Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3641– Lenguajes de Programación I
Profesor: Fernando Lovera

Sistema de Tipos y Tipos Compuestos

Basado en *Programming Languages Pragmatics* por Michael L. Scott

Gabriel De Ornelas **15-10377**

Capítulo 7: Sistema de Tipos

Sección 7.1: Vista General

7.1.1: Significado de Tipos

Hay 3 puntos de vista para definir un tipo:

> El denotacional, un tipo es un conjunto de valores.
>
> El estructural, un tipo es o un pequeña colección de tipos primitivos o un tipo compuesto creado aplicando un constructor de tipos a uno o más tipos primitivos.
>
> El basado en abstracción, un tipo es una interfaz que consiste en un conjunto de operaciones con semánticas mutuamente consistente y bien definida.

En semántica denotacional, un conjunto de valores es llamado dominio. Un tipo es un dominio, y el significado de una expresión es un valor del dominio que representa el tipo de la expresión.

La visión denotacional de los tipos generalmente ignora problemas de implementación.

Capítulo 7: Sistema de Tipos

Sección 7.1: Vista General

7.1.2: Polimorfismo

"Tener múltiples formas".

Los tipos deben tener ciertas características en común para mantener la correctitud. Los puntos en común se dan por:

Polimorfismo paramétrico: El código toma un tipo (o un conjunto de tipos) como parámetro, de forma explícita o implícita.

Explícito: Lenguajes estáticamente tipados. Implementación en tiempo de compilación.

Implícito: Implementación en tiempo de compilación. Más frecuente con tipados dinámicos con implementación en tiempo de ejecución.

Polimorfismo de subtipos: El código debe trabajar con valores de algún tipo específico T, pero se definen tipos adicionales para extensiones o refinamientos de T, el código debe funcionar con esos subtipos también.

Principalmente en lenguajes orientados a objetos; utiliza tipado estático. (Ejemplo: C++, Eiffel, OCaml, Java y C#)

La combinación del polimorfismo de subtipos y el paramétrico es útil para contenedores como "Lista de T" o "Pila de T", donde T no está especificada y se instancia luego como casi cualquier tipo.

Capítulo 7: Sistema de Tipos

Sección 7.1: Vista General

7.1.3: Ortogonalidad

Las características de un lenguaje deben ser independientes entre sí, para que cada una pueda utilizarse sin afectar ni ser afectada por las demás.

Formas de mejorar la ortogonalización:

Algol 68 y C: Al eliminar la distinción entre sentencias y expresiones.

Los procedimientos suelen declararse con un tipo de retorno trivial "void".

Para "eliminar" el valor de una variable es común usar "null", para enum se suele usar el tipo constructor "Option" o "Maybe".

Haskell: Los tipos que permiten a una variable ser de un tipo u otro (Maybe).

Especificar valores literales para objetos de tipo compuesto. Dichos literales a veces se conocen como agregados (aggregates).

Capítulo 7: Sistema de Tipos

Sección 7.1: Vista General

7.1.4: Clasificación de Tipos

Cada lenguaje los implementa de diferente forma:

-Numéricos: Enteros (+, -), flotantes, reales, complejos, racionales, etc.

-Discretos u ordinales: su dominio es finito y tienen una noción bien definida de predecesor y sucesor para cada elemento salvo el primero y el último. Como los Enteros.

-Booleanos como tipos de 1 byte (1 para True, 0 para False).

-Caracteres como tipos de 1 byte, aunque no siempre. Recientemente se utiliza una representación de 2 bytes para utilizar mejor el conjunto de caracteres Unicode, que es el estándar internacional.

-Enumeración: Un conjunto de elementos nombrados. type weekday = (sun, mon, tue, wed, thu, fri, sat);

-De Subrango: Es un tipo cuyos valores componen un subconjunto continuo de valores de alguna base de tipo discreta. subtype workday is weekday range mon..fri;

-Compuestos: Se crean al aplicar un constructor a uno o más tipos simples.

Capítulo 7: Sistema de Tipos
Sección 7.2: Revisión de Tipos

Reglas que restringen los tipos.

La Revisión de los Sistemas de tipos se divide en tres áreas:

Equivalencia de Tipos: Dos tipos son del mismo tipo.

Mecanismos: Conversión de Tipos (Casting), Casts sin Conversión:

Compatibilidad de Tipos: Determinan cuándo un objeto puede ser usado válidamente en un contexto determinado.

En muchos lenguajes, la compatibilidad es una relación más laxa que la equivalencia.

Mecanismo de Compatibilidad: Coerción de Tipo.

Inferencia de Tipos: Determina el tipo de una expresión compuesta a partir de los tipos de sus subexpresiones.

Aunque a veces es trivial (Ejemplo, la suma de dos enteros es un entero), puede ser complicada (Ejemplo, al tratar con conjuntos).

Capítulo 7: Sistema de Tipos

Sección 7.2: Revisión de Tipos

7.2.1: Equivalencia de Tipos

En lenguajes con tipado estático, se espera que los valores cumplan con un tipo específico en diversas situaciones: Asignación, Operaciones y Llamadas a subrutinas:

Equivalencia Estructural:

Dos tipos son iguales si están compuestos por los mismos componentes, organizados de la misma manera.

Ambigüedad:

Orden de los campos: ¿Los registros R1 (a: int; b: int) y R3 (b: int; a: int) son equivalentes? ML dice que sí; la mayoría de los demás lenguajes estructurales dicen que no.

Índices de Arreglos: ¿Los arreglos array [1..10] of char y array [0..9] of char son equivalentes? La mayoría dice que no (aunque Fortran y Ada los consideran compatibles, que es un concepto diferente).

Su principal defecto es que no puede distinguir entre tipos que el programador considera conceptualmente diferentes, pero que casualmente tienen la misma estructura interna.

Equivalencia Nominal:

Dos tipos son iguales sólo si provienen de la misma ocurrencia léxica de una definición de tipo. Básicamente, cada definición de tipo introduce un tipo nuevo y distinto, incluso si su estructura interna es idéntica.

Capítulo 7: Sistema de Tipos

Sección 7.2: Revisión de Tipos

7.2.1: Equivalencia de Tipos

Variantes de Equivalencia Nominal

Un alias de tipo se crea cuando se declara un nuevo nombre para un tipo existente (Ejemplo, type new_type = old_type; o typedef old_type new_type;).

