リージョン変数の動的なエイリアス判定によるメモリ効率向上

米田匡史[†] 鵜川始陽[††] 花井亮[††]
八杉昌宏[††] 湯淺太一[††]

リージョン推論と呼ばれる静的にオブジェクトの寿命を見積もる手法に基づくメモリ管理が，Tofte らによって提案されている．リージョン推論に基づくメモリ管理では，オブジェクトはリージョンと呼ばれるメモリブロックのいずれかに生成される．リージョンは特定のスコープを抜けると解放され，そのリージョン内に生成されたオブジェクトも同時に解放される．そのため，再帰呼び出しのように，関数呼び出しが連続する場合にはリージョンの解放ができない．Tofte らの処理系では，今後の計算においてアクセスされる可能性のあるオブジェクトが入っていないリージョンにオブジェクトを生成する際に，そのリージョンに入っているオブジェクトに上書きして生成しようとしている．しかし，関数がリージョンをリージョン変数に受け取ることができるため，リージョン変数のエイリアスが生じ，静的にオブジェクトを上書きしてもよいと判定できる箇所が限られる．本論文では，問題となるエイリアスが存在するかどうかを，実行時にオブジェクトを生成する際に調べる手法を提案する．これにより，Tofte らの手法では上書きしてよいか分からなかったオブジェクト生成の箇所で，より厳密に判定できるようになる．その結果，メモリ効率が向上し，これまで妥当なメモリ使用量で動かなかったプログラムが動くようになると期待される．

Memory Usage Improvement Using Runtime Alias Detection

Masashi Yoneda,[†] Tomoharu Ugawa,[††] Ryo Hanai,[††]
Masahiro Yasugi[††] and Taiichi Yuasa[††]

A technique for memory management based on region inference, which statically estimates the life-time of objects, was proposed by Tofte, et al. With this technique, objects are created in one of the memory blocks, called regions. Each region is deallocated when the control flow exits its corresponding scope and all objects in the region are deallocated at that time. This means that systems cannot deallocate regions while function calls are repeated without returning. This often happens in the case of recursive function calls. Tofte implemented a system which creates a new object by overwriting existing objects in a region if the region has no object that might be accessed in the rest of the computation. However, there are not a few points of object creation at which his static analysis cannot find it possible to overwrite. This is because functions may receive regions as region variables and there may be aliases of region variables. In this paper, we propose a technique to improve memory usage by checking the existence of problematic aliases at runtime. Our technique can determine exactly whether it is possible to overwrite in many points of object creation where Tofte's analysis fails. As a result, we expect more programs to run with a relatively small amount of memory in region-based systems.

1. はじめに

Standard ML [10] のような静的に型付けされた関数型言語におけるメモリ管理の手法として，リージョンを用いたメモリ管理が Tofte らによって提案されている[15]．リージョンを用いたメモリ管理では，オブジェクトはリージョンと呼ばれるメモリブロックのいずれかに生成される．リージョンは特定のスコープを抜けると解放され，そのリージョン内に生成されたオブジェクトも同時に解放される．ML などの言語では，リージョンの割当て，解放を行うコードを自動的に挿入することがある．これはリージョン推論[13] と呼ばれている．リージョン推論は，静的にオブジェクトの寿命を見積もり，リージョンの割当て，解放を行う場所を決定する．また，リージョン推論は，オブジェクトをどのリージョンに生成するかも決定する．

† 日本電気株式会社
NEC Corporation
†† 京都大学大学院情報学研究科通信情報システム専攻
Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University

たとえば，次のプログラム

```
let x = (0, 1) in fn y => #1 x end   (1)
```

は，リージョン推論によって以下のように変換される．

```
letregion ρ2 in
let x = (0 at ρ1, 1 at ρ2) at ρ3
    in (fn y => #1 x) at ρ4 end      (2)
end
```

ここで，`#1` は組の最初の要素を返す関数であり，ρ_i ($i = 1, 2, 3, 4$) はリージョンを表すためのリージョン変数である．e `at` ρ は，e を評価し，その結果として生成されるオブジェクトをリージョン ρ に生成する．e `at` ρ は，評価中にオブジェクトを生成するすべての箇所に現れる．`letregion` ρ `in` e `end` は，静的なスコープを持つリージョン変数 ρ を導入する．この式は，まず新しくリージョンを割り当て，そのリージョンを ρ に束縛する．そして，e を評価した後，ρ に束縛されたリージョンを解放し，e の評価結果を返す．プログラム (1) ではリージョン推論によって，整数 `1` は `let` 式のスコープのみで使われると推論される．そのため，`letregion` ρ_2`...end` が挿入されたプログラム (2) に変換される．ρ_2 以外のリージョン変数は，この式の外側の `letregion` で導入されているとする．

リージョン推論を用いたメモリ管理は，他のメモリ管理に比べ次のような長所を持っている．

- C 言語のような手動のメモリ管理に比べ安全である．
- メモリ管理の操作はすべて定数時間で行われる．これは，特に組み込みシステムのようなリアルタイムプログラミングを行う場合に有効である．
- 静的に不要と分かるオブジェクトは，ポインタの到達可能性を使っている GC に比べ早く解放できる．

しかし，リージョン推論では再帰呼び出しのように，関数呼び出しが連続する場合には呼び出し元で作られたリージョンの解放ができない．Tofte らが実装したリージョン推論を用いた ML コンパイラ MLKit[2),5)] では，今後の計算においてアクセスされる可能性のあるオブジェクトが入っていないリージョンにオブジェクトを生成する際に，そのリージョンに入っているオブジェクトに上書きして生成している．このようなリージョンを上書きしてオブジェクトを生成するかどうかの判定を行う解析は Storage Mode Analysis[3)] と呼ばれる．

リージョン推論を用いたシステムでは，メモリ効率を考えると，関数が引数にリージョンを受け取るリージョン多相性[15)] が必要になる．しかし，リージョン多相性を導入すると，同じリージョンを束縛する複数のリージョン変数，すなわちリージョン変数のエイリアスが生じる．Storage Mode Analysis は，今後の計算においてアクセスされる可能性のあるオブジェクトが置かれているリージョンを解析するため，リージョン変数のエイリアスの存在は，解析の精度を損う．

本論文では上記の問題を解決するために，オブジェクトを上書きして生成してよいかどうかの判断を一部実行時に遅延させる手法を提案する．具体的には，オブジェクトを生成するリージョンに対してエイリアスが存在するかどうかを，実行時に判断する．これにより，Tofte らの手法では上書きしてよいか分からなかったオブジェクト生成の箇所で，より厳密に判定できるようになる．

