UNIVERSITY OF GOTHENBURG

API-Driven Generation of Well-Typed Terms

Master of Science Thesis in Computer Science

David Spångberg

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, June 2014

API-Driven Generation of Well-Typed Terms
David Spångberg

Examiner: Emil Axelsson

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

Department of Computer Science and Engineering
Göteborg, Sweden June 2014

Abstract

In this thesis, a reusable library for defining generators for well-typed expressions in standard Haskell is presented. The expressions are randomly constructed from a set of functions and values specified by a user. Both the types of these functions and the type of the generated expression can be polymorphic and/or higher order, i.e. containing functions with polymorphic types as arguments. The main motivation for this library is for generating test data when testing an Embedded Domain Specific Language (EDSL) where constructing a generator for the language by hand might be both tedious and error-prone.

The library was successfully used to define a generator producing terms similar to those computed by such a hand-made generator. The code size and complexity of the final generator was significantly reduced when compared to the hand-made one.

Contents

Chapter 1

Introduction

When developing a compiler for an Embedded Domain Specific Language (EDSL) one often want to test certain aspects of the compiler, for instance, testing that the result of evaluating an expression produces the same result both before and after certain optimizations. One way to achieve this is to design unit tests that cover all developed optimizations. This approach has several limitations, for instance, when new optimizations are added, new unit tests have to be developed. Similarly, when optimizations are modified, the corresponding unit tests have to be updated.

Instead, when testing a compiler, it is possible to construct a generator for random Abstract Syntax Trees (ASTs) of the given language and use this generator to test the implementation. However, these ASTs might have several invariants and constructing a generator producing only valid ASTs might be non-trivial. One solution to this problem is to only use smart constructors given by an Application Programming Interface (API) to produce values of the requested type. However, combining these smart constructors in a type-correct way might be equally non-trivial.

Aspects of the compiler, other than optimizations, can also be tested in similar ways. One example is testing that interpreting a program versus compiling and executing it produces the same result. This test and several others have similar preconditions as the optimization tests; it requires that a program is generated or constructed before the test in question can be run.

1.1 Random generation

Instead of creating a generator for an AST manually, one might want to randomly generate values of the AST by using a list of functions from an API instead. This is reasonable since all meaningful terms in the AST should be describable somehow from the API provided by the EDSL. What is missing is a procedure, taking a list of functions together with a goal type, producing random expressions of the goal type by combining the given functions in a type-correct way. To create a generator for an AST, one simply specifies the type of the AST as the goal type and the list of functions as the functions from an API, working on that AST. If functions are added or removed from the API, these functions only need to be added or removed from the list of available functions supplied to the procedure. Even better, if the AST or functions in the API are modified, no modification is required for the term generation procedure.

1.2 Introducing QuickGen

In this report, the library QuickGen, written in the programming language Haskell [Mar10], is presented. The library can be used to generate random functions and values, with higher order types, containing polymorphic type variables. A *Language* (i.e. an API) is simply defined using Template Haskell [Lyn14], by specifying the functions and values available during term generation, for example:

```haskell
lang :: Language
lang = $(defineLanguage [| ( map   :: (a -> b) -> [a] -> [b]
                           , const :: a -> b -> a
                           , foldr :: (a -> b -> b) -> b -> [a] -> b
                           , nil   :: [a]
                           , cons  :: a -> [a] -> [a]
                           , n     :: Int
                           )
                         |])
```

A specific library function is responsible for generating expressions given a language, a *goal type* and a random seed:

```haskell
generate :: Language -> Type -> Seed -> Maybe Exp
```

A generator for terms of type `a -> [a]` using `lang` above can then be defined in the following way:

```haskell
f :: Seed -> Maybe Exp
f seed = generate lang $(getType [t| a -> [a] |]) seed
```

Different seeds supplied to this function then produce random well typed expressions, using the available functions and values from `lang`. The generated expressions can later be turned into real Haskell terms by using, for instance, the GHC API [ghc14].

1.3 Related work

Generating higher-order terms and functions have been the subject of research recently. For instance, in [Pal11] a generator very similar to the one presented in this thesis is introduced. This generator was successfully used to discover bugs in the Glasgow Haskell Compiler (GHC) [Mar13]. The following sections provide a brief overview over related research that has been used as inspiration for the duration of this project.

1.3.1 Inductive programming

In the field of inductive programming [Hof13], the interest lies in generating a program from an incomplete specification. For instance, in [Kat10], Katayama presents a generator that searches for all expressions, in a breadth-first manner, matching a goal type and at the same time satisfies a predicate. Since this generator produces and examines all well-typed expressions, instead of randomly searching for one, it was deemed not suitable for producing random expressions with deep nesting[1].

1.3.2 Efficient enumeration

In [DJW12] a library for defining enumerations of arbitrary algebraic data types is developed. These enumerations can also be indexed efficiently. As an example, the term at position 10^{100} for the complex `Exp` data type defined in Template Haskell is generated in less than a second on a normal desktop computer. However, Feat is currently not able to enumerate terms with complicated invariants, such as well-typed terms, [DJW12, p. 71].

1.3.3 Theorem proving

Augustsson presents a library for generating expressions in Haskell given a type [Aug05]. These terms are constructed with the help of a theorem prover for intuitionistic propositional logic by encoding types as logical statements. However, this

[1]However, the interface in the `Testing.QuickGen.TH` Haskell module was greatly inspired by the one used in [Kat10].

theorem prover, and theorem provers in general, will search for any, often minimal, proof of a statement and is therefore not suitable for generating random terms.

1.4 Problem and goal

The problem is that there currently is no automatic way to construct specialized generators for an EDSL, i.e. by specifying an API, in the programming language Haskell. The generators, when constructed, should be capable of generating any well-typed standard Haskell expression, of a given goal type. Any, in this context, means functions and values, possibly higher order and/or containing polymorphic type variables. In the best case scenario, the generators should be able to construct terms from functions, in the API, using type classes.

The goal is to implement a library that automatically constructs such generators. The functions and values produced from the generators should possibly have higher order and/or polymorphic types. The goal of the project is considered reached when the library has been used to successfully test a real world EDSL, such as FELDSPAR [Axe+10].

1.5 Structure

This thesis starts in chapter 2 with a formal definition of the algorithms used when generating expressions. Some limitations are also discussed here. After this, in chapter 3, the current implementation in the programming language Haskell is discussed. This chapter can be used as an extended documentation to the implementation. Example usage of the library is presented in chapter 4. For instance, the implementation of a generator with similar behaviour to that of the hand-made generator used when testing the EDSL Copilot [Pik+12] is defined and discussed. The last two chapters, chapters 5 and 6, discuss future work and conclusions of the project, respectively.

Chapter 2

Algorithm

This section is dedicated to the formal definition of the algorithm used to generate expressions from a user API. The first part contains an example run of the complete algorithm, followed by a discussion regarding some tricky parts. The last two subsections contain more formal definitions, with pseudo code, of the functions responsible for *type matching* and *term generation*, respectively.

2.1 A complete example

Suppose a user wants to generate an expression of type $[Int]$ using values and functions from the following API:

Constructor name	Type
map	$(a \to b) \to [a] \to [b]$
$sing$	$a \to [a]$
nil	$[a]$
n	Int
d	$Double$

Table 2.1: Simple API

The first step is to choose a random function or value where the return type matches our current goal type. In table 2.1 above, the term *constructor* is used to refer to one of the functions or values that can be used by the algorithm to generate expressions, this is also the term used in the rest of this thesis. Just by looking at the available constructors, one finds that all but the last two values have return types that match our current goal: $[Int]$. The constructors that do match, however, need to be slightly specialized to correctly match our goal type.

Assume that map was the first randomly selected matching constructor. In this case, for map to match our goal type of $[Int]$, the universally quantified type variable b has to be instantiated to the type Int. The other type variable, a has nothing to do with the current goal and can therefore be instantiated to anything. We might at this point generate a random type for a based on the constructors in scope – this is done by Palka in [Pal11]. Here, another tactic is employed: we say that the type variable is *undecided*. We introduce the special notation $?a$ to mean exactly this; that the type variable a is undecided. What this means is that the type of a has not been specialized yet, but might be in a later stage of the generation algorithm. In the end, the final specialized type for map is $(?a \to Int) \to [?a] \to [Int]$. At this point, it might help to visualize the current expression as:

$$map\ e_1\ e_2$$

where e_1 and e_2 are two placeholder expressions with types $(?a \to Int)$ and $[?a]$, respectively. To succeed, we need to generate these new subexpressions (subgoals) in some order. In this example, we choose to do it from left to right, starting with e_1:

- *Generating e_1 with type $?a \to Int$:*

  At this point, the algorithm differs slightly from what was done when starting to generate $[Int]$ above; the difference is that this time, we are trying to generate a function. In the end, we want a lambda abstraction of the form:

  $$\lambda x \to body$$

  where the argument x can be used inside $body$. To proceed, all arguments are added as constructors available when generating the body of the lambda abstraction. In this case, the only constructor added is $x :: ?a$, resulting in the following set of constructors:

Constructor name	Type
map	$(a \to b) \to [a] \to [b]$
$sing$	$a \to [a]$
nil	$[a]$
n	Int
d	$Double$
x	$?a$

Table 2.2: Extended API

  From here on, the algorithm is exactly the same as for the case when generating an expression of type $[Int]$ as seen above: a random matching constructor $\mathcal{C}$

is chosen and the arguments of $\mathcal{C}$, $y_1 \dots y_n$, are recursively generated. If all arguments are successfully generated, then *body* will be equal to $\mathcal{C}\ y_1\ \dots\ y_n$ and the complete expression e_1 will be $(\lambda x \rightarrow \mathcal{C}\ y_1\ \dots\ y_n)$.

Here, only the two constructors n and x match the current goal, which is Int, and are therefore the only candidates for $\mathcal{C}$. If we choose $\mathcal{C} = n$, we return with $e_1 = (\lambda x \rightarrow n)$ and continue generating $e_2 ::\ [?a]$. If we choose $\mathcal{C} = x$, we return with $e_1 = (\lambda x \rightarrow x)$ [1]. At this point, instead of continuing generating $e_2 ::\ [?a]$ as before, we now need to generate $e_2 ::\ [Int]$. The reason is that in the original types of e_1 and e_2, $(?a \rightarrow Int)$ and $[?a]$, respectively, the undecided type variable $?a$ refers to the same type in both expressions. If we start generating e_1 and decide that $?a$ has to be an Int, then this choice has to be remembered when generating e_2. To illustrate the difference, both of these cases will be considered below:

- *Case $e_1 = (\lambda x \rightarrow n)$: Generating e_2 with type* $[?a]$:

  This time, no new constructors are added to the API since we are not generating a function. The API at this point is therefore the one found in table 2.1. Similar to before, when choosing map as a constructor, the first three constructors are the only ones matching the current goal. Suppose the second one, *sing*, is chosen. In this case, the specialized type for $sing$ would be $(?a \rightarrow [?a])$ and a new subgoal for an expression of type $?a$ is created. Here, $?a$ can be matched with any constructor in the API, but let us assume $d :: Double$ is chosen, resulting in $e_2 = sing\ d$. Since this is the last subgoal, the term generation algorithm terminates with the complete expression $map\ (\lambda x \rightarrow n)\ (sing\ d)$.

