Lenguajes de Programación, 2016-1
Nota de clase 15: Polimorfismo de Subtipos (*Casting*)*

Favio E. Miranda Perea Lourdes del Carmen González Huesca
Facultad de Ciencias UNAM

19 de noviembre de 2018

1. Introducción

La palabra *polimorfismo* significa, según su etimología griega, "*tener muchas formas*". En lenguajes de programación el concepto de polimorfismo significa que una misma expresión, operación o fragmento de código puede ser usado en diversos contextos, es decir, puede recibir diversos tipos. Existen al menos cuatro clases distintas de polimorfismo:

- Polimorfismo universal:
  - Polimorfismo paramétrico.
  - Polimorfismo de inclusión (subtipos).

  El polimorfismo de subtipos es indispensable para el paradigma orientado a objetos, debido intrínseca de estos lenguajes para soportar el mecanismo de herencia. En esta nota nos dedicamos a estudiar el polimorfismo de subtipos.
- Polimorfismo *ad-hoc*:
  - Sobrecarga
  - Conversión o coerción.

  Un ejemplo de polimorfismo ad-hoc es la operación + en diversos tipos como enteros, naturales o reales, pero tambien en tipos no numéricos como cadenas o booleanos. En este caso la operación se denota con el mismo símbolo pero los algoritmos subyacentes para evaluar suelen ser muy distintos, por ejemplo la suma en enteros o en números de punto flotante. En este curso no discutiremos más acerca del polimorfismo ad-hoc.

Muchas veces se critica a los lenguajes fuertemente tipados diciendo que los tipos obstaculizan el código y evitan su reuso al exigir una versión distinta de una misma función para cada tipo posible. Por ejemplo, en ausencia de tipos, la función `swap` que toma un par e intercambia sus componentes puede definirse como

*Estas notas se basan en el libro de Pierce y en material de Ralph Hinze y Gerwin Klein.

$$\mathsf{swap} =_{def} \mathsf{fun}(x) \ \Rightarrow \ \langle \mathsf{snd}\, x, \mathsf{fst}\, x \rangle$$

Sin embargo debido a la rigidez del sistema de tipos, dependiendo del tipo del par x es necesario declarar diversas funciones que difieren únicamente en los tipos. Por ejemplo

$$\begin{array}{lll} \mathsf{swap1} & =_{def} & \mathsf{fun}(x : \mathsf{Nat} \times \mathsf{Nat}) \ \Rightarrow \ \langle \mathsf{snd}\, x, \mathsf{fst}\, x \rangle \\ \mathsf{swap2} & =_{def} & \mathsf{fun}(x : \mathsf{Nat} \times \mathsf{Bool}) \ \Rightarrow \ \langle \mathsf{snd}\, x, \mathsf{fst}\, x \rangle \\ \mathsf{swap3} & =_{def} & \mathsf{fun}(x : (\mathsf{Nat} \to \mathsf{Bool}) \times \mathsf{Float}) \ \Rightarrow \ \langle \mathsf{snd}\, x, \mathsf{fst}\, x \rangle \\ & \vdots & \end{array}$$

De manera que si queremos aplicar la operación swap a argumentos de diferente tipo en un mismo programa tendríamos que definir distintas versiones de dicha función para cada tipo. Esto viola un principio básico de la ingeniería de software llamado el principio de abstracción: *Cada parte significativa de funcionalidad en un programa debe ser implementada en un único sitio en el código fuente. Si funciones similares se implementan en distintas partes del código es benéfico combinarlas en una abstrayendo las partes que difieren.*

En nuestro caso las partes que difieren son los tipos, de manera que necesitamos un mecanismo de abstracción de tipos que permita dar una definición genérica de una función para posteriormente instanciarla con tipos concretos, de esto se encarga el polimorfismo paramétrico. Tema de gran importancia pero que no estudiaremos aquí por falta de tiempo.

