Una Aproximación Práctica al Desarrollo de Comprobadores de Tipos basada en Patrones de Diseño

Francisco Ortin, Luis Vinuesa, Juan Manuel Cueva

Universidad de Oviedo, Departamento de Informática, Calvo Sotelo s/n, 33007, Oviedo
{ortin, vinuesa, cueva}@uniovi.es

Abstract. Los sistemas de tipos son mayoritariamente empleados para asegurar que las distintas construcciones de un lenguaje de programación puedan ser ejecutadas sin la aparición de comportamientos erróneos en tiempo de ejecución. Se suelen emplear formalismos como el cálculo lambda con tipos para demostrar que los sistemas de tipos poseen determinadas características, pudiendo demostrarse que están bien definidos o que son seguros. Un comprobador de tipos es un algoritmo encargado de verificar si un programa cumple todas las reglas especificadas en el sistema de tipos del lenguaje en el que fue escrito. Aunque existen herramientas de descripción y procesamiento de sistemas de tipos, éstas no suelen ser empleadas en el desarrollo de procesadores de lenguajes comerciales por el conjunto de carencias que poseen.

Buscando la independencia del lenguaje de implementación, hemos desarrollado un diseño orientado a objetos encaminado a implementar comprobadores de tipos de cualquier lenguaje de programación, empleando para ello un conjunto de patrones de diseño. Elementos existentes en la mayoría de los sistemas de tipos, tales como expresión, constructor, equivalencia y coerción de tipos, o más avanzados como el polimorfismo paramétrico, han sido desarrollados con el diseño presentado. No es necesario que el desarrollador del procesador tenga conocimiento alguno de formalismos ni notaciones matemáticas, empleados frecuentemente en la teoría de tipos. El diseño es fácilmente comprensible, permitiendo así una sencilla depuración y extensión del comprobador y sistema de tipos. Adicionalmente se obtienen los beneficios de emplear determinados patrones de diseño orientados a objetos, altamente probados en el desarrollo de sistemas informáticos.

Keywords. Sistemas de tipos, comprobador de tipos, patrones de diseño, orientación a objetos.

1 Introducción

Un tipo es una colección de objetos poseedores de una estructura similar; la teoría de tipos, en sus comienzos, fue una rama de las matemáticas y la lógica basada en clasificar objetos mediante tipos [1]. Con la aparición de los ordenadores y los lenguajes de programación, la teoría de tipos se ha constituido como una aplicación práctica del desarrollo de lenguajes de programación. Un sistema de tipos es un método dirigido por sintaxis para demostrar la ausencia de comportamientos erróneos de programas, clasificando las construcciones sintácticas en función del tipo de valor que pueden procesar [2]. Así, los sistemas de tipos dividen los posibles valores en tipos, identificando ciertas construcciones del lenguaje como ilegales, en función de su tipo.

La comprobación de tipos es el análisis que detecta inconsistencias y anomalías semánticas en un programa, asegurándose de que las entidades se ajusten a su declaración y sean combinadas con otras entidades de tipos compatibles; el algoritmo que lleva a cabo este proceso se llama comprobador de tipos [3]. Un procesador de lenguaje acostumbra a implementar un comprobador de tipos en su fase de análisis semántico [4]. Los principales beneficios obtenidos en la comprobación de tipos efectuada en tiempo de compilación (estática) son:

- Detección temprana de errores. La comprobación estática de tipos reduce el número de errores que un programa puede generar en tiempo de ejecución. Éste es el beneficio más obvio ya que, gracias a la detección temprana de los errores, el programador podrá reparar éstos de un modo

casi inmediato –y no cuando se esté ejecutando la aplicación, pudiendo incluso haber sido implantada.

- Abstracción. Otra ventaja de emplear tipos en los lenguajes de programación es que su uso fuerza al programador a dividir el problema en diversos tipos de módulos, de un modo disciplinado. Los tipos identifican la interfaz de los módulos (funciones, clases, paquetes o componentes) proporcionando una simplificación de los servicios que ofrece cada módulo; un tipo de contrato parcial entre los desarrolladores del módulo y sus clientes.

- Eficiencia. Una entidad de un programa declarada con un tipo específico indica información relativa a lo que se intenta hacer con ella. De este modo, al conocer el tipo de las construcciones del lenguaje, se podrá generar código de carácter más específico y eficiente que si no tuviésemos esa información. En los procesadores que no poseen tipos en tiempo de compilación se tiene que descubrir la ejecución específica en tiempo de ejecución, con la consecuente pérdida de eficiencia.

- Seguridad. Un sistema de tipos es seguro si es capaz de garantizar la integridad de las abstracciones de alto nivel ofrecidas al programador. Por tanto, un programa aceptado por un sistema de tipos seguro se comportará de un modo adecuado, sin generar errores de tipo en tiempo de ejecución.

- Legibilidad. Un tipo de una entidad (variable, objeto o función) transmite información acerca de lo que se intenta hacer con ella, constituyendo así un modo de documentación del código.

Aunque el uso tradicional de los sistemas de tipos es la validación semántica de los programas de un lenguaje, también se emplean en otros campos como la ingeniería inversa [5], pruebas de software [6], metadatos del web [7] o incluso en seguridad [8].

1.1 Herramientas de Sistemas de Tipos

Los sistemas de tipos son uno de los métodos formales dentro de la computación empleados para asegurar que un sistema se comporta adecuadamente en función de una especificación del comportamiento deseado. Su formalización requiere notaciones precisas, válidas para demostrar propiedades de los sistemas de tipos [3]. Los formalismos más empleados en la actualidad son los cálculos lambda con tipos y la semántica operacional [9, 10]; en el caso de los comprobadores de tipos, lo más común es emplear gramáticas con atributos [11].

Sin embargo, aunque la formalización es necesaria para realizar estas demostraciones, también se requiere tener una infraestructura para la implementación de comprobadores de tipos basados en estos sistemas de tipos. La mayoría de las herramientas existentes se basan en estos formalismos: TinkerType utiliza cálculo lambda con tipos para demostrar propiedades de los sistemas de tipos, permitiendo implementar comprobadores de tipos en ML [12]; los métodos formales *ligeros*, tales como ESC/Java [13], SLAM [14] o SPIN [15], realizan análisis exhaustivos y verificación de programas para detectar errores que un compilador pasa por alto; las escasas herramientas de construcción de compiladores que ofrecen la generación de comprobadores de tipos están basadas en gramáticas con atributos [16] o modificaciones de éstas [17].

Aunque los sistemas de tipos y sus herramientas son utilizadas en distintas áreas dentro de la informática (como la investigación en lenguajes de programación), la mayoría de los compiladores comerciales no son desarrollados empleando estas herramientas [18]. Los principales motivos son:

- Los métodos formales están más enfocados a demostrar propiedades de los sistemas de tipos, más que a implementar comprobadores. Por ello, las notaciones matemáticas empleadas pueden ser desconocidas para el desarrollador del compilador.

