23 Universal Types

担当 遠藤 侑介*

2003 年 6 月 24 日

23 Universal Types

前章でおこなった let-polymorphism より一般的な計算体系 System F について学ぶ。

23.1 Motivation

§22.7 の例を考える。

```
doubleNat = λ f:Nat→Nat. λ x:Nat. f (f x);
doubleRcd = λ f:l:Bool→l:Bool. λ x:l:Bool. f (f x);
doubleFun = λ f:(Nat → Nat)→(Nat→Nat). λ x:Nat→Nat. f (f x);
```

これらの関数は、引数の型は異なるが動作は同じである。このように同じ機能を何回も記述することは「プログラムにおける機能の核は、ソースコードの一箇所にまとめて記述せよ。同じような関数を複数箇所に散在させるより、異なる部分を抽象化して 1 つにまとめた方がよい。」というソフトフェア工学の格言に反する。

この例において、異なる部分とは引数の型である。ということは、項から型を抽象化し、あとで具体的な型で instantiate できるような機能があればよいということになる。

23.2 Varieties of Polymorphism

コードの 1 部分が複数の型を持つことができるシステムを、多相型システムという。

- Parametric polymorphism: コードの 1 部分を「一般的に」型付けすることができる。具体的な型の代わりに型変数を使い、必要に応じて具体的な型で instantiate する。このインスタンスはすべて同じ動作をする (型によって動作を変えない) 。

  最も強力なのは本章の impredicative (first-class) polymorphism であるが、実際に使われているのは ML-style (let-polymorphism) である。

- Ad-hoc polymorphism: 多相型の値の、具体的な型ごとに応じて、異なる動作を記述する。最も使われているのは overloading であり、これを一般化したものが multi-method dispatch の基礎となっている。intensional polymorphism や typecase (Java の instanceof を一般化したもの) といった実行時に型情報を利用できる強力な機構もある。

*mame@is.s.u-tokyo.ac.jp

23.3 System F

System F は Girard が 1972 年に、Reynolds が 1974 年にそれぞれ独立に発見したシステムで、polymorphism の研究や、多くの言語設計の基礎として利用されてきた。また System F は second-order lambda-calculus とも呼ばれ、second-order intuitionistic logic と Curry-Howard correspondence を通して対応する。

System F の定義は型付きラムダ計算 $\lambda_{\to}$ の自然な拡張になっている。$\lambda_{\to}$ のラムダ抽象やラムダ適用と同じように、型の抽象化の構文 $\lambda \mathsf{X}.\mathsf{t}$ (型抽象) と、型の適用の構文 $\mathsf{t}[\mathsf{T}]$ (型適用) を導入する。また評価規則にも同様に

$$(\lambda \mathsf{X}.\mathsf{t}_{12})[\mathsf{T}_2] \longrightarrow [\mathsf{X} \mapsto \mathsf{T}_2]\mathsf{t}_{12} \text{ (E-TappTabs)}$$

を追加し、型付け規則にも以下を追加する。

$$\frac{\Gamma, \mathsf{X} \vdash \mathsf{t}_2 \ : \ \mathsf{T}_2}{\Gamma \vdash \lambda \mathsf{X}.\mathsf{t}_2 \ : \forall \mathsf{X}.\mathsf{T}_2} \text{ (T-Tabs)}$$

$$\frac{\Gamma \vdash \mathsf{t}_2 \ : \forall \mathsf{X}.\mathsf{T}_{12}}{\Gamma, \mathsf{X} \vdash \mathsf{t}_1[\mathsf{T}_2] \ : \ [\mathsf{X} \mapsto \mathsf{T}_2]\mathsf{T}_{12}} \text{ (T-App)}$$

追加された構文を利用すると次のような多相恒等関数が書ける。

$$\mathsf{id} = \lambda \mathsf{X}.\lambda \mathsf{x} : \mathsf{X}.\mathsf{x}$$

$\mathsf{id}[\mathsf{Nat}]$ は E-TappTabs より $\lambda x : \mathsf{Nat}.x$ となる。つまり関数 id は、引数として任意の型 T を与えると、型 $\mathsf{T} \to \mathsf{T}$ の関数を返す。よってこの関数の型は $\forall \mathsf{X}.\mathsf{X} \to \mathsf{X}$ となる。

