Capítulo 1

Introducción

La ingeniería del software reconoce un amplio conjunto de métodos formales que ayudan a garantizar que un sistema funciona correctamente con respecto a una especificación de su comportamiento deseado.

- Técnicas complejas: lógica de Hoare, lenguajes de especificación algebraica, lógica modal, semántica denotacional, etc.
- Model checking: métodos que encuentran errores en sistemas con un conjunto finito de estados.
- Monitorización en tiempo de ejecución: colección de técnicas que permiten detectar, dinámicamente, cuando uno de los componentes de un sistema no está funcionando correctamente.
- **Sistemas de tipos:** método sintáctico para comprobar la ausencia de ciertos comportamientos en los programas, clasificando frases de acuerdo a los tipos de valores que calculan.

Un sistema de tipos calcula una aproximación estática al comportamiento en tiempo de ejecución de los términos de un programa. Debido a su naturaleza estática, los sistemas de tipos son *conservadores*: pueden probar categóricamente la ausencia de "malos comportamientos", pero no pueden probar su presencia y, por ello, a veces rechazarán programas que funcionan bien en tiempo de ejecución. Por ejemplo:

```
if <test complejo> then 5 else <error de tipo>
```

será rechazado aunque el <`test complejo`> siempre se evalue a verdadero. La investigación en el área de los sistemas de tipos va encaminada a permitir el tipado de cada vez más programas.

De la misma forma, debido a su naturaleza estática, los sistemas de tipos no pueden garantizar:

- Divisor distinto de cero.
- Acceso a posición de array fuera de los límites.

Los sistemas de tipos sí pueden garantizar que:

- Los argumentos de una operación aritmética son números.
- El objeto receptor de una petición proporciona método requerido.
- No se producen violaciones de acceso a información privada.

Los sistemas de comprobación de tipos suelen estar implementados en el compilador, por lo que deben ser capaces de realizar su función automáticamente. En muchos casos, el programador "ayuda" al compilador indicando los tipos explícitamente, sin embargo esta anotación es bastante limitada para facilitar la escritura y lectura de los programas. Algunos lenguajes, como la familia ML, reducen las anotaciones al mínimo y utilizan un sistema de inferencia de tipos para anotar automáticamente los programas. Otros lenguajes, como el C o el Java, son más verbosos.

La realidad es que las anotaciones de tipos podrían proporcionar una prueba completa de que un programa cumple las especificaciones. En este caso, un sistema de tipos podría convertirse en un sistema de pruebas.

1.1. Funcionalidad de los sistemas de tipos

1.1.1. Detección de errores

El beneficio más obvio de los sistemas de tipos es la detección de errores de programación ya que muchos errores se manifiestan como una inconsistencia al nivel de los tipos. Los errores que se detectan durante la compilación suelen ser mucho más sencillos de corregir que los errores que se producen en tiempo de ejecución, cuando los efectos del error pueden no manifestarse hasta pasado algún tiempo desde que las cosas empezaron a ir mal.

Los sistemas de tipos pueden resultar de gran utilidad para el mantenimiento de programas. Por ejemplo, a la hora de modificar la definición de una estructura de datos, en lugar de buscar en el código todos los puntos en los que se utiliza dicha estructura, bastará con modificar su definición, compilar y comprobar los errores de tipos que se producen.

1.1.2. Abstracción

Los sistemas de tipo también favorecen una forma de programación más disciplinada. En el diseño de software a gran escala, los sistemas de tipos constituyen uno de los pilares básicos de los sistemas modulares. De hecho, la interfaz puede ser vista como el "tipo de un módulo" que indica los servicios implementados por dicho módulo.

Al estructurar los sistemas en módulos con sus correspondientes interfaces, se está favoreciendo un estilo de diseño más abstracto en el que las interfaces se diseñan independientemente de sus implementaciones.

1.1.3. Documentación

Los tipos también facilitan la lectura de los programas y constituyen una forma de documentación, que proporciona pistas sobre el funcionamiento. Además, esta forma de documentación no puede quedarse obsoleta, ya que se utiliza cada vez que se compila el programa.