- La salida de las herramientas está restringida a un lenguaje de programación. Si tomamos TinkerType como un ejemplo, los comprobadores generados están desarrollados en ML.

- Si se emplea una herramienta de generación de comprobadores de tipo a partir de un método formal, existirá un salto en la abstracción de la representación formal al código generado. De

esta forma, el código producido es más difícil de depurar y comprender puesto que proviene de la traducción automática de otra notación.

– Las herramientas basadas en métodos formales no están inmersas en entornos de desarrollo comerciales. Puesto que la orientación a objetos se ha constituido como el principal paradigma del desarrollo de software comercial, la implementación de un comprobador de tipos empleando estas tecnologías se podrá beneficiar de sus aportaciones [19], de los patrones de diseños existentes [20] y de todos los componentes, herramientas y *frameworks* reutilizables, disponibles en el mercado.

Sin tratar de crear una nueva herramienta que supere las limitaciones mencionadas, definiremos un diseño de ingeniería del software capaz de implementar comprobadores de tipos cuyos sistemas de tipos puedan haberse demostrado correctos previamente –empleando en dicho caso métodos formales. Para ello, describiremos un conjunto de diseños orientados a objetos que empleen los beneficios de este paradigma, así como la utilización de patrones de diseño ampliamente conocidos. El diseño será independiente del lenguaje de programación, fácilmente comprensible y mantenible por el desarrollador del procesador del lenguaje, y cubrirá las características estáticas y dinámicas de los sistemas de tipos más comunes, permitiendo extender y adaptar éste en función de las necesidades del lenguaje a procesar.

El resto del documento está organizado de la siguiente manera. La sección 2 introduce el modo de modelar las expresiones de tipo. La equivalencia de expresiones de tipo es descrita en la sección 3, y en el siguiente apartado se modelan éstas reduciendo el consumo de memoria y procesador. Cómo desarrollar las propiedades de coerción de tipos y polimorfismo paramétrico son abordados en las secciones 5 y 6 respectivamente. Finalmente se presentan las conclusiones y trabajo futuro.

2 Expresiones de Tipo

A la hora de implementar un comprobador de tipos, el procesador de lenguaje deberá representar internamente los tipos del lenguaje de algún modo. Cualquier representación, interna o externa, del tipo de cada construcción sintáctica del lenguaje se denomina expresión de tipo [4]. Un procesador de lenguaje deberá poseer una representación interna de las expresiones de tipo del lenguaje que procesa, para llevar a cabo, entre otras tareas, las comprobaciones de tipo oportunas.

Las expresiones de tipo se centran en la definición constructiva de un tipo: Un tipo es definido como un conjunto de tipos simples (también llamados predefinidos, básicos o primitivos), o bien como un tipo compuesto o construido formado a partir de otros tipos. Un tipo simple es un tipo atómico cuya estructura interna no puede ser modificada por el programador (`integer`, `float`, `boolean`...). Un tipo construido o compuesto es un tipo construido por el programador a partir de un constructor de tipos (`record`, `array`, `set`, `class`...) aplicado a otros tipos –básicos o construidos.

Un comprobador de tipos asocia (dinámica o estáticamente) un tipo a las distintas construcciones del lenguaje, calculando en todo momento el tipo de las expresiones, para posteriormente realizar las comprobaciones oportunas del análisis semántico. Este proceso de asociación de una expresión de tipo a las distintas construcciones del lenguaje se denomina inferencia de tipos [3]. Diversas características de los lenguajes de programación como la equivalencia y coerción de tipos, el polimorfismo, la genericidad y la sobrecarga hacen que la inferencia de tipos de un lenguaje se convierta en una tarea compleja. Un buen diseño de la representación interna de las expresiones de tipo del lenguaje será clave para poder inferir los tipos del modo más sencillo, extensible, mantenible y reutilizable.

2.1 Implementación de las Expresiones de Tipo

Cada compilador podrá representar internamente sus expresiones de tipo de un modo distinto en función de las características del lenguaje a procesar y del lenguaje de programación empleado en su implementación. Como primer ejemplo, las expresiones de tipo de un lenguaje que únicamente posea tipos simples, sería tan sencillo como utilizar una representación mediante variables enteras, enumeradas o caracteres. Sin embargo, estas representaciones no serían válidas cuando el lenguaje a

procesar posea algún constructor de tipo –el constructor de tipo puntero, por ejemplo, puede dar lugar a infinitas expresiones de tipo.

Con la aparición de constructores de tipo se crean infinitas combinaciones a la hora de enumerar las posibles expresiones de tipo. El empleo de cadenas de caracteres para representar las expresiones de tipos sería una solución factible, utilizando algún lenguaje de representación de expresiones de tipo como el descrito por Alfred Aho [4]. No obstante, cada vez que queramos extraer un tipo que forme parte de otro tipo compuesto, deberíamos procesar la cadena de caracteres como si de un lenguaje se tratase, incurriendo en una excesiva e innecesaria complejidad computacional. Una representación más sencilla y eficiente sería con estructuras de datos recursivas. A modo de ejemplo, lo siguiente es una implementación posible en el lenguaje C de un conjunto de tipos básicos y el constructor de tipo puntero:

```
typedef enum enum_tipos {
        entero, caracter, logico, real, puntero
} tipo_t;
typedef struct struct_puntero {
        tipo_t tipo;
        struct struct_puntero *puntero;
} expresion_tipo;
```

Figura 1: Alternativa de representación de expresiones de tipo con registros.

Esta estructura de tipos es más fácil de manipular que una cadena de caracteres y requiere una menor capacidad de cómputo a la hora de procesar una expresión de tipo compuesta.

La principal limitación de este tipo de estructura de datos es que, precisamente, sólo modela datos. En el desarrollo de un procesador de lenguaje será necesario asociar a estas estructuras un conjunto de rutinas tales como inferencia, equivalencia, coerción y comprobaciones de tipos para la fase de análisis semántico, o tamaño y representación de datos para la fase de generación de código. Existirán rutinas específicas para cada expresión de tipo, y otras comunes a todas ellas. Haciendo uso de la orientación a objetos, podremos asociar estas rutinas generales y específicas a los datos oportunos mediante el empleo de clases, obteniendo adicionalmente una mayor calidad en el diseño mediante el empleo de técnicas de encapsulamiento, abstracción, herencia y polimorfismo, reduciendo así la complejidad de la implementación de un sistema de tipos.

El problema de representar estructuras compuestas recursivamente de un modo jerárquico, aparece en diversos contextos dentro del campo de la computación. El patrón de diseño orientado a objetos *Composite* [20] ha sido utilizado para modelar y resolver este tipo de problemas. Permite crear estructuras compuestas recursivamente, tratando tanto los objetos simples como los compuestos de un modo uniforme, de forma que la utilización de un objeto compuesto y simple pueda realizarse sin conllevar distinción alguna. Su modelo estático se representa mediante el diagrama de clases de la Figura 2.