System F の完全な定義は本文の最後に示す。

23.4 Examples

Warm-up

例:doubling 関数

$$\mathsf{double} = \lambda \mathsf{X}.\lambda \mathsf{f} : \mathsf{X} \to \mathsf{X}.\lambda \mathsf{a} : \mathsf{X}.f(fa) \ : \ \forall \mathsf{X}.(\mathsf{X} \to \mathsf{X}) \to \mathsf{X} \to \mathsf{X}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathsf{X}, f : \mathsf{X} \to \mathsf{X}, a : \mathsf{X} \vdash f \ : \ \mathsf{X} \to \mathsf{X} \quad \mathsf{X}, f : \mathsf{X} \to \mathsf{X}, a : \mathsf{X} \vdash a \ : \ \mathsf{X}}{\mathsf{X}, f : \mathsf{X} \to \mathsf{X}, a : \mathsf{X} \vdash fa \ : \ \mathsf{X}}}{\mathsf{X}, f : \mathsf{X} \to \mathsf{X}, a : \mathsf{X} \vdash f(fa) \ : \ \mathsf{X}}}{\mathsf{X}, f : \mathsf{X} \to \mathsf{X} \vdash \lambda a : \mathsf{X}.f(fa) \ : \ \mathsf{X} \to \mathsf{X}}}{\mathsf{X} \vdash \lambda f : \mathsf{X} \to \mathsf{X}.\lambda a : \mathsf{X}.f(fa) \ : \ (\mathsf{X} \to \mathsf{X}) \to \mathsf{X} \to \mathsf{X}}}{\vdash \lambda \mathsf{X}.\lambda f : \mathsf{X} \to \mathsf{X}.\lambda a : \mathsf{X}.f(fa) \ : \ \forall \mathsf{X}.(\mathsf{X} \to \mathsf{X}) \to \mathsf{X} \to \mathsf{X}}$$

$$\mathsf{doubleNat} = \mathsf{double}[\mathsf{Nat}]$$

などとできる。

例:selfApp

$$\mathsf{selfApp} = \lambda \mathsf{x} : \forall \mathsf{X}.\mathsf{X} \to \mathsf{X}.\mathsf{x}[\forall \mathsf{X}.\mathsf{X} \to \mathsf{X}]\mathsf{x} : (\forall \mathsf{X}.\mathsf{X} \to \mathsf{X}) \to (\forall \mathsf{X}.\mathsf{X} \to \mathsf{X})$$

$$\dfrac{\dfrac{\dfrac{\mathsf{x} : \forall \mathsf{X}.\mathsf{X} \to \mathsf{X} \vdash \mathsf{x} : \forall \mathsf{X}.\mathsf{X} \to \mathsf{X}}{\mathsf{x} : \forall \mathsf{X}.\mathsf{X} \to \mathsf{X} \vdash \mathsf{x}[\forall \mathsf{X}.\mathsf{X} \to \mathsf{X}] \ : \ (\forall \mathsf{X}.\mathsf{X} \to \mathsf{X}) \to (\forall \mathsf{X}.\mathsf{X} \to \mathsf{X})} \quad \mathsf{x} : \forall \mathsf{X}.\mathsf{X} \to \mathsf{X} \vdash \mathsf{x} : \forall \mathsf{X}.\mathsf{X} \to \mathsf{X}}{\mathsf{x} : \forall \mathsf{X}.\mathsf{X} \to \mathsf{X} \vdash \mathsf{x}[\forall \mathsf{X}.\mathsf{X} \to \mathsf{X}]\mathsf{x} \ : \ \forall \mathsf{X}.\mathsf{X} \to \mathsf{X}}}{\vdash \lambda \mathsf{x} : \forall \mathsf{X}.\mathsf{X} \to \mathsf{X}.\mathsf{x}[\forall \mathsf{X}.\mathsf{X} \to \mathsf{X}]\mathsf{x} \ : \ (\forall \mathsf{X}.\mathsf{X} \to \mathsf{X}) \to (\forall \mathsf{X}.\mathsf{X} \to \mathsf{X})}$$

Polymorphic Lists

型コンストラクタ List と以下のプリミティブ、Bool 、Nat 、fix などは組み込みとする。

$$\begin{aligned}
&\mathsf{nil} : \forall \mathsf{X}.\mathsf{ListX} \\
&\mathsf{cons} : \forall \mathsf{X}.\mathsf{X} \to \mathsf{ListX} \to \mathsf{ListX} \\
&\mathsf{isnil} : \forall \mathsf{X}.\mathsf{ListX} \to \mathsf{Bool} \\
&\mathsf{head} : \forall \mathsf{X}.\mathsf{ListX} \to \mathsf{X} \\
&\mathsf{tail} : \forall \mathsf{X}.\mathsf{ListX} \to \mathsf{ListX}
\end{aligned}$$

