Hacia un Sistema de Tipos Estático y Dinámico[1]

Francisco Ortin[2], Daniel Zapico[3]

Departamento de Informática
Universidad de Oviedo
Oviedo, España

Abstract

Si bien la comprobación estática de tipos ofrece robustez, legibilidad, abstracción y eficiencia en el desarrollo de software, los sistemas de tipos dinámicos también otorgan una elevada flexibilidad en tiempo de ejecución, permitiendo al programador realizar un desarrollo más interactivo y ágil, idóneo para el desarrollo rápido de prototipos. En este artículo se presenta el sistema de tipos del núcleo de un lenguaje de programación cuyo principal objetivo es aunar los beneficios de ambas aproximaciones en un mismo lenguaje. Manteniendo la declaración explícita de tipos, se añade la declaración implícita para realizar reconstrucción de tipos estática y dinámica en un mismo sistema de tipos. Se combinan la inferencia de tipos dirigida por sintaxis con un sistema basado en restricciones, utilizando tipos unión como principal mecanismo para distinguir entre inferencia de tipos dinámica y estática. Las principales ventajas obtenidas son la detección temprana de errores en código dinámico, la integración de código estático y dinámico en un mismo lenguaje de programación y una notable optimización de código.

Keywords: Sistemas de tipos, lenguajes dinámicos, tipos unión, sistemas de tipos basados en restricciones, reconstrucción de tipos.

1 Introducción

Los sistemas con comprobación estática de tipos han demostrado ser una herramienta indispensable en el desarrollo de software, ofreciendo al programador ventajas tales como la detección temprana de errores, mejor documentación y abstracción, y un incremento de oportunidades para la optimización de código [21]. No obstante, los lenguajes con comprobación dinámica de tipos ofrecen una gran flexibilidad en tiempo de ejecución, valiosa en el desarrollo rápido de prototipos cuando existen numerosos cambios en los requisitos, o en aplicaciones que acceden a información (u otros sistemas) que se encuentren bajo un cambio continuo.

[1] Este trabajo ha sido parcialmente financiado por *Microsoft Research* con el proyecto titulado *Extending dynamic features of the SSCLI*, premiado en 2006 en la llamada a proposición de projectos de investigación *Phoenix and SSCLI, Compilation and Managed Execution.* Este proyecto también ha sido financiado por el Ministerio de Ciencia e Innovación dentro del Plan Nacional de I+D, con el proyecto TIN2008-00267 titulado *Mejora del Rendimiento y Robustez de los Lenguajes Dinámicos para el Desarrollo de Software Eficiente, Escalable y Fiable.*

[2] Email: ortin@lsi.uniovi.es

[3] Email: danzapico@gmail.com

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Puesto que ambos sistemas de tipos (estáticos y dinámicos) ofrecen beneficios, ha habido estudios previos orientados a obtener las ventajas de ambos, siguiendo la consigna de *comprobación estática de tipos cuando sea posible y dinámica cuando sea necesaria* [12]. Uno de los primeros estudios fue *Soft Typing* [3], que aplicaba comprobación estática a un lenguaje con tipado dinámico tal como *Scheme* [4]. No obstante, el programador no puede controlar sobre qué fragmentos de código desea que se lleve a cabo la comprobación, y la información estática de tipos no es utilizada para optimizar el código generado. La aproximación de [1] añade el tipo `Dynamic` al cálculo lambda, introduciendo dos operaciones de conversión (`dynamic` y `typecase`), produciendo un código prolijo y acoplado al dinamismo de la aplicación. Los trabajos de *Quasi-Static Typing* [25], *Hybrid Typing* [5] y *Gradual Typing* [23] realizan conversiones implícitas de estático a dinámico y viceversa, utilizando reglas de subtipado en las dos primeras y la relación de consistencia entre tipos en la última. La principal diferencia entre estos sistemas y el presentado en este artículo es que nuestro trabajo realiza comprobación de tipos incluso para los tipos dinámicos, pudiendo generar errores de tipo en código dinámico, aumentando así su robustez. Cabe destacar que el trabajo de *Gradual Typing* ya identificó un sistema de inferencia de tipos basado en unificación para la resolución de restricciones como mecanismo de fusión del tipado dinámico y estático [24]. No obstante, en la aproximación de Siek un tipo dinámico se puede siempre convertir a cualquier tipo estático, puesto que no se infiere tipo alguno cuando éste es dinámico. Los estudios teóricos para aunar los beneficios de ambos sistemas de tipos también se han incluido parcialmente en la implementación de determinados lenguajes tales como *Boo*, *Visual Basic .Net*, *Cobra*, *Dylan*, *Strongtalk* o el futuro *C#* 4.0.

Este artículo presenta el sistema de tipos del núcleo del lenguaje de programación *StaDyn*, cuyo principal objetivo es ofrecer las ventajas de ambos sistemas de tipos. El programador podrá obtener la robustez, fiabilidad y eficiencia de los sistemas estáticos y, sin sacrificar estas ventajas, también podrá beneficiarse de la flexibilidad de los lenguajes dinámicos. Puesto que la comprobación de tipos también se lleva a cabo en el código dinámico, éste será más seguro y eficiente, ofreciendo además una interoperabilidad directa entre el código dinámico y estático. En función del tipo de aplicación (o parte de ésta) que estemos desarrollando, podremos seleccionar el dinamismo apropiado para su desarrollo. Para ello, hemos definido un sistema de tipos sensible al flujo de ejecución [6], con inferencia de tipos dinámicos, que une la declaración explícita e implícita de tipos basada en restricciones [14], y utiliza tipos unión [20] como principal mecanismo para distinguir la comprobación e inferencia de tipos estática y dinámica.

El resto del artículo se estructura como sigue. La siguiente sección describe la sintaxis abstracta del núcleo del lenguaje, así como el código de ejemplo que utilizaremos a lo largo de todo el artículo. La Sección 3 detalla el sistema de tipos y en la Sección 4 presentamos una descripción somera de la implementación del algoritmo de comprobación de tipos. Finalmente las conclusiones se exponen en la Sección 5.

Programa	P	::=	$F^*\ D^*\ S+$
Función	F	::=	$T\ id\ (T\ id)^*\ D\ S\ R?$
Declaración	D	::=	$T\ id$
Sentencia	S	::=	E | `if` $E\ S+\ S^*$ | `while` $E\ S^*$
Retorno	R	::=	`return` E
Expresión	E	::=	id | id `(` E^* `)` | $E \oplus E$ | $E \otimes E$ | $E \# E$ | $E = E$ | $E.id$ | $E[E]$ | `new` $\{(id = (E \mid VT))^*\}$ | `true` | `false` | $ConstanteEntera$
Tipo sintáctico	T	::=	`int` | `bool` | `Array(`TI`)` | `{` $(id{:}TI)^*$ `}`
Variable de tipo	VT	::=	$Din?\ X_i$
Dinamismo	Din	::=	`sta` | `dyn`
Tipo interno	TI	::=	T | $T \times \ldots \times T \rightarrow T$ | $Rest$ | $Din?\ T \vee \ldots \vee T$
Tipo de restricción	TR	::=	T | $[(id{:}TR)^*]$ | $Din?\ T \vee \ldots \vee T$ | $Din?\ T \wedge \ldots \wedge T$
Restricción	$Rest$	::=	$TR \leq TR$ | $TR \leftarrow TR$

Figure 1. Sintaxis abstracta del núcleo del lenguaje.

2 Sintaxis

El lenguaje descrito en este artículo es la parte central del lenguaje de programación *StaDyn*, un lenguaje orientado a objetos que posee características heredadas de *C#* tales como herencia, ocultación de miembros, sobrecarga de métodos, polimorfismo de inclusión y enlace dinámico. La parte central del lenguaje presentada en este artículo no incluye estas características, pero modela sus elementos centrales tales como objetos, funciones, vectores, asignaciones y expresiones aritméticas, relacionales y de comparación. También se añaden a *C#* variables de tipo (ofreciendo reconstrucción de tipos mediante un polimorfismo implícito [11]), y la especificación del dinamismo de este tipo de referencias.