1.1.4. Seguridad

Un lenguaje seguro puede definirse como aquel que protege sus propias abstracciones. Todos los lenguajes de programación de alto nivel proporcionan abstracciones de sus servicios. Por ejemplo, un lenguaje puede proporcionar arrays como una abstracción de la memoria subyacente. Si el lenguaje es seguro, dichos arrays podrán ser accedidos y modificados utilizando únicamente sus correspondientes operaciones y no accediendo a la memoria directamente. Con un lenguaje no seguro se puede corromper, no sólo las propias estructuras de datos, sino también la memoria del sistema.

A continuación presentamos ejemplos de lenguajes seguros y no seguros.

	Chequeo estático	**Chequeo dinámico**
Seguros	ML, Haskell, Java, etc.	Lisp, Scheme, Perl, Postscript, etc.
No seguros	C, C++, etc.	

Normalmente, la seguridad en tiempo de ejecución no puede garantizarse únicamente a través del tipado estático. De hecho, los lenguajes listados como seguros en la tabla anterior llevan a cabo dinámicamente las comprobaciones relativas a los límites de los arrays.

La seguridad de un lenguaje no suele ser absoluta. Incluso los lenguajes considerados seguros incluyen mecanismos que permiten "saltarse" la seguridad (por ejemplo, permitir la llamada a funciones escritas en otro lenguaje).

En 1996, Cardeli introdujo la distinción entre los errores capturados y no-capturados. Los errores capturados son aquellos que provocan la detención del programa o el lanzamiento de una excepción. Los errores no capturados permiten que la ejecución continúe, al menos durante un tiempo. En este sentido, un lenguaje seguro sería el que previene los errores no capturados.

1.1.5. Eficiencia

El primer sistema de tipado (Fortran) se introdujo para mejorar la eficiencia de los cálculos matemáticos, distinguiendo entre expresiones enteras y reales. Cada uno tenía su propia representación y sus propias instrucciones máquina.

Además, con los lenguajes seguros es posible eliminar muchas de las comprobaciones dinámicas necesarias para garantizar la seguridad, lo cual implica también una mejora de la eficiencia.

1.1.6. Prueba automática de teoremas

De cara a la prueba automática de teoremas, los sistemas de tipos se utilizan para representar las proposiciones lógicas y las pruebas. Muchos asistentes de pruebas, como Coq, están basados en teoría de tipos.

1.2. Influencia de los sistemas de tipos en el diseño de los lenguajes

Introducir un sistema de tipos en un lenguaje diseñado sin él puede ser complicado. Por eso, es mejor que el diseño del lenguaje y el diseño del sistema de tipos

sean paralelos.

La sintaxis de un sistema tipado suele ser más complicada que la de un sistema no tipado. Es por ello que el diseño paralelo permite la generación de lenguajes con una sintaxis más clara y comprensible.

1.3. Un poco de historia

Final de la década de los 50: Aparece el primer sistema de tipos (en Fortran) para diferenciar enteros de flotantes.

Comienzo de la década de los 60: El tipado se extiende a los datos estructurados (registros) y funciones.

Década de los 70: Aparecen conceptos más complejos como el polimorfismo paramétrico, los tipos de datos abstractos, el subtipado, etc. Los sistemas de tipos se convierten en un tema de estudio.

Capítulo 2

Sistemas no tipados

En este capítulo vamos a definir un pequeño lenguaje no tipado para trabajar con booleanos y números. El lenguaje contiene un pequeño conjunto de términos: las constantes booleanas `true` y `false`, la constante numérica `0`, los operadores aritméticos `pred` y `succ` y la operación `iszero`. La gramática de nuestro lenguaje es la siguiente:

```
t::=
   true
   false
   if t then t else t
   0
   succ t
   pred t
   iszero t
```

La primera línea (`t::=`) declara que estamos definiendo un conjunto de términos y que vamos a utilizar `t` de forma genérica para referenciarlos. `t` es una *metavariable*. Es una variable porque se trata de un contenedor que será sustituido por un término particular. En este contexto, el prefijo *meta* indica que no es una variable del lenguaje de programación en si, sino del *metalenguaje* que utilizaremos para la descripción.

Un programa en nuestro lenguaje será un término construido a partir de la gramática. A continuación citamos algunos programas con los resultados de su evaluación:

```
if false then 0 else 1;
> 1
iszero (pred(succ 0));
> true
```

Por brevedad, vamos a utilizar números al describir nuestros programas, aunque en realidad los números se representan formalmente a través de aplicaciones de la función succ a 0. Por ejemplo `succ(succ(succ 0))` equivale a 3.