以下では，2 章でターゲット言語と型を導入し，リージョン推論の概略を述べる．そして，3 章で Storage Mode Analysis について説明した後，4 章で Storage Mode Analysis における問題点と解決策を提案し，その実装について述べる．5 章では評価と考察を行い，6 章で今後の課題を，7 章で関連研究について述べる．最後に，8 章で本研究のまとめを行う．

2. リージョン推論

2.1 ターゲット言語

リージョン推論のターゲット言語の式 e を，次のように定義する[15)]．

$$
\begin{aligned}
e \;::=\;& c\ a \mid x \mid \lambda x.e\ a \mid e_1\, e_2 \\
\mid\;& \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \\
\mid\;& \texttt{letrec}\ f\,[\vec{\rho}]\,(x)\ a\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \\
\mid\;& \texttt{letregion}\ \rho\ \texttt{in}\ e\ \texttt{end} \\
\mid\;& f\ [\vec{a}]\ a \\
a \;::=\;& \texttt{at}\ \rho
\end{aligned}
$$

x や f はプログラム変数，ρ はリージョン変数，c は定数である．また，a はアロケーションディレクティブであり，新しいオブジェクトをどのリージョンに割り当てるかを指定する．$\vec{\rho}$ は空か $\rho_1 \ldots \rho_n$ を，$\vec{a}$ は空か `at` $\rho_1 \ldots$ `at` ρ_n をそれぞれ表す．ここで，a と `letregion` はプログラマが記述する必要はなく，リージョン推論によって自動的に挿入される．

次に式 e の簡潔な説明を行う（詳細な操作的意味論は文献 15) で述べられている）．式 c `at` ρ は定数 c をリージョン ρ にストアし，その定数へのポインタを返す．同様に式 $\lambda x.e$ `at` ρ はクロージャ $\lambda x.e$ をリージョン ρ にストアし，そのクロージャへのポインタを返す．式 $e_1 e_2$ は関数適用を表す．式 `let` x `=` e_1 `in` e_2 `end`

は e_1 の評価結果を変数 x に束縛し，e_2 を評価した結果を返す．式 `letrec` f`[`$\vec{\rho}$`](`x`)` a `=` e_1 `in` e_2 `end` は，リージョン多相関数 f を定義する．`letrec` 式は，x が f の通常のパラメータであり，e_1 は f の本体である．f はさらに 0 個以上のリージョン $\vec{\rho}$ を受け取り，f 自体は a によって与えられるリージョン ρ にストアされる．`letrec` 式は，e_2 を評価した結果を返す．式 `letregion` ρ `in` e `end` は新しいリージョンを作り，ρ にその新しいリージョンを束縛する．それから，e を評価し ρ を解放した後，e の結果を返す．関数 f はリージョン多相関数であり，異なる呼び出しに対して異なるリージョンを引数にして適用ができる．式 f `[`$\vec{a}$`]` `at` ρ は，リージョン多相関数 f を $\vec{a}$ に適用し，その結果であるクロージャをリージョン ρ にストアし，そのクロージャへのポインタを返す．

2.2 リージョン付きの型

リージョン推論は，プログラムの適切な箇所にリージョンの割当て，解放を行うコードを挿入する．このために，型システムによって各式に対し，評価結果がストアされているリージョン，および評価中にアクセスするリージョンの集合であるエフェクト[8),9),12)] を求める．

2.1 節で定義した式を型付けするための型とエフェクトを次のように定義する．ただし，型変数を α，リージョン変数を ρ，エフェクト変数を ϵ とする．

$$\eta ::= \mathbf{get}(\rho) \mid \mathbf{put}(\rho) \mid \epsilon$$
$$\tau ::= \texttt{int} \mid \alpha \mid \mu \xrightarrow{\epsilon.\varphi} \mu$$
$$\mu ::= (\tau, \rho)$$

μ は，リージョン付きの型を表す．η は原子エフェクトと呼ばれる．$\mathbf{get}(\rho)$ は ρ に置かれているオブジェクトを読み出す可能性があることを，$\mathbf{put}(\rho)$ は ρ にオブジェクトをストアする可能性があることを意味する．エフェクト φ は原子エフェクトの集合である．型 τ は整数型か，型変数か関数型である．関数型 $\mu \xrightarrow{\epsilon.\varphi} \mu'$ に現れる $\epsilon.\varphi$ はアローエフェクトと呼ばれる．ここで φ は関数の本体を評価するときのエフェクトである．また，エフェクト変数 ϵ は φ のハンドルであり，エフェクト間の依存を表すのに用いられる．関数が本体の実行中に未知の関数を呼び出す場合，呼び出した関数によるエフェクトは本体の推論だけでは決まらない．そこで，未知の関数のエフェクトをエフェクト変数で抽象化し，関数適用の式に対して推論するときにエフェクト変数を具体化する．

リージョン多相関数が扱えるように，次の単純型スキーム（simple type scheme）σ と複合型スキーム（compound type scheme）π を導入する[15)]．

$$\sigma ::= \forall().\tau$$
$$\pi ::= \forall\rho_1 \cdots \rho_k \alpha_1 \cdots \alpha_n \epsilon_1 \cdots \epsilon_m.\underline{\tau}$$

複合型スキームは `let` および `letrec` で束縛された変数の型を表すのに用いられ，関数適用によって具体化される．複合型スキーム $\forall().\underline{\tau}$ は，単純型スキームと区別する必要がある．$\forall().\underline{\tau}$ は，リージョン多相関数の型でありリージョン引数の空リストを受け取るからである．このため，複合型スキームと単純型スキームを明確に区別するためにアンダーラインを付けている．

リージョン推論では，リージョンの記述を含まない式に対しリージョン付きの型とエフェクトを推論規則に従って推論する．そして，適切な箇所にアロケーションディレクティブを付加し，`letregion` を挿入する．その例をプログラム (1) を使って説明する．`let` 式の本体は，関数クロージャを返す．本体のリージョン付きの型は，$((\alpha,\ \rho_5) \xrightarrow{\epsilon.\{\mathbf{get}(\rho_3)\}} (\texttt{int}, \rho_1),\ \rho_4)$ と推論される．すなわち，ρ_5 にストアされている型が α のオブジェクトを受け取って `y` に束縛し，ρ_1 にストアされている `int` 型のオブジェクトを返す関数型である．そして，この関数は評価中に，ρ_3 にアクセスする．この型には，ρ_2 は含まれていないことが分かるので，`letregion` ρ_2 `in ... end` が挿入され，プログラム (2) を得る．

