CENG 218 Programlama Dilleri

Bölüm 8: Veri Türleri

Öğr.Gör. Şevket Umut Çakır

Pamukkale Üniversitesi

Hafta 11

Hedefler

- Veri türlerini ve tür bilgilerini anlamak
- Basit türleri anlamak
- Tür yapıcılarını anlamak
- Örnek dillerdeki tür isimlendirmesini ayırt edebilme
- Tür eşdeğerliğini anlamak
- Tür denetimini anlamak
- Tür dönüşümünü anlamak
- Polimorfik tür denetimini anlamak
- Açık polimorfizmi anlamak
- TinyAda'da tür denetimi gerçekleştirmek

Giriş

- Her program, bir sonuca ulaşmak için açık veya örtük olarak verileri kullanır.
- Veri türü(Data type): verilerin programlama dillerinde temsilinin altında yatan temel kavram
- En ilkel biçimindeki veriler, yalnızca bir bit koleksiyonudur
  - Bu, büyük programlar için gerekli olan soyutlama türlerini sağlamaz
- Programlama dilleri, bir dizi basit veri varlığı ve yenilerini oluşturmak için mekanizma içerir

Giriş

- Makine bağımlılıkları genellikle bu soyutlamaların uygulanmasının bir parçasıdır
- Verinin **sonluluğu**(**Finitude** of data):
  - Matematikte tam sayı kümesi sonsuzdur
  - Donanımda her zaman en büyük ve en küçük tamsayı vardır
- Hangi tür bilgisinin açık hale getirilmesi ve programın doğruluğunu onaylamak için kullanılması gerektiği konusunda dil tasarımcıları arasında çok fazla anlaşmazlık vardır

Giriş I

Statik Tip Kontrolü Gereksinimi

- Bir tür statik tip kontrolüne sahip olmanın nedenleri:
  - **Yürütme verimliliği(Execution efficiency)**: derleyicilerin belleği verimli bir şekilde tahsis etmesine izin verir
  - **Çeviri verimliliği(Translation efficiency)**: statik türler, derleyicinin derlenecek kod miktarını azaltmasına izin verir
  - **Yazılabilirlik(Writability)**: birçok yaygın programlama hatasının erken yakalanmasına izin verir
  - **Güvenlik** ve **güvenilirlik**(**Security** and **Reliability**): yürütme hatalarının sayısını azaltır
  - **Okunabilirlik(Readability)**: açık türler(explicit types) veri tasarımını belgelemeye yardımcı olur
  - **Belirsizlikleri kaldırma(Remove ambiguities)**: aşırı yüklemeyi çözmek için açık türler kullanılabilir
  - **Tasarım aracı(Design tool)**: açık türler, tasarım hatalarını vurgular ve çeviri zamanı hataları olarak görünür

Giriş II

Statik Tip Kontrolü Gereksinimi

- **Arayüz tutarlılığı ve doğruluğu(Interface consistency and correctness)**: açık veri türleri, büyük programların doğrulanmasına yardımcı olur

- **Veri türü(Data type)**: temel soyutlama mekanizması

Veri Türleri ve Tür Bilgisi

- Program verileri türlerine göre sınıflandırılabilir
- Tür adı, bu türden bir değişkenin tutabileceği olası değerleri ve bu değerlerin dahili olarak temsil edilme şeklini temsil eder.
- Veri türü (tanım 1): bir dizi değer
  - `int x;`, $x \in Tamsayilar$ ile aynı anlama gelir
- Veri türü (tanım 2): belirli özelliklere sahip, bir dizi değer ve bu değerler üzerinde bir dizi işlem
  - Bir veri türü aslında matematiksel bir cebirdir

Veri Türleri ve Tür Bilgisi

- **Tür denetimi(Type checking)**: bir programdaki tür bilgilerinin tutarlı olup olmadığını belirlemek için bir çevirmenin geçtiği süreç
- **Tür çıkarımı(Type inference)**: ifadelere türleri ekleme/iliştirme işlemi
- **Tür oluşturucular(Type constructors)**: daha karmaşık türler oluşturmak için bir grup temel türle kullanılan mekanizmalar
  - Örnek: Dizi, bir temel türü ve bir boyut veya aralık göstergesini alır ve yeni bir veri türü oluşturur
- **Kullanıcı tanımlı türler(User-defined types)**: tür oluşturucular kullanılarak oluşturulan türler

Veri Türleri ve Tür Bilgisi

- **Tür bildirimi(Type declaration)** (veya **tür tanımı(type definition)**): bir adı yeni bir veri türüyle ilişkilendirmek için kullanılır
- **Anonim tür(Anonymous type)**: adı olmayan bir tür
  - Bir ad atamak için C'de `typedef` kullanabilir
- **Tür eşdeğerliği(Type equivalence)**: iki türün aynı olup olmadığını belirleme kuralları
- **Tür sistemi(Type system)**: tür oluşturma yöntemleri, tür eşdeğerlik algoritması, tür çıkarım kuralları ve tür doğruluğu kuralları

Veri Türleri ve Tür Bilgisi

- **Güçlü türlü(Strogly typed)**: Tüm veri bozucu hataların mümkün olan en erken noktada algılanacağını garanti eden statik olarak uygulanan bir tür sistemi belirten bir dil
  - Birkaç istisna dışında (dizi alt simge sınırları gibi) hatalar çeviri sırasında tespit edilir
- **Güvenli olmayan programlar(Unsafe programs)**: veri bozucu hatalar içeren programlar
- **Geçerli programlar(Legal programs)**: güvenli programların uygun alt kümesi; bir çevirmen tarafından kabul edilen programlar

Veri Türleri ve Tür Bilgisi

- **Zayıf türlü dil(Weakly-typed language)**: güvenli olmayan programlara izin verebilecek boşluklara sahip olan dil
- **Türlenmemiş(Untyped)** (veya **dinamik türlü(dynamically typed)**) diller: statik tip sistemleri olmayan diller
  - Tüm güvenlik kontrolleri yürütme zamanında gerçekleştirilir
- **Çok biçimlilik(Polymorphism)**: adların birden çok türe sahip olmasına izin verirken, statik tür kontrolüne de izin verir

Basit Türler

- **Önceden tanımlanmış türler(Predefined types)**: diğer tüm türlerin oluşturulduğu bir dil ile sağlanan türler
  - Genellikle anahtar kelimeler(keywords) veya önceden tanımlanmış tanımlayıcılar(predefined identifiers) kullanılarak belirtilir
  - Sayısal türler gibi temel türlerde bazı varyasyonlar içerebilir
- **Basit türler(Simple types)**: doğal aritmetik veya sıralı yapılarından başka bir yapıya sahip değildir
  - Genellikle önceden tanımlanmış türleri içerir
  - **Numaralandırılmış(Enumerated)** türleri ve **alt aralık(subrange)** türlerini içerir

Basit Türler

- Numaralandırılmış türler: öğeleri açıkça adlandırılmış ve listelenmiş olan kümeler
  - Örnek: C dilinde: `enum Color {Red, Green, Blue};`
  - Çoğu dilde **sıralanır(ordered)**: değerlerin listelendiği sıra önemlidir
  - Çoğu dil, numaralandırılmış türler için önceden tanımlanmış bir **ardıl(successor)** ve **öncül(predecessor)** işlem içerir
  - Listelenen değerlerin dahili olarak nasıl temsil edildiğine dair hiçbir varsayımda bulunulmaz

Basit Türler

```
with Text_IO; use Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Enum is
   type Color_Type is (Red, Green, Blue);
   -- Color_Type değerlerini yazdırabilmek için Color_IO
   ↪  tanımlanır --
   package Color_IO is new Enumeration_IO(Color_Type);
   use Color_IO;
   x : Color_Type := Green;
begin
   x := Color_Type'Succ(x); -- x şimdi Blue
   x := Color_Type'Pred(x); -- x şimdi Green
   put(x); -- GREEN yazar
   new_line;
end Enum;
```

Şekil: Numaralandırılmış türü gösteren bir Ada programı

Basit Türler

```c
#include <stdio.h>
enum Color {Red, Green, Blue};
enum NewColor {NewRed = 3, NewGreen = 2, NewBlue = 2};
main(){
    enum Color x = Green; // x aslında 1
    enum NewColor y = NewBlue; // y aslında 2
    x++; // x şimdi 2 veya Blue
    y--; // y şimdi 1, bir enum bile değil
    printf("%d\n", x); // 2 yazar
    printf("%d\n", y); // 1 yazar
    return 0;
}
```

Şekil: Numaralandırılmış türü gösteren bir C programı

Basit Türler

- **Alt aralık türleri(Subrange types)**: en küçük ve en büyük öğeleri vererek belirtilen basit türlerin bitişik alt kümeleri
  - Örnek: `type Digit_Type is range 0..9;`
- **Sıra türleri(Ordinal Types)**: değerler kümesinde ayrı bir sıra sergileyen türler
  - Tüm sayısal tam sayı türleri sıralı türlerdir
  - Her zaman karşılaştırma operatörlerine sahiptir
  - Genellikle ardıl ve öncel işlemlerine sahiptir
- Gerçek sayılar sıralı değildir; ardıl ve öncel işlemleri yok
- Tahsis şemaları genellikle verimlilik için temeldeki donanıma bağlıdır
- IEEE 754 standardı, standart gösterimleri tanımlamaya çalışır

Tür Oluşturucular

- Veri türleri küme olduğundan, mevcut olanlardan yeni türler oluşturmak için küme işlemleri kullanılabilir.
- Kullanılabilen küme işlemleri arasında kartezyen çarpım, birleşim, kuvvet kümesi, fonksiyon kümesi ve alt küme bulunur
  - Bu küme işlemlerine tür oluşturucular denir
- Örnek: alt aralık türü, alt küme yapısı kullanılarak oluşturulur
- Matematiksel küme yapılarına karşılık gelmeyen tür oluşturucular ve tür oluşturuculara karşılık gelmeyen bazı küme işlemleri vardır.

Kartezyen Çarpım

- U ve V kümeleri verildiğinde, Kartezyen çarpım(Cartesian product) (veya çapraz çarpım(cross product)) U ve V'den tüm sıralı eleman çiftlerinden oluşur:

  $U \times V = \{(u, v)\} \mid u,\ U$ kümesindedir ve $v,\ V$ kümesindedir
- Birçok dilde, Kartezyen çarpım tür oluşturucusu, **kayıt(record)** veya **yapı inşası(structure construction)** olarak mevcuttur.

