15.5– : Subtyping

大根田裕一
oyuichi@yl.is.s.u-tokyo.ac.jp

15.5 Subtyping and Other Features

以前の章では simply typed lambda calculus に対して色々と拡張を行った (リスト、参照、バリアント等)。この節では simply typed lambda calculus with subtyping に対して色々と拡張を行い、それらの拡張が subtyping とどのように結びつくのかをみる。

15.5.1 Ascription and Casting

11 章では simply typed lambda calculus に対して t as T という項を追加した。この項は、項 t が型 T であるということを明示的にプログラム中に示しているだけで、評価の際には何の影響も及ぼさない。あくまでコーディングのしやすさを図っただけのものだった。ちなみに型推論規則は

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t \text{ as } T : T}\ T-\text{ASCRIBE}$$

だった。

ところが simply typed lambda calculus with subtyping ではもっと有用な役割を果たす。C++とか Java でいうところのキャストの役割をはたす。キャストには up-cast と down-cast の 2 種類がある。

up-cast(t as T) は、t に本来推測される型の supertype を割り当てる。up-cast は上の規則をそのまま用いれば実現できる。

$$\frac{\dfrac{\vdots}{\Gamma \vdash t : S} \quad \dfrac{\vdots}{S <: T}\ T-\text{SUB}}{\Gamma \vdash t : T}\ T-\text{ASCRIBE}$$

down-cast は up-cast の逆で、本来推測される型の subtype に cast してしまう。down-cast が正しく行われているかどうかは、静的には求めることができない。down-cast を認めるために次のような規則を用意する。

$$\frac{\Gamma \vdash t : S}{\Gamma \vdash t \text{ as } T : T}\ T-\text{DOWNCAST}$$

これは t が何らかの型に型付けできれば、どのような型にでも型付けできる、という規則である。

down-cast が正しく行われているかどうかは実行時にチェックする。評価規則として次のものを付け加える。

$$\frac{\vdash v : T}{v \text{ as } T \to v}\ E-\text{DOWNCAST}$$

ここまでの拡張で元の言語 $\lambda_{<:}$ の preservation は保たれている。しかし progress は失ってしまった (正しくない down-cast をするとプログラムの評価が stuck する)。progress を回復させる方法は 2 つある。1 つは、例外処理を導入して、正しくない cast が起こった際に例外を発生するようにすることである。例外はきちんと catch 節で捕らえられるようにする。もう 1 つの方法は、down-cast 演算を次のような動的な型

チェック演算で置き換えてしまうことである。

$$\frac{\Gamma \vdash t_1 : S \quad \Gamma, x : T \vdash t_2 : U \quad \Gamma \vdash t_3 : U}{\Gamma \vdash \text{if } t_1 \text{ in } T \text{ then } x \to t_2 \text{ else } t_3 : U} \; T - \text{TYPETEST}$$

$$\frac{\vdash v_1 : T}{\text{if } t_1 \text{ in } T \text{ then } x \to t_2 \text{ else } t_3 \longrightarrow [x \mapsto v_1]t_2} \; E - \text{TYPETEST1}$$

$$\frac{\text{not } (\vdash v_1 : T)}{\text{if } t_1 \text{ in } T \text{ then } x \to t_2 \text{ else } t_3 \longrightarrow t_3} \; E - \text{TYPETEST2}$$

15.5.2 Variants

11 章では simply typed lambda calculus にバリアントを付け加えた。バリアント型は $< l_i : T_i^{i \in 1 \ldots n} >$ だった。l_i はラベルで T_i がそれに対応する型である。そしてバリアント型の項として、$< l = t >$ as T を加えた。as T の T によって、$< l = t >$ の (バリアント) 型を明示しなくちゃ駄目だった。

$\lambda_{<:}$ をバリアントで拡張した結果が Figure 15-5(別紙) である。subtyping のおかげで、バリアント型の項に as T とかいう風に型を明示する必要がなくなった。

