Lenguajes de Programación I

Verificación de Tipos

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

Verificación de Tipos

El mecanismo para implantar las políticas

- Proceso que asegura que un programa obedece las reglas especificadas en el Sistema de Tipos.
  - Llamado "análisis semántico" – es preferible "análisis de contexto".
  - Estático, dinámico o una mezcla de ellos.
- La verificación analiza objetos del programa y les asigna un tipo.
  - Definición del objeto – especificación **explícita** del tipo.
  - Contexto de uso – especificación **implícita** del tipo.
- Se detecta y reporta escandalosamente un *Type Clash* cuando hay alguna violación a las reglas del sistema de tipos.
  - Ambigüedad en la definición.
  - Inconsistencia entre uso y contexto.
  - Imposibilidad de asignación de tipo.

Equivalencia de Tipos

Estos tipos se parecen igualito...

- Se contempla cuando el programador define sus propios tipos.
- Trata de establecer si los tipos T_1 y T_2 son el mismo.
- Si son diferentes, hemos encontrado un *Type Clash*.
- Si son el mismo, ¿es porque son idénticos o simplemente similares?

Necesitamos criterios para comparar tipos.

Equivalencia Estructural

"Byrdes of on kynde and color flok and flye allwayes together"

- Comparar la estructura – estudiar el *contenido* de las definiciones.
  - ¿Tienen los mismos componentes?
  - ¿Están en el mismo orden?
  - Utilizada en Modula-3, C (con matices perversos) y ML.
- Implantación recursiva en compilador o interpretador.
  - Tipos representados como grafos.
  - Sustituir cada nombre de tipo por su grafo.
  - Comparar recursivamente las estructuras resultantes aplicando los criterios de equivalencia particulares.
  - Apuntadores y referencias complican la implantación.
  - Técnicas se estudian y practican en Lenguajes de Programación II.

¿Cuáles diferencias son importantes?
¿Cuáles no tanto?

Equivalencia Estructural

Variedad de criterios de comparación

- ¿Les parece que estos tipos son iguales?

```
type foo = record
  a, b : integer
end;
```

```
type bar = record
  a : integer;
  b : integer;
end;
```

- ¿Y este?

```
type baz = record
  b : integer;
  a : integer;
end;
```

- La mayoría de los lenguajes dice que no, ML dice que si.

Equivalencia Estructural

No puede distinguir abstracciones diferentes

```
type credito = record
  concepto : string
  monto    : float
end;

type debito = record
  concepto : string
  monto    : float
end;

var x : credito,
    y : debito;

x = y   -- ¿Esto está bien?
```

Equivalencia por Nombres

"What's in a name? That which we call a rose by any other name..."

- Si dos definiciones de tipo tienen nombre diferente, entonces lo más *probable* es que sean tipos diferentes.
- Comparar los nombres – identidad *lexicográfica* de las definiciones.
  - Nombres diferentes – tipos diferentes.
  - Popular en lenguajes modernos (Java, C#, Ada).
- ¿Y si uso un nombre para definir otro nombre?

```
type foo = ...
type bar = foo
```

  - Se dice que `bar` es un **type alias** para `foo`.
  - ¿Son equivalentes o no?

Equivalencia por Nombres

Uso de tipos para generalizar interfaces de programación

- Construimos una librería "genérica" para pilas...

```
type stack_element = integer;

module stack;
import stack_element;
export push, pop;
...
procedure push(e : stack_element);
...
procedure pop() : stack_element;
```

- Necesitamos que la librería maneje pilas de "cualquier cosa".

¡Más vale que los alias sean equivalentes de lo contrario el módulo es inútil!

Equivalencia por Nombres

Al the bugs are belong to alias

- ¿Y en éste caso?

```
type grados_celsius = float,
     grados_fahrenheit = float;
var c : grados_celsius,
    f : grados_fahrenheit;
...
f := c
```

- Son dos escalas diferentes – eso debería impedirse.

¡Más vale que los alias no sean equivalentes
o tendremos *bugs* horrorosos!