Los resultados de la evaluación de un programa serán siempre constantes booleanas o números. A estos términos se les denomina *valores*. Cabe destacar que el presente lenguaje permite la formación de términos como `if 0 then 0 else 0` o `succ true`, que son ejemplos del tipo de programas que serán excluidos por el sistema de tipos.

2.0.1. Sintaxis

La sintaxis de nuestro lenguaje se puede definir de varias formas:

Definición inductiva. El conjunto de términos es el menor conjunto τ tal que:

1. $\{\texttt{true, false, 0}\} \subseteq \tau$.
2. Si $\texttt{t}_1 \in \tau$, entonces $\{\texttt{succ t}_1\texttt{, pred t}_1\texttt{, iszero t}_1\} \subseteq \tau$.
3. Si $\texttt{t}_1\texttt{, t}_2\texttt{, t}_3 \in \tau$, entonces $\{\texttt{if t}_1\ \texttt{then t}_2\ \texttt{else t}_3\} \in \tau$.

El calificativo "menor" nos indica que τ no contiene ningún otro término aparte de los anteriormente mencionados.

Definición por reglas inductivas. El conjunto de términos se define como:

1. $\texttt{true} \in \tau$, $\texttt{false} \in \tau$, $\texttt{0} \in \tau$.
2. $\frac{\texttt{t}_1 \in \tau}{\texttt{succ t}_1 \in \tau}$, $\frac{\texttt{t}_1 \in \tau}{\texttt{pred t}_1 \in \tau}$, $\frac{\texttt{t}_1 \in \tau}{\texttt{iszero t}_1 \in \tau}$
3. $\frac{\texttt{t}_1 \in \tau \quad \texttt{t}_2 \in \tau \quad \texttt{t}_3 \in \tau}{\texttt{if t}_1\ \texttt{then t}_2\ \texttt{else t}_3 \in \tau}$

Si hemos establecido las premisas encima de la línea, entonces podemos derivar las conclusiones de debajo de la línea. Las reglas sin premisas suelen llamarse *axiomas*.

Definición por enumeración de los términos. Para cada número natural i, definimos el conjunto S_i como:

$$\begin{aligned} S_0 &= \emptyset. \\ S_{i+1} &= \{\texttt{true}, \texttt{false}, \texttt{0}\} \\ &\quad \cup \{\ \texttt{succ t}_1,\ \texttt{pred t}_1,\ \texttt{iszero t}_1 \mid \texttt{t}_1 \in S_i\} \\ &\quad \cup \texttt{if t}_1\ \texttt{then t}_2\ \texttt{else t}_3 \mid \texttt{t}_1, \texttt{t}_2, \texttt{t}_3 \in S_i\} \\ S &= \cup_i S_i. \end{aligned}$$

S_0 está vacío, S_1 contiene las constantes, S_2 contiene las constantes y las frases que se pueden construir a partir de las constantes, S_2 contiene las constantes y todas las frases que pueden construirse utilizando `succ, pred, iszero, if` con las frases de S_2, etc.

Ejercicio: Cuántos elementos tiene S_3?

Ejercicio: Demostrar que $S_i \subseteq S_{i+1}, \forall i$.

2.1. Inducción en base a términos

Según la definición inductiva del conjunto de términos τ, si $\texttt{t} \in \tau$ existen tres posibilidades: `t` es una constante, `t` tiene la forma $\texttt{succ t}_1\texttt{, pred t}_1\texttt{, iszero t}_1$ o bien la forma $\texttt{if t}_1\ \texttt{then t}_2\ \texttt{else t}_3$ donde $\texttt{t}_1\texttt{, t}_2\texttt{, t}_3$ son términos *menores*. Esta estructura nos permite enunciar definiciones inductivas y probar inductivamente diferentes propiedades de los términos. A continuación algunos ejemplos:

1. El conjunto de constantes que aparecen en el término `t`, denominado $Consts(\texttt{t})$, se define como:

$$\begin{array}{lcl} Consts(\texttt{true}) & = & \{\texttt{true}\} \\ Consts(\texttt{false}) & = & \{\texttt{false}\} \\ Consts(\texttt{0}) & = & \{\texttt{0}\} \\ Consts(\texttt{succ t}_1) & = & Consts(\texttt{t}_1) \\ Consts(\texttt{pred t}_1) & = & Consts(\texttt{t}_1) \\ Consts(\texttt{iszero t}_1) & = & Consts(\texttt{t}_1) \\ Consts(\texttt{if t}_1\ \texttt{then t}_2\ \texttt{else t}_1) & = & Consts(\texttt{t}_1) \cup Consts(\texttt{t}_2) \cup Const(\texttt{t}_3) \end{array}$$

2. El tamaño de un término t, denominado $size(\texttt{t})$ y que indica el número de nodos en el árbol sintáctico abstracto, se define como:

$$\begin{array}{lcl} size(\texttt{true}) & = & 1 \\ size(\texttt{false}) & = & 1 \\ size(\texttt{0}) & = & 1 \\ size(\texttt{succ t}_1) & = & 1 + size(\texttt{t}_1) \\ size(\texttt{pred t}_1) & = & 1 + size(\texttt{t}_1) \\ size(\texttt{iszero t}_1) & = & 1 + size(\texttt{t}_1) \\ size(\texttt{if t}_1\ \texttt{then t}_2\ \texttt{else t}_3) & = & 1 + size(\texttt{t}_1) + size(\texttt{t}_2) + size(\texttt{t}_3) \end{array}$$

3. La profundidad de un término, denominada $depth(\texttt{t})$, se define como:

$$\begin{array}{lcl} depth(\texttt{true}) & = & 1 \\ depth(\texttt{false}) & = & 1 \\ depth(\texttt{0}) & = & 1 \\ depth(\texttt{succ t}_1) & = & 1 + depth(\texttt{t}_1) \\ depth(\texttt{pred t}_1) & = & 1 + depth(\texttt{t}_1) \\ depth(\texttt{iszero t}_1) & = & 1 + depth(\texttt{t}_1) \\ depth(\texttt{if t}_1\ \texttt{then t}_2\ \texttt{else t}_3) & = & 1 + max(depth(\texttt{t}_1), depth(\texttt{t}_2), depth(\texttt{t}_3)) \end{array}$$

Vamos a probar ahora por inducción un lema que relaciona el número de constantes de un término con su tamaño. Introduciremos así las pruebas inductivas, que serán ampliamente usadas a lo largo de este curso.

La inducción en base a términos, o inducción estructural, es equivalente a la inducción en base a números naturales. Dado un término t para el cual se quiere probar cierta propiedad P, la prueba se lleva a cabo asumiendo que P se cumple para todos los subtérminos (o términos menores).

Lema: El número de constantes diferentes en un término no será superior a su tamaño, $|Consts(\texttt{t})| \leq size(\texttt{t})$

Demostración: Por inducción, con tres casos a considerar:

Caso 1: el término t es una constante.

$|Consts(\texttt{t})| = 1 = size(\texttt{t})$.

Caso 2: `t = succ` $\texttt{t}_1$`, pred` $\texttt{t}_1$`, iszero` $\texttt{t}_1$.

Según la hipótesis de inducción $|Consts(\texttt{t}_1)| \leq size(\texttt{t}_1)$, por lo tanto $|Consts(\texttt{t})| = |Consts(\texttt{t}_1)| \leq size(\texttt{t}_1) < size(\texttt{t})$

Caso 3: `t = if t1 then t2 else t3`.

Según la hipótesis de inducción, $|Consts(\texttt{t}_1)| \leq size(\texttt{t}_1)$, $|Consts(\texttt{t}_2)| \leq size(\texttt{t}_2)$ y $|Consts(\texttt{t}_3)| \leq size(\texttt{t}_3)$, por lo tanto $|Consts(\texttt{t})| = |Consts(\texttt{t}_1) \cup Consts(\texttt{t}_2) \cup Consts(\texttt{t}_3)| \leq |Consts(\texttt{t}_1)|+|Consts(\texttt{t}_2)|+|Consts(\texttt{t}_1)| < size(\texttt{t}_1) + size(\texttt{t}_2) + size(\texttt{t}_3) < size(\texttt{t})$.

