Macalester College
DigitalCommons@Macalester College

Mathematics, Statistics, and Computer Science Honors Projects
Mathematics, Statistics, and Computer Science

4-26-2016

Blossom: A Language Built to Grow

Jeffrey Lyman
Macalester College

Follow this and additional works at: https://digitalcommons.macalester.edu/mathcs_honors

Part of the Computer Sciences Commons, Mathematics Commons, and the Statistics and Probability Commons

Recommended Citation

Lyman, Jeffrey, "Blossom: A Language Built to Grow" (2016). *Mathematics, Statistics, and Computer Science Honors Projects*. 45.
https://digitalcommons.macalester.edu/mathcs_honors/45

This Honors Project - Open Access is brought to you for free and open access by the Mathematics, Statistics, and Computer Science at DigitalCommons@Macalester College. It has been accepted for inclusion in Mathematics, Statistics, and Computer Science Honors Projects by an authorized administrator of DigitalCommons@Macalester College. For more information, please contact scholarpub@macalester.edu.

In Memory of Daniel Schanus

Macalester College

Department of Mathematics, Statistics, and Computer Science

Blossom
A Language Built to Grow

Jeffrey Lyman

April 26, 2016

Advisor
Libby Shoop

Readers
Paul Cantrell, Brett Jackson, Libby Shoop

Contents

1 Introduction **4**
- 1.1 Blossom 4

2 Theoretic Basis **6**
- 2.1 The Potential of Types 6
- 2.2 Type basics 6
- 2.3 Subtyping 7
- 2.4 Duck Typing 8
- 2.5 Hindley Milner Typing 9
- 2.6 Typeclasses 10
- 2.7 Type Level Operators 11
- 2.8 Dependent types 11
- 2.9 Hoare Types 12
- 2.10 Success Types 13
- 2.11 Gradual Typing 14
- 2.12 Synthesis 14

3 Language Description **16**
- 3.1 Design goals 16
- 3.2 Type System 17
- 3.3 Hello World 18
- 3.4 Variables 18
- 3.5 Functions 19
- 3.6 Datatypes 21
- 3.7 Message passing 26
- 3.8 Polymorphism 28
- 3.9 Error Handling 30
- 3.10 Example Program 32

4 Blossom's Mathematical Model **34**
- 4.1 Desugaring 34
- 4.2 Blossom Lambda Calculus 36
- 4.3 Type Structure 36
- 4.4 Patterns 38

4.5 Expressions . . . 38
4.6 Grammar . . . 41
4.7 Program theory/CHRs . . . 41
4.8 Type Environment . . . 44

5 Proof **45**
5.1 Proof roadmap . . . 45
5.2 S&S's abstract language . . . 45
5.3 Equivalence with Blossom . . . 46
5.4 Type Judgments . . . 47
5.5 Damas Milner Inference Rules . . . 48
5.6 Stuckey and Sulzman's judgments . . . 49
5.7 Confluence . . . 50
5.8 Ambiguity . . . 53

6 Building Blossom **54**
6.1 Parsing . . . 54
6.2 Preprocessor . . . 55
6.3 Type Checker . . . 58
6.4 Evaluation . . . 62

7 Future Work **65**
7.1 Language Features . . . 65
7.2 Implementation . . . 67
7.3 Beyond . . . 68
7.4 Conclusions . . . 69

8 Notes **70**
8.1 Why Blossom is Strict . . . 70

Chapter 1

Introduction

When Mark Zuckerberg sat down to write the code that would one day become facebook, he made a choice that has haunted the company ever since. He started writing in PHP. Because of its expressive syntax and web-first design PHP was the obvious choice for creating the small application Zuckerberg had in mind. Besides, it was a popular language that was already powering a sizable chunk of the internet. However as facebook grew and grew, it quickly became apparent that the language could not meet the demands of millions and millions of users [17].

This problem became worse as more developers were hired. Inevitably documentation and comments began to lag behind the code and errors crept in. PHP eventually became such a hindrance that facebook created a whole new language called Hack. Hack is fully compatible with PHP but uses techniques called gradual typing and type inference to catch small errors before they bring the whole website crashing down [17].

This is the part of the script where I am supposed to tell you that Mark Zuckerberg made a mistake and should have written facebook in a compiled language with static type checking. Yet, Mark Zuckerberg made a perfectly rational choice when he picked PHP, it was simple and easy to program. He had no idea that one day a billion people would use his application nor did anyone fully understand the impact that massively distributed cloud systems would have on software engineering. Besides, predicting the lifespan of a program is generally believed to be undecidable, and facebook could have easily remained a quirk of Harvard that never picked up steam.

If we truly want to improve the quality of code, we need to move past the idea that proper planning will solve all of the problems that arise when applications are scaled up. Instead we need assume that developers will chose a language that meets their immediate needs, and design new languages that can grow with applications.

1.1 Blossom

In this paper I will present a new language of my own design called Blossom that fills this role. Blossom is a general purpose applications language aimed at developers who want the comforts of garbage collection, automatic memory allocation, and high level abstractions as well as the speed and power of a compiled language. It is a functional language with immutable data and built-in support

for message passing. Its clear and familiar syntax attempts to merge the power of its ideological ancestors, Haskell, ML, and Erlang, with the intuitive ease of Python and Ruby.

Blossom is served with just enough syntactic sugar to make coding with immutable data a joy not a pain. We hope that developers will try Blossom for its simple syntax and fall in love with its expressiveness and unprecedented ability to support applications as they scale up. This scalability is at the heart of Blossom's design philosophy. Even the name Blossom comes from the idea that programs grow in an organic bottom up fashion and may sprout in new and unpredictable directions.

The two main ingredients in Blossom's special sauce are message passing and its strong type system. Message passing is a versatile framework that can help us broaden our idea of a what a program is beyond the single-machine, single-process model, which is proving increasingly inadequate. It can also be used to formalize the notion of side effects and allow developers to prove that functions are pure.

The true star of this paper though, is Blossom's completely inferable type system, which allows developers to write code without any type signatures without losing the benefits of static type checks. It supports both vertical polymorphism through inheritance and horizontal polymorphism through overloaded functions and behaviors.

Before I present Blossom, I will begin by exploring some of the theoretic frontiers of type systems and discuss how they inform Blossom's design. Then I will describe Blossom, first as a developer would use it, then in mathematic terms, which will be used to prove that Blossom's type checker does indeed infer the most general type for any expression. I will also present a working Blossom interpretor that features a type checker for Blossom programs based on Jones's implementation of a Haskell type checker [8]. Finally I will discuss the next steps needed to turn Blossom into a fully functioning language as well as the exciting implications of its design.

Chapter 2

Theoretic Basis

2.1 The Potential of Types

Types are usually thought of as a safety mechanism, since static type checks prove that programs will not encounter a certain subset of errors at runtime [13]. However types also serve a number of other, less formal roles. For example, the type signature of a function often helps explain what that function does and serves as a kind of automatic documentation [9]. Type information has even been used to create a search engine called hoogle, which searches the Haskell libraries for functions based on their type signature. This search engine has proved to be an invaluable tool in implementing the Blossom interpretor described in chapter 6.

Types can also be used to make side effects explicit [12]. For instance, failure types in Swift formalize the side effect of failure, allowing developers to identify fault points and perform the necessary error checking. Haskell takes this principle a step further and makes all side effects explicit though the use of monads. This means that functions have no secrets: if the return value is not wrapped in the IO monad, then a function definitely won't write to some file, and a function that doesn't return a State monad definitely didn't change the state of the program.

Types also help programmers handle higher level abstractions and support code reuse through polymorphism. When developers can write functions that operate on several types instead of just one, they tend to work faster, and code becomes easier to refactor.

Researchers are constantly pushing the boundaries of what types can do. The following section summarizes the most common type systems and presents several exciting new systems that aren't well known outside of academia.

2.2 Type basics

At their core types are sets of values. For example, the type Int describes the set $\mathbb{Z}$ and Boolean describes the set $\{True, False\}$ Most languages also contain support for record types, which describe the set of values that are grouped together into a single structure [13].

If a type system supports recursive types, records can contain values of the same type as themselves. This allows us to defined stacks, linked lists, trees, and several other useful data structures.

Functions also have types, which are commonly referred to in the literature as arrow types or function types and are written as $t_1 \to t_2$ where t_1 is the type of the domain of the function and t_2 is the type of the range of that function.

Programs are considered to be well typed if it can be proved that every function call in the program is applied to a set of arguments with the same type as the domain of the function. These proofs are constructed by creating a set of constraints of the form $A \vdash e\colon \tau$, which should be read as "the type of expression e with respect to the type environment A is τ" and then creating an algorithm that proves that each expression in the language fulfills these constraints [13].

For some languages, it is impossible to generate these constraints. In common parlance, these are called weakly typed languages although that phrase has a tendency to change its meaning based on the personal biases of the author. These languages are often dynamically typed, meaning that a value of type t_1 can be assigned to a variable instantiated with a value of type t_2. These languages therefore must perform dynamic type checks to catch type errors such as multiplying an integer with a string.

2.3 Subtyping

Simple type systems are systems that require identifiers to have a single concrete type. This makes things easy for a type checker to verify that the types of expressions are correct, but simply typed languages tend to be very verbose since you have to define define separate functions with different names for related operations such as adding integers and adding floats.

While simple type systems mirror the internal working of the processor, they do not have much in common with the way that programmers think, and so we have come up with a number of abstractions that reduce the amount of code we have to write. The most common of such abstractions is the notion of subtypes.

The type environment of a subtyped program can be thought of as an acyclic directed graph of inheritance. If there is an edge from type t_2 to t_1 then we say that t_1 is a subtype of t_2 or $t_1 <: t_2$. Subtypes are transitive such that if $t_1 <: t_2$ and $t_2 <: t_3$, then $t_1 <: t_3$. Because inheritance graphs are acyclical, if $t_1 <: t_2$ and $t_2 <: t_1$ then $t_1 = t_2$. Parent types always describe records with the same or fewer fields than their children and so we say that they are smaller even though they describe a larger set of values [13].

In some languages such as Java, which only allow single inheritance, the inheritance graph is a tree. This model of subtypes is both easy for programmers to understand and easy for a type checker to verify since it prevents cases where the compiler has to make a choice about which parent to inherit a given function from. Regardless of whether the graph is a tree, we can construct a smallest type, Top [13]. Every other type is a subtype of Top and Top has no parent type. If the inheritance graph is a tree, then Top is the root of that tree (in languages like Java, Top is referred to as Object). Although it is trivial to create a Top type in any subtyping system, not all subtyping systems contain a top type.

Top can also be thought of as the set of all values that a given language can describe. This leads naturally to the notion of a bottom type Bot, which describes the empty set. This means that Bot is a subtype of all other types [13]. Bottom types are useful for describing functionality such as errors and undefined functions; however, they significantly complicate type checking and violate the single inheritance model favored by many subtyped languages.

Top types are especially useful if the language includes support for casting, treating values of one type as values of a different type. There are two types of cast, upcast and downcast. Upcasts tell the language to treat a value as though it were a value of one of its ancestor types. For instance, if we define two Java-esque classes Corgi, and Dog, such that Corgi inherits dog, passing a value of type Corgi to a function that expects a value of type Dog results in an implicit upcast since that function will treat is argument as though it were a dog. Since subtypes can always be treated as members of their supertypes, upcasts will never result in a runtime error.

However the same is not true of downcasts, which are inherently dangerous and rely on the programmer to make sure that the value being downcast is actually a member of the target type. For instance, the following code will pass the Java type checker but still results in a runtime error:

```
Dog dog = (Dog) new Corgi();
(Rotweiller) dog; //Runtime error
```

Since subtyping type checkers handle down casts by assuming they will always succeed, they can be thought of as islands of dynamism in a statically typed sea.

Subtyping systems tend to be axiomatized, meaning that the type checker knows what types exist in the language and their relationships with one another before it preforms any checks. However as Frische et al. [6] have demonstrated, it is possible to build up a subtyping model of the data types within a program as the type checker progresses. Once this model is complete, the type checker attempts to reconcile it with the information it knows about the program and fails if a contradiction has been generated.

One of the biggest problems with subtyping systems is that the set of objects that developers wish to model does not always fit nicely into the directed graph model of subtypes. Some behaviors, such as comparability and hashability, cannot be captured by subtyping and for this reason most advanced languages provide typing capabilities beyond the simple subtyping model. For instance, Java provides Interfaces, which allow developers to require two different types to implement the same function without forcing the two types to share a common ancestor.

2.4 Duck Typing

Where subtyping is the most common approach to adding polymorphism to statically typed languages, duck typing is the most common approach to polymorphism in dynamic languages. Objects in subtyped languages are treated as dictionaries where the keys are the names of the fields and methods of the given object. Method calls and field references are little more than syntactic sugar that allow developers to access or modify these values.

Since keys may be added and removed to records at runtime it is impossible to perform static type checks and so errors that would have been caught at compile time in a static languages are caught during run time in a duck-typed language. This flexibility is duck typing's greatest strength and its greatest weakness. One the one hand, it allows developers to express complicated non-hierarchical behaviors with simple naming conventions and dynamic remodeling. On the other, these naming conventions rely on developers' familiarity with the code and the existence of clear documentation, two criteria that are notoriously difficult to meet.

As duck typed applications grow and expand, developers are forced to rely on third party tools like control flow analyzers and unit testing frameworks to be confident that their programs will not

encounter type errors. While these tools are powerful in their own right and also play an important role in the development of static-typed applications, they are rarely able to prove the absence of type errors the way that static type checkers can. Much research is being done to increase the type safety of duck-typed languages. Two of the leading areas of exploration, gradual typing, and success typing are described later on in this chapter.

2.5 Hindley Milner Typing

One of the biggest downsides of subtyping systems is that they do not distinguish between structures and objects. For example, in a pure subtyping system a list of integers and a list of strings are treated as separate types, which may inherit from the list of top level objects. Because of this, it is necessary to implement functions, such as taking the head of a list, for each type of list, or to implement it for the list of top objects and then use dangerous down casts to return a value of the appropriate type.

The solution to this problem is parametric polymorphism. Instead of creating one function from a list of top objects to a top object, we describe a family of functions from a parameterized list of α's to an α where α is a type variable that can stand for any type and list is a type constructor. Type constructors can be thought of as functions that take types as arguments and return types [7]. Several languages such as Java, Go, and C++ support type constructors. For example, Java implements lists as unary type constructors and the result of applying the List constructor to the type String is written as List<String>.

However not all types can be combined in this manner. For instance, the type Int<String> doesn't make any sense because how can you have an integer of strings? To overcome this confusion we need the notion of kinds, which are the 'types of types.' Types that take no parameters, like Int and String are referred to as a monotypes and have the kind *. The result of applying a Type constructor to a full set of arguments is also a monotype For instance, List<Int> has the kind *.

Type Constructors, also referred to as polytypes, have higher order kinds. For example, we write the kind of the List constructor as $* \to *$ to represent the fact that it must be applied to a single monotype before it becomes a monotype. The number of arguments a type constructor takes is equivalent to the number of arrows in its kind. For instance, a binary type constructor for dictionaries that takes types representing keys and values as parameters has the kind $* \to * \to *$.

Hindley Milner type systems (also referred to as HM systems) also support quantified types called type schemes. Type schemes are the bridge between type constructors and monotypes. They work by binding the type variables of a monotype to a quantifier. For instance, the type scheme representing the types of all lists can be written $\forall\alpha.List < \alpha >$. Note that this is different from the type $List < \alpha >$, which represents a single fixed, but unknown, type.

Type schemes support high level abstractions because they allow developers to write code that operates on data structures independent of the data that they contain. For instance, we can write the type of a function that takes the head of a list as $\forall\alpha.List < \alpha > \to \alpha$, which can be read as "A function from a list of objects of any type to an object of that type."

A type scheme can be instantiated by applying a substitution to it. Substitutions are sets of mappings from type variables to types. For instance, applying the substitution $\{\alpha \mapsto Int\}$ to the type $List < \alpha >$ results in the type $List < Int >$. A substitution θ is called a unifier of two types τ_1, τ_2 if $\theta(\tau_1) = \theta(\tau_2)$. A substitution θ is the Most General Unifier of two types if every other unifier of those types, θ_1 can be written as $\theta_2\dot{\theta}$ for some θ_2.

One of the biggest advantages of HM systems is that they allow us to find the principal type scheme of any expression [7, 3]. A type scheme is a principle type scheme if any other type that could represent that expression can be unified with the principle type [7].

Arguably the most useful feature of the HM type system is its ability to infer the principal types of expressions without the need for the developer to specify types at all [3]. These inferred types form a sort of free documentation that helps developers working on large teams [9]. Blossom will borrow elements from Hindley Milner systems so that the types of its expressions can be inferred without annotations.