Kartezyen Çarpım

- Örnek: C'de, bu struct bildirimi $int \times char \times double$ türünün Kartezyen çarpımını oluşturur

```
struct IntCharReal {
    int i;
    char c;
    double r;
};
```

Kartezyen Çarpım

- Kartezyen çarpım ile kayıt yapısı arasındaki fark:
  - Bir kayıt yapısında, bileşenlerin adları vardır
  - Kartezyen çarpımda, konum olarak belirtilirler
- Çoğu dil, bileşen adlarının bir kayıt yapısı tarafından tanımlanan türün parçası olduğunu düşünür.
- **Demet(Tuple)**: Temelde kartezyen çarpım ile aynı, ML'de daha saf bir kayıt yapısı biçimi

```
type IntCharReal = int * char * real;
```

Kartezyen Çarpım

- **Sınıf(Class)**: nesne yönelimli dillerde bulunan bir veri türü
  - Üye fonksiyonları veya metotları içerir
  - Veriler üzerinde işlem yapan fonksiyonları içeren veri türünün ikinci tanımına daha yakın

Birleşim(Union)

- **İki türün birleşimi(Union of two types)**: değer kümelerinin teorik küme birleşimi alınarak oluşturulur
- İki türü vardır
  - **Ayrım gözeten birleşim(Discriminated union)**: Birleşime, öğelerinin türünü ayırt etmek için bir etiket veya ayırıcı eklenir
  - **Ayrım gözetmeyen birleşim(Undiscriminated unions)**: etiketlerden yoksun; Herhangi bir değerin türü hakkında varsayımlar yapılmalıdır
- Ayrım gözetmeyen birleşimlere sahip bir dilin güvenli olmayan bir tür sistemi vardır

Birleşim

- C ve C ++ 'da, birleşim türü yapıcısı ayrım gözetmeyen birleşimler oluşturur Örnek:

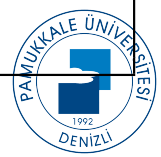
```
union IntOrReal {
    int i;
    double r;
};
```

- `x`, `IntOrReal` türünde bir değişkense, `x.i` bir `int` olarak yorumlanır ve `x.r` bir `double` olarak yorumlanır

Birleşim

- Ada, **varyant kaydı(variant record)** adı verilen tamamen güvenli bir birleşim mekanizmasına sahiptir.

```
type Disc is (IsInt, IsReal);
type IntOrReal (which: Disc) is
record
  case which is
    when IsInt => i: integer;
    when IsReal => r: float;
  end case;
end record;
--...
x: IntOrReal := (IsReal, 2.3);
```



Birleşim

- ML'de "veya" için dikey çubuk ile numaralandırılmış veriler bildirilir

```
datatype IntOrReal = IsInt of int | IsReal of real;
```

- Daha sonra desen eşleştirme(pattern matching) kullanılır:

```
fun printInt x = (print("int: "); print(Int.toString x);
↪  print("\n"));
fun printReal x = (print("real: "); print(Real.toString x);
↪  print("\n"));
fun printIntOrReal x =
  case x of
    IsInt(i) => printInt i |
    IsReal(r) => printReal r;
```

Birleşim

- ML'deki `IsInt` ve `IsReal` etiketleri, bir birleşim içinde her türden veri oluşturdukları için veri yapıcıları olarak adlandırılır.
- Birleşim, aynı anda farklı veri öğelerine ihtiyaç duyulmadığında yapılar için bellek tahsis gereksinimlerini azaltmada kullanışlıdır.
- Nesne yönelimli dillerde birleşimlere ihtiyaç yoktur
  - Örtüşmeyen farklı veri gereksinimlerini temsil etmek için mirası kullanılır

Alt Küme

- Matematikte bir alt küme, öğelerini ayırt etmek için bir kural verilerek belirtilir
- Bilinen türlerin alt kümeleri olarak yeni türler oluşturmak için programlama dillerinde benzer kurallar verilebilir.
- Ada'nın bir alt tür mekanizması vardır:

```
subtype IntDigit_Type is integer range 0..9
```

- Kayıtların değişken(varyant) bölümleri alt tür kullanılarak düzeltilebilir

```
subtype IRInt is IntOrReal(IsInt);
subtype IRReal is IntOrReal(IsReal);
```

Alt Küme

- Bu tür alt küme türleri, işlemleri üst türlerinden **devralır(inherit)**
  - Çoğu dil, programcının hangi işlemlerin miras alınacağını ve hangilerinin devralınmadığını belirlemesine izin vermez.
- Nesne yönelimli dillerdeki kalıtım, bir alt tip mekanizması olarak da görülebilir.
  - Hangi işlemlerin miras alınacağı üzerinde büyük ölçüde daha fazla kontrol ile

Diziler ve Fonksiyonlar

- Tüm fonksiyonlar kümesi $f : U \to V$, iki şekilde yeni bir türe yol açabilir:
  - **Dizi türü(Array type)**
  - **Fonksiyon türü(Function type)**
- U bir sıra(ordinal) türüyse, f fonksiyonu, indeks türü U ve bileşen türü V olan bir dizi olarak düşünülebilir.
  - Eğer i, U içindeyse, $f(i)$ dizinin $i'nci$ bileşenidir
  - Tam fonksiyon, değerlerinin dizisi veya demetleriyle temsil edilebilir $(f(low), \ldots, f(high))$

Diziler ve Fonksiyonlar

- Diziler bazen sekans(sequence) türleri olarak adlandırılır
- Tipik olarak, dizi türleri boyutlarla veya boyutsuz olarak tanımlanabilir
  - Bir dizi türünün bir değişkenini tanımlamak için, diziler normalde statik olarak tahsis edildiğinden, genellikle çeviri zamanında boyutu belirtmek gerekir.
- C'de, bir dizinin boyutu hesaplanmış bir sabit değil, değişmez olmalıdır.
- C veya C++'da bir dizi boyutu dinamik olarak tanımlanamaz

Diziler ve Fonksiyonlar

- C, boyut belirtilmemiş dizilerin işlevler için parametre olmasına izin verir (bunlar esasen işaretçilerdir), ancak boyut sağlanmalıdır
  - Dizinin boyutu, C veya C++'da dizinin parçası değil

```
int array_max(int a[], int size){
    int temp, i;
    assert(size > 0);
    temp = a[0];
    for (i = 1; i < size; i++)
        if(a[i] > temp) temp = a[i];
    return temp;
}
```

Diziler ve Fonksiyonlar

- Java'da diziler her zaman dinamik olarak(yığın/heap) ayrılır ve boyut dinamik olarak belirtilebilir(ancak değiştirilemez)
  - Bir dizi tahsis edildiğinde boyutu length özelliğinde saklanır

Diziler ve Fonksiyonlar

```
import java.io.*;
import java.util.Scanner;
public class ArrayTest {
    static private int array_max(int[] a){//[] konumuna dikkat
        int temp;
        temp = a[0];
        //uzunluk a'nin bir parçası
        for(int i = 1; i < a.length; i++)
            if(a[i] > temp) temp = a[i];
        return temp;
    }
    public static void main(String[] args){
        System.out.print("Pozitif tamsayı girin: ");
        Scanner reader = new Scanner(System.in);
        int size = reader.nextInt();
        int[] a = new int[size]; // Dinamik bellek tahsisi
        for(int i = 0; i< a.length; i++) a[i] = i;
        System.out.println(array_max(a));
    }
}
```

Diziler ve Fonksiyonlar

- Ada, boyut olmadan bildirilen dizi türlerine izin verir, **kısıtlanmamış diziler(unconstrained arrays)** olarak adlandırılır, ancak dizi değişkenleri bildirildiğinde bir boyut gerektirir
- Çok boyutlu diziler de mümkündür
- Diziler belki de en yaygın kullanılan tür yapıcıdır
- Uygulama son derece verimlidir
  - Alan belleğe sıralı olarak tahsis edilir
  - İndeksleme, başlangıç adresinden bir öteleme(ofset) hesaplaması ile gerçekleştirilir.

Diziler ve Fonksiyonlar

- Çok boyutlu diziler için, tahsis şemasında ilk olarak hangi dizinin kullanılacağına karar vermelidir
  - **Satır ana biçimi(Row-major form)**: ilk satırın tüm değerleri önce, sonra ikinci satırın tüm değerleri vb. Tahsis edilir.
  - **Sütun-ana biçim(Column-major form)**: ilk sütunun tüm değerleri, sonra ikinci sütunun tüm değerleri vb. Ayrılır.
- Fonksiyonel diller genellikle bir dizi türü sağlamaz; çoğu **listeyi(list)** bir dizi yerine kullanır
  - Scheme'in bir **vektör(vector)** türü vardır

Diziler ve Fonksiyonlar

- Bazı dillerde genel fonksiyon ve prosedür türleri oluşturulabilir
- Örnek: C'de tam sayılardan tam sayılara bir işlev türü tanımlamak:
  `typedef int (*IntFunction) (int);`
- Bu tür değişkenler veya parametreler için kullanılır:

```c
typedef int (*IntFunction)(int);
int square(int x){ return x*x; }
IntFunction f = square;
int evaluate(IntFunction g, int value){
    return g(value);
}
//...
printf("%d\n", evaluate(f, 3)); // 9 yazar
```

Diziler ve Fonksiyonlar

- ML'de bir fonksiyon türü tanımlanabilir:

```
type IntFunction = int -> int;
```

- Benzer şekilde kullanın:

```
fun square (x: int) = x * x;
val f = square;
fun evaluate(g: IntFunction, value: int) = g value;
(* ... *)
evaluate(f, 3); (* 9 döndürür *)
```

İşaretçiler ve Özyinelemeli Türler

- **Referans** veya **işaretçi** yapıcısı(Reference or pointer constructor): belirli bir türe başvuran tüm adresler kümesini oluşturur
  - Bir küme işlemine karşılık gelmiyor
- C örneği: `typedef int* InPtr;`
  - Tam sayıların depolandığı tüm adreslerin türünü oluşturur
- İşaretçiler, otomatik bellek yönetimi gerçekleştiren dillerde örtüktür(implicit)
  - Java'da, tüm nesneler, açıkça tahsis edilen(`new` operatörü kullanılarak) ancak çöp toplama tarafından otomatik olarak serbest bırakılan örtük olarak işaretçilerdir.

İşaretçiler ve Özyinelemeli Türler

- **Referans(Reference)**: Sistemin kontrolü altındaki, değer olarak kullanılamayan veya herhangi bir şekilde çalıştırılamayan bir nesnenin adresi (kopyalama hariç)
- **İşaretçi(Pointer)**: bir değer olarak kullanılabilir ve programlama tarafından değiştirilebilir
- C++'daki referanslar bir postfix & operatörü tarafından oluşturulur.
- **Özyinelemeli tür(Recursive type)**: bildiriminde kendini kullanan bir tür

İşaretçiler ve Özyinelemeli Türler

- Özyinelemeli türler, veri yapıları ve algoritmalarda önemlidir
  - Önceden boyutu ve yapısı bilinmeyen verileri temsil eder ve hesaplama ilerledikçe değişebilir
  - Örnekler: listeler ve ikili ağaçlar
- Şu C benzeri karakter listeleri bildirimini düşünün:

```
struct CharList{
    char data;
    struct CharList next; // C'de geçerli değil
}
```

İşaretçiler ve Özyinelemeli Türler

- C, her veri türünün çeviri sırasında belirlenen sabit bir maksimum boyuta sahip olmasını gerektirir
  - Bu sorunun üstesinden gelmek için manuel dinamik ayırmaya izin vermek için işaretçi kullanılmalıdır

```
struct CharListNode{
    char data;
    struct CharListNode* next; // şu an geçerli
};
typedef struct CharListNode* CharList;
```

- `CharListNode`'daki her bir öğenin artık sabit bir boyutu vardır ve bunlar rastgele boyutta bir liste oluşturmak için bir araya dizilebilirler.

