JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 9, November-December 2005

A Formal Description Of A Type Checking Algorithm

Majid Naeem, University of Central Punjab, Lahore, Pakistan
Chris Harrison, School of Informatics, Manchester University, UK

Abstract

In type system theory, a program variable can assume a range of values during the execution of a program. In a statically typed language, every expression of the language is assigned a type at compile time. There are many advantages to having a statically type-checked language including provision of earlier information on programming errors, documenting component interfaces, eliminating the need for run-time type checks, and providing extra information that can be used in compiler optimization. In this paper a formal description of a static type checker is presented and used to construct a static type for each expression in an object-oriented language called POOL.

1 INTRODUCTION

In object-oriented languages, a subclass enables reuse of the code of its superclass and relies on the type-correctness of the corresponding source code. A particular task associated with statically type checking an object-oriented language is designing the type checking rules which ensure that methods provided in a superclass will continue to be type-correct when inherited in a subclass. A set of typing rules, based on the structure of expressions, can be used to construct a static type for each expression at compile time by the type checker. The type checker is thus able to guarantee that if an expression has a static type T, evaluation of that expression at run-time will result in a value of type T.

POOL [1], [2] is a class-based strongly typed object-oriented language. In POOL, a program may consist of a number of user-defined types which may appear in any declaration order. POOL also supports parametric type definitions, a general technique that enables the same piece of code to be used by different types. In POOL, a parametric type that has been inferred from a program fragment may take on a different instance in every context in which it is used. Languages which also support type parameters include Trellis/Owl [8], Eiffel [9], and PolyTOIL [11].

In POOL, a relationship between types enables an object of a subtype to be used in any context that expects an object of the supertype. This process is termed Subsumption [3, 12]. Thus, if β is a subtype of α, i.e. ($\beta \leq \alpha$), then any expression of type β may be

Cite this article as follows: M. Naeem and Ch. Harrison: "A Formal Description of a Type Checking Algorithm", in *Journal of Object Technology*, vol. 4, no. 9, November - December 2005, pp. 93 - 100 http://www.jot.fm/issues/issue_2005_11/article2

used without type error in any context that requires an expression of type α. Thus, if Γ is a static typing environment and it is well-formed ($\Diamond$), then we can assign (:=) the value of an expression (or term) of type β to an object of type α without any type error.

$$\frac{\Gamma \vdash \Diamond \quad \Gamma \vdash b:\beta \quad \Gamma \vdash a:\alpha \quad \Gamma \vdash \beta \leq \alpha}{\Gamma \vdash a:=b}$$

The same substitutivity rule can be applied to functions. If α, α', β, and β' are four types such that α' is a subtype of α and β' is a subtype of β, and two functions ϕ and φ are defined in these types such that the type of function ϕ is $\phi : \alpha' \rightarrow \beta'$ and the type of function φ is $\varphi : \alpha \rightarrow \beta$, then the function ϕ can be substituted for function φ [13].

$$\frac{\Gamma \vdash \Diamond \quad \Gamma \vdash \varphi : \alpha \rightarrow \beta \quad \Gamma \vdash \phi : \alpha' \rightarrow \beta' \quad \Gamma \vdash \alpha' \leq \alpha \quad \Gamma \vdash \beta \leq \beta'}{\Gamma \vdash \varphi \leq \phi}$$

This substitution principle is extended to the class-based model of a language. Thus, a type β is a subclass of a class α if β is derived from α, either by updating or modification of its methods ($\beta <_m \alpha$) or by extension, i.e. extending its attributes or introducing new methods ($\beta <_e \alpha$) [4, 12].

If α is composed of m_i attributes and methods, and β is composed of m_j attributes and methods, then if m_j is the same as m_i (or m_j has additional attributes or methods), then β is a subclass of α by extension $<_e$.

$$\frac{\Gamma, \alpha, \beta \vdash \Diamond \quad \Gamma \ \alpha \equiv \{m_i : M_i\} \quad \Gamma \ \beta \equiv \{m_j : M_j\} \quad i \in 1\ldots n \quad j \in 1\ldots m \quad m >= n}{\Gamma \vdash \beta <_e \alpha}$$

If α has a method m_i of type $m_i : P_i \rightarrow R_i'$ and β has a method m_j of type $m_j : P_i \rightarrow R_i$ such that R_i' is a subtype of R_i, then β is a subclass of α by modification $<_m$.

$$\frac{\Gamma, \alpha, \beta \vdash \Diamond \quad \Gamma \vdash \alpha \equiv \{m_i : P_i \rightarrow R_i'\} \quad \Gamma \vdash \beta \equiv \{m_i : P_i \rightarrow R_i\} \quad \Gamma \vdash R_i' \leq R_i \quad i \in 1\ldots n}{\Gamma \vdash \beta <_m \alpha}$$

