal{K}(\alpha) \;=\; \texttt{if } \{\alpha : k\} \in \mathcal{K} \texttt{ then } k \texttt{ else } \{\!\{\}\!\}$$

カインド規則は、カインド環境 $\mathcal{K}$ のもとで型 τ がカインド k に属することを定義する。これを $\mathcal{K} \vdash \tau :: k$ と表記する。

$$\mathcal{K}\{\alpha :: k\} \vdash \alpha :: k$$

$$\mathcal{K} \vdash \tau :: \{\!\{\}\!\}$$

$$\mathcal{K} \vdash \{l_1 : \tau_1, \cdots, l_n : \tau_n\} :: \{\!\{l_1 : \tau_1, \cdots, l_n : \tau_n\}\!\}$$

$$\frac{\mathcal{K} \vdash \tau :: \{\!\{l_1 : \tau_1, \cdots, l_n : \tau_n\}\!\}}{\mathcal{K} \vdash \tau :: \{\!\{l_1 :: \tau_1, \cdots, l_m : \tau_m\}\!\}} \quad (m < n)$$

つぎに定義する FTV は、型 τ 中の自由型変数を返す。

$$\begin{array}{rcl}FTV(b) & = & \emptyset \\ FTV(\alpha) & = & \{\alpha\} \\ FTV(\tau_1 \to \tau_2) & = & FTV(\tau_1) \cup FTV(\tau_2) \\ FTV(\{l_1 : \tau_1, \cdots, l_n : \tau_n\}) & = & \bigcup_{1 \le i \le n} FTV(\tau_i) \\ FTV(\forall(\alpha_1 :: k_1, \cdots, \alpha_n :: k_n).\tau) & = & ((\bigcup_{1 \le i \le n} FTV(k_i)) \cup FTV(\tau)) \backslash \{\alpha_1, \cdots, \alpha_n\}\end{array}$$

型に関して理論的に考察する際、自由型変数は、型が、その置かれた文脈からどのような影響を受けるか決定するための基準となる。カインド環境によって型が制約されている場合、型自体に現れる型変数を考慮するだけでは十分ではない。たとえば $FTV(\tau)$ が与える自由型変数 α が、カインド環境を通して他の型変数と関係付けられている場合、つまり、つぎのようにカインド環境中で α を制約するカインド中に型変数 β が含まれている場合、

$$\mathcal{K}\{\alpha :: \{\!\{Name : \beta\}\!\}\}$$

τ はこの β を通して文脈の影響を受ける。つぎに定義する $EFTV$ は、型 τ の自由型変数と、カインド環境 $\mathcal{K}$ 中のカインド制約によってそれらと関連づけられる型変数の集合を返す。

$$\begin{array}{rcl}
EFTV(\mathcal{K},\tau) & = & \texttt{let fun}\ trace\ (\{\alpha\}\cup V,\ S)\ = \\
& & \quad \texttt{let val}\ V'\ =\ \texttt{case}\ \mathcal{K}(\alpha)\ \texttt{of} \\
& & \quad\quad \{\!\{l_1:\tau_1,\cdots,l_k:\tau_k\}\!\}\ \Rightarrow\ (FTV(\tau_1)\cup\cdots\cup FTV(\tau_k))\backslash S \\
& & \quad \texttt{in}\ trace\ (V'\cup V,\ \{\alpha\}\cup S)\ \texttt{end} \\
& & \ |\ trace\ (\emptyset,\ S)\ =\ S \\
& & \texttt{in}\ trace\ (FTV(\tau),\ \emptyset)\ \texttt{end}
\end{array}$$

また、$EFTV$ の定義を型環境 $\mathcal{T}$ に対して次のように拡張する。

$$EFTV(\mathcal{K},\mathcal{T}) \quad = \quad \bigcup\{EFTV(\mathcal{T}(x))|x\in dom(\mathcal{T})\}$$

$Cls(\mathcal{K},\mathcal{T},\tau)$ は、カインド環境 $\mathcal{K}$ および型環境 $\mathcal{T}$ のもとで型 τ を quantify した型と、quantify された型変数を $\mathcal{K}$ から取り除いて得られるカインド環境の組を返す。

$$\begin{array}{rcl}
Cls(\mathcal{K},\mathcal{T},\tau) & = & \texttt{let val}\ \{\alpha_1,\cdots,\alpha_n\}\ =\ EFTV(\mathcal{K},\tau)\backslash EFTV(\mathcal{K},\mathcal{T}) \\
& & \texttt{in}\ (\mathcal{K}\backslash\{\alpha_1::\mathcal{K}(\alpha_1),\cdots,\alpha_n::\mathcal{K}(\alpha_n)\},\forall(\alpha_1::\mathcal{K}(\alpha_1),\cdots,\alpha_n::\mathcal{K}(\alpha_n)).\tau)\ \texttt{end}
\end{array}$$

カインドを導入しない型システムでは、$\forall\alpha.\tau_1$ と τ_2 に関して、$S(\tau_1)=\tau_2$ とする型置換 S が存在する場合、これを

$$\forall\alpha.\tau_1 > \tau_2$$

と表記する。カインドを導入した型システムでは、型変数に代入することができる型を、カインドが制約している。したがって、その制約を満たさない型に型変数を置換しないように型置換 S を構成する必要がある。カインド環境 $\mathcal{K}$ のもとでの generic instance 関係 $\mathcal{K}\vdash\sigma>\tau$ をつぎのように定義する。

$$\mathcal{K}\vdash\forall(\alpha_1::k_1,\cdots,\alpha_n::k_n).\tau_1>\tau_2 \iff \begin{array}{ll}
& dom(S)=\{\alpha_1,\cdots,\alpha_n\} \\
\wedge & S(\tau_1)=\tau_2 \\
\wedge & \forall\alpha\in dom(\mathcal{K}).\mathcal{K}\vdash S(\alpha)::S(\mathcal{K}(\alpha)) \\
\wedge & \forall(1\le i\le n).\mathcal{K}\vdash S(\alpha_i)::S(k_i)
\end{array}$$

また、$\mathcal{K}\vdash\sigma\ge\tau$ をつぎのように定義する。

$$\mathcal{K}\vdash\sigma\ge\tau \iff \sigma=\tau\ \vee\ \mathcal{K}\vdash\sigma>\tau$$

以上の概念を用いて、式に関する型づけ規則を次のように定義する。

$$\mathcal{K},\mathcal{T}\triangleright c^b : b$$

$$\frac{\mathcal{K} \vdash \sigma \geq \tau}{\mathcal{K}, \mathcal{T}\{x:\sigma\} \triangleright x : \tau}$$

$$\frac{\mathcal{K}, \mathcal{T}\{x:\tau_1\} \triangleright e : \tau_2}{\mathcal{K}, \mathcal{T} \triangleright \lambda x.e : \tau_1 \to \tau_2}$$

$$\frac{\mathcal{K}, \mathcal{T} \triangleright e_1 : \tau_1 \to \tau_2 \quad \mathcal{K}, \mathcal{T} \triangleright e_2 : \tau_1}{\mathcal{K}, \mathcal{T} \triangleright e_1\ e_2 : \tau_2}$$

$$\frac{\forall (1 \leq i \leq n).(\mathcal{K}, \mathcal{T} \triangleright e_i : \tau_i)}{\mathcal{K}, \mathcal{T} \triangleright \{l_1 = e_1, \cdots, l_n = e_n\} : \{l_1 : \tau_1, \cdots, l_n : \tau_n\}}$$

$$\frac{\mathcal{K}, \mathcal{T} \triangleright e_0 : \tau_0 \quad \forall (1 \leq i \leq n).(\mathcal{K} \triangleright p_i : \mathcal{T}_i, \tau_0 \quad \mathcal{K}, \mathcal{T}\mathcal{T}_i \triangleright e_i : \tau)}{\mathcal{K}, \mathcal{T} \triangleright \texttt{case}\ e_0\ \texttt{of}\ p_1\ \Rightarrow\ e_1 \mid \cdots \mid p_n\ \Rightarrow\ e_n : \tau}$$

$$\frac{\mathcal{K}, \mathcal{T} \triangleright e_1 : \tau_1 \quad Cls(\mathcal{K}, \mathcal{T}, \tau_1) = (\mathcal{K}', \sigma) \quad \mathcal{K}', \mathcal{T}\{x:\sigma\} \triangleright e_2 : \tau_2}{\mathcal{K}, \mathcal{T} \triangleright \texttt{let val}\ x\ =\ e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau_2}$$

カインド環境 $\mathcal{K}$ のもとでパターン p が型 τ をもち (すなわち、型 τ の値と照合することが可能であり)、パターン中の変数パターンにより型環境 $\mathcal{T}$ を生成することを、$\mathcal{K} \triangleright p : \mathcal{T}, \tau$ と表記する。

$$\mathcal{K} \triangleright c^b : \emptyset, b$$

$$\mathcal{K} \triangleright x : \{x:\tau\}, \tau$$

$$\frac{\forall (1 \leq i \leq n).(\mathcal{K} \triangleright p_i : \mathcal{T}_i, \tau_i)}{\mathcal{K} \triangleright \{l_1 : p_1, \cdots, l_n : p_n\} : \mathcal{T}_n \cdots \mathcal{T}_1, \{l_1 : \tau_1, \cdots, l_n : \tau_n\}}$$

$$\frac{\forall 1 \leq i \leq n.(\mathcal{K} \triangleright p_i : \mathcal{T}_i, \tau_i) \quad \mathcal{K} \vdash \tau :: \{\!\{l_1 : \tau_1, \cdots, l_n : \tau_n\}\!\}}{\mathcal{K} \triangleright \{l_1 : p_1, \cdots, l_n : p_n, \ldots\} : \mathcal{T}_n \cdots \mathcal{T}_1, \tau}$$

以上で定義したカインド付型システムは、レコードを操作する式に、適度に polymorphic な型を与えることができる。

2.1.3 例

例として、つぎのプログラムに型を与える。(e_1, e_2) は $\{1 = e_1, 2 = e_2\}$ の略記である。

```
let val getName = fn x => case x of {Name = n, ...} => n
in
    (getName{Age = 30, Name = "YAMADA"}, getName{Name = "TANAKA", ID = 19})
end
```

まず、$getName$ に束縛される式の型をつぎのように判定できる。スペースの都合上、$\{\alpha :: \{\!\{Name : \beta\}\!\}\}$ を $\mathcal{K}$ と略記する。

$$\frac{\mathcal{K}, \{x:\alpha\} \triangleright x : \alpha \qquad \dfrac{\mathcal{K} \triangleright n : \{n:\beta\}, \beta \quad \mathcal{K} \vdash \alpha :: \{\!\{Name : \beta\}\!\}}{\mathcal{K} \triangleright \{Name = n, \ldots\} : \{n:\beta\}, \alpha} \qquad \mathcal{K}, \{x:\alpha\}\{n:\beta\} \triangleright n : \beta}{\dfrac{\mathcal{K}, \{x:\alpha\} \triangleright \texttt{case}\ x\ \texttt{of}\ \{Name = n, \ldots\} \Rightarrow n : \beta}{\mathcal{K}, \emptyset \triangleright \texttt{fn}\ x\ \Rightarrow \texttt{case}\ x\ \texttt{of}\ \{Name = n, \ldots\} \Rightarrow n : \alpha \to \beta}}$$

そして、カインド環境 $\mathcal{K} = \{\alpha :: \{\!\{Name : \beta\}\!\}\}$ および型環境 $\emptyset$ のもとでの $\alpha \to \beta$ のクロージャを求め、

$$Cls(\{\alpha :: \{\!\{Name : \beta\}\!\}\}, \emptyset, \alpha \to \beta) \quad = \quad (\emptyset, \forall (\alpha :: \{\!\{Name : \beta\}\!\}, \beta).\alpha \to \beta)$$

型環境中で $getName$ を $\forall(\alpha :: \{\!\{Name : \beta\}\!\}, \beta).\alpha \to \beta$ で束縛する。こうして得られたカインド環境 $\emptyset$ と型環境 $\{getName : \forall(\alpha :: \{\!\{Name : \beta\}\!\}, \beta).\alpha \to \beta\}$ のもとで $getName\{Age = 30, Name = "YAMADA"\}$ と $getName\{Name = "TANAKA", ID = 19\}$ の両方に型を与えることができる。まず、$\{Age = 30, Name = "YAMADA"\}$ に適用される個所では、$getName$ の出現に対して型 $\{Name : \texttt{string}, Age : \texttt{int}\} \to \texttt{string}$ が与えられる。スペースの都合上、$\{getName : \forall(\alpha :: \{\!\{Name : \beta\}\!\}, \beta).\alpha \to \beta\}$ を $\mathcal{T}$ と略記する。

$$\dfrac{\dfrac{\emptyset \vdash \quad \begin{array}{l}\forall(\alpha :: \{\!\{Name : \beta\}\!\}, \beta).\alpha \to \beta \\ > \left\{\begin{array}{l} Age : \texttt{int} \\ Name : \texttt{string} \end{array}\right\} \to \texttt{string}\end{array}}{\emptyset, \mathcal{T} \triangleright \begin{array}{l} getName \\ : \{Age : \texttt{int}, Name : \texttt{string}\} \to \texttt{string}\end{array}} \quad \emptyset, \mathcal{T} \triangleright \left\{\begin{array}{l} Age = 30 \\ Name = "YAMADA" \end{array}\right\} : \left\{\begin{array}{l} Age : \texttt{int} \\ Name : \texttt{string} \end{array}\right\}}{\emptyset, \mathcal{T} \triangleright getName\{Age = 30, Name = "YAMADA"\} : \texttt{string}}$$

同様に、$\{Name = "TANAKA", ID = 19\}$ に適用される個所では、$getName$ の出現に対して型 $\{Name : \texttt{string}, ID : \texttt{int}\} \to \texttt{string}$ が与えられる。

$$\dfrac{\dfrac{\emptyset \vdash \quad \begin{array}{l}\forall(\alpha :: \{\!\{Name : \beta\}\!\}, \beta).\alpha \to \beta \\ > \left\{\begin{array}{l} ID : \texttt{int} \\ Name : \texttt{string} \end{array}\right\} \to \texttt{string}\end{array}}{\emptyset, \mathcal{T} \triangleright \begin{array}{l} getName \\ : \{ID : \texttt{int}, Name : \texttt{string}\} \to \texttt{string}\end{array}} \quad \emptyset, \mathcal{T} \triangleright \left\{\begin{array}{l} Name = "TANAKA" \\ ID = 19 \end{array}\right\} : \left\{\begin{array}{l} Name : \texttt{string} \\ ID : \texttt{int} \end{array}\right\}}{\emptyset, \mathcal{T} \triangleright getName\{ID = 19, Name = "TANAKA"\} : \texttt{string}}$$

こうして、`let` 式全体の型を $(\texttt{string} * \texttt{string})$ と判定できる。$(\tau_1 * \tau_2)$ は $\{1 : \tau_1, 2 : \tau_2\}$ の略記である。

$$\dfrac{\begin{array}{l}\{\alpha :: \{\!\{Name : \beta\}\!\}\}, \emptyset \triangleright \texttt{fn}\ x\ \Rightarrow \texttt{case}\ x\ \texttt{of}\ \{Name = n, \ldots\} \Rightarrow n : \alpha \to \beta \\ Cls(\{\alpha :: \{\!\{Name : \beta\}\!\}\}, \emptyset, \alpha \to \beta) = (\emptyset, \forall(\alpha :: \{\!\{Name : \beta\}\!\}, \beta).\alpha \to \beta) \\ \emptyset, \{getName : \forall(\alpha :: \{\!\{Name : \beta\}\!\}, \beta).\alpha \to \beta\} \triangleright \begin{array}{l}(getName\{Age = 30, Name = "YAMADA"\}, \\ \ getName\{Name = "TANAKA", ID = 19\})\end{array} : \begin{array}{l}(\texttt{string} \\ \ * \texttt{string})\end{array}\end{array}}{\begin{array}{l}\texttt{let val}\ getName\ =\ \texttt{fn}\ x\ \Rightarrow \texttt{case}\ x\ \texttt{of}\ \{Name = n, \ldots\} \Rightarrow n \\ \texttt{in} \\ \emptyset, \emptyset \triangleright \quad (getName\{Age = 30, Name = "YAMADA"\}, \quad : (\texttt{string} * \texttt{string}) \\ \qquad\quad getName\{Name = "TANAKA", ID = 19\}) \\ \texttt{end}\end{array}}$$

2.2 外部オブジェクトのための型システム

以上で述べたカインド付型システムを応用し、1.2.1 節、1.2.2 節で述べたような概念上のモデルを反映する形で外部データを扱うための型システムを構築する。

2.2.1 オブジェクト型

1.3.1 節で述べているように、オブジェクトあるいはコンポーネントのもつ、物理的表現を隠蔽しながらも仮想的な「状態」を公開できるという性質を表現するために、以下のように表記する「オブジェクト型」

を導入する。

$$(\alpha_1, \cdots, \alpha_m)\ T[\{l_1 : \tau_1, \cdots, l_n : \tau_n\}] \qquad (0 \leq m,\ 0 \leq n)$$

詳細については以降で説明するが、オブジェクト型 $T[\{l_1 : \tau_1, \cdots, l_n : \tau_n\}]$ に属する値は、レコード型 $\{l_1 : \tau_1, \cdots, l_n : \tau_n\}$ と同様に l_1 から l_n のフィールドを持つと考えることができる。ただし、レコード型とは異なり、ソースプログラム上でオブジェクト型の値を直接構成することはできない。つまり、

$$Person[\{Name = "YAMATO"\}]$$

のように記述して型 $Person[\{Name : string\}]$ の値を生成することはできない。実際の言語処理系上ではオブジェクト型の値はかならず外部関数により生成されることを想定している。したがって、型理論的に解析する際には、次のように

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

オブジェクト型を結果型とするプリミティブ関数が存在すると仮定し、これらの関数を呼び出すことでのみオブジェクト型の値が生成できると考える。

以降では場合により $T[\{l_1 : \tau_1, \cdots, l_n : \tau_n\}]$ を $T[\{F\}]$ あるいは単に T と略記する。

2.2.2 要件と解法

オブジェクト型を導入する目的は、型 $T[\{l_1 : \tau_1, \cdots, l_n : \tau_n\}]$ の値をレコード型 $\{l_1 : \tau_1, \cdots, l_n : \tau_n\}$ の値と同様に扱える型システムを構築することにより、外部ライブラリが生成するオブジェクトにオブジェクト型を与えてプログラム上で組み込みのレコードと同様に扱う基盤とすることにある。

その型システムはオブジェクト型に関して以下の要件を満たすことが必要だろう。ただし、T を $T[\{l_1 : \tau_1, \cdots, l_n : \tau_n\}]$ の略記、T'を $T'[\{l_1 : \tau_1, \cdots, l_n : \tau_n\}]$ の略記とし、さらに、v を T 型の値とし、v' を T' 型の値とする。

1. $f\ :\ T\ \rightarrow\ \tau$ である場合、$f\ v$ は認めるが、$f\ v'$ および $f\ \{l_1 = e_1, \ldots, l_n = e_n\}$ は認めない。
2. $g\ :\ \forall\alpha :: \{\!\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\!\}.\alpha \rightarrow \tau$ である場合、$g\ v$ は認める。
3. $h\ :\ \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \rightarrow \tau$ である場合、$h\ v$ は認めない。
4. T と T' とは互いに相容れない型とする。実装の観点からいえば、型推論時に単一化できないことを意味する。たとえば、`if` e `then` v `else` v' は認めない。

そして、パターンマッチについては次の二つの条件を満たすことが求められる。$p_1, \cdots, p_n$ はそれぞれ型を $\tau_1, \cdots, \tau_n$ とする。

1. v は flexible レコードパターン $\{l_1 : p_1, \ldots, l_n : p_n, \ldots\}$ と照合することができる。
2. v は fixed レコードパターン $\{l_1 : p_1, \ldots, l_n : p_n\}$ と照合することができる。

まず、パターンマッチに関して前者の条件のみ満たす解法を説明し、その後、後者の条件も満たす解法を説明する。

typing rule によれば、型 τ の式をパターン $\{l_1 = p_1 : \tau_1, \cdots, l_n = p_n : \tau_n, \ldots\}$ と照合するための条件は、$\mathcal{K} \vdash \tau :: \{l_1 : \tau_1, \cdots, l_n : \tau_n\}$ である。そこで、カインド規則につぎの規則を追加する。

$$\mathcal{K} \vdash (\sigma_1, \cdots, \sigma_m)\ T[\{l_1 : \tau_1, \cdots, l_n : \tau_n\}] :: [\sigma_1/\alpha_1, \cdots, \sigma_m/\alpha_m]\{\!\{l_1 : \tau_1, \cdots, l_n : \tau_n\}\!\}$$

これにより、つぎのように flexible レコードパターンでオブジェクト型をパターンマッチすることができる。

$$\dfrac{\emptyset,\emptyset \triangleright \begin{array}{l} genPerson(\text{"}YAMADA\text{"}) \\ : Person \end{array} \quad \dfrac{\begin{array}{l}\emptyset \triangleright n : \{n\!:\!\texttt{string}\}, \texttt{string} \\ \emptyset \vdash Person[\{Name\!:\!\texttt{string}\}] :: \{\!\{Name\!:\!\texttt{string}\}\!\}\end{array}}{\emptyset \triangleright \{Name = n, \ldots\} : \{n\!:\!\texttt{string}\}, Person} \quad \begin{array}{l}\emptyset, \{n\!:\!\texttt{string}\} \\ \triangleright\ n\!:\!\texttt{string}\end{array}}{\emptyset, \emptyset \triangleright \texttt{case}\ genPerson(\text{"}YAMADA\text{"})\ \texttt{of}\ \{Name = n, \ldots\} \Rightarrow n : \texttt{string}}$$

後者の条件は、たとえば次のプログラムを型づけできることを要求する。

$\texttt{val}\ genPerson : (string * int) \to Person[\{Name : string, Age : int\}]$

$\vdots$

$\texttt{let val}\ getName\ =\ \texttt{fn}\ r\ \Rightarrow\ \texttt{case}\ x\ \texttt{of}\ \{Name = n, Age = a\}\ \Rightarrow\ n\ \hat{}\ Int.toString\ a$

$\texttt{in}$

$(getName(genPerson(\text{"}TANAKA\text{"}, 20)),\ getName\ \{Name = \text{"}YAMATO\text{"}, Age = 36\})$

$\texttt{end}$

$getName$ 中の case 式で fixed レコードパターン $\{Name = n, Age = a\}$ を使っていることに注意。

まず、このパターンに型 $\{Name : string, Age : int\}$ を与えたとする。この場合、この型と $Person$ とは単一化できないので適切ではない。

$\{Name : string, Age : int\}$ と $Person$ とを単一化可能とすると、組み込みのレコード型のみを引数にとる関数、あるいは $Person$ 型のみを引数にとる関数を定義できないので、これも適切ではない。

このパターンの型として、カインド $\{\!\{Name : string, Age : int\}\!\}$ で制約された型変数 α を与えたとする。この場合、パターン $\{Name = n, Age = a\}$ と、polymorphic レコードパターン $\{Name = n, Age = a, ...\}$ とが区別できなくなるのでこれもまた適切ではない。つまり、$getName$ を $\{Name = \text{"}YAMATO\text{"}, Age = 36, Addr = \text{"}Ishikawa\text{"}\}$ のように $Name, Age$ 以外のフィールドを持つ型の式にも適用できてしまう。

カインドの定義に若干の修正を加えることで、この問題に対処する。

これまで、カインド $\{\!\{Name : string, Age : int\}\!\}$ は、**少なくとも** $Name$ と Age の 2 個のフィールドをもつ型の集合を指すものと定義していた。今、オブジェクト型を導入したことにより、$string$ 型の $Name$ と int 型の Age の 2 個のフィールド**のみ**をもつ型が複数存在する状況がありえることとなった。これまでの定義によるカインドでは、これらの型を他の型と区別して扱うことができない。これまでのカインドの定義では、$\{\!\{Name : string, Age : int\}\!\}$ には $Name$ および Age 以外のフィールドを持つ型も含まれてしまう。そこで、$Name$ と Age の 2 個のフィールドだけをもつ型のみから構成される集合をひとつのカインドとして扱う。つまり、$Name$ と Age の 2 個のフィールド**だけ**をもつ型のみから構成されるカインドと、**少なくとも** $Name$ と Age の 2 個のフィールドをもつ型から構成されるカインドとを区別する。当然、前者は後者の部分集合である。

パターン $\{Name = n, Age = a\}$ は、前者のカインドに属する型の値**のみ**を照合することができ、パターン $\{Name = n, Age = a, ...\}$ は、後者のカインドに属する型の値を照合することができるものとすることで、オブジェクト型と、組み込みのレコード型とをパターンマッチにおいて同等に扱うことができる。

2.2.3 カインド付型システムの改良

上記で述べた方針に従って、カインドを次のように定義し直す。

$$k\ ::=\ \{\!\{l : \tau, \cdots, l : \tau\}\!\} \mid \{\!\{l : \tau, \cdots, l : \tau, \ldots\}\!\}$$

カインド $\{\!\{l_1 : \tau_1, \cdots, l_n : \tau_n\}\!\}$ は、l_1 から l_n までの**ちょうど** n 個のフィールドを持つ型の集合を指す。カインド $\{\!\{l_1 : \tau_1, \cdots, l_n : \tau_n, \ldots\}\!\}$ は、**少なくとも** l_1 から l_n までの n 個のフィールドを持つ型の集合を指す。つまり、

$$\mathcal{K} \vdash \{Name : string, Age : int\} :: \{\!\{Name : string, Age : int\}\!\}$$
$$\mathcal{K} \vdash \{Name : string, Age : int\} :: \{\!\{Name : string, \ldots\}\!\}$$

は成り立つが、次は成り立たない。

$$\mathcal{K} \vdash \{Name : string, Age : int\} :: \{\!\{Name : string\}\!\}$$

また、これまで $\{\!\{\}\!\}$ と表記していたカインドは $\{\!\{\ldots\}\!\}$ と表す。

カインド規則を以下のように定義する。

$$\mathcal{K}\{\alpha :: k\} \vdash \alpha :: k$$

$$\mathcal{K} \vdash \tau :: \{\!\{\ldots\}\!\}$$

$$\mathcal{K} \vdash \{l_1 : \tau_1, \cdots, l_n : \tau_n\} :: \{\!\{l_1 : \tau_1, \cdots, l_n : \tau_n\}\!\}$$

$$\mathcal{K} \vdash (\sigma_1, \cdots, \sigma_m)\ T[\{l_1 : \tau_1, \cdots, l_n : \tau_n\}] :: [\sigma_1/\alpha_1, \cdots, \sigma_m/\alpha_m]\{\!\{l_1 : \tau_1, \cdots, l_n : \tau_n\}\!\}$$

$$\frac{\mathcal{K} \vdash \tau :: \{\!\{l_1 : \tau_1, \cdots, l_n : \tau_n, \ldots\}\!\}}{\mathcal{K} \vdash \tau :: \{\!\{l_1 : \tau_1, \cdots, l_m : \tau_m, \ldots\}\!\}\ \ (m < n)}$$

$$\frac{\mathcal{K} \vdash \tau :: \{\!\{l_1 : \tau_1, \cdots, l_n : \tau_n\}\!\}}{\mathcal{K} \vdash \tau :: \{\!\{l_1 : \tau_1, \cdots, l_n : \tau_n, \ldots\}\!\}}$$

パターンに関する型づけ規則をつぎのように定義する。

$$\mathcal{K} \triangleright c^b : \{\}, b$$
$$\mathcal{K} \triangleright x : \{x : \tau\}, \tau$$

$$\frac{\forall (1 \leq i \leq n).\mathcal{K} \triangleright p_i : \mathcal{T}_i, \tau_i \quad \mathcal{K} \vdash \tau :: \{\!\{l_1 : \tau_1, \cdots, l_n : \tau_n\}\!\}}{\mathcal{K} \triangleright \{l_1 : p_1, \cdots, l_n : p_n\} : \mathcal{T}_n \cdots \mathcal{T}_1, \tau}$$

$$\frac{\forall (1 \leq i \leq n).\mathcal{K} \triangleright p_i : \mathcal{T}_i, \tau_i \quad \mathcal{K} \vdash \tau :: \{\!\{l_1 : \tau_1, \cdots, l_n : \tau_n, \ldots\}\!\}}{\mathcal{K} \triangleright \{l_1 : p_1, \cdots, l_n : p_n, \ldots\} : \mathcal{T}_n \cdots \mathcal{T}_1, \tau}$$

式に関する型づけ規則に変更の必要はない。

以上の改良により、前に示した要件を満たすことができる。

例

例として、つぎのプログラムに型を与える。

```
let val getName = fn x => case x of {Name = n} => n
in
    (getName{Name = "YAMADA"}, getName(genPerson("TANAKA")))
end
```

まず、$getName$ に束縛される式の型をつぎのように判定できる。スペースの都合上、$\{\alpha :: \{\!\{Name : \beta\}\!\}\}$ を $\mathcal{K}$ と略記する。

$$\dfrac{\mathcal{K}, \{x:\alpha\} \triangleright x : \alpha \qquad \dfrac{\mathcal{K} \triangleright n : \{n:\beta\}, \beta \quad \mathcal{K} \vdash \alpha :: \{\!\{Name:\beta\}\!\}}{\mathcal{K} \triangleright \{Name = n\} : \{n:\beta\}, \alpha} \qquad \mathcal{K}, \{x:\alpha\}\{n:\beta\} \triangleright n : \beta}{\dfrac{\mathcal{K}, \{x:\alpha\} \triangleright \texttt{case}\ x\ \texttt{of}\ \{Name = n\} \Rightarrow n : \beta}{\mathcal{K}, \emptyset \triangleright \texttt{fn}\ x\ \Rightarrow \texttt{case}\ x\ \texttt{of}\ \{Name = n\} \Rightarrow n : \alpha \to \beta}}$$