Equivalencia por Nombres

Ni calvo, ni con dos pelucas...

- **Equivalencia Estricta por Nombres** (*Strict Name Equivalence*)
  - Tipos alias se consideran diferentes.
  - La solución más frecuente en lenguajes modernos.
- **Equivalencia Relajada por Nombres** (*Loose Name Equivalence*)
  - Tipos alias se consideran equivalentes (Pascal, Modula-2).
  - Se gana poco con usar nombres diferentes.
- Una solución más práctica – lo mejor de ambos mundos
  - Tipo alias como *subtipo* **compatible** con su *tipo padre* – `type` en Haskell, `typedef` en C/C++.
  - Tipo alias como *tipo derivado* **incompatible** con su *tipo padre* – `newtype` en Haskell.

Conversión de Tipos

`foo :=` *expresion*
`bar + baz`
`grok(` arg_1 `, ..., ` arg_k `)`

- Verificación estática de tipos – contexto sugiere el tipo.
- ¿Los tipos (esperados o provistos) han de ser *exactamente* iguales?
  - Programador *debe* indicar **conversión de tipos** explícita (*typecast*).
  - Lenguaje *debe* provee una conversión implícita.
- En ambos casos podría haber código ejecutable adicional – escrito por el programador o generado por el compilador.

Conversiones Explícitas

Primer Caso

- $T_{provisto}$ y $T_{esperado}$ tienen nombres diferentes – el lenguaje usa Equivalencia por Nombres.
- $T_{provisto}$ y $T_{esperado}$ son Estructuralmente Equivalentes
  - Tienen la misma representación de bajo nivel.
  - Tienen el mismo conjunto de valores posibles.
- La conversión no require generar código adicional – basta copiar los bits de un lado a otro.

Conversiones Explícitas

Segundo Caso

- $T_{provisto}$ y $T_{esperado}$ tienen nombres diferentes – el lenguaje usa Equivalencia por Nombres.
- $T_{provisto}$ y $T_{esperado}$ son Estructuralmente Equivalentes
  - Tienen la misma representación de bajo nivel.
  - Conjuntos de valores diferentes, pero con intersección no vacía.
- Si $T_{provisto} \subset T_{esperado}$, no es necesario hacer verificaciones – ¡esto se puede deducir estáticamente!
- Si $T_{esperado} \subset T_{provisto}$, es necesario generar código extra para verificar la validez del valor a usar.
  - La verificación se hace a tiempo de *ejecución*.
  - Cuando tiene éxito, se usa la representación común.
  - Cuando falla, se genera un error semántico a tiempo de ejecución.

Conversiones Explícitas

Tercer Caso

- $T_{provisto}$ y $T_{esperado}$ tienen nombres diferentes – el lenguaje usa Equivalencia por Nombres.
- $T_{provisto}$ y $T_{esperado}$ **no** son Estructuralmente Equivalentes.
  - Tienen representaciones de bajo nivel diferentes.
  - Existe una correspondencia entre sus valores posibles.
- Es necesario generar código extra para realizar la conversión, y posiblemente verificaciones adicionales.

Conversiones Explícitas

Un ejemplo

```
n : integer;  - Asumir 32 bits
r : float;     - Asumir IEEE doble
t : 0..100;
c : grados_celsius;
...
t := test_score(n);     Caso 2: Emitir código para verificar
n := integer(t);        Caso 2: No hace falta código extra
r := float(n);          Caso 3: Emitir código para convertir
n := integer(r);        Caso 3: Emitir código para convertir
n := integer(c);        Caso 1: Copiar directamente
c := grados_celsius(r); Caso 1: Copiar directamente
```

Non-Converting Type Casts

Conversiones que convierten sin cambiar

- Cambian el *tipo* del valor, pero *matienen* la representación.
- Típicas en programación de sistemas de operación o algoritmos especializados de cálculo numérico.
- C++ es el más "abusador"
  - `static_cast` – conversión completa al tipo destino.
  - `reinterpret_cast` – cambia el tipo, mantiene los *bits*.
  - `dynamic_cast` – ignora el tipo estático, permitiendo asignaciones que serán válidas *dinámicamente*.
  - `const_cast` – ignorar la supuesta inmutabilidad del objeto destino.
- En C se simulan a través de apuntadores

```
r = *((float *) &n)
```

