修士学位論文要約（平成29年３月）

より完全で利便性の高い外部関数呼び出し機構の実現

佐藤 友昭
指導教員：大堀 淳 教授

Implementation of a More Complete and Convenient Foreign Function Calling Method

Tomoaki SATO
Supervisor: Atsushi OHORI

This article refines the direct C language interface of SML#. For functional languages, seamless connection with C is an important subject. In SML#, C functions can be directly imported without data conversion. We refine this mechanism in the following three ways. Firstly, we introduce labeled argument type to increase readability. Secondly, we develop a mechanism to import data allocated by C functions by using data conversion function implemented in SML#. This mechanism allows the programmer to treat C struct data as SML# records. Thirdly, we propose a mechanism to associate a union data containing a tag field to a variant type in SML#.

1.　はじめに

関数型言語 SML#[1] には C との直接連携機能が実装されており，多くの C で書かれた関数を特別なデータ変換処理を書かずに直接インポートできる[2]．一方で，いくつかの C 関数は SML#の機能不足のため，そのままインポートすることができなかったり，インポートした関数を使う上での有用な情報が失われる場合がある．現在の SML#では，同じ型の引数を複数持つ C 関数をインポートした際に，関数型の可読性が低下する場合がある．本論文では，SML#に実装されている順序付きレコード型を C 関数の引数の型に対応付けることでこの問題を解決する．また，C 関数によって返されるポインタが指す先のデータを，対応する SML#の型の値としてインポートできない場合がある．本論文では，外部の動的なデータをサポートするために SML#に実装されたデータ変換関数を用いて，C のヒープ上のデータをインポートする方法を提案する．さらに，現在の SML#では，共用体を扱う C 関数に対して何もサポートしていない．本論文では，SML#のヴァリアントと同じ使い方をする共用体に限り，ヴァリアントに対応付ける方法を提案する．

2.　引数名を含めた C 関数のインポート

C 関数の引数は，型の並びを区別するという特徴を持つ．SML#のレコード型はフィールドの並びが順不同である．このため通常のレコード型では，C 関数の引数の並びを表現することができない．本論文では，SML#にインポートする C 関数の型に仮引数名を含めることを可能にするために，引数の型として順序付きレコード型を用いる．順序付きレコード型は，フィールドの名前と並び順の両方を区別することができるレコード型である．順序付きレコードのラベルは，自然数と名前をアンダーバー（_）でつなげたものであり，自然数の部分がフィールドの順序を示す．本研究では，以下のように書ける C 関数インポートのための構文を追加する．

$$\texttt{\_import "}f\texttt{"}\ :\ (l_1\ :\ \tau_1, \ldots,\ l_n\ :\ \tau_n)\ \texttt{->}\ \tau$$

ここで，l_i は順序付きレコードのラベルと同様な表現を持つ．本研究では，この式に対して型付けをする際に，インポートする関数の引数の型として順序付きレコード型を対応付けるように，SML#のコンパイル方式を拡張する．この方式によってインポートされた関数 `f` は以下の型を持つ．

$$\{l_1\ :\ \tau_1, \ldots,\ l_n\ :\ \tau_n\}\ \texttt{->}\ \tau$$

本機能により，C 関数をインポートする際に，ラベルとして仮引数名を含めることが可能となり，インポートした関数の可読性が向上する．

3.　C のヒープ上にあるデータのインポート

SML#のヒープに確保されたオブジェクト全体の構造は，自動的なメモリ管理機能における管理情報などを含むため，C のヒープ上に確保されたオブジェクト全体の構造とは異なる．本研究では，C のヒープ上のオブジェクトを対応する SML#の型のデータとしてインポートする機能を実装する．このような機能を実現するためには，以下に示すような多相関数を実装することが望ましい．

```
importPtr : ['a. 'a ptr -> 'a]
```

この関数を実現するためには，ポインタが指す先のデータの型 τ の種類に応じて異なる処理を行う必要がある．例えば，τ が `int` 型の場合は，ポインタが指す先のデータを返すだけでよいが，構造体型の場