- *Case $e_1 = (\lambda x \rightarrow x)$: Generating e_2 with type* $[Int]$:

  Similar to the case for $[?a]$, no new constructors are added and the API is therefore the one found in table 2.1. Suppose that the randomly selected constructor chosen is the same as in the last step, *sing*. Here is where the difference in choice in e_1 is visible when generating e_2. In the case above, all constructors matched the goal of $?a$. However, in this case, $?a$ has already been instantiated to a type when generating e_1, namely Int. This forces us to choose n as our constructor, since no other constructors match the current goal. Again, this terminates the algorithm leaving us with the final expression $map\ (\lambda x \rightarrow x)\ (sing\ n)$.

2.1.1 Undecided variables

When generating e_1 in section 2.1 above, two different cases were considered. In the first case, the undecided variable $?a$ was never instantiated. The second case showed

[1]There is actually an infinite number of ways to instantiate $?a$ to match the current goal of Int. Only considering one of them is a simplification to the original problem. See section 2.2.2.

that a guess made for an undecided variable in one part needs to be remembered in the rest of the generation process. How this information about guesses for undecided variables should be handled is non trivial. Let us look at an example:

Example 2.1: Imagine that we are generating a term of type t using the following constructor:

$$c :: t_1 \to \ldots \to t_n \to t$$

Furthermore, assume that the undecided variable $?a$ is part of the type of c, i.e., $?a$ is present in at least one, and potentially all, of the types $t_1, \ldots, t_n$ and t. If we start generating a subgoal, for instance t_1, we might select another constructor containing additional subgoals which in turn might introduce even more subgoals. At any point in these subtrees we might decide that $?a$ should have type Int. It is critical that this information is shared between all subgoals. One could try to update the API by exchanging every usage of $?a$ with Int. However, this does not solve the problem that $?a$ might be present in one of the subgoals at the same level or higher up. If the subgoals and intermediate constructors were saved on a stack one might traverse this stack updating the types for subgoals containing $?a$. However, this stack, and the API, might grow to be very large making it inefficient to traverse the stack every time an undecided variable is updated.

An alternative tactic, that is employed in this project, is to have a global set of guesses for undecided variables, henceforth referred to as $\mathbb{U}$, that is consulted before trying to generate a subgoal containing undecided variables. Suppose that we have a goal type t containing the undecided variable $?a$. Further assume that $\mathbb{U}$ contain a guess, $?a \mapsto Int$, indicating that $?a$ should be substituted by Int. To continue, we substitute each occurence of $?a$ with Int inside the type t.

The set of guesses, $\mathbb{U}$, is also consulted before trying to match a goal with any constructor c from the API since the type of c might also contain undecided variables. Finally, when a guess for an undecided variable is performed, this guess is recorded in $\mathbb{U}$. For instance, the guess $?a \mapsto Int$ was performed when x was used as a constructor when generating e_1 in section 2.1.

2.1.2 Generating polymorphic expressions

One more type of expression needs to be discussed before continuing, namely expressions with polymorphic types. To see how this is done in this project, let us first look at a problematic example run:

- Imagine that we want to generate an expression with goal type $a \to a \to [a]$. We proceed in the same way as was done when generating the first argument to map in section 2.1 above, i.e. generating a lambda abstraction by generating names for the variables, adding the arguments with their respective types to the API and finally generating the lambda body with the updated API. The goal type when generating the body would be $[a]$. Furthermore, let us assume that the API is the following:

Constructor name	Type
map	$(a \to b) \to [a] \to [b]$
$succ_{Int}$	$Int \to Int$
$sing$	$a \to [a]$
x_1	a
x_2	a

Table 2.3: The constructors marked in red were added by the algorithm when generating the lambda abstraction.

Further imagine that *map* was randomly chosen as our constructor, introducing two subgoals $e_1 :: (?b \to a)$ and $e_2 :: [?b]$. Suppose we start with the second subgoal, with goal type $[?b]$, and randomly select *sing* followed by $succ_{Int}$ as our next constructors. Our expression at this point would be:

$$\lambda\ x_1\ x_2 \to map\ e_1\ (sing\ (succ_{Int}\ e_3))$$

Note that selecting $succ_{Int}$ above also introduced the guess $?b \mapsto Int$ in $\mathbb{U}$. The variables e_1 and e_3 refers to the currently unsolved subgoals. At this point, the current goal type, the type of e_3, is Int. By looking at the types in the API alone, it would be perfectly reasonable to choose x_1 as a constructor for this goal, since x_1 is universally quantified and can be matched with anything. However, if we substitute e_1 by x_1 and try to type check the resulting expression in, for instance, `ghci` we get the following:

```
λ> :t (λ x_1 x_2 -> map undefined (sing (succ_int x_1))) :: a -> a -> [a]

<interactive>:1:44:
    Couldn’t match expected type ‘a’ with actual type ‘Int’
    ...
```

The problem above is that the type of x_1 and x_2 should not be a universally quantified type variable ($\forall\ a.\ a$), when introduced to the API above. A solution to this problem is to substitute every universally quantified type variable with a dummy unique type constructor. This is the tactic employed by Katayama in [Kat10]. Thus, in the example above, before starting the generation process, each occurrence of the type variable a in $a \rightarrow a \rightarrow [a]$ should be exchanged by a unique type constructor, for instance A_1, resulting in the type $A_1 \rightarrow A_1 \rightarrow [A_1]$. Later during matching, the type A_1 will be matched using the same procedure used to match other type constructors. The exact procedure for matching type constructors can be found in listing 1.

2.1.3 Termination

There is one simplification to the simple algorithm presented above that need to be mentioned. The algorithm, if implemented directly, is not guaranteed to terminate. To see why, consider the following example:

Constructor name	Type
id	$a \rightarrow a$
n	Int

Say that we want to generate a term of type Int. The generation algorithm might choose id as the first constructor. After type matching and specialization we have exactly the same API and subgoal as in the original problem, we need to generate an Int. At this point there is nothing that stops the algorithm from choosing id indefinitely making this a non terminating process. In this particular example, the probability of termination is quite high but this might not be the case if constructors, requiring several subgoals to be generated, are introduced to the API.

The solution used to solve this problem in this project is to limit the number of uses for each constructor of functional type, i.e. constructors requiring subgoals. Constructors with zero subgoals, such as Int, will have an infinite number of uses. The notation $\textsc{Uses}(t)$ will henceforth be used to denote the number of uses available for a constructor with type t.

$$\textsc{Uses}(t) = \begin{cases} 10 & \text{if } t \text{ is a function type} \\ \infty & \text{otherwise} \end{cases}$$

The number 10 here was chosen after some experimentation and seems to enable complicated expressions in a reasonable big API while still limiting the search space enough to make the algorithm terminate if no solution can be found.

The solution with Uses above is not the only way to solve the issue of termination. In section 5.6, some other termination strategies are discussed.

2.1.4 Subgoal ordering

After choosing *map* as the first constructor in the original algorithm in section 2.1, we choose to generate the subgoals to *map* from left to right. With the addition of limited uses of constructors, as mentioned in section 2.1.3 above, the order in which subgoals are generated influence the final shape of generated expressions[2]. Again, let us illustrate this by looking at an example where we want to generate an expression with type *Int* using the following API:

Constructor name	Uses	Type
const	2	$a \to b \to a$
n	∞	*Int*

Table 2.4: API containing constructors with a limited number of uses

As before, the API contains the constructors available for use when generating expressions. The difference this time is that a limited number of uses, here two, is also imposed on the first constructor. The second constructor, having no subgoals, is given an infinite number of uses. Suppose *const*, with the specialized type $Int \to ?b \to Int$, is our first randomly chosen constructor. After choosing *const*, we must also update the API decreasing the number of uses for *const* by one. From here, we can choose to generate either of the subgoals, *Int* respectively $?b$. If we choose to generate from left to right starting with *Int*, and *const* is our next random choice of constructor, then we have effectively used up all usages of *const* available in this run of the algorithm. This forces us to choose *n* as a constructor in all remaining subgoals, including the goal for $?b$. If we go right to left instead, we may experience the same problem, i.e. we may limit the available constructors, thereby forcing the shape of the finished expression.

In general, if there are several constructors in the API containing at least one subgoal, the generated expressions are biased towards the direction of the first subgoals, i.e. if we start from the left, most usages of constructors will be present in the leftmost subgoals.

Due to implementation details, the algorithm used in this project generates goals from right to left making the expressions biased towards the right. In one of the sections in future work, section 5.5, some alternative tactics regarding subgoal ordering are discussed.

[2]This is also true for undecided variables as was shown in section 2.1.1.

2.2 Matching expressions

Type matching is the process of finding a minimal substitution for a type t_1 that makes it equal to a goal type t_2 [JP08, pp. 7–8]. For instance, imagine we have the following variables: $x :: a \to Int \to b$ and $y :: Int \to Int \to Bool$. If we try to match the type of x with y then the substitution $\{a \mapsto Int, b \mapsto Bool\}$ will be returned. If no match is found a failure is raised instead.

The Match algorithm presented below is similar to regular type matching in the respect that it finds a substitution for the universally quantified type variables. However, there are some differences. Let us look at one example to see how it differs from normal type matching:

$$\textsc{Match}(Int,\ a \to b) \Longrightarrow (?a \to Int)$$

The first difference we see is that Match actually returns a new type instead of a substitution. Match did find a substitution but then immediately applied it to the second argument to produce a specialized type. Further, a normal type matching algorithm would not find a substitution at all, it would fail on this particular input, since a value type normally cannot be matched against a function. Match however returns the type $(?a \to Int)$.

From the original type we can see that b has been turned into Int which might not be all that surprising. The type variable a, however, has lost its quantifier and been turned into an undecided type variable, as explained in section 2.1. Further, the Match function only looks at the return type of the second argument during matching. Therefore, in this example, the only type variable considered during matching was b. All of the remaining universally quantified type variables, which in this case is a, will be turned into undecided type variables, here $?a$.

The intuition is that a function f of type $(a \to Int)$ can be used to construct a value of type Int if we give it another value of type a. Since a is universally quantified, a value of any type can be given to f for it to produce an Int.

With this small introduction it is time to look at the algorithm for the match function.