2.6 Typeclasses

One of the problems with pure HM systems is that they cannot express behavioral polymorphism. For example, you can write a function that operates on the family of list types, but you cannot write a function that operates on all types that can be checked for equality. This means you would need separate functions for integer equality, string equality, boolean equality etc. The most famous extension to HM systems overcomes this limitation by using typeclasses to constrain which types a variable in a type scheme may represent [20].

Each typeclass consists of a list of functions and a constraint. The constraint is a predicate that takes a type and returns true if that type implements the functions. This allows us to write constrained type schemes as $\forall alpha.P(\alpha) \Rightarrow \tau$ where P is a predicate. [20]. For instance, Haskell defines a typeclass Eq, which consists of the predicate Eq, and the function ==. This allows us to write a function find that returns true if an element is in a list as follows:

```
find :: forall (Eq a) => a -> [a] -> Bool
find _ []       = False
find a (x:xs) = if a == x
                  then True
                  else find a xs
```

The first line of this code is a type signature that states that find takes a type a that can be equated, a list of objects of type a and returns a boolean. The second line uses Haskell's pattern matching syntax to state that the result of applying find to an empty list is False. The rest of the code states that if the head of the list is equal to the element we're searching for then the result is true, otherwise it calls find on the tail of the list.

Type classes can also be qualified by other type classes. For instance, the class Ord requires that its instances also instantiate the Eq class. Type classes are equivalent to first-order logic formulas [18] and they allow for a wide range of polymorphic behavior without sacrificing decidability or type inference.

One down side of type classes is that they do introduce the ability for expressions to be ambiguously typed [5]. In practice, this occurs quite infrequently and the introduction of optional type annotations and typeclass defaulting means that Haskell programmers often never run into any problems with ambiguity at all [8]. Furthermore the benefits gained by formalizing behavioral polymorphism far outweigh the negatives and so Blossom will incorporate typeclasses into its design.

2.7 Type Level Operators

One of the benefits of HM systems is that they allow the return types of functions to be determined by the types of their arguments. However this functionality is quite limited. Suppose you want to write a general multiplication function in an HM system with typeclasses, that can operate on integers and floats. You would start by writing a typeclass Mult that defines a single function * whose type is given by the scheme $\forall\alpha.Mult(\alpha).\alpha- > \alpha- > \alpha$, which can be read "a function from an object of a type that can be multiplied and another of those objects, to a third object of that type." However you can't use this typeclass to describe matrix vector multiplication because matrices and vectors are not the same type.

One obvious way around this limitation is to allow type schemes to contain operators that transform types. For instance, if we define a type level operator MultRT that takes two types and returns the type that is returned when those types are multiplied, we can write the type scheme of the multiplication function as $\forall\alpha\beta.Mult(\alpha,\beta) \Rightarrow \alpha- > \beta- > MultRT(\alpha,\beta)$.

Type level operators allow us to fine tune the types of functions to a higher degree than any static system discussed thus far. In fact they can be used to write entire programs that are evaluated by the type checker [4]. However, they have a major downside; they are usually undecidable. Since type signatures can contain functions, they can be viewed as a version of lambda calculus, meaning that the type signatures themselves are turing complete [12].

However this doesn't mean that type level operators are useless. Schrijvers et al [16] showed that it is possible to implement decidable type operations by limiting type operators to equality checks and unary operators that may not depend on other type operators. Unfortunately these limitations increase the likelihood that the type checker will reject valid programs because the operators depend on the developer to explicitly state all of the mappings that the type operators characterize[10].

In its first iteration, Blossom will not support type level operators because they add extra complexity to type signatures with little benefit. However, Blossom's has been designed so that its syntax can be easily extended and it is conceivable that developers may wish to implement type level operators as a language extension.

2.8 Dependent types

Types are often thought of as a safety mechanism because they prevent the developer from making errors like subtracting a string from a list, but they can't catch every error. For instance, division by zero is an error but it can't be caught by most type checkers because the error depends on the value, not the type.

Dependent types are a way of skirting around this limitation by basing the type of an object off of its value. For instance, the numbers 1 and 0 are both integers, but 1 has the dependent types Whole and Odd while 0 has the dependent types Natural and Even. By using these dependent types a type checker can in theory eliminate all runtime errors from a program [2].

If that sounds too good to be true, that's because it is. In Turing complete languages, dependent types are inherently undecidable [2]. For instance, suppose you want to write a function that converts strings to integers. It would make sense for you to specify that the type of this function as NumericString $\rightarrow$ Integer where NumericString is a dependent type that strings have if they only contain numeric characters. It is generally impossible to enforce these dependent types with static

checks because the data may come from user input which cannot be checked before the program is run. You could avoid this limitation by introducing a second function from Strings to NumericStrings that produces an error if the string is not numeric but that sort of defeats the purpose of dependent types in the first place.

However all is not lost; Brady and Hammond demonstrated that dependent types can be used in combination with a staged compiler that compiles code as it runs and performs runtime optimizations [2]. In practice their language functions like an interpreted language that carries out dynamic type checks until it reaches a point where the dependent type checks are decidable. The compiler then produces the final executable can run without carrying out any type checks at all.

Brady and Hammond's approach is to essentially gather all of the input data, freeze it and then create a dependent typed program that operates on this data. Essentially they flip the model of static code/dynamic data so that once the data is acquired it remains static while the code changes dynamically [2].

Brady and Hammond's approach is not the only way that dynamic types can be simulated, and much work has been done to make usable decidable dependent types through the introduction of functional dependencies [11]. Functional dependencies allow you to define multi-parameter type classes where the types of some of the parameters may depend on the types of the other parameters.

2.9 Hoare Types

An even more expressive variant of dependent types are Hoare types. Hoare Types assign each function a three tuple that describes the preconditions necessary for the function to execute properly, the type of the returned value and the postconditions that the return values will fulfill [12].

Like Dependent types, Hoare types are undecidable except in special circumstances. Because of this limitation, Hoare Type checkers do not fully validate programs; rather they perform basic type checking (such as making sure that you don't pass an int when you need a string) and generate a set of verification conditions. These verification conditions are decidable under some circumstances and can be used to prove the correctness of programs. However these cases are quite rare and the verification conditions tend to be used as a framework for code analysis tools that require lower standards of proof than traditional type checkers [12].

In an attempt to describe the types of expression as accurately as possible, Hoare types include subtypes, union types, intersect types, top types, bottom types, type negation, quantified types and kitchen sink types. They can even be used to express memory allocations, pointers, the size of the heap, and IO operations, which means that the complete Hoare type of a function will tell you in great detail exactly what the side effects of a function are.

However all of this complexity makes Hoare types extremely difficult to use. Type signatures are sometimes longer than the functions they describe and there are three different ways to express variable assignment. For instance, a generic swap function that switches the values at two memory addresses has the following type signature [12]:

swap : $\forall\alpha.\forall\beta.\Pi x\colon$ nat. $\Pi y\colon$ nat.

$m\colon \alpha, n\colon \beta.\{x \mapsto_\alpha m * y \mapsto_\beta n\}r\colon 1\{x \mapsto_\beta n * y \mapsto_\alpha m\}$

In plain English this can be read as, "For all types α and β and references x and y such that a value m of type α is stored at the location pointed to by x and a value n of type β is stored at location y the function swap will return the unit type and fulfill the postcondition that the value n

of type β will be stored at the location x and value m of type α will be stored at location y."

Like dependent types, Hoare types are interesting not because they're useful but because they show that types can be used as a framework for static program analysis that extends beyond the usual type safety checks. Neither dependent and Hoare types are supported by Blossom, but we have designed its syntax such that the static analysis tools used to verify Hoare types and dependent types could also be ported to analyze Blossom code.

2.10 Success Types

One of the fundamental problems with type checkers is that they will always reject some programs that will execute without type errors [13]. This problem stems from the fact that they only accept programs that can be proven to be free of type errors. We can avoid this problem by creating what is known as a success type checker that only rejects programs if it can prove that they will encounter a type error [9].

Success type checkers were invented by Lindahl and Sagonas to increase the quality of code written in Erlang. Data in Erlang is immutable so objects do have static types. However, functions cannot specify the types of their arguments and may return values with different types depending on their input. This means that Erlang programs cannot be validated by a traditional static type checker without removing many of Erlang's most important features [9].

Success type checkers use type inference similar to Hindley Milner inference. However success type systems also support union types, which are used to express the return types of functions that may return more than one type of value. Success type checkers are inherently optimistic, which means that when given a function F with the type $\alpha \rightarrow Int \cup String$ and a function G with the type $Int \rightarrow Int$, a success type checker will not reject the function $H = G \circ F$ because the program will never fail so long as the programmer has made sure that H is never passed a value that would cause F to return a String.

This lower standard of proof can still increase the quality of programs. To begin with, the inferred types serve as a free documentation aid, which can help developers understand the purpose of the function, the values it accepts, and what to expect it to return. Furthermore these annotations can help programmers catch subtle bugs in their programs. For instance, if a developer meant to write a function from a Float to a Float but the type checker reports that its type is actually a function from a Float to an Int (an error most likely produced by accidentally using integer division instead of float division), the programmer is alerted to the fact that their code probably contains a semantic error.

Success types can be thought of as a tool to help formalizing the shortcuts programmers often make when prototyping code and they could also serve as the starting point for code cleanup because they highlight problem functions that should be refactored or thoroughly tested [15].

Success types have gained little traction outside of Erlang because they only work for languages with statically typed values that also allow for functions to return more than one type, which is a rather uncommon design. Blossom's current design specification does not fit this model. However, we believe that success types could be used to help developers rapidly develop prototypes which could then be formalized into strictly typed code. Because this fits nicely with Blossom's growth oriented design, we hope to someday incorporate them into Blossom (see Section 7.1.7 for more details).

2.11 Gradual Typing

Success typing is closely related to the more general notion of gradual typing. Gradual typing seeks to improve the safety and performance of dynamically typed languages by allowing developers to add annotations that specify the types of functions. These annotations are usually enforced by dynamic type checks although in some cases it is possible for the interpreter to statically prove that they are unnecessary and ignore them [14]. This process allows static analysis tools to catch type errors even if they cannot perform complete type analysis.

Ideally this leads to a process of iterative improvement where the programmer adds more annotations or changes the code and then runs the type checker and then fixes any errors it finds. However this approach has its limits. Developers can't annotate third party libraries and introducing type annotations to an existing project can be a near Herculean task. Rastogi et al. [14] demonstrated that Type inference can be used to improve the effectiveness of gradual types but full type inference tends to be impossible in dynamic language unless you remove several of the features that would lead a developer to use that language in the first place [14].

Takikawa et al. [19] argue that because gradual types provide little benefit to the developer and because they significantly slow down code by preforming unnecessary dynamic type checks. Since any unannotated libraries must be treated as ill-typed by the gradual type checker, it is very difficult to completely remove the need for these dynamic checks. Although their analysis focuses on the language Typed Racket, their results suggest that unless the interpreter can statically differentiate between 'safe' calls and calls that require type checking, the cost of constant dynamic type checks is not worth the benefits [19].

As mentioned in the previous section, we hope to some day incorporate success types into Blossom and so we need to be weary of the potential downside of gradually typed system. However, Blossom's support for message passing provides a perfect method for separating strictly typed modules from gradually typed modules and so we believe that we will not encounter the issues raised by Takikawa et al.

2.12 Synthesis

Although the type systems proposed in the latter half of this chapter offer exciting new ideas about what types can do for the developer, they are all have fatal flaws. Type level operators, Hoare types and dependent types are all too complex to be practical, and in most cases, undecidable. Success types and gradual types are actually used in the real world however, they were created to address deficits in existing languages and have limited application outside of their existing uses.

The creation of these superimposed type systems speaks to the need for a new breed of language that allows for rapid prototyping and development combined with type safety. In such a language, developers could write poorly typed code that 'just works' and then use a typechecker to find the parts of code that need to be revised when it's time to prepare a release.

Such a type system would not be entirely without precedent. In fact, very few languages rely on just one type system. In particular, languages like C++ and Java, which rely primarily on subtyping, also incorporate elements of HM systems like parameterized types and limited inference in the case of C++.

In the following chapter I will present Blossom's type system, which combines elements of constrained Hindley Milner systems with bits of Subtyping systems. This system allows for complete

type inference, ad hoc overloading, and inheritance in such a way that developers can write code that looks like duck-typed code, but carries all of the guarantees that come with static type systems.

Chapter 3

Language Description

3.1 Design goals

Blossom is a language designed to help fallible humans build infallible systems. We believe that developers are more likely to chose the easy approach than the hard approach. We believe that the future is inherently unknowable and that no amount of planning will prevent an application from experiencing growing pains as more developers join the team and new features are added. Blossom's ability to infer types means that code always comes with free documentation and its message passing framework helps developers write small self contained modules with clearly defined interfaces.

Blossom's syntax aims to be simple and clear and is closely related to Erlang and Python. It provides syntactic sugar for common constructs and uses indentation rather than brackets and semicolons to delimitate expressions. Blossom strives to follow the famed principle of least surprise and its syntax is composed of easily understood rules that have as few exceptions as possible. Type annotations are only required for overloaded functions and in ambiguous constructs, but developers can add as many or as few type signatures as they wish. Blossom uses capital letters, and required parenthesis to create clear visual distinction between types and functions.

Error messages are written in clear, easy to understand language that is accessible to beginners and informative to experts. The standard libraries adhere to uniform conventions and names are simple and intuitive. Although Blossom borrows many concepts from theoretical computer science and advanced math, we attempt to codify these concepts in language that is more familiar to developers than it is to theoreticians.

Blossom will run on as many platforms as possible while minimizing the need for platform specific libraries. This means that Blossom provides abstractions similar to the python os library that deal with tricky issues such as file paths, IO operations, streams, etc.

Blossom's syntax also pushes developers into writing loosely coupled code with well defined interfaces that can be easily split off into discrete modules. Blossom's syntax is also designed to interface nicely with static analysis tools such as purity analyzers and code linters that can help developers find stylistic and semantic errors.

Blossom's support for message passing provides a great way of formalizing side effects. All IO methods and all changes to mutable data are implemented through message passing. Message passing also allows a single program to run on several machines at once and promotes the use of

asynchronous code.

3.2 Type System

Before I describe Blossom in greater detail, I will describe its type system. Like most languages, Blossom's type system is a hybrid system that borrows components from several systems. In particular Blossom combines elements of Constrained Hindley Minler and Subtyping systems. This allows us to combine the expressive power and inferability of HM systems with the common sense, easy to learn model of subtyping systems.

Types in Blossom are written with a capital first letter, e.g. Int, Char. Blossom also support type constructors that take types as parameters and return types. These constructors use the same syntax as Java so the type of a list of integers is written as List<Int>. We also define syntactic sugar for list types so we can write the type of a list of integers as [Int] instead.

The types of functions are written as the types of their arguments wrapped in angle brackets followed by an arrow to the return type. For instance, the type of a function from an Int to a Char is $< Int > - >$ Char the type of a function that takes two integers and returns an integer is $< Int, Int > - > Char$.

Blossom also allows types to contain variables, which are written as lower case words such as var and v. These variables may stand for any type. If the same variable is used multiple times in the same scheme, all instances of it represent the same type. For instance, the type of an append function is written as $< a, [a] > - > [a]$, which can be read as "a function from an object with type a, and a list of that type of object to another list of that type of object."

We can use type variables as type constructors. For instance, the function map operates on data structures that contain objects. It's arguments are a structure and a function that transforms the objects contained by the structure. The type signature of the function map can be written in Blossom as:

$< m < a >, < a > - > b > - > m < b >$

where the variables a and b represent monotypes like Int and Char and the variable m represents a unary type constructor like List or Set.

However not every unary type constructor can be mapped over. We need some way to express the fact that the type of the variable m must implement the map function. We do this by defining a Behavior. A Behavior is similar to a Haskell Typeclass or a Java Interface. It specifies a set of functions that will work on any type that implements the Behavior. For instance, we can define a Behavior called Mappable that specifies that all types that implement the Mappable Behavior define a function named map with the type given above.

We can specify that a type implements a behavior by writing a *Given* clause before the type. For instance, we can define a function lookup that looks for an element in a list that requires the types of the items to be equatable by stating that they must implement the behavior Eq as follows:

$Given\, Eq(a).\ < a, [a] > - > Bool$

We allow types to inherit other types. If the type $Corgi$ inherits the type Dog then all of the functions that can take a Dog will also take a $Corgi$. For a more formal description of Blossom's types, see section 4.3.

