日本ソフトウェア科学会第 27 回大会 (2010 年度) 講演論文集

多相レコード計算に基づく軽量な第一級オーバーロードの設計と実装

上野 雄大　大堀 淳

本発表では，多相レコード計算のコンパイル方式を応用し軽量に実現可能な，型付き関数型言語の第一級オーバーロード方式を提案する．この方式の下では，オーバーロードされた関数は，オーバーロードされた型上のみを動く型変数で束縛された多相型を持つ第一級の関数として扱われる．本機能の実現に要求されるものは，型抽象におけるインスタンス変数の生成と，型適用におけるインスタンスの選択のみであり，これらは多相レコードコンパイルと同様の方式で達成できる．本方式は SML# コンパイラ上に実装され公開されている．本大会の予稿では，方式と実装の概要を説明する．

1　序論

Standard ML では，+や>等の演算子は，組み込みの整数型や浮動小数点型に対してオーバーロードして定義されている．例えば，`1 + 2` という式の型は `int` 型と推論され，`1.0 + 2.0` の型は `real` 型と推論される．しかし，それらオーバーロードは全て静的に解決されるため，それらの演算子を第一級の多相関数として取り扱うことはできない．演算子が多相的に使用されている場合は，コンパイラはその演算子のデフォルトのインスタンスが選択されたものとみなして強制的にオーバーロードを解決する．例えば，

```
fun f x = x + x
```

のような関数 `f` の定義では，型推論によって+演算子のインスタンスを選択するための型情報が得られない．そこでコンパイラは，+のデフォルトのインスタンスとして `int` 型上の加算演算を選択する．その結果，`f` の型は `int -> int` となる．

このような強制的なオーバーロードの解決は，プログラマの直感に反しており，好ましくない．+がオーバーロードされているならば，`f` も同様にオーバーロード関数として定義されているほうが自然であり，またコードの汎用性を高めることができる．このようなことを普遍的に実現するための一つの方法は，オーバーロード関数を，ある型の集合上のみを動く多相型を持つ第一級の関数として取り扱えるように，言語を拡張することである．第一級のオーバーロード関数は，Haskell の型クラス [2] や OCaml の拡張言語である G'Caml [1] 等ですでに実現されている．しかし，これらのオーバーロード方式は汎用性は高いものの複雑な型システムを持ち，コンパイラの実装が容易ではなく，ユーザーが使いこなすことも難しい，また，それらの拡張によって言語の互換性が失われる懸念がある．

本稿では，ML 系高階関数型言語における，より軽量な第一級のオーバーロード関数を実現する方式を提案する．本方式は以下のような特徴を持つ．

- Standard ML に対して本方式の拡張を行っても互換性を失わない．ユーザーは，本方式の拡張を意識することなく，従来通りの記述方法で，オーバーロードされた関数を第一級の多相関数として取り扱うことができる．既存のプログラムは全て改変することなくそのまま利用可能である．
- オーバーロード関数を使用している式の最も一

Design and Implementation of Lightweight First-class Overloading.

Katsuhiro Ueno, Atsushi Ohori, 東北大学電気通信研究所, Research Institute of Electrical Communication, Tohoku University.

般的な型を推論することができる．

- 多相レコード計算 [3] のコンパイル方式の自然な応用で第一級オーバーロードを実現する．多相レコード計算をすでに実装しているコンパイラならば，多相レコードコンパイルに対する若干の拡張で本方式を実装することができる．
- 1 つのオーバーロード関数につき，1 つの単相型をキーとしてインスタンスを選択することができる．キーとなる型は，引数付きの型構築子を含んでもよい．

例えば，本方式の下では，+は以下のような型を持つ多相関数として定義される[†1]．

```
+ : ['a::{int, real}. 'a * 'a -> 'a]
```

ここで，'a::{...} の ::{...} はオーバーロードを表す型カインドであり，型変数 'a が取り得る例が制限されていることを意味する．この例では，型変数'a は int 型もしくは real 型にのみインスタンス化可能であることを表している．また，オーバーロードカインドを持つ型変数をネストすることによって，基底型だけでなく，型構築子を含む任意の単相型に対してオーバーロード関数を定義することができる．例えば，型

```
['a::{int, real, 'b list, 'c array,
      'd option},
 'b::{bool, char}, 'c::{int, bool}, 'd.
 'a * 'a -> 'a]
```