3. Storage Mode Analysis

リージョン推論では，リージョンを関数の引数に渡すことができる．しかし，再帰呼び出しのように，関数呼び出しが連続する場合に，その呼び出し元で作られたリージョンの解放はできない．MLKit では，このような呼び出し元で作られたリージョンに新しくオブジェクトを生成するときに，リージョン内のオブジェクトを上書きしても安全な場合は，上書きしてオブジェクトを生成する手法を用いている．この手法を Storage Mode Analysis と呼んでいる．Storage Mode Analysis は，各アロケーションポイントで新しくオブジェクトを生成するリージョンに，今後の計算においてアクセスされるオブジェクトが入っている可能性を調べ，適切な方法で処理を行う．このために，アロケーションディレクティブを次のように拡張する．

$$a ::= \texttt{at}\ \rho \mid \texttt{attop}\ \rho \mid \texttt{atbot}\ \rho \mid \texttt{sat}\ \rho$$

リージョンにオブジェクトを生成する際，`attop` の場合はリージョン内の新しい領域にオブジェクトを生成し，`atbot` の場合はすでに入っているオブジェクトに上書きしてオブジェクトを生成する．残りの計算においてリージョン ρ に入っているオブジェクトが使われないとき，`at` ρ は，`atbot` ρ に変換される．`sat`

(“somewhere at”) の場合は，どのように生成するかの決定は実行時まで遅延される．これは，一般に `letrec` で束縛されたときに起こる．Storage Mode Analysis によりすべての `at` は，`attop`，`atbot`，`sat` のいずれかになる．以下で簡単に Storage Mode Analysis について述べる．以下ではリージョン内にあるオブジェクトは解放するが，リージョンは解放しないことをリージョンのリセットと呼ぶことにする．また，関数の静的なスコープ内にあり，今後の計算で使われる可能性のある変数を生きている変数，今後の計算でアクセスされる可能性のあるリージョンを生きているリージョンとする．

Storage Mode Analysis では，アロケーションの際に今後の計算で使われるオブジェクトを解析する．そのため，計算の中間結果に対しても適切な型を必要とする．そこで，Storage Mode Analysis で扱う言語は 2 章の言語を K 正規化[3] した言語とする．つまり，すべての計算の中間結果は変数に束縛されているものと仮定する．

Storage Mode Analysis は以下のような有向グラフ G（Region Flow Graph と呼ぶ）を利用する．G の節は，K 正規化されたプログラムに現れるすべてのリージョン変数とすべてのエフェクト変数である．プログラム中に `letrec` で束縛されたプログラム変数 f があるとき，その型が

$$\pi = \forall \cdots \rho_i \cdots, \vec{\alpha}, \cdots \epsilon_j \cdots . \underline{\tau}$$

であり，次のような f の適用があるとする．

$$f([\cdots \rho_i' \cdots], [\vec{\tau}], [\cdots \epsilon_j'.\varphi_j' \cdots])$$

ただし，上の f の適用はリージョン以外に型とエフェクトに関する具体化も明示的に書いてある．このとき，ρ_i から ρ_i' への枝と ϵ_j から ϵ_j' への枝が存在する．リージョン変数 ρ_i からリージョン変数 ρ_i' への枝の意味は「ρ_i' の指すリージョンが ρ_i に渡されることがある．つまり，ρ_i は，ρ_i' が指すリージョンを指すことがある」である．エフェクト変数に関しても同様である．また，`let` で束縛された変数も `letrec` と同様の枝が存在する．また，プログラム中に現れるすべてのアローエフェクト $\epsilon.\varphi$ に対して，φ に現れるすべての自由なリージョン変数とエフェクト変数に ϵ からの枝が存在する．すなわち，`letregion` で束縛されたリージョン変数はつねに葉（節から枝が出ていない節）であり，リージョン変数からはリージョン変数への枝のみ存在する．G の中のすべての変数 n に対して，n を含めた n から到達できるすべての変数の集合を $\langle n \rangle$ と記述することにする．また，変数の集合 N に対して，その要素から到達できるすべての変数の集合を $\langle N \rangle$ と記述する．

以下では，関数内で局所的に導入されたリージョン変数を使ったオブジェクト生成のみを議論の対象とし，それ以外は `attop` に変換する．変換は，4 つの場合に分けることができる．簡単のため以下の説明において，e は自由変数を持たず，すべての束縛変数は互いに異なり，すべてのリージョン多相関数はリージョン引数を 1 つだけ持つとする．また，新しくオブジェクトを生成するリージョンを ρ_1 とし，ρ_1 にオブジェクトを生成するときに生きていると分かるリージョンを ρ_2 とする．実際には，生きているリージョンは複数存在しうるが，ここでは簡単のため ρ_2 のみ考える．ここで述べることは，ρ_2 がエフェクトに含まれる場合を含む一般の場合についても容易に拡張できる．

ρ_1 が `letregion` で作られた場合，すなわち

$$\texttt{letregion}\ \rho_1\ \texttt{in}\ \ldots(e\ \texttt{at}\ \rho_1)\ldots\texttt{end} \quad (3)$$

の場合，ρ_1 は葉である．ρ_1 が生きていないリージョンであると分かる場合，ρ_1 内のオブジェクトは今後の計算で使われないので，リージョン ρ_1 をリセットしてオブジェクトを生成することができる．そのため，式 (3) は次のように変換できる．

$$\texttt{letregion}\ \rho_1\ \texttt{in}\ \ldots(e\ \texttt{atbot}\ \rho_1)\ldots\texttt{end} \quad (4)$$

ρ_1 が生きているリージョン変数の場合は，ρ_1 のオブジェクトを上書きしてオブジェクトを生成することはできない．そのため，式 (3) は次のように変換する．

$$\texttt{letregion}\ \rho_1\ \texttt{in}\ldots(e\ \texttt{attop}\ \rho_1)\ldots\texttt{end} \quad (5)$$

次に，ρ_1 が `letrec` の仮引数の場合，すなわち

$$\texttt{letrec}\ f\texttt{[}\rho_1\texttt{](}x\texttt{)}\ a\ = \ldots(e_1\ \texttt{at}\ \rho_1)\ldots\texttt{in}\ e_2\ \texttt{end} \quad (6)$$

の場合，**図 1** と**図 2** の場合がある．図 1 は，ρ_2 から $\langle \rho_1 \rangle$ のいずれかに到達することができる場合である．このとき，ρ_1 と ρ_2 が同じリージョンを指す可能性があるので，ρ_1 のオブジェクトを上書きすることはできない．そのため，式 (6) は次のように変換する．

$$\texttt{letrec}\ f\texttt{[}\rho_1\texttt{](}x\texttt{)=}\ \ldots(e_1\ \texttt{attop}\ \rho_1)\ldots\texttt{in}\ e_2\ \texttt{end} \quad (7)$$