La gramática abstracta del núcleo del lenguaje se muestra en la primera parte de la Figura 1. `*` representa repetición de cualquier número de elementos, `?` opcionalidad y `|` alternativa. Puesto que se ha incluido la asignación como una expresión, los nodos expresión del árbol de sintaxis abstracta se crearán con un valor lógico indicando si es hijo directo a la izquierda de una expresión de asignación. La función `izqAsign` que utilizaremos posteriormente en las reglas de inferencia devuelve este valor lógico. Aunque el programador podrá utilizar la sentencia `return` igual que en *C#*, la sintaxis abstracta sólo reconoce ésta al final de una función. Este proceso de transformación lo lleva a cabo el analizador sintáctico para facilitar la inferencia de tipos en las estructuras de control (Sección 3.7).

La Figura 2 muestra un programa de ejemplo del núcleo del lenguaje. Aquellas líneas de código que posean un comentario de error son ejemplos de programas rechazados por nuestro sistema de tipos. En la parte derecha se muestra parte del entorno y restricciones generados, que serán referenciados a lo largo del artículo.

3 Sistema de Tipos

Los tipos utilizados en el núcleo de nuestro sistema se muestran en la segunda parte de la Figura 1. Los tipos sintácticos son los que el programador puede utilizar directamente (aceptados por el analizador sintáctico); los internos son aquéllos que utiliza el compilador, ocultos al programador; los de restricción sólo pueden aparecer en las restricciones, totalmente transparentes al programador. Los objetos se especifican mediante la colección de sus campos (atributos), no incluyendo dentro

```
01:  void f(int a, var p, var q) {          Γ(p):X1, Γ(q):X2
02:    var b, s, e; dyn var d; var[] ve;    Γ(b):X3, Γ(s):X4, Γ(e):X5, Γ(d):dyn X6, Γ(ve):Array(X7)
03:    b = a>=3;                            Γ(X3):bool
04:    if (b) d=e=s=p&&true&&q;             X1<:bool, X2<:bool, Γ(X5):bool, Γ(dyn X6):bool, Γ(X4):bool
05:    else d = s = a + p;                  X1<:int, Γ(X4):int, Γ(dyn X6):int
06:    a = s + 1;          // * Error       Γ(X4):int∨bool
07:    a = d + 1;                           Γ(dyn X6):dyn int∨bool
08:    a = s;              // * Error
09:    ve[2] = new { atributo = 3 };        Γ(X7):{atributo:int}
10:    vector(ve);
11:  }                                      Γ(X7):{atributo:int}∨int∨bool
12:  void setCampo(var obj, var valor) {    Γ(obj):X8, Γ(valor):X9
13:    obj.campo = valor;                   X8<:[campo:X18], X18←X9
14:  }                                      Γ(setCampo):X8xX9→void | X8<:[campo:X18], X18←X9
15:  bool sensibleFlujo() {
16:    int n; bool b; var obj;
17:    obj = new { campo = var };           Γ(obj):{campo:X10}
18:    obj.campo = 3;                       Γ(X10): int
19:    n = obj.campo + 45;
20:    obj.campo = obj.campo%2==0;          Γ(X10): bool
21:    return obj.campo;
22:  }
23:  void vector(var[] w) {                 Γ(w):Array(X11)
24:    var[] v;                             Γ(v):Array(X12)
25:    a = v[3];           // * Error
26:    v[0] = w[0] = 0;                     Γ(X12):int, Γ(X11):X11∨int
27:    v[1] = w[1] = true;                  Γ(X12):int∨bool, Γ(X11):X11∨int∨bool
28:  }                                      Γ(vector):Array(X11)→void | Γ(X11):X11∨int∨bool
29:  void setCampoInc(var p, var v) {       Γ(p):X13, Γ(v):X14
30:    int a; var obj;                      Γ(a):X15, Γ(obj):X16
31:    setCampo(p, v);
32:    obj = new {campo=var,otro=3};        Γ(X16):{campo:X17, otro:int}
33:    setCampo(obj, 3);                    Γ(X17):int
34:    a = obj.campo + 1;
35:  }
36:  void main() { sensibleFlujo(); }
```

Figure 2. Ejemplo de programa, siguiendo la sintaxis abstracta.

de éstos a las funciones (métodos). Aunque las variables de tipo se especifican con la palabra reservada `var` (introducida en C# 3.0 para otro propósito), el analizador sintáctico numerará éstas secuencialmente, asignándole un número distinto a cada una. El dinamismo de los tipos es estático por omisión. Finalmente, los tipos unión son internos al compilador, y los tipos miembro (tipos objeto con subtipado estructural) sólo pueden aparecer en las restricciones.

3.1 Entorno, Contexto y Restricciones

En la especificación del sistema de tipos utilizaremos aseveraciones con la siguiente estructura general: $\Gamma; \Omega \vdash E : T \mid C; \Gamma'$. Esta aseveración tiene por significado que, bajo las restricciones C, el entorno de entrada Γ y el contexto Ω, la expresión E tiene un tipo T, produciendo un entorno de salida Γ'.

Los entornos (Γ) asocian tipos a los identificadores dentro de un ámbito; también asocian a cada variable de tipo su tipo inferido (si existe). Dado el código de la Figura 2, para el ámbito de la función `f`, se cumple que Γ(a):`int` y $\Gamma(X_3)$:`bool`, siendo X_3 el tipo de la variable `b`. El contexto (Ω) alberga la información relativa a la función que se está procesando actualmente, para que se puedan hacer determinadas comprobaciones de tipo. $\Omega._{locals}$ colecciona el conjunto de identificadores que constituyen los parámetros de la función actual, y $\Omega._{params}$ sus parámetros; $\Omega._{tr}$ almacena el tipo de retorno declarado por la función, y $\Omega._{tip}$ los tipos inferidos a partir de los parámetros (se detallará en la Sección 3.3).

Las restricciones pueden ser de dos tipos distintos y su estructura está definida

mediante última producción de Figura 1. La restricción de subtipado requiere que un tipo sea un subtipo de otro dado [13]. Las de asignación no sólo comprueban que se pueda realizar una asignación, sino que también asocian un tipo a una variable de tipo en el entorno indicado. En la línea 4 del ejemplo de código de la Figura 2, se generan dos restricciones de subtipado para las variables `q` y `p`; ambas deberán ser subtipos de `bool` ($X_1 \leq$ `bool`, $X_2 \leq$ `bool`), puesto que forman parte de una expresión lógica. En la línea 13, se genera una restricción de asignación indicando que tras la invocación a `setCampo` se deberá asociar el tipo de `valor` (X_9) a la variable de tipo del `campo` del parámetro (X_{18}).

Las aseveraciones poseen entornos de entrada y salida. El entorno de entrada es necesario para sustituir las expresiones cuyo tipo es una variable de tipo asociada a otro tipo. Esto permite que el sistema de tipos sea sensible al flujo de ejecución [6], puesto que los identificadores y las variables de tipo pueden cambiar su tipo asociado en función del flujo de ejecución [7]. Un ejemplo de este comportamiento se aprecia en la función `sensibleFlujo` de la Figura 2. En la línea 17, el tipo del campo de `obj` es una variable de tipo libre (X_{10}). En la asignación de la línea 18 esta variable de tipo pasa a estar asociada a `int`, siendo, por tanto, correcta la línea 19. En la línea 20 su nuevo tipo pasa a ser `bool`, compilando sin errores la línea 21. Vemos pues cómo una misma variable puede tener varios tipos en el mismo ámbito, siendo éstos dependientes del flujo de ejecución. Este proceso también es aplicable a las estructuras de control (Sección 3.7).

