SML#の動的型付け機構

大堀 淳　上野 雄大

静的型システムを持つ言語における動的型付け（dynamic typing）は、Abadi 等によってその型理論が提案されているが、そこで提案されている操作的意味論理論には、コンパイルと実行のフェーズを超えた機能が暗黙に使われており、ネイティブコードコンパイラへ直接適用するのは困難である。特に、型推論に基づく多相型言語において、この問題は顕著である。我々は、我々が開発を進める ML 系高階関数型言語 SML#の基盤技術のひとつである型主導コンパイル方式を基礎に、動的型付け機構の系統的な実現方法を開発し、SML#に実装した。本発表では、方式と SML#への実装の概要を報告する。

1　はじめに

動的型付け（dynamic typing）とは、実行時に値の型を分析し、型に応じた処理を行う機構である。この動的型付けは、静的型システムを持たない言語では、基本的なプログラム機能のひとつであるが、静的に型付けられた言語でも、外部データの操作などの際に必要となる機能である。Abadi 等 [1] によって型システムの機能として提案され以来、静的型システムを持つ言語における有用性が広く認識され、多相型言語における実装も報告されている [4] [2]。しかしながら、現時点では、ML 系多相型言語の機能として確立されているとは言えず、実用的なコンパイラにおいても十分にサポートされているとは言えない。この現状は、単に実装の努力がなされていない、とのことではなく、型付き多相型プログラミング言語コンパイラにおける動的型付け機能の実現には、従来の研究成果を超える新たな理論や技術課題の克服を必要とすることを反映している。

動的型付けの基本は、実行時にプログラムによって作られる値に対して、その値の型情報をプログラムから利用可能にすることである。Lisp などの動的に型付けられた言語（型が静的に決定されないデータを扱う言語）では、この機能が言語の基本機能として組み込まれており、例えば Lisp では `stringp` などの型述語で調べることができる。この機能から理解される通り、動的に型付けられた言語における実行時の値は、型の表現と型に応じた値とを持った一様なデータとみることができる。Abadi 等の文献では、この型情報を伴う値の一様な表現を、`dynamic` 型と呼んでいる。動的に型付けされた言語では、このような型情報を含む値の表現とその分析機能は、プログラミングの基本である。Abadi 等の提案は、この動的に型付けられた言語の値の表現である `dynamic` 型を、静的型システムに導入し、動的型付け機能をプログラムに与えることを目指したシステムである。これによって、データ永続性 [3] などの外部データの型安全な操作が可能となると期待される。

静的型システムの枠組みでの動的型付けの基本は、以下の形の型規則の導入である。

$$\text{(dyn intro)}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{dynamic}(e : \tau) : \texttt{dynamic}}$$

$$\text{(dyn elim)}\ \frac{\Gamma \vdash e : \texttt{dynamic}}{\Gamma \vdash e\ \texttt{as}\ \tau : \tau}$$

式 $\texttt{dynamic}(e : \tau)$ の効果の直感的な理解は、e の値を

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

型 τ の表現と組にし、動的言語の値の表現と同等の型情報をもった値の生成である。このような機能は、プログラムの実行が型チェック機能を含むインタープリタで行われるような言語システムでは、比較的容易に実現できる。しかしながら、静的多相型言語のコンパイラでの実現方式が確立しているとは言えない。困難さの一旦を理解するために、Standard ML の構文で書かれた以下の仮想的な多相関数宣言を考えてみよう。

```
fun ('a) toDyn x = dynamic (x:'a)
  : 'a -> dynamic
fun ('a) toValue x = (x as 'a)
  : dynamic -> 'a
```

ここで、`('a)` は、この関数で束縛されるべき多相型変数の宣言である。これら関数の意味を考えるなら、コンパイルされたコードは、任意の型 τ に対して、τ の表現を生成する能力および任意の型 τ の値の実行時表現をヒープ上に生成する能力を持たねばならないが、従来のコンパイル技術に基づく多相型言語コンパイラには、そのようなコードを生成する能力はない。