合はメンバを全てレコードフィールドにコピーした上で，自動的なメモリ管理のための情報も適切に付加しなければならない．ゆえに，このような関数をSML#の言語機能だけで実装するのは困難である．本研究では，外部の動的な型を持つデータを扱うためにSML#に実装されたデータ変換関数やプリミティブを用いて，目標とする関数の一部を実装する．関数 `reifiedTermToML` は，`reifiedTerm` を受け取り，対応するSML#の型の値へと変換する関数である．また，`_reifyTy` プリミティブを利用する．このプリミティブは，SML#の型を与えると，その型に対応する型情報 `reifiedTy` とデータタイプのコンストラクタに関する情報 `conSetEnv` をフィールドとして含むレコード `tyRep` を返す．型 `reifiedTy` の定義の一部を以下に示す．

```
datatype reifiedTy
= INTty
| REALty
| DTATYPEty of
  {longsymbol : Symbol.longsymbol,
   id : TypID.id,
   args : refiedTy list,
   layout : layout,
   size : int}
| RECORDty of reifiedTy RecordLabel.Map.map
| ...
```

`reifiedTerm` は，型 `reifiedTy` で表されるデータの実行時表現であり，以下に定義の一部を示す．

```
datatype reifiedTerm
= INT of int
| REAL of real
| DATATYPE of string * reifiedTerm option
| RECORD of (string * reifiedTerm) list
| ...
```

本研究ではこれらの関数を用いて，以下に示す関数を実装する．

```
  importPtr :  ['a#reify.  'a ptr -> 'a]
```

この関数は，型変数`'a`にインポートしたいデータの型を指定すると，その型の値をCのヒープ上から取り出す関数である．この関数の処理内容は以下の通りである．まず型変数`'a`を`_reifyTy`プリミティブに与えることで，対応する `jsonTy` を取得する．次に，`jsonTy` で表される型情報を基に，Cのヒープ上のデータを取得することで，`reifiedTerm` を作る．そして最後に `reifiedTermToML` 関数に `reifiedTerm` を与えることで，対応するSML#の型の値を取得する．本機能を用いることで，Cのヒープ上にある構造体をSML#のレコードとしてインポートすることなどが可能となる．

4.　共用体型のサポート

SML#のヴァリアント型の値は，どのヴァリアントが選択されているかを表すタグをそのデータの先頭に含む．一方でCの共用体型は，その共用体の先頭にタグを含むとは限らない．つまり，共用体型とヴァリアント型は同様の機能を有する反面，メモリレベルのデータ表現には厳密な互換性がない．Cの共用体は，ヴァリアント型と同じ使い方をすることができる．このような共用体は，タグ付き共用体と呼ばれ，各メンバが先頭に4ビットの整数タグを含む構造体で構成される．本研究では，このようなタグ付き共用体に限り，SML#のデータタイプに対応付ける機能を実装する．本機能を実現するにあたり，まずはじめに，ユーザが自由にタグの値を設定できるタグ付きデータタイプを導入する．例えば，タグ付きデータタイプは以下のように宣言する．

```
datatype d = Var1(_tag : 1) of int
           | Var2(_tag : 2) of real
           | ...
```

n を自然数とした場合に，上記の宣言中の（`_tag :` n ）の部分が本研究で新たに加えた構文であり，n に32ビットの整数を指定することで，各コンストラクタに与えるタグを設定することができる．本研究では，ユーザが指定したタグの値を `int` 型の値として保持し，各ヴァリアントに割り当てるようにコンパイル方式を拡張した．次に，C関数が共用体を指すポインタを返す際に，そのポインタが指す先のヒープ上の共用体をタグ付きデータタイプとしてインポートする必要がある．この機能は，前節で述べたCのヒープ上のデータを対応するSML#の型の値としてインポートする機能を応用する．タグ付き共用体の各メンバは先頭に32ビットの整数タグを含む構造体であるため，まず先頭の4バイトをタグとして取得し，それ以外のメンバをフィールドに含むレコードとしてインポートする処理を行う．本機能を実現することで，共用体へのポインタを引数や戻り値の型として扱うC関数をインポートすることが可能となる．

5.　まとめ

本研究では，インポートするC関数の引数の並びに順序付きレコード型を対応付けることで，SML#の引数型に仮引数名を含めることを可能とした．また，C関数が返すポインタからそのポインタが指す先のヒープ上のデータを `reifiedTerm` を介して，対応するSML#の型の値としてインポートする機能を実装した．さらに，ヴァリアント型と同じ使い方をする共用体に制限し，その共用体をタグ付きデータタイプに対応づけてSML#にインポートする機能を実装した．

参考文献

1) SML# Compiler, available at http://www.pllab.riec.tohoku.ac.jp/smlsharp/

2) 上野 雄大，大堀 淳，"SML#の外部関数インターフェース"，コンピュータソフトウェア，Vol. 27，pp. 142-168，2010.