3.2 Funciones

Las reglas de inferencia, en el caso de programa, declaraciones y funciones, se tratan más de reglas de comprobación de correcta formación que de inferencia de tipos: $\diamond$ indica la validez semántica de un término. No obstante, las reglas mostradas en la Figura 3 infieren restricciones y entornos de salida necesarias para la inferencia de tipos del resto del sistema. La regla T-Func infiere en su entorno de salida el tipo de la función ($T_1 \times \ldots \times T_n \to T \mid C$) para que ésta pueda invocarse en adelante. Como puede apreciarse en la Figura 3, la función incluye las restricciones que deberán satisfacer sus argumentos en el momento de la invocación. Cabe destacar que la declaración de una función no genera restricciones y el único elemento que añade al entorno de salida es la asociación de su propio tipo a su identificador. Esto significa que las restricciones, tipos de los identificadores y variables de tipo incluidos en todo entorno son siempre locales a las funciones. El resto de reglas no genera restricción alguna y los entornos sintetizados son la entrada de los siguientes, consiguiendo así establecer una sensibilidad al flujo de ejecución.

3.3 Contexto

Como veremos posteriormente, la información relativa a la función procesada que se almacena en el contexto (Ω) es necesaria para realizar la comprobación de tipos. La Figura 4 muestra cómo se calculan sus elementos. En la declaración de una función (T-Func), se almacenan en $\Omega._{locals}$ las variables locales, en $\Omega._{params}$ los parámetros y en $\Omega._{tr}$ el tipo de retorno declarado para la función.

Más complejo es el cálculo de aquellos tipos que se han inferido a partir de los

$$\text{(T-Func)}\quad \frac{\texttt{id} \notin dom(\Gamma) \qquad \Gamma;\Omega \vdash D : \diamond \mid \emptyset;\Gamma' \qquad \texttt{id}_i \notin dom(\Gamma')^{i\in 1..n} \qquad \Gamma',\texttt{id}_1 : T_1 \ldots \texttt{id}_n : T_n;\Omega \vdash S : \diamond \mid C;\Gamma'' \qquad \Gamma''' = \Gamma,\texttt{id} : T_1 \times \ldots \times T_n \to T \mid C}{\Gamma;\Omega \vdash T\ \texttt{id}(T_1\ \texttt{id}_1 \ldots T_n\ \texttt{id}_n)\ D\ S : \diamond \mid \emptyset;\Gamma'''}$$

$$\text{T-Decl}\quad \frac{\texttt{id} \notin dom(\Gamma) \qquad \Gamma' = \Gamma,\texttt{id} : T}{\Gamma;\Omega \vdash T\ \texttt{id} : \diamond \mid \emptyset;\Gamma'}$$

$$\text{T-Decls}\quad \frac{\Gamma;\Omega \vdash D_1 : \diamond \mid \emptyset;\Gamma' \qquad \Gamma';\Omega \vdash D_2 : \diamond \mid \emptyset;\Gamma''}{\Gamma;\Omega \vdash D_1\ D_2 : \diamond \mid \emptyset;\Gamma''}$$

$$\text{T-Funcs}\quad \frac{\Gamma;\Omega \vdash F_1 : \diamond \mid \emptyset;\Gamma' \qquad \Gamma';\Omega \vdash F_2 : \diamond \mid \emptyset;\Gamma''}{\Gamma;\Omega \vdash F_1\ F_2 : \diamond \mid \emptyset;\Gamma''}$$

$$\text{T-Prog}\quad \frac{\Gamma;\Omega \vdash F : \diamond \mid \emptyset;\Gamma' \qquad \Gamma';\Omega \vdash S : \diamond \mid \emptyset;\Gamma''}{\vdash P : \diamond \mid \emptyset}$$

Figure 3. Reglas de inferencia de programa, función y declaración.

$$\text{(Ω-Func)}\quad \frac{\Gamma;\Omega.params = \texttt{id}_1 \ldots \texttt{id}_p, \Omega.locals = \texttt{id}_{p+1} \ldots \texttt{id}_{p+l}, \Omega.tr = T, \Omega.tip = T_1 \ldots T_p \vdash S : \diamond}{\Gamma;\Omega \vdash T\ \texttt{id}(T_1\ \texttt{id}_1 \ldots T_p\ \texttt{id}_p)\ T_{p+1}\ \texttt{id}_{p+1} \ldots T_{p+l}\ \texttt{id}_{p+l}\ \ S : \diamond}$$

$$\text{($\Omega.tip$-Field)}\quad \frac{\Gamma;\Omega_1 \vdash E : T_1 \qquad \Gamma \vdash T_1 \leq [\texttt{id} : T_2] \qquad T \in \Omega_1.tip \qquad \Omega_2.tip = T_2}{\Gamma;\Omega_1 \cup \Omega_2 \vdash E.\texttt{id} : T_2}$$

$$\text{($\Omega.tip$-Array)}\quad \frac{\Gamma;\Omega_1 \vdash E_1 : T_1 \qquad \Gamma \vdash T_1 \leq Array(T_2) \qquad T \in \Omega_1.tip \qquad \Omega_2.tip = T_2}{\Gamma;\Omega_1 \cup \Omega_2 \vdash E_1[E_2] : T_2}$$

Figure 4. Reglas de cálculo de Ω.

parámetros ($\Omega.tip$). Como se verá más adelante, este conjunto de tipos es necesario para realizar comprobaciones de expresiones de asignación, obtención de campos de objetos e indexación de vectores. Inicialmente, en la regla T-Func se introducen los tipos de los parámetros en $\Omega.tip$; en $\Omega.tip$-Field se añade el tipo del campo cuando el tipo del objeto pertenece a $\Omega.tip$; $\Omega.tip$-Array hace lo propio con los vectores.

3.4 Expresiones

En este apartado analizaremos las variables, expresiones aritméticas, de comparación, lógicas, el acceso a campos de objetos y la indexación de vectores. Las asignaciones y las invocaciones a funciones (ambas expresiones) serán analizadas en las secciones 3.6 y 3.9 respectivamente.

En la Figura 5 se muestran las reglas de inferencia de variables. Las funciones `vt` y `vtl` devuelven el conjunto de variables de tipo y variables de tipo libres existentes en un entorno dado. El predicado `izqAsign` indica si una expresión es el hijo directo a la izquierda (en el árbol de sintaxis abstracta) del operador de asignación.

Ninguna de las reglas genera restricción alguna ni añade elementos nuevos al entorno existente. La regla T-Var infiere el tipo de un identificador previamente declarado, cuando su tipo no es una variable de tipo. T-EVar tipa un identificador declarado con una variable de tipo, estando ésta ya asociada con un tipo concreto. Esto sucede, por ejemplo, en la línea 4 de la Figura 2, puesto que la variable de tipo de `b` (X_3) se asoció a `bool` (línea 3).

Tanto T-PVar como T-AVar infieren identificadores cuando tienen por tipo una variable de tipo. En el primer caso, se podrá utilizar un identificador si éste es un parámetro (y por tanto ya posee un valor) y no esté a la izquierda del operador de asignación. En el caso de T-AVar se permite la utilización de variables de tipo libres que no sean parámetros, siempre y cuando se les vaya a asignar un valor y, por tanto, su variable de tipo pase a estar enlazada con un tipo concreto. Por ejemplo,

$$\text{(T-Var)}\ \frac{\Gamma(\texttt{id}) : T \quad T \notin vt(\Gamma)}{\Gamma;\Omega \vdash \texttt{id} : T \mid \emptyset;\Gamma} \qquad \text{(T-EVar)}\ \frac{\Gamma(\texttt{id}) : X \quad \Gamma(X) : T}{\Gamma;\Omega \vdash \texttt{id} : T \mid \emptyset;\Gamma}$$

$$\text{(T-PVar)}\ \frac{\Gamma(\texttt{id}) : X \quad X \in vtl(\Gamma) \quad \texttt{id} \in \Omega._{params} \quad \neg izqAsign(\texttt{id})}{\Gamma;\Omega \vdash \texttt{id} : X \mid \emptyset;\Gamma} \qquad \text{(T-AVar)}\ \frac{\Gamma(\texttt{id}) : X \quad X \in vtl(\Gamma) \quad \texttt{id} \notin \Omega._{params} \quad izqAsign(\texttt{id})}{\Gamma;\Omega \vdash \texttt{id} : X \mid \emptyset;\Gamma}$$

Figure 5. Reglas de inferencia de variables.