11 章で $\lambda_{\to}$ にリストを導入した時は、nil や cons のために特別な型付け規則が必要だったが、System F ではそのような規則を必要としない。13 章の Ref 型など、他の一般的なデータ構造や制御構造にも同じことが言える。

例:Nat のリスト

$$\mathsf{l} = \mathsf{cons}[\mathsf{Nat}]4(\mathsf{cons}[\mathsf{Nat}]3(\mathsf{cons}[\mathsf{Nat}]2(\mathsf{nil}[\mathsf{Nat}]))) : \mathsf{ListNat}$$

例:map

$$\begin{aligned}
&\mathsf{map} = \lambda \mathsf{X}.\lambda \mathsf{Y}.\lambda f : \mathsf{X} \to \mathsf{Y}. \\
&\ (\mathsf{fix}(\lambda \mathsf{m} : (\mathsf{ListX}) \to (\mathsf{ListY}). \\
&\quad \lambda \mathsf{l} : \mathsf{ListX}. \\
&\quad\ \mathtt{if}\ \mathsf{isnil}[\mathsf{X}]\mathsf{l} \\
&\quad\quad \mathtt{then}\ \mathsf{nil}[\mathsf{Y}] \\
&\quad\quad \mathtt{else}\ \mathsf{cons}[\mathsf{Y}](\mathsf{f}(\mathsf{head}[\mathsf{X}]\mathsf{l}))(\mathsf{m}(\mathsf{tail}[\mathsf{X}]\mathsf{l})) \\
&\ )) \\
&\mathsf{map} : \forall \mathsf{X}.\forall \mathsf{Y}.(\mathsf{X} \to \mathsf{Y}) \to \mathsf{ListX} \to \mathsf{ListY}
\end{aligned}$$

例:reverse

$$\begin{aligned}
&\mathsf{reverse} = \lambda \mathsf{X}.\lambda l : \mathsf{ListX}. \\
&\ \mathsf{fix}(\lambda \mathsf{f} : \mathsf{List\ X} \to \mathsf{List\ X}.\lambda \mathsf{l} : \mathsf{List\ X}. \\
&\quad\ \mathtt{if}\ \mathsf{isnil}\ [\mathsf{X}]\ \mathsf{l} \\
&\quad\quad \mathtt{then}\ \mathsf{nil}\ [\mathsf{X}] \\
&\quad\quad \mathtt{else}\ \mathsf{cons}\ [\mathsf{X}]\ (\mathsf{head}\ [\mathsf{X}]\ \mathsf{l})\ (\mathsf{f}\ (\mathsf{tail}\ [\mathsf{X}]\ \mathsf{l}))) \\
&\mathsf{reverse} : \forall \mathsf{X}.\mathsf{List\ X} \to \mathsf{List}\ X
\end{aligned}$$

Church Encodings

System F での真偽値や自然数の Church encoding について考える。System F をコアとするようなプログラミング言語を設計するときに、System F で表現できる機能は、コアを拡張するせずプリミティブとして追加できる。もちろん参照型のようにコアを書き換える必要がある機能も存在する。

5 章の型なしラムダ計算において、真偽値は

$$\begin{aligned}\mathsf{tru} &= \lambda \mathsf{t}.\lambda \mathsf{f}.\mathsf{t}\\ \mathsf{fls} &= \lambda \mathsf{t}.\lambda \mathsf{f}.\mathsf{f}\end{aligned}$$

と表現した。System F ではこれに型を付ける。tru と fls を同じ型にしたいので、引数 t と f の型は同じとするが、その型は任意であるとする。以上を満たすように

$$\begin{aligned}\mathsf{CBool} &= \forall \mathsf{X}.\mathsf{X} \to \mathsf{X} \to \mathsf{X}\\ \mathsf{tru} &= \lambda \mathsf{X}.\lambda \mathsf{t} : \mathsf{X}.\lambda \mathsf{f} : \mathsf{X}.\mathsf{t} \ : \ \mathsf{CBool}\\ \mathsf{fls} &= \lambda \mathsf{X}.\lambda \mathsf{t} : \mathsf{X}.\lambda \mathsf{f} : \mathsf{X}.\mathsf{f} \ : \ \mathsf{CBool}\end{aligned}$$