図 2 は，ρ_2 から $\langle \rho_1 \rangle$ のいずれにも到達することができない場合である．このとき，ρ_1 が ρ_2 のエイリアスである可能性はない．つまり，f の $(e_1\ \texttt{at}\ \rho_1)$ より後の実行で，この時点で ρ_1 が指すリージョンに置かれているオブジェクトにアクセスすることはない．呼び出し元の関数でもこのことが成り立っていれば，ρ_1 はリセットできる．そこで，リージョン多相関数にリージョン r を渡す際，呼び出し元の関数の残りの実行で，r に置かれているオブジェクトにアクセスするかどうかのフラグも同時に渡す．ρ_1 へのオブジェク

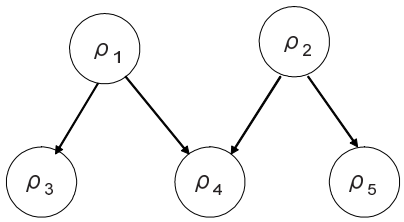


図 1 attop に変換する場合（letrec）
Fig. 1 A case of converting to attop (letrec).

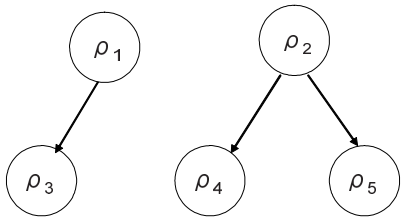


図 2 sat に変換する場合（letrec）
Fig. 2 A case of converting to sat (letrec).

ト生成の際には，このフラグによってリセットするかを決定する．式 (6) は sat を使って，次のように変換し実行時までリセットするかの決定を遅延する．

$$\texttt{letrec } f\texttt{[}\rho_1\texttt{](}x\texttt{) } a = \ldots(e_1 \texttt{ sat } \rho_1)\ldots\texttt{in } e_2 \texttt{ end} \quad (8)$$

これらの変換を規則にまとめると，(9)〜(12) になる．以下では，式 e の中に現れる変数 x に対して，その型を (T,ρ) とする．T は σ か π の形をしている．x を介して参照される可能性のあるリージョン変数の集合を $\bigcup\{\langle\rho\rangle \mid \rho \in \mathit{frv}(T,\rho)\}\cup\bigcup\{\langle\epsilon\rangle\cap \mathit{RegVar} \mid \epsilon \in \mathit{fev}(T)\}$ と定義し，$\mathit{lrv}(x)$ と書くことにする．ここで，RegVar はプログラムに現れるすべてのリージョン変数の集合，$\mathit{frv}(T,\rho)$ は T と ρ 中に現れるすべての自由なリージョン変数の集合，$\mathit{fev}(T)$ は T 中に現れるすべての自由なエフェクト変数の集合である．また，e の中に現れる変数の集合 X に対して，$\mathit{lrv}(X) = \bigcup\{\mathit{lrv}(x) \mid x \in X\}$ と定義する．$R[-]$ は，注目しているアロケーションディレクティブの出現位置に穴（$-$）があるローカルアロケーションコンテキストである．$LV(R)$ は，コンテキスト R の穴で生きている変数の集合を表す（正確な R の定義は文献 3) を参照）．

$$\frac{\rho \notin \mathit{lrv}(LV(R))}{\texttt{letregion } \rho \texttt{ in } R\texttt{[at } \rho\texttt{] end}} \Rightarrow \texttt{letregion } \rho \texttt{ in } R\texttt{[atbot } \rho\texttt{] end} \quad (9)$$

$$\frac{\rho \in \mathit{lrv}(LV(R))}{\texttt{letregion } \rho \texttt{ in } R\texttt{[at } \rho\texttt{] end}} \Rightarrow \texttt{letregion } \rho \texttt{ in } R\texttt{[attop } \rho\texttt{] end} \quad (10)$$

$$\frac{\mathit{lrv}(LV(R)) \cap \langle\rho\rangle = \emptyset}{\texttt{letrec } f\texttt{[}\rho\texttt{](}x\texttt{) } a\texttt{=}R\texttt{[at } \rho\texttt{] in } e \texttt{ end}} \Rightarrow \texttt{letrec } f\texttt{[}\rho\texttt{](}x\texttt{) } a\texttt{=}R\texttt{[sat } \rho\texttt{] in } e \texttt{ end} \quad (11)$$

$$\frac{\mathit{lrv}(LV(R)) \cap \langle\rho\rangle \neq \emptyset}{\texttt{letrec } f\texttt{[}\rho\texttt{](}x\texttt{) } a\texttt{=}R\texttt{[at } \rho\texttt{] in } e \texttt{ end}} \Rightarrow \texttt{letrec } f\texttt{[}\rho\texttt{](}x\texttt{) } a\texttt{=}R\texttt{[attop } \rho\texttt{] in } e \texttt{ end} \quad (12)$$

4. 提 案 手 法

4.1 atif アロケーションディレクティブ

3 章で述べた Storage Mode Analysis では，Region Flow Graph 上で新しくオブジェクトを生成するリージョンと生きているリージョンが同じエイリアスであるとき，そのリージョンはリセットできない．たとえば図 1 において，ρ_1 と ρ_2 が ρ_4 のエイリアスである可能性があるため，ρ_1 をリセットしてオブジェクトを割り当てることができない．しかし，実行時に ρ_1 が ρ_3 のエイリアスである場合，すなわち ρ_1 と ρ_3 が同じリージョンを束縛している場合，ρ_1 が ρ_2 のエイリアスではないため，ρ_1 をリセットすることができる．すなわち，従来はリセットできなかったリージョンに対して，実行時にそのリージョンが生きているかどうかを判定することで，リセットが可能なオブジェクト生成箇所を増やすことができる．

以上のことを実現するために，アロケーションディレクティブに以下のように atif を追加する．

$$a \ ::= \ldots \mid \texttt{atif } \rho \texttt{ in } \{\rho_1,\ldots,\rho_n\}$$

ρ がオブジェクトを割り当てるリージョン変数で，$\{\rho_1,\ldots,\rho_n\}$ が ρ と比較する必要のあるリージョン変数の集合を表す．この集合の要素を比較リージョン，集合を比較リージョン集合と呼ぶことにする．実際に ρ にオブジェクトを割り当てる際は，まず ρ に束縛されているリージョンと，ρ_i（$i = 1,\ldots,n$）のいずれかに束縛されているリージョンが一致するかどうかを判定し，もし一致する場合は attop ρ として動作し，そうでなければ sat ρ として動作する．リージョンに関数を適用する場合も，同様の処理を行う．