$$\text{(T-Arit)}\ \frac{\Gamma;\Omega \vdash E_1 : T_1 \mid C_1;\Gamma' \quad \Gamma' \vdash T_1 \leq \texttt{int} \mid C_2;\Gamma'' \quad \Gamma'';\Omega \vdash E_2 : T_2 \mid C_3;\Gamma''' \quad \Gamma''' \vdash T_2 \leq \texttt{int} \mid C_4;\Gamma''''}{\Gamma;\Omega \vdash E_1 \oplus E_2 : \texttt{int} \mid C_1 \cup C_2 \cup C_3 \cup C_4;\Gamma''''}$$

$$\text{(T-Log)}\ \frac{\Gamma;\Omega \vdash E_1 : T_1 \mid C_1;\Gamma' \quad \Gamma' \vdash T_1 \leq \texttt{bool} \mid C_2;\Gamma'' \quad \Gamma'';\Omega \vdash E_2 : T_2 \mid C_3;\Gamma''' \quad \Gamma''' \vdash T_2 \leq \texttt{bool} \mid C_4;\Gamma''''}{\Gamma;\Omega \vdash E_1 \otimes E_2 : \texttt{bool} \mid C_1 \cup C_2 \cup C_3 \cup C_4;\Gamma''''}$$

$$\text{(T-Rel)}\ \frac{\Gamma;\Omega \vdash E_1 : T_1 \mid C_1;\Gamma' \quad \Gamma' \vdash T_1 \leq \texttt{int} \mid C_2;\Gamma'' \quad \Gamma'';\Omega \vdash E_2 : T_2 \mid C_3;\Gamma''' \quad \Gamma''' \vdash T_2 \leq \texttt{int} \mid C_4;\Gamma''''}{\Gamma;\Omega \vdash E_1 \# E_2 : \texttt{bool} \mid C_1 \cup C_2 \cup C_3 \cup C_4;\Gamma''''}$$

$$\text{(T-Objeto)}\ \frac{\Gamma;\Omega \vdash E : T_1 \mid C_1;\Gamma' \quad \Gamma' \vdash T_1 \leq [\texttt{id} : T_2] \mid C_2;\Gamma'' \quad T_2 \in vtl(\Gamma'') \wedge T_2 \notin \Omega._{tip} \Rightarrow izqAsign(E.\texttt{id})}{\Gamma;\Omega \vdash E.\texttt{id} : T_2 \mid C_1 \cup C_2;\Gamma''}$$

$$\text{(T-Vector)}\ \frac{\Gamma;\Omega \vdash E_1 : T_1 \mid C_1;\Gamma' \quad \Gamma' \vdash T_1 \leq Array(T) \mid C_2;\Gamma'' \quad \Gamma'';\Omega \vdash E_2 : T_2 \mid C_3;\Gamma''' \quad \Gamma''' \vdash T_2 \leq \texttt{int} \mid C_4;\Gamma'''' \quad T \in vtl(\Gamma'''') \wedge T \notin \Omega._{tip} \Rightarrow izqAsign(E_1[E_2])}{\Gamma;\Omega \vdash E_1[E_2] : T \mid C_1 \cup C_2 \cup C_3 \cup C_4;\Gamma''''}$$

Figure 6. Reglas de inferencia de expresiones.

la utilización de la variable `v` en la línea 25 de la Figura 2 no está permitida porque es una variable de tipo libre, no es un parámetro, y no está a la izquierda de la asignación (T-AVar). Sin embargo, la variable `p` sí puede utilizarse en la línea 4, al tratarse de un parámetro (T-PVar).

Las reglas de inferencia de expresiones aritméticas, lógicas y relacionales (Figura 6) se apoyan en la relación de subtipado entre tipos (siguiente sección). En el caso de las aritméticas y relacionales, las dos subexpresiones han de ser subtipos de entero; en el caso de las lógicas han de promocionar a `bool`. Los entornos de salida se utilizan como entornos de entrada de las subexpresiones secuencialmente, siendo el último entorno de salida el propio de la expresión. Las restricciones que deberán satisfacerse para inferir el tipo de la expresión serán la unión de todas las restricciones creadas en las cuatro operaciones del antecedente.

En el acceso a un campo de un objeto (T-Objeto), el tipo del objeto tiene que promocionar al miembro [`id`:T_2]. Un miembro es un tipo interno que denota cualquier objeto que al menos contenga los campos del miembro (sus reglas de subtipado se describen en la siguiente sección), de igual forma que los tipos objeto descritos en [2]. Además, si el tipo del miembro es una variable de tipo libre y no está inferida a partir de los parámetros, ésta sólo podrá estar a la izquierda de la asignación. A modo de ejemplo, la línea 13 de la Figura 2 es correcta porque, aunque el tipo del campo es una variable libre (X_{18}) y no está a la izquierda del operador de asignación, su tipo ha sido inferido a partir de los parámetros (pertenece a $\Omega._{tip}$).

T-Vector requiere que la primera expresión promocione a vector, el índice a entero y, al igual que con los objetos, si se infiere una variable de tipo libre no dependiente de parámetros, ésta tendrá que estar a la izquierda del operador de asignación. Un ejemplo de error de compilación debido a este predicado es la línea 25 de la Figura 2. El tipo de los elementos de `v` (X_{12}) es una variable libre no inferida a partir de los parámetros (es local), produciéndose un error de compilación por no

encontrarse a la izquierda del operador de asignación.

3.5 Subtipado

Las reglas de subtipado utilizan entornos de entrada y salida y generan restricciones. El entorno de entrada es necesario para sustituir las expresiones que tienen por tipo una variable de tipo y ésta ha sido asociada a otro tipo. En este caso, el tipo final de la expresión será este último (T-Sub mostrada en la Figura 7). El entorno de salida se utiliza para asociar una variable de tipo libre a un tipo dado, cuando ésta tenga que promocionar a dicho tipo. Este comportamiento es precisamente el de la regla S-VTL de la Figura 7. Para esta regla, además, se genera una restricción de promoción. De este modo, el funcionamiento es inferir que la promoción es cierta si se satisface la condición creada. Para futuras expresiones, se sustituirá X por T, puesto que esta asociación ha sido añadida al entorno de salida.

La relación de subtipado es reflexiva y transitiva (Figura 7). Si las dos variables de tipo son libres, se genera una restricción de promoción (S-VTLs). Los constructores de tipo Objeto (S-Objeto) y vector (S-Vector) son covariantes. Sin embargo, un objeto es subtipo de otro dado cuando ambos tienen igual número de campos, los campos poseen las mismas etiquetas y cada tipo de éstos es covariante (S-Objeto).

La relación de subtipado entre miembros ofrece equivalencia estructural (S-EMiembro). Un objeto es un subtipo de un miembro si posee todos los campos del miembro y sus tipos son subtipos de los propios del miembro (S-Miembro). En la línea 32 de la Figura 2 se crea un objeto con dos campos (`campo` y `otro`) y éste promociona en la línea 33 al tipo [`campo`:X_{18}] (parámetro `obj` de la función `setCampo`). La promoción a nuevos miembros es posible gracias a la regla (S-IMiembro). Esta regla es necesaria porque en T-Objeto (Figura 6) añadimos un único campo tanto al entorno de salida como a las restricciones generadas. Esto significa que el tipo es un objeto que posee *al menos* el campo que aparece en el miembro, pudiendo poseer más (S-IMiembro).

El lenguaje *StaDyn* no ofrece aún funciones de orden superior (llamados delegados en C#), por lo que no se estudia el subtipado de funciones.

3.6 Asignaciones

Hemos utilizado cuatro reglas distintas para la inferencia de tipos de las asignaciones (Figura 8). T-Asign infiere el tipo cuando la expresión a la izquierda del operador no es una variable de tipo libre. En este caso, el tipo de la derecha ha de ser un subtipo del de la izquierda. En el caso de que se trate de una variable de tipo libre, ésta se asociará al tipo de la expresión de la derecha (T-VTLAsign).