3.3 Hello World

All Blossom programs begin with a single function called main. Main may take zero or one argument. If it does take an argument, that argument will be a list of strings containing the command line parameters specified when the program was executed. A simple hello world program can be written as follows:

```
1 fun main() -> print("Hello World")
```

3.4 Variables

Blossom variables are strings that start with a lowercase letter and are composed of alphanumeric characters and underscores. Variables are assigned with a single equal sign.

```
1 name = "Joanna"
```

The left hand side of an equals sign may be a pattern instead of a variable name (see section 3.6 for more on patterns).

```
1 (a,b) = someFunction()
```

Aside from data declarations, class definitions, and instantiation, (all of which will be described later) everything in Blossom is an expression, which means a variable assignment must have a value. We have chosen to represent that value of variable assignment with Null, which means that the following code will result in a type error since, Null is not a Boolean.

```
1 if (a = f(x))
2 ...
```

Furthermore, all variables are immutable, which means that the following bit of code is a syntax error.

```
1 a = 12
2 a = 11
```

Immutability has a number of benefits, in particular, it makes the implementation of message passing much easier, however it also has its downsides. For instance, random number generators are required to keep a seed that is used to generate the next number in the pseudo-random sequence. This means that random number generators need to return both the next seed and a random number. Therefore if you want to write a function that roles two dice you would have to implement it as follows:

```
1 fun roll2Dice(seed) ->
2     (seed1, d1) = randInt(seed)
3     (seed2, d2) = randInt(seed1)
4     (d1 % 6, d2 % 6)
```

That's a lot of extra bookkeeping and so we provide syntactic sugar in the form of the rebind operator @ that allows developers to reinitialize a variable. The @ operator can be implemented

as a preprocessor command that simply replaces all instances of the variable name that has been rebound with a new variable name for the remainder of the scope.

However we choose to leave the precise details of the rebind implementation up to the compiler since it may be able to simply overwrite the memory of the old variable. This means that we can remove the unnecessary variables seed1 and seed2 and just write the following code:

```
1 fun roll2Dice(seed) ->
2     (@seed, d1) = randInt(seed)
3     (@seed, d2) = randInt(seed)
4     (d1 % 6, d2 % 6)
```

However this code is still messy. Fortunately, since this pattern is so common we allow developers to use the rebind operator as an infix operator such that the expression:

```
1 (@seed, d1) = randInt(seed)
```

can be rewritten as a more familiar looking:

```
1 d1 = seed@randInt()
```

This means that we can reimplement our original example in syntax that is very familiar to users of object oriented languages:

```
1 fun roll2Dice(seed) ->
2     d1 = seed@randInt() % 6
3     d2 = seed@randInt() % 6
4     (d1, d2)
```

3.5 Functions

Functions in Blossom are first class objects. They are declared with the keyword *fun* and must have a name that begins with a lowercase letter followed by parenthesized list of arguments. The body of the function is a block of expressions separated from the declaration by an arrow written ->.

Blocks consist of one or more expressions. If there is only one expression that expression may be placed on the same line as the -> deliminator, or it may exist on a newline with a higher indentation level than the previous non-empty line. Blocks with multiple expressions must start on a new line with higher indentation than the previous line. Each expression in the block must have the same indentation level.

This syntax is illustrated by the following examples:

```
1  fun func1(arg1, arg2 ...) -> inlineExpression
2
3  fun func2(arg, arg, ... ) ->
4      expression1
5      expression2
6
7  fun func3(arg, arg, ... ) ->
8      expression1
9    expression2  //error, mismatched indentation
10
11 fun func4(arg, arg, ... ) -> expression1
12     expression2 //error, blocks must start on a new line
```

Function bodies are always treated as a single expression. If a functions body is a multi-line block, Blossom treats the newline character as a built-in sequence operator that evaluates the first expression and then returns the second expression. For instance, $func2$ from the above example can be rewritten as:

```
1  fun func2(arg, arg, ... ) ->
2      sequence(expression1, expression2)
```

The return value of a function is the value of the last expression in its body. If developers wish to suppress that value, they can place a trailing semicolon which will force the value of the expression on that line to be Null.

Functions are called by passing a parenthesized, comma separated list of values to the function just like C, Java, and Python. As of the time of writing Blossom does not support partially applied functions (also known as curried functions) although the syntax has been designed so that they can be implemented in the future without any radical changes to the syntax.

Functions may explicitly state the types of their arguments by adding a colon and a type signature after the names. They can specify their return type by adding the same annotation after the argument list like so:

```
1  fun square(x : Int) : Int -> x * x
```

We can also use type variables to annotate the types of functions.

```
1  fun append(x : a, list : [a] ) : [a] -> ...
```

We also allow qualified variables to be used in function definitions. Because writing qualifiers takes up a lot of space we allow them to be written on the preceding line so the following declarations are equivalent (For more detail on qualifiers, see section 3.2).

```
1  fun find(x : Given Eq(a). a, list : [a] ) : [a] -> ...
2
3  Given Eq(a)
4  fun find(x : a, list : [a] ) : [a] -> ...
```

3.5.1 Overloading

We allow for programmers to overload function definitions as long as they they provide type signatures for all of the arguments and the return type of the function. We also require that the types of overloaded functions do not overlap, meaning that at most one instance of an overloaded function can apply to any given value. For instance, the overloaded function in the following example produces an error because Blossom has no way of knowing which declaration to use when the function overloaded is passed an Int.

```
1  fun overloaded(arg : Int) : Int -> ...
2
3  fun overloaded(arg : a) : Int -> ...
```

3.5.2 Lambdas

Blossom also supports anonymous functions or lambdas. Lambdas are declared with the same syntax as functions minus the keyword fun and the function name. Lambdas can be declared anywhere and used in any instance when a function would be required.

```
1  (a,b) -> a + (b/2)
2
3  //lambdas can be assigned to variables
4  lambda = (a,b) -> a + (b/2)
5
6  //lambdas are called the same way as functions
7  ((a,b) -> a + (b/2))(2,4)
8  lambda(2,4)
```

Lambdas can capture variables defined in the current scope. Since all values are immutable, captured values will never change and they will never effect the value of the original variable that was captured. Lambdas also exist within their own scope so any variable declarations or rebinds will not pollute any other namespace.

3.6 Datatypes

Blossom defines a number of primitive types including Booleans, Integers, Floats, Characters, Tuples, and Lists. Users may define their own data types. These user defined types come in one of two forms, Record Types and Algebraic Types.

3.6.1 Records

Record types allow users to group several values together to form a single object and are quite similar to C structs. They can be declared using the keyword data and they must specify the names and types of their fields using the following syntax:

```
1 data Student where
2     .name  : String
3     .class : String
4     .id    : Int
```

Record types can be created by using the name of the record type as a constructor function and passing in the values of the fields as parameters. The order of the parameters that the constructor takes is the same as the order of the fields in the data declaration so a Student can be initialized with the following code:

```
1 john = Student("John", "Sophomore", 10013423)
```

You can access the fields of records by using the standard object oriented dot notation:

```
1 john.class == "Sophomore"
```

Records types may also be parameterized. For instance, one way of implementing a 2-Tuple as record types would be to create a record with two fields labeled one and two. Because we want tuples to be able to hold any values, we would need to specify variable types for the fields of the tuple like so:

```
1 data Tup2<a,b> where
2     .one : a
3     .two : b
4
5 tup = Tup2(1, "word")
6 tup.two == "word"
```

Blossom's type checker would then be able to infer the type $Tup2 < Int, String >$ for the value tup.

3.6.2 Algebraic Datatypes

Algebraic data types are similar to record types however, instead of defining a list of fields, they define a list of constructors. For instance, we can create a type called *Color* that can handle RGB, RGBa, and HSV colors, by declaring a datatype with 3 constructors.

```
1 data Color where
2     RGB(Int, Int, Int)
3     RGBa(Int,Int,Int,Int)
4     HSV(Int,Int,Int)
```

Algebraic constructors behave identically to record constructors so we can create solid red by writing:

```
1     red = RBGa(255,0,0,1)
```

Developers can also declare parameterized algebraic types. For instance, lists can be represented in Blossom with the following algebraic type:

```
1 data List<a> where
2     Cons(a, List<a>)
3     Nil
```

The Nil constructor represents the empty list and the Cons constructor takes an item and an list and returns a new list with the item prepended to the list. Notice that we are allowed to leave off the parentheses for the Nil constructor since it takes no arguments. Also note that types are parameterized with a comma separated list of types surrounded by angle brackets. Given the above type we can write the list 1,2 as:

```
1 list = Cons(1, Cons(2, Nil))
```

3.6.3 Sugar

Because it's no fun to write Cons and Nil all over the place we provide the following syntactic sugar for lists:

```
1 [] == Nil
2 [a,b] == Cons(a, Cons(b, Nil))
3 [a|b] == Cons(a,b)
4 [a,b|c] == Cons(a, Cons(b, c))
```

and the type $List < a >$ can be written as $[a]$. We also provide syntactic sugar for tuples such that

```
1 (a,b) == Tup2(a,b)
2 (a,b,c) == Tup3(a,b,c)
3 etc.
```

3.6.4 Patterns

Fields of record types can be extracted by using dot notation, but the fields of algebraic types are unnamed. To access them we have to deconstruct them using pattern matching. For instance, the following code shows how a list can be deconstructed:

```
1     list = Cons(1, Cons(2, Nil))
2     Cons(a, Cons(b, Nil)) = list
3     a == 1
4     b == 2
```

Note that because of Blossom's syntactic sugar, the above code is equivalent to the following code:

```
1     list = [1, 2]
2     [a, b] = list
3     a == 1
4     b == 2
```

Blossom patterns can come in 5 forms. The simplest patterns are the variable patterns. Variable patterns are just variable names that are free within a scope. These patterns match any value and bind the value that they are matched with to the variable name.

```
1  var // variable pattern
```

Like variable patterns, wild card patterns can be used to match any value, however instead of binding that value to a name, variable patterns simply ignore the value.

```
1  _ // wildcard pattern
```

The third kind of patterns are literal patterns which, as the name would suggest, are denoted by a literal. These patterns only match expressions that evaluate to the same literal and they do not bind any values.

```
1  1        // literal pattern
2  "string" // literal pattern
```

Constructor patterns are compound patterns formed by replacing the arguments of a constructor with patterns. They match values that have been created by the same constructor as the constructor appearing in the pattern whose sub values match the subpatterns of the constructor pattern. If any of those subpatterns contain variable patterns, those variables will be bound to the appropriate values.

```
1  list = Cons(1, Cons(2, Nil))
2  Cons(a, Cons(b, _)) = list // match succeeds, a = 1, b = 2
3  Cons(3, Cons(b, _)) = list // match fails because the literal
4                             // pattern 3 does not match the value 1
```

Since it's sometimes useful to bind both a constructed values and one of the subvalues Blossom also supports 'as patterns,' which consist of a variable name followed by pound symbol and another pattern. The variable is bound the entire expression if the pattern on the right hand side matches. If the right hand side does not match, the next the whole match fails.

```
1  var#Pat // as pattern
2  var#Cons(a,_) //matches [1,2,4] a is bound to 1
3                // var is bound to [1,2,4]
```

3.6.5 Case Expressions

The examples in the section above show how patterns can be used for variable assignment, however they are more commonly used in used in case expressions.

A case expression consists of the keyword *case* followed by an expression followed by the keyword *of* and then several branches consisting of a pattern and an expression block separated by an arrow. Expression blocks for case expressions follow the same syntax as function expression blocks, so single expression statements may be in-lined but multi-expression blocks must follow a newline and have greater indentation than the pattern.

```
1 case expr of
2     Pat1 -> expr
3     Pat2 ->
4         expr
5         expr
6     Pat3 -> expr
```

When case expressions are evaluated, the value of the expression at the top of the expression is matched against each pattern in the order that they are declared. As soon as a match is found, the variables in the pattern are bound to the values of the expression, and the expression following the pattern is evaluated.

If the value does not match any of the branches in the case expression then an error is thrown(see error handling for more info). This allows us to write a factorial function using pattern matching:

```
1 fun factorial(x) ->
2     case x of
3         1 -> 1
4         _ -> x * factorial(x-1)
```

The branches of case expressions exist within a separate scope, which means that variables declared within case branches are not accessible to the surrounding functions. For instance, the following code will result in an error:

```
1 fun badFunc() ->
2     case someFunc() of
3         _ -> msg = "Hello World"
4     print(msg) //err msg is not defined
```

Like all Blossom expressions, case expressions have a value. The value of a case expression is the last expression the expression block that is pointed to by the first pattern that matches the expression at the head of the case expression. This means that we can rewrite the above function to remove the error as follows:

```
1 fun goodFunc() ->
2     msg = case someFunc() of
3         _ -> "Hello World"
4     print(msg)
```

This means that all the branches of a case expression must have the same type.

3.6.6 If expressions

Blossom also supports if expressions with optional elif branches using the following syntax:

```
1 fun factorial(x) ->
2     if x == 1 ->
3         1
4     elif x > 0 ->
5         x * factorial(x - 1)
```

The scoping rules for cases also apply to if expressions because they are simply syntactic sugar of case expressions that only match boolean values. This means that if expressions that don't contain an else block may result in an error, however since these errors are lazily handled, they will not always result in a runtime error (see section 3.9 for more details about how that works).

3.7 Message passing

Message passing is a versatile tool that allows developers to formalize side effects, write asynchronous code, perform interprocess communication and more.

3.7.1 Handlers

Developers can write their own message handlers to using the following syntax that closely mirror function syntax:

```
1  handler myHandler(arg1, arg2) -> expr
2
3  handler myHandler(arg : Int) : String ->
4      expr
5      expr
```

Like functions, handlers can take any number of arguments and they return the result of evaluating the expression that is their body. However we also allow handlers to own a mutable state. Handlers must define a initial value for the state using a *with* clause above the handler definition. The default value is an expression that is evaluated the first time a message is sent to the handler. The state can then be accessed and modified from within the handler body using the getState and putState functions. This allows us to write a simple counting handler as follows:

```
1  with 0
2  handler counter() ->
3      val = getState()
4      putState(val + 1)
5      getState()
```

A more useful example would be to implement random number generation as a handler:

```
1  with collectInitialSeed()
2  handler rand() ->
3      seed = getState()
4      num = seed@randInt()
5      putState(seed)
6      num
```

Although their syntax intentionally mirrors that of functions, handlers are not functions, they cannot be passed as arguments to other functions and they cannot be overloaded. However they are specially designed so that the type inference algorithm can treat them as though they are functions.

Messages can be sent to handlers using the built-in *send* and *request* functions that are treated as implicit fields that are defined whenever a new handler is created.

3.7.2 send()

Send is a one way function that will ignore the output of the handler and return immediately while the handler's body may or may not execute asynchronously. Blossom guarantees that messages will be handled in the order they are received, which is not necessarily the order that they were sent.

```
1 myHandler.send(x,y) //returns immediately
```

The one way nature of send means if the only messages a function passes are send messages then we can be sure that the function will always return the same value for the same parameters. This feature allows us to write purity analysis tools that could highlight impure code. It also allows compilers to preform optimizations such as automatic memoization, provided they send the same messages every time a value is looked up instead of recomputed.

3.7.3 request()

The other way of passing a message to a handler is to use the *request* function. *request* is the same as *send* but it blocks until the result of the handler is computed and returned.

```
1 myHandler.request(x,y) //blocks until the handler returns
```

Expressions that depend on request are not pure since the results of a potentially impure function may impact its result. Since all impure functions (including IO functions) eventually call request somewhere, the compiler knows which functions cannot be cached and purity analysis tools can highlight those sections.

3.7.4 Applications

Handlers can be used for a variety of computations that involve side effects. For instance, all IO functions rely on message passing in some form. Messages can also be used to implement nondeterminism, stateful computations, generator functions, and more.

Message passing encourages developers to write modular code with clearly defined interfaces. It allows developers to separate their algorithms from the implementation. *send* and *request* can also be used as to write interprocess and interthread communication and so asynchronous message passing can easily be promoted to parallel computation.

3.7.5 Message Passing Implementations

Ideally Blossom would allow users to implement the details of message passing by specifying that certain handlers run on certain threads or that messages from a certain part of the code to another should be handled synchronously. However, I have not yet developed a formal mechanism that would capture this behavior.

Currently Blossom does not allow handlers to be generated dynamically. This is in keeping with the restriction that handlers cannot be treated as values and passed to functions. However, this restriction may be lifted in the future if it becomes apparent that user defined handler implementations require the ability to add handlers to running applications. More research is required before this question can be answered.

3.8 Polymorphism

Blossom provides two methods for polymorphic abstraction, Behaviors and Inheritance. Behaviors allow for horizontal polymorphism similar to Java Interfaces and Haskell Typeclasses while Inheritance provides vertical polymorphism in the style of object oriented languages like Python and Java.

