SML#の永続性拡張に向けて

大堀 淳　上野 雄大　大塚祐貴　高城光平

プログラミング言語への永続性の付与は、1980 年台に盛んに研究されたテーマであり、PS-algol、Galileo、PJava などの言語が開発されたが、汎用言語における機能として確立するに至っていない。特に、汎用の高階型付プログラミング言語においては、系統的な永続性のサポートはほとんどなされていない。その主な要因は、コンパイル時の静的な型と、外部永続データとの適切な関連の維持に関する方法が確立されていない点にあると思われる。本発表では、我々が開発を進める ML 系高階関数型言語 SML#をベースに、この問題を探求し、対話型環境に永続機能を付加した永続 SML#の実験的な試作を報告する。

1　はじめに

プログラミング言語の機能としての永続性（persistence）の概念は 1980 年代に盛んに研究され、直交永続性（orthogonal persistence）の概念が提案され、永続ルートからの到達可能性（reachability）による直交永続性の実現方式などが提案され、それらを取り入れた永続プログラミング言語（persistent programming language）が開発された（研究動向については、例えばサーベイ [3] を参照）。代表的永続プログラミング言語には、PS-Algol [4]、Napier [5], Gallileo [1]、P-Java [2] などがある。これら研究にもかかわらず、直交永続性は汎用のプログラミングの機能として確立し定着しているとは言いがたい。

プログラミング言語の文脈での永続性とは、元々は、データをファイルに書き込む機能およびデータをファイルから読み込む機能のことであり、通常の言語では、文字列またはビット列といった限られた型のデータに対してファイル入出力プリミティブとして提供される。プログラムが必要とする多様なデータの扱いは、それら文字列やビット列を、プログラムで解析することによって実現される。プログラミング言語からのデータベースの問い合わせ言語の呼び出しも、この枠組みでなされている。

直交永続性の概念は、このファイル上のデータへのアクセスを、文字列やビット列といった基底データから、プログラミング言語が提供する種々の型、さらにユーザプログラムで定義可能な任意の型に一般化することを目指す研究上のマニフェストと言うべき概念である。このような性格上、直交永続性に関する厳密な技術的定義や実現基準が広く共有されているとは言えない。しかしながら、「任意の型のデータが永続する」との直交永続性の考え方は、直感的には、おおよそ以下のように理解されている。

- 任意の型のデータを、プログラムでエンコード（マーシャリング）やデコード（アンマーシャリング）をすることなしに、読み込み、書き出しができる。
- さらに、データのファイルへの読み込みや書き出しは、自動的に、あるいは宣言的に行われる。

ここから理解される通り、直交永続性は、豊富な型を扱うプログラミング言語の機能として提唱された課題である。特に、高階の関数型や多相型などを含む豊富な静的な型を持ちユーザーが静的な型を柔軟に定義できるような多相型言語の機能を格段に強化する可能性をもつ。

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

しかし上述の通り、現状では、Haskell や OCaml、Standard ML などの近代的な型付き多相型プログラミング言語では、残念ながら直交永続性は、ほとんどサポートされていない。問題点を明確にするために、以下の簡単な問題を考えてみよう。これら近代的な型付き多相型プログラミング言語では、対話型環境を提供している。対話型環境のトップレベルで束縛された変数は、それに続くプログラムで使用される名前であるので、これら名前を永続データの名前として扱えると便利であろう。そこで、前述した永続言語の研究における永続ルートからの到達可能性を、対話型環境で参照可能な変数集合と捉えると、直交永続性の機能は、以下のようなプログラミングを即座に可能にするはずである。

1. 電車の中で対話型環境を立ち上げ、高階多相関数 `setX` を定義した後、対話型セッションを一旦終了する。
2. 帰宅後、再度対話型環境を立ち上げ、前のセッションで定義した `setX` にデータを部分適用し、特殊化された多相関数 `incX` を構築し、対話型セッションを終了する。
3. 後日再度対話型環境を立ち上げ、`incX` にオブジェクト `p` を適用し、結果を得る。

対話型プログラミングは、もともと漸進的に複雑なプログラムを作っていくのに適したプログラミングモデルであるため、上記のようなサポートは、データ永続性の基本と言える。しかし残念ながら、データベースなどが絡まないこのようなごく簡単な永続性でさえ、十分にサポートされているコンパイラは現時点では存在しないと思われる。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における直交永続性の実現には、従来の永続性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。

