型輪講
第13章 References

小林 義徳

平成15年5月13日

概 要

この章では、$\lambda_{\rightarrow}$with Unit に Reference を追加する。それに伴い、新たな概念である store (ヒープ) および store typing を導入する。

また、よく型付けされた term は何回評価しても stuck にならない[1]、ということを示すために、preservation theorem, progress theorem を示す。最後に、Reference を使った再帰の書き方を述べる。再帰が書けてしまうので、$\lambda_{\rightarrow}$ に Reference を導入すると正規性 (Normalization[2]) は成り立たなくなる。

1 Reference とは

1.1 Basics

Reference とは、value が入った書き換え可能な cell への参照のことである。この際、cell が何なのかについては、ひとまず置いておく。基本操作は以下の通りである。

- allocation (新しい cell を作成、結果は作られた cell への参照)
- dereferencing (参照の指す cell の中身を取り出す)
- assignment (cell の中身を新しい value に書き換える)

例

```
       Objective Caml version 3.05

# let r = ref 5;;   (* 新たな cell を作成 (allocation)、それへの参照を r に代入 *)
val r : int ref = {contents = 5}
# !r;;              (* r の指す cell の中身を参照 (dereference) *)
- : int = 5
# r := 7;;          (* r の指す cell の中身を 7 に変更 (assignment) *)
- : unit = ()
# !r;;              (* r の指す cell の中身を参照 (dereference) *)
- : int = 7
#
```

[1] value でなく、かつどの評価規則も適用できない term

[2] 一般に、$\lambda_{\rightarrow}$ の範囲で型付けできる term は、reduction が止まるという性質。Normalization は再帰が入った言語では成り立たない。

また、複数の reference が同じ cell を指す、という状態をつくることもできる (Aliasing) この際、片方の refernce を通して代入した値が、もう一方の reference から読み出される。

```
# let r = ref 13;;
val r : int ref = {contents = 13}
# let s = r;;
val s : int ref = {contents = 13}
# s := 82;;
- : unit = ()
# !r;;
- : int = 82
#
```

1.2 References to Compound Types

さらに、refernce の参照する cell の中身は、関数値でもよい。以下に、配列 (のようなものの効率の悪い実装) を $int \to int$ 型の関数値への reference を使って実現する例を示す。ここで言う「配列」とは、update によって値が更新された index については lookup 時にその値が返される。そうでない index については、0 が返される。

```
# type natArray = (int -> int) ref;;
type natArray = (int -> int) ref
# let newArray (_:unit) = ref (fun (n:int) -> 0);; (* 新しい配列の作成 *)
val newArray : unit -> (int -> int) ref = <fun>
# let lookup (a:natArray) (n:int) = (!a) n;;  (* 配列 a の n 番目の要素を取り出す *)
val lookup : natArray -> int -> int = <fun>
# let update (a:natArray) (m:int) (v:int) =
  let oldf = !a in
  a := (fun (n:int) -> if m = n then v else oldf n);; (* 配列 a の m 番目の要素を v に
書き換える *)
val update : natArray -> int -> int -> unit = <fun>
#
```

動作は簡単には、以下の通り。

```
# let a := newArray ();;
```

とした直後、a の指す cell には

```
 fun (n:int) -> 0
```

が入っている。

```
# update a 3 2;;
```

とすると、a の指す cell が更新され、内容が

```
 fun (n:int) -> if n = 3 then 2 else (fun (n:int) -> 0) n
```

となる。例えば、

```
# lookup a 4;;
```

は !a 4 となり、結果は 0:int である。
ちなみに、

```
# let update (a:natArray) (m:int) (v:int) =
  a := (fun (n:int) -> if m = n then v else (!a) n);;
```

としてしまうと、うまく行かない (簡単ですので各自考えてみましょう)。

1.3 Garbage Collection

Reference に対する基本操作には deallocation (reference の指す cell の明示的な解放) がないことに注意されたい。ML など多くの 言語処理系では、deallocation が無いかわりに、プログラムから使うことのできなくなった cell は、ランタイムシステムである Garbage Collectior によって回収される。

というのも、明示的な deallocation が存在する場合、型による安全性を保証するのが非常に難しいからである。これば、deallocation によって dangling reference が生じてしまい、それによって指されている cell が再利用されたときに、型エラーを起こすためである。下の例では、r の指していた cell が t により再利用された場合、succ(!s) の部分で型エラーを起こす。

```
# let r = ref 0 in
  let s = r in
  free r;
  let t = ref true in t := false;
  succ(!s);;
```

2 Reference の追加

$\lambda_{\to}$with Unit に Reference を追加する。

2.1 Typing

ref (allocation), := (assignment), ! (dereference) の型付け規則は以下のようになる。

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{ref } t_1 : \text{Ref } T_1} \text{ (T-REF)}$$

$$\frac{\Gamma \vdash t_1 : \text{Ref } T_1}{\Gamma \vdash !t_1 : T_1} \text{ (T-DEREF)}$$

$$\frac{\Gamma \vdash t_1 : \text{Ref } T_1 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 := t_2 : \text{Unit}} \text{ (T-ASSIGN)}$$