- Let $\mathbb{U}$ be the set of guesses for undecided variables, as discussed in section 2.1.1. Then the MATCH function takes two types, t_1 and t_2, and finds out if the type t_2 can be specialized in such a way that a value of this specialized type can be used in the construction of a value of type t_1. If MATCH is successful, the specialized version of t_2 is returned. In addition to returning the specialized type, the set of guesses might be updated during matching. If no match is found, the algorithm will fail.

```
1: **function** MATCH($t_1, t_2$)
2:     **if** $t_2$ is a function type $t_2 = (x_1 \to \ldots \to x_n)$ **then**
3:         $s \leftarrow$ MATCHAUX($t_1, x_n$)
4:     **else**
5:         $s \leftarrow$ MATCHAUX($t_1, t_2$)
6:
7:     Update $t_2$ by applying the substitution $s$ to the type
8:     Update $t_2$ by converting all *Forall* quantified types to *Undecided* ones
9:
10:    **return** $t_2$
11: **end function**
```

The first step is trying to find a substitution that either makes t_2, or the return type of t_2, match t_1. If such a substitution is found then it is applied to t_2. The last step before returning t_2 is to make all *Forall* quantified type variables into *Undecided* ones. The algorithm that finds a substitution is found in listing 1 [3]. Let us look at an example of how the complete algorithm works:

- Let $t_1 = [Int]$ and $t_2 = (a \to b) \to [a] \to [b]$. MATCH will proceed by trying to find a substitution for $[Int]$ and the return type of t_2, $[b]$, since t_2 is a function type.
  - In MATCHAUX the second case will match with $\mathcal{C} = [\,]$ and $y_1 = b$. Since t_1 is also of this form, $\mathcal{C} = [\,]$ and $x_1 = Int$, then MATCHAUX($Int,\ b$) will be called recursively.
  - Now since the second argument is universally quantified, the singleton substitution $\{b \ \mapsto \ Int\}$ is returned. This is also the value returned to MATCH.
- The substitution returned from MATCHAUX is applied to $t_2 = \ (a \to b) \to [a] \to [b]$ resulting in $(a \to Int) \to [a] \to [Int]$.
- In the last step the remaining universally quantified variable is turned into an undecided one and $(?a \to Int) \to [?a] \to [Int]$ is returned.

[3]This function is closer to traditional type matching compared to MATCH.

```
function MatchAux($t_1, t_2$)
    case $t_2$ of
        $\forall\ b.\ b$ then
            return $\{b \mapsto t_1\}$
        $\mathcal{C}(y_1, \ldots, y_n)$ then
            if $t_1$ is not the same type constructor as $t_2$, i.e. $t_1 \neq \mathcal{C}(x_1, \ldots, x_n)$ [4] then
                raise No_Match
            else
                return $\bigcup_i$ MatchAux($x_i, y_i$)
        $?b$ then
            if $t_1 = t_2 = ?b$ then
                return $\emptyset$
            else if $?b \in$ Vars($t_1$) then
                raise No_Match
            else if $\exists\ t$, s.t. $(?b \mapsto t) \in \mathbb{U}$ then
                return MatchAux($t_1, t$)
            else
                Add the mapping $(?b \mapsto t_1)$ to $\mathbb{U}$
                return $\emptyset$
        else
            case $t_1$ of
                $?a$ then
                    ▷ Similar to the case for $?b$ except in the last else
                    ...
                    else
                        Convert all $\forall$ type variables in $t_2$ to undecided
                        Add the mapping $(?a \mapsto t_2)$ to $\mathbb{U}$
                        return $\emptyset$
                else
                    raise No_Match
end function
```

Listing 1: The complete matching algorithm.

2.2.1 Unique types

One important property which has been left out of the discussion so far is that all undecided type variables introduced in Match are assumed to be unique. Let us look at an example to explain this:

[4]For some types $x_1 \ldots x_n$.

Example 2.2: Imagine we are generating a value with type $[[Int]]$ using the API found in table 2.1. Further suppose that the final, well-typed, expression we want to generate is the following:

$$map\ (\lambda xs \to map\ (\lambda x \to x)\ xs)\ [[n]]$$

If we follow the general algorithm introduced in section 2.1 in minute detail we would use the same undecided type variable $?a$ for both uses of map, which is not correct. To see this, imagine the generation algorithm proceeds as follows:

- We choose map as our first constructor, introducing $?a \to [Int]$ and $[[?a]]$ as subgoals.
- We continue with the second subgoal, i.e. we generate the expression $[[n]]$, introducing the guess $(?a \mapsto [Int])$ in $\mathbb{U}$.
- The first subgoal, $\lambda xs \to e_3$, to the outer map is generated. Since $?a$ was resolved to $[Int]$, xs will also have this type.
- The remaining subgoal is that for the placeholder expression e_3 with type $[Int]$. Again, we choose to use map as our constructor, introducing the subgoals $?a \to Int$ and $[?a]$. This time, since we already have a guess for the undecided variable $?a$ in $\mathbb{U}$, the subgoal for the inner map will be updated to $([Int] \to Int)$ and $[[Int]]$, respectively. However, these types do not permit us to choose, for instance, xs as a second argument for the inner map which is incorrect.

A simple way to solve this problem is to exchange every universally quantified type variable in a type t with a natural number before supplying t as the second argument to MATCH. Start with $n := 0$ which represents the next unique natural number to be used in a type. The general procedure is defined as follows:

- For some type t find the set of universally quantified type variables encountered in t:
$$\textsc{Vars}(t) = \{a_1, \ldots, a_m\}$$
- Create the substitution $s = \{a_1 \mapsto n, \ldots, a_m \mapsto n + m - 1\}$
- Let $n := n + m$
- Update t by applying the substitution s to t

At this point every type variable in t is unique for the entire run of the algorithm since it is not legal to have natural numbers as types in standard Haskell. This makes it completely safe to introduce mappings for undecided variables in $\mathbb{U}$ in MATCHAUX.

2.2.2 Matching functions

One simplification to the problem of type matching was made in this project. A simple example illustrates how this simplification affects the type matching algorithm. Suppose we want to generate an expression with goal type Int using the API found in Table 2.5 below:

Constructor name	Type
$head$	$[a] \to a$
$succ_{Int}$	$Int \to Int$
n	Int
$sing$	$a \to [a]$

Table 2.5: API containing *head*

When the matching algorithm in section 2.2 matches *head* with our current goal type, it would immediately notice that the return type of *head* is universally quantified and produce the substitution $\{a \mapsto Int\}$. The specialized type returned by the algorithm would be $[Int] \to Int$. However, this is not the only valid specialization of this type to produce a constructor for this goal. For instance, the expression below with type Int can be constructed manually from the API:

$$head\ (sing\ succ_{Int})\ n$$

To be able to generate this expression, another specialization of the type of *head* would have to be considered:

$$[?b \to Int] \to ?b \to Int$$

In general, there are an infinite number of valid instantiations for a universally quantified type variable when matching against a goal type t; each on the form: $?a_1 \to \ldots \to ?a_n \to t$ where n can be zero. Katayama uses this tactic when enumerating expressions in [Kat10].

In this project, it was decided to only consider the simple case, when $n = 0$. This is since instantiations where $n > 1$ does not make sense in several EDSL's, for instance in the Feldspar core language [Axe+10]. Furthermore, such instantiations taking extra arguments are not interesting for testing most of the time. In [Pal11], it was argued that setting $n > 1$ did not give any advantage in testing. In any case, if this functionality is required in the future, it can be implemented by redefining the MATCH procedure, see 2.2, to return a list of specialized types, for each $0 \leq i \leq n$, letting n be a user configurable parameter.

2.3 Generating expressions

Here we look at the complete term generation algorithm that was informally introduced in section 2.1. Similar to the MATCH algorithm, the first function we look at, GENERATE, does some basic computations and then delegates the more complicated work to an auxiliary function.

```
1: function GENERATE($t$)
2:     Bind all $\forall$ quantified variables in $t$ to some unique Data constructor types.  ▷ If for instance $t = a \to b$ then the resulting type might be $A1 \to B2$
3:     Substitute the undecided type variables in $t$ by some unique undecided variables.
4:     return GENERATEAUX($t$)
5: end function
```

On the second line of the algorithm, all universally quantified variables are substituted with unique dummy types as discussed in section 2.1.2. A similar transformation is done with the undecided type variables on the third line, to avoid capturing these undecided variables in later stages of the generation process[5]. The last line calls and returns the result of the auxiliary function. Before introducing this function, we need some additional definitions:

- Let USES(t) denote the number of uses for a constructor of type t, see 2.1.3, Γ a context (API) and $\mathbb{U}$ a set of guesses for undecided type variables, the rest of the generation algorithm can be found in listing 2.

The following literal interpretation of the algorithm is also included for clarity. The first step of the GENERATEAUX algorithm above is to case match on the current goal type t:

- If t is a function type, a lambda abstraction is constructed and the body of the lambda abstraction is generated in a context that has been extended to include the arguments of the lambda abstraction. This process may fail as can be seen from the usage of the standard Haskell type *Maybe* in the if statement from line 7 to 10.
- If t is not a function type, we first make local copies of $\mathbb{U}$ and Γ and try to find a matching constructor. Once again, this process may fail if no matching constructors are found in Γ. If a constructor is found, the number of uses for this constructor is decreased by one and there is once again a case match on a type, this time on the type of the constructor. Since the second case can be seen as a special case of the first one, with $m = 0$, only the first case will be considered.

[5]The procedure used to do this is the same one introduced in section 2.2.1.

For each of the argument types of the constructor c, t_i where $1 \leq i \leq m$, a respective expression e_i is generated. In this particular algorithm, the subgoals are generated from right to left as discussed in section 2.1.4. If any of the expressions fails to generate, i.e. if $me = \texttt{Nothing}$, then Γ and $\mathbb{U}$ is reset and `Nothing` is returned. Otherwise, all the expressions e_i were set and we can return the complete expression n applied to the arguments $e_1 \ldots e_m$.

```
function GenerateAux($t$)
    if $t$ is a function type $t_1 \to \ldots \to t_m$ then
        Generate unique variable names $x_1, \ldots, x_{m-1}$
        Add the constructors $(\textsc{Uses}(t_i), (x_i, t_i)),\ 1 \leq i < m$ to $\Gamma$
        $v \leftarrow \textsc{GenerateAux}(t_m)$
        Remove the $x_i,\ 1 \leq i < m$ constructors from $\Gamma$
        if $v$ is Just an expression then
            return $(\lambda x_1\ x_2\ \ldots\ x_{m-1} \to v)$
        else                                                    ▷ $v$ is here Nothing
            return Nothing
    else Retry up to 3 times                                    ▷ $t$ is here a value type
        $\mathbb{U}' \leftarrow \mathbb{U}$
        $\Gamma' \leftarrow \Gamma$
        $c \leftarrow$ A random matching constructor with positive uses in $\Gamma$        ▷ This line
    might introduce guesses for undecided type variables
        if $c$ is Just a constructor $c = (n, t')$ then
            decrease the number of uses for $c$ in $\Gamma$ by one
            if $t' = t_1 \to \ldots \to t_m \to t$ then             ▷ $t$ is here the same $t$ as on line 1
                for $i = m, m-1, \ldots, 1$ do
                    $me \leftarrow \textsc{GenerateAux}(t_i)$
                    case $me$ of
                        Nothing then
                            $\Gamma := \Gamma'$
                            $\mathbb{U} := \mathbb{U}'$
                            return Nothing
                        Just $e$ then
                            $e_i = e$
                return Just $(n\ e_1\ \ldots\ e_m)$
            else                                                ▷ $c$ has here a value type
                return Just $n$
        else                                                    ▷ $c$ is here Nothing
            return Nothing
end function
```

Listing 2: The auxiliary generate function

Chapter 3

Implementation

In this chapter the current implementation of the algorithm,as defined in chapter 2,is discussed. First, the data types used to represent types and values are presented, together with the functions used to work with the respective data types. Some of the limitations imposed by the specific representation scheme for the types presented are also discussed. After this, the Template Haskell module, and its functions, found in the library is introduced. Some usage examples of these functions are also presented. Finally, the current implementations of the MATCH and GENERATE algorithms are presented.