15.5.3 Lists

List コンストラクタは covariant(共変的) である。つまり、$S_1 <: T_1$ ならば List S_1 <: List T_1 としてよいということ。

$$\frac{S_1 <: T_1}{\text{List } S_1 <: \text{List } T_1} \; S - \text{List}$$

15.5.4 References

参照に関しては、次のような subtyping 規則にしないといけない。

$$\frac{S_1 <: T_1 \quad T_1 <: S_1}{\text{Ref } S_1 <: \text{Ref } T_1} \; S - \text{REF}$$

何故このように、List やバリアントよりも厳しい規則になっているのかというと、参照は、「読み込み先」としても「書き込み先」としても使われるからである。

15.1 で紹介された *principle of safe substitution* を基に考えてみる。Ref T_1 型の参照 (v) から値を読んでいるコンテキスト $(\cdots(!v)\cdots)$ を考える。このとき Ref S_1 <: Ref T_1 と仮定すると、このコンテキストにおいて、v の代わりに Ref S_1 型の参照を用いることができる。この時、dereference によって S_1 型の値が返ってきてしまうが、もともと !v は T_1 型であったことを考えると、$S_1 <: T_1$ という条件が必要になる。

同様に、Ref T_1 型の参照 v に値を書き込むコンテキスト $v := t$ を考える。書き込む値 (t) は明らかに型 T_1 である。このとき Ref S_1 <: Ref T_1 と仮定すると、v の代わりに Ref S_1 型の参照を使うことができるようになる。つまりこれは、S_1 型の値が読まれるところ !v で、さっき書き込んだ T_1 型の値が読まれてもかまわないということなので、$T_1 <: S_1$ という条件がが必要になる。

15.5.5 Arrays

配列は参照がたくさん並んだようなものなので、subtyping 規則は参照のものと同じ。

$$\frac{S_1 <: T_1 \quad T_1 <: S_1}{\text{Array } S_1 <: \text{Array } T_1} \; S - \text{ARRAY}$$

15.5.6 References Again

Reynolds(1988) は Source T と Sink T という型を用いてより優れた参照の型付けをした。直感的には、Source は読み込み専用の参照で、Sink は書き込み専用の参照であるといえる。型付け規則は次のようになる。

$$\frac{\Gamma|\Sigma \vdash t_1 : \mathsf{Source}\ T_{11}}{\Gamma|\Sigma \vdash !t_1 : T_{11}}\ \mathrm{T-DEREF}$$

$$\frac{\Gamma|\Sigma \vdash t_1 : \mathsf{Sink}\ T_{11} \quad \Gamma|\Sigma \vdash t_2 : T_{11}}{\Gamma|\Sigma \vdash t_1 := t_2 : \mathsf{Unit}}\ \mathrm{T-ASSIGN}$$

Source T 型の値には!演算しか施されないので、subtyping の規則として

$$\frac{S_1 <: T_1}{\mathsf{Source}\ S_1 <: \mathsf{Source}\ T_1}\ \mathrm{S-SOURCE}$$

を加える。同様に Sink T 型の値には:=しか施されないので、

$$\frac{T_1 <: S_1}{\mathsf{Sink}\ S_1 <: \mathsf{Sink}\ T_1}\ \mathrm{S-SINK}$$

を加える。最後に下の規則を加える。

$$\mathsf{Ref}\ T_1 <: \mathsf{Source}\ T_1 \quad (\mathrm{S-REFSOURCE})$$
$$\mathsf{Ref}\ T_1 <: \mathsf{Sink}\ T_1 \quad (\mathrm{S-REFSINK})$$

15.5.7 Base Types

もっといろいろな基底型があるときは、例えば Int <: Float とか Bool <: Int のような規則があると便利。

15.6 Coercion Semantics for Subtyping

今までこの章では、subtyping はより柔軟な型付けを認めてくれる、とかいうようなことをずっとやってきたが、subtyping の意味的な側面についてはあまり詳しく触れられていなかった。今まで述べてきたような式の評価方法は単純で分かりやすいものだったが、それは効率のいい評価方法ではない (特に数値演算、レコードのフィールドへのアクセス)。この章では、subtyping を含む言語に別の意味を与える coercion semantics というものについて述べる。