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での直交永続性を実現する見通しを得た。さらに、それら成果を SML#言語コンパイラ [8] に適用した SML#の永続性拡張の試作に成功した。以下は、実際の対話型セッションである。

```
$ smlsharp
SML# 3.4.1-8812.hgf05a (2019-08-03 ...
# val p = {X = 1, Y = 2};
val p = {X = 1, Y = 2} : {X: int, Y: int}
# fun setX x p = p # {X = x};
val setX = fn
  : ['a. 'a -> ['b#{X: 'a}. 'b -> 'b]]
# $
$ smlsharp
SML# 3.4.1-8812.hgf05a (2019-08-03 ...
# fun incX p = setX (#X p + 1) p;
val incX = fn : ['a#{X: int}. 'a -> 'a]
# $
$ smlsharp
SML# 3.4.1-8812.hgf05a (2019-08-03 ...
# p;
val it = {X = 1, Y = 2} : {X: int, Y: int}
# incX p;
val it = {X = 2, Y = 2} : {X: int, Y: int}
# $
```

ここで、`$`はシェルプロンプト、`#`は SML#のプロンプトである。この例から、レコード `p` のみならず、高階のレコード多相関数 `setX` やクロージャ`incX` などが、永続性データとして実現されていることが分かる。そればかりか、これら永続データは SML#の永続性ヒープである `.persist.json` ファイルに以下のような JSON 形式で保存されているため、整数のレコード `p` のような言語の実装に依存しないデータは、SML#に限らず JSON を理解できるプログラムならどの言語からもアクセスできる。

```
{"value":
   {"p": {"x" : 1, "y" : 2},
     ...
   },
 "type":
   {"p": {"recordTy":{"x":"int","y":"int"}},
     ...
   }
}
```

本発表では、ML 系の多相型言語の直交永続性拡張

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つつある SML#永続性拡張の実装を紹介する。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 多相型言語の永続性拡張の技術的課題

多相型言語において永続性を実現する上で達成すべき主な課題には以下のものが含まれる。

1. 動的型付け機構の実現
2. 静的環境の動的な構築
3. 暗黙の型情報の扱い
4. `datatype` 等、言語特有の機能の表現
5. 多相型関数の値と型の表現
6. 参照型等の変更可能なデータや循環構造の表現

多相型言語においては、これらは理論的分析さえ十分になされておらず、実装技術も未開拓な課題である。ML 系多相型言語で理想的な永続性を達成するためには、これら課題のすべてを解決し、コンパイラに実装する必要がある。ここでは、それらのいくつかについて、技術的な問題点と解決方針の概要を述べる。

2.1 動的型付け機構の実現

一般に、静的に型付けられた言語で永続性を実現するためには、言語がその内部で定義する型と値の関係を明示的に表現する必要がある。例えば、ML で `real` 型の変数 `x` は、実行時に浮動小数点データに束縛される。しかし、この型情報はコンパイラがコンパイル時に持つものであり、実行時にはその情報は存在しない。従って、単に `x` のデータのビット列をファイルに書き出しても、そのビットパターンが何を意味するかわからなければ、後の読み込み時に `x` を元の `real` 型の値として復元することはできない。この問題は、古くから静的言語における動的型付け（dynamic typing）として研究されてきた問題であり、永続性実現の基本技術の一つである。しかしながら、多相型を含む言語の動的型付け機構の実現技術は十分に確立しておらず、型理論的な枠組みと実装方式の双方の構築が必要である。例えば、直交永続性のためには、直感的には、以下のような関数で表現される動的型付けの機能が必要と思われる。

- `toDynamic : 'a -> dynamic`
- `toValue : dynamic -> 'a`

ここで、`dynamic` 型は、ファイルに書き出し可能な型情報を伴う値の表現を持つ型である．`toDynamic(`e`)` は任意の型の式 e をファイルに書き出し可能な `dynamic` 型へ変換する関数、`toValue(`d`)` はファイル等から読み込んだ `dynamic` 型の表現 d を受け取り、静的な型を持つ（ヒープ上の）値の表現を構築する関数である。

