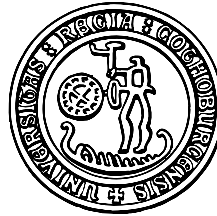
Thesis for the Degree of Licentiate of Engineering

Compiling Grammar-based Speech Application Components

Björn Bringert

CHALMERS | GÖTEBORG UNIVERSITY



Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden

Göteborg, July 2007

Compiling Grammar-based Speech Application Components
BJÖRN BRINGERT

© BJÖRN BRINGERT, 2007

Technical report no. 40L
ISSN 1652–876X
Department of Computer Science and Engineering
Language Technology Research Group

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31–772 1000

Printed at Chalmers, Göteborg, Sweden, 2007

Abstract

It is easy to imagine machines that can communicate using spoken natural language. Constructing such machines is more difficult. The available methods for development of interactive speech applications are costly, and current research is mainly focused on producing more sophisticated systems, rather than on making it easier to build them.

This thesis describes how components used in interactive speech applications can be automatically derived from natural language grammars written in Grammatical Framework (GF). By using techniques borrowed from the field of programming language implementation, we can generate speech recognition language models, multimodal fusion and fission components, and support code for abstract syntax transformations.

By automatically generating these components, we can reduce duplicated work, ensure consistency, make it easier to build multilingual systems, improve linguistic quality, enable re-use across system domains, and make systems more portable.

Table of Contents

Acknowledgments **vii**

Introduction **1**

1 Interactive Speech Applications . . . 1
  1.1 Problems . . . 3
2 This work . . . 4
  2.1 Advantages . . . 4
  2.2 Limitations . . . 7
3 Grammatical Framework . . . 7
  3.1 An Example . . . 8
4 Paper I: Speech Recognition Grammar Compilation in Grammatical Framework . . . 10
  4.1 An Example . . . 10
  4.2 Contribution . . . 10
  4.3 Publication . . . 12
5 Paper II: Multimodal Dialogue System Grammars . . . 12
  5.1 An Example . . . 13
  5.2 Contribution . . . 14
  5.3 Publication . . . 14
6 Paper III: A Pattern for Almost Compositional Functions . . . 14
  6.1 An Example . . . 15
  6.2 Contribution . . . 15
  6.3 Publication . . . 15
7 Related Work . . . 16
  7.1 GF in Interactive Speech Applications . . . 16
  7.2 Compiler-like Grammar Development . . . 16
  7.3 Interactive Development Environments for Dialogue Systems . . . 17
8 Future work . . . 17

Paper I: Speech Recognition Grammar Compilation in Grammatical Framework **25**

1 Introduction . . . 25
2 Speech Recognition Grammars . . . 26
3 Grammatical Framework . . . 27
  3.1 The Resource Grammar Library . . . 27
  3.2 An Example GF Grammar . . . 27

4 Generating Context-free Grammars . . . 28
4.1 Algorithm . . . 28
4.2 Discussion . . . 32
5 Finite-State Models . . . 32
5.1 Algorithm . . . 32
5.2 Discussion . . . 33
6 Semantic Interpretation . . . 33
6.1 Algorithm . . . 34
6.2 Discussion . . . 34
7 Related Work . . . 34
7.1 Unification Grammar Compilation . . . 34
7.2 Generating SLMs from GF Grammars . . . 35
8 Results . . . 35
9 Conclusions . . . 36

Paper II: Multimodal Dialogue System Grammars **43**
1 Introduction . . . 43
2 The Grammatical Framework and multilingual grammars . . . 44
3 Extending multilinguality to multimodality . . . 47
4 Proof-of-concept implementation . . . 47
4.1 Transport network . . . 48
4.2 Multimodal input . . . 48
4.3 Multimodal output . . . 51
5 Related Work . . . 53
6 Conclusion . . . 53

Paper III: A Pattern for Almost Compositional Functions **59**
1 Introduction . . . 59
1.1 Some motivating problems . . . 59
1.2 The solution . . . 60
1.3 Article overview . . . 60
2 Abstract Syntax and Algebraic Data Types . . . 61
3 Compositional Functions . . . 61
3.1 Monadic compositional functions . . . 62
3.2 Generalizing *composOp*, *composM* and *composFold* . . . 63
4 Systems of Data Types . . . 65
4.1 Several algebraic data types . . . 65
4.2 Categories and trees . . . 65
4.3 Compositional operations . . . 66
4.4 A library of compositional operations . . . 67
4.5 Migrating existing programs . . . 67
4.6 Examples . . . 68
4.7 Properties of compositional operations . . . 70
5 Almost Compositional Functions and the Visitor Design Pattern 71
5.1 Abstract syntax representation . . . 71
5.2 ComposVisitor . . . 72

5.3 Using ComposVisitor 72
6 Language and Tool Support for Compositional Operations 74
7 Related Work 75
7.1 Scrap Your Boilerplate 75
7.2 The Tree set constructor 80
7.3 Related work in object-oriented programming 82
7.4 Nanopass framework for compiler education 82
8 Future Work 82
8.1 Automatic generation of *compos* for existing types 82
8.2 Applications in natural language processing 83
8.3 Tree types and generic programming 83
9 Conclusions 83

Acknowledgments

First and foremost, I would like to thank my supervisor Aarne Ranta for his superb support, constantly interesting discussions, and for getting me involved in language technology research to begin with. I would also like to thank the members of my PhD committee, Bengt Nordström, Koen Claessen, and Robin Cooper, the current and former PhD students in the language technology research group, Harald Hammarström, Markus Forsberg, Håkan Burden, Peter Ljunglöf, Kristofer Johannisson, and Janna Khegai, the other local members of the TALK project, David Hjelm, Staffan Larsson, Stina Ericsson, Rebecca Jonson, Jessica Villing, Ann-Charlotte Forslund, Andreas Wallentin, and Mikael Sandin, and all members of the Department of Computer Science and Engineering. Angela, thank you for your love, and for sharing your wisdom about what it is to be a PhD student. I would not be here today without you. In order to make the reader continue past this page, there are more acknowledgments in the included papers. This work has been partly funded by the EU TALK project, IST-507802, and Library-Based Grammar Engineering, Swedish Research Council project dnr 2005-4211.

Björn Bringert Göteborg
July 2007

Introduction

This thesis shows how Grammatical Framework (GF) grammars can be used to simplify development of interactive speech applications. This chapter provides some background on interactive speech applications and introduces the three articles which make up the bulk of the thesis. The three articles describe how GF grammars can be used for speech recognition grammars, multimodality and semantic transfer.

The articles are presented in the context of interactive speech applications since this is an area where they can be used together. There are also other applications of these results, which is perhaps most readily apparent in the third article, where most of the examples are from programming language implementation rather than natural language processing.

1 Interactive Speech Applications

What do we mean by interactive speech applications? By *interactive*, we mean that the system gets input from a user, and delivers timely output as a result of user input. There is some relation between the input and output. A single task may be accomplished using one or more input/output interactions. We are mainly concerned with *speech applications*, that is, applications which get speech input from the user, and deliver speech output to the user. In the case of a speech translator, one user produces the input, and another receives the output. An interactive speech application may also be *multimodal*, that is, it may use multiple modes of communication, or *modalities*. Gestures and drawings are possible examples of modalities other than speech. Both the user input and the system output can be multimodal. Systems which are not multimodal are called *unimodal*.

Interactive speech applications have long been a staple of science fiction. Figures 1 and 2 illustrate two such applications: a *speech translator*, and a *dialogue system*. Interactive speech applications are already in limited commercial use. Examples of such applications include phone-based travel reservation systems and speech-enabled phrase books. However, interactive speech applications have yet to have a significant impact on everyday life. There are three major problems with current interactive speech applications:

1. They are not natural enough.

2. They are not usable enough.

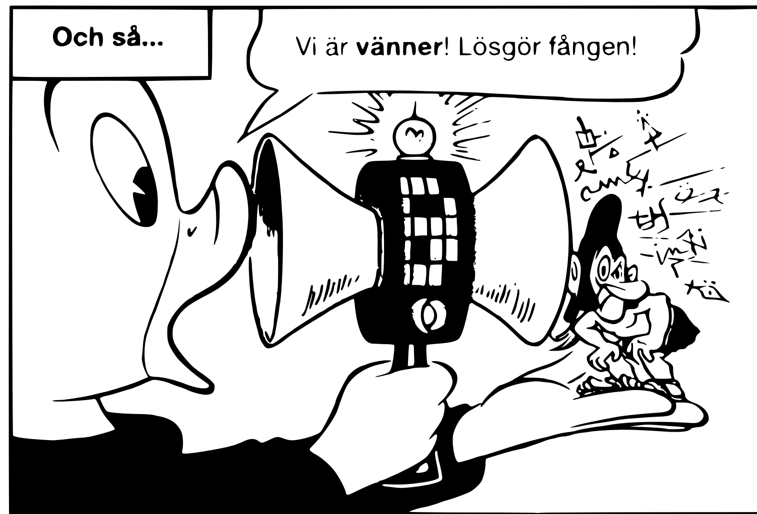


Figure 1. A speech translator, from the Uncle Scrooge story “Planet-planering” (English title “Scrooge’s Space Show”) (Branca et al. 1987). Louie says “We are friends! Release the prisoner!”, though in the original English version he says “Our uncle is a mega-merchant come to trade with you guys!”. ©Disney. Located with help from Sivebæk, Willot and Jensen (2007).

Figure 2. A dialogue system, from the Uncle Scrooge story “Operation Hajön” (English title “Operation Gootchy-goo”) (Strobl and Steere 1985). Uncle Scrooge says: “Stop babbling about the weather! I want to know if everything is proceeding according to plan!”. Smedly, the computer, responds “Right! Well, let me see. . . ”, quite like a GoDiS (Larsson 2002) dialogue system would. ©Walt Disney Productions. Located with help from Sivebæk, Willot and Jensen (2007).

3. They are not cheap enough.

According to Pieraccini (2005), academic dialogue system research is largely focused on the problem of naturalness, whereas industrial dialogue system development is more concerned with usability. Waibel (2004) considers high development cost and limited domains to be the major problems in speech translation research.

There are many applications where the current state of the art means that it is *not possible* to build systems that are natural or usable enough. However, there are also many applications which could benefit from use of even the current state of interactive speech technology, but where it is *not economically viable*, because of the high cost of implementing the systems.

1.1 Problems

The problems of naturalness, usability and cost are large and complex. This thesis deals with the following sub-problems:

Duplicated work In current practise, multiple components are developed semi-independently, with much duplicated effort. For example, speech recognition grammars and semantic interpretation components both need to take into account the linguistic and domain-specific coverage of the system.

Consistency Because of the lack of abstraction mechanisms and consistency checks, it is difficult to modify a system which uses multiple hand-written components. The problem is multiplied when the system is multilingual. The developer then has to modify each of the components, such as speech recognition grammars and semantic interpretation, manually for each language. A simple change may require touching many parts of the system, and there are no automatic consistency checks.

Localization With hand-written components, it is about as difficult to add support for a new language as it is to write the grammar, semantic interpretation, and generation components for the first language.

Linguistic quality Because of the lack of powerful language description tools, achieving high syntactic and morphological quality of the system output and the input language models can be costly. This is more pronounced for languages with a richer morphology than English, since current methods are often developed with English in mind.

Domain portability Components implemented for a given application domain can often not be easily reused in other domains.

Platform portability Systems implemented for a given platform (speech recognizer, operating system, programming language, etc.) can often not be used on other platforms.

2 This work

Interactive speech applications are still often written in low-level, platform-specific languages, which require much duplicated work throughout the system. Our aim is to make construction of interactive speech applications easier by compiling high-level specifications to the low-level code used in the running system. GF is "the working programmer's grammar formalism". In this spirit, the approach that we have taken is to use techniques borrowed from programming language implementation to automatically generate system components from grammars.

In the early days of computer programming, programs were written in machine code or assembly languages, very low-level languages which give the programmer full control, but make programs hard to write, limited to a single machine architecture, and difficult to maintain. Today, programs are written in high-level languages which give less control, but make programs easier to write, portable across different machines and operating systems, and easier to maintain. Programs written in high-level languages are compiled to code in low-level languages, which can be run by machines.

The approach to development of interactive speech applications which we describe here is *grammar-based*, since we use high-level grammars to define major parts of the functionality. Several different components used in interactive speech applications can be generated automatically from the grammars. The systems which we generate are *rule-based*, rather than *statistical*. In an experiment by Rayner et al. (2005a), a rule-based speech understanding system was found to outperform a statistical one, and the advantage of the rule-based system increased with the users' familiarity with the system.

In our description of the components which we generate, we consider interactive speech applications which can be implemented as pipelines. The system receives input, which is processed step by step, and in the end output is produced. A multimodal dialogue system may have components such as speech recognition, multimodal fusion, semantic interpretation, dialogue management, domain resources, output generation, linearization, multimodal fission, speech synthesis. Figure 3 shows a schematic view of such a system. In a speech translator, the dialogue management and domain resources may be replaced by a semantic transfer component, as shown in Figure 4. Larger systems, such as the Spoken Language Translator (Rayner et al. 2000), are more complex with more components and a architecture which is not a simple pipeline. The individual components that we generate can be used in more complex architectures, as has been done in experimental dialogue systems (Ericsson et al. 2006) which use the GoDiS (Larsson 2002) implementation of issue-based dialogue management.

2.1 Advantages

This work addresses the problems listed in Section 1.1 in the following ways:

Duplicated work The duplicated work involved in developing multiple com-

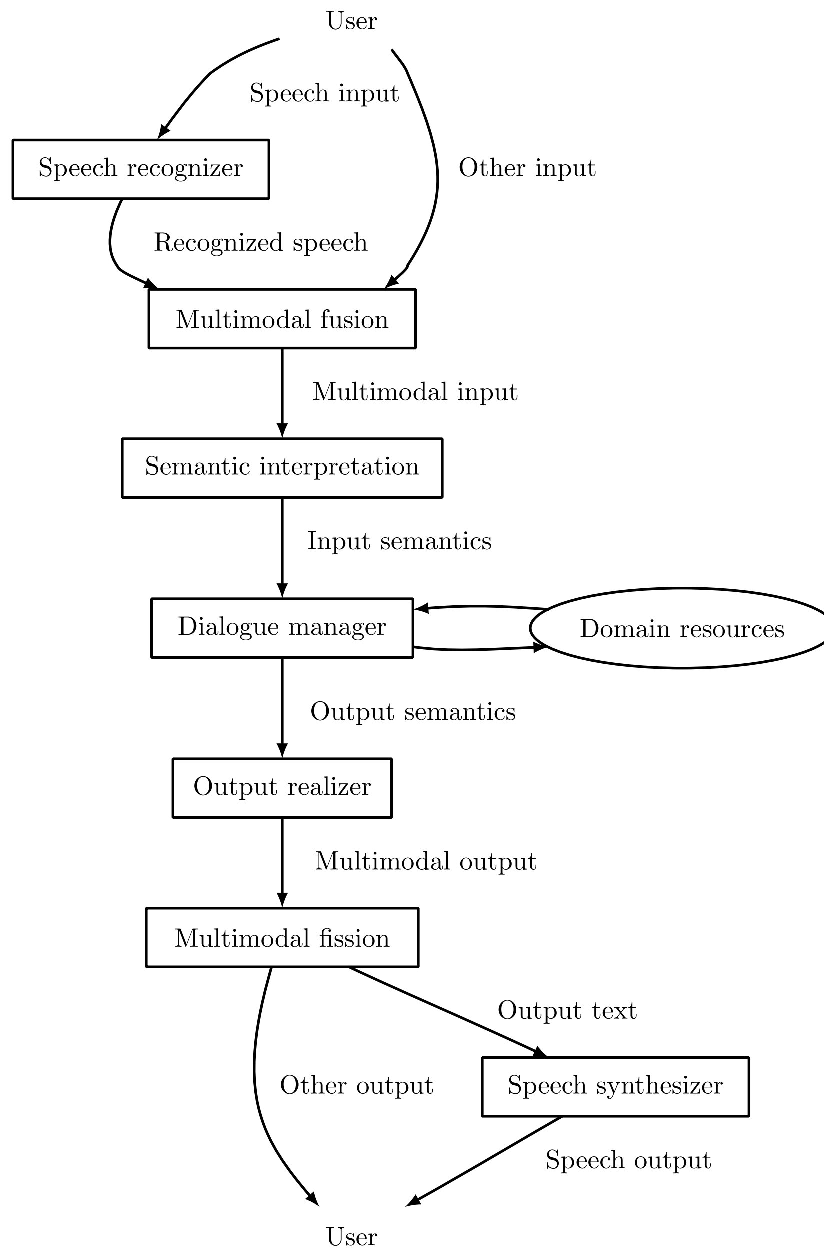


Figure 3. Architecture of a grammar-based multimodal dialogue system. In a unimodal system, there would be no multimodal fission and fusion components.

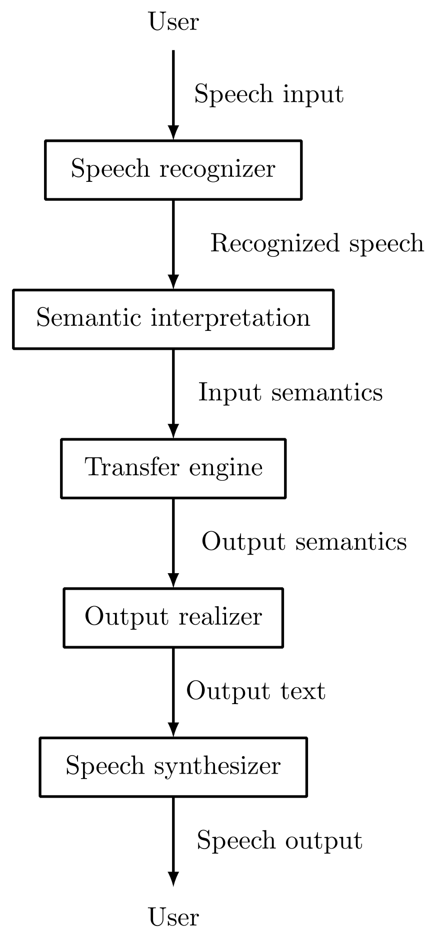


Figure 4. Architecture of a grammar-based speech translator. Compared to Figure 3, there is no multimodality, and the dialogue manager and domain resources have been replaced by a semantic transfer engine.

ponents is avoided by generating all the components from a single declarative source, a GF grammar.

Consistency The strong typing of the GF language enforces consistency between the abstract syntax and the concrete syntaxes for each language. This makes it easier to keep the semantics and the implementations for different languages in sync.

Localization GF's support for multilingual grammars and the common interface implemented by all grammars in the GF resource grammar library makes it easy to translate a system to a new language.

Linguistic quality GF's powerful constructs and the multilingual resource grammar library allows for high morphological and syntactic quality at a low cost.

Domain portability A large portion of the grammar implementation effort is in the resource grammar library. This library can be re-used in multiple domains, instead of being re-implemented for each new application.

Platform portability In our approach, a GF grammar is used as the canonical representation which the developer works with, and components in many formats can be generated automatically from this representation.