T-OAsign tiene en cuenta el caso particular de asignación de campos de un objeto, siendo su tipo una variable de tipo. En este caso, el objeto ha de promocionar a un miembro. El nuevo tipo del campo será el tipo del valor asignado, obteniéndose así una inferencia de tipo sensible al flujo de ejecución (función `sensibleFlujo` de la Figura 2). Finalmente, si el tipo depende del parámetro, se generará una restricción de asignación indicando que, en la invocación a la función, el tipo del argumento cambiará al indicado en la restricción. Un ejemplo de este tipo de restricción es la

$$\text{(T-Sub)}\ \frac{\Gamma;\Omega \vdash E : X \mid C;\Gamma' \qquad \Gamma'(X) : T}{\Gamma;\Omega \vdash E : T \mid C;\Gamma'} \qquad \text{(S-Reflex)}\ \Gamma \vdash T \leq T \mid \emptyset;\Gamma \qquad \text{(S-Trans)}\ \frac{\Gamma \vdash T_1 \leq T_2 \mid C_1;\Gamma' \quad \Gamma' \vdash T_2 \leq T_3 \mid C_2;\Gamma''}{\Gamma \vdash T_1 \leq T_3 \mid C_1 \cup C_2;\Gamma''}$$

$$\text{(S-VTLs)}\ \frac{X_1 \in vtl(\Gamma) \qquad X_2 \in vtl(\Gamma) \qquad C = X_1 \leq X_2}{\Gamma \vdash X_1 \leq X_2 \mid C;\Gamma} \qquad \text{(S-VTL)}\ \frac{X \in vtl(\Gamma) \qquad T \notin vtl(\Gamma) \qquad C = X \leq T \qquad \Gamma' = \Gamma, X : T}{\Gamma \vdash X \leq T \mid C;\Gamma'}$$

$$\text{(S-Vector)}\ \frac{\Gamma \vdash T_1 \leq T_2 \mid C;\Gamma'}{\Gamma \vdash Array(T_1) \leq Array(T_2) \mid C;\Gamma'}$$

$$\text{(S-Objeto)}\ \frac{\Gamma \vdash T_1 \leq T_{n+1} \mid C_1;\Gamma_1 \ldots \ \Gamma_{n-1} \vdash T_n \leq T_{2n} \mid C_n;\Gamma_n}{\Gamma \vdash \{\mathtt{id}_1 : T_1 \ldots \mathtt{id}_n : T_n\} \leq \{\mathtt{id}_1 : T_{n+1} \ldots \mathtt{id}_n : T_{2n}\} \mid C_1 \cup \ldots \cup C_n;\Gamma_n}$$

$$\text{(S-Miembro)}\ \frac{\forall\, i \in [n+1, n+m], \exists\, j \in [1,n], \mathtt{id}_i = \mathtt{id}_j \wedge \Gamma_{i-n} \vdash T_i \leq T_j \mid C_i;\Gamma_{i-n+1}}{\Gamma_1 \vdash \{\mathtt{id}_1 : T_1 \ldots \mathtt{id}_n : T_n\} \leq [\mathtt{id}_{n+1} : T_{n+1} \ldots \mathtt{id}_{n+m} : T_{n+m}] \mid C_1 \cup \ldots \cup C_i;\Gamma_{m+1}}$$

$$\text{(S-EMiembro)}\ \frac{\Gamma(X) : [id_1 : T_1 \ldots id_n : T_n]}{\Gamma \vdash [id_i : T_i]^{i \in 1..n} \leq X \mid \emptyset;\Gamma} \qquad \text{(S-IMiembro)}\ \frac{\Gamma(X) : [id_1 : T_1 \ldots id_n : T_n] \quad \Gamma' = \Gamma, X : [id_1 : T_1 \ldots id_{n+1} : T_{n+1}] \quad C = X \leq [id_1 : T_1 \ldots id_{n+1} : T_{n+1}]}{\Gamma \vdash X \leq [id_{n+1} : T_{n+1}] \mid C;\Gamma'}$$

Figure 7. Reglas de subtipado.

$$\text{(T-Asign)}\ \frac{\Gamma;\Omega \vdash E_1 : T_1 \mid C_1;\Gamma' \qquad T_1 \notin vtl(\Gamma') \qquad \Gamma';\Omega \vdash E_2 : T_2 \mid C_2;\Gamma'' \qquad \Gamma'' \vdash T_2 \leq T_1 \mid C_3;\Gamma'''}{\Gamma;\Omega \vdash E_1 = E_2 : T_1 \mid C_1 \cup C_2 \cup C_3;\Gamma'''}$$

$$\text{(T-VTLAsign)}\ \frac{\Gamma;\Omega \vdash E_1 : X \mid C_1;\Gamma' \qquad X \in vtl(\Gamma') \qquad \Gamma';\Omega \vdash E_2 : T \mid C_2;\Gamma'' \qquad \Gamma''' = \Gamma'', X : T}{\Gamma;\Omega \vdash E_1 = E_2 : T \mid C_1 \cup C_2;\Gamma'''}$$

$$\text{(T-OAsign)}\ \frac{\Gamma;\Omega \vdash E_1 : T_1 \mid C_1;\Gamma' \qquad \Gamma' \vdash T_1 \leq [\mathtt{id} : X] \mid C_2 \qquad X \in vt(\Gamma') \qquad \Gamma;\Omega \vdash E_2 : T_2 \mid C_3;\Gamma'' \qquad \Gamma''' = \Gamma'', X : T_2 \qquad \mathtt{if}\ X \in \Omega_{.tip}, \mathtt{then}\ C_4 = X \leftarrow T_2, \mathtt{else}\ C_4 = \emptyset}{\Gamma;\Omega \vdash E_1.\mathtt{id} = E_2 : T_2 \mid C_1 \cup C_2 \cup C_3 \cup C_4;\Gamma'''}$$

$$\text{(T-VAsign)}\ \frac{\Gamma;\Omega \vdash E_1[E_2] : X \mid C_1;\Gamma' \qquad X \in vt(\Gamma') \qquad \Gamma';\Omega \vdash E_3 : T \mid C_2;\Gamma'' \qquad \Gamma''' = \Gamma'', X : \Gamma''(X) \vee T \qquad \mathtt{if}\ X \in \Omega_{.tip}, \mathtt{then}\ C_3 = X \leftarrow X \vee \Gamma''(X) \vee T, \mathtt{else}\ C_3 = \emptyset}{\Gamma;\Omega \vdash E_1[E_2] = E_3 : T \mid C_1 \cup C_2 \cup C_3;\Gamma'''}$$

Figure 8. Reglas de inferencia de asignaciones.

generada en la función `setCampo` de la Figura 2 ($X_{18} \leftarrow X_9$). La invocación a esta función con otro objeto (línea 33) hace que el tipo del `campo` cambie tras la llamada (por ello la línea 34 es correcta).

Para los vectores cuyos elementos sean variables de tipo (T-VAsign), el nuevo tipo de sus elementos será el tipo unión formado por el que tenía previamente más el tipo asignado. Un tipo unión indica la posibilidad de tener alguno de los tipos que se hallan dentro de él [20]. Por ello en la línea 27 de la Figura 2, el tipo de `v` es un vector de enteros o booleanos (`int` $\vee$ `bool`), ya que puede albergar ambos. En el caso de que la variable de tipo dependa de los parámetros de la función, se generará una restricción con el tipo unión anterior, añadiendo además la propia variable de tipo. Éste es el caso de la variable `w` de la función `vector` de la Figura 2 (línea 27); de forma distinta a `v`, el tipo de `w` (X_{11}) se incluye a sí mismo en su tipo unión ($X_{11} \vee$ `int` $\vee$ `bool`), indicando así que el tipo que se utilice en la invocación se añadirá como elemento de la unión. Por ello, el tipo que tendrá `w` en la invocación producida en la línea 10 será {`atributo` : `int`} $\vee$ `int` $\vee$ `bool` ya que la variable `ve` poseía el tipo {`atributo`:`int`} en la llamada.

