Type Systems and Programming Languages

前田俊行

平成 13 年 5 月 7 日

9 Simply Typed Lambda-Calculus

9.1 Function Types

この章では、pure lambda-calculus に boolean 値を付け加えた言語の上に型システムをつくる。目標は以下の 2 つである。

- type safety を満たすこと (type preservation と type progress を満たすこと)。
- 厳しすぎないこと (たいていの正しいプログラムを型づけできること)。

lambda-calculus には関数 (abstraction) があるので、それに型を付けたい。そこで、まず最初に最も安易な方法として、λ-abstraction に型 $\rightarrow$ をつけることにしてみる。

$$\lambda \mathsf{x}.\mathsf{t} :\rightarrow$$

ところが、これだとあまり嬉しくない。例えば、$\lambda \mathsf{x}.\mathsf{true}$ も $\lambda \mathsf{x}.\lambda \mathsf{y}.\mathsf{x}$ も同じ型 $\rightarrow$ となる。これは、前者が何か値を apply すると boolean 値 true を返すのに対し、後者は別の関数を返すという違いを無視してしまうことになる。またこの方法では、関数が正しく実行される (スタックしない) ためには、どのような引数を渡せばよいかを知ることができない。

そこで、関数がどんな型の値を受け取ってどんな型の値を返すのか、を型としてあらわすために、単なる $\rightarrow$ ではなく、$\mathsf{T}_1 \rightarrow \mathsf{T}_2$ のようにあらわすことにする。ここで、T_1 は関数の受け取る値の型、T_2 は関数の返す値の型、にそれぞれ対応する。

9.1.1 定義: Bool 型上の型の集合は以下の文法から生成される

$$\begin{array}{lll} \mathsf{T} & ::= & \textit{types:} \\ & \mathsf{T} \rightarrow \mathsf{T} & \textit{type of functions} \\ & \mathsf{Bool} & \textit{type of booleans} \end{array}$$

□

ここで *type constructor* $\rightarrow$ はright-associativeである。例えば$\mathsf{T}_1 \rightarrow \mathsf{T}_2 \rightarrow \mathsf{T}_3$ は$\mathsf{T}_1 \rightarrow (\mathsf{T}_2 \rightarrow \mathsf{T}_3)$ である。

9.2 The Typing Relation

次に、実際に lambda-calculus に型をつける、が、その前に、abstraction の引数の型をどのように決定するかを決める。

lambda-calculus には λx.t のような abstraction がある。このような abstraction に型をつけるには、abstraction に値を apply した後にどのような型の値を生成するのかを考えなければならない。そのためには結局、abstraction がどのような型の値を引数として apply できるかを決定しなければならない。

abstraction の引数の型を決める方法は 2 つ考えられる。

1. λ-abstraction に引数の型を *annotate* する
2. abstraction の本体を *analyze* して引数の型を推論する

ここでは、1 の方法をとることにする。そのため、単に λx.t と書くかわりに $\lambda\mathsf{x}:\mathsf{T}_1.\mathsf{t}_2$ のように書くことにする。ここで、 T_1 が引数 x の型をあらわす注釈 (annotation) である。(ちなみに 1 の方法を用いる言語を explictly typed language、2 の方法を用いる言語を implicitly typed language と呼ぶ。)

さて、abstraction の引数 (x) の型 (T_1) はもうこれでわかるので、後は abstraction の本体 (t_2) の型を決めればよい。そのためには、t_2 中の x の出現を、型 T_1 をもつ term であるとみなして型を決めればよい。このことは、以下のようなルールとしてあらわせる。

$$\frac{\mathsf{x}:\mathsf{T}_1 \vdash \mathsf{t}_2 : \mathsf{T}_2}{\vdash \lambda\mathsf{x}:\mathsf{T}_1.\mathsf{t}_2 : \mathsf{T}_1 \rightarrow \mathsf{T}_2}$$

型の関係 (relation) が、もはや term t と型 T の 2 項関係 t : T ではなく、3 項関係 $\Gamma \vdash \mathsf{t} : \mathsf{T}$ となる。ここで、Γ は t 中の自由変数の型づけに関する仮定の集合である。より正確には Γ は typing context といい、変数とその型のリストである。また、”,”を使って新たな binding を Γ の最後に付け足すことができる。また、混乱をさけるため、abstraction の引数 x は、Γ 中に存在しない変数名に置き換えることにする。