2.2 Limitations

The goal is not to mainly allow more sophisticated applications, but rather to reduce the development cost of medium complexity applications. Just like high-level programming languages take away some of the control that the assembly language programmer has, generating system components from grammars places some limits on how systems can be implemented. These limits of course depend on how sophisticated the generation is.

Taken together, the components that we generate fit best in systems with pipeline architectures like the one shown in Figure 3. However, the individual components could also be used as parts of systems with other architectures. For example, a hybrid system could use our components to attempt a deep analysis, and fall back to a separate surface analysis when that fails.

3 Grammatical Framework

We use Grammatical Framework (GF) as the source language for our component generation. This section gives a short introduction to GF, with a small example grammar for a dialogue system.

Grammatical Framework (Ranta 2004) is a type theoretic grammar formalism based on Martin-Löf (1984) type theory. GF makes a distinction between *abstract syntax* and *concrete syntax*, corresponding to Curry's (1961) division of grammar into *tectogrammar* and *phenogrammar*.

abstract Pizza = {
cat Input; Order; Number; Size; Topping; [Topping] {1};
fun *order* : Order → Input;
pizza : Number → Size → [Topping] → Order;
one, *two* : Number;
small, *large* : Size;
cheese, *ham* : Topping;
cat Output;
fun *price* : Order → Int → Output;
}

Figure 5. `Pizza.gf`: A GF abstract syntax module.

The abstract syntax declares *what* can be said in the language. The concrete syntax describes *how* to say it. This is done by associating a concrete representation with each construct in the abstract syntax. In the simplest case, this concrete representation is a record containing a single string field. Records, tables and enumerations can be used to implement more complex representations, for example with gender agreement between nouns and adjectives. The process of generating a concrete syntax term from a term in the abstract syntax is called *linearization.*

One of GF's strong points is multilinguality. The division of grammar into abstract and concrete syntax means that it is easy to have multiple concrete syntaxes for one abstract syntax. This makes it possible to implement multilingual grammars. In order to avoid re-implementing the domain-independent linguistic details of a language for each new application grammar, the GF resource grammar library has been created. It implements the morphological and syntactic details of a number of languages, and presents a language-independent API to the application grammar writer. This significantly reduces the effort involved in translating grammars (Perera and Ranta 2007).

3.1 An Example

As an example of a GF grammar for an interactive speech application, Figure 5 and Figure 6 show the abstract and English concrete syntax for a small pizza ordering dialogue system. The Input category contains user input, such as "two large pizzas with ham and cheese", which corresponds to the abstract syntax term *order* (*pizza two large* [*ham*, *cheese*]). The Output category describes system output, for example "two large pizzas with ham and cheese cost 7 euros", for the abstract syntax term *price* (*pizza two large* [*ham*, *cheese*]) 7.

This concrete syntax module uses the new Resource Grammar API with overloading (Ranta 2006). Each function *mkX* constructs terms in the resource grammar category X. For example, the linearization category of the Order category is the resource grammar category NP. This means that an order

```
concrete PizzaEng of Pizza = open SyntaxEng, ParadigmsEng in {
  flags startcat   = Input;
  lincat Input     = Utt;
         Order     = NP;
         Number    = Det;
         Size      = AP;
         Topping   = NP;
         [Topping] = NP;
  lin order o             = mkUtt o;
      pizza n s ts        = mkNP n (mkCN (mkCN s pizza_N)
                                 (mkAdv with_Prep ts));
      one                 = mkDet n1_Numeral;
      two                 = mkDet n2_Numeral;
      small               = mkAP small_A;
      large               = mkAP large_A;
      cheese              = mkNP massQuant cheese_N;
      ham                 = mkNP massQuant ham_N;
      BaseTopping t       = t;
      ConsTopping t ts    = mkNP and_Conj t ts;
  lincat Output    = Utt;
  lin price o p = mkUtt (mkCl o (mkV2 cost_V)
                     (mkNP (mkNum p) euro_N));
  oper pizza_N  = mkN "pizza";
       small_A  = mkA "small";
       large_A  = variants {mkA "large"; mkA "big" };
       cheese_N = mkN "cheese";
       ham_N    = mkN "ham";
       cost_V   = mkV "cost";
       euro_N   = mkN "euro";
}
```

Figure 6. `PizzaEng.gf`: English concrete syntax for the abstract syntax in Figure 5.

is represented by a noun phrase in this concrete syntax. The *mkNP* function is overloaded, and the version of it that is used in the linearization of *pizza* takes two arguments, one of type Det (determiner, the linearization category of Number) and one of type CN (common noun, here constructed from another common noun and an adverbial phrase). In the linearization of *cheese*, a version of *mkNP* is used that takes a determiner and a noun as arguments. The linearization of *ConsTopping*, one of the two constructors in the [Topping] category, uses a third version of *mkNP* that takes a conjunction and two noun phrases as arguments.

Another noteworthy feature is that the linearization of *large* uses **variants**, to allow alternative ways to express a given input. This is used extensively in realistic dialogue system grammars, to handle variation in how input can be expressed without complicating the semantics.

Figure 7 shows the German concrete syntax for the abstract syntax in Figure 5. Note that the only difference compared to the English concrete syntax in Figure 6 is that it imports German resource grammar modules, and specifies the German application-specific words. GF's *parameterized modules* (Ranta 2007) can be used to create a shared implementation of the language independent part, but that has not been done here.

4 Paper I: Speech Recognition Grammar Compilation in Grammatical Framework

Speech recognizers use *speech recognition grammars* (also known as *language models*) to limit the input language in order to achieve acceptable recognition performance. In the paper "Speech Recognition Grammar Compilation in Grammatical Framework", we show how speech recognition grammars in several commonly used context-free and finite-state formalisms can be generated from GF grammars. We also describe generation of semantic interpretation code which can be embedded in speech recognition grammars.

4.1 An Example

For the Input category in the example grammar in Figure 5 and Figure 6, we can generate the finite-state model shown in Figure 8. Finite-state models such as this one are used to guide the HTK speech recognizer.

4.2 Contribution

I wrote the paper myself, and I implemented the various grammar translations it describes.

```
concrete PizzaGer of Pizza = open SyntaxGer, ParadigmsGer in {
  flags startcat     = Input;
  lincat Input       = Utt;
         Order       = NP;
         Number      = Det;
         Size        = AP;
         Topping     = NP;
         [Topping]   = NP;
  lin order o            = mkUtt o;
      pizza n s ts       = mkNP n (mkCN (mkCN s pizza_N)
                              (mkAdv with_Prep ts));
      one                = mkDet n1_Numeral;
      two                = mkDet n2_Numeral;
      small              = mkAP small_A;
      large              = mkAP large_A;
      cheese             = mkNP massQuant cheese_N;
      ham                = mkNP massQuant ham_N;
      BaseTopping t      = t;
      ConsTopping t ts   = mkNP and_Conj t ts;
  lincat Output      = Utt;
  lin price o p          = mkUtt (mkCl o (mkV2 cost_V)
                              (mkNP (mkNum p) euro_N));
  oper pizza_N   = mkN "Pizza" "Pizzas" feminine;
       small_A   = mkA "klein";
       large_A   = mkA "groß" "größer" "größte";
       cheese_N  = mkN "Käse" "Käse" masculine;
       ham_N     = mkN "Schinken";
       cost_V    = mkV "kostet";
       euro_N    = mkN "Euro" "Euros" masculine;
}
```

Figure 7. `PizzaGer.gf`: German concrete syntax for the abstract syntax in Figure 5.

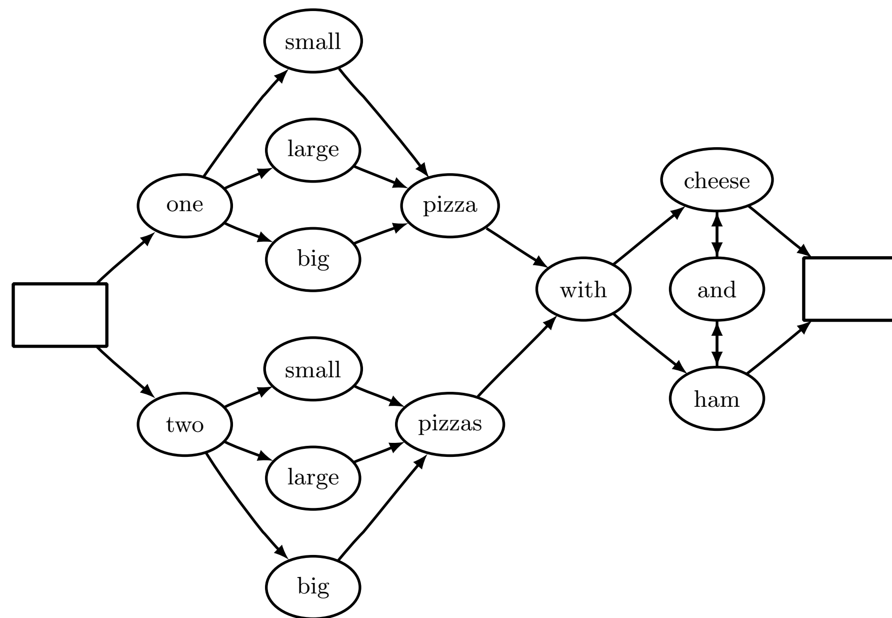


Figure 8. Finite-state language model generated from the English concrete syntax in Figure 6.

4.3 Publication

This paper was presented at SPEECHGRAM 2007, Workshop on Grammar-Based Approaches to Spoken Language Processing, Prague, Czech Republic, June 29, 2007.

5 Paper II: Multimodal Dialogue System Grammars

The paper "Multimodal Dialogue System Grammars" describes how GF grammars can be used to handle multimodality, that is, information presented using multiple modes of communication. Multimodal systems can for example combine speech and pointing gestures for input, or speech and graphics for output. *Multimodal fusion*, the integration of information from multiple modalities into a single semantic representation, and *multimodal fission*, the conversion of a single semantic representation into information in multiple modalities, are handled by using GF's facilities for parsing and linearization, respectively.

```
concrete PizzaDraw of Pizza = {
  lin
    order o          = o;
    pizza n s ts     = {s = call2 "scale" s.s (call2 "replicate" n.s
                              (call2 "above" ts.s (image "pizza")))};
    one              = {s = "1"};
    two              = {s = "2"};
    small            = {s = "0.5"};
    large            = {s = "1.0"};
    cheese           = {s = image "cheese"};
    ham              = {s = image "ham"};
    BaseTopping t    = {s = t.s};
    ConsTopping t ts = {s = call2 "above" t.s ts.s};
  oper
    call0 : Str → Str = λf → f ++ "(" ++ ")";
    call1 : Str → Str → Str = λf → λx → f ++ "(" ++ x ++ ")";
    call2 : Str → Str → Str → Str =
      λf → λx → λy → f ++ "(" ++ x ++ "," ++ y ++ ")";
    image : Str → Str = λx → call1 "image" ("\"" + x + "\"");
}
```

Figure 9. `PizzaDraw.gf`: A concrete syntax which generates drawing instructions from pizza orders.

5.1 An Example

We can extend the example grammar from Section 3 to make a multimodal application. For example, we can write another concrete syntax which generates drawing instructions instead of utterances in natural language. We refer to this as *parallel multimodality*, since the complete information is presented independently in each of the modalities. Figure 9 shows a concrete syntax which generates instructions in a simple drawing language. This can be used to draw graphical representations of pizza orders. Figure 10 shows the graphical representation of the order "two large pizzas with ham and cheese". The abstract syntax representation of this order is *order* (*pizza two large* [*ham*, *cheese*]) and from this, the PizzaDraw concrete syntax generates the drawing instructions: *scale* (1.0, *replicate* (2, *above* (*above* (*image* ("ham"), *image* ("cheese")), *image* ("pizza")))).

Another possible multimodal extension would be to allow spoken pizza orders to contain non-speech parts. For example, we could allow the user to say "I want a large pizza with cheese and that", accompanied by a click on a picture of some topping. This is what we call *integrated multimodality*.

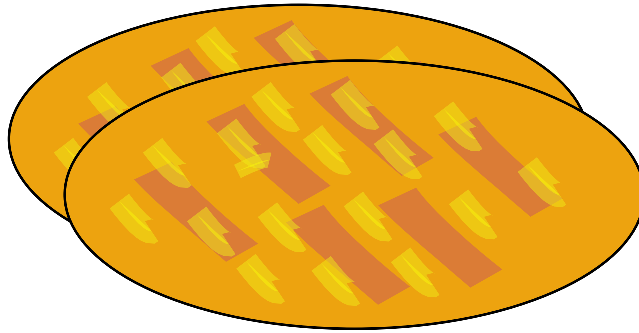


Figure 10. A graphical representation of the pizza order *order* (*pizza two large* [*ham*, *cheese*]), drawn using instructions generated by the concrete syntax in Figure 9.

5.2 Contribution

I designed and implemented the demonstration system, including the grammars, and wrote the sections about the proof-of-concept implementation and related work.

5.3 Publication

This paper was presented at DIALOR'05, Ninth Workshop on the Semantics and Pragmatics of Dialogue, Nancy, France, June 9-11 2005.

6 Paper III: A Pattern for Almost Compositional Functions

The paper "A Pattern for Almost Compositional Functions" introduces a method for simplifying a common class of functions over rich tree-like data types, such as abstract syntax trees in compilers or natural language applications. The method uses a type-specific traversal function, which can be automatically generated from the definition of the data type. This method helps reduce the amount of repetitive traversal code in programs which process rich tree structures.

Dialogue managers and semantic transfer engines process the abstract syntax representation of the input in various ways. There is a significant set of such transformations what are only concerned with some of the constructs in the often quite rich abstract syntax. This paper describes a way to express such transformations succinctly.

$$
\begin{array}{l}
\mathit{uniqueToppings} :: \mathsf{Tree}\ a \rightarrow \mathsf{Tree}\ a \\
\mathit{uniqueToppings}\ t = \textbf{case}\ t\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{pizza}\ n\ s\ \mathit{ts} \rightarrow \mathit{pizza}\ n\ s\ (\mathit{nub}\ \mathit{ts}) \\
\qquad\qquad\qquad\qquad _ \qquad\qquad\ \rightarrow \mathit{composOp}\ \mathit{uniqueToppings}\ t
\end{array}
$$

Figure 11. Haskell-like pseudo-code for an abstract syntax transformation function which removes duplicate toppings from terms in the abstract syntax in Figure 5. The *nub* function removes duplicate elements from a list.

6.1 An Example

Figure 11 shows a small example function which transforms abstract syntax terms. It goes through the term, removing any duplicate toppings in pizza orders. For example, the order *pizza two large* $[\mathit{ham}, \mathit{cheese}, \mathit{ham}]$ ("two large pizzas with ham and cheese and ham") is transformed to *pizza two large* $[\mathit{ham}, \mathit{cheese}]$ ("two large pizzas with ham and cheese"). The *composOp* function, which is the topic of this paper, is used to avoid specifying what the function does for terms other than pizza orders. In this small example, not a lot of code is saved, but in a realistic system with many abstract syntax constructors, the code savings can be very large.

6.2 Contribution

Aarne Ranta used a first version of *composOp* in the GF implementation. He then generalized this to constructive type theory and wrote a first version of the paper, describing the single data-type Haskell versions of *composOp* and *composM*, and a type family version of *composOp* in Agda.

I extended this first version to the full paper included here. Aarne's original paper represents about one fifth of the text of the final version. My contributions include support for *composOp* over type families in extended Haskell, the general *compos* function, the library of functions which use *compos*, the Java Visitor version of the pattern, the description of the relationship to applicative functors, including some identities with proofs, and descriptions of the relationships to generic programming in Haskell, tree types in type theory, traversals in object-oriented programming and the Nanopass framework.

6.3 Publication

This thesis includes a somewhat extended version of a paper presented at ICFP 2006, the 11th ACM SIGPLAN International Conference on Functional Programming, Portland, Oregon, September 18-20, 2006.

7 Related Work

7.1 GF in Interactive Speech Applications

In addition to the work presented in this thesis, there are several other possibilities for using GF grammars in interactive speech applications.

- The main GF implementation has a Haskell API[1] which can be used for parsing and linearization in natural language applications.
- The Embedded GF Interpreter (Bringert 2005) can be used for parsing and linearization in Java programs, and as an OAA (Martin et al. 1999) agent. Several experimental dialogue systems have been developed (Ericsson et al. 2006) using this interpreter along with the speech-recognition grammar generation described in this thesis.
- Peter Ljunglöf has implemented a Prolog library[2] for parsing with GF grammars.
- Jonson (2006) shows how to generate statistical language models for dialogue systems from GF grammars, instead of the grammar-based models that we generate.

7.2 Compiler-like Grammar Development

The Regulus grammar compiler (Rayner et al. 2006b) generates speech recognition grammars, with the possibility of embedding basic semantic interpretation, from unification grammars. Regulus has been used in several interactive speech applications, including the MedSLT (Bouillon et al. 2005; Rayner et al. 2006a) speech translator, and the Clarissa (Rayner et al. 2005b,c) dialogue system.

UNIANCE (Bos 2002) is another system for compiling unification grammars to speech recognition grammars. It includes interpretation code for compositional semantics in the generated grammars.

The SGStudio (Wang and Acero 2005) grammar authoring tool uses a hybrid model for development of speech recognition language models and semantic interpretation. A library of parameterized grammars are used for slot-filling, while a statistical model handles the non-slot-filling parts of user input.

ARIADNE (Denecke 2002) is a dialogue system architecture for rapid prototyping. To build a dialogue system, the developer creates an ontology, parsing grammars, generation templates, database conversion rules, and a description of the services offered by the system. In ARIADNE, the developer writes a set of declarative specifications which together are used in a fixed dialogue system architecture, whereas we generate a number of components, which can be used separately, from a specification in a single formalism. Compared to ARIADNE, we lack the generic dialogue management and database interface components.

[1] See `http://www.cs.chalmers.se/~aarne/GF/src/GF/Embed/EmbedAPI.hs`
[2] See `http://www.ling.gu.se/~peb/software.html`

On the other hand, we have support for multimodality, and GF is more linguistically expressive than the context-free grammars and output templates that ARIADNE uses.

7.3 Interactive Development Environments for Dialogue Systems

There has been substantial work in graphical tools for semi-automatic construction for dialogue systems. One example is the CSLU Rapid Application Developer (McTear 1999), which has support for multilinguality (Cole et al. 1999). The Application Generation Platform (AGP) (Hamerich et al. 2004) can generate multilingual and multimodal interfaces to existing databases semi-automatically. DUDE (Lemon and Liu 2006) is an environment for dialogue system development where the user can semi-automatically construct a dialogue system based on a Business Process Model. DUDE generates GF grammars and makes use of the work described in this thesis to generate speech recognition grammars for the Nuance and HTK speech recognizers. DiaMant (Fliedner and Bobbert 2003) is a GUI tool for rapid development of dialogue systems based on finite state dialogue models, extended with variables. Variant Transduction (Alshawi and Douglas 2001) is an example-based approach to rapid spoken language interface development. Interaction Builder (Katsurada et al. 2005) is a GUI tool for constructing web-based multimodal applications.

These tools all use graphical user interfaces to construct complete dialogue systems, whereas we use a grammar formalism as the user interface, and create general components which can also be used in other kinds of interactive speech applications, such as speech translators.