3.8.1 Behaviors

Behaviors describe what an object can do. They are declared by defining a list of stubs that a type must implement in order for it to implement the behavior. For example, the definition of the Mappable behavior is shown below:

```
1 m is Mappable when
2     fun map(m<a>, <a> -> b) : m<b>
```

Data types can implement behaviors by providing implementations for the stubs described by the behavior.

```
1 List is Mappable because
2     fun map(val, fn) ->
3         case val of
4             [a|b] -> [fn(a) | map(b,fn)]
5             _ -> []
```

Behaviors each define a predicate that can be used to describe the types of expressions by using the *Given* syntax. For instance, if you want to write a function that takes a generic container of integers and increases each of those integers by one, you could write that function as follows:

```
1 Given Mappable(m)
2 fun addOne(container : m<Int>) : m<Int> ->
3     map(container, (x) -> x + 1)
```

Given qualifiers may have multiple predicates separated by commas.

```
1 Given Mappable(m), Eq(a)
2 fun mapEqual(container : m<a>, value : a) : m<Bool> ->
3     map(container, (x) -> x == value)
```

Qualifiers can also be used to constrain the behaviors. For instance, if a type implements the Ord behavior, which allows values of a type to be put in order, then it must also implement the Eq behavior:

```
1 Given Eq(a)
2 a is Ord when
3     leq(a,a) : Bool
4     geq(a,a) : Bool
```

We can also constrain data types. Suppose we want to define a data type that represents a binary search tree by declaring a new type $BST < a >$. Since the objects that a binary search tree holds must be orderable, we should qualify the definition of BST like so:

```
1 Given Ord(a)
2 data BST<a> where
3     Node(a, BST<a>, BST<a>)
4     Empty
```

Qualifiers can also be used to implement a behavior for a family of types. For instance, we can express the fact that any list of equatable objects is also equatable using the following qualified implementation:

```
1 Given Eq(a)
2 List<a> is Eq because
3     fun equals(x,y) = case (x,y) of
4         ([], []) -> True
5         ([x1|xs], [y1|ys]) -> (x1 == y2) and (xs == ys)
6         _ -> False
```

In general you don't need to explicitly write out qualifiers since the type checker will be able to infer them. However there are a few cases where the introduction of qualified types introduces genuine ambiguity. For instance, if we have a behavior called Show, which defines a function show from a Showable object to a string and another behavior Parsable, which takes a string and returns the parsable object, the following code is untypable:

```
1 fun strPlusOne(str) ->
2     num = parse(str)
3     show(num + 1)
```

The typechecker is smart enough to figure out that the argument is a string but the only thing it knows about the variable num is that you can show it and add one to it. Unfortunately you can do both of those things to integers and floats so Blossom has no way of knowing whether to use the function from the integer implementation of Parsable or the float implementation.

Haskell deals with this kind of ambiguity by providing default types for several types. However since Blossom favors explicit over implicit syntax, we require the developer to explicitly state the type of the ambiguous value, which we believe is a light burden and may even help developers discover deeper flaws in their program logic. The above function can be fixed by providing an annotation:

```
1 fun strPlusOne(str) ->
2     num = parse(str) : Int
3     show(num + 1)
```

The ability to qualify behaviors allows us to create a hierarchy of behaviors. This opens us up to circular behavioral dependencies, which could lead to undecidable types, which would be no good. Fortunately there is a simple way to ensure that behaviors have no circular dependencies: we just stipulate that each behavior must be defined within a module before it is used in a given clause or instantiated.

3.8.2 Inheritance

The second form of polymorphism is inheritance. Record datatypes may inherit other record datatypes with the following syntax.

```
1 data Dog where
2     .name  : String
3     .owner : string
4
5 data Corgi inherits Dog where
6     .name    : String
7     .owner   : Sound
8     .royalty : Bool
```

The fields of the child type must contain all of the fields of parent type and the types of those must be the same. Subclasses can then be used anywhere their parent classes can be used.

In practice, inherited types can be viewed as existential behaviors. Each super type defines a new behavior whose stubs are the fields of the super type. Under the hood every reference to the super type is changed to a reference to a type variable that implements the super type.

Types may inherit multiple supertypes if they meet the requirements for inheriting from all of their superclasses. Because we wish to avoid the problem of circular inheritance, we require all subtypes to be defined after their parent types within the file. Subtypes must also explicitly implement each of the behaviors of their supertypes.

We require explicit implementations of the behaviors because often times developers will constrain behaviors with informal rules. For instance, the Mappable behavior expects implementations to respect the following two rules for any pure functions f and g:

```
1 map(data, id) == data
2
3 map(data, (x) -> f(g(x))) == map(map(data, g), f)
```

Enforcing these rules is beyond the capabilities of the Blossom type system (or any decidable type system for that matter) so it's up to the developers to make sure that they hold.

3.9 Error Handling

Blossom supports lazily evaluated errors in a manner similar to Swift. This means that functions will always return a value, but that value may be an error. Errors will cause a program to stop executing in one of two ways. The first is by passing an error value to a function or a case statement that does not explicitly accept error values.

If you want to write a function that won't force a runtime error when it is passed a failed value, you can append a question mark to the end of the argument name in the function declaration like so:

```
1 fun takeFallible(arg?) -> ...
```

Types may also be declared fallible by appending a ? (note that $<>-> t?$ and $(<>-> t)?$ are not the same.) This allows us to write functions like so:

```
1 fun concatError(lst : [a?]) : [a] -> case lst of
2     []     -> []
3     [a|as] -> [a | concatError(as)]
4     [?|as] -> concatError(as)
```

Note that we used ? as a pattern. We can do this because ? is actually a literal value that represents a failed value. However often times we want to differentiate between errors. For instance, you might want to handle a case expression failing to find a match differently from the way you would handle a divide by zero error. We solve this problem by creating a special behavior called Error, which defines a single stub that prints an error message. This behavior is defined as follows:

```
1 a is Error when
2     fun showError(a) : String
```

Users can then create their own errors by creating new types that implement the Error Behavior:

```
1 data DivByZero where
2     DivByZero
3
4 DivByZero is Error because
5     fun showError(dbz) -> "Divide by Zero"
```

These error types can then be matched in case expressions. Because the type checker knows that the patterns represent errors, not values, they will not impact the type of the case expression.

The second way to force an error to cause a runtime error is to use the force operator, which is represented by a exclamation point.

```
1     a = !(1/0) //fails here
2     someExpensiveFunction()
3     someOtherFunction(a) //not here
```

The force operator only forces errors that occur at the highest level of a constructed value. This means that the following code does not produce an error:

```
1 !(1, 1/0) //no runtime error
2
3 (1, !(1/0)) // runtime error
```

Unlike regular functions, passing a fallible parameter to a constructor will not force the error. However this behavior can be overridden by adding the force operators to their constructor functions like so:

```
1 data ErrorFreeList<a> where
2     EFCons(!a, !ErrorFreeList<a>)
3     EFNil
4
5 a = 1 / 0
6 b = [a] // no error
7 c = EFCons(a, EFNil) //error
```

You can also have a function body break early by throwing an error using the throw function.

```
1 fun alwaysFails() ->
2     throw(DivByZero)
3     someExpensiveFunction() //will never execute
```

You can also use ? as a literal that represents the default error. This is nice because it lets you write function stubs without defining the body like so:

```
1 fun stub(a:Int) -> ?
```

3.10 Example Program

Here is an example of a short Blossom program that makes use of the concepts described in this chapter.

```
1  data ParseError where
2      .msg : String
3
4  ParseError is Error because
5      showError(err) -> "ParseError: " + err.msg
6
7  a is Parsable when
8      parse(String) -> a
9
10 Int is Parsable because
11     parse(str) ->
12         if str == "" ->
13             throw(ParseError("Converting an empty string to a number"))
14         else ->
15             ints = map(str, charToInt)
16             foldRight(ints, 0, (acc, val) -> acc*10 + val)
17
18 fun foldRight(list : [a], acc : b, fn : <b,a> -> b) : b ->
19     case list of
20         [] -> acc
21         [x | xs] ->
22             rest = foldRight(xs, acc, fn)
23             fn(rest, x)
24
25 fun factorial(x) ->
26     case x of
27         1 -> 1
28         _ -> factorial(x - 1)
29
30 fun charToInt(char) ->
31     case char of ->
32         '1' -> 1
33         '2' -> 2
34         '3' -> 3
35         '4' -> 4
```

```
36          '5' -> 5
37          '6' -> 6
38          '7' -> 7
39          '8' -> 8
40          '9' -> 9
41          '0' -> 0
42          _   -> ParseError(``Non numeric character")
43
44  fun print(val) -> outstream.send(val)
45
46  fun getLine() -> instream.request()
47
48  fun main() ->
49      print("What is your name?")
50      name = getLine()
51      print("Hello " + name)
52      print("What is your favorite number:")
53      num = parseInt(getLine())
54      print("The factorial of ")
55      print(num)
56      print("is")
57      print(factorial(num))
```

Chapter 4

Blossom's Mathematical Model

Before we can begin proving the correctness of Blossom's type system we must first define a mathematical model of our language. This model will be based on lambda calculus, which is a method of writing recursive expressions that was invented by Alonzo Church in order to formally express algorithms. Lambda calculus is Turing equivalent and has been extensively studied. Nearly all of the literature on Hindley Milner type inference uses lambda calculus notation. This conversion is necessary because it reduces Blossom's plethora of sugary syntactic constructs into a more manageable set of 6 expressions.

The first half of this chapter will describe how regular Blossom can be translated into the Blossom Lambda Calculus (hereafter written BLC), culminating with the presentation of a grammar that defines BLC. The second half will show how the various data declarations, behavior declarations and implementations define a Program Theory and a Type Environment, which are mathematical objects used by the type checker to describe the relationships between types.

4.1 Desugaring

Before we model Blossom in as an abstract language, we first should first explain how several of Blossom's syntactic constructs are just syntactic sugar and show how that sugar can be removed.

4.1.1 Overloading

Overloaded functions are just syntactic sugar for behaviors. Each overloaded function corresponds to an implicit behavior that is implemented by the types of each expression that implements it. For instance, the following Blossom code snippets are equivalent.

```
1  fun bark(corgi : Corgi) : Sound -> expr1
2
3  fun bark(poodle : Poodle) : Sound -> expr2
```

```
1  a is Bark when
2      bark(a) -> Sound
3
4  Corgi is Bark because
5      bark(c) -> ...
6
7  Poodle is Bark because
8      bark(c) -> ...
```

4.1.2 Subtyping

One of the major innovations of Blossom's type system is that it supports subtype relationships as special behaviors. For instance, suppose we define a type Dog that is inherited by the type Corgi:

```
1  data Dog where
2      .name : String
3      .owner : String
4
5  data Corgi inherits Bark where
6      .name : String
7      .owner : String
8      .royal : Bool
```

We can use the fact that record fields are special functions to define an implicit behavior as follows:

```
1  a is _Dog when
2      name(a) -> String
3      owner(a) -> String
4
5  Dog is _Dog because
6      name(dog) -> dog.DOG_name
7      owner(dog) -> dog.DOG_owner
8
9  Corgi is _Dog because
10     name(corgi) -> dog.CORGI_name
11     owner(corgi) -> dog.CORGI_owner
```

Note that we have renamed the fields for Corgi and Dog so that they are unambiguous.

All of the functions that operate on the type Dog are replaced with qualified functions. For instance, the function:

```
1  fun bark(dog : Dog) : Sound -> expr1
```

Is converted to:

```
1  Given _Dog(a)
2  fun bark(dog : a) : Sound -> expr1
```

If a subtype overrides an inherited function, then that function is treated as a special case of function overloading. For example, writing:

```
1 fun bark(corgi : Corgi) : Sound -> expr2
```

Forces us to promote bark to an implicit behavior Bark:

```
1 Given _Dog(a). a is Bark when
2     fun bark(a) -> Sound
3
4 Corgi is Bark because
5     bark(corgi) -> ...
```

Every subtype that does not explicitly overload the bark function implicitly implements the Bark behavior with the same expression as the original bark function. This method is useful since it allows us to preserve decidable type inference. However it can lead to some tricky situations discussed in more detail in section 7.1.

4.2 Blossom Lambda Calculus

Lambda Calculus describes a grammar for writing expressions (symbolized with the letter e), which can be combined to generate larger expressions. Each expression has a type and the types of larger expressions can be inferred by combining the types of their sub-expressions.

However before we can define the expressions in BLC we must first define a model for types and patterns, which will be used to help us define those expressions.

4.3 Type Structure

Hindley Milner type systems typically support both types and type schemes. Types, often referred to as monotypes, represent a set of values. For instance, the type Int is equivalent to $\mathbb{Z}$. In contrast, type schemes represent families of types and are used to express the types of polymorphic functions.

4.3.1 Monotypes

Monotypes can take one of three forms. The simplest of these is the variable form. Variable types are written as lower case Greek letters, most often α and β. Variables represent any monotype, but not type schemes.

The second form are constructed types, which consist of a type constructor that is written T and an array of types $\bar{\tau}$. Constructed types are traditionally written $T\bar{\tau}$. However, when talking about actual types it is often useful to separate the constructor from the types it is applied to by wrapping the arguments in angle braces to distinguish between them, so that the type of a list of integers is written as $List < Int >$ instead of $List\,Int$.

Not all type constructors take arguments. For example, Int and Char are constructed types. We will also define the notion of concrete types, which are any constructed types that contain no type variables. It is also important to note that type constructors themselves are not types until they have been applied a full list of arguments. This is why $List$ is not a type but $List < \alpha >$ is.

The last form of monotypes are arrow types, which are used to represent the types of functions. They written $\bar{\tau} \to \tau$ where $\bar{\tau}$ is an array of the types of the arguments of a function and τ is the return type of the function. Like with constructed types, we will wrap the array of types that describes the function arguments with angle brackets to ease readability.

Monotypes can be described by the following grammar.

Monotypes τ	::=	α	Type Variable
	|	$T\bar{\tau}$	Constructed Type
	|	$\bar{\tau} \to \tau$	Arrow Type

4.3.2 Unification

Here I will define a binary operation called unification that equates two monotypes τ_1 and τ_2. Unification seeks to find a substitution ϕ that maps variables to types such that $\phi\tau_1 = \phi\tau_2$

This substitution ϕ is the most general unifier of two types τ_1 and τ_2 if every other unifier for those types, θ can be written as the composition of the substitution ϕ and some other substitution θ_1

4.3.3 Type schemes

Type schemes, written σ can represent either a monotype τ or a polytype. Polytypes are monotype τ whose variables $\bar{\alpha}$ have been quantified and satisfy a constraint C. We write these types as $\forall\bar{\alpha}.C \implies \tau$ They can be read in English as "The family of types that can be written in the form τ such that the variables $\bar{\alpha}$ in τ satisfy the constraint C."

4.3.4 Constraints

These constraints come in three forms. The first kind are equality constraints, which are written $\tau_1 = \tau_2$. These constraints are satisfied if τ_1 and τ_2 can be unified.

The second form of constraints are user defined predicates written $U\bar{\tau}$ where U is the predicate and $\bar{\tau}$ is the array of types to which the predicate is applied. These predicates are roughly equivalent to Blossom behaviors. For example, behavior Eq corresponds to the predicate $Eq(a)$ which evaluates true whenever a is a type that can be equated. For example the type scheme of the function "contains" which is defined as:

```
1  Given Eq(a)
2  fun contains(lst : List<a>, val : a) : Bool -> ...
```

is written $\forall\alpha.Eq(\alpha) \Rightarrow < List < a >, a > \to Bool$.

The third kind of constraints are conjunctions, which are written $C_1 \wedge C_2$. Conjunctions hold if both C_1 and C_2 hold.

4.3.5 Quantification

In traditional HM systems variables may be bound or free. Bound variables are so called because they are bound to either the existential or universal quantifiers whereas free variables are not bound to any quantifiers. All user defined type variables in Blossom are implicitly quantified with the universal quantifier. This means that the type scheme of the identity function, which is written in

regular Blossom as $< a > - > a$ is converted to the quantified type scheme $\forall a.\top \implies a \to a$ where $\top$ is a constraint that always holds.

Put it all together and you can describe the grammar of type schemes as.

σ	::=	$\tau \mid \forall\alpha.C \implies \tau$	Type scheme
C	::=	$\tau = \tau \mid U\bar{\tau} \mid C \wedge C \mid \top$	constraint

4.4 Patterns

We also provide a formal definition for patterns that is nearly identical to the Blossom syntax for patterns. We write variable patterns as x, literal patterns as k and wild card patterns as _. As patterns are written as $x\#p$ where x is a variable and p is a pattern. Constructor patterns are a data constructor P applied to an array of patterns $\bar{p}$.