そして、カインド環境 $\mathcal{K} = \{\alpha :: \{\!\{Name : \beta\}\!\}\}$ および型環境 $\emptyset$ のもとでの $\alpha \to \beta$ のクロージャを求め、

$$Cls(\{\alpha :: \{\!\{Name : \beta\}\!\}\}, \emptyset, \alpha \to \beta) \quad = \quad (\emptyset, \forall(\alpha :: \{\!\{Name : \beta\}\!\}, \beta).\alpha \to \beta)$$

型環境中で $getName$ を $\forall(\alpha :: \{\!\{Name : \beta\}\!\}, \beta).\alpha \to \beta$ で束縛する。こうして得られたカインド環境 $\emptyset$ と型環境 $\{getName : \forall(\alpha :: \{\!\{Name : \beta\}\!\}, \beta).\alpha \to \beta\}$ のもとで $getName\{Name = \text{"}YAMADA\text{"}\}$ と $getName(genPerson(\text{"}TANAKA\text{"}))$ の両方に型を与えることができる。まず、$\{Name = \text{"}YAMADA\text{"}\}$ に適用される個所では、$getName$ の出現に対して型 $\{Name : \texttt{string}\} \to \texttt{string}$ が与えられる。スペースの都合上、$\{getName : \forall(\alpha :: \{\!\{Name : \beta\}\!\}, \beta).\alpha \to \beta\}$ を $\mathcal{T}$ と略記する。

$$\dfrac{\emptyset \vdash \begin{array}{l}\forall(\alpha :: \{\!\{Name : \beta\}\!\}, \beta).\alpha \to \beta \\ \quad > \{Name : \texttt{string}\} \to \texttt{string}\end{array} \qquad \emptyset, \mathcal{T} \triangleright \{Name = \text{"}YAMADA\text{"}\} : \{Name : \texttt{string}\}}{\emptyset, \mathcal{T} \triangleright getName\{Name = \text{"}YAMADA\text{"}\} : \texttt{string}}$$

同様に、$genPerson(\text{"}TANAKA\text{"})$ に適用される個所では、$getName$ の出現に対して型 $Person[\{Name : \texttt{string}\}] \to \texttt{string}$ が与えられる。

$$\dfrac{\emptyset \vdash \begin{array}{l}\forall(\alpha :: \{\!\{Name : \beta\}\!\}, \beta).\alpha \to \beta \\ \quad > Person[\{Name : \texttt{string}\}] \to \texttt{string}\end{array} \qquad \dfrac{\emptyset, \mathcal{T} \triangleright (\text{"}TANAKA\text{"}) : \texttt{string}}{\emptyset, \mathcal{T} \triangleright genPerson(\text{"}TANAKA\text{"}) : Person}}{\emptyset, \mathcal{T} \triangleright getName(genPerson(\text{"}TANAKA\text{"})) : \texttt{string}}$$

こうして、`let` 式全体の型を $(\texttt{string} * \texttt{string})$ と判定できる。

$$\dfrac{\begin{array}{l}\{\alpha :: \{\!\{Name : \beta\}\!\}\}, \emptyset \triangleright \texttt{fn}\ x\ \Rightarrow \texttt{case}\ x\ \texttt{of}\ \{Name = n\} \Rightarrow n : \alpha \to \beta \\ Cls(\{\alpha :: \{\!\{Name : \beta\}\!\}\}, \emptyset, \alpha \to \beta) = (\emptyset, \forall(\alpha :: \{\!\{Name : \beta\}\!\}, \beta).\alpha \to \beta) \\ \quad \emptyset, \{getName : \forall(\alpha :: \{\!\{Name : \beta\}\!\}, \beta).\alpha \to \beta\} \\ \qquad \triangleright\ (getName\{Name = \text{"}YAMADA\text{"}\}, getName(genPerson(\text{"}TANAKA\text{"}))) : (\texttt{string} * \texttt{string})\end{array}}{\begin{array}{ll}\emptyset, \emptyset \triangleright & \texttt{let val}\ getName\ =\ \texttt{fn}\ x\ \Rightarrow \texttt{case}\ x\ \texttt{of}\ \{Name = n\} \Rightarrow n \\ & \texttt{in} \\ & \quad (getName\{Name = \text{"}YAMADA\text{"}\}, \\ & \quad\ getName(genPerson(\text{"}TANAKA\text{"}))) \\ & \texttt{end}\end{array} : (\texttt{string} * \texttt{string})}$$

2.3 カインド付型システムによるクラス階層の表現

クラスの継承関係によって定義されるオブジェクト指向的な型システムを、以上で述べたオブジェクト型を導入した型システムのもとで encode 可能であることを以下で示す。

2.3.1 クラスとカインド

クラスに基礎を置くオブジェクト指向言語において、クラスは関数やフィールドを包含するパッケージングの単位であると同時に、その名前はソースコード上で変数に与えられた型の名前としても現れる。以降では型を議論の直接の対象とするが、両者は次に述べるように密接に関連している。

オブジェクト指向言語の型システムの特徴は、制約された polymorphism をサブタイプ関係 $\leq$ によって実現している点にある。つまり、$\tau_1 \leq \tau_2$ ならば、型を $\tau_2 \to \tau$ とする関数を τ_1 型の式に適用することができる。この関係 $\leq$ は、クラス間の継承関係から導き出される。クラス c_1 がクラス c_2 を継承するならば、クラス型 c_1 とクラス型 c_2 について $c_1 \leq c_2$ が成り立つ。

クラス型とクラスとの関係は、前節までに示したレコード型やオブジェクト型とカインドとの関係に似ている。

オブジェクト指向言語においては、クラス型 c_1 の変数 v には、クラス c_1 のインスタンスを束縛することができる。クラス c_1 のインスタンスはその任意のスーパークラス c_2 のインスタンスでもあるから、このとき、v にはクラス c_2 のインスタンスが束縛されているということもできる。

一方、カインド付き型システムでは、カインド $\{\!\{l_1 : \tau_1, \cdots, l_m : \tau_m, \ldots\}\!\}$ に属する型は、同時に、任意の $n \leq m$ についてカインド $\{\!\{l_1 : \tau_1, \cdots, l_n : \tau_n, \ldots\}\!\}$ にも属している。カインドは型の集合であるから、これはカインド間の包含関係を意味している。

このようにクラス間の継承関係とカインド間の包含関係は、型に関連づけられるクラスおよびカインドを定義しており、これによって、それぞれの型システム上で polymorphism を限定する役割を果たしている。本節で述べるオブジェクト指向型システム上の型とカインドつき型システム上の型との間の変換手法は、この類似性を発想の出発点としている。

2.3.2 仮想的オブジェクト指向言語

まず、問題を解析するための仮想的なオブジェクト指向言語 OL を仮定する。OL の型がつぎのように定義できるとする。

$$\begin{aligned} \sigma_{ol} &= b \mid b \to b \\ b &= c \mid \texttt{int} \end{aligned}$$

c は OL で定義されるクラスを指す。

クラス系図 $\mathcal{C}$ はつぎのように OL で宣言されたクラス c と、クラス c が継承するクラス (c 自身も含む) の名前の集合を対応づける。

$$\mathcal{C} ::= \{c : \{c, \cdots, c\}, \cdots, c : \{c, \cdots, c\}\}$$

たとえばつぎのように定義される C++クラス

```
class A{}
class B{}
class C: A, B {}
```

の関係を、クラス系図 $\mathcal{C}$ により

$$\mathcal{C} = \{\texttt{C} : \{\texttt{A}, \texttt{B}, \texttt{C}\}, \texttt{B} : \{\texttt{B}\}, \texttt{C} : \{\texttt{C}\}\}$$

と表す。

クラス系図 $\mathcal{C}$ のもとでの、クラス型のサブタイプ関係をつぎのように定義する。

$$\mathcal{C}, OL \vdash c_1 \leq c_2 \Longleftrightarrow C(c_1) \supseteq C(c_2)$$

クラスベースの一般的なオブジェクト指向言語は、クラス型を引数にとる関数の適用についてつぎのような型づけ規則を持つ。

$$\frac{\mathcal{T} \triangleright f : c_1 \to \tau \quad \mathcal{T} \triangleright e : c_2 \quad \mathcal{C}, OL \vdash c_2 \leq c_1}{\mathcal{T} \triangleright f\ e : \tau}$$

2.3.3 OL 型と ML 型の対応

クラス系図 $\mathcal{C}$ のもとで、OL の型 σ_{ol} と ML の型 τ_{ml} を対応づけることを、

$$\mathcal{C} \vdash \sigma_{ol} = \tau_{ml}$$

と表記する。オブジェクト型

$$\alpha\ T[]$$

を用いて、変換規則をつぎのように定義する。

$$\mathcal{C} \vdash \mathtt{int} = \mathtt{int}$$

$$\frac{\mathcal{C}(c) = \{c_1, \cdots, c_n\}}{\mathcal{C} \vdash c = \{c_1 : \mathtt{unit}, \cdots, c_n : \mathtt{unit}\}\ T}$$

$$\frac{\mathcal{C} \vdash \sigma = \tau}{\mathcal{C} \vdash c \to \sigma = \forall(\alpha :: \{\!\{c : \mathtt{unit}, \ldots\}\!\}).\alpha\ T \to \tau}$$

$\mathcal{C}[\![\sigma_{ol}]\!] = \tau_{ml}$ をつぎのように定義しておく。

$$\mathcal{C} \vdash \sigma_{ol} = \tau_{ml} \Longleftrightarrow \mathcal{C}[\![\sigma_{ol}]\!] = \tau_{ml}$$

また、$C = \{c_1, \cdots, c_n\}$ であるときに、$\{C\}$ を $\{c_1 : \mathtt{unit}, \cdots, c_n : \mathtt{unit}\}$ の略記とする。

2.3.4 安全性

上記の変換規則は、型を $\mathcal{C}[\![c_1 \to \sigma]\!]$ とする ML の関数 f が、$\mathcal{C}, OL \vdash c_2 \leq c_1$ かつ $\mathcal{C} \vdash c_2 = \tau_2$ であるような型 τ_2 の値のみに適用できることを保証する。これは、

$$\mathcal{C}[\![c_1 \to \sigma]\!] = \forall(\alpha :: \{\!\{c_1 : \mathtt{unit}, \ldots\}\!\}).\alpha\ T \to \mathcal{C}[\![\sigma]\!]$$

より、つぎのように言い換えることができる。

定理 2.1 $\mathcal{K}, \mathcal{T}\{f : \forall(\alpha :: \{\!\{c_1 : \mathtt{unit}, \ldots\}\!\}).\alpha\ T \to \mathcal{C}[\![\sigma]\!]\} \triangleright f\ M_2 : \tau$ かつ
$\mathcal{K}, \mathcal{T}\{f : \forall(\alpha :: \{\!\{c_1 : \mathtt{unit}, \ldots\}\!\}).\alpha\ T \to \mathcal{C}[\![\sigma]\!]\} \triangleright M_2 : \tau_2$ ならば、$\mathcal{C}, OL \vdash c_2 \leq c_1$ かつ $\mathcal{C}[\![c_2]\!] = \tau_2$ を満たす c_2 が存在する。

仮定　定理 2.1 を証明する準備として、この変換方法が実際に応用される状況を想定したいくつかの仮定をおく。

T がオブジェクト型である場合、ソースプログラム中で任意の τ について $\tau\ T$ 型の値を生成することはできない。あらかじめ宣言されるプリミティブ関数 (あるいは外部関数) によってのみオブジェクト型の値を生成できる。また、これらのプリミティブ関数は、OL で定義されている関数に対応し上記の変換規則に従った適切な型を与えられていると仮定する。したがって、つぎの仮定をおくことができる。

仮定 2.2 $\mathcal{K}, \mathcal{T} \triangleright e : \tau\ T$ ならば、$\mathcal{C} \vdash \sigma_{ol} = \tau\ T$ であるような σ_{ol} が必ず存在する。

この仮定と変換規則から、さらに次の仮定が導き出される。

仮定 2.3 $\mathcal{K}, \mathcal{T} \triangleright e : \tau\ T$ ならば、τ はレコード型である。

さらに、クラス系図とプリミティブ関数が、次のより強い仮定を満たすと仮定する。

仮定 2.4 $\mathcal{K}, \mathcal{T} \triangleright e : \{F\}\ T$ であり、あるクラス c について $c \in dom(F)$ ならば、$\mathcal{C}(c) \subseteq dom(F)$ である。

証明　仮定の

$$\mathcal{K}, \mathcal{T}\{f : \forall(\alpha :: \{\!\{c_1 : \mathtt{unit}, \ldots\}\!\}).\alpha\ T \to \mathcal{C}[\![\sigma]\!]\} \triangleright f\ M_2 : \mathcal{C}[\![\sigma]\!]$$
$$\mathcal{K}, \mathcal{T}\{f : \forall(\alpha :: \{\!\{c_1 : \mathtt{unit}, \ldots\}\!\}).\alpha\ T \to \mathcal{C}[\![\sigma]\!]\} \triangleright M_2 : \tau_2$$

より、関数適用に関する型づけ規則にしたがい、

$$\mathcal{K}, \mathcal{T}\{f : \forall(\alpha :: \{\!\{c_1 : \mathtt{unit}, \ldots\}\!\}).\alpha\ T \to \mathcal{C}[\![\sigma]\!]\} \triangleright f : \tau_2 \to \mathcal{C}[\![\sigma]\!]$$

である。さらに変数に関する型づけ規則にしたがい、

$$\mathcal{K} \vdash \forall(\alpha :: \{\!\{c_1 : \mathtt{unit}, \ldots\}\!\}).\alpha\ T \to \mathcal{C}[\![\sigma]\!] > \tau_2 \to \mathcal{C}[\![\sigma]\!]$$

である。$>$ の定義より、$\tau_2'\ T = \tau_2$ であるような τ_2' と、つぎのような S が存在して、

$$S\ =\ \{\alpha : \tau_2'\}$$

かつ、

$$\mathcal{K} \vdash S(\alpha) :: S(\{\!\{c_1 : \mathtt{unit}, \ldots\}\!\})$$

すなわち

$$\mathcal{K} \vdash \tau_2' :: \{\!\{c_1 : \mathtt{unit}, \ldots\}\!\}$$

である。ここで、カインド規則および、仮定 2.3 より τ_2' がレコード型であることから、ある C が存在して、

$$\tau_2'\ =\ \{\{c_1\} \cup C\}$$

であり、さらに仮定 2.4 よりある $C' \subseteq C$ が存在して

$$\tau_2'\ =\ \{\{c_1\} \cup C\}\ =\ \{\mathcal{C}(c_1) \cup C'\}$$

である。また、仮定より $\mathcal{K}, \mathcal{T}\{f : \forall(\alpha :: \{\!\{c_1 : \mathtt{unit}, \ldots\}\!\}).\alpha\ T \to \mathcal{C}[\![\sigma]\!]\} \triangleright M_2 : \tau_2$ すなわち

$$\mathcal{K}, \mathcal{T}\{f : \forall(\alpha :: \{\!\{c_1 : \mathtt{unit}, \ldots\}\!\}).\alpha\ T \to \mathcal{C}[\![\sigma]\!]\} \triangleright M_2 : \tau_2'\ T$$

であることから、仮定 2.2 より、

$$\mathcal{C} \vdash c \ = \ \{\mathcal{C}(c_1) \cup C'\}\ T$$

であるようなクラス c が存在する。そして、変換規則より、

$$\mathcal{C}(c) \ = \ \mathcal{C}(c_1) \cup C'$$

である。このとき、

$$\mathcal{C}(c) \ = \ \mathcal{C}(c_1) \cup C' \ \supseteq \ \mathcal{C}(c_1)$$

であるから、

$$\mathcal{C}, OL \vdash c \leq c_1$$

つまり、

$$\mathcal{C}, OL \vdash c \leq c_1 \text{ かつ } \mathcal{C}[\![c]\!] = \tau_2$$

が成り立つ。

2.3.5 例

前にとりあげた以下のクラスを例にとる。

```
class A{}
class B{}
class C: A, B {}
```

クラス系図 $\mathcal{C}$ は以下の通り。

$$\mathcal{C} \ = \ \{\mathtt{A} : \{\mathtt{A}\}, \mathtt{B} : \{\mathtt{B}\}, \mathtt{C} : \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}\}$$

そして、クラス A,B,C に対応する ML 型はつぎのように得られる。

$$\begin{aligned}
\mathcal{C}[\![\mathtt{A}]\!] &= \{\mathtt{A} : \mathtt{unit}\}\ T \\
\mathcal{C}[\![\mathtt{B}]\!] &= \{\mathtt{B} : \mathtt{unit}\}\ T \\
\mathcal{C}[\![\mathtt{C}]\!] &= \{\mathtt{A} : \mathtt{unit},\ \mathtt{B} : \mathtt{unit},\ \mathtt{C} : \mathtt{unit}\}\ T
\end{aligned}$$

クラス A を引数にとり int を返す OL 上の関数型は、ML 上ではつぎの型に対応する。

$$\mathcal{C}[\![\mathtt{A} \to \mathtt{int}]\!] \ = \ \forall(\alpha :: \{\mathtt{A} : \mathtt{unit}, \ldots\}).\alpha\ T \to \mathtt{int}$$

そして、この型を持つ ML 上の関数 g に、クラス C に対応する型をもつ式 x を引数として与えることができる。$\{g : \forall(\alpha :: \{\mathtt{A} : \mathtt{unit}, \ldots\}).\alpha\ T \to \mathtt{int}, x : \{\mathtt{A} : \mathtt{unit},\ \mathtt{B} : \mathtt{unit},\ \mathtt{C} : \mathtt{unit}\}\ T\}$ を $\mathcal{T}$ と略記している。

$$\dfrac{\dfrac{\emptyset \vdash \begin{array}{l}\forall(\alpha :: \{\mathtt{A} : \mathtt{unit}, \ldots\}).\alpha\ T \to \mathtt{int} \\ \quad > \{\mathtt{A} : \mathtt{unit}, \mathtt{B} : \mathtt{unit}, \mathtt{C} : \mathtt{unit}\}\ T \to \mathtt{int}\end{array}}{\emptyset, \mathcal{T} \triangleright g : \{\mathtt{A} : \mathtt{unit}, \mathtt{B} : \mathtt{unit}, \mathtt{C} : \mathtt{unit}\}\ T \to \mathtt{int}} \quad \emptyset, \mathcal{T} \triangleright x : \{\mathtt{A} : \mathtt{unit}, \mathtt{B} : \mathtt{unit}, \mathtt{C} : \mathtt{unit}\}\ T}{\emptyset, \mathcal{T} \triangleright g\ x : \mathtt{int}}$$

より具体的な応用例として、4.4.4 節で Java のクラス定義を ML の型で表現する手法を述べている。

第3章　スクリプティング指向関数型言語 Amethyst

2 章で述べた型理論上の成果が実用的な関数型言語に応用できることを実証するため、Standard ML ベースのプログラミング言語 Amethyst を設計した。1 章で述べたように、Amethyst は、コンポーネントを組み合わせてアプリケーションを記述するスクリプティング言語としての用途を想定し、外部リソース操作を実現する機構を導入している。

本章は、外部リソース操作およびカインド付型システムに関連する部分に対象を限定して述べる。A 章に、その他も含めた構文規則を示す。

3.1　外部宣言

3.1.1　ドメイン宣言

次の構文はドメインを宣言する。

$$\texttt{domain}\ dom\ =\ \texttt{imports}\ \text{"}initializer\text{"}\ \texttt{of}\ \text{"}module\text{"}$$
$$\texttt{domain}\ dom\ =\ \texttt{imports}\ \text{"}initializer\text{"}\ \texttt{with}\ \text{"}initarg\text{"}\ \texttt{of}\ \text{"}module\text{"}$$

この構文は、ドメイン *dom* が提供する外部リソースに関する処理を *module* モジュールが実装していること、ドメインの初期化処理を *module* モジュールの関数 *initializer* で実装していることを宣言する。ここでいう「モジュール」とは、Windows の `.dll` や Unix の `.so` のような共有ライブラリあるいは動的モジュールを指す。初期化関数については後で説明する。また、省略可能な `with` "*initarg*" で、初期化関数に渡すパラメータを指定することができる。

例

$$\texttt{domain postgres}\ =\ \texttt{imports "init" of "pglib";}$$

この文は、`postgres` ドメインを宣言している。同時に、ネイティブライブラリ `pglib` 中の `init` 関数がドメイン初期化関数であることも宣言している。

3.1.2　外部型

外部リソースを表現する型を、つぎの構文によって宣言する。

$$\texttt{external type}\ (\alpha_1, \cdots, \alpha_m)\ T\ =\ \texttt{imports}\ \text{"}name\text{"}\ \texttt{of}\ dom$$

この構文は、dom ドメインが提供する $name$ で指定される外部リソースの型を T と表すことを宣言する。

とくに、$name$ を外部名 (external name) と呼ぶ。これは外部リソースに関する他の各種宣言構文でも同様である。

例

$$\texttt{external type connection} \ = \ \texttt{imports "RGconn" of postgres}$$

この文は、postgres ドメインの RGconn という名前で識別される種類のリソースを connection 型の値として扱うことを宣言している。

3.1.3 レコード型の外部型

外部型宣言の際につぎの構文を用いることで、この外部型の値に対して組み込みレコードと同様のパターンマッチをおこなえることを宣言できる。

$$\texttt{external type } (\alpha_1, \cdots, \alpha_m)\ T = \{l_1 : \tau_1\ \texttt{"}attr_1\texttt{"}, \cdots, l_n : \tau_n\ \texttt{"}attr_n\texttt{"}\}\ \texttt{imports "}name\texttt{" of } dom$$

この構文で宣言される型は、前章で述べたオブジェクト型 $(\alpha_1, \cdots, \alpha_m)\ T[\{l_1 : \tau_1, \cdots, l_n : \tau_n\}]$ に相当する。この構文は、外部型 T の値が、型を $\tau_1, \cdots, \tau_n$ とするフィールド $l_1 \cdots l_n$ を持つことを宣言する。$attr_1$ から $attr_n$ は、各フィールドと外部データの構成要素との対応関係を決定する際に参考となる属性情報を指定する。属性情報の形式はドメイン依存である。属性情報は省略可能である。

また、$\tau_1, \cdots, \tau_n$ 中の自由型変数は $\alpha_1, \cdots, \alpha_n$ に含まれなければならない。つまり、以下が成り立つことが必要である。

$$FTV(\tau_1) \cup \cdots \cup FTV(\tau_n) \subseteq \{\alpha_1, \cdots, \alpha_n\}$$

例

$$\texttt{external type emprec} = \{\texttt{Name} : \texttt{string "NAME"}, \texttt{Rank} : \texttt{int}, \texttt{Salary} : \texttt{int "SALARY"}\}$$
$$\texttt{imports "RECORD" of postgres}$$

この文は、Name と Rank、Salary の三つのフィールドを含むデータベースレコードを表現する emprec 型を宣言する。

3.1.4 直和型の外部型

$$\texttt{external type } (\alpha_1, \cdots, \alpha_m)\ T = C_1\ \texttt{of}\ \tau_1\ \texttt{"}attr_1\texttt{"} \mid \cdots \mid C_n\ \texttt{of}\ \tau_n\ \texttt{"}attr_n\texttt{" imports "}name\texttt{" of } dom$$

この構文で宣言される外部型の値は、datatype 文により宣言されるタグつき直和型と同様に、値構成子によるパターンマッチをおこなうことができる。$attr_1, \cdots, attr_n$ は、各値構成子に与えられた属性情報である。また、$\tau_1, \cdots, \tau_n$ 中の自由型変数は、$\alpha_1, \cdots, \alpha_m$ のいずれかでなければならない。この構文により宣言される外部型の値を、値構成子 C_i を用いて生成することはできない。外部関数のみが、この型の値を生成することができる。

例

external type 'a dbrec = R of 'a "R" | BOR "BOR" | EOR "EOR" imports "PGresult" of postgres

dbrec は postgres ドメインのデータベースレコードを指すカーソルを表す。dbrec を構成する各値構成子のうち、BOR はレコードセットの始点を指すカーソル、EOR はレコードセットの終点を指すカーソル、R は有効なレコードを指すカーソルを表している。

3.1.5 外部変数

つぎの構文は、外部ライブラリが Amethyst に対して公開する変数を宣言する。

$$\texttt{external val } v : \tau \texttt{ = imports "}name\texttt{" of } dom$$

$$\texttt{external fun } f : \tau_1 \to \tau_2 \texttt{ = imports "}name\texttt{" of } dom$$

後者の external fun は ML の fun に合わせたもので、関数型を指定する必要がある以外は external val と同じである。external val でも関数型を指定することができる。

例

external val queryEmployee : connection → string → emprec dbrec =
imports "query : select NAME, RANK, SALARY from employee where @1"
of postgres

この文は、employee テーブルに対するクエリーを実行して emprec dbrec 型の値を返す外部関数を宣言する。

3.2 外部型の仕様

SML では、モジュールの実装とインターフェイスとをストラクチャとシグネチャとに分けて宣言することができる。シグネチャは、ストラクチャで定義されている値や型についての抽象的な情報のみを明示する仕様宣言から構成される。

external type 宣言は、その構文中にドメインや外部名など、実装の詳細に関わる情報を含んでいる。external type 宣言を含むストラクチャにシグネチャを与える場合、これらの情報をシグネチャ上で公開すべきではない。

以降で、外部型宣言に対応する仕様宣言文について検討する。

3.2.1 レコード型の外部型

SML には、レコード型の外部型の性質を表現する適当な仕様宣言文がない。よって、つぎの構文を導入する。

$$\texttt{objecttype } (\alpha_1, \cdots, \alpha_m)T \texttt{ = } \{lbl_1 : \tau_1, \cdots, lbl_n : \tau_n\}$$

この構文は、$lbl_1, \cdots, lbl_n$ のフィールドを持つ型を指す仕様を宣言する。この構文は signature 宣言中でのみ使用できる。組み込みのレコード型に対しても objecttype で仕様を与えることができる。

たとえば、つぎのようにシグネチャを明示するストラクチャを定義できる。

```
signature SIG =
  sig
    objecttype emprec = {Name:string, Rank:int}
  end

structure STR1:SIG =
  struct
    external type emprec = {Name:string "S:NAME", Rank:int "I:RANK"}
                             imports "RECORD" of postgres;
  end

structure STR2:SIG =
  struct
    type emprec = {Name:string, Rank:int}
  end
```

3.2.2 直和型の外部型

直和型の外部型には、datatype によって定義された型と同様の仕様を与えることができる。たとえば、つぎのようにシグネチャを与えたストラクチャが定義できる。

```
signature SIG =
  sig
    datatype 'a dbrec = R of 'a | BOR | EOR
  end

structure STR1:SIG =
  struct
    external type 'a dbrec = R of 'a | BOR | EOR imports "PGresult" of postgres
  end

signature STR2:SIG =
  struct
    datatype 'a dbrec = R of 'a | BOR | EOR
  end
```

3.3 型式

カインド付型システムは、カインドで制約された polymorphic な型という概念を導入する。これに対応し、カインドに制約された polymorphic な型をソースコード上でプログラマが指定できるよう型に関する構文を拡張する。

3.3.1 カインドに制約される型変数

val 宣言文の構文をつぎのように拡張し、型変数にカインドによる制約を与えることを認める。

$$
\begin{array}{rcl}
dec & ::= & \vdots \\
 & | & \texttt{val}\ kindedtyvarseq\ id = exp \\
 & | & \vdots
\end{array}
$$

$$
\begin{array}{rcl}
kindedtyvar & ::= & tyvar \\
 & | & tyvar\texttt{:}\ \{lbl\texttt{:}ty, \cdots, lbl\texttt{:}ty\} \\
 & | & tyvar\texttt{:}\ \{lbl\texttt{:}ty, \cdots, lbl\texttt{:}ty, \ldots\}
\end{array}
$$

3.3.2 カインドに制約される quantification

次の構文拡張により、カインドにより制約された polymorphic な型を明示できる。

$$
\begin{array}{rcl}
ty & ::= & \vdots \\
 & | & \texttt{forall}\ kindedtyvarseq \Rightarrow ty \\
 & | & \vdots
\end{array}
$$