8 Future work

The overall goal is to make it easier to develop grammar-based interactive applications by generating as much as possible from GF grammars. These are some areas that could be explored further:

- Generation of dialogue management from GF grammars, along the lines of Ranta's and Cooper's (2004) work relating dialogue systems to proof editors.
- Investigation of platforms for deployment of complete generated systems. One promising candidate is XHTML+Voice (Axelsson et al. 2004), which allows web-based multimodal systems which can be accessed by users without requiring installation.
- A programming language for abstract syntax term transformations, which could be used to implement application specific functionality.
- Compilation of GF linearization rules to components usable in mainstream programming languages.

- Robust parsing algorithms for GF grammars. This would allow us to take advantage of the possibility of generating statistical language models from GF grammars (Jonson 2006).
- Development of realistic demonstration systems using our methods.

References

Hiyan Alshawi and Shona Douglas. Variant transduction: a method for rapid development of interactive spoken interfaces. In *Proceedings of the Second SIGdial Workshop on Discourse and Dialogue*, pages 1–9, Morristown, NJ, USA, 2001. Association for Computational Linguistics. URL `http://portal.acm.org/citation.cfm?id=1118080`.

Jonny Axelsson, Chris Cross, Jim Ferrans, Gerald McCobb, T. V. Raman, and Les Wilson. XHTML+Voice profile 1.2. Specification, VoiceXML Forum, 2004. URL `http://www.voicexml.org/specs/multimodal/x+v/12/`.

Johan Bos. Compilation of unification grammars with compositional semantics to speech recognition packages. In *Proceedings of the 19th international conference on Computational linguistics*, pages 1–7, Morristown, NJ, USA, 2002. Association for Computational Linguistics. URL `http://portal.acm.org/citation.cfm?id=1072323`.

P. Bouillon, M. Rayner, N. Chatzichrisafis, B. A. Hockey, M. Santaholma, M. Starlander, H. Isahara, K. Kanzaki, and Y. Nakao. A generic Multi-Lingual Open Source Platform for Limited-Domain Medical Speech Translation. pages 5–58, May 2005. URL `http://www.issco.unige.ch/pub/MedSLT_demo_EAMT05_final.pdf`.

Daniel Branca, Patsy Trench, and Dave Angus. Planet-planering. *Kalle Anka & C:o*, 1987(31), July 1987. ISSN 0345-6048. Disney story code D 8560.

Björn Bringert. Embedded Grammars. Master's thesis, Chalmers University of Technology, Göteborg, Sweden, February 2005. URL `http://www.cs.chalmers.se/~bringert/publ/exjobb/embedded-grammars.pdf`.

Ronald A. Cole, Ben Serridge, John-Paul Hosom, Andrew Cronk, and Ed Kaiser. A Platform for Multilingual Research in Spoken Dialogue Systems. In *Proceedings of the Workshop on Multi-Lingual Interoperability in Speech Technology (MIST)*, pages 43–48, Leusden, The Netherlands, September 1999. URL `http://www.cslu.ogi.edu/people/hosom/pubs/cole_MIST-platform_1999.pdf`.

Haskell B. Curry. Some Logical Aspects of Grammatical Structure. In Roman O. Jakobson, editor, *Structure of Language and its Mathematical Aspects, volume 12 of Symposia on Applied Mathematics*, pages 56–68. American Mathematical Society, Providence, 1961.

Matthias Denecke. Rapid prototyping for spoken dialogue systems. In *Proceedings of the 19th international conference on Computational linguistics*, pages 1–7, Morristown, NJ, USA, 2002. Association for Computational Linguistics. URL http://portal.acm.org/citation.cfm?id=1072375.

Stina Ericsson, Gabriel Amores, Björn Bringert, Håkan Burden, Ann C. Forslund, David Hjelm, Rebecca Jonson, Staffan Larsson, Peter Ljunglöf, Pilar Manchón, David Milward, Guillermo Pérez, and Mikael Sandin. Software illustrating a unified approach to multimodality and multilinguality in the in-home domain. Technical Report 1.6, TALK Project, 2006. URL http://www.talk-project.org/.

Gerhard Fliedner and Daniel Bobbert. DiaMant: A Tool for Rapidly Developing Spoken Dialogue Systems. In *Proceedings of the 7th Workshop on the Semantics and Pragmatics of Dialogue (DiaBruck)*, Wallerfangen, Germany, 2003. URL http://www.coli.uni-saarland.de/conf/diabruck/submission_finals/abstracts/320/demo_320.pdf.

Stefan Hamerich, Volker Schubert, Volker Schless, Ricardo de Córdoba, José M. Pardo, Luis F. D'haro, Basilis Kladis, Otilia Kocsis, and Stefan Igel. Semi-Automatic Generation of Dialogue Applications in the GEMINI Project. In Michael Strube and Candy Sidner, editors, *Proceedings of the 5th SIGdial Workshop on Discourse and Dialogue*, pages 31–34, Cambridge, Massachusetts, USA, 2004. Association for Computational Linguistics. URL http://acl.ldc.upenn.edu/hlt-naacl2004/sigdial04/pdf/hamerich.pdf.

Rebecca Jonson. Generating Statistical Language Models from Interpretation Grammars in Dialogue Systems. In *Proceedings of EACL'06*, 2006. URL http://citeseer.ist.psu.edu/jonson06generating.html.

K. Katsurada, H. Adachi, K. Sato, H. Yamada, and T. Nitta. Interaction builder: A rapid prototyping tool for developing web-based MMI applications. *IEICE Trans Inf Syst*, E88-D(11):2461–2467, 2005. doi: 10.1093/ietisy/e88-d.11.2461. URL http://dx.doi.org/10.1093/ietisy/e88-d.11.2461.

Staffan Larsson. *Issue-based Dialogue Management*. PhD thesis, Göteborg University, 2002.

Oliver Lemon and Xingkun Liu. DUDE: a Dialogue and Understanding Development Environment, mapping Business Process Models to Information State Update dialogue systems. In *EACL 2006, 11st Conference of the European Chapter of the Association for Computational Linguistics*, 2006. URL http://homepages.inf.ed.ac.uk/olemon/dude-final.pdf.

David L. Martin, Adam J. Cheyer, and Douglas B. Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1-2):91–128, 1999.

URL http://www.scopus.com/scopus/record/display.url?view=extended&origin=resultslist&eid=2-s2.0-0032805927.

Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Naples, 1984.

Michael F. McTear. Software to support research and development of spoken dialogue systems. In *Proceedings of Eurospeech'99*, Budapest, Hungary, 1999. URL http://citeseer.ist.psu.edu/548113.html.

Nadine Perera and Aarne Ranta. An Experiment in Dialogue System Localization with the GF Resource Grammar Library. In *SPEECHGRAM 2007*, 2007.

Roberto Pieraccini and Juan Huerta. Where do we go from here? Research and commercial spoken dialog systems. In *Proceedings of the 6th SIGdial Workshop on Discourse and Dialogue*, Lisbon, Portugal, September 2005. URL http://www.sigdial.org/workshops/workshop6/proceedings/pdf/65-SigDial2005_8.pdf.

Aarne Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2):145–189, March 2004. ISSN 0956-7968. doi: 10.1017/S0956796803004738. URL http://portal.acm.org/citation.cfm?id=967507.

Aarne Ranta. Grammars as software libraries. Manuscript, 2006. URL http://www.cs.chalmers.se/~aarne/articles/grammars-libraries.pdf.

Aarne Ranta. Modular Grammar Engineering in GF. To appear in Research on Language and Computation, 2007. URL http://www.cs.chalmers.se/~aarne/articles/multieng3.pdf.

Aarne Ranta and Robin Cooper. Dialogue Systems as Proof Editors. *Journal of Logic, Language and Information*, 13(2):225–240, 2004. ISSN 0925-8531. doi: 10.1023/B:JLLI.0000024736.34644.48. URL http://dx.doi.org/10.1023/B:JLLI.0000024736.34644.48.

Manny Rayner, David Carter, Pierrette Bouillon, Vassilis Digalakis, and Mats Wirén, editors. *The Spoken Language Translator.* Studies in Natural Language Processing. November 2000. doi: 10.2277/0521770777. URL http://dx.doi.org/10.2277/0521770777.

Manny Rayner, Pierrette Bouillon, Nikos Chatzichrisafis, Beth A. Hockey, Marianne Santaholma, Marianne Starlander, Hitoshi Isahara, Kyoko Kanzaki, and Yukie Nakao. A Methodology for Comparing Grammar-Based and Robust Approaches to Speech Understanding. In *Proceedings of Interspeech 2005*, 2005a. URL http://www.issco.unige.ch/pub/RaynerEAInterspeech2005.pdf.

Manny Rayner, Beth A. Hockey, Nikos Chatzichrisafis, Kim Farrell, and Jean-Michel Renders. A voice enabled procedure browser for the International Space Station. In *ACL '05: Proceedings of the ACL 2005 on Interactive poster and demonstration sessions*, pages 29–32, Morristown, NJ, USA, 2005b. Association for Computational Linguistics. URL http://portal.acm.org/citation.cfm?id=1225761.

Manny Rayner, Beth A. Hockey, Jean-Michel Renders, Nikos Chatzichrisafis, and Kim Farrell. Spoken Language Processing in the Clarissa Procedure Browser. Technical report, International Computer Science Institute, Berkeley, California, April 2005c. URL ftp://ftp.icsi.berkeley.edu/pub/techreports/2005/tr-05-005.pdf.

Manny Rayner, Pierrette Bouillon, Nikos Chatzichrisafis, Marianne Santaholma, Marianne Starlander, Beth A. Hockey, Yukie Nakao, Hitoshi Isahara, and Kyoko Kanzaki. MedSLT: A Limited-Domain Unidirectional Grammar-Based Medical Speech Translator. In *Proceedings of the First International Workshop on Medical Speech Translation*, pages 40–43, New York, New York, 2006a. Association for Computational Linguistics. URL http://acl.ldc.upenn.edu//W/W06/W06-3707.pdf.

Manny Rayner, Beth A. Hockey, and Pierrette Bouillon. *Putting Linguistics into Speech Recognition: The Regulus Grammar Compiler.* CSLI Publications, Ventura Hall, Stanford University, Stanford, CA 94305, USA, July 2006b. ISBN 1575865262.

Anders Christian Sivebæk, François Willot, and Lars Jensen. IRC conversation, March 2007. irc://irc.inducks.org:6667/dcml.

Tony Strobl and Steve Steere. Operation Hajön. *Musse Pigg & C:o*, 1985(5), May 1985. ISSN 0349-1463. Disney story code S 81111.

Alex Waibel. Speech Translation: Past, Present and Future. In *INTERSPEECH-2004*, pages 353–356, October 2004. URL http://scholar.google.com/scholar?q=cache:data.cstr.ed.ac.uk/internal/library/proceedings/2004/icslp2004/contents/TuB_pdf/Spec3801o/Spec3801o.1_p1342.pdf.

Ye-Yi Wang and Alex Acero. SGStudio: Rapid Semantic Grammar Development for Spoken Language Understanding. In *Proceedings of the Interspeech Conference*, Lisbon, Portugal, September 2005. URL http://research.microsoft.com/srg/papers/2005-yeyiwang-eurospeech.pdf.

Paper I

Speech Recognition Grammar Compilation in Grammatical Framework

SPEECHGRAM 2007, Prague

Speech Recognition Grammar Compilation in Grammatical Framework

Björn Bringert
Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
bringert@cs.chalmers.se

Abstract

This paper describes how grammar-based language models for speech recognition systems can be generated from Grammatical Framework (GF) grammars. Context-free grammars and finite-state models can be generated in several formats: GSL, SRGS, JSGF, and HTK SLF. In addition, semantic interpretation code can be embedded in the generated context-free grammars. This enables rapid development of portable, multilingual and easily modifiable speech recognition applications.

1 Introduction

Speech recognition grammars are used for guiding speech recognizers in many applications. However, there are a number of problems associated with writing grammars in the low-level, system-specific formats required by speech recognizers. This work addresses these problems by generating speech recognition grammars and semantic interpretation components from grammars written in Grammatical Framework (GF), a high-level, type-theoretical grammar formalism. Compared to existing work on compiling unification grammars, such as Regulus (Rayner et al. 2006), our work uses a type-theoretical grammar formalism with a focus on multilinguality and modular grammar development, and supports multiple speech recognition grammar formalisms, including finite-state models.

We first outline some existing problems in the development and maintenance of speech recognition grammars, and describe how our work attempts to address these problems. In the following two sections we introduce speech recognition grammars and Grammatical Framework. The bulk of the paper then describes how we generate context-free speech recognition grammars, finite-state language models and semantic interpretation code from GF grammars. We conclude by giving references to a number of experimental dialogue systems which already use our grammar compiler for generating speech recognition grammars.

Expressivity Speech recognition grammars are written in simple formalisms which do not have the powerful constructs of high-level grammar formalisms. This makes speech recognition grammar writing labor-intensive and error prone, especially for languages with more inflection and agreement than English.

This is solved by using a high-level grammar formalism with powerful constructs and a grammar library which implements the domain-independent linguistic details.

Duplicated work When speech recognition grammars are written directly in the low-level format required by the speech recognizer, other parts of the system, such as semantic interpretation components, must often be constructed separately.

This duplicated work can be avoided by generating all the components from a single declarative source, such as a GF grammar.

Consistency Because of the lack of abstraction mechanisms and consistency checks, it is difficult to modify a system which uses hand-written speech recognition grammars. The problem is multiplied when the system is multilingual. The developer has to modify the speech recognition grammar and the semantic interpretation component manually for each language. A simple change may require touching many parts of the grammar, and there are no automatic consistency checks.

The strong typing of the GF language enforces consistency between the semantics and the concrete representation in each language.

Localization With hand-written grammars, it is about as difficult to add support for a new language as it is to write the grammar and semantic interpretation for the first language.

GF's support for multilingual grammars and the common interface implemented by all grammars in the GF resource grammar library makes it easier to translate a grammar to a new language.

Portability A grammar in any given speech recognition grammar format cannot be used with a speech recognizer which uses another format.

In our approach, a GF grammar is used as the canonical representation which the developer works with, and speech recognition grammars in many formats can be generated automatically from this representation.

2 Speech Recognition Grammars

To achieve acceptable accuracy, speech recognition software is guided by a *language model* which defines the language which can be recognized. A language model may also assign different probabilities to different strings in the language. A language model can either be a *statistical language model (SLM)*, such as an

n-gram model, or a *grammar-based language model*, for example a *context-free grammar (CFG)* or a *finite-state automaton (FSA)*. In this paper, we use the term *speech recognition grammar (SRG)* to refer to all grammar-based language models, including context-free grammars, regular grammars and finite-state automata.

3 Grammatical Framework

Grammatical Framework (GF) (Ranta 2004) is a grammar formalism based on constructive type theory. In GF, an *abstract syntax* defines a semantic representation. A *concrete syntax* declares how terms in an abstract syntax are *linearized*, that is, how they are mapped to concrete representations. GF grammars can be made multilingual by having multiple concrete syntaxes for a single abstract syntax.

3.1 The Resource Grammar Library

The GF Resource Grammar Library (Ranta et al. 2006) currently implements the morphological and syntactic details of 10 languages. This library is intended to make it possible to write grammars without caring about the linguistic details of particular languages. It is inspired by *library-based software engineering*, where complex functionality is implemented in reusable software libraries with simple interfaces.

The resource grammar library is used through GF's facility for *grammar composition*, where the abstract syntax of one grammar is used in the implementation of the concrete syntax of another grammar. Thus, an application grammar writer who uses a resource grammar uses its abstract syntax terms to implement the linearizations in the application grammar.

The resource grammars for the different languages implement a common interface, i.e. they all have a common abstract syntax. This means that grammars which are implemented using resource grammars can be easily localized to other languages. Localization normally consists of translating the application-specific lexical items, and adjusting any linearizations which turn out to be unidiomatic in the language in question. For example, when the GoTGoDiS (Ericsson et al. 2006) application was localized to Finnish, only 3 out of 180 linearization rules had to be changed.

3.2 An Example GF Grammar

Figure 1 contains a small example GF abstract syntax. Figure 2 defines an English concrete syntax for it, using the resource grammar library. We will use this grammar when we show examples of speech recognition grammar generation later.

In the abstract syntax, **cat** judgements introduce syntactic categories, and **fun** judgements declare constructors in those categories. For example, the *items*

```
abstract Food = {
  cat Order; Items; Item; Number; Size;
  fun order : Items → Order;
      and   : Items → Items → Items;
      items : Item → Number → Size → Items;
      pizza, beer   : Item;
      one, two      : Number;
      small, large  : Size;
}
```

Figure 1. `Food.gf`: A GF abstract syntax module.

constructor makes an Items term from an Item, a Number and a Size. The term *items pizza two small* is an example of a term in this abstract syntax.

In the concrete syntax, a **lincat** judgement declares the type of the concrete terms generated from the abstract syntax terms in a given category. The linearization of each constructor is declared with a **lin** judgement. In the concrete syntax in Figure 2, library functions from the English resource grammar are used for the linearizations, but it is also possible to write concrete syntax terms directly. The linearization of the term *items pizza two small* is $\{s =$ "two small pizzas"$\}$, a record containing a single string field.

By changing the imports and the four lexical items, this grammar can be translated to any other language for which there is a resource grammar. For example, in the German version, we replace (*regN* "beer") with (*reg2N* "Bier" "Biere" *neuter*) and so on. The functions *regN* and *reg2N* implement paradigms for regular English and German nouns, respectively. This replacement can be formalized using GF's *parameterized modules*, which lets one write a common implementation that can be instantiated with the language-specific parts. Note that the application grammar does not deal with details such as agreement, as this is taken care of by the resource grammar.

4 Generating Context-free Grammars

4.1 Algorithm

GF grammars are converted to context-free speech recognition grammars in a number of steps. An overview of the compilation pipeline is show in Figure 3. The figure also includes compilation to finite-state automata, as described in Section 5. Each step of the compilation is described in more detail in the sections below.

Conversion to CFG The GF grammar is first converted into a context-free grammar annotated with functions and profiles, as described by Ljunglöf (2004).

```
concrete FoodEng of Food = open SyntaxEng, ParadigmsEng in {
  flags startcat   = Order;
  lincat Order     = Utt; Items    = NP;
         Item      = CN; Number = Det;
         Size      = AP;
  lin order x      = mkUtt x;
      and x y      = mkNP and_Conj x y;
      items x n s  = mkNP n (mkCN s x);
      pizza        = mkCN (regN "pizza");
      beer         = mkCN (regN "beer");
      one          = mkDet n1_Numeral;
      two          = mkDet n2_Numeral;
      small        = mkAP (regA "small");
      large        = mkAP (regA "large");
}
```

Figure 2. `FoodEng.gf`: English concrete syntax for the abstract syntax in Figure 1.

Cycle elimination All directly and indirectly cyclic productions are removed, since they cannot be handled gracefully by the subsequent left-recursion elimination. Such productions do not contribute to the coverage to the grammar, only to the set of possible semantic results.

Bottom-up filtering Productions whose right-hand sides use categories for which there are no productions are removed, since these will never match any input.

