Herramientas de Refactor e Información Contextual basado en Inferencia de Tipos para lenguajes tipo-inconsientes. – Propuesta de tesis doctoral –

Pablo Tesone

26 de junio de 2015

1. Objetivos

1.1. Marco General

Los sistemas de tipos[9] son una aplicación de la teoría de tipos[9], surgida en estudios sobre fundamentos de la matemática, a los lenguajes de programación.

El uso de sistemas de tipos en programación tiene como objetivo inicial la detección de los llamados errores de tipo[7], que ocurren cuando se aplica una operación a un valor para el que dicha operación no tiene sentido. Por ejemplo, pasar a mayúscula un número constituye un error de tipo, porque la operación de pasar a mayúscula sólo tiene sentido para textos, no para números. "Texto" y "número" son posibles tipos básicos de un sistema de tipos.

La aplicación de un sistema de tipos permite restringir el conjunto de programas correctos a aquellos para los que pueda garantizarse la ausencia de este tipo de errores. Para ello, se debe conocer el tipo de cada elemento incluido en un programa, y restringir cada operación imponiendo que sus argumentos deben respetar un determinado tipo[7]. Este es el comúnmente llamado *chequeo estático de tipos* presente en varios lenguajes de programación, como Java o C entre otros.

En el siguiente ejemplo, la operación tiene sentido ya que el valor almacenado en `v1` es un "Texto":

```
var v1 = "hola"
v1.pasarAMayuscula()
```

Mientras que en el siguiente, la misma no tiene sentido ya que el valor almacenado en `v2` es un "Número" y la operación `pasarAMayuscula` no esta definida para los números. Es deseable que esta situación sea detectada en tiempo de compilación y que no se produzca un error al ejecutar el programa:

```
var v2 = 42
v2.pasarAMayuscula()
```

Observamos que el uso de un sistema de tipos implica contar con información valiosa sobre los distintos componentes de un programa. Esto da lugar a un efecto adicional, que es el aprovechamiento que un entorno de desarrollo[4] puede realizar de dicha información, proponiendo posibilidades para completar una línea de código que se está escribiendo (la llamada ayuda sensible al contexto)[4], y permitiendo una mayor precisión y riqueza de las acciones de reorganización de código conocidas como *refactors*[8]. La potencia de los entornos de desarrollo actualmente disponibles les permite brindar capacidades extensivas en este sentido, lo que redunda en un ahorro de tiempo considerable en las tareas que llevan a cabo los programadores en su trabajo cotidiano. Por lo tanto, este aspecto del uso de sistemas de tipos en lenguajes de programación tiene una gran relevancia hoy en día.

El uso de sistemas de tipos también tiene consecuencias no deseadas[6, 5]. Entre ellas, destacamos la necesidad de escribir código adicional, cuyo único efecto es especificar los tipos de cada componente de un programa. Esto puede implicar que se agregue, dentro del código que compone un programa, una gran cantidad de definiciones, que deben ser generadas y luego mantenidas. Otra desventaja potencial del uso de sistemas de tipos es que el criterio de corrección de un programa podría resultar demasiado restrictivo, impidiendo formas de estructurar programas que podrían resultar útiles[3] para lograr programas más fácilmente extensibles, o componentes que pudieran ser reutilizados en distintos programas.

La industria del desarrollo de software está presenciando, en estos años, una creciente utilización de lenguajes de programación que no utilizan sistemas de tipos. En distintas fuentes de la literatura se asocian los términos *débilmente tipado*, *dinámicamente tipado* y *no tipado*[9, 3] para referirse a esta característica que puede presentar un lenguaje de programación. En este documento, hablaremos de lenguajes *tipo-inconscientes*. Algunos de estos lenguajes, tales como Javascript, Ruby, Php, Smalltalk y Python, son extensamente utilizados, en particular en el desarrollo de aplicaciones Web y para dispositivos móviles.

Los lenguajes de programación tipo-inconsientes evitan contener en el texto del programa información de tipos, haciendo que el código que se requiere escribir para realizar una determinada actividad sea mucho más sintético, y por lo tanto mantenible y flexible.

Por otro lado, en estos lenguajes no se cuenta con información sobre el tipo requerido de cada componente de un programa. En consecuencia no se pueden proporcionar, en los entornos de desarrollo, herramientas automatizadas que basan su comportamiento en la información que brinda un sistema de tipos, de acuerdo a lo descripto anteriormente.

Un ejemplo de estas herramientas es la capacidad de cambiar el nombre de una operación en su definición y en todos los lugares donde se usa; todo de forma automática. Supongamos que tenemos dos operaciones llamadas "multiplicar", una sobre los números (que multiplica el número original con el pasado como parámetro) y una sobre los textos (que repite el texto la cantidad de veces recibida por parámetro). Por motivos de calidad del código, el programador desea cambiar el nombre la de la segunda. Pero este cambio solo se debe hacer en los usos donde el objeto modificado es un texto, manteniendo sin cambio el uso sobre los números. Para poder lograrlo es necesario contar con el detalle del tipo de cada parte del programa.

