TAPL Chap11
Simple Extensions

Kinebuchi Tomohiko

May 7, 2011

言語の拡張

普段見慣れているプログラミング言語に近付けよう.
→ 各種言語要素を追加していく.
derived form とかが出てくる.

11.1 Base Type

Base Type の集合を抽象的に $\mathcal{A}$ と表記する.
個々の Base Type は A, B, C と表記する.

11.2 The Unit Type

Unit 型を導入する.
unit は Unit 型の「唯一」の value
Unit 型が下半分に出現する導出規則が無いため具体的な term の形は不明
副作用は Unit で表現できる.
C や Java の void に近い. void というと Bot 型 (Bottom §15.4) に近いように聞こえるが実際には Unit 型に近い.

11.2.1 Exercise

t_n と t_{n+1} は定数分のサイズしか違わないのに t_{n+1} の簡約は t_n の 2 倍かかる, ような t_n.

$\mathsf{t}_0 = \lambda \mathsf{x}.\ \mathsf{x},\ \mathsf{t}_n = (\lambda \mathsf{x}.\ \mathsf{x}\ \mathsf{x})\ \mathsf{t}_{n-1}$

c.f. 高階関数クイズ

11.3 Derived Forms

sequence と wildcard を導入する.

Sequence

意味としては t_1 を評価してから, t_2 を評価する.
文法として組み込む方法は,

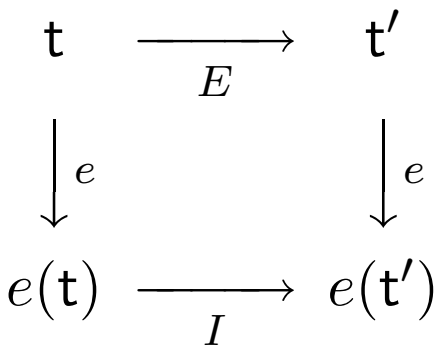
1. E-Seq, E-SeqNext, T-Seq を文法に追加する.
2. $t_1; t_2$ を $(\lambda x : \mathrm{Unit}.\ t_2)\ t_1$ の略記と見る. (derived form)

文法の追加の方は, E-AppAbs (P. 72) を良く見ると t_1 が評価されてから t_2 が評価される仕組みになっていることが分かる.

11.3.1 derived form であること

Theorem

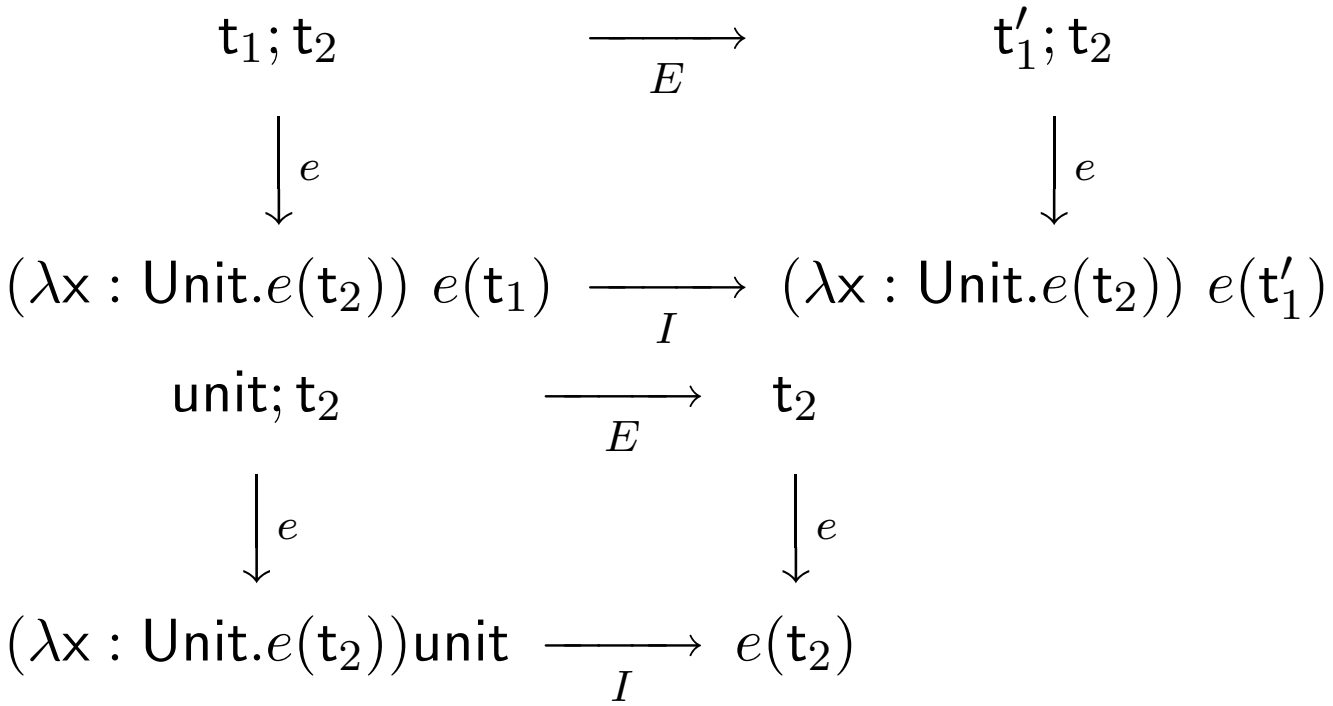
$t_1; t_2$ と $(\lambda x : \mathsf{Unit}.t_2)\ t_1$ はいつでも書き換えできる.
= 以下の可換図式が成立する.