Los tipos que son alias deben tratarse como dos nombres para el mismo tipo (equivalentes) o como nombres para dos tipos diferentes (distintos), aunque tengan la misma estructura interna.

Equivalencia Nominal Suelta (Loose Name Equivalence):

Los tipos que son alias se consideran equivalentes. El alias (type A = B;) se considera solo una declaración; A comparte la definición de B.

Equivalencia Nominal Estricta (Strict Name Equivalence):

Los tipos que son alias se consideran distintos. El alias se considera una definición; A es un tipo nuevo y distinto.

Es preferible para prevenir errores lógicos.

El lenguaje Ada ofrece una solución permitiendo al programador elegir:

Capítulo 7: Sistema de Tipos

Sección 7.2: Revisión de Tipos

7.2.1: Equivalencia de Tipos

Conversión de Tipos y Casts

Si se requiere que los tipos sean exactamente iguales, el programador debe usar una conversión de tipo explícita (o type cast) para usar un valor en un contexto que espera otro tipo.

Tipos Principales de Conversiones

1. Equivalencia Estructural:
   Los tipos son estructuralmente equivalentes pero el lenguaje usa equivalencia nominal estricta. Tienen la misma representación de bajo nivel; la conversión es solo conceptual. Coste en Tiempo de Ejecución (Run-time): Ninguno.

2. Comprobación de Subrango:
   Los tipos tienen la misma representación, pero diferentes conjuntos de valores (Ejemplo, un tipo es un subrango del otro, o uno es con signo y el otro sin signo). Se necesita código para verificar que el valor sea válido en el nuevo tipo. Si falla, hay un error semántico dinámico. Coste en Tiempo de Ejecución (Run-time): Revisión requerida.

3. Representaciones Diferentes:
   Los tipos tienen diferentes representaciones de bajo nivel (Ejemplo, entero de 32 bits a flotante de doble precisión). Se requiere una instrucción de máquina para cambiar la representación.Coste en Tiempo de Ejecución (Run-time): Conversión de Código requerida.

Lenguajes como Ada y C tienen distintas formas de abordar estas conversiones.

Capítulo 7: Sistema de Tipos
Sección 7.2: Revisión de Tipos
7.2.1: Equivalencia de Tipos
Casts sin Conversión

Es un cambio de tipo que no altera la implementación subyacente de un valor; es decir, se interpretan los bits de un valor de un tipo como si fueran de otro tipo.

Implementación en Lenguajes

Ada: Utiliza una subrutina genérica incorporada llamada unchecked_conversion.

C++: Proporciona un operador de cast específico para esta operación: reinterpret_cast.

Otros casts de C++: static_cast (conversión de tipo regular), dynamic_cast (para polimorfismo en tiempo de ejecución) y const_cast (para eliminar la calificación read-only).

El cast estilo C se define en términos de estos operadores más limpios.

Seguridad

Un Casteo de Tipo sin conversión es una subversión peligrosa del sistema de tipos del lenguaje, ya que desvirtúa las garantías de tipo.
En lenguajes con un sistema de tipos débil, estas subversiones son difíciles de encontrar.
En lenguajes con un sistema de tipos fuerte, el uso de un cast explícito (como unchecked_conversion o reinterpret_cast) al menos etiqueta los puntos peligrosos del código, facilitando la depuración si surgen problemas.

Capítulo 7: Sistema de Tipos

Sección 7.2: Revisión de Tipos

7.2.2: Compatibilidad de Tipos

La mayoría de los lenguajes de programación no exigen una equivalencia de tipos en todos los contextos, sino que el tipo de un valor sea compatible con el tipo esperado en ese contexto.

-Coerción de Tipos:

Es una conversión automática e implícita que realiza la implementación del lenguaje cuando se utiliza un valor de un tipo en un contexto que espera otro tipo.

Puede requerir código de tiempo de ejecución para realizar una comprobación.

Riesgo de Seguridad: La coerción es controvertida porque al permitir la mezcla de tipos sin indicación explícita del programador, debilita significativamente la seguridad de tipos (type security).

Flexibilidad: Algunos diseñadores argumentan que facilita la abstracción y la extensibilidad del programa, permitiendo usar tipos nuevos más fácilmente con los existentes.

Cada lenguaje ofrece cierta flexibilidad o no para la coerción, en C, por ejemplo, es peligroso porque puede cambiar el valor o la interpretación.

Capítulo 7: Sistema de Tipos

Sección 7.2: Revisión de Tipos

7.2.2: Compatibilidad de Tipos

Distinción entre Sobrecarga y Coerción de Tipos

Un nombre sobrecargado se refiere a múltiples objetos o implementaciones. La ambigüedad se resuelve por el contexto.

> En un lenguaje sin coerción, los operandos a y b deben ser del mismo tipo y el compilador elige la interpretación apropiada de la operación.
>
> En un lenguaje con coerción, si uno de los operandos no es del mismo tipo, se coacciona al tipo del otro, dependiendo del contexto.

Los literales (ej. números, cadenas, nil) son manejados dependiendo el caso:

> Constantes como nil están sobrecargadas, cuando se refirieren al puntero nulo para cualquier tipo que se necesite en el contexto.
>
> El compilador les asigna un tipo interno especial ("tipos constantes" como int const, real const, null) y coacciona estos tipos constantes especiales a un tipo más apropiado según sea necesario, incluso en lenguajes que generalmente no admiten coerciones.
>
> > Ada formaliza esto con los tipos universales para números:

Capítulo 7: Sistema de Tipos

Sección 7.2: Revisión de Tipos

7.2.2: Compatibilidad de Tipos

Tipos de Referencia Universal

Varios lenguajes proporcionan un tipo de referencia universal que pueden almacenar referencias a cualquier otro tipo de objeto.

Este tipo tiene diferentes nombres según el lenguaje: void* (C/C++), any (Clu), address (Modula-2), refany (Modula-3), Object (Java), y object (C#).

Cualquier l-value puede ser asignado a un objeto de TRU sin preocuparse por la seguridad de tipos.

Cuando una referencia es de tipo universal, el compilador no permite realizar ninguna operación sobre el objeto al que se refiere, porque su tipo específico es desconocido.

La asignación de un TRU de vuelta a un tipo de referencia específico (ej. un puntero a entero) es peligrosa si se quiere mantener la seguridad de tipos.

Para garantizar la seguridad en las asignaciones de un tipo menos específico a uno más específico es hacer que los objetos sean auto-descriptivos. Esto se logra incluyendo una etiqueta (tag) en la representación de cada objeto que indica su tipo real en tiempo de ejecución.