15.6.1 Problems with the Subset Semantics

15.5.7 では Int <: Float とかいうような規則があると便利だと述べた。これがあるとプログラマは、$4.5 + \mathsf{intToFloat}(6)$ と書かなくちゃいけないところを $4.5 + 6$ と書くだけでよくなる。しかし、Int と Float のメモリ表現は異なるので、明示的な cast を省いてしまうと、実行時に operand の型を check して、適切に intToFloat をはさみ込んでやる、とかいうようなことが必要になる。このような実行時の型 check はコンパイラの最適化によってある程度は取り除けるかもしれないけど、それでもやはり完全には取り除けなくて、実行時パフォーマンスは落ちてしまう。

レコードのフィールドアクセスの際にもパフォーマンス上の問題が生じる。subtyping を認めていない[1]言語では、各レコード型の値のフィールドの配置は静的に決まっている。その一方 subtyping を認めている言語では、あるレコード型の値のフィールドの配置を変えてしまってもよいということなので、フィールドにアクセスする際は先頭フィールドから順に実行時 check していかなくてはならない。

[1] もしくは subtyping の permutation rule を認めていない

15.6.2 Coercion Semantics

以上述べたような問題を、別の Semantics を用いることで解決する。その Semantics では、subtyping を run-time coercion によって取り去ってしまう。例えば、Int 型の a 値と Float の値 b が足し算されている式は、「a を Float に変換してから b と足す」という意味を持つようになる。また、レコードのフィールドアクセスは「そのレコードのフィールド配置を一定の順序にしてからアクセス」という意味になる。

直感的には、ある言語の Coercion Semantics は、その言語から別のより低レベルな (subtyping の無い) 言語への変換として表現される。ここでは、$\lambda_{<:}$ から、simply typed lambda calculus にレコードと Unit 型を付け加えた言語への変換と考える。

変換する際には、元の項のどの部分でどのように coercion が起こっていたのかを知る必要がある。それらは元の項の型付けの導出木を見ることで分かる。なので、変換を定式化するには、その変換を型規則に基づく関数 ($\lambda_{<:}$ における導出木を受け取って、$\lambda_{\to}$ + Unit + record の項を返す) とするのがよい。

その関数の前にまず、「型」のための関数と「subtyping」のための関数を定義する。「型」のための関数 $[\![\mathsf{T}]\!]$ は、$\lambda_{<:}$ における型を $\lambda_{\to}$ + Unit + record における型に変換する。これは単に $\lambda_{<:}$ の Top を Unit に変換するだけである。

$$
\begin{array}{lcl}
[\![\mathsf{Top}]\!] & = & \mathsf{Unit} \\
[\![\mathsf{T}_1 \to \mathsf{T}_2]\!] & = & [\![\mathsf{T}_1]\!] \to [\![\mathsf{T}_2]\!] \\
[\![\{\mathsf{l}_i : \mathsf{T}_i^{\,i\in 1\ldots n}\}]\!] & = & \{\mathsf{l}_i : [\![\mathsf{T}_i]\!]^{i\in 1\ldots n}\}
\end{array}
$$

次に「subtyping」のための関数。以下の式の中で、$\mathcal{C} :: \mathsf{S} <: \mathsf{T}$ とは、$\mathsf{S} <: \mathsf{T}$ の導出木が $\mathcal{C}$ であることを意味する。「subtyping」のための関数 $[\![\mathcal{C}]\!]$ は、$\mathsf{S} <: \mathsf{T}$ の導出木 $\mathcal{C}$ を受け取り、「$[\![\mathsf{S}]\!]$ 型の項を受け取り、$[\![\mathsf{T}]\!]$ 型の項を返す関数」を返す。

$$
\left[\!\!\left[\frac{}{\mathsf{T} <: \mathsf{T}} (\textsc{s-refl}) \right]\!\!\right] = \lambda \mathsf{x} : [\![\mathsf{T}]\!] . \mathsf{x}
$$

$$
\left[\!\!\left[\frac{}{\mathsf{S} <: \mathsf{Top}} (\textsc{s-top}) \right]\!\!\right] = \lambda \mathsf{x} : [\![\mathsf{S}]\!] . \mathsf{unit}
$$