とできる。CBool に対する演算は

$$\begin{aligned}&\mathsf{not} = \lambda \mathsf{b} : \mathsf{CBool}.\lambda \mathsf{X}.\lambda \mathsf{t} : \mathsf{X}.\lambda \mathsf{f} : \mathsf{X}.\mathsf{b}[\mathsf{X}]\ \mathsf{f}\ \mathsf{t} \ : \ \mathsf{CBool} \to \mathsf{CBool}\\ &\mathsf{and} = \lambda \mathsf{b}_1 : \mathsf{CBool}.\lambda \mathsf{b}_2 : \mathsf{CBool}.\mathsf{b}_1[\mathsf{CBool}]\mathsf{b}_2\mathsf{fls} \ : \ \mathsf{CBool} \to \mathsf{CBool} \to \mathsf{CBool}\end{aligned}$$

などと書ける。

自然数も同様に型なしラムダ計算での表現に適当な型注釈と型抽象を付ける。

$$\begin{aligned}&\mathsf{CNat} = \forall \mathsf{X}.(\mathsf{X} \to \mathsf{X}) \to \mathsf{X} \to \mathsf{X}\\ &\mathsf{c}_0 = \lambda \mathsf{X}.\lambda \mathsf{s} : \mathsf{X} \to \mathsf{X}.\lambda \mathsf{z} : \mathsf{X}.\mathsf{z};\\ &\mathsf{c}_1 = \lambda \mathsf{X}.\lambda \mathsf{s} : \mathsf{X} \to \mathsf{X}.\lambda \mathsf{z} : \mathsf{X}.\mathsf{s}\ \mathsf{z};\\ &\mathsf{c}_2 = \lambda \mathsf{X}.\lambda \mathsf{s} : \mathsf{X} \to \mathsf{X}.\lambda \mathsf{z} : \mathsf{X}.\mathsf{s}\ (\mathsf{s}\ \mathsf{z});\\ &\quad\vdots\\ &\mathsf{c}_n = \lambda \mathsf{X}.\lambda \mathsf{s} : \mathsf{X} \to \mathsf{X}.\lambda \mathsf{z} : \mathsf{X}.\underbrace{\mathsf{s}(\mathsf{s}(\mathsf{s}\cdots(\mathsf{s}}_{n\ 個}\mathsf{z})\cdots));\\ &\mathsf{csucc} = \lambda \mathsf{n} : \mathsf{CNat}.\lambda \mathsf{X}.\lambda \mathsf{s} : \mathsf{X} \to \mathsf{X}.\lambda \mathsf{z} : \mathsf{X}.\mathsf{s}(\mathsf{n}[\mathsf{X}]\mathsf{sz});\ : \ \mathsf{CNat} \to \mathsf{CNat}\\ &\mathsf{cplus} = \lambda \mathsf{m} : \mathsf{CNat}.\lambda \mathsf{n} : \mathsf{CNat}.\mathsf{m}\ [\mathsf{CNat}]\ \mathsf{csucc}\ \mathsf{n} \ : \ \mathsf{CNat} \to \mathsf{CNat} \to \mathsf{CNat};\\ &\mathsf{iszero} = \lambda \mathsf{n} : \mathsf{CNat}.\mathsf{n}[\mathsf{CBool}](\lambda \mathsf{x} : \mathsf{CBool}.\mathsf{fls})\mathsf{tru} \ : \ \mathsf{CNat} \to \mathsf{CBool}\\ &\mathsf{ctimes} = \lambda \mathsf{m} : \mathsf{CNat}.\lambda \mathsf{n} : \mathsf{CNat}.\lambda \mathsf{X}.\lambda \mathsf{s} : \mathsf{X} \to \mathsf{X}.\mathsf{n}\ [\mathsf{X}]\ (\mathsf{m}\ [\mathsf{X}]\ [\mathsf{s}]) \ : \ \mathsf{CNat} \to \mathsf{CNat} \to \mathsf{CNat};\\ &\mathsf{cexp} = \lambda \mathsf{m} : \mathsf{CNat}.\lambda \mathsf{n} : \mathsf{CNat}.\lambda \mathsf{X}.\mathsf{n}\ [\mathsf{X} \to \mathsf{X}]\ (\mathsf{m}\ [\mathsf{X}]) \ : \ \mathsf{CNat} \to \mathsf{CNat} \to \mathsf{CNat};\\ &\mathsf{vpred} = \lambda \mathsf{n} : \mathsf{CNat}.\lambda \mathsf{X}.\lambda \mathsf{s} : \mathsf{X} \to \mathsf{X}.\lambda \mathsf{z} : \mathsf{X}.\\ &\quad \mathsf{n}\ [(\mathsf{X} \to \mathsf{X}) \to \mathsf{X}]\\ &\qquad (\lambda \mathsf{p} : (\mathsf{X} \to \mathsf{X}) \to \mathsf{X}.\lambda \mathsf{q} : \mathsf{X} \to \mathsf{X}.\mathsf{q}\ (\mathsf{p}\ \mathsf{s}))\\ &\qquad (\lambda \mathsf{x} : \mathsf{X} \to \mathsf{X}.\mathsf{z})\\ &\qquad (\lambda \mathsf{x} : \mathsf{X} \to \mathsf{x});\\ &\qquad\ : \ \mathsf{CNat} \to \mathsf{CNat} \to \mathsf{CNat}\end{aligned}$$