Capítulo 7: Sistema de Tipos

Sección 7.2: Revisión de Tipos

7.2.3: Inferencia de Tipos

Determinar cuál es el tipo de una expresión completa.

En muchos casos, la respuesta es directa:

Operadores Aritméticos: El resultado suele tener el mismo tipo que los operandos.

Comparaciones: El resultado es casi siempre Booleano.

Llamadas a Funciones: El resultado tiene el tipo declarado en el encabezado (header) de la función.

Asignaciones (si son expresiones): El resultado tiene el mismo tipo que el lado izquierdo de la asignación.

En algunos casos, la determinación del tipo no es evidente, ya que las operaciones no necesariamente preservan los tipos de los operandos. Esto ocurre notablemente en operaciones sobre:

Subrangos.

Objetos Compuestos.

Capítulo 7: Sistema de Tipos

Sección 7.2: Revisión de Tipos

7.2.3: Inferencia de Tipos

Subrangos y Conjuntos

Subrangos

El resultado de cualquier operación aritmética en un subrango adquiere el tipo base del subrango.

Verificación Dinámica: Si el resultado de la operación se asigna a una variable de tipo subrango, se requiere una comprobación semántica dinámica (en tiempo de ejecución) para asegurar que el valor resultante esté dentro de los límites del subrango de destino.

Optimización del Compilador: Para evitar la costosa verificación en tiempo de ejecución, el compilador puede rastrear el rango de valores posibles de la expresión en tiempo de compilación.

Operaciones de Conjuntos (Sets)

Tipos Compatibles: En lenguajes como Pascal y Modula, las operaciones con conjuntos (unión +, intersección *, diferencia -) eran compatibles si sus elementos tenían el mismo tipo base.

Tipo Resultante: El resultado de la operación de conjuntos es de tipo conjunto de T.

Optimización de Conjuntos: Al igual que con los subrangos, el compilador puede rastrear los posibles miembros mínimos y máximos de la expresión del conjunto, evitando la necesidad de comprobaciones de límites en tiempo de ejecución en ciertos casos.

Capítulo 7: Sistema de Tipos

Sección 7.2: Revisión de Tipos

7.2.3: Inferencia de Tipos

Declaraciones

Los lenguajes modernos (como Scala, C# 3.0, C++11, Go y Swift) permiten al programador omitir explícitamente el tipo en la declaración de una variable cuando la intención puede ser inferida a partir del contexto (es decir, el valor asignado).

El tipo de la variable se infiere automáticamente del tipo del lado derecho de la asignación.

La inferencia es especialmente útil en declaraciones complejas que involucran tipos funcionales o estructuras largas.

C++ va un paso más allá con la palabra clave decltype, que permite igualar el tipo de cualquier expresión existente.

Es particularmente útil en plantillas genéricas donde el tipo de un resultado puede ser desconocido de antemano.

Capítulo 7: Sistema de Tipos

Sección 7.2: Revisión de Tipos

7.2.4: Revisión de Tipos en ML (Meta Lenguage)

Los programadores pueden elegir omitir la declaración explícita de tipos. El compilador los infiere basándose en:

- Tipos conocidos de constantes literales (Ejemplo, 1 es int).
- Tipos explícitamente declarados en otras partes del código.
- La estructura sintáctica del programa (cómo se utilizan las variables).

La corrección de tipos se reduce a la consistencia de tipos. Un programa es correcto si el algoritmo puede deducir un tipo único para cada expresión sin contradicciones ni ambigüedades.

Cualquier expresión cuyo tipo quede incompletamente especificado tras la inferencia se vuelve automáticamente polimórfica

Como todo es una expresión en los lenguajes funcionales, las funciones también tienen tipos.

Capítulo 7: Sistema de Tipos

Sección 7.4: Testeo de Igualdad y Asignación

Testeo de Igualdad:

Para tipos simples la igualdad es una comparación binaria obvia. Para tipos complejos o abstractos, surgen ambigüedades sobre qué significa "ser igual":

Interpretaciones para Cadenas: ¿Son referencias al mismo objeto? ¿Son idénticos a nivel de bits en toda su longitud? ¿Contienen la misma secuencia de caracteres? ¿Se mostrarían igual al ser impresos?

Comparación Superficial (Shallow Comparison): Determina si dos expresiones se refieren al mismo objeto.

Comparación Profunda (Deep Comparison): Determina si los objetos a los que se refieren son iguales en algún sentido. Para estructuras complejas (listas, grafos), esto requiere una recorrida recursiva.

Asignación:

Modelo de Referencia:

Asignación Superficial (a := b): *a* pasa a referirse al mismo objeto al que se refiere *b*.

Asignación Profunda: Crea una copia completa del objeto al que refiere *b* y hace que *a* se refiera a esa copia.

Modelo de Valor:

Asignación Superficial (a := b): Copia el valor de *b* en *a*. Si ese valor contiene punteros, los objetos a los que apuntan no se copian.

Asignación Profunda: Implica copiar recursivamente la estructura de datos completa.

Capítulo 8: Tipos Compuestos

Sección 8.1: Registros (Estructuras)

8.1.1: Sintaxis y Operaciones

Una colección de campos, cada uno con un tipo simple distinto. Cada lenguaje tiene su sintaxis específica.

En C se define usando la palabra clave struct. Ejemplo: `struct element {char name[2]; int atomic_number; double atomic_weight; _Bool metallic; };`

Cada componente dentro de un registro (como name, atomic_number, etc.) se denomina un campo.

Notación específica para acceder a un campo de un registro: C (.), esta es la más común. Ejemplo: `copper.atomic_weight`, siendo que `copper` es de tipo `element`.

La mayoría de los lenguajes permiten que las definiciones de registros sean anidadas (un campo de un registro es a su vez otro registro).

C permite dos formas de anidamiento: Definición Léxica Anidada: Se define el registro interno dentro del registro externo.

Referencia por Tipo: Se define el registro interno por separado y luego se usa como tipo de un campo del registro externo.

Hay lenguajes que solo permiten la referencia por tipo; no pueden anidarse léxicamente.

Para acceder a campos anidados C utiliza la notación de punto repetida,por ejemplo: `malachite.element_yielded.atomic_number`.

ML y sus derivados especifican que el orden de los campos es insignificante.

