UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ARTHUR GIESEL VEDANA

The V Language

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr. Rodrigo Machado
Coadvisor: Prof. Dr. Álvaro Freitas Moreira

Porto Alegre
December 2018

ABSTRACT

Functional programming languages are known for their ease of use and conciseness. Pure functional languages are a subset of these languages and provide further improvements in certain areas, such as making it easier to reason about programs due to referential transparency. One area in which pure functional languages usually lack, however, is when manipulating records.

We present the V language, a purely functional programming language with a novel approach to record manipulation. By using polymorphic accessors, V attempts to solve issues commonly found when manipulating records in purely functional programming languages. This approach required the use of traits to support ad-hoc polymorphism in the language.

This work presents a complete picture of the V language. A formal specification is given, with syntax, semantics and a type-inference system. The current implementation, comprising of an interpreter with an interactive read-eval-print loop (REPL), is shown, along with a brief analysis of some preliminary tests.

Keywords: Functional Programming Languages. records. traits.

A Linguagem V

RESUMO

Linguagens de programação funcionais são conhecidas por sua facilidade de uso e concisão. As linguagens funcionais puras são um subconjunto dessas linguagens e fornecem melhorias adicionais em determinadas áreas, como facilitar raciocínio sobre programas devido à transparência referencial. Uma área na qual linguagens funcionais puras geralmente pecam, no entanto, é a manipulação de registros.

Apresentamos a linguagem V, uma linguagem de programação puramente funcional com uma nova abordagem a registros. Usando acessores polimórficos, V tenta resolver problemas comumente encontrados quando manipulando registros em linguagens de programação puramente funcionais. Essa abordagem exigiu o uso de traits para prover polimorfismo ad-hoc para a linguagem.

Este trabalho apresenta uma visão completa da linguagem V. Uma especificação formal é dada, com sintaxe, semântica e um sistema de inferência de tipos. A implementação atual, composta de um interpretador com um read-eval-print loop (REPL), é mostrada, juntamente com uma breve análise de alguns testes preliminares.

Palavras-chave: linguagens de programação funcionais, registros, traits.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Functional programming languages, or simply functional languages, are languages in which programs are primarily written by means of the definition and combination of functions. Haskell, Ocaml, Scheme, F#, Elm, and Scala are examples of functional languages (or mostly functional in case of OCaml, Scheme, F# and Scala). Using features such as higher-order functions and currying, functional languages can (generally) express complex operations in fewer lines of code than imperative or object-oriented languages. Also, a strong type system, featured in many functional languages, allows the development of code that is free from some kinds of errors, such as missing cases in switch statements or trying to perform an operation on a type that does not support that operation.

Records (or structs) are one of the basic data structures in computer science. They are one of the most common way to represent structured and heterogeneous data, since having named fields allow easy understanding and manipulation of each component of the record. In most languages, access and manipulation of a single field in a record is easy and straightforward, and complex data structures can be expressed using nested records without becoming too unwieldy or cumbersome.

Purely functional languages (SABRY, 1998) are languages in which referential transparency holds, meaning that the result of a function application is solely defined by its arguments. There are few languages with this property, from which we can mention Haskell (MARLOW et al., 2010) and Elm (CZAPLICKI; CHONG, 2013). A consequence of referential transparency is the absence of implicit side effects during function evaluation, which means, among other things, that values stored in memory be cannot modified. Because of this, purely functional languages are known for not being as convenient for dealing with records as languages that allow side effects. This limitation is so serious that, at least in Haskell, the use of a library to work with large nested records is practically required. Although many different approaches have been suggested to mitigate this problem, they are usually limited. For instance, the popular Lenses (KMETT, 2012) approach for Haskell uses very complicated types (since it exploits a specific type isomorphism) and requires template meta-programming for its notation. These problems come from the fact that lenses are implemented as a library and not as a primitive construct of the language.

This paper presents V, a purely functional programming language inspired by

Haskell, with a new approach to record manipulation. V introduces first-class accessors that *focus* on a region of a given record. These accessors are used together with *getter* and *setter* functions to read and modify records. It is important to clarify that, when a *modification* or *update* of a record is mentioned in a purely functional language, what is meant is the construction of a new record based on an existing one. Accessors in V can be combined using primitive operations, allowing to focus on very specific parts of large, compound nested records. Accessors are polymorphic, allowing the access and update of a record whenever the necessary fields are present – which is verified by a trait-based type system.

Although the V language is still under development, its specification and implementation evolve in parallel with each other and are kept as consistent as possible whenever changes occur. The specification consists of a big-step operational semantics and a type system, and the implementation consists of a working REPL (read-eval-print loop) interpreter and an improvised library system (with a simple standard library), implemented in F# (SYME et al., 2010). Both specification and implementation can be found on the project page on Github[1]. This work presents the current status of the language and, in particular, the record subsystem and how traits are used in the type system.

1.1 Motivation

New programming languages are constantly being created, and the motivations for their existence are as varied as the languages themselves. Some languages, like Ada (TAFT; DUFF, 1997) and Algol (BACKUS et al., 1960), were designed by committees with a top-down approach, fully specifying the language before any code had been written in it. Others, such as Lua (IERUSALIMSCHY; FIGUEIREDO; FILHO, 1996) or Python (ROSSUM; JR, 1995), were first created to satisfy a small and specific need, and then grew in popularity from there. Some languages are created as a way to study programming languages academically, while others are created with the intent of being used in production. Some languages try to introduce novel concepts or propose a new syntax to simplify the work of programming, while others try to be as efficient as possible.

Out of all these scenarios, V evolved from a simple functional language called L1, introduced in a project in Formal Semantics class, as a way to explore language design, type systems and interpreters. The language started to evolve with new features such as

[1] <https://github.com/AvatarHurden/V>

strings, lists, type inference, polymorphism, etc., always looking for inspiration in other languages and trying to take the best parts of each.

For a while, the language was very similar to other functional languages, and its value existed solely as a learning tool. This started to change when records were added to the language. The author discovered that there is no universally accepted way to add records to functional languages, with each one taking its own approach, and all of them with serious limitations and drawbacks.

Because of this, the focus turned to finding a better way to use records, trying to reconcile the ease of use that exists in imperative or object-oriented languages with the purity of a functional language. This led the author to not only change the core of the language, but also drastically alter its original type system, making it distinct from those normally found in functional programming languages.

The result of this journey is V, a purely functional programming language with record accessors and a trait-based type system. The language is still being worked on as a general language, but now the focus is on what accessors and traits can bring of value to the programming language community.

The structure of this work is as follows. In Chapter 2, required background knowledge is provided. In Chapter 3, the V language as a whole is introduced, explaining its characteristics, type system and operational semantics while focusing on its record and accessor system. In Chapter 4, details about the implementation of V are provided. Finally, in Chapter 5, we conclude this work with the current state of the language, along with known limitations and future work.