2.2 Operational Semantics

2.2.1 Store

ここで、型 Ref T を持つ value って何？という問題が生じる。そこで、store (ヒープ) および、型 Ref T を持つ value として、store location という概念を導入する。

$$\mathcal{L} \quad : \quad \text{store locations の集合} \tag{1}$$

$$store \quad : \quad \mathcal{L} \to \text{values} \tag{2}$$

直感的には、 store locations = ヒープ上のアドレス、と思ってよい[3]。allocation の際には、store のどこかに value を格納されるだけの cell が確保され、それへの参照が、store location として返されるものとする。このようにすると、store location が型 Ref T を持つ value として定義できる。

また、store の状態も評価の際に考慮に入れなければならなくなる。これは、ある term t の評価の際に、副作用によって store の状態が書き換わり、それが将来他の term の評価に効いてくるという場合があるためである。このため、今までの評価規則の、$t \longrightarrow t'$ の部分を、すべて $t \mid \mu \longrightarrow t' \mid \mu'$ の形に改める。ここで、μ は store の状態、つまり、どの location にどの value が対応しているかを表し、しばしば $(l_1 \mapsto v_1, l_2 \mapsto v_2)$ のように書かれる。

既存の評価規則は、次のように書き換えられる。

$$(\lambda x : T_{11}.t_{12})v_2 \mid \mu \longrightarrow [x \mapsto v_2]t_{12} \mid \mu \text{ (E-APPABS)}$$

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{t_1 t_2 \mid \mu \longrightarrow t_1' t_2 \mid \mu'} \text{ (E-APP1)}$$

$$\frac{t_2 \mid \mu \longrightarrow t_2' \mid \mu'}{v_1 t_2 \mid \mu \longrightarrow v_1 t_2' \mid \mu'} \text{ (E-APP2)}$$

Syntax に対する追加は次の通りである。

v ::=

- $\lambda x : T.t$
- $unit$
- l (store location)

t ::=

- x
- $\lambda x : T.t$
- $unit$
- ref t
- $!t$
- $t := t$
- l

[3]そうでない場合も多い。これは実装依存の話なので、ここではこれ以上の議論はしない。

Syntax 中に store location l が現れている。location はプログラマが明示的に書くことを意図したものではなく、評価の途中の結果として使われるものである。

Reference の評価規則については、別紙を参照されたい。[4]

また、Garbage Collection は評価結果の正しさには影響しないため、ここでは議論しないことにする。GC をモデル中に入れることは可能ではある。

2.3 Store Typing

ここでの鍵となる問題は、「location l の型は何？」である。これは、store の中身に依存する。例えば、 $!l_2$ の型は store $(l_1 \mapsto unit, l_2 \mapsto 3)$ の下では Nat となり、store $(l_1 \mapsto unit, l_2 \mapsto unit)$ の下では Unit となる。

location の型規則としてまず思いつくのは、

$$\frac{\Gamma \mid \mu \vdash \mu(l) : T_1}{\Gamma \mid \mu \vdash l : Ref(T_1)}$$

とすることである。しかしこれではうまく行かない。

まず、型チェックの際 reference の中身を見に行かなければならないため、効率が悪い。例えば store が

$$(l_1 \to \lambda x : Nat.0, l_2 \to \lambda x : Nat.(!l_1)x, l_3 \to \lambda x : Nat.(!l_2)x)$$

のとき、l_3 の型を計算する際、l_2, l_1 の型も計算する必要がある。

また、型チェックが止まらない場合がある。例えば、

$$\begin{aligned} letr1 &= ref(\lambda x : Nat.0)in \\ letr2 &= ref(\lambda x : Nat.(!r1)x)in \\ (r1 &:= \lambda x : Nat.(!r2)x); r2);; \end{aligned}$$

とすると、store の状態が

$$(l_1 \mapsto \lambda x : Nat.(!r2)x, l_2 \mapsto \lambda x : Nat.(!r1)x)$$

となるので、$r2$ の型チェックが止まらない。

そこで、 store typing と呼ばれるものを導入する。 store μ が location に value を bind したものだったのに対し、 store typing Σ は、 location に 型を bind したものである。同じ location には、同じ型の value しか代入することができないため、このようなことが可能である。例えば、store typing Σ が $(l_1 \mapsto Unit, l_2 \mapsto Unit \to Unit)$ のとき、$!l_2$ の型は $Unit \to Unit$ となる。

Store Typing 導入後の型規則については、別紙を参照。

3 Safety (= Progress + Preservation)

この節では、progress theorem (よく型付けされた項は stuck でない)、および preservation theorem (型 T をもつ項が一ステップ評価されたとき、その結果も 同じ型 T をもつ) をいう。

始めに、preservation について述べる。$\lambda_{\to}$ では、以下の通りであった。

[4] E-REFV : store 中でまた value に bind されていない location を取ってきて、値 v_1 を bind

定理 ($\lambda_{\to}$ での preservation)