Ejemplo: OCaml `{name="Cu"; atomic_number=29}` es igual a `{atomic_number=29; name="Cu"}`

Las tuplas en OCaml se parecen a registros cuyos campos están ordenados, pero sin nombre. En SML, las tuplas son esencialmente azúcar sintáctico para registros cuyos nombres de campo son números enteros pequeños (1, 2, etc.).

Ejemplo: SML Los valores `("Cu", 29)`, `{1="Cu", 2=29}` y `{2=29, 1="Cu"}` son considerados iguales.

Capítulo 8: Tipos Compuestos

Sección 8.1: Registros (Estructuras)

8.1.2: Disposición de Memoria y Su Impacto

Los campos se almacenan contiguamente, y el compilador utiliza un desplazamiento (offset) conocido en la tabla de símbolos para acceder a cada campo rápidamente.

Sin embargo, para un acceso eficiente, las arquitecturas de hardware a menudo requieren que los tipos de datos estén alineados a direcciones específicas (ej. múltiplos de 4 bytes)

Para lograr esta alineación, el compilador debe insertar bytes de relleno o huecos entre campos, lo que aumenta el tamaño del registro.

Para ahorrar espacio, se puede solicitar el empaquetamiento (packing), que elimina estos huecos, pero ralentiza el acceso a los campos no alineados.

En cuanto a la comparación de igualdad de registros, una simple comparación de bloques (block_compare) puede fallar debido a que los huecos contienen datos "basura" diferentes. La solución más segura es que el compilador genere una rutina de comparación campo por campo personalizada.

Aunque la mayoría de los compiladores reordenan campos para optimizar la alineación, en la programación de sistemas (como kernels), C y C++ garantizan el orden declarado para coincidir con layouts de hardware específicos.

Capítulo 8: Tipos Compuestos

Sección 8.1: Registros (Estructuras)

8.1.3: Variante de Registros (Uniones)

Las Uniones (union) en C. Permiten que diferentes variables (que se presupone que nunca se usarán simultáneamente) se asignen "encima" una de otra, compartiendo los mismos bytes en memoria.

La sintaxis de C (union) es muy similar a la de un registro (struct): union { int i; double d; _Bool b;}, El tamaño total de la unión es igual al tamaño de su miembro más grande (en el ejemplo, double d).

Históricamente, se han usado con dos propósitos:

Primero, para la Interpretación Múltiple de Bytes en la programación de sistemas, permitiendo que el mismo bloque de memoria se interprete como datos de distinto tipo o como información administrativa.

Segundo, para representar campos alternativos dentro de un registro, como los detalles salariales variables de un empleado.

Aunque las uniones tradicionales de C eran inseguras, lenguajes como Pascal introdujeron la sintaxis de registros variantes más limpios, y C++11 añadió las uniones anónimas para lograr un propósito similar de manera más moderna.

Capítulo 8: Tipos Compuestos

Sección 8.2: Arreglos

Son los tipos de datos compuestos más comunes e importantes. Son generalmente homogéneos.

Históricamente, el tipo de índice se limitaba a enteros, pero la mayoría de los lenguajes modernos permiten cualquier tipo discreto como caracteres o enumeraciones.

En cuanto al tipo de elemento, los lenguajes actuales permiten cualquier tipo, incluyendo otros arreglos o registros.

Arreglos Asociativos: Algunos lenguajes (scripting languages, Go, Swift) permiten tipos de índice no discretos (ej. cadenas de texto). Estos se implementan típicamente con tablas hash o árboles de búsqueda. Son similares a los tipos diccionario o mapa en librerías estándar.

En C++, pueden usar la sintaxis convencional de arreglo gracias a la sobrecarga de operadores.

Los arreglos dispersos presentan desafíos especiales, por lo que las librerías a menudo ofrecen implementaciones alternativas optimizadas para almacenar sólo los valores no predeterminados.

Capítulo 8: Tipos Compuestos

Sección 8.2: Arreglos

8.2.1: Sintaxis y Operaciones

La mayoría de los lenguajes acceden a un elemento de un arreglo usando corchetes ([]) con el subíndice (Ejemplo, A[3]). Aunque algunos lenguajes, usan paréntesis (()) para la indexación (Ejemplo, A(3)).

Arreglos Unidimensionales: Sintaxis C: Declara el arreglo adjuntando la notación de subíndice a la declaración del escalar (char upper[26];).

Límites de Índice: C: El límite inferior siempre es cero. Los índices de un arreglo de *n* elementos van de *0* a *n-1*.

Arreglos Multidimensionales:

Sintaxis: Muchos lenguajes permiten declararlos especificando múltiples rangos de índice separados por comas.

Arreglos de Arreglos:

En Modula-3, declarar un arreglo con subíndices múltiples (mat: ARRAY [1..10], [1..10] OF REAL;) es azúcar sintáctico para declarar un arreglo de arreglos (ARRAY [1..10] OF ARRAY [1..10] OF REAL;). Los accesos también son intercambiables (mat[3, 4] es igual a mat[3][4]).

En Ada, el arreglo multidimensional es un tipo distinto que se accede con un solo par de paréntesis, mientras que un arreglo de arreglos es una jerarquía que se accede con múltiples pares (ej. mat2(3)(4)), lo que permite operaciones como el seccionamiento o slices de las filas.

Arreglos en C: En C, las matrices multidimensionales también se declaran como un arreglo de arreglos (Ejemplo, double mat[10][10];).

Debido a la integración de punteros y arreglos en C, las slices (secciones) no son compatibles.

mat[3][4] es un elemento individual, pero mat[3] es una referencia (puntero) a la tercera fila o al primer elemento de esa fila, dependiendo del contexto.

Capítulo 8: Tipos Compuestos

Sección 8.2: Arreglos

8.2.1: Sintaxis y Operaciones

Secciones y Operaciones de Arreglos

Una sección es una porción rectangular de un arreglo o matriz.

En la mayoría de los lenguajes, las únicas operaciones permitidas en un arreglo son: Selección de un elemento y Asignación de un arreglo a otro.

Sin embargo, lenguajes más avanzados, influenciados por APL, como Fortran 90 y Ada, permiten operaciones mucho más ricas.

> Específicamente, Fortran 90 sobresale con un rico conjunto de operaciones integradas que tratan a los arreglos completos como argumentos: los operadores aritméticos se aplican elemento por elemento y sus 60 funciones intrínsecas, definidas para escalares, también operan de esta forma. Gracias a su equivalencia estructural, Fortran permite que los operandos sean compatibles con solo tener el mismo tipo de elemento y forma (shape), sin importar su definición original, facilitando manipulaciones poderosas y concisas.