Furthermore, there are 4 appendices for the work. Appendix A provides the complete syntax for the V language. Appendix B gives the rules governing operational semantics for the language. Appendix C provides the rules for the language's type system. Appendix D gives information about the extended syntax of the language. Finally, Appendix E provides a simple implementation of accessors in Haskell.

2 BACKGROUND

2.1 Pure Functional Programming Languages

Functional programming is a style of programming that focuses on functions, with concepts such as function composition, currying and anonymous functions. Furthermore, this style of programming treats functions as first-class values, allowing them to be passed as arguments, used as return values of other functions, and manipulated in multiple ways. In pure functional programming, all functions are pure mathematical functions. This means that they do not have any side-effects, and their outputs are purely dependent on their inputs. The lack of side-effects also means that there is no way to change the state of a pure functional program, so the concept of mutable variables or global state is non-existent in purely functional programming.

Although the precise difference between *pure* and *impure* functional languages is a matter of controversy (SABRY, 1998), this text will call pure functional languages those that incorporate *all* aspects of pure functional programming. Some languages, such as F#, allow a programmer to create side-effects in a function (such as IO, mutating state, etc) to facilitate an imperative style of programming. They are still considered functional because they incorporate some aspects of functional programming, such as first-class functions or function composition, but their multi-paradigm approach separates them from pure functional languages. More modern languages, such as Kotlin or Swift, although not considered functional, incorporate several characteristics of functional languages, such as immutability and first-class functions.

Functional language programmers are among those with the highest paying salaries now (STACKOVERFLOW, 2018), and the popularity of functional programming (be it in pure or impure functional languages) is growing constantly. Among functional languages, Haskell and Elm are a couple of the more popular that can be said to be *pure*.

Haskell In 1987, a committee was formed to consolidate the more than a dozen existing functional languages that were created after the release of Miranda. In 1990, the committee released the first version of the Haskell language, serving as a vehicle for further research into functional programming. Despite an explicit stated goal of being used as a basis for language design (HUDAK et al., 2007), Haskell has grown popular in many different fields, including teaching and industry.

Haskell is a general purpose programming language, deriving its name from

Haskell Curry (HUDAK et al., 2007). It features a type system with complete type inference and has lazy evaluation (HENDERSON; MORRIS JR., 1976), which is a kind of non-strict evaluation.

Being a functional language, it supports first-class functions, function composition and pattern matching. By virtue of being pure, without any side-effects in functions, the language can use lazy evaluation as a means of representing very powerful abstractions, such as infinite lists.

Its type system supports type polymorphism in two flavors: parametric polymorphism and ad-hoc polymorphism. Parametric polymorphism is obtained by using Hindley-Milner (MILNER, 1978)(HINDLEY, 1969) type inference, while ad-hoc polymorphism (or overloading) is a result of type classes (HUDAK; FASEL, 1992)(WADLER; BLOTT, 1989).

Elm Elm is a domain-specific programming language for creating simple MVC-based web applications. It is purely functional, and its strong static type system makes it a very robust language for the task. It was initially designed in Evan Czaplicki's thesis in 2013 (CZAPLICKI; CHONG, 2013), with a compiler targeting HTML, CSS and Javascript. It has since expanded to include a read-eval-print loop (REPL), package manager and community-created libraries.

Because of its limited domain scope, it lacks support for many features that languages like Haskell provide, such as parametric polymorphism. Still, the advantages provided by functional purity and type inference, together with its libraries and the MVC app architecture, have allowed Elm to grow rapidly since its creation in 2013.

2.2 Records

Records, also called structs, are one of the basic data structures in computer science. They are collections of *fields*, where each field has a name and value. Typically, the number of fields in a record is fixed upon creation, and thus the only operations that can be performed on a field are accessing the current value and updating the field with a new value. Records are the most common way to represent structured and heterogeneous data in imperative languages, and objects (in object-oriented languages) are essentially records with added functions (or methods).

One of the greatest advantages of records over other data structures is the fact

that the programmer can easily access and update a single field in a record (at least in most languages). This means that even large or nested records do not become unwieldy, and thus they cover many different uses, from simple pairs of values up to complex hierarchies of nested records.

To support modifying a single field of a record, most languages make use of mutability. A record is stored in a specific place of memory, and updating the value of a field simply changes a portion of that memory. By doing this, the same variable can refer to both versions of the record (before and after mutating).

Pure functional languages, however, do not allow this sort of memory mutation. Even though most languages have a way of updating a single field of a record, they require that a new variable be created to refer to this modified record. This change creates a series of issues when dealing with records in pure functional languages, and many different approaches have been taken by different languages.

Below will be presented a brief overview of how Haskell and Elm allow the creation and use of records, and also some examples illustrating why their approaches are not adequate for several uses.

Haskell In the Haskell language, the following code declares a new record type `A` with two fields (`a` and `b`) of type `Int`, along with a value `x` of type `A`. The Haskell language automatically creates two *getter* functions, `a :: A → Int` and `b :: A → Int`, to access the fields `a` and `b`, respectively, within the same scope as that of the data type itself.

```haskell
data A = A { a :: Int,
             b :: Int }

x =  A { a = 3,
         b = 5 }

-- Code below is automatically generated
a :: A -> Int
a (A a _) = a

b :: A -> Int
b (A _ b) = b
```

One of the problems with this approach is that getters are defined as simple functions in the current namespace. This choice forbids two distinct record types from having

the same field names, as occurs commonly in real applications. For this reason, the record type B, defined below, cannot have a field named a, and the programmer is required to use a variation of the name.

```
data B = B { a' :: A,
             c  :: Int }

-- record with another record as field
z = B { a' = A { a = 3,
                 b = 4 },
        c = 6 }
```

Haskell offers a special syntax for record update. By providing a name and a value (or multiple names and values separated by commas) between curly braces, one can treat existing records as functions and "apply" any desired updates. The code below creates a new record, y, which is created exactly the same as x, except for the field a, which now has value 7.

```
y = x { a = 7 }
-- y = A { a = 7, b = 5 }
```

This syntax becomes cumbersome when inner fields of records need to be updated. For instance, the following code would be required in order to update the inner field a of record z to have value 8.

```
z' = z { a' = (a' z) { a =  8 } }
```

As the example shows, we are forced to expose the intermediate updates whenever we are trying to update fields within fields. This can be even worse when dealing with multiple levels of nested records, as it becomes necessary to extract and "repackage" every nested record individually.