我々は、この問題に取り組み、種々の新たな概念と技術を開発し、型付き多相型プログラミング言語における動的型付けの実現方式を確立し、ほぼ完全な動的型付け機構を SML#コンパイラ [7] に実装することに成功した。図 1 に実際の対話型セッションの簡単な例を示す。型変数`'a` の`#reify` は型の reify 機能を使うとの注釈であり、型変数の取りうる型には制約はない。`Dynamic` は動的型付けを実現するシステムモジュール、`Dynamic.dyn` は動的に型付けられた値の型、`Dynamic.dyn` の型パラメタは、部分的に検証された静的な型情報を表す。静的型情報が不明の`Dynamic.void Dynamic.dyn` が従来の `dynamic` 型に相当する型である。この例から理解される通り、上記の 2 つの関数が多相関数として定義され、ユーザ定義の再帰的なデータ型を含む任意の型の動的型付けが実現されている。

本発表では、ML 系の多相型言語での動的型付け実現上の技術的課題とその解決戦略の概要を述べ、それら課題を達成して実現した SML#の動的型付けの実装を紹介する。技術的課題とその実現方式及び実装技術の詳細、さらにそれらと従来の研究との詳細の比較は別の機会に発表する予定である。

2 静的言語での動的型付け実現上の課題

前節で触れた通り、プログラムの実行時に、コンパイラが内部で管理する型情報、型推論情報、シンボルの束縛情報などを自由に利用できるインタープリタ形式の言語処理系では、動的な言語同様、動的な型付け機能を実現する上での本質的な困難は伴わないと考える。静的言語での動的型付け実現上の課題は、コンパイラとは独立に実行するコードを生成しなければならないコンパイラにとっての課題である。

静的に型付けられた言語のコンパイラが動的な型付けを実現する上での主な技術的課題には、以下のものが含まれる。

- 動的型付け機構の多相的な利用
  多相型言語において動的型付け機能を十分に利用するためには、前節で例にあげた `toDyn` 関数のような多相的な関数の中での使用が重要である。多相型言語のネイティブコードコンパイラは、多相関数も、もちろんネイティブコードにコンパイルする。従って、動的型付け機能を使用する多相関数をコンパイルするには、任意の型を取りうる多相的な動的型付け機構をコードで実現する必要がある。このことは、コンパイルされたコードが、将来の型適用時にインスタンス化される型の実行時型表現を生成・分析する必要があることを意味する。
- 静的型情報の reify
  静的な型情報は、コンパイラがコンパイル時に構築し利用するメタな情報である。これらをプログラムから利用するためには、それらメタ情報のオブジェクト化、すなわち、自己反映計算における reify 処理が必要である。
- 動的型（dynamic 型）からの値の構築
  dynamic 型をプログラムが e `as` τ などの構文を通じて利用するためには、その実際の型に応じた内部表現を持つ値を構築する必要がある。このデータの型に応じた実行時表現の構築は、コンパイラが行う処理である。この処理は、種々の最適

```
$ smlsharp
SML# 3.4.1-... (2019-07-27 10:34:04 JST ...) for x86_64-pc-linux-gnu with LLVM 6.0.0
# val toDyn = Dynamic.dynamic;
val toDyn = fn : ['a#reify. 'a -> Dynamic.void Dynamic.dyn]
# fun ('a#reify) toValue x = _dynamic x as 'a;
val toValue = fn : ['a,'b#reify. 'a Dynamic.dyn -> 'b]
# datatype foo = A of int | B of real * foo;
# val x = Dynamic.dynamic (B (1.1, B (2.2, A 3)));
val x = _ : Dynamic.void Dynamic.dyn
# val y = toValue x : foo;
val y = B (1.1, B (2.2, A 3)) : foo
```

図 1　SML#の動的型付けの例

化を含む複雑な処理である。dynamic 型からの値の構築を実現するためには、コンパイルされたコードが、データレイアウトの決定や最適化を含むコンパイラの処理を利用可能である必要がある。

- ユーザ定義のデータ型の扱い
  ML の `datatype` 宣言で定義される一般に相互再帰的なユーザ定義のデータ型は、データ構造とみれば、大きさに上限の無い循環的な構造である。もちろん、これらデータ構造は、コンパイラの型チェック・型推論モジュールが表現し操作しているデータ構造であり、表現や操作方法自体に本質的で未解明の課題はない。しかし、型付き言語コンパイラの型チェック・型推論モジュールは、コンパイラコードの中でも有数の大きさを占めるコードである。型の reify や型からの値の再構築を行うコードは、これら処理と同等の処理を実行時に行うことを要求するものであり、その系統的な実装は技術的な課題と言える。

3　課題解決戦略と SML#での実装概要

我々は、前節で議論したものを含む、多相型言語コンパイラが動的型付け機構を実現する型理論的な枠組みとその実装技術の構築に成功し、それらを実装した SML#コンパイラをすでに実現している。型理論的な枠組みとその実装技術の詳細は、他の論文に譲り、本発表では、課題実現戦略と実装の概要を報告する。