2.2 静的環境の動的な構築

静的型システムの枠組みを理解する者には、上記の動的型付け機能の実現が困難であることが理解されると思われる。特に、`val x = toValue(`d`)` にどのような型を与え、さらに、その型情報をどのように静的に反映させるか、不明である。動的型付けがプログラムの機能であれば、この問題に対しては、種々のアドホックな解決が可能である。例えば、`toValue(`d`)` の型の明示的な宣言を要求し、型の整合性をプログラマに委ねることができる。しかし、これは、ファイルの読み込み時に文字列やビット列データを、期待される型に変換することをプログラマが宣言することに等しく、直交永続性を実現していることにならないだけでなく、膨大な数の変数の型を正確に知り指定することも現実的には不可能である。直交永続性の実現には、外部データが表現する型の情報を基に、プログラムの静的な環境を構築する機構が必要である。

2.3 暗黙の型情報の扱い

動的型付けの実現戦略のひとつは、コンパイラが持つ（メタレベルの）型情報を、プログラムが扱い得る値として表現する、自己反映計算分野で研究されてきた型の reify 機能の実現である。多相型言語の場合、この実現方式自体十分に確立しているとは言えないが、仮に完全な型の reify 機能が実現されたとしても、直交永続性の実現には、いくつかの困難な課題が残されている。その一つが暗黙な型情報の扱いである。コンパイラは、生成する値のすべての型情報を明示的に扱うとは限らない。例えば、関数型 `int -> int` の値は関数クロージャであるが、その中には、クロー

ジャ環境として他のクロージャも含む無制限の値の束縛が保存されている可能性があり、さらに、それらの型情報はコンパイル時さえ明示的に表現されていない。直交永続性の実現には、これらを含む値の明示的な表現とその再構築の方式と実装技術が要求される。

2.4 言語依存の機能

大部分の ML プログラムは、`datatype` 定義機構を用いて独自のデータを定義し、利用している。この状況は ML に限らない。例えば、Java では、種々のクラスを定義し、インスタンスを生成している。直交永続性の実現には、これら言語依存の定義機構を明示的に表現し、さらに、その表現からプログラムが構築した環境を再構築する機能が必要である。これら言語特有の機能には、値としては表現困難なものがいくつか含まれている。例えば、ほぼすべての近代的なプログラミング言語は、静的なスコープ規則でプログラムが構造化されている。静的スコープ規則に従いプログラムが定義した種々の型を持つ値について、その型を正確に表現するためには、スコープによって隠された型の表現等が必要となるが、それらの系統的な扱いは従来の研究では示されていない。

3 SML#永続性拡張の概要

我々は、前節で議論したものを含む、直交永続性を実現する型理論的な枠組みとその実装技術の構築にほぼ成功し、現在、永続 SML#の開発を進めている。冒頭で紹介した通り、発表時点で、実際に動作する対話型環境を実現している。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、実証戦略の概要と実装状況を報告する。

3.1 SML#直交永続性拡張の実現戦略と開発状況

SML#は、レコード多相性、C との直接連携を可能にする高い相互運用性、SQL のシームレスな統合などを実現した ML 系関数型言語である。これらの機能は、それら自身有用なばかりでなく、直交永続性を実現する基盤ともなっている。これら基盤の上に直交永続性機能を構築する戦略の概要は以下の通りである。

1. 動的型付け機構

   SML#のレコード多相性や C との直接連携は、コンパイラのメタレベルの機能としてではなく、コンパイルされたコードの機能として実現している。その中心となる技術が型主導コンパイル方式[6][7] である。この型主導コンパイル方式を用いて、`int` や `bool` などの基底型の表現をベースに、系統的な動的型付けを実現している。例えば、SML#では、

   ```
   # Dynamic.dynamic;
   val it = fn
      : ['a#reify.
           'a -> Dynamic.void Dynamic.dyn]
   ```

   のような関数が実装されている。ここで、型変数 `'a#reify` に付けられた注釈`#reify` は、型主導コンパイラが型の reify を行う型変数であることを示す。そのインスタンスに制約はない。動的型付けの詳細は、別の機会に発表予定である。