Veri Türleri ve Ortam

- İşaretçi türleri, yinelemeli türler ve genel fonksiyon türleri, dinamik olarak tahsis edilecek alan gerektirir
  - Otomatik ayırma ve serbest bırakma (çöp toplama) ile tamamen dinamik ortamlar gerektirir
  - Fonksiyonel dillerde ve daha dinamik nesne yönelimli dillerde bulunur
- Daha geleneksel diller (C ++ ve Ada) bu türleri bir yığın (programlama kontrolü altında dinamik bir alan) yeterli olacak şekilde kısıtlar.
- Ortam sorunları Bölüm 10'da tam olarak tartışılacaktır.

Örnek Dillerdeki Tür İsimlendirmesi

- Çeşitli dil tanımları, benzer şeyleri tanımlamak için farklı ve kafa karıştırıcı terminoloji kullanır.
- Bu bölüm, üç dil arasındaki farkların kısa bir açıklamasını verir: C, Java ve Ada

C

- Basit veri türlerine temel türler denir, örneğin:
  - `void` türü
  - Sayısal türler:
    - Sıralı olan **integral** türleri (12 olası tür)
    - **Kayan(floating)** türler (3 olası tür)
- İntegral türleri işaretli veya işaretsiz olabilir
- **Türetilmiş türler(Derived types)**: tür oluşturucuları kullanılarak oluşturulur

C

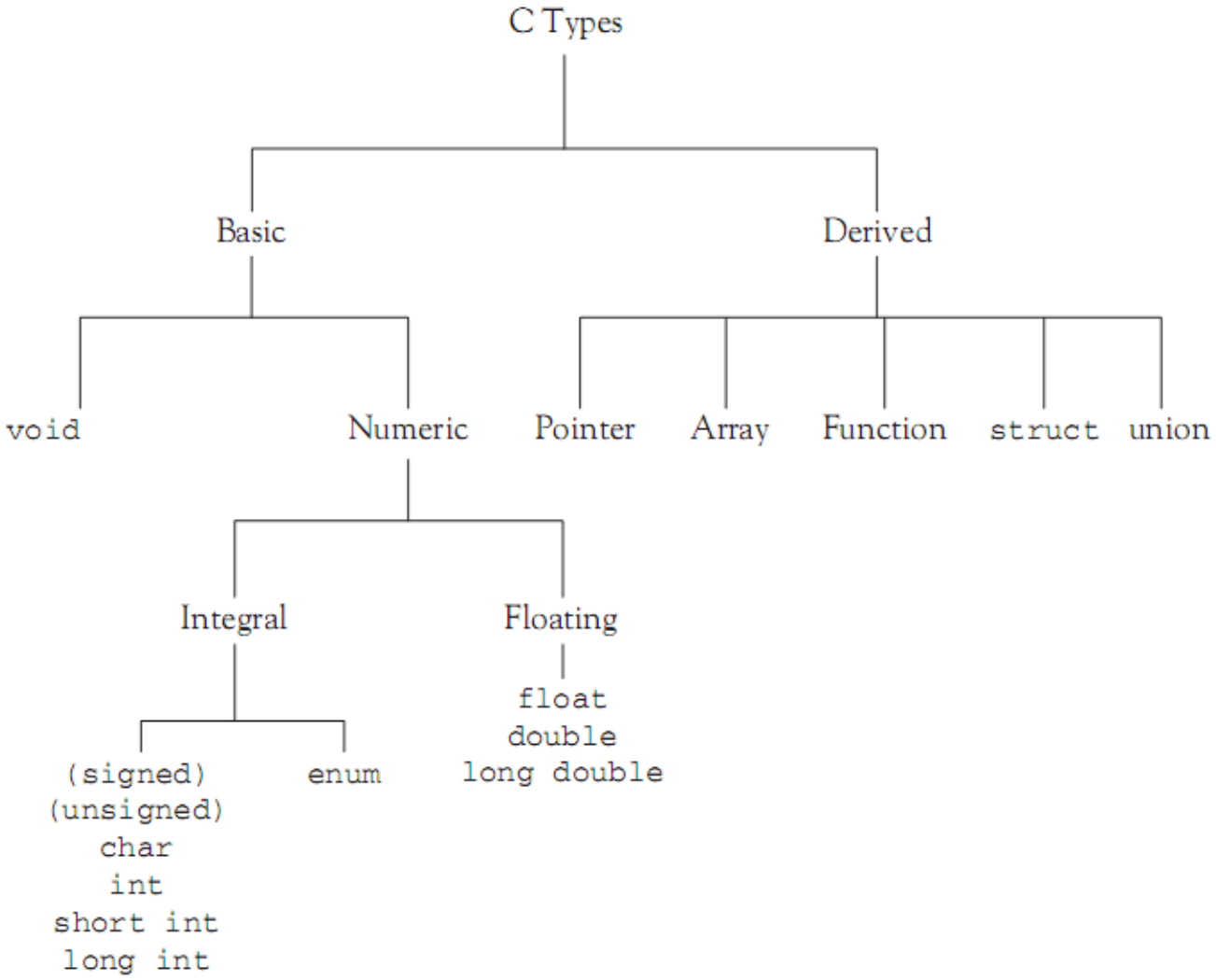


Figure 8.5 The type structure of C

Java

- Basit türlere ilkel türler(primitive types) denir, örneğin:
  - Boole (sayısal veya sıralı değil)
  - Aşağıdakiler dahil sayısal:
    - İntegral (sıra)
    - Kayan nokta(Floating point)
- **Referans türleri(Reference types)**: tür oluşturucuları kullanılarak oluşturulmuştur
  - Dizi(Array)
  - Sınıf(Class)
  - Arayüz(Interface)

Java

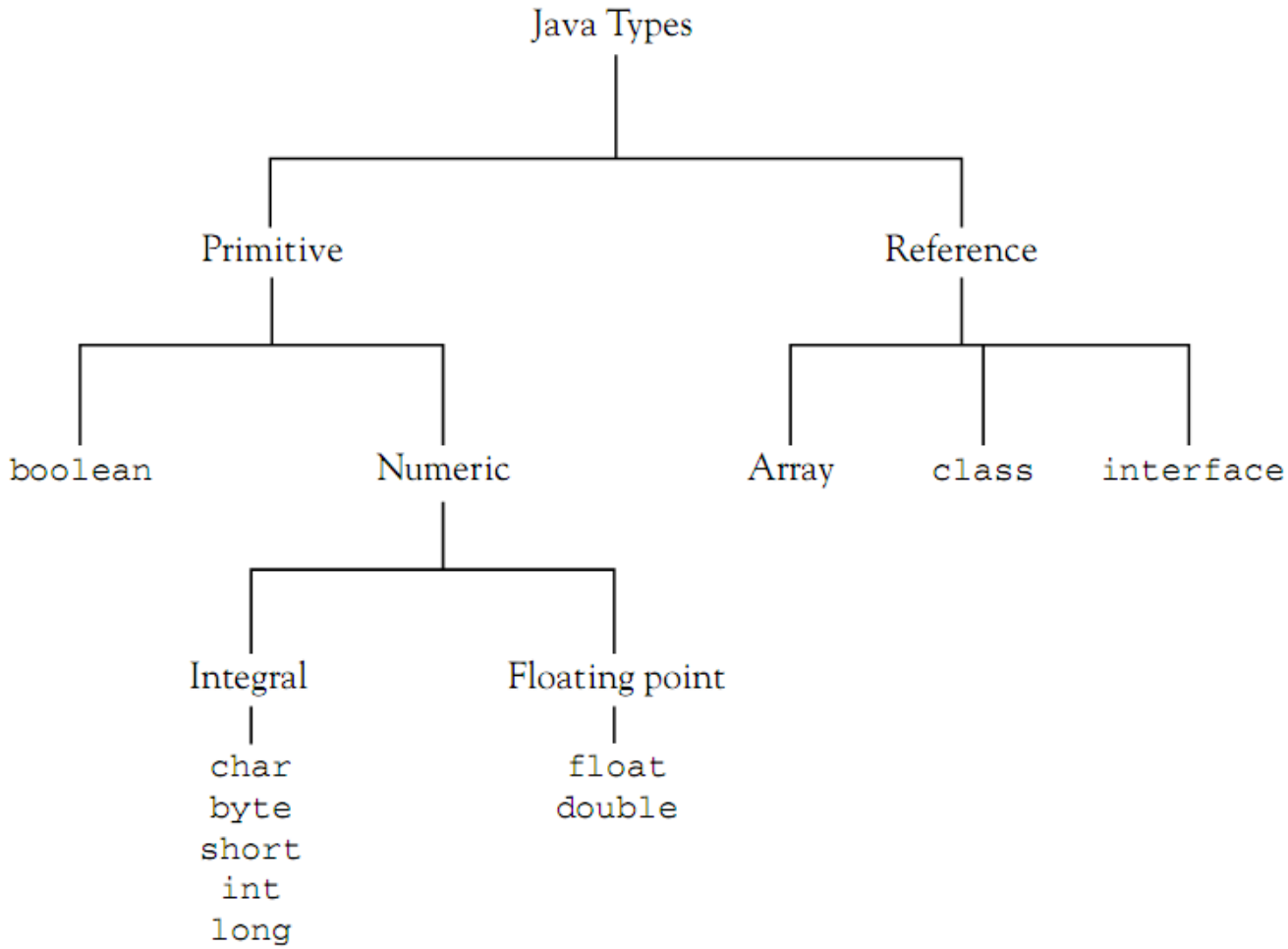


Figure 8.6 The type structure of Java

Ada

- Ada zengin türlere sahiptir
  - Basit türlere **skaler türler(scalar types)** denir
  - Sıralı türlere **ayrık türler(discrete types)** denir
  - **Sayısal türler(Numeric types)**, **gerçek(real)** ve **tam sayı(integer)** türlerini içerir
  - İşaretçi türlerine `access` türleri denir
  - Dizi ve kayıt türlerine **bileşik türler(composite types)** denir