2. Subtipos

La idea de subtipos es fundamental en el diseño de lenguajes de programación. La subtipificación permite escribir programas más concisos y de lectura más fácil al eliminar la necesidad de hacer conversiones explícitas de tipos. También puede ser usada para expresar propiedades adicionales de programas y es un concepto absolutamente fundamental en la programación orientada a objetos donde la noción de subclase y herencia está fuertemente ligada a la noción de subtipo.
Las relaciones específicas de subtipo que se incorporen a un lenguaje en particular dependen de muchos factores. Además de las propiedades teóricas que buscamos satisfacer, tenemos que tomar en cuenta los aspectos pragmáticos de la verificación de tipos y de la semántica operacional.

Las reglas de tipificación de $\lambda^{\to}$ son muy rígidas. En una función dada el tipo del argumento debe corresponder exactamente con el tipo del dominio de la función, situación que llevará al verificador de tipos a rechazar muchos programas que, desde el punto de vista del programador, se comportan adecuadamente. Veamos un ejemplo utilizando tipos registro:

$$\vdash \lambda r : \{\ell : \mathsf{Nat}\}.(r.\ell) : \{\ell : \mathsf{Nat}\} \to \mathsf{Nat}$$

$$\vdash \{\ell = 0, \ell' = 1\} : \{\ell : \mathsf{Nat}, \ell' = \mathsf{Nat}\}$$

Sin embargo el término

$$\big(\lambda r : \{\ell : \mathsf{Nat}\}.(r.\ell)\big)\{\ell = 0, \ell' = 1\}$$

no es tipificable, aún cuando su evaluación no causa problemas:

$$\big(\lambda r : \{\ell : \mathsf{Nat}\}.(r.\ell)\big)\{\ell = 0, \ell' = 1\} \to \{\ell = 0, \ell' = 1\}.\ell \to 0$$

La rigidez de las reglas de tipificación es chocante, en este ejemplo el problema es que se le esta pasando a la abstracción un "mejor" argumento del que necesita, es decir, un argumento con más información.

En general debería permitirse cierta flexibilidad en la tipificación si un tipo es mejor que el que se necesita, en el sentido de que un elemento de aquel puede usarse siempre de manera segura cuando se espera un argumento de este.

Esta intuición se formaliza introduciendo la relación de subtipos $\leq$, la cual permite usar valores de un tipo en lugar de otro. El principio básico de esta relación es la propiedad de sustitución: si S es subtipo de T, es decir, $\mathsf{S} \leq \mathsf{T}$, entonces cualquier expresión de tipo S puede emplearse sin causar un error de tipos en cualquier contexto que requiera una expresión de tipo T. Las propiedades más importantes de la relación de subtipo son:

- Debe ser reflexiva y transitiva.

$$\frac{}{\mathsf{S} \leq \mathsf{S}} \qquad\qquad \frac{\mathsf{R} \leq \mathsf{S} \qquad \mathsf{S} \leq \mathsf{T}}{\mathsf{R} \leq \mathsf{T}}$$

- Una regla de subsunción la cual afirma que si $\mathsf{S} \leq \mathsf{T}$ entonces una expresión de tipo S puede usarse en cualquier contexto que necesite una expresión de tipo T.

$$\frac{\Gamma \vdash e : \mathsf{S} \quad \mathsf{S} \leq \mathsf{T}}{\Gamma \vdash e : \mathsf{T}}$$

Obsérvese que la regla de subsunción no está dirigida por la sintaxis; la expansión o promoción de la expresión e a un nuevo tipo T ocurre de manera implícita, es decir la expresión e no se modifica al modificar su tipo de S a T. Además la expansión implica una pérdida de información, es decir, el tipado $e : \mathsf{T}$ es más general que $e : \mathsf{S}$.