Una estrategia utilizada para paliar la ausencia de información de tipos en un programa es el proceso conocido como *inferencia de tipos*[7], en el cual se obtiene cierto nivel de información a partir del análisis de las estructuras de un programa dado. Sin embargo, la inferencia de tipos utilizando los enfoques más estudiados en la literatura, no puede ser aplicado a los lenguajes tipo-inconsientes orientados a objetos que son utilizados en la industria, como los mencionados anteriormente.

1.2. Objetivos Generales

Para programas escritos en lenguajes tipo-inconscientes, la información relacionada con los tipos de cada componente que puede obtener un entorno de desarrollo es limitada. Esto es consecuencia de la situación descripta en el marco general. Por lo tanto, los entornos integrados de desarrollo y otras herramientas que utilizan los usuarios de dichos lenguajes, resultan limitados respecto de las características de refactors e información contextual que proveen, si los comparamos con herramientas para lenguajes con sistemas de tipos explicito como son *Eclipse* para Java o *Visual Studio* para C#. Existen algunas propuestas de definir sistemas de tipos y herramientas que puedan aplicarse a lenguajes tipo-inconscientes, que no apuntan a poder realizar chequeos estáticos de tipos, sino a proporcionar información sobre los componentes de un programa, que pueda ser explotada de distintas formas. Estas propuestas existentes no permiten su utilización en tiempo real en programas de nivel industrial, ya que o no cubren la totalidad del las funciones del lenguaje[10] o no lo logran con la eficiencia requerida para funcionar en tiempo real[1].

Además existen sistemas de tipos conocidos como *Pluggable Type Systems*[2], que estan concebidos como elementos externos al lenguaje de programación que pueden ser utilizados de forma opcional como herramientas en distintas etapas del desarrollo.

El primer objetivo general del trabajo propuesto es definir un conjunto de algoritmos y mecanismos para la obtención de información de tipos, que pueda ser utilizado en entornos de desarrollo y lenguajes de programación de uso en la industria del software. Esta información, aunque incompleta para realizar una validación de los programas, es extremadamente útil para mejorar la asistencia automática que las herramientas de desarrollo pueden realizar, de acuerdo a lo descripto en el marco general; siendo posible habilitar o deshabilitar las mismas siguiendo las ideas planteadas por *Pluggable Type Systems*.

Como caso de estudio de los algoritmos y técnicas desarrollados, realizaremos las implementaciones correspondientes para el lenguaje Smalltalk, en el ambiente de desarrollo Pharo[1]. Por lo tanto, el segundo objetivo general es el desarrollo de herramientas de análisis y modificación automáticos de programas, que se integren en el entorno Pharo. Estas herramientas podrán ser aprovechadas por el conjunto de desarrolladores que utilizan, durante su desempeño profesional, el lenguaje y entorno de desarrollo mencionados.

[1] Pharo es un ambiente de desarrollo e implementación de Smalltalk. http://www.pharo.org

1.3. Objetivos Específicos

Se propone:

1. Formular un algoritmo de inferencia de tipos que tenga en cuenta todas las características dinámicas del lenguaje Smalltalk, y en particular de la implementación de dicho lenguaje incluida en el entorno Pharo.

2. Realizar una implementación del algoritmo de inferencia de tipos, que permita una ejecución, de forma eficiente e incremental, en tiempo real durante la utilización del entorno de desarrollo. El objetivo de esta implementación es proveer información y sugerencias mientras el usuario escribe el programa.

3. Realizar un análisis del sistema de tipos del conjunto de librerías (componentes básicos) y programas escritos en Pharo para validar la correcta generación de la información y los tiempos de ejecución del algoritmo.

4. Integrar la implementación del algoritmo dentro del conjunto de herramientas presentes en Pharo.

5. Aprovechar la información de tipos para realizar sugerencias y proveer información contextual durante la edición del programa.

6. Aprovechar la información de tipos para la implementación de herramientas automatizadas de modificación del programa.

2. Antecedentes

Existen trabajos previos junto a los directores de esta propuesta en temas directamente relacionados con el trabajo propuesto[2]; cuyos detalles se listan abajo.

- Nicolás Passerini, Pablo Tesone, Stéphane Ducasse, An extensible constraint based type inference algorithm for object-oriented dynamic languages supporting blocks and generic types, International Workshop of Smalltalk Technologies (IWST 2014), Reino Unido, 2014.

