Polymorphism and System F

米澤研 M1 田渕 直 (tabee@yl.is.s.u-tokyo.ac.jp)

22 Type Reconstruction (後半)

22.7 Let-Polymorphism

Polymorphism という言葉は、同じプログラムテキストを文脈に応じて違った使い方ができるプログラミング言語の機能を指す。

前回に見た type reconstruction のアルゴリズムは let-polymorphism というシンプルな形式の polymorphism に一般化できる。let-polymorphism は Milner (1978) によりオリジナルの ML に最初に導入された (アイデアそのものはもう少し遡る)。

Polymorphism の何が嬉しいか、ということについては、改めて説明するまでもないと思われるが、一応確認しておくと、

$$\begin{array}{l}\texttt{let doubleNat} \;=\; \lambda \texttt{f} : \texttt{Nat} \to \texttt{Nat}.\lambda \texttt{a} : \texttt{Nat}.\ \texttt{f(f(a))}\ \texttt{in}\\ \texttt{let doubleBool} \;=\; \lambda \texttt{f} : \texttt{Nat} \to \texttt{Nat}.\lambda \texttt{a} : \texttt{Bool}.\ \texttt{f(f(a))}\ \texttt{in}\\ \mathit{let-body}\end{array}$$

というようなプログラムを考えたときに、[1]`doubleNat` と `doubleBool` は型の annotation 以外は全く同じなのだから例えば、

```
let double = λf : X → X.λa : X. f(f(a)) in
let a = double (λx : Nat. succ(succx)) 1 in
let b = double (λx : Bool. if x then false else true) false in
...
```

のようにひとまとめに書いてしまって、`a = ...` の部分では `double:(Nat→Nat)→Nat→Nat` に、`b = ...` の部分では `double:(Bool→Bool)→Bool→Bool` に型付けしてもらえると二度手間が省ける、ということである。

ちなみに、今までの規則では上のように書いても型付けはできない。`let` の型付け規則は

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}_1 \qquad \Gamma, \texttt{x} : \texttt{T}_1 \vdash \texttt{t}_2 : \texttt{T}_2}{\Gamma \vdash \texttt{let x} = \texttt{t}_1 \ \texttt{in}\ \texttt{t}_2 : \texttt{T}_2}\ (\texttt{T} - \texttt{LET})$$

[1] 本節では $\lambda_{\to}$ + let-binding な言語で話を進める

だったので (11 章)、結局 double の型を唯 1 つの $\mathtt{T}_1$ に固定しなければならないからである。

そのため、規則 T-LET を

$$\frac{\Gamma \vdash [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{let\ x} = \mathtt{t}_1 \mathtt{\ in\ } \mathtt{t}_2 : \mathtt{T}_2} \ (\mathtt{T-LETPOLY})$$

で置き換え、type reconstruction の為に同様の constraint-typing rule

$$\frac{\Gamma \vdash [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{t}_2 : \mathtt{T}_2 \mid\!\chi\ C}{\Gamma \vdash \mathtt{let\ x} = \mathtt{t}_1 \mathtt{\ in\ } \mathtt{t}_2 : \mathtt{T}_2 \mid\!\chi\ C} \ (\mathtt{CT-LETPOLY})$$

を導入する。

以上の変更は本質的には

$$\mathtt{let\ x} = \mathtt{v}_1 \mathtt{\ in\ } \mathtt{t}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_1]\mathtt{t}_2$$

という evaluation を型計算の前に行ってしまうことに相当する。

上の変更だけではまだ不十分で、このままでは double で annotate された型変数 X に対して { X = Nat, X = Bool } のような矛盾した constraint が出来てしまう。これを防ぐには、前回導入した implicit type annotation を使う。すなわち、

```
let double = λf : .λa. f(f(a)) in
let a = double (λx : Nat. succ(succx)) 1 in
let b = double (λx : Bool. if x then false else true) false in
...
```