Capítulo 8: Tipos Compuestos

Sección 8.2: Arreglos

8.2.2: Dimensiones, Límites y Asignación de memoria

La gestión de memoria de los arreglos se clasifica según el momento en que se conoce su tamaño.

1. Arreglos de Forma Estática: Son arreglos cuyo número de dimensiones y sus límites se especifican en la declaración (el tamaño es conocido en tiempo de compilación).

   El almacenamiento se gestiona de forma estándar:

   Estática: Para arreglos cuya vida útil es la duración total del programa.

   Pila: Para arreglos cuya vida útil está ligada a la invocación de una subrutina.

   Heap: Para arreglos asignados dinámicamente con una vida útil más general.

2. Arreglos de Forma Dinámica: Son arreglos cuya forma (tamaño o límites) no se conoce hasta el tiempo de elaboración o cuya forma puede cambiar durante la ejecución.

   La gestión de almacenamiento es más compleja porque el compilador debe: Asignar el espacio y hacer que la información de la forma (tamaño) esté disponible en tiempo de ejecución, ya que sin ella, la indexación no sería posible.

   En general, los arreglos cuyo tamaño puede cambiar durante la ejecución deben asignarse en el heap para manejar la variabilidad.

Capítulo 8: Tipos Compuestos

Sección 8.2: Arreglos

8.2.2: Dimensiones, Límites y Asignación de memoria

Vector Descriptor

Un vector descriptor está diseñado para contener la información de la forma del arreglo en tiempo de ejecución. En el caso general, debe especificar: El límite inferior de cada dimensión y el tamaño de cada dimensión, excepto la última (cuyo tamaño suele ser el tamaño del tipo de elemento y se conoce estáticamente).

Para optimizar el rendimiento de la verificación dinámica de límites, el descriptor a menudo almacena el límite superior, aunque sea redundante, para evitar recálculos constantes. El contenido de este descriptor se inicializa o actualiza cuando cambia la forma del arreglo, y en lenguajes como Fortran 90, la asignación de un arreglo puede requerir copiar no solo los datos sino también el contenido del propio vector descriptor.

Los contenidos del vector descriptor se inicializan en el momento de la elaboración o cada vez que cambian el número o los límites de las dimensiones.

Capítulo 8: Tipos Compuestos

Sección 8.2: Arreglos

8.2.2: Dimensiones, Límites y Asignación de memoria

Asignación en la Pila

Parámetros y Variables Locales de Forma Dinámica

Los parámetros de subrutinas y las variables locales son los ejemplos más sencillos de arreglos con forma dinámica.

Asignación en Pila en Tiempo de Elaboración: En Ada y C, la forma de un arreglo local se fija cuando se ejecuta la declaración:

Permiten que la forma de los arreglos locales se fije durante la elaboración de la subrutina. Para gestionar este tamaño variable, el marco de pila se divide en una porción de tamaño fijo y otra de tamaño variable.

Los arreglos dinámicos se ubican en la porción variable, pero un puntero hacia ellos, junto con el vector descriptor, se coloca en la porción fija. Este mecanismo permite que el compilador gestione la memoria de forma segura.

Capítulo 8: Tipos Compuestos

Sección 8.2: Arreglos

8.2.2: Dimensiones, Límites y Asignación de memoria

Asignación en el Heap

Los Arreglos Completamente Dinámicos son aquellos que pueden cambiar de forma y tamaño en cualquier momento, y debido a que estos cambios no siguen una estricta disciplina FIFO (First-In, First-Out), no pueden asignarse en la Pila y deben residir en el Heap.

Esto es común en cadenas dinámicas y en estructuras como ArrayLists en Java y C#. En estos casos, una operación como la concatenación de strings o el aumento de tamaño de un arreglo requiere un proceso costoso: asignar un nuevo y mayor bloque de memoria en el heap, copiar los datos del bloque viejo al nuevo, y luego liberar el bloque original.

Para optimizar el acceso, si el número de dimensiones es estático, el vector descriptor y el puntero a los datos pueden residir en el marco de pila para una rápida recuperación, aunque los datos mismos permanezcan en el heap.

Capítulo 8: Tipos Compuestos

Sección 8.2: Arreglos

8.2.3: Disposición de Memoria

Almacenamiento Unidimensional: En la mayoría de las implementaciones, los arreglos se almacenan en ubicaciones contiguas en la memoria.

Almacenamiento Multidimensional: Existen dos esquemas principales que definen qué elemento sigue al anterior en la memoria:

Por Fila (Row-Major): El índice que cambia más rápido es el último subíndice. A[i, j] es seguido por A[i, j+1].
La mayoría de los lenguajes lo usa como C, C++.

Por Columna (Column-Major): El índice que cambia más rápido es el primer subíndice. A[i, j] es seguido por A[i+1, j].
Fortran (históricamente adoptado por particularidades de la IBM 704).

Consecuencias del Orden:

El orden por fila facilita la definición de un arreglo multidimensional como un arreglo de subarreglos, ya que los elementos del subarreglo (fila) son contiguos en la memoria.

En el orden por columna, los elementos de un subarreglo (fila) no serían contiguos.

Impacto en el Rendimiento (Caché): El orden de disposición es crítico para el rendimiento debido a la caché de memoria:

Principio de Caché: La velocidad de los bucles que acceden a arreglos grandes está limitada por el rendimiento del sistema de memoria y la efectividad del caching. Cuando hay un fallo de caché, se trae no solo el elemento solicitado, sino también varios elementos siguientes (una línea de caché).

Capítulo 8: Tipos Compuestos

Sección 8.2: Arreglos

8.2.3: Disposición de Memoria

Disposición con puntero de fila

Una alternativa a la disposición de memoria contigua para arreglos multidimensionales es la Disposición con Puntero de Fila o Row Pointer Layout.

En este esquema, se utiliza un arreglo auxiliar de punteros, donde cada puntero apunta al inicio de una fila separada de datos, lo que significa que las filas no son adyacentes en la memoria. Lenguajes como Java utilizan esta disposición de forma obligatoria, mientras que C y C# la ofrecen como opción.

Aunque requiere espacio extra para los punteros, su ventaja clave es que permite la creación de arreglos irregulares o ragged arrays, donde las filas pueden tener longitudes diferentes sin desperdicio de espacio. Además, facilita la construcción de un arreglo a partir de filas preexistentes sin necesidad de copiar datos.