はオーバーロードされた二項演算子の型であり，int，real，bool list，char list，int array，bool array および'd option 型のインスタンスを持つことを表している．

本稿では，以下の構成に従い，提案するオーバーロード方式の概要を述べる．第 2 節では，多相レコードコンパイルを応用しオーバーロード方式を実現するための方針を述べる．第 3 節ではコンパイル方式の概要を述べる．第 4 節では，本方式の SML# コンパイラにおける実装を簡単に報告する．第 5 節でまとめと今後の課題について述べる．

†1 実際の Standard ML の処理系では，+は様々な数を表す組み込み型に対してオーバーロードされている．この例では簡便のため，それらの型のうち代表的なもののみを示した．

2 実現方針

[3] で提案されている多相レコード計算のコンパイル方式では，以下の方針でレコード多相性を効率の良いコードにコンパイルする．

1. あるフィールドを持つレコード型のみにインスタンス化可能であることを表す型カインド（レコードカインド）を型変数に与える．
2. あるレコード型中のあるフィールドのインデックスを，単一の値のみを持つ型（シングルトン型）で表現する．
3. レコードカインド付きの多相型抽象から，シングルトン型のラムダ抽象を生成する．
4. 3. の結果と多相型に対する型適用から，シングルトン型の引数を生成する．

これらの各ステップは，インスタンス化可能な型を制限する機構と，型インスタンスから実行時に必要な情報を受け渡すコードを生成するためのフレームワークと見なすことができる．

本稿では，このフレームワークを応用し，第一級オーバーロードに必要な機構を実現する．まず，オーバーロードインスタンスの型を表す型カインド **ov** を導入する．このカインドは，インスタンス化可能な型の集合を保持する．次に，ひとつのオーバーロードインスタンスを表現するシングルトン型 *Inst* を定義する．最後に，オーバーロードカインドとシングルトン型のラムダ抽象との対応を与える．

3 コンパイル方式

本節では，[3] に合わせて，ソース言語と実装言語を定義し，第一級オーバーロードを実現するコンパイル方式の概要を述べる．

ソース言語の式 e，単相型 τ，多相型 σ および種類 k を以下のように定義する．

$$e ::= c^b \mid i \mid x \mid \lambda x.e \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e$$
$$\tau ::= b \mid t \mid \tau \to \tau \mid d(\tau, \ldots, \tau)$$
$$\sigma ::= \tau \mid \forall t :: k.\sigma$$
$$k ::= U \mid \mathbf{ov}(I, T)$$

ここで，x は変数，b は基底型，d は型構成子，T は単

相型の集合を表すメタ変数である．e，τ，σ および k の定義は，[3] における暗黙型付けレコード計算 $\lambda^{let,\blacksquare}$ からレコードに関する項を取り除き，オーバーロードに関する項を加えたものに等しい．$d(\tau_1,\ldots,\tau_n)$ は，引数 $\tau_1,\ldots,\tau_n$ を取る型構成子である．関数型 $\tau_1 \to \tau_2$ は，関数型構成子に 2 つの型引数を適用した型 $\to(\tau_1,\tau_2)$ の別表記と見なす．i は，式中ではオーバーロードカインド $\mathbf{ov}$ を含む多相型を持つ定数である．i の集合はシステムによって与えられ，ユーザーは新たなオーバーロード関数を定義できないものとする．オーバーロードカインド $\mathbf{ov}(I,T)$ は，型の例を T の要素に限定する種類である．I は，このカインドを持つ型変数の型インスタンスがオーバーロードインスタンスを選択するキーとなるオーバーロード関数群を表す．例えば，以下の 2 つのオーバーロード関数が与えられているとき，

$$\texttt{+} : \forall t{::}\mathbf{ov}(\{\texttt{+}\},\{\texttt{int},\texttt{real}\}).t\times t\to t$$
$$\texttt{-} : \forall t{::}\mathbf{ov}(\{\texttt{-}\},\{\texttt{int},\texttt{real}\}).t\times t\to t$$

