Anche il teorema di progressione non cambia, ma si basa sul lemma delle forme canoniche, la cui dimostrazione è complicata dal subtyping.

Lemma 6.10 (Forme canoniche)**.**

- *Se v è un valore di tipo $T_1 \to T_2$ allora v è della forma* $\mathsf{fn}\, x{:}S_1.M$.
- *Se v è un valore di tipo $\{\ell_i : T_i\ ^{i\in 1..n}\}$, allora v è della forma $\{k_j = v_j\ ^{j\in 1..m}\}$ tale che $\{\ell_i\ ^{i\in 1..n}\} \subseteq \{k_j\ ^{j\in 1..m}\}$*

□

Theorem 6.11 (Progressione)**.** *Sia M un termine chiuso e ben tipato, i.e. $\emptyset \vdash M : T$, allora o M è un valore, oppure esiste un termine M' tale che $M \longrightarrow M'$.* □

6.3 Il tipo massimo e il tipo minimo

Può essere conveniente introdurre un tipo distinto che sia supertipo di tutti gli altri. Introduciamo quindi la costante di tipo Top e la regola corrispondente:

$$\frac{}{T <: \mathsf{Top}}\ (\textsc{Top}) \qquad \frac{}{\mathsf{Bot} <: T}\ (\textsc{Bot})$$

Questa aggiunta non modifica le propretà del sistema, che continuano a valere. Il tipo Top è analogo al tipo `Object` in Java. In realtà in Java i tipi primitivi `int`, `char`, ... non sono sottotipi di `Object` (ma ci sono i corrispondenti tipi wrapper). In Scala invece esiste davvero un tipo massimo: `Any`, che ha due sottotipi: `AnyVal`, supertipo di ogni *value class* built-in, e.s. `Int`, `Char`, `Boolean`, `Unit`, .., e il tipo `AnyRef`, supertipo di ogni *reference class*, che corrisponde a `java.lang.Object` su piattaforma Java e a `System.Object` su piattaforma .NET.

È inoltre possibile aggiungere al sistema di tipi un tipo minimo, sottotipo cioè di ogni altro tipo. Introduciamo quindi la costante di tipo Bot e la regola corrispondente (BOT). Si noti che non c'è nessun valore che ha tipo Bot. Ciò perchè se fosse $\varnothing \vdash v : \mathsf{Bot}$ allora per subsumption avrei ad esempio sia $\varnothing \vdash v : \mathsf{Nat} \to \mathsf{Nat}$ che $\varnothing \vdash v : \{\ell : \mathsf{Nat}\}$, ma allora il lemma delle forme canoniche mi direbbe che v è sia una funzione che un record, cosa impossibile. In realtà un tipo senza alcun valore è utile per tipare quelle operazioni che non intendono restituire alcun risultato, come il sollevamento di un'eccezione[5]. Ad esempio: $\mathsf{fn}\, x : \mathsf{Nat}.\mathsf{fn}\, y : \mathsf{Nat}.\mathsf{if}\ y! = 0\ \mathsf{then}\ x/y\ \mathsf{else}\ \mathsf{throw}\ y$ è una funzione di tipo $\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$ a patto che entrambi i rami del branching abbiano tipo Nat, cosa possibile se si tipa il termine $\mathsf{throw}\ M$ con il tipo Bot:

$$\frac{\Gamma \vdash M : T_{exn}}{\Gamma \vdash \mathsf{throw}\ M : \mathsf{Bot}}\ (\textsc{Raise})$$

In Java non c'è un tipo minimo, in Scala esiste invece il tipo `Nothing`, sottotipo di qualsiasi tipo e senza che alcun valore abbia questo tipo. In Scala ad esempio c'è il metodo predefinito `error` che permette proprio di usare il meccanismo delle eccezioni rispettando il typing in modo agevole:

```
def error(msg:String) :Nothing =
  throw new RuntimeError(msg)

def dividi(x:Int, y:Int):Int =
  if (y!=0) x/y
  else error("divisione per 0!")
```

Si osservi che senza il tipo `Nothing` sarebbe possibile tipare la funzione

[5] o l'invocazione di una continuazione.

```
def dividiBIS(x:Int, y:Int):Int =
  if (y!=0) x/y   else  throw new RuntimeError(msg)
```

utilizzando la regola (RAISE) definita nella sezione precedente, cioè quella che assegna a throw M un qualsiasi tipo T. Ma in questo modo il typing della funzione `error` richiederebbe un tipo parametrico per il tipo di ritorno.