If α has a method m_i of type $m_i : P_i \rightarrow R_i$ and β has a method m_j of type $m_j : P_i' \rightarrow R_i'$ such that P_i' is a subtype of P_i, then β is a subclass of α by modification $<_m$.

$$\frac{\Gamma,\alpha,\beta \vdash \Diamond \quad \Gamma \vdash \alpha \equiv \{m_i : P_i \to R_i\} \quad \Gamma \vdash \beta \equiv \{m_i : P_i' \to R_i\} \quad \Gamma \vdash P_i' \leq P_i \quad i \in 1\ldots n}{\Gamma \vdash \beta <_m \alpha}$$

If α has a method m_i of type $m_i : P_i \to R_i'$ and β has a method m_j of type $m_j : P_i' \to R_i$ such that P_i' is a subtype of P_i and R_i' is a subtype of R_i, then β is a subclass of α by modification $<_m$.

$$\frac{\Gamma,\alpha,\beta \vdash \Diamond \quad \Gamma \vdash \alpha \equiv \{m_i : P_i \to R_i'\} \Gamma \vdash \beta \equiv \{m_i : P_i' \to R_i\} \quad \Gamma \vdash R_i' \leq R_i \quad \Gamma \vdash P_i' \leq P_i \quad i \in 1\ldots n}{\Gamma \vdash \beta <_m \alpha}$$

2 TYPE CHECKING IN POOL

The task of the type checker is to verify that a program is type correct. Ideally, type checking takes place before the program is run, in which case the program is said to be statically type correct and the corresponding type system is termed a static type system. If an implementation verifies type correctness during a program's execution, then the program is said to be dynamically type correct and its system is a dynamic type system.

The static type checker designed and implemented for POOL exploits an enhanced static type-checking mechanism to minimize run-time checks for late bindings. This mechanism is realized via a small set of type-operation look-up tables that provide run-time support for dynamic type-checks. POOL follows the typing rules of a general class-based object-oriented language, the most common of which are given in [6, 7].

The type checker is responsible for ensuring that the typing rules of the language are enforced. In practice, type checking is done by 'bottom up' inspection of a program, matching and synthesizing types while proceeding towards the root; the type of predefined identifiers is already "known" and contained in the initial environment, whereas the type of an expression is computed from the type of its sub-expressions and type constraints imposed by the expression's context. In POOL, the type of an object, an operation and an expression is determined by the rules defined in the following sections.

Object Type

In POOL, an object 'τ' can have more than one type, i.e. it may be defined as a variant object type. Thus, if Γ is a well-formed ($\Diamond$) static typing environment which has types $\alpha_1, \alpha_2, \ldots, \alpha_n$ an object τ can be defined with a single type $\alpha_1, \alpha_2, \ldots, \alpha_n$, or more than one type $\alpha_1, \alpha_2, \ldots, \alpha_n$,.

$$\frac{\Gamma \vdash \Diamond \quad \Gamma \vdash \alpha_i \quad i \in 1..n}{\Gamma \vdash (\tau : \alpha_1 | \ldots | \alpha_n)}$$

Operation Type

In POOL, an operation defined in a type can have a parameter of a variant type and return a value of more that one type as an allowed return type, i.e. POOL supports polymorphic functions and operations. Thus, if Γ is a well-formed static typing environment which has types $\alpha_1, \alpha_2, \ldots, \alpha_n$ and $\beta_1, \beta_2, \ldots, \beta_m$ an operation f can be defined in type β_j or α_i, where `i ∈ 1...n`, with a parameter of type α_i and a return type β_j or vice versa.

$$\frac{\Gamma \vdash \Diamond \quad \Gamma \vdash \alpha_i \quad \Gamma \vdash \beta_j \quad i \in 1\ldots n \quad j \in 1\ldots m}{\Gamma \vdash (f : \alpha_1 | \ldots | \alpha_n \rightarrow \beta_1 | \ldots | \beta_m)}$$

$f : \alpha_1 | \ldots | \alpha_n \rightarrow \beta_1 | \ldots | \beta_m$ indicates that function f has a parameter of type $\alpha_1 | \ldots | \alpha_n$ and returns a value of type $\beta_1 | \ldots | \beta_m$.