Es importante notar que, aunque se acceda a los elementos con la misma sintaxis de doble subíndice (ej. A[i][j]), el cálculo de la dirección en memoria es fundamentalmente diferente.

Capítulo 8: Tipos Compuestos

Sección 8.2: Arreglos

8.2.3: Disposición de Memoria

Cálculo de direcciones

Cálculo de Dirección para Arreglo Contiguo: ¿Cómo se calcula la dirección de un elemento $A[i, j, k]$ en un arreglo tridimensional con distribución por fila (row-major), donde las dimensiones van de L_n a U_n.

Se definen los tamaños de las dimensiones intermedias (Ejemplo):

S_3: sizeof elem type S_2: (número de elementos en fila) * S_3 S_1: (número de filas en plano) * S_2

La dirección del elemento $A[i, j, k]$ es: Dirección($A[i, j, k]$) = Dirección de A + (i - L_1) * S_1 + (j - L_2) * S_2 + (k - L_3) * S_3

Esta fórmula involucra tres multiplicaciones y seis sumas/restas en tiempo de ejecución.

Optimización en Tiempo de Compilación (Estático vs Dinámico):

Toda expresión que depende de constantes o de un índice estático se puede calcular en tiempo de compilación.

Esta parte estática calcula el desplazamiento de un arreglo imaginario con límites inferiores de cero.

Las multiplicaciones que dependen de los índices dinámicos i y j deben realizarse en tiempo de ejecución.

Rendimiento: Si las cargas de puntero intermedias aciertan en la caché, el costo de ejecución es comparable al de la distribución contigua (aunque requiere dos accesos a memoria).Si las cargas de puntero intermedias fallan en la caché, la ejecución será sustancialmente más lenta.

Capítulo 8: Tipos Compuestos

Sección 8.4: Conjuntos

Un conjunto es una colección desordenada de un número arbitrario de valores distintos de un tipo común. El tipo base del que provienen los elementos del conjunto se conoce como tipo base o tipo universo.

Las operaciones de conjuntos se traducen en rápidas instrucciones lógicas a nivel de bits:

Unión: OR a nivel de bit. Intersección: AND a nivel de bit. Diferencia: NOT a nivel de bit seguido de AND a nivel de bit.

Implementaciones Modernas:

Para conjuntos con un universo grande, se utilizan implementaciones cuyo tamaño es proporcional al número de elementos presentes, y no al tamaño total del tipo base.

Distinguimos entre Conjuntos Ordenados, implementados con árboles, que devuelven elementos de menor a mayor, y Conjuntos Desordenados, implementados con tablas hash.

Soporte en Lenguajes y Librerías:

Algunos lenguajes (como Python y Swift) proporcionan conjuntos como un constructor de tipo incorporado.

En muchos lenguajes orientados a objetos, los conjuntos se soportan a través de la librería estándar (Ejemplo, std::set en C++).

Los lenguajes que no tienen un constructor de conjunto incorporado, pero sí arreglos asociativos (diccionarios, mapas o hashes), pueden emular conjuntos desordenados.

Capítulo 8: Tipos Compuestos

Sección 8.5: Punteros y Tipos Recursivos

Un tipo recursivo es aquel cuyos objetos pueden contener referencias a otros objetos del mismo tipo. Se emplean para construir una amplia variedad de estructuras de datos "enlazadas", como listas y árboles. La mayoría son registros, ya que necesitan contener otros campos además de la referencia recursiva. La forma de implementar tipos recursivos depende del modelo de variables del lenguaje:

- Modelo de Referencia (Java): Incluir una referencia al mismo tipo es sencillo, ya que todos los campos son referencias.
- Modelo de Valor (como C): Los tipos recursivos requieren la noción de puntero: una variable cuyo valor es una referencia a algún objeto.

La Fuga de Memoria (Memory Leak) ocurre cuando un programa falla al recuperar el espacio para objetos no necesarios.

Reclamación de Almacenamiento:
- Reclamación Explícita (Manual)
- Reclamación Automática (Garbage Collection)

Capítulo 8: Tipos Compuestos

Sección 8.5: Punteros y Tipos Recursivos

8.5.1: Sintaxis y Operaciones

Las operaciones sobre punteros incluyen la asignación y desasignación de objetos en el heap, la desreferenciación para acceder a los objetos, y la asignación de un puntero a otro. El comportamiento de estas operaciones depende del tipo de lenguaje.

Lenguajes Funcionales: típicamente utilizan un modelo de referencia, donde los objetos se asignan automáticamente.

Lenguajes Imperativos: Pueden usar un modelo de valor, un modelo de referencia, o una combinación.

> Modelo por Valor (Ej: C o Ada): La asignación A=B coloca el valor de B en A. Para que A y B apunten al mismo objeto, deben ser explícitamente punteros.
>
> Modelo por Referencia (Smalltalk o Ruby): La asignación A=B siempre hace que A se refiera al mismo objeto al que se refiere B.

Java y C# (Modelo Intermedio):

> Java hace explícita la implementación del modelo de referencia dependiendo del contexto.

Capítulo 8: Tipos Compuestos

Sección 8.5: Punteros y Tipos Recursivos

8.5.1: Sintaxis y Operaciones

Modelo por Referencia

En el Modelo por Referencia, los lenguajes como la familia ML (ej. OCaml) utilizan el mecanismo de variantes o etiquetas para declarar tipos recursivos, como un árbol, donde un nodo puede ser Empty o un Node que contiene referencias a otros nodos del mismo tipo.

> En memoria, cada parte del árbol es un bloque etiquetado asignado en el heap, y las referencias simplemente apuntan a estas tuplas.
>
> Por su parte, Lisp utiliza una representación más uniforme para estructuras recursivas: un árbol se modela como una lista anidada donde cada lista es una celda cons con referencias a la cabeza y al resto.

Dado que los lenguajes ML son tipados estáticamente, tienen reglas para manejar tipos que se referencian mutuamente, declarándolos juntos con la sintaxis type ... and ... and ... para evitar problemas de dependencia. Si se programa en un estilo puramente funcional, las estructuras recursivas son inherentemente acíclicas; la creación de estructuras circulares requiere el uso de las características imperativas del lenguaje.

Capítulo 8: Tipos Compuestos

Sección 8.5: Punteros y Tipos Recursivos

8.5.1: Sintaxis y Operaciones

Modelo por Valor

En el Modelo por Valor (C y Ada), la declaración de tipos recursivos requiere el uso de punteros y declaraciones adelantadas del tipo.

