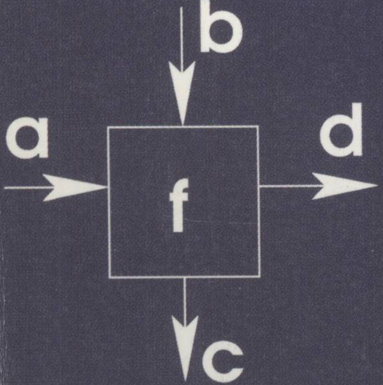
LNCS 3622

Tutorial

Varmo Vene
Tarmo Uustalu (Eds.)

Advanced Functional Programming

5th International School, AFP 2004
Tartu, Estonia, August 2004
Revised Lectures



Springer

TP311.1-53
A257
2004

Varmo Vene   Tarmo Uustalu (Eds.)

Advanced Functional Programming

5th International School, AFP 2004
Tartu, Estonia, August 14 – 21, 2004
Revised Lectures

Springer

Volume Editors

Varmo Vene
University of Tartu
Department of Computer Science
J. Liivi 2, EE-50409 Tartu, Estonia
E-mail: varmo@cs.ut.ee

Tarmo Uustalu
Institute of Cybernetics
Akadeemia tee 21, EE-12618 Tallinn, Estonia
E-mail: tarmo@cs.ioc.ee

Library of Congress Control Number: 2005931987

CR Subject Classification (1998): D.1.1, D.1, D.3, F.3, D.2

ISSN	0302-9743
ISBN-10	3-540-28540-7 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-28540-3 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper   SPIN: 11546382   06/3142   5 4 3 2 1 0

Preface

This volume contains the revised lecture notes corresponding to nine of the lecture courses presented at the 5th International School on Advanced Functional Programming, AFP 2004, held in Tartu, Estonia, August 14–21, 2004.

The goal of the AFP schools is to inform the wide international communities of computer science students and software production professionals about the new and important developments in the area of functional programming. The schools put a special emphasis on practical applications of advanced techniques. The Tartu school was preceded by four earlier schools in Båstad, Sweden (1995, LNCS 925), Olympia, WA, USA (1996, LNCS 1129), Braga, Portugal (1998, LNCS 1608) and Oxford, UK (2002, LNCS 2638).

The scientific programme of AFP 2004 consisted of five preparatory ("intermediate") courses, given by John Hughes (Chalmers University of Technology, Göteborg, Sweden), Doaitse Swierstra (Universiteit Utrecht, The Netherlands) and Rinus Plasmeijer (Radboud Universiteit Nijmegen, The Netherlands), and nine regular ("advanced") courses, presented by Atze Dijkstra (Universiteit Utrecht, The Netherlands), Doaitse Swierstra, John Hughes, Conor McBride (University of Nottingham, UK), Alberto Pardo (Universidade de la República, Montevideo, Uruguay), Rinus Plasmeijer, Bernard Pope (University of Melbourne, Australia), Peter Thiemann (Universität Freiburg, Germany), and Simon Thompson (University of Kent, UK). There was also a student session.

The school attracted a record number of 68 participants from 16 countries (inclusive of the lecturers and organizers).

This volume contains the notes for the advanced courses. Following the school, the lecturers revised the notes they had prepared for the school. The revised notes were each carefully checked by two or three second readers selected from among the most qualified available and then revised once more by the lecturers. We are proud to commend the final texts to everyone wishing to acquire first-hand knowledge about some of the exciting and trendsetting developments in functional programming.

We are grateful to our sponsors, to the Faculty of Mathematics and Computer Science of the University of Tartu, to the lecturers and the second readers for their hard work on the oral presentations, and the notes, and to all our participants. You made the school what it was.

Tartu and Tallinn, June 2005

Varmo Vene
Tarmo Uustalu

Organization

Host Institution

AFP 2004 was organized by the Department of Computer Science of the University of Tartu in cooperation with the Center for Dependable Computing (CDC), an Estonian center of excellence in research.

Programme Committee

Varmo Vene (University of Tartu, Estonia) (chairman)
Johan Jeuring (Universiteit Utrecht, The Netherlands)
Tarmo Uustalu (Institute of Cybernetics, Tallinn, Estonia)

Organizing Committee

Varmo Vene (University of Tartu, Estonia) (chairman)
Härmel Nestra (University of Tartu, Estonia)
Vesal Vojdani (University of Tartu, Estonia)
Tarmo Uustalu (Institute of Cybernetics, Tallinn, Estonia)

