A Reusable Modular Extension Mechanism for a Practical Compiler

Yutaka Saeki

2001,1,12

Abstract

Since the application areas of information technologies is increasing widely, and each application is being more complex, use of languages designed for particular domains (called domain-specific languages) is regarded as an important paradigm for rapid development and low-cost maintenance. As a methodology for providing such languages, using extensible programming languages are suitable. Because language designers concern about only minimum faculties, there is no necessity to take care about the others. And for users of DSLs,the extended language is based on a fixed language, so programmers need little effort for leaning a new environment.

In this thesis, we propose a technique for implementing a compiler that is build as composition of compiler modules. There are some difficulties to build compilers enable to be extended by composable modules, so we propose a method to solve them. In this work, we try to satisfy the following requirements.

1. The designer of compiler modules should be able to program/understand each module easily.
   It is generally difficult for users to understand all the mechanism of language construct. We need a good abstraction for defining a new faculty of a compiler.

2. Each compiler module should be composable.
   Practically, the programmer might want to apply several extensions to the language at the same time. To help this, we must introduce a framework to detect and avoid conflicts among extension modules.

We adopted a metalevel architecture for describing the modules. The metalevel architecture is a good abstraction mechanism for extensible systems . A software based on metalevel architecture is separated into a metalevel and a baselevel. The semantics of a baselevel is defined in a metalevel.

In our system, the definition of each module consists of two parts: action part and meta part.

The action part describes a compilation subprocess (e.g., how a sequence of compiled expressions should be composed), while the meta part defines the low level behaviors of each expression in the action part and describes access constraints on compiler states. An action part is a baselevel definition of a compiler module, and meta part is a metalevel of the action part.

Using this approach, we can define every compiler module (including low level code generators and runtime administrators) in a quite high-level manner.

Moreover, using explicit constraints on the access to compiler states, the composed compiler can guarantee a kind of safety by detecting "conflicts" among modules. Along this way, we designed and implemented an extensible compiler of a dynamically-typed functional language.

Key Words: language extension,compiler,conflict avoidance,metalevel architecture