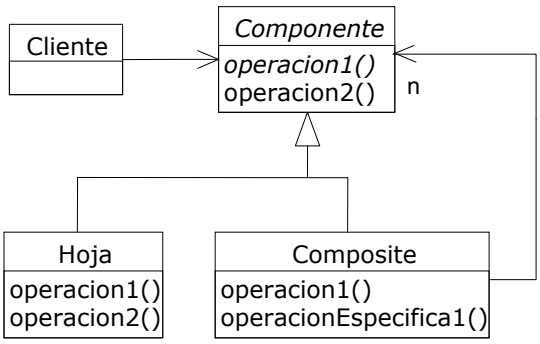

Figura 2: Diagrama de clases del patrón de diseño *Composite*.

Los distintos elementos del modelo son:

– Componente. Clase típicamente abstracta que declara la interfaz de todos los elementos, independientemente de que sean simples o compuestos. Puede implementar en sus métodos un comportamiento por omisión, o declararlo como abstracto para que cada uno de los subtipos lo implementen.

– Hoja. Representa cada nodo hoja de la estructura jerárquica. Un nodo hoja no tienen ningún tipo "hijo". Definirá en sus métodos las operaciones concretas para ese nodo específico.

– Compuesto (*Composite*). Modela aquellos nodos que se construyen como composición de otros nodos, almacenando referencias a sus nodos "hijo". Implementa las operaciones en función de los hijos que posee, e incluso en función de los resultados de cada una de las operaciones de sus hijos.

– Cliente. Cualquier elemento software que utiliza el patrón de diseño presentado.

Siguiendo este patrón de diseño, cada expresión de tipo primitivo será una clase hoja y los tipos compuestos con cada constructor de tipo serán clases *Composite*. Las operaciones comunes a todos los tipos, propias de la fase de análisis semántico (equivalencia, coerción, inferencia o unificación) y de generación de código (tamaño o representación), serán ubicadas en la clase componente. El hecho de ubicar estos métodos en la clase raíz de la jerarquía, implica que se pueda llevar a cabo un tratamiento uniforme de todos los tipos respecto a este conjunto de mensajes públicos, indistintamente de su estructura interna.

Si existe un comportamiento por omisión para la mayoría de las expresiones de tipos, será implementado al nivel de la clase Componente. Cada clase derivada redefinirá, si fuese necesario, la operación general definida en el componente para su caso específico –u obtendrá, en caso contrario, el comportamiento por defecto. Adicionalmente, cualquier clase derivada podrá definir métodos específicos propios de su expresión de tipo, sin necesidad de que éstos estén declarados en su clase base. Estos métodos definirán los mensajes específicos de la expresión de tipo que representa la clase, no teniendo sentido éstos para el resto de tipos.

El diagrama de clases de la Figura 3 muestra la estructura preliminar del diseño empleado para modelar las expresiones de tipo de un subconjunto del lenguaje C, siguiendo el patrón de diseño *Composite*.

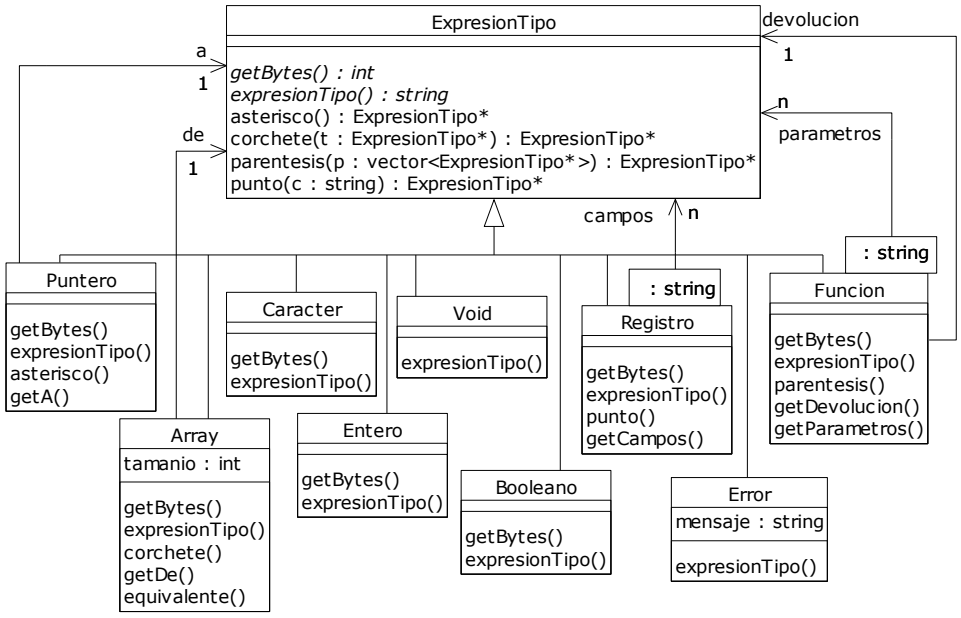

Figura 3: Diagrama de clases empleado para modelar expresiones de tipo del lenguaje C.

La clase que juega el papel de componente en el patrón *Composite* es la clase abstracta `ExpresionTipo`. Ésta posee el comportamiento general de todas las expresiones de tipo, alguno de ellos predefinido por omisión. Los métodos de ejemplo definidos son:

– Métodos `asterisco`, `corchete`, `parentesis` y `punto`: Estos métodos tienen dos responsabilidades: calcular (inferir) el tipo resultante tras aplicar un operador del lenguaje, a partir de una expresión de tipo y, en algún caso, otras expresiones de tipo pasadas como parámetro, y realizar la comprobación de tipos; devolverán el tipo `Error` con una descripción del mismo, si

existe un error de tipo.
La implementación por omisión es devolver siempre un error, indicando así que dicha operación no está semánticamente definida para el tipo. Cada tipo que implemente este operador deberá redefinir el método relacionado. Por ejemplo, el tipo `Puntero` implementa el método `asterisco` puesto que esta operación está permitida para este tipo; la expresión de tipo que devuelve (infiere) es el tipo al que apunta.

– Método `expresionTipo`: Devuelve una cadena de caracteres representativa de su expresión de tipo, siguiendo la notación utilizada en muchos compiladores [4]. Sus objetivos principales son facilitar las tareas de depuración y el ahorro de memoria (explicado en siguiente punto). El código C++ de la Figura 4 muestra un ejemplo de implementación de este método para la clase `Funcion`. Así, una función que reciba un puntero a un entero y un real y no devuelva nada, poseerá la expresión de tipo "`(Puntero(int),float)->void`".

```
string Funcion::expresionTipo() const {
    ostringstream o;
    if (parametros.size())
          o<<"(";
    unsigned i=0;
    for (;i< parametros.size()-1;i++)
          o<< parametros[i]-> expresionTipo()<<',';
    if (parametros.size())
          o<< parametros [i]-> expresionTipo()<<')';
    o<<"->"<<devolucion-> expresionTipo();
    return o.str();
}
```

Figura 4: Implementación del método que calcula expresiones de tipo.