E は E-Seq, E-SeqNext, T-Seq を追加した世界
I は追加してない世界

Proof.

e を再帰的に定義する. 話はそこからだ.
(evaluation)
$\Rightarrow$ $\mathsf{t} = \mathsf{t}_1; \mathsf{t}_2$ $(\mathsf{t}_1 \neq \mathsf{unit}), \mathsf{unit}; \mathsf{t}_2$ を考える
$\Leftarrow$ t の形について場合分け (実質 $\mathsf{t} = \mathsf{t}_1; \mathsf{t}_2$ だけ扱う)
(typing)
$\Rightarrow$ $t = \mathsf{t}_1; \mathsf{t}_2$ を考える
$\Leftarrow$ t の形について場合分け (実質 $\mathsf{t} = \mathsf{t}_1; \mathsf{t}_2$ だけ扱う) □


$\mathsf{t}_1;\mathsf{t}_2$
$\longrightarrow_E$
$\mathsf{t}_1';\mathsf{t}_2$
$\downarrow e$
$\downarrow e$
$(\lambda \mathsf{x}:\mathsf{Unit}.e(\mathsf{t}_2))\ e(\mathsf{t}_1)$
$\longrightarrow_I$
$(\lambda \mathsf{x}:\mathsf{Unit}.e(\mathsf{t}_2))\ e(\mathsf{t}_1')$
$\mathsf{unit};\mathsf{t}_2$
$\longrightarrow_E$
t_2
$\downarrow e$
$\downarrow e$
$(\lambda \mathsf{x}:\mathsf{Unit}.e(\mathsf{t}_2))\mathsf{unit}$
$\longrightarrow_I$
$e(\mathsf{t}_2)$

derived form について

言語には手を加えないで, 表面の文法を拡張すること.
型安全性とかは証明する必要有→ §11.3.1
Derived Form = Syntactic Sugar
逆は desugar (脱糖?)

Wildcard

wildcard _ を derived form として導入する.
$\lambda x : S.\ t$ を $\lambda \_ : S.\ t$ と略記する.

Exercise 11.3.2

追加すべき evaluation & typing rules

$$(\lambda\_ : \mathsf{T}_{11}.\mathsf{t}_{12})\ \mathsf{v}_2 \rightarrow \mathsf{t}_{12}$$

$$\frac{\Gamma \vdash \mathsf{t}_2 : \mathsf{T}_2}{\Gamma \vdash \lambda\_ : \mathsf{T}.\mathsf{t}_2 : \mathsf{T} \rightarrow \mathsf{T}_2}$$

(c.f. Figure 9-1, P.103)
derived form であること
→ §11.3.1 と一緒

11.4 Ascription

term の一部として型を明示

1. term の型を覚えておかなくても良い (特に文書中)
2. 複雑な term の型の表示 (型の別名も使用する)

抽象化としての ascription (ascription と cast の関係は §15.5 で再度議論する)

型の略記

UU = Unit → Unit とか
型と型の略記は, 書き手の都合で自由に入れ換えられる
こういう仕組みは言語によって有ったり (OCaml), 無かったり (Java)

