より“シンプルな”領域・エフェクトシステムの試み

A Possibly “Simpler” Region and Effect System

香川 考司
Koji KAGAWA

香川大学工学部信頼性情報システム工学科
(kagawa@eng.kagawa-u.ac.jp)

概 要

Tofte と Talpin は、ML のプログラムに対して、領域・エフェクト推論 (region & effect inference) を用いて、オブジェクトの寿命を推論し、通常のゴミ集めに頼らずに使用済のメモリを回収する方法を提案した。

この方法は、高度な型推論システムに基づいており、プログラマはオブジェクトの寿命に関する明示的な指定はいっさい行なくて済む。その一方で、プログラマの意図どおりにオブジェクトの寿命を推論させるために、ある種のテクニックが必要になる場合がある。

本研究では、相対的参照 (compositional reference) の概念に基づく、Tofte と Talpin のシステムと同じ目的を持つ、別の型システムを提案し、いくつかのプログラム例を挙げる。このシステムでは、プログラマが明示的にオブジェクトの寿命を指定して、その指定の正当性をシステムがチェックする。再帰関数の型推論も Tofte & Talpin とは異なる方法で行なう。

1 はじめに

Lisp や ML、Java などの関数型言語やオブジェクト指向言語の多くはゴミ集め (garbage collection) の機能を持っており、ゴミ集めを持たない場合に比べて、生産性や安全性が大幅に向上している。その反面、プログラムの実行時間に関する予測が難しくなる、あるいは性能を良くするためにはメモリが比較的大量に必要になる、などの問題点が生じる。これらは通常のコンピュータで通常のアプリケーションを実行する場合には大きな問題とならないが、例えば小型の組込み機器上で、時間的要求の厳しいアプリケーションを実行する場合には、問題となりうる。

このような場合、あらかじめ必要な領域を確保し、プログラムの実行中には、新しいオブジェクトを生成しないように工夫する必要がある。C 言語の用語で言えば静的 (`static`) な変数のみ用いるようにする。しかし、このようなメモリ管理の方式のみを用いると、寿命の限られたデータに対しても、常に領域を確保する必要があるなど柔軟性がなく、かえって大量のメモリが必要になる場合も考えられる。

そこで、C 言語の自動 (`auto`) 変数のように、関数やブロックの実行時にだけ確保される変数を用いることが考えられる。C 言語の `auto` 変数は、そのままでは unsafe であり、Lisp や ML、Java のように安全性を看板の一つに掲げる言語では、そのままでは採用することはできない。そこで Tofte と Talpin [8] は ML のプログラムに対してオブジェクトの寿命を推論し、使用されなくなった時点で、メモリを解放することができるようにするために領域・エフェクト推論 (region & effect inference) という手法を提案した。この方式ではオブジェクトをスタック (正確には領

域 — region のスタック) に割り当てることができる。その結果、少ないメモリでプログラムを実行し、プログラムの実行時間を予測することが可能になる。

一方、この方式は、通常の ML プログラムからメモリに割り当てられたオブジェクトの寿命を “推論” するのであって、プログラマが明示的にオブジェクトの寿命を指定し、それをコンパイラがチェックするのではない。そうすると、プログラマの意図と推論結果がうまく合わない場合がある。例えばリストの長さを計算する `length` 関数の場合が [7] に報告されている。

```
fun length ([], acc) = acc  (* A *)
  | length (_::xs, acc)
           = length (xs, acc+1)

fun length l = (* B *)
  let fun loop (p as ([], acc)) = p
        | loop (_::xs, acc)
               = loop (xs, acc+1)
  in snd (loop (l, 0)) end
```

`(* A *)` のように定義すると、(プログラマの意図に反して、) 再帰呼出しのたびにタプルのための新しい領域が割り当てられ、スタックが消費されてしまう。一方で同じ関数を `(* B *)` のように定義すると、今度は再帰呼出しの際には、新しい領域を割り当てないようになる。さらに storage mode analysis と呼ばれる解析を行なうと、以前の領域を上書きしても良いことがわかり、コンスタントなメモリ消費で計算を行なうことができるようになる。