– Método `getBytes`: Es un mero ejemplo de cómo los tipos del lenguaje poseen funcionalidad propia de la fase de generación de código. Este método devuelve el tamaño en bytes necesario para albergar una variable de ese tipo.

Las clases derivadas poseen métodos adicionales propios de su comportamiento específico, tales como `getCampos` (registro), `getDevolucion` y `getParametros` (función), `getA` (puntero) y `getDe` (*array*).

Nótese cómo los tipos compuestos poseen asociaciones al tipo base: un puntero requiere el tipo al que apunta (`a`); un *array* necesita el tipo que colecciona (`de`); un registro requiere una colección de sus campos (`campos`), cualificada por el nombre del campo (`string`); una función requiere una asociación al tipo que devuelve (`devolucion`), así como una colección cualificada por el nombre de cada uno de sus parámetros (`parametros`). El hecho de que todas las asociaciones estén dirigidas hacia la clase base de la jerarquía, hace que cada tipo compuesto pueda formarse con cualquier otro tipo –incluyendo él mismo– gracias al polimorfismo.

Otra de las ventajas de este diseño, es la facilidad con la que los tipos se pueden ir construyendo con las herramientas existentes de análisis, tales como Lex/Yacc, AntLR o JavaCC. Las expresiones de tipo se pueden construir dirigidas por la sintaxis, por medio de sus constructores. Inicialmente se crean los tipos básicos y éstos se emplean para construir posteriormente nuevos tipos compuestos en las rutinas semánticas de las herramientas de análisis.

2.2 Un Primer Ejemplo

Mostraremos un escenario de ejemplo de utilización del diseño mostrado para representar las expresiones de tipo de un subconjunto del lenguaje C. La parte léxica y sintáctica se ha especificado mediante las herramientas Lex y Yacc y el resto del procesador se ha implementado en C++ ISO/ANSI. Cuando una variable es definida (líneas 1 a la 11 de la Figura 5), se asocia en una tabla de símbolos su identificador con su tipo –un puntero de tipo `ExpresionTipo`. En estas rutinas semánticas, se crean los tipos mediante la invocación del constructor apropiado, componiendo los tipos complejos conforme el Yacc vaya realizando las reducciones.

Las sentencias de una función son una repetición de expresiones. Cada expresión tiene que tener un tipo y éste se representa mediante un puntero a `ExpresionTipo`. En cada reducción será necesario inferir el tipo de la subexpresión, aplicando en cada caso el operador oportuno. Esta inferencia será procesada mediante el envío del mensaje oportuno al tipo de la subexpresión. Por ejemplo, la línea 17 de la Figura 5 obtiene el tipo puntero de la primera subexpresión `puntero`; en la siguiente reducción aparece el operador `*`, enviándose el mensaje asterisco, infiriéndose el tipo apuntado `Entero`; el siguiente paso es obtener de la tabla de símbolos el tipo `Array` de `vector`; a éste se le pasa el mensaje `corchete`, pasándole el tipo `Entero` previamente inferido; el tipo resultante de la expresión es entero.

A modo de ejemplo, la Figura 5 muestra una traza de las expresiones de tipos inferidas para cada sentencia del programa principal. En la parte izquierda se muestra el código fuente con su número de línea. Lo mostrado en la parte derecha es la cadena de caracteres devuelta por el envío del mensaje `expresionTipo`, a cada tipo inferido.

```
1:     int vector[10];
2:     int *puntero;
3:     int **punteroDoble;
4:     char **v[10];
5:     char *w[10];
6:     struct Fecha {
7:       int dia, mes, anio;
8:       };
9:     struct Fecha fecha;
10:    int *f(int,char*);
11:    void p(int*);
12:
13:    int main() {
14:      fecha;                    Registro((dia x int)x(mes x int)x(anio x int))
15:
16:      v;                        Array(10,Puntero(Puntero(char)))
17:      vector[*puntero];         int
18:      **v[**punteroDoble];      char
19:      w[*f(3,w[1])];            Puntero(char)
20:      p(f(fecha.dia,w[2]));     Void
       }
```

Figura 5: Traza de inferencia de tipos en un programa de ejemplo.

3 Equivalencia de Expresiones de Tipo

Como hemos mencionado con anterioridad, el principal objetivo de un sistema de tipos es evitar la aparición de errores de tipo en tiempo de ejecución. En diversas construcciones de un lenguaje de programación (como la invocación a funciones o la asignación de una expresión a una variable) es necesario detectar la incorrecta utilización de expresiones que no posean tipos equivalentes. Por tanto, uno de los elementos a tener en cuenta cuando se diseña un sistema de tipos es determinar las condiciones que hacen que dos tipos sean equivalentes entre sí, es decir, que sean el mismo tipo. Existen diversas alternativas de equivalencia de tipos, siendo las dos principales clasificaciones las siguientes [21]:

- Equivalencia estructural. Dos tipos son estructuralmente equivalentes, únicamente si poseen la misma estructura, es decir, o son el mismo tipo básico, o bien están formadas aplicando el mismo constructor a tipos estructuralmente equivalentes. Lenguajes tales como ML, Algol-68 y Modula-3 emplean un sistema de tipos basado en equivalencia estructural. C++ implementa equivalencia estructural, excepto para clases, registros y uniones.

- Equivalencia de nombres (nominal). Establece que todo tipo ha de tener un nombre único, considerando dos tipos equivalentes sólo si tienen el mismo nombre. Lenguajes con equivalencia de nombres son Java y Ada.

El diseño de la segunda es más sencillo que el primero, puesto que se reduce a la comparación de los identificadores únicos de cada tipo. Sin embargo, la equivalencia estructural requiere un proceso de comparación recursiva de cada uno de los elementos que constituyen un tipo.

Utilizando el diseño presentado para las comprobaciones de tipos, la implementación de un algoritmo de equivalencia estructural se convierte en una tarea realmente sencilla. Puesto que todos los tipos han de poder ser comparados con cualquier otro tipo, se añadirá a la clase `ExpresionTipo` un método `equivalente` encargado de devolver si el tipo pasado como parámetro es equivalente al actual.