Por otra parte con la ayuda de los subtipos nuestro ejemplo anterior puede ser tipado. Si postulamos por ejemplo que $\{\ell : \mathsf{Nat}, \ell' : \mathsf{Nat}\} \leq \{\ell : \mathsf{Nat}\}$ entonces la regla de subsunción nos lleva al siguiente tipado :

$$\frac{\vdash \{\ell = 0, \ell' = 1\} : \{\ell : \mathsf{Nat}, \ell' : \mathsf{Nat}\} \qquad \{\ell : \mathsf{Nat}, \ell' : \mathsf{Nat}\} \leq \{\ell : \mathsf{Nat}\}}{\vdash \{\ell = 0, \ell' = 1\} : \{\ell : \mathsf{Nat}\}}$$

lo cual nos permite concluir que

$$\vdash \big(\lambda r : \{\ell : \mathsf{Nat}\}.(r.\ell)\big)\{\ell = 0, \ell' = 1\} : \mathsf{Nat}$$

Esto ejemplifica la propiedad fundamental de los subtipos es:

Si $\mathsf{S} \leq \mathsf{T}$ *entonces siempre que se requiera una expresión de tipo* T*, es posible usar una expresión de tipo* S *en su lugar.*

Este principio se formaliza mediante dos estilos de manejo de subtipos:

- **Interpretación mediante subconjuntos**:
  Si $\mathsf{S} \leq \mathsf{T}$ entonces toda expresión de tipo S es también una expresión de tipo T.

- **Interpretación mediante coerción**[1]:
  Si $\mathsf{S} \leq \mathsf{T}$ entonces cualquier expresión de tipo S puede convertirse de forma única en una expresión de tipo T. Es decir hay una coerción (forzamiento) del tipo S al tipo T mediante una conversión implícita. Esta coerción puede generarse en tiempo de compilación o de ejecución aunque esto no es necesario.
  Por ejemplo, la relación entre los tipos Int y Float debe ser $\mathsf{Int} \leq \mathsf{Float}$ para permitir operaciones en donde se mezclen ambos tipos de datos como en $(5 + 3.2)$, esto mediante una coerción del entero a un flotante para resolver la operación en cuestión.

3. Definición de Subtipos

En esta sección discutimos reglas de subtipos que involucran a los distintos constructores de tipo estudiados durante el curso.

3.1. Declaración básica de subtipos

Todo lenguaje de programación proporciona tipos básicos o primitivos en su implementación. Para poder definir una relación general de subtipos debemos empezar postulando la relación de subtipos en dichos tipos primitivos, por medio de reglas de inferencia de la forma

$$\frac{}{\mathsf{S}_i \leq \mathsf{T}_i}$$

desde la implementación del lenguaje. Por ejemplo

$$\frac{}{\mathsf{Nat} \leq \mathsf{Int}} \qquad \frac{}{\mathsf{Int} \leq \mathsf{Float}}$$

Una vez declaradas las relaciones iniciales de subtipo se dan juicios para subtipificar tipos más complicados como funciones, sumas, productos, etc. Veamos con detalle algunos casos.

3.2. Subtipos Función

¿ Cuándo un tipo $\mathsf{S}_1 \to \mathsf{T}_1$ es subtipo de $\mathsf{S}_2 \to \mathsf{T}_2$? veamos un ejemplo.

Supongamos que $\mathsf{Int} \leq \mathsf{Float}$ y sea $f : \mathsf{Int} \to \mathsf{Int}$. Entonces se tiene $fn : \mathsf{Int}$ para cualquier expresión $n : \mathsf{Int}$, así que podemos concluir, por subsunción, que $fn : \mathsf{Float}$. Por lo tanto es claro que:

$$\mathsf{Int} \to \mathsf{Int} \leq \mathsf{Int} \to \mathsf{Float}$$

Es decir la relación de subtipos original se preserva en el segundo argumento (el codominio) del tipo función. En tal caso se dice que esta posición es mónotona o covariante.