$$
\left[\!\!\left[\frac{\mathcal{C}_1 :: \mathsf{S} <: \mathsf{U} \qquad \mathcal{C}_2 :: \mathsf{U} <: \mathsf{T}}{\mathsf{S} <: \mathsf{T}} (\textsc{s-trans}) \right]\!\!\right] = \lambda \mathsf{x} : [\![\mathsf{S}]\!] . [\![\mathcal{C}_2]\!] \, ([\![\mathcal{C}_1]\!] \, \mathsf{x})
$$

$$
\left[\!\!\left[\frac{\mathcal{C}_1 :: \mathsf{T}_1 <: \mathsf{S}_1 \qquad \mathcal{C}_2 :: \mathsf{S}_2 <: \mathsf{T}_2}{\mathsf{S}_1 \to \mathsf{S}_2 <: \mathsf{T}_1 \to \mathsf{T}_2} (\textsc{s-arrow}) \right]\!\!\right] = \lambda \mathsf{f} : [\![\mathsf{S}_1 \to \mathsf{S}_2]\!] . \lambda \mathsf{x} : [\![\mathcal{C}_2]\!] \, (\mathsf{f}([\![\mathcal{C}_1]\!] \, \mathsf{x}))
$$

$$
\left[\!\!\left[\frac{}{\{\mathsf{l}_i : \mathsf{T}_i^{\,i\in 1\ldots n+k}\} <: \{\mathsf{l}_i : \mathsf{T}_i^{\,i\in 1\ldots n}\}} (\textsc{s-rcdwidth}) \right]\!\!\right] = \lambda \mathsf{r} : \{\mathsf{l}_i : [\![\mathsf{T}_i]\!]^{i\in 1\ldots n+k}\} . \{\mathsf{l}_i = \mathsf{r}.\mathsf{l}_i^{\,i\in 1\ldots n}\}
$$

$$
\left[\!\!\left[\frac{\text{for each i} \quad \mathcal{C}_i :: \mathsf{S}_i <: \mathsf{T}_i}{\{\mathsf{l}_i : \mathsf{S}_i^{\,i\in 1\ldots n}\} <: \{\mathsf{l}_i : \mathsf{T}_i^{\,i\in 1\ldots n}\}} (\textsc{s-rcddepth}) \right]\!\!\right] = \lambda \mathsf{r} : \{\mathsf{l}_i : [\![\mathsf{S}_i]\!]^{i\in n}\} . \{\mathsf{l}_i = [\![\mathcal{C}_i]\!] \, (\mathsf{r}.\mathsf{l}_i^{\,i\in 1\ldots n})\}
$$

$$
\left[\!\!\left[\frac{\{\mathsf{k}_j : \mathsf{S}_j^{\,j\in 1\ldots n}\} \quad \text{perm. of} \quad \{\mathsf{l}_i : \mathsf{T}_i^{\,i\in 1\ldots n}\}}{\{\mathsf{l}_i : \mathsf{S}_i^{\,i\in 1\ldots n}\} <: \{\mathsf{l}_i : \mathsf{T}_i^{\,i\in 1\ldots n}\}} (\textsc{s-rcdperm}) \right]\!\!\right] = \lambda \mathsf{r} : \{\mathsf{k}_j : [\![\mathsf{S}_j]\!]^{j\in 1\ldots n}\} . \{\mathsf{l}_i = \mathsf{r}.\mathsf{l}_i^{\,i\in 1\ldots n}\}
$$

LEMMA 15.1 $\mathcal{C} :: \mathsf{S} <: \mathsf{T}$ ならば、$\vdash [\![\mathcal{C}]\!] : [\![\mathsf{S}]\!] \to [\![\mathsf{T}]\!]$ ∎

proof: $\mathcal{C}$ に関する帰納法。 ∎

最後に、$[\![\mathsf{T}]\!]$ と $[\![\mathcal{C}]\!]$ を用いて、目的となる関数 $[\![\mathcal{D}]\!]$ を定義する。$\mathcal{D} :: \Gamma \vdash \mathsf{t} : \mathsf{T}$ は、$\Gamma \vdash \mathsf{t} : \mathsf{T}$ の導出木が $\mathcal{D}$