Ada

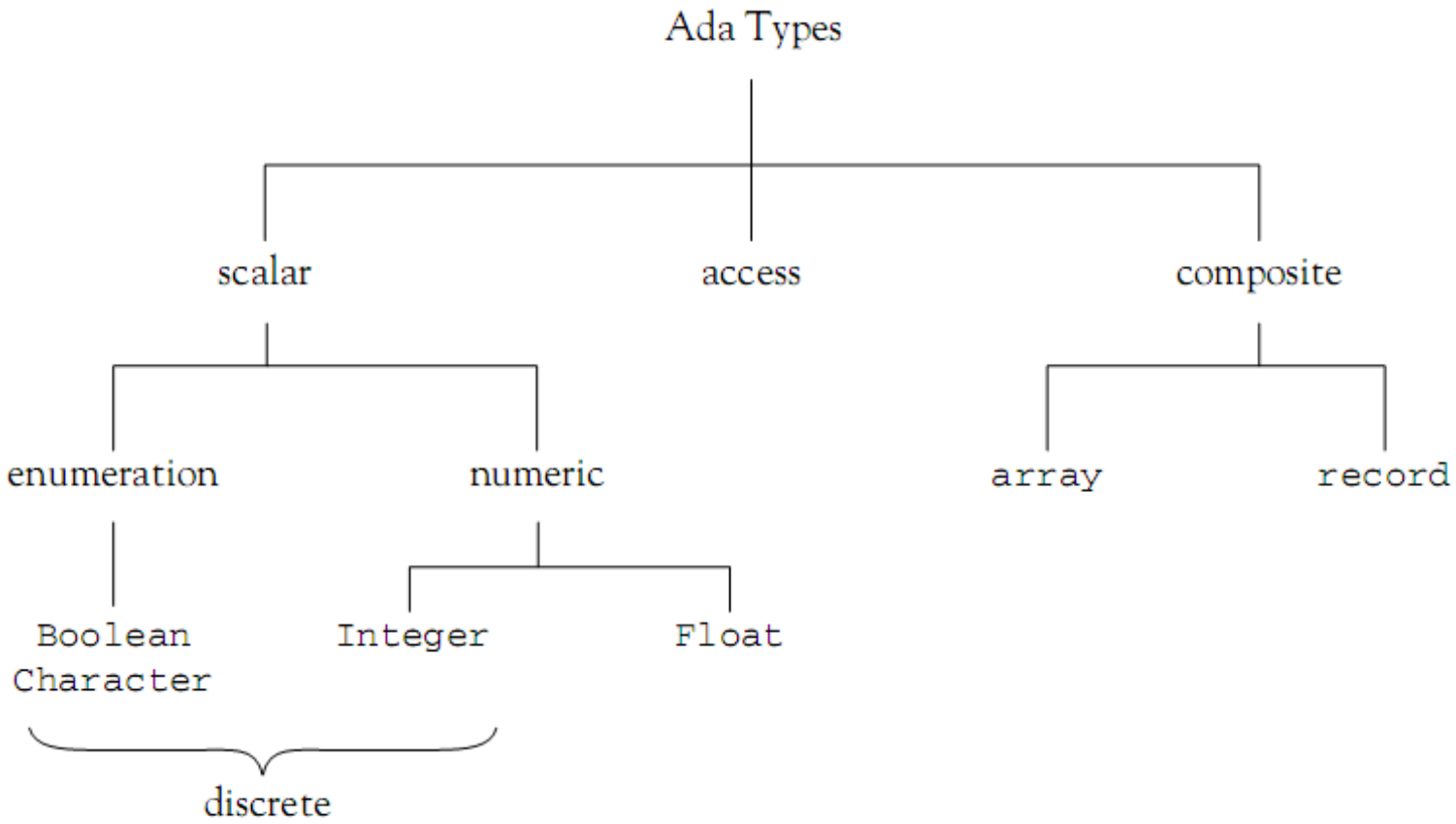


Figure 8.7 The type structure of Ada (somewhat simplified)

Tür Eşdeğerliği

- **Tür eşdeğerliği(Type equivalence)**: iki tür ne zaman aynıdır?
- Değer kümelerini kümeler halinde karşılaştırabilir
  - Aynı değerleri içeriyorlarsa aynıdır
- **Yapısal eşdeğerlik(Structural equivalence)**: aynı yapıya sahiplerse iki veri türü aynıdır
  - Aynı basit türlerdeki aynı tür oluşturucular kullanılarak aynı şekilde oluşturulmuştur
  - Bu, programlama dillerinde tür eşdeğerliğinin temel biçimlerinden biridir

Tür Eşdeğerliği

- Rec1 ve Rec2 yapısal olarak eşdeğerdir
- Rec1 ve Rec3 yapısal olarak eşdeğer değildir(`char` ve `int` alanları yer değiştirmiştir)

```
struct Rec1{
    char x;
    int y;
    char z[10];
};
struct Rec2{
    char x;
    int y;
    char z[10];
};
struct Rec3{
    int y;
    char x;
    char z[10];
};
```

Tür Eşdeğerliği

- Yapısal eşdeğerliğin uygulanması nispeten kolaydır (özyinelemeli türler hariç)
  - Hata kontrolü ve depolama tahsisi gerçekleştirmek için gereken tüm bilgileri sağlar
- Yapısal eşdeğerliği kontrol etmek için, bir çevirmen türleri ağaç olarak temsil edebilir ve alt ağaçlarda eşdeğerliği yinelemeli olarak kontrol edebilir.
- Bir tür oluşturucunun uygulaması altındaki bir türe ne kadar bilgi dahil edildiğine ilişkin sorular hala ortaya çıkmaktadır.

Tür Eşdeğerliği

- Örnek: A1 ve A2 yapısal olarak eşdeğer midir?

```
typedef int A1[10];
typedef int A2[20];
```

  - Evet, dizin kümesinin boyutu bir dizi türünün parçası değilse
  - Aksi takdirde hayır

- Yapıların üye isimleriyle ilgili benzer sorular ortaya çıkmaktadır

Tür Eşdeğerliği

- Örnek: Bu iki yapı yapısal olarak eşdeğer mi?

```
struct RecA {
        char x;
        int y;
};
```

```
struct RecB {
        char a;
        int b;
};
```

- Yapılar sadece kartezyen çarpım olarak kabul edilirse, evet
- Farklı yapıdaki değişkenlerin üye verilerine erişmek için farklı isimler kullanması gerektiğinden, bunlar tipik olarak eşdeğer olarak kabul edilmezler.

Tür Eşdeğerliği

- Bildirimlerdeki **tür isimleri(type names)** açıkça verilebilir veya verilmeyebilir
  - C'de, değişken bildirimleri **anonim türleri(anonymous types)** kullanabilir
  - İsimler ayrıca `struct` ve `union` veya `typedef` kullanılarak da verilebilir.
- Tür adları mevcut olduğunda yapısal eşdeğerlik, her adın bildirimindeki ilişkili tür ifadesiyle değiştirilmesiyle yapılabilir (özyinelemeli türler hariç)

Tür Eşdeğerliği

- Örnek: C kodunda
  - A değişkeninin iki adı vardır: `struct RecA` ve `RecA` (`typedef` tarafından verilir)
  - Değişken b yalnızca `RecB` adına sahiptir (`struct` adı boş bırakılmıştır)
  - Değişken c'nin tür adı yoktur (yalnızca programcı tarafından kullanılamayan dahili bir ad)

```
struct RecA{
    char x;
    int y;
} a;

typedef struct RecA RecA;

typedef struct{
    char x;
    int y;
} RecB;

RecB b;

struct{
    char x;
    int y;
} c;
```

Tür Eşdeğerliği

- Adları türlerle değiştirerek yapısal eşdeğerlik, özyinelemeli türlere uygulandığında bir tür denetleyicide sonsuz döngülere yol açabilir

```
typedef struct CharListNode* CharList;
typedef struct CharListNode2* CharList2;

struct CharListNode{
    char data;
    CharList next;
};

struct CharListNode2{
    char data;
    CharList2 next;
};
```

Tür Eşdeğerliği

- Örnek: C kodunda:
  - a, b, c ve d yapısal olarak eşdeğerdir
  - a ve c ismen eşdeğerdir ve b veya d'ye ismen eşdeğer değildir
  - b ve d diğer herhangi bir değişkene ismen eşdeğer değildir

```
struct RecA{
    char x;
    int y;
};

typedef struct RecA RecA;

struct RecA a;
RecA b;
struct RecA c;
struct{
    char x;
    int y;
} d;
```

Tür Eşdeğerliği

- Ada çok saf bir isim eşdeğerliği uyguluyor
  - Neredeyse tüm durumlarda değişken ve işlev bildirimlerinde tür adları gerektirir
- C, ad ve yapısal eşdeğerlik arasında kalan bir tür eşdeğerliği kullanır:
  - `struct` ve `union` için isim eşdeğerliği
  - Diğer her şey için yapısal eşdeğerlik
- Pascal, C'ye benzer, ancak hemen hemen tüm tip kurucular yeni, eşdeğer olmayan türlere yol açar.

Tür Eşdeğerliği

- Java'nın yaklaşımı basittir:
  - `typedef` yok
  - `class` ve `interface` bildirimleri örtük olarak yeni tür adları oluşturur ve bu türler için isim eşdeğerliği kullanılır
  - Diziler, temel tür eşdeğerliğini oluşturmak için özel kurallarla yapısal eşdeğerlik kullanır.

Tür Denetimi

- **Tür denetimi(Type checking)**: bir çevirmenin tüm yapıların tutarlı olduğunu doğruladığı süreç
  - İfadelere(expressions) ve komutlara(statements) bir tür eşdeğerlik algoritması uygular
  - Bağlama uyması için tür eşdeğerlik algoritmasının kullanımını değiştirebilir

  İki tür tür denetimi:
  - **Dinamik(Dynamic)**: tür bilgileri çalışma zamanında ele alınır ve kontrol edilir
  - **Statik(Static)**: türler program metninden belirlenir ve çevirmen tarafından kontrol edilir

Tür Denetimi

- Güçlü türlü(Strongly typed) bir dilde, tüm tür hataları çalışma zamanından önce yakalanmalıdır
  - Bu diller statik olarak yazılmalıdır
  - Tür hataları, yürütmeyi engelleyen derleme hata mesajları olarak rapor edilir
- Bir dil tanımı, dinamik veya statik yazmanın kullanılıp kullanılmadığını belirtemez

Tür Denetimi

- Örnek 1:
  - C derleyicileri, çeviri sırasında statik tür denetimi uygular, ancak birçok tutarsızlık derleme hatalarına neden olmadığından C güçlü türlü(strongly typed) değildir.
  - C ++, güçlü tür denetimi ekler, ancak esas olarak hataları yerine derleyici uyarıları biçimindedir, bunlar yürütmeyi engellemez

Tür Denetimi

- Örnek 2:
  - Şema dinamik türlü bir dildir, ancak türler titizlikle kontrol edilir
  - Tür hataları programın sonlandırılmasına neden olur
  - Bildirimlerde tür yoktur ve açık tür adları yoktur
  - Değişkenlerin önceden bildirilmiş türleri yoktur, ancak sahip oldukları değerin türünü alırlar

Tür Denetimi

- Örnek 3:
  - Ada güçlü türlü(strongly typed) bir dildir
  - Tüm tür hataları derleme hata mesajlarına neden olur
  - Dizi indislerindeki aralık hataları gibi bazı hatalar, yürütmeden önce yakalanamaz.
  - Bu tür hatalar, program tarafından ele alınmadığı takdirde programın sonlandırılmasına neden olacak istisnalara neden olur.

Tür Denetimi

- **Tür çıkarımı(Type inference)**: ifade türleri, alt ifadelerinin türlerinden çıkarılır
  - Tür denetiminin önemli bir parçasıdır
- Tür denetim kuralları ve tür çıkarım kuralları genellikle birbirine karıştırılır
  - Ayrıca tür eşdeğerlik algoritmasıyla yakın bir etkileşime sahiptirler.
- Tür çıkarımı ve doğruluk kuralları, bir dilin semantiğinin en karmaşık kısımlarından biridir.