Second Readers

Venanzio Capretta (University of Ottawa, Canada)
James Cheney (University of Edinburgh, UK)
Catarina Coquand (Chalmers University of Technology, Sweden)
Jeremy Gibbons (University of Oxford, UK)
Thomas Hallgren (Oregon Graduate Institute, Portland, OR, USA)
Michael Hanus (Christian-Albrechts-Universität zu Kiel, Germany)
Johan Jeuring (Universiteit Utrecht, The Netherlands)
Jerzy Karczmarczuk (Université Caen, France)
Ralf Lämmel (CWI, Amsterdam, The Netherlands)
Andres Löh (Universiteit Utrecht, The Netherlands)
Nicolas Magaud (University of New South Wales, Sydney, Australia)
Simon Marlow (Microsoft Research, Cambridge, UK)
Ross Paterson (City University, London, UK)
Simon Peyton Jones (Microsoft Research, Cambridge, UK)
Colin Runciman (University of York, UK)
Tim Sheard (Portland State University, Portland, OR, USA)
Joost Visser (Universidade do Minho, Braga, Portugal)
Eric Van Wyk (University of Minnesota, Minneapolis, MN, USA)

Sponsoring Institutions

Tiigriülikool programme of the Estonian Information Technology Foundation

National Centers of Excellence programme of the Estonian Ministry of Education and Research

EU FP5 IST programme via the thematic network project APPSEM II

¥462.56元

Table of Contents

Typing Haskell with an Attribute Grammar

Atze Dijkstra and S. Doaitse Swierstra

Institute of Information and Computing Sciences,
Utrecht University, P.O.Box 80.089,
3508 TB Utrecht, Netherlands
{atze, doaitse}@cs.uu.nl

Abstract. A great deal has been written about type systems. Much less has been written about implementing them. Even less has been written about implementations of complete compilers in which all aspects come together. This paper fills this gap by describing the implementation of a series of compilers for a simplified variant of Haskell. By using an attribute grammar system, aspects of a compiler implementation can be described separately and added in a sequence of steps, thereby giving a series of increasingly complex (working) compilers. Also, the source text of both this paper and the executable compilers come from the same source files by an underlying minimal weaving system. Therefore, source and explanation is kept consistent.

1 Introduction and Overview

Haskell98 [31] is a complex language, not to mention its more experimental incarnations. Though also intended as a research platform, realistic compilers for Haskell [1] have grown over the years and understanding and experimenting with those compilers is not an easy task. Experimentation on a smaller scale usually is based upon relatively simple and restricted implementations [20], often focusing only on a particular aspect of the language and/or its implementation. This paper aims at walking somewhere between this complexity and simplicity by

- Describing the implementation of essential aspects of Haskell (or any other (functional) programming language), hence the name Essential Haskell (EH) used for simplified variants of Haskell[1] in these notes.
- Describing these aspects separately in order to provide a better understanding.
- Adding these aspects on top of each other in an incremental way, thus leading to a sequence of compilers, each for a larger subset of complete Haskell (and extensions).
- Using tools like the Utrecht University Attribute Grammar (UUAG) system [3], hereafter referred to as the AG system, to allow for separate descriptions for the various aspects.

[1] The 'E' in EH might also be expanded to other aspects of the compiler, like being an Example.

The remaining sections of this introduction will expand on this by looking at the intentions, purpose and limitations of these notes in more detail. This is followed by a short description of the individual languages for which we develop compilers throughout these notes. The last part of the introduction contains a small tutorial on the AG system used in these notes. After the introduction we continue with discussing the implementation of the first three compilers (sections 2, 3 and 4) out of a (currently) sequence of ten compilers. On the web site [11] for this project the full distribution of the code for these compilers can be found. We conclude these notes by reflecting upon our experiences with the AG system and the creation of these notes (section 5).

1.1 Purpose

For whom is this material intended?

- For students who wish to learn more about the implementation of functional languages. This paper also informally explains the required theory, in particular about type systems.
- For researchers who want to build (e.g.) a prototype and to experiment with extensions to the type system and need a non-trivial and realistic starting point. This paper provides documentation, design rationales and an implementation for such a starting point.
- For those who wish to study a larger example of the tools used to build the compilers in these notes. We demonstrate the use of the AG system, which allows us to separately describe the various aspects of a language implementation. Other tools for maintaining consistency between different versions of the resulting compilers and the source code text included in these notes are also used, but will not be discussed.

For this intended audience these notesprovide:

- A description of the implementation of a type checker/inferencer for a subset of Haskell. We describe the first three languages of a (currently) sequence of ten, that end in a full implementation of an extended Haskell.
- A description of the semantics of Haskell, lying between the more formal [16,14] and more implementation oriented [21,33] and similar to other combinations of theory and practice [34].
- A gradual instead of a big bang explanation.
- Empirical support for the belief that the complexity of a compiler can be managed by splitting the implementation of the compiler into separate aspects.
- A working combination of otherwise usually separately proven or implemented features.