Top-down filtering Only productions for categories which can be reached from the start category are kept. This is mainly used to remove parts of the grammar which are unused because of the choice of start category. One example where this is useful is when a speech recognition grammar is generated from a multimodal grammar (Bringert et al. 2005). In this case, the start category is different from the start category used by the parser, in that its linearization only contains the speech component of the input. Top-down filtering then has the effect of excluding the non-speech modalities from the speech recognition grammar.

The bottom-up and top-down filtering steps are iterated until a fixed point is reached, since both these steps may produce new filtering opportunities.

Left-recursion elimination All direct and indirect left-recursion is removed using the LC_{LR} transform described by Moore (2000). We have modified the

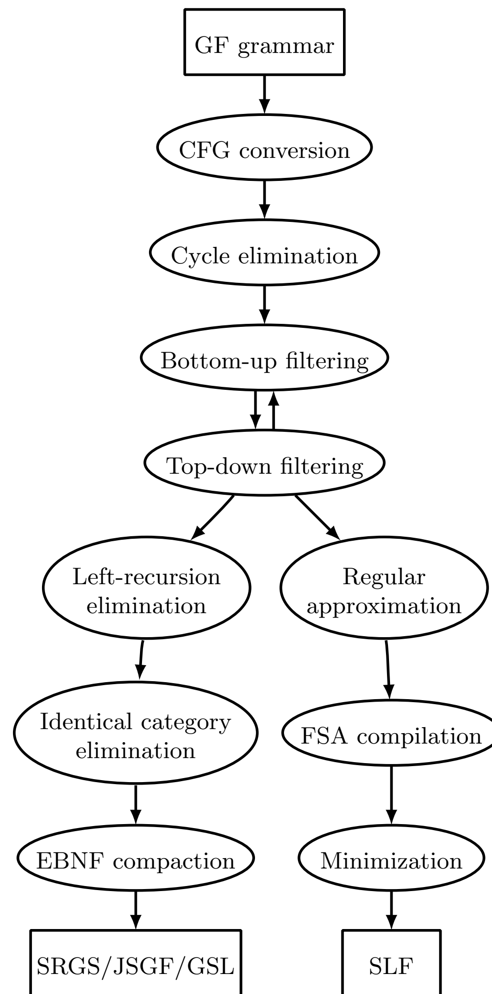


Figure 3. Grammar compilation pipeline.

LC_{LR} transform to avoid adding productions which use a category A–X when there are no productions for A–X.

Identical category elimination In this step, the categories are grouped into equivalence classes by their right-hand sides and semantic annotations. The categories $A_1 \dots A_n$ in each class are replaced by a single category $A_1 + \dots + A_n$ throughout the grammar, discarding any duplicate productions. This has the effect of replacing all categories which have identical sets of productions with a single category. Concrete syntax parameters which do not affect inflection is one source of such redundancy; the LC_{LR} transform is another.

EBNF compaction The resulting context-free grammar is compacted into an *Extended Backus-Naur Form (EBNF)* representation. This reduces the size and improves the readability of the final grammar. The compaction is done by, for each category, grouping all the productions which have the same semantic interpretation, and the same sequence of non-terminals on their right-hand sides, ignoring any terminals. The productions in each group are merged into one EBNF production, where the terminal sequences between the non-terminals are converted to regular expressions which are the unions of the original terminal sequences. These regular expressions are then minimized.

Conversion to output format The resulting non-left-recursive grammar is converted to SRGS, JSGF or Nuance GSL format.

A fragment of a SRGS ABNF grammar generated from the GF grammar in Figure 2 is shown below. The left-recursive *and* rule was removed from the grammar before compilation, as the left-recursion elimination step makes it difficult to read the generated grammar. The fragment shown here is for the singular part of the *items* rule.

```
$FE1 = $FE6 $FE9 $FE4;
$FE6 = one;
$FE9 = large | small;
$FE4 = beer  | pizza;
```

The corresponding fragment generated from the German version of the grammar is more complex, since the numeral and the adjective must agree with the gender of the noun.

```
$FG1  = $FG10 $FG13 $FG6 | $FG9 $FG12 $FG4;
$FG9  = eine;
$FG10 = ein;
$FG12 = große  | kleine;
$FG13 = großes | kleines;
$FG4  = Pizza;
$FG6  = Bier;
```

4.2 Discussion

The generated grammar is an overgenerating approximation of the original GF grammar. This is inevitable, since the GF formalism is stronger than context-free grammars, for example through its support for reduplication. GF's support for dependently typed and higher-order abstract syntax is also not yet carried over to the generated speech recognition grammars. This could be handled in a subsequent semantic interpretation step. However, that requires that the speech recognizer considers multiple hypotheses, since some may be discarded by the semantic interpretation. Currently, if the abstract syntax types are only dependent on finite types, the grammar can be expanded to remove the dependencies. This appears to be sufficient for many realistic applications.

In some cases, empty productions in the generated grammar could cause problems for the cycle and left-recursion elimination, though we have yet to encounter this in practice. Empty productions can be removed by transforming the grammar, though this has not yet been implemented.

For some grammars, the initial CFG generation can generate a very large number of productions. While the resulting speech recognition grammars are of a reasonable size, the large intermediate grammars can cause memory problems. Further optimization is needed to address this problem.

5 Finite-State Models

5.1 Algorithm

Some speech recognition systems use finite-state automata rather than context-free grammars as language models. GF grammars can be compiled to finite-state automata using the procedure shown in Figure 3. The initial part of the compilation to a finite-state model is shared with the context-free SRG compilation, and is described in Section 4.

Regular approximation The context-free grammar is approximated with a regular grammar, using the algorithm described by Mohri and Nederhof (2001).

Compilation to finite-state automata The regular grammar is transformed into a set of *non-deterministic finite automata (NFA)* using a modified version of the *make_fa* algorithm described by Nederhof (2000). For realistic grammars, applying the original *make_fa* algorithm to the whole grammar generates a very large automaton, since a copy of the sub-automaton corresponding to a given category is made for every use of the category.

Instead, one automaton is generated for each category in the regular grammar. All categories which are not in the same mutually recursive set as the category for which the automaton is generated are treated as terminal symbols. This results in a set of automata with edges labeled with either terminal symbols or the names of other automata.

If desired, the set of automata can be converted into a single automaton by substituting each category-labeled edge with a copy of the corresponding automaton. Note that this always terminates, since the sub-automata do not have edges labeled with the categories from the same mutually recursive set.

Minimization Each of the automata is turned into a minimal *deterministic finite automaton (DFA)* by using Brzozowski's (1962) algorithm, which minimizes the automaton by performing two determinizations and reversals.

Conversion to output format The resulting finite automaton can be output in HTK *Standard Lattice Format (SLF)*. SLF supports sub-lattices, which allows us to convert our set of automata directly into a set of lattices. Since SLF uses labeled nodes, rather than labeled edges, we move the labels to the nodes. This is done by first introducing a new labeled node for each edge, and then eliminating all internal unlabeled nodes. Figure 4 shows the SLF model generated from the example grammar. For clarity, the sub-lattices have been inlined.

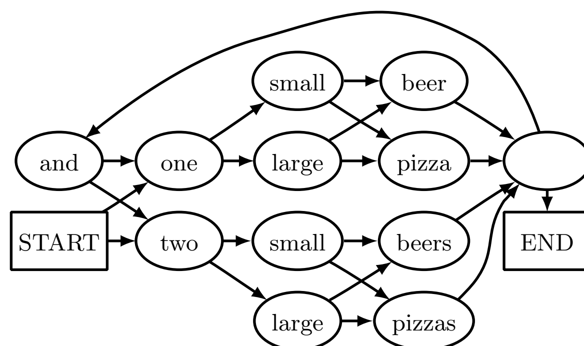


Figure 4. SLF model generated from the grammar in Figure 2.

5.2 Discussion

Finite-state models are even more restrictive than context-free grammars. This problem is handled by approximating the context-free grammar with an overgenerating finite-state automaton. This may lead to failure in a subsequent parsing step, which, as in the context-free case, is acceptable if the recognizer can return all hypotheses.

6 Semantic Interpretation

Semantic interpretation can be done as a separate parsing step after speech recognition, or it can be done with semantic information embedded in the speech recognition grammar. The latter approach resembles the *semantic actions* used

by parser generators for programming languages. One formalism for semantic interpretation is the proposed *Semantic Interpretation for Speech Recognition (SISR)* standard. SISR tags are pieces of ECMAScript code embedded in the speech recognition grammar.

6.1 Algorithm

The GF system can include SISR tags when generating speech recognitions grammars in SRGS and JSGF format. The SISR tags are generated from the semantic information in the annotated CFG (Ljunglöf 2004). The result of the semantic interpretation is an abstract syntax term.

The left-recursion elimination step makes it somewhat challenging to produce correct abstract syntax trees. We have extended Moore's (2000) LC_{LR} transform to preserve the semantic interpretation. The LC_{LR} transform introduces new categories of the form $A-X$ where X is a proper left corner of a category A. The new category A–X can be understood as "the category A, but missing an initial X". Thus the semantic interpretation for a production in A–X is the semantic interpretation for the original A-production, abstracted (in the λ-calculus sense) over the semantic interpretation of the missing X. Conversely, where-ever a category A–X is used, its result is applied to the interpretation of the occurrence of X.

6.2 Discussion

As discussed in Section 4.2, the semantic interpretation code could be used to implement the non-context-free features of GF, but this is not yet done.

The slot-filling mechanism in the GSL format could also be used to build semantic representations, by returning program code which can then be executed. The UNIANCE grammar compiler (Bos 2002) uses that approach.

7 Related Work

7.1 Unification Grammar Compilation

Compilation of unification grammars to speech recognition grammars is well described in the literature (Moore 1999; Dowding et al. 2001). Regulus (Rayner et al. 2006) is perhaps the most ambitious such system. Like GF, Regulus uses a general grammar for each language, which is specialized to a domain-specific one. Ljunglöf (Ljunglöf 2007a) relates GF and Regulus by showing how to convert GF grammars to Regulus grammars. We carry compositional semantic interpretation through left-recursion elimination using the same idea as the UNIANCE grammar compiler (Bos 2002), though our version handles both direct and indirect left-recursion.

The main difference between our work and the existing compilers is that we work with type-theoretical grammars rather than unification grammars. While the existing work focuses on GSL as the output language, we also support

a number of other formats, including finite-state models. By using the GF resource grammars, speech recognition language models can be produced for more languages than with previous systems. One shortcoming of our system is that it does not yet have support for weighted grammars.

7.2 Generating SLMs from GF Grammars

Jonson (2006) has shown that in addition to generating grammar-based language models, GF can be used to build statistical language models (SLMs). It was found that compared to our grammar-based approach, use of generated SLMs improved the recognition performance for out-of-grammar utterances significantly.

8 Results

Speech recognition grammars generated from GF grammars have already been used in a number of research dialogue systems.

GOTTIS (Bringert et al. 2005; Ericsson et al. 2006), an experimental multimodal and multilingual dialogue system for public transportation queries, uses GF grammars for parsing multimodal input. For speech recognition, it uses GSL grammars generated from the speech modality part of the GF grammars.

DJ-GoDiS, GoDiS-deLUX, and GoTGoDiS (Ericsson et al. 2006) are three applications which use GF grammars for speech recognition and parsing together with the GoDiS implementation of issue-based dialogue management (Larsson 2002). GoTGoDiS has been translated to 7 languages using the GF resource grammar library, with each new translation taking less than one day (Ericsson et al. 2006).

The DICO (Villing and Larsson 2006) dialogue system for trucks has recently been modified to use GF grammars for speech recognition and parsing (Ljunglöf 2007b).

DUDE (Lemon and Liu 2006) and its extension REALL-DUDE (Lemon et al. 2006b) are environments where non-experts can develop dialogue systems based on Business Process Models describing the applications. From keywords, prompts and answer sets defined by the developer, the system generates a GF grammar. This grammar is used for parsing input, and for generating a language model in SLF or GSL format.

The Voice Programming system by Georgila and Lemon (Georgila and Lemon 2006; Lemon et al. 2006a) uses an SLF language model generated from a GF grammar.

Perera and Ranta (2007) have studied how GF grammars can be used for localization of dialogue systems. A GF grammar was developed and localized to 4 other languages in significantly less time than an equivalent GSL grammar. They also found the GSL grammar generated by GF to be much smaller than the hand-written GSL grammar.

9 Conclusions

We have shown how GF grammars can be compiled to several common speech recognition grammar formats. This has helped decrease development time, improve modifiability, aid localization and enable portability in a number of experimental dialogue systems.

Several systems developed in the TALK and DICO projects use the same GF grammars for speech recognition, parsing and multimodal fusion (Ericsson et al. 2006). Using the same grammar for multiple system components **reduces development and modification costs**, and makes it easier to **maintain consistency** within the system.

The feasibility of **rapid localization** of dialogue systems which use GF grammars has been demonstrated in the GoTGoDiS (Ericsson et al. 2006) system, and in experiments by Perera and Ranta (2007).

Using speech recognition grammars generated by GF makes it easy to **support different speech recognizers**. For example, by using the GF grammar compiler, the DUDE (Lemon and Liu 2006) system can support both the ATK and Nuance recognizers.

Implementations of the methods described in this paper are freely available as part of the GF distribution[1].

Acknowledgments

Aarne Ranta, Peter Ljunglöf, Rebecca Jonson, David Hjelm, Ann-Charlotte Forslund, Håkan Burden, Xingkun Liu, Oliver Lemon, and the anonymous referees have contributed valuable comments on the grammar compiler implementation and/or this article. We would like to thank Nuance Communications, Inc., OptimSys, s.r.o., and Opera Software ASA for software licenses and technical support. The code in this paper has been typeset using `lhs2TeX`, with help from Andres Löh. This work has been partly funded by the EU TALK project, IST-507802.

References

Johan Bos. Compilation of unification grammars with compositional semantics to speech recognition packages. In *Proceedings of the 19th international conference on Computational linguistics*, pages 1–7, Morristown, NJ, USA, 2002. Association for Computational Linguistics. URL `http://portal.acm.org/citation.cfm?id=1072323`.

Björn Bringert, Robin Cooper, Peter Ljunglöf, and Aarne Ranta. Multimodal Dialogue System Grammars. In *Proceedings of DIALOR'05, Ninth Workshop on the Semantics and Pragmatics of Dialogue*, pages 53–60,

[1] `http://www.cs.chalmers.se/~aarne/GF/`

Nancy, France, 2005. URL http://www.cs.chalmers.se/~bringert/publ/mm-grammars-dialor/mm-grammars-dialor.pdf.

Janusz A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Mathematical theory of Automata*, Volume 12 of MRI Symposia Series, pages 529–561. Polytechnic Press, Polytechnic Institute of Brooklyn, N.Y., 1962.

John Dowding, Beth A. Hockey, Jean M. Gawron, and Christopher Culy. Practical issues in compiling typed unification grammars for speech recognition. In *ACL '01: Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, pages 164–171, Morristown, NJ, USA, 2001. Association for Computational Linguistics. URL http://portal.acm.org/citation.cfm?id=1073034.

Stina Ericsson, Gabriel Amores, Björn Bringert, Håkan Burden, Ann C. Forslund, David Hjelm, Rebecca Jonson, Staffan Larsson, Peter Ljunglöf, Pilar Manchón, David Milward, Guillermo Pérez, and Mikael Sandin. Software illustrating a unified approach to multimodality and multilinguality in the in-home domain. Technical Report 1.6, TALK Project, 2006. URL http://www.talk-project.org/.

Kallirroi Georgila and Oliver Lemon. Programming by Voice: enhancing adaptivity and robustness of spoken dialogue systems. In *BRANDIAL'06, Proceedings of the 10th Workshop on the Semantics and Pragmatics of Dialogue*, pages 199–200, 2006. URL http://www.ling.uni-potsdam.de/brandial/Proceedings/brandial06_georgila_etal.pdf.

Rebecca Jonson. Generating Statistical Language Models from Interpretation Grammars in Dialogue Systems. In *Proceedings of EACL'06*, 2006. URL http://citeseer.ist.psu.edu/jonson06generating.html.

Staffan Larsson. *Issue-based Dialogue Management*. PhD thesis, Göteborg University, 2002.

Oliver Lemon and Xingkun Liu. DUDE: a Dialogue and Understanding Development Environment, mapping Business Process Models to Information State Update dialogue systems. In *EACL 2006, 11st Conference of the European Chapter of the Association for Computational Linguistics*, 2006. URL http://homepages.inf.ed.ac.uk/olemon/dude-final.pdf.

Oliver Lemon, Kallirroi Georgila, David Milward, and Tommy Herbert. Programming Devices and Services. Technical Report 2.3, TALK Project, 2006a. URL http://www.talk-project.org/.

Oliver Lemon, Xingkun Liu, Daniel Shapiro, and Carl Tollander. Hierarchical Reinforcement Learning of Dialogue Policies in a development environment for dialogue systems: REALL-DUDE. In *BRANDIAL'06, Proceedings of*

the 10th Workshop on the Semantics and Pragmatics of Dialogue, pages 185–186, September 2006b. URL http://www.ling.uni-potsdam.de/brandial/Proceedings/brandial06_lemon_etal.pdf.

Peter Ljunglöf. *Expressivity and Complexity of the Grammatical Framework*. PhD thesis, Göteborg University, Göteborg, Sweden, 2004. URL http://www.ling.gu.se/~peb/pubs/p04-PhD-thesis.pdf.

Peter Ljunglöf. Converting Grammatical Framework to Regulus. In *SPEECHGRAM 2007*, 2007a.

Peter Ljunglöf. Personal communication, March 2007b.

Mehryar Mohri and Mark J. Nederhof. Regular Approximation of Context-Free Grammars through Transformation. In Jean C. Junqua and Gertjan van Noord, editors, *Robustness in Language and Speech Technology*, pages 153–163. Kluwer Academic Publishers, Dordrecht, 2001. URL http://www.coli.uni-sb.de/publikationen/softcopies/Mohri:2001:RAC.pdf.

Robert C. Moore. Removing left recursion from context-free grammars. In *Proceedings of the first conference on North American chapter of the Association for Computational Linguistics*, pages 249–255, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. URL http://portal.acm.org/citation.cfm?id=974338.

Robert C. Moore. Using Natural-Language Knowledge Sources in Speech Recognition. In K. M. Ponting, editor, *Computational Models of Speech Pattern Processing*, pages 304–327. Springer, 1999. URL http://research.microsoft.com/users/bobmoore/nato-asi.pdf.

Mark J. Nederhof. Regular Approximation of CFLs: A Grammatical View. In Harry Bunt and Anton Nijholt, editors, *Advances in Probabilistic and other Parsing Technologies*, pages 221–241. Kluwer Academic Publishers, 2000. URL http://www.dcs.st-and.ac.uk/~mjn/publications/2000d.pdf.

Nadine Perera and Aarne Ranta. An Experiment in Dialogue System Localization with the GF Resource Grammar Library. In *SPEECHGRAM 2007*, 2007.

Aarne Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2):145–189, March 2004. ISSN 0956-7968. doi: 10.1017/S0956796803004738. URL http://portal.acm.org/citation.cfm?id=967507.

Aarne Ranta, Ali El Dada, and Janna Khegai. The GF Resource Grammar Library, June 2006. URL http://www.cs.chalmers.se/~aarne/GF/doc/resource.pdf.