Patterns can be described by the following grammar:

Pattern	p	::=	x	Name
		|	$P\bar{p}$	Constructor
		|	k	Literal
		|	$x\#p$	As pattern
		|	_	wild card pattern

4.5 Expressions

4.5.1 Constants

The simplest expressions in BLC are Constants, which are written k. Blossom supports Integer, Float, and Character constants out of the box. Not every expression with a constant value is a constant in BLC. For example, True and False have a constant value but they are implemented as constructors and so Blossom treats them the same way it would treat a constructor function like Cons.

4.5.2 Variables

The next kind of BLC expressions are variable expressions, which we will denote x. Everything in Blossom that is referred to by a name is a variable. This includes functions and constructors in addition to what we traditionally think of as variables.

Variables represent bindings to other expressions and they are evaluated by replacing the variable name with the expression that the variable points to.

4.5.3 Lambdas

Lambda expressions can be thought of as anonymous functions. They are written $\lambda p.e$ where p is a pattern that matches the arguments of the anonymous function and the expression e represents the function body. The following anonymous function.

```
1  (x) -> x + 1
```

is represented in our BLC as $\lambda x.x + 1$.

If a Blossom anonymous function takes multiple parameters, they will be represented as a single pattern in BLC. These patterns are created with the n-ary product constructor. For instance:

```
1  (x,y) -> x * y - 2
```

can be converted into the BLC expression $\lambda 2Prod(x, y).x * y - 2$ where 2Prod is the only constructor of the 2Prod type, which could be defined as:

```
1  data 2Prod<a,b> where
2      2Prod(a,b)
```

Like constant expressions, lambda expressions are considered base values, which means that they cannot be reduced to a simpler value.

4.5.4 Application

Lambdas can be applied to other expressions to generate new values. These application expressions can be thought of as function calls and they are written as $e_1 e_2$ where e_1 is an expression that evaluates to a lambda and e_2 is the expression that is passed as an argument to the lambda.

Note that unlike regular Blossom, which allows functions to be applied to multiple arguments, BLC functions are always applied to exactly one argument. Furthermore this argument is expected to match a product pattern. It makes sense therefore to wrap the arguments in a product constructor before passing them to the lambda. These product constructors also play a crucial role in ensuring that Blossom evaluates strictly (see the chapter 8 for more info).

These product constructors are the only values in Blossom that are allowed to have kind larger than $* \to *$. For instance, the product constructor 2Prod has kind $* \to * \to *$ can be written as in BLC as $\lambda x.(\lambda y.2Prod(x, y))$.

This means that the Blossom function call foo(a, b) is written in BLC as the compound application $foo((2Prod\, a)\, b)$ as opposed to $(foo\, a)b$.

An application expression $(\lambda p.e_1)e_2$ is evaluated by matching the value of e_2 with the pattern p. If this operation succeeds, then the variable patterns in p are bound to the values in e_2 to which they correspond and the expression e_1 is evaluated.

For instance, if e_1 evaluates to $\lambda 1Prod(x).x+1$ and e_2 is the constant expression 2, the application of $e_1 e_2$ can be evaluated by replacing x with 2 in the equation $x + 1$ yielding the expression $2 + 1$ which can be evaluated to 3.

4.5.5 Let

We also provide support for let expressions, which are a more powerful version of lambdas and application. They are written in our abstract language as $\textbf{let}\, \bar{x} = \bar{e}\, \textbf{in}\, e$ where $\bar{x}$ is an array of a variable names, and $\bar{e}$ is an array of expressions with the same length as $\bar{x}$. We will also refer to $\bar{x}$ and $\bar{e}$ collectively as a bindgroup.

Let expressions are evaluated by binding the name $x_i \in \bar{x}$ to the corresponding $e_i \in \bar{e}$ in both every other expression $e \in \bar{e}$ and in the expression e. Then the value of the expression e is computed and returned.

Blossom expressions such as:

```
1  a = 12
```

are not enough to define a let expression on their own. Rather they treat the rest of the block of expressions as the body of the let statement. For instance, the Blossom expression block:

```
1  a = 12
2  a + 3
3  print(a)
```

is represented in BLC as **let** $a = 12$ **in** $(seq(a + 3, print(a)))$ where seq is a built-in sequence operator that simply evaluates its first expression and then returns the second expression. If an expression block ends with an assignment:

```
1  x = exp
```

that assignment is treated as a let expressions that binds x to the value of exp in a null expression.

Every Blossom program is one giant let expression where $\bar{x}$ is the array of functions and global variable names, $\bar{e}$ is the array of expressions that the names are bound to, and the expression e is the function main.

4.5.6 Case

The expressions that have been defined so far are not unique to BLC, in fact they are identical to the expressions defined by Damas and Milner, with the exception of pattern matching. Fortunately pattern matching is expressible using only the expressions defined so far. Therefore most of the literature does not bother treating case expressions as separate expressions. However, it is easier to implement a type checker that supports case expressions than it is to write a preprocessor that translates them into other expressions, so for the sake of completeness I will include them in BLC. (The actual transformation from pattern matching syntax to pure lambda calculus is described in the next chapter).

A case expression is written as **case** e **of** $\bar{e}$ where e is an expression and $\bar{e}$ is an array of lambdas expressions that represent the different branches of the case statement. For example:

```
1  case x of
2      1 -> 1
3      a -> a / 2
```

is converted into the BLC expression **case** e **of** $[\lambda 1.1, \lambda a.a/2]$. We could reduce this further by treating case expressions as a lambda that takes an expression, and applies it a primitive match operation that selects a lambda from the list, and then applies that lambda to the expression.

We require that the types of all the lambda expressions in $\bar{e}$ are the same. This allows the type checker to skip the translation between case expressions and lambda expressions and just treat the case expression the same way that it treats lambda application.

4.5.7 Annotations

Blossom also supports – and in a few cases, requires – type annotations. In BLC, we write type annotations as $e : \sigma$ where e is the expression being annotated and σ is the type scheme that arises from transforming the type from a Blossom type to a type scheme in BLC. For example, the following blossom expression:

```
1  x : Int
```

is expressed as $x : Int$ and the expression:

```
1  x : <a> -> Int
```

is expressed as $x : \forall\alpha.\top \implies \alpha \rightarrow Int$. The result of evaluating an annotation is the same as evaluating the expression component of the annotation. In fact, annotations have no effect on how a program runs but they are sometimes necessary to remove ambiguity.

4.6 Grammar

The syntax of Blossom is therefore reducible to the following grammar:

Expression	e	::=	k	constant
		|	x	variable
		|	$e\bar{e}$	application
		|	$\textbf{case}\, e\, \textbf{of}\, \bar{e}$	case
		|	$\lambda p.e$	Lambda abstraction
		|	$\textbf{let}\, \bar{x} = \bar{e}\, \textbf{in}\, e$	let
		|	$e : \sigma$	type annotation
Pattern	p	::=	x	Name
		|	$P\bar{p}$	Constructor
		|	k	Literal
		|	$x\#p$	As pattern
		|	$_$	wildcard pattern
Monotype	τ	::=	α	type variable
		|	$T\bar{\tau}$	Constructed Type
		|	$\tau_1 \rightarrow \tau_2$	function type
Type Scheme	σ	::=	τ	Monotype
		|	$\forall\bar{\alpha}.c \implies \tau$	Polytype
Constraint	c	::=	$\tau = \tau$	Equality Constraint
		|	$U\bar{\tau}$	Predicate
		|	$c \wedge c$	Conjunction

4.7 Program theory/CHRs

In our proofs about Blossom's type system we will also make use of the concept of a Program theory introduced by Stuckey and Sulzman [18]. A program theory is a set of Constraint Handling Rules (hereafter abbreviated CHRs), which describe the relationships between constraints.

4.7.1 CHRs

CHRs are a rewriting system that is used to solve sets of constraints called a constraint store. There are two kinds of rules, propagation and simplification. Both are written as of a set of constraints called the head, a relational operator, and a set of constraints on the right hand side [1].

Simplification rules are written as:

$c_1, \dots c_n \iff d_1, \dots d_m$

If this simplification rule is applied to a constraint store that contains the constraints $c_1, \dots, c_n$ it replaces those constraints with the set $d_1, \dots, d_m$.

The other kind of Constraint Handling Rules are Propagation rules. They can be written:

$c_1, \dots c_n \implies d_1, \dots d_m$

If this rule is applied to a constraint store that contains the constraints $c_1, \dots, c_n$, it adds the set of constraints $d_1, \dots, d_m$ to the store.

CHRs can be used to determine whether a set of constraints holds by repeatedly selecting a rule at random and applying it to the store until either all the constraints are removed from the store or a failure constraint, written $\perp$, is generated. We will write the empty set of constraints as $\top$. $\top$ is only generated if all of the constraints in a store have been satisfied.

4.7.2 Transitions

We will define a transition $C_1 \to_t C_2$ that converts a set of constraints C_1 to a set of constraints C_2 as the result of successively applying CHRs to the set C_1. For instance, given a program theory consisting of the following CHRs:

CHR 1. $c_1 \iff \top$

CHR 2. $c_1 \implies c_2$

we can define a transition $\to_1$ that applies CHR 1 to a set of constraints. The result of applying $\to_1$ to the set $\{c_1, c_2\}$ is the set $\{c_2\}$

4.7.3 Termination

CHRs may result in infinite computations. For instance, the program theory:

CHR 1. $c_1 \iff c_2$

CHR 2. $c_2 \iff c_1$

will never terminate because successive applications of CHR 1 and CHR 2 will never generate a failure constraint or an empty constraint. Program theories that always generate a failure constraint or an empty constraint are considered to be terminating.

Although simplification rules may be applied infinitely many times when solving a constraint store, we need to put some restrictions on the application of propagation rules. For instance, applying the following Program theory to the store c_1 could generate an endless cycle of applying CHR 3 then CHR 2 then 3 etc.

CHR 1. $c_1 \iff \top$

CHR 2. $c_2 \iff \top$

CHR 3. $c_1 \implies c_2$

Since allowing this sort of cycle would severly cripple the ability of CHRs to solve problems, we prevent a propagation rule from being applied to a store to which it has already been applied [1]. Therefore the following transition is forbidden since rule 3 is applied twice to the same store.

c_1

$\to_3 c_1, c_2$

$\to_1 c_1$

$\to_3 c_1, c_2$

4.7.4 Confluence

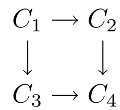
Program theories are confluent if the order in which their CHRs is applied to a store does not change the value of the final store. Confluence is easiest to demonstrate by counter example, the program theory:

CHR 1. $c_1 \iff \top$
CHR 2. $c_1 \implies c_2$
CHR 3. $c_2 \implies \bot$

is not confluent because it can generate two different final constraint stores from the initial store $\{c_1\}$. Applying CHR 1 first results in the truth constraint $\top$ while applying CHR 2 first adds c_2 to the constraint set and c_2 simplifies to $\bot$ not $\top$.

More formally, we can say that a program theory is confluent if for every pair of transitions $\rightarrow_1, \rightarrow_2$ such that $C_0 \rightarrow_1 C_1$, $C_0 \rightarrow_2 C_2$ and $C_1 \neq C_2$ there exists a C_3, $\rightarrow_3$ and a $\rightarrow_4$ such that $C_1 \rightarrow_3 C_3$ and $C_1 \rightarrow_4 C_3$.

This relationship can be modeled with a diagram:



4.7.5 Behaviors generate CHRs

Behavior implementations generate simplification rules. For instance, if a developer implements the Eq behavior for the type Student with the following code:

```
1 Student is Eq because
2     equals(student1, student2) -> ...
```

Then we add the simplification rule $Eq(Student) \iff \top$ to our program theory. Qualified implementations also generate simplification rules. For example, the Blossom code:

```
1 Given Eq(a). [a] is Eq because
2     equals(a,b) -> ...
```

gives rise to the simplification rule $Eq([a]) \iff Eq(a)$.

Propagation rules are generated by qualified behavior declarations. For instance, the behavior Ord is implemented with the Blossom code:

```
1 Given Eq(a)
2 a is Ord when
3     leq(a,a) -> Bool
```

This generates the propagation rule $Ord(a) \implies Eq(a)$ that can be read as "whenever a type implements the Ord behavior, it must also implement the Eq behavior."

4.7.6 Equivalence with First Order Logic

Program Theories can be converted into First Order Formulas that model the relationships between types in a program. We will define an operation [[]] that converts a program theory into a First

Order Formula by converting each individual CHR into a formula and then combining them with the conjunction operator.

[[]] converts simplification rules into biconditional formulas:

$[[c_1, \ldots, c_n \iff d_1, \ldots, d_m]] = \forall\bar{\alpha}(c_1 \wedge \ldots \wedge, c_n \leftrightarrow (\exists\bar{\beta}d_1, \wedge \ldots \wedge, d_m))$

This states that for all the variables $\bar{\alpha}$ that satisfy the constraints $c_1 \wedge \ldots \wedge c_n$ there exists a set of variables $\bar{\beta}$ such that the constraints $d_1 \wedge \ldots \wedge d_m$ are equivalent to the constraints $c_1 \wedge \ldots \wedge c_n$.

Propagation rules are converted into logical implications using the following method:

$[[c_1, \ldots, c_n \implies d_1, \ldots, d_m]] = \forall\bar{\alpha}(c_1 \wedge \ldots \wedge, c_n \supset (\exists\bar{\beta}d_1, \wedge \ldots \wedge, d_m))$

where $\supset$ is used to symbolize logical implication. the resulting formula can be read as stating that if there exists a set of variable $\bar{\alpha}$ that satisfy the constraint $c_1 \wedge \ldots \wedge c_n$ then there exists a set of variables $\bar{\beta}$ that satisfy the constraints $d_1 \wedge \ldots \wedge d_m$.

4.8 Type Environment

In our proofs we will also make use of the concept of a Type environment written Γ which is a set of mapping between variables and types. For instance, the Type Environment $\Gamma = \{x : \tau\}$ describes an environment where the type variable x is bound to the type τ.

We will use the notation Γ_x, which represents the type environment Γ where all of the bindings from the variable x to types have been removed. The predicate $(x : \sigma) \in \Gamma$ holds when the variable x is bound to the type σ in the type environment Γ.

Chapter 5

Proof

5.1 Proof roadmap

In this chapter I will prove that Blossom's type system is decidable and that the principle types of expressions can be inferred. This means that not only do developers not need to specify types, but that the types that are inferred capture the largest set of values possible.

Fortunately much work has already been done to prove the decidability and soundness of Constrained HM type systems, which have a great deal in common with Blossom's type system. In particular, Stuckey and Sulzman proved in their paper "A Theory of Overloading" that it is possible to overload identifiers HM types systems while preserving the type inference if the program theory that describes a program is confluent and terminating [18].

The proof will be described in the following steps: first I shall present and comment on Stuckey and Sulzman's abstract language and then I will demonstrate that for the purposes of type inference, BLC can be converted to Stuckey and Sulzman's abstract language. I will then briefly describe the type judgements that are used to infer the types of expressions. Next I will show that the program theories that Blossom creates are always confluent and terminating, which is sufficient to prove that it is possible to infer the principle types of expressions in Blossom.

Unfortunately, the introduction of constraints means that it is possible to write expressions whose types are ambiguous. In the last section of this chapter I will detail how these ambiguities arise, how they can be resolved, and introduce a novel method for automatically resolving ambiguities according to subtype relationships.

5.2 S&S's abstract language

Stuckey and Sulzman describe an abstract language (hereafter written SSLC for Stuckey and Sulzman's Lambda Calculus) for describing expressions that is very similar to BLC. Perhaps the most striking difference is the way in which they handle overloaded identifiers. In most cases, it is impossible to overload functions without specifying their types. Because of this limitation, Stuckey and Sulzman introduce a new syntactic construct that they call a Program abbreviated p. Programs are either an expression e or a statement that overloads the identifier f with the expression e. We will

write these statements as **overload** $f = e$ **in** p. It is useful to think of overload statements as top level let statements that require the user to specify the type of the expression that is bound to the identifier.

The rest of the expressions in SSLC closely resemble expressions in BLC and are described by the following grammar:

Program p ::= $\textbf{overload} f = e \textbf{in} p | e$
Expression e ::= $k|x|e\,e\prime|\lambda x.e|\textbf{let} x = e \textbf{in} e|(e : \sigma)$
Monotypes $\tau ::= \tau \rightarrow \tau | T\bar{\tau} | \alpha$
Polytypes $\sigma ::= \tau | \forall \bar{\alpha}.C \Rightarrow \tau$
Constraint C ::= $t_1 = t_2 | C_1 \wedge \ldots \wedge C_2 | U\bar{t}$

There are a few key differences between BLC and SSLC. For instance, SSLC lambda expressions bind expressions directly to identifiers instead of using pattern matching. There is no support for case expressions and let expressions only bind one identifier at a time. Fortunately there are more similarities than differences. Constraint, Types, and Type schemes are the same in SSLC and BLC. So too are constant, variable, application and annotation expressions.

