Thesis for the Degree of Licentiate of Engineering

Enumerative Testing and Embedded Languages

Jonas Duregård

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden 2012

Enumerative Testing and Embedded Languages
Jonas Duregård

© 2012 Jonas Duregård

Technical Report 97L
ISSN 1652-876X
Department of Computer Science and Engineering
Functional Programming Research Group

Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 10 00

Printed at Chalmers
Göteborg, Sweden 2012

Enumerative Testing and Embedded Languages
Jonas Duregård
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

This thesis explores rapid experimental development of programming languages, with particular emphasis on effective semi-automatic testing. Our results are actualised in two Haskell libraries: BNFC-meta and Feat.

BNFC-meta is an extension of the BNF Converter (BNFC) tool. As such it is capable of building a complete compiler front end from a single high level language specification. We merge this with the practice of embedding languages in Haskell, both by embedding BNFC itself and embedding all languages defined using BNFC-meta. Embedding is carried out by means of *quasi-quotation* enabling use of the languages concrete syntax inside Haskell code. A simple extension to the grammar formalism adds *anti-quoting*, in turn allowing Haskell code embedded in the concrete syntax of the embedded languages. The end user can thus seamlessly mix concrete and abstract syntax. Our automatic approach improve on existing manually defined Haskell anti-quoters by not polluting the AST datatypes.

Our second major contribution, Feat (Functional Enumeration of Algebraic Types) automatically enables property based testing on the large AST types generated by BNFC-meta and such tools, but it is useful more generally for algebraic types. Feat is based on the mathematical notion of an enumeration as a bijective function from natural numbers to an enumerated set. This means that unlike previous list-based enumeration methods it is not intrinsically serial and can be used for both random and exhaustive testing. We describe a theory of functional enumeration as a simple algebra closed under sums, products, guarded recursion and bijections. We implement these ideas in a library and show that it compares favourably to existing tools when testing AST types.

This thesis is based on the work contained in the following papers.

I. Jonas Duregård and Patrik Jansson (2011). "Embedded Parser Generators". In: *Proceedings of the 4th ACM Symposium on Haskell*. Tokyo, Japan: ACM, pp. 107–117. ISBN: 978-1-4503-0860-1. DOI: 10.1145/2034675.2034689

II. Jonas Duregård, Patrik Jansson, and Meng Wang (2012). "Feat: functional enumeration of algebraic types". In: *Proceedings of the 2012 symposium on Haskell*. Copenhagen, Denmark: ACM, pp. 61–72. ISBN: 978-1-4503-1574-6. DOI: 10.1145/2364506.2364515

Contents

Acknowledgements

First and foremost I would like to thank my supervisors and co-authors Patrik Jansson and Meng Wang for helping me with practically every aspect of making this thesis a reality.

I would also like to thank all my other co-workers in the functional programming group at Chalmers for being such a wonderful lot and for letting me be a part of it. Among you I would single out Koen Claessen for motivating me to become a functional programming researcher in the first place, and for a great deal of inspiration and enthusiasm along the way.

I would also like to thank the people who have used my ideas in their work (and have been telling me about it). Every bug report I get makes me very happy that someone is actually using the software I developed as part of my academic effort.

Finally I would like to thank all the open source developers that make the tools we depend on every day to carry out our research (and particularly the wonderful members of the Haskell community). There would be no such thing as software science if it where not for people like you.

Introduction

The traditional view of choosing which programming language to use is that of general purpose languages competing over being *the* best programming language. But in reality, all larger software systems involve several programming languages to solve different tasks. It is widely accepted among software professionals that there is no language which is best suited for all the problems a software engineer faces, no "one language to rule them all". It is rather a matter of selecting the right tool for the right job.

The Domain Specific Language (DSL) paradigm shifts focus further away from the ideal of a monolithic programming language. Instead each domain of computable problems has its own set of languages and tools. The analogy to other engineering problems is clear: there are tools that solve a very wide range of problems, but a tool that is tailored for the specific problem at hand is potentially more effective and most likely easier to use.

Understandably this approach has spawned a very large number of languages. In extreme cases a DSL can even be developed and used exclusively within a single project to solve a frequently recurring problem. Using DSLs stresses the need for tools that allow programmers to rapidly prototype, maintain and test new languages.

In this chapter we explain the most fundamental concepts used in the remainder of the thesis and in §4 we give a concise summary of the contributions focusing on motivating examples.

1 Algebraic datatypes

The algebraic datatype (ADT) is one of the foremost tools in a functional programmers toolkit. Each datatype is defined by a number of constructors. Each constructor is just a label representing a unique method of constructing values. Thus each constructor represents a disjoint subset of the type. In addition to the label, each constructor has a sequence of *component datatypes*. To build a value from the constructor its label is combined with a value from each such component type.

A very simple algebraic datatype is the type of Boolean values. It has two constructors labelled "False" and "True", and the constructors have no additional components so each constructor label is a value in its own. In Haskell, we define this type by:

```
data Bool = False | True
```

We can define pairs of Boolean values as a datatype with a single constructor that has two components of type Bool.

```
data BoolPair = BP Bool Bool
```

A pair of booleans can be constructed by applying the label BP to any two boolean values e.g. BP True False. The BoolPair type demonstrates the algebraic nature of ADTs: complex types are built by combining simpler ones. The algebra becomes clearer if we consider the *sum of product* view of datatypes: Adding a constructor to a datatype corresponds to addition, extending a constructor with an additional component type corresponds to multiplication. Thus Bool could be expressed as $\mathsf{True} + \mathsf{False}$ and if we disregard the label for BoolPair we get the following expression: $(\mathsf{False} + \mathsf{True}) * (\mathsf{False} + \mathsf{True})$. Expanding this using the distributive property as we would in ordinary arithmetics we get:

$$\mathsf{False} * \mathsf{True} + \mathsf{False} * \mathsf{False} + \mathsf{True} * \mathsf{False} + \mathsf{True} * \mathsf{True}$$

This sum corresponds directly to each of the four existing pairs of boolean values. Quite frequently the labels are abstracted away altogether and constructors with no components are simply expressed as 1 giving $\mathsf{BoolPair} = (1+1) * (1+1)$. Of course these types contain only a finite number of values. More generally ADTs capture the notion of *recursively defined sets* such as the set of (Peano) natural numbers. Each number is either zero or the successor of another number. Such types are easily defined by a recursive datatype:

```
data Nat = Zero
         | Succ Nat
```

The introduction of recursion makes the algebraic view of datatypes slightly more complicated. It is higly desirable to have a finite algebraic expression for datatypes such as Nat. This is usually handled by extending the algebra with a least fixed point operator μ such that $\mathsf{Nat} = \mu\ \mathsf{n}.\ 1 + \mathsf{n}$. The fixed point operator can be extended to enable mutual recursion (cf. polyvariadic fixed point combinators) although often algebraic representations of data types used in generic programming do not support this.

Type constructors and regular types A *type constructor* (not to be confused with the *data* constructors described above) is a datatype definition with variables that when substituted for specific ADTs form a new ADT. An example is the tuple type (a, b) which is a generalisation of BoolPair where BoolPair = (Bool, Bool). A larger example is for instance a binary tree with data in each leaf:

```
data Tree a = Leaf a
            | Branch (Tree a) (Tree a)
```

The type constructor Tree can be applied to the Nat type to yield the ADT Tree Nat of trees with natural numbers in their leaves. The combination of recursive types and type constructors allow types which can not be expressed as a least fixed point. For instance the type of balanced binary trees:

```
data Balanced a = BLeaf a
                | BBranch (Balanced (a, a))
```

These types are referred to as non-regular types and are often not supported by generic programming tools (such as Jansson and Jeuring (1997)) and other tools that use the algebraic representation of types.

Pattern matching The principal tool for defining functions on algebraic datatypes, *pattern matching* breaks a function into cases for each constructor. Each case binds the component values of the matching constructor to variables. For instance addition of natural numbers:

```
add Zero     m = m
add (Succ n) m = Succ (add n m)
```

As is often the case with functions on natural numbers, addition has two interesting cases: either the first operand is zero or it is the successor of another natural number. In the latter case we bind that number to the variable n so that we can recursively add it to the second operand.

2 Abstract syntax trees and context free languages

The availability of ADTs and pattern matching has made functional languages such as Haskell popular choices for implementing compilers and language technology software.

One reason for this is the close connection between algebraic datatypes and Abstract Syntax Trees (ASTs) of context free languages. The *concrete syntax* of a language is the set of all character strings that it contains. The *abstract syntax* is a tree representation of the same information. It is called abstract because it usually abstracts away from things in the concrete syntax which are not important for the semantics of the language e.g. if spaces or tabs are used or if there are redundant brackets around a number.

For example consider a language of boolean expressions with literals, negation and conjunction. The concrete syntax of an expression could be reminiscent of natural language e.g. `"true and not false"`. We define an algebraic datatype to represent the abstract syntax:

```
data Exp = F
         | T
         | Not Exp
         | And Exp Exp
```

The abstract syntax of the sentence `"true and not false"` is And T (Not F). Abstract syntax trees are not as easy for humans to read and maintain, but for implementing programs that manipulate or evaluate expressions it is easier to work with the abstract syntax. For instance to evaluate expressions to a truth-value we only need this tiny program:

```
eval :: Exp → Bool
eval F             = False
eval T             = True
eval (Not e)       = not (eval e)
eval (And e1 e2) = eval e1 ∧ eval e2
```

The BNF Converter The popularity of context free languages in software engineering is largely due to the Backus-Naur Form (BNF) grammar formalism (Naur, 1963). In BNF a language is specified by a set of production rules, each rule consisting of a category name and a sequence of terminal symbols and non-terminal categories. These production rules describe how symbols can be combined into sentences in the language: If we start with a category name and expand it using any production rule, repeating the process for all resulting non-terminals until only terminal symbols remain, then that sequence of symbols is a sentence in the specified language. Overly simplified the BNF grammar of our little expression language is as follows:

```
Exp ::= "true";
Exp ::= "false";
Exp ::= "not" Exp;
Exp ::= Exp "and" Exp;
```

We implicitly allow whitespace characters between our terminal symbols. From a language specification such as this it is possible to automatically derive efficient parsers that convert linear concrete syntax to an abstract tree structure. Note that each production rule in our grammar corresponds to a single constructor in our AST datatype: each category is a datatype and each of the non-terminals in a rule correspond to a data field in a constructor. The only thing missing are the constructor names. The BNF Converter (Forsberg and Ranta, 2003) uses this fact to automatically generate an AST type definition from a grammar. This requires the BNF formalism to be extended with a *label* for each rule, corresponding to the constructor name. In Labelled BNF, the grammar can be specified as follows:

```
F.   Exp ::= "false";
T.   Exp ::= "true";
Not. Exp ::= "not" Exp;
And. Exp ::= Exp "and" Exp;
```

It is easy to recreate the AST datatype from this description; remove the non terminals and you are left with the constructor names and components. This grammar has a few problems however. Particularly it is *ambiguous*. For instance it is not clear how `"not false and true"` should be parsed, is it Not (And F T) or And (Not F) T. This problem is solved by specifying *precedence*: does conjunction take precedence over negation or vice versa? Also `"false and true and false"` can be parsed as either And F (And T F) or And (And F T) F. This problem is solved by specifying *associativity*: is conjunction right or left associative? Regardless of which answers we choose for these questions, a third question arises: how can we specify the other meanings of the sentence in our expression language? For this we need to add some kind of parentheses in the concrete syntax, but these are not needed in the abstract syntax where everything is already in an unambiguous tree form. In BNFC these issues are solved by two mechanisms:

- *Indexed categories* are used to distinguish a single abstract category into several syntactic sub-categories by adding trailing numbers to category names.
- *Syntactic dummies* allows users to specify syntactic rules which have no influence on the abstract syntax type definition, by replacing the label of the rule with an underscore.

Applying these techniques yields the following grammar for our expression type:

```
And. Exp0 ::= Exp0 "and" Exp1;
Not. Exp0 ::= "not" Exp1;
F.   Exp1 ::= "false";
T.   Exp1 ::= "true";
_.   Exp1 ::= "(" Exp0 ")";
```

This parses `"not false and true"` as And (Not F) T (conjunction takes precedence) and `"not (false and true)"` as Not (And F T).

Given this grammar, BNFC produces source code (in Haskell or any of several available languages) for three primary components:

- The AST datatype definition.
- A *parser* that can convert from concrete to abstract syntax (or fail with a syntax error).
- A *pretty printer* which takes an abstract syntax tree and converts it to a concrete syntax string. The output is "pretty" in the sense that it contains no redundant brackets and has coherent indentation etc.

Together these components form a complete compiler front end. With all syntactic issues handled in a single self-contained file, the developer is free to focus on the semantics of the language i.e. writing functions on the abstract syntax type.

This ability to rapidly go from a high level syntax description to a complete compiler front end has proven valuable in programming language teaching and research, as well as in some industrial applications (Forsberg and Ranta, 2004).

Domain Specific Languages General purpose programming languages such as C, Java and Haskell have been designed to enable programmers to solve any problem with a computable solution. Domain Specific Languages (DSLs) on the other hand are designed to solve a much more narrow set of problems (called the problem domain). More general problems might be difficult or even impossible to solve in the DSL. Examples of well known DSLs include SQL for database queries and HTML for web pages.

The motivation for using DSLs instead of general purpose programming languages is usually some advantage that the specialization offers at the cost of more general functionality. For instance domain specific code optimisations that work well in the problem domain of the DSL but which could be detrimental in other contexts.

Embedded Languages The reason that it is so simple to make a function that evaluates the boolean expression example is that the language is essentially a subset of Haskell. Each construct in the expression language is simply delegated to the corresponding Haskell function. It is often the case in DSL design that the language you are designing has a large array of standard features (such as arithmetic expressions) and a few domain specific features on top of that.

The principle of embedded languages is that instead of building a DSL from scratch, the DSL is embedded in a *host language*. Each program in the DSL is a program in the host language (but not vice versa). The domain specific elements of the language are implemented as a library. The details of the library are hidden from the user with abstraction features of the host language, in some sense it becomes a language of its own rather than a mere library (Hudak, 1996).

3 Property based testing

Verifying the correctness of software is essential to software engineering, and testing is a major part of virtually any software project. *Property based testing* is a semi-automatic approach where developers write predicates (the properties) that universally quantify over some input set (typically a datatype). Each predicate is a specification for some part of the software under test. A testing framework is then used to automatically construct test cases from the input set and to search for a counter-example to the tested property. If a counter-example is found there is either a bug in the program or the specification is incorrect.

Random testing The most well known testing framework for functional programming is QuickCheck, described by Claessen and Hughes (2000). One of the foremost merits of QuickCheck is the ease with which properties are defined and the short trail from a high level specification to an executable test suite. The simplest predicates are just functions from input data to booleans. For instance to test the relation between the reverse function on strings and string concatenation we define the following function:

```
prop_RevApp :: String → String → Bool
prop_RevApp xs ys = reverse (xs ++ ys) == reverse ys ++ reverse xs
```

All the parameters of the function are implicitly universally quantified. To verify the correctness of this property we only need to pass it to the QuickCheck test driver:

```
Main> quickCheck prop_RevApp
OK! passed 100 tests.
```

As the output suggest, the property is only partially verified. Specifically 100 pairs of strings were tested without finding a counterexample. The test data generated by QuickCheck is selected at random. Datatypes are associated with default random generator using a type class (called Arbitrary), and the library includes combinators to build generators for user defined types.

When writing generators the user must ensure termination and reasonable size of generated values. The library provides several tools for making this easier. It is more difficult to assess the statistical coverage of the generator, i.e. how likely it is to exercise all relevant parts of the tested code. If a property passes it is difficult to verify that it is not due to some flaw in the generator that causes it to avoid generating significant test cases. One way to mitigate this uncertainty is to simply run more tests, but the library also provides some tools for manually inspecting the test cases generated to ensure the tester that good coverage is achieved.

The small scope hypothesis A common observation in software testing is that if a program fails to meet its specification, there is typically a small input that exhibits the failure (by some definition of small). The small scope hypothesis states that it is at least as effective to exhaustively test a class of smaller values (the small scope) as it is to randomly or manually select test cases from a much larger scope. The Haskell library SmallCheck (Runciman, Naylor, and Lindblad, 2008) applies the small scope hypothesis to algebraic datatypes, and argues that most bugs can be found by exhaustively testing all values below a certain depth limit. The depth of a value is the largest number of nested constructor applications required to construct it. So in the ADT for arithmetic expressions, `And T (Not F)` is of depth two because the constructor `F` is nested inside the constructor `Not` which in turn is nested in `And`. Exhaustive testing by depth has at least two advantages over random generation:

- Generators are mechanically defined. There is usually no thought involved in writing the enumeration procedure for a datatype, it just mirrors the definition of the type itself.
- When a property succeeds the testing driver can give a concise and meaningful description of coverage: the depth limit to which it was able to exhaustively test.

The disadvantage is that the number of values can grow extremely fast and exhaustively testing even to a small depth might not be feasible. Even in our small example the number of values grows as a double exponential and by depth five there are already 10^{15} values which is well beyond feasibility for exhaustive search. The SmallCheck library provides combinators to mitigate this by manually changing the depth cost of selected constructors e.g. make the conjunction operator increase the “depth” of values by two instead of one. Unfortunately this procedure partly eliminates both the advantages described above: generator definition is no longer mechanical and it is no longer easy to understand the inclusion criteria of a test run.

Enumerative combinatorics Mathematically speaking, ADTs are a special case of *combinatorial structures*. As such they are studied in the branch of mathematics called *combinatorics*. There is ample literature on the subject, and some of this work might have applications in property based testing. Particularly *enumerative* combinatorics deals with the problem of counting the number of structures of a given size. A central concept of enumerative combinatorics is *combinatorial species*, first described by Joyal (1981). Species define structures composed of sums and products, but also of more complicated structures such as sets and cycles. From each species several *generating functions* can be computed and these are used to count the number of structures of a given size. Yorgey (2010) argues that species can be used both for random and exhaustive testing of functional programs.

4 Contributions

Our contributions are divided in two parts, corresponding to the chapters of this thesis. Each part is also identified by a Haskell library: *BNFC-meta* which is a "totally embedded" version of the BNF Converter, and *Feat* which is a random-systematic hybrid test case generator which works well for properties quantifying over Abstract Syntax Trees (but also for algebraic datatypes in general).

Embedded parser generators Our first paper describes BNFC-meta, a library that keeps all the benefits of the BNF Converters Haskell Back End and extends it in three ways:

- Grammars are embedded in Haskell source files rather than kept in a separate file. Haskell extensions for compile time metaprogramming replace external generation tools. This makes experimentation and prototyping even faster because of a shorter and more dependable tool chain.

- In addition to the parser and the pretty printer, a quasi-quoter (Mainland, 2007) is constructed for the specified language. This allows users to write constants in the concrete syntax of the language directly in Haskell source files and they are statically (at compile time) expanded to their corresponding abstract syntax.

- The grammar formalism is extended with a simple mechanism for defining anti-quoting for the generated quasi-quoters. This allows users to leave "holes" in their quasi-quoted strings which are converted to Haskell variables in the AST values. This gives the programmer the ability to express functions and patterns in terms of the concrete syntax.

The last point may seem to contradict the statement that the abstract syntax is easier to manipulate than the concrete syntax, and in some way that is true. Most recursive functions are easier to write on the abstract syntax, where there is one case for each rule. Some more complex procedures, particularly program transformations like constant unfolding, are often easier to express by a pattern on the concrete syntax (Mainland, 2007).

Motivating example Suppose we have a small language with Java-like syntax. The complete grammar for the example is presented in table 1. Using the abstract syntax of the language we define a function that takes an expression and builds a complete program that prints the result of evaluating the expression.

```
printing :: Expr → Prog
printing e = Fun TInt (Ident "main") [SFunApp (Ident "print") [e]]
```

To apply this function to the expression 5 + 5 we declare a value print10:

```
print10 :: Prog
print10 = printing (EPlus (EInt 5) (EInt 5))
```

Evaluating print10 in an interpreter (GHCI) gives the following output:

```
Main> print10
Fun TInt (Ident "main")
  [SFunApp (Ident "print") [EPlus (EInt 5) (EInt 5)]]
```

A more readable output is produced using the pretty printer provided by BNFC-meta (as well as by BNFC):

```
Main> putStrLn (printTree print10)
int main () {
  print (5 + 5);
  }
```

This division of syntax into illegible abstract syntax and legible concrete syntax raises the following question: why are we using the illegible syntax in our definitions?