3.4 例

以上で述べた外部宣言の使用例を示す。

PostgreSQL データベースに対して検索を実行し、取得したデータベースレコードからフィールド値を取り出すプログラムを以下のように記述できる。

pglib.ams

```
 1 domain postgres = imports "init" of "pglib";
 2
 3 exception PGerror of string;
 4
 5 external type connection = imports "PGconn" of postgres;
 6 external type 'a dbrec = R of 'a "R" | BOR "BOR" | EOR "EOR"
 7                               imports "PGresult" of postgres;
 8
 9 external fun open : string -> string -> string -> string -> connection
10                                       = imports "open:" of postgres;
11 external fun moveNext : 'a dbrec -> 'a dbrec = imports "movenext:" of postgres;
12 external fun close : connection -> unit = imports "close:" of postgres;
```

pglib.ams は、PostgreSQL へのアクセスに必要な基本的な外部型および外部関数を宣言する。

pgtest.ams

```
 1 (* load PostgreSQL library *)
 2 :load "pglib.ams";
 3
 4 (* declare external types consisting of fields of result of query *)
 5 external type emprec = {Name:string "S:NAME", Rank:int "I:RANK"}
 6          imports "RECORD" of postgres;
 7
 8 (* declare query function *)
 9 (* '@n' in query string will be substituted with value of string argument *)
10 external fun queryEmployee : connection -> string -> (emprec dbrec option) =
11          imports "query:select NAME, RANK from EMPLOYEE where @1" of postgres;
12
13 fun allNames rs =
14     let val rss = moveNext rs
15     in
16     (case rss of
17          EOR => []
18        | R{Name=n,...} => n::(allNames rss))
19     end;
20 fun getNames optrs =
21     case optrs of
22         NONE => []
23       | SOME rs => allNames(rs);
24
25 (*
26   open database connection
27   arguments are host name running database server, database name, user name, password
28 *)
29 val c = open "localhost" "testdb" "kiyoshiy" "kiyoshiy"
30         handle e as (PGerror msg) => (print msg;raise e);
31
32 (*
33   send query
34 *)
35 val emp = queryEmployee c "RANK = 1";
36
37 (*
38   retrieve value of name field from all records
39 *)
40 val empnames = getNames emp;
41
```

```
42 (*
43   close database connection explicitly.
44 *)
45 val _ = close c;
```

pglib.ams で定義されている外部型、外部関数を用いて、上記のように PostgreSQL データベースにアクセスするアプリケーションコードを記述できる。18 行目に見えるように、データベースレコードのフィールド値を、ML 組み込みのレコードに対するのと同様のパターンマッチによって取出すことができる。

実行例　つぎのようにデータベースを用意したうえで

```
bash-2.04$ psql testdb
testdb=# select * from EMPLOYEE;
      name       | rank
-----------------+------
 ISHIZAKA Taizou |    1
 DOKOU Toshio    |    2
 HIRAIWA Gaishi  |    3
(3 rows)
```

pgtest.ams を Amethyst で実行すると、つぎの出力を得る。

```
bash-2.04$ amethysti preludes.ams pgtest.ams
domain postgres = imports "init" of pglib
exception PGerror of string
external type  connection =  imports "PGconn" of postgres
external type 'a dbrec = R of 'a "R" | BOR "BOR" | EOR "EOR"
                                  imports "PGresult" of postgres
external fun  open:string -> string -> string -> string -> connection
                                  = imports "open:" of postgres
external fun  moveNext:'a dbrec -> 'a dbrec = imports "movenext:" of postgres
external fun  close:connection -> unit = imports "close:" of postgres
external type  emprec = {Name:string "S:NAME",Rank:int "I:RANK"}
                                  imports "RECORD" of postgres
external fun  queryEmployee:connection -> string -> emprec dbrec option =
         imports "query:select NAME, RANK from EMPLOYEE where @1" of postgres
val allNames = fn : forall ('a,'b:{Name:'a,...}) => 'b dbrec -> 'a list
val getNames = fn : forall ('a,'b:{Name:'a,...}) => 'b dbrec option -> 'a list
val c = ??? : connection
val emp = SOME ??? : emprec dbrec option
val empnames = ["ISHIZAKA Taizou"] : string list
```

外部型の値は"???"と表示している。また、最後の行から、RANK フィールドを 1 とするレコードの NAME フィールドの値を獲得できていることが分かる。

第4章　実装

4.1　実装の概要

本研究の成果として、1 章で掲げた目標を実現するスクリプティング指向関数型言語の処理系を実装した。この処理系は、3 章で設計した言語 Amethyst で記述されたプログラムを、2 章で定義した型システムのもとで解析した上でコンパイルし、さまざまな外部ライブラリと連携しながら実行する。

4.1.1　問題点とその解決

処理系の設計および実装に際して、外部ライブラリとの連携に関する問題点を以下のように解決した。

- 外部型の導入により、ソースコード上で同じように見えるデータ構造が実行時にも同じように表現されると仮定することは不可能となった。

  一方、2.1 節で型理論的な部分を紹介した多相型レコード計算は、それを準備として、引数で渡されるレコードのフィールド構成がコンパイル時に判定不可能な polymoprhic 関数の効率的なコンパイルを実現している。

  本研究はこれを応用し、値の物理的構造がコンパイル時はおろか実行時でも処理系には不可知であるような状況を扱えるコンパイル方法を開発した。

- 外部ライブラリを記述する言語および形式について、Amethyst の処理系があらかじめ仮定を置くことはできない。したがって、さまざまな外部ライブラリを扱うためには、これらに対応して動的に調整可能な機構を処理系の側で用意する必要がある。

  異種の言語/形式で記述されたモジュール間のデータ形式や関数呼び出し手順に関する差異を吸収するコードを**ブリッジコード**と呼ぶ。Amethyst では、ブリッジコードを実装して外部ライブラリとランタイムとの間を仲介するレイヤーをドメインモジュールと呼んでいる。ランタイムと外部ライブラリとの間に、動的にプラグイン可能なドメインモジュールを挟み込むことで、複数の言語/形式で記述されたモジュールをランタイムが扱えるようにした。

- 外部型の値は、リモートオブジェクトへのハンドルやデータベース接続など有限な外部リソースを占有する。必要のなくなった外部リソースは、各リソースごとに定められた所定の手続きによって確実に解放する必要がある。したがって、処理系が実装するガーベジコレクションを利用し、不要な外部リソースに対する解放手続きの実行を確実とすることが望ましい。

  また、外部型の値に格納される情報の物理的表現形式は、個々の外部ライブラリの実装に依存する。一方で、関数型言語処理系のランタイムはヒープ管理を中心に設計されており、ランタイム上の値の物理的な表現形式はガーベジコレクタの実装方法に強く制約される。したがって、外部データへの対応を組み込んだヒープ管理方式を、ランタイム全体への影響を考慮して注意深く検討する必要がある。

Amethyst は、オブジェクト指向の言語処理系で実際に採用されている方法を組み合わせて、上記の問題に対処している。

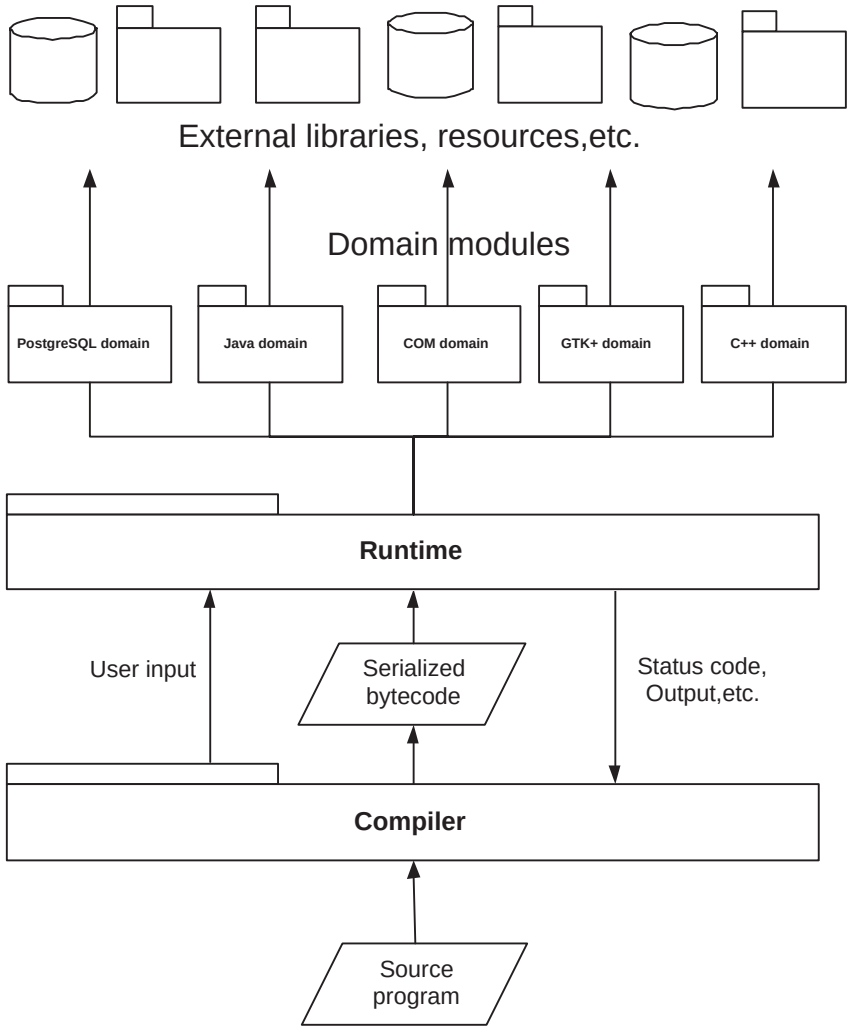
- 以上で挙げた問題点はいずれも、外部ライブラリと Amethyst との動的側面における差異に起因するものであった。これと並んで、両者の静的側面での差異、すなわち型システムに関する差異も解消する必要がある。

  この点に関しては、すでに 2 章において基本的な準備を済ませている。仮想的な言語を対象としてそこで示した手法を、Amethyst は現実のオブジェクト指向言語に応用している。

本章は、以上の点を中心に処理系の実装について述べる。

4.1.2 システム構成

Amethyst 処理系は下図に示すサブシステムで構成する。



コンパイラはソース言語をバイトコードにコンパイルする。Amethyst のコンパイラは、2 章に示した外

部リソース操作のための型システムを実装し、3 章で示した Amethyst 言語で記述されたプログラムをコンパイルする。

ランタイムはコンパイラが生成するバイトコードを解釈・実行する。Amethyst のランタイムは ZINC 抽象機械 [Ler90] をベースとし、外部型および外部関数の導入に対応して、既存の命令の修正および新たな命令の追加を施している。また、外部データを適切に扱えるようガーベジコレクションおよび値の物理的表現形式に工夫を加えている。

ドメインモジュールはランタイムと外部ライブラリとの間の差異を吸収する。それは `int` や `string` といった基本的なデータ形式の変換から、より抽象的なモデル間の変換にまで及ぶ。外部ライブラリの基礎にあるモデルが複雑である場合、ドメインモジュールが果たすべき役割も大きなものとなる。

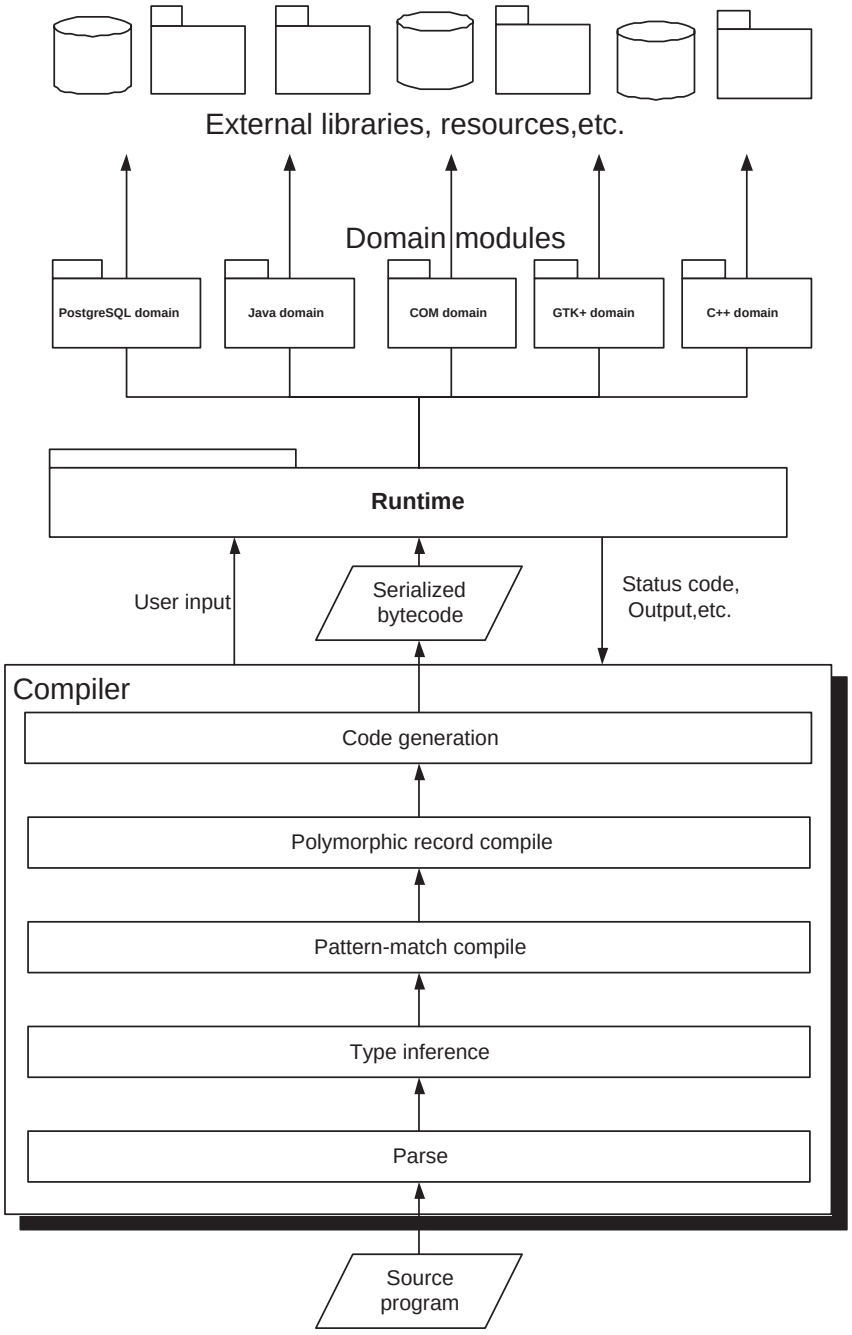
以降で、ここで挙げた順に各サブシステムを説明する。

また、コンパイラおよびランタイムに関する説明は、ZINC 抽象機械についての知識を前提としている。B 章で ZINC 抽象機械を説明しているので参考にされたい。

4.2 コンパイラ

4.2.1 全体像

コンパイラは、下図に示す phase を経てソースプログラムをバイトコード列にコンパイルする。



本節はとくに、外部型の値を操作するソースプログラムを入力として受け取り、適切な外部ライブラリ関数の呼び出しを指示するバイトコード列を生成するまでの過程について述べる。

4.2.2 多相型レコード計算のコンパイル

概要

2 章で述べたように、外部型を加えた Amethyst の型システムは [Oho95] が示す多相型レコード計算を応用している。外部型を扱うプログラムのコンパイルについても、[Oho95] をもとにしている。そこで、まず [Oho95] で示されているコンパイル手法について概観する。

たとえば式#Name{Name="YAMADA",Age=30} のようにレコードからフィールド値を取り出す式は、実行時には、いくつかのセルにより構成されるメモリブロックからひとつのセルを取出す操作として実現される。ソースプログラム上では取出すフィールドを名前 (ラベル) により指定するが、実行時にはメモリブロック中のセルを指すインデックス (整数) により指定する。したがって、いずれかの時点で、名前による指定からインデックスによる指定に変換しなければならない。

#Name に適用される式の型がただひとつに決定されるならば、取出すセルの位置はコンパイル時に決定できる。たとえば次の式

$$\#\mathtt{Name}\ \{\mathtt{Age} = 30,\ \mathtt{Name} = \text{"}\mathtt{YAMADA}\text{"}\}$$

において、#Name に渡される式の型は {Age:int, Name:string} のみである。ここで、実行時にレコード型の値はそのフィールドの個数のセルから成るメモリブロックとして表現され、ブロック内では、フィールド名のアルファベット順にセルが並べられるものとする。すると、この#Name は、引数で渡されるメモリブロックの二番目のセルを取出すコードにコンパイルすればよい。

2 章で述べたカインド付型システムでは、getName に polymorphic な型 $\forall(\alpha :: \{\!\{\mathtt{Name} : \beta\}\!\}, \beta).\alpha \to \beta$ を与えることができる。したがって、つぎのようなプログラムを記述できる。

```
let fun getName r = #Name r
in
  (getName {Age = 30, Name = "YAMADA"}_a, getName {Name = "TANAKA", Salary = 1000}_b)
end
```

しかし、先ほどの式とは異なり、この式では、取出すべきセルの位置をコンパイル時に決定できない。型 {Age:int, Name:string} の値は二つのセルから成るメモリブロックとして表現され、Name フィールドはその二番目のセルに配置される。また、型 {Name:string, Salary:int} の値も二つのセルから成るメモリブロックとして表現されるが、Name フィールドは一番目のセルに配置される。したがって上の getName 中の#Name r は、a の呼び出しに際して r の二番目のセルを取出し、b の呼び出しに際しては r の一番目のセルを取出すコードにコンパイルしなければならない。

上記の問題は、式 #Name r において r の型が決定できないことに原因がある。r の型が分かれば、r の何番目のフィールドを取り出すべきかを決定できる。そこで、まず、polymorphic 関数に対して、引数の値とともに引数の型に関しての情報を渡してやることを考える。型に関するすべての情報を渡す必要はない。r の型 τ を構成するフィールド中での Name の位置を教えてやれば十分である。つまり、カインドにより引数の型が制約されている polymorphic 関数は、ソースプログラム上の仮引数に加えて、その引数を通して呼び出し側から渡されるレコード中の必要なフィールドを指すインデックスを引数として受け取る。「必要なフィールド」とは、引数を制約するカインド中に現れるフィールドにほかならない。また、その関数を呼び出す際には、ソースプログラム上で指定されている実引数に加え、その引数に含まれるレコードのフィールドのうち、その関数が必要とするフィールドを指すインデックスを渡す。

たとえば

```
fun getName r = #Name r
```

で定義される関数 getName は、型が $\forall(\alpha :: \{\!\{\mathtt{Name} : \beta\}\!\}, \beta).\alpha \to \beta$ であり、ひとつのフィールド (Name) を要求する。そこで、これを

$$\mathtt{fun\ getName\ I\ r\ =\ \#[I]\ r}$$

へと変換する。追加された引数 I は、型を α とする引数、すなわち r が指すレコード中の Name フィールドを指すインデックスを受け取るために用いられる。#[I] r は、実行時に r が指すメモリブロックの I 番目のセルを取出すようなコードにコンパイルされる。以降では、I のようにインデックスを受け渡すことを目的として追加される変数をインデックス変数と呼ぶ。

これに対応し、getName を呼び出す式

$$\mathtt{getName\ \{Age = 30,\ Name = "YAMADA"\}}$$

は、getName の型が $\forall(\alpha :: \{\!\{\mathtt{Name} : \beta\}\!\}, \beta).\alpha \to \beta$ であることより、getName が Name フィールドのインデックスを要求することが分かるので、

$$\mathtt{getName\ 2\ \{Age = 30,\ Name = "YAMADA"\}}$$

に変換する。{Age=30, Name="YAMADA"} の型は {Age:int, Name:string} であり、Name フィールドは実行時に二番目のセルに配置されるので getName には 2 を渡す。

変換

上記の変換を、二段階に分けて説明する。

1. 型情報の注釈
2. インデックス引数の挿入

この過程で、ソースプログラムは次ページで示す形の式に順に変換される。

以降では、プログラム全体の一括変換を二回おこなうように説明している。Amethyst の実装では、これらの変換を型推論、パターンマッチコンパイル、多相型レコード計算コンパイルの各段階に分散するコードによって実行しており、必ずしもプログラム全体を一括して変換するものではない。

$$
\begin{array}{rcl}
e & ::= & c^b \\
& | & x \\
& | & \texttt{fn}\ x\ \Rightarrow\ e \\
& | & e\ e \\
& | & \#l\ e \\
& | & \{lbl = e, \cdots, lbl = e\} \\
& | & \texttt{let val}\ x\ =\ e\ \texttt{in}\ e\ \texttt{end}
\end{array}
$$

⇓

型情報の注釈

⇓

$$
\begin{array}{rcl}
e & ::= & c^b \\
& | & x\ (\alpha :: k, \cdots, \alpha :: k)\ (\tau, \cdots, \tau) \\
& | & \texttt{fn}\ x\ \Rightarrow\ e \\
& | & e\ e \\
& | & \#(l, \tau)\ e \\
& | & \{lbl = e, \cdots, lbl = e\} \\
& | & \texttt{let val}\ x\ =\ \texttt{fn}(\alpha :: k, \cdots, \alpha :: k) \Rightarrow e\ \texttt{in}\ e\ \texttt{end}
\end{array}
$$

⇓

インデックス引数の挿入

⇓

$$
\begin{array}{rcl}
e & ::= & c^b \\
& | & x\ \mathcal{I}\ \cdots\ \mathcal{I} \\
& | & \texttt{fn}\ x\ \Rightarrow\ e \\
& | & e\ e \\
& | & \#[\mathcal{I}]\ e \\
& | & \{lbl = e, \cdots, lbl = e\} \\
& | & \texttt{let val}\ x\ =\ \texttt{fn}\ I_{(lbl,\alpha)} \Rightarrow \cdots \texttt{fn}\ I_{(lbl,\alpha)} \Rightarrow e\ \texttt{in}\ e\ \texttt{end} \\
\\
\mathcal{I} & ::= & i \\
& | & I_{(lbl,\alpha)}
\end{array}
$$

型情報の注釈　最初の変換では、型推論により得られる型情報を式に付加する。

型環境 $\{\texttt{id} : \forall\alpha.\alpha \to \alpha\}$ のもとで、次の式を考える。

$$\texttt{id 1}$$

$$\texttt{id true}$$

最初の式では id を int → int 型の式として使用し、二番目の式では bool → bool 型の式として使用している。このように polymorphic な型、すなわち $\forall\alpha.\tau$ を型とする変数は、それが使用される個所のそれぞれにおいて、τ 中の α を文脈から決定されるいずれかの型 σ に置き換えて得られる型 $[\sigma/\alpha]\tau$ の式として扱われる。言い換えれば、$\forall\alpha.\tau$ のような型は、その型変数 $(=\alpha)$ に代入する型を引数にとる「型に関する関数」であり、文脈から与えられる型 σ を用いて $[\sigma/\alpha]\tau$ を得ることは、型に関する関数適用であるとみることができる。

ここでは、以下のように型に関する関数とその適用およびフィールド取出し式のそれぞれにおいて、α および σ のように文脈から与えられる型を明示する。

- let 式

  $$\texttt{let val } x = e_1 \texttt{ in } e_2 \texttt{ end}$$

  において、e_1 の型を generalize した結果が $\forall(\alpha_1 :: k_1, \ldots, \alpha_n :: k_n).\sigma_1 \to \sigma_2$ であるならば、e_1 を、ソース上で宣言されている引数に加えて α_1 から α_n に対応する n 個の型を引数にとる関数に変換する。つまり、let 式を

  $$\texttt{let val } x = \texttt{fn}(\alpha_1 :: k_1, \cdots, \alpha_n :: k_n) \Rightarrow e_1 \texttt{ in } e_2 \texttt{ end}$$

  に変換する。

- polymorphic に束縛されている x が出現している個所で、x を型 $[\tau_1/\alpha_1]\ldots[\tau_n/\alpha_n]\tau$ の式として用いているならば、この変数式を

  $$x\ (\alpha_1, \cdots, \alpha_n)\ (\tau_1, \cdots, \tau_n)$$

  に変換する。

- 式 e からラベル l で指定するフィールドを取り出す式

  $$\#l\ e$$

  を、e が型を τ と判定されるならば、

  $$\#(l, \tau)\ e$$

  に変換する。

例

$$
\begin{array}{l}
\texttt{let val } getName = \texttt{fn } r \Rightarrow \#Name\ r \\
\texttt{in} \\
\quad (getName\{Age = 30, Name = "YAMADA"\}, \\
\quad\ getName\{Name = "TANAKA", Salary = 1000\}) \\
\texttt{end}
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{l}
\texttt{let val } getName = \texttt{fn}(\alpha :: \{\!\{Name : \beta\}\!\}, \beta) \Rightarrow \texttt{fn } r \Rightarrow \#(Name, \alpha)\ r \\
\texttt{in} \\
\quad ((getName\ (\alpha :: \{\!\{Name : \beta\}\!\}, \beta)\ (\{Age : \texttt{int}, Name : \texttt{string}\}, \texttt{string})) \\
\qquad\quad \{Age = 30, Name = "YAMADA"\}, \\
\quad\ (getName\ (\alpha :: \{\!\{Name : \beta\}\!\}, \beta)\ (\{Name : \texttt{string}, Salary : \texttt{int}\}, \texttt{string})) \\
\qquad\quad \{Name = "TANAKA", Salary = 1000\}) \\
\texttt{end}
\end{array}
$$

インデックス引数の挿入　上述のように polymorphic に束縛された変数の出現を $(x\ (\alpha_1 :: k_1, \cdots, \alpha_n :: k_n)\ (\tau_1, \cdots, \tau_n))$ に置き換える目的は、この変数 (に束縛される関数) が参照するレコードフィールドを明らかにすることにある。次の段階では、x が必要とするフィールドを $\tau_1, \cdots, \tau_n$ から抽出し、それらのインデックスを x に与える式に変換する。同様に、カインドに制約されて polymorphic に束縛される式を、この式が必要とするフィールドのインデックスを引数として受け取る関数に変換する。

変換の手順を示す前に、フィールドのインデックスに関していくつかの定義を示す。

フィールドは型とラベル名の組によって識別できるので、(l, τ) と表記する。これは、型 τ を構成するフィールドのうち、ラベル l を名前とするフィールドを指す。このフィールドを指すインデックスの集合を $idx(l, \tau)$ と表記する。以降では、$idx(l, \tau)$ をインデックス型、その要素をインデックス値と呼ぶ。τ がレコード型である場合、$idx(l, \tau)$ の要素は静的に決定できる。たとえば $idx(\texttt{Name}, \{\texttt{Name} : \texttt{string}\}) = \{1\}$ である。τ が型変数である場合、インデックス値を静的に判定することはできず、この変換により追加される引数を通して実行時に取得する。この追加される引数はインデックス値の受け渡しに用いられるのでインデックス変数と呼ぶ。

つぎの $idxOf$ は、フィールドを指すインデックス値あるいは実行時にインデックス値を保持するインデックス変数を返す。

$$
\begin{array}{rcl}
idxOf\ idx(l,\ \{l_1 : \tau_1, \ldots l : \tau_i, \ldots l_n : \tau_n\}) & = & i \\
idxOf\ idx(l,\ \alpha) & = & I_{(l,\ \alpha)}
\end{array}
$$

つぎの $IdxList$ は、カインドを与えられた型変数の組 $(\alpha_1 :: k_1, \ldots, \alpha_n :: k_n)$ を受け取り、この型の値が必要とするインデックス型のリストを返す。集合ではなく、順序を保存するリストであることに注意する。

$$
IdxList((\alpha_1 :: k_1, \ldots, \alpha_n :: k_n)) \quad = \quad (idxlist\ (\alpha_1,\ k_1)) @ \ldots @ (idxlist\ (\alpha_n,\ k_n))
$$

$$
\begin{aligned}
idxlist\ (\alpha,\ \{\!\{l_1 :: \tau_1, \ldots, l_n :: \tau_n\}\!\}) &= [idx(l_1, \alpha), \ldots, idx(l_n, \alpha)] \\
idxlist\ (\alpha,\ \{\!\{l_1 :: \tau_1, \ldots, l_n :: \tau_n, \ldots\}\!\}) &= [idx(l_1, \alpha), \ldots, idx(l_n, \alpha)]
\end{aligned}
$$

先の変換によって式に注釈された型情報を、このように定義されるインデックスに関する情報に置き換える。

- `let` 式によって polymorphic に束縛される関数は、その式が必要とするフィールドのインデックスを引数として受け取る関数に変換する。つぎの `let` 式を考える。

  $$\texttt{let val}\ x\ =\ \texttt{fn}(\alpha_1 :: k_1, \cdots, \alpha_m :: k_m) \Rightarrow M_1\ \texttt{in}\ M_2\ \texttt{end}$$

  まず、M_1 が要求するインデックス型のリストを得る。

  $$[idx(l_1, \beta_1), \ldots, idx(l_n, \beta_n)]\ =\ IdxList(\alpha_1 :: k_1, \ldots, \alpha_m :: k_m)$$

  得られたインデックス型のそれぞれに対応し、実際のインデックスを受け取るインデックス変数を M_1 の仮引数に加え、

  $$\texttt{let val}\ x\ =\ \texttt{fn}\ I_{(l_1,\beta_1)}\ \Rightarrow\ \ldots\ \texttt{fn}\ I_{(l_n,\beta_n)}\ \Rightarrow\ M_1\ \texttt{in}\ M_2\ \texttt{end}$$

  に変換する。

- 変数式にはつぎのように型に関する注釈が付加されている。

  $$x\ (\alpha_1 :: k_1, \cdots, \alpha_m :: k_m)\ (\tau_1, \cdots, \tau_m)$$

  これを、x が要求するインデックス値を x に渡す式に変換する。まず、x が要求するインデックス型のリストを得る。

  $$[idx(l_1, \beta_1), \cdots, idx(l_n, \beta_n)]\ =\ IdxList(\alpha_1 :: k_1, \cdots, \alpha_m :: k_m)$$