このように、領域をコントロールするためには、上の `(* B *)` の `as` パターンのように、あまり自明でないテクニックが必要であり、また storage mode analysis という (おそらくこれも自明でない) 解析に頼る必要がある。

そこで考えられるのが、すべてを自動的に推論させることをあきらめ、

- 領域の指定をプログラマが、ある程度明示的に行なう。
- 場合によって、繰り返しや破壊的代入などの命令的なプログラムの書き方を併用する。

という方針である。

原理的には、Tofte と Talpin のシステムでも、プログラマが型と領域を明示し、コンパイラがそれを型チェックすることは可能であると考えられるが、領域とエフェクトに関する型の表現が、通常のプログラミング言語に比べると高度で複雑であるため、プログラマが直接、このような指定をすることは難しいと思われる。

例えば、

```
( let x = (2, 3)
  in λ y.(fst x, y) end)
5
```

というプログラムに領域に関する操作を加えると、

```
letregion ρ4, ρ5
in letregion ρ6
   in let x = (2 at ρ2, 3 at ρ6) at ρ4
      in (λ y. (fst x, y) at ρ1) at ρ5
      end
   end
   5 at ρ3
end
```

のようになる [8]。ここで **letregion**が領域の有効範囲を示す。領域に関する記述の部分が多くなり、気軽に用いることはできない。また関数の型も複雑になりがちである。

そこで、さらに次のような方法が考えられる。

- 領域の間に階層を考え、複数の領域をまとめて、ひとつの領域として表現することができるようにする。
- デフォルトの領域のようなものを考え、いちいち領域を指定しなくても良いようにする。

一方で Peyton Jones と Launchbury [6] は、Haskell で状態を用いた計算 (stateful computation) を純粋 (pure) な計算の中に言わば “封じ込める” ための基本的なオペレータ (`runST`) を提案している。Haskell は遅延評価を行なう言語であり、Tofte & Talpin の研究と異なり、メモリ管理は彼らの研究の対象ではないが、ML のように先行 (eager) 評価を採用する言語では、メモリ管理のためにも応用できると考えられる。このシステムではモナドの手法を用いて、状態 (Tofte と Talpin のいう領域) を表すデータをプログラム中で明示的に扱わなくても良いようになっている。それでいて、型には領域に関する情報が現れ、領域の有効範囲の指定 (`runST`) は明示的である。しかし、`runST` では、領域は一度に一つだけで、スタックのような領域の階層が表現できない。

本研究では、Peyton Jones と Launchbury の型システムに相対的参照 [3] の概念を適用して領

域を階層化する。領域に関する記述は、デフォルトの領域を利用する場合は、とくにしなくても良いし、プログラマが明示的に行なうことも可能である。相対的参照は、領域の異なるオブジェクトのキャストや、領域を明示的に指定するときに識別子として用いる。

さらに、領域を明示的に扱うためには、もう一つ大事な点がある。それは再帰関数の型推論である。領域を扱う場合、関数は再帰呼出しの時に別の型を割り当てられるようになっていなければ、実用的ではない。しかし、このようないわゆる多相再帰 (polymorphic recursion) に対しては、ML の型システムの範囲内では一般に pricipal type を計算するアルゴリズムは存在しないことが知られている [2] [4]。Tofte と Talpin のシステムでは、再帰関数の型を推論するときに、領域に関するパラメータを特別扱いし、polymorphic recursion を扱っている。しかし、領域に関する情報をユーザに明示的に見せる場合には、領域をそのように特別扱いするわけにはいかない。