Encoding Lists

2 セクション前に登場した多相リストのプリミティブ群は、純粋な System F の体系だけで扱える。

自然数の表現を参考にリストの表現を考える。3 つの要素 x 、y 、z を持つようなリストは、2 引数関数 f と初期値 v を導入して f x (f y (f z v)) と表現することにする。要素の型が X のリストの型は次のように定義する。

$$\mathsf{ListX} = \forall \mathsf{R}.(\mathsf{X} \to \mathsf{R} \to \mathsf{R}) \to \mathsf{R} \to \mathsf{R};$$

nil と cons 、isnil は以下のように記述できる。

$$\mathsf{nil} = \lambda \mathsf{X}.(\lambda \mathsf{R}.\lambda \mathsf{c} : \mathsf{X} \to \mathsf{R} \to \mathsf{R}.\lambda \mathsf{n} : \mathsf{R}.\mathsf{n}) \text{ as } \mathsf{ListX} \ : \ \forall \mathsf{X}.\mathsf{List\ X}$$

$$\mathsf{cons} = \lambda \mathsf{X}.\lambda \mathsf{hd} : \mathsf{X}.\lambda \mathsf{tl} : \mathsf{ListX}.(\lambda \mathsf{R}.\lambda \mathsf{c} : \mathsf{X} \to \mathsf{R} \to \mathsf{R}.\lambda \mathsf{n} : \mathsf{R}.\mathsf{c\ hd\ (tl\ [R]\ c\ n)}) \text{ as } \mathsf{List\ X}$$
$$: \ \forall \mathsf{X}.\mathsf{X} \to \mathsf{List\ X} \to \mathsf{List\ X}$$

$$\mathsf{isnil} = \forall \mathsf{X}.\lambda \mathsf{l} : \mathsf{List\ X}.\mathsf{l}[\mathsf{Bool}](\lambda \mathsf{hd} : \mathsf{X}.\lambda \mathsf{tl} : \mathsf{Bool}.\mathsf{false})\mathsf{true} \ : \ \forall \mathsf{X}.\mathsf{List\ X} \to \mathsf{Bool}$$

head の定義は少し難しい。リスト nil に対して head を適用した場合どうするべきか。ここではその場合には fix を用いて評価を発散させることにする。

$$\mathsf{diverge} = \lambda \mathsf{X}.\lambda \_ : \mathsf{Unix.fix}(\lambda \mathsf{x} : \mathsf{X}.\mathsf{x}) : \forall \mathsf{X}.\mathsf{Unit} \to \mathsf{X}$$

これを利用して

$$\mathsf{head} = \lambda \mathsf{X}.\lambda \mathsf{l} : \mathsf{List\ X}.\mathsf{l}[\mathsf{X}](\lambda \mathsf{hd} : \mathsf{X}.\lambda \mathsf{tl} : \mathsf{X}.\mathsf{hd})(\mathsf{diverge}[\mathsf{X}]\texttt{unit}) \ : \ \forall \mathsf{X}.\mathsf{List\ X} \to \mathsf{X}$$