  変数式 x は型 $[\tau_m/\alpha_m] \cdots [\tau_1/\alpha_1]\tau$ の値として用いられているので、つぎの代入 S を用いて、

  $$S\ =\ [\tau_m/\alpha_m] \cdots [\tau_1/\alpha_1]$$

  変数式 x が要求するインデックス型はつぎのようになる。

  $$[idx(l_1, S(\beta_1)), \cdots, idx(l_n, S(\beta_n))]$$

  それぞれのインデックス型について、そのインデックス値を得る。

  $$\mathcal{I}_i\ =\ idxOf\ idx(l_i, S(\beta_i))$$

  そして変数式 $x\ (\alpha_1 :: k_1, \cdots, \alpha_m :: k_m)\ (\tau_1, \cdots, \tau_m)$ を、つぎの式に変換する。

  $$x\ \mathcal{I}_1\ \cdots\ \mathcal{I}_n$$

- τ 型のレコード中でラベルを l とするフィールドを指すインデックスは、$idxOf\ idx(l, \tau)$ により得られる。これを $\mathcal{I}$ とすると、$\#(l, \tau)\ e$ を

  $$\#[\mathcal{I}]\ e$$

  に変換する。

例

$$
\begin{array}{l}
\texttt{let val } getName = \texttt{fn}(\alpha :: \{\!\{Name : \beta\}\!\}, \beta) \Rightarrow \texttt{fn } r \Rightarrow \#(Name, \alpha)\ r \\
\texttt{in} \\
\quad ((getName\ (\alpha :: \{\!\{Name : \beta\}\!\}, \beta)\ (\{Age : \texttt{int}, Name : \texttt{string}\}, \texttt{string})) \\
\qquad \{Age = 30, Name = \text{"}YAMADA\text{"}\}, \\
\quad (getName\ (\alpha :: \{\!\{Name : \beta\}\!\}, \beta)\ (\{Name : \texttt{string}, Salary : \texttt{int}\}, \texttt{string})) \\
\qquad \{Name = \text{"}TANAKA\text{"}, Salary = 1000\}) \\
\texttt{end}
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{l}
\texttt{let val } getName = \texttt{fn } I_{(Name,\alpha)} \Rightarrow \texttt{fn } r \Rightarrow \#[I_{(Name,\alpha)}]\ r \\
\texttt{in} \\
\quad ((getName\ 1)\{Age = 30, Name = \text{"}YAMADA\text{"}\}, \\
\quad (getName\ 2)\{Name = \text{"}TANAKA\text{"}, Salary = 1000\}) \\
\texttt{end}
\end{array}
$$

4.2.3 オブジェクト型への対応

2 章で示したオブジェクト型を導入した型システムを言語処理系に適用するには、第一に、型推論アルゴリズムを修正し、第二に、前節で説明したコンパイル方法をオブジェクト型に応用する必要がある。

型推論

2.2.3 節でのカインドの再定義に対応して、型推論アルゴリズム中の単一化規則を修正する。
まず、[Oho95] で示される単一化規則に現れるカインド $\{\!\{F\}\!\}$ をカインド $\{\!\{F, \ldots\}\!\}$ に読み替える。
さらに、カインド $\{\!\{F\}\!\}$ に関する二つの規則を新たに加える。
最初に示す規則は 2.2.3 節で定義したカインド規則の

$$
\begin{array}{l}
\mathcal{K} \vdash \{l_1 : \tau_1, \cdots, l_n : \tau_n\} :: \{\!\{l_1 : \tau_1, \cdots, l_n : \tau_n\}\!\} \\
\mathcal{K} \vdash (\sigma_1, \cdots, \sigma_m)\ T[\{l_1 : \tau_1, \cdots, l_n : \tau_n\}] :: [\sigma_1/\alpha_1, \cdots, \sigma_m/\alpha_m]\{\!\{l_1 : \tau_1, \cdots, l_n : \tau_n\}\!\}
\end{array}
$$

に対応する。

$$
\begin{array}{l}
(E \cup \{(\alpha_1, \tau_2)\},\ \mathcal{K} \cup \{(\alpha_1, \{\!\{F_1\}\!\})\},\ S,\ \mathcal{SK}) \Longrightarrow \\
([\tau_2/\alpha_1](E \cup \{(F_1(l), F_2(l)) | l \in dom(F_1)\}),\ [\tau_2/\alpha_1](\mathcal{K}), \\
\ [\tau_2/\alpha_1](S) \cup \{(\alpha_1, \tau_2)\}, [\tau_2/\alpha_1](\mathcal{SK}) \cup \{(\alpha_1, \{\!\{F_1\}\!\})\}) \\
\text{if } \mathcal{K} \vdash \tau_2 :: \{\!\{F_2\}\!\} \text{ and } dom(F_1) = dom(F_2) \text{ and } \alpha_1 \notin FTV(\{F_2\})
\end{array}
$$

この規則は、カインド $\{\!\{F_1\}\!\}$ を与えられた型変数 α_1 と、F_1 と同じラベルのフィールドをもつ型 τ_2 との単一化を試みる。ここで現れる τ_2 には、$\{F_2\}$ あるいは $T[\{F_2\}]$ が該当する。

つぎに示す規則は、カインド規則の

$$\frac{\mathcal{K} \vdash \tau :: \{\!\{l_1 : \tau_1, \cdots, l_n : \tau_n\}\!\}}{\mathcal{K} \vdash \tau :: \{\!\{l_1 : \tau_1, \cdots, l_n : \tau_n, \ldots\}\!\}}$$

に対応する。

$$\begin{aligned}
&(E \cup \{(\alpha_1, \alpha_2)\},\ \mathcal{K} \cup \{(\alpha_1, \{\!\{F_1, \ldots\}\!\}), (\alpha_2, \{\!\{F_2\}\!\})\},\ S,\ \mathcal{SK}) \Longrightarrow \\
&([\alpha_2/\alpha_1](E \cup \{(F_1(l), F_2(l)) | l \in dom(F_1)\}), \\
&\ [\alpha_2/\alpha_1](\mathcal{K}) \cup \{(\alpha_2, [\alpha_2/\alpha_1](\{\!\{F_2\}\!\}))\}, \\
&\ [\alpha_2/\alpha_1](S) \cup \{(\alpha_1, \alpha_2)\},\ [\alpha_2/\alpha_1](\mathcal{SK}) \cup \{(\alpha_1, \{\!\{F_1\}\!\})\}) \\
&\text{if } dom(F_1) \subseteq dom(F_2)
\end{aligned}$$

これらの規則を加えることにより、型推論機構がオブジェクト型を扱うことができる。

コード生成

オブジェクト型を導入したことにより、式

$$\#lbl\ r$$

において r からフィールドを取出す手順は、r に束縛される値の属するドメインにより全く異なりうる。そこで、polymorphic な関数を呼び出す際に、フィールドを指すインデックスを受け渡す替わりにフィールド値を取出すコード (=セレクタ関数) を受け渡すことで、物理表現が不明なデータからのフィールド取出しを実現する。

前節で示した変換手順のうち、「インデックス引数の挿入」変換を、以降で述べる「セレクタ引数の挿入」変換に置き換える。

⇓

型情報の注釈

⇓

セレクタ引数の挿入

⇓

$$
\begin{array}{rcl}
e & ::= & c^b \\
& | & x\ \mathcal{S}\ \cdots\ \mathcal{S} \\
& | & \texttt{fn}\ x\ \Rightarrow\ e \\
& | & e\ e \\
& | & \mathcal{S}\ e \\
& | & \{lbl = e, \cdots, lbl = e\} \\
& | & \texttt{let val}\ x\ =\ \texttt{fn}\ S_{(lbl,\alpha)} \Rightarrow \cdots \texttt{fn}\ S_{(lbl,\alpha)} \Rightarrow e\ \texttt{in}\ e\ \texttt{end} \\
\\
\mathcal{S} & ::= & \texttt{fn}\ r \Rightarrow sel(lbl, \tau)(r) \\
& | & S_{(lbl,\alpha)}
\end{array}
$$

$idxOf$ の定義をつぎのように変更する。

$$
\begin{array}{rcl}
idxOf\ idx(l,\ \{l_1 : \tau_1, \cdots, l_n : \tau_n\}) & = & sel(l, \{l_1 : \tau_1, \cdots, l_n : \tau_n\}) \\
idxOf\ idx(l,\ T[\{l_1 : \tau_1, \cdots, l_n : \tau_n\}]) & = & sel(l, T) \\
idxOf\ idx(l,\ \alpha) & = & S_{(l,\ \alpha)}
\end{array}
$$

$sel(l, \tau)(r)$ は、型が τ である式 r を評価して得られる値から、l が指すフィールドの値を取り出す。

例

$$
\begin{array}{l}
\texttt{let val}\ getName = \texttt{fn}(\alpha :: \{\!\{Name : \beta\}\!\}, \beta) \Rightarrow \texttt{fn}\ r \Rightarrow \#(Name, \alpha)\ r \\
\texttt{in} \\
\quad ((getName\ (\alpha :: \{\!\{Name : \beta\}\!\}, \beta)\ (Person[\{Age : \texttt{int}, Name : \texttt{string}\}], \texttt{string})) \\
\qquad (genPerson(\text{"}YAMADA\text{"}, 30)), \\
\quad (getName\ (\alpha :: \{\!\{Name : \beta\}\!\}, \beta)\ (\{Name : \texttt{string}, Salary : \texttt{int}\}, \texttt{string})) \\
\qquad \{Name = \text{"}TANAKA\text{"}, Salary = 1000\}) \\
\texttt{end}
\end{array}
$$

⇓

```
let val getName = fn I_(Name,α) ⇒ fn r ⇒ #[I_(Name,α)] r
in
  (getName (fn r ⇒ sel(Name, Person[{Age : int, Name : string}])(r))
        (genPerson("YAMADA", 30)),
   getName (fn r ⇒ sel(Name, {Name : string, Salary : int})(r))
        {Name = "TANAKA", Salary = 1000})
end
```

セレクタ関数のコンパイル　上記の変換により生成される $sel(l,\tau)(r)$ は、最終的につぎのように仮想マシン命令 GetField あるいは ExtGetField にコンパイルされる。

$$\begin{aligned}\mathcal{C}[\![sel(l,\{l_1:\tau_1,\cdots,l:\tau_i,\cdots,l_n:\tau_n\})(r)]\!] &= \mathcal{C}[\![r]\!];\texttt{GetField}(i)\\ \mathcal{T}[\![sel(l,\{l_1:\tau_1,\cdots,l:\tau_i,\cdots,l_n:\tau_n\})(r)]\!] &= \mathcal{C}[\![r]\!];\texttt{GetField}(i)\end{aligned}$$

$$\begin{aligned}\mathcal{C}[\![sel(l,T[\{l_1:\tau_1,\cdots,l:\tau_i,\cdots,l_n:\tau_n\}])(r)]\!] &= \mathcal{C}[\![r]\!];\texttt{ExtGetField}(l,T)\\ \mathcal{T}[\![sel(l,T[\{l_1:\tau_1,\cdots,l:\tau_i,\cdots,l_n:\tau_n\}])(r)]\!] &= \mathcal{C}[\![r]\!];\texttt{ExtGetField}(l,T)\end{aligned}$$

GetField,ExtGetField はつぎのようにマシン状態を遷移する。

Code	Accu	Env.	Arg.stack	Return stack	Heap
$\texttt{GetField}(i);c_0$	b	e	s	r	$H\{b \mapsto C[v_1,\ldots,v_n]\}$
c_0	v_i	e	s	r	$H\{b \mapsto C[v_1,\ldots,v_n]\}$
$\texttt{ExtGetField}(l,T);c_0$	v	e	s	r	H
c_0	a (a は T に対応するセレクタを l と v を引数にして呼び出した返り値)	e	s	r	H

GetField(i) は、Accum 上に置かれたポインタが指すブロックから i 番目のフィールドの値を得て Accum に置く。

ExtGetField(l,T) は、Accum 上の値 v とラベル l とを引数として、T に関連づけられている外部ライブラリの関数 (selector) を呼び出し、その返り値を Accum に置く。

4.2.4　直和型外部型のコンパイル

直和型に関するパターンマッチは、パターンマッチコンパイルにより switch 式に変換される。

$$\texttt{switch}(e:\tau,[C_1:e_1,\ldots,C_n:e_n],e_d)$$

は、e を評価して得られる値からタグ C を取出し、それが C_1 ならば e_1、C_n ならば e_n を評価する。C が $C_1,\ldots,C_n$ のいずれにも一致しない場合、e_d を評価する。

e の値からタグ C を得る手順は τ により異なる。τ が `datatype` 文により宣言された型である場合、実行時に e を評価するとヒープ中のブロックを指すポインタが得られるので、そのポインタが指すブロックからタグを取り出す。τ が外部型ならば、実行時に e を評価して得られる値 v は外部ライブラリによって生成された値である。この場合、τ に対してあらかじめ対応づけられている外部ライブラリの関数 (getTag) に v を渡して呼び出し、タグを得る。

`switch` 式は、仮想機械命令 `SwitchTag` あるいは `ExtSwitchTag` にコンパイルされる。τ_0 が `datatype` 文で宣言された型である場合は `SwitchTag` を用いる。

$$\begin{aligned}\mathcal{T}[\![\texttt{switch}(e_0:\tau_0,[C_1:e_1,\ldots,C_n:e_n],e_d)]\!] &= \mathcal{C}[\![e_0]\!];\texttt{SwitchTag}([C_1:\mathcal{C}[\![e_1]\!],\cdots,C_n:\mathcal{C}[\![e_n]\!]],\mathcal{C}[\![e_d]\!]);\\ \mathcal{C}[\![\texttt{switch}(e_0:\tau_0,[C_1:e_1,\ldots,C_n:e_n],e_d)]\!] &= \mathcal{C}[\![e_0]\!];\texttt{SwitchTag}([C_1:\mathcal{C}[\![e_1]\!],\cdots,C_n:\mathcal{C}[\![e_n]\!]],\mathcal{C}[\![e_d]\!]);\end{aligned}$$

τ_0 が `externaltype` 文で宣言された型である場合は、`ExtSwitchTag` を用いる。

$$\begin{aligned}\mathcal{T}[\![\texttt{switch}(e_0:\tau_0,[C_1:e_1,\ldots,C_n:e_n],e_d)]\!] &= \mathcal{C}[\![e_0]\!];\texttt{ExtSwitchTag}(\tau_0,[C_1:\mathcal{C}[\![e_1]\!],\cdots,C_n:\mathcal{C}[\![e_n]\!]],\mathcal{C}[\![e_d]\!]);\\ \mathcal{C}[\![\texttt{switch}(e_0:\tau_0,[C_1:e_1,\ldots,C_n:e_n],e_d)]\!] &= \mathcal{C}[\![e_0]\!];\texttt{ExtSwitchTag}(\tau_0,[C_1:\mathcal{C}[\![e_1]\!],\cdots,C_n:\mathcal{C}[\![e_n]\!]],\mathcal{C}[\![e_d]\!]);\end{aligned}$$

`SwitchTag` および `ExtSwitchTag` はつぎのように仮想機械の状態を遷移する。

Code	Accu	Env.	Arg.stack	Return stack	Heap
$\texttt{SwitchTag}([C_1:c_1,\cdots,C_n:c_n],c_d);c_0$	b	e	s	r	$H\{b\mapsto C[\ldots]\}$
$\begin{cases} c_i & C_i = C \text{ である場合} \\ c_d & \text{それ以外の場合}\end{cases}$	b	e	s	r	$H\{b\mapsto C[\ldots]\}$
$\texttt{ExtSwitchTag}(\tau,[C_1:c_1,\cdots,C_n:c_n],c_d);c_0$	v	e	s	r	H
$\begin{cases} c_i & \tau\text{に対応する } getTag \text{ を呼び出して} \\ & \text{得られたタグと } C_i\text{が一致する場合} \\ c_d & \text{それ以外の場合}\end{cases}$	v	e	s	r	H

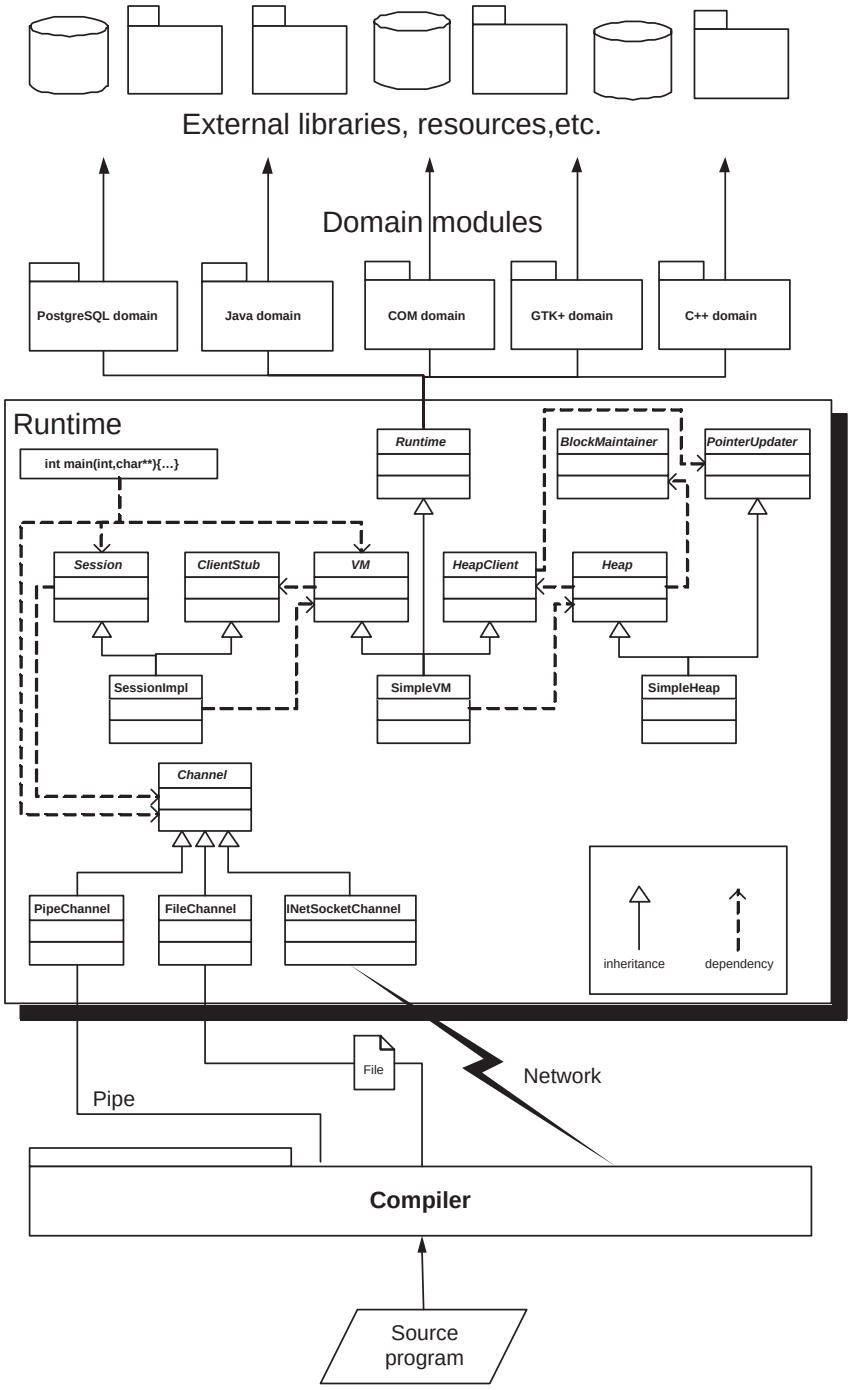
`SwitchTag`$([C_1:c_1,\cdots,C_n:c_n],c_d)$ は、Accum 上に置かれたポインタ b が指すブロックからタグ C を得て、$C_1,\ldots,C_n$ と比較する。いずれかの C_i と C が一致する場合、c_i が指すコードにジャンプする。C がいずれの C_i とも一致しない場合、c_d が指すコードにジャンプする。

`ExtSwitchTag`$(\tau_0,[C_1:c_1,\cdots,C_n:c_n],c_d)$ は、Accum 上に置かれた値 v を引数として τ_0 と関連づけられている外部ライブラリの関数 (getTag) を呼び出し、その返り値としてタグ C を得る。この C を $C_1,\ldots,C_n$ と比較する。いずれかの C_i と C が一致する場合、c_i が指すコードにジャンプする。C がいずれの C_i とも一致しない場合、c_d が指すコードにジャンプする。

4.3 ランタイム

4.3.1 全体像

Amethyst のランタイム (バイトコードインタプリタ) は、下図に示すクラスにより実装されている。



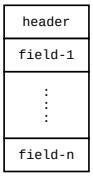
これらのクラスは、レイヤー構造を成すと考えることができる。最下部にはヒープが位置する。ヒープは上位レイヤに対してメモリ管理に関するサービスを提供する。ヒープの上位には仮想機械が位置する。このレイヤは、ヒープの提供するサービスを利用して、バイトコードインタプリタを実装する。仮想機械の上位にはフロントエンドとのインターフェイスを担当するセッションレイヤが位置する。セッションレイヤは、フロントエンドとの通信を実装するチャネルレイヤにも依存している。本節では、このうち、外部データおよび外部関数の操作に関係するヒープと仮想機械について述べる。

また、ネットワークを構成する各レイヤがネットワークを異なるレベルでとらえるように、これらのレイヤは「値」を異なる粒度で扱っている。ヒープは、即値と管理ブロック、カスタムブロックの 3 種類でしか値を区別しない。仮想機械は、よりソースコードに近い粒度で値を識別する。たとえば `int`、`string`、`real` やその他のブロック構造を区別して扱う。本節では、このような「値」の粒度の違いについても説明する。

4.3.2 メモリ管理

ヒープは `CELLSIZE` バイトのセルを要素とする配列として実装する。`CELLSIZE` は、プラットホームでのアドレス値を保持するに十分なサイズ (=`sizeof(void*)`) とする。

ヒープモジュールは、セルをグループ化したブロック単位でヒープを管理する。ブロックは、先頭 1 セルを占有するヘッダと後続のフィールドから構成される。ヘッダはデータエリアに保持されているデータの種類を識別するタグのほか、サイズやガーベジコレクションに関係する情報を保持する。



即値とポインタ

ヒープマネージャからみれば、ブロックのフィールドに格納される値は即値と他のブロックを指すポインタの二種類しかない。ガーベジコレクションの際にヒープマネージャはブロックのフィールドを検査し、ポインタならばそれが指すブロックを再帰的に検査する。このため、即値とポインタとは容易に判別できる方法で表現されなければならない。

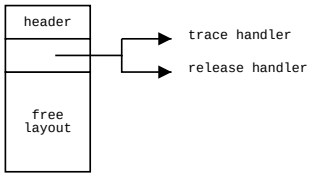
CAML-light やその他ガーベジコレクションを組み込んでいる多数の言語処理系が、つぎの方法で即値とポインタとを判別している。

> ひとつのセルが w ビットから構成されるとする。alignment によりブロックを指すポインタは最下位ビットがつねに 0 である。そこで、即値を $w-1$ ビットで表現し、最下位ビット (LSB) を 1 とすることで、ポインタと区別する。

Amethyst のヒープマネージャもこの方法をとっている。

カスタムブロック

タプルやレコードについては上記のブロックで表現できるが、`real` や `string` などの値を格納するには、セル単位でしかアクセスできないブロックは適当ではない。そこで、ヒープマネージャはカスタムブロック (Custom block) と呼ぶ種類のブロック形式をクライアントに提供している。外部データとの関連で述べると、外部ライブラリは、データベース接続を指すハンドルやその他のパラメータなど、外部リソースに関する情報をドメイン独自の形式でカスタムブロックに保持することができる。



これまで述べてきたブロックは管理ブロック (Managed block) と呼ぶ。管理ブロックについては、ヒープマネージャがその構成内容を管理する。カスタムブロックについては、ヒープマネージャは以下に述べる二点について、そのブロックを使用するモジュールの助けをかりて管理する。

ブロック間の参照　他のヒープブロックへのポインタをカスタムブロック中に保持する場合、ガーベジコレクションの際にヒープマネージャがそのポインタをたどれるようブロック中のポインタの所在をヒープマネージャに通知する必要がある。このため、trace ハンドラと呼ぶ関数をカスタムブロックに対応づける。ヒープマネージャは、ガーベジコレクション時にポインタをたどってカスタムブロック b に到達すると、b の trace ハンドラを呼び出す。trace ハンドラは、他のヒープブロック b' を指すポインタ p' を b 中に保持するならば、p' のアドレスをヒープマネージャに通知する。ガーベジコレクションの最中に b' は移動する可能性があるが、その場合、移動先を指すように p' の値がヒープマネージャにより書き換えられる。

カスタムブロックの解放　外部のリソースを使用する場合、データベース接続を解放したり、ファイルをクローズするなど、不要となったリソースに対して所定の手続きを呼び出す必要があることが多い。この手続きを確実に呼び出すようプログラマが注意してコードを記述すればよいが、クロージャのために変数の寿命が動的に変りうる関数型言語プログラムの場合、どの時点でそれらの手続きを呼び出すべきかプログラマが決定することは難しい。そこで、参照解放時にヒープマネージャが確実にこれらの手続きを呼び出す仕組みを考える。

trace ハンドラと同様の方法をここで用いる。カスタムブロックと、そのブロック中に含まれるデータの後始末を実行する release ハンドラと呼ぶ関数を対応させる。ヒープマネージャは、ガーベジコレクションの際にガーベジと判定されたカスタムブロックのそれぞれについて、対応する release ハンドラを呼び出す。

ここで、release ハンドラを呼び出すタイミングには注意が必要である。まず、あるブロック b が到達不能と判定した直後に、b に対して release ハンドラを呼び出す方法は安全ではない。release ハンドラ中でヒープからブロックを新たに割り当てるような処理をおこない、それが再びガーベジコレクションを起動したと仮定する。このとき、ガーベジコレクタは b が占めるセルが使用されていないとみなしているので、それらのセルを他の値で上書きする可能性がある。

問題は、ブロック b に対して release ハンドラが呼び出されるには b がガーベジであると認識されることが前提だが、release ハンドラ呼び出しが完了するまでは b をガーベジとしてはならないという点にある。

そこで、b がガーベジと認識された時点で b を指すポインタを一時的にルートセットに加え、そのうえで release ハンドラを呼び出し、呼び出しが完了した時点でルートセットから取り除くことでこの問題に対処する [Ric00]。以下に具体的に説明する。

この手法は、生存リストと解放待ちリストと呼ぶ二つのリストを用いる。生存リストは、ルートセットから到達可能なカスタムブロックを指すポインタを保持する。解放待ちリストは、ルートセットから到達不可能であることが判明したが、まだ release ハンドラを呼び出していないカスタムブロックを指すポインタを保持する。

まず、ヒープマネージャは、カスタムブロックを生成する際に、そのブロックを指すポインタを生存リス

トに追加する。ガーベジコレクション時にヒープを走査した後生存リスト中のポインタが指すブロックがガーベジと認識されていた場合、そのポインタを生存リストから取り除き解放待ちリストに追加する。stop & copy 方式を採用しているならば、その前にブロックを移動する。生存リスト中のポインタはガーベジコレクトの際のルートセットには含まれない。解放待ちリストに含まれるポインタは、ガーベジコレクション時に、通常のルートセットと同様に扱われる。このため、ポインタが指すブロックがガーベジと認識されることは避けられる。

そして、以降の任意の時点で解放待ちリスト中のポインタ p を選び、p が指すブロック b に対して release ハンドラを呼び出す。release ハンドラが完了した時点で p を解放待ちリストから取り除く。b が解放待ちリスト中の他のポインタから到達可能でなければ、次回のガーベジコレクション時に b が占有するセルはガーベジとされ回収される。

4.3.3 仮想機械

値の表現形式

`int` 型の値をヒープマネージャの即値として表現するのは適切ではない。前述のヒープマネージャで採用している即値の表現形式では、ポインタと即値を判別するために使用される 1 ビット分、即値として表現できる数値の範囲が狭められる。Amethyst は外部ライブラリとの連携を重視しているので、プラットホームの native な整数値をそのまま扱えることが望ましい。

boxing とよばれるつぎの方法を用いると、native な整数値を損なうことなく表現することができる。

boxing `int` 型の値をつねにカスタムブロック中に保持する。

この方法は、non-boxing よりも多くのメモリを消費することと、常にヒープアクセスが伴うために処理速度の点で劣るという欠点をもっている。Amethyst では、non-boxing と boxing の双方の利点を取り入れ、欠点を補い合う方法として、つぎの手法を採用している [GR83]。

automatic boxing-unboxing この方法は、通常は non-boxing と同じであるが、必要に応じて boxing をおこなう。つまり、$w-1$ ビットでおさまる整数値は non-boxing 手法で表現し、w ビットを必要とする整数値は boxing 手法で表現する。

数値が $w-1$ ビットで表現できる範囲に収まらない場合、カスタムブロックを生成してそのフィールドに数値を格納する。

`real` 型の値および `string` 型の値はカスタムブロックに保持する。レコードはすでに述べたように管理ブロックに保持する。

外部関数

外部関数を表現する情報は、カスタムブロックに保持する。ただし、以下に述べる二点について注意が必要である。

引数の個数 ML における関数は、原理的にただひとつの引数のみをとる。では、つぎのように n 個の引数をとる C 関数は ML 上でどのような型で表現すべきだろうか。

$$\sigma\ \mathtt{f}(\sigma_1\ a_1, \ldots, \sigma_n\ a_n);$$

つぎの二通りの方法が考えられる。ただし、C 上の型 $\sigma, \sigma_1, \ldots, \sigma_n$ に対応する ML 上の型を、$\tau, \tau_1, \ldots, \tau_n$ とする。

- n タプルを引数にとる関数として表現する。

$$\texttt{f} : (\tau_1 \ * \ \ldots \ * \ \tau_n) \to \tau$$

- n 重にネストした関数として表現する。

$$\texttt{f} : \tau_1 \ \to \ \ldots \ \to \ \tau_n \ \to \ \tau$$

現在の実装では後者の方法を採用している。

ただし、この方法では、外部関数が受け取る引数の個数をその型をもとに決定することができないので注意が必要である。たとえば外部関数 f をつぎのように宣言したとする。

$$\texttt{external val } f \texttt{ :int} \to \texttt{int} \to \texttt{int } = \texttt{ imports "}foo\texttt{" of } C$$