In Scala esiste inoltre il tipo `Null`, sottotipo di ogni reference class, che ha come unico valore `null`. In questo modo il valore `null` può essere promosso a qualsiasi tipo classe, i.e., qualsiasi sottotipo di `AnyRef`[6].

Osservare che il tipo `void` ad esempio di C e Java corrisponde al tipo `Unit` più che al tipo Bot perché un'espressione di tipo `void` indica che quell'espressione ha un valore non interessante e non che quell'espressione non si valuta ad alcun valore.

6.4 Subtyping algoritmico

Abbiamo osservato che nel sistema di tipi che abbiamo dato le regole non sono *syntax directed*, non possono cioè dare un algoritmo di type-checking semplicemente leggendole dal basso verso l'alto. Ciò dipende dalla presenza della regola di subsumption e dalla transitività della relazione di subtyping. La regola di subsumption infatti si applica ad un termine qualsiasi M; quindi quando cerchiamo di derivare il tipo di M, possiamo applicare sia subsumption, sia la regola di tipo specifica per la forma di quel termine. Anche la regola di transitività si può applicare ogni volta che si cerca di derivare un qualsiasi giudizio di subtyping, ma ha anche un problema in più: richiede di inventarsi un tipo intermedio per poter dimostrare le due premesse.

La soluzione a questo problema è quella di modificare le regole di tipo e di sottotipo dandone una *versione algoritmica* che sia syntax-directed. Per dimostrare la correttezza di questa nuova versione sarà sufficiente dimostrare che un giudizio di (sotto)tipo è derivabile nel nuovo sistema di regole se e solo se era derivabile nel vecchio.

Le regole che definiscono la relazione di subtyping algoritmico $S <:: T$ sono le seguenti, in cui la regola transitiva è stata eliminata[7], e le regole di subtyping per i tipi record sono state compattate (anche loro erano sovrapponibili e la regola di transitività serviva proprio per collegare "squenze" di applicazioni delle varie regole di subtyping in larghezza, profondità e permutazione).

(ALGO TOP)
$$\frac{}{T <:: \mathsf{Top}}$$

(REFLEX)
$$\frac{}{T <:: T}$$

(ALGO ARROW)
$$\frac{T_1 <:: S_1 \quad S_2 <:: T_2}{S_1 \to S_2 <:: T_1 \to T_2}$$

(ALGO SUB RECORD)
$$\frac{\{\ell_i\ ^{i\in 1\ldots n}\} \subseteq \{k_j\ ^{j\in 1\ldots m}\} \quad \text{if } k_j = \ell_i \text{ then } S_j <:: T_i}{\{k_j : S_j\ ^{j\in 1\ldots m}\} <:: \{\ell_i : T_i\ ^{i\in 1\ldots n}\}}$$

Proposition 6.12 (Correttezza e completezza del subtyping algoritmico)**.** *$S <: T$ se e solo se $S <:: T$* □

A questo punto un algoritmo di decisione del subtyping si scrive in modo diretto, leggendo le nuove regole dal basso verso l'alto. Per ottenere la versione algoritmica delle regole di tipo bisogna eliminare la regola di subsumption, o meglio, capire esattamente dove serve inserirla nelle altre regole per ottenere lo stesso effetto in modo più syntax directed.

Si può osservare che nelle derivazioni di tipo, l'uso della regola di subsumption è essenziale quando si passa ad una funzione un argomento di un sottotipo rispetto al tipo atteso. In tutti gli altri casi l'applicazione

[6] In Java Language Specification si legge: "There is also a special null type, the type of the expression null, which has no name. Because the null type has no name, it is impossible to declare a variable of the null type or to cast to the null type. The null reference is the only possible value of an expression of null type. The null reference can always undergo a widening reference conversion to any reference type."

[7] Si può eliminare anche la regola riflessiva, ma allora bisogna lasciare l'assioma $B <:: B$ di riflessività per i tipi base