Some would argue that the definitions above can be improved greatly using the ordinary abstraction tools of Haskell (factoring out code, type classes, infix operators etc.). This is correct to some extent but it does require some work and in the end the programmer has to keep two programming languages in mind: the actual concrete syntax of the language and the syntax of what is essentially an embedded language for the abstract syntax. With some effort you can bend Haskell into making these languages very similar, but not so similar that knowledge of the concrete syntax is sufficient to define values in the abstract syntax.

```
import Language.LBNF (lbnf, dumpCode, bnfc)
bnfc [lbnf|
Fun.      Prog ::= Typ Ident "(" ")" "{" [Stm] "}";
SDecl.    Stm  ::= Typ Ident ";";
SAss.     Stm  ::= Ident "=" Expr ";";
SIncr.    Stm  ::= Ident "++" ";";
SWhile.   Stm  ::= "while" "(" Expr ")" "{" [Stm] "}";
SFunApp.  Stm  ::= Ident "(" [Expr] ")" ";";
_.        Stm  ::= Stm ";";
ELt.      Expr0 ::= Expr1 "<" Expr1;
EPlus.    Expr1 ::= Expr1 "+" Expr2;
ETimes.   Expr2 ::= Expr2 "*" Expr3;
EVar.     Expr3 ::= Ident;
EInt.     Expr3 ::= Integer;
$.        Expr3 ::= "$" Ident;
[].       [Stm] ::=;
(:).      [Stm] ::= Stm [Stm];
separator Expr ",";
_. Expr  ::= Expr0;
_. Expr0 ::= Expr1;
_. Expr1 ::= Expr2;
_. Expr2 ::= Expr3;
_. Expr3 ::= "(" Expr ")";
TInt.     Typ ::= "int";
TVoid.    Typ ::= "void";
|]
```

Table 1: A complete executable definition of a small Javalette-like language

In BNFC-meta we get a quasi-quoter for each datatype (i.e. for each grammar category) for free, so the definition of print10 can be changed to:

```
print10 :: Prog
print10 = printing [expr | 5 + 5 |]
```

This is only a minor improvement, and we cannot use the quasi-quoter for Prog directly to define the body of printing since it contains the free variable e. However, by adding a simple *anti-quoting rule* to the grammar we can enable expressions with variables:

```
$. Expr3 ::= "$" Ident;
```

The label is replaced by $ which indicates that the non-terminal on the right hand side is parsed as Haskell code. We add a terminal "$" to distinguish Haskell variables from Javalette variables. Now we can define a version of printing in terms of the concrete syntax:

```
printing' :: Expr → Prog
printing' e = [prog |
  int main () {
    print ($e);
  } |]
```

The type of printing′ is automatically inferred if we omit it. In fact after the quasi-quotation is resolved at compile time, the Haskell code from this meta-program is completely identical to the abstract syntax definition of printing.

Functional Enumeration of Algebraic Datatypes Our second paper is about Feat: a theory of efficient functional enumerations and a library based on this theory. The motivation was initially to test code produced by BNFC-meta and other applications involving large Abstract Syntax Tree types, but it may be applicable in a wider field of testing and perhaps for other purposes that benefit from quick access enumerated data.

Given the prominent position of testing and verification in the software engineering process, no development tool can be called complete without taking testing into account. With BNFC-meta intended for rapid prototyping, and with the popularity of test driven development strategies it seems essential to provide facilities for equally fast test suite prototyping. It turns out that Abstract Syntax Trees, although a very popular application of Haskell, is a bit of a blind spot for existing tools. Particularly we identify two main issues that we overcome with our approach:

- Writing random generators by hand for large systems of types is painstaking, and so is verifying their statistical soundness.

- The small scope hypothesis does not apply directly to large ADTs.

The first issue is exacerbated by the use of BNFC-meta as a prototype tool. Writing and verifying test data generators for large types is particularly bothersome if the AST type is constantly undergoing changes.

The second issue is demonstrated in the paper. Applying SmallCheck to properties that quantify over a large AST (in our case that of Haskell itself with some extensions) proved improductive for the purpose of finding bugs. The reason is the extreme growth of the test space as depth increases, which practically prevents SmallCheck from reaching deep enough to find bugs.

To overcome these problems we provide *functional enumerations*. We consider an enumeration as a sequence of values. In serial enumerations this sequence is typically represented by a lazy infinite list starting with small elements and moving to progressively larger ones. For example the enumeration of the values in our expression type might start with [F, T, Not F, Not T, And F F, And F T, ...

A functional enumeration is instead characterised by an efficient indexing function that computes the value at a specified index of the sequence, essentially providing random access to enumerated values. The difference is best demonstrated by an example:

```
Main> index (10^100) :: Exp
Not (And (And (And (And (Not T) (And (And T T) T))
...
(And (Not T) F))) F)
```

Here we access the value at position 10^{100} in the enumeration of the Exp type (with ten lines of additional data cut away). Clearly accessing this position in a serial enumeration is not practical.

This “random access” allows Functional enumerations to be used both for SmallCheck-style exhaustive testing of small scopes and QuickCheck-style random testing with a statistical guarantee of uniform distribution over a well specified subset of the type. We show in a case study that this flexibility helps discover bugs that can not practically be reached by the serial enumeration provided by SmallCheck.

Motivating example Returning to the example with the Java-like language. Suppose we want to test some property of the language, for instance the print-parse-cycle which states that composing the parser with the pretty-printer yields the identity function (if this is not the case there is a bug in either the pretty-printer or the parser). We can formalise this as the following predicate:

```
prop_cycle :: Prog → Bool
prop_cycle a = err (const False) (fmap (a ==)
  (pExpr (tokens (printTree a))))
```

Using err (const False) means we consider parse errors as failures. We can test properties like these with QuickCheck, provided that the type of the function argument is an instance of the Arbitrary type class. There is no such instance for Prog, and writing one by hand is difficult and error prone. In Feat there is a Template Haskell metaprogram for automatically constructing an instance of the Enumerable type class (part of Feat), and from that we can easily construct an instance of arbitrary. The code to construct the Enumerable instances is simple:

```
deriveEnumerable ''Prog
deriveEnumerable ''Stm
deriveEnumerable ''Expr
deriveEnumerable ''Typ
```

Although it is convenient to use the Template Haskell script, the generated code is manageable. For instance this is the generated instance for Expr:

```
instance Enumerable Expr where
  enumerate = consts
    [unary (funcurry ELt)
    ,unary (funcurry EPlus)
    ,unary (funcurry ETimes)
    ,unary EVar
    ,unary EInt
    ]
```

Without explaining the semantics of unary, the pattern is still simple: for a constructor with $k + 1$ components we apply funcurry k times (there is a special function nullary for constructors with no components). Now we can use the function uniform :: Enumerable a ⇒ Int → Gen a together with the QuickCheck function sized to construct an Arbitrary instance:

```
instance Arbitrary Prog where
  arbitrary = sized uniform
```

This gives a random generator that given a size bound n (by QuickCheck) chooses a value uniformly at random from the set of value of size n or less. The result of a test run:

```
Main> quickCheck prop_cycle
*** Failed! Falsifiable (after 15 tests):
Fun TInt (Ident "a")
  [SWhile (ETimes (EInt (-1)) (EVar (Ident "a’"))) []]
```

As is typically the case with random testing, this is not a minimal failing example. The usual method to get one is to define a *shrinking function*, and it is possible to derive one automatically if needed (Duregård, 2009). Feat offer an alternative approach however, since we can enumerate values exhaustively as well as randomly. Doing so for all values of size 15 or less, starting with the smallest yields the following result:

```
Main> ioAll 15 (showReport prop_cycle)
Main> main2
--- Testing 0 values at size 0
--- Testing 0 values at size 1
--- Testing 0 values at size 2
--- Testing 0 values at size 3
--- Testing 10 values at size 4
--- Testing 0 values at size 5
--- Testing 0 values at size 6
--- Testing 50 values at size 7
--- Testing 150 values at size 8
--- Testing 300 values at size 9
--- Testing 370 values at size 10
Failed: Fun TInt (Ident "a") [SAss (Ident "a") (EInt (-1))]
```

The concrete syntax of this is int a () {a = −1; } and it is not parseable, indicating a lexical error at −1. Investigations show that the BNFC built-in type Integer is indeed intended to parse only non-negative integers, so this is arguably a "bug" in the generated AST type. Specifically the AST typ is too "wide" i.e. it contains values that are not represented by any concrete sentence in the language. This kind of imprecision is very common in AST types and there needs to be a method of flagging these as false positives. One way to ignore the error is by altering the property to pass for any value with a negative literal. This method has the problem that it generates and discards many values which are not in the language, at great computational cost. The ratio of discarded values typically grows as the size of expressions grows. Instead we modify the definition of the enumeration and make it enumerate only non-negative numbers. This is done by providing a bijective function from another enumerated type to the set of non-negative integers. One simple bijection is from the integers themselves, mapping negative numbers to odd numbers and positive to even:

```
toNat :: Integer → Integer
toNat n = if n < 0 then abs (n * 2 + 1) else n * 2
```

To modify the enumeration we simply change the instance of Enumerable Expr above, replacing unary EInt with unary (EInt ∘ toNat). Rerunning the test immediately yields a reassuring answer from QuickCheck:

```
Main> quickCheck prop_cycle
+++ OK, passed 100 tests.
```

Significantly slower, we can enumerate all values with 14 or fewer constructors:

```
Main> main2
--- Testing 0 values at size 0
--- Testing 0 values at size 1
...
--- Testing 1870 values at size 11
--- Testing 10340 values at size 12
--- Testing 16480 values at size 13
--- Testing 39930 values at size 14
--- Done. Tested 69500 values
```

Paper I

Embedded Parser Generators

This chapter was originally published in the proceedings of the 2011 Haskell Symposium under the same title. It has undergone only minor corrections since.

Embedded Parser Generators

Jonas Duregård, Patrik Jansson

Abstract

We present a novel method of embedding context-free grammars in Haskell, and to automatically generate parsers and pretty-printers from them. We have implemented this method in a library called BNFC-meta (from the BNF Converter, which it is built on). The library builds compiler front ends using metaprogramming instead of conventional code generation. Parsers are built from labelled BNF grammars that are defined directly in Haskell modules. Our solution combines features of parser generators (static grammar checks, a highly specialised grammar DSL) and adds several features that are otherwise exclusive to combinatory libraries such as the ability to reuse, parameterise and generate grammars inside Haskell.

To allow writing grammars in concrete syntax, BNFC-meta provides a *quasi-quoter* that can parse grammars (embedded in Haskell files) at compile time and use metaprogramming to replace them with their abstract syntax. We also *generate* quasi-quoters so that the languages we define with BNFC-meta can be embedded in the same way. With a minimal change to the grammar, we support adding *anti-quotation* to the generated quasi-quoters, which allows users of the defined language to mix concrete and abstract syntax almost seamlessly. Unlike previous methods of achieving anti-quotation, the method used by BNFC-meta is simple, efficient and avoids polluting the abstract syntax types.

1 Introduction

The underlying motivation of this paper is to support rapid development of, and experimentation with, Domain Specific Languages (DSLs). Especially if the desired syntax of the DSL makes it difficult or impossible to embed it in a general purpose language using conventional methods (as a combinator library). We aim to eliminate the "barrier" associated with employing a parser generator such as the BNF Converter (Forsberg and Ranta, 2003), and make it as easy to use as a parser combinator library. The title of this paper is deliberately ambiguous about what is embedded, referring both to the parser generators and the generated parsers.

Embedded (parser generators) Like the original BNF Converter on which it is built, BNFC-meta builds compiler front ends (abstract syntax types, parsers, lexers and pretty-printers) from grammar descriptions. Unlike the BNF Converter: 1) BNFC-meta is a metaprogram and 2) our grammar descriptions are embedded as Haskell values.

By “metaprogram” we mean that it is a Haskell function from grammars to abstract Haskell code. This abstract code can be “spliced” by the compiler using the non-standard language extension Template Haskell (Sheard and Jones, 2002). Shortening the compilation tool chain in this way has many practical advantages, including faster and simpler compilation. Embedding the language definitions also allow users to define their own functions in the same module as the parser.

The fact that grammars are embedded in the target language (i.e. Haskell) is a major advantage of our approach. This lends BNFC-meta features which are typically reserved for combinator parsers, namely the possibility of building grammar definitions using all the abstraction features of a functional language. This can drastically reduce the complexity of code, enabling features such as reusable grammars, parameterised grammars and programmer defined grammar transformations. Even though BNFC-meta grammars are embedded in Haskell rather than defined in separate files, users can still write grammars in the same concrete syntax as in BNFC. This is achieved using another metaprogramming facility called quasi-quoting, which essentially provides programmer defined syntax extensions to Haskell. Users can mix concrete and abstract grammar syntax depending on which is most suited for the task at hand.

(Embedded parser) generators The alternative interpretation of the title (that the generated parsers are embedded) highlights another innovation in BNFC-meta. By embedded we mean that any language defined with BNFC-meta (an object language) can be used as a syntactic extension to Haskell, and the compiler will statically parse the concrete syntax of the object language into its corresponding abstract syntax. This is achieved using the same technique as when we embed our grammar definitions; we automatically generate quoters for the object language. As indicated by their name, a quoter allow the object language to be used in a syntactically defined scope (a quote).

Quotes can be used to define both patterns and expression, but in order to define useful patterns we need to be able to bind variables. In general we want to have “holes” in our quotes that are filled with Haskell values. This is called anti-quoting, and it is what separates a *quasi*-quoter from just a quoter. In BNFC-meta we provide built in support for defining anti-quotation.

The embedded parsers generated by BNFC-meta enables the programmer to “mix and match” abstract and concrete syntax of the object language seamlessly, reducing code size and increasing readability for several common language processing tasks.

Paper contents The remainder of this paper is structured as follows. In §2 we present the tools on which BNFC-meta is built and give some examples of using it. In §3 we present the details of embedding BNFC into a Haskell library. We also argue that this method is not only useful for BNFC, but can be applied in several other contexts. In §4 we explain the quasi-quoting mechanisms generated by BNFC-meta, and explain why these are a natural consequence of embedding BNFC as a library. In §5 we show some performance results and a large scale example. We conclude in §6 with a discussion of related and future work.

2 Preliminaries

BNF, LBNF and BNFC The Backus-Naur Form (BNF) is a notation for context free grammars. The notation is widely used in computer language processing, mainly due to the fact that efficient parsers can be automatically constructed from BNF grammars. There are many variants of BNF but they all have production rules which are combined to form sentences:

```
Foo ::= "Foo!" Foo
      | Bar
Bar ::= "Bar."
```

The category Foo (also called a non-terminal, as opposed to the terminal strings) represents all sentences on the form "Foo!Foo! ... Foo!Bar.". In this paper we use a variant of BNF called Labelled BNF (LBNF). In LBNF each production rule represents a single production (there is no vertical bar operator) and each production rule carries a descriptive label. The grammar above can be expressed as:

```
FooCons. Foo ::= "Foo!" Foo;
FooNill. Foo ::= Bar;
BarDot.  Bar ::= "Bar.";
```

The primary advantage of LBNF is that a system of algebraic datatypes can be extracted from the rules by using categories as types and labels as constructors. These types capture the abstract syntax tree of the specified language. In our example:

```
data Foo = FooCons Foo | FooNill Bar
data Bar = BarDot
```

The BNF Converter (BNFC) is a program that uses LBNF grammars to generate complete compiler front ends. This includes abstract syntax tree types, a lexical analyser (lexer), a parser and a pretty printer. BNFC is written in Haskell but can generate parser code for many target languages, including Haskell, C/C++ and Java. BNFC-meta does not support any target language other than Haskell. When we refer to the BNFC in the remainder of this paper we actually mean only its Haskell back end. The LBNF grammar formalism has many features which are not described in this paper (see (Forsberg and Ranta, 2003) for details).

Template Haskell and Quasi-quoters Template Haskell is a non-standard metaprogramming extension to Haskell, first described by (Sheard and Jones, 2002). The only compiler that currently supports the extension is the Glasgow Haskell Compiler. The features of Template Haskell relevant to this paper include:

- A library of datatypes for the abstract syntax of Haskell, including types for declarations, expressions, patterns etc. We call values of these types *metaprograms*.
- A language extension for "splicing" the metaprograms into Haskell source code at compile time.

Suppose we have defined a few metaprograms (in this case simple code fragments) in a module MyTemplates. The following program is possible:

```
{-# LANGUAGE TemplateHaskell #-}
import Language.Haskell.TH (Q, Dec, Exp)
import MyTemplates (myDeclarations,    -- :: Q [Dec]
                    myExpression)      -- :: Q Exp
  -- Top-level Haskell declaration splice
myDeclarations
  -- Expression splice
splicedExpression = $(myExpression)
```

In this example the expression myDeclarations is used in place of a sequence of declarations at the top level. The compiler will evaluate it, splice in the resulting declarations in its place and continue to compile the resulting code. In expression contexts splices must be indicated by a $ but the idea is the same: evaluate, splice in, re-compile. Note that the Q monad can perform IO actions so even this simple example could potentially splice "dynamic" code e.g. by reading a description from a file or base the particular code on the architecture of the compiling machine.

Quasi-quotes The term quasi-quotation is not used entirely consistently in the functional programming community. In a broader linguistic setting, the term quasi-quote was coined by W. V. Quine in 1940 as a means of formally distinguishing metavariables from surrounding text. In computer languages such as Lisp, the term quasi-quote is almost synonymous with template metaprogram (Bawden, 1999) i.e. a metaprogram that may contain "holes" for the programmer to fill. This type of quasi-quotes are also in the original Template Haskell design. Understanding exactly how this kind of quasi-quoter works is not essential for this paper, but we show a (somewhat contrived) example of how it can be used:

```
sharedPair :: Q Exp → Q Exp
sharedPair e = [| let x = $e in (x, x) |]
```

The [| initiates the quote, which means that the Haskell code within it is a metaprogram. The dollar sign marks holes in the metaprogram, where e is the parameter of sharedPair. The general term for dollar sign is *anti-quotation* operator, since it escapes from the quoted context into the surrounding metalanguage.

The QuasiQuotes extension to Haskell, introduced in (Mainland, 2007), offers a generalisation of quasi-quotes where any *object language* can reside in a quote. The Lisp and Template Haskell quasi-quotes are the special case where the object language is the same as the enclosing language.

These general quasi-quoters are often used to embed domain specific languages, which have an established concrete syntax. Often Haskell's syntax is enough to embed at least an approximation of the DSL as a combinator library. But there are cases when such approximations are not close enough. Also combinators are not useful when pattern matching. In both these cases, quasi-quoters enable programmers to insert program fragments written in the concrete syntax of the DSL, and the compiler will translate these into Haskell expressions or patterns.