Por otro lado, tal vez resulte sorpresiva a primera vista la relación

$$\mathsf{Float} \to \mathsf{Int} \leq \mathsf{Int} \to \mathsf{Int}$$

[1] Del ingles *coercion*. Según el diccionario de la lengua española la palabra coerción significa *presión ejercida sobre alguien para forzar su voluntad o su conducta*

la cual es válida puesto que podemos obtener una función del tipo de la derecha a partir de una función del tipo de la izquierda mediante coerción en el argumento. Es decir restringiendo el dominio de $f : \mathsf{Float} \to \mathsf{Int}$ a elementos $x : \mathsf{Int}$ forzando a que f sea una función $f : \mathsf{Int} \to \mathsf{Int}$.
Obsérvese que la relación de subtipos original $\mathsf{Int} \leq \mathsf{Float}$ usada en el primer argumento del tipo función se invierte en el tipo función, en tal caso decimos que este argumento es antimonótono o contravariante.
Finalmente mediante transitividad se obtiene que:

$$\mathsf{Float} \to \mathsf{Int} \leq \mathsf{Int} \to \mathsf{Float}$$

En el caso general se tiene la siguiente regla

$$\frac{\mathsf{T}_1 \leq \mathsf{S}_1 \quad \mathsf{S}_2 \leq \mathsf{T}_2}{\mathsf{S}_1 \to \mathsf{S}_2 \leq \mathsf{T}_1 \to \mathsf{T}_2}$$

Veamos otros ejemplos:

$$\begin{aligned}
\mathsf{Nat} \to \mathsf{Nat} &\leq \mathsf{Nat} \to \mathsf{Int} \\
\mathsf{Int} \to \mathsf{Nat} &\leq \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{Int} \to \mathsf{Nat} &\leq \mathsf{Nat} \to \mathsf{Int} \\
(\mathsf{Nat} \to \mathsf{Int}) \to \mathsf{Nat} &\leq (\mathsf{Int} \to \mathsf{Nat}) \to \mathsf{Int} \\
((\mathsf{Int} \to \mathsf{Nat}) \to \mathsf{Int}) \to \mathsf{Nat} &\leq ((\mathsf{Nat} \to \mathsf{Int}) \to \mathsf{Nat}) \to \mathsf{Int}
\end{aligned}$$

3.3. Subtipos para Sumas y Productos

Las reglas para subtipificar sumas y productos expresan el hecho de que los argumentos son covariantes en ambos argumentos, es decir que se respeta el orden $\leq$.

$$\frac{\mathsf{S}_1 \leq \mathsf{T}_1 \quad \mathsf{S}_2 \leq \mathsf{T}_2}{\mathsf{S}_1 + \mathsf{S}_2 \leq \mathsf{T}_1 + \mathsf{T}_2} \qquad \frac{\mathsf{S}_1 \leq \mathsf{T}_1 \quad \mathsf{S}_2 \leq \mathsf{T}_2}{\mathsf{S}_1 \times \mathsf{S}_2 \leq \mathsf{T}_1 \times \mathsf{T}_2}$$

Desde el punto de vista de coerciones, las reglas nos dicen que para forzar a que un valor de tipo $\mathsf{S}_1 \star \mathsf{S}_2$ funcione como uno de tipo $\mathsf{T}_1 \star \mathsf{T}_2$, para $\star \in \{+, \times\}$, basta forzar apropiadamente cada uno de los argumentos.

3.4. Subtipos para registros

Los tipos registro son de gran importancia para modelar objetos, veamos sus reglas de subtipado:

- Amplitud. Un tipo registro con campos adicionales a otro tipo registro dado es subtipo de este. Mientras más campos tenga un tipo registro, más restricciones impone en su uso y por lo tanto describe a menos valores posibles.

$$\frac{}{\{\ell_1 : \mathsf{T}_1, \ldots, \ell_{n+k} : \mathsf{T}_{n+k}\} \leq \{\ell_1 : \mathsf{T}_1, \ldots, \ell_n : \mathsf{T}_n\}}$$

- Profundidad. Los tipos de cada campo pueden variar, siempre y cuando la relación de subtipos se mantenga en cada uno de los campos del registro.

$$\frac{\mathsf{S}_1 \leq \mathsf{T}_1 \ldots \mathsf{S}_n \leq \mathsf{T}_n}{\{\ell_1 : \mathsf{S}_1, \ldots, \ell_n : \mathsf{S}_n\} \leq \{\ell_1 : \mathsf{T}_1, \ldots, \ell_n : \mathsf{T}_n\}}$$

- Permutación. El orden de los campos de un valor de tipo registro no importa.