4.2 変換規則の変更

我々の手法では，リージョンを引数として受け取る場合のみ atif になりうるので，3 章の規則のうち規則 (12) のみ変更する必要がある．変更後の規則は規則 (13)，(14) になる．ただし，リージョン変数の集合を $LR(R)$=$\bigcup\{\mathit{frv}(T,\rho) \mid x : (T,\rho) \in LV(R)\}$，エフェクト変数の集合を $LE(R)$=$\bigcup\{\mathit{fev}(T) \mid x : (T,\rho) \in LV(R)\}$ と定義する．また，比較リージョン

集合 $C(R,\rho)$ は，$\{\rho' \mid \rho' \in LR(R), \langle\rho\rangle \cap \langle\rho'\rangle \neq \emptyset\}$ とする．ここで，ρ は新しくオブジェクトを生成するリージョンである．これらの集合を用いて規則 (11) を書き直すと規則 (15) のようになる．

$$\frac{\rho \in LR(R) \ \vee \ \langle\rho\rangle \cap \langle LE(R)\rangle \neq \emptyset}{\texttt{letrec}\ f\texttt{[}\rho\texttt{](}x\texttt{)}\ a\texttt{=}R\texttt{[at}\ \rho\texttt{]}\ \texttt{in}\ e\ \texttt{end}} \Rightarrow \texttt{letrec}\ f\texttt{[}\rho\texttt{](}x\texttt{)}\ a\texttt{=}R\texttt{[attop}\ \rho\texttt{]}\ \texttt{in}\ e\ \texttt{end} \quad (13)$$

$$\frac{C(R,\rho) \neq \emptyset \quad \rho \notin LR(R) \quad \langle\rho\rangle \cap \langle LE(R)\rangle = \emptyset}{\texttt{letrec}\ f\texttt{[}\rho\texttt{](}x\texttt{)}\ a\texttt{=}R\texttt{[at}\ \rho\texttt{]}\ \texttt{in}\ e\ \texttt{end}} \Rightarrow \texttt{letrec}\ f\texttt{[}\rho\texttt{](}x\texttt{)}\ a\texttt{=}R\texttt{[atif}\ \rho\ \texttt{in}\ C(R,\rho)\texttt{]}\ \texttt{in}\ e\ \texttt{end} \quad (14)$$

$$\frac{C(R,\rho) = \emptyset \quad \rho \notin LR(R) \quad \langle\rho\rangle \cap \langle LE(R)\rangle = \emptyset}{\texttt{letrec}\ f\texttt{[}\rho\texttt{](}x\texttt{)}\ a\texttt{=}R\texttt{[at}\ \rho\texttt{]}\ \texttt{in}\ e\ \texttt{end}} \Rightarrow \texttt{letrec}\ f\texttt{[}\rho\texttt{](}x\texttt{)}\ a\texttt{=}R\texttt{[sat}\ \rho\texttt{]}\ \texttt{in}\ e\ \texttt{end} \quad (15)$$

$LR(R)$ に ρ が含まれる場合，ρ が生きているオブジェクトが置かれているリージョンであることが分かるので，実行時に判定を行う必要はなく，リセットできないことが分かる．また，$LE(R)$ の要素から到達できるリージョンと，ρ から到達できるリージョンが一致した場合は，その一致したリージョン変数が現在のスコープでは参照できない可能性がある．そのため，これらの条件を満たす場合は `attop` に変換する．

提案手法は，従来では `attop` に変換されたものの一部を `atif` に変換するため，`sat` への変換に関して増減はない．しかし，`letrec` に対する変換規則がすべての場合を満たすように規則 (11) を規則 (13)，(14) にあわせて規則 (15) に変更する．規則 (14)，(15) から `sat` は `atif` の特別な場合（リージョン比較集合が空の場合）であることが分かる．

4.3 変換アルゴリズム

変更した Storage Mode Analysis におけるアロケーションディレクティブを求めるアルゴリズムは，次のようになる．以下で Region Flow Graph G はすでに作られているとする．

(1) 比較リージョン集合 C を空にする．
(2) 新しいオブジェクトを割り当てるリージョン変数 ρ_0 から G 上で到達できるリージョン変数をすべてマークする．
(3) LR と LE を求める．このとき，LR の要素に ρ_0 が入っていた場合 `attop` ρ_0 を返す．
(4) LE の要素 ϵ_i $(i = 1, \ldots, n)$ から G 上で到達できるマークされたリージョン変数に到達した場合 `attop` ρ_0 を返す．
(5) LR の要素 ρ_i $(i = 1, \ldots, m)$ から G 上で到達できるマークされたリージョン変数に到達した場合，比較リージョン集合 C に ρ_i を含める．
(6) 比較リージョン集合 C が空の場合は `sat` ρ_0 を，それ以外の場合は `atif` ρ_0 `in` C を返す．

4.4 コード生成

変更した Storage Mode Analysis によって，プログラムを変換すると，`atif` は 2 種類の箇所で現れる．オブジェクトを生成する箇所とリージョン多相関数呼び出しの引数の位置である．それぞれの場合について，生成されるコードを**図 3** と**図 4** に示す．ここで，$\texttt{reg}_i$ $(i = 1, 2)$ はレジスタとし，$[\rho]$ はリージョン ρ へのポインタとする．生成されるコードは AT&T 形式とする．また，コード中の foreach 文は比較リージョン集合の全要素に対して本体のコードを生成することを意味する．

MLKit では，リージョンへのポインタの下位から 2 ビット目はリージョンをリセットするかのフラグ（リ

```
    mov [ρ0], reg1
    btl $1, reg1
    jnc L1
    foreach ρ in C{
        mov [ρ], reg2
        cmpl reg1, reg2
        je L1
    }
    call resetregion
L1: call allocate
```

図 3 オブジェクト生成時のコード
Fig. 3 Code for object allocation.

```
    mov [ρ0], reg1
    btl $1, reg1
    jnc L2
    foreach ρ in C{
        mov [ρ], reg2
        cmpl reg1, reg2
        je L1
    }
    jmp L2
L1: btrl $1, reg1
L2: push reg1
```

図 4 リージョン引数を受け渡す際のコード
Fig. 4 Code for region parameter passing.