- 領域 (やエフェクト) は、型の一種として扱い、特別扱いしない。

本研究では、再帰的な関数の型推論も Tofte& Talpin とは異なる方法で行なう。

2 型システム

本研究では、エフェクトや領域を特別扱いしないので、Tofte と Talpin の 領域・エフェクトシステムの表記法をそのまま使わずに、モナド (monad)に基づく型システムの表記法を用いる。しかし、エフェクトシステムとモナドに基づくシステムに本質的な違いはない。

われわれの言語は、モナド的な型システムを持つ以外は通常の値呼びの関数型言語である (図 1)。

m は Haskell 風の型構成子変数 (type constructor variable) で、この部分が、式のもつ副作用を表す。特に副作用を持っていない場合は、この部分をアンダースコア (_) で表すことにする。これは、(パターン中の変数に対する慣習と同様に) 常にフレッシュな変数を意味する。(Var) や (Lambda) は副作用を持たないので、この部分がアンダースコアになっている。なお、再帰的な定義 (letrec) については後で触れる。

Peyton Jones と Launchbury のオペレータは次のようなものである。

```
newVar : τ → ST s (Ref s τ)
readVar : Ref s τ → ST s τ
writeVar : Ref s τ
            → _ (τ → ST s unit)
runST : (∀s. unit → ST s τ) → _ τ
            (s ∉ FV(τ))
```

本研究は先行評価を行なう言語を対象とするため、unit を余分に引数にとるなど、若干オリジナルの型から変更されている。(ただし文法は、ほぼ Haskell 風を採用する。)

ST は領域操作関数* (state transformer) を表す型構成子である。ST s xは s という型の領域に作用を及ぼし、x という型の値を返す計算 (computation) の型である。ST sが副作用を表す部分に当たる。

runST は領域に依存する計算が外部に影響を及ぼさないように閉じ込める役割をする。このため内部で副作用を用いても、外部からは純粋な関数に見えることが保証される。runST の引数の領域操作関数の型の領域パラメータは、総称的であることが要求される。つまり、引数の中の自由変数の型 (つまり環境) のなかに s が現れてはいけない。さらに、τ の中に s が現れてはいけない。この s は領域の型を表すが、この “閉じ込め” のチェックのために使う抽象的な型である。一種の存在型 (existential type) と考えることができる。

相対的参照 (compositional reference) [3] とはあるオブジェクト (領域) の中の部分オブジェクト (領域) の相対的な位置を指し示すために用いられるデータ型である。“compositional” という名は、2 つの相対的参照が (関数を合成するように) 合成できることを表している。s という型のオブジェクトの中の t という型の部分オブジェクトを示す相対的参照の型は、Mut s tと表される。s をこの相対的参照の台、t を対象と呼ぶことにする。 例えば、3 つ組 (a, b, c) の中の最

* 状態操作関数と訳すべきかも知れないが、本研究では Launchbury & Peyton Jones の “state” は、Tofte & Talpin の “region” に対応するため、この文章中は領域という言葉を使うことにする。

$$\frac{x :: \tau \in \Gamma}{\Gamma \vdash x :: _\,\tau}\ (\mathrm{Var})$$

$$\frac{\Gamma \vdash e_1 :: m\,(\tau_1 \to m\,\tau_2) \quad \Gamma \vdash e_2 :: m\,\tau_1}{\Gamma \vdash e_1\,e_2 :: m\,\tau_2}\ (\mathrm{App})$$

$$\frac{\Gamma, x :: \tau_1 \vdash e :: m\,\tau_2}{\Gamma \vdash \lambda x \to e :: _\,(\tau_1 \to m\,\tau_2)}\ (\mathrm{Lambda})$$

$$\frac{\Gamma \vdash e_1 :: _\,\tau_1 \quad (e_1 : \mathrm{value}) \quad \Gamma, x :: \forall \overline{\alpha}.\,\tau_1 \vdash e_2 :: m_2\,\tau_2 \quad (\overline{\alpha} = FV(\tau)/FV(\Gamma))}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 :: m_2\,\tau_2}\ (\mathrm{Let})$$