Manny Rayner, Beth A. Hockey, and Pierrette Bouillon. *Putting Linguistics into Speech Recognition: The Regulus Grammar Compiler*. CSLI Publications, Ventura Hall, Stanford University, Stanford, CA 94305, USA, July 2006. ISBN 1575865262.

Jessica Villing and Staffan Larsson. Dico: A Multimodal Menu-based In-vehicle Dialogue System. In *BRANDIAL'06, Proceedings of the 10th Workshop on the Semantics and Pragmatics of Dialogue*, pages 187–188, 2006. URL `http://www.ling.uni-potsdam.de/brandial/Proceedings/brandial06_villing_etal.pdf`.

Paper II | Multimodal Dialogue System Grammars

DIALOR 2005, Nancy

Multimodal Dialogue System Grammars*

Björn Bringert, Peter Ljunglöf, Aarne Ranta
Department of Computer Science and Engineering
Chalmers University of Technology
and Göteborg University
{bringert,peb,aarne}@cs.chalmers.se

Robin Cooper
Department of Linguistics
Göteborg University
cooper@ling.gu.se

Abstract

We describe how multimodal grammars for dialogue systems can be written using the Grammatical Framework (GF) formalism. A proof-of-concept dialogue system constructed using these techniques is also presented. The software engineering problem of keeping grammars for different languages, modalities and systems (such as speech recognizers and parsers) in sync is reduced by the formal relationship between the abstract and concrete syntaxes, and by generating equivalent grammars from GF grammars.

1 Introduction

We are interested in building multilingual multimodal grammar-based dialogue systems which are clearly recognisable to users as the same system even if they use the system in different languages or in different domains using different mixes of modalities (e.g. in-house vs in-car, and within the in-house domain with vs without a screen for visual interaction and touch/click input). We wish to be able to guarantee that the functionality of the system is the same under the different conditions.

Our previous experience with building such multilingual dialogue systems is that there is a software engineering problem keeping the linguistic coverage in sync for different languages. This problem is compounded by the fact that for each language it is normally the case that a dialogue system requires more than one grammar, e.g. one grammar for speech recognition and another for

* This project is supported by the EU project TALK (Talk and Look, Tools for Ambient Linguistic Knowledge), IST-507802

interaction with the dialogue manager. Thus multilingual systems become very difficult to develop and maintain.

In this paper we will explain the nature of the Grammatical Framework (GF) and how it may provide us with a solution to this problem. The system is oriented towards the writing of multilingual and multimodal grammars and forces the grammar writer to keep a collection of grammars in sync. It does this by using computer science notions of abstract and concrete syntax. Essentially abstract syntax corresponds to the domain knowledge representation of the system and several concrete syntaxes characterising both natural language representations of the domain and representations in other modalities are related to a single abstract syntax.

GF has a type checker that forces concrete syntaxes to give complete coverage of the abstract syntax and thus will immediately tell the grammar writer if the grammars are not in sync. In addition the framework provides possibilities for converting from one grammar format to another and for combining grammars and extracting sub-grammars from larger grammars.

2 The Grammatical Framework and multilingual grammars

The main idea of Grammatical Framework (GF) is the separation of abstract and concrete syntax. The abstract part of a grammar defines a set of abstract syntactic structures, called abstract terms or trees; and the concrete part defines a relation between abstract structures and concrete structures.

As an example of a GF representation, the following abstract syntax tree represents a possible user input in our example dialogue system.

GoFromTo (PStop Chalmers) (PStop Valand)

The English concrete syntax relates the query to the string

"I want to go from Chalmers to Valand"

The Swedish concrete syntax relates it to the string

"Jag vill åka från Chalmers till Valand"

The strings are generated from the tree in a compositional rule-to-rule fashion. The generation rules are automatically inverted to parsing rules.

The abstract theory of Grammatical Framework (Ranta 2004) is a version of dependent type theory, similar to LF (Harper et al. 1993), ALF (Magnusson and Nordström 1994) and COQ (Coq). What GF adds to the logical framework is the possibility of defining concrete syntax. The expressiveness of the concrete syntax has developed into a functional programming language, similar to a restricted version of programming languages like Haskell (Peyton Jones 2003) and ML (Milner et al. 1997).

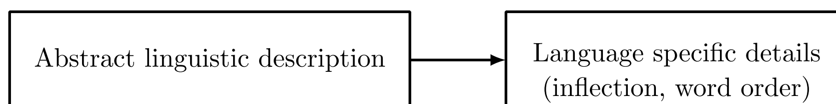


Figure 1. Higher-level language descriptions

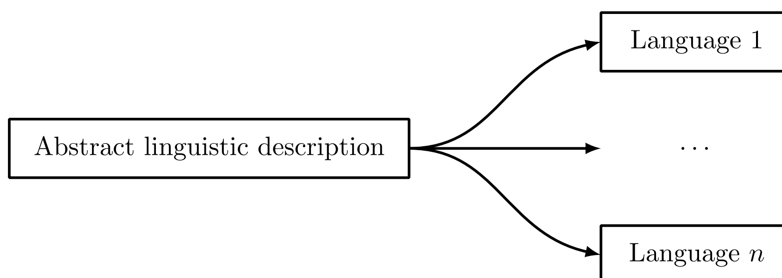


Figure 2. Multilingual grammars

The separation between abstract and concrete syntax was suggested for linguistics in (Curry 1961), using the terms "tectogrammatical" and "phenogrammatical" structure. Since the distinction has not been systematically exploited in many well-known grammar formalisms, let us summarize its main advantages.

Higher-level language descriptions The grammar writer has a greater freedom in describing the syntax for a language. As illustrated in figure 1, when describing the abstract syntax he/she can choose not to take certain language specific details into account, such as inflection and word order. Abstracting away smaller details can make the grammars simpler, both to read and understand, and to create and maintain.

Multilingual grammar writing It is possible to define several different concrete syntax mappings for one particular abstract syntax. The abstract syntax could e.g. give a high-level description of a family of similar languages, and each concrete mapping gives a specific language instance, as shown in figure 2.
This kind of multilingual grammar can be used as a model for interlingual translation between languages. But we do not have to restrict ourselves to only multilingual grammars; different concrete syntaxes can be given for different modalities. As an example, consider a grammar for displaying time table information. We can have one concrete syntax for writing the information as plain text, but we could also present the information in the form of a table output as a LaTeX file or in Excel format, and a third possibility is to output the information in a format suitable for speech synthesis.

Several descriptional levels Having only two descriptional levels is not a restriction; this can be generalized to as many levels as is wanted, by equating

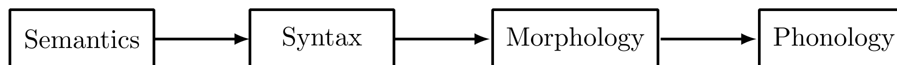


Figure 3. Several descriptional levels

Figure 4. Using resource grammars

the concrete syntax of one grammar level with the abstract syntax of another level. As an example we could have a spoken dialogue system with a semantical, a syntactical, a morphological and a phonological level. As illustrated in figure 3, this system has to define three mappings; *i*) a mapping from semantical descriptions to syntax trees; *ii*) a mapping from syntax trees to sequences of lexical tokens; and *iii*) a mapping from lexical tokens to lists of phonemes.

This formulation makes grammars similar to transducers (Karttunen et al. 1996; Mohri 1997) which are mostly used in morphological analysis, but have been generalized to dialogue systems by (Lager and Kronlid 2004).

Grammar composition A multi-level grammar as described above can be viewed as a "black box", where the intermediate levels are unknown to the user. Then we are back in our first view as a grammar specifying an abstract and a concrete level together with a mapping. In this way we can talk about *grammar composition*, where the composition $G_2 \circ G_1$ of two grammars is possible if the abstract syntax of G_2 is equal to the concrete syntax of G_1.

If the grammar formalism supports this, a composition of several grammars can be pre-compiled into a compact and efficient grammar which doesn't have to mention the intermediate domains and structures. This is the case for e.g. finite state transducers, but also for GF as has been shown by (Ranta 2005).

Resource grammars The possibility of separate compilation of grammar compositions opens up for writing *resource grammars* (Ranta 2005). A resource grammar is a fairly complete linguistic description of a specific language. Many applications do not need the full power of a language, but instead want to use a more well-behaved subset, which is often called a *controlled language*. Now, if we already have a resource grammar, we do not even have to write a concrete syntax for the desired controlled language, but instead we can specify the language by mapping structures in the controlled language into structures in the resource grammar, as shown in figure 4.

3 Extending multilinguality to multimodality

Parallel multimodality *Parallel multimodality* is a straightforward instance of multilinguality. It means that the concrete syntaxes associated with an abstract syntax are not just different natural languages, but different representation modalities, encoded by language-like notations such as graphic representation formalisms. An example of parallel multimodality is given below when a route is described, in parallel, by speech and by a line drawn on a map. Both descriptions convey the full information alone, without support from the other.

This raises the dialogue management issue of whether all information should be presented in all modalities. For example, in the implementation described below all stops are indicated on the graphical presentation of a route whereas in the natural language presentation only stops where the user must change are presented. Because GF permits the suppression of information in concrete syntax, this issue can be treated on the level of grammar instead of dialogue management.

Integrated multimodality *Integrated multimodality* means that one concrete syntax representation is a combination of modalities. For instance, the spoken utterance *"I want to go from here to here"* can be combined with two pointing gestures corresponding to the two *"here"*s. It is the two modalities in combination that convey the full information: the utterance alone or the clicks alone are not enough.

How to define integrated multimodality with a grammar is less obvious than parallel multimodality. In brief, different modality "channels" are stored in different fields of a record, and it is the combination of the different fields that is sent to the dialogue system parser.

4 Proof-of-concept implementation

We have implemented a multimodal route planning system for public transport networks. The example system uses the Göteborg tram/bus network, but it can easily be adapted to other networks. User input is handled by a grammar with integrated speech and map click modalities. The system uses a grammar with parallel speech and map drawing modalities. The user and system grammars are split up into a number of modules in order to simplify reuse and modification.

The system is also multilingual, and can be used in both English and Swedish. For every English concrete module shown below, there is a corresponding Swedish module. The system answers in the same language as the user made the query in.

In addition to the grammars shown below, the application consists of a number of agents which communicate using OAA (Martin et al. 1999). The grammars are used by the Embedded GF Interpreter (Bringert 2005) to parse user input and generate system output.

4.1 Transport network

The transport network is represented by a set of modules which are used in both the query and answer grammars. Since the transport network is described in a separate set of modules, the Göteborg transport network may be replaced easily. We use **cat** judgements to declare categories in the abstract syntax.

```
abstract Transport = {
  cat
    Stop;
}
```

The Göteborg transport network grammar extends the generic grammar with constructors for the stops. Constructors for abstract syntax terms are declared using **fun** judgements.

```
abstract Gbg = Transport ** {
  fun
    Angered : Stop;
    AxelDahlstromsTorg : Stop;
    Bergsjon : Stop;
    ...
}
```

4.2 Multimodal input

User input is done with integrated speech and click modalities. The user may use speech only, or speech combined with clicks on the map. Clicks are expected when the user makes a query containing *"here"*.

Common declarations The QueryBase module contains declarations common to all input modalities. The Query category is used to represent the sequentialization of the multimodal input into a single value. The Input category contains the actual user queries, which will have multimodal representations. The Click category is also declared here, since it is used by both the click modality and the speech modality, as shown below.

```
abstract QueryBase = {
  cat
    Query;
    Input;
    Click;
  fun
    QInput : Input → Query;
}
```

Since QueryBase is language neutral and common to the different modalities, it has a single concrete syntax. In a concrete module, **lincat** judgements are

used to declare the linearization type of a category, i.e. the type of the concrete representations of values in the category. Note that different categories may have different linearization types. The concrete representation of abstract syntax terms is declared by **lin** judgements for each constructor in the abstract syntax.

Values in the Input category, which are intended to be multimodal, have records with one field per modality as their concrete representation. The *s1* field contains the speech input, and the *s2* field contains the click input. Terms constructed using the QInput constructor, that is sequentialized multimodal queries, are represented as the concatenation of the representations of the individual modalities, separated by a semicolon.

```
concrete QueryBaseCnc of QueryBase = {
  lincat
    Query = {s : Str};
    Input  = {s1 : Str; s2 : Str};
    Click  = {s : Str};
  lin
    QInput i = {s = i.s1 ++ ";" ++ i.s2};
}
```

Click modality Click terms contain a list of stops that the click might refer to:

```
abstract Click = QueryBase ** {
  cat
    StopList;
  fun
    CStops     : StopList → Click;
    NoStop     : StopList;
    OneStop    : String → StopList;
    ManyStops : String → StopList → StopList;
}
```

The same concrete syntax is used for clicks in all languages:

```
concrete ClickCnc of Click = QueryBaseCnc ** {
  lincat
    StopList = {s : Str};
  lin
    CStops xs        = {s = "[" ++ xs.s ++ "]"};
    NoStop           = {s = ""};
    OneStop x        = {s = x.s};
    ManyStops x xs = {s = x.s ++ "," ++ xs.s};
}
```

Speech modality The Query module adds basic user queries and a way to use a click to indicate a place.

```
abstract Query = QueryBase ** {
  cat
    Place;
  fun
    GoFromTo : Place → Place → Input;
    GoToFrom : Place → Place → Input;
    PClick   : Click → Place;
}
```

This module has a concrete syntax using English speech. Like terms in the Query category, Place terms are linearized to records with two fields, one for each modality.

```
concrete QueryEng of Query = QueryBaseCnc ** {
  lincat
    Place = {s1 : Str; s2 : Str};
  lin
    GoFromTo x y = {
      s1 = ["i want to go from"] ++ x.s1 ++ "to" ++ y.s1;
      s2 = x.s2 ++ y.s2
    };
    GoToFrom x y = {
      s1 = ["i want to go to"] ++ x.s1 ++ "from" ++ y.s1;
      s2 = x.s2 ++ y.s2
    };
    PClick c = {s1 = "here"; s2 = c.s};
}
```

Indexicality To refer to her current location, the user can use *"here"* without a click, or omit either origin or destination. The system is assumed to know where the user is located. Since *"here"* may be used with or without a click, inputs with two occurrences of *"here"* and only one click are ambiguous. A query might also be ambiguous even if it can be parsed unambiguously, since one click can correspond to multiple stops when the stops are close to each other on the map.

These are the abstract syntax declarations for this feature (in the Query module):

```
fun
  PHere    : Place;
  ComeFrom : Place → Input;
  GoTo     : Place → Input;
```

The English concrete syntax for this is added to the QueryEng module. Note that the click (*s2*) field of the linearization of an indexical *"here"* is empty, since there is no click.

```
lin
  PHere = {s1 = "here"; s2 = []};
  ComeFrom x = {
    s1 = ["i want to come from"] ++ x.s1;
    s2 = x.s2
  };
  GoTo x = {
    s1 = ["i want to go to"] ++ x.s1;
    s2 = x.s2
  };
```

Tying it all together The TransportQuery module ties together the transport network, speech modality and click modality modules.

```
abstract TransportQuery = Transport, Query, Click ** {
  fun
    PStop : Stop → Place;
}
```

4.3 Multimodal output

The system's answers to the user's queries are presented with speech and drawings on the map. This is an example of parallel multimodality as the speech and the map drawings are independent. The information presented in the two modalities is however not identical, as the spoken output only contains information about where to change trams/buses. The map output shows the entire path, including intermediate stops.

Abstract syntax for routes The abstract syntax for answers (routes) contains the information needed by all the concrete syntaxes. All concrete syntaxes might not use all of the information. A route is a non-empty list of legs, and a leg consists of a line and a list of at least two stops.

```
abstract Route = Transport ** {
  cat
    Route;
    Leg;
    Line;
    Stops;
  fun
    Then      : Leg → Route → Route;
```

```
    OneLeg     : Leg → Route;
    LineLeg    : Line → Stops → Leg;
    NamedLine  : String → Line;
    ConsStop   : Stop → Stops → Stops;
    TwoStops   : Stop → Stop → Stops;
}
```

Concrete syntax for drawing routes The map drawing language contains sequences of labeled edges to be drawn on the map. The string

> drawEdge (6, [Chalmers, Vasaplatsen]); drawEdge (2, [Vasaplatsen, Gronsakstorget, Brunnsparken]);

is an example of a string in the map drawing language described by the RouteMap concrete syntax. The TransportLabels module extended by this module is a simple concrete syntax for stops.

```
concrete RouteMap of Route = TransportLabels ** {
  lincat
    Route, Leg, Line, Stops = {s : Str};
  lin
    Then l r       = {s = l.s ++ ";" ++ r.s};
    OneLeg l       = {s = l.s ++ ";"};
    LineLeg l ss   =
      {s = "drawEdge" ++ "(" ++ l.s ++ "," ++ "[" ++ ss.s ++ "]" ++ ")"};
    NamedLine n  = {s = n.s};
    ConsStop s ss = {s = s.s ++ "," ++ ss.s};
    TwoStops x y = {s = x.s ++ "," ++ y.s};
}
```

English concrete syntax for routes In the English concrete syntax we wish to list only the first and last stops of each leg of the route. The TransportNames module gives English representations of the stop names by replacing all non-English letters with the corresponding English ones in order to give the speech recognizer a fair chance.

```
concrete RouteEng of Route = TransportNames ** {
  lincat
    Route, Leg, Line = {s : Str};
    Stops = {start : Str; end : Str};
  lin
    Then l r        = {s = l.s ++ "." ++ r.s};
    OneLeg l        = {s = l.s ++ "."};
    LineLeg l ss    =
      {s = "Take" ++ l.s ++ "from" ++ ss.start ++ "to" ++ ss.end};
```

```
    NamedLine n      = {s = n.s};
    ConsStop s ss    = {start = s.s; end = ss.end};
    TwoStops s1 s2 = {start = s1.s; end = s2.s};
}
```

5 Related Work

Johnston (1998) describes an approach to multimodal parsing where chart parsing is extended to multiple dimensions and unification is used to integrate information from different modalities. The approach described in this paper achieves a similar result by using records along with the existing unification mechanism for resolving discontinuous constituents. The main advantages of our approach are that it supports both parsing and generation, and that it does not require extending the existing formalism.

6 Conclusion

GF provides a solution to the problems named in the introduction to this paper. Abstract syntax can be used to characterise the linguistic functionality of a system in an abstract language and modality independent way. The system forces the programmer to define concrete syntaxes which completely cover the abstract syntax. In this way, the system forces the programmer to keep all the concrete syntaxes in sync. In addition, since GF is oriented towards creating grammars from other grammars, our philosophy is that it should not be necessary for a grammar writer to have to create by hand any equivalent grammars in different formats. For example, if the grammar for the speech recogniser is to be the same as that used for interaction with dialogue management but the grammars are needed in different formats, then there should be a compiler which takes the grammar from one format to the other. Thus, for example, we have a compiler which converts a GF grammar to Nuance's format for speech recognition grammars. The idea of generating context-free speech recognition grammars from grammars in a higher-level formalism has been described by Dowding et al. (2001), and implemented in the Regulus system (Rayner et al. 2003).