以下のように定義された関数 `f` を考える．

```
fun f x = x + x - x
```

このとき，`f` は以下のような型を持つ．

$$\texttt{f} : \forall t{::}\mathbf{ov}(\{\texttt{+},\texttt{-}\},\{\texttt{int},\texttt{real}\}).t\to t$$

関数 `f` の定義中には 2 つのオーバーロード関数 `+` および `-` が使用されている．しかし，それら 2 つの関数は共に x の型上の演算として使用されているため，x の型が決まれば，`+`，`-` 共にひとつのインスタンスが決まらなければならない．x の型 t の種類に含まれるオーバーロード関数識別子の集合 $\{\texttt{+},\texttt{-}\}$ は，この状況を明示するためのものである．

オーバーロードカインド $\mathbf{ov}(I,T)$ の T に型変数以外を型引数に取る型構成子が含まれていた場合，最も一般的な型を推論することが難しい．例えば，以下のような型を持つオーバーロード関数 `f` と，`f` を用いて定義した関数 `g` を考える．

$$\texttt{f} : \forall t{::}\mathbf{ov}(\{\texttt{f}\},\{\texttt{int list},\texttt{bool list}\}).\ t\to t$$

```
fun g x = f [x]
```

このときの `g` の型は，`f` の型から型構築子 `list` のみが外れた，以下のような型であることが期待される．

$$\texttt{f} : \forall t{::}\mathbf{ov}(\{\texttt{f}\},\{\texttt{int},\texttt{bool}\}).\ t\to t$$

しかし，`f` の $\{\texttt{int list},\texttt{bool list}\}$ と式 `[x]` の型 `'t list` から $\{\texttt{int},\texttt{bool}\}$ を単一化アルゴリズムによって求めることは難しいため，このような型を推論することと，型推論アルゴリズムが最も一般的な型を推論する性質を両立させることは難しい．そこで，本提案では，T の要素を基底型あるいは型変数のみを型引数に取る型構築子に限定する．型構成子から成る単相型をオーバーロードする場合は，オーバーロードカインドを持つ型変数をネストすることで，その単相型の構造を表すことにする．例えば，上記の関数 `f` の型は，以下のように表現する．

$$\texttt{f} : \forall t_1{::}\mathbf{ov}(\{\texttt{f}\},\{t_2\ \texttt{list}\}),\quad t_2{::}\mathbf{ov}(\{\texttt{f}\},\{\texttt{int},\texttt{bool}\}).\ t_1\to t_1$$

このような形に限定することで，インスタンス集合の共通部分の単一化を求めることによって，`g` のような関数の最も一般的な型を推論することができる．また，このように限定したとしても，インスタンス型を表現する能力は変わらない．ただし，オーバーロードインスタンスを選択するための型情報が複数の型変数に分散していることに注意する必要がある．

ソース言語から実装言語へのコンパイルは，多相レコードコンパイルと同様に，型推論による型が明示された計算系への変換と，型抽象からラムダ抽象を生成する型主導コンパイルの 2 つのステップで行う．実装言語の型 τ を以下のように定義する．

$$\tau ::= b \mid t \mid \tau\to\tau \mid d(\tau,\ldots,\tau) \mid \mathit{Inst}(i,\overline{\tau})\Rightarrow\tau$$

$\mathit{Inst}(i,\overline{\tau})$ は，型の列 $\overline{\tau}$ で選択されるオーバーロード関数 i のインスタンスを表すシングルトン型である．$\mathit{Inst}(i,\overline{\tau})\Rightarrow\tau_2$ は，オーバーロードインスタンスを受け取り τ_2 型の値を返すラムダ抽象の型である．