図 1　型規則

初の 2 つの要素の組 (a, b) を示す相対的参照の型は Mut (a, b, c) (a, b) となる。(ただし、本研究では台が (a, b, c) のような具体的な型になる相対的参照は扱わない。) ちなみに参照型 Ref s aは大きさ 1 の配列に対する相対的参照 Mut s (Array a) と考えることができる。

相対的参照に関する基本的なオペレータは次のとおりである。

```
appR : Mut s t
       →  _ ((unit → ST t τ)
       → ST s τ)
cmpR : Mut s t
       → _ (Mut t u → _ (Mut s u))
```

appR は、t に対する領域操作関数 ST t τを、t を含む領域である s に対する領域操作関数 ST s τとして利用するために用いる。cmpR は、2 つの相対的参照を (関数合成のように) 合成する。

さらに、runST の一般化として、次の extendST というやはり特殊な型を持つ関数を導入する。

```
extendST : (∀s. Mut s t → ST s τ)
           → ST t τ
```

s と τ に関する条件は runST と同様である。直感的には s は t を “拡張した” 領域である。extendST の引数となる領域操作関数は、内部ではこの拡張された領域を対象に計算を行なうが、外部からはこの拡張された部分は見えず、t の領域操作関数に見える。この拡張された部分に生成されたオブジェクトは、extendST の実行が終わったときは不要になるため、メモリを回収することができる。

例えば、

```
extendST
(λ_ → var a = newArr n true in
      var _ = sieve a in
      readArr a n)
```

では、a が、extendST の中で生成されているので、このオブジェクトが占めていたメモリは、extendST の実行が終了後回収される。(ここで **var** $x = e_1$ **in** e_2 は $(\lambda x \to e_2)\,e_1$ の略記法として用いている。)

Mut s tという型の相対的参照は s の領域操作関数から t の領域操作関数を利用したり、t を台とする相対的参照を s を台とする相対的参照にキャストしたりするために用いられる。例えば、$s \supset t$ の時、s/t の部分 (extendST が拡張した領域) にあるリストを t (元からあった領域) にコピーする関数は次のように定義される。

```
copyD : Mut s t
        → ST s (Mut s (List s a)
        → ST s (Mut t (List t a)))
copyD t xs =
  if isNull xs then
       appR t (λ_ → nil unit)
  else var a  = head xs in
       var as = copyD t (tail xs) in
       appR t (λ_ → cons a as)
```

nil unit や cons a as は 領域 t で行なわれなければならないため、appR を用いている。

ただしリストに関する基本的なオペレータの型は図 2 の通りである。 実際に isNull, head, tail を使うときは、t は s の部分領域になっている。(つまり $t \subseteq s$。)

Mut s (List s a) は、対象が List のような具体的な型になっている相対的参照だが、これは直感的には領域 s にあるリスト型を表す。(さらに List の場合は tail の部分も相対的参照になるため、s を型パラメータとして持っている。) 本研究ではこのような相対的参照は破壊的代入を行なうかどうかに関係なく、単にデータを領域に関連づけるために用いている。

```
nil    : unit → ST s (Mut s (List s a))
cons   : a → _ (Mut s (List s a) → ST s (Mut s (List s a)))
isNull : Mut s (List t a) → ST s Bool
head   : Mut s (List t a) → ST s a
tail   : Mut s (List t a) → ST s (Mut t (List t a))
```

図 2　リスト処理関数

逆に、t (元からあった領域) にあるリストを s/t の部分(extendST が拡張した領域)にコピーするときは、次のような関数を用いる。

```
copyU : Mut s t
        → ST s (Mut t (List t a)
        → ST s (Mut s (List s a)))
copyU t xs =
  if appR t (λ_ → isNull xs) then
    nil unit
  else
    var a = appR t (λ_ → head xs)
    in
    var as = copyU t
       (appR t (λ_ → tail xs)) in
    cons a as
```