Elm Elm has no concept of named record types, but type aliases can be used to obtain the same results as record types in Haskell.

```
type alias A = { a : Int,
                 b : Int }

x = { a = 3, b = 5 }
```

Elm also creates *getter* functions for each field of the record: `.a :: A -> Int` and `.b :: A -> Int`. Notice the function names are preceded by a dot, and in Elm one can write `x.a` to access the field `a` in the record `x`. One advantage of Elm over Haskell is that getters in Elm are not restricted to a single record type. This means that it is possible to define the `B` type as follows:

```
type alias B = { a : A,
               c : Int }

-- record with another record as field
z = { a = { a = 3, b = 4},
      c = 6 }
```

Like Haskell, there is a special syntax for record update. At the surface level, the only difference is that the existing record is positioned inside the curly braces, separated from the updated fields by a pipe (`|`).

```
y = { x | a = 7 }
-- y = { a = 7, b = 5 }
```

This syntax, however, has a limitation that makes updating nested fields even more cumbersome than Haskell: it only accepts names before the pipe, not arbitrary expressions (in other words, one cannot write `{ { a = 3, b = 5 } | a = 7 }` in Elm). This means that, to update an inner field, it is necessary to first bind the inner record to a name.

```
let x = z.a in
   z' = { z | a = { x | a = 8 } }
```

Lenses

One possible approach to manipulate records in functional languages is through the use of Lenses (O'CONNOR, 2011), which are functional references to parts of a complex structure. They allow both accessing and updating these parts and, since they are functions, they can be composed to create lenses for more complex structures.

Because they combine functional purity with the idea of accessing a specific part of a structure, they can be used as a way to allow easier record manipulation in pure functional languages.

Lenses in Haskell Haskell has a Lens package that brings lenses to the language, but it is still experimental and is not a part of the language or standard libraries.

Since Lens is a non-standard library, it must be imported before being used in a program. Furthermore, since creating lenses requires some meta-programming, a pragma must be included in the language (this could be removed if one creates the lenses by hand). In order to create lenses with the expected names `a` and `b`, the actual field names of the data type must be preceded by an underscore.

```haskell
{-# LANGUAGE TemplateHaskell #-}

import Control.Lens

data A = A { _a :: Int,
             _b :: Int }

x =  A { _a = 3,
         _b = 5 }

makeLenses ''A
-- creates lenses a and b
```

These lenses can then be used both as *getters* and as *setters*. This unifies the syntax for these two operations, while also transforming the *set* operation into a first class construct.

```haskell
val = get a x
-- val = 3

y = set a 7 x
-- y = A { _a = 7, _b = 5 }
```

The issue of namespace remains, however. Because of this, the data type `B` must still use a variation of the field name `a`.

```haskell
data B = B { _a' :: A,
             _c  :: Int }

-- record with another record as field
z = B { _a' = A { _a = 3,
                  _b = 4 },
        c = 6 }

makeLenses ''B
```

Finally, lenses resolve the issue of exposing intermediate updates. Since lenses are functions, they can be composed using the regular function composition operator `(.)` to access inner fields. Below, we compose the `a'` and `a` lenses to update an inner field.

```haskell
z' = set (a' . a) 7 z
-- z' = B { _a' = A { _a = 7,
--                    _b = 4 },
--          c = 6 }
```

Although lenses do solve some of the issues that Haskell has when dealing with records, they are far from a perfect solution. They do not address the problem of repeating field names and namespaces, and they are notorious for being extremely difficult for beginners in the language. Not only do users have to understand Haskell metaprogramming, but, since lenses are created on top of complicated type isomorphisms and manipulations, they can create tricky to understand type error messages.

2.3 Polymorphism

Polymorphism (STRACHEY, 2000) is a way to allow functions to operate on different types. There are many different kinds of polymorphism, but the ones most common to functional languages are parametric polymorphism and ad-hoc polymorphism.

Parametric polymorphism is when a function operates uniformly on any type. This kind of polymorphism is what allows a function such as *map* to operate on any list. This kind of polymorphism restricts functions to work on the structure of data, without any knowledge of the actual values contained in it.

As an example, below is the definition of the *map* function in the V standard library. Its type is $\forall a, b.(a \to b) \to [a] \to [b]$, where a and b are type variables, which means that it takes a function that maps values of type a to values of type b, and a list of values of type a, returning a list of values of type b. Note that the definition does not make any restriction on the types a or b, allowing them to be replaced by any concrete type. This means that the same implementation can be used for lists of integers, strings or any other value.

```
let rec map f ls =
  match ls with
  | [] -> []
  | x :: xs -> (f x) :: (map f xs);
```

Ad-hoc polymorphism is when the same name is defined on multiple types, but the behavior is different for each type. This is used, for instance, in equality: the exact behavior when testing for equality depends on the type of the values. Furthermore, ad-hoc polymorphism is restricted to specific types; equality, for example, does not apply to functions in general (due to known undecidability).

As an example, below are two implementations for the equality operation in V, one for numbers and one for booleans. For every new type for which equality is desired, a new function would have to be implemented, describing the expected behaviour for that specific type.

```
let rec (==) n1 n2 =
  match (n1, n2) with
  | (0, 0) -> True
  | (a, b) -> a - b == 0;
```

```
let (==) b1 b2 =
  match (b1, b2) with
  | (True, True) -> True
  | (False, False) -> True
  | _ -> False;
```

While parametric polymorphism is widely used and understood in functional languages through the Hindler-Milner-style type system (HINDLEY, 1969) (MILNER, 1978), ad-hoc polymorphism is rarer and different languages have taken different approaches. Haskell's Type Classes are one way to implement ad-hoc polymorphism, and V implements Traits to achieve a similar effect.

Traits The term trait was first introduced by (UNGAR et al., 1991) to denote a par-

ent object to which an object may delegate some of its behavior. Since then, the term has been used to describe systems of delegating and defining behavior by many different languages, such as Squeak/SmallTalk (NIERSTRASZ; DUCASSE; POLLET, 2009) and Scala (ODERSKY et al., 2004). Other languages have different names for similar tools, such as interfaces in Java (GOSLING et al., 2014) and protocols in Swift. Most instances of traits occur in object-oriented languages, but Type Classes are a Haskell system that also defines specific behavior for types.

In this work, traits are used to support ad-hoc polymorphism (or overloading). A trait defines one or more behaviors or characteristics (such as equality, comparison or accessing fields in records), and is associated with a set of types. When a type exhibits the behavior or characteristic defined by a trait (and, therefore, is associated with it), it is said that the type *conforms* to the trait.

As an example, the *Equatable* trait defines the characteristic of being testable for equality, and is associated to, among other types, integers and booleans. When the equality (=) operator is used in the language, the types of its operands are tested for conformance to the *Equatable* trait in order to check if a program is valid.

So, for instance, the program

```
3 = 4
```

would be valid, as 3 and 4 have type *Int*, and *Int* conforms to *Equatable*. However, the program

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

would be invalid, since the operands are functions, and functions do not conform to *Equatable*.

2.4 Formal Semantics and Type Systems

Formal semantics of programing (FLOYD, 1967) is concerned with precisely defining the meaning of programs. It provides abstract entities that allow only the relevant features of computation to be described. Other features more specific to implementation, such as running time and storages addresses, can be ignored when specifying formal semantics.