この f が、ひとつの int 型の値を受け取って int → int 型の関数 (クロージャ) を返すことを意図しているのか、あるいは、二つの int 型の値を受け取り int 型の値を返すことを意図しているのか、この宣言からは判定できない。むしろ外部ライブラリがいずれかを選択できる方が望ましい。

この点を考慮し、外部関数をつぎのように 3 つの要素から成るデータ構造 (=外部クロージャ) で表現する。

$$ExtCls(fnptr, arity, args)$$

$fnptr$ は外部関数を指すポインタを表す。$arity$ は外部関数が受け取る引数の個数を表す。ただし、この $arity$ は ML 上の関数定義をもとにコンパイラもしくはランタイムが決定するのではなく、外部ライブラリがクロージャを生成する際に決定する。もちろん、ML 上での型が $\tau_1 \to \ \ldots \to \tau_n \to \tau$ である場合、$arity \leq n$ でなければならない。$args$ は、この外部クロージャが保持する引数のリストである。

$ExtCls$ は、external val/fun 宣言あるいは外部関数の呼び出しによって生成される。いずれの場合も、外部ライブラリ中からランタイム関数を呼び出して生成する。

関数適用をおこなう仮想マシン命令 Appterm,Apply の動作を、外部クロージャに対応して次のように定義する。

Code	Accu	Env.	Arg.stack	Return stack	Heap
Appterm; c_0	$ExtCls(f, k, [a_1, \ldots, a_n])$ (where $n = k - 1$)	e_0	$v.s$	r	H
c_0	a (a は外部関数呼び出し $f(a_1, \ldots, a_n, v)$ の返り値)	e_0	s	r	H
Appterm; c_0	$ExtCls(f, k, [a_1, \ldots, a_n])$ (where $n < k - 1$)	e_0	$v.s$	r	H
c_0	$ExtCls(f, k, [a_1, \ldots, a_n, v])$	e_0	s	r	H

Accum 上に外部クロージャ($ExtCls(fnptr, arity, args)$) が置かれている場合、Appterm はつぎのように動作する。$args$ に Arg.stack の先頭に置かれた値を加えた個数が、$arity$ に達するならば、$args$ および Arg.stack 先頭の値を引数として $fnptr$ を呼び出し、その返り値 a を Accum に置く。$arity$ に達しない場合、$fnptr$ および $arity$ と、$args$ に Arg.stack 先頭の値を追加したリストから、新たな外部クロージャを生成し、これを Accum に置く。Apply についても同様である。

セレクタ関数　4.2.3 節に述べたように、polymorphic な型を与えられた変数の出現は、文脈により推論される型から決定されるセレクタ関数をその変数に引数として渡す式に変換される。polymorphic な型を外部関数に与えた場合も同様に、この外部関数の出現は、セレクタ関数を引数として渡す式に変換される。

このセレクタ関数は、関数内部で他の引数からフィールドを取り出す場合に必要となる。しかし、Amethyst は、基本的に外部関数はそのドメインによって生成されるデータのみを扱うことを想定している。したがって、その内部構造については把握しているはずで、セレクタ関数を必要とするケースは、まず無い。また、セレクタ関数の適用順序やその形式はコンパイラの実装の詳細に関する部分である。外部ライブラリがこれに依存するのは適切ではない。このような理由から、外部関数にセレクタ関数を渡すことは現実的ではない。

そこで、セレクタ関数を引数として渡す関数適用を、通常の関数適用とは異なるコード列にコンパイルする。

$$\begin{aligned}\mathcal{T}[\![(M\ S_1\ \ldots\ S_k)]\!] &= \mathcal{C}[\![S_k]\!]; \texttt{Push}; \ldots; \mathcal{C}[\![S_1]\!]; \texttt{Push}; \mathcal{C}[\![M]\!]; \texttt{AppSelTerm}(k)\\ \mathcal{C}[\![(M\ S_1\ \ldots\ S_k)]\!] &= \texttt{Pushmark}; \mathcal{C}[\![S_k]\!]; \texttt{Push}; \ldots; \mathcal{C}[\![S_1]\!]; \texttt{Push}; \mathcal{C}[\![M]\!]; \texttt{AppSel}(k)\end{aligned}$$

`AppSelTerm`,`AppSel` はつぎのように仮想マシンの状態を遷移する。

Code	Accu	Env.	Arg.stack	Return stack	Heap
$\texttt{AppSelTerm}(k); c_0$	$a = Cls(c_1, e_1)$	e_0	$v.s$	r	H
c_1	a	$v.e_1$	s	r	H
$\texttt{AppSelTerm}(k); c_0$	$a = ExtCls(\ldots)$	e_0	$v_1.\ldots.v_k.s$	r	H
c_0	a	e_0	s	r	H
$\texttt{AppSel}(k); c_0$	$a = Cls(c_1, e_1)$	e_0	$v.s$	r	H
c_1	a	$v.e_1$	s	$Cls(c_0, e_0).r$	H
$\texttt{AppSel}(k); c_0$	$a = ExtCls(\ldots)$	e_0	$v_1.\ldots.v_k.\varepsilon.s$	r	H
c_0	a	e_0	s	r	H

`AppSel` および `AppSelTerm` は、Accum に置かれている値が外部関数を指すクロージャ($ExtCls$) ならば、引数スタックの先頭に積まれている k 個のセレクタ関数 (および `AppSel` の場合、スタックマークも) を引数スタックから取り除く。Accum に置かれている値が通常のクロージャであれば、`AppSel` および `AppSelTerm` の動作は `Apply` および `AppTerm` と同様である。

4.4　ドメインモジュール

本章は、ブリッジコードを実装するドメインモジュールについて述べる。まず、ドメインモジュールの実装方法について Amethyst が採る方針について触れる。つぎに、ドメインモジュールがランタイムに対して公開すべきインターフェイスについて説明し、その実装例として `postgres` ドメインを実装するコードを示す。`postgres` ドメインは、PostgreSQL データベースへのアクセスを提供する。最後に、Java のクラスライブラリを Amethyst から使用可能にする `Java` ドメインについて説明する。これは 2.3 節で述べたクラスベース型システムと関数型言語型システムと間の変換手法の応用例でもある。

4.4.1　記述言語の選択

ブリッジコードを記述する言語について、つぎの二つの選択肢がある。

- 関数型言語 (ホスト言語) で記述する。ホスト言語に、ネイティブ言語で記述された任意の関数を呼び出せるプリミティブを用意する。ドメインモジュールはこれを利用してポインタ操作やマーシャリングコードを記述する。Haskell や SMLNJ では、`int` や `char*`など基本的なデータ型のマーシャリングやネイティブな C 関数の呼び出しをのみをプリミティブとして用意し ([FLMJ99b],)、関数型言語でブリッジコードを記述する方法をとっている ([Blu01], [FLMJ99a],[MF01])。

- ネイティブ言語で記述する。まず、ドメインモジュールが定義すべきネイティブ関数の集合を規定する。各ドメインモジュールはこれらの関数を公開し、その中でポインタ操作やマーシャリング処理を記述する。そして抽象機械はこれらの関数を呼び出してフィールド取り出しやタグジャンプをおこなう。Java、Python、Ruby が採用している方法。

既存の関数型言語処理系では前者の方法をとるものが多い。しかし、関数型言語で記述するとはいえ、その内容はポインタ演算やメモリ操作など非関数型言語的な処理が占め、関数型言語で記述するメリットは少ないように思える。そのような処理は、C 言語などシステムプログラミング言語の方が適しているだろう。そこで Amethyst では、ブリッジコードを C 言語などネイティブな言語で記述し、これを動的にランタイムにロードする方法をとっている。

4.4.2 ドメインモジュールのインターフェイス

ドメインモジュールのインターフェイスを共通とすることで、ランタイムは複数のドメインモジュールを一様に扱うことができる。ドメインモジュールは、以下に挙げる関数を実装して Amethyst ランタイムに対して公開しなければならない。これらの関数を**ドメインメソッド**と呼ぶ。

domain initializer ランタイムはドメインに属するいずれかの外部型あるいは外部関数を使用する前に一度だけこの関数を呼ぶ。domain initializer は `domain` 宣言で指定された名前と一致しなければならない。たとえばつぎのようにドメインが宣言されている場合、

```
domain SomeDom = imports "init" of "SomeDom"
```

domain initializer は `init` という名前でなければならない。

domain finalizer ドメインモジュールを解放する直前に、ランタイムはこの関数を呼ぶ。

global name resolver ランタイムは、`external val` および `external fun` によって宣言される変数の値を得るためにこの関数を呼ぶ。

field selector resolver ランタイムは、`external type` 宣言時に、外部型に対応する field selector を得るためにこの関数を呼ぶ。

tag selector resolver ランタイムは、`external type` 宣言時に、外部型に対応する tag selector を得るためにこの関数を呼ぶ。

さらに、ドメインに対して宣言された各外部型について以下の関数を用意する。

field selector 4.2.4 節で述べたように、レコード型の外部型 T の値からフィールド値を取り出す式は仮想機械命令 `ExtGetField` にコンパイルされる。ランタイムは、`ExtGetField` を実行する際に、T が属するドメインの field selector を呼んで T 型の値のフィールド値を得る。

tag selector 4.2.3 節で述べたように、直和型の外部型 T に対する switch 式は仮想機械命令 ExtSwitchTag にコンパイルされる。ランタイムは、ExtSwitchTag を実行する際に、T が属するドメインの tag selector を呼んで T 型の値のタグを得る。

4.4.3 PostgreSQL ドメイン

実例として、PostgreSQL データベースへのアクセスを提供する postgres ドメインをとりあげる。

PostgreSQL ライブラリ

PostgreSQL は、データベースへのインターフェイスを提供するネイティブライブラリ libpq を用意している。

libpq を利用して PostgreSQL データベースからレコードを取得するには、以下の手順にしたがう。

1. ホスト名、データベース名等を引数に PQsetdb を呼び出し、データベース接続を表す PGconn へのポインタを得る。
2. PGconn*とクエリー文を引数として PQexec を呼び出し、レコードセットを表す PGresult へのポインタを得る。
3. PGresult*と、レコードを指定するインデックスおよびフィールドを指定するインデックスを指定して PQgetvalue を呼び出し、フィールド値を得る。フィールド値はすべて文字列として表現されている。フィールドの総数は PQnfields、レコードの総数は PQntuples により取得できる。
4. 使い終わった PGresult*を引数として PQclear を呼ぶ。
5. 使い終わった PGconn*を引数として PQfinish を呼ぶ。

モデル変換

上記で述べたように、libpq から返されたレコードセットからレコードのフィールド値を得るには、レコードを指すインデックスとフィールドを指すインデックスの両方を指定して PQgetvalue を呼ぶ。これは、レコードセットを二次元配列状に表現していることを意味する。アプリケーションは、およそつぎのようなコードでフィールド値を取得する。

```
numfs = PQnfields(rs);
numrs = PQntuples(rs);
for(i = 0;i < numrs;i++){
  for(j = 0;j < numfs;j++){
    doProcess(PQgetvalue(numrs, i, j));
  }
}
```

一方、データベースへのインターフェイスを提供する他のライブラリの多くは、レコードセットをレコードのシーケンシャルなリストとみなすモデルを採用している。この場合、アプリケーションコードは、次のコードのように目的のレコードを指すようカーソルを移動した上で、フィールドを指定して値を取得する。

```
numfs = DBnfields(rs);
```

```
cur = DBBOR(rs);        // BOR = Beginning Of Recordset
while( ! DBEOR(cur)){   // EOR = End Of Recordset
  for(j = 0;j < numfs;j++){
    doProcess(DBgetvalue(cur, j));
  }
  DBmovenext(cur);
}
```

Amethyst においてドメインモジュールが果たす役割を強調するため、以降で示す postgres ドメインモジュールは、libpq が採用する二次元配列モデルをそのまま Amethyst プログラマに提示するのではなく、一般的なシーケンシャルモデルにしたがったインタフェースを提示するよう設計している。

外部宣言

postgres ドメインは、以下で宣言される外部型および関数から構成される。

pglib.ams

```
 1 domain postgres = imports "init" of "pglib";
 2
 3 exception PGerror of string;
 4
 5 external type connection = imports "PGconn" of postgres;
 6 external type 'a dbrec = R of 'a "R" | BOR "BOR" | EOR "EOR"
 7                                imports "PGresult" of postgres;
 8
 9 external fun open : string -> string -> string -> string -> connection
10                                       = imports "open:" of postgres;
11 external fun moveNext : 'a dbrec -> 'a dbrec = imports "movenext:" of postgres;
12 external fun close : connection -> unit = imports "close:" of postgres;
```

dbrec 型がレコードセット上のカーソルを表現している。moveNext は、引数で渡されたカーソルが指しているレコードの次のレコードを指すカーソルを返す。

このほかに、検索対象とするテーブルに対応した外部型と関数をアプリケーション側で個別に宣言する必要がある。たとえば EMPLOYEE テーブルの Name, Rank フィールドを検索するならば、つぎのようにテーブルのスキーマに対応する型と、クエリー文を含む外部名と型を適切に指定した外部関数とを宣言する。

```
external type emprec = {Name:string "S:NAME", Rank:int "I:RANK"}
         imports "RECORD" of postgres;
external fun queryEmployee : connection -> string -> (emprec dbrec option) =
         imports "query:select NAME, RANK from EMPLOYEE where @1" of postgres;
```

レコードセットは、dbrec 型とアプリケーションごとに宣言する外部型 (ここでは emprec) との二つの型によって表現する。つまり、アプリケーション独立の部分とアプリケーション依存の部分とを分けている。

postgres ドメインは、アプリケーション側の外部関数宣言がつぎのような条件を満たすことを前提としている。

- connection と n 個の string を引数にとること。
- 返り値の型が dbrec 型のインスタンスであること。
- 外部名の先頭を query: とし、それ以降にレコードを返すクエリー文がつづくこと。クエリー文中に現れる $@i$ は、引数で渡された n 個の文字列中の i 番目の文字列で置き換えられる。

ドメインモジュール実装

この postgres ドメインを実装するドメインモジュールのコードを以下に示す。

pglib.C

```
 1 #include "config.h"
 2
 3 #include <stdlib.h>
 4 #include <string.h>
 5 #include <postgresql/libpq-fe.h>
 6 #include <string>
 7
 8 #include "amethyst.h"
 9
10 extern "C"{
11   PUBLICFUNCTION T_DOMAININITIALIZER init;
12 }
13
14 static const char* DBREC_TYPENAME = "dbrec";
15 static const char* DBREC_EXNAME = "PGresult";
16 static const char* PGERRORNAME = "PGerror";
17
18 struct Result{
19   AMValue     con;
20   PGresult* pgres;
21   int curpos;
22 };
23
24 int g_tagOfNONE;
25 int g_tagOfSOME;
26
27 Runtime* g_Runtime = NULL;
28 DomainInfo* g_dm = NULL;
29
```

```
30 ////////////////////////////// Utility functions ////////////////////////////
31 static char* strappend(const char* prefix,const char* suffix){
32   // append 'prefix' and 'suffix'
33 }
34 static char* strnappend(const char* prefix,const char* suffix,int length){
35   // append 'prefix' and substring of 'suffix'
36 }
37 static int numCharInStr(const char* str, const char ch){
38   // return the number of ch in str
39 }
40
41 ////////////////////////////// Maintainer //////////////////////////////
42 class ResultMaintainer:public BlockMaintainer{
43 public:
44   void onRelease(BLOCK);
45   void onTrace(BLOCK,PointerUpdater*);
46   static ResultMaintainer* singleton;
47 private:
48   ResultMaintainer(){};
49 };
50 ResultMaintainer* ResultMaintainer::singleton = new ResultMaintainer();
51
52 void ResultMaintainer::onTrace(BLOCK arg, PointerUpdater* tellMeRoots)
53 {
54   Result* res = (Result*)g_Runtime->CRaw(arg);
55   res->con = tellMeRoots->tellMeRoot(res->con);
56 }
57
58 void ResultMaintainer::onRelease(BLOCK arg)
59 {
60   Result* res = (Result*)g_Runtime->CRaw(arg);
61   PQclear(res->pgres);
62 }
63
64 class ConnectionMaintainer:public BlockMaintainer{
65 public:
66   void onRelease(BLOCK);
67   static ConnectionMaintainer* singleton;
68 private:
69   ConnectionMaintainer(){};
70 };
71 ConnectionMaintainer* ConnectionMaintainer::singleton = new ConnectionMaintainer();
```

```
 72
 73 void ConnectionMaintainer::onRelease(BLOCK arg)
 74 {
 75   PQfinish(*(PGconn**)g_Runtime->CRaw(arg));
 76 }
 77
 78 ////////////////////// PostgresSQL specific utility functions ////////////////////
 79
 80 static void raiseError(const char* msg){
 81   // notify error to Amethyst runtime
 82 }
 83
 84 static AMValue DbValToValue(char type, const char* val)
 85 {
 86   switch(type){
 87   case 'I':
 88     return g_Runtime->fromLong(atol(val));
 89   case 'S':
 90     return g_Runtime->fromString(val);
 91   }
 92 }
 93
 94 static char* replaceQuery(const char* tmpl, int numargs, const char** args)
 95 {
 96   // replace '@n' in tmpl with args[n]
 97 }
 98
 99
100 ////////////////////// implementations of external function ////////////////////
101
102 static AMValue openConnection(const T_EXFUNINFO funinfo, int argc, AMValue* args)
103 {
104   const char* hostname = g_Runtime->CString(args[0]);
105   const char* dbname = g_Runtime->CString(args[1]);
106   const char* login = g_Runtime->CString(args[2]);
107   const char* pwd = g_Runtime->CString(args[3]);
108   PGconn* con = PQsetdbLogin(hostname, NULL, NULL, NULL, dbname, login, pwd);
109   if(PQstatus(con) == CONNECTION_BAD){
110     raiseError( PQerrorMessage(con) );
111     return g_Runtime->InvalidValue();
112   }else{
113     AMValue blk = g_Runtime->allocCustomBlock(g_dm, 0, sizeof (PGconn*), &con,
```

```
114                                             ConnectionMaintainer::singleton);
115     return blk;
116   }
117 }
118
119 static AMValue doQuery(const T_EXFUNINFO funinfo, int argc, AMValue* args)
120 {
121   // retrieve db connection out of first argument
122   PGconn* con = *(PGconn**)g_Runtime->CRaw(args[0]);
123
124   // build query string from second argument
125   const char* exname = g_Runtime->getFunExternalName(funinfo);
126   const char* querytmpl = exname + 6;  // 6 = strlen('query:')
127
128   int numargs = argc - 1; // excluding first argument (= connection)
129   const char** conditions = new (const char*)[numargs];
130   for(int i = 0;i < numargs;i++){
131     conditions[i] = g_Runtime->CString(args[i+1]);
132   }
133   char* query = replaceQuery(querytmpl,numargs,conditions); // replace "@n" with arg-n
134   delete[] conditions;
135
136   // send query and receive result
137   PGresult* pgres = PQexec(con, query);
138   free(query);// replaceQuery use strdup
139   if(!pgres || PQresultStatus(pgres) != PGRES_TUPLES_OK){
140     raiseError( PQresultErrorMessage(pgres) );
141     return g_Runtime->allocManagedBlock(g_tagOfNONE, 0, NULL);
142   }else{
143     Result res = {args[0], pgres, -1};
144     AMValue blk =
145       g_Runtime->allocCustomBlock(g_dm,
146                                   rt->getTypeElementIndex(DBREC_TYPENAME,"BOR"),
147                                   sizeof(Result),
148                                   &res,
149                                   ResultMaintainer::singleton);
150     return g_Runtime->allocManagedBlock(g_tagOfSOME, 1, &blk);
151   }
152 }
153
154 static AMValue moveNext(const T_EXFUNINFO funinfo, int argc, AMValue* args)
155 {
```

```
156   int tag,curpos;
157   Result* res = (Result*)g_Runtime->CRaw(args[0]);
158   if(PQntuples(res->pgres) > res->curpos){
159     tag = rt->getTypeElementIndex(DBREC_TYPENAME,"R");
160     curpos = res->curpos+1;
161   }else{
162     tag = rt->getTypeElementIndex(DBREC_TYPENAME,"EOR");
163     curpos = res->curpos;
164   }
165   Result newres = {res->con, res->pgres, curpos};
166   AMValue blk = g_Runtime->allocCustomBlock(g_dm, tag, sizeof(Result), &newres,
167                                             ResultMaintainer::singleton);
168   return blk;
169 }
170
171 static AMValue closeConnection(const T_EXFUNINFO funinfo, int argc, AMValue* args)
172 {
173   PQfinish(*(PGconn**)g_Runtime->CRaw(args[0]));
174   return g_Runtime->allocManagedBlock(0, 0, NULL); // return unit
175 }
176
177 ////////////////////// domain methods ////////////////////
178
179 static AMValue retself(const char*,const char*,AMValue blk,int index)
180 {
181   return blk;
182 }
183
184 static AMValue fieldSelector(const char* name, const char* extname,
185                              AMValue blk, int index)
186 {
187   Result *res = (Result*)g_Runtime->CRaw(blk);
188   const char* attr = g_Runtime->getTypeElementAttribute(name, index);
189   int colindex = PQfnumber(res->pgres, attr+2);
190   const char* val = PQgetvalue(res->pgres, res->curpos, colindex);
191   return DbValToValue(*attr, val);
192 }
193
194 static T_FIELDSELECTOR getFieldSelector(const char* name, const char* exname)
195 {
196   if(!strcmp(exname, DBREC_EXNAME)){
197     return retself;
```

```
198   }else{
199     return fieldSelector;
200   }
201 }
202
203
204 static int tagSelector(const char* name, const char* exname, AMValue blk)
205 {
206   Result* res = (Result*)g_Runtime->CRaw(blk);
207   if(res->curpos < 0){
208     return rt->getTypeElementIndex(DBREC_TYPENAME,"BOR");
209   }else if(res->curpos >= PQntuples(res->pgres)){
210     return rt->getTypeElementIndex(DBREC_TYPENAME,"EOR");
211   }else{
212     return rt->getTypeElementIndex(DBREC_TYPENAME,"R");
213   }
214 }
215
216 static T_TAGSELECTOR getTagSelector(const char* name, const char* exname)
217 {
218   // name must be 'PGresult'
219   return tagSelector;
220 }
221
222 static AMValue getGlobalValue(const char* name, const char* exname){
223   if(!strncmp(exname,"query:",6)){
224     int nPlaceholders = numCharInStr(exname + 6, '@');
225     return g_Runtime->allocCustomClosure(g_dm, name, exname,
226                                          1 + nPlaceholders, // connection + place holders
227                                          doQuery, 0, NULL);
228   }else if(!strncmp(exname,"open:",5)){
229     return g_Runtime->allocCustomClosure(g_dm, name, exname, 4, openConnection, 0, NULL);
230   }else if(!strncmp(exname,"movenext:",9)){
231     return g_Runtime->allocCustomClosure(g_dm, name, exname, 1, moveNext, 0, NULL);
232   }else if(!strncmp(exname,"close:", 6)){
233     return g_Runtime->allocCustomClosure(g_dm, name, exname, 1, closeConnection, 0, NULL);
234   }
235 }
236
237 static void shutdownDomain()
238 {
239   g_dm = NULL;
```

```
240 }
241
242 ////////////////////// domain initializer ////////////////////
243 void init(Runtime* rt, DomainInfo* dm, const char* initarg)
244 {
245   g_Runtime = rt;
246
247   g_dm = dm;
248   g_dm->fieldSelectorResolver = getFieldSelector;
249   g_dm->tagSelectorResolver = getTagSelector;
250   g_dm->getGlobalValue = getGlobalValue;
251   g_dm->shutDown = shutdownDomain;
252
253   g_tagOfNONE = rt->getTypeElementIndex("option","NONE");
254   g_tagOfSOME = rt->getTypeElementIndex("option","SOME");
255 }
256
```

1-29 行目 ヘッダーファイル amethyst.h は、Amethyst ランタイムのインターフェイスやデータ形式定義を含んでいる。とくにランタイム上の値表現形式である AMValue 型の宣言を含んでいる。

Result 構造体は、Amethyst コード上の dbrec 型と、emprec のようにアプリケーションで使用するテーブルごとに定義される型の値を表現する。Amethyst コード上では二つの型は区別されるが、ランタイム上ではこれらの値はおなじ値によって表現される。つまり、emprec dbrec 型の値 v と、(fn (R e) ⇒e) v によって得られる値は、ランタイム上ではまったく同じである。Result の con フィールドは、このレコードセットを生成したデータベース接続への参照を保持する。これは、レコードセットがまだ使用中であるうちに、ガーベジコレクションによってデータベース接続が閉じられてしまうのを防ぐためである。pgres フィールドは実際のレコードセットを保持する。curpos フィールドは、レコードセット中のひとつのレコードを指すインデックスを保持する。

g_Runtime は Amethyst ランタイムを参照するポインタを保持する。g_dm はこのドメインモジュールがランタイムに公開するドメインメソッドを指すポインタのテーブルである。

30-40 行目 strappend、strnappend、numCharInStr の各関数は、文字列に関するユーティリティ関数である。コードは省略する。

41-77 行目 4.3.2 節で述べたように、カスタムブロックには trace ハンドラと release ハンドラが関連づけられる。実際には、onTrace と onRelease の二つのメソッドを持つ BlockMaintainer クラス (あるいはそのサブクラス) のインスタンスを各カスタムブロックと結び付けることで実現する。カスタムブロックを利用する側は、各自で BlockMaintainer のサブクラスを定義して、そのインスタンスをヒープマネージャに渡す必要がある。

ResultMaintainer は dbrec 型の値を保持するカスタムブロックと関連づけられる (145-149 行目、166-167 行目)。Result 構造体は、connection 型の値を保持するブロックへのポインタを con フィールドに保持するので、ResultMaintainer の onTrace(52-56 行目) は、このポインタの所在をランタイムに教える。また、onRelease(58-62 行目) では、不要となったレコードセットを解放する。

ConnectionMaintainer は、connection 型の値を保持するカスタムブロックと関連づけられる (113-114 行目)。onRelease(73-76 行目) で、不要となったデータベース接続を閉じる。前に述べたように、connection 型の値を保持するブロックへの参照を Result が持つので、このデータベース接続から生成されたレコードセットがまだ存在する間に、この関数が呼び出されることはない。

78-99 行目 コードは省略するが、raiseError は PGerror 型の例外を生成してランタイムに通知する。

DbValToValue は、PostgreSQL ライブラリの値表現形式を Amethyst ランタイムの値表現形式に変換する。PostgreSQL ライブラリはすべての値を文字列として表現するので、ここでの処理は簡単である。

replaceQuery は、tmpl 中の "@n" を args[n] の文字列で置換し、データベースに送信するクエリー文字列を生成する。

100-176 行目 ここで定義する関数は、pglib.ams 中で external val 文によって宣言される外部関数を実装する。

openConnection は、open 関数を実装する。

doQuery は、外部名を"query:"で始まる外部関数を実装する。外部名には、データベースに送信するクエリー文字列も含まれる。外部名はランタイムインターフェイスの getFunExternalName 関数によって取得できる (125 行目)。

moveNext は、同名の moveNext 関数を実装する。この関数は、引数で渡された値から取出した Result 構造体を複製し、その curpos フィールドの値をひとつ増やした上でランタイムに返す。

closeConnection は、close 関数を実装する。

177-241 行目 retself 関数は、dbrec 型に対応するセレクタ関数を実装する。dbrec を構成する値構成子中で R だけが引数をとる。retself 関数は、case e_1 of R $v \Rightarrow e_2$ のように R を指定したパターンマッチを実行する際に呼び出され、v に相当する値をランタイムに返す。ここで、実行時に τ dbrec 型の値は τ 型の値と物理表現を共有しているので、retself は引数で渡された dbrec 型の値をそのまま返す。

fieldSelector は、emprec のようにレコードセットを表す外部型に対応するセレクタ関数を実装する。この関数は、Result 構造体の pgres フィールドに保持されている PGresult から、実際のデータベースレコードのフィールド値を取出してランタイムに返す。

getFieldSelector は、このドメインの field selector resolver を実装する。

tagSelector は、dbrec 型に対応する $getTag$ 関数 (4.2.4 節) を実装する。この関数は、Result 構造体の curpos フィールドの値をみて、BOR、EOR、R のうちの適切なタグをランタイムに返す。

getTagSelector は、このドメインの tag selector resolver を実装する。

getGlobalValue は、このドメインの global name resolver を実装する。

shutdownDomain は、このドメインの domain finalizer を実装する。

242-256 行目 init は、このドメインの domain initializer を実装する。引数としてランタイムインターフェイスへのポインタと、ドメインメソッドを格納するテーブル、および domain 文で指定される初期化引数を渡される。

4.4.4 Java ドメイン

2.3 節で述べた方法を応用して、Java のクラスおよびメンバー宣言を Amethyst の宣言に変換する手法について説明する。

クラスによる二重の制約

最初に、Java の型システムにおいてクラスが果たしている役割について考察する。
Java のクラスは、オブジェクト参照に関する操作に対して次の二つの制約を定義している。

関数適用 クラス C への参照を引数にとるメソッドに対し、C およびそのサブクラスへの参照を実引数として渡すことができる。しかし、その他のクラスへの参照を渡すことはできない。

メンバーアクセス その参照を通してアクセスできるメソッドおよびフィールドを定義する。

以下のように Java でクラスを定義したとする。

```
class A{
  public static void sm(A a){...};
}
class B extends A{
  public String Name;
}
class C extends A{}

class D{
  public String Name;
}
```