Another reason for using GF grammars has to do with the use of resource grammars and cascades of levels of representation as described in section 2. This allows for the hiding of grammatical detail from language and the precise implementation of modal interaction for other modalities. This enables the dialogue system developer to reuse previous grammar or modal interaction implementations without herself having to reprogram the details for each new dialogue system. Thus the dialogue engineer need not be a grammar engineer or an expert in multimodal interfaces.

References

Björn Bringert. Embedded Grammars. Master's thesis, Chalmers University of Technology, Göteborg, Sweden, February 2005. URL http://www.cs.chalmers.se/~bringert/publ/exjobb/embedded-grammars.pdf.

Coq. *The Coq Proof Assistant Reference Manual*. The Coq Development Team, 1999. Available at http://pauillac.inria.fr/coq/

Haskell B. Curry. Some Logical Aspects of Grammatical Structure. In Roman O. Jakobson, editor, *Structure of Language and its Mathematical Aspects, volume 12 of Symposia on Applied Mathematics*, pages 56–68. American Mathematical Society, Providence, 1961.

John Dowding, Beth A. Hockey, Jean M. Gawron, and Christopher Culy. Practical issues in compiling typed unification grammars for speech recognition. In *ACL '01: Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, pages 164–171, Morristown, NJ, USA, 2001. Association for Computational Linguistics. URL http://portal.acm.org/citation.cfm?id=1073034.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993. ISSN 0004-5411. doi: 10.1145/138027.138060. URL http://portal.acm.org/citation.cfm?id=138060.

Michael Johnston. Unification-based multimodal parsing. In *Proceedings of the 36th annual meeting on Association for Computational Linguistics*, pages 624–630, Morristown, NJ, USA, 1998. Association for Computational Linguistics. URL http://portal.acm.org/citation.cfm?id=980949.

Lauri Karttunen, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328, 1996.

Torbjörn Lager and Fredrik Kronlid. The Current platform: Building conversational agents in Oz. In *2nd International Mozart/Oz Conference*, October 2004.

Lena Magnusson and Bengt Nordström. *The Alf proof editor and its proof engine*, volume 806 of *Lecture Notes in Computer Science*, pages 213–237. Springer, 1994. doi: 10.1007/3-540-58085-9_78. URL http://dx.doi.org/10.1007/3-540-58085-9_78.

David L. Martin, Adam J. Cheyer, and Douglas B. Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1-2):91–128, 1999. URL http://www.scopus.com/scopus/record/display.url?view=extended&origin=resultslist&eid=2-s2.0-0032805927.

Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML - Revised*. The MIT Press, May 1997. ISBN 0262631814.

Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–312, 1997.

Simon Peyton Jones. The Haskell 98 Language. *Journal of Functional Programming*, 13(1), 2003.

A. Ranta. Modular Grammar Engineering in GF. *Research in Language and Computation*, 2005. To appear.

Aarne Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2):145–189, March 2004. ISSN 0956-7968. doi: 10.1017/S0956796803004738. URL `http://portal.acm.org/citation.cfm?id=967507`.

Manny Rayner, Beth A. Hockey, and John Dowding. An open source environment for compiling typed unification grammars into speech recognisers. In *EACL '03: Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics*, pages 223–226, Morristown, NJ, USA, 2003. Association for Computational Linguistics. ISBN 1111567890. URL `http://portal.acm.org/citation.cfm?id=1067790`.

Paper III | A Pattern for Almost Compositional Functions

ICFP 2006, Portland

A Pattern for Almost Compositional Functions

Björn Bringert and Aarne Ranta
Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
{bringert,aarne}@cs.chalmers.se

Abstract

This paper introduces a pattern for almost compositional functions over recursive data types, and over families of mutually recursive data types. Here "almost compositional" means that for a number of the constructors in the type(s), the result of the function depends only on the constructor and the results of calling the function on the constructor's arguments. The pattern consists of a generic part constructed once for each data type or family of data types, and a task-specific part. The generic part contains the code for the predictable compositional cases, leaving the interesting work to the task-specific part. Examples of the pattern implemented in dependent type theory with inductive families, in Haskell with generalized algebraic data types and rank-2 polymorphism, and in Java using a variant of the Visitor design pattern are given. The relationship to the "Scrap Your Boilerplate" approach to generic programming, and to general tree types in dependent type theory are also investigated.

1 Introduction

This paper addresses the issue of repetitive code in operations on rich data structures. To give concrete examples of what we would like to be able to do, we start by giving some motivating problems.

1.1 Some motivating problems

Suppose that you have an abstract syntax definition with many syntactic types such as statement, expression, and variable.

1. Write a function that renames all variables in a program by prepending an underscore to their names. Do this with a case expression that has just two branches: one for the variables, another for the rest.
2. Write a function that constructs a symbol table containing all variables declared in a program, and the type of each variable. Do this with a case expression that has just two branches: one for declarations, another for the rest.

3. Write a function which gives fresh names to all variables in a program. Do this using only three cases: one for variable bindings, another for variable uses, and a third for the rest.

One problem when writing recursive functions which need to traverse rich data structures is that the straightforward way to write them involves large amounts of traversal code which tends to be repeated in each function. There are several problems with this:

- The repeated traversals are probably implemented using copy-and-paste or retyping, both of which are error-prone and can lead to maintenance problems.
- When we add a constructor to the data type, we need to change all functions that traverse the data type, many of which may not need any specific behavior for the new constructor.
- Repeated traversal code obscures the interesting cases where the functions do their real work.
- The need for complete traversal code for the whole family of data types in every function could encourage a less modular programming style where multiple operations are collected in a single function.

1.2 The solution

The pattern which we present in this paper allows the programmer to solve problems such as the above in a (hopefully) intuitive way. First we write the traversal code once and for all for our data type or family of data types. We then reuse this component to succinctly express the operations which we want to define.

1.3 Article overview

We first present the simple case of a single recursive algebraic data type, and show examples of using the pattern for this case, with examples in plain Haskell 98 (Peyton Jones 2003a). After that, we generalize this to the more complex case of a family of data types, and show how the pattern can be used in dependent type theory (Martin-Löf 1984; Nordström et al. 1990) with inductive families (Dybjer 1994) and in Haskell with generalized algebraic data types (Peyton Jones et al. 2006; Augustsson and Petersson 1994) and rank-2 polymorphism. We then prove some properties of our compositional operations, using the laws for applicative functors (McBride and Paterson 2007). We go on to express the pattern in Java (Gosling et al. 2005) with parametric polymorphism (Bracha et al. 1998), using a variant of the Visitor design pattern (Gamma et al. 1995). In the following section, we briefly describe some tools which can be used to automate the process of writing the necessary support code for a given

data type. Finally, we discuss some related work in generic programming, type theory, object-oriented programming and compiler construction, and provide some conclusions.

2 Abstract Syntax and Algebraic Data Types

Algebraic data types provide a natural way to implement the abstract syntax in a compiler. To give an example, the following Haskell type defines the abstract syntax of lambda calculus with abstractions, applications, and variables. For more information about using algebraic data types to represent abstract syntax for programming languages, see for example Appel's (1997) text books on compiler construction.

```
data Exp = EAbs String Exp | EApp Exp Exp | EVar String
```

Pattern matching is the technique for defining functions on algebraic data types. These functions are typically recursive. An example is a function that renames all the variables in an expression by prepending an underscore to their names:

```
rename :: Exp → Exp
rename e = case e of
             EAbs x b → EAbs ("_" ++ x) (rename b)
             EApp c a → EApp (rename c) (rename a)
             EVar x   → EVar ("_" ++ x)
```

3 Compositional Functions

Many functions used in compilers are *compositional*, in the sense that the result for a complex argument is constructed from the results for its parts. The *rename* function is an example of this. The essence of compositional functions is defined by the following higher-order function:

```
composOp :: (Exp → Exp) → Exp → Exp
composOp f e = case e of
                 EAbs x b → EAbs x (f b)
                 EApp c a → EApp (f c) (f a)
                 _        → e
```

Its power lies in that it can be used when defining other functions, to take care of cases which are just compositional. Such is the EApp case in *rename*, which we thus omit by writing:

```
rename :: Exp → Exp
rename e = case e of
```

```
    EAbs x b → EAbs ("_" ++ x) (rename b)
    EVar x   → EVar ("_" ++ x)
    _        → composOp rename e
```

In general, an abstract syntax has many more constructors, and this pattern saves much more work. For instance, in the implementation of GF (Ranta 2004), the Exp type has 30 constructors, and *composOp* is used in more than 20 functions, typically covering 90 % of all cases.

A major restriction of *composOp* is that its return type is Exp. How do we use it if we want to return something else? If we simply want to compute some result using the abstract syntax tree, without modifying the tree, we can use *composFold*:

```
composFold :: Monoid o ⇒ (Exp → o) → Exp → o
composFold f e = case e of
                   EAbs x b → f b
                   EApp c a → f c ⊕ f a
                   _        → ∅
```

This function takes an argument which maps terms to a monoid, and combines the results. The Monoid class requires an identity element ∅, which we return for leaf nodes, and an associative operation (⊕), which we use to combine results from nodes with more than one child.

```
class Monoid a where
  ∅ :: a
  (⊕) :: a → a → a
```

Using *composFold* we can now, for example, write a function which gets the names of all free variables in an expression:

```
free :: Exp → Set String
free e = case e of
           EAbs x b → free b \ {x}
           EVar x   → {x}
           _        → composFold free e
```

This example uses a Set type with the operations \, {·}, ∅ and ∪, with a Monoid instance such that ∅ = ∅ and (⊕) = ∪.

3.1 Monadic compositional functions

When defining a compiler in Haskell, it is convenient to use monads instead of plain functions, to deal with errors, state, etc. To this end, we generalize *composOp* to a monadic variant:

```
composM :: Monad m ⇒ (Exp → m Exp) → Exp → m Exp
composM f e = case e of
```

```
EAbs x b → return EAbs `ap` return x `ap` f b
EApp c a → return EApp `ap` f c `ap` f a
_        → return e
```

Here we are using the Monad type class and the *ap* function from the Haskell 98 Libraries (Peyton Jones 2003b):

```
class Monad m where
  (>>=)  :: m a → (a → m b) → m b
  return :: a → m a

ap :: Monad m ⇒ m (a → b) → m a → m b
ap mf mx = mf >>= λf → mx >>= λx → return (f x)
```

If we want to maintain some state across the computation over the tree, we can use *composM* with a state monad (Jones 1995). In the example below, we will use a state monad State with these operations:

```
readState  :: State s s
writeState :: s → State s ()
runState   :: s → State s a → (a, s)
```

Now we can, for example, write a function that gives fresh names of the form `"_n"`, where n is an integer, to all bound variables in an expression. Here the state is an infinite supply of fresh variable names, and we pass a table of the new names for the bound variables to the recursive calls.

```
fresh :: Exp → Exp
fresh = fst ∘ runState names ∘ f []
  where names = ["_" ++ show n | n ← [0 ..]]
        f :: [(String, String)] → Exp → State [String] Exp
        f vs t = case t of
                   EAbs x b → do x′ : ns ← readState
                                 writeState ns
                                 let vs′ = (x, x′) : vs
                                 return (EAbs x′) `ap` f vs′ b
                   EVar x   → do let x′ = lookup′ x x vs
                                 return (EVar x′)
                   _        → composM (f vs) t
lookup′ :: Eq a ⇒ b → a → [(a, b)] → b
lookup′ def _ []             = def
lookup′ def k ((x, y) : xs) = if x ≡ k then y else lookup′ def k xs
```

3.2 Generalizing *composOp*, *composM* and *composFold*

McBride and Paterson (McBride and Paterson 2007) introduce *applicative functors*, which generalize monads. An applicative functor has two operations, *pure* and ⊛, corresponding to the *return* and *ap* operations of a Monad.

```
class Applicative f where
  pure :: a → f a
  (⊛)  :: f (a → b) → f a → f b
```

Since the *composM* function only uses *return* and *ap*, it actually works on all applicative functors, not just on monads. We call this generalized version *compos*:

```
compos :: Applicative f ⇒ (Exp → f Exp) → Exp → f Exp
compos f e = case e of
               EAbs x b → pure EAbs ⊛ pure x ⊛ f b
               EApp g h → pure EApp ⊛ f g ⊛ f h
               _        → pure e
```

By using wrapper types with appropriate Applicative instances, we can now define *composOp*, *composM* and *composFold* in terms of *compos*:

```
composOp :: (Exp → Exp) → Exp → Exp
composOp f = runIdentity ∘ compos (Identity ∘ f)

newtype Identity a = Identity {runIdentity :: a}

instance Applicative Identity where
  pure                    = Identity
  Identity f ⊛ Identity x = Identity (f x)

composM :: Monad m ⇒ (Exp → m Exp) → Exp → m Exp
composM f = unwrapMonad ∘ compos (WrapMonad ∘ f)

newtype WrappedMonad m a = WrapMonad {unwrapMonad :: m a}

instance Monad m ⇒ Applicative (WrappedMonad m) where
  pure = WrapMonad ∘ return
  WrapMonad f ⊛ WrapMonad v = WrapMonad (f `ap` v)

composFold :: Monoid o ⇒ (Exp → o) → Exp → o
composFold f = getConst ∘ compos (Const ∘ f)

newtype Const a b = Const {getConst :: a}

instance Monoid m ⇒ Applicative (Const m) where
  pure _ = Const ∅
  Const f ⊛ Const v = Const (f ⊕ v)
```

Further operators, such as *composM_* below can be defined by using other wrapper types.

```
composM_ :: Monad m ⇒ (Exp → m ()) → Exp → m ()
composM_ f = unwrapMonad_ ∘ composFold (WrapMonad_ ∘ f)

newtype WrappedMonad_ m = WrapMonad_ {unwrapMonad_ :: m ()}

instance Monad m ⇒ Monoid (WrappedMonad_ m) where
  ∅ = WrapMonad_ (return ())
  WrapMonad_ x ⊕ WrapMonad_ y = WrapMonad_ (x ≫ y)
```

4 Systems of Data Types

4.1 Several algebraic data types

For many languages, the abstract syntax is not just one data type, but many, which are often defined by mutual induction. An example is the following simple imperative language with statements, expressions, variables, and types. In this language, statements that return values (for example assignments or maybe blocks that end with a return statement) can be used as expressions.

```
data Stm = SDecl Typ Var | SAss Var Exp | SBlock [Stm] | SReturn Exp
data Exp = EStm Stm | EAdd Exp Exp | EVar Var | EInt Int
data Var = V String
data Typ = TInt | TFloat
```

Now we cannot any longer easily define general *composOp* functions, as some of the recursive calls must be done on terms which have different types than the value on which the function was called. Implementing operations such as α-conversion on this kind of family of data types quickly becomes very laborious.

4.2 Categories and trees

An alternative to separate mutual data types for abstract syntax is to define just one type Tree, whose constructors take Trees as arguments:

```
data Tree = SDecl   Tree Tree
          | SAss    Tree Tree
          | SBlock  [Tree]
          | SReturn Tree
          | EStm Tree
          | EAdd Tree Tree
          | EVar  Tree
          | EInt  Int
          | V String
          | TInt
          | TFloat
```

This is essentially the representation one would use in a dynamically typed language. It does not, however, constrain the combinations enough for our liking: there are many Trees that are even syntactically nonsense.

A solution to this problem is provided by dependent types (Martin-Löf 1984; Nordström et al. 1990). Instead of a constant type Tree, we define an **inductive family** (Dybjer 1994) Tree c, indexed by a **category** c. The category is just a label to distinguish between different types of trees. We must now leave standard Haskell and use a Haskell-like language with dependent types and inductive families. Agda (Coquand 2000) is one such language. What one would define in Agda is an enumerated type:

```
data Cat = Stm | Exp | Var | Typ
```

followed by an **idata** (inductive data type, or in this case an inductive family of data types) definition of Tree, indexed on Cat. We omit the Agda definitions of the Tree family and the *compos* function as they are virtually identical to the Haskell versions shown below, except that in Agda the index for Tree is a value of type Cat, whereas in Haskell the index is a dummy data type.

We can also do our exercise with the limited form of dependent types provided by Haskell since GHC 6.4: **Generalized Algebraic Data Types** (GADTs) (Peyton Jones et al. 2006; Augustsson and Petersson 1994). We cannot quite define a *type* of categories, but we can define a set of dummy data types:

```
data Stm
data Exp
data Var
data Typ
```

To define the inductive family of trees, we write, in this extension of Haskell:

```
data Tree :: * → * where
  SDecl   :: Tree Typ → Tree Var → Tree Stm
  SAss    :: Tree Var → Tree Exp → Tree Stm
  SBlock  :: [Tree Stm] → Tree Stm
  SReturn :: Tree Exp → Tree Stm
  EStm    :: Tree Stm → Tree Exp
  EAdd    :: Tree Exp → Tree Exp → Tree Exp
  EVar    :: Tree Var → Tree Exp
  EInt    :: Int → Tree Exp
  V       :: String → Tree Var
  TInt    :: Tree Typ
  TFloat  :: Tree Typ
```

In Haskell we cannot restrict the types used as indices in the Tree family, which makes it entirely possible to construct types such as Tree String. However, since there are no constructors of this type, ⊥ is the only element in it.

4.3 Compositional operations

The power of inductive families is shown in the definition of the function *compos*. We now define it simultaneously for the whole syntax, and can then use it to define tree-traversing programs concisely.

```
compos :: Applicative f ⇒ (∀a. Tree a → f (Tree a)) → Tree c → f (Tree c)
compos f t = case t of
               SDecl x y  → pure SDecl  ⊛ f x ⊛ f y
               SAss x y   → pure SAss   ⊛ f x ⊛ f y
               SBlock xs  → pure SBlock ⊛ traverse f xs
```

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Compos}\ t\ \textbf{where} \\
\quad \mathit{compos} :: \mathsf{Applicative}\ f \Rightarrow (\forall a.\ t\ a \rightarrow f\ (t\ a)) \rightarrow t\ c \rightarrow f\ (t\ c) \\
\mathit{composOp} :: \mathsf{Compos}\ t \Rightarrow (\forall a.\ t\ a \rightarrow t\ a) \rightarrow t\ c \rightarrow t\ c \\
\mathit{composOp}\ f = \mathit{runIdentity} \circ \mathit{compos}\ (\mathsf{Identity} \circ f) \\
\mathit{composFold} :: (\mathsf{Monoid}\ o, \mathsf{Compos}\ t) \Rightarrow (\forall a.\ t\ a \rightarrow o) \rightarrow t\ c \rightarrow o \\
\mathit{composFold}\ f = \mathit{getConst} \circ \mathit{compos}\ (\mathsf{Const} \circ f) \\
\mathit{composM} :: (\mathsf{Compos}\ t, \mathsf{Monad}\ m) \Rightarrow (\forall a.\ t\ a \rightarrow m\ (t\ a)) \rightarrow t\ c \rightarrow m\ (t\ c) \\
\mathit{composM}\ f = \mathit{unwrapMonad} \circ \mathit{compos}\ (\mathsf{WrapMonad} \circ f) \\
\mathit{composM_} :: (\mathsf{Compos}\ t, \mathsf{Monad}\ m) \Rightarrow (\forall a.\ t\ a \rightarrow m\ ()) \rightarrow t\ c \rightarrow m\ () \\
\mathit{composM_}\ f = \mathit{unwrapMonad_} \circ \mathit{composFold}\ (\mathsf{WrapMonad_} \circ f)
\end{array}
$$

Figure 1. The ComposOp module.

$$
\begin{array}{lll}
\mathsf{SReturn}\ x & \rightarrow \mathit{pure}\ \mathsf{SReturn} & \circledast f\ x \\
\mathsf{EAdd}\ x\ y & \rightarrow \mathit{pure}\ \mathsf{EAdd} & \circledast f\ x \circledast f\ y \\
\mathsf{EStm}\ x & \rightarrow \mathit{pure}\ \mathsf{EStm} & \circledast f\ x \\
\mathsf{EVar}\ x & \rightarrow \mathit{pure}\ \mathsf{EVar} & \circledast f\ x \\
_ & \rightarrow \mathit{pure}\ t &
\end{array}
$$

The first *compos*, the function to apply to the subtrees, is now a polymorphic function, since it is applied to subtrees of different types. The argument to the SBlock constructor is a list of statements, which we handle by visiting the list elements from left to right, using the *traverse* function (McBride and Paterson 2007), which generalized *mapM*:

$$
\begin{array}{l}
\mathit{traverse} :: \mathsf{Applicative}\ f \Rightarrow (a \rightarrow f\ b) \rightarrow [a] \rightarrow f\ [b] \\
\mathit{traverse}\ f\ [\,] \qquad\quad = \mathit{pure}\ [\,] \\
\mathit{traverse}\ f\ (x : xs) = \mathit{pure}\ (:) \circledast f\ x \circledast \mathit{traverse}\ f\ xs
\end{array}
$$

The other *compos** functions are special cases of *compos* in the same way as before.