$$\frac{\{k_1 : \mathsf{S}_1, \ldots, k_n : \mathsf{S}_n\} \text{ permutación de } \{\ell_1 : \mathsf{T}_1, \ldots, \ell_n : \mathsf{T}_n\}}{\{k_1 : \mathsf{S}_1, \ldots, k_n : \mathsf{S}_n\} \leq \{\ell_1 : \mathsf{T}_1, \ldots, \ell_n : \mathsf{T}_n\}}$$

Como ejemplos tenemos:

$$\begin{array}{rcl} \{x : \mathsf{Nat}, y : \mathsf{Nat}\} & \leq & \{x : \mathsf{Nat}\} \\ \{x : \mathsf{Nat}) & \leq & \{x : \mathsf{Int}\} \\ \{x : \mathsf{Nat}, y : \mathsf{Nat}\} & \leq & \{x : \mathsf{Int}\} \\ \{x : \mathsf{Nat}, y : \mathsf{Nat}\} & \leq & \{y : \mathsf{Nat}, x : \mathsf{Nat}\} \end{array}$$

Dependiendo del lenguaje en particular las reglas recién definidas pueden o no ser válidas. Por ejemplo en JAVA no hay permutación.

3.5. Subtipos para variantes y tipos recursivos

Los tipos variantes y tipos recursivos son generalmente covariantes, por ejemplo para listas y árboles tenemos

$$\frac{\mathsf{S} \leq \mathsf{T}}{\mathsf{list}\,\mathsf{S} \leq \mathsf{list}\,\mathsf{T}} \qquad \frac{\mathsf{S} \leq \mathsf{T}}{\mathsf{Tree}\,\mathsf{S} \leq \mathsf{Tree}\,\mathsf{T}}$$

Si bien estas reglas parecen adecuadas y son intuitivas, en muchos casos sólo funcionan cuando las estructuras de datos en cuestión son persistentes, como en el caso de los programas funcionales puros. Como veremos más adelante, algunas restricciones adicionales deben imponerse en presencia del enunciado de asignación, pues este permite cambiar los valores de una estructura como listas o árboles.

3.6. El tipo máximo Top

El tipo Top, también denotado $\top$, se define como un elemento máximo para la relación de subtipificación $\leq$ de manera que para todo tipo S se cumple

$$\frac{}{\mathsf{S} \leq \mathsf{Top}}$$

Intuitivamente Top corresponde al tipo de todos los programas correctamente tipados. Este tipo puede ser removido de cualquier lenguaje sin dañar sus propiedades. Sin embargo puede ser de utilidad de manera que no es conveniente removerlo de un sistema con subtipos. Dos aplicaciones de Top son:

- Es de utilidad en sistemas sofisticados que combinan subtipos y polimorfismo. La definición de records como azucar sintáctico en sistemas sofisticados como $\mathsf{F}_<$ depende fuertemente del tipo Top.
- Top es importante para los lenguajes orientados a objetos pues corresponde al tipo Object.
- En presencia de Top puede simularse el polimorfismo universal con el polimorfismo de subtipos. Por ejemplo en lugar de definir una función identidad $\mathsf{id} : \mathsf{T} \to \mathsf{T}$ para cada tipo T, definimos únicamente una función $\mathsf{id} : \mathsf{Top} \to \mathsf{Top}$. De esta manera id puede recibir un argumento de cualquier tipo T pues siempre sucede que $\mathsf{T} \leq \mathsf{Top}$.

3.7. El tipo mínimo Bot

Por otro lado resulta natural preguntarse si sería conveniente agregar un tipo mínimo a la relación de subtipos, el tipo Bot o $\bot$ que sea subtipo de cualquier tipo

$$\frac{}{\mathsf{Bot} \leq \mathsf{S}}$$

Tal tipo tiene que ser vacío, es decir, no debe existir un término t tal que $\vdash t : \mathsf{Bot}$. Si este fuera el caso la regla de subsunción nos permitiría obtener por ejemplo que $\vdash v : \mathsf{Top} \to \mathsf{Top}$ y también que $\vdash v : \{\ell : \mathsf{Top}\}$, pero entonces el lema de formas canónicas (que sigue siendo válido) nos llevaría a que v es una abstracción y un record a la vez lo cual es sintácticamente imposible. Entonces ¿ para qué sirve tal tipo? Bot proporciona una manera conveniente de expresar el hecho de que algunas operaciones no deben devolver un valor, operaciones como la creación de una excepción o la llamada a una continuación. Darle a tales expresiones el tipo Bot tiene dos efectos:

- Señala al programador el hecho de que no se espera un resultado (si la expresión devolviera un resultado sería de tipo Bot, lo cual es absurdo).
- Le informa al verificador de tipos que tal expresión puede ser usada con seguridad en cualquier contexto que espera un valor de cualquier tipo.

Por ejemplo en el programa

$$\lambda x : \mathsf{T}.\mathsf{if}\ \langle x\ razonable \rangle\ \mathsf{then}\ \langle calcula \rangle\ \mathsf{else\ error}$$

la expresión error puede recibir el tipo Bot en cuyo caso el programa estará correctamente tipificado pues error puede entonces recibir cualquier tipo mediante subsunción, de manera que las dos ramas del if sean compatibles.
Por otro lado la presencia de Bot complica de manera significativa la construcción del verificador de tipos. Por ejemplo en la presencia de Bot la regla "si una aplicación rs esta tipificada correctamente entonces r debe tener tipo de función" debe modificarse en "si una aplicación rs esta tipificada correctamente entonces r es de tipo Bot o tiene tipo de función". De manera que agregar Bot a un sistema es mucho más delicado que agregar Top.

3.8. Atribuciones y *Casting*

En lenguajes con subtipos como Java y C++ las atribuciones son de gran interés y se conocen con el nombre de *casting*[2]. Una atribución es simplemente la acción de atribuir explícitamente un tipo T a una expresión e.

[2]Se agradecerá cualquier sugerencia para usar un nombre en español.

Existen dos maneras de emplear el *casting* :

- *Casting* hacia arriba (*upcasting*): A un término dado se le atribuye un supertipo del tipo esperado. Esta clase de *casting* puede verse como una forma de "abstracción", una manera de esconder la existencia de algunas partes de un valor de manera que no puedan ser usadas en cierto contexto circundante. Por ejemplo si t es un registro podemos usar un *casting* hacia arriba para esconder algunos de sus campos. En este caso la semántica estática es:

$$\frac{\Gamma \vdash e : \mathsf{T}}{\Gamma \vdash \langle \mathsf{T} \rangle\ e : \mathsf{T}}$$

  aquí $\langle \mathsf{T} \rangle\ e$ está haciendo explícito el tipo T de e dentro del código de programa.
  De esta forma en presencia de subtipos si se tiene $\mathsf{T} \leq \mathsf{R}$ entonces podemos concluir, usando subsunción que $\Gamma \vdash \langle \mathsf{R} \rangle\ e : \mathsf{R}$. Este proceso genera la siguiente regla derivada:

$$\frac{\Gamma \vdash e : \mathsf{T} \qquad \mathsf{T} \leq \mathsf{R}}{\Gamma \vdash \langle \mathsf{R} \rangle\ e : \mathsf{R}}$$

- *Casting* hacia abajo (*downcasting*): Asigna a una expresión un tipo arbitrario que probablemente el verificador de tipos no asignaría. La regla de atribución se modifica de la siguiente manera:

$$\frac{\Gamma \vdash e : \mathsf{S}}{\Gamma \vdash \langle \mathsf{T} \rangle\ e : \mathsf{T}}$$

Primero se verifica que e esté correctamente tipada y luego se le atribuye un tipo T sin ninguna restricción en la relación entre S y T.
Este tipo de atribuciones pueden ser desastrosas para la seguridad del lenguaje, sin embargo el lema a seguir es "confía pero verifica". El verificador de tipos acepta el tipo dado por el *casting* hacia abajo en tiempo de compilación. Sin embargo inserta una señal que, en tiempo de ejecución, causará que se verifique que el valor en cuestión tenga realmente el tipo que se afirma. Esto causa que la regla de evaluación $\langle \mathsf{T} \rangle\ v \to v$ se modifique como sigue:

$$\frac{\vdash v : \mathsf{T}}{\langle \mathsf{T} \rangle\ v \to v}$$

El uso de este tipo de reglas produce diversos problemas, en particular dicha regla causa que se mezclen las semánticas estática y dinámica, es decir, las reglas del tipado con las de evaluación, de manera que cualquier razonamiento formal, como lo son las pruebas por inducción, serían más complicadas que hasta ahora.