Haskell quasi-quoters are functions that provide the translation from text to code. As such they are first class citizens of the language. Since multiple quoters can be in scope at any given quotation site, each quote is labelled with the name of the quasi-quoter to use. If q is quasi-quoter, then we can write the declaration $x = [q \mid \phi \mid]$ and the compiler applies the translation function in q to the String containing the text ϕ. This produces a metaprogram of type Q Exp which is spliced by Template Haskell, replacing the quasi-quote. Note that this generalised type of quoters does not have a universal anti-quotation operator like the built in Template Haskell quasi-quoter does. Instead the programmer of each quasi-quoter must define the syntax for anti-quotation and the proper translation of the anti-quoted text into Haskell expressions and patterns. Many Haskell quasi-quoters don't support anti-quotation at all, meaning they are really only quoters (and not actually "quasi"). In the rest of this paper we use the term quasi-quoters to refer to any quoters regardless their anti-quotation support.

```
{-# LANGUAGE QuasiQuotes, TemplateHaskell #-}             1
import Language.LBNF                                      2
bnfc [lbnf |                                              3
RAlt.  Reg1 ::= Reg1 "|" Reg2;                            4
RSeq.  Reg2 ::= Reg2 Reg3;                                5
RStar. Reg3 ::= Reg3 "*";                                 6
REps.  Reg3 ::= "eps";                                    7
RChar. Reg3 ::= Char;                                     8
_.     Reg  ::= Reg1;                                     9
_.     Reg1 ::= Reg2;                                     10
_.     Reg2 ::= Reg3;                                     11
_.     Reg3 ::= "(" Reg ")";                              12
|]                                                        13
example :: Reg                                            14
example = RStar (RAlt (RChar 'a') (RChar 'b'))            15
```

Figure 1.1: Basic usage of the Language.LBNF module

2.1 Running example

In this section we demonstrate the use of BNFC-meta, and explain how this differs from the original BNF Converter. Our object language represents regular expressions (it is actually a subset of the regular expression syntax of LBNF). Henceforth we refer to this language as Reg and it is used as a running example in the rest of the paper. The language has six syntactic constructs: choice, concatenation, repetition (Kleene star), empty string (epsilon), single characters and parentheses. These constructs all reside in a single grammar category, that we also call Reg. Since there are infix binary operators (choice and concatenation) we need to indicate precedence and associativity. In LBNF this is done by using *indexed categories*. Figure 1.1 (lines 4–12) shows the grammar of Reg, divided into three levels of precedence indexed 1,2 and 3 (the subscripting is just for readability, in the actual source the indices are plain text). We label the first five syntactic constructs (RAlt, RSeq, RStar, REps and RChar). The other rules (including the parenthesis rule) have no semantic importance and are not labelled; an underscore is placed instead of a label to indicate this.

Note that in Figure 1.1, we have embedded the grammar into a Haskell module using a quasi-quoter (named lbnf) and the value produced by the quoter is passed to the metaprogram bnfc. The end result is that the code produced by bnfc is spliced by the compiler, replacing the grammar definition. The details of this embedding are covered in §3. With the original BNF Converter, we would need to put the grammar in a `.cf` file and run the BNFC tool instead. Doing so would produce a lexer module, a parser module, a printing module and an abstract syntax module (each module in a separate output file). The module we have defined, on the other hand, can be loaded into the GHC interpreter just like any other module and the generated tools for Reg are readily available:

```
Ok, modules loaded: Main.
*Main> :i Reg
data Reg = RAlt Reg Reg | RSeq Reg Reg
         | RStar Reg | REps | RChar Char
instance Eq Reg
instance Ord Reg
instance Show Reg
instance Print Reg
```

The datatype Reg is the abstract syntax type extracted by BNFC-meta. Note that all the indices are ignored for the purpose of building the AST types, so the different levels of precedence are not reflected here. As shown in Figure 1.1 we can also write code that uses this abstract syntax directly in the module (in the definition of example). The Print instance for Reg (indicated above) provides a pretty printer:

```
*Main> printTree example
"('a' | 'b')*"
```

There is also a lexer for the grammar and a parser for each category. The name of the parser is the name of the category prefixed by the letter p and the name of the lexer is tokens:

```
*Main> pReg (tokens "'a' | 'b' *")
Ok (RAlt (RChar 'a') (RStar (RChar 'b')))
```

Quoters for free Apart from the embedding itself, the major addition in BNFC-meta compared to the original BNFC is the automatic generation of quasi-quoters. This feature generates a quasi-quoter for each category of the grammar (the name of the quasi-quoter is the name of the category, but with initial lowercase, so in our example we have a quoter named reg). Abstract values in the language may thus be specified using the concrete syntax. Because of the Template Haskell stage restrictions, we can not use the quasi-quoters directly in the same module. But if we import the module in which we defined the grammar we can write code like this:

```
r1, r2 :: Reg
r1 = [reg | 'a' * 'b' * 'c'* |]
r2 = RSeq (RSeq (RStar (RChar 'a'))
                (RStar (RChar 'b')))
          (RStar (RChar 'c'))
```

Here r_1 and r_2 are exactly equal, but r_1 is defined using concrete syntax whereas r_2 uses abstract. Note that the parsing of the regular expression syntax occurs at compile time, so there is no run time overhead and no risk of run time errors.

Anti-quoting We can also generate *patterns* with the reg quoter:

```
isEps₁, isEps₂ :: Reg → Bool
isEps₁ [reg | eps |] = True
isEps₁ _             = False

isEps₂ REps = True
isEps₂ _    = False
```

As you can see the beautification we saw in the expression example is absent here, in fact the value of using a quoter at all for this function is very questionable. The reason for this is that we can only generate closed expressions or patterns with the reg quoter, so we can only build constant patterns that do not bind any variables. It is difficult to think of a useful constant pattern which is more advanced than the one in isEps. In fact we cannot even make a corresponding function that checks if the argument is a choice (RAlt) without binding variables.

To express patterns with variable bindings, or expressions that use variables, there needs to be a facility to inject Haskell code into the generated expressions. In other words we need to extend the quasi-quoters with anti-quoting. To add anti-quoting to Reg we need to determine a syntax which does not clash with our other syntactic constructs. For instance we can use % as the anti-quoting operator for single identifiers, so that %x is translated into the Haskell identifier x. In a pattern context this means that x becomes bound to a value, and in an expression context it references which ever x is in scope at the quotation site. This anti-quotation allows definitions like these:

```
plus :: Reg → Reg
plus r = [reg | %r %r*|]

xs :: Reg
xs = plus [reg | 'x' |]
```

where xs evaluates to the abstract syntax of the Reg expression `'x' 'x'*`. To express anti-quotation in the grammar, we introduce a special label for grammar rules in BNFC-meta: $. We can use this together with the built in Ident category[1] to add anti-quotation to Reg:

```
$. Reg₃ ::= "%" Ident;
```

[1] Ident is a predefined LBNF-category for identifiers. We could also roll our own identifiers or use any other category to syntactically define allowed Haskell expressions.

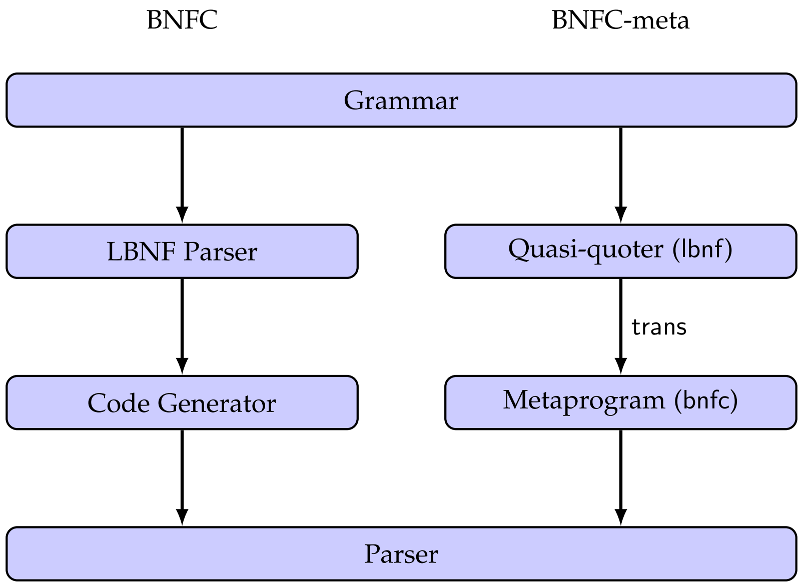

Figure 1.2: A comparative overview of BNFC and BNFC-meta. The annotation trans indicates programmer defined transformations

This adds exactly the anti-quotation mode we want to the Reg parser and the example above works as intended. Replacing the label with the dollar sign indicates to BNFC-meta that this is an anti-quotation rule. When the quasi-quoter parses using this rule, the concrete syntax of the Ident is sent to a Haskell parser and the resulting expression (or pattern) is used instead of an abstract syntax expression. Adding this anti-quotation does not change the parser pReg nor does it change the abstract syntax type Reg.

3 Embedding BNFC

This section discusses the process of elevating BNFC from a command line tool into a metaprogram, and the advantages of doing so. We also show that the process can be generalised and applied to any tool that satisfies certain criteria. Figure 1.2 gives an overview of the basic changes in BNFC-meta compared to BNFC. The benefits of embedding BNFC into Haskell are:

- Libraries that use BNFC-meta do not depend on any installed applications other than the Haskell compiler, hence they do not require custom install scripts. An alternative way to achieve this is to include the BNFC-generated code in the library; but that is generally bad practice because it makes the source more difficult to understand and it invites collaborators to edit the generated code directly, as opposed to properly changing the source grammar.

- Like with any other embedding, the features of the host language (Haskell) are available to the embedded one. Since the Grammar type is exported by Language.LBNF it is possible to write functions that combine, manipulate or analyse grammars at compile time, before they are spliced by the bnfc function.

While the importance of the first feature should not be underestimated, it has no groundbreaking impact on the process of defining languages. The second one has more substantial benefits and is the focus of the remainder of this section.

3.1 Parser generators and combinators

Traditionally, parser generators are the dominating software tools for defining parsers. Most parser generators use some domain specific language for defining grammars, and from such definitions they produce a much more verbose parser implementation in a particular language (the target language). Sometimes a parser generator can use the same grammar to generate parsers in several different target languages. Parser generators can also generate other useful language tools such as pretty-printers and abstract syntax tree types, if the grammar DSL carries enough information (Ranta, 2004; Forsberg and Ranta, 2003).

In functional programming there is a compelling alternative to parser generators, known as parser combinators. Here the grammar DSL is embedded in the target language, using ordinary language features such as higher order functions to construct parsers by combining simpler ones. Major advantages of parser combinators include the familiarity to the programmer (the parser is written in the same language as the code that uses it) and the comfort with which parsers can be developed (no need to generate code between test-runs). Also all the features of the target language are available when defining the parsers so general patterns for eliminating code duplication can be applied, which might not be possible in a standalone grammar DSL. But parser combinators also have several downsides compared to parser generators:

- A parser generator (especially one that is limited to context free object languages) can analyse grammars and detect ambiguities and other anomalies statically. With parser combinators these errors are either detected at runtime, or not detected at all (resulting in unpredictable behaviour).

- Combinator parsing has generally lower performance compared to parser generators, often relying heavily on backtracking or requiring the user to perform manual optimisations.
- Since combinator libraries usually rely on the recursion mechanism of the target language, they typically can't deal with left recursive grammars (which causes an infinite expansion and subsequently failure to terminate). Most grammars for real object languages rely on left recursion at some point, but any grammar with left recursion can be rewritten into an equivalent form without it (Moore, 2000). The rewriting might not be obvious though, and may impair the readability of the grammar considerably.
- Sometimes parser combinator libraries have a significant syntactic overhead compared to the streamlined grammar DSLs used by parser generators.

While BNFC-meta is definitely a parser generator (the parsers are generated Haskell code), it has several of the advantages of parser combinators. Like parser combinators it is available as a library instead of an application, thus it does not require any other tools than the compiler. More importantly, grammars are *almost* first class citizens of the target language (Haskell). We write almost first class because the grammars are only values at compile time, and sometimes this kind of values are referred to as second class citizens. This means that users can apply arbitrary Haskell functions to the grammar after parsing it with lbnf but before splicing it with bnfc. In Figure 1.2 this is indicated by the "trans" annotation in the flow chart. In practice this means you can write code like this:

```
import OtherModule (f)   -- f :: Grammar → Grammar
bnfc (f [lbnf | Γ |])
```

The Template Haskell stage restrictions prevent f from being defined (at top level) in the same module as the splice. Note that f is evaluated at compile time so if it fails then a compile time error is raised (see §4.4 for a discussion on error messages). In this case the type of f indicates that it is a grammar transformer but one could also have functions that do not take an existing grammar as an argument (i.e. constant grammars) or one that takes some other parameter like a list of operators and constructs a grammar from that (i.e. parameterised grammars).

Example: grammar reuse In the original BNF Converter, the only way to extend an existing grammar is to copy the file and add the required rules. This procedure is very pervasive from a maintenance perspective. In BNFC-meta on the other hand, grammars are Haskell values (at compile time) and can thus be manipulated using all the features of Haskell (including a rich module system).

```
module RegexGrammars where
import Language.LBNF
import Language.LBNF.Grammar
combine :: Grammar → Grammar → Grammar
combine (Grammar i) (Grammar j) = Grammar (i ++ j)
minimal, extended :: Grammar
minimal = [lbnf |
   RAlt.  Reg1 ::= Reg1 "|" Reg2;
   RSeq.  Reg2 ::= Reg2 Reg3;
   RStar. Reg3 ::= Reg3 "*";
   REps.  Reg3 ::= "eps";
   RChar. Reg3 ::= Char;
   _.Reg   ::= Reg1;
   _.Reg1  ::= Reg2;
   _.Reg2  ::= Reg3;
   _.Reg3  ::= "(" Reg ")";
 |]
extended = combine minimal [lbnf |
   RPlus.     Reg3 ::= Reg3 "+";
   ROpt.      Reg3 ::= Reg3 "?";
 |]
```

Figure 1.3: Grammar reuse

Suppose that we want to create an extended version of our Reg grammar (from Figure 1.1) that has the postfix operators $+$ (non empty repetition) and ? (optional). First we factor the grammar out of the module in which it is defined into a new module. Then, instead of applying bnfc to the grammar, we give it a top level name. Thus bnfc [lbnf | Γ |] becomes minimal = [lbnf | Γ |]. The type of minimal is Grammar which is just a wrapper around a list of grammar rules. Figure 1.3 demonstrates how we exploit this fact to create a crude function for combining two grammars, and how this function is used to define the extended grammar in terms of the original one.

The extended parser is defined by importing RegexGrammars and splicing in bnfc extended. If we still want to use the original language we can splice bnfc minimal. Both languages can not be spliced into the same module however, since they define datatypes and values with overlapping names. In §4 we will come back to the combine function to add anti-quotation support to our extended grammar (see Figure 1.5).

3.2 A general method for embedding compilers

The principle behind the embedding is remarkably simple. Like most compilers BNFC has two distinct components:

- A front end, corresponding to a function of type String $\rightarrow$ Grammar, where the string is the grammar written by the user (the concrete syntax) which is parsed into a value of type Grammar (an abstract syntax tree).
- Several back ends, each producing a parser written in a specific programming language. The Haskell back end can (somewhat simplified) be thought of as a function Grammar $\rightarrow$ String which takes the grammar of the parser and produces concrete Haskell syntax.

Conveniently, the front end corresponds exactly to the lbnf quasi-quoter. All that is needed to construct lbnf is a function that converts a Grammar into a Haskell expression of type ExpQ, such that the expression evaluates to the given grammar value. The function is trivial but a bit verbose. One method of doing this automatically for any type (using generic programming) was presented by Mainland (2007).

Likewise, the back end corresponds exactly to the bnfc function. The only difference is that BNFC produces concrete Haskell syntax (String) whereas we need abstract syntax (DecsQ). The quickest way of coding this conversion from concrete to abstract syntax is to plug in a Haskell parser. A more elegant way might be to alter the BNFC back end.

This simple technique can be generalised to embed any compiler under the conditions that 1) the target language of the compiler is Haskell and

2) the compiler is implemented in Haskell. The first requirement enables Template Haskell to splice the resulting code into a Haskell module. The second requirement means that the code from the original application can be put directly into a library. Note that it is not required to have separated front and back-ends. If the application only provides a function α :: String → String we can just consider the abstract syntax tree type to be String. The front end quoter may then be built from the identity function, and the back end may be built from α. If we have no distinction between front and back ends we do not get the programmer defined transformation however (since these are applied after front end processing but before back end processing).

Embedding BNFC and friends In practice it proved complicated to use the general method for BNFC because the back end does not only produce Haskell code. BNFC also produces intermediate code for the Happy parser generator and the Alex lexer generator, violating the first condition for applying our method. When using BNFC this means that after processing your grammar with BNFC, you will need to process parts of the output with Happy and parts of it with Alex. This is not desirable in a library setting, especially since avoiding external software is one of the perks of the embedding.

The generality of our approach enabled us to overcome this problem, by embedding Alex and Happy in much the same way as we embedded BNFC. Alex and Happy 1) both produce Haskell code and 2) are both written in Haskell, so they satisfy the criteria for embedding. The result of applying the embedding method to these programs are two new libraries: happy-meta and alex-meta. Similar to BNFC-meta, these libraries allow users to write Happy and Alex syntax directly into Haskell modules using quasi-quoters, and splice the result into the module using Template Haskell. As a by-product, they provide the functions necessary to adapt BNFC to the first condition (by composing the back end of BNFC with the front ends of Alex and Happy). Although we used the "quick and dirty" approach (parsing code we have generated ourselves) this is only a performance issue at compile time, the produced code is essentially identical to the code produced by the original tools. We have made the three resulting libraries (BNFC-meta, happy-meta and alex-meta) available through Hackage.

4 Quasi-quoters

In this section we explain in more detail how the quasi-quoters are generated by BNFC-meta and what separates them from previous methods of specifying quasi-quoters.

4.1 Where do quoters come from?

All quasi-quoters, like reg and lbnf, are of the type QuasiQuoter which is a record type containing at least an expression and a pattern quoter (recent versions of the Template Haskell library also include type and declaration quoters).

```
data QuasiQuoter = QuasiQuoter
  {quoteExp :: String → Q Exp
  , quotePat :: String → Q Pat
  }
```

The quote [q | ϕ |] is equivalent to splicing the expression produced by quoteExp q ϕ if the quote is in an expression context, or quotePat q ϕ in a pattern context. To explain the basic quoters (that only generate abstract syntax trees) we can restrict ourselves to this tiny subset of the Template Haskell library API:

```
mkName :: String → Name
data Exp
  = ConE Name        -- Constructors
  | AppE Exp Exp     -- Application
  | LitE Lit
  |  ...
data Pat
  = ConP Name [Pat]   -- Constructor application
  | LitP Lit
  | ...
data Lit = CharL Char | ...
```

With these definitions we can simply "lift" a parsed value into an expression that evaluates back to the given value, or a pattern that matches only the given value. For instance `"'a'"` is parsed to RChar `'a'`, which as an expression is

```
AppE (ConE (mkName "RChar")) (LitE (CharL 'a'))
```

and as a pattern

```
ConP (mkName "RChar") [LitP (CharL 'a')]
```

Lifting values in this way is a completely mechanical task, and Mainland (2007) defines datatype generic functions that perform the lift automatically (for any AST type that is a member of the Data typeclass).

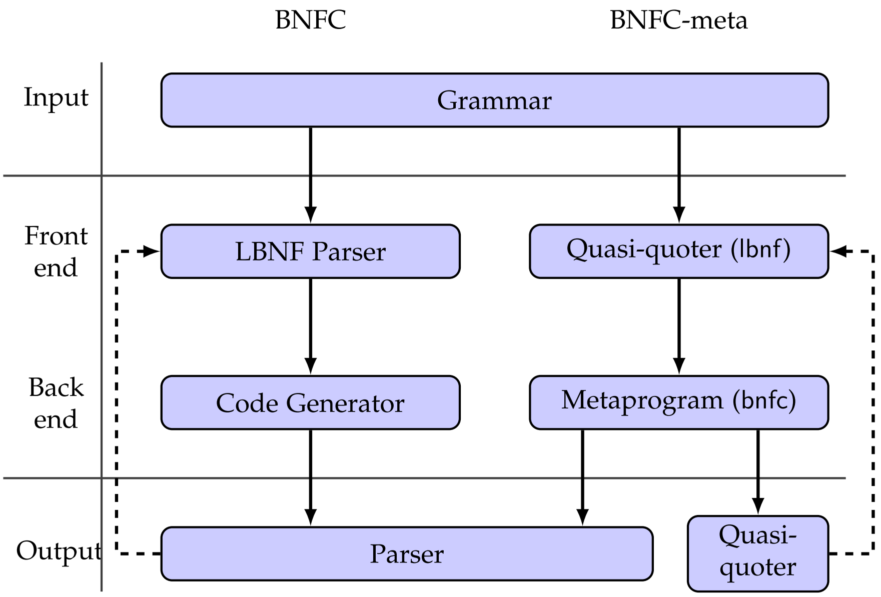

Figure 1.4: A comparative overview of BNFC and BNFC-meta. Dashed lines indicate bootstrapping capability

4.2 Anti-quoting

The obvious use of anti-quotation in our Reg example is to allow any subexpression to be replaced by a snippet of Haskell code. In our example in §2.1 we introduced a quasi-quoting operator for single identifiers. For this example we choose to allow a slightly larger subset of Haskell, with function applications and identifiers in the anti-quoted expressions. We encapsulate all anti-quoted code in curly brackets. This allows us to write expressions like [reg | {f r}∗|] that translate into RStar (f r).