$$\text{(T-Exp)}\ \frac{\Gamma;\Omega \vdash E : T \mid C;\Gamma'}{\Gamma;\Omega \vdash E : \diamond \mid C;\Gamma'} \qquad \text{(T-Return)}\ \frac{\Gamma;\Omega \vdash E : T \mid C_1;\Gamma' \qquad \Gamma' \vdash T \leq \Omega._{tr} \mid C_2}{\Gamma;\Omega \vdash \texttt{return}\ E : \diamond \mid C_1 \cup C_2;\Gamma'}$$

$$\text{(T-If)}\ \frac{\begin{array}{c}\Gamma;\Omega \vdash E : T \mid C_1;\Gamma' \qquad \Gamma' \vdash T \leq \texttt{int} \mid C_2;\Gamma'' \\ \Gamma'';\Omega \vdash S_1 : \diamond \mid C_3;\Gamma''' \qquad \Gamma'';\Omega \vdash S_2 : \diamond \mid C_4;\Gamma''''\end{array}}{\Gamma;\Omega \vdash \texttt{if}\ E\ S_1\ S_2 : \diamond \mid C_1 \cup C_2 \cup unir(C_3, C_4);\Gamma''' \cup \Gamma''''}$$

$$\text{(T-While)}\ \frac{\begin{array}{c}\Gamma;\Omega \vdash E : T \mid C_1;\Gamma' \qquad \Gamma' \vdash T \leq \texttt{int} \mid C_2;\Gamma'' \\ \Gamma'';\Omega \vdash S : \diamond \mid C_3;\Gamma'''\end{array}}{\Gamma;\Omega \vdash \texttt{while}\ E\ S : \diamond \mid C_1 \cup C_2 \cup unir(C_3, \emptyset);\Gamma'' \cup \Gamma'''}$$

Figure 9. Reglas de inferencia de sentencias.

$$\text{(}\Gamma\text{-}\cup\text{)}\ \frac{\begin{array}{c}\forall \texttt{id} \in dom(\Gamma'), \texttt{id} \in dom(\Gamma'') \wedge \Gamma'(\texttt{id}) = \Gamma''(\texttt{id}) \\ \forall \texttt{id} \in dom(\Gamma''), \texttt{id} \in dom(\Gamma') \wedge \Gamma''(\texttt{id}) = \Gamma'(\texttt{id}) \qquad avt(\Gamma) = unir(avt(\Gamma'), avt(\Gamma''))\end{array}}{\Gamma' \cup \Gamma'' = \Gamma}$$

Figure 10. Definición de la operación $\cup$ sobre entornos.

3.7 Sentencias

Las sentencias que incluye nuestro lenguaje son `return`, `if` y `while`, así como las expresiones. La Figura 9 muestra las reglas de inferencia que comprueban la validez semántica de estas expresiones. Una expresión constituirá una sentencia cuando para la expresión sea posible inferir un tipo; las restricciones y tipos de salida serán los generados por la expresión. En el caso de la sentencia `return`, se ha de cumplir que la expresión de retorno sea un subtipo del tipo declarado por el programador.

Las sentencias `if` y `while` poseen saltos en el flujo de ejecución, dificultando la inferencia de tipos. Por ello se definen la función `unir` sobre restricciones y la unión de entornos, que tienen en cuenta la información inferida en cada ramal de ejecución, componiéndola en un nuevo entorno y conjunto de restricciones. Por ejemplo, el tipo de la variable `s` (`int` $\vee$ `bool`) en la línea 6 de la Figura 2 se compone a partir de los tipos que tiene en la línea 4 (`bool`) y 5 (`int`). Lo mismo sucede con las restricciones: la restricción final del tipo de la variable `p` ($X_1 \leq$ `int` $\vee$ `bool`) se forma teniendo en cuenta que en la línea 4 se genera una restricción de promoción a `int` y la línea 5 requiere que sea subtipo de `bool`.

La unión de entornos utilizada en la Figura 9 se apoya en la función `unir` del modo descrito en la Figura 10: la asociación de identificadores ha de ser igual en ambos entornos; los tipos asociados a las variables de tipo, cuyo conjunto es obtenido mediante la función `avt`, será el resultado de `unir` los conjuntos que tengan en cada ramal.

La Figura 11 muestra el algoritmo implementado para la función `unir`. Cada elemento del conjunto pueden ser restricciones de subtipado y asignación, o asociaciones de variables de tipo de un entorno. El algoritmo se define empleando dos operaciones `comp` y `union`, definidas en la Figura 12. El algoritmo toma los elementos de los dos conjuntos y los une sustituyendo su tipo por la unión de ambos [20]. Inicialmente toma los del primer conjunto y posteriormente los del segundo que no estén en el primero ($\div$).

Tal y como se muestra en la Figura 12, las comparaciones de restricciones tienen en cuenta siempre el primer tipo. Esto es debido a que tal y como generamos las

$unir(Cjto_1, Cjto_2) \triangleq Cjto$ `in`
  $Cjto \leftarrow \emptyset$
  $\forall\, elem_1 \in Cjto_1$
    `if` $\exists\, elem_2 \in Cjto_2, comp(elem_1, elem_2)$
      $Cjto \leftarrow Cjto \cup union(elem_1, elem_2)$
    `else`
      $Cjto \leftarrow Cjto \cup union(elem_1)$
  $\forall\, elem \in Cjto_2 \div Cjto_1$
    $Cjto \leftarrow Cjto \cup union(elem)$

$Cjto_1 \div Cjto_2 \triangleq Cjto$ `in`
$Cjto \leftarrow \emptyset$
$\forall\, elem_1 \in Cjto_1$
  `if` $\nexists\, elem_2 \in Cjto_2, comp(elem_1, elem_2)$
    $Cjto \leftarrow Cjto \cup elem_1$

Figure 11. Algoritmo `unir` de unión de restricciones y asociaciones de variables de tipo.

(J-Comp)
$comp(X_1 \leftarrow T_1, X_1 \leftarrow T_2)$ $\quad$ $comp(X_1 \leq T_1, X_1 \leq T_2)$ $\quad$ $comp(X_1 : T_1, X_1 : T_2)$

(J-Union)
$union(X \leftarrow T_1, X \leftarrow T_2) = X \leftarrow T_1 \vee T_2$ $\quad$ $union(X \leq T_1, X \leq T_2) = X \leq T_1 \vee T_2$

$union(X : T_1, X : T_2) = X : T_1 \vee T_2$ $\quad$ $union(X \leftarrow T) = X \leftarrow X \vee T$ $\quad$ $union(X \leq T) = X \leq X \vee T$

$union(X : T) = X : X \vee T$

Figure 12. Definición de las operaciones de comparación y unión.

restricciones, el primer tipo siempre será una variable de tipo (si no fuese así, no se generaría restricción alguna), y por ello componemos la restricción en torno a ésta. La única restricción que no cumple este requisito es la generada en la instrucción `return`. No obstante, esta instrucción no podrá aparecer en una sentencia alternativa o condicional gracias a la trasformación del árbol de sintaxis abstracta descrita en la Sección 2.

Cabe mencionar el funcionamiento de la operación unión cuando la restricción se ha generado sólo en un ramal (tres últimas reglas de inferencia de la Figura 12). Para este caso, la restricción generada incluye al propio tipo, indicando que el resultante ha de ser la unión del previo (si lo hubiese) más el inferido en ese ramal. A modo de ejemplo, la variable `e` en la función `f` en la Figura 2 tiene por tipo (tras el condicional) $X_5 \vee$ `bool`. Como veremos en la siguiente sección, esto implica que su tipo puede ser `bool` o la variable libre de tipo X_5. Este caso particular, que también aparecía en la asignación de elementos de un vector (T-VAsign), es tenido en cuenta en la resolución de restricciones (Sección 3.9).