Pure simply typed lambda-calculus ($\lambda_\rightarrow$)

以下は pure lambda-calculus に型をつけた言語 pure simply typed lambda-calculus の定義である。ちなみに、ベース型が無いので実際にこの言語の term に型をつけることはできない。(Exercise 9.2.1)

Syntax

t	::=		*terms:*
		x	*variable*
		$\lambda\mathsf{x}{:}\mathsf{T}.\mathsf{t}$	*abstraction*
		$\mathsf{t}\ \mathsf{t}$	*application*
v	::=		*values:*
		$\lambda\mathsf{x}{:}\mathsf{T}.\mathsf{x}$	*abstraction value*
T	::=		*types:*
		$\mathsf{T}\to\mathsf{T}$	*type of functions*
Γ	::=		*contexts:*
		$\emptyset$	*empty contexts*
		$\Gamma,\ \mathsf{x}{:}\mathsf{T}$	*term variable binding*

Evaluation

$$\frac{\mathsf{t}_1 \longrightarrow \mathsf{t}_1'}{\mathsf{t}_1\mathsf{t}_2 \longrightarrow \mathsf{t}_1'\mathsf{t}_2} \quad \text{(E-App1)}$$

$$\frac{\mathsf{t}_2 \longrightarrow \mathsf{t}_2'}{\mathsf{v}_1\mathsf{t}_2 \longrightarrow \mathsf{v}_1\mathsf{t}_2'} \quad \text{(E-App2)}$$

$$(\lambda\mathsf{x}:\mathsf{T}_{11}.\mathsf{T}_{12})\,\mathsf{v}_2 \longrightarrow [\mathsf{x}\mapsto\mathsf{v}_2]\,\mathsf{t}_{12} \quad \text{(E-AppAbs)}$$

Typing

$$\frac{\mathsf{x}:\mathsf{T}\in\Gamma}{\Gamma\vdash\mathsf{x}:\mathsf{T}} \quad \text{(T-Var)}$$

$$\frac{\Gamma,\mathsf{x}:\mathsf{T}_1\vdash\mathsf{t}_2:\mathsf{T}_2}{\Gamma\vdash\lambda\mathsf{x}:\mathsf{T}_1.\mathsf{t}_2:\mathsf{T}_1\to\mathsf{T}_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma\vdash\mathsf{t}_1:\mathsf{T}_{11}\to\mathsf{T}_{12} \quad \Gamma\vdash\mathsf{t}_2:\mathsf{T}_{11}}{\Gamma\vdash\mathsf{t}_1\mathsf{t}_2:\mathsf{T}_{12}} \quad \text{(T-App)}$$

Booleans

これらは、Boolean 型をベース型に取るように pure simply typed lambda-calculus を拡張するルールである。

Syntax for Boolean

$$
\begin{array}{llr}
\mathsf{t} & ::= & \textit{terms:} \\
& \mathsf{true} & \textit{constant true} \\
& \mathsf{false} & \textit{constant false} \\
& \mathsf{if\ t\ then\ t\ else\ t} & \textit{conditional} \\
\\
\mathsf{v} & ::= & \textit{values:} \\
& \mathsf{true} & \textit{true value} \\
& \mathsf{false} & \textit{false value} \\
\\
\mathsf{T} & ::= & \textit{types:} \\
& \mathsf{Bool} & \textit{type of booleans}
\end{array}
$$

Evaluation for Boolean

$$\mathsf{if\ true\ then\ t_2\ else\ t_3} \longrightarrow \mathsf{t_2} \qquad \text{(E-IfTrue)}$$

$$\mathsf{if\ false\ then\ t_2\ else\ t_3} \longrightarrow \mathsf{t_3} \qquad \text{(E-IfFalse)}$$

$$\frac{\mathsf{t_1} \longrightarrow \mathsf{t_1'}}{\mathsf{if\ t_1\ then\ t_2\ else\ t_3} \longrightarrow \mathsf{if\ t_1'\ then\ t_2\ else\ t_3}} \qquad \text{(E-If)}$$