The standard method for adding anti-quoting was introduced by Mainland (2007). The method consists of changing the parser of the language to support the anti-quoting syntax, and then using the same kind of lifting from values to metaprograms as we do for the basic quoters. In our case this would mean changing the grammar in much the same way as in our earlier anti-quoting example (in §2.1), but without using the special anti-quoting label:

RegAQ. Reg_3 ::= "{" [Ident] "}";

Now we can parse Regs with anti-quoted code, but there are two problems:

- When lifting parsed values to expressions or patterns we must add a special case for the constructor RegAQ. These values should not be lifted, instead the list of Idents should be translated into a function application (or a constructor application in a pattern context).

- The abstract syntax for Reg now contains a constructor that can hold Haskell expressions. Functions that operate on the Reg type must now assume that this constructor is never used at run time, or handle it appropriately if it does.

The first issue is solved by Mainland (2007) by extending the generic lifting functions with type-specific exceptions for anti-quoting. The second issue is more difficult. Without access to open datatypes there is no way of reusing the abstract syntax and parser without compromising type safety.

In BNFC-meta we solve this issue by generating two parsers from each grammar category. One for run time parsing directly to a specific AST type (pReg in our example) and one for compile time parsing to a generic QuasiAST datatype (named qReg in our example). The vital part of the type is defined like this:

```
data QuasiAST
   = AstConApp Name [QuasiAST]   -- Constructor app.
   | AstLit Lit                  -- Litteral values
   | AstAnti (Q Exp) (Q Pat)     -- Anti-quotation
```

This is essentially a common subset of the Exp and Pat types, and it can be converted into either. Since BNFC-meta only generates abstract syntax types, the first two constructors are sufficient to express any expression or pattern that we may generate with the basic quoters. The AstAnti constructor allows the injection of arbitrary code by supplying it both as a pattern and an expression (although only one of these are evaluated depending on the context of the quote). When our grammar is compiled, qReg :: String → QuasiAST is automatically generated. It is trivial to define a function toQuoter :: (String → QuasiAST) → QuasiQuoter, and this gives us the definition of the quoter for the Reg category: reg = toQuoter qReg.

Now we can explain the semantics of the special anti-quoting label ($). As before, just replacing the label of our rule will magically do what we want:

```
$ . Reg3 ::= "{" [Ident] "}";
```

More generally, the $ is followed by an optional identifier and omitting the identifier as above is equivalent to this:

```
$ printAq. Reg3 ::= ...
```

The identifier must be a function of type $\Delta_1 \dots \Delta_n \rightarrow$ QuasiAST where Δ_k is the AST type of the k:th non terminal in the right hand side of the rule (so in our case [Ident] → QuasiAST). The function has much the same purpose as the type specific exceptions in the generic programming approach from

```
topdown :: (Reg → Reg) → Reg → Reg
topdown f rx = case f rx of
  [reg | %e1   % e2 |] → [reg | {r e1}   {r e2} |]
  [reg | %e1 | % e2 |] → [reg | {r e1} | {r e2} |]
  [reg | %e1 *      |] → [reg | {r e1} *        |]
  [reg | %e1 +      |] → [reg | {r e1} +        |]
  [reg | %e1 ?      |] → [reg | {r e1} ?        |]
  e                    → e
  where r = topdown f

transform :: Reg → Reg
transform = topdown step where
  step rx = case rx of
    [reg | %e | eps     |] → [reg | %e ?   |]
    [reg | %e1  % e2 * |]
        | e1 ≡ e2          → [reg | %e1 + |]
        | otherwise        → rx
    e                      → e
```

Figure 1.5: Implementing a program transformation on regular expressions

Mainland (2007), it specifies how a parsed value can be translated into Haskell code. The difference here is that the [Ident] is never placed in the abstract syntax tree, which means that the abstract syntax type Reg and the original parser pReg are totally unaffected by the introduction of the anti-quotation syntax.

The default anti-quotation function takes the result of pretty printing the non-terminal on the right hand side, and parses it with a Haskell parser:

```
printAq :: Print a ⇒ a → QuasiAST
printAq = stringAq ∘ printTree

stringAq :: String → QuasiAST
stringAq s = AstAnti (parseExp s) (parsePat s)
```

Although we use a very restricted subset of Haskell in this example we could just as well accept a superset, e.g. by defining a grammar category for strings starting with `'{'` and ending with `'}'`. Defining this category in LBNF involves making a custom lexical token, which we do not cover in this paper. Using a Haskell superset also moves some syntax error handling from our parser to the Haskell parser used for anti-quotes.

In Figure 1.5 we use both our anti-quotation operators to 1) define a general top-down traversal function on regexps and 2) use the function to transform regular expressions that do not make use of + and ? into equivalent ones that do. Note that the entire module is written without assuming anything about the abstract syntax other than the existence of a category named Reg. Thus, the names of constructors and other grammar details can be freely changed as long as the concrete syntax remains the same. This appeals to the saying that concrete syntax is more abstract than abstract syntax.

Bootstrapping The original BNF Converter is bootstrapped, "eating its own dog food". By this we mean that it generates its own front end: the syntax of LBNF is specified as an LBNF grammar, and feeding this grammar to BNFC produces the front end modules of the BNFC implementation. The addition of automatically generated quasi-quoters preserves this property[2]: the quasi-quoter lbnf can be automatically generated by processing the grammar for LBNF in BNFC-meta. Just like BNFC can generate a complete front end for a traditional compiler, BNFC-meta can generate a front end for an embedded one. Figure 1.4 shows a comparison between the components of BNFC and BNFC-meta, and the dotted lines indicate the bootstrapping capacity of each.

4.3 Example: Monadic quoters

A very frequent situation when dealing with abstract syntax is to traverse a syntax tree in a monadic computation, e.g. carrying an environment or a state. In a non-monadic setting we can define a recursive case of a transformation on Reg in a simple way:

```
transform :: Reg → Reg
transform [reg | {r1} {r2} |] = [reg | {transform r1} {transform r2} |]
```

If we carry an environment the transformation becomes much more verbose, and we can not make use of the fact that we allow function application in anti-quotes:

```
transform :: Reg → Env Reg
transform [reg | {r1} {r2} |] = do
  r1' ← transform r1
  r2' ← transform r2
  return [reg | {r1'} {r2'} |]
```

The problem here is that an expression such as $\lambda r \rightarrow$ [reg | {r}∗|] has type Reg → Reg but we want to lift it to Monad m ⇒ m Reg → m Reg. In general it is not possible to lift a given quasi-quoter like this because we can not separate the monadic components from non-monadic in the resulting

[2]The concept of bootstrapping is slightly misleading in a library context, since libraries cannot depend on previous versions of themselves. In BNFC-meta this is solved by adding a small bootstrapping utility that flushes the produced code into a file instead of splicing it.

expression. The QuasiAST type on the other hand, supplies exactly this distinction: all the anti-quoted values are monadic and the rest are non-monadic. A completely generic function for building monadic quoters can be written as follows (note that the monadic nature of the code is not reflected in the type signatures here, the change is only in the type of the *generated* code):

```
toMonQuoter :: (String → QuasiAST) → QuasiQuoter
toMonQuoter f = QuasiQuoter {
  quoteExp = toMonExp ∘ f,   -- Monadic expr. lifting
  quotePat = toPat ∘ f       -- Ordinary pattern lifting
  }
toMonExp :: QuasiAST → Q Exp
toMonExp q = case q of
  AstConApp con qs → foldl mAppE
                        (returnE (conE con))
                        (map toMonExp qs)
  AstAnti e _      → e   -- We assume e is monadic
  AstLit l         → returnE (litE l)
returnE :: Q Exp → Q Exp
returnE e = [| return ($e) |]
  -- Monadic function application
mAppE :: Q Exp → Q Exp → Q Exp
mAppE mf ma = [| ($mf) >>= flip liftM ($ma) |]
```

We can now define mReg = toMonQuoter qReg and the new quoter mReg has exactly the property we needed.

4.4 Error handling

Using metaprogramming in general, and quasi-quoters in particular, adds a new dimension to compile time error handling. There are no problems with errors "slipping through" to run time: all parsing is performed at compile time and the generated code is type checked after it is spliced. What can be problematic is the precision of the error messages, with regards to source location. The compiler will automatically indicate which quasi-quote the error occurred in, anything beyond that has to be provided by the programmer of the quasi-quoter. Because BNFC uses the Alex lexer generator, source locations are already included in the lexer result. This means that when a syntax error occurs, BNFC-meta quasi-quoters can automatically present accurate source locations and show the concrete syntax of the code that causes the error.

The situation is worse for type errors. With the basic quoters type errors can not occur unless there is a bug in BNFC-meta. In the presence of anti-quoting however, the user is free to introduce any imaginable type errors in the generated code. BNFC-meta has little control over the error messages in these situations, and they will be presented without an accurate source position and in terms of the generated code. As an example, consider the monadic quoters we introduced in §4.3. It relies on the type inference system to determine which monad to use. A constant value is polymorphic: $[\mathsf{mReg} \mid \mathsf{eps} \mid] :: \mathsf{Monad}\ \mathsf{m} \Rightarrow \mathsf{m}\ \mathsf{Reg}$. When anti-quoting the monad type is determined by the type of the anti-quoted code. If non-monadic code is accidentally used, a no-instance error arises and the error message informs us that it occurs in the second argument of $\gg=$, leaking the generated code to the user. This is a problem for quasi-quoters in general and not specific to BNFC-meta.

5 Performance and scalability

This section provides some examples and experimental results. All tests where performed on a 3.0 GHz Intel Xeon processor. Version 6.12.1 of the Glasgow Haskell Compiler was used, with optimisation level `-O2`.

5.1 Performance analysis

When using the quasi-quoters generated by BNFC-meta, performance is not a major issue (since the parsers are only used at compile time, when quasi-quotes are resolved). For applications that use the parsers at runtime there may be some requirements though. Since BNFC-meta is based on Alex and Happy the performance should be comparable to that of hand written parsers using them, which is usually considered good.

There are not many reliable performance comparisons between parser generators and parser combinators, possibly because it is difficult to make a fair comparison when the choice of object language and the amount of work put into each parser is very relevant to end performance. The general understanding seems to be that parser generators are faster. Parsec (Leijen and Meijer, 2001) is considered a fast parser combinator library and it is seeing heavy use in the Haskell community.

For this reason we decided to compare the performance of Parsec and BNFC-meta. In order to avoid bias in the choice of object language or implementation of the Parsec parser we decided to use an existing one and write an equivalent LBNF grammar for comparison. There are a number of examples included in the distribution of Parsec 2, some of them where excluded because they do not have context free syntax and as such can not be implemented in BNFC-meta. From the remaining, the largest example language was chosen and it proved to be a language called *Mondrian*. The source code for the Parsec parser (and abstract syntax) is a few hundred lines of Haskell code and the BNFC-meta implementation we

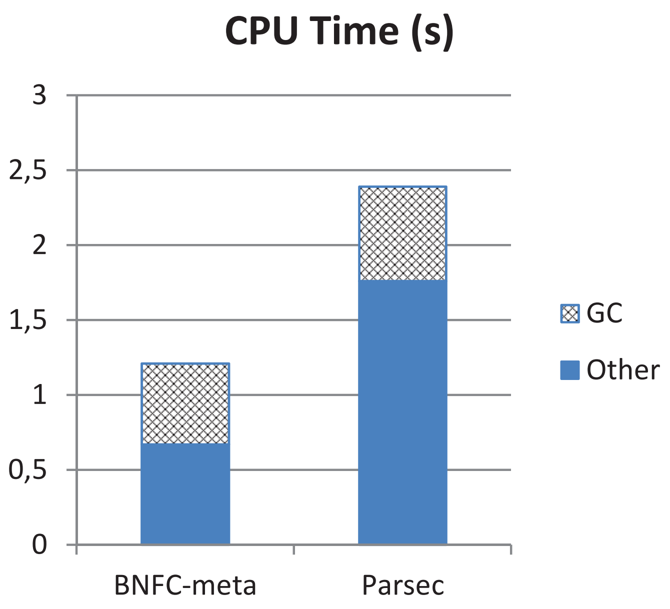

Figure 1.6: Average CPU usage when parsing a Mondrian file

wrote is around 50 lines of Haskell/LBNF code. The two implementations were considered “similar enough” to be comparable when the BNFC-meta parser could parse all the examples that the Parsec parser could, and the level of detail of the abstract syntax types seemed equivalent in a quick inspection. Figure 1.6 shows the relative CPU time usage. The times are measured using the criterion benchmark tool. A very large Mondrian file (1 MB) was used in the test to ensure measurable time consumption. Initially we speculated that the time difference was caused by a difference in memory usage and specifically by garbage collection (GC), but this was not confirmed by measurements which showed only a slightly increased GC activity for Parsec.

Compilation time Since BNFC-meta does compile time calculations it is reasonable to assume that the compilation time should be somewhat greater than for Parsec. Also the amount of code that BNFC-meta produces can be quite big and the compiler needs to process and optimise it, which could increase compilation times further. This hypothesis is confirmed by our test results (see Figure 1.7), with the BNFC-meta parser taking twice as long to compile as the Parsec one.

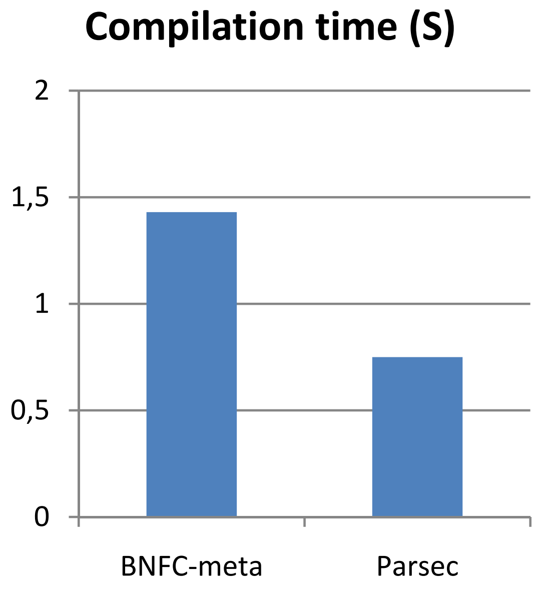

Figure 1.7: Compilation time of a Mondrian parser

5.2 Expressiveness

When comparing to flexible parser generators and combinator libraries such as Parsec and hand written Alex+Happy code it is important to note that BNFC-meta is limited to defining the same set of object languages as the original BNFC. This set is essentially all context-free languages and some context sensitive extensions like a limited kind of layout syntax (by automatic preprocessing). This means that it is difficult to capture many real world languages exactly using LBNF.

The fact that BNFC generates a lexer automatically and that this lexer does not distinguish newlines from white spaces imposes some further restrictions. Specifically it is not trivial to implement object languages with significant newline characters, which is a common feature. In principle it should be possible to give the user stronger control over the lexer without sacrificing the simplicity in cases where this is not needed. In spite of these limitations LBNF is *not* limited to "toy languages", as we will demonstrate below.

5.3 A large scale example

In order to fit a complete definition, the running example in this article is very simple (only a single grammar category Reg). Nonetheless, the applications can scale up to fully equipped real world languages. To demonstrate this, we applied BNFC-meta to the grammar of (pre-processed) ANSI C available from the BNFC webpage. The grammar is roughly 250 LBNF rules so the module is too large to include in this paper, but not much larger than an average Haskell module. By default BNFC-meta generates quasi-quoters for all entry point categories (or for all categories if no entry point is specified in the grammar). In this case it means we are given quasi-quoters for C programs, statements and expressions. Using the program quoter we can define a C program using concrete syntax:

```
p :: Program
p = [program |
int main () {
  printf ("Hello World");
  return 0;
}
 |]
```

The "least pretty equivalent" to this definition (which only uses the abstract syntax constructors) looks like this:

```
p :: Program
p = Progr [Afunc (NewFunc [Type Tint] (NoPointer
  (OldFuncDec (Name (Ident "main")))) (ScompTwo
  [ExprS (SexprTwo (Efunkpar (Evar (Ident "printf"))
  [Estring "Hello World"])), JumpS (SjumpFive
  (Econst (Eoctal (Octal "0"))))])))]
```

Anti-quoting There is an existing open source quasi-quoter library for C, called language-c-quote (Mainland, 2009). It offers a kind of "typed" anti-quotation where expressions like

```
[cexp | $int : a + $exp : b |]
```

means that a and b must be Haskell identifiers. Furthermore a must be an integer (i.e. of type Int) and b must be a C expression (i.e. of the same type as the quote).

We want to add a more liberal quotation syntax that allows more advanced Haskell expressions (not just single identifiers) in the quotes. We chose to have two anti-quotation modes for each category:

- An explicitly typed mode where [C : h:] is a member of the category C for any Haskell expression h of type C (h may not contain the string ":]" and may not contain nested quasi-quotes). Note the syntactic similarity with quasi-quoters.

```
import C
zeroFill :: Integer → [Ident] → Stm
zeroFill n arrs = [stm |
{
  int tmp;
  for (tmp = 0; tmp < [Integer : n:]; tmp++) {
    [:map zeroOneSlot arrs:]
  }
}
|]
zeroOneSlot :: Ident → Stm
zeroOneSlot arr = [stm |
  [Ident : arr:] [tmp] = 0;
|]
```

Figure 1.8: Using the C grammar with anti-quoting

- An implicitly typed mode where the category name is dropped meaning [:h:] is a member of all categories.

Figure 1.8 demonstrates how these are used to define a program that is parameterised over an array length n and a list of identifiers (names of arrays), such that the produced C program fills each array with n zeroes. Note that the program deals only with the abstract syntax of C at runtime, so the Haskell type system statically guarantees that the program will produce syntactically correct code independent of the arguments to zeroFill. The second anti-quotation mode is controversial because it introduces massive ambiguities in the grammar. For instance if we have a quasi-quoted C statement [stm | [:h:] |] then the anti-quote may refer to a statement, an expression or even an integer. The decision is made by the parser generator (automatically resolving ambiguities) but the user is warned of the ambiguities. In most cases assuming the wrong category will simply incur a type error, but if h is polymorphic there may be serious problems. One solution is to avoid the implicitly typed mode if the expression is polymorphic.

Observe that we describe the anti-quotation without mentioning that C is our object language. In fact this description of anti-quotation is expressed only in general terms of the existing grammar categories and as such it can be applied to any grammar. This has a tremendous potential for code reuse: instead of adding this mode of anti-quotation to the C grammar (and write a hundred or so grammar rules by hand) we can define a grammar transformation that adds this anti-quotation mode to any grammar. We can even parameterise the grammar transformation over the exact syntax of the anti-quotes; how quotes are initiated, what delimits the optional category name from the Haskell expression and how quotes are ended. The grammar transformation is thus a function:

```
antiquote :: String → String → String → Grammar → Grammar
```

And to add the desired anti-quotation to the C grammar we simply interpose this function between the quasi-quoter front end and the splicing back-end:

```
bnfc $ antiquote "[" ":" ":]" $ [lbnf |
... the C Grammar ...
|]
```

5.4 Implementing a general anti-quotation mode

The implementation of antiquote adds two things to the grammar: a lexical token (which is a grammar category built from a regular expression) and two anti-quotation rules for each category. The individual rules contain the start string "[" and optionally the name of the category, the token matches the delimiter ":" an arbitrary string and the ending string ":]". The LBNF rules for a single category C looks something like this:

```
$ g ::= "[C" AntiQuotingToken;
$ f ::= "[" AntiQuotingToken;
```

where AntiQuotingToken is the shared anti-quotation token. The functions g and f need to do the appropriate pruning of the matched string before the Haskell expression is parsed. Defining these rules is a simple task of collecting the categories of the original grammar, defining a set of new rules from these and adding this set to the original grammar.