Tür Uyumluluğu

- Belirli şekillerde birleştirildiğinde doğru kabul edilebilecek iki farklı türe uyumlu denir
  - Ada'da, aynı temel türün herhangi iki alt aralığı uyumludur
  - C ve Java'da, tüm sayısal türler uyumludur (ve dönüştürmeler gerçekleştirilir)
- **Atama uyumluluğu(Assignment compatibility)**: bir atama ifadesinin sol ve sağ tarafları aynı türde olduklarında uyumludur
  - Sol tarafın bir **l-değeri(l-value)** ve sağ tarafın bir **r-değeri(r-value)** olması gerektiğini yok sayar

Tür Uyumluluğu

- Atama uyumluluğu, her iki tarafın aynı türe sahip olmadığı durumları içerebilir
- Java'da, e, değeri bilgi kaybı olmadan x türüne dönüştürülebilen sayısal bir tür olduğunda x = e geçerlidir

Örtülü Türler

- **Örtülü türler(Implicit types)**: bildirimde açıkça belirtilmeyen türler
- Tür, çevirmen tarafından bağlam bilgisinden veya standart kurallardan çıkarılmalıdır.
- C'de, hiçbir tür belirtilmezse değişkenler örtülü olarak tamsayıdır ve dönüş türü verilmemişse işlevler örtülü olarak bir tamsayı değeri döndürür.
- Pascal'da, adlandırılmış sabitler temsil ettikleri değişmez değerlere göre dolaylı olarak yazılır.
- Değişmezler, örtülü olarak yazılmış varlıkların başlıca örnekleridir

Örtüşen Türler ve Çok Türlü Değerler

- Ortak değerlere sahip iki tür çakışabilir
- Türlerin ayrık olması tercih edilse de, bu, nesne yönelimli dillerde kalıtım yoluyla alt türler oluşturma yeteneğini ortadan kaldıracaktır.
- C'de, `unsigned int` ve `int` gibi türler çakışır
- C'de değişmez 0(literal 0), her integral türü için, her işaretçi türünün değeridir ve boş işaretçiyi(null pointer) temsil eder
- Java'da, değişmez değer `null`, her referans türünün değeridir

Paylaşılan İşlemler

- Her tür, genellikle örtülü olarak bir dizi işlemle ilişkilendirilir
- İşlemler birkaç tür arasında paylaşılabilir veya farklı olabilecek diğer işlemlerle aynı ada sahip olabilir.
- Örnek: + operatörü gerçek sayı toplama, tamsayı toplama veya küme birleşimi(set union) olabilir
- **Aşırı yüklenmiş işlem(Overloaded operator)**: farklı işlemler için aynı isim kullanılır
  - Çevirmen, işlenenlerin(operand) türlerine göre hangi işlemin kastedildiğine karar vermelidir

Tür Dönüşümü

- **Tür dönüşümü(Type conversion)**: bir türden diğerine dönüştürme
  - Otomatik olarak gerçekleşmesi için tür sistemine entegre edilebilir
- **Örtülü dönüştürme(Implicit conversion)** (veya **zorlama(coercion)**): çevirmen tarafından eklenen
- **Genişleten dönüştürme(Widening conversion)**: hedef veri türü, dönüştürülen tüm verileri veri kaybı olmadan tutabilir
- **Daraltan dönüştürme(Narrowing conversion)**: dönüştürme, veri kaybını içerebilir

Tür Dönüşümü

- Örtülü dönüştürme:
  - Hataların yakalanmaması için tür denetimini zayıflatabilir
  - Dönüşüm programcının beklediğinden farklı bir şekilde yapılırsa beklenmeyen davranışlara neden olabilir
- **Açık dönüştürme(Explicit conversion)** (veya **cast**): dönüştürme yönergeleri koda yazılır
  - Dönüşümler kodda belgelenir
  - Daha az beklenmedik davranış olasılığı
  - Çevirmenin aşırı yüklemeyi çözmesini kolaylaştırır

Tür Dönüşümü

- C ++ Örneği:

```
double max (int, double);
double max(double, int);
max(2, 3);
```

  - Birinci veya ikinci parametrede `int` türünden `double` türüne olası örtülü dönüşümler nedeniyle belirsiz

- Java yalnızca aritmetik türler için örtülü dönüşümlerin genişletilmesine izin verir
- C++, daraltma için uyarı mesajları yayar

Tür Dönüşümü

- Açık dönüştürmelerin bir şekilde kısıtlanması gerekir
  - Genellikle basit türlere veya sadece aritmetik türlere
- Yapılandırılmış türler için dönüştürmeye izin veriliyorsa, bunlar bellekte aynı boyutlara sahip olmalıdır
  - Çevirinin belleği farklı bir tür olarak **yeniden yorumlamasına(reinterpret)** izin verir
- Örnek: C'de `malloc` ve `free` fonksiyonları, genel bir işaretçi veya **anonim işaretçi türü(anonymous pointer type)** `void*` kullanılarak bildirilir
- Nesne yönelimli diller, alt türlerden süper türlere ve bazı durumlarda geri dönüşümlere izin verir

Tür Dönüşümü

- Açık dönüştürmeye alternatif, dönüştürmeleri gerçekleştirmek için önceden tanımlanmış veya kütüphane fonksiyonlarını kullanmaktır.
  - Ada, dönüşümlere izin vermek için **öznitelik fonksiyonlarını(attribute functions)** kullanır
  - Java, int'ten String'e dönüştürmek için `toString` ve String'den int'e dönüştürmek için `parseInt` gibi işlevler içerir.
- Ayrım gözetmeyen birleşimler(undiscriminated unions) farklı türden değerleri tutabilir
  - Herhangi bir ayrımcı veya etiket olmadan, bir çevirmen bir türdeki değerleri diğerinden ayırt edemez

Polimorfik Tür Denetimi

- Statik türlü dillerin çoğu, bildirimlerdeki tüm adlar için açık tür bilgilerinin verilmesini gerektirir
- Açık bir bildirim olmaksızın isimlerin türlerini belirlemek mümkündür:
  - Bir adın kullanımları hakkında bilgi toplayabilir ve tüm kullanımlar kümesinden türü çıkarabilir
  - Bazı kullanımlar diğerleriyle uyumsuz olduğu için bir tür hatası bildirebilir
- Bu tür çıkarım ve tür denetimi, **Hindley-Milner tür denetimi** olarak adlandırılır.

Polimorfik Tür Denetimi

- C kodundaki örnek: `a[i] + i`
  - a bir tamsayı dizisi olarak ve i bir tamsayı olarak bildirilmeli, bir tamsayı sonucu verir
- Tür denetleyicisi bu ağaçla başlar:

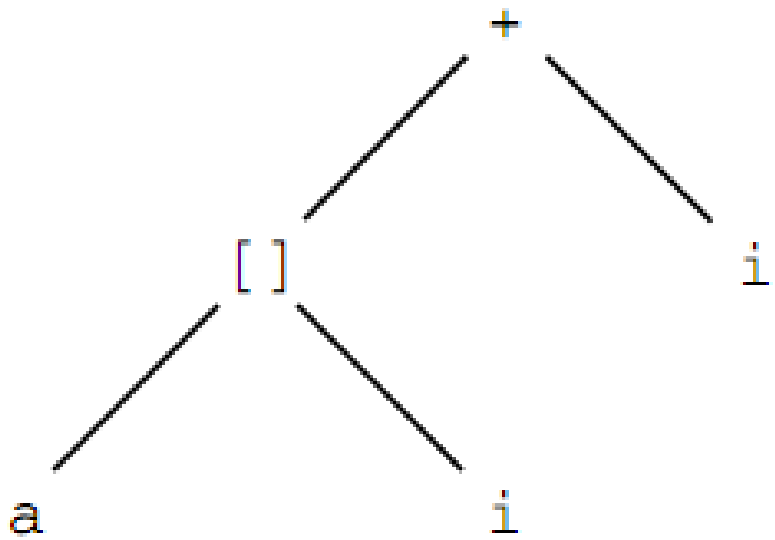


Figure 8.8 Syntax tree for a[i] + i

Polimorfik Tür Denetimi

- İsimlerin türleri (yaprak düğümler) bildirimlerden doldurulur

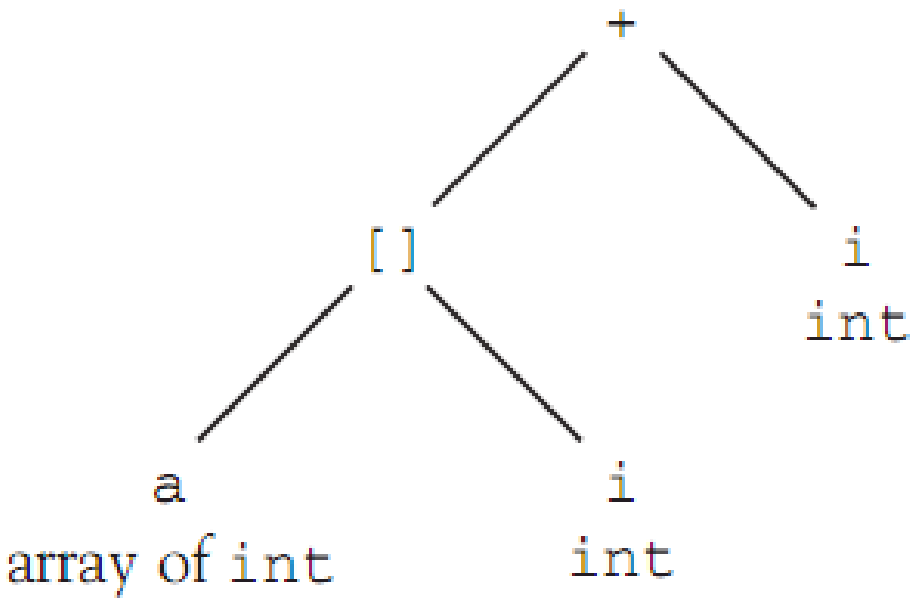


Figure 8.9 Types of the names filled in from the declarations

Polimorfik Tür Denetimi

- Tür denetleyici artık alt simge düğümünü kontrol ediyor ([] etiketli)
  - Sol işlenen bir dizi olmalıdır
  - Sağ işlenen bir int olmalıdır
  - Alt simge düğümünün çıkarım yapılan türü, dizinin bileşen türüdür - int

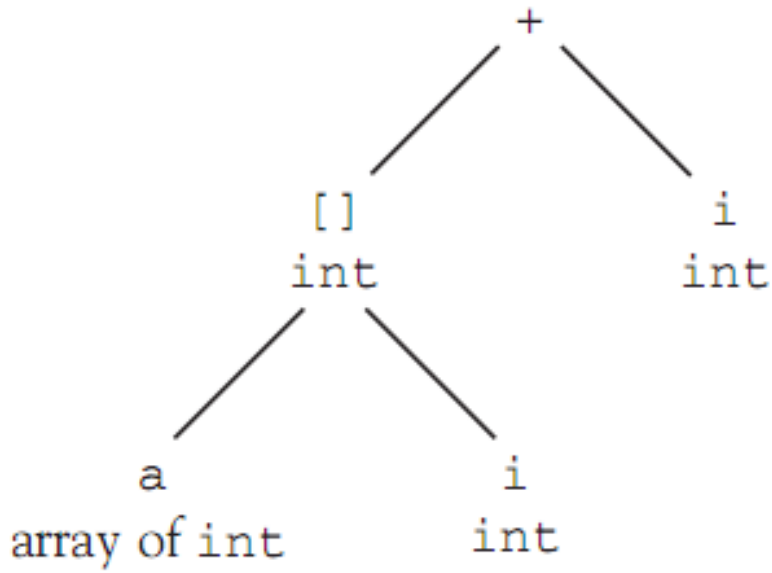


Figure 8.10 Inferring the type of the subscript node

Polimorfik Tür Denetimi

- + düğüm türü kontrol edilir
  - Her iki işlenen de aynı türe sahip olmalıdır
  - Bu tür bir + işlemi içermelidir
  - Sonuç, işlenenlerin türüdür - int