セットする場合は1）が入っている．このフラグは，関数への引数としてリージョンが atbot で渡された場合に1になっている．sat の場合は，このフラグを用いてリージョンをリセットするかどうかを決定する．

図3，図4ともにまずリセットフラグを調べる．もし，0であれば attop として渡されたリージョンなのでリージョンの比較は行わない．次に，比較リージョン集合の全要素に対して比較を行う．図3の場合，等しいリージョンがあれば，リセットせずにアロケートを行い，どの要素とも異なるときは，リセットを行った後アロケートを行う．図4の場合，等しいリージョンがあれば，フラグを0にしてからリージョンをスタックに積む．どの要素とも異なるときは，フラグをそのままにしてリージョンをスタックに積む．

以上のことを，atif に対して行えばよい．ただし，MLKit では，リージョン推論の後の処理でリージョンのサイズを計算し[3),16)]，リージョンのサイズが静的に決定できる場合は，リージョンをレジスタやスタックに割り付ける．このようなリージョンにオブジェクトを生成するときは，リージョンをリセットする必要がないので，比較を行うコードは生成しない．また，比較リージョンがレジスタなどに割り当てられている場合も，そのリージョンに対する比較のコードは生成しない．

4.5 メモリ効率の改善

提案する手法がどのような場合にメモリ効率を改善できるかについて述べる．ここで述べる条件に完全には合わないプログラムでもメモリ効率が改善されることはあるが，ここでは典型的な条件を考える．

関数 f において，

(1) 返り値がリージョン ρ に生成されるときに，ρ 以外のリージョン変数 ρ' に生きている可能性があるオブジェクトが入っており，

(2) ρ と ρ' がリージョン変数のエイリアスになっている可能性がある，

という条件が満たされると，従来の手法ではリージョンのエイリアスが起こるため返り値は，attop で割り当てなければならなかった．しかし，動的なエイリアス判定を行うことで実際にエイリアスになっていない場合には atbot にすることができ，メモリ効率が改善される．条件(1)はたとえば，fun f x y = (x, y) のように，引数として受け取った値を含むようなオブジェクトを返り値とするような関数が該当する．この例では，組 (x, y) を生成する際に x や y が入っているリージョンは生きており，リセットしてはいけない．条件(2)はたとえば以下のような場合が該当する．

- f がトップレベルで定義されている場合．トップレベルで定義された関数は他のコンパイルモジュールから呼び出される可能性があるため，どのようなリージョンが引数に渡されるか分からない．そのため，MLKit ではこの関数のリージョン引数は互いにエイリアスしている可能性があるとしてコンパイルする．
- f のリージョン引数 ρ と ρ' に同じリージョンを渡す場合．たとえば，if e then x else f x の式では型付けの制約から，x と f x の返り値は同じリージョンに置かれなければならない．そのため，この呼び出しでは ρ と ρ' に同じリージョンを渡すことになる．その結果，これ以外の f の呼び出しで ρ と ρ' に異なるリージョンを渡したとしても，関数 f 内では ρ と ρ' がエイリアスしている可能性があるとしてコンパイルする．
- f が2カ所以上で呼び出され，ある呼び出しでは ρ に渡すリージョンを，別の呼び出しでは ρ' に渡す場合．たとえば，式 (f (f x)) は (f[ρ_1, ρ_2] (f [ρ_2, ρ_3] x)) と変換される．その結果，実際には異なるリージョンが渡されているにもかかわらず，f 内では ρ と ρ' がエイリアスしている可能性があるとしてコンパイルする．

さらに，上記の条件を満たす関数 f を繰り返し呼び出し f の結果を返すと，従来の手法では f の返り値を置くリージョンに不要なオブジェクトがたまってしまう．これに対して，提案手法では f の返り値を生成するたびに以前に生成した返り値のメモリを解放でき，顕著にメモリ効率が改善される．

上記の条件を満たしたプログラムであっても，MLKit では整数や真偽値の unboxing やインライン展開といった最適化により，条件を満たさないプログラムに書き換え，不要なオブジェクトが蓄積しないようにできることがある．しかし，すべてのプログラムがそうなるわけではない．

たとえば，条件を満たすプログラム compound interest（**図5**）を考える．このプログラムは，複利計算を行うプログラムで10年で元本が10倍になるために必要な金利を求める．このプログラムでは，power は再帰呼び出しをするのでインライン展開はできない．このプログラムにおいて，power の返り値は acc + 0.0 で生成された浮動小数点数である．このとき acc は生きている．ここで 0.0 を足しているのは，acc * r で作られるオブジェクトを power 内で作ったリージョンに入れるためであり，この種の書き換えはリー

```
fun power 0 acc r = acc + 0.0
  | power n acc r = power (n - 1) (acc * r) r

val _ =
  let fun loop i =
    let val r = (1.0 + 0.0000001 * (real i))
        val x = power 10 1.0 r
    in
      if x > 10.0 then (r, x)
      else loop (i + 1)
    end
  in
    let val (r, x) = loop 0
    in
      print ((Real.toString r) ^ " ^ 10 = " ^
             (Real.toString x) ^ " > 10\n")
    end
  end
```

図 5 メモリ効率改善が顕著な例
Fig. 5 An example program where memory usage is significantly improved.

ジョン推論では定石☆である．なお，MLKit では浮動小数点数は unboxing されない．また，`power` は他のコンパイルモジュールから呼び出すことが可能であるため，`acc` の値が入っているリージョン変数と，返り値の入っているリージョン変数がエイリアスである可能性がある．

関数 `loop` は `power` を繰り返し呼び出し，最後に呼び出した結果を返す．したがって，`power` は計算結果を `loop` の呼び出し元が作ったリージョンに置くことになり，各 `power` の結果が蓄積される．しかし，我々の手法を用いると `power` の結果は `atif` で生成され，`power` の呼び出しごとにリセットされる．

5. 評　　価

4 章で述べた手法を MLKit [14] に実装し，その性能を測定した．ベンチマークプログラムには，MLKit のテストプログラムである dangle，Knuth-Bendix，FFT，Mandelbrot と 4.5 節で示した例である compound interest（図 5）を用いた．今回評価に用いた実行環境は以下のとおりである．

OS: Debian Linux (sarge)
Kernel: 2.4.27
CPU: Pentium 4 3.0 GHz
Memory: 512 MB
MLKit: 4.1.4

提案手法によってコンパイルしたものを実行したときに，リージョンの比較が行われた回数，従来と提案手法のリセットの回数，増加したリセットの回数とその増加率を**表 1** にまとめる．Mandelbrot は，リージョンの比較およびリセットの増加がないため，`atif` に変換されなかったことが分かる．dangle，Knuth-Bendix，FFT では，リージョンの比較が行われ，さらにリージョンのリセットも増えている．compound interest においてはリージョンの比較が行われていないが，リージョンのリセットは増加している．これは，4.4 節で触れたように，比較リージョンがスタックやレジスタに割り当てられるときには比較を省いているためである．Knuth-Bendix はリセットの増加率は小さいが，リセットが増加した他のプログラムにおいては 2 倍程度リセットが増加している．