Semantics comes in two flavors: *static* semantics and *dynamic* semantics. Static semantics refers to any checks that can be performed before running a program, and is

usually referred to as type checking. Dynamic semantics, on the other hand, refers to the behavior of programs when run.

There are many different approaches to dynamic semantics, such as denotational semantics and axiomatic semantics. In this work, we will focus on operational semantics, in which computations are modeled explicitly. This modeling is done by way of rules that specify transition relations from programs to programs and values.

There are two variants of operational semantics: small-step and big-step. Rules given in small-step semantics evaluate expressions one computation step at a time. This means that evaluation can be described as transitions between subsequent configurations (system states). In big-step semantics, rules describe the computation of any expression in a single derivation. They relate expressions directly to their value and, as such, are usually more concise. On the other hand, the representation of non-termination is not as natural as when using small-step semantics.

In this work, the big-step style will be used to express the operational semantics of V, as it is the most natural way to define an interpreter (KAHN, 1987). This means that the evaluation of expression will be given as a set of rules that map expressions to values.

These rules will have the form:

$$\frac{prem_1 \qquad \cdots \qquad prem_n}{env \vdash e \Downarrow v} \qquad \text{(Example)}$$

where e is an expression; v is the resulting value; $prem_i$, for $1 \leq i \leq n$, are any requirements for the use of the rule; and env is an evaluation environment. The exact definitions of expressions, values and environments will be given later.

Similarly, the type system of V will be expressed as a set of rules of the form:

$$\frac{\texttt{pre-req}}{\Gamma \vdash e : T} \qquad \text{(T-Example)}$$

where e is an expression; T is the resulting type; `pre-req` are any requirements for the use of the rule; and Γ is a typing environment. Again, exact definitions of each of these terms will be given later in the text.

In practice, however, typing is done in V with a type inference system that uses constraint collection and unification. This means that rules given in the format above do not capture the real way that V decides types for expressions. In actuality, the rules for typing expressions are of the form given below:

$$\frac{\texttt{pre-req}}{\Gamma \vdash e : T \mid C} \qquad \text{(T-Example2)}$$

where C is a set of constraints generated by the typing rule.

Although the simplified rules of the form of `T-Example` do not capture the exact way that V's type system works, they are much simpler to understand and do not change much when constraint collection is introduced. Because of this, they will be the preferred way to express typing rules, and only when necessary will the constraint-collection version of these rules be used.

3 THE V LANGUAGE

This chapter will focus on the design of the V language, first describing the language informally and providing an in-depth exploration of its record system. Following this, the chapter will provide, in a formal manner, the syntax (both core and extended) of the language, its operational semantics, and its type system.

3.1 Overview

V is a purely functional strict general purpose programming language with a strong and static type system. It draws heavily from Haskell, both in its syntax and feature-set, although inspiration was also taken from F# to provide some of the syntax.

Being general purpose means that the language does not have any specific applications in which it excels. The language tries to provide the basic tools that are needed by any programming scenario, and leaves the task of creating domain-specific tools to the developer. That being said, V tries to improve upon other languages that already exist in the field, especially in relation to record manipulation in functional languages.

The syntax of the language is divided into two parts: the core syntax, which forms the basis for program evaluation and type inference; and the extended syntax, which provides higher level expressions and can be translated into the core syntax.

Separating the syntax this way simplifies the process of defining the language. By having a small core syntax, the number of typing and evaluation rules can be greatly reduced, making the implementation of these components easier. Furthermore, by having a higher level syntax, it is much easier to add new expressions to the language without having to worry as much about breaking compatibility with existing expressions.

V's type system has support for parametric polymorphism through Hindley-Milner style let-polymorphism (PIERCE, 2002). It also uses the concept of traits, which provide ad-hoc polymorphism, or overloading, to the language. This mechanism is the basis for V's approach to record manipulation, but it also has many other applications, such as allowing a universal equality function.

V has full type inference, meaning that the type system is capable of determining the type of every expression without depending on type annotations from the developer. However, for clarity (or to restrict the type of an expression further than is required), V allows type annotations in certain expressions.

3.2 Record System

This section introduces, in an informal way, the approach V takes for records and accessors. First, the structure and construction of records are given. Second, accessors are described, explaining their use, construction and manipulation. Then, a few helper definitions are provided, allowing easier use and manipulation of accessors. Finally, a complete example of a small program in V which uses records and accessors is given.

Records Records are a comma-separated set of associations between field names (also known as labels) and values enclosed in curly braces. Each field name can only appear once in a given record. For example, the record below has three fields named, respectively, `name`, `level` and `health`.

```
{ name:   "Hero",
  level:  6,
  health: 100 }
```

The type of a record is defined completely by its field names and associated types. Because of this, a record can be constructed without declaring its type beforehand, as opposed to what happens in Haskell. In the example above, the type of the record is `{name:String, level:Int, health:Int}`.

Accessors In its most basic form, an accessor is a field name preceded by an octothorp (#). To actually use accessors, two built-in functions, `get` and `set`, are defined:

1. `get` takes an accessor and a record, returning the value associated with the accessor's name in the record.

   ```
   get #health {stamina: 30, health: 20}
   // returns 20
   ```

2. `set` takes an accessor, a value and an initial record. It returns a new record, replacing the value associated with the accessor's name in the initial record.

   ```
   set #health 0 {stamina: 30, health: 20}
   // {stamina: 30, health: 0}
   ```

An accessor can be used on any record that contains the field name associated to it. In the example below, the `#health` accessor is used on two records of different types,

since both contain a field named `health`.

```
get #health { name:  "P1",
              level: 6,
              health: 20 }
// 20

get #health { stamina: 30,
              health: 100 }
// 100
```

Manipulating Accessors V offers three ways to manipulate accessors: stacking, joining and distorting. All of them take accessors as input and return a (composite) accessor as output, which can then be used with the `get` and `set` functions.

Stacking Updating the fields of a nested record in V by using only basic accessors is still cumbersome. This is illustrated by the following example, in which the inner field `name` is updated to `"John"`.

```
let player = { name: "Hero",
               level: 6,
               health: 100 };

let game = { player: player,
             enemies: [] };

let oldPlayer = get #player game;

set #player
    (set #name "John" oldPlayer)
    game
// { player: { name:   "John",
//             level:  6,
//             health: 100 },
//   enemies: [] };
```

By using V's stacked accessors, however, it is possible to hide the intermediate updates for an inner field, making the respective update syntax much more convenient. In the V language, accessors are stacked with the built-in `stack` function. The following example presents how a stacked accessor is used to perform the same update shown in the previous example.

```
let player = { name: "Hero",
               level: 6,
               health: 100 };

let game = { player: player,
             enemies: [] };

let playerName = stack #player #name;

set playerName "John" game
// { player: { name:   "John",
//             level:  6,
//             health: 100 },
//   enemies: [] };
```

In the example above, `playerName` is the composite accessor which results of stacking accessors `#player` and `#name`. The `stack` function only receives two accessors as inputs, but these can themselves be composite accessors, allowing to focus on an inner field no matter how deep it is located within a nested record structure.