今度は head や tail を領域 t で行なわなければならないので appR を用いている。

実用的には、head や tail を呼ぶ度に appR を使うのは面倒である。サブタイピングか、それに類する型システムを導入することによって、この appR の挿入を自動化する必要があると思われる。つまり copyU の理想形は次のようになる。

```
copyU xs =
  if isNull xs then
    nil unit
  else
    var a = head xs in
    var as = copyU (tail xs) in
    cons a as
```

しかし、今のところは appR を用いる必要がある。

一方、copyD の cons の場合は appR の挿入を自動化するのはより難しいかも知れない。しかし、appR が必要なのが、このようにコンストラクタを呼ぶ場合だけにすることができれば、充分実用的になると思われる。

ちなみに、copyU は cmpR を用いて

```
copyU t xs =
  if isNull (cmpR t xs) then
    nil unit
  else
    var a  = head (cmpR t xs) in
    var as = copyU t
             (tail (cmpR t xs)) in
    consM a as
```

のようにも書くことができる。

3　再帰関数と extendST

extendST は、領域パラメータについて多相型であることを要求する。このため、領域を扱う関数の中で extendST を使用し、その中で再帰呼出しをすると、通常の letrec の型推論ルールでは型を推論できない。ML の型推論では、再帰呼出しは、もとの呼出しと同じ型でなければいけないからである。

ただし、このような多相再帰 (polymorphic recursion) でも型推論ではなく型チェックならば可能である。(図 3) しかし、再帰関数に対していちいち型を明示的に与えるのは、プログラマにとって大きな負担となりがちである。

Tofte と Talpin は、再帰関数は “領域に関するパラメーターについてのみ” 多相であれば良いとして、そのような多相再帰を推論するために、次のような方法を取った。

1. 最初は領域パラメータを無視して、e_1 の型付けを行なう。
2. 次に、この型の領域パラメータにすべて別々の型変数を割当て、これらを総称化した型スキームを x に割当て、e_1 の型チェックを行なう。この e_1 の型スキームと x の型が一致すれば e_1 の型チェックは終了する。
3. そうでなければ、この e_1 の型スキームを新しい x の型として e_1 の型チェックをやり直し、x と e_1 の型が一致するまで繰り返す。

領域パラメータは数が有限で、繰り返しにしたがって数が減少するので型チェックは必ず停止する。

この方法で、もちろん必要な多相型を発見することができるが、このアルゴリズムの停止性の証明は、この領域・エフェクトシステムの性質に強く依存している。Tofte & Talpin のシステムでは領域をユーザが明示的に指定せず、領域に関することはすべてシステムが推論する。そして letregion もシステムが挿入することになっ

$$\frac{\Gamma, x :: \forall\overline{\beta}.\tau_1 \vdash e_1 :: _\,\tau_1 \quad (e_1 : \text{value}) \quad \Gamma, x :: \forall\overline{\alpha}.\tau_1 \vdash e_2 :: m_2\,\tau_2 \quad (\overline{\beta} \subseteq \overline{\alpha} = FV(\tau_1)/FV(\Gamma))}{\Gamma \vdash \texttt{letrec}\ x = e_1\ \texttt{in}\ e_2 :: m_2\,\tau_2}\ (\text{LetRec})$$

図 3　型規則 (letrec)

ているので、このような方法をとらざるを得ないと考えられる。

一方で本研究では、相対的参照により領域を指定し、オブジェクトの寿命も extendST で明示的に指定するので、もう少し単純化した方法がとれる。つまり、次のような (少しアドホックな) 方法を取る。

1. e_1 の中の x のそれぞれの出現ごとに別々の型変数を割り当て、e_1 の型を推論する。
2. x のそれぞれの出現 (occurrence) に割り当てられる型から、総称変数にならなければならない部分を割り出す。
3. e_1 の型と x のすべての出現の型を単一化 (unify) し、その結果、必要な部分が総称変数になっているか、チェックする。