であることを意味する。関数 $[\![\mathcal{D}]\!]$ は、$\Gamma \vdash t : T$ の導出木 $\mathcal{D}$ を引数として受け取り、$[\![T]\!]$ 型の項を返す。

$$\left[\!\!\left[\frac{x : T \in \Gamma}{\Gamma \vdash x : T}(\mathsf{T-VAR})\right]\!\!\right] = x$$

$$\left[\!\!\left[\frac{\mathcal{D}_2 :: \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 : T_1 \to T_2}(\mathsf{T-ABS})\right]\!\!\right] = \lambda x : [\![T_1]\!] . [\![\mathcal{D}_2]\!]$$

$$\left[\!\!\left[\frac{\mathcal{D}_1 :: \Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \mathcal{D}_2 :: \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}(\mathsf{T-APP})\right]\!\!\right] = [\![\mathcal{D}_1]\!]\,[\![\mathcal{D}_2]\!]$$

$$\left[\!\!\left[\frac{\text{for each } i \qquad \mathcal{D}_1 :: \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i^{i\in 1\ldots n}\} : \{l_i : T_i^{i\in 1\ldots n}\}}(\mathsf{T-RCD})\right]\!\!\right] = \{l_i = [\![\mathcal{D}_i]\!]^{i\in 1\ldots n}\}$$

$$\left[\!\!\left[\frac{\mathcal{D}_1 :: \Gamma \vdash t_1 : \{l_i : T_i^{i\in 1\ldots n}\}}{\Gamma \vdash t_1.l_j : T_j}(\mathsf{T-PROJ})\right]\!\!\right] = [\![\mathcal{D}_1]\!] .l_j$$

$$\left[\!\!\left[\frac{\mathcal{D} :: \Gamma \vdash t : S \qquad \mathcal{C} :: S <: T}{\Gamma \vdash t : T}(\mathsf{T-SUB})\right]\!\!\right] = [\![\mathcal{C}]\!]\,[\![\mathcal{D}]\!]$$

THEOREM 15.2 $\mathcal{D} :: \Gamma \vdash t : T$ ならば、$[\![\Gamma]\!] \vdash [\![\mathcal{D}]\!] : [\![T]\!]$。ただしここで、$[\![\Gamma]\!]$ は次のように再帰的に定義される*:*　$[\![\emptyset]\!] = \emptyset$　かつ　$[\![\Gamma, x : T]\!] = [\![\Gamma]\!], x : [\![T]\!]$。 ∎

proof:　$\mathcal{D}$ に関する帰納法。 ∎

以上のように coercion semantics を定義してしまえば、もう $\lambda_{<:}$ の evaluation rule を用いる必要はなくなる。$\lambda_{<:}$ の項を評価する際は、その項の型付けの導出木を $\lambda_{\to}$ + unit + record の項に変換し、その項を $\lambda_{\to}$ + unit + record の evaluation rule を元に評価すれば良いからである。

15.6.3 Coherence

subtyping を含む言語に coercion semantics を与える際に気をつけなくてはいけないことがある。例として、Base type として Int、Bool、Float、String がある言語を考える。次のような coercion があると便利なので定義する。

$$\begin{aligned}
[\![\mathsf{Bool} <: \mathsf{Int}]\!] &= \lambda b : \mathsf{Bool}.\mathsf{if}\ b\ \mathsf{then}\ 1\ \mathsf{else}\ 0 \\
[\![\mathsf{Int} <: \mathsf{String}]\!] &= \mathsf{intToString} \\
[\![\mathsf{Bool} <: \mathsf{Float}]\!] &= \lambda b : \mathsf{Bool}.\mathsf{if}\ b\ \mathsf{then}\ 1.0\ \mathsf{else}\ 0.0 \\
[\![\mathsf{Float} <: \mathsf{String}]\!] &= \mathsf{floatToString}
\end{aligned}$$

