U. Kastens P. Pfahler (Eds.)

Compiler Construction

4th International Conference, CC '92
Paderborn, FRG, October 5-7, 1992
Proceedings

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Volume Editors

Uwe Kastens
Peter Pfahler
Universität-GH-Paderborn, Mathematik-Informatik (FB 17)
Warburger Str. 100, W-4790 Paderborn

CR Subject Classification (1991): D.3.4, D.3.1, F.4.2, D.2.6, I.2.2

ISBN 3-540-55984-1 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-55984-1 Springer-Verlag New York Berlin Heidelberg

Typesetting: Camera ready by author/editor
Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
45/3140-543210 - Printed on acid-free paper

Preface

The *International Workshop on Compiler Construction CC '92* provides a forum for the presentation and discussion of recent developments in the area of compiler construction. Its scope ranges from compilation methods and tools to implementation techniques for specific requirements of languages and target architectures.

The workshop is held every two years and continues the series of *Compiler Compiler* workshops organized in the former GDR since 1986. Its new title *Compiler Construction* documents the extension of its scope. It is intended to establish this series as a forum in Europe that covers the whole range of compilation aspects. The good response to the call for papers shows the need for such a forum: 64 papers on a broad variety of topics were submitted by authors from all over the world. The program committee selected 16 contributions for full presentation and 12 for short presentation, both published in this volume. Further contributions were selected to be presented at a poster exhibition. Their titles are listed at the end of this volume – abstracts are published in a report of the Computer Science Series of the Universität-GH Paderborn. The workshop program is completed by a keynote speech given by N. Wirth on "30 Years of Programming Languages and Compilers".

CC '92 is hosted by the Universität-GH Paderborn. The German Gesellschaft für Informatik (GI) is the main sponsor of CC '92. IFIP working Group 2.4 (System Implementation Languages) and GI Fachgruppe 2.1.3 (Implementierung von Programmiersprachen) also support CC '92.

We thank all who contributed to the workshop and its organization.

Paderborn, June 1992

Uwe Kastens
Peter Pfahler

Program Committee

P. Fritzson (Sweden)
Tibor Gyimothi (Hungary)
R. Nigel Horspool (Canada)
Martin Jourdan (France)
Uwe Kastens (Germany)
Kai Koskimies (Finland)
Günter Riedewald (Germany)
A. Vladimir Serebriakov (Russia)
Reinhard Wilhelm (Germany)

Chairman Uwe Kastens, Paderborn

Organization Peter Pfahler, Paderborn

Contents

Transformation of Attributed Trees Using Pattern Matching

Josef Grosch

GMD Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1, D-7500 Karlsruhe, Germany
+721-662226
grosch@karlsruhe.gmd.de

Abstract. This paper describes a tool for the transformation of attributed trees using pattern matching. The trees to be processed are defined by a formalism based on context-free grammars. Operations for trees such as composition and decomposition are provided. The approach can be characterized as an amalgamation of trees or terms including pattern matching, with recursion, attribute grammars, and imperative programming. Transformations can either modify the input trees or map them to arbitrary output. Possible applications are the various transformation tasks in compilers such as semantic analysis, optimization, or the generation of intermediate representations. The design goals have been to combine an expressive and high level technique for transformation with flexibility, efficiency, and practical usability. A reliable development style is supported by static typing and checks for the single assignment property of variables. We give some example transformations and describe the input language of our tool called *puma*. The relationship to similar work is discussed. Finally, experimental results are presented that demonstrate the efficiency of our approach.

Keywords. transformation, attributed trees, pattern matching

1 Introduction

The transformation of trees using pattern matching becomes an accepted technique. Several tools have been constructed recently that follow this principle [CoP90, HeS91, LMW89, Vol91]. Tools for code generation successfully use the same technique, too [AGT89, ESL89]. We present a new tool called *puma* and its input language for the transformation and manipulation of attributed trees and graphs [Gro91a]. *Puma* stands for *p*attern *m*atching and *u*nification. Its intended application areas are the various transformation tasks in a compiler operating on abstract syntax trees or arbitrary graph structures. This includes semantic analysis, optimization, intermediate code generation, source-to-source translation, and eventually machine code generation.

The trees that are subject to pattern matching are described by a formalism based on context-free grammars. The tree nodes may be associated with attributes of arbitrary types. Node types are used to specify the properties of tree nodes. An extension mechanism induces a subtype relation among the node types. Pattern matching is extended to handle subtypes and attributes, too. Operations for the composition and decomposition of trees are supported by a concise notation.

The building blocks for a transformation are recursive subroutines, classified as procedures, functions, and predicates, with an arbitrary number of input and output parameters. The bodies of the subroutines consist of rules which are made up of patterns, conditions, statements, and expressions. The first two components control the applicability of a rule. The statements determine what has to be done whenever a rule is applicable.

The expressions provide values for the output parameters and the function result.

Static type checking with respect to trees is provided. Variables are declared implicitly and they are checked for the single assignment property. There is read and write access to attributes stored in the tree which allows the construction of attribute evaluators. In all places it is possible to escape to hand-written code which provides the power and flexibility of the imperative programming style.