Expression Type

If Γ is a well-formed ($\Diamond$) static typing environment which has types α, $\beta_1, \beta_2, \ldots, \beta_n$ and $\gamma_1, \gamma_2, \ldots, \gamma_m$, and a polymorphic function or operation, $f : \beta_1, \beta_2, \ldots, \beta_n \rightarrow \gamma_1, \gamma_2, \ldots, \gamma_n$, then an object τ of type α can be applied to an operation of type $f(\tau) \rightarrow f(\tau)$ iff α and β_i `(i ∈1…n)` have a Least Common Type (LCT).

$$\frac{\Gamma \vdash \Diamond \quad \Gamma \vdash \tau : \alpha \quad \Gamma \vdash f : \beta_1 | \ldots | \beta_n \rightarrow \gamma_1 | \ldots | \gamma_m \quad \Gamma \vdash \mathtt{LCT}(\alpha) \ \mathtt{in}\ \mathtt{LCT}\ (\beta_1 | \ldots | \beta_n)}{\Gamma \vdash f(\tau) \rightarrow \gamma_1 | \ldots | \gamma_m}$$

LCT (α) in `LCT` $(\beta_i | \ldots | \beta_n)$ indicates that both α and $\beta_1, \beta_2, \ldots, \beta_n$ have the same Least Common Type.

3 THE TYPE CHECKING ALGORITHM

The main task of a type checker is to make sure that whenever an expression is evaluated and assigns a value to an object there should be no typing error. In order to do so, the type checker ensures that the type of the value resulting from an expression's evaluation is the same type as the object to which this value is to be assigned. Many type-checking algorithms do this checking by computing a Least Common Type (LCT) for the expression and object [10].

In POOL, the algorithm for type checking is composed of four cases. Each case is based upon the type of expressions (which may have either simple or variant types) and the type of objects (which again may have either simple or variant types), where objects will receive the expression's value. The types of objects are termed henceforth '`term b`' and the types of expressions '`term a`' respectively.

Case 1. If **`term a`** has a type α and **`term b`** has a type β, then if β is a subtype of α ($\beta \leq \alpha$) and **`term b`** is assigned (`:=`) to **`term a`**, then the static typing environment will remain well formed ($\Diamond$).

$$\frac{\Gamma \vdash a:\alpha \quad \Gamma \vdash b:\beta \quad \Gamma \vdash \beta \leq \alpha \quad \Gamma \vdash a:= b}{\Gamma \vdash \Diamond}$$

It is a simple Subsumption Principle [13].

Case 2. If **`term a`** has a type α and **`term b`** has a variant type $\beta_1|\ldots|\beta_n$, then **`term b`** can be assigned to **`term a`** iff types $\beta_1, \ldots, \beta_n$ are all subtypes of α, i.e. α is a Least Common Type of $\beta_1, \ldots, \beta_n$.

$$\frac{\Gamma \vdash a:\alpha \quad \Gamma \vdash b:\beta_1|\ldots|\beta_n \quad \Gamma \vdash \beta_i \leq \alpha \quad \Gamma \vdash a := b \quad i \in 1\ldots n}{\Gamma \vdash \Diamond}$$

Thus, if `x` is an object of type α, `y` is an object of type β, and `z` is an object of type β_1, where β_1 and β_2 are subtypes of type α, then according to the Subsumption Principle we can assign objects '`y`' and '`z`' to object '`x`'.

`x:=y` `x:=z`

This also means that if an object `y` has a type β_1 or β_2, then it too can also be assigned to object '`x`'.

`x:=y` `iff` $y:\beta_1|\beta_2$

If there are `n` types $\beta_1 \ldots \beta_n$ such that all these types are subtypes of type α, i.e. α is a Least Common Type of $\beta_1 \ldots \beta_n$, $\beta_i \leq \alpha \ \forall \ i \in 1\ldots n$, then an object `y` of type $y:\beta_1|\ldots|\beta_n$ can be assigned to an object x of type α.

`x:=y` `iff` $x:\alpha$

Case 3. If **`term a`** has a variant type $\alpha_1|\ldots|\alpha_n$ and **`term b`** has type β, then **`term b`** can be assigned to **`term a`** iff there is a type in $\alpha_1,\ldots,\alpha_n$ such that β is a subtype of one of α_i, $i \in 1\ldots n$.

$$\frac{\Gamma \vdash a:\alpha_1|\ldots|\alpha_n \quad \Gamma \vdash b:\beta \quad \Gamma \vdash \beta \leq \alpha_1|\ldots|\alpha_n \quad \Gamma \vdash a:=b}{\Gamma \vdash \Diamond}$$

Thus, if x is an object of type α_1, y is an object of type α_2 and z an object of type β, such that β is a subtype of α_1 , i.e. $\beta \leq \alpha_1$, then according to the Subsumption Principle we can assign object z to object x.

$$x:=z$$

This also means that if an object x has a type α_1 or α_2 then an object y of type β can be assigned to x.

$$x:\alpha_1|\alpha_2 \quad x:=y \quad \text{iff} \quad y:\beta$$

If there are types $\alpha_1,\ldots,\alpha_n$ such that one of these is a supertype of the type β, where $\beta \leq \alpha_i \quad i\in 1\ldots n$, then an object y of type β can be assigned to an object x of type $\alpha_1 |\ldots| \alpha_n$.