Joining Joined accessors operate on different fields of the same record. The field values are treated as tuples, both for setting new values and for getting the current field values. Below is a simple example of accessing the fields `level` and `health` in a record using a joined accessor.

```
get #(#level, #health) player
// (6, 100)

set #(#level, #health) (7, 80) player
// {name: "Hero", level: 7, health: 80}
```

When setting, joined accessors are "applied" from left to right. This means that, if multiple components of the accessor refer to the same field, the last component is used.

```
set #(#level, #level) (6, 7) player
// {name: "Hero", level: 7, health: 100}
```

When updating a record, joined accessors provide the same functionality as the update syntax in Elm and Haskell, allowing multiple updates to be performed simultaneously.

Distorting It is possible to distort accessors, defining `getter` and `modifier` functions to be applied on an existing field value. These functions allow a field to "store" a value in a different format (or even type) than the one used when operating on it through accessors.

The `modifier` function receives two parameters: the value provided by the caller of the accessor and the old value stored in the field. The function can then use both values to generate a new value to be stored.

Distorting is most useful when the value in a field represents structured data that is not a record, such as a list or a bitmask. By defining a function to destructure the data (the `getter` function) and a function to reconstruct the data (the `modifier` function), it is possible to create an accessor to manipulate only a specific portion of this data.

As an example, the code below allows editing the first enemy of a game.

```
let enemy = { stamina: 20,
              health: 40};

let game  = { player: player,
              enemies: [enemy]};

let getter ls     = head ls;

let modifier x ls = x :: (tail ls);

let firstEnemy = distort #enemies
                         getter
                         modifier;
get firstEnemy game
// { stamina: 20, health: 40 }
```

Another use for distortion is to allow a different view of the data in a field, such as transforming a number into a string. In these cases, usually the stored value is not necessary to generate a new value (i.e the new value is generated purely based on the provided value), and so the second parameter of the `modifier` function is ignored. As an example, the accessor below shows health as a string, by using the built-in `parseInt` and `printInt` functions.

```
let getter h     = printInt h;

let modifier h _ = parseInt h;
```

```
let healthString = distort #health
                           getter
                           modifier;

get healthString player
// "100"
```

Modify In addition to the base `set` function, a built-in `modify` function is also available. This function, instead of taking a value to be inserted into the record, takes a function to modify the existing value in the field. Using it, it is possible to specify a new value for a field taking into account the current value, such as in the following case:

```
let player' = modify #level
                     (\x -> x + 1)
                     player;

// player' = { name: "Hero",
//             level: 7,
//             health: 100 };
```

Dot Access Although the functions and expressions shown above provide all the functionality needed to manipulate records and accessors, using prefixed functions is cumbersome. For this reason, a new syntax is provided that greatly simplifies the most common uses for accessors. This syntax is very similar to accessing attributes in object-oriented languages, commonly referred to as dot notation.

To access a field of any identifier bound to a record, simply put a dot after the identifier and write the name of the field being accessed. This removes both the use of the octothorp and `get` function, and can also be used to access inner fields. As an example, the code below is used to access the name subfield of the player field:

```
let player = { name: "Hero",
               level: 6,
               health: 100 };

let game = { player: player,
             enemies: [] };

game.player.name
```

```
// "Hero"
```

This syntax can also be used with joined accessors. In this scenario, what follows the dot is between parentheses, and, just like with stacked accessors, does not need to be preceded by an octothorp.

```
player.(level, health)
// (6, 100)
```

Inside the parentheses, every accessor can itself be stacked or joined. The example, although contrived, is perfectly legal V code:

```
let getFields record = record.(company.(number, name), ceo.lastName);
```

The dot notation also works with accessors that have been bound to identifiers. In this case, it is necessary to precede the identifier with a single quote ('). This is necessary for using distorted accessors (such as `healthString` defined above), but can also be used for stacked or joined accessors. In the example below, we use an alias for accessing a nested field (notice how the dot notation can also be used on pure accessors, and these must still start with an octothorp).

```
let playerName = #player.name;
game.'playerName
// "Hero"
```

Update There is also a special `update` command to allow simpler updates of fields. An update is written as a field (using the dot notation seen above), followed by a left-facing arrow, and finally the new value to be set in the field. An update can be used exactly like a function, simply passing the record to be updated as its argument.

Below is a simple example of updating a nested field:

```
(update player.name <- "John") game
// { player: { name:   "John",
//             level:  6,
//             health: 100 },
//   enemies: [] };
```

The update syntax supports updating multiple fields at the same time by separating each update with a semicolon (;) and wrapping them inside curly braces. Below is an example updating both the name and level of a player:

```
update {
  name <- "John";
  level <- 7
} player
// { name:   "John",
//   level:  7,
//   health: 100 }
```

This syntax also allows using a function to `modify` an existing field. By replacing the straight line of the arrow with a tilde (~) you can pass a function to modify the existing value in the field.

```
let player' = update { level <~ (\x -> x + 1) } player;
// player' = { name: "Hero",
//             level: 7,
//             health: 100 };
```

This syntax is especially intuitive for defining functions that operate on records. As an example, the code above could be also written as:

```
let increaseLevel = update level <~ (\x -> x + 1);

let player' = increaseLevel player
// player' = { name: "Hero",
//             level: 7,
//             health: 100 };
```

Example

Maintaining the thematic of a game, Figure 3.1 presents a series of functions that manipulate records which can constitute a library for a game. A few helper functions are defined to allow easier manipulation of individual players and enemies, decreasing either their stamina or their health. These functions are then used to compose the final game manipulations: the player attacking either all enemies or a single enemy. As shown in the comments, the functions that implement record types do not define specific record types as arguments. Instead, they require that the records have *at least* a specific set of fields. This is represented in the comments by adding ellipsis (`...`) to the end of the record type, as in `{health:Int,...}`. The comments also display type information for each function, and more details about the type system are provided in Section 3.5.