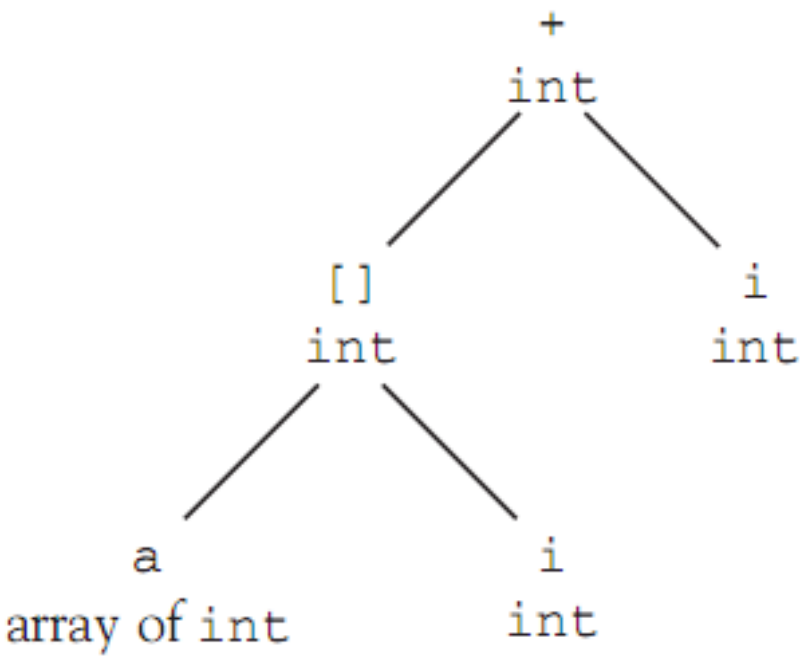


Figure 8.11 Inferring the type of the + node

Polimorfik Tür Denetimi

- Örnek: C kodunda: `a[i] + i`
  - Ya a ve i'nin bildirimleri eksik olsaydı?
- Tür denetleyicisi, ilk önce tür değişkenlerini henüz türü olmayan tüm adlara atar

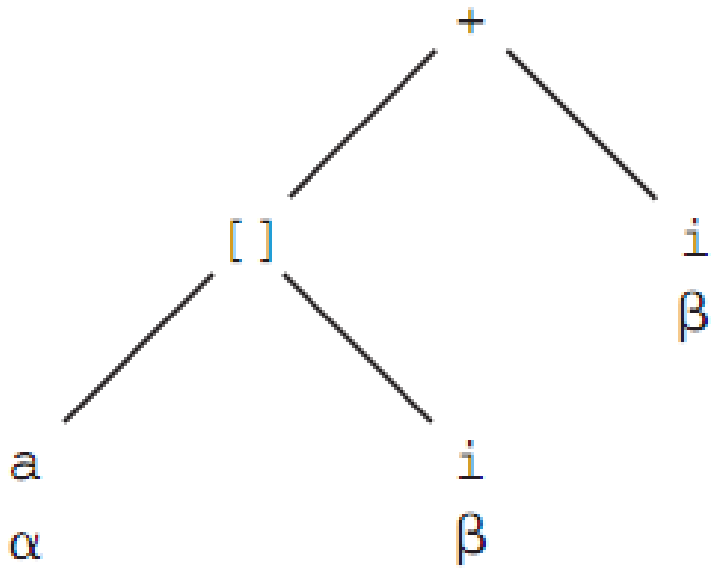


Figure 8.12 Assigning type variable to names without types

Polimorfik Tür Denetimi

- Tip denetleyicisi şimdi alt simge düğümünü denetler
  - A'nın bir dizi olması gerektiğini çıkarır
  - i'nin int olması gerektiğini çıkarır
  - Tüm ağaçta β'yı int ile değiştirir

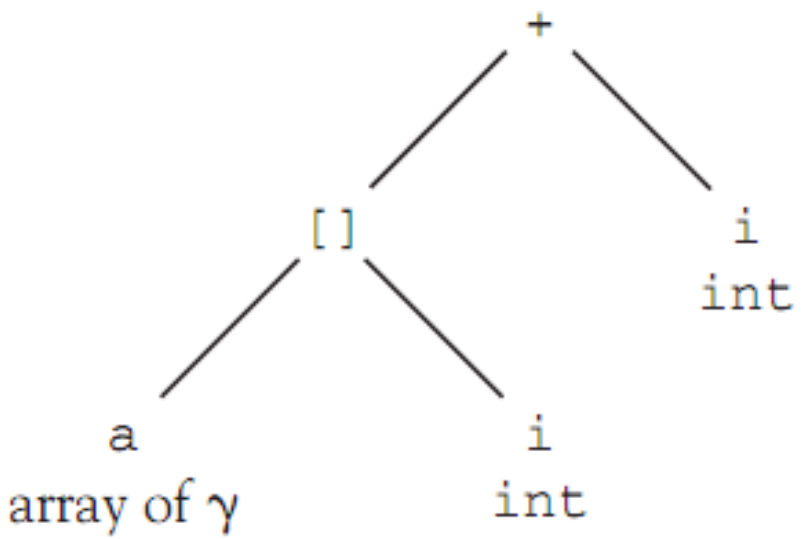


Figure 8.13 Inferring the type of the variable

Polimorfik Tür Denetimi

- Tür denetleyicisi artık alt simge türünün doğru türde olduğu ve γ türüne sahip olduğu sonucuna varır

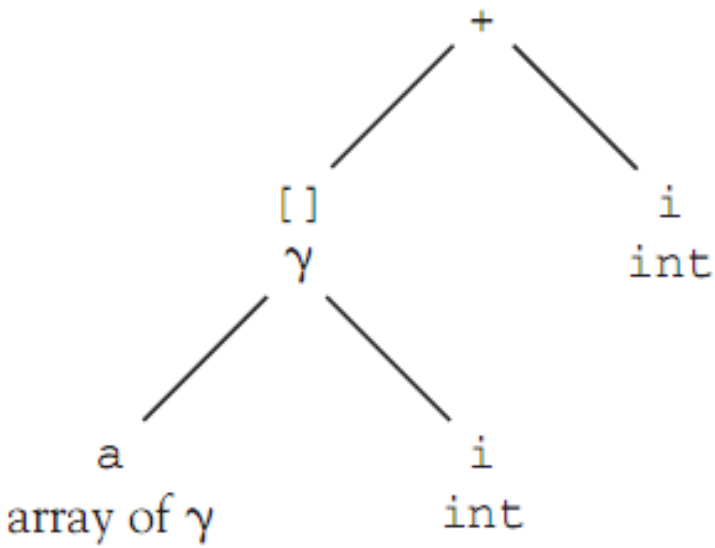


Figure 8.14 Inferring the type of the subscript noc

Polimorfik Tür Denetimi

- $+$ düğüm türü kontrol edilir
  - γ türünün int olması gerektiği sonucuna varır
  - γ her yerde int ile değiştirir
- Bu, Hindley-Milner tip kontrolünün temel işlem şeklidir.

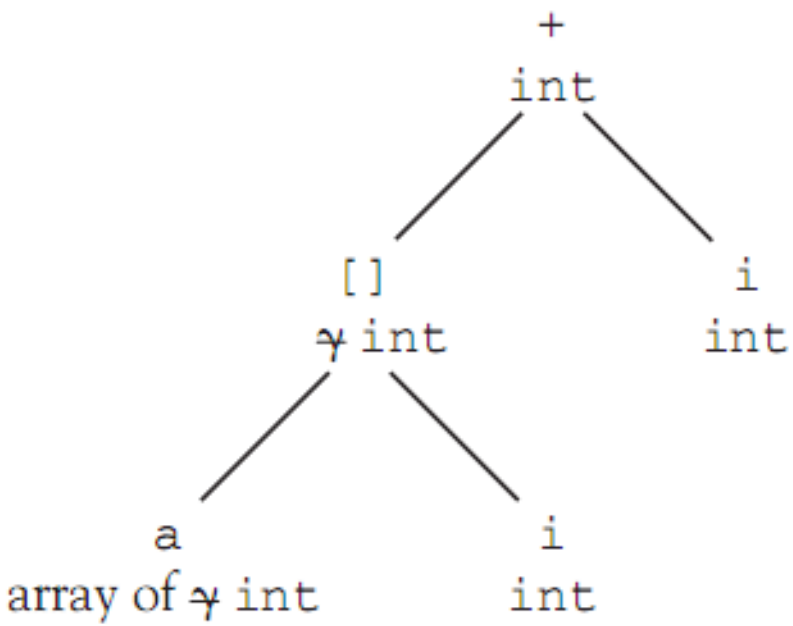


Figure 8.15 Substituting int for the type variable

Polimorfik Tür Denetimi

- Bir tür değişkeni gerçek bir türle değiştirildiğinde, bu değişken adının tüm örnekleri yeni türle güncellenmelidir.
  - Tür değişkenlerinin **somutlaştırılması(instantiation)** olarak adlandırılır
- **Birleştirme(Unification)**: değişkenler için tür ifadeleri, tür kontrolünün başarılı olması için değiştiğinde
  - Örnek α dizisi ve β dizisi: $\alpha == \beta$ olmasını istiyoruz, bu nedenle β her yerde α olarak değiştirilmelidir
  - Bir tür desen eşleştirmesidir(pattern matching)

Polimorfik Tür Denetimi

- Birleştirme üç durumu içerir:
  - Herhangi bir tür değişkeni, herhangi bir tür ifadesiyle birleşir (ve bu ifadeye somutlaştırılır(instantiated))
  - Herhangi iki tür sabiti yalnızca aynı türdeyse birleşir
  - Herhangi iki tür yapı (dizi veya struct gibi), yalnızca aynı tür yapıcıya sahip uygulamalarsa ve tüm bileşen türleri de yinelemeli olarak birleştirilirse birleştirilir.

