Type System in Programming Languages

JIANG Hui (蒋 慧)[1], LIN Dong (林 东)[2], ZHANG Xingyuan (张兴元)[1] and XIE Xiren (谢希仁)[1]

[1] *Department of Computer Engineering, Institute of Communications Engineering Nanjing 210016, P.R. China*

[2] *National Defense University of China, Beijing 100091, P.R. China*

E-mail: osman@263.net; Donglin66@263.net

Received June 4, 2000; revised September 21, 2000.

Abstract Type system provides a precise description of a programming language. This is a prerequisite for the implementation and use of language. It also conducts mechanical and transparent type-checking on programs to prevent the occurrence of execution error during the running of programs. So, it can be said that, on the one hand, type system works as a formal tool to do mathematical analysis of language; on the other hand, it is a formal method for rigorously and precisely designing and implementing language. In this paper, some basic concepts of type system are discussed first. And then, the implementation of a graph-rewriting-based functional language — SClean's type system is given in details. It is hoped that the proposed method of using and implementing type system is of practical usefulness.

Keywords type system, type inference, type-checking, type theory, semantic model

1 Introduction

With its more and more involving into computer programming languages, type system is drawing an increasing attention of computer community. There are a lot of researches on the theory and application of type and type system. Barendregt[1,2] discusses two kinds of type systems for simple typed λ-calculus of Curry's and Church's, which gives the most original concepts of type system. Hindley[3] discusses a wide range of topics in functional language. Milner[4] gives the newest definition of Standard ML. They implemented the polymorphism and module. In [5] and [6], there are some classical discussions on polymorphism and object-oriented problems. [7] and [8] are two good comprehensive reviews of the first- and second-order type systems. Cardelli[9] and Hindley[10] discuss typeful programming and type assignment system in a unified type theoretic framework. Hofmann, Fisher, Benjamin *et al.* have done a lot on type-theoretic foundation for the object theory and application in process calculus[11−13]. Hudak *et al.* revised a high-order type system for the functional language Haskell[14]. Mitchell[15] gives a more semantics-oriented overview of type systems.

There are two major application directions of type system. One is its use as a formal analysis tool for a language. Type system provides a well-formed mathematical model for describing and analyzing a language's semantics. The other is its use as a formal method for rigorously and precisely developing a language. A well-defined type system is a rigorous specification of a language[16]. It is a prerequisite for the use and implementation of a language. Nowadays, type system has been seen as an important programming discipline, and it is the core of many newly developed languages.

The remainder of this paper is structured as follows. Section 2 discusses the motivations of type and type system in programming languages. Definitions of some new relevant

It is supported by the National Natural Science Foundation of China (No.69931040), and is also supported by Defense Science and Technology Funding No.GF00J6.2.1.JB0501.

concepts and terminology are also given. Section 3 and Section 4 discuss the theoretical and implementary part of type system. Section 5 gives the details of the implementation of SClean's type system. We hope our method to build and implement an actual type system will be of practical usefulness.

2 The Role of Type and Type System in Programming Languages

There are four motivations for type and type system in programming languages.

The first programming language that used type in 1954 is Fortran, where the first character is used to distinguish integer and floating type so as to provide good space management for the variables in computer memory and improve run-time efficiency. This is a basic motivation for the presence of type in modern languages.

ALGOL60 is the first language that explicitly used type information and did type-checking at compiling time. By associating type with each constant, operator, variable and function symbol, a static type structure is imposed on a program. This enforced restrictions on programs and provided a protective covering that hid the underlying representation and constrains appropriate operations at run time. Thus, some flaws can be detected in the early stage, i.e., at compiling time rather than at run time later. This improved the program's correctness and shortened debugging sessions. This is the second motivation for the use of type and type system.

The third motivation for type and type system in programming languages is their supporting for abstract data type and modularity. In some languages such as Modula-2, Adam, C++ and ML, type information can be organized as interfaces and hidden in independent modules or head files. Implementation modules depend only on their interfaces or head files. So compilation is more efficient because modules can be compiled independently. And as long as the interfaces are stable, changes to a module do not affect other modules. These are very useful in large-scale modularization software design and system reuse, which provide the foundation of software engineering.

The fourth motivation of type and type system in programming languages is their significant role in building a language's semantic model. The so-called mathematical semantics of programming language means to build a mathematics-based frame or model for the language, such as set theoretic model, recursion-theoretic model, category-theoretic model, etc. Then all the terms of the language can be denoted by objects in the model and explained based on the mathematics underlying the model. Type system bridges a language syntax and its semantic model, and helps to map syntactic objects to semantic correspondences.