Typing for Boolean

$$\Gamma \vdash \mathsf{true} : \mathsf{Bool} \qquad \text{(T-True)}$$

$$\Gamma \vdash \mathsf{false} : \mathsf{Bool} \qquad \text{(T-False)}$$

$$\frac{\Gamma \vdash \mathsf{t_1} : \mathsf{Bool} \quad \Gamma \vdash \mathsf{t_2} : \mathsf{T} \quad \Gamma \vdash \mathsf{t_3} : \mathsf{T}}{\Gamma \vdash \mathsf{if\ t_1\ then\ t_2\ else\ t_3} : \mathsf{T}} \qquad \text{(T-If)}$$

9.3 Introduction and Elimination Forms

さて、上の $\lambda_{\rightarrow}$ の定義で登場した type constructor $\rightarrow$ は以下の 2 つの種類のルールで登場している。

1. introduction rule (T-Abs) どうやって型の要素がつくられるかを記述している
2. elimination rule (T-App) どうやって型の要素が使われるかを記述している

elimination rule の term の直接の subterm に introduction rule の term が来ていると、その term は redex となる。

これらのお話はこの本の後ろの方で複雑な型システムを議論するときにまた登場する。

9.4 Properties of Typing

次に pure simply typed lambda-calculus(with Booleans) の型安全性を証明する、が、その証明のためにまずいくつかの補題や定理を示す。

まずは、*inversion lemma* を示す。これは、導出がどのように構築されるかを述べるものである。

9.4.1 Lemma [Inversion of the typing relation]

1. If $\Gamma \vdash \mathsf{x} : \mathsf{R}$, then $\mathsf{x} : \mathsf{R} \in \Gamma$
2. If $\Gamma \vdash \lambda \mathsf{x} : \mathsf{T}_1.\mathsf{t}_2 : \mathsf{R}$, then $\mathsf{R} = \mathsf{T}_1 \to \mathsf{R}_2$ for some R_2 with $\Gamma, \mathsf{x} : \mathsf{T}_1 \vdash \mathsf{t}_2 : \mathsf{R}_2$
3. If $\Gamma \vdash \mathsf{t}_1\mathsf{t}_2 : \mathsf{R}$, then there is some type T_{11} such that $\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_{11} \to \mathsf{R}$ and $\Gamma \vdash \mathsf{t}_2 : \mathsf{T}_{11}$
4. If $\Gamma \vdash \mathsf{true} : \mathsf{R}$, then $\mathsf{R} = \mathsf{Bool}$
5. If $\Gamma \vdash \mathsf{false} : \mathsf{R}$, then $\mathsf{R} = \mathsf{Bool}$
6. If $\Gamma \vdash \mathsf{if t_1\ then t_2\ else\ t_3} : \mathsf{R}$, then
   - $\Gamma \vdash \mathsf{t}_1 : \mathsf{Bool}$
   - $\Gamma \vdash \mathsf{t}_2 : \mathsf{R}$
   - $\Gamma \vdash \mathsf{t}_3 : \mathsf{R}$

□

次に、uniqueness of types を示す。

9.4.3 Theorem [Uniqueness of Types]

ある typing context Γ で、ある term は、多くても 1 つだけ型をもつ。つまり term を型づけできるなら、その型は唯一のものである。さらにいえば、その型づけの導出も唯一のものである。 □

次に、cannonical forms lemma を示す。これは、value がある型を持つときに、具体的にどのようなかたちをとるかを示すものである。

9.4.4 Lemma [Cannonical Forms]

1. if v is a value of type Bool, then v is either trueor false
2. if v is a value of type $\mathsf{T}_1 \to \mathsf{T}_2$, then $\mathsf{v} = \lambda \mathsf{x} : \mathsf{T}_1.\mathsf{t}_2$

$\square$

ここで、type safety の 1 つ、progress theorem を示す。

9.4.5 Theorem [Progress]

もし閉じた term t が、ある T について $\vdash \mathsf{t} : \mathsf{T}$ であるならば、t は value であるか、または、$\mathsf{t} \longrightarrow \mathsf{t}'$ なる、t' が存在する。 $\square$