5.3 Equivalence with Blossom

In this section I will show how BLC can be converted into SSLC.

5.3.1 Overloading

BLC does not treat functions that are overloaded differently from functions that are not. In the previous section, I proved that overloaded functions, subtyping and behaviors can all be written with using the behavior syntax. Because of the way that the behavior syntax has been defined, we can derive a type annotation σ from each behavior implementation.

Since Blossom behavior declarations and implementations are all top level statements, we can enforce the restriction that overloading only happens at the top level of Blossom programs. Blossom also requires that the types of behavior implementations do not overlap. This means that if we write an implementation for Eq like so:

```
1       [Int] is Eq because ...
2       [a] is Eq because ...
```

a compiler error will be thrown because there will be no way for Blossom to decide which implementation to use when you write:

```
1       if [1] = [2,3]
```

5.3.2 Cases and Patterns

Most papers discussing type checking in HM systems do not explicitly support case expressions in their respective abstract languages. This is because a case statement acts exactly like function application for the purposes of type checking.

5.3.3 Matching

Each pattern p describes a match function m_p that takes a value of the same type as the pattern and returns a boolean. For variable and wild card patterns, this match function always returns true. For literal patterns it runs the primitive equality function on the value and the literal in the pattern.

For each algebraic type T we define an implicit function c_T which returns the constructor used to create a value of type T. A constructor patterns $C\bar{p}$ matches a value v if $c_T(v) = C$ and the patterns $\bar{p}$ match the subvalues of v.

5.3.4 Binding

For each constructor C we can define a function f_c that takes a value created by that constructor, and returns one of its sub values. For instance, the Cons constructor from the list datatype

```
1       data List<a> where
2           Cons(a,List<a>)
3           Nil
```

generates the naming function f_{head} and f_{tail}.

If a pattern can match a value, we can simulate binding its variables by binding the value to a new variable x and then replacing all of the instances of names that were bound in the matching operations with the $f(x)$ where f is formed by composing the appropriate naming functions. Therefore we can re-write the BLC expression.

$\lambda 2 Prod(a, b).a + b$

as the SSLC expression:

$\lambda x.f_1(x) + f_2(x)$

Because the patterns in BLC lambda expressions are only ever product or variable patterns, and the type checker validates the number of arguments before they are applied to the function, the match operation will never fail and so we can skip straight to the bind operation.

Case Expressions are a little more complex. They have to be implemented as a lambda that binds an expression to x . Then for every branch with pattern p_i and expression e_i it attempts to match e with p_i until it finds a branch that succeeds, and then it replaces the bindings in that branch with the functions described above, evaluates it and returns the result. If none of the branches match, a runtime error is generated.

5.4 Type Judgments

I will now explain the type inference judgements used by Stuckey and Sulzman. Judgements are written $P, C, \Gamma \vdash e : \sigma$. where P is a program theory, C is a constraint and Γ is a type environment and $e : \sigma$ is a binding from a type e to a scheme σ. Judgments are true iff the binding $e : \sigma$ is in the Type environment and any constraints in σ hold for the given constraint and program theory. We use these judgements to create type inference rules written as:

$$\frac{A_1, \ldots, A_n}{A}$$

Which can be read in English as "If the Judgments A_1 to A_n are true then the Judgment A is also true."

5.5 Damas Milner Inference Rules

Hindley Milner Type inference was first described by Roger Hindley in 1969 [7] and later rediscovered by Robin Milner [3]. However it wasn't formally proved to be complete until 1982 when Luis Damas working in conjunction with Robin Milner defined the following inference rules and introduced an algorithm to verify them [3].

5.5.1 Constants

The first inference rule simply assigns constants to their types. Since there are no preconditions this is an axiomatic inference.

$$\frac{}{P, C, \Gamma \vdash k\colon \tau}$$

5.5.2 Variables

Inferring the types of variables is not much more complex. Damas and Milner's inference rule simply states that the type of a variable expression is the type to which that variable is bound within the type environment.

$$\frac{x\colon \sigma \in \Gamma}{P, C, \Gamma \vdash x\colon \sigma}$$

5.5.3 Application

Application is only valid when the type of the first expression is an arrow type and the type of the second expression is in the domain of the function. If those conditions are met, then the type of the application is the type of the range of the function.

$$\frac{P, C, \Gamma \vdash e\colon \tau \qquad P, C, \Gamma \vdash e\prime\colon \tau \to \tau\prime}{P, C, \Gamma \vdash ee\prime\colon \tau\prime}$$

5.5.4 Lambda

The inference rule for lambda expressions can be read as "If with respect to a program theory, a constraint, and a type environment where all bindings of the variable x have been replaced with the binding x : τ, the type of the expression e is $\tau\prime$, then the type of the lambda expression $\lambda x.e$ is $\tau \to \tau\prime$"

$$\frac{P, C, \Gamma_x.x\colon \tau \vdash e\colon \tau\prime}{P, C, \Gamma_x \vdash \lambda x.e\colon \tau \to \tau\prime}$$

5.5.5 Let

The inference rule for let expressions states that if the value of an expression e is the type scheme σ when the binding x has been removed from the type environment, and the type of the expression $e\prime$ is τ when the binding $e : \sigma$ is added to the type environment then the type of the expression $\textbf{let}\, x = e \,\textbf{in}\, e$ is τ.

$$\frac{P, C, \Gamma_x \vdash e \colon \sigma \qquad P, C, \Gamma_x.x \colon \sigma \vdash e\prime \colon \tau\prime}{P, C, \Gamma_x \vdash let x = e in e\prime \colon \tau\prime}$$

5.6 Stuckey and Sulzman's judgments

The inference rules described thus far are exactly the same as those originally proposed by Damas and Milner, and are well known to be sound. Stuckey and Sulzman expanded these rules with three novel rules, which handle overloaded and constraints. These rules make use of the function fv that returns the set of free variables of an object.

5.6.1 Annotations

The first new rule handles type annotation expressions, which are necessary to resolve cases where constrained types result in ambiguity. The actual judgements for annotations is pretty simple. All we require is that all of the variables of the annotating type σ are not free in the type environment or any constraints. Blossom ensures that this condition is met by renaming the type vars of each user generated scheme with a unique name.

$$\frac{P, C, \Gamma \vdash e \colon \sigma \qquad fv(\sigma) = \emptyset}{P, C, \Gamma(e \colon \sigma) : \sigma}$$

5.6.2 $\forall I$

The next rule they introduce is the $\forall I$ rule, which handles constraint conjunctions. In English it states that if the expression e has type τ with respect to the constraints C_1 and C_2 then its type with respect to the constraint C_1 is the scheme $\forall \bar{\alpha} C_2. \Rightarrow \tau$ where the variables $\bar{\alpha}$ are not free in either C_1 or Γ.

$$\frac{P, C_1 \wedge C_2, \Gamma e \vdash \tau \qquad \bar{\alpha} \notin fv(\Gamma) \cup fv(C_1)}{P, C_1, \Gamma \vdash e : \forall \bar{\alpha} C_2 \Rightarrow \tau}$$

5.6.3 $\forall E$

They also describe a judgement they call $\forall E$, which simplifies constraints that are entailed by the program theory. In English this rule states that if an expression e is represented with the scheme $\forall \bar{\alpha}.C_2 \Rightarrow \tau$ with respect to the program theory P and the constraint C_1 and the constraint C_1 implies C_2 when the variables $\bar{\alpha}$ replaced by the types $\bar{\tau}$ in any first order formula that can be derived from the program theory when the variables $\bar{\alpha}$ replaced by the types $\bar{\tau}$, then the type of e is τ.

$$\frac{P, C_1, \Gamma \vdash e : \forall \bar{\alpha}.C_2 \Rightarrow \tau \qquad [[P]] \models C_1 \supset [\bar{\tau}/\bar{\alpha}]C_2}{P, C_1, \Gamma \vdash e : \forall \bar{\alpha} C_2 \Rightarrow \tau}$$

5.6.4 Overload

The last new rule is also the most complex. It describes how an identifier f can be overloaded so that it can evaluate to the expression e that has type $\forall\bar{\alpha}.C_f \Rightarrow \tau_f$.

The first axiom stipulates that the type environment Γ contains a binding from the identifier f to the scheme $\forall a.F \Rightarrow a$ where F is the implicit constraint that holds for all of the types that the identifier F may represent. This condition is always met in Blossom because the compiler will reject any overloaded functions that don't specify their types before the type checker runs.

Next we stipulate that the type of the expression e, namely the type scheme $\forall\bar{\alpha}.C_f \Rightarrow \tau_f$, does not have any free variables, which is met by default since Blossom implicitly quantifies all of the variables of user defined type schemes.

The next axiom requires us to define h_{C_f}, which is the set of all the equality constraints of the form $\tau_1 = \tau_2$ in the in the constraint C_f. We will also define ϕ the most general unifier of those types and a Predicate F, which corresponds to the Behavior that the overloaded function describes. We then require that the simplification rule $F\phi\tau_f \iff \phi C_f$ is in the program theory. This axiom captures the notion that constrained behavior implementations generate simplification rules as detailed in section 4.7.

The last axiom stipulates that the type of the program p with respect to the constraint, type environment, and Program theory is τ.

If all of these judgments are correct, then the type of the top level expression overloading the identifier f with the expression e in the program p is τ. Notice that this roughly equates to the inference judgement for let expressions with some extra axioms to ensure that overloading functions fits nicely into a constraint system.

$$\frac{\begin{array}{c}(f : \forall a.F \Rightarrow a) \in \Gamma \qquad fv(\forall\bar{\alpha}.C_f \Rightarrow \tau_f) = \emptyset \\ P, C, \Gamma \vdash e : \forall\bar{\alpha}.C_f \Rightarrow \tau_f \\ F\phi\tau_f \iff \phi C_f \in P \text{ where } \phi = \text{mgu of } h_{C_f} \\ P, C, \Gamma \vdash p : \tau\end{array}}{P, C, \Gamma \vdash \mathit{overloadf} :: (\forall\bar{\alpha}.C_f \Rightarrow \tau_f) = e\,\mathbf{in}\,p : \tau}$$

5.7 Confluence

Now I will prove that the Constraint Handling Rules generated by Blossom are confluent and terminating, which is sufficient to prove that Blossom's type system is sound, and decidable.

This proof will make use of the concept of critical pairs, which are two stores S_1 and S_2 that can both be derived from a common ancestor store S_a. As Abdennadher demonstrated, we only need to prove confluence for minimal critical pairs that are formed when S_a only contains the constraints that appear in the heads of the rules that generate S_1 and S_2 [1].

5.7.1 Constraint ordering

Before we deal with the CHRs themselves, we must first discuss the underlying structure of Constraints in a Blossom Program. It is useful to define an order among constraints written as $C_1 \preceq C_2$. Which means that if C_2 holds then C_1 must also hold. This ordering arises from qualified behavior declarations. For example, the declaration of the Ord behavior is written as follows:

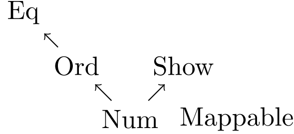
```
1  Given Eq(a). a is Ord when
2      leq(a,a) -> Bool
```

This introduces the order $Eq \preceq Ord$. Blossom forbids the use of Behaviors in qualifiers before they have been defined, which means that the ordering among constraints can be physically seen in the ordering of their declarations in the program. Because of this fact we know that if $C_1 \preceq C_2$ and $C_2 \preceq C_1$ then $C_1 = C_2$ and there are no nontrivial identities among constraints. Blossom also requires types that implement a behavior to implement all of the behaviors that precede that behavior. This means that our ordering is transitive so if $C_1 \preceq C_2$ and $C_2 \preceq C_3$ then $C_1 \preceq C_3$.

This means that we can construct a forest of constraints where the parents of each node precede their children. For instance, the following set of behavior declarations:

```
1  a is Eq when ...
2
3  a is Show when ...
4
5  Given Eq(a). a is Ord when ...
6
7  Given Show(a), Eq(a). a is Num when ...
8
9  m is Mappable when ...
```

Can be modeled as the forest:



As discussed in Chapter 4, propagation CHRs are generated by qualified behavior declarations. Since multiple declarations of the same behavior are forbidden, we know that propagation rules cannot overlap with other propagation rules. They are also single headed since blossom does not allow you to declare more than one behavior in a single declaration.

Furthermore because we require all types that implement a behavior to implement the behaviors that precede it, we know that if there is a propagation rule $c_1 \implies c_2$ in the program theory, and a simplification rule $c_1 \iff c_3$ then there must also be a simplification rule $c_2 \iff c_3$ in our program theory.

5.7.2 Type scheme ordering

Stuckey and Sulzman also define an order among type schemes that they write $F \vdash (\forall \bar{\alpha_1}.C_1 \implies \tau_1) \preceq (\forall \bar{\alpha_2}.C_2 \implies \tau_2)$ where there are no name clashes between $\bar{\alpha_1}$ and $\bar{\alpha_2}$. If with respect to the first order formula F that arises from the program theory, the constraint C_2 implies that there exists a set of variables $\bar{\alpha_1}$ such that the constraint $C_1 \wedge \tau_1 = \tau_2$ holds [18]. More formally:

$F \vdash (\forall \bar{\alpha_1}.C_1 \implies \tau_1) \preceq (\forall \bar{\alpha_2}.C_2 \implies \tau_2)$
$\iff$
$F \models C_2 \supset \exists \bar{\alpha_1}.(C_1 \wedge \tau_1 = \tau_2)$

For instance, the type scheme $[Int]$ precedes the type scheme $\forall a. \implies [a]$ because the variable a can be replaced with the type Int to equate the types.

Blossom requires that the types of overloaded functions do not overlap. This means that it is an error to declare a behavior implementation for both [Int] and [a] because Blossom would have to make a random choice of which function to use when it is applied to a list of Ints. Therefore the following Blossom code results in an error:

```
1 Given Eq(a). [a] is Eq because
2     equals(list1,list2)  ->
3         case (list1,list2) of
4             ([],[]) -> True
5             ([x|xs],[y|ys]) -> x == y and xs == ys
6             _ -> False
7
8 [Int] is Eq because
9     equals(i1,i2) -> True
```

As discussed in section 4.7, simplification rules are generated by behavior implementation. Blossom forbids these behavior implementations from overlapping so we know that there are no overlapping simplification rules in Blossom Program Theories. This means that if the simplification rule $C(\sigma_1) \iff C(\sigma_2)$ exists in the program theory, then $\sigma_1 \preceq \sigma_2$.

Furthermore, since a type may only implement one behavior at a time, we also know that all simplification rules generated by Blossom programs will be single headed.

5.7.3 Putting it together

Propagation rules in Blossom will never propagate a constraint that does not precede the head of the propagation rules, so repeated applications of propagation rules will not generate infinite cycles.

Simplification rules on the other hand only replace constraints with constraints that precede their heads, therefore it is impossible for simplification rules to generate infinite cycles.

It is possible to generate cycles by repeatedly applying propagation and simplification rules, however these cycles are all broken because of the rule that a propagation rule may only be applied to a given store once (see 4.7 for more details about this fact).

Therefore theories are terminating. If a program theory is terminating and all of its critical pairs are joinable then it is confluent. According to Abdennadher the only way for Joinability to be violated is if the application of one rule prohibits the application of another. The only way for this to happen is if one of the two rules is a simplification rule, and the heads of the rules overlap.

Since the heads of simplification rules do not overlap with each other, and the heads of propagation rules do not overlap with each other, critical pairs only arise when the head of a propagation rule overlaps with a simplification rule. An example of two such rules that create a critical pair are the rules:

$Ord(a) \implies Eq(a)$

$Ord(Int) \iff \top$

When applied to the constraint store $Ord(Int)$, we can derive two possible states, $Eq(Int)$ and $\top$. Since Blossom requires all types that implement a behavior to implement its super behaviors,

we know that if a Blossom Program contains the rule $Ord(Int) \iff \top$ then it also contains the rule $Eq(Int) \iff \top$, thereby joining the pair. Therefore Blossom program theories are confluent.

5.8 Ambiguity

Stuckey and Sulzman point out that ambiguity may arise from the use of constrained types. For instance, if the function readNum returns a value that behaves like a number, i.e. either an integer or a float and the function numToString can take anything that behaves like a number and returns a string, then the type of the Blossom expression:

```
1  numToString(readNum("12") + readNum("3"))
```

Cannot be inferred since there is no way for the compiler to decide whether to interpret the strings as floats or integers. Stuckey and Sulzman avoid this problem by treating these cases as type errors and requiring developers to add in type annotations to resolve this issue.