Figure 3.1: A simple game in V.

```
let player  = { health:  100,
                level:   6,
                name:    "Hero",
                stamina: 40 };

let enemies = [ { health: 20, stamina: 10 },
                { health: 30, stamina: 10 } ];

let game    = { player: player,
                enemies: enemies };

// This function takes any accessor that points to a field of type Int
// taking advantage of accessors being first-class expressions
// reduce :: X#Int -> Int -> (X -> X)
let reduce accessor byAmount = update 'accessor <~ (\x -> x - byAmount);

// damageBy :: Int -> (X -> X)
//   X = { health: Int, ... }
let damageBy = reduce #health;

// staminaDrain :: Int -> (X -> X)
//   X = { stamina: Int, ... }
let staminaDrain = reduce #stamina;

// Drains the stamina of an attacker and reduces the health of an attacked entity
// Both attacker and attacked can be players or enemies
// attack :: (X, Y) -> (X, Y)
//   X = { stamina: Int, ... }, Y = { health: Int, ... }
let attack (attacker, attacked) =
  (staminaDrain 10 attacker, damageBy 10 attacked);

// The player attacks all enemies in the game, draining all his stamina
// swipe :: X -> X
//   X = { enemies: [{ health: Int, ... }], player: { stamina: Int, ... }}
let swipe =
  update {
    enemies <~ map (damageBy 10);
    player.stamina <- 0
  };

// The player attacks one specific enemy
// lungeAt :: Int -> X -> X
//   X = { enemies: { health: Int, ... }, player: { stamina: Int, ... }}
let lungeAt number game =
  let getter ls = ls !! number;
  let modifier enemy ls = setNth number enemy ls;
  let distortedEnemies = distort #enemies getter modifier;
  update {
    (player, 'distortedEnemies) <~ attack
  } game;
```

3.3 Syntax

Figure 3.2 shows a simplified abstract core syntax of the V language. The first few expressions are common to most functional programming languages. In order, we have integers, booleans, characters, empty list, list construction, and tuples; then, identifiers and `let` binding with patterns, followed by lambda abstraction (function), recursive function, function application, and `match` expression; the $error$ expression represents a run-time error or exception. These expressions have the usual behavior, as seen in textbooks such as (PIERCE, 2002). The $builtin$ meta-variable refers to any function that is directly built into the language. This simplified syntax only has functions that relate to records and accessors, while the full language defines many more (such as arithmetic operators, comparisons, etc). Every builtin function has a predefined arity, which represents the number of arguments it takes to become fully evaluated.

Records and Accessors The expression $\{l_1 : e_1, \ldots, l_n : e_n\}$ denotes a record by explicitly presenting the association between labels and expressions. There are two expressions used to create accessors. The first, $\#l$ is the simple field accessor, while the second, $\#(e_1, \ \ldots \ e_n)$ is used to create joined accessors. The other kinds of accessors, stacked and distorted, are created by using the builtin functions `stack` and `distort`, respectively.

Patterns Both `match` and `let` expressions use patterns, which are denoted by the letter p in Figure 3.2. The four first patterns are straightforward: the first, x, is a pattern that always succeeds and creates an identifier binding; the second, _, always succeeds and creates no bindings; the following are patterns for integers, booleans, characters, lists (empty and constructed), and tuples. The last two patterns, record and partial record, are noteworthy. Both match record expressions with fields in any order, with the distinction that the partial record pattern matches records with *at least* the same fields as the pattern, while the record pattern only matches records with *exactly* the same fields.

IO V has a very simple IO system, powered by the builtin functions `read` and `write`. Both of these functions operate on a single character at a time, reading from the standard input and writing to the standard output.

Although not fully developed, V uses a simple Monadic approach (LIANG; HUDAK; JONES, 1995) to support IO. The basis for this system are the `return` and `bind` functions, and more detail can be found in Appendix A.

Figure 3.2: The core V syntax.

e	::=	n	
	|	b	
	|	c	
	|	nil	
	|	$e_1 :: e_2$	
	|	$(e_1, \ldots e_n)$	$(n \geq 2)$
	|	x	
	|	$\texttt{let}\ p = e_1\ \texttt{in}\ e_2$	
	|	$\texttt{fn}\ x \Rightarrow e$	
	|	$\texttt{rec}\ f\ x \Rightarrow e$	
	|	$e_1\ e_2$	
	|	$\texttt{match}\ e\ \texttt{with}\ match_1, \ldots, match_n$	$(n \geq 1)$
	|	$error$	
	|	$builtin$	
	|	$\{l_1 : e_1, \ldots l_n : e_n\}$	$(n \geq 1)$
	|	$\#l$	
	|	$\#(e_1, \ldots e_n)$	$(n \geq 2)$
$match$	::=	$p \rightarrow e$	
	|	$p\ \texttt{when}\ e_1 \rightarrow e_2$	
$builtin$	::=	`get`	(binary)
	|	`set`	(ternary)
	|	`stack`	(binary)
	|	`distort`	(ternary)
	|	`read`	(unary)
	|	`write`	(unary)
	|	`return`	(unary)
	|	`bind`	(unary)
	|	$=$	(binary)
p (pattern)	::=	x	
	|	$\_$	
	|	n	
	|	b	
	|	c	
	|	nil	
	|	$p_1 :: p_2$	
	|	$(p_1, \ldots p_n)$	$(n \geq 2)$
	|	$\{l_1 : p_1, \ldots l_n : p_n\}$	$(n \geq 1)$
	|	$\{l_1 : p_1, \ldots l_n : p_n, \ldots\}$	(partial record, $n \geq 1$)
$x, x_0, x_1, \ldots$	$\in$	$Ident$	(set of identifiers)
$l, l_1, l_2, \ldots$	$\in$	$Label$	(set of record labels)

Extended Language As a way to facilitate actual programming in V, an extended language (syntactic sugar) is defined on top of the core terms and expressions outlined previously. This language is then used as the basis for the parser, and a translation algorithm is used to generate the syntax tree for the core V language.

The extended language provides constructions ranging from a simple `if-then-else` expression to complex multi-parameter functions with pattern matching. It also provides simpler syntax for list and tuple creation, and the extended language is where the special syntax for record access is defined.

Although it will be only briefly mentioned here, the full specification of the extended language can be found in Appendix D.

3.4 Semantics

The operational semantics of V is specified by means of an eager, big-step evaluation semantics with an evaluation environment and static scope. This environment is a mapping from identifiers to values. Besides basic data values (numbers, booleans, characters, lists, tuples and records), closures and accessors are also considered values.

Figure 3.3: Environment and Values

$$
\begin{array}{lcll}
env & ::= & \overline{x \mapsto v} & \\
 & & & \\
v & ::= & n & \\
 & | & b & \\
 & | & c & \\
 & | & nil & \\
 & | & v_1 :: v_2 & \\
 & | & (v_1,\ \ldots\ v_n) & (n \geq 2) \\
 & | & \{l_1 : v_1,\ \ldots\ l_n : v_n\} & (n \geq 1) \\
 & | & \langle x, e, env \rangle & \text{(closure)} \\
 & | & \langle f, x, e, env \rangle & \text{(recursive closure)} \\
 & | & \ll builtin\ .\ v_1,\ \ldots v_n \gg & (n < \text{arity of } builtin) \\
 & | & \#path & \\
 & & & \\
path & ::= & l & \\
 & | & path\ .\ path & \\
 & | & (path_1,\ \ldots\ path_n) & (n \geq 2) \\
 & | & path\ [v_1,\ v_2] &
\end{array}
$$

Environment and Values The definitions for environments and values are depicted

in Figure 3.3. We employ the notation $\overline{x \mapsto v}$ to refer to an unordered mapping from identifiers (x) to values (v). Since V has static scope, a closure $\langle x, e, env \rangle$ represents a function `fn` $x \Rightarrow e$ under environment env, captured at the moment of its evaluation. This environment is then used to evaluate the function's body when the function is applied. The value $\ll builtin\,.\,v_1, \ldots v_n \gg$ represents partially applied built-in functions. They are used when the arity of the function is greater than 1, because application is performed one argument at a time.