として、各 double の derivation 毎に (CT-ABSINF) で新しい型変数を導入するようにしてやればよい。

以上、2 点の変更で一応 double の例は typable になったが、まだ問題が残っている。

$$\mathtt{let\ x} = (\mathtt{stuck}\ \text{する}\ \mathtt{term})\ \mathtt{in}\ 0$$

というプログラムは新しい規則の元では typable なのに evaluate できないことになってしまう。let-binding の右辺は値になるまで先に評価する、という評価規則なので、型システムでもこの部分の正当性をチェックするべきである。というわけで、最終的な規則の形は

$$\frac{\Gamma \vdash [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{t}_2 : \mathtt{T}_2 \quad \Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \vdash \mathtt{let\ x} = \mathtt{t}_1 \mathtt{\ in\ } \mathtt{t}_2 : \mathtt{T}_2} \ (\mathtt{T-LETPOLY})$$

となる。(CT-LETPOLY) も同様。

これまでで一通り polymorphism を実現するための枠組みは整ったが、この枠組みのままでは実用にならない。問題は $\mathtt{t}_2$ の中に複数の let-束縛変数が出現する場合で、今の規則を素直に使うと、最悪の場合型チェックにかかる時間は項の長さに対して指数的に増えてしまう。そこで、実際のプログラミング言語では以下のように型チェックを行う。

1. $\mathtt{t}_1$ に constraint typing を使って $\mathtt{S}_1$ (型) と $\mathtt{C}_1$ (constraint) を求める。
2. $\mathtt{C}_1$ から most general solution σ と $\mathtt{t}_1$ の主型 $\mathtt{T}_1$ を求める。
3. $\mathtt{T}_1$ に残っている型変数のうち、文脈 Γ に現れないものを $\mathtt{X}_1 \ldots \mathtt{X}_n$ とする。$\mathtt{t}_1$ の「(主) 型スキーム」を $\forall \mathtt{X}_1 \ldots \mathtt{X}_n.\mathtt{T}_1$ と書く。Γ の中にある型変数は $\mathtt{t}_1$ の「外」と関連を持っているので $\forall$ で限量してはいけない。これをすると

$$(\lambda \mathtt{f} : \mathtt{X} \to \mathtt{X}.\lambda \mathtt{x} : \mathtt{X}.\ \mathtt{let\ g} = \mathtt{f\ in\ g}(0))$$
$$(\lambda \mathtt{x} : \mathtt{Bool}.\ \mathtt{if\ \ x\ then\ true\ else\ false})$$
$$\mathtt{true};$$

   が型付けできてしまう。
4. 変数と型スキームのペア $\mathtt{x} : \forall \mathtt{X}_1 \ldots \mathtt{X}_n.\mathtt{T}_1$ を文脈に追加し、$\mathtt{t}_2$ を型チェックする。
5. $\mathtt{t}_2$ の中で $\mathtt{x}$ が現れる毎に fresh な型変数 $\mathtt{Y}_1 \ldots \mathtt{Y}_n$ を導入して、$[\mathtt{X}_1 \mapsto \mathtt{Y}_1 \ldots \mathtt{X}_n \mapsto \mathtt{Y}_n]\mathtt{T}_1$ をその場での $\mathtt{x}$ の型とする (これによって、上で出てきた implicit type annotation を使う、という約束は必ずしも必要でなくなる)。

推論規則の形で書けば、

$$\frac{\mathtt{x} : \forall \mathtt{X}_1 \ldots \mathtt{X}_n.\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}[\mathtt{X}_1 \mapsto \mathtt{Y}_1 \ldots \mathtt{X}_n \mapsto \mathtt{Y}_n]\ \mathtt{where\ each\ Y_i : fresh}}\ (\mathtt{Alt.\ T-VAR})$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \qquad \Gamma, \mathtt{x} : \forall \mathtt{X}_1 \ldots \mathtt{X}_n.\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{let\ x} = \mathtt{t}_1\ \mathtt{in\ t}_2 : \mathtt{T}_2}\ (\mathtt{Alt.\ T-LETPOLY})$$

