A Type-Theoretic Approach to Program Development

Li Wei
Beijing University of Aeronautics and Astronautics
Received July 9, 1988; revised April 5, 1989.

Abstract

A paradigm of program development using type theories is given after analyzing some typical examples. In order to carry this approach forward, a language ALT is designed. It is a linguistic description of a generalized higher order typed lambda calculus with $\Pi$, $\Sigma$ types and $\Pi$, $\Sigma$ kinds (supertypes). Four examples are given to show how ALT can be used to implement many concepts of software engineering and artificial intelligence. They are intuitionistic logic, Peano arithmetic, approximate reasoning and program transformations. ALT is described formally, using a structural operational approach.

1. Introduction

In his Turing Award lecture of 1977, John Bacus pointed out that von Neumann architecture is just one choice for implementing computation among many computational models, and it has played a great important role in the development of computer industry. Furthermore, he mentioned that thirty years practice of computer industry and research has proved that the von Neumann architecture has two fundamental defects: the von Neumann bottleneck, that is too many data transmissions between ALU and memory, heavily affects the efficiency of computers; and the mathematical model based on the von Neumann architecture is not the one which people study and use everyday in schools, universities, industries and other fields. To overcome these defects Bacus suggested to use applicative (or functional) languages based on the theory of Lambda calculus.

Since then, the research about applicative languages becomes a very active and fruitful area of computer science. Many languages, such as FP, ML, Hope and Miranda etc. have been designed and implemented. The applicative language-oriented architectures and computers have been designed and produced. The TI's Explorers, Flagship of ESPRIT and Connection machine are the typical representatives among them.

Thus how to build program development environments using applicative style languages, what is the relation between these languages and various logical inference systems, and how to implement these systems become the problems which are badly in need of solving.

In 1980 Marti-Löf generalized a typed Lambda calculus and provided a mechanism for implementing intuitionistic first order logic. In 1985 Huet and Coquand proposed their calculus of constructions. In 1987, Plotkin and his colleagues defined a second order-typed Lambda calculus called EFL (Edinburgh Logical Framework), and showed that many logical inference systems, including the classical first and second order logic, can be implemented using ELF.
On the one hand, various type structures have been enriched rapidly, so that using them we could specify and implement many high-level concepts in software, such as assertions, problems, rules, transformations, strategies and even "program development process", which are more difficult to deal with using traditional approaches. Many applications have been investigated along this direction, for example, using Martin-Löf's theory Constable built a prototype of software environment for implementing mathematics called Nuprl\textsuperscript{17}, Paulson designed his Isabelle\textsuperscript{13}, and Edinburgh built their proof editors using ELF\textsuperscript{11}. All of these efforts herald a new approach in constructing computer-aided tools and program development environments in both software engineering and artificial intelligence.

On the other hand, if we study these works carefully, we will find that their type theory is introduced only to serve some special purposes. For example, the type theory given by Martin-Löf is introduced to define intuitionistic (first order) logic under the principle of "propositions as types"; the Π type and Σ type are introduced to specify universal and existential qualifiers, but there are no "kinds" (supertypes) in his theory (there is a hierarchy of universes, such as in Nuprl, but this is not general enough to make it useful like ELF). The Edinburgh LF is introduced to provide a general framework for expressing various logic systems and for constructing interactive proof development environments of them under the principle of "high order judgments as types"; a supertype "kind" is introduced to define high order judgments, but there is no Σ type, and Σ kinds.

What we need now is to abstract the paradigm of this new approach from those works mentioned above, and to design a new kind of language or tool incorporating the advances within type theory, and to show how those widespread concepts of software engineering and artificial intelligence can be implemented using such a language.

Thus the paradigm of this approach is given in Section 2, after analyzing some typical examples concerning type theories. As a contribution to this approach the language ALT is designed, and its abstract syntax is given in Section 3. Some programming examples in ALT are discussed in Section 4 to show the role of this language in program environment design. The language is described using structural operational semantics, and a static semantics, dynamic semantics and the interaction between the two are described in Sections 5, 6 and 7.

2. A Paradigm of Program Construction Using Type Theories

In this section we will give a general paradigm of program development using type theories after analyzing two major examples.

Example 1. Martin-Löf Type Theory and Intuitionistic Logic

In his lecture note, Martin-Löf introduced a type theory to implement intuitionistic (first order) logic. His approach can be summarized by the following steps:

1. Viewing propositions as types.

When we state a proposition, for example, "John is a thief", we have actually made a judgment; therefore a proposition can be identified as a judgment. When we say "a proposition is true" we mean that the judgment is formed upon