Paths are values to which accessor expressions evaluate. This is because accessors view records as trees and, therefore, paths describe how to traverse these trees, from the outermost structure towards the innermost fields. The basic accessor expressions $\#l$ and $\#(e_1, \ldots e_n)$ produce l and $(path_1, \ldots path_n)$ accessors, respectively. The built-in `stack` function evaluates to $path\,.\,path$ accessors. The `distort` function plays the same role for the $path\,[v_1, v_2]$ accessors, where v_1 is the getter distortion and v_2 is the modifier distortion.

Evaluation Rules The judgement $env \vdash e \Downarrow v$ denotes that the expression e evaluates to value v under environment env. Below we present a few evaluation rules to give the reader a sense of the semantics of the language. Appendix B contains the complete list of evaluation rules.

The evaluation of functions is straightforward, producing closures as the following rule shows.

$$env \vdash \texttt{fn}\ x \Rightarrow e \Downarrow \langle x, e, env \rangle \qquad \text{(BS-Fn)}$$

Records require all field expressions to be fully evaluated in order to be considered values. If, at any point in the evaluation, an exception is encountered ($error$), then the whole expression evaluates to $error$. The following rules present record evaluation.

$$\frac{\forall\, k \in [1,n] \;\; env \vdash e_k \Downarrow v_k}{env \vdash \{l_1 : e_1, \ldots l_n : e_n\} \Downarrow \{l_1 : v_1, \ldots l_n : v_n\}} \qquad \text{(BS-Record)}$$

$$\frac{\begin{array}{c}\exists\, k \in [1,n] \;\; env \vdash e_k \Downarrow error \\ \forall\, j \in [1,k) \;\; env \vdash e_j \Downarrow v_j\end{array}}{env \vdash \{l_1 : e_1, \ldots l_n : e_n\} \Downarrow error} \qquad \text{(BS-RecordError)}$$

Pattern Matching A new identifier binding is added to the evaluation environment whenever a pattern x is matched against a value. Pattern matching takes a pattern and

a value to match against, returning an environment if successful. As was done for evaluation rules, only a few examples will be given here, and the full list can be found in Appendix B.

Below is the matching rule for the simple x pattern.

$$match(x, v) = \{x \mapsto v\}$$

If a match fails, it is represented by the following syntax:

$$\neg\, match(p, v)$$

In single-pattern expressions, such as let expressions, failing to match a pattern will result in a runtime exception and will evaluate to $error$. In case expressions, every pattern is matched, from left to right. When the first match succeeds, the associated expression is evaluated. If all patterns fail, then the whole case expression also evaluates to $error$.

$$\frac{\text{env} \vdash e_1 \Downarrow v \qquad match(p, v) = \text{env}_1 \qquad \text{env}_1 \cup \text{env} \vdash e_2 \Downarrow v_2}{\text{env} \vdash \texttt{let}\ p = e_1\ \texttt{in}\ e_2 \Downarrow v_2} \quad \text{(BS-Let)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v \qquad \neg\, match(p, v)}{\text{env} \vdash \texttt{let}\ p = e_1\ \texttt{in}\ e_2 \Downarrow error} \quad \text{(BS-LetError)}$$

Accessors As previously mentioned, accessors evaluate to paths. To illustrate this, some of the rules for accessor evaluation are presented as follows.

$$\text{env} \vdash \#l \Downarrow \#l \quad \text{(BS-Label)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow\!\ll \text{stack} \,.\, \#path_1 \gg \qquad \text{env} \vdash e_2 \Downarrow \#path_2}{\text{env} \vdash e_1\ e_2 \Downarrow \#path_1 \,.\, path_2} \quad \text{(BS-Stacked)}$$

$$\frac{\forall\, k \in [1, n] \;\; \text{env} \vdash e_k \Downarrow \#path_k}{\text{env} \vdash \#(e_1, \ldots\ e_n) \Downarrow \#(path_1, \ldots\ path_n)} \quad \text{(BS-Joined)}$$

`Get` and `Set` Although most evaluation rules will be omitted here for brevity, the rules for `get` and `set` deserve special attention. Since these functions require multiple arguments, their evaluation rules use partially applied built-in functions.

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \ll \texttt{get} \;.\; \#path \gg \\ \text{env} \vdash e_2 \Downarrow \{l_1 : v_1,\; \ldots\; l_n : v_n\} \\ traverse(path, \{l_1 : v_1,\; \ldots\; l_n : v_n\}) = v', r'\end{array}}{\text{env} \vdash e_1\; e_2 \Downarrow v'} \quad \text{(BS-Get)}$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \ll \texttt{set} \;.\; \#path,\; v \gg \\ \text{env} \vdash e_2 \Downarrow \{l_1 : v_1,\; \ldots\; l_n : v_n\} \\ traverse(path, \{l_1 : v_1,\; \ldots\; l_n : v_n\}, v) = v', r'\end{array}}{\text{env} \vdash e_1\; e_2 \Downarrow r'} \quad \text{(BS-Set)}$$

These rules use an auxiliary function, $traverse$, to evaluate the result of applying the accessor to the record. The function takes three arguments: a path, a record and an update value; and returns two values: the value associated with the field specified by the path; and an updated record. This updated record uses the value provided as input to the function to update the field specified by the path.

Traverse There are four cases for this function, one for each path type: simple, stacked, joined and distorted.