という感じになる。たぶん...。

このアルゴリズムは非常に効率が良く、項長に関して本質的に線形オーダでしかないと長い間思われていたが、最悪のケースでは依然として指数オーダとなることが Kfoury et al. と Mairson によって示された。

最後に、let-polymorphism 付きの言語に、参照等副作用を伴う機能を加える時には注意がいる。例えば

$$\mathtt{let\ r\ =\ ref}(\lambda \mathtt{x.x})\mathtt{in}$$
$$(\mathtt{r\ :=\ }(\lambda \mathtt{x} : \mathtt{Nat.succ\ x});$$
$$(!\mathtt{r})\ \mathtt{true};$$

という式を考えてみる。最終的に

$$\mathtt{r} : \forall \mathtt{X.Ref}(\mathtt{X} \to \mathtt{X})$$

が文脈に加わり、2 行目で

$$\mathtt{Ref}(\mathtt{Nat} \to \mathtt{Nat})$$

、3 行目で

$$\mathtt{Ref}(\mathtt{Bool} \to \mathtt{Bool})$$

となるが、これは明らかにおかしい。

こういうことが起こるのは、評価規則が call-by-value なのに型付けの方ではいきなり

$$\Gamma \vdash [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{t}_2 : \mathtt{T}_2$$

としてしまっているのが原因なので、評価規則か型付け規則のどちらかを変更すればよい。評価規則を

$$\mathtt{let\ x} = \mathtt{t}_1 \ \mathtt{in}\ \mathtt{t}_2 \ \longrightarrow [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{t}_2$$

に変えると上の例は

$$(\mathtt{ref}\ (\lambda \mathtt{x}.\mathtt{x})) \ := \ (\lambda \mathtt{x} : \mathtt{Nat}.\mathtt{succ}\ \mathtt{x});$$
$$(!(\mathtt{ref}\ (\lambda \mathtt{x}.\mathtt{x})))\ \mathtt{true};$$

という正当な式に評価されるが、いきなり call-by-value でない semantics を導入したため理解しにくいという欠点がある。

結論としては型付けの方に変更を加える方が望ましい。具体的には let-束縛される項が値の時にのみ $\forall$ による一般化を許す、という制限 (value restriction) を加える。強すぎる制限に見えるかもしれないが、実用上はほとんど問題にならないことが経験的に知られている (Wright 1995)。

22.8 Notes

λ 計算の主型についての記述は 50 年代の Curry の論文などに既に見られる。主型の具体的な計算法は Hindley(1969)・Morris(1968)・Milner(1978) 等。

Milner のオリジナルの ML-style let-polymorphism で使われている type reconstruction のアルゴリズム (Algorithm W) は、完全に untyped な項に対して定義されているという点で本章で扱ったものとは多少異なっている。

その他、let-polymorphism に対する様々な拡張 (Record, Variant, etc...) についての参考文献がテキストに記載されているので、適宜、参照して頂きたい。

23 Universal Types

23.2 Varieties of Polymorphism

- Parametric Polymorphism: 前節と本節のメイン。型チェックではとりあえず「generic な」型を当てておいて、必要に応じて具体的な型で考えるタイプ。Polymorphic なコードはプログラムテキスト上は同じ見かけになる。

  最も強力なのは本節の impredicative (or first-class) polymorphism だが、実用上広く使われるのは let-polymorphism である。

- Ad-hoc Polymorphism: 型ごとに違ったコードを書くタイプ。最も広く使われているものに overloading、それを一般化した multi-method dispatch 等がある。intensional polymorphis や `typecase` のように、実行時に型情報を利用できる強力な機構もある。

  OOP 界で polymorphism というときには ad-hoc polymorphism の一種である subtype polymorphism を指すことが多い。

23.3 System F

本節で紹介する System F は Reynolds (1974) と Girard (1972) によって独立に発見された体系で、研究・言語デザインの基礎として広く利用されている。