$$y:\beta \quad x:=y \quad \text{iff} \quad x:\alpha_1|\ldots|\alpha_n$$

Case 4. If **term a** has a variant type $\alpha_1 |\ldots| \alpha_n$ and **term b** has a variant type $\beta_1 |\ldots| \beta_m$, then **term b** can be assigned to **term a** iff for every type β_j in $\beta_1 |\ldots| \beta_m$ there is a type α_i in $\alpha_1, \ldots, \alpha_n$ such that β_j is a subtype of $\alpha_i \quad i\in 1\ldots n$.

$$\frac{\Gamma \vdash a:\alpha_1|\ldots|\alpha_n \quad \Gamma \vdash b:\beta_1|\ldots|\beta_m \quad \Gamma \vdash \beta I \leq \alpha_1|\ldots|\alpha_n \quad \Gamma \vdash a:= b}{\Gamma \vdash \Diamond}$$

Thus, if x is an object of type α_1, a is an object of type β_1 and b an object of type β_2, such that β_1 and β_2 are supertypes of type α_1, then according to the Subsumption rule we can assign object a or b to object x.

$$x:=a \qquad x:=b$$

This also means that if the object x has a type α_1 or α_2 then the object a of type β_1 or β_2 can be assigned to x.

$$x:\alpha_1|\alpha_2 \quad x:=a \quad \text{iff} \quad a:\beta_1|\beta_2$$

If there are types $\beta_1,..,\beta_n$ such that all these types are subtypes of the type α_j, i.e. $\beta_I \leq \alpha_j$ $\forall i\in 1...n, \quad j\in 1...m$, then an object y of type $y:\beta_1|\ldots|\beta_n$ can be assigned to an object x of type $\alpha_1|\ldots|\alpha_m$.

4 CONCLUSION

The main justification for statically checking expressions lies in the nature of the guarantee that can be given, i.e. in a strongly-typed system, all code is guaranteed to be type-correct before execution. In practice, checking may be static or dynamic and involves analyzing the program text, or alternatively, accessing the run-time environment. As a "rule-of-thumb" it is almost always preferable to check statically as much as possible [5].

The type checker presented here supports type checking of polymorphic operations and variant objects, i.e. operations which have variant parameter types and variant return types, and guarantees that run-time errors of the kind "Message not Understood" are not generated. Once a class is type checked in this manner, it will remain type correct when inherited in other classes.

REFERENCES

[1] C.J. Harrison, M. Naeem. POOL: A Persistent Object-Oriented Language. ACM Symposium on Applied Computing, 2000.

[2] C. J. Harrison, M. Naeem and S. E. Eldridge. A Model-Oriented Programming Support Environment for Understanding Object-Oriented Concepts. Workshop 8, ECOOP 2000, France, in Lecture Notes in Computer Science, Vol. 1964/2000, Springer-Verlag Heidelberg, ISSN: 0302-9743

[3] J. Fisher and C. Mitchell. Notes on Typed Object-Oriented Programming. In Proceedings Theoretical Aspects of Computer Software, pages 844-885. Lecture Notes in Computer Science, Springer-Verlag Vol. 789, 1994.

[4] K.B. Bruce. A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics. Journal of Functional Programming 1(1):1-10, Jan 1999.

[5] Luca Cardelli. Type Systems. Handbook of Computer Science and Engineering, Chapter 103, CRC Press, 1997.

[6] Luca Cardilli, Jim Donahue, Misk Jordan, Bill Kalsow, Greg Nelson. The Modula-3 Type System. In Conference record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, Pages 202-212, January 1989.

[7] Martin Abadi, Luca Cardelli. A theory of Objects. Springer-Verlag New York, Inc. 1996.

[8] Schaffert, T. Cooper, B. Bullis, M. Kilian, C. Wilpolt. An introduction to Trellis/Owl. In OOPSLA'89 Proceedings, pages 9-16. ACM SIGPLAN Notices, 21(11), November 1986.

[9] Bertrand Meyer. Eiffel: the language. Prentice-Hall, 1992.

[10] P.L. Curien, G. Ghelli. Coherence of Subsumption, minimum typing and the type checking in $F_{\leq}$. Mathematical Structures in Computer Science, 2(1), pp 55-91, 1992.

[11] K.B. Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Object Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Gary T. Leavenand, Benjamin Pierce. On Binary Methods. Theory and Practice of Object Systems, 1(3), pp. 221-242, 1996.

[12] Luca Cardelli. A semantics of multiple inheritance. Information and Computation 76(2/3):138-164, 1988.

About the authors

Majid Naeem is a Professor in the Department of Computer Science at University of Central Punjab Pakistan. E-Mail: majid.naeem@pcit.ucp.edu.pk

Chris Harrison is a lecturer in the School of Informatics at Manchester University. E-Mail: cjharrison@manchester.ac.uk