3.1　多相的な動的型付け機能の実現

多相型言語における動的型付けの最大の課題は、環境で束縛された型変数を含む型を持つ変数に対する動的型の導入を許す体系の構築である。この機能なしには、動的型付け機能を用いるプログラムは多相型を持つことができず、多相型言語の利点を十分に活用できないことになる。

困難を理解するための典型的な例は、冒頭で挙げた多相関数

```
fun ('a) toDyn (x:'a) = dynamic (x)
```

である。多相型ラムダ計算の標準的な文法では、

$$\Lambda t.\lambda x : t.\texttt{dynamic}(x : t)$$

と表現される。この関数をコンパイルした結果のコードは、束縛型変数 t に対して将来与えられる可能なインスタンス型 τ すべてに対応する動的値 (v, τ) を作る能力が要求される。注意深い読者はすでに理解されていることと思うが、この問題の本質は、引数の型に応じて振る舞いが異なる多相関数をネイティブコードにコンパイルする問題と共通であり、その系統的な一つの解決方法は、レコード多相性のコンパイルの過程で提案された型主導コンパイル方式である。

型主導コンパイル方式 [5] [6] では、型抽象時に、抽

象の対象となる式に含まれる型依存の処理を表現するカインド（型の型）を導入し、型変数を束縛する際にカインド注釈を付加し、そのカインドによって決まる実行時に必要となる属性を受け取るラムダ抽象を導入する。さらに、そのように変換された型抽象式が型適用される時、コンパイラは適用されるインスタンス型から、必要となる属性情報を表現する式を生成し、適用する。この考え方を適用し、上記のラムダ式を、以下のような形の式にコンパイルすることによって、この問題は解決される。

$\Lambda t\#\texttt{reify}.\lambda T : \texttt{reifiedTy}(t).\lambda x : t.\texttt{dynamic}(x, T))$

ここで、$\texttt{reifiedTy}(t)$ は型 t の型表現を唯一の値としてもつ単元型（singleton type）、`dynamic(x, T)` は、インスタンス型 τ を持つ値 `x` と τ の型表現 `T` を受け取って処理を行う関数である。

SML#コンパイラには、モジュラーな型主導コンパイルフェーズが組み込まれている。この型主導コンパイルフェーズに、上記の機能を実現するモジュールを組み込むことで、この方式を実装することができる。

3.2 型の reify と動的型の値の構築

問題点であげた、型の reify、動的型からの値の構築、ユーザ定義のデータ型の扱いの 3 つ課題の本質は、いずれも、再帰的な型の表現、型の実行時表現、最適化処理、などのコンパイラの複雑な機能を、コンパイルされたユーザコードから呼び出すための系統的機構の開発といえる。そのような機構があれば、適宜コンパイル機能の呼び出しを行うようにコンパイルすることによって解決できる。このコンパイラ機能の呼び出し方式の開発は、コンパイラとユーザコードの関連に依存し、本質的に処理系依存の問題である。

SML#コンパイラは、それ自身 SML#で書かれており、コンパイラの種々の機能は、SML#のモジュールとして実現されている。さらに、それらモジュールは、分割コンパイルされ、リンクされてコンパイラが作られている。この構造は、ユーザコードの構造と同じである。従って、ユーザコードからのコンパイラコードの呼び出しが、コンパイラが使用するモジュール単位であるなら、リンクや実行時の呼び出しに関する本質的な問題は存在しない。残る課題は、コンパイラが知っているライブラリ関数を、ユーザコードから呼び出すコードの生成である。これは、コンパイラを構成する変数の静的環境を、ユーザコードのコンパイル環境に追加し、さらに、ユーザレベルでのコンパイラの関数呼び出しを行う式を生成するコンパイルレベルの式を導入することによって解決できる。

3.3 SML#の動的型付け拡張

我々は、これら型理論的な枠組みの構築およびそれらに基づくコンパイラへの実装を行い、動的型付け機構を装備した SML#コンパイラの開発に成功した。図 2 に、SML#の動的型付けを実現するシステムモジュールである `Dynamic` ストラクチャの一部と、動的型付け機能のための特殊構文の概要を示す。