4.4 A library of compositional operations

In order to provide generic implementations of the different functions, we overload *compos* and define the other operations in terms of it. The code for this is shown in Figure 1.

4.5 Migrating existing programs

Replacing a family of data types with a GADT does not change the appearance of the expressions and patterns in the syntax tree types. However, the types now have the form Tree c for some c. If we want, we can give the dummy types names other than those of the original categories, for example Stm_, Exp_,Var_, and

Typ_, and use type synonyms to make the types also look like they did when we had multiple data types:

```
type Stm = Tree Stm_
type Exp = Tree Exp_
type Var = Tree Var_
type Typ = Tree Typ_
```

This allows us to modify existing programs to switch from a family of data types to a GADT, simply by replacing the abstract syntax type definitions. All existing functions remain valid with the new abstract syntax definition, which makes it possible to take advantage of our operators when writing new functions, without being forced to change any existing ones.

4.6 Examples

Example: Rename variables

It would be very laborious to define a renaming function for the original Haskell definition with separate data types (as shown in Section 4.1). But now it is easy:

```
rename :: Tree c → Tree c
rename t = case t of
             V x → V ("_" ++ x)
             _   → composOp rename t
```

Example: Warnings for assignments

To encourage pure functionality, this function sounds the bell each time an assignment occurs. Since we are not interested in the return value of the function, but only in its IO outputs, we use the function *composM_* (like *composM* but without a tree result, see Figure 1 for its definition).

```
warnAssign :: Tree c → IO ()
warnAssign t = case t of
                 SAss _ _ → putChar (chr 7)
                 _        → composM_ warnAssign t
```

Example: Symbol table construction

This function constructs a variable symbol table by folding over the syntax tree. Once again, the return value is of no interest. This function uses the Monoid instance for lists, where the associative operation is ++, and the identity element is [].

```
symbols :: Tree c → [(Tree Var, Tree Typ)]
symbols t = case t of
              SDecl typ var → [(var, typ)]
              _             → composFold symbols t
```

Example: Constant folding

We want to replace additions of constants by their result. Here is a first attempt:

```
constFold :: Tree c → Tree c
constFold e = case e of
                EAdd (EInt x) (EInt y) → EInt (x + y)
                _                      → composOp constFold e
```

This works for simple cases, but what about for example $1 + (2 + 3)$? This is an addition of constants, but is not matched by our pattern above. We have to look at the results of the recursive calls:

```
constFold' :: Tree c → Tree c
constFold' e = case e of
                 EAdd x y → case (constFold' x, constFold' y) of
                              (EInt n, EInt m) → EInt (n + m)
                              (x', y')         → EAdd x' y'
                 _        → composOp constFold' e
```

This illustrates a common pattern used when the recursive calls can introduce terms which we want to handle.

Example: Syntactic sugar

This example shows how easy it is to add syntax constructs as syntactic sugar, i.e. syntactic constructs that can be eliminated. Suppose that you want to add increment statements. This means a new branch in the definition of Tree c from Section 4.2:

```
SIncr :: Tree Var → Tree Stm
```

Increments are eliminated by translation to assignments as follows:

```
elimIncr :: Tree c → Tree c
elimIncr t = case t of
               SIncr v → SAss v (EAdd (EVar v) (EInt 1))
               _       → composOp elimIncr t
```

4.7 Properties of compositional operations

The following laws hold for our definitions of the *compos** functions:

Identity 1 $compos\ pure = pure$
Identity 2 $composOp\ id = id$
Identity 3 $composFold\ (\lambda_ \rightarrow \emptyset) = \lambda_ \rightarrow \emptyset$

In the proofs below, we will make use of the laws for applicative functors (McBride and Paterson 2007):

Identity $pure\ id \circledast u = u$
Composition $pure\ (\circ) \circledast u \circledast v \circledast w = u \circledast (v \circledast w)$
Homomorphism $pure\ f \circledast pure\ x = pure\ (f\ x)$
Interchange $u \circledast pure\ x = pure\ (\lambda f \rightarrow f\ x) \circledast u$

Below, T refers to some type for which we have defined *compos* according to the scheme exemplified in Section 4.3. For example, we would like *compos* not to modify the tree on its own, i.e. that:

Theorem 1. *For all total values* $t :: \mathsf{T}$*,* $compos\ pure\ t = pure\ t$*.*

Proof. The *compos* function has the general form:

$$
\begin{aligned}
compos\ f\ t = \textbf{case}\ t\ \textbf{of}& \\
&\mathsf{C}\ x_1 \ldots x_n \rightarrow pure\ \mathsf{C} \circledast g_1\ x_1 \circledast \ldots \circledast g_n\ x_n \\
&\ldots \\
&_ \qquad\quad \rightarrow pure\ t
\end{aligned}
$$

where each g_i is either *pure*, f, or *traverse* f, depending on the type of x_i. Since $f = pure$ in the case that we are reasoning about, the functions $g_1 \ldots g_n$ are either *pure* or *traverse pure*.

Lemma 1. *For all total* $ys :: [a]$*,* $traverse\ pure\ ys = pure\ ys$*.*

Proof. By induction on the structure of *ys*, using the homomorphism law for applicative functors. □

Using Lemma 1, we see that all the $g_1 \ldots g_n$ functions are *pure*. Thus, the constructor cases all have the form:

$$\mathsf{C}\ x_1 \ldots x_n \rightarrow pure\ \mathsf{C} \circledast pure\ x_1 \circledast \ldots \circledast pure\ x_n$$

By repeated use of the homomorphism law for applicative functors, we have that

$$pure\ \mathsf{C} \circledast pure\ x_1 \circledast \ldots \circledast pure\ x_n = pure\ (\mathsf{C}\ x_1 \ldots x_n)$$

Thus, for all total $t : \mathsf{T}$, $compos\ pure\ t = pure\ t$. With the definitions of *composOp* and *composFold* given above, Identity 2 and Identity 3 follow straightforwardly from Identity 1. □

It should also be possible to perform formal reasoning about our compositional operations using dependent type theory with tree sets, as discussed in Section 7.2.

5 Almost Compositional Functions and the Visitor Design Pattern

The Visitor design pattern (Gamma et al. 1995) is a pattern used in object-oriented programming to define an operation for each of the concrete elements of an object hierarchy. We will show how an adaptation of the Visitor pattern can be used to define almost compositional functions in object-oriented languages, in a manner quite similar to that shown above for languages with algebraic data types and pattern matching.

First we present the object hierarchies corresponding to the algebraic data types. Each object hierarchy has a generic Visitor interface. We then show a concrete visitor that corresponds to the *composOp* function. Our examples are written in Java (Gosling et al. 2005) with parametric polymorphism (Bracha et al. 1998).

5.1 Abstract syntax representation

We use a standard encoding of abstract syntax trees in Java (Appel 2002), along with the support code for a type-parametrized version of the Visitor design pattern. For each algebraic data type in the Haskell version (as shown in Section 4.1), we have an abstract base class in the Java representation:

```
public abstract class Stm {
  public abstract<R, A>R accept (Visitor<R, A>v, A arg);
  public interface Visitor<R, A> {
    public R visit (SDecl p, A arg);
    public R visit (SAss p, A arg);
    public R visit (SBlock p, A arg);
    public R visit (SReturn p, A arg);
    public R visit (SInc p, A arg);
  }
}
```

The base class contains an interface for visitors with methods for visiting each of the inheriting classes. It also specifies that each inheriting class must have a method for accepting the visitor. This method dispatches the call to the correct method in the visitor.

For each data constructor in the algebraic data type, we have a concrete class which inherits from the abstract base class:

```
public class SDecl extends Stm {
  public final Typ typ_;
  public final Var var_;
  public SDecl (Typ p1, Var p2) {typ_ = p1; var_ = p2; }
  public<R, A>R accept (Visitor<R, A>v, A arg) {
    return v.visit (this, arg);
```

```
  }
}
```

The Visitor interface can be used to define operations on all the concrete classes in one or more of the hierarchies (when defining an operation on more than one hierarchy, the visitor implements multiple Visitor interfaces). This corresponds to the initial examples of pattern matching on all of the constructors, as shown in Section 2. It suffers from the same problem: lots of repetitive traversal code.

5.2 ComposVisitor

We can create a class which does all of the traversal and tree rebuilding. This corresponds to the *composOp* function in the Haskell implementation.

```
public class ComposVisitor<A> implements
  Stm.Visitor<Stm, A>, Exp.Visitor<Exp, A>,
  Var.Visitor<Var, A>, Typ.Visitor<Typ, A> {
  public Stm visit (SDecl p, A arg) {
    Typ typ_ = p.typ_.accept (this, arg);
    Var var_ = p.var_.accept (this, arg);
    return new SDecl (typ_, var_);
  }
  // ...
}
```

The ComposVisitor class implements all the Visitor interfaces in the abstract syntax, and can thus visit all of the constructors in all of the types. Each *visit* method visits the children of the current node, and then constructs a new node with the results returned from these visits.

The code above could be optimized to eliminate the reconstruction overhead when the recursive calls do not modify the subtrees. For example, if all the objects which are being traversed are immutable, unnecessary copying could be avoided by doing a pointer comparison between the old and the new child. If all the children are unchanged, we do not need to construct a new parent.

5.3 Using ComposVisitor

While the *composOp* function takes a function as a parameter, and applies that function to each constructor argument, the ComposVisitor class in itself is essentially a complicated implementation of the identity function. Its power comes from the fact that we can override individual *visit* methods.

When using the standard Visitor pattern, adding new operations is easy, but adding new elements to the object hierarchy is difficult, since it requires changing the code for all the operations. Having a ComposVisitor changes this, as we can add a new element, and only have to change the Visitor interface, the

ComposVisitor, and any operations which need to have special behavior for the new class.

The Java code below implements the desugaring example from Section 4.6 where increments are replaced by addition and assignment. Note that in Java we only need the interesting case, all the other cases are taken care of by the parent class.

```
class Desugar extends ComposVisitor⟨Object⟩ {
  public Stm visit (SInc i, Object arg) {
    Exp rhs = new EAdd (new EVar (i.var_), new EInt (1));
    return new SAss (i.var_, rhs);
  }
}
Stm desugar (Stm stm) {
  return stm.accept (new Desugar (), null);
}
```

The Object argument to the *visit* method is a dummy since this visitor does not need any extra arguments. The *desugar* method at the end is just a wrapper used to hide the details of getting the visitor to visit the statement, and passing in the dummy argument.

This being an imperative language, we do not have to do anything special to be able to thread a state through the computation. Here is the symbol table construction function from Section 4.6 in Java:

```
class BuildSymTab extends ComposVisitor⟨Object⟩ {
  Map⟨Var, Typ⟩ symTab = new HashMap⟨Var, Typ⟩();
  public Stm visit (SDecl d, Object arg) {
    symTab.put (d.var_, d.typ_);
    return d;
  }
}
Map⟨Var, Typ⟩ symbolTable (Stm stm) {
  BuildSymTab v = new BuildSymTab ();
  stm.accept (v, null);
  return v.symTab;
}
```

You may wonder why this function was implemented as a stateful computation instead of as a fold like in the Haskell version. Creating a visitor which corresponds to *composFold* would be less elegant in Java, since we would have to pass a combining function and a base case value to the visitor. This could be done by adding abstract methods in the visitor, but in most cases the stateful implementation is probably more idiomatic in Java.

Our final Java example is the example from Section 3, where we compute the set of free variables in a term in the small functional language introduced in Section 2.

```
class Free extends ComposVisitor<Set<String>> {
  public Exp visit (EAbs e, Set<String> vs) {
    Set<String> xs = new TreeSet<String>();
    e.exp_.accept (this, xs);
    xs.remove (e.ident_);
    vs.addAll (xs);
    return e;
  }
  public Exp visit (EVar e, Set<String> vs) {
    vs.add (e.ident_);
    return e;
  }
}
Set<String> freeVars (Exp exp) {
  Set<String> vs = new TreeSet<String>();
  exp.accept (new Free (), vs);
  return vs;
}
```

Here we make use of the possibility of passing an extra argument to the *visit* methods. The argument is a set to which the *visit* method adds all the free variables in the visited term.

6 Language and Tool Support for Compositional Operations

A drawback of using the method we have described is that one needs to define the *compos* function for each type or type family. Another problem when working in Haskell is that the current version of GHC does not support type class deriving for GADTs, which means that we often also have to write instances for the common built-in type classes, such as Eq, Ord and Show.

To create Compos instances automatically, we could extend the Haskell compiler to allow deriving instances of Compos. Another possibility would be to generate the instances using Template Haskell (Sheard and Jones 2002) or DrIFT (Winstanley et al. 2007), though these tools do not yet support GADTs.

We have added a new back-end to the BNF Converter (BNFC) (Forsberg and Ranta 2003, 2006) tool which generates a Haskell GADT abstract syntax type along with instances of Compos, Eq, Ord and Show. We have also extended the BNFC Java 1.5 back-end to generate the Java abstract syntax representation shown above, along with the ComposVisitor class. In addition to the abstract syntax types and traversal components described in this paper, the generated code also includes a lexer, a parser, and a pretty printer. We can generate all the Haskell or Java code for our simple imperative language example using the grammar shown below. It is written in LBNF (Labelled Backus-Naur Form), the input language for BNFC.

```
SDecl.   Stm ::= Typ Var ";";
SAss.    Stm ::= Var "=" Exp ";";
SBlock.  Stm ::= "{" [Stm] "}";
SReturn. Stm ::= "return" Exp ";";
SInc.    Stm ::= Var "++" ";";
separator Stm "";

EStm.    Exp1 ::= Stm;
EAdd.    Exp1 ::= Exp1 "+" Exp2;
EVar.    Exp2 ::= Var;
EInt.    Exp2 ::= Integer;
EDbl.    Exp2 ::= Double;
coercions Exp 2;

V.       Var  ::= Ident;

TInt.    Typ  ::= "int";
TDbl.    Typ  ::= "double";
```

7 Related Work

7.1 Scrap Your Boilerplate

The part of this work dealing with functional programming languages can be seen as a light-weight solution to a subset of the problems solved by generic programming systems. We use traversal operations similar to those in the "Scrap Your Boilerplate" (SYB) (Lämmel and Peyton Jones 2003) approach. However, no attempt is made to support completely generic functions such as those in "Generics for the Masses" (Hinze 2004) or PolyP (Jansson and Jeuring 1997). In this section we attempt to compare and contrast our work and SYB.

Introduction to Scrap Your Boilerplate

SYB uses generic traversal functions along with a type safe cast operation implemented by the use of type classes. This allows the programmer to extend fully generic operations with type-specific cases, and use these with various traversal schemes. Data types must have instances of the Typeable and Data type classes to be used with SYB.

The original "Scrap Your Boilerplate" paper (Lämmel and Peyton Jones 2003) contains a number of examples, some of which we will show as an introduction and later use for comparison. In the examples, some type synonyms (GenericT and GenericQ) have been inlined to make the function types more transparent. The examples work on a family of data types:

```
data Company  = C [Dept]                  deriving (Typeable, Data)
data Dept     = D Name Manager [Unit]     deriving (Typeable, Data)
data Unit     = PU Employee | DU Dept     deriving (Typeable, Data)
data Employee = E Person Salary           deriving (Typeable, Data)
```

```
data Person    = P Name Address     deriving (Typeable, Data)
data Salary    = S Float            deriving (Typeable, Data)
type Manager   = Employee
type Name      = String
type Address   = String
```

The first example increases the salary of all employees:

```
increase :: Data a ⇒ Float → a → a
increase k = everywhere (mkT (incS k))

incS :: Float → Salary → Salary
incS k (S s) = S (s ∗ (1 + k))
```

More advanced traversal schemes are also supported. This example increases the salary of everyone in a named department:

```
incrOne :: Data a ⇒ Name → Float → a → a
incrOne n k a | isDept n a = increase k a
              | otherwise = gmapT (incrOne n k) a

isDept :: Data a ⇒ Name → a → Bool
isDept n = False ‘mkQ‘ isDeptD n

isDeptD :: Name → Dept → Bool
isDeptD n (D n′ _ _) = n ≡ n′
```

SYB also supports queries, that is, functions which compute some result from the data structure rather than returning a modified structure. This example computes the sum of the salaries of everyone in the company:

```
salaryBill :: Company → Float
salaryBill = everything (+) (0 ‘mkQ‘ billS)

billS :: Salary → Float
billS (S f) = f
```

SYB examples using compositional operations

We will now show the above examples implemented using our compositional operations. We first lift the family of data types from the previous section into a GADT:

```
data Company; data Dept;    data Unit
data Employee; data Person; data Salary

type Manager = Employee
type Name    = String
type Address = String

data Tree :: ∗ → ∗ where
```

```
C  :: [Tree Dept] → Tree Company
D  :: Name → Tree Manager → [Tree Unit] → Tree Dept
PU :: Tree Employee → Tree Unit
DU :: Tree Dept → Tree Unit
E  :: Tree Person → Tree Salary → Tree Employee
P  :: Name → Address → Tree Person
S  :: Float → Tree Salary
```

We define *compos* as in Section 4.3, and use the operations from the library of compositional operations described in Section 4.4 to implement the examples.

```
increase :: Float → Tree c → Tree c
increase k c = case c of
                 S s → S (s * (1 + k))
                 _   → composOp (increase k) c
```

Here is the richer traversal example:

```
incrOne :: Name → Float → Tree c → Tree c
incrOne d k c = case c of
                  D n _ _ | n ≡ d → increase k c
                  _               → composOp (incrOne d k) c
```

Query functions are also easy to implement:

```
salaryBill :: Tree c → Float
salaryBill c = case c of
                 S s → s
                 _   → composFold 0 (+) salaryBill c
```

These examples can all be written as single functions, whereas with SYB they each consist of two or three functions. With SYB, the type class based system for type-specific cases forces functions that have specific cases for multiple types to be split into multiple definitions.