3.1 Types and expressions

This section provides an overview of the data types that were defined to more easily describe the different parts of the algorithm as defined in chapter 2.

3.1.1 Variables, Forall and Undecided

In Haskell, a type can contain universally quantified type variables. For instance, in the type for `id :: a -> a`, an implicit `forall` for the type variable `a` is introduced resulting in the final type `id :: forall a. a -> a`. This type says that `id` works for **any** type `a`, be it integers, list of strings or functions containing their own universally quantified type variables. QUICKGEN introduces another kind of quantifier, `Undecided`, as first introduced in section 2.1. Internally, the types used for representing variables are the following:

```haskell
data Quantifier = Forall | Undecided
type Nat = Int
type Variable = (Nat, Quantifier)
```

Simply put, a `Variable` in QuickGen is a natural number paired together with a value of type `Quantifier`. A natural number is used instead of, for instance, a string since comparing two natural numbers is far more efficient than comparing two strings.

3.1.2 Constructors

A constructor is the term used for the Haskell functions and values found in a user specified API. The intuition is that one of these terms can be used to construct parts of, or a complete, Haskell expression. `Constructor`'s are also returned by the function `randomMatching`, discussed in section 3.3.3.

The internal representation of a constructor is very simple:

```haskell
type Name = TH.Name
type Constructor = (Name, Type)
```

A `Name` is simply a type alias for names in Template Haskell [Lyn14]. A `Type` corresponds to the, possibly specialized, type of the constructor. Types are discussed in the next section. How to a specialize a type for a constructor when defined in an API is explained in section 3.2.

3.1.3 Types and simple types

Below are the two data types in QuickGen that are used to represent types in Haskell:

```haskell
data Type = Type [Variable] Cxt SType

data SType =
    FunT [SType]
  | VarT Variable
  | ConT Name [SType]
  | ListT SType
```

The first data type is used to introduce variables, and constraints on these variables, in types. The second data type, `SType`, has constructors for representing functions, variables, constructors and lists. For instance, the implicitly universally quantified type $a \to b \to b$ could be represented as:

```
Type [(0, Forall), (1, Forall)] [] (FunT [ VarT (1, Forall)
                                          , VarT (1, Forall)
                                          , VarT (0, Forall)
                                          ])
```

Each name of a type variable is turned into natural a number and a quantifier (here `Forall`), as explained in 3.1.1 . The type variable a is here turned into `(0, Forall)` and b is turned into `(1, Forall)`. A not so obvious transformation is done for the inner `SType`. The order of the type variables in the function type is reversed, compared to the original type. The reason this is done is to make the implementation of the type matching more efficient since only the return type of functions are considered during matching, see section 2.2. For now, it is enough to remember that function types are reversed.

The rest of the constructors, `VarT`, `ConT` and `ListT`, represent type variables, type constructors and lists, respectively. The observant reader may notice that there is currently no way to represent type variables with arguments, i.e. there is no way to represent the type of `return :: Monad m => a -> m a` This limitation, and ways to solve it, is discussed in section 5.1.

The reason there is an extra constructor `ListT` for lists instead of representing them as `ConT "List" a` [1] is just a convenience, making the implementation simpler. It also follows the representation for types used in Template Haskell [Lyn14].

The last type to mention here is that of constraints in types:

```
data Pred = ClassP Name SType
type Cxt = [Pred]
```

A constraint is simply a list of predicates. E.g. the constraints in the Haskell type `(Monoid a, Monoid b) => Monoid (a,b)` would be:

```
[ ClassP "Monoid" (VarT (0, Forall))
, ClassP "Monoid" (VarT (1, Forall))
]
```

[1] Also note that in this example you cannot use the name "List" for the list type constructor since a user might add this data type themselves.

3.1.4 Expressions

The following data type is used for the generated expressions in QuickGen:

```haskell
data Exp =
    ConE Name
  | AppE Exp Exp
  | LamE [Name] Exp
```

An expression is either the name of a `Constructor` 3.1.2, an expression applied to another expression or a lambda expression. The list of `Name`'s in a lambda expression will always be non empty. This data type is very simple when compared to the expression data type used by Template Haskell [Lyn14]. This implies that some Haskell expressions, like case- and let-expressions, cannot be generated by the library. This has very little effect on the usability of the library as an EDSL testing framework, since functions and values are the only visible parts outwards in an EDSL. Furthermore, if a user, for instance, wants a case match for a specific data type to be generated, a function directly corresponding to this case match can be added to the API. Below is an example showing how such a function would be defined for the `Maybe` Haskell data type:

```haskell
data Maybe a = Nothing | Just a

maybe :: b -> (a -> b) -> Maybe a -> b
maybe b f m = case m of
    Nothing -> b
    Just a  -> f a
```

It might be worth noting that this particular function, `maybe`, is already present in the Haskell Prelude[2]. Defining it again is probably unnecessary. In any case, the last step is to simply add this function to the API.

3.1.5 Other types

1. **Substitutions:** A mapping from type identifiers (natural numbers) to simple types and a list of variables contained in the simple type.

```haskell
type Substitution = Map Nat ([Variable], SType)
```

[2]It can be found here: `http://hackage.haskell.org/package/base-4.7.0.0/docs/Prelude.html#v:maybe`.

There is a value of type `Substitution` in the `EGState`, see section 3.3.1, when generating expressions. This value only contains mappings for `Undecided` variables and represents the set of guesses, $\mathbb{U}$, first introduced in section 2.1.1. The `Testing.QuickGen.Types` module contains several functions [3], for transforming and getting information from `Substitution`'s.

2. **Contexts** and **Uses**: A `Context` is a mapping from type identifiers to constructors paired together with the available `Uses` left for each particular constructor.

```haskell
type Uses = Maybe Nat
type Context = Map Id (Uses, Constructor)
```

The number of uses can either be `Just` a natural number or `Nothing`, the latter indicating that this particular constructor can be used an unlimited number of times.

3. **Class environments:** A mapping from names of Haskell type classes to a list of super classes paired with the Template Haskell instance declaration.

```haskell
type ClassEnv = Map Name ([Name], [TH.InstanceDec])
```

Currently the class environment is not used internally apart from being constructed in the Template Haskell module. Future work regarding the usage of this type is discussed in section 5.3.

4. **Languages:** Basically a `ClassEnv` paired together with a list of `Constructor`'s.

```haskell
data Language = L ClassEnv [Constructor]
```

A `Language` is one of the arguments for the library function `generate`, the other two being a `Type` and a `Seed` (`Integer`). The `generate` function, together with a value of these three types, are the only thing a user needs in order to generate well-typed terms using this library. Currently, the only way for an end user to construct a value of this type is via the function `defineLanguage`, introduced in the next section.

3.2 Template Haskell

The library contains a small module named `Testing.QuickGen.TH` with two exported Template Haskell functions, `defineLanguage` and `getType`, that a user can use to construct a `Language` or a `Type`, respectively. Example usage of these functions is shown in listing 3 below:

[3]For instance `lookupSubst :: Nat -> Substitution -> Maybe ([Variable], SType)` and `(|->) :: Nat -> SType -> Substitution`.

```haskell
ty :: Type
ty = $(getType [t| forall a b. (a -> b) -> [a] -> [b] |])

genInt = 0 :: Int
nil   = []
cons = (:)

lang :: Language
lang = $(defineLanguage [| ( genInt, nil, cons
                           , id, map
                           ) |])
```

Listing 3: Example usage of `getType` and `defineLanguage`.

The `getType` function simply converts a type, represented as a Template Haskell data type, into the representation used by this library. This function can be used to easily construct goal types to be used together with the `generate` function introduced in section 3.3.4 below.

The `defineLanguage` function, also seen in the example above, takes a tuple containing the constructors that should be available when generating expressions. The observant reader might notice the usage of `nil` and `cons` instead of the more common `[]` and `(:)`. In the used version of the library, it is not possible to directly include the latter constructors in the API definition, since variables are the only form of expression currently accepted. How one might extend the definition of `defineLanguage`, to include other types of expressions, is discussed in section 5.4.

One additional form when specifying a constructor in the API is also allowed:

```haskell
$(defineLanguage [| (map :: (a -> Int) -> [a] -> [Int], id) |])
```

This would specialize the type of `map` so that it can only be used to construct expressions of type `[Int]`. The constructor `id` however, having no type annotation, would still be associated with its most general type: $a \to a$.

3.2.1 Calculating a class environment

As mentioned when discussing Class Environments in section 3.1.5, a `Language` is isomorphic to a `ClassEnv` paired with a list of `Constructor`'s. The easiest way to explain how a class environment is calculated is probably by presenting the documentation for the internal function `getClassEnv` together with some Haskell type class instances:

```haskell
type ClassEnv = Map Name ([Name], [TH.InstanceDec])

-- | Given a list of class names iteratively find new classes
-- mentioned in either the constraints of a class name or in any of
-- the instances. Returns the 'ClassEnv' with information about all
-- instances for the initial classes and the discovered classes.
getClassEnv :: [Name] -> TH.Q ClassEnv

class Functor f => Applicative f where
    pure :: a -> f a

instance             Applicative [a]
instance Monoid a => Applicative ((,) a)

class Monoid a

instance             Monoid [a]
instance Monoid a => Monoid (Maybe a)
```

Example 3.1: Suppose a user includes the function `pure :: Applicative f => a -> f a` as a constructor in the language. If this is the only function in the language containing a type class constraint, the initial list of names, *ns*, will be `[Applicative]`. The algorithm proceeds as follows:

- Initialize a class environment *cenv* to the empty set.
- Loop until *ns* is empty:
    1. Remove the first name *n* from *ns* and ask Template Haskell about the superclasses *sups* and instances *is* of *n*.
    2. Extend *cenv* by adding a mapping from *n* to *is*.
    3. Extend *ns* by adding all type classes in *sups*, not yet mentioned in neither *ns* nor *cenv*.
    4. Extend *ns* by adding all type classes mentioned in any of the instances in *is*, but not yet mentioned in neither *ns* nor *cenv*.
- Return *cenv*

In our example with `[Applicative]` as the initial list, we would start by finding all information about the `Applicative` type class and proceed by adding `Functor`, a superclass of `Applicative`, and `Monoid`, mentioned in one of the instances, to *ns*. The next step would be looking up the information of, for instance, `Monoid`. Since there are no superclasses for this class, only the classes mentioned in the instances are added. In this case, one of the instances mentions a class which is already in *cenv*, `Monoid`, and this particular class is therefore not added to *ns*.