3.8 Dinamismo

El sistema de tipos presentado está parametrizado en función del dinamismo que el programador asocie a las variables de tipo. Los cambios producidos en la inferencia de tipos en base al dinamismo están centralizados en los tipos unión. Un tipo unión denota los posibles tipos que puede poseer una referencia [20]. Si la referencia ha sido declarada como estática, las operaciones permitidas sobre ella serán aquéllas que se satisfagan para todos los tipos de la unión. Así se ofrecerá la seguridad de que no existan errores de tipo en tiempo de ejecución. No obstante, si el programador ha indicado que la variable de tipo es dinámica, la inferencia será más optimista, permitiendo la operación si al menos un tipo de la unión satisface las reglas de tipo. Nótese cómo, si no existe ninguna posibilidad de que la operación sea aplicable, el sistema producirá un error de compilación aunque la referencia sea dinámica. Este funcionamiento se aprecia en las líneas 6 y 7 de la Figura 2. Tanto `s` como `d` tienen

$$\text{(S-EUnion)}\ \frac{\forall\, i \in [1,n], \Gamma \vdash T_i \leq T \mid C_i; \Gamma_i}{\Gamma \vdash \texttt{sta}\, T_1 \vee \ldots \vee T_n \leq T \mid C_1 \cup \ldots \cup C_n; \Gamma_1 \cup \ldots \cup \Gamma_n} \qquad \text{(S-DUnion)}\ \frac{\exists\, i \in [1,n], \Gamma \vdash T_i \leq T \mid C_i; \Gamma_i}{\Gamma \vdash \texttt{dyn}\, T_1 \vee \ldots \vee T_n \leq T \mid \cup C_i; \cup \Gamma_i}$$

Figure 13. Reglas de subtipado para tipos unión.

$$\text{(T-Inv)}\ \frac{\begin{array}{c} clonar(\Gamma(\texttt{id})) : Tp_1 \times \ldots \times Tp_n \to T \mid C \qquad \forall\, i \in [1,n], \Gamma_i; \Omega \vdash E_i : T_i \mid C_i; \Gamma_{i+1} \\ \forall\, i \in [1,n], T_i \notin vtl(\Gamma_{i+1}) \qquad \forall\, i \in [1,n], \Gamma_{n+i} \vdash T_i \leq Tp_i \mid C_{n+i}; \Gamma_{2n+1} \qquad \sigma, \Gamma_{2n+1} \Vdash C; \Gamma_{2n+2} \end{array}}{\Gamma_1; \Omega \vdash \texttt{id}(E_1 \ldots E_n) : \sigma T \mid C_1 \cup \ldots \cup C_{2n}; \Gamma_{2n+2}}$$

$$\text{(T-VTLInv)}\ \frac{\begin{array}{c} clonar(\Gamma(\texttt{id})) : Tp_1 \times \ldots \times Tp_n \to T \mid C \\ \forall\, i \in [1,n], \Gamma_i; \Omega \vdash E_i : T_i \mid C_i; \Gamma_{i+1} \qquad \exists\, i \in [1,n], T_i \in vtl(\Gamma_{i+1}) \qquad \forall\, i \in [1,n], \Gamma_{n+i} \vdash T_i \leq Tp_i \mid C_{n+i}; \Gamma_{2n+1} \end{array}}{\Gamma_1; \Omega \vdash \texttt{id}(E_1 \ldots E_n) : T \mid C \cup C_1 \cup \ldots \cup C_{2n}; \Gamma_{2n+2}}$$

Figure 14. Reglas de inferencia de invocación a función.

el tipo `int∨bool`, pero la primera variable es estática y la segunda dinámica. Esta diferencia provoca que la línea 6 genere un error de compilación (no se puede sumar un `bool`) cuando la línea 7 es correcta (el tipo `int` acepta la suma).

La comprobación de tipos descrita más arriba se formaliza con las reglas de subtipado para los tipos unión mostrados en la Figura 13 (no es necesario definir la promoción a un tipo unión puesto que la sintaxis de nuestro lenguaje nunca lo permite). Nótese cómo la promoción es la operación utilizada para inferir los tipos en nuestro sistema, facilitando así la obtención del comportamiento deseado. En el caso del comportamiento estático, el subtipado es correcto si todos los tipos satisfacen la promoción. Las variables de tipo y restricciones generadas en cada una de las promociones serán la unión de todas las obtenidas (así definimos la unión de entornos en la Figura 9). Para el comportamiento dinámico es necesario que exista al menos una promoción y las restricciones y variables de tipo generadas se definen para aquellas promociones satisfactorias ($\cup\Gamma_i$ representa la unión de los Γ_i).

Nótese cómo las reglas de subtipado de la Figura 13 también son válidas para la conversión de tipos implícitos (tanto dinámicos como estáticos) a tipos explícitos. La línea 8 de la Figura 2 intenta asignar el tipo `int∨bool` al tipo `int`, generando un error porque, al ser la referencia `s` estática, no se cumple que `bool` sea un subtipo de `int`; si `s` fuera dinámica, la conversión sería satisfactoria —aunque no lo sería si la asignación fuese a `ve`, independientemente de que `s` sea un tipo dinámico.

3.9 Invocación a funciones

La Figura 14 muestra las dos reglas de inferencia en la invocación de funciones. La diferencia entre ambas radica en la existencia (T-VTLInv) o no (T-Inv) de argumentos que sean variables de tipo libre. Para ambos casos, la función `clonar` toma todas las variables de tipo de un tipo función, incluyendo las de sus restricciones, y devuelve un tipo equivalente con nuevas variables de tipo libres. Este proceso permite realizar cualquier número de invocaciones a una misma función, separando las variables de tipo para cada llamada. Para cada invocación, se infieren los tipos de los argumentos y se comprueban que sean subtipos de los parámetros.

$$\text{(R-Rest)}\quad \frac{\forall\, T_i \leq T_i' \in C, unificar(T_i, T_i', \sigma), \Gamma_i \vdash T_i \leq T_i' \mid \emptyset; \Gamma_{i+1}\ ^{i\in 1..n} \quad \forall\, T_j \leftarrow T_j' \in C, \Gamma_j \vdash T_j' \leq T_j; \Gamma_j', \forall X \in alias(\sigma, T_j), \Gamma_{j+1} = \Gamma_j, X : T_j'\ ^{j\in n+1..m}}{\sigma, \Gamma_1 \Vdash C; \Gamma_{n+m+1}}$$

Figure 15. Condiciones que satisface el algoritmo de resolución de restricciones.

En el caso de que los argumentos no sean variables de tipo libre, se procede a la resolución de las restricciones. La aseveración $\sigma, \Gamma \Vdash C; \Gamma'$ significa que el conjunto de sustituciones σ es una solución de C, tomando como entrada el entorno Γ, y generando Γ' como salida. En este caso el tipo inferido es la sustitución σT y las restricciones de la función no se añaden a las generadas por la invocación. En el caso de que existan variables de tipo libre en los argumentos (T-VTLinv), el tipo de la invocación es el declarado en la función y las restricciones de ésta se añaden a las propias de la invocación. Por ejemplo, la función `setCampo` de la Figura 2 tiene por tipo $\Gamma(\texttt{setCampo}) : X_8 \times X_9 \rightarrow void \mid X_8 \leq [campo : X_{18}], X_{18} \leftarrow X_9$. Puesto que los parámetros de la invocación de la línea 31 son variables de tipo libre, las restricciones de `setCampo` ($X_8 \leq [\texttt{campo} : X_{18}], X_{18} \leftarrow X_9$), correctamente clonadas, son añadidas a las restricciones de la función `setCampoInc`.

En la Figura 15 se muestra de un modo declarativo el funcionamiento de nuestro algoritmo de resolución de restricciones. Para restricciones de subtipado se unifican sus dos tipos antes de comprobar si existe promoción. El algoritmo de unificación empleado es el descrito por Paterson en [19]. Tanto la unificación como la promoción han de ser válidas para que el conjunto de sustituciones σ constituya una solución. En el caso de las restricciones de asignación se verifica que el valor asignado sea un subtipo de la expresión a asignar. Posteriormente se toman todas las variables de tipo que pertenezcan a la clase de equivalencia [19] de la variable a ser asignada (función `alias`), y se le asigna en el entorno de salida el tipo indicado por la restricción. Siguiendo con nuestro ejemplo de la Figura 2, la invocación de la línea 33 satisface la restricción de subtipado de `setCampo` ($\{\texttt{campo} : X_{17}, \texttt{otro} : \texttt{int}\} \leq [\texttt{campo} : X_{18}]$) y asigna el tipo entero al objeto pasado al satisfacer la restricción de asignación ($X_{17} = alias(X_{18}) \leftarrow \texttt{int}$).