と書ける。ただしこれでは call-by-value のとき常に発散してしまう。

$$\mathsf{head} = \lambda \mathsf{X}.\lambda \mathsf{l} : \mathsf{List\ X}.(\mathsf{l}[\mathsf{Unit} \to \mathsf{X}](\lambda \mathsf{hd} : \mathsf{X}.\lambda \mathsf{tl} : \mathsf{Unit} \to \mathsf{X}.\lambda \_ : \mathsf{Unit}.\mathsf{hd})(\mathsf{diverge}[\mathsf{X}]))\texttt{unit}$$
$$: \ \forall \mathsf{X}.\mathsf{List\ X} \to \mathsf{X}$$

と書けば call-by-value でも正しく動作する。

tail の定義はさらに難しいので略。

Exercise の insert をやってみましたが動作させて見たことがないので間違っているかもしれません……。

```
SubListX = Bool → X → ListX
insert =
  λX.λcmp : X → X → Bool.λl : ListX.λe : X.
    l [SubListX]
      (λe2 : X.λl2 : Bool → X → List X.
        cons [SubList X] e2
          (if cmp e e2
            then (λf : Bool.λe3 : X.l2 f e3)
            else (λf : Bool.λe3 : X.
              if f
              then (cons [SubList X] e3 (p false e3))
              else p false e3))
      (λf : Bool.λe3 : X.
        if f then (cons [SubList X] e3 (nil [SubList X])) else (nil [SubList X]))
      false e
```

23.5 Basic Properties

System F の基本的な性質は 9 章の型付きラムダ計算とよく似ている。特に、preservation 定理と progress 定理は型付きラムダ計算の場合を自然に拡張したものである。

Theorem 23.5.1. *[PRESERVATION]* もし $\Gamma \vdash t : T$ かつ $t \longrightarrow t'$ が成り立つならば、$\Gamma \vdash t' : T$ が成り立つ。

Proof. 9 章の PRESERVATION の証明に型抽象と型適用の場合を追加する。

$\Gamma \vdash t : T$ の導出に関する帰納法。型抽象と型適用の場合だけ示す。

$t = \lambda X.t_2$ のとき、t は値であるから、t と異なる t' に対して $t \to t'$ となる評価は存在しない。よって定理の前提が偽になるため定理を満たしている。

$t = t_1[T_2]$ のとき、$\Gamma \vdash t_1 : \forall X.T'$ ただし T' は $T = [X \to T_2]T'$ を満たし $X \notin \Gamma$ 。このとき、評価規則の適用によって 2 つの場合にわけられる。

E-TAPP の場合 $t_1 \to t'_1$ かつ、$t' = t'_1[T_2]$ 。帰納法の仮定より $\Gamma \vdash t'_1 : \forall X.T'$ 。よって T-APP より $\Gamma \vdash t'_1[T_2]$ 。

E-TAPPTABS の場合 $t_1 = \lambda X.t_{12}$ 。従って $\Gamma, X \vdash t_{12} : T'$ 。型代入における型保存性 (22 章)

$$\Gamma \vdash t : T \text{ ならば} \sigma\Gamma \vdash \sigma t : \sigma T$$

より、

$$[X \mapsto T_2](\Gamma, X) \vdash [X \mapsto T_2]t_{12} : [X \mapsto T_2]\ T'$$

となる。$X \notin \Gamma$ より $[X \mapsto T_2]\Gamma = \Gamma$ であり、X への型適用だから環境から X は消える。つまり $[X \mapsto T_2](\Gamma, X) = \Gamma$。また $T = [X \to T_2]T'$ より

$$\Gamma \vdash [X \mapsto T_2]t_{12} : T$$

であり、これにより E-TAPPABS によって型保存性があることが示された。 □

Theorem 23.5.2. *[PROGRESS]* もし t が *closed* かつ *well-typed* な項ならば、t は値、もしくは、ある t' が存在して $t \longrightarrow t'$ が成り立つ。

Proof. 9 章の PROGRESS の証明に型抽象と型適用の場合を追加する。

項の構造に関する帰納法。型抽象と型適用の場合だけ示す。

$t = \lambda X.t_2$ のとき、t は値である。

$t = t_1[T_2]$ のとき、帰納法の仮定より t_1 は値であるか他の項に関する簡約が存在する。前者の場合 E-TappTabs が、後者の場合 E-Tapp が適用できる。つまり常に他の項に関する簡約が存在する。 □