The algorithm above is the most straightforward way known to the author to find all information about the type classes possibly used when generating values. However, the proposed algorithm has some problems: when applied to, for instance, the list `[Num]`, the resulting class environment is calculated very quickly, but then compiling this value again might take noticable time even on a modern computer. The reason is that the class environment calculated using this algorithm grows very large due to the large amount of instances available in GHC. Furthermore, several type classes that do not seem to be relevant for generating functions using the `Num` type class are present in the final class environment[4]. The text representation of the final class environment calculated from `[Num]` is around 50000 characters long. The fourth step in example 3.1 was therefore removed from the algorithm used in the library.

In future versions, a modification to the fourth step should be added again, i.e. so that only classes that are relevant to the current problem are added to the class environment. Possible ways to solve this are discussed further in section 5.2.

3.3 ExpGen

The ExpGen module contains the core algorithm and methods to generate type-safe expressions. The generation starts in the appropriately named function `generate` which works by finding a random matching `Constructor`'s for the current goal type and recursively tries to generate expressions of the argument types of the constructor.

[4]I.e. the type class `MVector` in `Data.Vector.Generic.Mutable` in the `vector` Haskell package can be found in the class environment.

3.3.1 The ExpGen state

The `ExpGen` type is basically a state monad keeping track of and updating relevant information when generating expressions.

```haskell
type Nat        = Int
type NextLambda = Nat
type NextType   = Nat

type EGState = (NextLambda, NextType, [Context], StdGen, Substitution)

newtype ExpGen a = EG (State EGState a)
instance Monad ExpGen
instance MonadState EGState ExpGen
```

The type `EGState` is a tuple with several elements. The first two elements, `NextLambda` and `NextType`, are used to generate unique identifiers for lambda variables and type variables, respectively. The list (stack) of `Context`'s contain all `Constructor`'s introduced either in the language definition or in a lambda abstraction generated by the algorithm. If, for instance, the starting language contains `map` and `id` and the type to generate is `Int -> Double -> Int`, then the starting stack of contexts would be a singleton list only containing `map` and `id`. The next step would introduce a lambda abstraction `\x y -> ...`, effectively adding one more `Context`, containing the values `x` and `y` [5], to the stack of contexts and continue to generate an expression of type `Int` at the point of the ellipsis. If the expression finishes successfully, the top-most context on the stack is popped off and the algorithm returns the generated expression.

The `StdGen` is from the System.Random module and is used when selecting random constructors when generating expressions. The last value, with type `Substitution`, contains the current guesses for all `Undecided` type variables, i.e. it represents the set $\mathbb{U}$.

3.3.2 Match function

The function `match` found in the ExpGen module implements the algorithm discussed in section 2.2. The observant reader might notice that this function does not pattern match on the type of t, as is done in the pseudo code for Match. Instead, this is done in the `match'` function. However, the complete algorithm is still the same.

[5] With the appropriate types `Int` respectively `Double`.

```
match :: Monad m => Type -> Type -> StateT Substitution m Type
match gt t = do
    s <- match' gt t

    let t2  = // apply the substitution s to t
        t2' = // Convert all Forall quantified variables in t2 to
              // Undecided variables

    return t2'

match' :: Monad m => Type -> Type -> StateT Substitution m Substitution
```

`match` takes a goal type *gt* and a matched against type *t* and returns a type inside a state monad. The state being kept, if called with an initial empty state, is the current guesses for the `Undecided` variables encountered during this particular run for the function. This is correct if the current set of guesses, represented as a substitution, is fully applied to both arguments, `gt` and `t`, before being sent as arguments to this function. The only undecided variables found in either `gt` or `t` are therefore variables without any previous guesses, i.e. these variables are not present in $\mathbb{U}$. This function, when called in this manner, can therefore only introduce guesses for variables not present in the current set of guesses.

3.3.3 Selecting a random matching constructor

The following function gets a goal type *gt* and randomly selects a `Constructor` from the current context matching the given type:

```
randomMatching :: Type -> ExpGen (Maybe (Id, Constructor, Substitution))
randomMatching gt = ...
```

The function works by looking through each `Context`, filtering out those `Constructor`'s having no uses left and then runs `match` with the goal type `gt` and the type `t` for each of the remaining `Constructor`'s. As discussed above, the initial state for `match` will be the empty set. Further, the substitution containing the current set of guesses, $\mathbb{U}$, will be fully applied to both `gt` and `t`. If `match` succeeds, it returns a, possibly specialized, constructor of type `t` and a `Substitution` containing new guesses for `Undecided` variables. The constructor is then saved to a list of constructor candidates. If `match` fails, the list of candidates is unchanged.

The last step of the function is simply to randomly select and return one of the candidate `Constructor`'s by using the `StdGen` from the `EGState`.

3.3.4 Generating expressions

`generate` will be the only exported function in the ExpGen module, i.e. in future versions of this library, it is the only function from this module that will be visible to the end user:

```haskell
generate :: Language -> Type -> Seed -> (Maybe Exp, EGState)
generate lang t seed = runEG seed lang $ do
    t’ <- bindForall <$> uniqueTypes t
    generate’ t’

generate’ :: Type -> ExpGen (Maybe Exp)
```

The function `generate` is extremely simple, as its basically a wrapper for the function `generate’` where the real work is done. Here the different functions presented in the last section are combined into a complete algorithm that is used to generate expressions. This is also the algorithm presented in pseudocode in section 2.3.

Chapter 4

Example usage

In this section, some example usage of QuickGen is presented. The first example we look at is a simple language from the test suite included in the library. This example includes generating both polymorphic and monomorphic functions from a simple API. After this, two real world examples are presented. More specifically, a generator designed to mimic the behaviour of the handmade generator, used when testing the *Copilot* EDSL [Pik+12], is presented followed by a simple generator used to discover an artificially introduced bug in Feldspar [Axe+10].

4.1 Simple usage

Bundled with the QuickGen library is a test suite where one of the tests include testing the complete usage of the library. First an API, or more more correctly a value of type `Language`, is defined using the function `defineLanguage` as discussed in section 3.2. This API includes a selection of some common functions found in the Haskell Prelude:

```haskell
lang :: Language
lang = $(defineLanguage [| ( arbiInt    :: Int
                           , arbiDouble :: Double
                           , nil        :: [a]
                           , cons       :: a -> [a] -> [a]
                           , id         :: a -> a
                           , foldr      :: (a -> b -> b) -> b -> [a] -> b
                           , const      :: a -> b -> a
                           , sing       :: a -> [a]
                           , map        :: (a -> b) -> [a] -> [b]
                           , app        :: (a -> b) -> a -> b
                           , succInt    :: Int -> Int
                           , succDouble :: Double -> Double
                           )
                         |])
```

Listing 4: One of the API's used by the test suite. For clarity, all types of the constructors are written out explicitly.

The value `lang`, defined in listing 4 above, is used as the first argument to the function `generate` presented in section 3.3.4. In the current setup, `generate` will be called multiple times with `lang` and the two goal types, $a \to [a]$ and $[Int]$:

```haskell
-- | Generates values of type '[Int]'
genListInt :: Seed -> Maybe (Exp, Type)
genListInt seed = generate lang ty seed
  where
    ty = $(getType [t| [Int] |])

-- | Generates values of type 'a -> [a]'
genPolyList :: Seed -> Maybe (Exp, Type)
genPolyList seed = generate lang ty seed
  where
    ty = $(getType [t| forall a. a -> [a] |])
```

For both of these functions, a random list of seed values will be generated and each seed will then be passed as an argument to its respective function. The only step left in the test suite is to compile the expressions using the GHC API [ghc14]. This is done to ensure that the types of the generated expressions are correct. For `genListInt`, something similar to the following will be executed[1]:

[1]The complete implementation of the compilation process using the GHC API is beyond the scope of

```
checkTypeListInt exp = do
    let expStr = "(" ++ show exp ++ ") :: [Int]"
    runGhc $ do
        -- Load required modules. Specifically the Haskell Prelude
        -- and the module containing the API shown above.
        ...
        compileExpr expStr
```

The function compileExpr above takes a normal Haskell string and compiles this string as an expression using the modules loaded into scope. If successful, a value that can safely be cast into a list of integers is returned. The compilation can fail, however, with an error message similar to what GHC report for incorrect source files. If this happens, the current test case will be aborted, and the error message will be displayed to the user.

The function checkTypeListInt and the respective function for the polymorphic test case are then called 50 times each to check that only well-typed expressions are generated by the library. Listing 5 below shows a sample from the polymorphic function generator:

```
\m_0 -> const nil (succInt (id (foldr (\e_3 d_3 -> e_3) arbiInt
  (app (\c_3 -> nil) (foldr (\b_3 a_3 -> a_3) arbiInt (cons
  arbiDouble (foldr (\z_2 y_2 -> z_2) nil (sing (map (\x_2 ->
  arbiDouble) (sing (succDouble (const arbiDouble (app (\w_2 -> nil)
  (map (\v_2 -> arbiInt) (sing (map (\u_2 -> m_0) (const nil (sing
  (id (succDouble (id (app (\t_2 -> arbiDouble) (sing (map (\s_2 ->
  arbiDouble) (sing arbiDouble)))))))))))))))))))))))
```

Listing 5: A randomly generated polymorphic function.

The running time for the complete test case, random generation followed by type checking of 100 expressions, averages around 10 seconds on a modern laptop. Furthermore, the memory usage remains low for the complete duration of the test.

4.2 a DIY High-Assurance compiler

The Copilot EDSL is designed to monitor C programs by periodically sampling variables, arrays and return values of side-effect free functions [Pik+12]. A stream of

this thesis but the curious reader can find it in the GenTest module found here: https://github.com/solarus/quickgen/blob/master/testing/GenTests.hs sampled values with type `t` can be specified in Copilot by constructing a value of type `Stream t`.

```haskell
fib :: Stream Word32
fib = [0,1] ++ (fib + drop 1 fib)
```

Listing 6: The fibonacci sequence as defined in Copilot.