オーバーロードカインド付き型抽象からのラムダ抽象の生成は以下の手順で行う．

1. オーバーロードカインドに現れるそれぞれの i について，カインド付き型変数の依存関係から，i のインスタンスを選択するためのネストした型変数群 T_i を求める．
2. 型変数間の依存関係を元に，ネストした型変数群 T_i を一列に並べ，型の列 $\overline{t_i}$ を得る．
3. $\overline{t_i}$ が i のインスタンスを選択するキーとして十分ならば，シングルトン型 $\mathit{Inst}(i,\overline{t_i})$ を生成する．

4. 生成された全てのシングルトン型について，その型の引数を受け取るラムダ抽象を決められた順に生成する．

型適用からの引数の生成は，多相レコードコンパイルと同様である．

4 実装

本稿で提案する第一級オーバーロード方式は，著者らが開発している Standard ML の拡張言語 SML# に実装されている．本稿の成果を含む SML# コンパイラは Web 上に公開されており，SML# の Web サイト [4] からソースコードを取得できる．

現在の SML# では，基本ライブラリの仕様に定められたオーバーロード関数はすべて第一級のオーバーロード関数として提供されており，ユーザーはこれらの関数を組み合わせて新たな第一級のオーバーロード関数を定義することができる．例えば，ひとつの数と数のリストの和を返す関数 `sumList` を以下のように入力すると，オーバーロードカインド付きの多相関数としてコンパイルされ，その型が表示される．

```
# fun sumList z nil = z
>   | sumList z (h::t) = h + sumList z t;
val sumList = fn
    : ['a::{int, IntInf.int, real,
            Real32.real, word,
            Word8.word}.
       'a -> 'a list -> 'a]
```

煩雑な型の表示を防ぐため，SML# コンパイラはオーバーロードカインド $\mathbf{ov}(I, T)$ のうち T のみを`::`の後に表示する．コンパイラ内部では，実際には I に相当する情報もオーバーロードカインドに含めて管理している．以下のように具体的な引数を与えて `sumList` を呼び出すことができ，`sumList` が正しくオーバーロード関数として定義されていることを確認できる．

```
# sumList 0 [1,2,3];
val it = 6 : int
# sumList 0.0 [1.0,2.0,3.0];
val it = 6 : real
```

Standard ML では演算子だけでなく定数リテラルもオーバーロードされている．これら定数リテラルもオーバーロードカインド付きの多相型を与え，多相性を最大化することも技術的には可能である．しかし SML# では，ユーザーの意図しない多相性が導入されることを防ぐため，また実行効率のために，定数リテラルについては従来の Standard ML と同様に，静的にオーバーロードを解決することにしている．従って，上記 `sumList` を初期値を 0 としてリストの和を計算する関数として以下のように記述した場合，`sumList` はオーバーロード関数にならない．

```
# fun sumList nil = 0
>   | sumList (h::t) = h + sumList t;
val sumList = fn : int list -> int
```

5 まとめ

本稿では，多相レコード計算のコンパイル方式を応用した軽量な第一級オーバーロード方式を提案した．本方式の下では，オーバーロード関数は，オーバーロードされた型上のみを動く型変数で束縛された多相型を持つ第一級の関数となる．本方式は多相レコードコンパイル方式の自然な拡張として構築されているため，Standard ML との互換性を失わず，実装も比較的容易である．また，SML# コンパイラにおける本方式の実装について簡単に報告した．

本稿で提案した第一級オーバーロード方式の型システム，コンパイルの理論，実装，および関連研究との比較の詳細は，別途報告する予定である．

参 考 文 献

[1] Furuse, J.: Extensional Polymorphism by Flow Graph Dispatching, *In Proceedings of Asian Symposium on Programming Languages and Systems (APLAS)*, Lecture Notes in Computer Science, Vol. 2895(2003).

[2] Hall, C. V., Hammond, K., Peyton Jones, S. L., and Wadler, P. L.: Type classes in Haskell, *ACM Transaction on Programming Languages and Systems*, Vol. 18, No. 2(1996), pp. 109–138.

[3] Ohori, A.: A polymorphic record calculus and its compilation, *ACM Transactions on Programming Languages and Systems*, Vol. 17, No. 6(1995), pp. 844–895.

[4] SML# Compiler, http://www.pllab.riec.tohoku.ac.jp/smlsharp/.