従来手法を 1 としたときの提案手法における実行中のリージョンのページの最大数と実行時間に関するグラフが**図 6** である．MLKit では，リージョンは `malloc` を使って確保されたページのリストで実現されており[4]，オブジェクトはページに割り当てられる．オブジェクトを生成する際にリージョンの空き領域が足りない場合，フリーリストから新しいページを取ってくる．実行におけるメモリの最大使用量は，リージョンのページの最大数と考えることができる．そのため，評価結果にはメモリ使用量としてリージョンページの最大数を載せる．また，今回リージョンのページサイズはデフォルトで使われている 1,016 Byte とした．

図 6 において，`atif` に変換されなかった Mandelbrot に関しては，コンパイル結果は変わらないのでページ最大数と実行時間ともに結果は変わらない．

dangle と compound interest は，リージョンのリセットを行うことで，必要なリージョンのページ数が減り，その結果リージョンの最大ページ数も減っている．特に compound interest は使用メモリ量が従来手法では 41,111 ページに対して提案手法では 12 ページと大幅に減っており，提案手法によるリージョンのリセットの増加が特に効果的であったことが分かる．これは，従来手法ではループの反復回数に比例して不要なオブジェクトが蓄積されていたが，提案手法ではそのようなオブジェクトが繰返しごとに解放されているためである．compound interest は従来手法では，非常に多くのページを使っており，`malloc` によるページの確保が繰り返されていたが，提案手法では妥当なメモリ量で動くようになった．

Knuth-Bendix と FFT はリージョンのリセットが行われているが，リージョンの最大ページ数に変化は見られない．これは，リージョンのリセットは行われ

☆ Advanced Topics in Types and Programming Languages [11] の p.129 においても同様の書き換えが行われている．

表 1 比較回数とリセット回数
Table 1 Number of region comparisons and resettings.

	number of comparing regions	number of resettings in conventional method, C	number of resettings in our method, O	increased number of resettings (100(O-C)/C%)
dangle	3,000	3,006	6,006	3,000 (99.8%)
Knuth-Bendix	33,134	2,916,741	2,920,233	3,492 (1.2%)
FFT	1,245,184	262,147	655,362	393,215 (150%)
Mandelbrot	0	6	6	0 (0%)
compound interest	0	2,589,256	5,178,512	2,589,256 (100%)

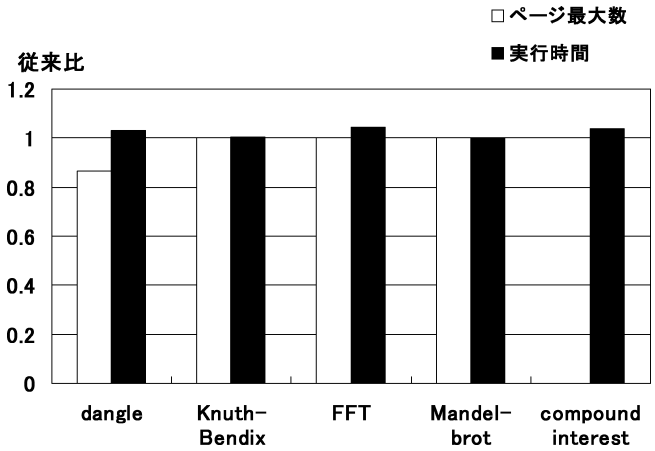


図 6 ページの最大数と実行時間
Fig. 6 Max number of allocated pages and elapsed time.

るが，その後にそのリージョンのページサイズを超えるほどオブジェクトが生成される前にリージョンが解放されるから，もしくはリセットが行われた箇所と別の箇所でより多くのリージョンのページが要求されたからと考えられる．

dangle，Knuth-Bendix，FFT，compound interest に関しては，リージョンの比較とリセットが起こっているため，実行時間が増加している．ただし，compound interest に関しては従来手法ではページが不足したときに，ページの確保のための `malloc` が何度も行われているが，提案手法ではページの要求が従来手法よりもなされないため，`malloc` の処理の分だけ実行時間は減る．FFT と Knuth-Bendix に関してはページの最大数が減っていないことから，`malloc` の回数は従来手法と提案手法で変わっていないことが分かる．dangle に関してはリージョンページの最大数は減っているにもかかわらず，`malloc` の回数は変わらなかった．これは，`malloc` が起こるのはフリーリストにページが足りなくなったときだからである．このため，dangle と FFT に関して実行時間が増えているのは，純粋にリージョンの比較とリセットによるオーバヘッドと考えられる．Knuth-Bendix も同様に比較とリセットが起こっているが，実行時間の変化は小さい．これはリセットの増加率が小さいためと考えられる．compound interest においては，`malloc` の起こる回数が減ったにもかかわらず実行時間が増えていることから，比較とリセットのオーバヘッドの方が大きいことが分かる．ただし，メモリがより少ない環境では実行時にメモリが足りなくなりスラッシングが起こることがあり，そのような場合には従来手法よりも実行時間が短くなる．

実行時間の増加には，比較によるものとリージョンのリセットによるものがあるが，主にリセットのためと考えられる．リージョンの比較は，4.4 節で述べたようにアセンブリ命令で数命令であるのに対し，リセットは現在 C のランタイム関数を呼び出し，その関数内ではページの管理など複雑な処理が行われているためである．また，`atif` アロケーションディレクティブの比較リージョン集合に含まれるリージョン変数の数は，ほとんどの場合 1 つ程度である．そのため，`atif` が現れる位置において，リージョンの比較によるオーバヘッドは小さいと考えられる．

6. 今後の課題

提案手法では，オブジェクトを生成するリージョンが，生きている変数の型に現れる自由なエフェクト変数（4.2 節の LE）から到達できるリージョンのエイリアスである可能性があるときには，`attop` に変換している．しかし，このエイリアスに関しても実行時に比較することでリセットできる場合がある．このような場合，すべてを `attop` に変換するのではなく，クロージャに比較リージョンを閉じ込めることが可能なリージョンに関しては，`atif` に変換することで，リセットできるリージョンが増加する．ただし，この場合はクロージャのサイズが従来よりも大きくなる．

また，現在の実装ではリージョンの比較を `atif` のアロケーションディレクティブに対してつねに行っている．一度行った比較の結果を再利用することによって，無駄な比較を省くことができる．

7. 関連研究

リージョン推論を用いたシステムでは，リージョンが解放されるまで，リージョン内に入っているオブジェクトがごみになっても解放することができない．Storage Mode Analysis は今後の計算でアクセスされるオブジェクトが入っていないリージョンをリセットすることで，この問題を改善している．我々の方式は Storage Mode Analysis を改良し，よりメモリ効率を高めている．