出現毎に別の型変数を割り当てるのは、以下で述べるように、runST と extendST の型推論に、存在型の型推論に使われるようなスコーレム型構成子 [5] が使われるため、これらの別々の出現の間では通常の単一化は行なわれないからである。通常、スコーレム型構成子を用いて runST と extendST の型規則を表現すると、図 4 のように表される。 しかし、今回は次のように変更を加える。

- runST, extendST の型を推論するときに、自由変数 (Γ) の中の変数のうち、letrec で束縛される変数の型についてはスコーレム定数 (κ) のエスケープのチェックをしない。エスケープのチェックは letrec の型チェックの時まで遅延する。そのために、κ に τ や自由変数の型の中の型変数を関連づけて記録しておく。これは κ とこれらの変数があとで単一化されないことを letrec の型推論時にチェックしなければならないからである。

以上をまとめると図 5 および図 6 のようになる。TC が型推論の本体である。TC では、再帰的な自由変数 ($::_r$) とそうでない自由変数 ($::_n$) を環境の中で区別している。NR は、環境のなかの letrec で束縛されていない変数の部分を取り出す。$FS(\tau)$ は、τ の中に現れるスコーレム型構成子を表す。

成り立つべき命題は以下の通りである。

$(S, \tau, I) = TC(\Gamma, e)$ で、かつ
Γ の中に $::_r$ が含まれなければ、
$S\Gamma \vdash e :: \tau$

TC は再帰関数の中で runST, extendST を使うための、いわば最小の拡張であって、 もちろん principal type を計算できるわけではない。しかし、我々の目的にはこれで十分であると思われる。

実際には領域・エフェクトシステムが必要となる分野で再帰関数を利用することは少ないかも知れないが、この方法だと、領域・エフェクトシステムに関係なく、再帰的な存在型 (existential type) を操作するための再帰関数の型を推論するときなどにも応用可能であると考えられる。

4　プログラム例

extendST を使った再帰関数として、図 7 にマージソートの例を挙げる。もちろんマージソートは、スタックを使わなくても破壊的代入を駆使して行なうことができるが、既存の関数的な定義を少し書きかえてできる、このような定義にも、それなりのメリットはあると思われる。

msort 中でリストを分割するときに、その結果を extendST で作られた拡張領域に置き、この領域の中で msort を再帰的に呼んでいる。split は旧領域から新領域にデータを読み込むために、merge は新領域から旧領域にデータを書き込むために appR を用いている。

これらの関数の型は図 8 のように推論される。なお、下線を引いた部分 ($\underline{\texttt{ST}\ s}$) は、本来は副作用がないので、単なる _ で良いはずであるが、現在の letrec の推論方式ではこうなってしまう。何らかの方法でこの部分を _ にできるように型推論ルールを変更することがより望ましいと思われる。

$$\frac{\Gamma \vdash' e :: \mathtt{ST}\,\kappa\,\tau \quad (\kappa : \text{新しいスコーレム定数}) \quad \kappa \notin FV(\tau) \cup FV(\Gamma)}{\Gamma \vdash' \mathtt{runST}\,e :: \tau}\ (\mathrm{RunST})$$

$$\frac{\Gamma \vdash' e :: \mathtt{Mut}\,\kappa\,t \to \mathtt{ST}\,\kappa\,\tau \quad (\kappa : \text{新しいスコーレム定数}) \quad \kappa \notin FV(\tau) \cup FV(\Gamma)}{\Gamma \vdash' \mathtt{extendST}\,e :: \mathtt{ST}\,t\,\tau}\ (\mathrm{extendST})$$

図 4　型規則 (runST)

5　おわりに

プログラマが明示的にオブジェクトの寿命を指定できる領域・エフェクトシステムを提案した。このシステムでは相対的参照の概念が重要な役割を果たしている。その型推論を説明し、簡単な例を挙げた。