System F の定義は $\lambda_{\to}$ の自然な拡張になっている[2]。具体的には型に対しての「λ-抽象・適用」を導入して、

$$(\lambda \mathtt{X}.\mathtt{t})[\mathtt{T}] \longrightarrow [\mathtt{X} \mapsto \mathtt{T}]\mathtt{t} \qquad (\mathtt{E}-\mathtt{TAPPTABS})$$

という評価規則を設ける。

型システムの方ではこれを型付けできるよう、下の規則を加える。

$$\frac{\Gamma \vdash \lambda \mathtt{X}.\mathtt{t} : \forall \mathtt{X}.\mathtt{T}}{\Gamma, \mathtt{X} \vdash \mathtt{t} : \mathtt{T}} \qquad (\mathtt{T}-\mathtt{TABS})$$

$$\frac{\Gamma \vdash \mathtt{t}[\mathtt{T}_2] : [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}_1}{\Gamma \vdash \mathtt{t} : \forall \mathtt{X}.\mathtt{T}_1} \qquad (\mathtt{T}-\mathtt{TAPP})$$

変数と型変数の freshness (Γ に現れない) は相変わらず仮定する。Freshness を満たすための型変数の α-rename も許すものとする。

System F の完全な定義は 図 23-1 に示した通り。

23.4 Examples

この節で System F による polymorphic プログラミングの例を見てみる。まず単純な例から始め、リストや自然数の Church encoding が System F で表現できることを見る。

- 例 1. `id` (identity function)

$$\mathtt{id} \;=\; \lambda \mathtt{X}.\lambda \mathtt{x}.\ \mathtt{x} : (\forall \mathtt{X}.\mathtt{X} \to \mathtt{X})$$

$$\dfrac{\dfrac{\mathtt{X}, \mathtt{x} : \mathtt{X} \vdash \mathtt{x} : \mathtt{X}}{\mathtt{X} \vdash \lambda \mathtt{x}.\ \mathtt{x} : (\mathtt{X} \to \mathtt{X})}}{\vdash \lambda \mathtt{X}.\lambda \mathtt{x}.\ \mathtt{x} : (\forall \mathtt{X}.\mathtt{X} \to \mathtt{X})}$$

[2] ので、とりあえず `let` は扱わない。

もう少し面白い例として、自己適用は System F では型付けできる。

- 例 2. `selfApp` (self application)

$$\texttt{selfApp} \;=\; \lambda \texttt{x} : \forall \texttt{X}.\texttt{X} \to \texttt{X}.\ \texttt{x}[\forall \texttt{X}.\texttt{X} \to \texttt{X}]\ \texttt{x}$$

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

Polymorphic Lists

次にリストの操作の例を挙げてみる。とりあえず type constructor `List` と

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

(と `Bool`・`Nat`・`fix`・`if ~ then ~ else`) は組み込みとする。

$\lambda_{\to}$ にリストを導入した時には (11 章)、`nil` や `cons...` のために特別な推論規則を導入したが、System F ではこれらを拡張なしで扱えるという利点がある。

- 例 1. `Nat` のリスト

$$\begin{aligned}
&\texttt{l} \;=\; \texttt{cons}[\texttt{Nat}]\ 4\ (\texttt{cons}[\texttt{Nat}]\ 3\ (\texttt{cons}[\texttt{Nat}]\ 2\ (\texttt{nil}[\texttt{Nat}])))\\
&\texttt{l} : \texttt{List Nat}
\end{aligned}$$