Compatibilidad de Tipos

- Equivalencia de Tipos no siempre es necesaria en todos los casos – basta que el tipo de cada valor sea **compatible** con el contexto.
- "Compatible" es muy relativo y depende del lenguaje
- Cuando el lenguaje hace la conversión implícita automática, se dice que hace **coerción de tipos** (*type coercion*).
- Debilitan el sistema de tipos.

Coerciones en C

```
short int s;
unsigned long int l;
char c;   /* Con o sin signo? */
float f;
double d;

s = l;  /* Bits bajos de l */
l = s;  /* s se extiende en signo */
s = c;  /* c se extiende en signo o en cero */
f = l;  /* l se reinterpreta como float */
d = f;  /* f se reinterpreta como double */
f = d;  /* d se reduce peligrosamente */
```

- Esto es verdad hasta C89 – se *sugirieron* conversiones implícitas.
- Son *obligatorias* a partir de C99.

La respuesta

. . . de la vida, el universo y todo

42

`int` (32 bits)	00000000 00000000 00000000 00101010
`float` (32 bits)	01000010 00101000 00000000 00000000
`double` (64 bits)	01000000 01000101 00000000 00000000 00000000 00000000 00000000 00000000

Inferencia de Tipos

- Determinar el tipo de una expresión a partir de sus componentes.
- Basados en el *Algoritmo de Unificación* y *Ecuaciones de Tipos*
  - Deducen tipos complejos a partir de tipos simples.
  - Calculan el tipo "más general posible".
- Consideremos

```
type A = 0..20, B = 10..20;
var a : A, b : B;
```

- ¿Cuál es el tipo de `a + b`?
- ¿Cuál es el tipo de `a := b - foo(10)`?

Correctitud del Sistema de Tipos

We aim to please and protect

- Dado un programa P, el sistema de tipos del lenguaje pretende impedir que ocurran los errores E_1, E_2, ... a tiempo de *ejecución*.
- Para prevenirlos, primero debe *detectarlos*
  - "Lo antes posible" cuando son estáticos.
  - "Nunca demasiado tarde" cuando son dinámicos.

¿Cómo sabemos si "hace lo correcto"?

Solidez y Completitud

Cualidades complementarias

- El sistema de tipos es **sólido** si nunca acepta un programa que al correr comete el error E_i – nunca reporta *falsos negativos*.
- El sistema de tipos es **completo** si nunca rechaza un programa que jamás cometería el error E_i – nunca reporta *falsos positivos*.

Debe ser correcto $\forall i : E_i$ para ser útil.

¡Pero yo quiero que sea estático!

... por aquello de no pegarme tiros en los pies

1. Quiero que el verificador estático *siempre termine* – para terminar de compilar y correr.
2. Quiero que el verificador estático sea sólido – para que mis programas no tengan errores de tipos.
3. Quiero que el verificador estático sea completo – para que nunca me niegue un programa correcto.

Para una gran cantidad de cosas que se quieren predecir, es *imposible* disfrutar de los tres beneficios.

En la práctica...

You can't always get what you want

- La verificación dinámica es necesariamente riesgosa – la verificación estática es necesariamente aproximada.
- Los sistema de tipos se diseñan para ser **sólidos**
  - Hasta el punto de *demostrarlo* matemáticamente.
  - Aunque se prevengan pocas cosas, *seguro* que se previenen.
- Es imposible garantizar la **completitud**
  - Muchas de las cosas que se quieren prevenir no son decidibles.
  - Otras cosas son simplemente incompletas – en el sentido Gödeliano.

Diseñamos sistemas de tipos que terminen y que sean sólidos – agregamos características para aproximar la completitud.

Bibliografía

- [*Scott*]
  - Sección 7.2
  - Ejercicios y Exploraciones