2.1 New Relevant Concepts and Terminology

Although type system cannot reflect all the complexity of practical programming languages, it provides useful model when investigating some basic and actual problems. Some new concepts have appeared recently. They represent new development of type system and programming languages.

• Second- and higher-order. Many modern languages, such as ML, Miranda, Haskell etc., have the constructs of type parameters. With type parameters, first-order type system is extended to second-order type system and the language is more expressive.

• Polymorphism. It is the ability of a function to handle objects of many types, i.e., a function can have a unified behavior. There are many kinds of polymorphism, such as overloading, inheritance, parametric polymorphism and subtyping. C++ implements overloading and inheritance. And its template type is a kind of parametric polymorphism.

• Abstract data type (ADT). The most essential property of ADT is independence of representation, which means that the behavior of a program does not depend on the way

where data are represented in programs. This formed the basis for the later development of modular programming languages[7].

• Subtyping. Subtyping exists in various typed object-oriented programming languages. It is a relation between types. When we say type A is a subtype of type B, it means that any object of type A is an object of type B. Subtyping provides a very flexible extension mechanism that allows users to extend the data structures and operations.

• Modules and interfaces. Type information is encapsulated into interfaces for modules. Modules can then be compiled separately, with each module depending only on the interfaces of the others. So each can be designed and developed independently, and changes to one module do not affect others. These are very important in large-scale system engineering.

Some terms for further discussion are given below:

- Explicit typing language: a typed language where type definitions are part of the syntax.
- Implicit typing language: a typed language where type definitions are not part of the syntax.
- Type inference (type reconstruction): the process that type system infers the needed type information when there is little or no type information in the program.
- Type-checking: the process that type system checks whether the use of variables and functions in the program is consistent with their type declarations.
- Static checking: the process of type-checking or type inference that is completed at compiling time.
- Dynamic checking: the process of finding type errors done at run-time.
- Statically checked language: a typed language in which all the type-checkings are finished at compiling time before execution.
- Dynamically checked language: a typed language in which type-checkings are done at run-time.
- Type checker: in the interpreter or compiler, a part of semantic routine that does the type inference or type-checking work.

3 Basic Construction of Type System

We classify two statuses of type system for clearness and succinctness: the theoretical and implementary statuses of type system. Type assignment system (TAS) — the theoretical status of type system — consists of axioms, basis and inference rules. Type system — the implementary status of type system — involves the problems of implementing inference algorithm in the compiler or interpreter.

3.1 Formal Definition of Type System

Definition 1 (Type). *Type is defined inductively as follows*:

(1) *Each base type is a type,*

(2) *Type variable is a type* (*indefinitely*),

(3) *If* χ *is a type constructor with n parameters,* $\tau_1, \ldots, \tau_n$ *are any kind of types, then* $\chi\tau_1 \ldots \tau_n$ (*represented by* $\chi(\overline{\tau_n})$) *is a type,*

(4) *Nothing else is type.*

Base types are atomic types that a language gives, such as the four base data types in PASCAL: Integer, Char, Boolean, Real.

Type variable is a variable whose value can be any kind of types. Type constructor is a function of type. Its parameters are types. When applied to type parameters, a new kind of type is got.

Definition 2 (Type Statement). *Type statement is a formula with the form* $M : \tau$, *where* M *is a term, the subject,* τ *is a type, the predicate. The formula means that "*τ *is assigned to* M*" or "*M *has the type* τ*".*

Definition 3 (Basis). *Basis is any finite or infinite set of formulas* $\Gamma = \{M_1 : \tau_1, \ldots, M_n : \tau_n\}$, $M_j \neq M_i$ $(i \neq j)$, *where the subjects are different from each other.*

Definition 4 (Type Assignment System (TAS, or Type System)). *TAS is a natural deduction system, consisting of axioms, basis, and (inference) rules. Each formula* F *in basis has a deduction in TAS. It is a tree of formulas, with which the top is axioms or formulas in basis, the others are formulas being deduced from those immediately above them by a rule, with the bottom being* F*. If such a deduction exists, we say* $\Gamma \mapsto F$.

3.2 Sublanguage of Type System

It is deserved to point out that type system is independent of the programming language. For a programmer, type system is transparent to him and he can only see the syntax of the programming language. Generally, some different kinds of abstract syntax are used to express type formulas, called type expression sublanguage here, which can be used by other programming languages' type systems. So the sublanguage has the characteristics of general-purpose and transparency. Table 1 is the type expression sublanguage that SClean uses.

