外部データの解釈を文脈ごとに与える動的型付け機構

大堀 淳　上野 雄大　高城光平

近年急速に集積されつつある種々のデータは、それらを操作するプログラミング言語とは独立した意味と表現を持つ。これら外部データを静的型付き言語で取り扱うには，意図する処理に応じてデータを解釈し、言語の型システムでの表現に変換する必要がある。そこで必要とされる解釈と変換は、データを処理するアプリケーションに固有であり，さらに、データの処理文脈に依存する。本研究の目的は，外部データを静的型システムで表現可能な値に変換す動的型付け機構に、プログラマが意図する解釈に応じた変換処理を統合し、型付き言語における実用性の高い外部データ操作体系を確立することである。その第一歩として、外部データの対象を JSON 等の構造化データに限定し、我々がすでに実現している SML#の動的型検査構文に、型検査の対象となる外部データの変換処理をプログラムとして自由に指定できる言語機構を導入し拡張し、その実行性を確認した。本発表ではその概要を報告する。

1　はじめに

およそ実用的なプログラムは、外部世界とのデータの入出力を必要とする。従来のシステムでは、外部データは、それぞれのプログラミングシステムを念頭に用意され、プログラムから読み込まれ、利用されてきた。このモデルでは、外部データもプログラミングのデータ同様、プログラムが定める解釈に従うデータであり、プログラムが作り出すデータと同様の意味と性質を持つ。そこでの技術的な課題は、多量のデータを外部記憶装置に格納する方式や表現形態とその内部形式への変換などであるが、これらは何れも、データの入出力ライブラリ等の閉じた課題であり、プログラムの処理対象としてのデータは、あくまで従来プログラミング言語が扱ってきたデータ型であった。

しかしながら、近年のデータサイエンスの隆盛により、種々のデータが、個々のプログラムとは独立に集積され、それら集積されたデータの一部を個々のプログラムがアクセスし、そのデータの意図に従って解釈し処理することが必要となってきている。このためには、外部データを、その意図する意味に従って、プログラムが処理可能な内部データへ変換する必要がある。この問題は、ビッグデータを扱うシステムなどで顕在化し始めているが、その系統的な手法は確立されていないと言える。現在の手法は、外部データの処理の都合に合わせ、外部データを一旦、処理プログラムにとって都合のよい形式に変換し、変換された外部データを、従来の手法でプログラムが読み込み、処理するというものである。

この外部データそのものを、事前にプログラムに合わせて変換する “データのクレンジング” アプローチは、従来の入出力処理を行うプログラムが、特的の目的で処理をするためには十分と思われるが、動的に変化する外部データレポジトリをアクセスし、外部データの種々の側面をリアルタイムに分析するようなシステムの構築には十分に対応できないと思われる。外部データの柔軟なアクセスには、外部データの意味とその処理目的に従い、それぞれの処理コンポーネントが、処理の文脈の応じて、外部データそのものを解釈し処理することが可能なプログラミング機能が必要と思われる。本研究の一般的な目的は、この洞察を下に、外部データの解釈を文脈ごとに与える機構を構築することである。この目的に向けて、外部データとして JSON のようなプログラミング言語に親和性の

Atsushi Ohori, Katsuhiro Ueno, Kohei Takagi , 東北大学, Tohoku University.

ある構造データを対象とし、これら外部データを目的に応じて柔軟にプログラムで処理する機構を試作した。本発表では、上述の目的を達成する上での技術的課題および試作の概要を報告する。

2 基礎とするアプローチとその課題

外部データを JSON のような構造化データに限っても、プログラムで柔軟に処理できる機構は、従来のプログラミング言語では十分にサポートされているとは言いがたい状況にある。現在のアプローチは、構文解析ツールを用いて JSON データを構文解析し、JSON データそのものを表すデータ構造に変換し、そのデータ構造を処理するアプローチである。このアプローチでは、

```
val r = "{\"age\" : 25,
          \"name\" : \"Bowmore\",
          \"region\" : \"Islay\"}"
```

のようなデータは、例えば

```
 OBJ [("age", INT 25),
      ("name", STRING "Bowmore"),
      ("region", STRING "Islay")]
```

のような表現としてプログラムに与えられ、プログラムが、"age"や"name"などの定性的な属性情報を含め、データとして処理する必要がある。

属性情報等を静的な情報として取り込む問題は、SML# の動的型付け機構 [1] によってほぼ解決されている。例えば以下のようなプログラムが可能である。

```
# _dynamic Dynamic.fromJson r
  as {age:int, name:string, region:string};
val it = {age = 25,
          name = "Bowmore",
          region = "Islay"}
 : {age: int, name: string, region: string}
```