Polimorfik Tür Denetimi

- Hindley-Milner tür denetimi avantajları:
  - Programcının yazması gereken tür bilgisi miktarını basitleştirir
  - Tutarlılık açısından güçlü bir şekilde kontrol edilirken türlerin olabildiğince genel kalmasına izin verir
- Hindley-Milner tür denetimi, polimorfik tür denetimini örtülü olarak uygular
- α dizisi, parametrik polimorfizm adı verilen sonsuz sayıda türden oluşan bir kümedir.
  - Hindley-Milner **örtülü parametrik polimorfizm(implicit parametric polymorphism)** kullanır

Polimorfik Tür Denetimi

- Bazen aşırı yüklemeden ayırmak için **plansız polimorfizm(ad hoc polymorphism)** olarak adlandırılır
- **Saf polimorfizm(Pure polymorphism)** (veya **alt tür polimorfizm(subtype polymorphism)**): ortak bir atayı paylaşan nesneler aynı zamanda ata için var olan operatörleri paylaştığında veya yeniden tanımladığında
- **Monomorfik(Monomorphic)**: çok biçimlilik göstermeyen bir dili tanımlar

Polimorfik Tür Denetimi

- Polimorfik fonksiyonlar, parametrik polimorfizm ve Hindley-Milner tip kontrolünün gerçek hedefidir.
- Örnek:

```
int max(int x, int y){
        return x > y ? x : y;
}
```

  - `int` başka bir aritmetik türle değiştirilirse gövde aynıdır
  - $>$ işlemini temsil eden yeni bir parametre eklenebilir

```
int max(int x, int y, int (*gt)(int a, int b) ){
        return gt(x, y) ? x : y;
}
```

Polimorfik Tür Denetimi

- C benzeri bir sözdizimi ile

```
max(x, y, gt) {
        return gt(x, y) ? x : y;
}
```

- ML sözdiziminde

```
fun max (x, y, gt) = if gt(x, y) then x else y;
```

Polimorfik Tür Denetimi

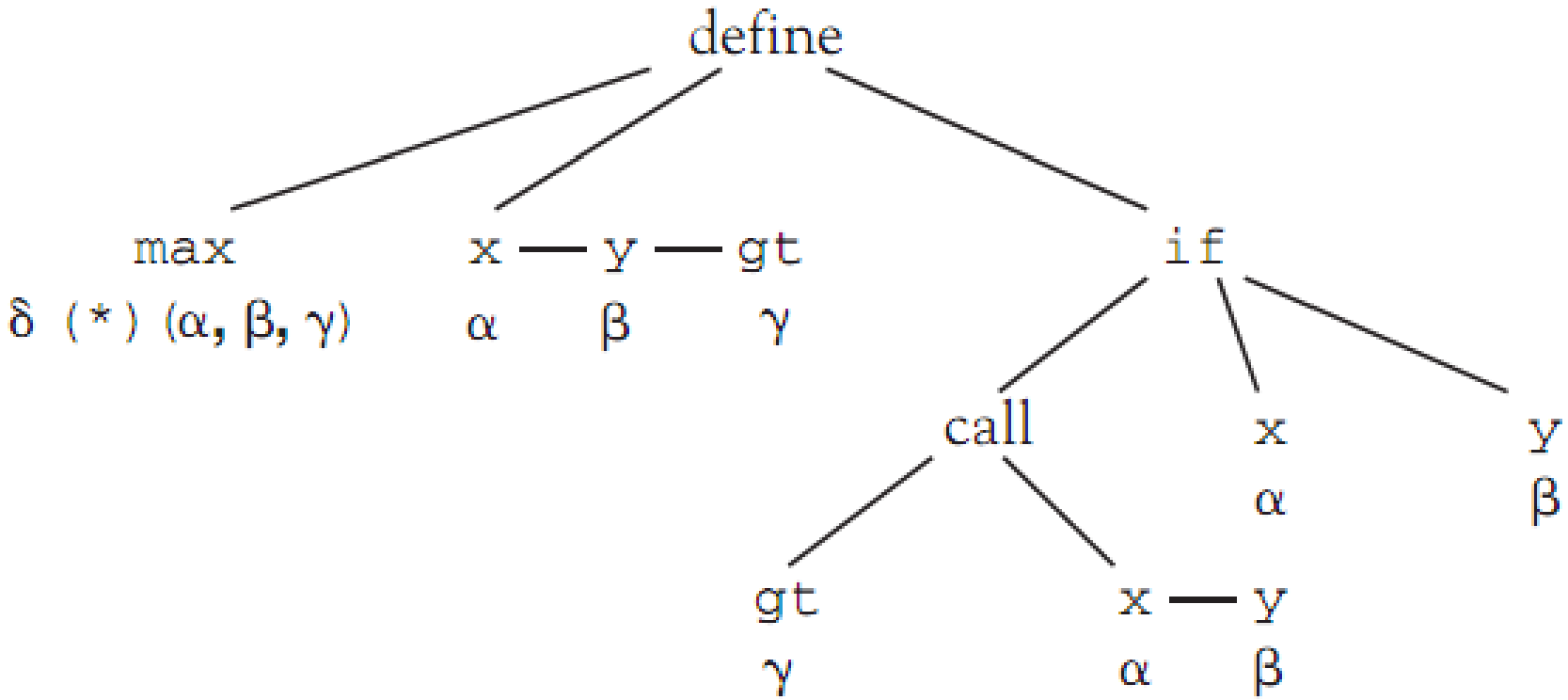


Figure 8.16 A syntax tree for the function max

Polimorfik Tür Denetimi

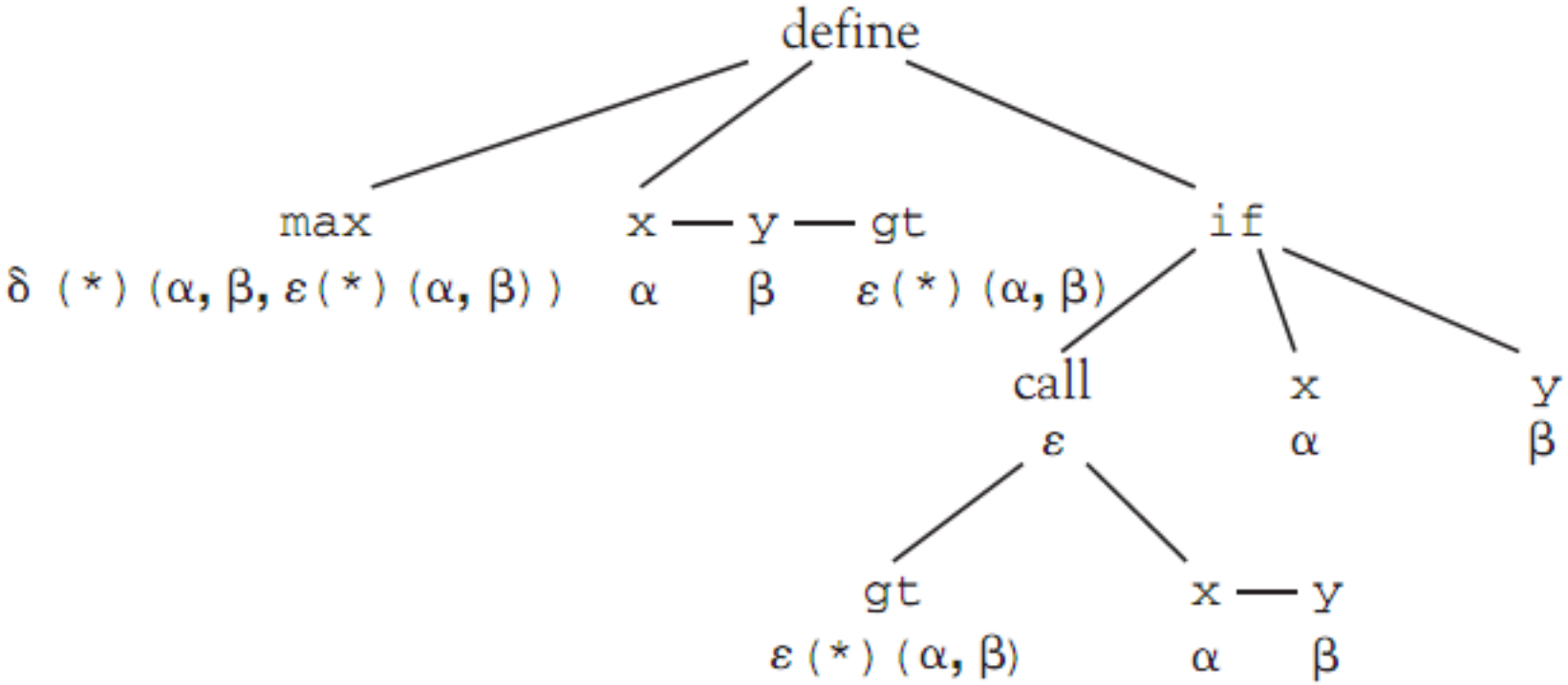


Figure 8.17 Substituting for a type variable

Polimorfik Tür Denetimi

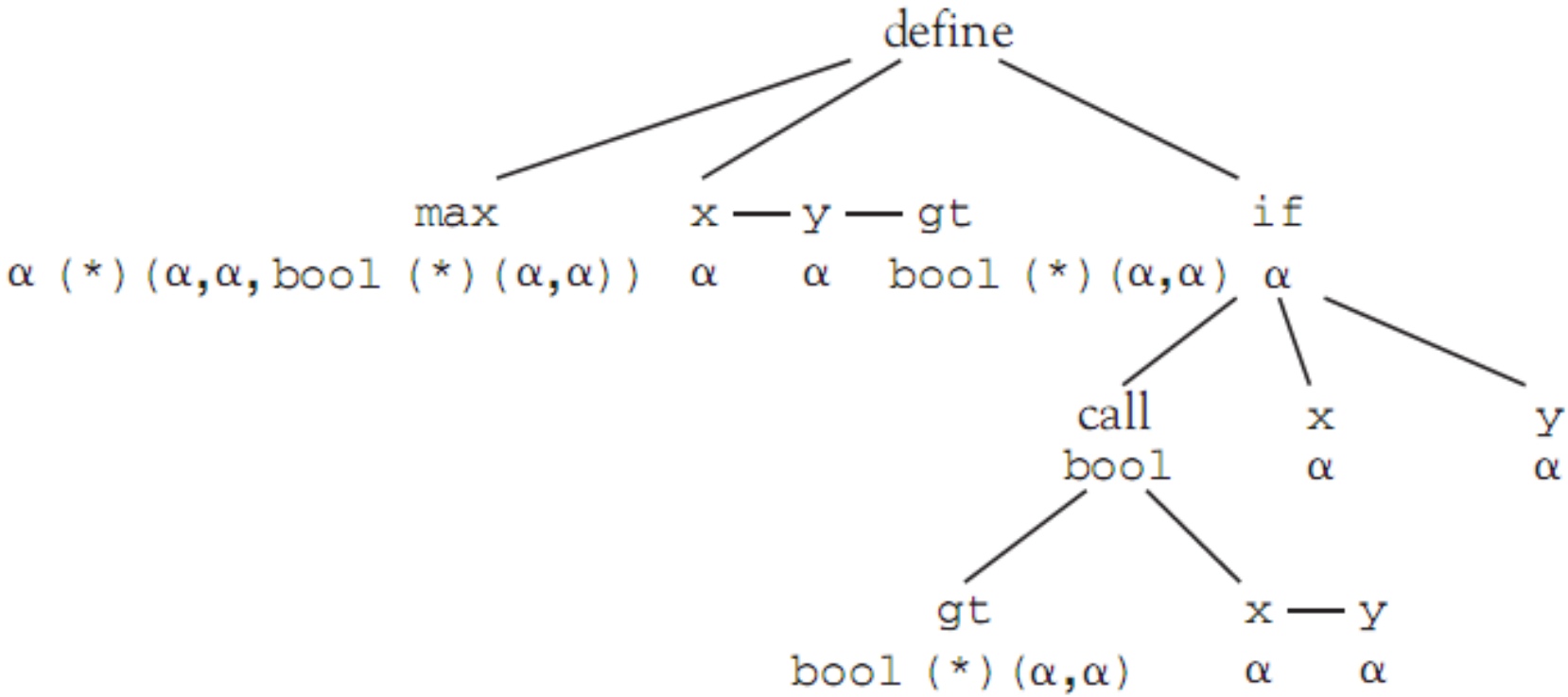


Figure 8.18 Further substitutions

Polimorfik Tür Denetimi

- Artık gerçek türlerin birleştiği her durumda max kullanabilir
- Bu tanımları ML'de verirsek:

```
fun gti(x:int, y) = x>y;
fun gtr(x:real, y) = x>y;
fun gtp((x, y), (z, w)) = gti (x, z);
```