Table 1. The Type Expression Sublanguage of SClean's Type System

Type	::=	BaseType | ConstructedType | TypeVariable
TypeVariable	::=	$\alpha\|\beta\|\gamma\|\ldots$
BaseType	::=	Int | Char | Bool | String | Real
ConstructedType	::=	ArrowType | CardProdType | ListType | TupleType | RecordType;
ArrowType	::=	$\text{Type}_1 \rightarrow \text{Type}_2$;
CardProdType	::=	$\text{Type}_1 \times \text{Type}_2 \times \cdots \times \text{Type}_n$;
ListType	::=	[ ] | [Type_1, $\text{Type}_2, \ldots,$ Type_n] | [Type_1| ListType];
TupleType	::=	(Type_1, $\text{Type}_2, \ldots,$ Type_n);
RecordType	::=	(label_1 : $\text{Type}_1, \ldots,$ label_n: Type_n);

4 Implementation of Type System

4.1 Type System in the Process of Compiling

Type system is implemented as a part of language's compiler. It is in the semantic routine after syntactic analysis, see Fig.1.

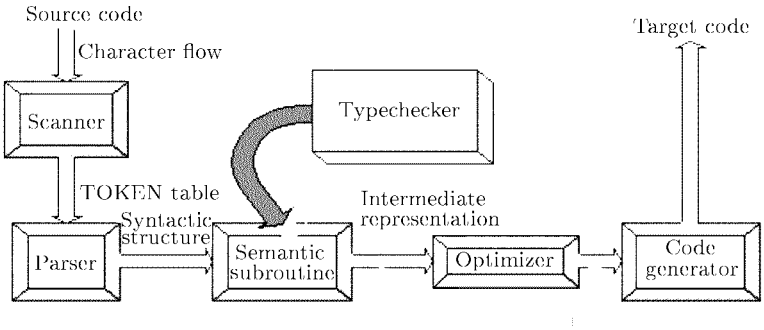

Fig.1. Where type system is located in the process of compiling.

The three stages of compiler: scanning, syntactic analysis and type analysis can be viewed as three composite functions that operate on programs: (typechecker ∘ parser ∘ scanner)

program, where '*program*' is the program that a programmer writes using a programming language. Scanner uses finite state automaton and maps the character string to a TOKEN table. By using some analysis algorithm, parser maps the TOKEN table to an abstract structure of the program (generally an abstract syntactic tree). Typechecker gives type semantics to the abstract syntactic structure. Typechecker is one semantic subroutine of compiler, but it is not necessary.

4.2 Type Inference and the Algorithm

The type inference algorithm we implement here is based on Hindley-Milner algorithm[4] in ML. Hindley-Milner algorithm is based on λ-calculus. Because SClean is a functional language based on graph rewriting, we have done some extension to the algorithm (see Fig.2). It mainly works as follows. (1) Traverse the program's abstract syntactic tree produced by the parser, assume a genetic type variable for each variable it meets. (2) Collect all the type constraints on the variables according to their usage in program. (3) Use Robinson's UNIFY algorithm[4] to compute the most general unifier (MGU) of all the assumed types of the same variable, and make sure that there is no conflicts among all the constraints. The MGU will be the final type of the variable.

find_type :: environment→program→type

I) find_type env ([],t) = σ

where, [basis, s, σ] = INFER env t

II) find_type env ([(n, body) : defs], t)

$$= \begin{cases} \text{find\_type(add\_to env } n(s_1\sigma_1))\ (\text{defs}, t), & \text{if occurs\_in } n \text{ body} \quad (2.1) \\ \text{find\_type(add\_to env } n\ \sigma_2)(\text{defs}, t), & \text{else} \quad (2.2) \end{cases}$$

$$\text{where, } \begin{cases} [bas_1, s_2, \sigma_1] = \text{INFER(add\_to env } n\,(\text{Typevar } \phi))\text{body} & (2.3.1) \\ s_1 = \text{UNIFY}(s_2(\text{Typevar } \phi), \sigma_1) & (2.3.2) \\ [bas_2, s_3, \sigma_2] = \text{INFER env body} & (2.3.3) \end{cases}$$

INFER :: env→term→typeresult

(a) INFER env (Var x) = [(x, Typevar α), {}, α]

(b) INFER env (DefrecName n) = [(), { }, env n]

(c) INFER env (Name n) = [(), { }, typeinstance]

where, typeinstance = fresh_instance(env n)

(d) INFER env (Appl f g) = [bas, s(Typevar α), $s \circ s_2 \circ s_1$]

$$\text{where, } \begin{cases} [basl, s_1, \rho] = \text{INFER env } f \\ [basr, s_2, \tau] = \text{INFER } s_1(\text{env})g \\ \{unifiable,\ s\} = \text{UNIFY}(\rho, \tau \to \alpha), \alpha \notin FV(A) \\ bas = s(basl{+}basr) \end{cases}$$

(e) INFER env (Abstr x g) =[result_bas, $s_2(\varphi \to \tau)$, $s_2 \circ s_1$]

$$\text{where, } \begin{cases} [bas, s_1, \tau] = \text{INFER env } g \\ \{\text{result\_bas}, s_2\} = \text{check\_assmptn}(x, \text{Typevar } \alpha)bas \end{cases}$$

Fig.2. The type inference SClean's type system uses.