External C values are accessed by using one of the functions found in the `Copilot.Extern` Haskell module, for instance: `extern :: Typed a => String -> Maybe [a] -> Stream a`. In addition to `Streams`, a mechanism called *triggers* are also discussed by Pike et al. in [Pik+12]. A trigger is constructed by using the following functions:

```haskell
arg :: Typed a => Stream a -> Arg

-- | The trigger function takes a string representing an external
-- function in C. This function will be called every time the
-- second argument, its guard, is true. The arguments supplied to
-- the C function will be the current values of the streams
-- supplied in the third parameter.
trigger :: String -> Stream Bool -> [Arg] -> Trigger
```

A complete Copilot program, as generated by the Copilot generator, can be seen as a list of streams together with a list of triggers[2].

[2]The *copilot-core* Haskell package [Pik+14] also defines one more mechanism in its `Spec` data type called an *observer*. This mechanism is not generated by the handmade generator for Copilot programs included in this same package and is therefore not considered in the QuickGen generator either.

```
1  let s_0 :: Stream Bool
2      s_0 = {- A random expression of type Stream Bool -}
3
4      s_1 :: Stream Word64
5      s_1 = {- A random expression of type Stream Word64 -}
6
7      ...
8
9      s_n :: Stream Int32
10     s_n = ...
11 in do
12   trigger "f_1" {- A random expression of type Stream Bool -}
13           [ arg {- A random expression of type Typed a => Stream a -}
14           , ...
15           , arg {- A random expression of type Typed a => Stream a -}
16           ]
17   ...
18   trigger "f_m" ... [ ... ]
```

Listing 7: Example structure of a Copilot specification

– Before continuing, it should be noted that the types for the streams given in listing 7 above are just a selection of the available types for Copilot streams. A complete list of instances can be found in [Pik+14]. Furthermore, it is not required that, for instance, `s_n` on line 9 to have type `Stream Int32`. Instead, any type `t` with an instance of the `Typed` type class can be chosen. –

As can be seen in listing 7, for each of the variables `s_i` above, a stream with a matching type will be generated in the location of the comment. It is possible for each of these stream expressions to use any of the streams `s_i` in the final expression. For the triggers the generation is slightly more complicated. First we generate an expression of type `Stream Bool`. This is followed by a non-empty list of expressions always starting with a call to `arg` and ending with an expression of type `Typed a => Stream a`. Similar to before, any of the streams `s_i` can be used while generating random guards and arguments for the triggers.

4.2.1 A Copilot generator in QuickGen

A generator using QuickGen was constructed with the goal to mimic the behaviour of the generator provided by Copilot. Unfortunately, since type classes were not fully implemented at the time of the experiment, some restrictions had to be made to the constructors available in the API. Specifically, since functions and values with type class constraints does not function properly, these functions were specialized to a selected subset of the types within the type class. For instance, instead of having the more general function `app :: Typed a => [a] -> Stream a -> Stream a` in the API, two specialized versions[3] of this function were defined and included instead. The complete API can be found in listing 9.

This API contains a selection of the functions and values available when creating specifications using the Copilot EDSL. Before continuing, some things need pointing out. First of all, the function `drop`, used in listing 6 above, has been left out of the API because of some issues with totality of this function. Secondly, the API contains functions called `cycle...` that have no counterpart in the Copilot standard library. These functions are used to create infinite streams from finite list in the same way as `cycle` from the Haskell standard library works.

```haskell
cycleBool xs = let s = xs `app` s in s

-- These two streams are equivalent
x1 = [True,False] `app` x1
x2 = cycleBool [True, False]
```

Listing 8: The definition of the cycle function together with example usage.

To generate Copilot expressions, a variant of the `generate` function was defined that reruns the generation process with new random seeds until it succeeds.

```haskell
-- | A Copilot expression is represented using a Quickgen Exp and Type
type CopilotExpr = (Exp, Type)

genExpr :: Language -> Type -> StdGen -> (CopilotExpr, StdGen)
genExpr l t g = case generate l t seed of
    Nothing -> genExpr l t g'
    Just r  -> (r, g')
  where (seed, g') = next g
```

[3]Where `a` was substituted with `Word64` and `Bool`, respectively.

```haskell
lang :: Language
lang =
  $(defineLanguage
      [| ( sing             :: a -> [a]
         , cons             :: a -> [a] -> [a]
         , ifBool           :: Stream Bool -> Stream a -> Stream a
         , ifWord64         :: Stream Bool -> Stream a -> Stream a

         , true             :: Stream Bool
         , false            :: Stream Bool
         , cycleBool        :: [Bool] -> Stream Bool
         , appBool          :: [Bool] -> Stream Bool -> Stream Bool
         , not              :: Stream Bool -> Stream Bool
         , and              :: Stream Bool -> Stream Bool -> Stream Bool
         , or               :: Stream Bool -> Stream Bool -> Stream Bool

         , cycleWord64      :: [Word64] -> Stream Word64
         , appWord64        :: [Word64] -> Stream Word64 -> Stream Word64
         , signumWord64     :: Stream Word64 -> Stream Word64
         , absWord64        :: Stream Word64 -> Stream Word64
         , eqWord64         :: Stream Word64 -> Stream Word64 -> Stream Bool
         , lteWord64        :: Stream Word64 -> Stream Word64 -> Stream Bool
         , gtWord64         :: Stream Word64 -> Stream Word64 -> Stream Bool
         , plusWord64       :: Stream Word64 -> Stream Word64 -> Stream Word64
         , minusWord64      :: Stream Word64 -> Stream Word64 -> Stream Word64
         , timesWord64      :: Stream Word64 -> Stream Word64 -> Stream Word64
         , divWord64        :: Stream Word64 -> Stream Word64 -> Stream Word64

         , externBool       :: ExtBool -> [Bool] -> Stream Bool
         , externWord64     :: ExtWord64 -> [Word64] -> Stream Word64
         , ext1, ext2       :: ExtBool
         , ext3, ext4       :: ExtWord64

         , arbiBool         :: Bool
         , arbiListBool     :: [Bool]
         , arbiStreamBool   :: Stream Bool
         , arbiWord64       :: Word64
         , arbiListWord64   :: [Word64]
         , arbiStreamWord64 :: Stream Word64
         )
       |])
```

Listing 9: The initial API used when generating Copilot expressions.

To generate the list of streams, `s_1 ... s_n`, a function that starts with `lang` as the initial API and incrementally adds streams to the API, as they are generated, was defined:

```haskell
type CopilotName = String
-- | A Copilot stream is represented as a name together with an expression
type CopilotStream = (CopilotName, CopilotExpr)

someStreamTy :: Type
someStreamTy = Type [u] [] (ConT (mkName "Stream") [VarT u])
  where u = (0, Undecided) :: Variable

genStreams :: Int -> StdGen -> (Language, [CopilotStream], StdGen)
genStreams n g = go lang (map (('s':) . show) [1..n]) [] g
  where
    go l []        acc g = (l, reverse acc, g)
    go l (name:ns) acc g =
        let (r@(_, ty), g') = genExpr l someStreamTy g
            c    = (mkName name, ty)
            l'   = [c] `addTo` l
        in go l' ns ((name, r) : acc) g'
```

The first element in the return value of `genStreams` is the final language value, containing the complete API together with all `s_i` variables. The second and third elements are the generated streams and the updated standard generator value, respectively. It should be noted that this is not exactly the same tactic as employed by the Copilot generator since an expression `s_i`, constructed by the latter generator, can use any of the other streams, `s_j`, in the final expression, including `s_i` itself. Since, at least, some recursion in streams can be achieved using the `cycleBool` and `cycleWord64` functions, it did not seem necessary to add this extra step in the generation process.

Finally the `genTriggers` and `genSpec` functions were defined to generate triggers and a complete spec, respectively.

```haskell
type CopilotTrigger = (CopilotName, Exp, [CopilotExpr])

-- | Generates 'n' Copilot triggers using the language 'lang'. The
-- number of arguments for each trigger will be between low and high
genTriggers :: Language -> Int -> (Int, Int) -> StdGen
            -> ([CopilotTrigger], StdGen)
genTriggers lang n (low,high) g = ...

type CopilotSpec = ([CopilotStream], [CopilotTrigger])

genSpec :: StdGen -> (CopilotSpec, StdGen)
genSpec g1 = let (numStreams,  g2) = randomR (2,12) g1
                 (numTriggers, g3) = randomR (1,6) g2
                 (l, streams,  g4) = genStreams numStreams g3
                 (triggers,    g5) = genTriggers l numTriggers (1,5) g4
             in ((streams, triggers), g5)
```

The implementation of `genTriggers` function has been left out since it is similar to the `genStreams` function found above. The only function left to explain, `genSpec`, ties together the other specialized generator functions, by choosing the number of streams and triggers to generate, and threads the standard generator through the complete computation.

What remains now is converting this representation into a representation understood by Copilot. In this particular case study, the generated `CopilotSpec` was transformed into an expression very similar to the example structure found in listing 7, the difference being a call to the Copilot `prettyPrint` function before the `do` on line 11. This expression was later type checked, compiled and executed using the GHC API [ghc14]. The resulting values were compared to random values produced by the Copilot random generator and the two generators were found, by visual inspection, to be similar in operation[4]. However, this test is not enough to draw any real conclusions but it is an indication that the library is heading the right way. Furthermore, the running time of the QuickGen generator was significantly higher than that of the corresponding generator in Copilot. On a modern laptop, the former generator requires around one second to generate a complete specification, the latter, in many cases, is perceived to finish instantly. Still, it is the author's understanding that one second, in this case, is short enough to render the QuickGen generator usable as a source for random test data.

What is worth noting is that the size of the complete QuickGen generator was estimated to be about 25 to 30 percent the size of the Copilot generator. This estimation

[4]Modulo the number of available types and the inclusion of `drop` in the Copilot generator.

was made by counting significant lines of code for both generators[5]. The complete generator implemented using QuickGen can be found in [Spå14].

4.3 Feldspar

Since one of the motivations for this project was to implement a random generator for the EDSL Feldspar [Axe+10], it seemed natural that one of the use case examples was to test this language. As in section 4.2, a language, `lang`, and a generator, `gen`, were defined. Since these definitions are very similar to those in the previous examples, the specification of the language and generator can be found in Appendix A.

To test that, for instance, optimizations do not change the behaviour of the program, we will require two evaluation functions – one that optimizes the program and evaluates it and another one just performing the evaluations. We will call these evaluation procedures $\textsc{Eval}_{opt}$ and $\textsc{Eval}$, respectively. To test the optimizations done by the language, we need to generate and compile an expression e. We proceed by comparing the output of running both $\text{Eval}_{opt}(e)$ and $\text{Eval}(e)$.