Theorem 23.5.3. *[NORMALIZATION] Well-typed な System F* の項は正規性を持つ。

つまり well-typed な System F のプログラムの評価は必ず停止する。証明はとても難しいらしく本文にも載っていなかったので略。

23.6 Erasure, Typability, and Type Reconstruction

System F の項を形なしラムダ計算の項に変換する型消去関数を定義する。

$$
\begin{array}{lcl}
erase(\mathsf{x}) & = & \mathsf{x} \\
erase(\lambda \mathsf{x} : \mathsf{T}_1.\mathsf{t}_2) & = & \lambda \mathsf{x}.erase(\mathsf{t}_2) \\
erase(\mathsf{t}_1\ \mathsf{t}_2) & = & erase(\mathsf{t}_1)\ erase(\mathsf{t}_2) \\
erase(\lambda \mathsf{X}.\mathsf{t}_2) & = & erase(\mathsf{t}_2) \\
erase(\mathsf{t}_1[\mathsf{T}_2]) & = & erase(\mathsf{t}_1)
\end{array}
$$

型なしラムダ計算の項 m に対して、$erase(\mathsf{t}) = \mathsf{m}$ となるような System F の項 t があるとき、m は System F で型付け可能であるという。型再構成は、型なしラムダ計算の項 m が与えられたとき、$erase(\mathsf{t}) = \mathsf{m}$ を満たすような well-typed な System F の項 t を見つけられるかを求めることになる。これは 1970 年代の初期から研究されていたが、1990 年代に Wells によって否定的に決着した。

THEOREM 23.6.1. *[WELLS, 1994]* 与えられた型なしラムダ計算の項 m に対して、$erase(t) = m$ を満たすような *well-typed* な *System F* の項 t が存在するかどうかは、決定不能である。

System F においては、完全な型再構成だけでなく、さまざまな種類の部分的な型再構成も決定不能だと知られている。たとえば次のような部分型消去関数に対しても、型再構成は決定不能である。

$$
\begin{array}{lcl}
erase_p(\mathsf{x}) & = & \mathsf{x} \\
erase_p(\lambda \mathsf{x} : \mathsf{T}_1.\mathsf{t}_2) & = & \lambda \mathsf{x}.erase(\mathsf{t}_2) \\
erase_p(\mathsf{t}_1\ \mathsf{t}_2) & = & erase(\mathsf{t}_1)\ erase(\mathsf{t}_2) \\
erase_p(\lambda \mathsf{X}.\mathsf{t}_2) & = & \lambda \mathsf{X}.erase(\mathsf{t}_2) \\
erase_p(\mathsf{t}_1[\mathsf{T}_2]) & = & erase(\mathsf{t}_1)\ [\,]\ \ (*\text{型適用のマーク } [\,] \text{ が付いている}*)
\end{array}
$$

THEOREM 23.6.2. *[BOEHM, 1985, 1989]* 型適用の印が付いているような閉じた項 m に対して、$erase_p(t) = m$ を満たすような *well-typed* な *System F* の項 t が存在するかどうかは、決定不能である。

有用な部分型再構成もあるらしい。

BOEHM は $erase_p$ の型構成を higher-order unification (これも決定不能) に帰着して決定不能を証明したので、higher-order uification の semi-algorithm を利用した部分型再構成がある。

20 章の recursive types で出てきた、datatype のコンストラクタやデストラクタを型注釈と見なして fold や unfold を入れる発想を、$\forall$ まで含めるように拡張する方法？や、ML 式の型再構成に、明示的に ($\forall$ を含んだ) 型注釈を付けられるようにする方法、subtyping と impredicative を共に含んだ polymorphism 、greedy type inference などなど……よくわかりませんでした。

23.7 Erasure and Evaluation Order

System F の操作的意味論は、type-passing semantics で、多相関数に型引数を渡すと型は関数本体に代入される。

System F に基づくプログラミング言語の現実的なインタプリタやコンパイラで、実行時に型を操作すると大きなコストがかかってしまう。型は実行に影響を与えないので、実行時に型注釈は何の意味も持たない。一旦 well-typed なプログラムを得れば、型注釈を好きなように書き換えて、全く同じ動作をする別のプログラムを得ることができる。多くの多相言語では、type-erasure semantics を採用している。これは型検査の後に型付きプログラムを形なしプログラムに変換し評価を行う。

しかし reference や例外など副作用を含む本格的なプログラム言語では、型消去関数の定義が多少慎重に定義しなければならない。例えば System F に例外を発生させるプリミティブ error を追加したプログラムを考えると

```
let f = (λ X. error) in 0;
```