- Max fonksiyonunu şu şekilde çağırabiliriz:

```
max(3, 2, gti); (* 3 döndürür *)
max(2.1, 3.2, gtr); (* 3.2 döndürür *)
max((2, "hello"), (1, "hi"), gtp); (* (2, "hello") döndürür *)
```

Polimorfik Tür Denetimi

- max fonksiyonu için en genel tür, onun **esas türü(principal type)** olarak adlandırılır: $\alpha\ (*)\ (\alpha, \alpha,\ \texttt{bool}\ \ (*)\ (\alpha, \alpha))$
- Her max çağrısı, bu esas türünü bir monomorfik türe özelleştirir
  - Parametrelerin türlerini de örtülü olarak özelleştirebilir
- Parametre olarak bir fonksiyona gönderilen polimorfik tipte herhangi bir nesne, fonksiyonun süresi boyunca sabit bir özelleşmeye sahip olmalıdır.
  - Buna **let-bağımlı polimorfizm(let-bound polymorphism)** denir

Polimorfik Tür Denetimi

- Hindley-Milner tip kontrolünü zorlaştıran iki problem vardır:
  - Let-bound polimorfizm
  - Tekrarı kontrol problemi
- Polimorfik türlerde de çeviri sorunları vardır
  - Türü bilmeden rastgele türdeki değerleri kopyalamak, çevirmenin değerlerin boyutunu belirleyemeyeceği anlamına gelir.
  - **Kod şişmesine(Code bloat)** neden olabilir

Açık Polimorfizm

- **Açık parametrik polimorfizm(Explicit parametric polymorphism)**: polimorfik bir veri türünü tanımlamak için, tür değişkeni açıkça yazılmalıdır
- Örnek: ML kodunda yığıt(stack) bildirimi

```
datatype 'a Stack = EmptyStack | Stack of 'a * ('a
↪   Stack);
```

  - Stack tipi değerler şu şekilde yazılabilir:

```
val empty = EmptyStack; (* empty, 'a Stack türündedir
↪   *)
val x = Stack(3, EmptyStack); (* x, int Stack
↪   türündedir *)
```

Açık Polimorfizm

- Açıkça parametreleştirilmiş polimorfik veri türleri, **kullanıcı tanımlı tür oluşturucular(user-defined type constructors)** oluşturmak için bir mekanizmadan başka bir şey değildir
- Bir tür yapıcısı, türlerden türlere bir fonksiyondur
- C'de inşa doğrudan `typedef` ile ifade edilebilir
- ML'de bu, tür yapıcı ile yapılır
- C++, açık parametrik çok biçimliliğe sahip, ancak ilişkili örtük Hindley-Milner tür denetimi olmayan bir dildir.
  - Şablon(Template) mekanizmasını kullanır

Açık Polimorifzm

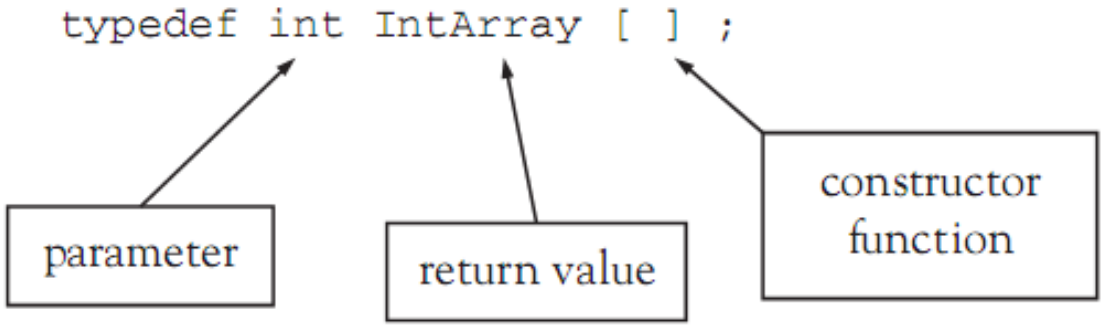


Figure 8.19 The components of a type definition in C

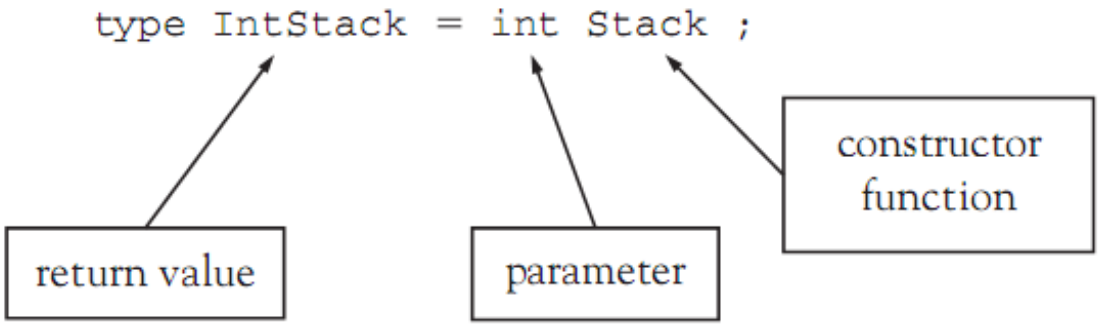


Figure 8.20: The components of a type definition in ML

Açık Polimorifzm

- **Örtülü olarak kısıtlanmış parametrik polimorfizm(Implicitly constrained parametric polymorphism)**: tür parametresine örtük olarak bir kısıtlama uygular
- **Açıkça kısıtlanmış parametrik polimorfizm(Explicitly constrained parametric polymorphism)**: hangi tür parametrelerin gerekli olduğunu açıkça belirtir

Örnek Olay: TinyAda'da Tür Denetimi

- Hedefler:
  - Kullanılmadan önce beyan edildiklerinden emin olmak için tanımlayıcıları kontrol edin
  - Tanımlayıcıların aynı blokta birden fazla beyan edilmediğini kontrol edin
  - Bir tanımlayıcının rolünü sabit, değişken, prosedür veya tür adı olarak kaydedin

Tür Uyumluluğu, Tür Eşdeğerliği ve Tür Tanımlayıcıları

- TinyAda ayrıştırıcısının şunları yapması gerekir:
  - Bir işlenenin türünün gerçekleştirilen işlem için uygun olup olmadığını kontrol edin
  - Bir atama ifadesinin sol tarafındaki adın, sağ taraftaki ifadeyle tür uyumlu olup olmadığını kontrol edin
  - Bir dizinin dizin türleri gibi belirli bildirim öğelerinin türlerini kısıtlayın

Tür Uyumluluğu, Tür Eşdeğerliği ve Tür Tanımlayıcıları

- TinyAda, tür uyumluluğunu belirlemek için gevşek bir ad denkliği biçimi kullanır
  - Diziler ve numaralandırmalar için, iki tanımlayıcı tür uyumludur, ancak ve ancak bildirimlerinde aynı tür adı kullanılarak bildirilmişlerse
  - Yerleşik türler INTEGER, CHAR ve BOOLEAN ve bunların programcı tanımlı alt aralık türleri için, iki tanımlayıcı, ancak ve ancak üst türleri adla eşdeğerse tür uyumludur

Tür Uyumluluğu, Tür Eşdeğerliği ve Tür Tanımlayıcıları

- Tür tanımlayıcı: tür özniteliklerini temsil etmek için kullanılan birincil veri yapısı
- Tür adı tanıtıldığında sembol tablosuna tür tanımlayıcı girilir
  - Yerleşik tür adları için başlangıçta INTEGER, CHAR ve BOOLEAN
  - Yeni tür bildirimleriyle her karşılaşıldığında

Tür Tanımlayıcı Sınıfların Tasarımı ve Kullanımı

- Tür tanımlayıcı, açıklanan veri türünün kategorisine bağlı olarak farklı öznitelikler içeren bir değişken kaydı gibidir.
- Her tanımlayıcı, veri türünün kategorisini tanımlamak için olası ARRAY, ENUM, SUBRANGE ve NONE değerlerini içeren bir tür formu alanı içerir.
- Dizi türü tanımlayıcısı, dizin türleri ve öğe türleri için öznitelikler içerir (bu öznitelikler aynı zamanda tür tanımlayıcılarıdır)
- Numaralandırma türü tanımlayıcısı, sabit adlarını numaralandırmak için sembol girişlerinin bir listesini içerir

Tür Tanımlayıcı Sınıfların Tasarımı ve Kullanımı

- Alt aralık türleri için tür tanımlayıcıları (INTEGER, CHAR ve BOOLEAN dahil), alt ve üst sınır değerlerini ve üst tür için bir tür tanımlayıcı içerir
- Java'da varyant kayıt yapısı yoktur
  - Bir TypeDescriptor sınıfı ve üç alt sınıfla modelleyebilir: ArrayDescriptor, SubrangeDescriptor ve EnumDescriptor

Bildirimlerde Tip Bilgilerinin Girilmesi

- Bir kaynak programda tanımlayıcıların bildirildiği her yere tip bilgisi girilmelidir
- Tür bilgileri, tür tanımlayıcılarından veya bir tür tanımından gelir
  - Tür tanımlayıcıları: tür tanımlayıcı, tanımlayıcının sembol girişinde bulunur
  - Tür tanımı: yeni bir tür oluşturulabilir

İfadelerdeki İşlemlerdeki Türleri Kontrol Etme

- TinyAda ifadelerinin kuralları, türlerinin nasıl kontrol edilmesi gerektiğine dair ipuçları verir.
- Her işlenenin türü kontrol edilmeli ve doğru tür tanımlayıcısı döndürülmelidir

İsimleri İşleme: Dizine Alınmış Bileşen Referansları ve Prosedür Çağrıları

- Prosedür en az bir parametre bekliyorsa, TinyAda dizinli bileşen referansları ve prosedür çağrıları için sözdizimi aynıdır
  - Önde gelen tanımlayıcının rolüne göre bu iki kelime öbeğini ayırt etmelidir

Statik Anlamsal Analizi Tamamlama

- Ayrıştırma sırasında diğer iki tür anlamsal kısıtlama getirilebilir:
  - Parametre modlarının kontrol edilmesi
  - Sayı bildirimlerinde ve aralık türü tanımlarında yalnızca statik ifadelerin kullanıldığını kontrol edin
- Tanya'nın üç parametre modu vardır:
  - Yalnızca giriş: in anahtar sözcüğüyle
  - Yalnızca çıktı: out anahtar sözcüğüyle
  - Girdi / çıktı: in out anahtar sözcükleriyle