5 Implementation of SClean

SClean's TAS[17] is made up of a few axioms, a set of type inference rules and a basis constructed dynamically during type inference. They together specify how to apply the above type inference algorithm to reconstruct type information for each term. This formal method has been found of great usefulness for detecting errors in language's prototype and extending the language. All the type inference rules are grammar-oriented and obey a principle: for each production of all the grammatical non-terminators, there is a corresponding inference rule, that is:

• For each grammatical non-terminator, there is a judgement;
• For each grammatical production of all non-terminators, there is an inference rule.

In our implementation of SClean's type system on the platform of Visual C++, the inference algorithm is realized by a set of recursively descending functions. With each rule, there is a function corresponding to it. When doing type inference on program's abstract syntactic tree built by parser, the typechecker applies corresponding inference rule to each grammatical symbol top-down recursively to complete the type inference progress.

SClean's inference environment is made up of four sub-environments: function symbol environment (FSE), node identification environment (NIE), type constructor environment (TCE), type variable environment (TVE), and the total environment is E.

SClean's type system has two categories of axioms:

$$\text{AXIOM}: \ \text{VAR}\ x:\alpha \mapsto x:\alpha \quad \text{CONST}\ \mapsto \text{BasicValue}:\text{BaseType}$$

VAR means an arbitrary variable without type declaration can have any fresh type variable as its type. CONST means that the type of BasicValue of some certain BaseType is BaseType.

$$\boxed{E_\phi \overset{\text{Program}}{\longmapsto} \text{Module}_1 \ldots \text{Module}_n;: E'}$$

$$\text{PROGRAM}\ \frac{\begin{array}{l} E_\phi \overset{\text{Module}}{\longmapsto} \text{Module}_1 :: E'_1 \\ E'_1 \overset{\text{Module}}{\longmapsto} \text{Module}_2 :: E'_2 \\ \ldots \\ E'_{n-1} \overset{\text{Module}}{\longmapsto} \text{Module}_n :: E' \end{array}}{E_\phi \overset{\text{program}}{\longmapsto} \text{Module}_1 \ldots \text{Module}_n;: E'}$$

Fig.3. Type inference rules for Program.

$$\boxed{E_\phi \overset{\text{Module}}{\longmapsto} \text{DEFINITION MODULE ModuleId; \{Import\}\{Def\} :: } E'}$$

$$\boxed{E_\phi \overset{\text{Module}}{\longmapsto} \text{[IMPLEMENTATION] MODULE ModuleId; \{Import\}\{Impl\} :: } E'}$$

$$\text{DEF-MODULE}\ \frac{\begin{array}{l} E_\phi \overset{\text{Import}}{\longmapsto} \text{Import}_1 :: E'_{I_1}, E'_{I_1} \overset{\text{Import}}{\longmapsto} \text{Import}_2 :: E'_{I_2}, \\ \qquad \ldots, E'_{I_{m-1}} \overset{\text{Import}}{\longmapsto} \text{Import}_m :: E'_{I_m} (m \geq 0) \\ E'_{I_m} \overset{\text{Def}}{\longmapsto} \text{Def}_1 :: E'_{D_1}, E'_{D_1} \overset{\text{Def}}{\longmapsto} \text{Def}_2 :: E'_{D_1}, \\ \qquad \ldots, E'_{D_{n-1}} \overset{\text{Def}}{\longmapsto} \text{Def}_n :: E' (n \geq 0) \end{array}}{\begin{array}{l} E_\phi \overset{\text{Module}}{\longmapsto} \text{DEFINITION MODULE ModuleId;} \\ \qquad \text{\{Import\}\{Def\}} :: E' \end{array}}$$