- Stéphane Ducasse, Robb Nebbe and Tamar Richner, Two Reengineering Patterns: Eliminating Type Checking, Proceedings of the 4th European Conference on Pattern Languages of Programming and Computing, 1999

- Tudor Gîrba, Jean-Marie Favre and Stéphane Ducasse, Using Meta-Model Transformation to Model Software Evolution, Proceedings of 2nd International Workshop on Meta-Models and Schemas for Reverse Engineering (ATEM 2004), 57–64, 2004

- Stéphane Ducasse, Matthias Rieger and Georges Golomingi, Tool Support for Refactoring Duplicated OO Code, Proceedings of the ECOOP '99 Workshop on Experiences in Object-Oriented Re-Engineering, Forschungszentrum Informatik, Karlsruhe, 1999

- Oscar Nierstrasz, Stéphane Ducasse and Serge Demeyer, Object-oriented Reengineering Patterns — an Overview, Proceedings of Generative Programming and Component Engineering (GPCE 2005), 1–9, LNCS 3676, 2005

Asímismo, tanto el postulante como los directores propuestos han realizado varios trabajos referidos al estudio de lenguajes de programación, la implementación de los mismos y la construcción de herramientas de desarrollo, temas ligados a la realización de la presente propuesta. En tal sentido, destacamos:

- Nicolás Passerini, Javier Fernandes, Ronny De Jesús, Pablo Tesone, Leonardo Gassman, Enhancing Binding-based User Interfaces with Transaction Support, Foundations of Object-Oriented Languages (FOOL 2013), Estados Unidos, 2013

- Nicolás Passerini, Javier Fernandes, Pablo Tesone, Wollok? Relearning How To Teach Object-Oriented Programming, Workshop de Ingeniería en Sistemas y Tecnología de la Información (WISIT 2014), Argentina, 2014

[2]Solamente se listan los trabajos que tienen mayor relación con la temática propuesta

- Jean-Baptiste Arnaud, Stéphane Ducasse, Marcus Denker and Camille Teruel, Handles: Behavior-Propagating First Class References For Dynamically-Typed Languages, Journal of Science of Computer Programming, 2014
- B. Accattoli, Eduardo Bonelli, D. Kesner and C. Lombardi,A Nonstandard Standardization Theorem, Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). ACM SIGPLAN-SIGACT, Estados Unidos, 2014.
- Eduardo Bonelli, Francisco Bavera, Justification Logic and History Based Computation, 7th International Colloquium on Theoretical Aspects of Computing (ICTAC'10), Brasil. LNCS 6255, pp. 337-351, 2010.

Cabe mencionar que se cuenta con amplia colaboración con el equipo que realiza el mantenimiento del entorno Pharo, en particular con el equipo que se encuentra trabajando en el laboratorio RMod de INRIA Nord Europe.

Un ejemplo de esta continua colaboración es que el postulante realizará una estadía durante los meses de Agosto a Diciembre de 2015, en el laboratorio antes señalado para la continuación de la línea de investigación propuesta en *An extensible constraint based type inference algorithm for object-oriented dynamic languages supporting blocks and generic types*. El financiamiento de dicha estadia es realizado por INRIA Nord Europe y Pharo Consortium.

También existe una amplia comunidad de usuarios de Pharo en Argentina y Europa; el postulante y el director propuesto mantienen un contacto regular con varios miembros de dicha comunidad.

3. Actividades y Metodología

Las actividades a realizar se dividen en las siguientes etapas:

1. Revisión de la literatura existente. Esto incluye métodos tradicionales de inferencia de tipos, y aplicaciones existentes sobre lenguajes de objetos tipo-inconscientes.
2. Desarrollar un algoritmo que de forma iterativa e incremental vaya generando la mayor cantidad posible de información a partir de un programa completo o incompleto.
3. Realizar un prototipo de este lenguaje para probar el nivel de información provisto sobre programas reales escritos para Pharo.
4. Realizar una implementación eficiente en tiempo de ejecución y espacio de memoria del algoritmo desarrollado.
5. Integrar esta herramienta desarrollada en el proceso de edición y compilación de los programas en Pharo.
6. Implementar herramientas de sugerencias y refactors basadas en la información obtenida por la inferencia de tipos.

4. Factibilidad

El Departamento de Ciencia y Tecnología de la Universidad Nacional de Quilmes cuenta con la infraestructura necesaria, que en este caso consiste principalmente de espacio físico, mobiliario de oficina, acceso a conectividad a Internet, impresora (tóner y papel) y una PC o portable.

Actualmente los directores cuentan con proyectos UBACyT y PUNQ que servirían para cubrir las necesidades antes mencionadas. También hay financiamiento por parte de INRIA Lille Nord Europe y de Pharo Consortium para estadías en Francia.

Referencias

[1] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications.* Ph.D. thesis, Stanford University, December 1996.

[2] Gilad Bracha. Pluggable type systems, October 2004. OOPSLA Workshop on Revival of Dynamic Languages.

[3] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[4] A Nico Habermann and David Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, (12):1117–1127, 1986.

[5] Stefan Hanenberg. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In *ACM Sigplan Notices*, volume 45, pages 22–35. ACM, 2010.

[6] Sebastian Kleinschmager, Stefan Hanenberg, Romain Robbes, É Tanter, and Andreas Stefik. Do static type systems improve the maintainability of software systems? an empirical study. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 153–162. IEEE, 2012.

[7] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[8] William F. Opdyke. *Refactoring Object-oriented Frameworks*. PhD thesis, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

[9] Benjamin Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[10] Frédéric Pluquet, Antoine Marot, and Roel Wuyts. Fast type reconstruction for dynamically typed programming languages. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 69–78, New York, NY, USA, 2009. ACM.