We will come back to this in our conclusion (see section 5).

We restrict ourselves in the following ways, partly because of space limitations, partly by design:

- We do not discuss extensions to Haskell implemented in versions beyond the last version presented in these notes. See section 1.3 for a preciser description of what can and cannot be found in these notes with respect to Haskell features.
- We concern ourselves with typing only. Other aspects, like pretty printing and parsing, are not discussed. However, the introduction to the AG system (see section 1.4) gives some examples of the pretty printing and the interaction between parsing, AG code and Haskell code.
- We do not deal with type theory or parsing theory as a subject on its own. This paper is intended to describe "how to implement" and will use theory from that point of view. Theoretical aspects are touched upon from a more intuitive point of view.

Although informally and concisely introduced where necessary, familiarity with the following will make reading and understanding these notes easier:

- Functional programming, in particular using Haskell
- Compiler construction in general
- Type systems, λ-calculus
- Parser combinator library and AG system [3,38]

For those not familiar with the AG system a short tutorial has been included at the end of this introduction (see section 1.4). It also demonstrates the use of the parser combinators used throughout the implementation of all EH versions.

We expect that by finding a balance between theory and implementation, we serve both those who want to learn and those who want to do research. It is also our belief that by splitting the big problem into smaller aspects the combination can be explained in an easier way.

In the following sections we give examples of the Haskell features present in the series of compilers described in the following chapters. Only short examples are given, so the reader gets an impression of what is explained in more detail and implemented in the relevant versions of the compiler.

1.2 A Short Tour

Though all compilers described in these notes deal with a different issue, they all have in common that they are based on the λ-calculus, most of the time using the syntax and semantics of Haskell. The first version of our series of compilers therefore accepts a language that most closely resembles the λ-calculus, in particular typed λ-calculus extended with **let** expressions and some basic types and type constructors such as *Int*, *Char* and tuples.

EH version 1: λ-calculus. An EH program is a single expression, contrary to a Haskell program which consists of a set of declarations forming a module.

```
let i :: Int
    i = 5
in i
```

All variables need to be typed explicitly; absence of an explicit type is considered to be an error. The corresponding compiler (EH version 1, section 2) checks the explicit types against actual types.

Besides the basic types *Int* and *Char*, composite types can be formed by building tuples and defining functions:

```
let id :: Int → Int
    id = λx → x
    fst :: (Int, Char) → Int
    fst = λ(a, b) → a
in id (fst (id 3, 'x'))
```

Functions accept one parameter only, which can be a pattern. All types are monomorphic.

EH version 2: Explicit/implicit typing. The next version (EH version 2, section 3) no longer requires the explicit type specifications, which thus may have to be inferred by the compiler.

The reconstructed type information is monomorphic, for example the identity function in:

```
let id = λx → x
in let v = id 3
   in id
```

is inferred to have the type *id* :: *Int* → *Int*.

EH version 3: Polymorphism. The third version (EH version 3, section 4) performs standard Hindley-Milner type inferencing [8,9] which also supports parametric polymorphism. For example,

```
let id = λx → x
in id 3
```

is inferred to have type $id :: \forall a.a \to a$.

1.3 Haskell Language Elements Not Described

As mentioned before, only a subset of the full sequence of compilers is described in these notes. Currently, as part of an ongoing work [11], in the compilers following the compilers described in these notes, the following Haskell features are dealt with:

EH 4. Quantifiers everywhere: higher ranked types [36,32,7,28] and existentials [30,25,27]. See also the longer version of these notes handed out during the AFP04 summerschool [13].
EH 5. Data types.
EH 6. Kinds, kind inference, kind checking, kind polymorphism.
EH 7. Non extensible records, subsuming tuples.
EH 8. Code generation for a GRIN (Graph Reduction Intermediate Notation) like backend [6,5].
EH 9. Class system, explicit implicit parameters [12].
EH 10. Extensible records [15,22].

Also missing are features which fall in the category syntactic sugar, programming in the large and the like. Haskell incorporates many features which make programming easier and/or manageable. Just to mention a few:

- Binding group analysis
- Syntax directives like infix declarations
- Modules [10,37].
- Type synonyms
- Syntactic sugar for **if**, **do**, list notation and comprehension.

We have deliberately not dealt with these issues. Though necessary and convenient we feel that these features should be added after all else has been dealt with, so as not to make understanding and implementating essential features more difficult.

1.4 An AG Mini Tutorial

The remaining part of the introduction contains a small tutorial on the AG system. The tutorial explains the basic features of the AG system. The explanation of remaining features is postponed to its first use throughout the main text. These places are marked with $\mathcal{AG}$. The tutorial can safely be skipped if the reader is already familiar with the AG system.