ここで、$\mathsf{intToString}(1) = \text{"1"}$ 、$\mathsf{floatToString}(1.0) = \text{"1.000"}$ と仮定してしまうと、coercion semantics の下での評価が一意に定まらなくなってしまう。その例が、$(\lambda x : \mathsf{String}.x)\mathsf{true}$。true の Bool 型が String 型に昇格する際、2 通りの方法 (Bool → Int → String と Bool → Float → String) が考えられ、互いに評価結果が変わってしまう ("1" と "1.000")。

coercion semantics に関して、*coherence* という性質を定義する。

DEFINITION 15.3 ある言語の型付けの導出木から、別のある言語の *term* への変換 $[\![-]\!]$ が coherent であるとは、変換元の言語の任意の型付け $\Gamma \vdash t : T$ に関する任意の導出木 $\mathcal{D}_1$、$\mathcal{D}_2$ について、変換先の言語において $[\![\mathcal{D}_1]\!]$ と $[\![\mathcal{D}_2]\!]$ の評価結果が等しいことである。 ∎

ちなみに、前節で述べた coercion semantics は coherent。この節で述べた coercion semantics に関しては、$\mathsf{floatToString}(0.0) = \text{"0"}$、$\mathsf{floatToString}(1.0) = \text{"1"}$ とすれば、coherent になる。もっと複雑な言語間において coherence を証明するのは、一般にはすごく大変らしい。

15.7 Intersection and Union Types

型の間に intersection の演算 ($\wedge$) を考えることで、もっと優れた subtype 関係を得られる。$\mathsf{T}_1 \wedge \mathsf{T}_2$ 型の項は、T_1 型にも T_2 型にも型付けできる項である、と定める。すると、次の 4 つの規則は自然だろう。

$$\mathsf{T}_1 \wedge \mathsf{T}_2 <: \mathsf{T}_1 \quad (\mathrm{S-INTER1})$$

$$\mathsf{T}_1 \wedge \mathsf{T}_2 <: \mathsf{T}_2 \quad (\mathrm{S-INTER2})$$

$$\frac{\mathsf{S} <: \mathsf{T}_1 \quad \mathsf{S} <: \mathsf{T}_2}{\mathsf{S} <: \mathsf{T}_1 \wedge \mathsf{T}_2} \quad (\mathrm{S-INTER3})$$

$$\mathsf{S} \to \mathsf{T}_1 \wedge \mathsf{S} \to \mathsf{T}_2 \ <: \ \mathsf{S} \to (\mathsf{T}_1 \wedge \mathsf{T}_2) \quad (\mathrm{S-INTER4})$$

intersection types は強い性質を持ち、$\lambda_{<:}$ + intersection types において、型付けできる項は正規化された[2]項であり、その逆も成り立つことが知られている。このことは、type reconstruction problem(Chapter22 でやる) が非決定的であることを示している。

union types というのもよく知られている型である。$\mathsf{T}_1 \vee \mathsf{T}_2$ 型の項は、T_1 型か T_2 型に型付けできる項である、と定める。バリアント型とは異なり、それぞれの型にタグなどは付いてないことに注意。

disjoint な union types と non-disjoint な union types の違いは、後者の型を持つ値に対しては case 文によるマッチングを使うことができないということである。このような値に関しては、T_1,T_2 両方の型にとって意味のある演算しか施すことができない。

15.8 Notes

省略。

15.5.4 Reference のつけたし

- 参照が covariant であると仮定、つまり、$\mathsf{Ref}\{\mathsf{a}:\mathsf{Int};\mathsf{b}:\mathsf{Int}\} <: \mathsf{Ref}\{\mathsf{a}:\mathsf{Int}\}$ が成り立つとしてしまうと、
$$[\mathsf{x} \mapsto \mathsf{Ref}\{\mathsf{a}:\mathsf{Int};\mathsf{b}:\mathsf{Int}\}] \vdash \mathsf{x} := \{\mathsf{a} := 1\};\ (!\mathsf{x}).\mathsf{b}$$
が型付けされてしまう。
- 参照が contra-variant であるとすると、つまり、$\mathsf{Ref}\{\mathsf{a}:\mathsf{Int}\} <: \mathsf{Ref}\{\mathsf{a}:\mathsf{Int};\mathsf{b}:\mathsf{Int}\}$ が成り立つとすると、
$$[\mathsf{x} \mapsto \mathsf{Ref}\{\mathsf{a}:\mathsf{Int}\}] \vdash (!\mathsf{x}).\mathsf{b}$$
が型付けされてしまう。

[2]評価が有限回で止まるということ