En el caso de los tipos simples, el método `equivalente` se limitará a comprobar si el parámetro es del mismo tipo que el de la clase –ésta será la implementación por omisión ubicada en la clase `ExpresionTipo`. En el caso de las expresiones de tipo compuestas es necesario comprobar que ambos han sido construidos con el mismo constructor de tipos, y además llevar a cabo una comparación recursiva de cada una de las expresiones de tipo empleadas para construir el tipo compuesto. Una implementación en C++ de estos dos ejemplos es la siguiente:

```
bool ExpresionTipo::equivalente(const ExpresionTipo *et) const {
    return typeid(*this)==typeid(*et); // * RTTI
}
bool Registro::equivalente(const ExpresionTipo *te) const {
    const Struct *record=dynamic_cast<const Struct*>(te); // * RTTI
    if (!record) return false;
    if (campos.size()!=record->campos.size())
          return false;
    map<string,TypeExpression*>::const_iterator it1,it2;
    for (it1=campos.begin(),it2=record->campos.begin();
                        it1!=campos.end();++it1,++it2) {
          if (it1->first!=it2->first)
                return false;
          if (!it1->second->equivalente(it2->second))
                return false;
    }
    return true;
}
```

Figura 6: Implementación de ejemplo de equivalencia estructural.

Empleando la información de tipos ofrecida por el C++ estándar (RTTI), la implementación por omisión devuelve la igualdad de la clase de la que las expresiones de tipo son instancia. En el caso del registro deberá cumplirse, además, que el número de campos coincida y que cada uno de ellos sean equivalentes entre sí.

4 Representación de Tipos mediante GADs

Del diagrama de clases utilizado para representar las expresiones de tipo (empleando el patrón *Composite*) se pueden crear las expresiones mediante árboles de objetos. El programa de ejemplo mostrado en la Figura 5 asociará al identificador `f` la estructura de objetos en forma de árbol de la Figura 7. La estructura de árbol duplica la representación de tipos para un identificador dado –aparece dos veces el tipo `Entero`. Esta duplicación de objetos se hace más latente conforme se van creando distintas expresiones de tipos para las diversas construcciones sintácticas del lenguaje, que en la mayoría de los programas reales supone cifras demasiado elevadas. El alto consumo de memoria y el tiempo excesivo requerido para crear y recorrer este número de estructuras de objetos hace necesario el rediseño de las expresiones de tipo.

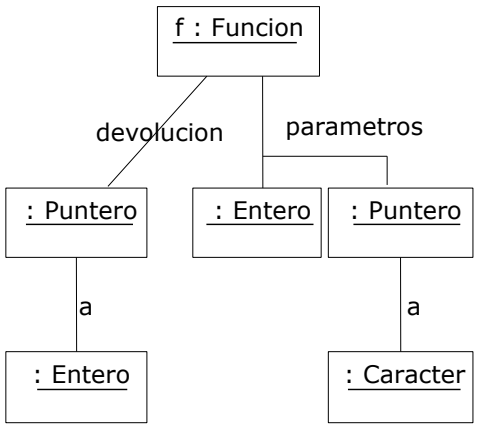

Figura 7: Diagrama de objetos en forma de árbol del identificador f.

Este tipo de problemas es resuelto de un modo sencillo por el patrón de diseño *Flyweight* [20]. Su modelo se basa en identificar objetos compartidos simultáneamente en múltiples contextos,

representando el estado compartido en todos los estados aparte de los estados particulares de cada objeto. Conforme se cree un número elevado de objetos, la parte compartida a todos ellos (*Flyweight*) se representa mediante un único objeto; otras instancias de un tamaño más reducido representarán la información particular de cada uno de los objetos por separado.

La separación de los estados compartidos y particulares en nuestro caso es trivial. Ante un programa de entrada, un procesador del lenguaje reconocerá un conjunto elevado de símbolos, teniendo cada uno de ellos un tipo –al igual que toda expresión del lenguaje.

- La información relativa a un símbolo es particular. Una variable tendrá un desplazamiento en memoria, un ámbito, un tipo y un identificador. Cada símbolo deberá tener esta información particular a cada uno.
- Un tipo posee información compartida por todo símbolo y construcción sintáctica de este tipo: su tamaño, equivalencia, coerción, inferencia y representación a bajo nivel.

Por este motivo, separando los símbolos de los tipos de un lenguaje y reutilizando los segundos, obtendremos un mayor aprovechamiento de la memoria y, como veremos en la sección 4.1, un mayor rendimiento. El patrón de diseño *Flyweight* ofrece un modelo para no repetir la creación de, en nuestro caso, expresiones de tipo. Una factoría de tipos será la encargada de obtener y crear (si fuese necesario) los tipos del sistema, sin necesidad de ir creando uno distinto cada vez que se requiera un tipo.

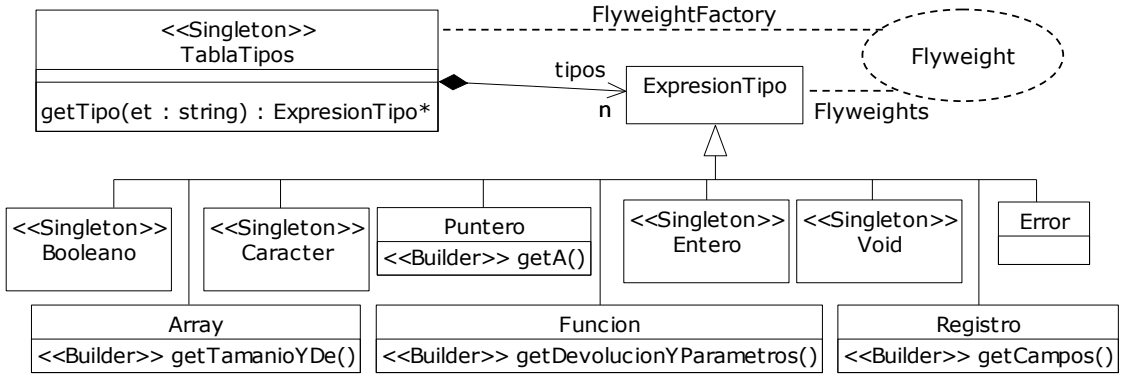

Figura 8: Diagrama de clases de la tabla de tipos (Flyweight).

A modo de resumen, los siguientes criterios son los empleados para conseguir controlar la creación de expresiones de tipo por medio de la tabla de tipos:

- El método `getTipo` de `TablaTipos` es la única forma de acceder a un tipo. Este método recibe un parámetro cadena de caracteres con la representación textual de la expresión de tipos, cuya notación fue mostrada con anterioridad en este artículo. Este método se encargará de localizar el tipo asociado a dicha expresión de tipo (creándolo si fuese necesario), controlando que éste no esté duplicado. Devolverá un puntero a `ExpresionTipo`.
- Prohibición de creación y destrucción de tipos. Para controlar la no duplicidad de los tipos, la construcción y destrucción de instancias de tipos será prohibida. En el lenguaje C++, este objetivo se consigue declarando los constructores y destructores de todos los tipos como protegidos y haciendo que la tabla de tipos sea una clase amiga de cada clase tipo. En otros lenguajes como Java, la implementación se puede conseguir con la utilización de paquetes.
  En el caso de que el lenguaje requiera destrucción explícita de objetos (como C++), la tabla de tipos será la encargada de destruir todos los tipos cuando ésta vaya a ser liberada.
- Construcción heterogénea de expresiones de tipo. El proceso de crear el tipo apropiado a partir de la cadena que representa su expresión de tipo no es una tarea elemental. Es necesario analizar ésta y llamar a los constructores apropiados componiendo la estructura recursiva de objetos. Este proceso sólo se emplea en la construcción de las expresiones de tipo; la posterior utilización de las mismas se realizará mediante un puntero `ExpresionTipo`.