Principios de inducción en base a términos

Sea P un predicado que queremos probar por inducción. Existen tres alternativas:

1. Por inducción en profundidad:

   Si, para cada término `s`,
   asumiendo que $P(\texttt{r})$ se cumple para todo `r` cuya $depth(\texttt{r}) < depth(\texttt{s})$,
   podemos probar $P(\texttt{s})$.
   entonces $P(\texttt{s})$ se cumple para todo `s`.

2. Por inducción en tamaño:

   Si, para cada término `s`,
   asumiendo que $P(\texttt{r})$ se cumple para todo `r` cuya $size(\texttt{r}) < size(\texttt{s})$,
   podemos probar $P(\texttt{s})$.
   entonces $P(\texttt{s})$ se cumple para todo `s`.

3. Por inducción estructural:

   Si, para cada término `s`,
   asumiendo que $P(\texttt{r})$ se cumple para todos los subtérminos `r` de `s`
   podemos probar $P(\texttt{s})$.
   entonces $P(\texttt{s})$ se cumple para todo `s`.

El tipo de inducción a utilizar debe decidirse en función del problema: debe elegirse el tipo que lleve a una solución más sencilla del problema.

2.2. Estilos semánticos

Una vez definida la sintaxis de nuestro lenguaje, falta por definir cómo se evaluarán los términos, lo cual se denomina *semántica* del lenguaje. Existen tres aproximaciones básicas para formalizar la semántica:

1. *Semántica operacional:* especifica el comportamiento de un lenguaje de programación definiendo una *máquina virtual*. En los lenguajes sencillos, los *estados* de la máquina son los términos. El comportamiento se define a través de las *funciones de transición* que, a partir de un estado, indican el estado siguiente a través de un paso de simplificación o declaran que la ejecución se ha interrumpido. El *significado* de un término puede interpretarse como el estado final de la máquina.

2. *Semántica denotacional:* en lugar de una secuencia de estados, el *significado* de un término se define como un objeto matemático denominado *denotación*. Por lo tanto, en la semántica denotacional resulta necesario definir un *dominio semántico*, un conjunto de objetos matemáticos que representen lo que los programas llevan a cabo, y de *funciones de interpretación* que mapeen los términos en elementos de este dominio. [1].

3. *Semántica axiomática:* en este caso, el *el significado* de un término viene dado por lo que se puede probar a través de él. Las semánticas axiomáticas centran la atención en el proceso de razonar sobre programas, generando aserciones sobre el estado del programa PSQ (la aserción Q se cumplirá en el estado generado al ejecutar el término S en un estado que cumpla la aserción P). Esta línea de pensamiento ha dado lugar a ideas como las *invariantes* de bucles.

En los años 60 y 70, la semántica operacional era considerada inferior a las otras dos, por resultar poco elegante y tener una base matemática menos sólida. Sin embargo, a partir de los años 80, las otras dos semánticas comenzaron a encontrar problemas de difícil solución y la semántica operacional resurgió, siendo ampliamente utilizada en la actualidad. De hecho, será la semántica que utilizaremos nosotros durante el curso.

2.3. Evaluación

Vamos a analizar la semántica operacional de los booleanos, vamos a olvidarnos de los números por el momento. La gramática de los booleanos se definiría como:

```
t::=
     true
     false
     if t then t else t

v::=
     true
     false
```

donde `t` son los términos y `v` los valores, es decir, el subconjunto de los términos que pueden ser los resultados finales de la evaluación.

La *relación de evaluación* de los términos $\texttt{t} \rightarrow \texttt{t}'$, entendida como "`t` se evalúa a `t'`" en un único paso, se define como:

$$\texttt{if true then } t_2 \texttt{ else } t_3 \rightarrow t_2 \qquad (\texttt{E} - \texttt{IfTrue})$$

$$\texttt{if false then } t_2 \texttt{ else } t_3 \rightarrow t_3 \qquad (\texttt{E} - \texttt{IfFalse})$$

$$\frac{t_1 \rightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \rightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \qquad (\texttt{E} - \texttt{If})$$

Las dos primeras reglas de inferencia son en realidad *axiomas*, ya que no tienen premisas.