Haskell and Attribute Grammars (AG). Attribute grammars can be mapped onto functional programs [23,19,4]. Vice versa, the class of functional programs (catamorphisms

[39]) mapped onto can be described by attribute grammars. The AG system exploits this correspondence by providing a notation (attribute grammar) for computations over trees which additionally allows program fragments to be described separately. The AG compiler gathers these fragments, combines these fragments, and generates a corresponding Haskell program.

In this AG tutorial we start with a small example Haskell program (of the right form) to show how the computation described by this program can be expressed in the AG notation and how the resulting Haskell program generated by the AG compiler can be used. The 'repmin' problem [4] is used for this purpose. A second example describing a 'pocket calculator' (that is, expressions) focusses on more advanced features and typical AG usage patterns.

Repmin a la Haskell. Repmin stands for "replacing the integer valued leaves of a tree by the minimal integer value found in the leaves". The solution to this problem requires two passes over a tree structure, computing the miminum and computing a new tree with the minimum as its leaves respectively. It is often used as the typical example of a circular program which lends itself well to be described by the AG notation. When described in Haskell it is expressed as a computation over a tree structure:

```
data Tree = Tree_Leaf Int
          | Tree_Bin  Tree Tree
          deriving Show
```

The computation itself simultaneously computes the minimum of all integers found in the leaves of the tree and the new tree with this minimum value. The result is returned as a tuple computed by function *r*:

```
repmin :: Tree → Tree
repmin  t
  = t'
  where (t', tmin)           = r t tmin
        r (Tree_Leaf i    ) m = (Tree_Leaf m   , i              )
        r (Tree_Bin  lt rt) m = (Tree_Bin lt' rt', lmin `min` rmin)
                           where (lt', lmin) = r lt m
                                 (rt', rmin) = r rt m
```

We can use this function in some setting, for example:

```
tr     = Tree_Bin (Tree_Leaf 3) (Tree_Bin (Tree_Leaf 4) (Tree_Leaf 5))
tr'    = repmin tr

main :: IO ()
main = print tr'
```

The resulting program produces the following output:

```
Tree_Bin (Tree_Leaf 3) (Tree_Bin (Tree_Leaf 3) (Tree_Leaf 3))
```

The computation of the new tree requires the minimum. This minimum is passed as a parameter *m* to *r* at the root of the tree by extracting it from the result of *r*. The result tuple of the invocation *r t tmin* depends on itself via the minimum *tmin* so it would seem we have a cyclic definition. However, the real dependency is not on the tupled result of *r* but on its elements because it is the element *tmin* of the result tuple which is passed back and not the tuple itself. The elements are not cyclically dependent so Haskell's laziness prevents a too eager computation of the elements of the tuple which might otherwise have caused an infinite loop during execution. Note that we have two more or less independent computations that both follow the tree structure, and a weak interaction, when passing the *tmin* value back in the tree.

Repmin a la AG. The structure of *repmin* is similar to the structure required by a compiler. A compiler performs several computations over an *abstract syntax tree* (*AST*), for example for computing its type and code. This corresponds to the *Tree* structure used by *repmin* and the tupled results. In the context of attribute grammars the elements of this tuple are called *attribute*'s. Occasionaly the word *aspect* is used as well, but an aspect may also refer to a group of attributes associated with one particular feature of the AST, language or problem at hand.

Result elements are called *synthesized* attributes. On the other hand, a compiler may also require information that becomes available at higher nodes in an AST to be available at lower nodes in an AST. The *m* parameter passed to *r* in *repmin* is an example of this situation. In the context of attribute grammars this is called an *inherited attribute*.

Using AG notation we first define the AST corresponding to our problem (for which the complete compilable solution is given in Fig. 1):

```
DATA Tree
  | Leaf int : {Int}
  | Bin  lt  : Tree
         rt  : Tree
```

The **DATA** keyword is used to introduce the equivalent of Haskell's **data** type. A **DATA**⟨*node*⟩ defines a *node* ⟨*node*⟩ (or *nonterminal*) of an AST. Its alternatives, enumerated one by one after the vertical bar |, are called *variants*, *productions*. The term *constructor* is occasionally used to stress the similarity with its Haskell counterpart. Each variant has members, called *children* if they refer to other nodes of the AST and *fields* otherwise. Each child and field has a name (before the colon) and a type (after the colon). The type may be either another **DATA** node (if a child) or a monomorphic Haskell type (if a field), delimited by curly braces. The curly braces may be omitted if the Haskell type is a single identifier. For example, the **DATA** definition for the repmin problem introduces a node (nonterminal) *Tree*, with variants (productions) *Leaf* and *Bin*. A *Bin* has children *lt* and *rt* of type *Tree*. A *Leaf* has no children but contains only a field *int* holding a Haskell *Int* value.