La asignación de nuevos nodos en el Heap se hace mediante funciones como malloc en C, donde el programador debe especificar el tamaño y la asignación no es segura a nivel de tipos, o con el operador new en Ada, C++, Java y C#, que es seguro a nivel de tipos y puede invocar constructores.

Para acceder al objeto apuntado, se usa la desreferenciación explícita: el asterisco * en C o la flecha ^ en Pascal. C ofrece la abreviatura -> para acceder a campos de estructuras apuntadas. Ada simplifica esto con la desreferenciación implícita, usando la sintaxis de punto (.) para registros directos y apuntados. Por otro lado, lenguajes funcionales imperativos como OCaml usan la asignación := solo para punteros (ref) y el operador ! para la desreferenciación explícita, haciendo muy clara la distinción entre los valores-L y los valores-R.

Capítulo 8: Tipos Compuestos

Sección 8.5: Punteros y Tipos Recursivos

8.5.1: Sintaxis y Operaciones

Punteros y Arreglos en C

Por defecto, un nombre de arreglo sin subíndices se convierte automáticamente en un puntero a su primer elemento, permitiendo la Aritmética de Punteros, donde la suma a + k se escala automáticamente por el tamaño del tipo base.

El operador de Subíndice [] es una abreviatura sintáctica de la aritmética de punteros y la desreferenciación, donde $E_1[E_2]$ es equivalente a $*((E_1) + (E_2))$. Aunque punteros (int *a) y arreglos (int b[10]) son distintos en sus declaraciones (uno asigna una dirección y el otro, espacio para todos los elementos), cuando un arreglo se pasa como argumento a una función, se pasa un puntero al primer elemento.

Es crucial notar que el operador sizeof es la única forma de distinguir entre ambos: si se aplica a un arreglo devuelve el tamaño total del arreglo, pero si se aplica a un puntero, devuelve solo el tamaño de la dirección en sí, lo cual se utiliza para calcular el número de elementos con la expresión {sizeof}(a) / {sizeof}(a[0]).

Capítulo 8: Tipos Compuestos

Sección 8.5: Punteros y Tipos Recursivos

8.5.2: Referencia Colgante

La Referencia Colgante (Dangling Reference) es un error peligroso donde un puntero vivo apunta a una dirección de memoria que ya ha sido liberada.

Este riesgo existe tanto para objetos en la Pila, cuando un puntero de un ámbito más amplio sobrevive a la función que declaró la variable local, como para objetos en el Heap, cuando el programador libera explícitamente el objeto (con free o delete) mientras otros punteros continúan referenciándolo.

La consecuencia es grave: el software de implementación puede reutilizar ese espacio liberado, y el uso posterior del puntero colgante puede resultar en la lectura o escritura de datos que pertenecen a otro objeto, lo que lleva a comportamiento indefinido y a la potencial corrupción de estructuras de gestión de la memoria.

Lenguajes como Ada buscan mitigar este problema aplicando una regla estricta: prohíben que un puntero apunte a cualquier objeto con un tiempo de vida más breve que el tipo del puntero, aunque la solución más robusta y moderna es el uso de la recolección de basura automática.

Capítulo 8: Tipos Compuestos

Sección 8.5: Punteros y Tipos Recursivos

8.5.3: Recolector de Basura

La Recuperación Explícita de memoria en el heap (con free o delete) es una fuente importante de errores como fugas de memoria y referencias colgantes. Por ello, la Recolección de Basura (GC) automática es una alternativa muy atractiva.

La GC es esencial para los lenguajes funcionales, ya que gran parte de su gestión de objetos ocurre en el heap. También ha sido adoptada ampliamente por lenguajes imperativos modernos como Java y C#, brindando una inmensa comodidad al eliminar la necesidad de que el programador gestione el tiempo de vida de los objetos y, por lo tanto, eliminando el riesgo de referencias colgantes.

Aunque su implementación es compleja y tiende a ser más lenta que la recuperación manual optimizada, los beneficios en la seguridad y la simplificación de la programación son indiscutibles.

Capítulo 8: Tipos Compuestos

Sección 8.5: Punteros y Tipos Recursivos

8.5.3: Recolector de Basura

Conteo de Referencias

Una vez que un objeto deja de ser referenciado, se convierte en basura. La técnica de recolección de basura más simple es el Conteo de Referencias, donde cada objeto del heap tiene un contador que registra cuántos punteros activos lo referencian.

El contador se incrementa o decrementa durante las asignaciones, y cuando llega a cero, el objeto se reclama y el sistema decrementa recursivamente los contadores de los objetos que este apuntaba. Para que esta técnica funcione, el lenguaje debe ser fuertemente tipado y la implementación debe rastrear la ubicación exacta de cada puntero usando descriptores de tipo generados por el compilador.

Sin embargo, la limitación más seria del Conteo de Referencias es que falla en liberar estructuras circulares (objetos que se referencian mutuamente), ya que sus contadores nunca llegarán a cero. Por esta razón, solo es completamente seguro en estructuras que están garantizadas como acíclicas, como las cadenas de longitud variable.

Capítulo 8: Tipos Compuestos

Sección 8.5: Punteros y Tipos Recursivos

8.5.3: Recolector de Basura

Punteros Inteligentes

Los Punteros Inteligentes son objetos de programa que imitan el comportamiento de los punteros, pero añaden semántica extra para automatizar tareas, como la gestión de memoria y la verificación de límites.

En lenguajes como C++, que requieren liberación manual, la librería estándar ofrece tres punteros clave que aprovechan la sobrecarga de operadores y destructores:

El std::unique_ptr otorga propiedad exclusiva sobre el objeto; cuando el puntero sale de su ámbito, su destructor reclama automáticamente el objeto.

El std::shared_ptr implementa el conteo de referencias, incrementando y decrementando un contador de manera segura, liberando el objeto solo cuando el contador llega a cero.

Finalmente, el std::weak_ptr se utiliza para referenciar un objeto sin contribuir al conteo de referencias, siendo esencial para romper ciclos en estructuras de datos y evitar extensiones artificiales en el tiempo de vida de un objeto.

Capítulo 8: Tipos Compuestos

Sección 8.5: Punteros y Tipos Recursivos

8.5.3: Recolector de Basura

Recolección de Basura por Rastreo

Conteo de Referencias

En principio un objeto se define como útil si existe al menos un puntero que lo referencie.