[1] Algo similar a lo que ocurre durante la compilación pero en lugar de traducir a otro lenguaje, traducimos a un formalismo matemático

La regla `E-IfTrue` indica que, si la guarda de un término condicional es literalmente `true` el nuevo estado de la máquina debería ser t_2. La regla `E-IfFalse` indica que, si la guarda de un término condicional es literalmente `false` el nuevo estado de la máquina debería ser t_3. Por último, la regla `E-If` indica que si la guarda t_1 se evalúa a t_1', entonces el término condicional `if` t_1 `then` t_2 `else` t_3 se evalúa a `if` t_1' `then` t_2 `else` t_3.

Según nuestras reglas de evaluación, cómo se evaluaría el término: `if true then (if false then false else false) else true`? No se evalúa a `if true then false else true`, sino que en primer lugar se evaluará la guarda del condicional usando la regla `E-If`. Las reglas `E-IfTrue` y `E-IfFalse` sólo se pueden utilizar cuando la guarda ya ha sido totalmente evaluada.

Vamos a definir formalmente la *relación evaluación* de términos:

Definición: Una *instancia* de una regla de inferencia se obtiene sustituyendo cada metavariable por un determinado término tanto en la conclusión de la regla como en todas sus premisas (si las tiene).

Por ejemplo: `if true then true else (if false then false else false)` → `true` es una instancia de la regla `E-IfTrue` en la que t_2 ha sido sustituida por `true` y t_3 ha sido sustituida por `if false then false else false`.

Definición: Una relación satisface una regla si, para cada instancia de la regla, o bien la conclusión está en la relación o bien alguna de las premisas no está en la relación.

Definición: La relación de *evaluación en un paso* → es la menor relación binaria de términos que satisface las tres reglas de nuestro sistema de evaluación. Cuando un par `(t,t')` satisface la relación de evaluación, decimos que `t` → `t'` es *derivable*.

La derivabilidad de una expresión puede justificarse a través de su *árbol de derivación*, cuyas hojas están etiquetadas con instancias de las reglas `E-IfTrue` y `E-IfFalse` y cuyos nodos internos están etiquetados con instancias de la regla `E-If`.

Ejemplo: A partir de las sentencias

```
s = if true then false else false
t = if s then true else true
u = if false then true else true
```

intentemos probar la derivabilidad de la sentencia `if t then false else false` → `if u then false else false`. El árbol de derivación sería el siguiente:

$$\dfrac{\dfrac{}{\texttt{s} \rightarrow \texttt{false}}\,\texttt{E-IfTrue}}{\texttt{t} \rightarrow \texttt{u}}\,\texttt{E-If}$$

$$\dfrac{\texttt{t} \rightarrow \texttt{u}}{\texttt{if t then false else false} \rightarrow \texttt{if u then false else false}}\,\texttt{E-If}$$

El árbol del ejemplo anterior no tiene bifurcaciones porque ninguna regla de evaluación tiene más de una premisa. El árbol de derivaciones permite introducir el concepto de *inducción basada en derivaciones.*

Teorema *Determinación de las evaluaciones en un paso*: Si `t` $\rightarrow$ `t'` y `t` $\rightarrow$ `t''`, entonces `t'` = `t''`.

Vamos a probar éste teorema por *inducción en base a derivaciones.* Consiste en asumir que el resultado es válido para todas las derivaciones menores y analizar la regla de evaluación utilizada en la raíz de la derivación.

Supongamos que la regla utilizada en la raíz de la derivación `t` $\rightarrow$ `t'` es `E-IfTrue`, en ese caso, sabemos que `t` es de la forma `if` t_1 `then` t_2 `else` t_3 y que t_1 `= true`. Ello implica que la regla ejecutada en la raíz de la derivación `t` $\rightarrow$ `t''` no puede ser ni `E-IfFalse` ni `E-If`, dado que hemos establecido que t_1 `= true`. Por lo tanto, la regla a aplicar en la raíz únicamente puede ser `E-IfTrue` de lo cual se deduce que `t'` = `t''`. El razonamiento es análogo para la regla `E-IfFalse`.