Blossom takes the same approach and treats ambiguity as an error; however, when the ambiguity is caused a subtyping constraint it is possible for the compiler to make a deterministic choice by always picking the type of the superclass. For instance, in the following code Dog is a superclass, and parseDog is a function that takes a string and returns a value of a type that inherits from dog. This means that the following code is ambiguous because there is no way for an algorithm to infer a concrete type for the variable dog:

```
1  fun getDogOwnerFromString(str : String)
2      dog = parseDog(str)
3      dog.owner
```

Blossom doesn't treat this an error, instead it assumes that the type of dog is Dog, the type that all other dogs inherit. If every dog can be parsed as the super type Dog then this doesn't cause any serious problems for developers. However it is very easy to imagine a case where this condition is not true. It is hoped that these cases will be rare but more work needs to be done to ensure that this problem will not make subtyping unusable. A similar method known as defaulting is used in Haskell although it is only used for resolving the types of numeric literals [8].

Chapter 6

Building Blossom

This chapter will describe a Blossom interpretor that is meant to serve as the basis for a fully functional compiler. I wrote this interpretor in Haskell and the source code for it can be found online at github.com/NaOHman/Blossom.

This interpretor does not implement all of the features of Blossom. Rather it is meant as a proof of concept that shows how the theory behind Blossom's type system can be put in practice. In particular, it does not support message passing or error handling, because neither of those functionalities have much impact on Blossom's type system.

The interpretor is composed of 4 main parts. The first part is the parser, which takes a source file and turns it into a set of objects that closely match the syntax of Blossom. The parser also catches some syntax errors such as mismatched parenthesis and indentation errors. If none of these errors are found, it passes the model of the program to the preprocessor. The preprocessor catches more syntax errors, such as duplicate data type declarations. It also preforms a number of transformations on the model produced by the parser to accommodate overloaded functions, inheritance, etc. It also constructs an expression tree that is used during type checking and evaluation stages.

This expression tree, as well as some additional information about the types declared by the user, is passed to the type checker whose job is to scan the program for type errors. Because of the way that we have designed Blossom's syntax, the type checker also catches errors such as using a variables before they are defined, and functions that are called with the wrong number of arguments. The type checker also tags each expression in the tree with its type so that overloaded functions can be properly applied. Finally if no type errors are found, this expression tree is passed to the evaluator that actually runs the program.

6.1 Parsing

The first stage of the interpretor is the parser, which generates lists of data declarations, behavior declarations, behavior implementations, and global bindings. Bindings are 3-tuples consisting of an identifier, an expression, and type scheme. Both function declaration and variable assignment at the top level of a program generate bindings.

The parser uses MegaParsec, which is a monadic parsing library written in Haskell. This library takes care of many of the trickier aspects of parsing for us, such as order of operations, comments,

and indentation sensitive syntax. Monadic parsers work by combining small parsers for syntactic units, called lexemes, to create larger parsers. For instance, we can parse a function call by combining a parser that represents a lowercase name with a parser that recognizes a list of expressions wrapped in parenthesis.

Each of these lexemes is modeled by a Haskell datatype. Since Blossom is heavily influenced by Haskell, these datatypes should not seem to foreign. For instance, Blossom patterns are modeled by the following Haskell datatype:

```
data Pat = PCons Id [Pat]
         | PAs   Id Pat
         | PLit  Literal
         | PVar  Id
         | PWild
```

One of the other advantages of using a language like Haskell is that these datatypes are almost identical to BLC's grammar. For example, the following is the BLC grammar for patterns:

Pattern	p	::=	$P\bar{p}$	Constructor
		|	$x\#p$	As pattern
		|	k	Literal
		|	x	variable
		|	_	wild card pattern

We can write the parser that recognizes a pattern as follows:

```
pattern =  try (PCons <$> capsWord <*> csl pattern)
       <|> try (PAs   <$> (lowerWord <* symbol '#') <*> pattern)
       <|> try (PLit  <$> literal)
       <|> try (PVar  <$> varName)
       <|> try (symbol ``_" >> return PWild)
```

The syntax may seem alien to those unfamiliar to Haskell, but it is my hope that you can see the similarities between the datatype, the monadic parser and Blossom's grammar.

As stated above, these monadic parsers are combined to create larger parsers. The largest parser is called the Blossom parser. It is composed of four smaller parsers that each parse one of the top level syntactic structures, namely Data Declarations, Behavior Declarations, Behavior Implementations, and Bindings. However these structures are not the same structures that Blossom uses for type checking and evaluation. Furthermore there are several syntax errors that are not checked by the parser because they are impossible or very difficult to detect before parsing is complete. In order to check these syntax errors, and to create a more usable model of Blossom, we pass the information gathered by the parser to the preprocessor.

6.2 Preprocessor

The preprocessor has two main jobs. It validates the parts of the Blossom language specification that cannot be checked by the parser and it transforms the parser's model into a list of assumptions, a class environment, and a syntax tree. Assumptions are written $Identifier : Type$ and are mappings from identifiers to type schemes. They are generated by global bindings and constructor definitions. We call them assumptions because the we don't necessarily know that the type scheme of a given

identifier is actually the same as the expression it is bound to. The class environment is a mapping from the names of behaviors to the list of types that implement them. It also contains mappings from behaviors to their respective super-behaviors, the behaviors that any type that implements that behavior must also implement. The class environment is roughly equivalent to Stuckey and Sulzman's concept of a program theory.

6.2.1 Datatypes

We need to create a list of assumptions for each datatype that represent the type schemes of its constructors. For instance, suppose we have the datatype List:

```
1 data List<a> where
2     Cons(a, List<a>)
3     Nil
```

We know that both of the constructors are functions that return lists, and we know the types of the arguments because they are specified by the constructors. Therefore we can deduce the assumption $Cons :< a, List < a >> - > List < a >$. However the type variable a in this type scheme is free, which means that it refers to a single specific, but unknown type. We need to bind it to the universal quantifier to represent the fact that we can have lists of any type. Therefore the actual assumption that is generated by the Cons constructor is $Cons : \forall a. < a, List < a >> - > List < a >$.

You might think, based on the previous section, that Blossom infers the assumption $Nil : \forall a. <> - > List < a >$ for the Nil constructor. However, that's not quite correct. Because the Nil constructor takes no arguments, and does not trigger any additional computations, it makes more sense for us to treat it as a constant value than it does to treat it as a function with no arguments. Therefore Blossom actually generates the assumption $Nil : \forall a. List < a >$.

We also check the list of constructor and type names for clashes. If any are found we reject the program. In theory we could allow users to overload constructors using the same technique that we use to overload functions but we believe that doing so would dramatically increase the chances for ambiguity to occur. Next we create an implicit function for every constructor that preforms the actual work of constructing a new value out of other values. We then bind these functions to the names of the constructors and add these bindings to the list of bindings produced by the parser.

The preprocessor also turns record fields into functions. For instance, the Corgi datatype:

```
1     data Corgi where
2         .name    : String
3         .owner   : String
4         .royalty : Bool
```

gives rise to 3 implicit functions,

```
1 fun _name(corgi : Corgi) : String -> ...
2 fun _owner(corgi : Corgi) : String -> ...
3 fun _Royalty(corgi : Corgi) : Bool -> ...
```

we add an underscore to the beginning of each field to differentiate fields from regular functions. Each of these functions is then turned into a binding from the modified name of the field, to a special

built-in function that knows how to access the value that the field function points to. We don't add any assumptions to the list of assumptions just yet since we still need to determine whether these field functions have been overloaded. Instead we add these bindings to the list of bindings generated by the parser since we detect overloaded functions while processing those bindings.

6.2.2 Behavior Generation

Next the preprocessor checks to see if any record types have been inherited. Each inherited type is promoted to a behavior using the method described in section 4.1 This generates a new behavior declaration and a list of behavior implementations which are added to their respective lists to be processed later. If an algebraic type is inherited, then the preprocessor rejects the program.

Then the preprocessor checks the list of bindings for duplicate names. If any are found, they are promoted to implicit behavior declarations and added to the list of behavior declarations waiting to be processed (for a detailed description of this method, see section 4.1). The overloaded functions are then removed from the list of bindings. Note that because we process data types first, the field functions may also be overloaded by the preprocessor during this step.

At this point all of the behaviors have been generated. The preprocessor then checks to make sure that no two behaviors define the same function and that there are no bindings with the same name as a function defined by a behavior. Then it generates an assumption for each binding left in the list.

The preprocessor also begins constructing the class environment by adding the qualifiers of each behavior declaration to that behavior's list of super-behaviors. It also checks to make sure that the ordering of behaviors described in section 5.5 is not violated. Since behaviors are processed in the same order that they are parsed, it can do this by simply checking to make sure that all of the super behaviors of a behavior have already been added to the class environment.

6.2.3 Declaration Validation

The preprocessor then checks to make sure that the types of behavior declarations do not overlap and that each implementation provides a definition for every stub specified by the behavior. The preprocessor also adds the types to each behavior's list of implementations in the class environment. Finally it makes one last pass through the class environment to make sure that each type that implements a behavior also implements that behavior's super behaviors.

Once this process is complete, the preprocessor generates special overloaded bindings for each of the functions declared by a behavior. Rather than binding an identifier to an expression, these bindings bind identifiers to mappings from types to expression.

If the preprocessor has not rejected the program, then we can be sure that it is syntactically correct and that our list of assumptions and the class environment is complete. The only thing left to do is to turn the list of bindings into a single expression tree. The preprocessor does this by identifying the expression bound to the identifier 'main' and then creating a giant let expression that binds each of the global bindings to their expressions in the main expression. (If main takes commandline arguments we add an implicit request call at the top of that expression to collect those arguments).

6.3 Type Checker

This monster let expression, the class environment, and the list of assumptions are then passed to the type checker, which examines the expression tree and either rejects it as ill-typed or passes it to the evaluator. My type checker is based off of the Haskell type checker written by Mark Jones [8] which is in turn based on the algorithm W which was described by Damas and Milner [3]. I chose to base my type checker off of Jones's since Haskell uses a variant of constrained HM types that supports pattern matching and is quite closely related to Blossom's type system.

Throughout this section I will use a high level pseudocode to explain the type inference algorithm. This pseudocode should be fairly easy to read however it is important to mention we will use pattern matching syntax to help break the code into more manageable chunks. For instance, the psudocode:

```
foo(arg, 1) do
    print(''Bar")
foo(arg, x) do
    print(''Baz")
```

Will print "Bar" when the second argument evaluates to 1 and "Baz" when it evaluates to any other value.

6.3.1 Substitutions

This type checker works by traversing the tree and generating new assumptions about the types of expressions. It then attempts to build a substitution that unifies all of the type variables in the list of assumptions. If no such substitution exists then the program is rejected because it is ill-typed. The algorithm also keeps track of a list of constraints that are verified once the main inference algorithm is complete.

As described earlier, a substitution is a mapping from a type variable to a type scheme. The type checker attempts to find the most general unifier, or MGU of the set of types in a program. The MGU is the the substitution θ such that for any other substitution s there exists a third substitution $s\prime$ such that the result of applying s to any type τ is equal to applying θ and then $s\prime$ to τ in other words $s(\tau) = s\prime(\theta(\tau))$.

The process of finding this substitution is defined by the unify operation which takes two types and modifies the global substitution or fails because no such unifier exists. It uses a Robinson's algorithm [7] to preform this task. Note that the following code and all code throughout this chapter will make use of BLC syntax.

```
unify(type1, type2) do
    sub = mgu(type1, type2)
    extend the global substitution with sub

mgu(τ1 → τ2, τ3 → τ4) do
    sub1 = mgu(τ1, τ3)
    sub2 = mgu(τ2, τ4)
    return (sub2 ∘ sub1)
```

```
mgu(α, τ) do
    return α ↦ τ

mgu(τ, α) do
    return α ↦ τ

mgu(T₁τ̄₁, T₁τ̄₂) do
    if T₁ == T₂
        return map mgu over τ̄₁ and τ̄₂
    else
        fail "Type constructors do not match"

mgu(a, b) do
    fail "Couldn't unify types a and b"
```

6.3.2 Constrained W Definition

The main type inference algorithm is implemented as a function called tiExpr which takes two arguments, the list of assumptions, and an expression and returns the type of that expression. We also define a function called putConstraints which adds a list of constraints to the global list of constraints, which will be processed after tiExpr completes. Throughout the algorithm, I will shorten type scheme to type.

I will also refer to a function called instantiate that takes a type scheme and replaces all of its variables with fresh type variables that do not appear in any other assumption. I will also define a function called getConstraint which returns the constraints of a given type scheme.

Without further ado I will now give the pseudocode for inferring the types of Blossom expressions. The inference algorithm for constants simply returns that literal which is a priori knowledge.

```
// Constant expression
tiExpr(assumps, k) do
    if k is an Int:
        return Int
    if k is a Float:
        return Float
    etc.
```

The type of a variable expression is simply the type to which that variable bound in the list of assumptions. If there are no assumptions about the variable, then it has not been defined yet and the type checker throws an error.

```
//variable expression
tiExpr(assumps, x) do
    if x in assumps:
        type = lookup(x, assumps)
        new_type = instantiate(type)
        putConstraint(getConstraint(new_type))
        return new_type
    else:
        fail "the variable x has not been defined yet."
```

Type inference for lambda expressions relies on a helper function called tiAlt, which infers a type from a pattern and an expression. It does this by running type inference on the pattern, and the expression, and then creating a type scheme representing a function from the type of the pattern to the type of the expression.

We also define a function tiPat which takes a pattern and returns a list of assumptions generated by binding the variables in the pattern, and the overall type of the pattern.

```
//lambda expression
tiExpr(assumps, λ p. e) do
    return tiAlt(p, e)

//Helper function for case and lambda types
tiAlt(assumps, p, e) do
    p_assumps, p_type = tiPat(p)
    e_type = tiExpr(p_assumps + assumps, e)
    return <p_type> -> e_type

tiPat(assumps, var#pat) do
   as, type = tiPat(assumps, pat2)
   as += [var : type]
   return as, type

tiPat(assumps, x)
   type is a new type variable with kind *
   return [x : type], type

tiPat(assumps, _) do
   type is a new type variable with kind *
   return [], type

tiPat(assumps, k)
   type = tiExpr(assumps, k)
   return [], type
```

```
tiPat(assumps, Constructor(pats))
   type = lookup cstr in assumps
   new_type = instantiate(type)
   putConstraint(getConstraint(new_type))
   new_assumps, p_types = map tiPat over pats
   rt is a new type variable with kind *
   unify(new_type, <p_types>->rt)
   return  new_assumps, rt
```

Application expressions are typed by first checking the types of the function, and the argument, and then making sure that the type of the argument can be unified with the type of the domain of the function.

```
//Application
tiExpr(assumps, e_1(e_2)) do
    fun_type = tiExpr(assumps, e_1)
    arg_type = tiExpr(assumps, e_2)
    rt = new type variable with kind *
    unify(fun_type, <arg_type> -> rt)
    return rt
```

Case expressions are typed like function applications but instead of running type inference on a single function, we run it on all of the branches of the case expression and make sure that those branches all can be unified.

```
//case inference
tiExpr(assumps, case e of es) do
    case_type = tiExpr(assumps, e)
    rt = new type variable with kind *
    for λ p.e in es do
        b_type = tiAlt(p, e)
        unify(b_type, <case_type> -> rt)
    return rt
```

Let expressions are typed by first inferring the types of the bindings in the bindgroup which represents the list of pairs of patterns and expression. This process generates a list of assumptions that represent the type schemes of the variables that appear in the bind group and then uses those assumptions to infer a type for the expression following the *in*. is then used to the type of the expression.

```
//Let expression inference
tiExpr(assumps, let bindgroup in e)
    new_assumps <- tiBindGroup(assumps, bindgroup)
    type <- tiExpr(new_assumps + assumps, e)
    return type
```

I will not define the tiBindGroup function here since it has to jump through a bunch of hoops because we to allow bindings to mutually dependent and to override existing bindings. Basically, it removes any assumptions that already exist for each binding and then assumes that each binding

corresponds to some type variable. Then it works its way through the bindings until types have been inferred for all of them.

Once this process is complete we are left with the global substitution and the global list of constraints. Because the preprocessor has already done the work to make sure that the propagation constraints have been fulfilled, validating the constraints is a simple matter of applying the substitution to all of the types within the constraints, and then looking them up in the Class Environment to make sure that if we have a constraint like $Eq(Int)$ then the type Int is in the list of Eq's implementing types.

6.4 Evaluation

At this point a fully formed Blossom implementation would compile an executable binary. However since writing a compiler is beyond the scope of this project, I will instead demonstrate how Blossom can be evaluated by an interpretor.