- 例 2. `map`

$$\begin{aligned}
&\texttt{map} \;=\; \lambda \texttt{X}.\lambda \texttt{Y} \lambda \texttt{f} : \texttt{X} \to \texttt{Y}\\
&\quad (\texttt{fix}\ (\lambda \texttt{m} : (\texttt{List X}) \to (\texttt{List Y}).\\
&\qquad \lambda \texttt{l} : \texttt{List X}.\\
&\qquad\quad \texttt{if isnil}[\texttt{X}]\ \texttt{l}\\
&\qquad\quad \texttt{then nil}[\texttt{Y}]\\
&\qquad\quad \texttt{else cons}[\texttt{Y}]\ (\texttt{f}\ (\texttt{head}[\texttt{X}]\ \texttt{l}))\ (\texttt{m}\ (\texttt{tail}[\texttt{X}]\ \texttt{l}))\\
&\quad))\\
&\texttt{map} : \forall \texttt{X}.\forall \texttt{Y}.\ (\texttt{X} \to \texttt{Y}) \;\to\; (\texttt{List X}) \;\to\; (\texttt{List Y})\\
&\texttt{head}[\texttt{Nat}]\ (\texttt{map}[\texttt{Nat}][\texttt{Nat}]\ (\lambda \texttt{x} : \texttt{Nat}.\ \texttt{succ x})\ \texttt{l})\\
&\longrightarrow^{*}\ 5 : \texttt{Nat}
\end{aligned}$$

Church Encodings

System F での論理型や自然数の Church encoding を見る。一旦コア言語で表現できた機能は、後でコア言語を無駄に拡張することなく primitive として追加することができる。もちろん、参照型のように根本的な変更が必要な機能も存在する。

Boolean

論理型の真偽値は、型無し λ 計算では

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

のように表現できた (5 章)。System F でも基本的にはこれに型を付けるだけとなる。考慮すべき点は、

- 引数 `t` と `f` の型は同じであるべき。
- ただし、`t` と `f` の型を天下り的に決めてしまうべきではない (折角、polymorphism を導入したのだから)。

以上をふまえると、`tru`・`fls` の型と表現は

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

となる。

`CBool` に対する演算は

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

等のように定義できる。

Natural Numbers

自然数も基本的に型無し λ での表現に適当な annotation と型抽象を付けるだけである。

$$\begin{aligned}\mathtt{CNat} &= \forall \mathtt{X}.(\mathtt{X} \to \mathtt{X}) \to \mathtt{X} \to \mathtt{X}\\ \mathtt{c}_0 &= \lambda \mathtt{X}.\lambda \mathtt{s}.\lambda \mathtt{z}.\ \mathtt{z}\\ \mathtt{c}_1 &= \lambda \mathtt{X}.\lambda \mathtt{s}.\lambda \mathtt{z}.\ \mathtt{s}\ \mathtt{z}\\ \mathtt{c}_n &= \lambda \mathtt{X}.\lambda \mathtt{s}.\lambda \mathtt{z}.\ \underbrace{\mathtt{s}(\mathtt{s}(\ldots(\mathtt{s}\ \mathtt{z})\ldots))}_{\mathtt{n}\text{ 個}}\\ \mathtt{csucc} &= \lambda \mathtt{n} : \mathtt{CNat}\lambda \mathtt{X}.\lambda \mathtt{s} : \mathtt{X} \to \mathtt{X}.\lambda \mathtt{z} : \mathtt{X}.\ \mathtt{s}(\mathtt{n}[\mathtt{X}]\ \mathtt{s}\ \mathtt{z})\\ \mathtt{iszero} &= \lambda \mathtt{n} : \mathtt{CNat}.\ \mathtt{n}[\mathtt{CBool}]\ (\lambda \mathtt{x} : \mathtt{CBool}.\ \mathtt{fls})\ \mathtt{tru} \quad : \mathtt{CNat} \to \mathtt{CBool}\end{aligned}$$

定数と successor ができたので足し算その他も順次定義できる。

$$\texttt{cplus} \;=\; \lambda \texttt{n} : \texttt{CNat} \lambda \texttt{m} : \texttt{CNat}.\ \texttt{m}[\texttt{CNat}]\ \texttt{csucc n} \quad : \texttt{CNat} \to \texttt{CNat} \to \texttt{CNat}$$
$$\texttt{ctimes} \;=\; \lambda \texttt{n} : \texttt{CNat}. \lambda \texttt{m} : \texttt{CNat}.\ \lambda \texttt{X}. \lambda \texttt{s} : \texttt{X} \to \texttt{X}.\ \texttt{n}[\texttt{X}]\ (\texttt{m}[\texttt{X}]\ \texttt{s}) \quad : \texttt{CNat} \to \texttt{CNat} \to \texttt{CNat}$$