della regola di subsumption si può posporre. Cioè se una derivazione di tipo termina con la regola di subsumption e poi con una regola R, allora è possibile riscrivere la derivazione posponendo l'applicazione di subsumption dopo l'applicazione di R. In altre parole, ogni derivazione di tipo si può riscrivere in modo tale che usi subsumption solo (i) quando si applica un parametro ad una funzione e (ii) al termine della derivazione. Se eliminiamo la regola di subsumption quindi, dobbiamo incorporare l'uso del subtyping direttamente nella regola che tipa l'applicazione delle funzioni. Inoltre, le applicazioni finali di subsumption nelle derivazioni di tipo non saranno più possibili, quindi daremo ai termini dei tipi più piccoli (dei sottotipi) dei tipi che davamo prima. Ad esempio sia $M = (\mathsf{fn}\, x : \{\ell{:}\mathsf{Nat}, \ell'{:}\mathsf{Nat}\}.x)\ \{\ell = 0,\ \ell' = 1\}$, allora il giudizio $\varnothing \vdash M : \{\ell : \mathsf{Nat}\}$ è derivabile solo usando subsumption, mentre nel sistema algoritmico si deriva $\varnothing \vdash M : \{\ell{:}\mathsf{Nat},\ \ell'{:}\mathsf{Nat}\}$, che è in effetti un tipo più preciso (un sottotipo).

Il sistema di tipi algoritmico è quindi dato dagli assiomi usuali e dalle seguenti regole, in cui l'unico campiamento è appunto nella regola (ALGO APP), e nel fatto che manca la regola di subsumtpion:

(ALGO RECORD)
$$\frac{\Gamma \Vdash M_i : T_i \quad i \in 1..n}{\Gamma \Vdash \{\ell_i = M_i\ ^{i\in 1..n}\} : \{\ell_i{:}T_i\ ^{i\in 1,..n}\}}$$

(ALGO PROJECT)
$$\frac{\Gamma \Vdash M : \{\ell_i{:}T_i\ ^{i\in 1,..n}\}}{\Gamma \Vdash M.\ell_i : T_i}$$

(ALGO VAR)
$$\frac{x : T \in \Gamma}{\Gamma \Vdash x : T}$$

(ALGO APP)
$$\frac{\Gamma \Vdash M : T_1 \to T_2 \quad \Gamma \Vdash N : T_1' \quad T_1' <:: T_1}{\Gamma \Vdash M\ N : T_2}$$

(ALGO FUN)
$$\frac{\Gamma, x : T_1 \Vdash M : T_2}{\Gamma \Vdash \mathsf{fn}\, x{:}T_1.M : T_1 \to T_2}$$

Theorem 6.13 (Correttezza)**.** *Se* $\Gamma \Vdash M : T$ *allora* $\Gamma \vdash M : T$.

Proof. Per induzione sulla derivazione di $\Gamma \Vdash M : T$. □

Il sistema di tipi con subsumtpion assegna diversi tipi ad uno stesso termine, cosa che non succede nel sistema algoritmico, che assegna sempre ad un termine il minimo tipo con cui il termine è tipabile.

Theorem 6.14 (Completezza o Tipi Minimi)**.** *Se* $\Gamma \vdash M : T$ *allora* $\Gamma \Vdash M : S$ *per qualche* $S <: T$.

Proof. Per induzione sulla derivazione di $\Gamma \vdash M : T$ □

EXERCISE 6.15. Torvare due termini M e N tali che $M \longrightarrow N$, $\Gamma \Vdash M : T$, $\Gamma \Vdash N : S$ con $S <:: T$ e $T \not<:: S$. Cioè esibire un caso in cui il tipo di un termine decresce durante la computazione. □

Join e meet di tipi Se aggiungiamo i termini if-then-else, come si tipa il seguente termine?

$$M = \mathsf{if}\,\mathsf{true}\,\mathsf{then}\,\{x = \mathsf{true}, y = \mathsf{false}\}\,\mathsf{else}\,\{x = \mathsf{false}, z = \mathsf{true}\}$$

Usando la regola di subsuption il termine si può tipare con $\{x : \mathsf{Bool}\}$, ma anche $\{x : \mathsf{Top}\}$. Nella versione algoritmica ha senso che M abbia come tipo *il minimo tipo T supertipo sia del ramo then che del ramo else*, cioè il lub/join (se esiste). Quindi la regola per la versione algoritmica è

(ALGO IFTHEELSE)
$$\frac{\Gamma \Vdash M_1 : \mathsf{Bool} \quad \Gamma \Vdash M_2 : T_2 \quad \Gamma \Vdash M_3 : T_3 \quad T = T_2 \vee T_3}{\Gamma \Vdash \mathsf{if}\, M_1\, \mathsf{then}\, M_2\, \mathsf{else}\, M_3 : T}$$

dove il join/lub è definito come segue: $T \vee T' = S$ se $T <: S$, $T' <: S$ e per ogni U talse che $T <: U$ e $T' <: U$ si ha $S <: U$.