同じ問題に対する別の解決として，ごみ集め（GC）と併用した手法[6] が提案されている．この手法の長所は，リージョン推論では解析できなかったごみを解放できるようになることである．しかし，リージョン推論だけを用いる場合と異なりプログラムの実行が決定的ではなくなり，実行時に停止時間が生じる可能性もある．また，リージョン推論と GC を組み合わせるためには，ダングリングポインタが発生しないようなリージョン推論を用いるか，実行時にリージョンと型の情報を持たせる必要があるという短所もある．

これとは別に，Aiken らが作ったシステム[1] や Henglein らのシステム[7] も Storage Mode Analysis とは別のアプローチで同じ問題に対処している．Storage Mode Analysis では，リージョン変数の導入と同時にリージョンが割り当てられ，リージョン変数のスコープを抜けるときにリージョンが解放される．それに対して，Aiken らのシステムや Henglein らのシステムでは，リージョン変数のスコープと，リージョンの割当ておよび解放を切り離すことでメモリ効率を改善している．そのため，再帰呼び出し中にリージョンの解放ができる可能性があるが，Tofte らのシステムでは可能であったリージョンをスタックやレジスタに割り当てる最適化が難しい．

8. おわりに

本論文では，リージョン推論を用いたシステムのメモリ効率を改善するために，実行時にリージョン変数のエイリアスを判定する手法を提案した．その手法を実装し，評価を行うことで，実際にメモリ効率が改善される例を示した．

リージョン推論に基づくシステムでは，導入で述べたように多くの利点を備える一方，ユーザが推論システムを意識したコードを書かない限り，妥当なメモリ使用量では動作しない場合も少なくない．本研究は，ユーザがシステムを意識せずに書けるプログラムの範囲を広げるものであり，実用的なシステムの応用に向けて大きな意義があると考える．

参考文献

1) Aiken, A., Fahndrich, M. and Levien, R.: Better Static Memory Management: Improving Region-Based Analysis of Higher-Order Languages, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.174–185 (1995).
2) Birkedal, L., Rothwell, N., Tofte, M. and Turner, D.N.: The ML Kit (Version 1), Technical Report 14, Department of Computer Science, University of Copenhagen (1993).
3) Birkedal, L., Tofte, M. and Talpin, J.-P.: From region inference to von Neumann machines via region representation inference, *The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.171–183, ACM Press (1996).
4) Elsman, M. and Hallenberg, N.: An Optimizing Backend for the ML Kit Using a Stack of Regions, Technical Report, Department of Computer Science, University of Copenhagen (1995).
5) Elsman, M. and Hallenberg, N.: A Region-Based Abstract Machine for the ML Kit, Technical Report TR-2002-18, Royal Veterinary and Agricultural University of Denmark and IT University of Copenhagen, IT University Technical Report Series (2002).
6) Hallenberg, N., Elsman, M. and Tofte, M.: Combining Region Inference and Garbage Collection, *ACM SIGPLAN Conference on Pro-*

gramming Language Design and Implementation (*PLDI '02*), Berlin, Germany, ACM Press (2002).
7) Henglein, F., Makholm, H. and Niss, H.: A direct approach to control-flow sensitive region-based memory management, *Proc. 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming* (*PPDP*), Montréal, Canada, pp.175–186, ACM (2001).
8) Jouvelot, P. and Gifford, D.: Algebraic Reconstruction of Types and Effects, *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, pp.303–310, ACM Press (1991).
9) Lucassen, J.M. and Gifford, D.K.: Polymorphic effect systems, *POPL '88: Proc. 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, pp.47–57, ACM Press (1988).
10) Milner, R., Tofte, M., Harper, R. and MacQueen, D.: *The Definition of Standard ML — Revised*, MIT Press (1997).
11) Pierce, B.C. (Ed.): *Advanced Topics in Types and Programming Languages*, MIT Press (2005).
12) Talpin, J.-P. and Jouvelot, P.: Polymorphic Type, Region and Effect Inference, *Journal of Functional Programming*, Vol.2, No.3, pp.245–271 (1992).
13) Tofte, M. and Birkedal, L.: A region inference algorithm, *ACM Trans. Program. Lang. Syst.*, Vol.20, No.4, pp.724–767 (1998).
14) Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N., Olesen, T.H. and Sestoft, P.: Programming with Regions in the ML Kit (for Version 4), Technical Report, IT University of Copenhagen (2001).
15) Tofte, M. and Talpin, J.-P.: Implementing the call-by-value lambda-calculus using a stack of regions, *The 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.188–201, ACM Press (1994).
16) Vejlstrup, M.: Multiplicity Inference, Master's thesis, Department of Computer Science, University of Copenhagen (1994).

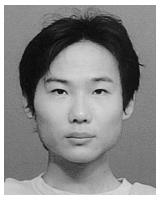
(平成 17 年 12 月 16 日受付)
(平成 18 年 3 月 22 日採録)

米田 匡史

2004 年京都大学工学部情報学科卒業．2006 年同大学大学院情報学研究科通信情報システム専攻修士課程修了．現在，日本電気株式会社に在籍．

鵜川 始陽

1978 年生．2000 年京都大学工学部情報学科卒業．2005 年同大学大学院情報学研究科博士課程修了．同年より同研究科特任助手．博士（情報学）．関数型言語，プログラミング言語処理系，プログラムの形式検証に興味を持つ．



花井 亮

2001 年京都大学理学部卒業．2003 年同大学大学院情報学研究科通信情報システム専攻修士課程修了．現在，同専攻博士後期課程に在籍．プログラミング言語処理系，並列処理に興味を持つ．

八杉 昌宏（正会員）

1967 年生．1989 年東京大学工学部電子工学科卒業．1991 年同大学大学院電気工学専攻修士課程修了．1994 年同大学院理学系研究科情報科学専攻博士課程修了．1993〜1995 年日本学術振興会特別研究員（東京大学，マンチェスター大学）．1995 年神戸大学工学部助手．1998 年京都大学大学院情報学研究科通信情報システム専攻講師．2003 年より同大学助教授．博士（理学）．1998〜2001 年科学技術振興事業団さきがけ研究 21 研究員．並列処理，言語処理系等に興味を持つ．日本ソフトウェア科学会，ACM 各会員．

湯淺 太一（フェロー）

1977 年京都大学理学部卒業．1982 年同大学大学院理学研究科博士課程修了．同年京都大学数理解析研究所助手．1987 年豊橋技術科学大学講師．1988 年同大学助教授，1995 年同大学教授，1996 年京都大学大学院工学研究科情報工学専攻教授．1998 年同大学院情報学研究科通信情報システム専攻教授となり現在に至る．理学博士．記号処理，プログラミング言語処理系，並列処理に興味を持っている．著書『Common Lisp 入門』（共著），『C 言語によるプログラミング入門』，『コンパイラ』ほか．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．