Encoding Lists

Church encoding の最後の例としてリストをとりあげる。基本的なアイデアは、要素 x，y，z を持つリストを、2 引数関数 f と「初期値」v を受け取って f x(f y (f z v)) を返す関数で表現するというものである。一般に要素の型と v の型は異なっていてもよいので、X 型のリストの型は

$$\texttt{List X} \;=\; \forall \texttt{R}.(\texttt{X} \to \texttt{R} \to \texttt{R}) \to \texttt{R} \to \texttt{R}$$

と表すことができる。

nil，cons，isnil はそれぞれ、

$$\texttt{nil} \;=\; \lambda \texttt{X}.(\lambda \texttt{R}. \lambda \texttt{c} : \texttt{X} \to \texttt{R} \to \texttt{R}. \lambda \texttt{n} : \texttt{R}.\ \texttt{n})\ \texttt{as List X} \quad : \forall \texttt{X}.\texttt{List X}$$
$$\texttt{cons} \;=\; \lambda \texttt{X}. \lambda \texttt{hd} : \texttt{X}. \lambda \texttt{tl} : \texttt{List X}.$$
$$(\lambda \texttt{R}. \lambda \texttt{c} : \texttt{X} \to \texttt{R} \to \texttt{R}. \lambda \texttt{n} : \texttt{R}.\ \texttt{c hd}\ (\texttt{tl}[\texttt{R}]\ \texttt{c n}))\ \texttt{as List X} \quad : \forall \texttt{X}.\texttt{X} \to \texttt{List X} \to \texttt{List X}$$
$$\texttt{isnil} \;=\; \lambda \texttt{X}. \lambda \texttt{l} : \texttt{List X}.$$
$$\texttt{l}[\texttt{Bool}]\ (\lambda \texttt{hd} : \texttt{X}. \lambda \texttt{tl} : \texttt{Bool}.\ \texttt{false})\ \texttt{true} \quad : \forall \texttt{X}.\texttt{List X} \to \texttt{Bool}$$

でよい。

やや面倒なのが head，tail で、まず head は nil に対してどういう結果を返すかという問題があるが、とりあえずは「未定義」として逃げることにする。具体的には

$$\texttt{diverge} \;=\; \lambda \texttt{X}. \lambda _ : \texttt{Unit}.\ \texttt{fix}\ (\lambda \texttt{x} : \texttt{X}.\ \texttt{x})$$

という関数を用意して、
nil の時には diverge unit の「結果を返す」ということにする。Naïve に考えると

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

だが、今は call-by-value で考えているので、これでは無条件に diverge[X] unit が評価されてしまい、非常にまずい。これを避けるにはもう一段 λ 抽象して、

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

としてやる。

tail はさらに複雑になる。まず、ペアを表す型と関数、

$$\begin{aligned}
&\texttt{Pair X Y} \;=\; \forall \texttt{R}.(\texttt{X} \to \texttt{Y} \to \texttt{R}) \to \texttt{R}\\
&\texttt{pair} \;:\; \forall \texttt{X}.\forall \texttt{Y}.\texttt{X} \to \texttt{Y} \to (\texttt{Pair X Y})\\
&\texttt{fst} \;:\; \forall \texttt{X}.\forall \texttt{Y}.(\texttt{Pair X Y}) \to \texttt{X}\\
&\texttt{snd} \;:\; \forall \texttt{X}.\forall \texttt{Y}.(\texttt{Pair X Y}) \to \texttt{Y}
\end{aligned}$$