While we would not go so far as to say that the generated quasi-quoters are identical to the ones provided by the language-c-quote library (such an analysis would not be practical), they are certainly quite similar. The most striking difference is the simplicity of the BNFC-meta implementation. Where language-c-quote has several hundred or even a few thousand lines of code, our implementation has a single Haskell source file with a readable and syntax-directed grammar definition. We also obtain a reusable pattern for anti-quoting and even for this single example the definition of this reusable pattern is shorter than the manual changes we would otherwise have needed.

6 Discussion

The closest predecessor to our work is the Quasi-quoting paper by Mainland (2007) where he gives the basic infrastructure to support user-defined generalized quasi-quoting in GHC. We build on this work by demonstrating how essential quasi-quotation is when converting a utility program (in our case BNFC, Alex and Happy) to a metaprogram because it allows the syntax of the original tool to be preserved completely. We also show that limiting quoters to generate ASTs of context free languages has several advantages over the general quoters:

- They can be given general anti-quotation operators.

- They can be transformed into variant quoters, including but not limited to monadic quoters.

We derive much of the strength of our tool from the implementation of the BNF Converter (Forsberg and Ranta, 2003). We internalise BNFC (+ Happy and Alex for parsing and lexing) and add general support for anti-quoting.

In his recent tutorial on combinator parsing Swierstra (2009) writes "Parser combinators occupy a unique place in the field of parsing: they make it possible to write expressions which look like grammars, but actually describe parsers for these grammars." In this paper we retain this property but also allow "off-line" parser generators to be used at compile time.

There is a long history of metaprogramming in the Lisp and Scheme community, starting already in the sixties. Twenty years ago *META* for Common Lisp used metaprogramming to construct parsers (Baker, 1991). The more recent survey of lexer and parser generators in Scheme (Owens et al., 2004) shows several examples and cites several Scheme-based generators.

From the C++ world, *Spirit* (Guzman and Kaiser, 2011) is a set of libraries (part of The Boost Initiative (2009)) that can build recursive descent parsers, using an embedded grammar DSL. It uses template metaprogramming to generate parsers.

The comparison of DSL implementations by (Czarnecki et al., 2003) looks at the meta-programming support of MetaOCaml, Template Haskell, and C++. All of those systems have the meta-programming strength to support compile time parser generation, but currently only Template Haskell supports quasi-quoting.

There are several attempts at bridging the gap between parser generators and combinators by improving combinator libraries. Devriese and Piessens (2011) attempt to limit the problems related to left recursive grammars in parser combinator libraries. They use Template Haskell to perform some grammar transformation at compile time. In (Rendel and Ostermann, 2010) a technique for getting "pretty-printers for free" from a parser combinator library is described.

6.1 Future work

While working on BNFC-meta we have found several interesting directions for future work. One direction is to work closely on the interface to and the implementation of Template Haskell. Here we would like to allow more control over error messages, especially when the following static checks on the generated code fail. We have also seen the need for generating not just declaration lists, but also module headers, including import and export lists. This would be helpful to avoid exporting some "junk" datatypes produced by Happy. Currently the only way to avoid polluting the name space of the module is to generate definitions in local **where** or **let** blocks, which is bad for error messages. Also we are not sure if the compiler responds well to a two-thousand-lines **where**-clause.

It might be interesting to make a more thorough performance analysis of BNFC-meta and a more conclusive comparison to other parsing frameworks.

Another direction we aim to pursue is to apply BNFC-meta to ongoing DSL projects (the Feldspar (Axelsson et al., 2010) backend, GPGPU programming with Obsidian (Svensson, Claessen, and Sheeran, 2010), language based security, etc.)

Finally we would also like to embed more libraries and tools in the same way as BNFC, Happy and Alex: candidates include the dependently typed language Agda and the Foreign Function Interface tool hsc2hs.

6.2 Conclusions

We use metaprogramming both to embed parser generators and to generate embedded parsers. There is a natural connection between these tasks: the latter provide a suitable front end for the first. This is evident from the bootstrapping ability of BNFC-meta.

As far as we know, BNFC-meta is the only system that provides both embedded parser generation and generation of embedded parsers in a single library.

Paper II

Feat: Functional Enumeration of Algebraic Types

This chapter is an extended version of a paper originally published in the proceedings of the 2012 Haskell Symposium under the same title.

Feat: Functional Enumeration of Algebraic Types

Jonas Duregård, Patrik Jansson, Meng Wang

Abstract

In mathematics, an enumeration of a set S is a bijective function from (an initial segment of) the natural numbers to S. We define "functional enumerations" as efficiently computable such bijections. This paper describes a theory of functional enumeration and provides an algebra of enumerations closed under sums, products, guarded recursion and bijections. We partition each enumerated set into numbered, finite subsets.

We provide a generic enumeration such that the number of each part corresponds to the size of its values (measured in the number of constructors). We implement our ideas in a Haskell library called `testing-feat`, and make the source code freely available. Feat provides efficient "random access" to enumerated values. The primary application is property-based testing, where it is used to define both random sampling (for example QuickCheck generators) and exhaustive enumeration (in the style of SmallCheck). We claim that functional enumeration is the best option for automatically generating test cases from large groups of mutually recursive syntax tree types. As a case study we use Feat to test the pretty-printer of the Template Haskell library (uncovering several bugs).

1 Introduction

Enumeration is used to mean many different things in different contexts. Looking only at the Enum class of Haskell we can see two distinct views: the list view and the function view. In the list view succ and pred let us move forward or backward in a list of the form [start .. end]. In the function view we have bijective function toEnum :: Int → a that allows direct access to any value of the enumeration. The Enum class is intended for enumeration types (types whose constructors have no fields), and some of the methods (fromEnum in particular) of the class make it difficult to implement efficient instances for more complex types.

The list view can be generalised to arbitrary types. Two examples of such generalisations for Haskell are SmallCheck (Runciman, Naylor, and Lindblad, 2008) and the less well-known enumerable package. SmallCheck implements a kind of enumToSize :: $\mathbb{N} \rightarrow$ [a] function that provides a finite list of all values bounded by a size limit. Enumerable instead provides only a lazy [a] of all values.

Our proposal, implemented in a library called Feat, is based on the function view. We focus on an efficient bijective function $\mathsf{index}_{\mathsf{a}} :: \mathbb{N} \rightarrow \mathsf{a}$, much like toEnum in the Enum class. This enables a wider set of operations to explore the enumerated set. For instance we can efficiently implement enumFrom :: $\mathbb{N} \to [a]$ that jumps directly to a given starting point in the enumeration and proceeds to enumerate all values from that point. Seeing it in the light of property-based testing, this flexibility allows us to generate test cases that are beyond the reach of the other tools.

As an example usage, imagine we are enumerating the values of an abstract syntax tree for Haskell (this example is from the Template Haskell library). Both Feat *and* SmallCheck can easily calculate the value at position 10^5 of their respective enumerations:

```
*Main> index (10^5) :: Exp
AppE (LitE (StringL "")) (CondE (ListE []) (ListE [])
                                           (LitE (IntegerL 1)))
```

But in Feat we can also do this:

```
*Main> index (10^100) :: Exp
ArithSeqE (FromR (AppE (AppE (ArithSeqE (FromR (ListE [])))
... -- and 20 more lines!
```

Computing this value takes less than a second on a desktop computer. The complexity of indexing is (worst case) quadratic in the size of the selected value. Clearly any simple list-based enumeration would never reach this far into the enumeration.

On the other hand QuickCheck (Claessen and Hughes, 2000), in theory, has no problem with generating large values. However, it is well known that reasonable QuickCheck generators are really difficult to write for mutually recursive datatypes (such as syntax trees). Sometimes the generator grows as complex as the code to be tested! SmallCheck generators are easier to write, but fail to falsify some properties that Feat can.

We argue that functional enumeration is the only available option for automatically generating useful test cases from large groups of mutually recursive syntax tree types. Since compilers are a very common application of Haskell, Feat fills an important gap left by existing tools.

For enumerating the set of values of type a we partition a into numbered, finite subsets (which we call *parts*). The number associated with each part is the size of the values it contains (measured in the number of constructors). We can define a function for computing the cardinality for each part i.e. card_a :: Part $\to \mathbb{N}$. We can also define select_a :: Part $\to \mathbb{N} \to$ a that maps a part number p and an index i within that part to a value of type a and size p. Using these functions we define the bijection that characterises our enumerations: index_a :: $\mathbb{N} \to$ a.

We describe (in §2) a simple theory of functional enumeration and provide an algebra of enumerations closed under sums, products, guarded recursion and bijections. These operations make defining enumerations for Haskell datatypes (even mutually recursive ones) completely mechanical. We present an efficient Haskell implementation (in §3).

The efficiency of Feat relies on memoising (of meta information, not values) and thus on *sharing*, which is illustrated in detail in §3 and §4.

We discuss (in §5) the enumeration of datatypes with invariants, and show (in §6) how to define random sampling (QuickCheck generators) and exhaustive enumeration (in the style of SmallCheck) and combinations of these. In §7 we show results from a case study using Feat to test the pretty-printer of the Template Haskell library and some associated tools.

2 Functional enumeration

For the type E of functional enumerations, the goal of Feat is an efficient indexing function $\mathsf{index} :: \mathsf{E}\ \mathsf{a} \to \mathbb{N} \to \mathsf{a}$. For the purpose of property-based testing it is useful with a generalisation of index that selects values by giving size and (sub-)index. Inspired by this fact, we represent the enumeration of a (typically infinite) set S as a *partition* of S, where each part is a numbered finite subset of S representing values of a certain size. Our theory of functional enumerations is a simple algebra of such partitions.

Definition 1 (Functional Enumeration)**.** A functional enumeration of the set S is a partition of S that is

- *Bijective*, each value in S is in exactly one part (this is implied by the mathematical definition of a partition).
- *Part-Finite*, every part is finite and ordered.
- *Countable*, the set of parts is *countable*.

□

The countability requirement means that each part has a number. This number is (slightly simplified) the size of the values in the part. In this section we show that this algebra is closed under disjoint union, Cartesian product, bijective function application and guarded recursion. In Table 2.2 there is a comprehensive overview of these operations expressed as a set of combinators, and some important properties that the operations guarantee (albeit not a complete specification).

To specify the operations we make a tiny proof of concept implementation that does not consider efficiency. In §3 and §4 we show an efficient implementation that adheres to this specification.

Enumeration combinators:

empty	$:: \mathsf{E\ a}$
singleton	$:: \mathsf{a} \to \mathsf{E\ a}$
$(\oplus)$	$:: E\ \mathsf{a} \to E\ \mathsf{b} \to E\ (\mathsf{Either\ a\ b})$
$(\otimes)$	$:: E\ \mathsf{a} \to E\ \mathsf{b} \to E\ (\mathsf{a}, \mathsf{b})$
biMap	$:: (\mathsf{a} \to \mathsf{b}) \to E\ \mathsf{a} \to E\ \mathsf{b}$
pay	$:: \mathsf{E\ a} \to \mathsf{E\ a}$

Properties:

index (pay e) i	$\equiv$ index e i
(index e i_1 $\equiv$ index e i_2)	$\equiv$ ($i_1 \equiv i_2$)
pay ($e_1 \oplus e_2$)	$\equiv$ pay $e_1 \oplus$ pay e_2
pay ($e_1 \otimes e_2$)	$\equiv$ pay $e_1 \otimes e_2$
	$\equiv e_1 \otimes$ pay e_2
fix pay	$\equiv$ empty
biMap f (biMap g e)	$\equiv$ biMap (f $\circ$ g) e
singleton a $\otimes$ e	$\equiv$ biMap (a,) e
e $\otimes$ singleton b	$\equiv$ biMap (, b) e
empty $\oplus$ e	$\equiv$ biMap Right e
e $\oplus$ empty	$\equiv$ biMap Left e

Table 2.2: Operations on enumerations and selected properties

Representing parts The parts of the partition are finite ordered sets. We first specify a datatype Finite a that represents such sets and a minimal set of operations that we require. The datatype is isomorphic to finite lists, with the additional requirement of unique elements. It has two consumer functions: computing the cardinality of the set and indexing to retrieve a value.

$$\begin{aligned}&\mathsf{card}_F :: \mathsf{Finite\ a} \to \mathbb{N}\\&(!!_F)\ :: \mathsf{Finite\ a} \to \mathbb{N} \to \mathsf{a}\end{aligned}$$

As can be expected, $\mathsf{f}\ !!_F\ \mathsf{i}$ is defined only for $\mathsf{i} < \mathsf{card}_F\ \mathsf{f}$. We can convert the finite set into a list:

$$\begin{aligned}&\mathsf{values}_F :: \mathsf{Finite\ a} \to [\mathsf{a}]\\&\mathsf{values}_F\ \mathsf{f} = \mathsf{map}\ (\mathsf{f}!!_F)\ [0\mathinner{..}\mathsf{card}_F\ \mathsf{f} - 1]\end{aligned}$$

The translation satisfies these properties:

$$\begin{aligned}&\mathsf{card}_F\ \mathsf{f} \equiv \mathsf{length}\ (\mathsf{values}_F\ \mathsf{f})\\&\mathsf{f}\ !!_F\ \mathsf{i} \quad \equiv (\mathsf{values}_F\ \mathsf{f})\ !!\ \mathsf{i}\end{aligned}$$

For constructing Finite sets, we have disjoint union, product and bijective function application. The complete interface for building sets is as follows:

$$\begin{aligned}&\mathsf{empty}_F && :: \mathsf{Finite\ a}\\&\mathsf{singleton}_F && :: \mathsf{a} \to \mathsf{Finite\ a}\\&(\oplus_F) && :: \mathsf{Finite\ a} \to \mathsf{Finite\ b} \to \mathsf{Finite\ (Either\ a\ b)}\\&(\otimes_F) && :: \mathsf{Finite\ a} \to \mathsf{Finite\ b} \to \mathsf{Finite}\ (\mathsf{a},\mathsf{b})\\&\mathsf{biMap}_F && :: (\mathsf{a} \to \mathsf{b}) \to \mathsf{Finite\ a} \to \mathsf{Finite\ b}\end{aligned}$$

The operations are specified by the following simple laws:

$$\begin{aligned}&\mathsf{values}_F\ \mathsf{empty}_F && \equiv [\,]\\&\mathsf{values}_F\ (\mathsf{singleton}_F\ \mathsf{a}) && \equiv [\mathsf{a}]\\&\mathsf{values}_F\ (\mathsf{f}_1 \oplus_F \mathsf{f}_2) && \equiv \mathsf{map\ Left}\ (\mathsf{values}_F\ \mathsf{f}_1) +\!\!+\ \mathsf{map\ Right}\ (\mathsf{values}_F\ \mathsf{f}_2)\\&\mathsf{values}_F\ (\mathsf{f}_1 \otimes_F \mathsf{f}_2) && \equiv [(\mathsf{x},\mathsf{y}) \mid \mathsf{x} \leftarrow \mathsf{values}_F\ \mathsf{f}_1, \mathsf{y} \leftarrow \mathsf{values}_F\ \mathsf{f}_2]\\&\mathsf{values}_F\ (\mathsf{biMap}_F\ \mathsf{g}\ \mathsf{f}) && \equiv \mathsf{map\ g}\ (\mathsf{values}_F\ \mathsf{f})\end{aligned}$$

To preserve the uniqueness of elements, the operand of biMap_F must be bijective. Arguably the function only needs to be injective, it does not need to be surjective in the type b. It is surjective into the resulting set of values however, which is the image of the function g on f.

A type of functional enumerations Given the countability requirement, it is natural to define the partition of a set of type a as a function from $\mathbb{N}$ to Finite a. For numbers that do not correspond to a part, the function returns the empty set (empty_F is technically not a part, a partition only has non-empty elements).

```
type Part = ℕ
type E a = Part → Finite a
empty :: E a
empty = const empty_F
singleton :: a → E a
singleton a 0 = singleton_F a
singleton _ _ = empty_F
```

Indexing in an enumeration is a simple linear search:

```
index :: E a → ℕ → a
index e i₀ = go 0 i₀ where
  go p i = if i < card_F (e p)
             then e p !!_F i
             else  go (p + 1) (i − card_F (e p))
```

This representation of enumerations always satisfies countability, but care is needed to ensure bijectivity and part-finiteness when we define the operations in Table 2.2.

The major drawback of this approach is that we can not determine if an enumeration is finite, which means expressions such as index empty 0 fail to terminate. In our implementation (§3) we have a more sensible behaviour (an error message) when the index is out of bounds.

Bijective-function application We can map a bijective function over an enumeration.

```
biMap f e = biMap_F f ∘ e
```

Part-finiteness and bijectivity are preserved by biMap (as long as it is always used only with bijective functions). The inverse of biMap f is biMap f^{-1}.

Disjoint union Disjoint union of enumerations is the pointwise union of the parts.

```
e₁ ⊕ e₂ = λp → e₁ p ⊕_F e₂ p
```

It is again not hard to verify that bijectivity and part-finiteness are preserved. We can also define an "unsafe" version using biMap where the user must ensure that the enumerations are disjoint:

```
union :: E a → E a → E a
union e₁ e₂ = biMap (either id id) (e₁ ⊕ e₂)
```

Guarded recursion and costs Arbitrary recursion may create infinite parts. For example in the following enumeration of natural numbers:

```
data N = Z | S N deriving Show
natEnum :: E N
natEnum = union (singleton Z) (biMap S natEnum)
```

All natural numbers are placed in the same part, which breaks part-finiteness. To avoid this we place a *guard* called pay on (at least) all recursive enumerations, which pays a "cost" each time it is executed. The cost of a value in an enumeration is simply the part-number associated with the part in which it resides. Another way to put this is that pay increases the cost of all values in an enumeration:

```
pay e 0 = empty_F
pay e p = e (p − 1)
```

This definition gives fix pay ≡ empty. The cost of a value can be specified given that we know the enumeration from which it was selected.

```
cost :: E t → t → ℕ
cost (singleton _) _          ≡ 0
cost (a ⊕ b)       (Left x)   ≡ cost a x
cost (a ⊕ b)       (Right y)  ≡ cost b y
cost (a ⊗ b)       (x, y)     ≡ cost a x + cost b y
cost (biMap f e)   x          ≡ cost e (f⁻¹x)
cost (pay e)       x          ≡ 1 + cost e x
```

We modify natEnum by adding an application of pay around the entire body of the function:

```
natEnum = pay (union (singleton Z) (biMap S natEnum))
```

Now because we pay for each recursive call, each natural number is assigned to a separate part:

```
*Main> map values_F (map natEnum [0..3])
[[], [Z], [S Z], [S (S Z)]]
```

Cartesian product Product is slightly more complicated to define. The specification of cost allows a more formal definition of part:

Definition 2 (Part). Given an enumeration e, the part for cost p (denoted as P_e^p) is the finite set of values in e such that

$$(v \in P_e^p) \Leftrightarrow (\mathsf{cost}_e\ v \equiv p)$$

□

The specification of cost says that the cost of a product is the sum of the costs of the operands. Thus we can specify the set of values in each part of a product: $\mathsf{P}^{\mathsf{p}}_{\mathsf{a} \otimes \mathsf{b}} = \bigcup_{k=0}^{\mathsf{p}} \mathsf{P}^{\mathsf{k}}_{\mathsf{a}} \times \mathsf{P}^{\mathsf{p}-\mathsf{k}}_{\mathsf{b}}$. For our functional representation this gives the following definition:

```
e₁ ⊗ e₂ = pairs where
  pairs p = concat_F (conv (⊗_F) e₁ e₂ p)
concat_F :: [Finite a] → Finite a
concat_F = foldl union_F empty_F
conv :: (a → b → c) → (ℕ → a) → (ℕ → b) → (ℕ → [c])
conv (⊘) fx fy p = [fx k ⊘ fy (p − k) | k ← [0 . . p]]
```

For each part we define pairs p as the set of pairs with a combined cost of p, which is the equivalent of $\mathsf{P}^{\mathsf{p}}_{\mathsf{e}_1 \otimes \mathsf{e}_2}$. Because the sets of values "cheaper" than p in both e_1 and e_2 are finite, pairs p is finite for all p. For surjectivity: any pair of values (a, b) have costs $\mathsf{ca} = \mathsf{cost}_{\mathsf{e}_1}\ \mathsf{a}$ and $\mathsf{cb} = \mathsf{cost}_{\mathsf{e}_2}\ \mathsf{b}$. This gives $(\mathsf{a}, \mathsf{b}) \in (\mathsf{e}_1\ \mathsf{ca} \otimes_F \mathsf{e}_2\ \mathsf{cb})$. This product is an element of conv $(\otimes_F)$ e_1 e_2 $(\mathsf{ca} + \mathsf{cb})$ and as such $(\mathsf{a}, \mathsf{b}) \in (\mathsf{e}_1 \otimes \mathsf{e}_2)\ (\mathsf{ca} + \mathsf{cb})$. For injectivity, it is enough to prove that pairs p1 is disjoint from pairs p2 for $\mathsf{p1} \not\equiv \mathsf{p2}$ and that (a, b) appears once in pairs $(\mathsf{ca} + \mathsf{cb})$. Both these properties follow from the bijectivity of e_1 and e_2.