また本文にも述べたが、実用的なシステムにするためには、copyU や split で appR の挿入をいちいちしなくても済むように型システムを工夫する必要がある。そうでなければ、Tofte と Talpin のシステムよりも “シンプル” であるとは主張できないだろう。これは将来の課題である。

現在、型推論アルゴリズムのプロトタイプは、Objective Caml で実装している。Tofte と Talpin の繰り返しによる再帰関数の型推論と計算時間の上での比較も行なってみたいと考えている。

本研究は関数型言語を対象にしているが、もともとの型システムが単純であれば、例えば Java のような命令型あるいはオブジェクト指向型の言語にも適用可能であると思われる。もちろん、オブジェクト指向言語の場合、仮想関数を多用しているプログラムでは (高階関数を多用しているプログラムと同様) 効果が限られる。主に static メソッド、final メソッドを利用する必要がある。最近は組込み機器用の RealTime Java に関する関心が高まっているので、Java Virtual Machine 側の対応がされれば、本研究を応用することも可能であると思われる。関連する研究として、UCB(カリフォルニア大学バークレイ校) の David Gay と Alex Aiken は、Java に明示的な領域を導入し、領域に対する reference counting を行なう手法を提案している [1]。これは実行時に領域の解放ができるかどうかをチェックする方法である。

謝辞

査読者の方々からは有意義なコメントを頂きました。ここに感謝します。本研究は、平成 10 年度文部省科学研究費補助金奨励研究 (A) 10780196 の支援を受けています。

参考文献

[1] David Gay and Alex Aiken. Memory management with explicit regions. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.

[2] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transaction on Programming Languages and Systems*, 15(2):253–289, 1993.

[3] Koji Kagawa. Compositional references for stateful functional programming. In *Proc. of the International Conference on Functional Programming 1997*. ACM Press, June 1997.

[4] Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transaction on Programming Languages and Systems*, 15(2):290–311, 1993.

[5] Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1411–1430, September 1994.

[6] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

[7] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML kit. Technical Report 97/12, Department of Computer Science, University of Copenhagen, April 1997.

[8] Mads Tofte and Jean-Pierre Talpin. Implementation of typed call-by-value λ-calculus using a stack of regions. In *Annual ACM Symp. on Principles of Prog. Languages*, pages 188–201, January 1994.

$TC(\Gamma, x) =$
$\quad (Id, \tau[\overline{\beta}/\overline{\alpha}], \emptyset)$ $\quad$ ここで $\Gamma(x) = \forall\overline{\alpha}.\tau,\ \overline{\beta}$は新しい型変数

$TC(\Gamma, e\, e') =$
$\quad$ let $\quad (S, \tau, I) = TC(\Gamma, e)$
$\quad\quad\quad (S', \tau', I') = TC(S\Gamma, e')$
$\quad\quad\quad \beta$は新しい型変数
$\quad\quad\quad U = mgu(S'\tau, \tau' \to \beta)$
$\quad$ in $\quad (US'S, U\beta, U(S'I \cup I'))$

$TC(\Gamma, \lambda x \to e, I) =$
$\quad$ let $\quad \beta$は新しい型変数
$\quad\quad\quad (S, \tau) = TC(\Gamma + \{x :_n \beta\}, e)$
$\quad$ in $\quad (S, S\beta \to \tau, I)$

$TC(\Gamma, \texttt{let}\ \ x = e\ \texttt{in}\ e') =$
$\quad$ let $\quad (S, \tau, I) = TC(\Gamma, e)$
$\quad\quad\quad (S', \tau', I') = TC(S\Gamma + \{x :_n gen(S\Gamma, \tau)\}, e')$
$\quad$ in $\quad (S'S, \tau', I \cup I')$

$TC(\Gamma, \texttt{runST}\, e) =$
$\quad$ let $\quad (S, \tau, I) = TC(\Gamma, e)$
$\quad\quad\quad U = mgu(\tau, \texttt{ST}\, \kappa\, a)$ $\quad$ (κは新しいスコーレム定数, a は新しい型変数)
$\quad$ in $\quad$ if $\kappa \notin FS(Ua) \cup FS(NR(US\Gamma))$
$\quad\quad\quad$ then $(US, Ua, UI \cup \{(\kappa, Ua, US\Gamma)\})$