実現された動的型付けは、永続性の実現、型依存の演算、汎用の値のプリンター、JSON 等の外部データの取り込みや書き出し等多くの実世的な機能の基盤となる技術である。`Dynamic` ストラクチャに定義された、`['a#reify.'a -> unit]` 型を持つ Dynamoc.pp 関数は、汎用のプリティプリンタである。特殊構文ではなく、通常の高階関数であるため、この関数を呼び出すだけで、任意の型の値をプリントすることができる。例えば `foldl` 関数の引数をトレースしたければ、以下のように `foldl` の冒頭で `Dynamic.pp` 関数を引数に対して呼び出すだけで良い。

```
# fun foldl f z l =
    (Dynamic.pp {l=l,z=z};
     case l of
       nil => z
     | h::t => foldl f (f (h,z)) t);
val foldl = fn
 : ['a#reify, 'b#reify.
    ('a * 'b -> 'b) -> 'b -> 'a list -> 'b]
```

ここで {`l=l,z=z`} は、Standard ML のレコード式である。以下の例が示すように、この関数は、適用される値の型に応じた適切なプリントを行う。

```
# foldl (fn (h,z) => h + z) 0 [1,2];
{l = [1, 2], z = 0}
{l = [2], z = 1}
{l = [], z = 3}
```

動的型付けを実現する Dynamic ストラクチャの一部

```
structure Dynamic =
  struct
    type 'a dyn  <hidden>
    type dynamic = void dyn
    datatype ty =
    datatype term =
    exception RuntimeTypeError = PartialDynamic.RuntimeTypeError
    val dynamic = fn : ['a#reify. 'a -> void dyn]
    val dynamicToString = fn : void dyn -> string
    val dynamicToTerm = fn : void dyn -> term
    val dynamicToTy = fn : void dyn -> ty
    val format = fn : ['a#reify. 'a -> string]
    val fromJson = fn : string -> void dyn
    val join = fn : void dyn * void dyn -> void dyn
    val pp = fn : ['a#reify. 'a -> unit]
    val toJson = fn : ['a. 'a dyn -> string]
    val valueToJson = fn : ['a#reify. 'a -> string]
    val view = fn : ['a#reify. 'a dyn -> 'a]
     ...
  end
```

動的型付けを操作する特殊構文

- _dynamic *exp* as τ　　動的型の値への変換
- _dynamiccase *exp* of *ruleList*　　型の混在を許すパターンマッチング

図 2　SML#の動的型付け拡張

```
val it = 3 : int
# foldl (fn (h,z) => h ^ z) "" ["a","b"];
{l = ["a", "b"], z = ""}
{l = ["b"], z = "a"}
{l = [], z = "ba"}
val it = "ba" : string
```

現在の SML#の対話型環境を実現する値のプリンタはこの動的型付け機能を用いて実現されている。

プリティプリンタに限らず、従来困難であった種々の機能が、この動的型付け機構を用いて実現可能である。たとえば、SML#の C 言語との直接連携機能と組み合わせれば、C 言語で書かれた JSON パーザ Yajil を直接呼び出し、結果を動的型付け機構を用いて、ML の値として取り込むことができる。Dynamic.fromJSON 関数はそのようにして実現されている。

今後リリース予定のコンパイラ SML# 3.5 版では、この動的型付け機構が提供される予定である。

4　まとめ

本発表では、静的に型付けされた多相型言語で動的型付けを実現する上での技術的課題を論じ、その解決戦略と、その戦略に基づき実現した SML#の動的型付け機能の概要を紹介した。SML#では、一般化された自然結合や対話型システムのプリンター始め多くの機能が、動的型付け機構を用いて実現している。今後、動的型付けの機能の拡張やその応用技術の研究

を進め、より柔軟で実用的な SML#処理系の開発を進めていく予定である。

参 考 文 献

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. *ACM Transcations on Programming Languages and Systems*, 13(2):237–268, 1991.

[2] M. Abadi, L. Cardelli, B. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. In *ACM SIGPLAN Workshop on ML and its Applications*, 1992.

[3] M.P. Atkinson and O.P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, June 1987.

[4] X. Leroy and M. Mauny. Dynamics in ml. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, 1991.

[5] A Ohori. A compilation method for ML-style polymorphic record calculi. In *Proc. ACM POPL Symposium*, pages 154–165, 1992.

[6] A. Ohori. A polymorphic record calculus and its compilation. *ACM Trans. Prog. Lang. and Syst.*, 17(6):844–895, 1995.

[7] SML#. http://www.riec.tohoku.ac.jp/smlsharp/, 2006–2019.