The output of the generator is a source module in one of the target languages C or Modula-2. This module allows for easy integration and cooperation with other modules, either hand-written or generated ones. The pattern matching is local and considers a region at the top of the current subtree, only. It is implemented by direct code and therefore efficient.

Our approach can be regarded from several points of view: From the point of view of imperative programming it is an extension by statically typed, attributed trees, constructs for composition and decomposition, and pattern matching. From the point of view of logic programming it omits backtracking and restricts pattern matching to one of the two terms being a ground term. It adds attributes which are stored in the terms (trees), static typing, input and output modes for parameters, and an easy escape to imperative features. From the point of view of functional programming it offers the simple style of functional programming which has always been present in imperative languages having functions and recursion. It adds the pattern matching facility. From the point of view of attribute grammars it allows the specification of attribute evaluation with explicit control of the evaluation order or visit sequences. This eases the use of global attributes and gives full control on side-effects.

The intended use of this tool proceeds in three steps: First, a tree is constructed either by a parser, a previous transformation phase, or whatever is appropriate. Second, the attributes in the tree are evaluated either using an attribute grammar based tool, by a *puma* specified tree traversal and attribute computations, or by hand-written code. Third, the attributed tree is transformed or mapped to another data structure by a *puma* generated transformation module. These steps can be executed one after the other or more or less simultaneously. Besides trees, *puma* can handle attributed graphs as well, even cyclic ones. Of course the cycles have to be detected in order to avoid infinite loops. A possible solution uses attributes as marks for nodes already visited.

A transformer module can make use of attributes in the following ways: If attribute values have been computed by a preceding attribute evaluator and are accessed in read only mode then this corresponds to the three step model explained above. A *puma* generated module can also evaluate attributes on its own. A further possibility is that an attribute evaluator can call *puma* subroutines in order to compute attributes. This is especially of interest when attributes depend on tree-valued arguments.

The tool supports two classes of tree transformations: *mappings* and *modifications*. Tree mappings map an input tree to arbitrary output data. The input tree is accessed in read only mode and left unchanged. Tree *modifications* change a tree by e. g. computing and storing attributes at tree nodes or by changing the tree structure. In this case the tree data structure serves as input as well as output and it is accessed in read and write mode.

The first class covers applications like the generation of intermediate languages or machine code. Trees are mapped to arbitrary output like source code, assembly code, binary machine code, linearized intermediate languages like P-Code, or another tree structure. A further variant of mapping is to emit a sequence of procedure calls which are

handled by an abstract data type.

The second class covers applications like semantic analysis or optimization. Trees are decorated with attribute values, properties of the trees corresponding to context conditions are checked, or trees are changed in order to reflect optimizing transformations.

Puma is part of the Karlsruhe Toolbox for Compiler Construction [GrE90]. In particular it cooperates with the generator for abstract syntax trees *ast* [Gro91b] and the attribute evaluator generator *ag* [Gro89]. The attributed trees are defined and managed by a module generated with *ast*. A second module generated by *puma* creates and handles these trees. This way all the powerful operations for trees and graphs provided by *ast* are available such as reader and writer procedures or the interactive browser. For sake of simplicity we will deviate from reality in this paper and treat the definition of the tree structure as part of *puma*.

The rest of this paper is organized as follows: Section 2 presents a few simple examples of how to describe transformations with *puma*. Section 3 describes the input language of the tool. Section 4 sketches the implementation of the generated transformer module. Section 5 compares our approach with related work. Section 6 presents experimental results. Section 7 contains concluding remarks.

2 Tree Transformation by Pattern Matching

The probably easiest way to get an impression of our approach can be obtained by having a look at a few introductory examples. We will use the abstract syntax of simple arithmetic expressions as input data structure. Besides a few intrinsic attributes describing e. g. the values of constants we use an attribute called *Type*. It describes the type of every subexpression. Its domain are trees, too. The tree definition based on a context-free grammar shown in Example 1 specifies the structure of expressions and types.

Example 1: Tree Definition

```
Expr            = Type <
   Plus         = Lop: Expr Rop: Expr .
   Minus        = Lop: Expr Rop: Expr .
   Const        = [Value] .
   Adr          = <
      Index     = Adr Expr .
      Select    = Adr [Ident: tIdent] .
      Ident     = [Ident: tIdent] .
   >.
>.
Type            = <
   Int          = .
   Real         = .
   Bool         = .
   Array        = [Lwb] [Upb] Type .
   Record       = Fields .
>.
Fields          = <
   NoField      = .
   Field        = [Ident: tIdent] Type Fields .
>.
```

The names before the character '=' can be regarded both as rule names or nonterminals. The possible right-hand sides for one nonterminal are enclosed in angle brackets '<' and '>'. Non-tree valued attributes are enclosed in square brackets '[' and ']'. The attributes *Lwb*, *Upb*, and *Value* are of the default type *int*. The attribute *Ident* is of the user-