Unfortunately, Feldspar does not export an evaluation function matching the behaviour of $\textsc{Eval}$, there is however, a function named `eval` that behaves like $\textsc{Eval}_{opt}$. To continue, another evaluation function, having the correct behaviour, was defined and added to Feldspar[6]. Next, a procedure was defined to repeatedly call `gen` followed by compiling the generated expressions using `compileExpr` from the GHC API. The result of evaluating the compiled expressions using the two evaluation functions, $\textsc{Eval}_{opt}$ and $\textsc{Eval}$, were then compared.

Using the API found in Appendix A, no bugs were discovered for the particular type of expressions generated. To proceed, a bug was artificially introduced into the `Feldspar.Core.Constructs.Num` module, incorrectly optimizing an expression $1 + n$ by replacing it with the value n. The bug was quickly discovered by the procedure and several counterexamples were produced. The exact modifications to introduce the bug can be found in Appendix B. The exact code used to compile and evaluate the generated expressions using the GHC API can be found in listing 12 in Appendix A

4.4 Summary

This chapter started with a demonstration of one of the test cases bundled with the QuickGen library. This particular example showed how to use the functions introduced in chapter 3, to describe an API used when generating, for instance, higher

[5]By removing comments, import statements and empty lines.

[6]The required changes can be found in Appendix B.

order functions. Fairly complicated expressions were generated reasonably quickly, averaging at about 100 ms per expression, while still using low amounts of memory on the host computer.

In the second example, a more refined generator was defined producing random expressions with form similar to those generated by the Copilot random expression generator. However, mainly due to type classes not being fully implemented, the API used in the former generator was more restrictive than the one available in the Copilot counterpart. The defined generator was also noticeably slower than the existing one but was still fast enough to be considered usable by the author. The biggest gain was noticed when comparing code size, where the generator defined in QuickGen was about 25 percent the size of its counterpart.

Finally, a generator for the EDSL Feldspar was defined. The API in this example, while small, contained functions taking higher order arguments with polymorphic types. However, no bugs were found in the Language using the API and a goal type of `Data WordN`. At this point, a bug was artificially introduced into the language. This bug was then quickly discovered by a simple testing procedure.

Chapter 5

Future work

Due to time limitations, the scope and complexity of the project was reduced by limiting the implementation of the final project. This chapter discusses some of these limitations and how they affect the final product.

5.1 Type variable arguments

Currently there is no way to represent type variables with multiple arguments. This functionality was never prioritized since type classes were not fully implemented in the library. To see the limitation, let us look at the `SType` data type found in the `Testing.QuickGen.Types` module:

```
1 data SType =
2   ...
3   | VarT Variable
4   | ConT Name [SType]
5   ...
```

This implementation makes it possible to represent, for instance, the type `Maybe Int` as `ConT "Maybe" [ConT "Int" []]`. However, it is not possible to represent `m a` as in `return :: Monad m => a -> m a` since there is no way to give arguments to the type variable `m`. The definition of `VarT` on line 3 above can be changed slightly, mimicking the definition of `ConT`, to allow type arguments:

```
1 data SType =
2   ...
3   | VarT Variable [SType]
4   ...
```

Now it is possible to, at least, represent type variables with type arguments. What is missing is to update the matching algorithm, introduced in section 2.2, to correctly handle type variables. However, this update is highly dependent on first implementing type classes correctly which is discussed in section 5.3.

5.2 Selecting a Class Environment

In the end of section 3.2.1, there is a problem presented where a large expression is constructed, consisting of around 50000 characters, when calculating the complete class environment for an API only containing the type class `Num`. Furthermore, the calculated class environment contained several type classes that did not seem relevent for the particular API. Two different approaches to handle this problem are discussed:

- Instead of automatically trying to calculate the complete class environment, a user could specify exactly which instances of a particular type class they are interested in at the same time as an API is specified. Then, only those specific instances are added to the class environment. Let us look at some of the instances of the `Monoid` type class as an example:

```
instance                                    Monoid [a]
instance                                    Monoid Ordering
instance (Monoid a, Monoid b)            => Monoid (a, b)
instance (Monoid a, Monoid b, Monoid c) => Monoid (a, b, c)
```

Listing 10: A selection of instances of the `Monoid` type class

  Furthermore, imagine that a user has the function `mempty :: Monoid m => m` in the API together with some functions working with lists and pairs. In this particular example it would probably not make sense to use the function `mempty` to produce values of type `Ordering` or `(Monoid a, ...) => (a,b,c)` since these values cannot be consumed by any of the other functions in the API. The user could instead specify in the API to only include the first and third instance removing the extraneous instances altogether. The function `mempty` could still be used, in this case, to produce values of complicated types; for instance `((([],[]),[]),([],[])) :: ((([a],[b]),[c]),([d],[e]))`.

- If the user is interested in a big set of instances, specifying all of them manually might be a very cumbersome task. Furthermore, new constructors added to the API might require additional instances to be added. Forgetting to do this last step might be easy, making the complete process error-prone.

Instead of trying to add every instance manually, it might be possible to look at the return types of the constructors, available in the API, to filter out instances that are not interesting. Using the same argument as in the suggestion above, if we only have functions and values producing lists and pairs it might be enough to only include the first and third instances. However, this kind of filtering might be to restrictive in some cases. For instance, if we add the following function to the API, `f :: a -> b -> a`, it is perfectly legal to apply `mempty :: Ordering` as a second argument to this function and a specific user might even be interested in expressions on this form.

None of the suggestions above completely solve the problem at hand. However, it might be possible to combine them, i.e. letting a user specify a set of instances that should be included and then taking the union with the set of instances that somehow relates to the constructors in the API.

5.3 Type Classes

Complete support for type classes was initially a goal of the project but was never fully realized. What is missing is to implement something similar to performing class *entailment* as defined in [Jon00]. This function would have a type similar to the following:

```haskell
data Pred = ClassP Name SType
type Cxt = [Pred]
data Type = Type [Variable] Cxt SType

entail :: ClassEnv -> Cxt -> Pred -> Bool
entail ce ps p = ...
```

The first three lines were introduced in section 3.1.3 and are repeated here for clarity. The intuition is that `entail` is given a class environment, a list of predicates (the initial constraints for the type) and a predicate that we want to find out if it is true or not given the class environment and the constraints. If `ps` is empty and `p` is equal to, for instance, `ClassP "Num" (VarT (a, Forall))`, this corresponds to finding an instance of `Num a` in the class environment `ce` [Jon00]. This function would be used, after type matching, to verify that all class constraints, for a given specialized constructor, can be satisfied using the current class environment and class constraints of the current goal type.

For instance, consider the following example during type matching with some goal type `Cxt => gt` against the following constructor `plus :: Num a => a -> a -> a`.

For these particular types, since a is universally quantified, the substitution $\{a \mapsto gt\}$ will be produced on line three in the `match` function found in section 3.3.2. This substitution is then applied to the type `Num a => a -> a -> a`, producing `Num gt => gt -> gt -> gt`. What needs to be done is to find out if gt really is an instance of the `Num` type class which is exactly what `entail` is defined to do. The constraints to send to this function are simply the constraints found in our goal type, `Cxt`. The next step is to identify our predicate(s) to verify. In this case, it will be `ClassP "Num" st`. In general, several predicates may need to be verified. For instance, if the type of our constructor is `(Num a, Num b) => (a, b)`, then both predicates `Num a` and `Num b` needs to be sent to `entail` as the third argument and both predicates must be satisfied.

If `entail` returns true, it successfully found an instance for `Num st`, and the constructor `plus` is safe to use as a constructor for a value of the current goal type.

5.3.1 Entailment and undecided variables

The `entail` function used above, as defined in [Jon00] and [Jon99], solves the problem for the definition of types used in standard Haskell, however, this project introduces another type that is not present in standard Haskell, namely the type of undecided type variables. Imagine that our current goal type is `?a` and that `mempty :: Monoid m => m` is the chosen constructor. This would introduce a guess for `?a` as `Monoid ?a => ?a`. If the type for `?a` is never fully realized, this could actually result in a compile error. To see why consider the following expressions:

```
λ> :t const (5 :: Int) mempty

<interactive>:1:18:
    No instance for (Monoid b0) arising from a use of ‘mempty’
    The type variable ‘b0’ is ambiguous
λ> :t const (5 :: Int) (mappend [] mempty)
const (5 :: Int) (mappend [] mempty) :: Int
```

In the first example, the type of `mempty` would still be `?a`. In the second one, the type would have been further specialized to `[?a]` giving us an unambiguous instance for the type class. This problem could be solved by trying to default the instances to some instance in the class environment. This is done in standard Haskell most notably with the `Num` type class. The `Exp` data type, see section 3.1.4, would also have to be updated, by adding a way to add type annotations to constructors if needed, possibly with the following definition of `ConE`:

```haskell
data Exp =
    ConE Name (Maybe Type)
  | ... -- As before
```

5.4 Supporting more expressions

Several types of expressions that are available in the Template Haskell `Exp` data type, see [Lyn14], are not yet understood by the function `defineLanguage` used when defining an API. For instance, it is currently not possible to directly specify `(:) :: a -> [a] -> [a]` and `[] :: [a]` to be available as constructors. This feature was never a priority since it has little effect on the type of expressions that can be generated by the library. In several examples throughout this thesis, a constructor named `cons` was used instead of `(:)`. The definition of this constructor would simply be `cons = (:)` in these examples making it possible to indirectly use this constructor without it being directly available in the API.

However, supporting additional expressions does make a big difference from a usability perspective and will therefore be a goal for the future. Doing so however might make it necessary to also add respective type constructors to the `Exp` data type as defined in section 3.1.4. An alternative would be to redefine the Template Haskell function `defineLanguage` to generate the kind of definitions seen above. I.e. if `(:)` is used as a constructor in the API, `defineLanguage` would generate a definition similar to `c1 = (:)` and then substitute `(:)` with `c1` in the language definition.

5.5 Subgoal ordering

In the current implementation of the `generate` function found in section 3.3.4, subgoals for constructors with functional types are generated from right to left due to an implementation detail. This has the effect that expressions generated by the current algorithm tend to be biased towards the right, i.e. the deepest nesting of constructors are more probable to be found on the rightmost arguments to functions. By instead generating the subgoals in a random order, one should be able to generate expressions with deep nesting in arbitrary subgoals. This makes the distribution of expressions more evenly spread out in the complete domain of the problem. However, this would most probably not effect the probability of balanced expressions, i.e. this kind of expressions might still be underrepresented in the final distribution.

5.6 Termination strategy

A method to ensure termination of the generation algorithm was presented in section 2.1.3. Several other strategies exist that might be worth examining. For instance, in [Pal11], an algorithm similar to the one presented in this project is discussed where each subgoal is limited by a size parameter. This parameter is then decreased for each recursive call.