3 Implementation

The implementation in the previous section is thoroughly inefficient; the complexity is exponential in the cost of the input. The cause is the computation of the cardinalities of parts. These are recomputed on each indexing (even multiple times for each indexing). In Feat we tackle this issue with *memoisation*, ensuring that the cardinality of each part is computed at most once for any enumeration.

Finite sets First we implement the Finite type as specified in the previous section. Finite is implemented directly by its consumers: a cardinality and an indexing function.

```
type Index    = Integer
data Finite a = Finite {card_F :: Index
                       , (!!_F)  :: Index → a
                       }
```

Since there is no standard type for infinite precision *natural* numbers in Haskell, we use Integer for the indices. All combinators follow naturally from the correspondence to finite lists (specified in §2). Like lists, Finite is a monoid under append (i.e. union):

```
(⊕_F) :: Finite a → Finite a → Finite a
f1 ⊕_F f2 = Finite car ix where
  car = card_F f1 + card_F f2
  ix i = if i < card_F f1
         then f1 !!_F i
         else  f2 !!_F (i − card_F f1)
empty_F = Finite 0 (λi → error "Empty")
instance Monoid (Finite a) where
  mempty = empty_F
  mappend = (⊕_F)
```

It is also an applicative functor under product, again just like lists:

```
(⊗_F) :: Finite a → Finite b → Finite (a, b)
(⊗_F) f1 f2 = Finite car sel where
  car = card_F f1 ∗ card_F f2
  sel i = let (q, r) = (i ‘divMod‘ card_F f2)
          in  (f1 !!_F q, f2 !!_F r)
singleton_F :: a → Finite a
singleton_F a = Finite 1 one where
  one 0 = a
  one _ = error "Index out of bounds"
instance Functor Finite where
  fmap f fin = fin {(!!_F) = f ◦ (fin!!_F)}
instance Applicative Finite where
  pure     = singleton_F
  f ⟨∗⟩ a  = fmap (uncurry ($)) (f ⊗_F a)
```

For indexing we split the index $i < c_1 * c_2$ into two components by dividing either by c_1 or c_2. For an ordering which is consistent with lists (s.t. $\mathsf{values}_F\ (f \langle * \rangle a) \equiv \mathsf{values}_F\ f \langle * \rangle \mathsf{values}_F\ a$) we divide by the cardinality of the second operand. Bijective map is already covered by the Functor instance, i.e. we require that the argument of fmap is a bijective function.

Enumerate As we hinted earlier, memoisation of cardinalities (i.e. of Finite values) is the key to efficient indexing. The remainder of this section is about this topic and implementing efficient versions of the operations specified in the previous section. A simple solution is to explicitly memoise the function from part numbers to part sets. Depending on where you apply such memoisation this gives different memory/speed tradeoffs (discussed later in this section).

In order to avoid having explicit memoisation we use a different approach: we replace the outer function with a list. This may seem like a regression to the list view of enumerations, but the complexity of indexing is not adversely affected since it already does a linear search on an initial segment of the set of parts. Also the interface in the previous section can be recovered by just applying (!!) to the list. We define a datatype Enumerate a for enumerations containing values of type a.

```
data Enumerate a = Enumerate {parts :: [Finite a]}
```

In the previous section we simplified by supporting only infinite enumerations. Allowing finite enumerations is practically useful and gives an algorithmic speedups for many common applications. This gives the following simple definitions of empty and singleton enumerations:

```
empty :: Enumerate a
empty = Enumerate []

singleton :: a → Enumerate a
singleton a = Enumerate [singleton_F a]
```

Now we define an indexing function with bounds-checking:

```
index :: Enumerate a → Integer → a
index = index' ∘ parts where
  index' []         i = error "index out of bounds"
  index' (f : rest) i
      | i < card_F f   = f !!_F i
      | otherwise      = index' rest (i − card_F f)
```

This type is more useful for a propery-based testing driver (see §6) because it can detect with certainty if it has tested all values of the type.

Disjoint union Our enumeration type is a monoid under disjoint union. We use the infix operator (◇) = mappend (from the library Data.Monoid) for both the Finite and the Enumerate union.

```
instance Monoid (Enumerate a) where
  mempty  = empty
  mappend = union

union :: Enumerate a → Enumerate a → Enumerate a
union a b = Enumerate $ zipPlus (◇) (parts a) (parts b)
  where
    zipPlus :: (a → a → a) → [a] → [a] → [a]
    zipPlus f (x : xs) (y : ys) = f x y : zipPlus f xs ys
    zipPlus _ xs       ys       = xs ++ ys   -- one of them is empty
```

It is up to the user to ensure that the operands are really disjoint. If they are not then the resulting enumeration may contain repeated values. For example pure True ◇ pure True type checks and runs but it is probably not what the programmer intended. If we replace one of the Trues with False we get a perfectly reasonable enumeration of Bool.

Cartesian product and bijective functions First we define a Functor instance for Enumerate in a straightforward fashion:

```
instance Functor Enumerate where
  fmap f e = Enumerate (fmap (fmap f) (parts e))
```

An important caveat is that the function mapped over the enumeration must be *bijective* in the same sense as for biMap, otherwise the resulting enumeration may contain duplicates.

Just as Finite, Enumerate is an applicative functor under product with singleton as the lifting operation.

```
instance Applicative Enumerate where
  pure    = singleton
  f ⟨∗⟩ a = fmap (uncurry ($)) (prod f a)
```

Similar to fmap, the first operand of ⟨∗⟩ must be an enumeration of bijective functions. Typically we get such an enumeration by lifting or partially applying a constructor function, e.g. if e has type Enumerate a then f = pure (,) ⟨∗⟩ e has type Enumerate (b → (a, b)) and f ⟨∗⟩ e has type Enumerate (a, a).

Two things complicate the computation of the product compared to its definition in §2. One is accounting for finite enumerations, the other is defining the convolution function on lists.

A first definition of conv (that computes the set of pairs of combined cost p) might look like this (with mconcat equivalent to foldr ($\oplus_F$) empty_F):

```
badConv :: [Finite a] → [Finite b] → Int → Finite (a, b)
badConv xs ys p = mconcat (zipWith (⊗_F) (take p xs)
                                          (reverse (take p ys)))
```

The problem with this implementation is memory. Specifically it needs to retain the result of all multiplications performed by ($\otimes_F$) which yields quadratic memory use for each product in an enumeration.

Instead we want to perform the multiplications each time the indexing function is executed and just retain pointers to e_1 and e_2. The problem then is the reversal. With partitions as functions it is trivial to iterate an inital segment of the partition in reverse order, but with lists it is rather inefficient and we do not want to reverse a linearly sized list every time we index into a product. To avoid this we define a function that returns all reversals of a given list. We then define a product funtion that takes the parts of the first operand and all reversals of the parts of the second operand.

```
reversals :: [a] → [[a]]
reversals = go [] where
  go _   []     = []
  go rev (x:xs) = let rev′ = x : rev
                  in  rev′ : go rev′ xs
```

```
prod :: Enumerate a → Enumerate b → Enumerate (a, b)
prod e₁ e₂ = Enumerate $ prod′ (parts e₁) (reversals (parts e₂))
prod′ :: [Finite a] → [[Finite b]] → [Finite (a, b)]
```

In any sensible Haskell implementation evaluating an inital segment of reversals xs uses linear memory in the length of the segment, and constructing the lists is done in linear time.

We define a version of conv where the second operand is already reversed, so it is simply a concatenation of a zipWith.

```
conv :: [Finite a] → [Finite b] → Finite (a, b)
conv xs ys = Finite card index
  where card    = sum $ zipWith (∗) (map card_F xs) (map card_F ys)
        index i = mconcat (zipWith (⊗_F) xs ys) !!_F i
```

The worst case complexity of this function is the same as for the conv that reverses the list (linear in the list length). The best case complexity is constant however, since indexing into the result of mconcat is just a linear search. It might be tempting to move the mconcat out of the indexing function and use it directly to define the result of conv. This is semantically correct but the result of the multiplications are never garbage collected. Experiments show an increase in memory usage from a few megabytes to a few hundred megabytes in a realistic application.

For specifying prod′ we can revert to dealing with only infinite enumerations i.e. assume prod′ is only applied to "padded" lists:

```
parts = let rep = repeat empty_F in Enumerate $
  prod′ (parts e₁ ++ rep) (reversals (parts e₂ ++ rep))
```

Then we define prod′ as:

```
prod′ xs rys = map (conv xs) rys
```

Analysing the behaviour of prod we notice that if e_2 is finite then we eventually start applying conv xs on the reversal of parts e_2 with a increasing chunk of empty_F prepended. Analysing conv reveals that each such empty_F corresponds to just dropping an element from the first operand (xs), since the head of the list is multiplied with empty_F. This suggest a strategy of computing prod′ in two stages, the second used only if e_2 is finite:

```
prod′ xs@(_ : xs′) (ys : yss) = goY ys yss where
  goY ry rys = conv xs ry : case rys of
    []          → goX ry  xs′
    (ry′ : rys′) → goY ry′ rys′
  goX ry = map (flip conv ry) ∘ tails
prod′ _ _                   = []
```

If any of the enumerations are empty the result is empty, otherwise we map over the reversals (in goY) with the twist that if the list is depleted we pass the final element (the reversal of all parts of e_2) to a new map (goX) that applies conv to this reversal and every suffix of xs. With a bit of analysis it is clear that this is semantically equivalent to the padded version (except that it produces a finite list if both operands are finite), but it is much more efficient if one or both the operands are finite. For instance the complexity of computing the cardinality at part p of a product is typically linear in p, but if one of the operands is finite it is max p l where l is the length of the part list of the finite operand (which is typically very small). The same complexity argument holds for indexing.

Assigning costs So far we are not assigning any costs to our enumerations, and we need the guarded recursion operator to complete the implementation:

```
pay :: Enumerate a → Enumerate a
pay e = Enumerate (empty_F : parts e)
```

To verify its correctness, consider that parts (pay e) !! 0 $\equiv$ empty_F and parts (pay e) !! $(p + 1)$ $\equiv$ parts e !! p. In other words, applying the list indexing function on the list of parts recovers the definition of pay in the previous section (except in the case of finite enumerations where padding is needed).

Examples Having defined all the building blocks we can start defining enumerations:

```
boolE :: Enumerate Bool
boolE = pay $ pure False ◊ pure True
blistE :: Enumerate [Bool]
blistE = pay $ pure []
             ◊ ((:) ⟨$⟩ boolE ⟨∗⟩ blistE)
```

A simple example shows what we have at this stage:

```
*Main> take 16 (map card_F $ parts blistE)
[0,1,0,2,0,4,0,8,0,16,0,32,0,64,0,128]
```

```
*Main> values_F (parts blistE !! 5)
[[False, False], [False, True], [True, False], [True, True]]
```

We can also very efficiently access values at extremely large indices:

```
*Main> length $ index blistE (10^1000)
3321
*Main> foldl1 xor $ index blistE (10^1000)
True
*Main> foldl1 xor $ index blistE (10^1001)
False
```

Computational complexity Analysing the complexity of indexing, we see that union adds a constant factor to the indexing function of each part, and it also adds one to the generic size of all values (since it can be considered an application of Left or Right). For product we choose between p different branches where p is the cost of the indexed value, and increase the generic size by one. This gives a pessimistic worst case complexity of $p * s$ where s is the generic size. If we do not apply pay directly to the result of another pay, then $p \leqslant s$ which gives s^2. This could be improved to $s \log p$ by using a binary search in the product case, but this also increases the memory consumption (see below).

The memory usage is (as always in a lazy language) difficult to measure exactly. Roughly speaking it is the product of the number of distinguished enumerations and the highest part to which these enumerations are evaluated. This number is equal to the sum of all constructor arities of the enumerated (monomorphic) types. For regular ADTs this is a constant, for non-regular ones it is bounded by a constant multiplied with the highest evaluated part.

Sharing As mentioned, Feat relies on memoisation and subsequently sharing for efficient indexing. To demonstrate this, we move to a more realistic implementation of the list enumerator which is parameterised over the underlying enumeration.

```
listE :: Enumerate a → Enumerate [a]
listE aS = pay $ pure []
                ◊ ((:) ⟨$⟩ aS ⟨*⟩ listE aS)
blistE2 :: Enumerate [Bool]
blistE2 = listE boolE
```

This simple change causes the performance of blistE2 to drop severely compared to blistE. The reason is that every evaluation of listE aS creates a separate enumeration, even though the argument to the function has been used previously. In the original we had blistE in the tail instead, which is a top level declaration. Any clever Haskell compiler evaluates such declarations at most once throughout the execution of a program (although it is technically not required by the Haskell language report). We can remedy the problem by manually sharing the result of the computation with a **let** binding (or equivalently by using a fix point combinator):

```
listE2 :: Enumerate a → Enumerate [a]
listE2 aS = let listE = pay $ pure []
                        ◇ ((:) ⟨$⟩ aS ⟨*⟩ listE)
            in  listE

blistE3 :: Enumerate [Bool]
blistE3 = listE2 boolE
```

This is efficient again but it has one major problem, it requires the user to explicitly mark recursion. This is especially painful for mutually recursive datatypes since all members of a system of such types must be defined in the same scope:

```
data Tree a = Leaf a | Branch (Forest a)
newtype Forest a = Forest [Tree a]

treeE   = fst ∘ treesAndForests
forestE = snd ∘ treesAndForests
treesAndForests :: Enumerate a → (Enumerate (Tree a)
                                 ,Enumerate (Forest a))
treesAndForests eA =
  let eT = pay $ (Leaf ⟨$⟩ eA) ◇ (Branch ⟨$⟩ eF)
      eF = pay $ Forest ⟨$⟩ listE2 eT
  in (eT, eF)
```

Also there is still no sharing between different evaluations of treeS and forestS in other parts of the program. This forces everything into the same scope and crushes modularity. What we really want is a class of enumerable types with a single overloaded enumeration function.

```
class Enumerable a where
  enumerate :: Enumerate a

instance Enumerable Bool where
  enumerate = boolE

instance Enumerable a ⇒ Enumerable (Tree a) where
  enumerate = pay ((Leaf ⟨$⟩ enumerate) ◇ (Branch ⟨$⟩ enumerate))

instance Enumerable a ⇒ Enumerable [a] where
  enumerate = listE2 enumerate

instance Enumerable a ⇒ Enumerable (Forest a) where
  enumerate = pay (Forest ⟨$⟩ enumerate)
```

This solution performs well and it is modular. The only potential problem is that there is no guarantee of enumerate being evaluated at most once for each monomorphic type. We write potential problem because it is difficult to determine if this is a problem in practice. It is possible to provoke GHC into reevaluating instance members, and even if GHC mostly does what we want other compilers might not. In the next section we discuss a solution that guarantees sharing of instance members.

4 Instance sharing

Our implementation relies on memoisation for efficient calculation of cardinalities. This in turn relies on sharing; specifically we want to share the instance methods of a type class. For instance we may have:

```
instance Enumerable a ⇒ Enumerable [a] where
  enumerate = pay $ pure []
                  ◇ ((:) ⟨$⟩ enumerate ⟨*⟩ enumerate)
```

The typical way of implementing Haskell type classes is using dictionaries, and this essentially translates the instance above into a function similar to enumerableList :: Enumerate a → Enumerate [a]. Determining exactly when GHC or other compilers recompute the result of this function requires significant insight into the workings of the compiler and its runtime system. Suffice it to say that when re-evaluation does occur it has a significant negative impact on the performance of Feat. In this section we present a practical solution to this problem.

A monad for type-based sharing The general formulation of this problem is that we have a value x :: C a ⇒ f a, and for each monomorphic type T we want x :: f T to be shared, i.e. to be evaluated at most once. The most direct solution to this problem seems to be a map from types to values i.e. Bool is mapped to x :: f Bool and () to x :: f (). The map can then either be threaded through a computation using a state monad and updated as new types are discovered or updated with unsafe IO operations (with careful consideration of safety). We have chosen the former approach here.

The map must be dynamic, i.e. capable of storing values of different types (but we still want a type safe interface). We also need representations of Haskell types that can be used as keys. Both these features are provided by the Typeable class.

We define a data structure we call a dynamic map as an (abstract) datatype providing type safe insertion and lookup. The type signatures of dynInsert and dynLookup are the significant part of the code, but the full implementation is provided for completeness.

```
import Data.Dynamic (Dynamic, fromDynamic, toDyn)
import Data.Typeable (Typeable, TypeRep, typeOf)
import Data.Map as M
newtype DynMap = DynMap (M.Map TypeRep Dynamic)
  deriving Show
dynEmpty :: DynMap
dynEmpty = DynMap M.empty
dynInsert :: Typeable a ⇒ a → DynMap → DynMap
dynInsert a (DynMap m) =
  DynMap (M.insert (typeOf a) (toDyn a) m)
```

To associate a value with a type we just map its type representation to the dynamic (type casted) value.

```
dynLookup :: Typeable a ⇒ DynMap → Maybe a
dynLookup (DynMap m) = hlp run ⊥ where
  hlp :: Typeable a ⇒ (TypeRep → Maybe a) → a → Maybe a
  hlp f a = f (typeOf a)
  run tr  = M.lookup tr m >>= fromDynamic
```

Lookup is also easily defined. The dynamic library provides a function fromDynamic :: Dynamic → Maybe a. In our case the M.lookup function has already matched the type representation against a type stored in the map, so fromDynamic is guaranteed to succeed (as long as values are only added using the insert function).

Using this map type we define a sharing monad with a function share that binds a value to its type.

```
type Sharing a = State DynMap a
runSharing :: Sharing a → a
runSharing m = evalState m dynEmpty
share :: Typeable a ⇒ Sharing a → Sharing a
share m = do
  mx ← gets dynLookup
  case mx of
    Just e    → return e
    Nothing → mfix $ λe → do
      modify (dynInsert e)
      m
```

Note that we require a monadic fixpoint combinator to ensure that recursive computations are shared. If it had not been used (i.e. if the Nothing case had been m >>= modify ∘ dynInsert) then any recursively defined m would eventually evaluate share m and enter the Nothing case. Using the fix point combinator ensures that a reference to the result of m is added to the map *before* m is computed. This makes any evaluations of share m inside m end up in the Just case which creates a cyclic reference in the value (exactly what we want for a recursive m). For example in x = share (liftM pay x) the fixpoint combinator ensures that we get runSharing x ≡ fix pay instead of ⊥.

Self-optimising enumerations Now we have a monad for sharing and one way to proceed is to replace Enumerate a with Sharing (Enumerate a) and re-implement all the combinators for that type. We don't want to lose the simplicity of our current type though and it seems a very high price to pay for guaranteeing sharing which we are used to getting for free.