Exercise 11.4.1

(1) t as T $\equiv (\lambda x : T\ x)\ t$
derived form であることは §11.3.1 と同様の方法で示す.
$e(t\ as\ T) = (\lambda x : T\ x)\ t$
(2) t_1 as T $\rightarrow t_1$

11.5 Let Bindings

複雑な式で, 部分式の繰り返しを避け読み易くする
ML 流の評価順序
let x=t_1 in t_2
t_1 を value になるまで評価してから β 簡約して t_2 を評価

derived form たり得るか?

let x=t_1 in $t_2 \equiv (\lambda x : T_1.t_2)\ t_1$
T_1 をどこかから調達しないといけない
→ 型付き導出木から調達

- Sequence, Wildcard, Ascription = term の変形
- Let Binding = 型付き導出木の変形 → “a little less derived”

Chap22 でも let を derived form としない理由を見る

11.6 Pairs

ここから Pair, Tuple, Record という直積集合を扱っていく.
Pair は $T_1 \times T_2$ という型を持つ.
直積 (direct product) やカーテシアン積 (cartesian product) と言う.
「直積 (product)」と「射影 (projection)」の 2 操作がある.
Pair の評価は左成分から, となっている. (本質的ではないルール.)

11.7 Tuples

直積成分を Pair よりも増やせるようにした型.
curly brace を使った型表記が特徴的.

$$\{\mathsf{T}_i^{\,i\in 1...n}\}$$

成分は左から評価.

11.8 Records

構造体, immutable dictionary, associative array など. index の代わりに label が出てくる.

Exercise 11.8.1

どこが explicit じゃないのか分からない……

$$\frac{\mathsf{l}_i = \mathsf{l}_j}{\{\mathsf{l}_i = \mathsf{v}_i{}^{i \in 1 \ldots n}\}.\mathsf{l}_j \longrightarrow \mathsf{v}_j}$$

Tuple is a special case of Record

$l_i = i$ という暗黙のラベルが付いている Record が Tuple
{a:Bool, Nat, c:Bool} も {a:Bool, 2:Nat, c:Bool} の略記と考えられる
しかし, ややこしいので止めましょう.

Record field の順序

多くの言語では Record の個々の要素の順番は無視して同値性を考える.

ex. Java の Map

> *boolean equals(Object o)*
> 指定されたオブジェクトがこのマップと等しいかどうかを比較します。指定されたオブジェクトもマップであり、*2* つの *Map* が同じマッピングを表している場合は *true* を返します。つまり、*t1.entrySet().equals(t2.entrySet())* である場合、*2* つのマップ *t1* と *t2* は同じマッピングを表します。これにより、*Map* インタフェースの実装が異なる場合でも、*equals* メソッドが正しく動作することが保証されます。

Record field の順序

しかしこの本では, 順序も考慮した同値性を選択.
Chap15 で subtyping を使って {a:1, b:2} と {a:2, b:1} を同一視する. そのパフォーマンスについては §15.6 で議論する.

Record Pattern Match

拡張された Let Binding
ex. destructuring-bind (Common Lisp), multiple-value-bind?? (elisp), multiple assignment (Python)
「同じ変数は 2 度現れない」という仮定のもと, M-Rcd では代入演算を合成している.
let a=c=x, d=y, b=z=a=c=1, d=2, b=3 in add x y z
M-Rcd のルールは, Tuple で組まれている木の, 右優先, 深さ優先, 帰り掛けでの適用, を意味する.

Exercise 11.8.2

1. Pattern に型を付けろ
2. ここまでの文法の type preservation と progress の証明の概形を書け.

11.9 Sums

型の和 (A+B で A 型または B 型を表す)
ex. Either (Haskell), 共通インターフェース (Java)
inl, inr でくるんで, T_1, T_2 型を $T_1 + T_2$ 型にする.

Uniqueness of Types

Type Sum を導入すると型の一意性が崩れてしまう.
1: Num のとき, inl 1: Num+Bool でもあり inl 1: Num+Num でもある.
解決策

- type checker の能力を上げる (Chap22)
- 継承関係を入れて解決 (Chap15)
- annotation を付けて明示 (←この章ではこれ)

11.10 Variants

Type Sum を任意個の型の和に拡張使い勝手とかは Sum と同じ
ex. union? (C), Variant? (VB)

Variants の応用

- Options
- Enumerations
- SingleField Variable: 型の目印としての名前