4. Subtipos para referencias

Para obtener una regla de subtipado para tipos referencia debemos preguntarnos cómo y bajo qué condiciones podemos usar una referencia de tipo $\mathsf{Ref}\ \mathsf{S}$ de manera segura en un contexto donde se espera una referencia de tipo $\mathsf{Ref}\ \mathsf{T}$.

Supóngase que $\ell : \mathsf{Ref}\ \mathsf{S}$. Hay dos situaciones importantes a analizar con respecto a ℓ cuando $\mathsf{Ref}\ \mathsf{S} \leq \mathsf{Ref}\ \mathsf{T}$.

- Si el valor de ℓ se recupera mediante la operación !, entonces dicho valor debe poder usarse de manera segura en cualquier contexto donde se espere un valor de tipo T, debido a la condición de subtipado $\mathsf{Ref\,S} \leq \mathsf{Ref\,T}$. Esto sólo es posible cuando $\mathsf{S} \leq \mathsf{T}$.

- Es posible asignar un nuevo valor a ℓ mediante la operación $\ell := e$ donde e es de tipo T, puesto que $\mathsf{Ref\,S} \leq \mathsf{Ref\,T}$. Ahora bien, como en otra parte de un mismo programa podría esperarse (mediante la recuperación $!\ell$) que el valor almacenado en la celda ℓ realmente sea de tipo S. Esto sólo puede permitirse si $\mathsf{T} \leq \mathsf{S}$.

De lo anterior se sigue que la regla de subtipos para referencias debe enunciarse como sigue:

$$\frac{\mathsf{S} \leq \mathsf{T} \quad \mathsf{T} \leq \mathsf{S}}{\mathsf{Ref\,S} \leq \mathsf{Ref\,T}}$$

Es decir el operador Ref es invariante respecto a subtipos.

Obsérvese que la relación de subtipos es sólo un preorden, es decir, no necesariamente cumple la propiedad de antisimetria, por lo que puede haber tipos S y T que cumplan las dos condiciones $\mathsf{S} \leq \mathsf{T}$ y $\mathsf{T} \leq \mathsf{S}$ sin ser iguales.

Curiosamente en JAVA se permite el subtipado covariante de arreglos (los cuales se pueden definir con referencias pues corresponden a celdas contiguas en memoria). Es decir, la regla de subtipado de arreglos es:

$$\frac{\mathsf{S} \leq \mathsf{T}}{\mathsf{S}[\,] \leq \mathsf{T}[\,]}$$

donde $\mathsf{S}[\,]$ es la sintaxis de Java para $\mathsf{Array\,S}$. Esta característica, introducida originalmente debido a la ausencia de polimorfismo que evitaba operaciones como copiar partes de un arreglo, se considera ahora como un defecto en el diseño del lenguaje puesto que afecta seriamente el desempeño de programas con arreglos. La regla de subtipos para arreglos covariante es incorrecta y el precio consiste en checar en tiempo de ejecución todas y cada una de las asignaciones a un arreglo para asegurarse de que el valor escrito es subtipo del tipo original de sus elementos.

4.1. Subtipos para continuaciones

Las continuaciones son contravariantes:

$$\frac{\mathsf{S} \leq \mathsf{T}}{\mathsf{Cont\,T} \leq \mathsf{Cont\,S}}$$

Esto debe ser claro, cualquier continuación que está esperando un valor de tipo T puede considerarse una continuación que está esperando un valor de tipo S siempre y cuando el valor S pueda considerarse un valor de tipo T.