このアプローチでは、以下のような機構で、外部データをプログラムで処理する。

- コンパイラは、言語がサポートする種々の型の値を、そのメタ情報も含むめて表現するデータ構造、reifiedTerm 表現を提供する。
- さらに、型付きデータと reifiedTerm 表現との相互変換を行う機構をプログラマに提供する。
- reifiedTerm 表現を、JSON などの外部データも表現可能に拡張し、外部データを reifiedTerm 表現に変換する多相関数を提供する。

この機構によって、JSON などの構造データを動的にプログラミングに取り入れることが可能になったが、以下の未解決な問題が残っている。

- 外部データは、独自の意味を持ったデータであり、プログラミング言語が定義する種々の制約された型を表現するデータではない。従って、例えば数データは抽象的な数を表し、32 ビット整数や 64 ビット浮動小数点データではない。これら基本データに限っても、用途に応じて、プログラミング言語の型に変換する必要がある。
- 外部データには、いわゆる NULL 値が含まれている。データベース分野の長年の研究が明らかにしている通り、NULL 値の柔軟な扱いは、外部データの扱いの鍵である。
- 最近のデータクレンジング研究から明らかなように、それぞれの外部データは、その意味に応じた独自のフォーマットを採用している場合が多く、複数の外部データの扱いには、これらフォーマットの差異を吸収する必要がある。

本研究の目的は、これら課題を解決し、型付き言語での実用性の高い外部データ操作体系を確立することである。この目的を達成するための第一歩として、前述の SML#の動的型付け機構を拡張し、これら問題を解決する機構の構築を試みる。

3 技術的課題とその解決戦略

これまでの分析から、本研究の具体的な課題は、外部データを処理の文脈に応じて解釈し変換する柔軟な機構を、プログラミング言語の機能として提供することである。

もちろん、任意のデータ変換をプログラムすることによって、個々のデータ変換は、当然実現可能である。データクレンジングアプローチは、外部データをプログラムで処理する前に、その個々の必要な変換をユーザが認識し、別なプログラムやツールなどによって個別に変換を実行している。また、Python などの動的言語では、外部データを含むデータ構造を、動的

な構造として保持し、外部データを含むすべてのデータを動的に処理するアプローチをとる。この場合は、プログラムがデータの構造の解釈を含む処理を行うため、データ変換の問題は、限定的である。

我々が直面する課題は、静的に型付けられた言語において、この変換を外部データの処理文脈と一体化し、例えば同一の外部データに対して、処理文脈によってそれぞれ必要な解釈変換を行う機構の構築である。前節で述べた通り、静的に型付けられた言語では、例えば、`int64` や `real32` などの基本型や {`name:string, age:int`} などの構造データは、それぞれ独自の表現を持つ。プログラムが処理できる値は、これら型で表現された値のみである。この構造から、プログラムの処理は、それらデータが内部の型に変換された後の値に限定される。われわれが必要とする機能は、必要な時点で、それぞれの処理に必要な変換をプログラムする機構である。

この問題の系統的な解決を目指す上での我々の洞察とそれに基づく実現戦略は以下の通りである。

- プログラミング言語は、当然、種々のデータを任意に変更する関数を定義する機構を提供している。それら関数は、言語の型システム内部の変換として定義される。
- プログラミング言語の型システムは十分に強力であり、外部データの構造を含む必要な構造も、もちろん型システムで表現可能である。
- 外部データの値の集合 $\mathcal{E}$ とプログラミング言語の型システムが定義する内部データの値の集合 $\mathcal{D}$ の間には、ガロア結合が構築可能である。

$$\phi \quad : \quad \mathcal{E} \longrightarrow \mathcal{D}$$
$$\psi \quad : \quad \mathcal{D} \longrightarrow \mathcal{E}$$

  この ϕ は、`_dynamic` e `as` τ などの変換と異なり、本来 ML の型として対応する値を持たない `NULL` 値の表現なども含む ML の値への embedding であり、その定義は、コンパイラによって与えることができる。
- 外部データを利用するプログラマは、必要な変換を、プログラミング言語の内部データの変換関数

$$f : \mathcal{D} \longrightarrow \mathcal{D}$$

  として記述し、コンパイラに、外部データの取り込みの前に、この関数を外部データに適用する事を要求する。

  さらに、この f の記述は、文脈に応じた `NULL` 値の変換や、フォーマットの変換など、必要な変換パターンの集合として記述できることが望ましい。

  ガロア結合から、この関数 f から、以下の外部データ変換関数を得ることができる。

$$\overline{f} = \lambda x : \mathcal{E}.\psi(f(\phi(x))) : \mathcal{E} \longrightarrow \mathcal{E}$$