クラス `B` への参照は、メソッド `void A::sm(A a)` に引数として渡すことができるという点でクラス `C` への参照 (および `A` への参照) と同じ集合に分類することができる。同時に、同一のシグネチャを持つメンバーを公開しているという点では、`B` への参照を `D` への参照と同じ集合に分類することができる。

この二種類の制約を型として別個に表現し、それらをパラメータにとる型として Java のクラスを表現する。

```
type ('c, 'm) JObject
```

型変数 `'c` はクラスの継承関係による制約を表現する。`'m` はメンバーに関する制約を表現する。クラスの継承関係に関する制約は、2.3 節で述べたように、クラスが継承するすべてのスーパークラスおよびインターフェイスをそれぞれ識別するラベルから構成するレコード型により表現する。メンバーに関する制約は、クラスが公開するメンバーそれぞれに対応するフィールドを公開するオブジェクト型により表現する。

この方法で Java の型を ML 上で表現すると、関数適用に関して Java の型システムが課す制約が守られることを保証すると同時に、メンバーアクセスに関して Java が課す制約を「補正」することができる。二つの制約の関係を検討すると、メンバーアクセスに関する制約が合理的な根拠をもつ一方で、関数適用に関する制約は型理論上意義のある根拠を持たず、むしろ、本来許されるべきメンバーアクセス操作を阻む要因となっていることに気づく。たとえば、つぎのコード

```
obj.Name
```

は obj に対して実際に操作を加えているのだから、obj が必ず Name フィールドを持つと保証されるように制約を課すことは必要である。しかし、関数適用においては、実引数を仮引数に束縛する以外に引数に対して何らの操作も加えていない。したがって、実引数として渡すことができる式の型を制約することは本来ならば不要である。不要であるはずの制約を Java が課しているのは、Java の型システムがメンバー単位ではなく、より粗いクラス単位でしか型を識別できないために、関数適用においてクラス単位で制約を課すことによって結果的にメンバーアクセスにおける制約が守られることを保証するという手段をとっているからである。これは 2.1 節で述べた SML の状況と似ている。SML で polymorphic な getName 関数を定義できないのと同様に、Java では、先ほど例示したクラス B への参照とクラス D への参照のどちらも受け入れて Name フィールドを取出すような関数を定義できない。SML におけるこの問題を 2.1 節で解決したのと同様に、ここで示す Java の型から ML の型へのマッピングは、Java オブジェクトに対する「getName」関数を定義可能とする。

Java ライブラリ

つぎのコードは、Java オブジェクトを扱うためのドメインおよび外部型を宣言する。

```
domain Java = imports "init" of "jnilib";
external type ('c, 'm) JObject = JObject of 'm imports "JObject" of Java;
fun JObj (JObject obj) = obj;
```

以降の説明では、以上の宣言があらかじめなされているものと仮定する。

記法

以降での説明では、Java で記述されるコードを $[\![M]\!]_{java}$、Amethyst で記述されるコードを $[\![M]\!]_{ML}$ と表記する。

仮定

以降では Java の構文を用いて説明しているが、以下を仮定する。

- Java クラス名として、つねにパッケージ名で修飾された名前 (FQN) を用いる。たとえば組み込みの Object クラスは java.lang.Object と表記する。
- Java では、各クラスおよびインターフェイスで宣言されたメンバーは、そのサブクラスおよびサブインターフェイスのメンバーに暗黙的に含まれる。ここでは、各クラスおよびインターフェイスはスーパークラスから暗黙的に継承したメンバーも明示的に宣言したものとして扱う。

補助関数

いくつかの補助関数を定義しておく。

classes は引数で渡されたクラスおよびインターフェイスが継承するすべてのクラスおよびインターフェイスの集合を返す。

$$classes([\![\texttt{class}\ C\ \texttt{extends}\ C_p\ \texttt{implements}\ I_1, \ldots, I_n\ \{\ldots\}]\!]_{java})$$

$$= \{C\} \cup classes(C_p) \cup classes(I_1) \cup \ldots \cup classes(I_n)$$

$$classes([\![\texttt{interface}\ I\ \texttt{extends}\ I_p\ \{\ldots\}]\!]_{java})$$
$$= \{I\} \cup classes(I_p)$$

$$classes([\![\texttt{class java.lang.Object}\ \{\ldots\}]\!]_{java})$$
$$= \{\texttt{java.lang.Object}\}$$

$methods$ は、クラスおよびインターフェイスが宣言する public インスタンスメソッドの集合を返す。

$$methods\left(\left[\!\!\left[\begin{array}{l}\texttt{class}\ C\ \ldots\ \{\\ \texttt{public}\ \tau_1\ m_1(\tau_{1,1},\ldots,\tau_{1,k_1})\{\ldots\}\\ \quad\vdots\\ \texttt{public}\ \tau_n\ m_n(\tau_{n,1},\ldots,\tau_{n,k_n})\{\ldots\}\\ \}\end{array}\right]\!\!\right]_{java}\right)$$
$$= \{(m_1,(\tau_{1,1}\ *\ \ldots\ *\ \tau_{1,k_1})\to\tau_1),\ldots,(m_n,(\tau_{n,1}\ *\ \ldots\ *\ \tau_{n,k_n})\to\tau_n)\}$$

$$methods\left(\left[\!\!\left[\begin{array}{l}\texttt{interface}\ I\ \ldots\ \{\\ \texttt{public}\ \tau_1\ m_1(\tau_{1,1},\ldots,\tau_{1,k_1});\\ \quad\vdots\\ \texttt{public}\ \tau_n\ m_n(\tau_{n,1},\ldots,\tau_{n,k_n});\\ \}\end{array}\right]\!\!\right]_{java}\right)$$
$$= \{(m_1,(\tau_{1,1}\ *\ \ldots\ *\ \tau_{1,k_1})\to\tau_1),\ldots,(m_n,(\tau_{n,1}\ *\ \ldots\ *\ \tau_{n,k_n})\to\tau_n)\}$$

スーパークラスおよびスーパーインターフェイスから継承するメソッドもサブクラスおよびサブインターフェイスで明示的に宣言されたものとして扱うので、スーパークラスおよびスーパーインタフェイスに対して再帰的に $methods$ を呼び出す必要はない。

コンストラクタおよびスタティックメソッドについても同様に $constructors, smethods$ を用意する。定義は省略する。

$$constructors[\![\texttt{class}\ C]\!]_{java} = \text{(all public constructors of } C\text{)}$$
$$smethods[\![\texttt{class}\ C]\!]_{java} = \text{(all public static methods of } C \text{ including inherited from super classes)}$$

$fields$ は、クラスおよびインターフェイスが宣言する public インスタンスフィールドの集合を返す。

$$fields\left(\left[\!\!\left[\begin{array}{l}\texttt{class}\ C\ \ldots\ \{\\ \texttt{public}\ \tau_1\ f_1;\\ \quad\vdots\\ \texttt{public}\ \tau_n\ f_n;\\ \}\end{array}\right]\!\!\right]_{java}\right)$$
$$= \{(f_1,\ \tau_1),\ldots,(f_n,\ \tau_n)\}$$

$$
\begin{aligned}
fields\left(\left[\!\!\left[\begin{array}{l}\texttt{interface}\ I\ \ldots\ \{\\ \texttt{public}\ \tau_1\ f_1;\\ \quad\vdots\\ \texttt{public}\ \tau_n\ f_n;\\ \}\end{array}\right]\!\!\right]_{java}\right)\\
= \quad \{(f_1,\ \tau_1),\ldots,(f_n,\ \tau_n)\}
\end{aligned}
$$

$FQN2LBL$ は、パッケージ名で修飾されたクラス名 (FQN) に対応する ML 識別子を返す。

$$FQN2LBL(p_1.\cdots.p_n.C) \quad = \quad p_1'\cdots' p_n'C$$

パッケージ名を連結する'.' は ML では変数名に使用できない。そこで、ML の変数名に使用可能な文字集合のうち、Java のクラス名を構成可能な文字集合に含まれない文字を用いて'.' を置換する。ここでは ′ を選んだ。

$JavaSig$ は、Java の型を引数にとり、JVM 仕様で定義されているようにその型を表現する文字列 (type descriptor) を返す。

$$JavaSig\ :\ \text{Java.type} \rightarrow (\texttt{string})$$

例

$$
\begin{aligned}
JavaSig[\![\texttt{int}]\!]_{java} &\Rightarrow \text{"}\texttt{I}\text{"}\\
JavaSig[\![\texttt{real}]\!]_{java} &\Rightarrow \text{"}\texttt{R}\text{"}\\
JavaSig[\![\texttt{java.lang.string}]\!]_{java} &\Rightarrow \text{"}\texttt{Ljava/lang/String;}\text{"}\\
JavaSig[\![\texttt{java.lang.string (int)}]\!]_{java} &\Rightarrow \text{"}\texttt{(I)Ljava/lang/String;}\text{"}
\end{aligned}
$$

型の変換

A 変換は、Java のメソッドおよびコンストラクタ中の引数に現れる型に対応する ML の型を与える。**R** 変換は、メソッドの返り値として現れる Java の型に対応する ML の型を与える。**A**,**R** のいずれも、ML の型とともに、その型中の自由型変数とカインドを返す。

$$
\begin{aligned}
\mathbf{A} &\ :\ \text{Java.type} \rightarrow (\text{ML.type} * kindedtyvars)\\
\mathbf{R} &\ :\ \text{Java.type} \rightarrow (\text{ML.type} * kindedtyvars)
\end{aligned}
$$

基本型および `String` への参照については次のように簡単に定義できる。

$$
\begin{aligned}
\mathbf{A}[\![\texttt{int}]\!]_{java} = \mathbf{R}[\![\texttt{int}]\!]_{java} &= ([\![\texttt{int}]\!]_{ML}, \emptyset)\\
\mathbf{A}[\![\texttt{java.lang.String}]\!]_{java} = \mathbf{R}[\![\texttt{java.lang.String}]\!]_{java} &= ([\![\texttt{string}]\!]_{ML}, \emptyset)\\
&\ \ \vdots\\
\mathbf{R}[\![\texttt{void}]\!]_{java} &= ([\![\texttt{unit}]\!]_{ML},\ \emptyset)
\end{aligned}
$$

その他の基本型については、[BK99] と同様に Java の `long` は `Int64.int` に、Java の `float` は `Real32.real` にというように対応させればよい。

クラスへの参照型については、引数の型として現れる場合と、結果型に現れる場合とで扱いが異なる。クラス C への参照型を引数にとるメソッドは、クラス C への参照型の値だけではなく、C のサブクラスへの参照型の値も引数にとることができる。一方、クラス C への参照型がメソッドの結果型に現れる場合、そのメソッドが返すのは必ずクラス C への参照型である。(もちろん、その参照が指すオブジェクトはクラス C ではなくそのサブクラスのインスタンスであり得る。)

$$
\begin{array}{rcl}
\mathbf{A}[\![C]\!]_{java} & = & \texttt{let val } id \ = \ FQN2LBL(C) \\
& & \qquad \texttt{val } \alpha \ = \ \text{fresh tyvar} \\
& & \qquad \texttt{val } \beta \ = \ \text{fresh tyvar} \\
& & \texttt{in} \\
& & \quad ([\![(\alpha, \beta)\ \texttt{JObject}]\!]_{ML},\ \{\alpha :: \{\!\{id : \texttt{unit}, \ldots\}\!\}, \beta\}) \\
& & \texttt{end}
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{R}[\![C]\!]_{java} & = & \texttt{let val } \{l_1, \ldots, l_n\} \ = \ \{FQN2LBL(c) | c \in classes(C)\} \\
& & \texttt{in} \\
& & \quad ([\![(\{l_1 : \texttt{unit}, \ldots, l_n : \texttt{unit}\}, C)\ \texttt{JObject}]\!]_{ML},\ \emptyset) \\
& & \texttt{end}
\end{array}
$$

メソッドの変換

M 変換は、Java のコンストラクタ、インスタンスメソッド、クラスメソッドの名前および型を受け取り、ML 上の仕様 (spec) と外部宣言の組を返す。

クラス C のコンストラクタに対し、つぎのように ML の宣言を得る。

$$
\begin{array}{rcl}
\mathbf{M}[\![C(\tau_1, \ldots, \tau_n)]\!]_{java} & = & \texttt{let val } (\sigma_1, Q_1) \ = \ \mathbf{A}[\![\tau_1]\!] \\
& & \qquad \vdots \\
& & \quad \texttt{val } (\sigma_n, Q_n) \ = \ \mathbf{A}[\![\tau_n]\!] \\
& & \quad \texttt{val } (\sigma, Q) \ = \ \mathbf{R}[\![C]\!] \\
& & \quad \texttt{val } javasig \ = \ JavaSig([\![\texttt{void } (\tau_1, \ldots, \tau_n)]\!]_{java}) \\
& & \texttt{in} \\
& & \quad ([\![\texttt{val } (Q_1 \cup \ldots \cup Q_n \cup Q)\ C \ : \ \sigma_1 \to \ldots \to \sigma_n \to \sigma]\!]_{ML}, \\
& & \quad [\![\texttt{external val } (Q_1 \cup \ldots \cup Q_n \cup Q)\ C \ : \ \sigma_1 \to \ldots \to \sigma_n \to \sigma \\
& & \qquad = \ \texttt{imports } \text{"}new : C : javasig\text{"} \ \texttt{of } Java]\!]_{ML}) \\
& & \texttt{end}
\end{array}
$$

クラス C のスタティックメソッド m に対し、つぎのような ML の宣言を得る。

$$
\begin{array}{rcl}
\mathbf{M}[\![\tau\ m(\tau_1, \ldots, \tau_n)]\!]_{java} & = & \texttt{let val } (\sigma_1, Q_1) \ = \ \mathbf{A}(\tau_1) \\
& & \qquad \vdots \\
& & \quad \texttt{val } (\sigma_n, Q_n) \ = \ \mathbf{A}(\tau_n)
\end{array}
$$

$$
\begin{array}{l}
\quad\texttt{val}\ (\sigma, Q)\ =\ \mathbf{R}(\tau) \\
\quad\texttt{val}\ javasig\ =\ JavaSig([\![\tau\ (\tau_1, \ldots, \tau_n)]\!]_{java}) \\
\texttt{in} \\
\quad([\![\texttt{val}\ (Q_1 \cup \ldots \cup Q_n \cup Q)\ m\ :\ \sigma_1 \to \ldots \to \sigma_n \to \sigma]\!]_{ML}, \\
\quad\ [\![\texttt{external val}\ (Q_1 \cup \ldots \cup Q_n \cup Q)\ m\ :\ \sigma_1 \to \ldots \to \sigma_n \to \sigma \\
\qquad =\ \texttt{imports}\ \text{"}smethod : C : m : javasig\text{"}\ \texttt{of}\ Java]\!]_{ML}) \\
\texttt{end}
\end{array}
$$

クラス C のインスタンスメソッド m に対し、つぎのような ML の関数を宣言する。

$$
\begin{array}{rcl}
\mathbf{M}[\![\tau\ m(\tau_1, \ldots, \tau_n)]\!]_{java} & = & \texttt{let val}\ (\sigma_0, Q_0)\ =\ \mathbf{A}(C) \\
& & \quad\texttt{val}\ (\sigma_1, Q_1)\ =\ \mathbf{A}(\tau_1) \\
& & \quad\ \vdots \\
& & \quad\texttt{val}\ (\sigma_n, Q_n)\ =\ \mathbf{A}(\tau_n) \\
& & \quad\texttt{val}\ (\sigma, Q)\ =\ \mathbf{R}(\tau) \\
& & \quad\texttt{val}\ javasig\ =\ JavaSig([\![\tau\ (\tau_1, \ldots, \tau_n)]\!]_{java}) \\
& & \texttt{in} \\
& & \quad([\![\texttt{val}\ (Q_0 \cup Q_1 \cup \ldots \cup Q_n \cup Q)\ m\ :\ \sigma_0 \to \sigma_1 \to \ldots \to \sigma_n \to \sigma]\!]_{ML}, \\
& & \quad\ [\![\texttt{external val}\ (Q_0 \cup Q_1 \cup \ldots \cup Q_n \cup Q)\ m\ :\ \sigma_0 \to \sigma_1 \to \ldots \to \sigma_n \to \sigma \\
& & \qquad =\ \texttt{imports}\ \text{"}method : C : m : javasig\text{"}\ \texttt{of}\ Java]\!]_{ML}) \\
& & \texttt{end}
\end{array}
$$

フィールドの変換

C 変換は Java のクラスが公開するメンバーを表現する ML 上での外部型宣言を返す。

$$
\begin{array}{rcl}
\mathbf{C}[\![\texttt{class}\ C]\!]_{java} & = & \texttt{let val}\ \{(fld_1, sig_1, \tau_1), \cdots, (fld_m, sig_m, \tau_m)\} \\
& & \qquad =\ \{(fld,\ JavaSig([\![\tau]\!]_{java}),\ \mathbf{A}[\![\tau]\!]_{java}) | (fld, \tau) \in fields(C)\} \\
& & \texttt{in} \\
& & \quad([\![\texttt{objecttype}\ C\ =\ \{fld_1 : \tau_1, \cdots, fld_m : \tau_m\}]\!]_{ML}, \\
& & \quad\ [\![\texttt{external type}\ C\ =\ \{fld_1 : \tau_1\ \text{"}field : fld_1 : sig_1\text{"}, \\
& & \qquad\qquad\qquad \cdots, fld_m : \tau_m\ \text{"}field : fld_m : sig_m\text{"}\} \\
& & \qquad\qquad\qquad \texttt{imports}\ \text{"}C\text{"}\ \texttt{of}\ Java]\!]_{ML}) \\
& & \texttt{end}
\end{array}
$$

クラスの変換

以上で示した型およびメンバーに関する変換を用いて、Java のクラスに対応する ML のシグネチャおよびストラクチャをつぎのように得る。

$$
\begin{array}{l}
\mathbf{S}[\![\texttt{class}\ C]\!]_{java} \\
\quad = \quad \texttt{let val}\ (cl^{sig}, cl^{def}) = \mathbf{C}[\![C]\!]_{java} \\
\qquad\quad \texttt{val}\ \{(ctor_1^{sig}, ctor_1^{def}), \cdots, (ctor_k^{sig}, ctor_k^{def})\} = \{\mathbf{M}[\![ctor]\!]_{java} | ctor \in constructors[\![C]\!]_{java}\} \\
\qquad\quad \texttt{val}\ \{(mtd_1^{sig}, mtd_1^{def}), \cdots, (mtd_m^{sig}, mtd_m^{def})\} = \{\mathbf{M}[\![mtd]\!]_{java} | mtd \in methods[\![C]\!]_{java}\} \\
\qquad\quad \texttt{val}\ \{(smtd_1^{sig}, smtd_1^{def}), \cdots, (smtd_n^{sig}, smtd_n^{def})\} = \{\mathbf{M}[\![smtd]\!]_{java} | smtd \in smethods[\![C]\!]_{java}\} \\
\quad \texttt{in} \\
\qquad \left(\left[\!\!\left[\begin{array}{l} \texttt{signature}\ C\ =\ \texttt{sig} \\ \quad cl^{sig} \\ \quad ctor_1^{sig} \\ \quad\quad \vdots \\ \quad ctor_k^{sig} \\ \quad mtd_1^{sig} \\ \quad\quad \vdots \\ \quad mtd_m^{sig} \\ \quad smtd_1^{sig} \\ \quad\quad \vdots \\ \quad smtd_n^{sig} \\ \texttt{end} \end{array} \right]\!\!\right]_{ML} , \left[\!\!\left[\begin{array}{l} \texttt{structure}\ C : C\ =\ \texttt{struct} \\ \quad cl^{def} \\ \quad ctor_1^{def} \\ \quad\quad \vdots \\ \quad ctor_k^{def} \\ \quad mtd_1^{def} \\ \quad\quad \vdots \\ \quad mtd_m^{def} \\ \quad smtd_1^{def} \\ \quad\quad \vdots \\ \quad smtd_n^{def} \\ \texttt{end} \end{array} \right]\!\!\right]_{ML} \right) \\
\quad \texttt{end}
\end{array}
$$

オーバーロード

Java では同一のクラス内で同一の名前をもつメソッドを複数定義できる (=オーバーロード)。しかし、ML はオーバーロードを自由に定義できる仕組みを用意していない。そこで、たとえば以下のような方法で、個々の関数に別個の名前を与えることで対処する。

- C++コンパイラがおこなうように、関数名を引数および返り値の型で修飾する。
- 一クラス内で `m` という名前のメソッドが n 個宣言されているとすると、それぞれにユニークなインデックスを割り振り、ML 内では `m'1`,...,`m'n` と名前を与える。

`null` 値

ML では、τ 型の値はかならず即値あるいは実在するブロックを指す。一方、Java では、クラス C への参照型を結果型とする関数が、実在するオブジェクトへの参照ではなく、無効な参照 (=`null`) を返す場合がある。これに対処するため、[BK99] では、Java のクラス C に対応する ML 型が τ であったとすると、C への参照型を τ `option` と表現する。そして、`null` は `NONE` で表し、オブジェクトを指す有効な参照は `SOME(r)` と表す。ただし、この方法はコーディングが煩雑となる。

そこで、Amethyst は次のような null の表現方法を選択した。

```
external val ('a:{java'lang'Object:'c,...},'b,'c) null : ('a, 'b) JObject =
                                        importing "null" of Java;
external fun ('a:{java'lang'Object:'c,...},'b,'c) isNull : ('a,'b) JObject -> bool =
                                        importing "isnull:" of Java;
```

そして、必要がある場合にかぎり isNull を呼び出して null であるかどうかを確認する。

例

つぎの Java クラス Person を例にとる。

```
class Person{
  public Person(String name, int age){
    this.name = name;
    this.Age = age;
  }
  private String name;
  public String getName(){
    return this.name;
  }
  public void setName(String name){
    this.name = name;
  }
  public int Age;
}
```

$\mathbf{S}[\![\texttt{Person}]\!]_{java}$ により、つぎの ML 宣言を得る。

```
signature Person =
  sig
    objecttype Person = {Age:int}
    val Person : string -> int -> ({java'lang'Object:unit, Person:unit}, Person) JObject
    val ('a:{Person:unit,...}, 'b) getName : ('a,'b) JObject -> string
    val ('a:{Person:unit,...}, 'b) setName : ('a,'b) JObject -> string -> unit
end
structure Person:Person =
  struct
    external type Person = {Age:int "field:Age:I"} imports "Person" of Java;
    external fun Person : string -> int -> ({java'lang'Object:unit,Person:unit}, Person) JObject
                    = imports "new:Person:(Ljava/lang/String;I)V" of Java;
    external fun ('a:{Person:unit,...}, 'b) getName : ('a,'b) JObject -> string
                    = imports "method:Person:getName:()Ljava/lang/String;" of Java;
    external fun ('a:{Person:unit,...}, 'b) setName : ('a,'b) JObject -> string -> unit
```

```
                    = imports "method:Person:setName:(Ljava/lang/String;)V" of Java;
end
```

そして、つぎのように Person クラスを Amethyst 上で扱うことができる[1]

```
>val yamato = Person.Person "YAMATO" 36;
val yamato = ??? : ({java'lang'Object:unit,Person:unit},Person.Person) JObject
>val a = #Age (JObj yamato);
val a = 36 : int
>val n = Person.getName yamato;
val n = "YAMATO" : string
```

つぎのコード例は、本節の最初で述べたように、メンバーアクセスに関して Java が課す制約を「補正」したコーディングを Amethyst が可能とすることを示す。

```
class B{
  public String Name = "B";
}
class D{
  public String Name = "D";
}
```

$\mathbf{S}[\![\texttt{B}]\!]_{java}$ と $\mathbf{S}[\![\texttt{C}]\!]_{java}$ により、次の Amethyst コードを得る。structure と signature は省略する。

```
external type B = {Name:string "field:Name:Ljava/lang/String;"} imports "B" of Java;
external type D = {Name:string "field:Name:Ljava/lang/String;"} imports "D" of Java;

external fun B : unit -> ({java'lang'Object:unit, B:unit}, B) JObject =
         imports "new:B:()V" of Java;
external fun D : unit -> ({java'lang'Object:unit, D:unit}, D) JObject =
         imports "new:D:()V" of Java;
```

B と D のいずれのインスタンスも引数にとることができ、その Name フィールドを取出す関数 getName をつぎのように定義できる。

```
bash-2.04$ amethysti -p BD.ams
       :
>fun getName r = #Name r;
val getName = fn : forall ('a,'b:{Name:'a,...}) => 'b -> 'a
>getName (JObj (B()));
val it = "B" : string
>getName (JObj (D()));
val it = "D" : string
```

つまり、Java オブジェクトに対する「カインドで制約された Polymorphic な関数」を定義している。このような関数を Java で記述することはできない。

[1] 現在の Amethyst はまだ structure を実装していないので、型および関数は実際にはトップレベルで宣言される。

変換スキームの改良

以上で述べた変換スキームに以下の二点で改良を加える。

- インスタンスメソッドをオブジェクトと直接関連づける。
- プロパティをフィールドと同様に扱えるようにする。

これにより、オブジェクト指向言語のコーディングスタイルに似た記述が可能になる。

インスタンスメソッドのフィールドへの追加　以上で示した方法で `Person` クラスを ML 上の宣言に変換した場合、`setName` メソッドを呼び出す際に、次のようにクラス名に対応するストラクチャ名を明示しなければならない。

```
Person.setName obj "YAMATO"
```

一方、Java ではつぎのようなコードで `setName` メソッドを呼び出すことができる。

```
obj.setName("YAMATO");
```

Amethyst でもこれと同様にオブジェクトに対して直接メソッドを指定したい。インスタンスフィールドと同様に、インスタンスメソッドを外部型 C を構成するフィールドに含めることにより、つぎのようなコードで `setName` メソッドを呼び出すことができる。

```
#setName obj "YAMATO"
```

これに対応した変換の手順を以下に示す。

次に示す $\mathbf{M}'$ 変換は、Java のインスタンスメソッドのシグネチャを受け取り、ML 上の型を返す。

$$
\begin{aligned}
\mathbf{M}'[\![\tau\ (\tau_1,\ldots,\tau_n)]\!]_{java} \quad = \quad & \texttt{let val}\ (\sigma_1, Q_1)\ =\ \mathbf{A}(\tau_1) \\
& \qquad \vdots \\
& \texttt{val}\ (\sigma_n, Q_n)\ =\ \mathbf{A}(\tau_n) \\
& \texttt{val}\ (\sigma, Q)\ =\ \mathbf{R}(\tau) \\
& \texttt{in} \\
& \quad [\![\texttt{forall}\ (Q_1 \cup \ldots \cup Q_n \cup Q)\ \Rightarrow\ \sigma_1 \to \ldots \to \sigma_n \to \sigma]\!]_{ML} \\
& \texttt{end}
\end{aligned}
$$

そして、$\mathbf{C}$ 変換をつぎのように修正する。

$$
\begin{aligned}
\mathbf{C}[\![\texttt{class}\ C]\!]_{java} \quad = \quad & \texttt{let val}\ \{(fld_1, sig_1, \tau_1), \cdots, (fld_m, sig_m, \tau_m)\} \\
& \qquad = \{(fld,\ JavaSig([\![\tau]\!]_{java}),\ \mathbf{A}[\![\tau]\!]_{java}) | (fld, \tau) \in fields(C)\} \\
& \quad \texttt{val}\ \{(mtd_1, msig_1, \sigma_1), \cdots, (mtd_n, msig_n, \sigma_n)\} \\
& \qquad = \{(mtd,\ JavaSig([\![\tau]\!]_{java}),\ \mathbf{M}'[\![\tau]\!]_{java}) | (mtd, \tau) \in fields(C)\} \\
& \texttt{in} \\
& \quad ([\![\texttt{objecttype}\ C\ =\ \{fld_1 : \tau_1, \cdots, fld_m : \tau_m, mtd_1 : \sigma_1, \cdots, mtd_n : \sigma_n\}]\!]_{ML}, \\
& \quad [\![\texttt{external type}\ C\ =
\end{aligned}
$$

$$\{fld_1 : \tau_1\ "field : fld_1 : sig_1", \cdots, fld_m : \tau_m\ "field : fld_m : sig_m",$$
$$mtd_1 : \sigma_1\ "method : mtd_1 : msig_1", \cdots, mtd_n : \sigma_n\ "method : mtd_n : msig_n"\}$$
$$\texttt{imports}\ "C"\ \texttt{of}\ Java\rrbracket_{ML})$$

`end`

この変換により、`Person` クラスに対応して次のような ML 宣言が得られる。

```
external type Person = {setName:string->unit "method:setName:(Ljava/lang/String;)V",
                        getName:unit->string "method:getName:()Ljava/lang/String;",
                        Age:int "field:Age:I"}
                         imports "Person" of Java
```

`Person` 型の値 `obj` に対して

$$\#\texttt{setName obj}$$

を実行すると、Java ドメインモジュールは、およそつぎの ML コードで得られるクロージャに相当する外部クロージャを生成してランタイムに返す。

$$\texttt{Person.setName obj}$$

polymorphic な型を持つフィールド　上記の変換によって得られる外部型宣言にはひとつ注意すべき点がある。つぎのように `Person` クラスへの参照型を受け取るメソッドをもつクラス `Student` を例にとる。

```
class Student: public Person{
  public void setSupervisor(Person sv){
    this.sv = sv;
  }
  private Person sv = null;
}
```

このクラスに対応する外部型宣言はつぎのようになる。(簡単のため、Person クラスから継承するメンバーは省略する。)

```
external type Student =
      {setSupervisor:forall('a:{Person:unit,...},'b) => ('a,'b) JObject -> unit
       "method:setSupervisor:(LPerson;)V" } imports "Student" of Java
```