Our solution extends the enumeration type with a self-optimising routine, i.e. all enumerations have the same functionality as before but with the addition of an optimiser record field:

```
data Enumerate a = Enumerate
  {parts      :: [Finite a]
  , optimiser :: Sharing (Enumerate a)
  } deriving Typeable
```

The combinator for binding a type to an enumeration is called eShare.

```
eShare :: Typeable a ⇒ Enumerate a → Enumerate a
eShare e = e {optimiser = share (optimiser e)}
```

We can resolve the sharing using optimise.

```
optimise :: Enumerate a → Enumerate a
optimise e = let e′ = runSharing (optimiser e) in
  e′ {optimiser = return e′}
```

If eShare is used correctly, optimise is semantically equivalent to id but possibly with a higher degree of sharing. But using eShare directly is potentially harmful. It is possible to create "optimised" enumerations that differ semantically from the original. For instance λe → eShare t e yields the same enumerator when applied to two different enumerators of the same type. As a general rule the enumeration passed to eShare should be a closed expression to avoid such problems. Luckily users of Feat never have to use eShare, instead we provide a safe interface that uses it internally.

An implication of the semantic changes that eShare may introduce is the possibility to replace the Enumerable instances for any type throughout another enumerator by simply inserting a value in the dynamic map before computing the optimised version. This could give unintuitive results if such enumerations are later combined with other enumerations. In our library we provide a simplified version of this feature where instances can be replaced but the resulting enumeration is optimised, which makes the replacement completely local and guarantees that optimise still preserves the semantics.

The next step is to implement sharing in all the combinators. This is simply a matter of lifting the operation to the optimised enumeration. Here are some examples where ... is the original definitions of parts.

```
fmap f e = e {...
  optimiser = fmap (fmap f) $ optimiser e}
f ⟨*⟩ a = Enumerate {...
  optimiser = liftM2 (⟨*⟩) (optimiser f) (optimiser a)}
pure a = Enumerate {...
  optimiser = return (pure a)}
```

The only noticeable cost of using eShare is the reliance on Typeable. Since almost every instance should use eShare and consequently require type parameters to be Typeable and since Typeable can be derived by GHC, we chose to have it as a superclass and implement a default sharing mechanism with eShare.

```
class Typeable a ⇒ Enumerable a where
  enumerate :: Enumerate a
shared :: Enumerable a ⇒ Enumerate a
shared = eShare enumerate
optimal :: Enumerable a ⇒ Enumerate a
optimal = optimise shared
```

The idiom is that enumerate is used to define instances and shared is used to combine them. Finally optimal is used by libraries to access the contents of the enumeration (see §6).

Non-regular enumerations The sharing monad works very well for enumerations of regular types, where there is a closed system of shared enumerations. For non-regular enumerations (where the number of enumerations is unbounded) the monadic computation may fail to terminate. In these (rare) cases the programmer must ensure termination.

Free pairs and boilerplate instances There are several ways to increase the sharing further, thus reducing memory consumption. Particularly we want to share the cardinality computation of every sequenced application (⟨*⟩). To do this we introduce the FreePair datatype which is just like a pair except constructing one carries no cost i.e. the cost of the pair is equal to the total costs of its components.

```
data FreePair a b = FreePair a b deriving (Show, Typeable)
instance (Enumerable a, Enumerable b) ⇒ Enumerable (FreePair a b)
  where enumerate = FreePair ⟨$⟩ shared ⟨*⟩ shared
```

Since the size of FreePair a b is equal to the sum of the sizes of a and b, we know that for these functions:

```
f :: a → b → c
g :: FreePair a b → c
g (FreePair a b) = f a b
```

We have f ⟨$⟩ shared ⟨∗⟩ shared isomorphic to g ⟨$⟩ shared but in the latter case the product of the enumerations for a and b are always shared with other enumerations that require it (because shared :: FreePair a b is always shared. In other words *deep uncurrying* functions before applying them to shared often improve the performance of the resulting enumeration. For this purpose we define a function which is equivalent to uncurry from the Prelude but that operates on FreePair.

```
funcurry :: (a → b → c) → FreePair a b → c
funcurry f (FreePair a b) = f a b
```

Now in order to make an enumeration for a data constructor we need one more function:

```
unary :: Enumerable a ⇒ (a → b) → Enumerate b
unary f = f ⟨$⟩ shared
```

Together with pure for nullary constructors, unary and funcurry can be used to map any data constructor to an enumeration. For instance pure [ ] and unary (funcurry (:)) are enumerations for the constructors of [a]. In order to build a new instance we still need to combine the enumerations for all constructors and pay a suitable cost. Since pay is distributive over ♢, we can pay once for the whole type:

```
consts :: [Enumerate a] → Enumerate a
consts xs = pay $ foldl (♢) mempty xs
```

This gives the following instance for lists:

```
instance Enumerable a ⇒ Enumerable [a] where
  enumerate = consts [pure [ ], unary (funcurry (:))]
```

5 Invariants

Datatype invariants are a major challenge in property-based testing. An invariant is just a property on a datatype, and one often wants to test that it holds for the result of a function. But we also want to test other properties only with input that is known to satisfy the invariant. In random testing this can sometimes be achieved by filtering: discarding the test cases that do not satisfy the invariant and generating new ones instead, but if the invariant is an arbitrary boolean predicate finding test data that satisfies the invariant can be as difficult as finding a bug. For systematic testing (with SmallCheck or Feat) this method is slightly more feasible since we do not repeat values which guarantees progress, but filtering is still a brute force solution.

With QuickCheck programmers can manually define custom test data generators that guarantee any invariant, but it may require a significant programmer effort and analysing the resulting generator to ensure correctness and statistical coverage can be difficult. Introducing this kind of complexity into testing code is hazardous since complex usually means error prone.

In Feat the room for customised generators is smaller (corresponding to the difference between monads and applicative functors). In theory it is possible to express any invariant by providing a bijection from a Haskell datatype to the set of values that satisfy the invariant (since functional enumerations are closed under bijective function application). In practice the performance of the bijection needs to be considered because it directly affects the performance of indexing.

A simple and very common example of an invariant is the non-empty list. The function uncurry (:) is a bijection into non-empty lists of a from the type (a, [a]). The preferred way of dealing with these invariants in Feat is by defining a **newtype** for each restricted type, and a *smart constructor* which is the previously mentioned bijection and export it instead of the data constructor.

```
newtype NonEmpty a = MkNonEmpty {nonEmpty :: [a]}
  deriving Typeable
mkNonEmpty :: a → [a] → NonEmpty a
mkNonEmpty x xs = MkNonEmpty (x : xs)
instance Enumerable a ⇒ Enumerable (NonEmpty a) where
  enumerate = consts [unary (funcurry mkNonEmpty)]
```

To use this in an instance declaration, we only need the nonEmpty record function. In this example we look at the instance for the datatype Type from the Template Haskell abstract syntax tree which describes the syntax of (extended) Haskell types. Consider the constructor for universal quantification:

```
ForallT :: [TyVarBndr] → Cxt → Type → Type
```

This constructor must not be applied to the empty list. We use nonEmpty to ensure this:

```
instance Enumerable Type where
  enumerate = consts [...
                     , funcurry $ funcurry $ ForallT ∘ nonEmpty]
```

Here ForallT ∘ nonEmpty has type:

```
NonEmpty TyVarBndr → Cxt → Type → Type
```

The only change from the unrestricted enumeration is post-composition with nonEmpty.

Enumerating Sets of natural numbers Another fairly common invariant is sorted lists of unique elements i.e. Sets. It is not obvious that sets can be built from our basic combinators. We can however define a bijection from lists of natural numbers to sets of natural numbers: scanl (((+) ∘ (1+)). For example the list [0,0,0] represents the set [0,1,2], the list [1,1,0] represents [1,3,4] and so on. We can define an enumerator for natural numbers using a bijection from Integer.

```
newtype Nat = Nat {nat :: Integer}
  deriving (Show, Typeable, Eq, Ord)
mkNat :: Integer → Nat
mkNat a = Nat $ abs $ a * 2 − if a > 0 then 1 else 0
instance Enumerable Nat where
  enumerate = unary mkNat
```

Then we define sets of naturals:

```
newtype NatSet = MkNatSet {natSet :: [Integer]}
  deriving Typeable
mkNatSet :: [Nat] → NatSet
mkNatSet = MkNatSet ∘ scanl1 ((+) ∘ (1+)) ∘ map nat
```

Generalising to sets of arbitrary types Sets of naturals are useful but what we really want is a datatype Set a = MkSet {set :: [a]} and a bijection to this type from something which we can already enumerate. Since we just defined an enumeration for sets of naturals, an efficient bijective mapping from natural numbers to a is all we need. Since this is the definition of a functional enumeration, we appear to be in luck.

```
mkSet :: Enumerate a → NatSet → Set a
mkSet e = MkSet ∘ map (index e) ∘ natSet

instance Enumerable a ⇒ Enumerable (Set a) where
  enumerate = unary (mkSet optimal)
```

This implementation works but it is slightly simplified, it doesn't use the cardinalities of a when determining the indices to use. This distorts the cost of our sets away from the actual size of the values.

6 Accessing enumerated values

This section discusses strategies for accessing the values of enumerations, especially for the purpose of property-based testing. The simplest function values is simply all values in the enumeration partitioned by size. We include the cardinalities as well because this is often useful e.g. to report to the user how many values are in a part before initiating testing on values. For this reason we give values type Enumerate a → [(Integer, [a])].

Given that Feat is intended to be used primarily with the Enumerable type class, we have implemented the library functions to use class members, but provide non-class versions of the functions that have the suffix With:

```
type EnumL a = [(Integer, [a])]
values :: Enumerable a ⇒ [(Integer, [a])]
values = valuesWith optimal
valuesWith :: Enumerate a → [(Integer, [a])]
valuesWith = map (λf → (card_F f, values_F f)) ∘ parts
```

Parallel enumeration A generalisation of values is possible since we can "skip" an arbitrary number of steps into the enumeration at any point. The function striped takes a starting index and a step size n and enumerates every n^th value after the initial index in the ordering. As a special case values = striped 0 0 1. One purpose of this function is to enumerate in parallel. If n processes execute uncurry striped k n where k is a process-unique id in the range $[0 \mathinner{..} n-1]$ then all values are eventually evaluated by some process and, even though the processes are not communicating, the work is evenly distributed in terms of number and size of test cases.

```
stripedWith :: Enumerate a → Index → Integer → EnumL a
stripedWith e o0 step = stripedWith' (parts e) o0 where
  stripedWith' (Finite crd ix : ps) o =
      (max 0 d, thisP) : stripedWith' ps o'
    where
      o'     = if space ⩽ 0 then o − crd else step − m − 1
      thisP  = map ix (genericTake d $ iterate (+step) o)
      space  = crd − o
      (d, m) = divMod space step
```

Bounded enumeration Another feature afforded by random-access indexing is the ability to systematically select manageable portions of gigantic parts. Specifically we can devise a function bounded :: Integer → EnumL a such that each list in bounded n contains at most n elements. If there are more than n elements in a part we systematically sample n values that are evenly spaced across the part.

```
samplePart :: Integer → Finite a → (Integer, [a])
samplePart m (Finite crd ix) =
  let step   = crd % m
  in if crd ⩽ m
     then (crd, map ix [0 .. crd − 1])
     else  (m,   map ix [ round (k ∗ step)
                         | k ← map toRational [0 .. m − 1]])
boundedWith :: Enumerate a → Integer → EnumL a
boundedWith e n = map (samplePart n) $ parts e
```

Random sampling A noticeable feature of Feat is that it provides random sampling with uniform distribution over a size-bounded subset of a type. This is not just nice for compatibility with QuickCheck, it is genuinely difficult to write a uniform generator even for simple recursive types with the tools provided by the QuickCheck library.

The function uniform :: Enumerable a ⇒ Part → Gen a generates values of the given size or smaller.

```
uniformWith :: Enumerate a → Int → Gen a
uniformWith = uni ∘ parts where
  uni :: [Finite a] → Int → Gen a
  uni [] _      = error "uniform: empty enumeration"
  uni ps maxp = let (incl, rest) = splitAt maxp ps
                    fin          = mconcat incl
                in case card_F fin of
                     0 → uni rest 1
                     _ → do i ← choose (0, card_F fin − 1)
                            return (fin !!_F i)
```

Since we do not make any local random choices, performance is favourable compared to hand written generators. The typical usage is sized uniform, which generates values bounded by the QuickCheck size parameter. In Table 2.3 we present a typical output of applying the function sample from the QuickCheck library to the uniform generator for [[Bool]]. The function drafts values from the generator using increasing sizes from 0 to 20.

7 Case study: Enumerating the ASTs of Haskell

As a case study, we use the enumeration technique developed in this paper to generate values of Haskell ASTs, specifically the abstract syntax of Template Haskell, taken from the module Language.Haskell.TH.Syntax.

```
*Main> sample (sized uniform :: Gen [[Bool]])
[]
[[]]
[[],[]]
[[True]]
[[False],[],[]]
[[],[False,False,True]]
[[False,True,False,True,True]]
[[False],[],[],[]]
[[True],[True],[],[False,True]]
[[False],[False,True,False,False,True]]
```

Table 2.3: Randomly chosen values from the enumeration of [[Bool]]

```
data Exp    = VarE Name | CaseE Exp [Match] | ...   -- 18 Cons.
data Match  = Match Pat Body [Dec]
data Body   = GuardedB [(Guard, Exp)] | NormalB Exp
data Dec    = FunD Name [Clause] | ...              -- 14 Cons.
data Clause = Clause [Pat] Body [Dec]
data Pat    = LitP Lit | ViewP Exp Pat | ...        -- 14 Cons.
```

Table 2.4: Parts of the Template Haskell AST type. Note that all the types are mutually recursive. The comments indicate how many constructors there are in total of that type

We use the generated ASTs to test the Template Haskell pretty-printer. The background is that in working with *BNFC-meta* (Duregård and Jansson, 2011), which relies heavily on meta programming, we noticed that the TH pretty printer occasionally produced un-parseable output. BNFC-meta also relies on the more experimental package *haskell-src-meta* that forms a bridge between the *haskell-src-exts* parser and Template Haskell. We wanted to test this tool chain on a system-level.

The AST types We limited ourselves to testing expressions, but following dependencies and adding a few **newtype** wrappers this yielded a system of almost 30 datatypes with 80+ constructors. A small part is shown in Table 2.4.

We excluded a few non-standard extensions (e.g. bang patterns) because the specification for these are not as clear (especially the interactions between different Haskell extensions).

Comparison to existing test frameworks We wanted to compare Feat to existing test frameworks. For a set of mutual-recursive datatypes of this size, it is very difficult to write a sensible QuickCheck generator. We therefore excluded QuickCheck from the case study.

On the other hand, generators for SmallCheck and Feat are largely boilerplate code. To avoid having the results skewed by trying to generate the large set of strings for names (and to avoid using GHC-internal names which are not printable), we fix the name space and regard any name as having size 1. But we do generate characters and strings as literals (and found bugs in these).

Test case distribution The result shows some interesting differences between Feat and SmallCheck on the distribution of the generated values. We count the number of values of each part (depth for SmallCheck and size for Feat) of each generator.

Size	1	2	3	4	5	6	...	20
SmallCheck	1	9	951	×	×	×	...	×
Feat	0	1	5	11	20	49	...	65072965

Table 2.5: The number of test cases below a certain size

It is clear that for big datatypes such as ASTs, SmallCheck quickly hits a wall: the number of values below a fixed size grows aggressively, and we are not able to complete the enumeration of size 4 (given several hours of execution time). In the case of Feat, the growth in the number of values in each category is more controlled, due to its more refined definition of size.

We looked more closely into the values generated by SmallCheck by sampling the first 10000 values of the series on depth 4. A count revealed that the maximum size in this sample is 35, with more than 50% of the values having a size more than 20. Thus, contrary to the goal of generating small values, SmallCheck is actually generating pretty large values from early on.

Testing the TH PrettyPrinter The generated AST values are used as test cases to find bugs in Template Haskell's prettyprinter (Language.Haskell.TH.Ppr). We start with a simple property: a pretty-printed expression should be syntactically valid Haskell. We use *haskell-src-exts* as a test oracle:

```
prop_parses e =
  case parse $ pprint (e :: Exp) :: ParseResult Exp of
    ParseOk _        → True
    ParseFailed _ s → False
```

After a quick run, Feat reports numerous bugs, some of which are no doubt false positives. A small example of a confirmed bug is the expression [Con..]. The correct syntax has a space after the constructor name (i.e. [Con ..]). As we can see, this counterexample is rather small (having size 6 and depth 4). However, after hours of testing SmallCheck is not able to find this bug even though many much larger (but not deeper) values are tested. Given a very large search space that is not exhaustible, SmallCheck tends to get stuck in a corner of the space and test large but similar values. The primary cause of SmallCheck's inability to deal with ASTs is that the definition of "small" as "shallowly nested" means that there are very many small values but many types can practically not be reached at all. For instance generating any Exp with a where-clause seems to require at least depth 8, which is far out of reach.

Comparatively, the behaviour of Feat is much better. It advances quickly to cover a wider range of small values, which maximises the chance of finding a bug. The guarantee "correct for all inputs with 15 constructors or less" is much stronger than "correct for all values of at most depth 3 and a few million of depth 4". When there is no bug reported, Feat reports a more meaningful portion of the search space that has been tested.

It is worth mentioning that SmallCheck has the facility of performing "depth-adjustment", that allows manual increment of the depth count of individual constructors to reduce the number of values in each category. For example, instead counting all constructors as 1, one may choose to count a binary constructor as having depth 2 to reflect the fact that it may create a larger value than a unary one (similar to our pay function). In our opinion, this adjustment is a step towards an imprecise approximation of size as used in our approach. Even if we put time into manually adjusting the depth it is unclear what kind of guarantee testing up to depth 8 implies, especially when the definition of depth has been altered away from generic depth.

Testing round trip properties We also tested an extension of this property that does not only test the syntactic correctness but also that the information in the AST is preserved when pretty printing. We tested this by making a round trip function that pretty prints the AST, parses it with *haskell-src-exts* and converts it back to Template Haskell AST with *haskell-src-meta*. This way we could test this tool chain on a system level, finding bugs in *haskell-src-meta* as well as the pretty printer. The minimal example of a pretty printer error found was StringL "\n" which is pretty printed to "", discarding the newline character. This error was not found by SmallCheck partly because it is too deep (at least depth 4 depending on the character generator), and partly because the default character generator of SmallCheck only tests alphabetical characters. Presumably an experienced

SmallCheck tester would use a **newtype** to generate more sensible string literals.

Refuting the small scope hypothesis SmallCheck is based on the *small scope hypothesis* which states that it is sufficient to exhaustively test a small part of the input set to find most bugs. SmallCheck in particular makes the assumption that it is sufficient to test all values bounded by some depth. As we have shown, this assumption does not hold for testing the Template Haskell pretty printer and other properties that quantify over ASTs: although there where several bugs, none where found within feasible range of a depth-bounded search.

Although Feat is not limited to exhaustive search, we found that using Feat to implement size-bounded search is sufficient to find bugs in the Template Haskell example. In other words we did not have to rely on the random access our enumerators provide, the difference between partitioning by depth and size seemed sufficient to find bugs. This raises the question of whether the small scope hypothesis is valid for the scope of size-bounded values, and ultimately if there is any practical value in the ability to select larger values by random or systematic sampling.

To test this we made an additional experiment where we disabled individual constructors from being generated until we where not able to find any errors in the first few million values of our exhaustive search. This is an abbreviated output from our test run:

```
* Testing 0 values at size 0
* Testing 0 values at size 1
* Testing 1 values at size 2
...
* Testing 984968 values at size 16
```

In less than a minute we where able to exhaustively search to size 16 without finding any new bugs. We then tested a systematic sampler that selected at most 10000 values of each size up to size 100 and saw if it found any additional errors.

```
* Testing 0 values at size 0
...
* Testing 4583 values at size 11
* Testing 10000 values at size 12
...
* Testing 10000 values at size 24
Failure!
Conrete Syntax: \'\NUL' -> let var :: [forall var . []] in []
Abstract Syntax: LamE [LitP (CharL '\NUL')] (LetE [SigD var
  (AppT ListT (ForallT [PlainTV var] [] ListT))] (ListE []))
```

This appears to be an error in our reference parser, which expects brackets around the forall type even when it is inside a list constructor. Regardless of where the error lies, it is a counterexample that is not found by exhaustive testing but which is found using systematic sampling.