SYB is a powerful system, but for many common uses such as the examples presented here, we believe that the *composOp* approach is more intuitive and easy to use. The drawback is that the data type family has to be lifted to a GADT, and that the *compos* function must be implemented. However, this only needs to be done once, and at least the latter can be automated, either by using BNFC, or by extending the Haskell compiler to generate instances of Compos (as is done for SYB).

Using SYB to implement compositional operations

Single data type Above we have shown how to replace simple uses of SYB with compositional operations. We will now show the opposite, and investigate to what extent the compositional operations can be reimplemented using SYB.

The renaming example for the simple functional language, as shown in Section 3, looks very similar when implemented using SYB:

```
rename :: Exp → Exp
rename e = case e of
              EAbs x b → EAbs ("_" ++ x) (rename b)
              EVar x   → EVar ("_" ++ x)
              _        → gmapT (mkT rename) e
```

For the single data type case, our *composOp* and *composM* can be implemented with *gmapT* and *gmapM*, *composFold* is like *gmapQ* with a built-in fold, and our *compos* corresponds to *gfoldl*. Here are their definitions for the Exp type:

```
composOp :: (Exp → Exp) → Exp → Exp
composOp f = gmapT (mkT f)

composM :: Monad m ⇒ (Exp → m Exp) → Exp → m Exp
composM f = gmapM (mkM f)

composFold :: b → (b → b → b) → (Exp → b) → Exp → b
composFold z c f = foldl c z ∘ gmapQ (mkQ z f)

compos :: (∀a. a → m a) → (∀a b. m (a → b) → m a → m b)
  → (Exp → m Exp) → Exp → m Exp
compos r a f e = gfoldl (λx → a x ∘ extM r f) r e
```

The *extM* function used above has been generalized to arbitrary unary type constructors (the *extM* from SYB requires the type constructor to be in the Monad class).

Families of data types For the multiple data type case, it is difficult to use SYB to implement our examples with the desired type. We can implement functions with a type which is too general or too specific, for example:

```
rename :: Data a ⇒ a → a
rename = gmapT (rename `extT` renameVar)
  where renameVar :: Var → Var
        renameVar (V x) = V ("_" ++ x)

renameStm :: Stm → Stm
renameStm = rename
```

What we would like to have is a *rename* function which can be applied to any abstract syntax tree, but not to things which are not abstract syntax trees. Using a family of normal Haskell data types, this restriction could be achieved by the use of a dummy type class:

```
class Data a ⇒ Tree a
instance Tree Stm
```

$$\textbf{instance}\ \mathsf{Tree\ Exp}$$
$$\textbf{instance}\ \mathsf{Tree\ Var}$$
$$\textbf{instance}\ \mathsf{Tree\ Typ}$$

$$\mathit{renameTree} :: \mathsf{Tree}\ a \Rightarrow a \rightarrow a$$
$$\mathit{renameTree} = \mathit{rename}$$

However, we would like the class Tree to be closed, something which is currently only achievable using hacks such as not exporting the class.

When using *composOp*, the type restriction is achieved as a side effect of lifting the family of data types into a GADT. Using a GADT to restrict the function types when using SYB is currently not practical, since current GHC versions cannot derive Data and Typeable instances automatically for GADTs.

Using compositional operations to implement SYB

We can also try to implement the SYB functions in terms of our functions. If we are only interested in our single data type, this works:

$$\mathit{gmapT} :: \mathsf{Data}\ a \Rightarrow (\forall b.\ \mathsf{Data}\ b \Rightarrow b \rightarrow b) \rightarrow a \rightarrow a$$
$$\mathit{gmapT}\ f = \mathit{mkT}\ (\mathit{composOp}\ f)$$
$$\mathit{gmapM} :: (\mathsf{Data}\ a, \mathsf{Monad}\ m) \Rightarrow (\forall b.\ \mathsf{Data}\ b \Rightarrow b \rightarrow m\ b) \rightarrow a \rightarrow m\ a$$
$$\mathit{gmapM}\ f = \mathit{mkM}\ (\mathit{composM}\ f)$$
$$\mathit{gmapQ} :: \mathsf{Data}\ a \Rightarrow (\forall b.\ \mathsf{Data}\ b \Rightarrow b \rightarrow u) \rightarrow a \rightarrow [u]$$
$$\mathit{gmapQ}\ f = \mathit{mkQ}\ []\ (\mathit{composFold}\ []\ (+\!\!+)\ ((:[]) \circ f))$$

Of course these functions are no longer truly generic: even though their types are the same as the SYB versions', they will only apply the function that they are given to values in the single data type Exp. Defining *gfoldl* turns out to be problematic, since the combining operation that *gfoldl* requires cannot be constructed from the operations of an applicative functor.

For the type family case, it does not seem possible to use compositional operations to implement SYB operations. It is even unclear what this would mean, since type families are implemented in different ways in the two approaches.

Scrap Your Boilerplate conclusions

We consider the main differences between Scrap Your Boilerplate and our compositional operations to be that:

- When using SYB, no changes to the data types are required (except some type class deriving), but the way in which functions over the data types are written is changed drastically. With compositional operations on the other hand, the data type family must be lifted to a GADT, while the style in which functions are written remains more natural.
- SYB functions over multiple data types are too generic, in that they are not restricted to the type family for which they are intended.

- Our approach is a general pattern which can be translated rather directly to other programming languages and paradigms.
- Compositional operations directly abstract out the pattern matching, recursion and reconstruction code otherwise written by hand. SYB uses runtime type representations and type casts, which makes for more genericity, at the expense of transparency and understandability.

7.2 The Tree set constructor

Introduction

Petersson and Synek (1989) introduce a set constructor for tree types into Martin-Löf's (1984) intuitionistic type theory. Their tree types are similar to the inductive families in for example Agda (Coquand 2000), and, for our purposes, to Haskell's GADTs. The value representation, however, is quite different. There is only one constructor for trees, and it takes as arguments the type index, the data constructor and the data constructor arguments.

Tree types are constructed by the following rule:

Tree set formation

$$\frac{A : set \qquad B(x) : set[x : A] \qquad C(x, y) : set[x : A, y : B(x)] \qquad d(x, y, z) : A[x : A, y : B(x), z : C(x, y)] \qquad a : A}{Tree(A, B, C, d, a) : set}$$

Here A is the set of names (type indices) of the mutually dependent sets. $B(x)$ is the set of constructors in the set with name x. $C(x, y)$ is the set of argument labels (or selector names) for the arguments of the constructor y in the set with name x. d is a function which assigns types to constructor arguments: for constructor y in the set with name x, $d(x, y, z)$ is the name of the set to which the argument with label z belongs. For simplicity, $\mathcal{T}(a)$ is used below, instead of $Tree(A, B, C, d, a)$.

Tree values are constructed using this rule:

Tree value introduction

$$\frac{a : A \qquad b : B(a) \qquad c(z) : \mathcal{T}(d(a, b, z))[z : C(a, b)]}{tree(a, b, c) : \mathcal{T}(a)}$$

Here a is the name of the set to which the tree belongs. b is the constructor. c is a function which assigns values to the arguments of the constructor (children of the node), where $c(z)$ is the value of the argument with label z.

Trees are eliminated using the *treerec* constant, with the computation rule:

$$treerec(tree(a, b, c), f) \to f(a, b, c, \lambda z.\ treerec(c(z), f))$$

Relationship to GADTs

As we have seen above, trees are built using the single constructor *tree*, with the type, constructor, and constructor arguments as arguments to *tree*. We can use this structure to represent GADT values, as long as all children are also trees. Using the constants $l_1 \ldots$ as argument labels for all constructors, we can represent GADT values in the following way:

$$\mathtt{b}\ \mathtt{t_1} \ldots \mathtt{t_n} :: \mathtt{Tree\ a} \equiv tree(a, b, \lambda z.\ \text{case } z \text{ of } \{l_1 : t_1; \ldots ; l_n : t_n\})$$

For example, the value SDecl TInt (V `"foo"`) :: Tree Stm in our Haskell representation would be represented as the term shown below. We use "*string*" to stand for some appropriate tree representation of a string.

$$\begin{aligned} tree(Stm, SDecl, \lambda x.\ \text{case } x \text{ of } \{ & \\ & l_1 : tree(Typ, TInt, \lambda y.\ \text{case } y \text{ of } \{\}); \\ & l_2 : tree(Var, V, \lambda y.\ \text{case } y \text{ of } \{l_1 : \text{"}foo\text{"}\}) \\ & \}) \end{aligned}$$

Tree types and compositional operations

We can implement a *composOp*-equivalent in type theory by using *treerec*:

$$composOp(f, t) = treerec(t, \lambda(a, b, c, c').\ tree(a, b, \lambda z.\ f(c(z))))$$

What makes this so easy is that all values have the same representation, and c which contains the child trees is just a function which we can compose with our function f. With this definition, we can use *composOp* like in Haskell. The code below assumes that we have wild card patterns in case expressions, and that ++ is a concatenation operation for whatever string representation we have.

$$\begin{aligned} rename(t) = treerec(t, \lambda(a, b, c, c').\ \text{case } b \text{ of } \{ & \\ & V : tree(Var, V, \lambda l.\ \text{"_"} +\!\!+ c(l)); \\ & _ : composOp(rename, t) \\ & \}) \end{aligned}$$

One advantage over the Haskell solution is that we have access to both the original child values (c in the example above), and the results of the recursive calls (c' in the example above) when writing our functions. This would simplify functions which need to use the results of the recursive calls, for example the constant folding example in Section 4.6.

7.3 Related work in object-oriented programming

The ComposVisitor class looks deceptively simple, but it combines a number of features, some of which are already known in the object-oriented programming community. It does however appear that the combination which we have presented is relatively novel.

- It uses type-parameterized visitor interfaces, which can only be implemented in a few object-oriented languages. Similar parameterized visitor interfaces can be found in the Loki C++ library (Alexandrescu 2001).
- It is a depth-first traversal combinator whose behavior can be overridden for each concrete class. A similar effect can be achieved by using the BottomUp and Identity combinators from Joost Visser's (2001) work on visitor combinators, and with the depth-first traversal function in the the Boost Graph Library (Lee et al. 2002).
- It allows modification of the data structure in a functional and compositional way. The fact that functional modification is not widely used in imperative object-oriented programming is probably the main reason why this area has not been explored further.

7.4 Nanopass framework for compiler education

The idea of structuring compilers as a large number of simple passes is central to the work on the Nanopass framework for compiler education (Sarkar et al. 2005), a domain-specific language embedded in Scheme. Using the Nanopass framework, a compiler is implemented as a sequence of transformations between a number of intermediate languages, each of which is defined using set of mutually recursive data types. Transformations are implemented by pattern matching, and a *pass expander* adds any missing cases, a role similar to that of our *composOp*.

One notable feature is that a language can be declared to inherit from an existing language, with new constructors added or existing ones removed. This makes it possible to give more accurate types to functions which add or remove constructions, without having to define completely separate languages which differ only in the presence of a few constructors.

8 Future Work

8.1 Automatic generation of *compos* for existing types

Some way of automatically declaring new Compos instances for existing data types should be developed. At the moment, none of the meta-programming and generic programming tools which we have looked at support reflection over GADTs.

8.2 Applications in natural language processing

While most of the examples in this paper are related to compiler writing, we think that this technique could also be useful in natural language processing, for example in rule-based translation. One example of this would be aggregation, e.g. by transforming sentence conjunction, for example "John walks and Mary walks", to noun phrase conjunction, such as "John and Mary walk". We want to be able to do this transformation wherever sentences of this form appear in a phrase, for example in "I know that John walks and Mary walks". The transformation is done on the level of abstract syntax, and is similar to the ones for formal languages shown earlier in this paper. Since a natural language grammar may have a very large number of constructors, using *composOp* for this kind of transformation could be very beneficial. We will explore this further in the Transfer language (Bringert 2006), which is intended for writing functions over GF (Ranta 2004) abstract syntax terms. The language is dependently typed, and has support for inductive families and automatic generation of *composOp* functions.

8.3 Tree types and generic programming

In *"Scrap Your Boilerplate" Reloaded* (Hinze et al. 2006), SYB is explained in terms of a lifting of all types to a GADT. We have already seen that the tree types of Petersson and Synek (1989) are a very powerful construct which can be used to represent GADTs and perform generic operations on them. It would be interesting to see to what extent generic programming systems such as Scrap Your Boilerplate can be explained using dependent type theory with these tree types.

9 Conclusions

We have presented a pattern for easily implementing almost compositional operations over rich data structures such as abstract syntax trees.

We have ourselves started to use this pattern for real implementation tasks, and we feel that it has been very successful. In the compiler for the Transfer language (Bringert 2006) we use a front-end generated by BNFC (Forsberg and Ranta 2003, 2006), including a Compos instance for the abstract syntax. The abstract syntax has 70 constructors, and in the (still very small) compiler the various *compos** functions are currently used in 12 places. The typical function using *compos** pattern matches on between 1 and 5 of the constructors, saving hundreds of lines of code. Some of the functions include: replacing infix operator use with function calls, beta reduction, simultaneous substitution, getting the set of variables bound by a pattern, getting the free variables in an expression, assigning fresh names to all bound variables, numbering meta-variables, changing pattern equations to simple declarations using case expressions, and replacing unused variable bindings in patterns with wild cards. Furthermore, we have noticed that using compositional operations to implement a compiler

makes it easy to structure it as a sequence of simple steps, without having to repeat large amounts of traversal code for each step. Modifying the abstract syntax, for example by adding new constructs to the front-end language, is also made easier since only the functions which care about this new construct need to be changed.

Acknowledgments

We would like to thank the following people for their comments on earlier versions of this work: Thierry Coquand, Bengt Nordström, Patrik Jansson, Josef Svenningsson, Sibylle Schupp, Marcin Zalewski, Andreas Priesnitz, Markus Forsberg, Alejandro Russo, Thomas Schilling, the anonymous ICFP referees, and everyone who offered comments during the talks at the Chalmers CS Winter Meeting, at Galois Connections, and at ICFP 2006. The code in this paper has been typeset using `lhs2TeX`, with help from Andres Löh and Jeremy Gibbons. This work has been partly funded by the EU TALK project, IST-507802.

References

Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, February 2001. ISBN 0201704315.

Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, second edition, October 2002. ISBN 052182060X.

Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, December 1997. ISBN 0521582741.

Lennart Augustsson and Kent Petersson. Silly type families. URL `http://www.cs.pdx.edu/~sheard/papers/silly.pdf`. 1994.

Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998. URL `http://citeseer.ist.psu.edu/bracha98making.html`.

Björn Bringert. The Transfer programming language, 2006. `http://www.cs.chalmers.se/~aarne/GF/doc/transfer.html`.

Catarina Coquand. Agda homepage, 2000. `http://www.cs.chalmers.se/~catarina/agda/`.

Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, July 1994. doi: 10.1007/BF01211308. URL `http://dx.doi.org/10.1007/BF01211308`.

Markus Forsberg and Aarne Ranta. BNF Converter homepage, 2006. `http://www.cs.chalmers.se/~markus/BNFC/`.

Markus Forsberg and Aarne Ranta. The BNF Converter: A High-Level Tool for Implementing Well-Behaved Programming Languages. In *NWPT'02 proceedings, Proceedings of the Estonian Academy of Sciences*, December 2003. URL `http://www.cs.chalmers.se/~markus/BNFC/BNF_Report.ps.gz`.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0201633612. URL `http://portal.acm.org/citation.cfm?id=186897`.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification*. Addison-Wesley Professional, third edition, July 2005. ISBN 0321246780.

Ralf Hinze. Generics for the masses. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, volume 39, pages 236–243. ACM Press, September 2004. doi: 10.1145/1016850.1016882. URL `http://portal.acm.org/citation.cfm?id=1016882`.

Ralf Hinze, Andres Löh, and Bruno C. D. S. Oliveira. "Scrap Your Boilerplate" Reloaded. In Masami Hagiya and Philip Wadler, editors, *FLOPS*, volume 3945 of *Lecture Notes in Computer Science*, pages 13–29. Springer, 2006. doi: 10.1007/11737414_3. URL `http://dx.doi.org/10.1007/11737414_3`.

Patrik Jansson and Johan Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 470–482, New York, NY, USA, 1997. ACM Press. ISBN 0897918533. doi: 10.1145/263699.263763. URL `http://dx.doi.org/10.1145/263699.263763`.

Mark P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 97–136, London, UK, 1995. Springer-Verlag. ISBN 3540594515. URL `http://portal.acm.org/citation.cfm?id=734150`.

Lie-Quan Lee, Andrew Lumsdaine, and Jeremy G. Siek. *The Boost graph library: user guide and reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0-201-72914-8.

Ralf Lämmel and Simon Peyton Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *TLDT03*, 2003. URL `http://citeseer.ist.psu.edu/702290.html`.

Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, 1984.

Conor McBride and Ross Paterson. Applicative Programming with Effects. *Journal of Functional Programming*, 17(5), 2007. URL `http://www.soi.city.ac.uk/~ross/papers/Applicative.html`.

Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's type theory: an introduction.* Oxford University Press, 1990. Available from `http://www.cs.chalmers.se/Cs/Research/Logic/book/`.

Kent Petersson and Dan Synek. *A set constructor for inductive sets in Martin-Löf's type theory*, volume 389 of *Lecture Notes in Computer Science*, pages 128–140. Springer, 1989. doi: 10.1007/BFb0018349. URL `http://dx.doi.org/10.1007/BFb0018349`.

Simon Peyton Jones. The Haskell 98 Language. *Journal of Functional Programming*, 13(1), 2003a.

Simon Peyton Jones. The Haskell 98 Libraries. *Journal of Functional Programming*, 13(1), 2003b.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 50–61, New York, NY, USA, 2006. ACM Press. ISBN 1595933093. doi: 10.1145/1159803.1159811. URL `http://dx.doi.org/10.1145/1159803.1159811`.

Aarne Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2):145–189, March 2004. ISSN 0956-7968. doi: 10.1017/S0956796803004738. URL `http://portal.acm.org/citation.cfm?id=967507`.

Dipanwita Sarkar, Oscar Waddell, and Kent R. Dybvig. EDUCATIONAL PEARL: A Nanopass framework for compiler education. *Journal of Functional Programming*, 15(5):653–667, 2005.

Tim Sheard and Simon P. Jones. Template meta-programming for Haskell. In *Haskell '02: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM Press, 2002. ISBN 1581136056. doi: 10.1145/581690.581691. URL `http://portal.acm.org/citation.cfm?id=581691`.

Joost Visser. Visitor combination and traversal control. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, volume 36, pages 270–282, New York, NY, USA, November 2001. ACM Press. ISBN 1581133359. doi: 10.1145/504282.504302. URL `http://portal.acm.org/citation.cfm?id=504302`.

Noel Winstanley, Malcom Wallace, and John Meacham. The DrIFT homepage, 2007. `http://repetae.net/~john/computer/haskell/DrIFT/`.