- コンパイラは、具体的な外部データ E に以下の変換を施し、プロフラムに提供する。

$$\texttt{\_daynamic}\ \overline{f}(E)\ \texttt{as}\ \tau$$

- 関数 f から $\overline{f}$ を生成する処理は、コンパイラが実現する必要があるが、必要な処理の構造は、SML#の関数を C 言語のデータに適用するための処理と同様に実現可能なはずである。そこで、SML#コンパイラを拡張し、上記の機能を取り込んだ新たな以下の構文を提供する。

$$\begin{array}{l}\texttt{\_daynamic}\ exp\ \texttt{as}\ \tau \\ \qquad \texttt{with}\ pat_1\ \texttt{=>}\ exp_1 \\ \qquad\quad \vdots \\ \qquad \texttt{|}\ pat_n\ \texttt{=>}\ exp_n\end{array}$$

  `with` 以下に書かれる規則は、種々の外部データの部分の変換規則を、SML#の世界で記述したものである。

  コンパイラは、この規則を、適当な戦略で、可能な限り外部データのマッチする部分に適用し、外部データを変換した後、型 τ の値に変換する。

4 実装の概要

上述の洞察に基づき新しい構文を実装するためには、外部データの集合 $\mathcal{E}$ を具体的に定め、データ変換関数 ϕ および ψ を実現した上で、関数 f を `with` 以下に書かれた変形規則の集合から生成する必要がある。実装では、集合 $\mathcal{E}$ として、2 節で述べた `reifiedTerm` 表現を用いた。この選択により、ϕ および ψ として、SML#コンパイラが動的型付け機構を実現するために提供しているプリミティブをほぼそのまま流用する

ことができる。SML#コンパイラは、reifiedTerm と ML の値を相互に変換する以下のプリミティブを提供する。

`reifiedTermToML` : $\forall\alpha\#reify.$`reifiedTerm` $\to \alpha$
`toReifiedTerm` : $\forall\alpha\#reify.\alpha \to$ `reifiedTerm`

ここで $\#reify$ は、型インスタンスを動的なデータ構造として取り出す（reify する）必要があることを表すカインドである。NULL 値の取り扱いを除けば、`reifiedTermToML` が ϕ に、`toReifiedTerm` が ψ にそれぞれ対応する。

`reifiedTermToML` を NULL 値を含む embedding とするためには、外部データの NULL 値に対応する ML の値をひとつ定める必要がある。前節で述べた通り、NULL 値は本来 ML で対応する値を持たない特殊な値である。しかし、データを変換する規則を表現する目的にその意味を限定するならば、NULL 値は他のどの値とも異なる単独のシンボルとみなすことができる。この考え方に従い、実装では、唯一の値構成子 `null` を持つ新しいデータ型 `null` を導入し、値構成子 `null` が外部データの NULL 値に対応する ML の値となるように、`reifiedTermToML` および `toReifiedTerm` を拡張した。

残る f は、引数 D を受け取り、`with` 以下に記述された変換規則を、可能な限り D のマッチする部分に適用した結果を返す関数である。この振る舞いを静的なコード生成によって実現することは容易ではない。一方、D の `reifiedTerm` 表現を別の `reifiedTerm` 表現に変形するプログラムならば、ユーザーレベルのプログラミングでも実現可能である。そこで、f を含む関数 $\overline{f}$ を以下の方針で実装した。

- コンパイラは、`with` 以下に与えられた変形規則 pat_i `=>` exp_i それぞれについて、前節で述べたガロア接続の考え方に基づき、関数 $\overline{f_i} : \mathcal{E} \to \mathcal{E}$ を生成する。$\overline{f_i}$ は、与えられた引数 E 全体が pat_i にマッチするとき、exp_i の評価結果の `reifiedTerm` 表現を返す。そうでなければ `Match` 例外を発生させる。$\overline{f_i}$ は具体的には以下のコードである。