El proceso de creación de expresiones de tipos es llevado a cabo mediante la utilización de otro patrón de diseño denominado *Builder*, que permite separar la construcción de objetos complejos [20]. Las clases representativas de tipos compuestos poseerán un método de clase (*static*) encargado de obtener los tipos compuestos de esa expresión de tipo. Por ejemplo, el método `getDevolucionYParametros` de las funciones, analizará la cadena y devolverá los tipos de los que está compuesto. Este es un proceso mutuamente recursivo, es decir, estos métodos se apoyarán en el método `getTipo` que a su vez los ha invocado.

– Única instancia de los tipos simples. Para asegurar la creación única de los tipos simples –nótese que los compuestos no han de ser únicos, puesto que dependen de sus tipos agregados– se empleará un patrón de diseño *Singleton* que asegurará su unicidad [20]. La tabla de tipos también empleará este patrón de diseño.

Mediante el diseño presentado, las estructuras de objetos que representan los tipos del lenguaje dejarán de tener una estructura de árbol con instancias repetidas, pasando a estar representadas mediante Grafos Acíclicos Dirigidos (GAD). El programa de ejemplo mostrado en la Figura 5 dará lugar al diagrama de objetos de la Figura 9.

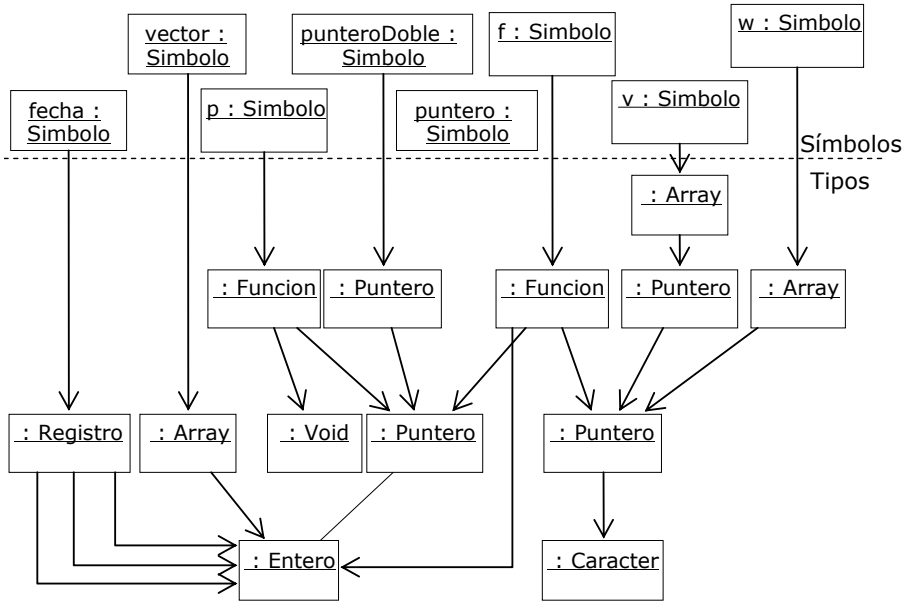

Figura 9: Diagrama de objetos mediante estructura de GAD.

4.1 Equivalencia Estructural mediante GADs

Puesto que la nueva estructura de objetos asegura la unicidad de las expresiones de tipo, la implementación de la equivalencia estructural de tipos (método `equivalente`) se simplifica en gran medida. No será necesario comprobar recursivamente los estados de los objetos `ExpresionTipo`, sino que nos apoyaremos en su identidad –en C++ se trata de comprobar direcciones de memoria en lugar de atributos, y en Java referencias en lugar del método `equals`. Dos expresiones de tipo serán equivalentes si son exactamente el mismo objeto. Este proceso requiere un tiempo de proceso ínfimo en relación con el necesario en las estructuras de árbol.

```
bool ExpresionTipo::equivalente(const ExpresionTipo *te) const {
          return this==te; }
```

Figura 10: Nueva implementación de equivalencia estructural de tipos, mediante estructuras GAD.

5 Coerción de Tipos

Ante diversas construcciones sintácticas, los lenguajes de programación permiten realizar conversiones de los tipos inferidos de un modo implícito (realizadas por el procesador) o explícito (por

el programador). Las conversiones implícitas reciben el nombre de coerción –en ocasiones también denominadas promoción. La mayoría de los lenguajes ofrecen coerción de tipos en aquellos contextos donde la conversión no supone pérdida alguna de información. La conversión implícita de números enteros donde se requiera un número real se da, por ejemplo, en los lenguajes C++, Pascal, Java y C#. En Modula-2 no existe coerción alguna.

Las conversiones implícitas entre tipos (coerciones) hacen que la definición de equivalencia se vea modificada en cierto modo. Si dos tipos no son equivalentes, puede ser que exista coerción entre ellos y, por tanto, sí sean finalmente equivalentes. Por tanto, crearemos un nuevo método en la jerarquía `igualQue`, cuyo código será el propio de la implementación de equivalencia estructural (el mostrado en la Figura 10). Otro método `coercion` devolverá, a partir del tipo pasado, la expresión de tipo resultante de la coerción del objeto implícito al tipo pasado como parámetro. Así, la nueva implementación del método `equivalente` pasará a ser:

```
const ExpresionTipo * ExpresionTipo::equivalente(const ExpresionTipo *te)
const {
    if (this->igualQue(te)) return this;
    return this->coercion(te);
}
```

Figura 11: Método equivalente teniendo en cuenta la coerción de tipos.

Gracias al encapsulamiento, la modificación de la implementación del método `equivalente` sin haber alterado su interfaz supone que el resto de la implementación del comprobador de tipos que hace uso de este método (como el operador de asignación o la invocación a una función) no sufra cambio alguno.