`setSupervisor` フィールドに対して型 $\forall(\alpha :: \{\texttt{Person} : \texttt{unit}, \ldots\}, \beta).(\alpha, \beta)$ `JObject` $\rightarrow$ `unit` が指定されている。これは、`setSupervisor` メソッドが第一引数に関して (制約されているが)polymorphic であることを意味する。現在の SML では、このように polymorphic な関数をフィールドに持つレコードの扱いに制限を設けている。たとえば次のようにレコードから取り出した関数を polymorphic な関数として扱うことができない。

```
- val r = {ID=fn x => x};
val r = <poly-record> : {ID:'a -> 'a}
- val id = #ID r;
stdIn:7.1-7.15 Warning: type vars not generalized because of
```

```
   value restriction are instantiated to dummy types (X1,X2,...)
val id = fn : ?.X1 -> ?.X1
- id 1;
stdIn:8.1-8.5 Error: operator and operand don't agree [literal]
  operator domain: ?.X1
  operand:         int
  in expression:
    id 1
```

このような制限を value restriction と呼ぶ。この制限は、式全体の型が monomorphic である場合は適用されない。次のように引数を与えてやれば結果は `boolean` あるいは `int` のように monomorphic な型となるので、SML はこれを扱うことができる。

```
- (#ID r 1, #ID r true);
val it = (1,true) : int * bool
```

この制限のため、SML の型システムでは、つぎのようにして得られる `setSupervisor` メソッドを polymorphic な関数として扱うことができない。

$$\#\texttt{setSupervisor}\ \texttt{obj}$$

この問題は、[OY99] に示された Rank-1 polymorphism を導入することで解決できる。

プロパティ　1.2.2 で述べたように、Java のコンポーネントフレームワークである JavaBeans の規約では、τ `get`X() というシグネチャのメソッドを、型を τ とするプロパティ X の値を設定するアクセサメソッドとみなす。そして、この規約に準拠した開発ツールでは、プロパティをインスタンスフィールドと同様に扱うことができる。Amethyst でも、Java ドメインモジュールをこの規約に対応させることで、プロパティをインスタンスフィールドと同様に外部型のフィールドに含めることができる。

プロパティに対応したドメインモジュールを用いると、つぎのように外部型 `Person` を宣言できる。

```
external type Person = {Name:string "propget:Person:getName:()Ljava/lang/String;",
                        Age:int "field:Person:Age:Ljava/lang/String;"}
                         imports "Person" of Java
```

これにより、

$$\texttt{val name} \ = \ \#\texttt{getName obj}\ ()$$

と記述する替りに、

$$\texttt{val name} \ = \ \#\texttt{Name obj}$$

と記述して `Name` プロパティの値を取り出すことができる。

第5章　結論と今後の課題

5.1　結論

コンポーネントを活用するアプリケーション構成方法の普及により、スクリプティング言語に対して高い信頼性が要求されている。一方、関数型言語は優れた記述力と安全性をすでに備えているが、外部のコンポーネントとの interoperability に弱点をもつ。本研究は、以下に述べる手法によって関数型言語の開放性を強化し、複雑なアプリケーションの記述にも堪えうるスクリプティング言語の基盤を構築した。

まず、コンポーネントの特徴を表現する「オブジェクト型」を提案し、多相型レコード計算を応用して ML の型システムに組み込んだ。さらに、このオブジェクト型と多相型レコード計算とを応用して、クラスベースのオブジェクト指向言語に見られるサブタイプ関係を関数型言語の型システムの枠内で表現する方法を示した。

次に、オブジェクト型をはじめ外部リソース操作に関する拡張を Standard ML に加えたプログラミング言語 Amethyst を設計した。

最後に、コンポーネントとの連携を念頭において、種々の外部ライブラリの差異を吸収するレイヤー構造や外部リソースへの対応を組み込んだヒープ管理方式などを採用した Amethyst 処理系を実装した。

5.2　今後の課題

実用的なプログラミング言語としての資格を Amethyst 言語が得るためには、本稿で達成した成果ではまだ十分ではない。たとえば以下に挙げる機能を合理的な方法で実現することが必要だろう。

型システム 本稿で対象とした外部ライブラリは、オブジェクトやデータベースレコードなど、もともと関数型言語のレコードと似たデータモデルを採用している。他方、たとえば XML のような非レコード的なデータモデルも存在する。これらのデータモデルを実用上問題無い精度で関数型言語の枠内で再現する方法が求められる。

実装 本稿はコンポーネントを組み合わせるスクリプティング言語として関数型言語を活用することを提案した。次はコンポーネントの記述言語としての可能性を探りたい。[FLMJ99a] は、Haskell を用いた COM オブジェクトの実装を実現している。ここで示されている方法は、適当なドメインモジュールを実装することで Amethyst にも適用できるだろう。しかし、組み込みのレコードあるいはストラクチャと、外部に公開するコンポーネントとを同様の構文で記述できることが望ましい。現在の Amethyst にはモジュール機構が欠けているが、この点を考慮して実装方法を検討したい。

付 録A　Amethyst文法定義

[MTHM97] で示される Standard ML の文法定義にならって、Amethyst の文法規則を示す。

Amethyst は Standard ML の文法規則のうち、以下をサポートしていない。

- `structure` や `signature` などモジュール機構をサポートしていない。したがって、[MTHM97] 中の *longvid* あるいは *longtycon* に替り、*vid* あるいは *tycon* としている。
- `abstype` , `withtype` , `local` をサポートしていない。
- `open` をサポートしていない。かわりに、コンパイラにソースファイルの読み込を指示する `load` 命令を用意している。
- 予約語 `op` による中置演算子から通常の関数識別子への変換をサポートしていない。

Amethyst は以下の構文を新たに導入している。

- 外部リソースに関する宣言 (`domain`,`external type`,`external val`)
- カインドで制約された型変数 (*kindedtyvar*)
- quantify された型を表す `forall`
- コンパイラの挙動を制御するコンパイラコマンド (*cmd*)

式

$$
\begin{array}{rcl}
atexp & ::= & scon \\
& | & vid \\
& | & \{\ \langle exprow \rangle \} \\
& | & \#lab \\
& | & () \\
& | & (exp_1, \cdots, exp_n) \\
& | & [exp_1, \cdots, exp_n] \\
& | & (exp_1; \cdots; exp_n) \\
& | & \texttt{let}\ dec\ \texttt{in}\ exp_1; \cdots; exp_n\ \texttt{end} \\
& | & (exp) \\
exprow & ::= & lab = exp\ \langle, exprow \rangle \\
appexp & ::= & atexp \\
& | & appexp\ atexp
\end{array}
$$

$$
\begin{array}{rcl}
infexp & ::= & appexp \\
& | & infexp_1\ vid\ infexp_2 \\
exp & ::= & infexp \\
& | & exp\ :\ ty \\
& | & exp_1\ \texttt{andalso}\ exp_2 \\
& | & exp_2\ \texttt{orelse}\ exp_2 \\
& | & exp\ \texttt{handle}\ match \\
& | & \texttt{raise}\ exp \\
& | & \texttt{if}\ exp_1\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3 \\
& | & \texttt{while}\ exp_1\ \texttt{do}\ exp_2 \\
& | & \texttt{case}\ exp\ \texttt{of}\ match \\
match & ::= & mrule\ \langle\ |\ match\rangle \\
mrule & ::= & pat\ =>\ exp
\end{array}
$$

宣言

$$
\begin{array}{rcl}
dec & ::= & \texttt{val}\ kindedtyvarseq\ valbind \\
& | & \texttt{val rec}\ kindedtyvarseq\ valbind \\
& | & \texttt{fun}\ kindedtyvarseq\ fvalbind \\
& | & dec_1\ \langle ;\rangle\ dec_2 \\
valbind & ::= & pat = exp\ \langle\texttt{and}\ valbind\rangle \\
fvalbind & ::= & atpat_{11} \cdots atpat_{1n} = exp_1 \\
& & |atpat_{21} \cdots atpat_{2n} = exp_2 \\
& & |\quad \cdots \quad \cdots \\
& & |atpat_{m1} \cdots atpat_{mn} = exp_m \\
& & \langle\texttt{and}\ fvalbind\rangle
\end{array}
$$

トップレベル宣言

$$
\begin{array}{rcl}
topdec & ::= & dec \\
& | & \texttt{datatype}\ datbind \\
& | & \texttt{exception}\ exbind \\
& | & \texttt{infix}\ \langle d\rangle\ vid_1 \cdots vid_n \\
& | & \texttt{infixr}\ \langle d\rangle\ vid_1 \cdots vid_n \\
& | & \texttt{nonfix}\ vid_1 \cdots vid_n \\
& | & \texttt{domain}\ vid = \texttt{imports}\ stringcon\ \langle\texttt{with}\ stringcon\rangle\ \texttt{of}\ stringcon \\
& | & \texttt{external type}\ tyvarseq\ tycon = \ \texttt{imports}\ stringcon\ \texttt{of}\ vid \\
& | & \texttt{external type}\ tyvarseq\ tycon = \ \ extconbind\ \texttt{imports}\ stringcon\ \texttt{of}\ vid
\end{array}
$$

	`external type` *tyvarseq tycon* $=$ $\{$*exttyrow*$\}$ `imports` *stringcon* `of` *vid*
	`external val` *vid* : *ty* $=$ `imports` *stringcon* `of` *stringcon*
	`external fun` *vid* : *ty* $=$ `imports` *stringcon* `of` *stringcon*
datbind	::= *tyvarseq tycon* = *conbind*
conbind	::= *vid* ⟨`of` *ty*⟩ ⟨| *conbind*⟩
exbind	::= *vid* ⟨`of` *ty*⟩
extconbind	::= *vid* ⟨`of` *ty*⟩ ⟨*stringcon*⟩ ⟨| *extconbind*⟩
exttyrow	::= *lab* : *ty* ⟨*stringcon*⟩ ⟨, *exttyrow*⟩

パターン

atpat	::=	_
	|	*scon*
	|	*vid*
	|	{ ⟨*patrow*⟩ }
	|	()
	|	($pat_1, cdots, pat_n$)
	|	[$pat_1, cdots, pat_n$]
	|	(*pat*)
patrow	::=	...
	|	*lab* = *pat* ⟨ , *patrow* ⟩
	|	*vid*⟨:*ty*⟩ ⟨`as` *pat*⟩ ⟨, *patrow*⟩
pat	::=	*atpat*
	|	*vid atpat*
	|	pat_1 *vid* pat_2
	|	*pat* : *ty*
	|	*vid*⟨: *ty*⟩ `as` *pat*

型式

ty	::=	*tyvar*
	|	{ ⟨*tyrow*⟩ }
	|	*tyseq tycon*
	|	$ty_1 * \cdots * ty_n$
	|	*ty* $->$ *ty*′
	|	`forall` *kindedtyvarseq* $=>$ *ty*
	|	(*ty*)

$$
\begin{array}{rcl}
kindedtyvar & ::= & tyvar\ \langle : \{\ tyrow\ \langle\ , \ldots\ \rangle\ \}\ \rangle \\
tyrow & ::= & lab : ty\ \langle, tyrow \rangle
\end{array}
$$

コンパイラコマンド

$$
\begin{array}{rcl}
cmd & ::= & : vid\ cmdarg_1\ \cdots\ cmdarg_n \\
cmdarg & ::= & vid \\
& | & stringcon
\end{array}
$$

プログラム

$$
\begin{array}{rcl}
prog & ::= & stmt_1\ \langle ; \rangle\ \cdots\ \langle ; \rangle\ stmt_n \\
stmt & ::= & exp \\
& | & topdec \\
& | & cmd
\end{array}
$$

付 録B　ZINC抽象機械

Amethyst の抽象機械命令セットは、ZINC 抽象機械 [Ler90] の命令セットをもとにしている。本編の理解を助けるため、本章は ZINC 抽象機械について説明する。

B.1　ZINC 抽象機械の構成

ZINC 抽象機械は以下の 5 つの要素で構成される。

Code 命令ポインタ。つぎに実行するコードを指す。

Accu アキュムレータ。計算の中間結果を保持する。

Env ローカル変数を保持するスタック。最後に束縛された変数が先頭に現れている。

Arg.stack 関数適用の引数を保持するスタック。

Return stack 関数呼び出しのリターン先の `Code,Env,Return Stack` の組を保持するスタック。

Amethyst の実装では、そのほかにヒープおよびグローバル変数環境なども抽象機械の状態を構成している。

抽象機械上で操作される値をつぎのように定義する。

$$\begin{aligned} v \quad ::= \quad & Int(n) \\ \mid \quad & Cls(c, e) \end{aligned}$$

$Int(n)$ は整数値を表す。$Cls(c, e)$ は、コード列 c と環境 e の組から成るクロージャを表す。

B.2　コンパイル

ラムダ式から ZINC 抽象機械命令へのコンパイルと、抽象機械命令の仕様について説明する。

ソース言語は、つぎのような基本的なラムダ式である。

$$\begin{aligned} e \quad ::= \quad & n \\ \mid \quad & x \\ \mid \quad & \lambda x.e \\ \mid \quad & e \ \ldots \ e \\ \mid \quad & p(e, \cdots, e) \\ \mid \quad & \texttt{let val } x \ = \ e \texttt{ in } e \texttt{ end} \\ \mid \quad & \texttt{letrec val } x \ = \ e \texttt{ in } e \texttt{ end} \end{aligned}$$

関数適用が複数の引数を含みうることに注意。$p(e,\cdots,e)$ はプリミティブ関数呼び出しを指す。ただし、$+(e_1,e_2)$ や $-(e_1,e_2)$ を e_1+e_2 および e_1-e_2 と表記する場合がある。また、`let` は非再帰的なローカル変数定義、`letrec` は再帰を含むローカル変数定義を表す。

さらに、ソースプログラムはコンパイル前に de Brujn 記法に変換されるものとする。つまり、変数は名前ではなく、その変数を参照している個所とその変数を束縛している個所との間で束縛されている変数の数によって表現される。たとえば

$$\lambda x.x(\lambda y.xy(\lambda z.xyz))$$

は

$$\lambda.0(\lambda.1\ 0(\lambda.2\ 1\ 0))$$

と表現する。よって、以降の説明では、つぎのように変数をインデックス i により Li と表記する式をコンパイル対象とする。

$$\begin{array}{rcl} e & ::= & n \\ & | & Li \\ & | & \lambda e \\ & | & e\ \ldots\ e \\ & | & p(e,\cdots,e) \\ & | & \texttt{let}\ \ L0\ =\ e\ \texttt{in}\ e\ \texttt{end} \\ & | & \texttt{letrec}\ \ L0\ =\ e\ \texttt{in}\ e\ \texttt{end} \end{array}$$

ラムダ式は、その式が現れる位置に応じて、二通りの方法のいずれかでコンパイルされる。$\mathcal{T}[\![E]\!]$ は、関数本体内の最後の式、すなわち、その値がそのまま関数の返り値となるような式 E を効率のよいコードにコンパイルする。$\mathcal{C}[\![E]\!]$ はそれ以外の状況で現れる式をコンパイルする。

抽象機械命令が抽象機械の状態を遷移する様子をつぎのような形式で記述する。

Code	Accu	Env.	Arg.stack	Return stack
$c_0;C$	a	e	s	r
C'	a'	e'	s'	r'

これは、抽象機械の状態が表の上行に示す状態であるとき、抽象機械は命令 c_0 を実行することによって表の下行に示す状態に遷移することを意味する。Code 列の $c_0;C$ は、コード列中で命令ポインタが指す位置に命令 c_0 が位置し、その後続が C であることを意味する。Env、Arg.stack、Return stack はいずれもスタック形式のデータ構造であり、先頭要素が v で後続の要素列が s であることを

$$v.s$$

と表記する。以降ではヒープを操作する命令を扱わないのでヒープに関する状態遷移は記述しないが、本編ではヒープに関する状態を次のように表記している。

$$H\{b \mapsto C[v_1,\cdots,v_n]\}$$

これは、ヒープ H 中でアドレス b が指す位置に、タグを C とし、$v_1,\cdots,v_n$ をフィールドとするブロックが存在することを表す。

B.2.1 定数

整数定数式 n は次のコード列にコンパイルする。

$$\mathcal{T}[\![n]\!] = \mathcal{C}[\![n]\!] = \texttt{ConstInt}(n)$$

`ConstInt`

Code	Accu	Env.	Arg.stack	Return stack
$\texttt{ConstInt}(n); c$	a	e	s	r
c	$Int(n)$	e	s	r

`ConstInt`(n) は、抽象機械上で整数値 n を表す値 $Int(n)$ を Accum に置く。

B.2.2 ローカル変数

インデックスを i とする変数を参照する式 Li は、次のようにコンパイルされる。

$$\mathcal{T}[\![Li]\!] = \mathcal{C}[\![Li]\!] = \texttt{Access}(i)$$

`Access`

Code	Accu	Env.	Arg.stack	Return stack
$\texttt{Access}(i); c$	a	$e = v_0 \ldots v_i \ldots$	s	r
c	v_i	e	s	r

`Access` は、環境の先頭から i 番目 (先頭は 0 番) の値をアキュムレータに置く。

B.2.3 関数適用

ラムダ計算では、すべての関数は引数をひとつだけとる。したがって関数適用についても、ひとつの引数につき一回の関数適用をおこなうものとされる。たとえば $f\ a\ b$ という式は、a への f の適用と、$f\ a$ により得られた値 (=クロージャ) の b への適用の二回の関数適用から成る。

SECD machine のような単純な抽象機械は、このように連続する関数適用を、それぞれ独立した関数適用として扱う。このため $f\ a\ b$ を評価する際には、最初の関数適用によりクロージャを生成し、直後に、そのクロージャを次の関数適用で展開する。

一方 ZINC は、連続する関数適用の関連性を考慮し、複数の引数を含む関数適用式を一括してコンパイルするとともに、中間的なクロージャを生成しないよう最適化した命令を ZINC 抽象機械に用意している。

$$\begin{aligned}
\mathcal{T}[\![(M N_1 \ldots N_k)]\!] &= \mathcal{C}[\![N_k]\!]; \texttt{Push}; \ldots; \mathcal{C}[\![N_1]\!]; \texttt{Push}; \mathcal{C}[\![M]\!]; \texttt{Appterm} \\
\mathcal{C}[\![(M N_1 \ldots N_k)]\!] &= \texttt{Pushmark}; \mathcal{C}[\![N_k]\!]; \texttt{Push}; \ldots; \mathcal{C}[\![N_1]\!]; \texttt{Push}; \mathcal{C}[\![M]\!]; \texttt{Apply}
\end{aligned}$$

Appterm/Apply

Code	Accu	Env.	Arg.stack	Return stack
Appterm; c_0	$a = Cls(c_1, e_1)$	e_0	$v.s$	r
c_1	a	$v.e_1$	s	r
Apply; c_0	$a = Cls(c_1, e_1)$	e_0	$v.s$	r
c_1	a	$v.e_1$	s	$Cls(c_0, e_0).r$

Appterm と Apply は、以下のように動作する。

1. Apply は、コードポインタと環境からクロージャを生成して、リターンスタックにプッシュする。このクロージャは、Apply によって呼び出された関数が終了した際に復元される。つまり、Apply の次のコード (=c_0 の先頭) から実行が再開される。
   一方、Appterm はリターンスタックを更新しない。したがって、Appterm により呼び出された関数が終了すると、c_0 に戻るのではなく、最後に Apply が実行された地点にリターンする。たとえば、関数 f が Apply によって g を呼び、g がその末尾において h を Appterm により呼び出したとすると、h が終了した際には、f が g を呼び出した時点のクロージャが復元される。もともと g は h の返り値をそのまま返すだけなので、このような最適化をおこなってもプログラムの意味は変わらない。
2. アキュムレータに保持されているクロージャ(=M を評価して得られたもの) を展開し、コードポインタと環境を置き換える。
3. 先頭の引数 (=N_1 の評価結果) を引数スタックからポップして環境の先頭に加える。(これにより、関数本体をコンパイルする度に先頭に Grab を挿入しなくてすむ。)

Push

Code	Accu	Env.	Arg.stack	Return stack
Push; c_0	a	e	s	r
c_0	a	e	$a.s$	r

Push は、アキュムレータに置かれている値 (=直前の式を評価して得られた値) を引数スタックの先頭に加える。

Pushmark

Code	Accu	Env.	Arg.stack	Return stack
Pushmark; c_0	a	e	s	r
c_0	a	e	$\varepsilon.s$	r

Pushmark は、現在評価している関数呼び出しの引数がここまでであることを示すマーク (=ε) を引数スタックの先頭に加える。

たとえば $g = \lambda x.\lambda y.(x + y)$ と定義された環境で、$(f(gN_1)N_2))$ という式をコンパイルして実行する。すると、g を呼び出す時点で引数スタックには N_1, N_2 が積まれている。ここで N_1 だけではなく N_2 も g に渡されてしまうのを防ぐために、N_1 と N_2 の間にマークを挟み、この g への呼び出しの引数が N_1 だけであることを認識できるようにする。

関数適用が $\mathcal{T}$ でコンパイルされる場合、その関数適用は $\lambda(M N_1 \ldots N_k)$ のように関数抽象によって囲まれているはずであり、$((M N_1 \ldots N_k)O_1, O_2 \ldots)$ のようにその関数適用を別の関数適用が直接囲むことはないので、Pushmark は必要ない。

B.2.4 関数抽象

$$\begin{aligned}\mathcal{T}[\![\lambda E]\!] &= \texttt{Grab};\mathcal{T}[\![E]\!]\\ \mathcal{C}[\![\lambda E]\!] &= \texttt{Cur}(\mathcal{T}[\![E]\!];\texttt{Return})\end{aligned}$$

Cur

Code	Accu	Env.	Arg.stack	Return stack
$\texttt{Cur}(c_1);c_0$	a	e	s	r
c_0	$Cls(c_1,e)$	e	s	r

Cur は、関数本体の先頭を指すコードポインタと現在の環境からクロージャを生成してアキュムレータに置く。

Grab

Code	Accu	Env.	Arg.stack	Return stack
$\texttt{Grab};c_0$	a	e_0	$\varepsilon.s$	$Cls(c_1,e_1).r$
c_1	$Cls(c_0,e_0)$	e_1	s	r
$\texttt{Grab};c$	a	e	$v.s$	r
c	a	$v.e$	s	r

Grab は、実質的には、複数の引数をとる関数を適用する際に、二番目以降の引数を引数スタックから環境に移すはたらきをする。「実質的には」としたのは、複数の引数をとる関数適用は ZINC 独自の拡張であるため。本来のラムダ計算の定義に従えば、入れ子の内側に位置するラムダ関数を適用することを意味する。

Grab が使用されるのは、λE が $\mathcal{T}$ によりコンパイルされる場合のみである。そして、λE が $\mathcal{T}$ によってコンパイルされるのは、

- λE が、別の関数の末尾に位置している、すなわち $\lambda\lambda E$ という形である
- λE が、それ自体が $\mathcal{T}$ によってコンパイルされる let 式の値となっている、すなわち $\texttt{let}\, L0 = N\, \texttt{in}\, \lambda\, E\, \texttt{end}$ という形である

のいずれかの場合に限られる。

いずれにしても、この λE への呼び出しは表面上（ソースコード上）には現れない。たとえば、$((\lambda x.\lambda y.x+y)N_1\ N_2)$ 中の内側の関数 $\lambda y.x+y$ への呼び出しは、外側の関数 $(=\lambda x.\ldots)$ に対する関数適用を評価した結果発生するものであり、コンパイル時には明らかではない。Appterm・Apply が用いられるのは、ソースコード上に現れている関数適用をコンパイルする場合に限られる。したがって、この λE が呼び出される際のコードには Appterm・Apply は使用されない。そこで、Appterm・Apply のかわりに、引数を引数スタックから取り出して環境に追加する処理を、E が実行される直前におこなう必要がある。その役割を果たすのが Grab である。すぐ後で述べるように、Return も同様の役割を果たす。

呼び出される際の状況がコンパイル時に明らかではないので、実行時にこの関数の先頭にコードポインタが達した時点で、引数が与えられていない場合がありえる。たとえば $((\lambda x.\lambda y.x+y)N_1)$ を実行すると、コードポインタが $\lambda y.x+y$ の先頭の Grab に達した時点で、引数スタックの先頭には、外側の関数適用の先頭の Pushmark によってプッシュされたマークが現れている。これは $\lambda y.x+y$ に対する引数がないこと

を意味している。$\lambda x.\lambda y.x+y$ を二引数をとる一つの関数としてみると、この状況は関数の部分適用にあたる。このような場合、Grab は関数本体の先頭を指すコードポインタと環境（関数内に現れる自由変数のその時点での束縛を含んでいる）からクロージャを生成し、それをこの関数呼び出しの値としてアキュムレータに置き、呼び出し元にリターンする。ここで生成されるクロージャは Grab の次のコードを指している。ここで、この Grab の次のコードは、Grab が常に引数を引数スタックから環境に移すと仮定している。したがって、このクロージャを復元して実行する際には、Grab の次のコードに移る前に、Grab の代理として引数を引数スタックから環境に移しておくことが必要になる。Appterm, Apply, Return がこれをおこなう。

Return

Code	Accu	Env.	Arg.stack	Return stack
Return; c_0	a	e_0	$\varepsilon.s$	$Cls(c_1,e_1).r$
c_1	a	e_1	s	r
Return; c_0	$a=Cls(c_1,e_1)$	e_0	$v.s$	r
c_1	a	$v.e_1$	s	r

引数スタックに引数が残っていない場合、すなわち引数スタックの先頭にマークが現れている場合、Return は呼び出し元への復帰動作をおこなう。つまり、リターンスタックの先頭のクロージャを取り出して、この呼び出しのリターン先のコードポインタと環境を復元する。Appterm/Apply で説明したように、リターン先は最後に Apply が実行された地点であり、この関数を直接呼び出した地点とは限らない。

引数スタックに引数が残っており、かつ関数本体の評価結果が関数である場合、Return は呼び出し元へ戻らず、引き続き次の関数呼び出しをおこなう。$((\lambda x.x)(\lambda y.M)N)$ を例にとると、最初の関数適用、すなわち $(\lambda x.x)(\lambda y.M)$ を実行すると、関数の本体 $(=\lambda x.x)$ が評価された後、$\lambda y.M$ を指すクロージャがアキュムレータに置かれ、引数スタックの先頭に N の値が現れた状態で、Return に達する。この場合、Return は、Grab を説明する際に述べた「表面上明らかではない」関数呼び出しをおこなう。すなわち、アキュムレータ上のクロージャを展開してコードポインタと環境を置き換え、引数スタックの先頭要素をポップして環境の先頭に加える。これにより、$(\lambda y.M)N$ が実行される。Return 命令が実行される際に、引数スタックに引数が残っていて、かつアキュムレータにクロージャ以外の値が置かれている状況はありえない。そのような状況が発生するのは (1 2) のように関数ではない値を適用しようとした場合であり、それはソースコードのバグである。

B.2.5 let 式

$$\begin{aligned}\mathcal{T}[\![\texttt{let } L0=N \texttt{ in } M \texttt{ end}]\!] &= \mathcal{C}[\![N]\!];\texttt{Let};\mathcal{T}[\![M]\!]\\ \mathcal{C}[\![\texttt{let } L0=N \texttt{ in } M \texttt{ end}]\!] &= \mathcal{C}[\![N]\!];\texttt{Let};\mathcal{C}[\![M]\!];\texttt{Endlet}\end{aligned}$$

再帰を含まない let 式は $((\lambda x.M)N)$ と同じ意味を持つが、頻繁に使われるので、実際の関数適用より高速で簡潔なコンパイル方式を用意した。

Let

Code	Accu	Env.	Arg.stack	Return stack
Let; c	a	e	s	r
c	a	$a.e$	s	r

Let は、アキュムレータに置かれている値を環境の先頭に追加する。つまり、Push;Grab に等しい。したがって、M 中では let で束縛された変数を関数の束縛変数と同様に扱うことができる。

Endlet

Code	Accu	Env.	Arg.stack	Return stack
Endlet; c	a	$v.e$	s	r
c	a	e	s	r

Endlet は、環境から先頭要素を取り除く。これは、Let によって環境に追加された値を取り除くことを意味する。

let 式が $\mathcal{T}$ によってコンパイルされた場合、Endlet は生成されない。これは、関数が let 式で終わっていること、および関数内で生成された環境は関数の外のコードからは参照されないことから、環境を元に戻す必要がないためである。

$$\begin{aligned}\mathcal{T}[\![\texttt{letrec } L0 = N \texttt{ in } M \texttt{ end}]\!] &= \texttt{Dummy}; \mathcal{C}[\![N]\!]; \texttt{Update}; \mathcal{T}[\![M]\!] \\ \mathcal{C}[\![\texttt{letrec } L0 = N \texttt{ in } M \texttt{ end}]\!] &= \texttt{Dummy}; \mathcal{C}[\![N]\!]; \texttt{Update}; \mathcal{C}[\![M]\!]; \texttt{Endlet}\end{aligned}$$

let 式では、変数の再帰的定義、すなわちその変数自身を自由変数として含む項に変数を束縛することができない。変数を束縛するには項を評価する必要があり、項を評価する際にはその項の自由変数が環境で束縛されている必要がある。したがって、再帰的定義においては、変数の束縛と項の評価が互いを必要としあう循環的な状況が発生する。項を評価した後で変数を束縛する let 式では、この状況を解決することはできない。