Esta definición falla al no poder reclamar estructuras circulares que ya no son accesibles desde el programa principal, incluso si sus contadores de referencias no son cero (basura no recolectada).

Una definición superior es que un objeto es útil si puede ser alcanzado siguiendo una cadena de punteros válidos que comienza desde algo que tiene un nombre (es decir, una raíz externa al heap).

Los recolectores de basura por rastreo operan siguiendo esta definición:

Exploran el heap recursivamente.

Comienzan desde punteros externos.

Determinan qué objetos son efectivamente alcanzables y, por lo tanto, útiles.

Capítulo 8: Tipos Compuestos

Sección 8.5: Punteros y Tipos Recursivos

8.5.3: Recolector de Basura

Marcar y Barrer - Tipo de Rastreo

Tomando en cuenta que un bloque es útil si es alcanzable siguiendo una cadena de punteros válidos que comienza fuera del heap.

El algoritmo Marcar y Barrer se ejecuta cuando la cantidad de espacio libre restante en el heap cae por debajo de un umbral mínimo predefinido.

El recolector de basura ejecuta el algoritmo en tres fases:

1. Marcado Inicial: El recolector recorre todo el heap, marcando tentativamente cada bloque como "inútil" (basura).

2. Rastreo y Marcado (Mark):
   El recolector comienza con todos los punteros fuera del heap (las raíces). Explora recursivamente todos los datos enlazados a partir de las raíces. Cada bloque recién descubierto en el heap se marca como "útil" (vivo).

   Si el recolector encuentra un bloque que ya está marcado como "útil", regresa sin hacer recursión para evitar el trabajo duplicado.

3. Barrido (Sweep):
   El recolector vuelve a recorrer el heap y todo bloque que todavía esté marcado como "inútil" se mueve a la lista de espacio libre y se recupera.

Capítulo 8: Tipos Compuestos

Sección 8.5: Punteros y Tipos Recursivos

8.5.3: Recolector de Basura

Mejoras de Marcar y Barrer

La fase de exploración o marcado (Paso 2) del algoritmo Marcar y Barrer es inherentemente recursiva (requiere de mucho espacio).

Inversión de punteros: Busca evitar el uso de una pila de llamadas explícita durante el recorrido de marcado.

Invierte los punteros a medida que recorre el heap.

Copiar y Parar (Recolección de Copia): Se utiliza para lograr la compactación de almacenamiento (reduciendo la fragmentación externa) mientras realiza simultáneamente la recolección de basura.

Este algoritmo divide el heap en dos mitades: el Espacio Desde (From-Space), donde se realizan las asignaciones, y el Espacio Hacia (To-Space), que está inicialmente vacío. Cuando el Espacio Desde se llena, el recolector comienza su rastreo desde las raíces y copia cada bloque alcanzable (vivo) a ubicaciones contiguas en el Espacio Hacia, eliminando inmediatamente la fragmentación externa.

Los punteros en la antigua ubicación se actualizan para redirigir a la nueva. Una vez finalizado el rastreo, el Espacio Desde se descarta, el programa continúa en el Espacio Hacia, y las mitades se intercambian.

El costo principal es que solo se utiliza la mitad del heap en un momento dado, lo que subutiliza el espacio, pero su gran beneficio es la compactación automática de la memoria, lo que mejora la localidad de referencia.

Capítulo 8: Tipos Compuestos

Sección 8.5: Punteros y Tipos Recursivos

8.5.3: Recolector de Basura

Recolección de Basura Generacional y Conservadora

La Recolección Generacional optimiza el rastreo explotando el principio de que la mayoría de los objetos del heap son de vida corta. Este mecanismo divide el heap en múltiples regiones o generaciones (generalmente joven y antigua).

> El recolector se centra primero en la región más joven, que tiene la mayor probabilidad de contener basura, y solo rastrea la región antigua si es necesario. Si un objeto sobrevive a algunas recolecciones en su región, es promovido a la siguiente generación más antigua, de manera similar al algoritmo de Copiar y Parar.
>
> Un desafío clave es rastrear los punteros que van de la región antigua a la nueva. Para manejar esto, el compilador instrumenta el código con una Barrera de Escritura (write barrier), que es código generado en cada operación de asignación para detectar y registrar los raros punteros "de antiguo a nuevo" en una lista, asegurando que sean tratados como raíces durante el rastreo de la región joven.

La Recolección de Basura Conservadora permite implementar la recuperación automática incluso en lenguajes no fuertemente tipados al eliminar la necesidad de identificar explícitamente los punteros.

> El algoritmo Marcar y Barrer procede de manera conservadora, asumiendo tentativamente que cualquier valor entero alineado por palabra que parezca ser una dirección válida dentro del heap es, de hecho, un puntero útil. El recolector escanea las raíces (pila y almacenamiento global) y marca recursivamente como útil cualquier bloque alcanzable a través de estas coincidencias.
>
> Este enfoque es completamente seguro ya que nunca reclama un bloque útil; sin embargo, tiene la limitación de que puede dejar basura sin reclamar (false memory leak) si un valor de datos coincide accidentalmente con una dirección del heap. Además, debido a esta ambigüedad, la recolección conservadora no puede realizar la compactación de la memoria, lo que puede llevar a una fragmentación a largo plazo.

Capítulo 8: Tipos Compuestos

Sección 8.6: Listas

Una Lista es una estructura recursiva ideal para lenguajes funcionales y lógicos que operan mediante recursión, siendo más eficientes en entornos con recolección de basura automática debido a las muchas operaciones de creación de objetos.

La estructura varía significativamente:

> En lenguajes ML (como OCaml), las listas son homogéneas (todos los elementos del mismo tipo) e implementadas como una cadena de bloques enlazados.
>
> En contraste, Lisp utiliza listas heterogéneas (cualquier tipo de elemento) implementadas como una cadena de celdas cons, donde cada celda tiene dos punteros: uno al elemento (car) y otro al resto de la lista (cdr).

Esto permite la existencia de Listas Impropias en Lisp, donde el puntero final (cdr) no apunta a la lista vacía (nil) sino a un átomo. Un aspecto clave de Lisp es que sus propios programas son listas, requiriendo que el programador las cite (ej. '(abcd)) para evitar que sean evaluadas como código.

Finalmente, las Listas por Comprensión ofrecen una sintaxis concisa para crear listas, inspirada en la notación de conjuntos, que consta de una expresión, un enumerador y filtros, y es soportada por lenguajes como Python y Haskell.