  `fn` x `=> toReifiedTerm`
  `(case reifiedTermToML` x `of`
  pat_i `=>` exp_i`)`

  $\overline{f_i}$ が E に対して期待通り動作するためには、$\phi(E)$ の型が pat_i の型と等しくなければならない。この制約を $\overline{f_i}$ を呼び出す前にチェックできるように、pat_i の構造を動的なデータ構造として取り出した（reify した）値 $\overline{pat_i}$ を $\overline{f_i}$ とペアにする。
- $\overline{f}$ を、$\overline{pat_i}$ および $\overline{f_i}$ の集合（リスト）と外部データ E の `reifiedTerm` 表現を受け取り、E の $\overline{pat_i}$ にマッチする部分にできる限り $\overline{f_i}$ を適用した結果を返すユーザーレベルの関数 `rewriteTerm` として実装する。コンパイラは `rewriteTerm` 関数を呼び出すコードを生成するのみとし、変換規則の適用戦略やマッチする部分の探索は `rewriteTerm` の実装に委ねる。

コンパイル例を図 1 に示す。

`rewriteTerm` は以下の戦略で外部データを書き換える。まず、いずれかの $\overline{pat_i}$ にマッチする最大の部分項を探す。これは、外部データの外側から順に、各 i について $\overline{pat_i}$ にマッチするかどうかを検査することで行う。次に、マッチした部分項を $\overline{f_i}$ を適用した結果した結果で置き換える。ただし、$\overline{f_i}$ が `Match` 例外を返した場合はマッチしなかったものとして扱う。一度マッチし置き換えられた部分項は、それ以上マッチの対象としない。以上の処理を、いずれかの $\overline{pat_i}$ にマッチする部分項がなくなるまで、外部データの構造に対して再帰的に繰り返す。

新しい構文の実行例を、SML#の対話セッションとして図 2 に示す。JSON データ `j` はオブジェクトのリストである。`j` の最初の要素は `with` の最初の変形規則に、最後の要素の `dept` の値は最後の変形規則にそれぞれマッチし、規則によって `null` は`""`に、`"Sendai"`は`"SDJ"`にそれぞれ書き換えられる。最後に、書き換え結果を SML#のレコードのリストとして読み込む。

5 まとめ

プログラムとは独立に集積された外部データをプログラムで処理するには、外部データを、その意図する意味に従って、プログラムが処理可能な内部データ

ソースコード:

```
open Dynamic
fn dyn => _dynamic dyn as {id:int, dept:string} list
          with {id:int, dept=null} => {id=id, dept=""} | "Sendai" => "SDJ"
```

コンパイル結果:

```
fn dyn =>
  _dynamic rewriteTerm
             [(PATRECORD [("id", PATWILD (_reifyTy(int))), ("dept", PATNULL)],
              fn x => toReifiedTerm (case reifiedTermToML x : {id:int, dept:null} of
                                       {id:int, dept=null} => {id=id, dept=""}),
              (PATWILD (_reifyTy(string)),
               fn x => toReifiedTerm (case reifiedTermToML x : string of
                                      "Sendai" => "SDJ"))]
             dyn
  as {id:int, dept:string} list
```

図 1 _datatype exp as τ with $\cdots$ **式のコンパイル例**

```
# open Dynamic;
# fun f dyn = _dynamic dyn as {id:int, dept:string} list
>             with {id:int, dept=null} => {id=id, dept=""} | "Sendai" => "SDJ";
(interactive):2.12-3.75(144) Warning: match nonexhaustive
      "Sendai" => ...
# val f = fn : ['a. 'a dyn -> {dept: string, id: int} list
# val j = fromJson "[{\"id\":0, \"dept\":null},\
>                  \ {\"id\":1, \"dept\":\"KIX\"},\
>                  \ {\"id\":2, \"dept\":\"Sendai\"}]";
val j = _ : void dyn
# f j;
val it = [{dept = "", id = 0}, {dept = "KIX", id = 1}, {dept = "SDJ", id = 2}]
         : {dept: string, id: int} list
```

図 2 _datatype exp as τ with $\cdots$ **式の実行例**

へ変換する必要がある。本報告では、静的型付き言語 SML#において、外部データと SML#の値とのガロア結合を考え、外部データの一部を SML#の世界で書き換える規則を与えることで、外部データの処理文脈に応じてそれぞれ必要な解釈変換を行う言語機構を提案し、その試作について報告した。今後は、この試作を通じて、本研究が提案するアプローチの実用上の課題や展望を明らかにする予定である。

謝辞 本研究の一部は JSPS 科研費 18K11233 および 19K11893 の助成を受けたものです。

参 考 文 献

[1] 大堀 淳, 上野 雄大. SML#の動的型付け機構. In **日本ソフトウェア科学会第** *36* **回大会論文集**, 2019.