$$\text{IMPL-MODULE}\ \frac{\begin{array}{l} E_\phi \overset{\text{Import}}{\longmapsto} \text{Import}_1 :: E'_{I_1}, E'_{I_1} \overset{\text{Import}}{\longmapsto} \text{Import}_2 :: E'_{I_2}, \\ \qquad \ldots, E'_{I_{m-1}} \overset{\text{Import}}{\longmapsto} \text{Import}_m :: E'_{I_m} (m \geq 0) \\ E'_{I_m} \overset{\text{Impl}}{\longmapsto} \text{Impl}_1 :: E'_{P_1}, E'_{P_1} \overset{\text{Impl}}{\longmapsto} \text{Impl}_2 :: E'_{P_2}, \\ \qquad \ldots, E'_{P_{n-1}} \overset{\text{Impl}}{\longmapsto} \text{Impl}_n :: E' (n \geq 0) \end{array}}{\begin{array}{l} E_\phi \overset{\text{Module}}{\longmapsto} \text{DEFINITION MODULE ModuleId;} \\ \qquad \text{\{Import\}\{Impl\}} :: E' \end{array}}$$

Fig.4. Type inference rules for Module.

Figs.3 and 4 are the type inference rules for Program and Module. Generally, judgement takes the form of $E \overset{\text{RhsGraph}}{\longmapsto} \text{RhsGraph} :: \tau \triangleright E'$, RhsGraph above $\mapsto$ means that this is the judgement of RhsGraph. For the production of RhsGraph, it can be referred to from envi-

ronment E that the result is $\tau \triangleright E'$, with which RhsGraph has type τ, and the environment is modified to E'. Sometimes there may be no type result but just a modified environment. The judgements in Figs.3 and 4 are the related inference rules, consisting of one or more premises (above the horizontal line) and a conclusion (beneath the horizontal line). Premises infer the types of all the components according to related inference rules and get a modified environment. Conclusion is the final result.

6 Conclusions

In the field of computer science, type system has been studied very deeply and widely and involves a lot of theoretical fields, such as logic, algebra, set theory and category. With the fast development of type systems of object-oriented languages and parallel programming languages, type parameters are used and quantified to extend the type system to higher-order systems. Higher-order type system can represent polymorphism and abstract data type that support modularization and software reuse. Now the formalization of type system has evolved into type theory, which is an important branch of logic. Investigation of type system will be of great significance in the formation of a new style of programming methodology and the popularity of a host of computer science fields.

References

[1] Barendregt H P. The Lambda Calculus: Its Syntax and Semantics. Amsterdam: North-Holland, 1984.
[2] Barendregt H P. Lambda Calculus with Types. In *Handbook of Logic in Computer Science,* Oxford University Press, Vol.2, 1991.
[3] Hindley J R, Seldin J P. Introduction to Combinators and λ-Calculus. Cambridge University Press, 1986.
[4] Milner R, Tofte M, Harper R *et al.* The Definition of Standard ML (revised). The MIT Press, 1997.
[5] Cardelli L, Wegner P. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys,* Dec. 1985, 17(4): 471–521.
[6] Donahue J, Demers A. Data types are values. *ACM Transaction on Program. Languages and System,* July 1985, 7(3): 426–445.
[7] Cardelli L. Type System. In *Handbook of Computer Science and Engineering,* Chapter 103, CRC Press, 1997.
[8] Gunter C A. Semantics of Programming Languages: Structures and Techniques. In *Foundations of Computing,* Garey M, Meyer A (eds.), MIT Press, 1992.
[9] Cardelli L. Typeful Programming. In *Formal Description of Programming Concepts,* Neuhold E J, Paul M (eds.), Springer-Verlag, 1991.
[10] Hindley J R. Basic Simple Type Theory. Cambridge University Press, 1997.
[11] Hofmann M, Pierce B. A unifying type-theoretic framework for objects. *Journal of Functional Programming,* Cambridge University Press, Jan., 1993, 1(1).
[12] Fisher K, Mitchell J C. Notes on typed object-oriented programming. In *Proc. Theoretical Aspects of Computer Software,* Springer, LNCS 789, 1994, pp.844–885.
[13] Benjamin C Pierce, Turner D N. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming,* Cambridge University Press, Jan. 1993, 1(1): 1–100.
[14] Hudak P, Wadler P *et al.* Report on the functional programming language Haskell. Department of Computing Science, Glasgow University, 1993.
[15] Mitchell J C. Type Systems for programming languages. In *Handbook of Theoretical Computer Science,* Van Leeuwen J (ed.), North Holland, 1990, pp.365–458.
[16] Jiang Hui, Lin Dong, Xie Xiren. Using formal specification languages in industrial software development. In *IEEE International Conference on Intelligent Processing Systems,* Beijing, October 28–31, 1997, pp.1847–1850.
[17] Jiang Hui. Implementation of SClean's type system [dissertation]. Institute of Communication Engineering, 1998.