$TC(\Gamma, \texttt{letrec}\ x = e\ \texttt{in}\ e') =$
$\quad$ let $\quad e^*$は、e の中の x の各出現を新しい変数 x_iで置き換えた式
$\quad\quad\quad (S, \tau_1, I) = TC(\Gamma + \{x_i :_r a_i\}, e^*)$
$\quad\quad\quad \sigma_i$は Sa_iの中のスコーレム定数の各出現を新しい型変数で置き換えた型
$\quad\quad\quad U = mgu(\bigcup_i \{\sigma_i\} \cup \{\tau_1\})$
$\quad\quad\quad V_i = mgu(U\tau_1, U(Sa_i))$
$\quad$ in $\quad$ if $\forall i.\, \forall \kappa \in FS(Sa_i).\, V_i^{-1}(\kappa) \cap (FV(\tau^\dagger) \cup FV(NR(\Gamma^\dagger))) = \emptyset$ where $(\kappa, \tau^\dagger, \Gamma^\dagger) \in UI$
$\quad\quad\quad$ and $\forall i.\, V_i^{-1}(FS(Sa_i)) \cap FV(U\tau_1) \cap FV(US\Gamma) = \emptyset$
$\quad\quad\quad$ then let $V = \Pi(V_i|_{FV(U(Sa_i))})$
$\quad\quad\quad\quad\quad (S', \tau', I') = TC(VUS\Gamma + \{x :_n gen(US\Gamma, U\tau_1)\}, e')$
$\quad\quad\quad\quad$ in $(S'VUS, \tau', S'VUI[x :_n gen(US\Gamma, U\tau_1)]) \cup I')$

図 5　型推論アルゴリズム

$$
\begin{aligned}
NR(\Gamma) &= \{x :_n \tau \mid (x :_n \tau) \in \Gamma\} \\
gen(\Gamma, \tau) &= \forall (FV(\tau) \setminus FV(\Gamma)).\tau
\end{aligned}
$$

図 6　型推論アルゴリズムの補助関数

```
split t xs l r = if appR t (λ_ → isNull xs) then mkPair l r
                 else if appR t (λ_ → isNull (tail xs)) then
                   mkPair (cons (appR t (λ_ → head xs)) l) r
                 else split t (appR t (λ_ → tail (tail xs)))
                              (cons (appR t (λ_ → head xs)) l)
                              (cons (appR t (λ_ → head (tail xs))) r)

merge t xs0 ys0 = if      isNull xs0 then copyD t ys0
                  else if isNull ys0 then copyD t xs0
                  else var y = head ys0 in
                       var x = head xs0 in
                       if x<y then var xs1 = merge t (tail xs0) ys0 in
                                   appR t (λ_ → cons x xs1)
                       else var ys1 = merge t xs0 (tail ys0) in
                            appR t (λ_ → cons y ys1)

msort xs0 = if (isNull xs0) then nil unit
            else if (isNull (tail xs0)) then xs0
            else extendST (λt →
                   var p = split t xs0 (nil unit) (nil unit) in
                   var l = fst p in
                   var r = snd p in
                   var l1 = msort l in
                   var r1 = msort r in
                   merge t l r)
```

図 7　split, merge, msort の定義

```
split : Mut s t → ST s (Mut t (List t a)
          → ST s (Mut s (List s a) → ST s (Mut s (List s a)
          → ST s (Mut s (Mut s (List s a), Mut s (List s a))))))

merge : Mut s t → ST s (Mut s (List s a)
          → ST s (Mut s (List s a) → ST s (Mut t (List t a))))

msort : Mut s (List s a) → ST s (Mut s (List s a))
```

図 8　split, merge, msort の型