Las coerciones de tipos varían de un lenguaje a otro. Nuestro diseño hace que la modificación de las coerciones de un lenguaje a otro sea modelada de un modo sencillo. Así, el método de coerción se ubicará en la clase raíz de la jerarquía (`ExpresionTipo`) devolviendo siempre el valor nulo (`0` en C++ o `null` en Java), indicando que no es posible hacer la coerción. Cada una de las expresiones de tipo deberá indicar en su método `coercion` aquellos tipos a los cuales puede promocionar. A modo de ejemplo, un valor lógico en C++ puede convertirse implícitamente a un entero, carácter o real. Adicionalmente, un *array* que colecciona elementos de tipo T promociona a un puntero que apunte a elementos de tipo T. Basándonos en este diseño, se pueden implementar estas conversiones en los métodos `coercion` de las clases `Booleano` y `Array`, tal y como se muestra en la Figura 12.

```
const ExpresionTipo *Booleano::coercion(const ExpresionTipo *te) const {
    string ste=te->expresionTipo();
    if ( ste=="int" || ste=="char" || ste=="float" )
          return TablaTipos::getInstance().getTipo(ste);
    return 0;
}
const ExpresionTipo *Array::coercion(const ExpresionTipo *te) const {
    const Puntero *puntero=dynamic_cast<const Puntero*>(te);
    if (puntero && de->igualQue(puntero->getA()) )
          return puntero;
    return 0;
}
```

Figura 12: Implementación de las coerciones de booleanos y arrays.

El resultado de este diseño es que, distribuyendo el código de coerción de tipos en las distintas clases, todo procesador de lenguaje realizará, si es necesario, una conversión implícita entre los tipos sin necesidad de programarlo explícitamente. La modificación de las reglas de promoción entre tipos es muy sencilla y directa. Nótese cómo el mecanismo de polimorfismo de los sistemas de tipos de los lenguajes con herencia es otra aplicación de este diseño –toda clase deberá permitir la coerción a sus superclases.

6 Polimorfismo

El polimorfismo (universal) es la propiedad de un lenguaje que hace que una parte de un programa pueda tener varios tipos [3]. Se produce cuando una función puede definirse sobre un conjunto determinado de tipos. Mediante el empleo de herencia, muchos lenguajes orientados a objetos ofrecen un polimorfismo restringido a una jerarquía de tipos, denominado polimorfismo de inclusión (o de subtipos) [22]. Los lenguajes orientados a objetos denominan al polimorfismo universal (también llamado paramétrico) como genericidad.

La principal aportación de los sistemas de tipos polimórficos es que añaden variables de tipo en las expresiones de tipo. Una variable de tipo representa cualquier aparición de una expresión de tipo como instancia de esta variable. Una variable de tipo sirve para representar cualquier tipo dentro de otra expresión de tipo. ML, Haskell y OCaml son lenguajes funcionales que ofrecen polimorfismo; C++ y Eiffel son lenguajes orientados a objetos que también lo implementan.

Un procesador de un lenguaje que ofrezca polimorfismo, deberá implementar un mecanismo de unificación: proceso de encontrar una sustitución para cada variable de tipo con otra expresión de tipo, acorde al empleo del operador o función polimórfico [23]. Si tenemos una función que recibe cualquier tipo y devuelve su dirección, ante la invocación de ésta con una variable entera, deberá ser capaz de inferir que dicha invocación posee el tipo puntero a entero. Esta inferencia se lleva a cabo mediante un algoritmo de unificación. A modo de ejemplo, la siguiente es una entrada válida para nuestro procesador:

```
1:      int vector[10];
2:      template <typename T> struct Registro { T field; };
3:      template <typename T> struct Lista {
4:            T uno;
5:            T* muchos; };
6:      template <typename T> T *f(T);
7:      template <typename T> T g(T,T);
8:      template <typename T1, typename T2> T2 *h(T1,T2);
9:      struct Registro<bool> registroBool;
10:     struct Lista<int> listaInt;
11:
12:     int main() {
13:           f(3);                          Puntero(int)
14:           g(3,'3');                      int
15:           h(3.3,vector);                 Puntero(Array(10,int))
16:           f<int>(true);                  Puntero(int)
17:     }
```

Figura 13: Programa de entrada que utiliza polimorfismo.

El programa C++ anterior requiere un proceso de unificación en la inferencia de tipos. Por ejemplo, la línea 15 invoca a la función `h` con dos parámetros; es necesario inferir el tipo de ambos y unificar éstos a las variables de tipo `T1` y `T2`; una vez unificadas, se podrá saber que el tipo devuelto es un puntero a *array* de enteros (puntero a T2). Nótese cómo, además de unificación, las líneas 14 y 16 también emplean coerción de tipos.

En nuestro diseño, es necesario añadir un nuevo tipo de expresión de tipo: `VariableTipo`. Esta clase deberá tener un atributo que indique el nombre de la variable de tipo, necesario para la unificación. Adicionalmente, se añadirá un método `unificar` a todos los elementos de la jerarquía en el que se recibirá la expresión de tipo a unificar y la lista de sustituciones (asociación de variables de tipo a expresiones de tipo). La devolución de unificar es el tipo unificado.

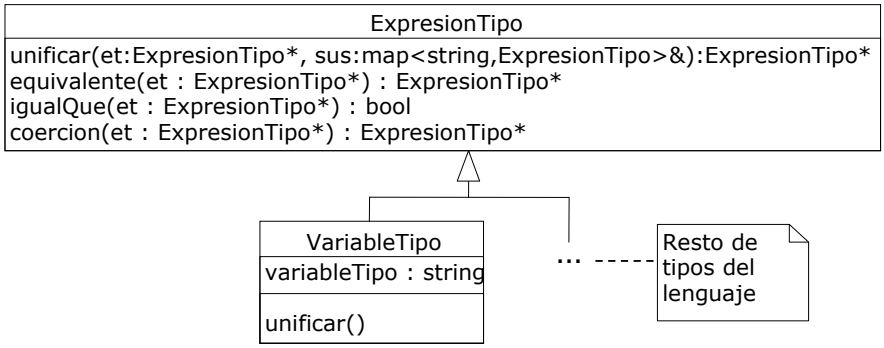

Figura 14: Diagrama de clases que permite polimorfismo.

La implementación del método `unificar` de cada clase es sencilla:

- Los tipos simples, al no tener variables de tipo, devolverán una llamada al método `equivalente`. Ésta será la implementación por omisión de `ExpresionTipo`.
- Los tipos compuestos comprobarán que son el mismo constructor de tipo y llamarán al método unificar de cada uno de sus tipos base. La devolución será una expresión de tipo con el mismo constructor de tipo y las unificaciones de cada uno de sus tipos base.
- Finalmente, la variable de tipo buscará en las sustituciones (segundo parámetro) si su variable de tipo existe. En caso afirmativo devuelve la expresión de tipo asociada a ésta; si no, asigna el tipo a unificar (primer parámetro) a su variable de tipo y devuelve el mismo (Figura 15).

```
ExpresionTipo *VariableTipo::unicar(ExpresionTipo *et, map<string,
ExpresionTipo *> &sus) {
    if ( sus.find(variableTipo)!=sus.end() )
          return sus[variableTipo];
    sus[variableTipo]=et;
    return et;
}
```

Figura 15: Implementación de la unificación de variables de tipo.