2. 静的な環境の動的な構築

   型主導コンパイルによる型の reify 機能によってコンパイラがコンパイル時に生成する型は、実行時にプログラムから参照可能であるだけでなく、プログラムが自由に型情報を生成することも可能である。永続データには、この型表現が付加されている。この型表現自体は基底型（`ReifiedTy.reifiedTy`）のデータとして扱われる。静的な型環境の動的な構築のためには、この動的に生成された型表現をコンパイラの静的型環境に束縛する reflect 機能が必要である。SML#では、ユーザレベルのプログラムとコンパイラが、ヒープを通じて値を共有する機構を通じて、この reflect 機能を実現している。

3. 暗黙な型の表現

   通常の型付き言語のコンパイラでは、コンパイラが生成する値の表現は、コンパイラ独自の仮定に依存している。従って、例えば、浮動小数点や整数などの基底型さえ、標準の表現との互換性が保証されない状況であり、レコードやクロージャなどの値を安全にファイルに書き出す等の処理は、なお困難である。SML#の C との直接連携

は、この問題を解決している。これは、基底型はすべてネイティブ表現を採用し、さらに、レコードは、そのレイアウト情報をデータ自身が保持することによって実現している。これらデータの情報により、暗黙な型情報を含むデータも、安全にファイルへの書き出しと読み込みが可能となる。SML#の non-moving GC は、この技術により実現されている。GC を含むこれら機構を洗練することにより、関数クロージャなどの暗黙な型情報を含むデータも型安全に系統的に永続化が可能である。

4. 言語依存の機能への対応
   言語依存の機能、特にユーザの `datatype` 定義とそれに従った値の表現には、静的スコープ規則に従う型の定義の再構築が必要である。これら機能をコンパイラに付加することに本質的な困難はないが、その結果得られる永続データの表現は、言語のソース構文の静的な規則を反映した複雑なものとなり、外部とのデータの相互運用の観点からは、適切とは言えない。そこで、SML#では、永続データの表現に JSON による標準的な表現以外に、値を再現するソースコード表現も採用し、この機能にとって `datatype` を含むデータの直交永続性の実現する戦略を取る。

以上の戦略の基づき、型理論的な枠組みを構築し、実装技術の開発し、それらを実現する SML#コンパイラの実装を行っている。本原稿執筆時には、すでに以下の機能を実現済みである。

- 動的型付け機構のほぼ完全な実現
- 変数束縛からの JSON 表現の生成
- JSON 表現から静的型情報の構築とコンパイラ環境への追加
- レコード多相やランク 1 多相を含む高階多相型関数の永続性の実現
- ネストしたクロージャを含む暗黙な型情報を含む値の表現の生成

さらに、これらの一部をコンパイラの対話型環境に統合した永続 SML#のプロトタイプの試作に成功した。現在のプロトタイプでは、クロージャに含まれる環境はポインタを含まない物に限定され、また、ユーザデータ型のクロージャの混在は未実装である、等の制約を含むものの、実際に対話型環境を起動して永続性の機能を確認することができる。本稿の冒頭の対話型セッションは、そのプロトタイプの実際の出力結果である。

4 まとめと今後の開発方針

ML 系多相型言語に直交永続性機能を追加する上での型理論的な枠組みと実装上の未解決課題を提示し、その解決戦略の概要を提示した。提示された解決戦略は、すでに実現済みであり、それに基づく永続 SML#のプロトタイプの試作に成功している。現在、直交永続性のサポートする SML#コンパイラの開発の完成を目指している。本発表では、完成しつつある永続 SML#のデモを交えて、この開発の現状を報告した。技術的課題の解決の詳細は他の論文に譲る。

参 考 文 献

[1] Albano. A., L. Cardelli, and R. Orsini. Galileo: A strongly typed, interactive conceptual language. Technical report, Bell Laboratories, Bell Telephone Laboratories, Internal Technical document Services, Murray Hill 1b-509, NJ, USA, 1983.

[2] M. P. Atkinson, L. Dayns, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent java. *ACM SIGMOD Record*, 25(4):68–75, 1996.

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

[4] Persistent Programming Research Group. PS-algol reference manual - fourth edition. Technical Report PPRP-12-87, Universities of Glasgow and St Andrews, 1987.

[5] R. Morrison, A.L. Brown, R.C.H. Connor, and A. Dearle. Napier88 reference manual. Technical report, Department of Computational Science, University of St Andrews, 1989.

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

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

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