This method does not guarantee a minimal counterexample and indeed it is possible to remove parts of the example above and still get the same error. The smallest size of a program exhibiting this bug turned out to be 17. We where able to find this bug using exhaustive search by letting it run for a few more minutes. For this reason we are reluctant to consider the outcome as a proper refutation of the small scope hypothesis and we see this part of our experiment as inconclusive.

8 Related Work

SmallCheck, Lazy SmallCheck and QuickCheck Our work is heavily influenced by the property based testing frameworks QuickCheck (Claessen and Hughes, 2000) and SmallCheck (Runciman, Naylor, and Lindblad, 2008). The similarity is greatest with SmallCheck and we improve upon it in two distinct ways:

- (Almost) Random access times to enumerated values. This presents a number of possibilities that are not present in SmallCheck, including random or systematic sampling of large values (too large to exhaustively enumerate) and overhead-free parallelism.
- A definition of size which is closer to the actual size. Especially for testing abstract syntax tree types and other "wide" types this seems to be a very important feature (see §7).

Since our library provides random generation as an alternative or complement to exhaustive enumeration it can be considered a "best of two worlds" link between SmallCheck and QuickCheck. We provide a generator which should ease the reuse of existing QuickCheck properties.

SmallCheck systematically tests by enumerating all values bounded by depth of constructor nestings. In a sense this is also a partitioning by size. The major problem with SmallCheck is that the number of values in each partition grows too quickly, often hitting a wall after a few levels of depth. For AST's this is doubly true; the growth is proportional to the number of constructors in the type, and it is unlikely you can ever test beyond depth 4 or so. This means that most constructors in an AST are never touched.

Lazy SmallCheck can cut the number of tests on each depth level by using the inherent laziness of Haskell. It can detect if a part of the tested value is evaluated by the property and if it is not it refrains from refining this value further. In some cases this can lead to an exponential decrease of the number of required test cases. In the case of testing a pretty printer (as we do in §7) Lazy SmallCheck would offer no advantage since the property fully evaluates its argument every time.

After the submission of this paper, a package named *gencheck* was uploaded to Hackage (Uszkay and Carette, 2012). GenCheck is designed to generalise both QuickCheck and SmallCheck, which is similar to Feat in goal. This initial release has very limited documentation, which prevents a more comprehensive comparison at the moment.

EasyCheck In the functional logic programming language Curry (Hanus et al., 2006), one form of enumeration of values comes for free in the form of a search tree. As a result, testing tools such as EasyCheck (Christiansen and Fischer, 2008) only need to focus on the traversal strategy for test case generation. It is argued in (Christiansen and Fischer, 2008) that this separation of the enumeration scheme and the test case generation algorithm is particularly beneficial in supporting flexible testing strategies.

Feat's functional enumeration, with its ability to exhaustively enumerate finite values, and to randomly sample very large values, lays an excellent groundwork for supporting various test case generation algorithms. One can easily select test cases of different sizes with a desired distribution.

AGATA AGATA (Duregård, 2009) is the previous work of Jonas Duregård. Although it is based entirely on random testing it is a predecessor of Feat in the sense that it attempts to solve the problem of testing syntactic properties of abstract syntax trees. It is our opinion that Feat subsumes AGATA in this and every other aspect.

Generating (Typed) Lambda Terms To test more aspects of a compiler other than the libraries that perform syntax manipulation, it is more desirable to generate terms that are type correct.

In (Yakushev and Jeuring, 2009), well-typed terms are enumerated according to their costs—a concept similar to our notion of size. Similar to SmallCheck, the enumeration in (Yakushev and Jeuring, 2009) adopts the list view, which prohibits the sampling of large values. On the other hand, the special-purpose QuickCheck generator designed in (Pałka et al., 2011), randomly generates well-typed terms. Unsurprisingly, it has no problem with constructing individual large terms, but falls short in systematicness.

It is shown (Wang, 2005) that well-scoped (but not necessarily well-typed) lambda terms can be uniformly generated. The technique used in (Wang, 2005) is very similar to ours, in the sense that the number of possible terms for each syntactic constructs are counted (with memoization) to guide the random generation for a uniform distribution. This work can be seen as a special case of Feat, and Feat can indeed be straightforwardly instrumented to generate well-scoped lambda terms.

Feat is at present not able to express complicated invariants such as type correctness of the enumerated terms. One potential solution is to adopt more advanced type systems as in (Yakushev and Jeuring, 2009), so that the type of the enumeration captures more precisely its intended range.

Combinatorial species In mathematics a *combinatorial species* is an endofunctor on the category of finite sets and bijections. Each object A in this category can be described by its cardinality n and a finite enumeration of its elements: $\mathsf{f} : \mathbb{N}_\mathsf{n} \to \mathsf{A}$. In other words, for each n there is a canoncial object (label set) $\mathbb{N}_\mathsf{n}$. Each arrow $\mathsf{phi} : \mathsf{A} \to \mathsf{B}$ in this category is between objects of the same cardinality n, and can be described by a permutation of the set $\mathbb{N}_\mathsf{n}$. This means that the object action S_0 of an endofunctor S maps a pair (n, f) to a pair $\mathsf{S}_0\ (\mathsf{n}, \mathsf{f})$ whose first component is the cardinality of the resulting set (we call it $\mathsf{card\ n}$). (The arrow action S_1 maps permutations on $\mathbb{N}_\mathsf{n}$ to permutations on $\mathbb{N}_{\mathsf{card\ n}}$.)

In the species Haskell library (decribed by Yorgey (2010)) there is a function $\mathsf{enumerate} : \mathsf{Enumerable\ f} \Rightarrow [\mathsf{a}] \to [\mathsf{f\ a}]$ which takes a (list representation of) an object a to all $(\mathsf{f\ a})$-structures obtained by the S_0 map. The key to comparing this with our paper is to represent the objects as finite enumerations $\mathbb{N}_\mathsf{n} \to \mathsf{a}$ instead of as lists $[\mathsf{a}]$. Then $\mathsf{enumerate}' : \mathsf{Enumerable\ f} \Rightarrow (\mathbb{N}_\mathsf{n} \to \mathsf{a}) \to (\mathbb{N}_{\mathsf{card\ n}} \to \mathsf{f\ a})$. We can further let a be $\mathbb{N}_\mathsf{p}$ and define $\mathsf{sel\ p} = \mathsf{enumerate}'\ \mathsf{id} : \mathbb{N}_{\mathsf{card\ p}} \to \mathsf{f}\ \mathbb{N}_\mathsf{p}$. The function sel is basically an inefficient version of the indexing function in the Feat library. The elements in the image of g for a particular n are (defined to be) those of weight n. The union of all those images form a set (a type). Thus a species is roughly a partition of a set into subsets of elements of the same size.

The theory of species goes further than what we present in this paper, and the species library implements quite a bit of that theory. We cannot (yet) handle non-regular species, but for the regular ones we can implement the enumeration efficiently.

Boltzmann samplers A combinatorial class is basically the same as what we call a "functional enumeration": a set C of combinatorial objects with a size function such that all the parts C_n of the induced partitioning are finite. A *Boltzmann model* is a probability distribution (parameterized over a small real number x) over such a class C, such that a uniform discrete probability distribution is used within each part C_n. A *Boltzmann sampler* is (in our terminology) a random generator of values in the class C following the Boltzmann model distribution. The datatype generic Bolztmann sampler defined in (Duchon et al., 2004) follows the same structure as our generic enumerator. We believe a closer study of that paper could help defining random generators for ASTs in a principled way from our enumerators.

Decomposable combinatorial structures. The research field of enumerative combinatorics has worked on what we call "functional enumeration" already in the early 1990:s and Flajolet and Salvy (1995) provide a short overview and a good entry point. They define a grammar for "decomposable" combinatorial structures including constructions for (disjoint) union, product, sequence, sets and cycles (atoms or symbols are the implicit base case). The theory (and implementation) is based on representing the counting sequences $\{C_i\}$ as generating functions as there is a close correspondance between the grammar constructs and algebraic operations on the generating functions. For decomposable structures they compute generating function *equations* and by embedding this in a computer algebra system (Maple) the equations can be symbolically manipulated and sometimes solved to obtain closed forms for the GFs. What they don't do is consider the pragmatic solution of just tabulating the counts instead (as we do). They also don't consider complex algebraic datatypes, just universal (untyped) representations of them. Complex ASTs can perhaps be expressed (or simulated) but rather awkwardly. They also don't seem to implement the index function into the enumeration (only random generation). Nevertheless, their development is impressive, both as a mathematical theory and as a computer library and we want to explore the connection further in future work.

9 Conclusions and Future work

Since there are now a few different approaches to property-based testing available for Haskell it would be useful with a library of properties to compare the efficiency of the libraries at finding bugs. The library could contain "tailored" properties that are constructed to exploit weaknesses or utilise strengths of known approaches, but it would be interesting to have naturally occurring bugs as well (preferably from production code). It could also be used to evaluate the paradigm of property-based testing as a whole.

Instance (dictionary) sharing Our solution to instance sharing is not perfect. It divides the interface into separate class functions for consuming and combining enumerations and it requires Typeable.

A solution based on stable names (Peyton Jones, Marlow, and Elliot, 1999) would remove the Typeable constraint but it is not obvious that there is any stable name to hold on to (the stable point is actually the dictionary function, but that is off-limits to the programmer). Compiler support is always a possible solution (i.e. by a flag or a pragma), but should only be considered as a last resort.

Enumerating functions For completeness, Feat should support enumerating function values. We argue that in practice this is seldom useful for property-based testing because non trivial higher order functions often have some requirement on their function arguments, for instance the $*$By functions in Data.List need functions that are total orderings, a parallel fold needs an associative function etc. This can not be checked as a precondition, thus the best bet is probably to supply a few manually written total orderings or possibly use a very clever QuickCheck generator.

Regardless of this, it stands to reason that *functional* enumerations should have support for functions. This is largely a question of finding a suitable definition of size for functions, or an efficient bijection from an algebraic type into the function type.

Invariants The primary reason why enumeration can not completely replace the less systematic approach of QuickCheck testing is invariants. QuickCheck can always be used to write a generator that satisfies an invariant, but often with no guarantees on the distribution or coverage of the generator.

The general understanding seems to be that it is not possible to use systematic testing and filtering to test functions that require e.g. type correct programs. Thus QuickCheck gives you something, while automatic enumeration gives you nothing. The reason is that the ratio type correct/syntactically correct programs is so small that finding valid non-trivial test cases is too time consuming.

It would be worthwhile to try and falsify or confirm the general understanding for instance by attempting to repeat the results of (Pałka et al., 2011) using systematic enumeration.

Invariants and costs We have seen that any bijective function can be mapped over an enumeration, preserving the enumeration criterion. This also preserves the cost of values, in the sense that a value x in the enumeration fmap f e costs as much as $f^{-1}x$.

This might not be the intention, particularly this means that a strong size guarantee (i.e. that the cost is equal to the number of constructors) is typically not preserved. As we show in §7 the definition of size can be essential in practice and the correlation between cost and the actual number of constructors in the value should be preserved as far as possible. There may be useful operations for manipulating costs of enumerations.

Conclusions We present an algebra of enumerations, an efficient implementation and show that it can handle large groups of mutually recursive datatypes. We see this as a step on the way to a unified theory of test data enumeration and generation. Feat is available as an open source package from the HackageDB repository:
`http://hackage.haskell.org/package/testing-feat`

References

Axelsson, Emil et al. (2010). "Feldspar: A domain specific language for digital signal processing algorithms". In: *MEMOCODE*, pp. 169–178 (cit. on p. 46).

Baker, Henry G. (1991). "Pragmatic parsing in Common Lisp; or, putting defmacro on steroids". In: *SIGPLAN Lisp Pointers* IV (2), pp. 3–15. ISSN: 1045-3563 (cit. on p. 45).

Bawden, Alan (1999). "Quasiquotation in LISP". In: *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. Ed. by O. Danvy. San Antonio: University of Aarhus, Dept. of Computer Science, pp. 88–99 (cit. on p. 23).

Christiansen, Jan and Sebastian Fischer (2008). "EasyCheck: test data for free". In: *FLOPS'08*. Springer, pp. 322–336 (cit. on p. 78).

Claessen, Koen and John Hughes (2000). "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *ICFP '00*. ACM, pp. 268–279 (cit. on pp. 7, 50, 77).

Czarnecki, Krzysztof et al. (2003). "DSL Implementation in MetaOCaml, Template Haskell, and C++ ". In: *Domain-Specific Program Generation*. LNCS. Berlin, Heidelberg: Springer-Verlag, pp. 51–72. DOI: `10.1007/b98156` (cit. on p. 45).

Devriese, Dominique and Frank Piessens (Jan. 2011). "Explicitly recursive grammar combinators — A better model for shallow parser DSLs". In: *Practical Aspects of Declarative Languages (PADL) 2011*. Vol. 6539. LNCS. Springer (cit. on p. 45).

Duchon, Philippe et al. (2004). "Boltzmann Samplers for the Random Generation of Combinatorial Structures". In: *Combinatorics, Probability and Computing* 13.4–5, pp. 577–625. DOI: `10.1017/S0963548304006315` (cit. on p. 79).

Duregård, Jonas (2009). "AGATA: Random generation of test data". MA thesis. Chalmers University of Technology (cit. on pp. 15, 78).

Duregård, Jonas and Patrik Jansson (2011). "Embedded Parser Generators". In: *Proceedings of the 4th ACM Symposium on Haskell*. Tokyo, Japan: ACM, pp. 107–117. ISBN: 978-1-4503-0860-1. DOI: 10.1145/2034675.2034689 (cit. on pp. v, 73).

Duregård, Jonas, Patrik Jansson, and Meng Wang (2012). "Feat: functional enumeration of algebraic types". In: *Proceedings of the 2012 symposium on Haskell*. Copenhagen, Denmark: ACM, pp. 61–72. ISBN: 978-1-4503-1574-6. DOI: 10.1145/2364506.2364515 (cit. on p. v).

Flajolet, Philippe and Bruno Salvy (1995). "Computer Algebra Libraries for Combinatorial Structures". In: *J. Symb. Comput.* 20.5/6, pp. 653–671 (cit. on p. 80).

Forsberg, Markus and Aarne Ranta (2003). "The BNF Converter: A High-Level Tool for Implementing Well-Behaved Programming Languages". In: *NWPT'02 proc., Proc. Estonian Academy of Sciences* (cit. on pp. 5, 19, 22, 28, 45).

— (2004). "BNF converter". In: *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*. Haskell '04. Snowbird, Utah, USA: ACM, pp. 94–95. ISBN: 1-58113-850-4. DOI: 10.1145/1017472.1017475. URL: http://doi.acm.org/10.1145/1017472.1017475 (cit. on p. 6).

Guzman, Joel de and Hartmut Kaiser (2011). *Boost Spirit*. http://www.boost.org/doc/libs/1_46_1/libs/spirit/ (cit. on p. 45).

Hanus, Michael et al. (2006). *Curry: An Integrated Functional Logic Language*. Version 0.8.2. Available from http://www.informatik.uni-kiel.de/~curry/report.html. (cit. on p. 78).

Hudak, Paul (Dec. 1996). "Building domain-specific embedded languages". In: *ACM Comput. Surv.* 28.4es. ISSN: 0360-0300. DOI: 10.1145/242224.242477. URL: http://doi.acm.org/10.1145/242224.242477 (cit. on p. 7).

Jansson, Patrik and Johan Jeuring (1997). "PolyP — a polytypic programming language extension". In: *Proc. POPL'97: Principles of Programming Languages*. ACM Press, pp. 470–482 (cit. on p. 3).

Joyal, André (1981). "Une théorie combinatoire des séries formelles". In: *Advances in Mathematics* 42.1, pp. 1 –82. ISSN: 0001-8708. DOI: 10.1016/0001-8708(81)90052-9. URL: http://www.sciencedirect.com/science/article/pii/0001870881900529 (cit. on p. 9).

Leijen, Daan and Erik Meijer (2001). *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Tech. rep. UU-CS-2001-35. `http://www.cs.uu.nl/~daan/parsec.html`. Departement of Computer Science, Utrecht University (cit. on p. 39).

Mainland, Geoffrey (2007). "Why it's nice to be quoted: quasiquoting for Haskell". In: *Proc. ACM SIGPLAN workshop on Haskell*. Haskell '07. Freiburg, Germany: ACM, pp. 73–82. ISBN: 978-1-59593-674-5 (cit. on pp. 9, 10, 23, 31, 33–36, 44).

— (2009). *The Haskell package* language-c-quote. `http://hackage.haskell.org/package/language-c-quote` (cit. on p. 42).

Moore, Robert C. (2000). "Removing left recursion from context-free grammars". In: *Proc. 1st North American chapter of the Association for Computational Linguistics conference*. Seattle, Washington: Morgan Kaufmann Publishers Inc., pp. 249–255 (cit. on p. 29).

Naur, P. (Ed.) (1963). "Revised report on the algorithmic language ALGOL 60". In: *Communications of the ACM* 6.1, pp. 1–17 (cit. on p. 4).

Owens, Scott et al. (2004). "Lexer and Parser Generators in Scheme". In: *2004 Scheme Workshop*. URL: `http://repository.readscheme.org/ftp/papers/sw2004/owens.ps.gz` (cit. on p. 45).

Pałka, Michał H. et al. (2011). "Testing an optimising compiler by generating random lambda terms". In: *AST '11*. ACM, pp. 91–97 (cit. on pp. 78, 81).

Peyton Jones, Simon, Simon Marlow, and Conal Elliot (1999). "Stretching the storage manager: weak pointers and stable names in Haskell". In: *IFL'99*. LNCS. Springer (cit. on p. 80).

Ranta, Aarne (2004). "Grammatical Framework". In: *J. Funct. Program.* 14 (2), pp. 145–189. ISSN: 0956-7968 (cit. on p. 28).

Rendel, Tillmann and Klaus Ostermann (2010). "Invertible syntax descriptions: unifying parsing and pretty printing". In: *Proc. third ACM Haskell symposium*. Haskell '10. Baltimore, Maryland, USA: ACM (cit. on p. 45).

Runciman, Colin, Matthew Naylor, and Fredrik Lindblad (2008). "Smallcheck and lazy smallcheck: automatic exhaustive testing for small values". In: *Haskell '08*. ACM, pp. 37–48 (cit. on pp. 8, 49, 77).

Sheard, Tim and Simon Peyton Jones (2002). "Template meta-programming for Haskell". In: *Proceedings of the Haskell workshop*. Pittsburgh, Pennsylvania: ACM Press, pp. 1–16. ISBN: 1-58113-605-6 (cit. on pp. 20, 22).

Svensson, Joel, Koen Claessen, and Mary Sheeran (2010). "GPGPU kernel implementation and refinement using Obsidian". In: *Procedia Computer Science* 1.1. ICCS 2010, pp. 2065 –2074. ISSN: 1877-0509. DOI: `10.1016/j.procs.2010.04.231` (cit. on p. 46).

Swierstra, S. Doaitse (2009). "Combinator Parsing: A Short Tutorial". In: Berlin, Heidelberg: Springer-Verlag, pp. 252–300. ISBN: 978-3-642-03152-6. DOI: `10.1007/978-3-642-03153-3_6` (cit. on p. 45).

The Boost Initiative (2009). *The Boost initiative for free peer-reviewed portable C++ source libraries*. `http://www.boost.org` (cit. on p. 45).

Uszkay, Gordon J. and Jacques Carette (2012). *The Haskell package* gencheck. `http://hackage.haskell.org/package/gencheck` (cit. on p. 78).

Wang, Jue (2005). *Generating random lambda calculus terms*. Tech. rep. Boston University (cit. on p. 78).

Yakushev, Alexey Rodriguez and Johan Jeuring (2009). "Enumerating Well-Typed Terms Generically". In: *AAIP 2009*. Vol. 5812. LNCS. Springer, pp. 93–116 (cit. on pp. 78, 79).

Yorgey, Brent A. (2010). "Species and Functors and Types, Oh My!" In: *Proceedings of the third ACM symposium on Haskell*. ACM, pp. 147–158. DOI: `10.1145/1863523.1863542` (cit. on pp. 9, 79).