を用意する。具体的な表現についてはテキスト 5 章・23 章などを参照して頂きたい。その上で

```
tail = λX.λl : List X.
  (fst[List X][List X](
    l[Pair (List X) (List X)]
      (λhd : X.λtl : Pair (List X) (List X).
        pair[List X][List X]
          (snd[List X][List X] tl) (cons[X] hd (snd[List X][List X] tl)))
      (pair[List X][List X] (nil[X]) (nil[X]))))
```

と書く。これを詳細に解説するのは紙面[3]の都合で避けるが、おおざっぱに言うと、空リストのペア (最後の行) から順次、(n-1 要素, n 要素) のリストのペアを構築して、その最初の要素を返す、ということをしている。

23.5 Basic Properties

$\lambda_{\to}$ で成り立つ性質の多くが System F でも成り立つ。

定理 23.5.1 (Preservation)

$\Gamma \vdash \texttt{t} : \texttt{T}$ かつ $\texttt{t} \longrightarrow \texttt{t}'$ ならば、$\Gamma \vdash \texttt{t}' : \texttt{T}$

証明は例によって typing derivation に関する帰納法。$\lambda_{\to}$ との差分だけ示せば良いが、最後の規則が (T-TABS) ならば t はそもそも値 $(\lambda \texttt{X}.\texttt{t}_1)$ なので、結局 (T-TAPP) の場合のみ示せばよい。

評価規則 (E-TAPP) によって $\texttt{t}_1[\texttt{T}_2]\ (= \texttt{t}) \longrightarrow \texttt{t}_1'[\texttt{T}_2]\ (= \texttt{t}')$ の場合、帰納法の仮定から $\Gamma \vdash \texttt{t}_1 : \forall \texttt{X}.\texttt{T}$ ならば $\Gamma \vdash \texttt{t}_1' : \forall \texttt{X}.\texttt{T}$ がいえるのでほぼ自明。

評価規則 (E-TAPPTABS) によって $(\lambda \texttt{X}.\ \texttt{t}_1)[\texttt{T}_2]\ (= \texttt{t}) \longrightarrow [\texttt{X} \mapsto \texttt{T}_2]\texttt{t}_1\ (= \texttt{t}')$ の場合、導出規則をよく見ると、ある $\texttt{T}_1$ について $\Gamma \vdash (\lambda \texttt{X}.\ \texttt{t}_1) : \forall \texttt{X}.\texttt{T}_1$ と $\Gamma, \texttt{X} \vdash \texttt{t}_1 : \texttt{T}_1$ がいえることがわかる。あとは「型についての Substitution Lemma」

$$\Gamma, \texttt{X}, \Delta \vdash \texttt{t}_1 : \texttt{T}_1 \;\Rightarrow\; \Gamma, [\texttt{X} \mapsto \texttt{T}_2]\Delta \vdash [\texttt{X} \mapsto \texttt{T}_2]\texttt{t}_1 : [\texttt{X} \mapsto \texttt{T}_2]\texttt{T}_1$$

[3] and/or 発表者の時間

を示すことによって証明できる。$\Delta \ldots$ という部分は lemma の証明に必要で、これがないと証明が途中で行き詰まってしまう[4]。

定理 23.5.2 (Progress)

t が closed かつ well-typed ならば、t は値か、ある t' に対して $t \longrightarrow t'$。

最後の規則が（T-TABS） ならば t は値 ($\lambda X.t_1$) なので OK。

（T-TAPP） の時、 t は $t_1[T_2]$ の形だが、t_1 が値でないとすると帰納法の仮定から $t_1 \longrightarrow t_1'$ となる t_1' があり、（E-TAPP）が使える。

t_1 が値ならば、t_1 は $\lambda X.t_2$ の形になるはず (厳密な証明は略) なので、（E-TAPPTABS）が使える。

定理 23.5.2 (Strong Normalization)

Well-typed な System F の項は強正規化可能である。

この証明は Girard(1972, 1989) の主要な業績だが、とても難しい (らしい) のでテキストでは扱われていない。上述の Girard や Gallier(1990) を参照。

[4] 実際に試してみるとわかる