Another strategy which seems reasonably to examine, is to instead associate each constructor with a function taking the current depth as a parameter and producing a weight for its particular constructor at the current depth. A higher weight would then equate to a higher probability to be chosen as a constructor for the current goal type. These functions could, for instance, be defined to favour constructors with many subgoals at lower depths and to favour constructors with few goals at deeper levels of recursion. It might even be interesting to let a user partially specify how these weights are calculated for certain constructors in the API since this would improve the usability of the library for generating expressions.

5.7 Compiling expressions

Currently, one of the only ways to compile generated expressions is to pretty print them followed by compiling the pretty printed value using the function `compileExpr` from the GHC API. This is also the way that was presented in chapter 4. A complete generator suffers from this limitation in that it has to depend on the GHC API. In addition, it has to include modules and source code, containing the API used by the generated expressions, while compiling the expression. A more automated way to construct real executable values from expressions is required.

In [Kat10], in addition to storing the type and name of a constructor in the language definition, a value of the data type `HValue` is also stored for each constructor:

```haskell
newtype HValue = HV (forall a. a)
```

This is used to store a real executable value. Later, when a generated expression is to be constructed, the value associated with each constructor is extracted and cast into its corresponding type. The constructors are then put together in the same fashion as was done when searching for a matching expression.

This tactic to construct executable values from expressions should currently be possible in QuickGen. However, with the addition of type classes, this is no longer possible since the complete instance for a type class needs to be fully realized to be able to find the concrete function to be executed. The following session in `ghci` should describe the problem:

```
λ> let hId = HV (unsafeCoerce id) -- OK
λ> let hMempty = HV (unsafeCoerce mempty) -- Not OK

<interactive>:21:32:
    No instance for (Monoid a0) arising from a use of ‘mempty’
    The type variable ‘a0’ is ambiguous
    ...
λ> let hMemptyList = HV (unsafeCoerce (mempty :: [a])) -- OK
```

Chapter 6

Conclusions

The goal of this project, as stated in section 1.4, is to implement a reusable library for automatically constructing generators for EDSL's by specifying an API. A library was implemented, and in chapter 4, we showed how to use the library to define generators for two different EDSL's, Copilot and Feldspar. In the case for Feldspar, the generator was simply defined by specifying an API together with a single call to a library function, very much in the spirit of the goal. However, to test the EDSL, another function had to be defined to compile the generated expressions into real values. Furthermore, due to type classes not being fully implemented, it did not generate expressions using the full range of types available in the language. Even so, the generated values were used to discover an artificially introduced bug in the language, leading us to conclude that the library can be used today for simple regression testing.

For QuickGen to become fully usable, and to completely reach the goal, the two issues, type classes and the need for compiled expressions, has to be solved. In the case for type classes, some of the work, calculating class environments, is already done. This was discussed in section 3.2.1. The actual entailment of type classes should be possible to implement using the discussion in section 5.2 as a base. To compile generated expressions, it might be possible to include a specialized wrapper for the GHC API in this library, making this process easy. In section 5.7, another tactic used by Katayama, is also discussed.

All in all, QuickGen is a big step forward in achieving the goal. Using the library and this thesis as a base, the goal can be realized in a timely fashion.

References

[Aug05] Lennart Augustsson. *Announcing Djinn, version 2004-12-11, a coding wizard*. 2005. URL: http://permalink.gmane.org/gmane.comp.lang.haskell.general/12747.

[Axe+10] Emil Axelsson et al. "Feldspar: A domain specific language for digital signal processing algorithms". In: *8th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE), 2010*. IEEE. 2010, pp. 169–178.

[DJW12] Jonas Duregård, Patrik Jansson, and Meng Wang. "Feat: Functional Enumeration of Algebraic Types". In: *Haskell'12*. ACM, 2012, pp. 61–72. DOI: 10.1145/2364506.2364515.

[ghc14] ghc-devs@haskell.org. *GHC/As a library*. 2014. URL: http://www.haskell.org/haskellwiki/GHC/As_a_library.

[Hof13] Martin Hoffman. *inductive-programming.org The IP Community*. 2013. URL: http://www.inductive-programming.org/.

[Jon00] Mark P. Jones. *Typing Haskell in Haskell: Revised version*. 2000. URL: http://web.cecs.pdx.edu/~mpj/thih/.

[Jon99] Mark P. Jones. "Typing Haskell in Haskell". In: *Haskell Workshop*. 1999. DOI: 10.1.1.41.470. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.470.

[JP08] Barry Jay and Simon Peyton Jones. "Scrap Your Type Applications". In: *Proceedings of the 9th International Conference on Mathematics of Program Construction*. MPC '08. Marseille, France: Springer-Verlag, 2008, pp. 2–27. ISBN: 978-3-540-70593-2. DOI: 10.1007/978-3-540-70594-9_2. URL: http://dx.doi.org/10.1007/978-3-540-70594-9_2.

[Kat10] Susumu Katayama. "Recent Improvements of MagicHaskeller". In: *Approaches and Applications of Inductive Programming*. Ed. by Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer. Vol. 5812. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 174–193. ISBN: 978-3-642-11930-9. DOI: 10.1007/978-3-642-11931-6_9. URL: http://dx.doi.org/10.1007/978-3-642-11931-6_9.

[Lyn14] Ian Lynagh. *Template Haskell hoogle documentation*. 2014. URL: http://hackage.haskell.org/package/template-haskell.

[Mar10] Simon Marlow. *Haskell 2010 Language Report*. 2010. URL: http://www.haskell.org/definition/haskell2010.pdf.

[Mar13] Simon Marlow. *The Glasgow Haskell Compiler*. 2013. URL: http://www.haskell.org/ghc/.

[Pal11] Michal H. Palka. “Testing an Optimising Compiler by Generating Random Lambda Terms”. In: (2011). URL: http://publications.lib.chalmers.se/publication/157525.

[Pik+12] Lee Pike et al. “Experience Report: a Do-It-Yourself High-Assurance Compiler”. In: *Proceedings of the Intl. Conference on Functional Programming (ICFP)*. Preprint available at http://www.cs.indiana.edu/~lepike/pub_pages/icfp2012.html. ACM, Sept. 2012.

[Pik+14] Lee Pike et al. *Hackage: copilot-core: An intermediate representation for Copilot*. 2014. URL: http://hackage.haskell.org/package/copilot-core.

[Spå14] David Spångberg. *Github: solarus/copilot-quickgen-test*. 2014. URL: https://www.github.com/solarus/copilot-quickgen-test.

Appendix A

Feldspar generator specification

```haskell
type DWord = Data WordN

lang :: Language
lang = $(defineLanguage [| ( plus'    :: DWord -> DWord -> DWord
                           , times'   :: DWord -> DWord -> DWord
                           , div'     :: DWord -> DWord -> DWord
                           , sum'     :: Vector DWord -> DWord
                           , zipWith' :: (DWord -> DWord -> DWord)
                                      -> Vector DWord
                                      -> Vector DWord
                                      -> Vector DWord
                           , map      :: (a -> b) -> Vector a -> Vector b
                           , range    :: DWord -> DWord -> Vector DWord
                           , id       :: a -> a
                           , const    :: a -> b -> a
                           , wordN0   :: DWord
                           , wordN1   :: DWord
                           , wordN2   :: DWord
                           , wordN3   :: DWord
                           )
                        |])

gen :: Seed -> Maybe (Exp, Q.Type)
gen seed  = generate lang ty seed
  where
    ty = $(getType [t| Data WordN |])
```

Listing 11: API and generator used when testing Feldspar. The `range` function is a similar to the Haskell function `enumFromTo`.

```
1  main = do
2      [n] <- getArgs
3      g   <- getStdGen
4      let rs = P.take (P.read n) $ randoms g :: [Seed]
5          vals :: [Q.Exp]
6          vals = P.map fst . catMaybes . P.map gen $ rs
7
8      let go = runGhc (Just libdir) $ do
9              _ <- getSessionDynFlags >>= setSessionDynFlags
10             addTarget =<< guessTarget "Language.hs" Nothing
11             load LoadAllTargets
12
13             setContext [ IIDecl . simpleImportDecl . mkModuleName $ "Prelude"
14                        , IIDecl . simpleImportDecl . mkModuleName $ "Language"
15                        ]
16
17             cs <- mapM (compileExpr . show) vals
18             P.length cs `P.seq` return (P.zip cs vals)
19
20     res <- catch go $ \e -> P.error $ "Should not happen" P.++ show (e :: SomeException)
21
22     forM_ res $ \(p, v) -> do
23         let p' = unsafeCoerce p :: Data WordN
24             doIt e prog = do
25                 let r = e prog
26                 r `P.seq` return (P.Right r)
27             f e prog = catch (doIt e prog) $ \e -> return (P.Left (show (e :: SomeException
28         r1 <- f eval  p'
29         r2 <- f eval' p'
30         if r1 P./= r2
31             then putStrLn $ "Not equal for: " P.++ show v
32             else return ()
```

.

Listing 12: Code used to compile expressions generated by gen above. The calls to the different evaluation functions can be found on lines 28 and 29.

Appendix B

Changes made to feldspar-language

All changes were made in version 0.6.0.3 of `feldspar-language`.

```haskell
----------------------------------------------------
-- In module Feldspar.Core.Interpretation

optimize' :: ( Typeable a
             , OptimizeSuper dom
             )
          => ASTF (dom :|| Typeable) a -> ASTF (Decor Info (dom :|| Typeable)) a
optimize' = S.fold $ \s as -> appArgs (Sym $ Decor undefined s) as


----------------------------------------------------
-- In module Felspar.Core.Frontend

reifyFeld' :: SyntacticFeld a
    => BitWidth n
    -> a
    -> ASTF (Decor Info FeldDomain) (Internal a)
reifyFeld' n = flip evalState 0 .
    (   return
    <=< codeMotion prjDict mkId
    .   optimize'
    .   targetSpecialization n
    <=< reifyM
    .   Syntactic.desugar
    )

eval' :: SyntacticFeld a => a -> Internal a
eval' = evalBind . reifyFeld' N32
```

Listing 13: `eval'` function that was added, i.e. evaluation function for unoptimized code.

```haskell
--------------------------------------------------
-- In module Feldspar.Core.Constructs.Num

-- Original code on line 110
constructFeatOpt (C' Add) (a :* b :* Nil)
    | Just 0 <- viewLiteral b = return a
    | Just 0 <- viewLiteral a = return b
    | alphaEq a b = constructFeatOpt (c' Mul) (a :* literalDecor 2 :* Nil)

-- Artificially introduced bug
constructFeatOpt (C' Add) (a :* b :* Nil)
    | Just 0 <- viewLiteral b = return a
    | Just 1 <- viewLiteral a = return b
    | alphaEq a b = constructFeatOpt (c' Mul) (a :* literalDecor 2 :* Nil)
```

Listing 14: Artificially introduced bug: optimizing $1 + n$ to n.