Supongamos ahora que la regla aplicada en la raíz de la derivación `t` $\rightarrow$ `t'` es `E-If`, en ese caso, `t` es de la forma `if` t_1 `then` t_2 `else` t_3 donde t_1 $\rightarrow$ t_1`'`. Por lo tanto, la regla aplicada en la raíz de la derivación `t` $\rightarrow$ `t''` ha de ser también `E-If` ya que hemos establecido que `t` es de la forma `if` t_1 `then` t_2 `else` t_3 donde, en este caso t_1 $\rightarrow$ t_1`''`. Según la hipótesis de inducción, para todas las derivaciones menores el resultado a probar es válido, por lo tanto, al ser t_1`'` y t_1`''` derivaciones menores, se ha de cumplir que t_1`'` = t_1`''`, lo cual implica que `t'` = `if` t_1`'` `then` t_2 `else` t_3 `=if` t_1`''` `then` t_2 `else` t_3 `= t''`.

Definición: Un término `t` está en *forma normal* si no se le puede aplicar ninguna regla de derivación, es decir, si no existe un término `t'` tal que `t` $\rightarrow$ `t'`.

Teorema: Todos los valores están en forma normal.

En nuestra gramática booleana, `false` y `true` están en forma normal y son valores. En general, la expresión contraria "todas las formas normales son valores" no tiene por qué cumplirse, de hecho, las formas normales que no son valores jugarán un papel muy importante en el análisis de los *errores en tiempo de ejecución.*

Definición: La relación de *evaluación en varios pasos* $\rightarrow^*$ es el cierre reflexivo y transitivo de la *evaluación en un sólo paso*, es decir:

1. Si `t` $\rightarrow$ `t'`, entonces `t` $\rightarrow^*$ `t'`.
2. Para todo `t`, se cumple que `t` $\rightarrow^*$ `t`.
3. Si `t` $\rightarrow^*$ `t'` y `t'` $\rightarrow *$ `t''`, entonces `t` $\rightarrow^*$ `t''`

Teorema *Unicidad de las formas normales*: Si `t` $\rightarrow^*$ `u` y `t` $\rightarrow^*$ `u'` y tanto `u` como `u'` son formas normales, en este caso `u` = `u'`.

La prueba de este teorema es muy similar a la prueba del teorema de la *determinación de la derivación en un único paso.*

Teorema *Terminación de la evaluación*: En nuestro lenguaje booleano, para cada término `t` existe una forma normal `t'` tal que `t` $\rightarrow^*$ `t'`. Este razonamiento puede ser utilizado para probar la terminación de un programa.

La prueba de este teorema es intuitiva, teniendo en cuenta que cada paso de la evaluación reduce el tamaño del término, y éste no es infinito.

Ejercicio: Supongamos que añadimos la regla `if true then` t_2 `else` t_3 a nuestra gramática. Cuál de los cuatro teoremas anteriores se siguen cumpliendo?

Vamos a extender ahora la definición de evaluación a las expresiones aritméticas. Los términos han de tener la forma:

```
t::= ...
     0
     succ t
     pred t
     iszero t

v::= ...
     nv

nv::=
      0
      succ nv
```

El resultado final de evaluar una expresión aritmética será un valor numérico, o bien el 0 o bien el sucesor de otro valor númerico.

Las reglas de evaluación son las siguientes:

$$\frac{t_1 \rightarrow t_1'}{\texttt{succ } t_1 \rightarrow \texttt{succ } t_1'} \quad (\texttt{E-Succ})$$

$$\texttt{pred 0} \rightarrow 0 \quad (\texttt{E-PredZero})$$

$$\texttt{pred (succ } nv_1\texttt{)} \rightarrow nv_1 \quad (\texttt{E-PredSucc})$$

$$\frac{t_1 \rightarrow t_1'}{\texttt{pred } t_1 \rightarrow \texttt{pred } t_1'} \quad (\texttt{E-Pred})$$

$$\texttt{iszero 0} \rightarrow \texttt{true} \quad (\texttt{E-IsZeroZero})$$

$$\texttt{iszero (succ } nv_1\texttt{)} \rightarrow \texttt{false} \quad (\texttt{E-IsZeroSucc})$$

$$\frac{t_1 \rightarrow t_1'}{\texttt{iszero } t_1 \rightarrow \texttt{iszero } t_1'} \quad (\texttt{E-IsZero})$$