El comprobador de tipos invocará al método unificar cada vez que se llame a una función y cuando se trate de una unificación explícita por parte del programador (líneas 9, 10 y 16 de la Figura 13).

7 Conclusiones y Trabajo Futuro

En la actualidad existen diversos formalismos empleados para diseñar y verificar la validez de los sistemas de tipos de los lenguajes de programación. Las escasas herramientas que utilizan dichas notaciones para implementar comprobadores de tipos poseen un conjunto de limitaciones como la dependencia de un lenguaje específico, dificultades en la depuración y carencias de interacción y reutilización de componentes software. Para paliar estos inconvenientes, hemos realizado un diseño de comprobadores de tipos para cualquier lenguaje de programación, aportando los siguientes beneficios:

1. No se emplean notaciones distintas a las utilizadas en desarrollo software. La descripción de los sistemas de tipos, así como la demostración de que satisfacen determinadas condiciones, es comúnmente llevada a cabo a través de notaciones matemáticas tales como el cálculo lambda con tipos. Un desarrollador de un comprobador de tipos requiere implementar las reglas de inferencia y validez de los tipos en un determinado lenguaje de programación, probablemente sin tener conocimientos de las notaciones que fueron empleadas para definir el sistema de tipos.
2. Independencia del lenguaje. Puesto que lo que hemos presentado es un diseño, éste podrá ser implementado en cualquier lenguaje de programación que soporte las abstracciones de clase, encapsulamiento, herencia y polimorfismo.

3. Comprensible, mantenible y reutilizable. La sencillez del diseño hace que sea fácil de depurar, modificar y extender. Si una vez desarrollado un comprobador de tipos para un lenguaje de programación fuese necesario implementar un comprobador para otro lenguaje, se podría reutilizar todo el código representante de aquellas partes comunes a ambos sistemas de tipos. Puesto que el diseño del comprobador nunca cambia, será muy sencillo introducir las nuevas características del segundo sistema de tipos.

4. El sistema puede ser empleado para implementar sistemas de tipos tanto estáticos como dinámicos. Determinados lenguajes realizan la comprobación de tipos total o parcialmente en tiempo de ejecución. Este diseño se puede emplear en ambos escenarios.

5. Independencia del lenguaje a procesar y de las herramientas. Este diseño para desarrollar comprobadores de tipos puede emplearse junto a cualquier tipo de herramientas de generación de compiladores. Adicionalmente, se basa en conceptos tales como expresión de tipo, tipo primitivo, constructor, equivalencia y coerción de tipos o polimorfismo; conceptos que no son propios de un paradigma concreto y que aparecen en sistemas de tipos de múltiples lenguajes.

6. Beneficios propios del empleo de patrones de diseño y orientación a objetos. La utilización de patrones de diseño orientados a objetos, altamente probados y revisados, hace que los diseños que los empleen correctamente adopten los beneficios de éstos [20].

El diseño aquí presentado ha sido empleado en el desarrollo de múltiples procesadores de lenguaje tales como la implementación de una máquina virtual [24] o un intérprete genérico independiente del lenguaje [25], así como su utilización a nivel docente e investigador (APLT, *Advanced Programming Languages and Technologies,* http://www.di.uniovi.es/~labra/APLT). El diseño se ha implementado en los lenguajes C++ y Java, junto con las herramientas Lex/Yacc, AntLR y JavaCC.

En la actualidad se está desarrollando una herramienta de generación de comprobadores de tipos capaz de, basándose en los diseños presentados, generar automáticamente comprobadores de tipos para lenguajes de programación (http://www.di.uniovi.es/tys).

References

1. Whitehead, A. N., Russell, B.: Principia Mathematica. Cambridge University Press, Cambridge, MA (1910).
2. Pierce, B.C.: Types and Programming Languages. MIT Press (2002).
3. Cardelli, L.: Type Systems. The Computer Science and Engineering Handbook, CRC Press (2004).
4. Aho, A.V., Ullman, J.D., Sethi, R.: Compilers: Principles, Techniques, and Tools. Addison Wesley (1985).
5. Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating objects with confined types. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) (2001).
6. Richardson, D.J.: TAOS: Testing with analysis and Oracle support. International Symposium on Software Testing and Analysis, (1994).
7. Hosoya, H., Pierce, B.C.: XDuce: A statically typed XML processing language. ACM Transactions on Internet Technology (TOIT) 3(2) (2003).
8. Abadi, M: Security Protocols and Specifications. Foundations of Software Science and Computation Structures (FOSSACS) (1999).
9. Pitts, A.M.: Operational Semantics and Program Equivalence. Applied Semantics, Lecture Notes in Computer Science, Tutorial, Volume 2395, Springer-Verlag (2002).
10. Berardi S.: Towards a mathematical analysis of the Coquand-Huet Calculus of Constructions and the other systems in Barendregt's cube. Technical report, Department of Computer Science, Carnegie-Mellon University (1988).
11. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory 2 (2). (1968)
12. Levin, M.Y., Pierce, B.C.: TinkerType: a language for playing with formal systems. Journal of Functional Programming 13(2) (2003).
13. Rustan, K., Leino, M., Nelson, G., Saxe, J.B.: ESC/Java User's Manual.Technical Note 2000-002, Compaq Systems Research Center (2000).
14. Ball, T., Rajamani, S.K.: The SLAM Project: Debugging System Software via Static Analysis. Principles of Programming Languages (POPL) (2002).
15. Holzmann, G.J., Holzmann, G.J.: The Spin Model Checker. Addison Wesley (2003).
16. Reps, T.W., Teitelbaum, T.: The Synthesizer Generator. Software Development Environments (SDE) (1984).
17. Brand, M., Deursen, A., Heering, J., Jong, H.A., Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., Visser, J.: The ASF+SDF Meta-environment: A Component-Based Language Development Environment. Compiler Construction (CC), European Joint Conference on Theory and Practice of Software (ETAPS) (2001)

18. Scott, M.L.: Programming Language Pragmatics. Morgan Kaufmann Publishers (2000).
19. Booch, G.: Object-Oriented Analysis and Design With Applications, 2nd Edition. Benjamin Cummings (1994).
20. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison Wesley (1995).
21. Welsh, J., Sneeringer, M.J., Hoare, C.A.R.: Ambiguities and Insecurities. Pascal, Software Practice and Experience7 (6) (1977).
22. Strachey, C.: Fundamental concepts in programming languages. Lecture notes for the International Summer School in Computer Programming (1967).
23. Milner, R.: A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences 17(3) (1978).
24. Ortin, F., Cueva, J.M., Martínez, A.B.: The reflective nitrO abstract machine. ACM SIGPLAN Notices 38(6) (2003)
25. Ortin, F, Cueva, J.M.: Dynamic Adaptation of Application Aspects. Elsevier Journal of Systems and Software 71(3) (2004)