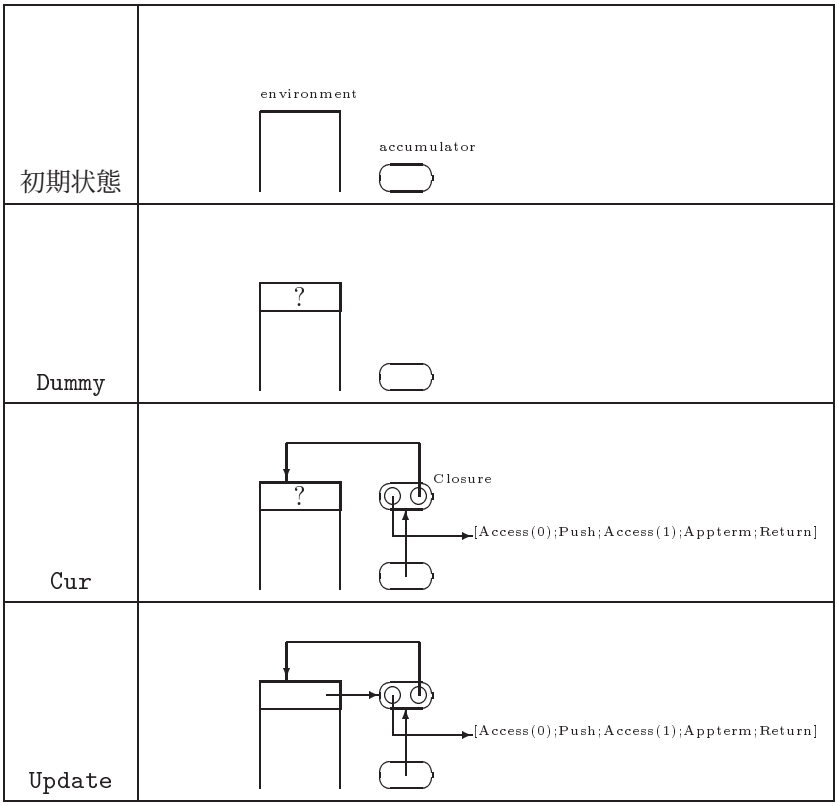
このため、ZINC 抽象機械は let とは別に letrec を用意している。letrec は、項を評価する前に Dummy によって変数を仮に束縛しておき、項を評価し終わったあとで Update によって変数を束縛しなおすことにより再帰的定義を実現する。たとえば、

$$\texttt{letrec val } f = \lambda x.(f\ x) \texttt{ in } E \texttt{ end}$$

をコンパイルすると

$$\texttt{Dummy;Cur(Access(0);Push;Access(1);Appterm;Return);Update;}\ldots$$

というコードを得る。このコードを実行した際の様子を以下の図で示す。



Dummy

Code	Accu	Env.	Arg.stack	Return stack
Dummy; c	a	e	s	r
c	a	$?.e$	s	r

Dummy は、環境の先頭にダミーの値を追加する。これによって環境中に割り当てられたエントリーは、後で Update によって更新される。

Update

Code	Accu	Env.	Arg.stack	Return stack
Update; c	a	$e = v.e_1$	s	r
c	a	$e[v \leftarrow a]$	s	r

環境の先頭要素の内容を、アキュムレータに置かれた値 (a) で更新する。ここで、環境の先頭に置かれていた値 ($=v$) は、Dummy によって環境に加えられたものである。

また、a がクロージャであった場合、そのクロージャに含まれているコードは、e 中の v が置かれていた場所を参照する可能性がある。その場合、v を a に更新したことによって、a が a 自身への参照を含むことになる。これは a が再帰的に定義されたことを意味する。

B.2.6 プリミティブ関数

$$\mathcal{T}[\![p(M_1 \ldots M_k)]\!] = \mathcal{C}[\![p(M_1 \ldots M_k)]\!] = \mathcal{C}[\![M_k]\!]; \mathtt{Push}; \ldots; \mathcal{C}[\![M_2]\!]; \mathtt{Push}; \mathcal{C}[\![M_1]\!]; \mathtt{Prim}(p)$$

+や=などのプリミティブ演算子の適用は Prim によって実行される。

Prim

Code	Accu	Env.	Arg.stack	Return stack
$\texttt{Prim}(p); c$	a	e	$v_2 \ldots v_k.s$	r
c	$p(a, v_2, \ldots, v_k)$	e	s	r

Prim は最初の引数をアキュムレータから取り出し、残りの引数を引数スタックから取り出して演算をおこない、その結果をアキュムレータに置く。

ユーザ定義関数の場合、必要とする引数がスタックに用意されていなければ、Grab 命令がクロージャを生成して関数の実行を休止する。一方、Prim は、適用するプリミティブ演算子が必要とする個数 ($=k$) の引数がアキュムレータおよびスタック上に用意されていることを前提としている。コンパイラは、正しい個数の引数がプリミティブ演算子に渡せられるよう適切なコードを生成しなければならない。

B.3 コンパイル／実行例

いくつかのソースプログラムを上記の方式でコンパイルした結果と、それを ZINC 抽象機械で実行した様子を示す。

それぞれの例について、つぎの 3 項目を示す。

- ソースプログラムと、それを de brujn 記法で表記したラムダ式
- コンパイルによって得られるコード列。
- コード列を ZINC 抽象機械で実行した際に機械状態が遷移する様子

B.3.1 関数適用

プログラム

$$\begin{aligned} \textbf{lambda} \quad & ((\lambda x.\lambda y.x - y)\ 3\ 2) \\ \textbf{de brujn} \quad & (\lambda(\lambda(L0 - L1))\ 3\ 2) \end{aligned}$$

コンパイル結果

$$\begin{aligned} &[\texttt{Pushmark}; \texttt{ConstInt}(2); \texttt{Push}; \texttt{ConstInt}(3); \texttt{Push}; \\ &\quad \texttt{Cur}[\texttt{Grab}; \texttt{Access}(1); \texttt{Push}; \texttt{Access}(0); \texttt{Prim}(\text{"}-\text{"}); \texttt{Return}]; \texttt{Apply}] \end{aligned}$$

実行トレース

Code	Accu	Env	Arg.stack	Return stack
`Pushmark`	$Int(0)$	[]	[]	[]
`ConstInt(2)`	$Int(0)$	[]	$[\varepsilon]$	[]
`Push`	$Int(2)$	[]	$[\varepsilon]$	[]
`ConstInt(3)`	$Int(2)$	[]	$[Int(2), \varepsilon]$	[]
`Push`	$Int(3)$	[]	$[Int(2), \varepsilon]$	[]
`Cur`	$Int(3)$	[]	$[Int(3), Int(2), \varepsilon]$	[]
`Apply`	$Cls(\texttt{Grab}, [...])$	[]	$[Int(3), Int(2), \varepsilon]$	[]
`Grab`	$Cls(\texttt{Grab}, [...])$	$[Int(3)]$	$[Int(2), \varepsilon]$	$[Cls(\$, [...])]$
`Access(1)`	$Cls(\texttt{Grab}, [...])$	$[Int(2), Int(3)]$	$[\varepsilon]$	$[Cls(\$, [...])]$
`Push`	$Int(3)$	$[Int(2), Int(3)]$	$[\varepsilon]$	$[Cls(\$, [...])]$
`Access(0)`	$Int(3)$	$[Int(2), Int(3)]$	$[Int(3), \varepsilon]$	$[Cls(\$, [...])]$
`Prim(−)`	$Int(2)$	$[Int(2), Int(3)]$	$[Int(3), \varepsilon]$	$[Cls(\$, [...])]$
`Return`	$Int(-1)$	$[Int(2), Int(3)]$	$[\varepsilon]$	$[Cls(\$, [...])]$
\$	$Int(-1)$	[]	[]	[]

B.3.2 部分適用

プログラム

lambda $((\lambda x.\lambda y.x - y)\ 3)$

de brujn $(\lambda(\lambda(L0 - L1))3)$

コンパイル結果

[`Pushmark`; `ConstInt(3)`; `Push`; `Cur`[`Grab`; `Access(1)`; `Push`; `Access(0)`; `Prim(" − ")`; `Return`]; `Apply`]

実行トレース `Grab` は、引数スタックの先頭にマークが現れているので、現時点のクロージャを作成してアキュムレータに置き、リターンスタックからクロージャを展開してリターンしている。

Code	Accu	Env	Arg.stack	Return stack
`Pushmark`	$Int(0)$	[]	[]	[]
`ConstInt(3)`	$Int(0)$	[]	$[\varepsilon]$	[]
`Push`	$Int(3)$	[]	$[\varepsilon]$	[]
`Cur`	$Int(3)$	[]	$[Int(3), \varepsilon]$	[]
`Apply`	$Cls(\texttt{Grab}, [...])$	[]	$[Int(3), \varepsilon]$	[]
`Grab`	$Cls(\texttt{Grab}, [...])$	$[Int(3)]$	$[\varepsilon]$	$[Cls(\$, [...])]$
\$	$Cls(\texttt{Access}(1), [...])$	[]	[]	[]

B.3.3 関数を返す関数

プログラム

lambda $((\lambda x.x)\ (\lambda y.2-y)\ 3)$

de brujn $((\lambda(L0))(\lambda(2-L0))3)$

コンパイル結果

[Pushmark; ConstInt(3); Push; Cur[Access(0); Push; ConstInt(2); Prim(" − "); Return];
Push; Cur[Access(0); Return]; Apply]

実行トレース 最初の Return が、リターンスタックからクロージャをポップすることなく、アキュムレータ上のクロージャを展開していることに注意。

Code	Accu	Env	Arg.stack	Return stack
Pushmark	$Int(0)$	[]	[]	[]
ConstInt(3)	$Int(0)$	[]	$[\varepsilon]$	[]
Push	$Int(3)$	[]	$[\varepsilon]$	[]
Cur	$Int(3)$	[]	$[Int(3), \varepsilon]$	[]
Push	$Cls(\texttt{Access(0)}, [...])$	[]	$[Int(3), \varepsilon]$	[]
Cur	$Cls(\texttt{Access(0)}, [...])$	[]	$[Cls(\texttt{Access(0)}, [...]), Int(3), \varepsilon]$	[]
Apply	$Cls(\texttt{Access(0)}, [...])$	[]	$[Cls(\texttt{Access(0)}, [...]), Int(3), \varepsilon]$	[]
Access(0)	$Cls(\texttt{Access(0)}, [...])$	$[Cls(\texttt{Access(0)}, [...])]$	$[Int(3), \varepsilon]$	$[Cls(\$, [...])]$
Return	$Cls(\texttt{Access(0)}, [...])$	$[Cls(\texttt{Access(0)}, [...])]$	$[Int(3), \varepsilon]$	$[Cls(\$, [...])]$
Access(0)	$Cls(\texttt{Access(0)}, [...])$	$[Int(3)]$	$[\varepsilon]$	$[Cls(\$, [...])]$
Push	$Int(3)$	$[Int(3)]$	$[\varepsilon]$	$[Cls(\$, [...])]$
ConstInt(2)	$Int(3)$	$[Int(3)]$	$[Int(3), \varepsilon]$	$[Cls(\$, [...])]$
Prim(−)	$Int(2)$	$[Int(3)]$	$[Int(3), \varepsilon]$	$[Cls(\$, [...])]$
Return	$Int(-1)$	$[Int(3)]$	$[\varepsilon]$	$[Cls(\$, [...])]$
$	$Int(-1)$	[]	[]	[]

B.3.4 末尾呼び出しの最適化

プログラム

lambda $((\lambda x.(\lambda y.(\lambda z.2-z)\ y)\ x)\ 3)$

de brujn $(\lambda(\lambda(\lambda(2-L0)L0)L0)3)$

コンパイル結果

[Pushmark; ConstInt(3); Push;
Cur
[Access(0); Push;

Cur

[Access(0); Push; Cur[Access(0); Push; ConstInt(2); Prim(" − "); Return]; Appterm;

Return]; Appterm; Return]; Apply]

実行トレース　Return が一回だけしか実行されていないことに注意。

Code	Accu	Env	Arg.stack	Return stack
Pushmark	$Int(0)$	[]	[]	[]
ConstInt(3)	$Int(0)$	[]	$[\varepsilon]$	[]
Push	$Int(3)$	[]	$[\varepsilon]$	[]
Cur	$Int(3)$	[]	$[Int(3), \varepsilon]$	[]
Apply	$Cls(\texttt{Access(0)}, [...])$	[]	$[Int(3), \varepsilon]$	[]
Access(0)	$Cls(\texttt{Access(0)}, [...])$	$[Int(3)]$	$[\varepsilon]$	$[Cls(\$, [...])]$
Push	$Int(3)$	$[Int(3)]$	$[\varepsilon]$	$[Cls(\$, [...])]$
Cur	$Int(3)$	$[Int(3)]$	$[Int(3), \varepsilon]$	$[Cls(\$, [...])]$
Appterm	$Cls(\texttt{Access(0)}, [...])$	$[Int(3)]$	$[Int(3), \varepsilon]$	$[Cls(\$, [...])]$
Access(0)	$Cls(\texttt{Access(0)}, [...])$	$[Int(3), Int(3)]$	$[\varepsilon]$	$[Cls(\$, [...])]$
Push	$Int(3)$	$[Int(3), Int(3)]$	$[\varepsilon]$	$[Cls(\$, [...])]$
Cur	$Int(3)$	$[Int(3), Int(3)]$	$[Int(3), \varepsilon]$	$[Cls(\$, [...])]$
Appterm	$Cls(\texttt{Access(0)}, [...])$	$[Int(3), Int(3)]$	$[Int(3), \varepsilon]$	$[Cls(\$, [...])]$
Access(0)	$Cls(\texttt{Access(0)}, [...])$	$[Int(3), Int(3), Int(3)]$	$[\varepsilon]$	$[Cls(\$, [...])]$
Push	$Int(3)$	$[Int(3), Int(3), Int(3)]$	$[\varepsilon]$	$[Cls(\$, [...])]$
ConstInt(2)	$Int(3)$	$[Int(3), Int(3), Int(3)]$	$[Int(3), \varepsilon]$	$[Cls(\$, [...])]$
Prim(−)	$Int(2)$	$[Int(3), Int(3), Int(3)]$	$[Int(3), \varepsilon]$	$[Cls(\$, [...])]$
Return	$Int(-1)$	$[Int(3), Int(3), Int(3)]$	$[\varepsilon]$	$[Cls(\$, [...])]$
$	$Int(-1)$	[]	[]	[]

B.3.5　let 式

プログラム

lambda　　let val $f = \lambda x.x - 3$ in f 2 end

de brujn　　let $L0 = \lambda(L0\ -\ 3)$ in $L0$ 2 end

コンパイル結果

[Cur[ConstInt(3), Push, Access(0), Prim(" − "), Return], Let, Pushmark, ConstInt(2), Push,

Access(0), Apply, Endlet]

実行トレース

Code	Accu	Env	Arg.stack	Return stack
Cur	$Int(0)$	[]	[]	[]
Let	$Cls(\texttt{ConstInt(3)}, [...])$	[]	[]	[]
Pushmark	$Cls(\texttt{ConstInt(3)}, [...])$	$[Cls(\texttt{ConstInt(3)}, [...])]$	[]	[]
ConstInt(2)	$Cls(\texttt{ConstInt(3)}, [...])$	$[Cls(\texttt{ConstInt(3)}, [...])]$	$[\varepsilon]$	[]
Push	$Int(2)$	$[Cls(\texttt{ConstInt(3)}, [...])]$	$[\varepsilon]$	[]
Access(0)	$Int(2)$	$[Cls(\texttt{ConstInt(3)}, [...])]$	$[Int(2), \varepsilon]$	[]
Apply	$Cls(\texttt{ConstInt(3)}, [...])$	$[Cls(\texttt{ConstInt(3)}, [...])]$	$[Int(2), \varepsilon]$	[]
ConstInt(3)	$Cls(\texttt{ConstInt(3)}, [...])$	$[Int(2)]$	$[\varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
Push	$Int(3)$	$[Int(2)]$	$[\varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
Access(0)	$Int(3)$	$[Int(2)]$	$[Int(3), \varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
Prim(−)	$Int(2)$	$[Int(2)]$	$[Int(3), \varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
Return	$Int(-1)$	$[Int(2)]$	$[\varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
Endlet	$Int(-1)$	$[Cls(\texttt{ConstInt(3)}, [...])]$	[]	[]
$	$Int(-1)$	[]	[]	[]

B.3.6 再帰的定義

プログラム

lambda `letrec val` $f = \lambda x.f(x+1)$ `in` f `end`

de brujn `letrec` $L0 = \lambda(L1\ (L0\ +\ 1))$ `in` $L0$ `end`

コンパイル結果

[Dummy; Cur[ConstInt(1); Push; Access(0); Prim(" + "); Push; Access(1); Appterm; Return]; Update; Access(0); Endlet]

実行トレース

Code	Accu	Env	Arg.stack	Return stack
Dummy	$Int(0)$	[]	[]	[]
Cur	$Int(0)$	$[Int(-1)]$	[]	[]
Update	$Cls(\texttt{ConstInt(1)}, [...])$	$[Int(-1)]$	[]	[]
Access(0)	$Cls(\texttt{ConstInt(1)}, [...])$	$[Cls(\texttt{ConstInt(1)}, [...])]$	[]	[]
Endlet	$Cls(\texttt{ConstInt(1)}, [...])$	$[Cls(\texttt{ConstInt(1)}, [...])]$	[]	[]
$	$Cls(\texttt{ConstInt(1)}, [...])$	[]	[]	[]

B.3.7 再帰関数の適用

プログラム

lambda	$\texttt{let val}\ f = \lambda x.f(x+1)\ \texttt{in}\ f\ 0\ \texttt{end}$
de brujn	$\texttt{letrec}\ L0 = \lambda(L1\ (L0\ +\ 1))\ \texttt{in}\ (L0\ 0)\ \texttt{end}$

コンパイル結果

[Dummy; Cur[ConstInt(1); Push; Access(0); Prim(" + "); Push; Access(1); Appterm; Return];
Update; Pushmark; ConstInt(0); Push; Access(0); Apply; Endlet]

実行トレース 無限ループに陥る。

Code	Accu	Env	Arg.stack	Return stack
`Dummy`	$Int(0)$	[]	[]	[]
`Cur`	$Int(0)$	$[Int(-1)]$	[]	[]
`Update`	$Cls(\texttt{ConstInt(1)}, [...])$	$[Int(-1)]$	[]	[]
`Pushmark`	$Cls(\texttt{ConstInt(1)}, [...])$	$[Cls(\texttt{ConstInt(1)}, [...])]$	[]	[]
`ConstInt(0)`	$Cls(\texttt{ConstInt(1)}, [...])$	$[Cls(\texttt{ConstInt(1)}, [...])]$	$[\varepsilon]$	[]
`Push`	$Int(0)$	$[Cls(\texttt{ConstInt(1)}, [...])]$	$[\varepsilon]$	[]
`Access(0)`	$Int(0)$	$[Cls(\texttt{ConstInt(1)}, [...])]$	$[Int(0), \varepsilon]$	[]
`Apply`	$Cls(\texttt{ConstInt(1)}, [...])$	$[Cls(\texttt{ConstInt(1)}, [...])]$	$[Int(0), \varepsilon]$	[]
`ConstInt(1)`	$Cls(\texttt{ConstInt(1)}, [...])$	$[Int(0), Cls(\texttt{ConstInt(1)}, [...])]$	$[\varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Push`	$Int(1)$	$[Int(0), Cls(\texttt{ConstInt(1)}, [...])]$	$[\varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Access(0)`	$Int(1)$	$[Int(0), Cls(\texttt{ConstInt(1)}, [...])]$	$[Int(1), \varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Prim(+)`	$Int(0)$	$[Int(0), Cls(\texttt{ConstInt(1)}, [...])]$	$[Int(1), \varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Push`	$Int(1)$	$[Int(0), Cls(\texttt{ConstInt(1)}, [...])]$	$[\varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Access(1)`	$Int(1)$	$[Int(0), Cls(\texttt{ConstInt(1)}, [...])]$	$[Int(1), \varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Appterm`	$Cls(\texttt{ConstInt(1)}, [...])$	$[Int(0), Cls(\texttt{ConstInt(1)}, [...])]$	$[Int(1), \varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`ConstInt(1)`	$Cls(\texttt{ConstInt(1)}, [...])$	$[Int(1), Cls(\texttt{ConstInt(1)}, [...])]$	$[\varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Push`	$Int(1)$	$[Int(1), Cls(\texttt{ConstInt(1)}, [...])]$	$[\varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Access(0)`	$Int(1)$	$[Int(1), Cls(\texttt{ConstInt(1)}, [...])]$	$[Int(1), \varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Prim(+)`	$Int(1)$	$[Int(1), Cls(\texttt{ConstInt(1)}, [...])]$	$[Int(1), \varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Push`	$Int(2)$	$[Int(1), Cls(\texttt{ConstInt(1)}, [...])]$	$[\varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Access(1)`	$Int(2)$	$[Int(1), Cls(\texttt{ConstInt(1)}, [...])]$	$[Int(2), \varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Appterm`	$Cls(\texttt{ConstInt(1)}, [...])$	$[Int(1), Cls(\texttt{ConstInt(1)}, [...])]$	$[Int(2), \varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`ConstInt(1)`	$Cls(\texttt{ConstInt(1)}, [...])$	$[Int(2), Cls(\texttt{ConstInt(1)}, [...])]$	$[\varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Push`	$Int(1)$	$[Int(2), Cls(\texttt{ConstInt(1)}, [...])]$	$[\varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Access(0)`	$Int(1)$	$[Int(2), Cls(\texttt{ConstInt(1)}, [...])]$	$[Int(1), \varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Prim(+)`	$Int(2)$	$[Int(2), Cls(\texttt{ConstInt(1)}, [...])]$	$[Int(1), \varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Push`	$Int(3)$	$[Int(2), Cls(\texttt{ConstInt(1)}, [...])]$	$[\varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Access(1)`	$Int(3)$	$[Int(2), Cls(\texttt{ConstInt(1)}, [...])]$	$[Int(3), \varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Appterm`	$Cls(\texttt{ConstInt(1)}, [...])$	$[Int(2), Cls(\texttt{ConstInt(1)}, [...])]$	$[Int(3), \varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`ConstInt(1)`	$Cls(\texttt{ConstInt(1)}, [...])$	$[Int(3), Cls(\texttt{ConstInt(1)}, [...])]$	$[\varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Push`	$Int(1)$	$[Int(3), Cls(\texttt{ConstInt(1)}, [...])]$	$[\varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Access(0)`	$Int(1)$	$[Int(3), Cls(\texttt{ConstInt(1)}, [...])]$	$[Int(1), \varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Prim(+)`	$Int(3)$	$[Int(3), Cls(\texttt{ConstInt(1)}, [...])]$	$[Int(1), \varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
$\vdots$				
`Appterm`	$Cls(\texttt{ConstInt(1)}, [...])$	$[Int(43), Cls(\texttt{ConstInt(1)}, [...])]$	$[Int(44), \varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`ConstInt(1)`	$Cls(\texttt{ConstInt(1)}, [...])$	$[Int(44), Cls(\texttt{ConstInt(1)}, [...])]$	$[\varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Push`	$Int(1)$	$[Int(44), Cls(\texttt{ConstInt(1)}, [...])]$	$[\varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
`Access(0)`	$Int(1)$	$[Int(44), Cls(\texttt{ConstInt(1)}, [...])]$	$[Int(1), \varepsilon]$	$[Cls(\texttt{Endlet}, [...])]$
Interrupt				

付 録C　ユーザーズガイド

C.1　Amethystの入手

Amethyst は以下の URL から入手できる。

http : //www.jaist.ac.jp/~kiyoshiy/amethyst/

C.2　インストール

まず、パッケージファイルを展開する。

```
tar zxf amethyst-x.y.z.tar.gz
```

そして、多くの GNU ソフトウェアと同様に、以下の手順でコンパイルおよびインストールをおこなう。

```
cd amethyst-x.y.z
./configure
make
make install
```

現時点では、以下のプラットホームでインストールおよび実行が可能であることを確認している。

- Solaris 2.7(Sparc)
- Windows 上の Cygwin 環境
- Linux Debian2.1(PC)
- Linux RedHat7.2(PC)

また、PostgreSQL ドメインおよび Java ドメインを利用する場合、`configure` を実行する際に PostgreSQL ライブラリおよび JDK がインストールされている場所を指定する必要がある。JAIST の環境向けの設定を記述した `configure-jaist` をパッケージに同梱している。JAIST の標準的な環境ならば `configure` の替りにこの `configure-jaist` を実行すればよい。その他の場合は各自の環境に合わせて `configure-jaist` を書き換えて実行すればよい。

C.3　コマンド

C.3.1　基本コマンド

Amethyst は、compiler と bytecode interpreter の二つの実行ファイルから構成される。上記の手順でインストールした場合、compiler のコマンド名は `amethyst`、bytecode interpreter のコマンド名は `amethystsvr` となる。

コンパイラ

amethyst file:$file$ {[-p $prelude_1$ ... $preluede_m$] | $file_1$ ... $file_n$}
amethyst pipe:$server$ {[-p $prelude_1$ ... $prelude_m$] | $file_1$ ... $file_n$}
amethyst inet:$host$: $port$ {[-p $prelude_1$ ... $prelude_m$] | $file_1$ ... $file_n$}

入力　$file_1$ ... $file_n$ を指定しない場合、amethyst はソースプログラムを標準入力から読み込んでコンパイルする。$prelude$ が指定されている場合には、$prelude_1, \ldots, prelude_m$ に記述されたソースプログラムを読み込んでコンパイルした後に標準入力からの読み込を開始する。

$file_1$ ... $file_n$ を指定すると、amethyst は $file_1$ から順に読み込み、ファイル中に記述されたソースプログラムをコンパイルする。

出力　file :$file$ を指定すると、コンパイルして得られたバイトコード列を $file$ が指すファイルに出力する。

pipe :$server$ を指定すると、amethyst は、まず $server$ で指定されたコマンドを実行して bytecode interpreter(通常は amethystsvr) を起動する。そして、コンパイルして得られたバイトコード列をパイプを通して bytecode interpreter に送信する。$server$ はフルパス名で指定することが必要。

inet : $host$: $port$ を指定すると、amethyst は、まずインターネットソケットによってホスト $host$ のポート番号 $port$ に接続する。そして、コンパイルして得られたバイトコード列をソケットを通じて送信する。この場合、bytecode interpreter をあらかじめ起動しておくことが必要である。

バイトコードインタプリタ

amethystsvr --file $file$
amethystsvr --pipe fd
amethystsvr --inet $port$

入力　--file が指定された場合、バイトコード列を $file$ で指定するファイルから読み込んで実行する。

--pipe が指定された場合、バイトコード列を fd で指定するファイル記述子から読み込んで実行する。

--inet が指定された場合、バイトコード列を $port$ で指定するポートから読み込んで実行する。

C.3.2　ラッパーコマンド

上記のコンパイラとランタイムをラップして使いやすくしたコマンドを用意している。

インタラクティブフロントエンド

amethysti {[−p $prelude_1$ ... $prelude_m$] | $file_1$ ... $file_n$}

amethysti はインタラクティブフロントエンドを提供する。引数の仕様はコンパイラと同じである。

実行例

```
bash-2.04$ amethysti
>val id = fn x => x;
```

```
val id = fn : forall ('a) => 'a -> 'a
>(id 1, id true);
val it = (1,true) : (int * bool)
>
```

バッチコンパイラ

$$\texttt{amethystc}\ [-\texttt{o}\ outputfile]\ \{[-\texttt{p}\ prelude_1\ \ldots\ prelude_m]\mid file_1\ \ldots\ file_n\}$$

amethystc は、コンパイラが生成するバイトコードとそれをランタイムに渡して起動するコードとをラップするスクリプトを生成する。-o オプションは、出力するスクリプトファイル名をしてする (デフォルトは a.sh)。その他の引数の仕様はコンパイラと同じである。

実行例

```
bash-2.04$ cat foo.ams
:set silent;
val id = fn x => x;
putInt(1 + id 2);
bash-2.04$ amethystc -o foo.sh foo.ams
bash-2.04$ ./foo.sh
3
```

関連図書

[BK99] Nick Benton and Andrew Kennedy. Interlanguage working without tears: Blending SML with java. In *International Conference on Functional Programming*, pages 126–137, 1999.

[Blu01] Matthias Blume. No-longer-foreign: Teaching an ml compiler to speak c "natively". In Nick Benton and Andrew Kennedy, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science Publishers, 2001.

[FLMJ99a] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon L. Peyton Jones. Calling hell from heaven and heaven from hell. In *International Conference on Functional Programming*, pages 114–125, 1999.

[FLMJ99b] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. *H/Direct*: A binary foreign language interface for haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 153–162. ACM, June 1999.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

[Ler90] Xavier Leroy. The ZINC experiment, an economical implementation of the ML language. Technical Report 117, INRIA, 1990.

[MF01] Erik Meijer and Sigbjorn Finne. Lambada, haskell as a better java. In Graham Hutton, editor, *Electronic Notes in Theoretical Computer Science*, volume 41. Elsevier Science Publishers, 2001.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[Oho95] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, November 1995.

[Oka98] C. Okasaki. Views for standard ml. In Proc. 1998 ACM SIGPLAN Workshop on ML, 1998., 1998.

[Ous98] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[OY99] Atsushi Ohori and Nobuaki Yoshida. Type inference with rank 1 polymorphism for type-directed compilation of ML. *ACM SIGPLAN Notices*, 34(9):160–171, September 1999. Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99).

[Ric00] Jeffery Richter. Automatic Memory Management in the Microsoft.NET Framework. *msdn magazine*, 2000.

[Wad87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In Steve Munchnik, editor, *Proceedings, 14th Symposium on Principles of Programming Languages*, pages 307–312. Association for Computing Machinery, 1987.