次に、2 つのちょっとした lemma を示す。これらの lemma は、その次の substitution lemma を示すために用いられる。

9.4.6 Lemma [Permutation]

もし $\Gamma \vdash \mathsf{t} : \mathsf{T}$ であり、$\triangle$ が Γ の要素をならびかえたものであるとすると、$\triangle \vdash \mathsf{t} : \mathsf{T}$ が成り立ち、さらにこれら 2 つの導出は同じ深さとなる。 $\square$

9.4.7 Lemma [Weakening]

もし $\Gamma \vdash \mathsf{t} : \mathsf{T}$ であり、かつ、$\mathsf{x} \notin dom(\Gamma)$ ならば、$\Gamma, \mathsf{x} : \mathsf{S} \vdash \mathsf{t} : \mathsf{T}$ が成り立ち、さらにこれら 2 つの導出は同じ深さとなる。 $\square$

以上の lemma より、以下の重要な lemma が証明できる。

9.4.8 Lemma [Substitution]

もし $\Gamma, \mathsf{x} : \mathsf{S} \vdash \mathsf{t} : \mathsf{T}$ かつ $\Gamma \vdash \mathsf{s} : \mathsf{S}$ が成り立つならば、$\Gamma \vdash [\mathsf{x} \mapsto \mathsf{s}]\mathsf{t} : \mathsf{T}$ が成り立つ。 $\square$

以上の準備により、ようやく type safety のもう 1 つの theorem、preservation を示すことができる。

9.4.9 Theorem [Preservation]

もし $\Gamma \vdash \mathsf{t} : \mathsf{T}$ かつ $\mathsf{t} \to \mathsf{t}'$ が成り立つならば、$\Gamma \vdash \mathsf{t}' : \mathsf{T}$ が成り立つ。 $\square$

これで、theorem progress と preservation より、$\lambda_{\to}$ の type safety が証明できた。

9.5 Erasure and Typability

さて、ここでよく $\lambda_{\rightarrow}$ の定義、特に evaluation rule をよく見直してみると、abstraction の引数に型を annotate するようにしたが、実は evaluation では直接的にはその annotation は何にも使われていないことがわかる。evaluation では単にその annotation を保持するだけである。

そこで、ほとんどの本格的なコンパイラでは、annotation は型チェックのときだけ用いて、実際にコードを生成するときは、annotation は無視して、untyped なかたちに変換する。このやり方を形式化するため、*eraseure* と呼ばれる、typed term を untyped term にマップする関数を使う。

以下はその関数 *erase* の定義である。

9.5.1 定義

$$
\begin{aligned}
erase\,(x) &= \mathsf{x} \\
erase\,(\lambda\mathsf{x}:\mathsf{T}_1.\mathsf{t}_2) &= \lambda\mathsf{x}.erase\,(\mathsf{t}_2) \\
erase\,(\mathsf{t}_1\mathsf{t}_2) &= erase\,(\mathsf{t}_1)\,erase\,(\mathsf{t}_2)
\end{aligned}
$$

□

当然のことながら、関数 *erase* を用いて untyped な言語にもどしてから評価する semantics と、もとの言語の semantics が一致して欲しい。このことは、以下の theorem で示される。

9.5.2 Theorem [Evaluation commutes with erasure]

1. もし typed evaluation のもとで、 $\mathsf{t} \longrightarrow \mathsf{t}'$ であれば、$erase\,(\mathsf{t}) \longrightarrow erase\,(\mathsf{t}')$ がなりたつ。
2. もし untyped evaluation のもとで、$erase\,(\mathsf{t}) \longrightarrow \mathsf{m}'$ であれば、$erase\,(\mathsf{t}') = \mathsf{m}'$ で $\mathsf{t} \longrightarrow \mathsf{t}'$ なる t' が存在する。

□

9.6 Curry vs. Church Semantics

Curry-style まず term を定義して、その term がどのように振舞うかをきめる semantics を定義する。最後に、好ましくない term をはじくために、型システムを構築する。

Church-style まず term を定義して、次に型づけできる term をよりぬいて、その term だけに semantics を定義する。