We begin evaluation with a scope which is just a mapping between identifiers and the expressions to which they correspond. The only bindings within the scope at the beginning of evaluation are built-in primitive operations such as addition (my interpretor also implements print and getLine as primitives since it does not include message passing support yet).

The evaluation begins with the let expression that describes the program and continues until it encounters a runtime error, or an expression which cannot be reduced any further. Blossom programs may also continue to evaluate indefinitely. This process is implemented by the *eval* function that takes an expression and a scope and returns a value.

Values are either primitive constants such as Ints and Chars, Lambda expressions, or constructed values which are defined as a stucture consisting of a string that represents the constructor and a list of values that represent the arguments of the constructor. We also define a special kind of value to represent overloaded identifiers. These overloaded values are mappings from types to other values.

The simplest expressions to evaluate are constants, all you do is return the value of the constant.

```
eval(scope, k) do
    return k
```

Lambdas are also base values and so they are returned as is.

```
eval(scope, lambda) do
    return lambda
```

Variables are only slightly more complex, all you have to do is look up the value of the literal in the scope. We know that this operation will always succeed because the type checker rejects programs with references to undefined variables.

```
eval(scope, var) do
    val = lookup var in scope
    return val
```

Let expressions make use of two special functions, match and bind. Match takes a pattern and a value and returns true if the value matches the pattern. Bind also takes pattern and a value but it returns a list of the name value pairs that represent a successful match. After these new bindings are calculated, they are added to the scope and the expression is evaluated.

```
eval(scope, let bindgroup in e) do
    bindings = []
    for pat,expr in bindgroup:
        val =  eval(scope, expr)
        if val does not match pat:
            fail "Could not bind expression to variable"
        else
            bindings += bind(pat, val)
    return eval(scope + bindings, e)
```

Case expressions are quite similar. They evaluate the expression at the top of the case expression, and then attempt to match that value against each of the patterns in the branches until one succeeds. At that point the bindings that arise from the pattern are added to the scope and the expression is evaluated. If no matches are found, an error is generated.

```
eval(scope, case e of branches) do
    val = eval(scope, e)
    for pat,expr in branches:
        if val matches pat:
            bindings = bind(pat, val)
            return eval(scope + bindings, expr)
    fail "Pattern match failed in a case expression"
```

Application expressions are the most complicated expressions to evaluate. First the argument of the function is evaluated, then the function itself is evaluated. If the function is a lambda value, then we attempt to match the argument to the pattern of the lambda and evaluate the expression. If the function is an overloaded value, we have to disambiguate it by using the type of the argument to select the appropriate function.

If the function is a built-in function like addition, then it is handled by a special execBuiltin function which executes all primitive operations. Finally if the function is none of the above, then the function must have evaluated to either a constructed value or a literal which is an error that should have been caught by the type checker, but we will make it explicit here.

```
eval(scope, f(e)) do
    arg = eval(scope, e)
    fn = eval(f)
    if fn is a lambda value written λ p.e:
        if arg matches p:
            bindings = bind(pat, arg)
            return eval(scope + bindings, e)
        fail "Function could not be applied to the given arguments"
    if fn is an overloaded expression:
        fn2 = disambiguate(fn, arg.type)
        return eval(fn2, arg)
    if fn is a built-in operator
        return execBuiltin(fn, arg)
    fail "you can't call something that isn't a function"
```

Although this interpretor is capable of running a large subset of Blossom programs it has a number of shortcomings. To begin with is makes no attempt at memory management or garbage collecting and passes every object by value not by reference. We can get away with these tricks because we wrote the interpretor in Haskell which supports automatic garbage collection and passes all objects by name. However, since we keep track of all bindings in the scope, Haskell's garbage collector is not very effective. Despite these flaws, this interpretor proves that Blossom can be used to create working programs and can be used as a jumping off point to create a fully fledged Blossom compiler.

Chapter 7

Future Work

Although I made a lot of progress towards implementing Blossom, much work remains to be done before it can be released to the public.

7.1 Language Features

7.1.1 Syntax

Blossom's syntax is still very fluid at this point and may be modified before the first public release. For instance, I would like to be able to remove Blossom's reliance on capital letters to distinguish types from type variables so that Blossom can be translated into languages that do not have capital letters. In addition, I am also considering changing the syntax of behavior declarations and implementations since I believe it is too easy to mistake one for the other.

7.1.2 Partial Application

I would also like to implement partially applied functions and partially applied type constructors since I consider both of these features essential to any functional programming language. For instance, because the Dictionaries are described by a binary type constructor $Dict < k, v >$ they can't implement the Mappable behavior since Blossom expects all Mappable types to be unary type constructor of the form $m < a >$.

As Blossom is currently implemented, You can mimic partially applied functions by defining lambdas that capture values from their scopes, for instance:

```
1    a = 12
2    partialFunc = (x) -> x + a
3    map([1,2,3], partialFunc) ==  [13,14,15]
```

However, it would be ideal if you could partially apply functions in a more concise manner.

7.1.3 Default arguments

I would also like to be able to support default values for function arguments similar to keyword arguments in python. These would be especially useful for defining record types that automatically fill in the fields of their supertypes. For instance:

```
1 data Corgi inherits Dog where
2     .name : String
3     .owner = "Queen Elizabeth II" : String
4     .royalty = True : String
5
6 pookie = Corgi("Pookie")
7 porkPie = Corgi("Pork Pie", owner="John", royalty=False)
```

However as of now, I have not been able to find a way to support default values without crippling the ability to overload functions.

7.1.4 Message Passing Implementations

Another feature that needs to be implemented for Blossom to become a fully functioning language is user designed message passing implementations. One way that this could be accomplished would be to allow users to provide a separate file along with any Blossom program that specifies how the different handlers interact. For instance, you could specify that messages from the main program to a specific handler are always carried out on a newly created pthread or on a different machine connected via ssh. By default all handlers would be implemented on a separate green thread on the same processor. However this method seems very rigid, since it does not allow the actual program to effect how the messages are handled and I would like to do more research before I commit Blossom to that approach.

7.1.5 Modules

Another obvious omission for a application programming language is the lack of a module system. Blossom currently does not define a module system because they make compiling and type checking much more difficult. In particular I would like to allow users to define codependent modules, however doing so would make it much more difficult to ensure that behaviors do not generate non-terminating Constraint Handling Rules since we would not be able to guarantee that behaviors are only qualified by other behaviors that appear above them in a file.

7.1.6 Subtyping

Although Blossom provides a working implementation for subtyping, it gives rise to some tricky situations. For instance, suppose you want to write a function that takes a dog and a person, and changes that dog's owner to that person. You might write a function adopt as:

```
1 fun adopt(dog : Dog, owner : String) : Dog ->
2     Dog(dog.name, owner)
```

However, Blossom translates this function into the qualified function:

```
1 Given _Dog(a)
2 fun adopt(dog : a, owner : String) : Dog ->
3     Dog(dog.name, owner)
```

Because we use the constructor Dog, which always returns an object with the type Dog. If we define a type Corgi that inherits dog, the following code will produce a type error since the types Dog and Corgi are not equal.

```
1 charles = Corgi("Charles", "", False)
2 @charles = adopt(charles, "Jim")
```

One possible solution would be to allow the rebind operator to rebind fields in a function like so:

```
1 Given _Dog(a)
2 fun adopt(dog : a, owner : String) : Dog ->
3     @dog.name =  owner
```

However this approach relies on the developers to use the rebind operator and not the constructor. Until a better solution can be found, we will simply rely on developers to overload such functions to achieve the desired behavior:

```
1 fun adopt(dog : Corgi, owner : String) : Corgi ->
2     Corgi(dog.name, owner, dog.royalty)
```

7.1.7 Success Typing

Blossom's syntax was heavily influenced by Lindahl and Sagonas's success typing system. At the outset, I had planned on allowing case expressions to be ill formed, which would allow different branches to return different types. The strict Blossom type checker would reject these programs but they still might pass the success type checker.

The goal was to allow developers to prototype code with as few restrictions as possible. Once they were ready to start revising their code, they could use the Blossom type checker and the success type checker in concert to identify the sections of code that need to be cleaned up. Furthermore because message handlers reject all ill-typed messages, Blossom could allow applications to deploy a mix of success typed code and strictly typed code, while minimizing the number of dynamic checks needed.

However as it became apparent that it would be very difficult to implement both a success type checker and the Blossom type checker in the time frame of this project. So I chose to focus my efforts on the Blossom type checker since it would be more useful to developers.

7.2 Implementation

7.2.1 Message passing

The most glaring omissions in my Blossom interpretor is its lack of true message passing support. At the moment, a few choice message handlers are treated as calls to built in functions. Future implementations of Blossom will allow for true message passing support that could be implemented by using green threads, system threads, and third party libraries such as openSSL. However as

mentioned above, since I still haven't finalized how exactly Blossom will implement this functionality, it could not be included in the interpretor.

7.2.2 Compiler

Blossom is also designed to be compiled rather than interpreted. This would make Blossom programs execute faster and would allow us to preform a number of optimizations on the syntax tree such as tail recursion. I also have not decided whether to attempt to compile Blossom to assembly code or whether my compiler should target the JVM. Compilation to the JVM would result in slower programs, but would make it easier to implement cross platform libraries and would also allow developers to write android applications in Blossom.

7.2.3 Bootstrapping

Haskell was a great choice for implementing the parser and type checker because of its support for pattern matching and algebraic datatypes, which make it very easy to model BLC in a natural manner. However, Blossom also supports these features and I would like to one day be able to compile Blossom with Blossom.

7.3 Beyond

7.3.1 Libraries

Once Blossom's syntax is stabilized it will need to provide number of libraries before it can really be used to develop applications. In particular we will need to provide libraries for basic data types like sets and dictionaries as well as libraries that handle network calls, file operations, os calls, foreign function calls, and more.

7.3.2 Tools

We can also imagine creating a number of useful tools which would help Blossom developers write better code. For instance, we could create a tool that automatically converts overloaded functions to behaviors. We can also imaging a code linting tool similar to jslint, hlint, etc. that preforms basic syntax checks to make sure we aren't using unnecessary parenthesis or an overly complicated construct when sugar for it exists. Another exciting tool would be a clone of Haskell's hoogle, which allows developers to search libraries for functions based on their types. This is an incredibly useful tool in a functional language and I can personally attest to the fact that it has saved me many hours of reimplementing existing functions.

Furthermore, we know that the only code in Blossom that generates side effects relies on message passing. We can use this fact to make all sorts of control flow analysis tools. For instance, we could create a purity analyzer that determines whether expressions have side effects. Developers could use this tool in conjunction with test suites to prove that their code is error free with more certainty than is possible in other languages like C and Java. We can also imagine an IDE that performs real-time purity checks and highlights all impure expressions. This gentle marker of side effects would help developers write better, purer code.

We can also use the fact that all user input funnels through the request function to create a security analyzer. This tool would be able to use message passing to determine which parts of the code can be effected by a given input source. For instance, it could warn you if you are making an SQL call with a value that came from a network call so that you can make sure that it isn't a malicious string that will drop all of your tables.

7.4 Conclusions

Although it is far from complete, Blossom represents an exciting new addition to the family of functional languages. It combines the power and safety of Haskell and ML with the ease and flexibility of Erlang. The type information provided by the inference algorithm will help large teams of developers work together since it will serve as an automatic source of documentation. Furthermore Blossom's message passing framework will encourage developers to write loosely coupled code that can easily grow from a single threaded application running on one machine to a massively parallel application running on thousands of machines distributed around the world.

Finally Blossom's familiar and easy to understand syntax will help introduce functional programming techniques and Hindley Milner type systems to developers who have been scared off by the strange and convoluted syntax of other languages like Haskell and ML.

Personally I plan on continuing to develop Blossom in my spare time. With luck, it will be ready for a public release in the near future and I hope that I will be able to create a community of developers who also see the benefits of a growth-oriented language to assist me in turning Blossom into a popular and well supported language.

Chapter 8

Notes

8.1 Why Blossom is Strict

Traditional HM systems treat functions that take multiple arguments as lambdas that return lambdas. For example, a function $multiply :< Int, Int > - > Int$ could be expressed as $\lambda x.\lambda y.x * y$.

This allows users to curry functions, that is to pass partially applied functions as values to other functions. However this means that any side effects will be evaluated lazily. For instance, if you write the functions foo and bar:

```
1  fun foo(a, b) -> a
2
3  fun bar(b) ->
4      print(b)
5      b
```

you would expect the expression foo(1, bar(2)) to print out the value 2 and then return 1. However in lazy languages like Haskell, the value of the second argument is not evaluated until it is used in an expression. Because b is not used, the side effect (printing 2) of bar is never triggered. Instead the call foo(1,bar(2)) would result in the following evaluation sequence:

- foo is applied to the expression 1
- the variable a is replaced with 1 in the expression a
- foo(1) is applied the expression bar(2)
- the variable b is replaced with the expression bar(2) in the expression 1
- the expression 1 is returned

We believe that it's much easier for developers to debug their programs if they can establish a clear timeline that precedes the error. Therefore Blossom requires functions to be strictly evaluated, which means that the values of the arguments are computed before they are passed to the function. The Blossom expression foo(1,bar(2)) is actually evaluated in the following order:

- bar is applied to the expression 2
- the variable b is replaced with the number 2
- the value 2 is printed
- the value 2 is returned
- foo is applied to 1 and 2
- the values 1 and 2 are bound to the arguments a and b
- the value a matches the pattern on line 3
- the value 1 is returned

Although this is means Blossom may do more work, it simplifies debugging enormously if expressions are evaluated strictly and you can simulate all the benefits of lazy computation by using message passing.

The implicit product constructors with which Blossom wraps function arguments can be used to enforce this strict evaluation model. Since we know that values will be wrapped in the product constructor before they are passed to the function, we can have the compiler force a complete evaluation of any expression that is passed to a product constructor.

Bibliography

[1] Slim Abdennadher. "Operational semantics and confluence of constraint propagation rules". In: *Principles and Practice of Constraint Programming-CP97*. Springer Berlin Heidelberg, 1997, pp. 252–266.

[2] Edwin Brady and Kevin Hammond. "A verified staged interpreter is a verified compiler". In: *Proceedings of the 5th international conference on Generative programming and component engineering*. ACM. 2006, pp. 111–120.

[3] Luis Damas and Robin Milner. "Principal type-schemes for functional programs". In: *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1982, pp. 207–212.

[4] Richard A Eisenberg and Jan Stolarek. "Promoting functions to type families in Haskell". In: *ACM SIGPLAN Notices* 49.12 (2015), pp. 95–106.

[5] Karl-Filip Faxen. "Haskell and principal types". In: *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*. ACM. 2003, pp. 88–97.

[6] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. "Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types". In: *Journal of the ACM (JACM)* 55.4 (2008), p. 19.

[7] Roger Hindley. "The principal type-scheme of an object in combinatory logic". In: *Transactions of the american mathematical society* 146 (1969), pp. 29–60.

[8] Mark P Jones. "Typing haskell in haskell". In: *Haskell workshop*. Vol. 43. 1999, pp. 45–59.

[9] Tobias Lindahl and Konstantinos Sagonas. "Practical type inference based on success typings". In: *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*. ACM. 2006, pp. 167–178.

[10] Andres Löh and Ralf Hinze. "Open data types and open functions". In: *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*. ACM. 2006, pp. 133–144.

[11] Conor McBride. "Faking it Simulating dependent types in Haskell". In: *Journal of functional programming* 12.4-5 (2002), pp. 375–392.

[12] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. "Hoare type theory, polymorphism and separation". In: *Journal of Functional Programming* 18.5-6 (2008), pp. 865–911.

[13] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[14] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. “The ins and outs of gradual type inference”. In: *ACM SIGPLAN Notices*. Vol. 47. 1. ACM. 2012, pp. 481–494.

[15] Konstantinos Sagonas, Josep Silva, and Salvador Tamarit. “Precise explanation of success typing errors”. In: *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation*. ACM. 2013, pp. 33–42.

[16] Tom Schrijvers et al. “Type checking with open type functions”. In: *ACM Sigplan Notices* 43.9 (2008), pp. 51–62.

[17] Jon Snyder. *Facebook Introduces Hack the Programming Language of the Future*. Mar. 2014. URL: http://www.wired.com/2014/03/facebook-hack/.

[18] Peter J Stuckey and Martin Sulzmann. “A theory of overloading”. In: *Acm transactions on programming languages and systems (toplas)* 27.6 (2005), pp. 1216–1269.

[19] Asumu Takikawa et al. “Is sound gradual typing dead?” In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM. 2016, pp. 456–468.

[20] Philip Wadler and Stephen Blott. “How to make ad-hoc polymorphism less ad hoc”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1989, pp. 60–76.