El algoritmo de resolución comprueba primero las restricciones de subtipado y posteriormente las de asignación. Tal y como vimos en la Figura 12, en ocasiones las restricciones de subtipado y asignación pueden ser recursivas, siendo el segundo tipo un tipo unión en el que se incluye al primero. En este caso, antes de la unificación y tomando el entorno de entrada, se sustituirá en el tipo unión la aparición del primer tipo por su tipo asociado en el entorno de entrada. En el caso de no existir, simplemente se eliminará de la parte derecha.

4 Algoritmo de Comprobación de Tipos

Las reglas de inferencia del núcleo del lenguaje aquí presentado requieren inferir el tipo de una función antes de poder inferir el tipo de la invocación (Figura 14). Una vez inferido el tipo de una función, ésta ya no vuelve a analizarse. Por tanto, el algoritmo de comprobación de tipos comienza con las sentencias del programa

principal y, conforme se producen invocaciones a funciones, pasa a inferir el tipo de éstas. Aunque en la implementación real de *StaDyn* sí soportamos recursividad, el sistema de tipos presentado en este artículo no ofrece esta característica.

Si bien las reglas de inferencia presentadas no son deterministas, existen pocas condiciones a añadir para transformar éstas a un algoritmo determinista de comprobación de tipos siguiendo el proceso descrito en el párrafo anterior. La primera de ellas es referente a las reglas de asignación; en T-VTLAsign hay que comprobar que la parte izquierda de la asignación no sea un acceso a un campo de un objeto ni una indexación de un vector. Las reglas de cálculo de Ω (Figura 4) se tienen que codificar junto a las reglas de invocación a función (T-Inv y T-VTLInv), indexación de un vector (T-Vector), acceso a miembro de un objeto (T-Objeto), y las reglas de asignación (Figura 8). En lo relativo a la subsunción (T-Sub), esta regla se encuentra diseminada a lo largo de todas las reglas de subtipado. La implementación de estas dos relaciones entre tipos se ha llevado a cabo empleando el patrón de diseño *Composite* [8], siguiendo las pautas que describimos en [15].

Este sistema de tipos ha sido implementado en C# como parte de un compilador del lenguaje de programación *StaDyn*. Hasta la fecha se genera código para la versión 2 del *CLR* aunque también se pretende generar código para el *DLR* [9] y *Reflective Rotor* [22,17]. La parte de análisis léxico y sintáctico se ha llevado a cabo con la herramienta *AntLR* [18] y el recorrido del árbol de sintaxis abstracta con el patrón de diseño *Visitor* [8]. La implementación de *StaDyn* está disponible en http://www.reflection.uniovi.es/stadyn

En [16] se describe una evaluación preliminar del rendimiento de *StaDyn*. La mejora obtenida es de 43 y 51 veces superior a Visual Basic 10 y C# 4.0 respectivamente, cuando se utilizan variables de tipo dinámicas [16]. En el caso de utilizar declaración explícita de tipos, nuestro compilador genera un código 4,6% más lento que C# 4.0 y un 14% más rápido que Visual Basic 10.

5 Conclusiones

Este artículo presenta el primer paso hacia la especificación del sistema de tipos del lenguaje de programación *StaDyn* cuyo principal objetivo es ofrecer los beneficios de la comprobación dinámica y estática de tipos en un mismo lenguaje de programación. La principal contribución de este trabajo es la realización de inferencia y comprobación de tipos incluso en escenarios dinámicos. Este procesamiento tiene por principales ventajas la detección de errores de tipo en código dinámico, la integración de código dinámico y estático en un mismo lenguaje, y suponer un valioso mecanismo para la optimización de código. Combinamos la declaración explícita e implícita de tipos con un sistema de reconstrucción de tipos basado en restricciones, centralizando el posible dinamismo de los tipos en el comportamiento de los tipos unión, consiguiendo así utilizar el mismo sistema de tipos en entornos donde prima la robustez y eficiencia (estáticos) como en escenarios donde se prioriza la flexibilidad y desarrollo rápido de prototipos (dinámicos).

Nuestro próximo trabajo será la formalización de la semántica del núcleo del lenguaje, para posteriormente demostrar su seguridad respecto al tipo. Una vez llegado a este punto, formalizaremos el resto de características de *StaDyn* tales como

clases, interfaces, herencia, polimorfismo de inclusión, enlace dinámico, recursividad y sobrecarga de métodos.

References

[1] Abadi, M., L. Cardelli, B.C. Pierce, and G. Plotkin, *Dynamic typing in a statically typed language*, ACM Transactions on Programming Languages and Systems 13, 2 (1991), 237–268.

[2] Abadi, M., and L. Cardelli, "A Theory of Objects", Springer, 1996.

[3] Cartwright, R., and M. Fagan, *Soft typing*, in *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation*, 1991.

[4] Flanagan, C., M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen, *Catching bugs in the web of program invariants*, in *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation*, 1996.

[5] Flanagan, C., S.N. Freund, and A. Tomb, *Hybrid types, invariants, and refinements for imperative objects*, in *International Workshop on Foundations and Developments of Object-Oriented Languages*, 2006.

[6] Foster, J.S., T. Terauchi, and A. Aiken, *Flow-Sensitive Type Qualifiers*, in *Programming Language Design and Implementation*, 2002.

[7] Furr, M., A. Jong-Hoon, J.S. Foster, and M. Hicks, *Static Type Inference for Ruby*, in *ACM Symposium on Applied Computing*, 2009.

[8] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley, 1995.

[9] Hugunin, J., *Just Glue It! Ruby and the DLR in Silverlight*, in *MIX Conference*, 2007.

[10] Hunt, A., and D. Thomas, "The Pragmatic Programmer", Addison-Wesley, 2000.

[11] McCracken, N., *The typechecking of programs with implicit type structure*, in *Semantics of Data Types*, Lecture Notes in Computer Science 173, 1984.

[12] Meijer, E., and P. Drayton, P., *Dynamic Typing When Needed: The End of the Cold War Between Programming Languages*, in *Proceedings of the OOPSLA Workshop on Revival of Dynamic Languages* (2004)

[13] Mitchell, J.C., *Type inference with simple subtypes*, Journal of Functional Programming 1, 3 (1991).

[14] Odersky, M., M. Sulzmann, and M.Wehr, *Type inference with constrained types*, Theory and Practice of Object Systems 5, 1 (1999).

[15] Ortin, F., D. Zapico, and J.M. Cueva, *Design Patterns for Teaching Type Checking in a Compiler Construction Course*, IEEE Transactions on Education 50, 3 (2007).

[16] Ortin, F., and Perez-Schofield, J.B.G., *Supporting both Static and Dynamic Typing*, in Programming and Languages Conference (PROLE), 2008.

[17] Ortin, F., Redondo, J.M., Perez-Schofield, J.B.G., *Efficient Virtual Machine Support of Runtime Structural Reflection*, Science of Computer Programming, *to be published*.

[18] Parr, T., "The Definitive ANTLR Reference: Building Domain-Specific Languages", Pragmatic Bookshelf, 2007.

[19] Paterson, M.S., and M.N. Wegman, *Linear Unification*, Journal of Computer and System Sciences 16 (1978).

[20] Pierce, B.C., "Programming with intersection types, union types, and polymorphism", Technical Report CMU-CS-91-106, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1991.

[21] Pierce, B.P., "Types and Programming Languages", The MIT Press. February, 2002.

[22] Redondo, J.M., F. Ortin, and J.M. Cueva, *Optimizing Reflective Primitives of Dynamic Languages*, International Journal of Software Engineering and Knowledge Engineering 18, 6 (2008).

[23] Siek, J., and W. Taha, *Gradual Typing for Objects*, in *Proceedings of the 21st European Conference on Object-Oriented Programing*, Lecture Notes In Computer Science 4609, 2007.

[24] Siek, J., and M. Vachharajani, *Gradual typing with unification-based inference*, in *Proceedings of the Symposium on Dynamic Languages*, 2008.

[25] Thatte, S., *Quasi-static typing*, in *Proceedings of the ACM Symposium on Principles of programming languages*, 1990.