$\Gamma \vdash t : T$ かつ、$t \longrightarrow t'$ のとき、$\Gamma \vdash t' : T$。

証明: $\Gamma \vdash t : T$ の導出に関する帰納法で証明済。

しかし、これをそのまま Reference で拡張した $\lambda_{\to}$ に持ってきて、「$\Gamma \mid \Sigma \vdash t : T$ かつ $t \mid \mu \longrightarrow t' \mid \mu'$ のとき、$\Gamma \mid \Sigma \vdash t' : T$ である」とするのは間違い。この条件だと、store μ と store typing Σ の一貫性がとれてない場合についても preservation を言ってしまっているためである。

定義

store μ が typing context Γ, store typing Σ の下で well typed であるとは、$dom(\mu) = dom(\Sigma)$ かつ、全ての $l \in dom(\mu)$ に対し $\Gamma \mid \Sigma \vdash \mu(l) : \Sigma(l)$ なことをいい、

$$\Gamma \mid \Sigma \vdash \mu$$

と書く。

これを用いて、「$\Gamma \mid \Sigma \vdash t : T, t \mid \mu \longrightarrow t' \mid \mu', \Gamma \mid \Sigma \vdash \mu$ のとき、$\Gamma \mid \Sigma \vdash t' : T$」とするのも、さっきよりはましだが、まだ正しいとはいえない。ほとんどの評価規則に関してはこの文は正しいが、allocation の評価規則 E-REFV についてのみ間違っている。allocation により、μ' には新しい location l の binding が入っている。しかし、l の binding が store typing Σ に入っていない。したがって、$t'(l$ を含む) は Σ のもとで型付けできない。

以下に示すものが正しい preservation theorem である。

3.1 定理 (preservation)

$$\begin{aligned} \Gamma \mid \Sigma \quad &\vdash \quad t : T \\ \Gamma \mid \Sigma \quad &\vdash \quad \mu \\ t \mid \mu \quad &\longrightarrow \quad t' \mid \mu' \end{aligned}$$

のとき、ある $\Sigma' \supseteq \Sigma$ なる Σ' に対し、

$$\begin{aligned} \Gamma \mid \Sigma' \quad &\vdash \quad t' : T \\ \Gamma \mid \Sigma' \quad &\vdash \quad \mu' \end{aligned}$$

が成り立つ。これを証明するにあたって、いくつかの補題を使う。

3.1.1 補題 (substitution)

$\Gamma,\ x : S \mid \Sigma \vdash t : T,\ \Gamma \mid \Sigma \vdash s : S$ のとき、$\Gamma \mid \Sigma \vdash [x \mapsto s]t : T$ が成り立つ。証明は、型の導出に関する帰納法による。

3.1.2 補題

$\Gamma \mid \Sigma \vdash \mu,\ \Sigma(l) = T,\ \Gamma \mid \Sigma \vdash v : T$ のとき、$\Gamma \mid \Sigma \vdash [l \mapsto v]\mu$ が成り立つ。

証明: $\Gamma \mid \Sigma \vdash \mu$ の定義より明らか。同じ location に、同じ型で違う値をいれても、μ と Σ の一貫性は保たれる、ということを言っている。

3.1.3 補題

$\Gamma \mid \Sigma \vdash t : T, \Sigma \subseteq \Sigma'$ のとき、$\Gamma \mid \Sigma' \vdash t : T$ が成り立つ。

証明: 簡単。

preservation theorem の証明は、以上の補題と、inversion property [5]を用い、型の導出に関する帰納法で行うことができる。

3.2 定理 (progress)

閉じた term t が、よく型付けされているとする ($\phi \mid \Sigma \vdash t : TforsomeTand\Sigma$)。このとき、$t$ が value であるか、または $\phi \mid \Sigma \vdash \mu$ なる任意の store μ に対し、ある term t' と ある store μ' が存在し、$t \mid \mu \longrightarrow t' \mid \mu'$ を満たす。

証明は、型の導出に関する帰納法で行うことができる。その際、canonical forms lemma[6] を使う。

4 Reference を使った再帰

最後に、reference を使って再帰関数を実現する方法を示す。この方法で再帰を実装している関数型言語の処理系もある。以下に示す例は 階乗を計算する再帰関数である。

1. cell を一つ allocate する。この際、型が同じである dummy の関数で初期化する。

```
# let fact_ref = ref (fun (n:int) -> 0);;
val fact_ref : (int -> int) ref = {contents = <fun>}
```

2. 関数の本体を記述する。再帰的に自分を呼び出す部分については、上で作った dummy の関数で書いておく。

```
# let fact_body = fun (n:int)-> if n = 0 then 1 else n * ((!fact_ref)(n-1));;
val fact_body : int -> int = <fun>
```

3. reference cell に関数の本体を代入する

```
# fact_ref := fact_body;;
- : unit = ()
```

4. reference cell の内容を取り出し、使う。

```
# let fact = !fact_ref;;
val fact : int -> int = <fun>
# fact 5;;
- : int = 120
```

[5]例えば、$\Gamma \vdash x : R$ のとき $x : R \in \Gamma$ など

[6]例えば、v が 型 Bool の value のとき、v = true または v = false など

この方法により再帰が記述できてしまうので、$\lambda_{\rightarrow}$ with References については、Normalization が成り立たない。