これを評価すると 0 になる。この項から単純に型を消去すると

```
let f = error in 0;
```

これを評価すると例外が発生してしまう。call-by-value 評価戦略では、値 (ここでは型抽象) の中の項 (副作用を含みうる。ここでは error) の評価を延期させる。この言語では型抽象が重要な意味を持っていたということになる。次のように型消去関数を定義することで、このような間違いは発生しなくなる。

$$
\begin{array}{lcl}
erase_v(\mathsf{x}) & = & \mathsf{x} \\
erase_v(\lambda \mathsf{x}:\mathsf{T}_1.\mathsf{t}_2) & = & \lambda \mathsf{x}.erase(\mathsf{t}_2) \\
erase_v(\mathsf{t}_1\ \mathsf{t}_2) & = & erase(\mathsf{t}_1)\ erase(\mathsf{t}_2) \\
erase_v(\lambda \mathsf{X}.\mathsf{t}_2) & = & \lambda\_\ .\ erase(\mathsf{t}_2) \\
erase_v(\mathsf{t}_1[\mathsf{T}_2]) & = & erase(\mathsf{t}_1)\ \mathsf{dummyv}\quad (*\text{この値はなんでもいい}*)
\end{array}
$$

THEOREM 23.7.1. $erase_v(t) = u$ が成り立つならば、*(1)* t と u は正規形であるか、*(2)* $erase_v(t') = u'$ であるような $t \longrightarrow t'$ と $u \longrightarrow u'$ が存在する。

23.8 Fragments of System F

System F の正確さと表現力は、多相性の研究に重要な役割を果たしてるが、言語設計において完全に System F を用いることはコストがかかるので滅多にない。そこで System F を制限して型再構成が実用になるような派生を考える。例えば prenex polymorphism (ML の多相性) 、rank-2 polymorphism などがある。

prenex polymorphism では型変数は quantifier を持たない型しかとりえず、quantifier を持つ型が関数の引数に現れることを許さない。

型を根からたどって ∀ quantifier に出会うまでの経路で、関数適用の矢印が 2 つ以上現れる経路がないとき、その型は rank-2 であると言う。例えば $(\forall \mathsf{X}.\mathsf{X} \to \mathsf{X}) \to \mathsf{Nat}$ や、$\mathsf{Nat} \to \mathsf{Nat}$ 、$\mathsf{Nat} \to (\forall \mathsf{X}.\mathsf{X} \to \mathsf{X}) \to \mathsf{Nat} \to \mathsf{Nat}$ などは rank-2 であるが、$((\forall \mathsf{X}.\mathsf{X} \to \mathsf{X}) \to Nat) \to Nat$ は rank-2 ではない。rank-2 polymorphism では、型はすべて rank-2 に制限される。rank-2 polymorphism はより多くの形なしラムダ計算の項を型付けできるという意味で prenex polymorphism より強力である。

rank-2 polymorphism は型付けが決定可能で、rank-3 polymorphism 以降は型付けが決定不能である。

23.9 Parametricity

CBool について考える。

$$\mathsf{CBool} = \forall \mathsf{X}.\mathsf{X} \to \mathsf{X} \to \mathsf{X}$$
$$\mathsf{tru} = \lambda \mathsf{X}.\lambda \mathsf{t} : \mathsf{X}.\lambda \mathsf{f} : \mathsf{X}.\mathsf{t} \ : \ \mathsf{CBool}$$
$$\mathsf{fls} = \lambda \mathsf{X}.\lambda \mathsf{t} : \mathsf{X}.\lambda \mathsf{f} : \mathsf{X}.\mathsf{f} \ : \ \mathsf{CBool}$$

CBool が与えられたとき、tru と fls の定義は、CBool の構造から機械的に構成することができる。CBool は ∀ で始まっているのでその値は λX で始まらなければならない。CBool の本体は $\mathsf{X} \to \mathsf{X} \to \mathsf{X}$ なので、CBool の値の本体も λt : X.f : X. で始まることになる。最終的に CBool は t を返すので、CBool の値は t か f を返す。というのは、この 2 つの値以外に X の型の値を手に入れたり生成したりできないからである。

これは多相プログラムの一様な動作を定式化する parametricity の強力な原理による現象である。

23.10 Impredicativity

System F の多相性は impredicative であると言われる。一般に、quantifier の定義域に、quantifier が含まれているとき、その定義を impredicative であると言う。一方 ML の多相性は predicative であると言われる。型変数に quantifier が含まれないからである。

predicative と impredicative の歴史については略。

23.11 Notes

略。