En la regla `E-PredSucc`, el hecho de que la parte izquierda de la regla se escriba `pred(succ` nv_1`)` en lugar de `pred(succ` t_1 implica que la regla no puede ser utilizada para evaluar `pred (succ (pred 0))` a `pred 0`, pues esto requeriría instanciar la metavariable nv_1 a `pred 0`, que no es un valor numérico. Por lo tanto, para evaluar el `pred (succ (pred 0))` tenemos que utilizar el siguiente árbol de derivación:

$$\frac{}{\texttt{pred 0} \rightarrow \texttt{0}}\ \texttt{E-PredZero}$$

$$\frac{}{\texttt{succ (pred 0)} \;\rightarrow\; \texttt{succ 0}}\ \texttt{E-Succ}$$

$$\frac{}{\texttt{pred(succ (pred 0))} \;\rightarrow\; \texttt{pred (succ 0)}}\ \texttt{E-Pred}$$

Ejercicio: Demostrar que el teorema de la *determinación de la evaluación en un paso* se cumple para las expresiones aritméticas. Es decir, demostrar que si `t` $\rightarrow$ `t'` y `t` $\rightarrow$ `t''`, entonces `t'` = `t''`;

Al formalizar la semántica operacional de un lenguaje estamos obligados a especificar la respuesta ante cualquier término, como por ejemplo, `succ false`. Según nuestro conjunto de reglas, `succ false` no está definido (por lo tanto está en forma normal). De este tipo de términos decimos que están *atascados*.

Definición: Decimos que un término `t` está atascado si está en forma normal pero no es un valor.

Los términos *atascados* nos dan una noción de error en tiempo de ejecución, cuando la semántica operacional no sabe qué hacer porque se ha llegado a un "estado inconsistente".

Ejercicio: Una alternativa a la hora de definir estados inconsistentes en una máquina abstracta sería definir un nuevo término llamado `wrong` y aumentar la semántica operacional con reglas que generen explícitamente `wrong` cuando algún término se quede *atascado*.

Para ello, habría que introducir dos nuevas categorías semánticas:

```
badnat ::=
    wrong
    true
    false

badbool ::=
    wrong
    nv
```

También tendríamos que aumentar la relación de evaluación con las siguientes reglas:

```
if badbool then t1 then t2 → wrong  (E-If-Wrong)
succ badnat → wrong                 (E-Succ-Wrong)
pred badnat → wrong                 (E-Pred-Wrong)
iszero badnat → wrong               (E-IsZero-Wrong)
```

Ejercicio: Existen dos estilos alternativos en la semántica operacional. El utilizado en este curso se denomina estilo *paso corto*, ya que la evaluación se define en sucesivos pasos individuales hasta que un término se evalúe a un valor. Existe

también un estilo de *paso largo* en la que evaluación se define directamente de los términos a los valores finales. En el estilo de *paso largo*, nuestro lenguaje se definiría de la siguiente manera:

$$\mathtt{v} \Downarrow \mathtt{v} \qquad (\mathtt{B-Value})$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{true} \quad \mathtt{t}_2 \Downarrow \mathtt{v}_2}{\mathtt{if\ t}_1\ \mathtt{then\ t}_2\ \mathtt{else\ t}_3 \Downarrow \mathtt{v}_2} \qquad (\mathtt{B-IfTrue})$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{false} \quad \mathtt{t}_3 \Downarrow \mathtt{v}_3}{\mathtt{if\ t}_1\ \mathtt{then\ t}_2\ \mathtt{else\ t}_3 \Downarrow \mathtt{v}_3} \qquad (\mathtt{B-IfFalse})$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{nv}_1}{\mathtt{succ\ t}_1 \Downarrow \mathtt{succ\ nv}_1} \qquad (\mathtt{B-Succ})$$

$$\frac{\mathtt{t}_1 \Downarrow 0}{\mathtt{pred\ t}_1 \Downarrow 0} \qquad (\mathtt{B-PredZero})$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{succ\ nv}_1}{\mathtt{pred\ t}_1 \Downarrow \mathtt{nv}_1} \qquad (\mathtt{B-PredSucc})$$

$$\frac{\mathtt{t}_1 \Downarrow 0}{\mathtt{iszero\ t}_1 \Downarrow \mathtt{true}} \qquad (\mathtt{B-IsZeroZero})$$

$$\frac{\mathtt{t}_1 \Downarrow \mathtt{succ\ nv}_1}{\mathtt{iszero\ t}_1 \Downarrow \mathtt{false}} \qquad (\mathtt{B-IsZeroSucc})$$