Simple Path For simple paths, the function accesses the field l specified by the path, creating a new record by associating the same label to the input value.

$$\frac{\begin{array}{c}1 \leq k \leq n \\ r = \{l_1 : v_1,\; \ldots\; l_k : v,\; \ldots\; l_n : v_n\}\end{array}}{traverse(l_k, \{l_1 : v_1, \ldots l_n : v_n\}, v) = v_k, r}$$

Stacked Paths Stacked paths require three recursive calls to the $traverse$ function. The first call omits the update value, and is used to extract a record rec associated with the first component of the path. This record is then passed, along with the second component of the path and the update value, to the second call of $traverse$. Finally, the third call uses the return rec' of the second call to update the internal record, returning a new updated outer record r'.

$$\frac{\begin{array}{c}traverse(path_1, \{l_1 : v_1, \ldots l_n : v_n\}) = rec, r \\ traverse(path_2, rec, v) = v', rec' \\ traverse(path_1, \{l_1 : v_1, \ldots l_n : v_n\}, rec') = rec, r'\end{array}}{traverse(path_1 \;.\; path_2, \{l_1 : v_1, \ldots l_n : v_n\}, v) = v', r'}$$

Joined Paths Joined paths also require multiple calls to $traverse$, but the exact number depends on the amount of paths joined. Pairing the paths with the components of the tuple provided as the update value, each pair is passed as input to a call to $traverse$. This happens from left to right, and each call returns a part of the old value and a partially updated record. Every call uses the previous partially updated record, and the last call to $traverse$ returns the fully updated record.

$$\frac{\begin{array}{c} path = (path_1,\ \ldots\ path_n) \qquad v = (v_1,\ \ldots\ v_n) \\ r_0 = \{l_1 : v_1, ...l_n : v_n\} \\ \forall i \in [1,\ n]\ .\ traverse(path_i, r_{i-1}, v_i) = v'_i, r_i \end{array}}{traverse(path, \{l_1 : v_1, ...l_n : v_n\}, v) = (v'_1,\ \ldots\ v'_n), r_n}$$

Distorted Paths Distorted paths require two calls to $traverse$, one before and one after applying the distortions. First, the current value of the field is extracted. This value is then passed to the first component (v_1) of the accessor, returning the distorted current value. Then, the new distorted value v, along with the current value of the field, is passed to the second component (v_2) of the distorted accessor. This value is then provided as the new update value for a call to $traverse$, returning the updated record.

$$\frac{\begin{array}{c} traverse(path, \{l_1 : v_1, ...l_n : v_n\}) = v_{old}, r \\ \{\} \vdash v_2\ v\ v_{old} \Downarrow v' \qquad \{\} \vdash v_1\ v_{old} \Downarrow v'_{old} \\ traverse(path, \{l_1 : v_1, ...l_n : v_n\}, v') = v_{old}, r \end{array}}{traverse(path\ [v_1,\ v_2], \{l_1 : v_1, ...l_n : v_n\}, v) = v'_{old}, r}$$

3.5 Type System

V has a Hindley-Milner type inference system, which means type annotations are not necessary to properly type a term. With let-polymorphism, it also supports parametric polymorphism. In other words, it allows functions to be defined for *all* types (such as the identity function). With the existence of traits, another type of polymorphism is allowed: ad-hoc polymorphism. This kind of polymorphism allows functions to accept *some* types (such as the equality function or record accessor). Figure 3.4 shows all available types and traits in the V language.

Types Most of the types, such as integers, tuples, functions, and record types, exist as they are usually presented in other functional languages. The IO T type is used for all

Figure 3.4: Types

T	::=	Int	
	|	Bool	
	|	Char	
	|	Void	
	|	T list	
	|	$(T_1, \ldots T_n)$	$(n \geq 2)$
	|	IO T	
	|	$T_1 \to T_2$	
	|	$\{l_1 : T_1, \ldots l_n : T_n\}$	$(n \geq 1)$
	|	$T_1 \# T_2$	$Accessor$
	|	$X^{\overline{Trait}}$	
$Trait$	::=	$Equatable$	
	|	$Orderable$	
	|	$\{l : T\}$	$(Record\ Label)$
$X, X_1, X_2, \ldots$	$\in$	$TypeVar$ (set of type variables)	
$l, l_1, l_2, \ldots$	$\in$	$Label$	

IO operations in the language. In it, T represents the type of the result of the operation: for the `read` function, it will be Char; the `write` function results in a type of IO Void.

V introduces accessor types $T_1 \# T_2$, where T_1 is a record type and T_2 is the type of the field being accessed. This allows us to define the type signature of the `get` and `set` function as follows:

1. `get` :: $T_1 \# T_2 \to T_1 \to T_2$
2. `set` :: $T_1 \# T_2 \to T_2 \to (T_1 \to T_1)$

Traits The traits *Equatable* and *Orderable* are used for equality (= and $\neq$) and comparison ($\geq, >, <, \leq$) operations, respectively. The other traits, named *Record Label*, specify a single label name and associated type that a record type must have. Below we present some of the rules that define the *conforms* relation between types and traits. As an example, since integers can be ordered and tested for equality, we have the following axioms.

$$\text{Int} \in Equatable$$

$$\text{Int} \in Orderable$$

Composite types such as tuples and records conform to $Equatable$ if all of their component types conform to $Equatable$, as the trait conformance rule for tuples, presented below, shows.

$$\frac{\forall i \in [1,n] \ . \ T_i \in Equatable}{(T_1, \ \ldots \ T_n) \in Equatable}$$

Functions, on the other hand, cannot be tested for equality. Therefore $(T_1 \rightarrow T_2) \notin Equatable$ and there is no conformance rule for $Equatable$ regarding functional types.

Record types are the only types that can conform to *Record Label* traits. Furthermore, the record must have a field with the same name and type as the one defined in the trait. This means that a record type with n fields automatically conforms to n Record Label traits, one for every association between a name and a type, as shown by the following rule.

$$\frac{\exists \, n \in [1,k] \ . \ l_n = l \ \wedge \ T_n = T}{\{l_1 : T_1, \ \ldots \ l_k : T_k\} \in \{l : T\}}$$

To illustrate the use of traits and how they are deployed in the language, below is the typing rule for the equality operator. It specifies that the type T of the arguments must conform to the $Equatable$ trait.

$$\frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T \qquad T \in Equatable}{\Gamma \vdash e_1 = e_2 : \text{Bool}} \qquad \text{(T-=)}$$

Type Variable The type $X^{\overline{Trait}}$, commonly referred to as a type variable, exists only as a tool for the type inference system. In it, a type variable is used to represent unknown types during the type inference algorithm. Each type variable is associated with a set of traits which it must satisfy, limiting the types which it can represent. If this set is empty (or if the type variable is shown without any associated traits), the type variable becomes universally quantified and can be substituted for any type available in the language.

Traits and Accessors The polymorphism observed in accessors is due to their use of traits. Every basic accessor (that is, an accessor for a single field) has a corresponding *Record Label* trait. By checking trait conformance, a basic accessor $\#l$ can be used with any record that conforms to a record label trait (i.e has a field with the label l).

$$\frac{T_2 \in \{l : T_1\}}{\Gamma \vdash \#l : T_2 \# T_1} \qquad \text{(T-Accessor)}$$

Compound accessors, be they stacked, joined or distorted, do not create additional trait requirements, but instead create new accessor types based on existing ones. As can be seen below, there is no premise testing for trait conformance in the rule for $stack$.

$$\frac{\Gamma \vdash e_1 : T_2 \# T_1 \qquad \Gamma \vdash e_2 : T_1 \# T_3}{\Gamma \vdash \mathsf{stack}\ e_1\ e_2 : T_2 \# T_3} \qquad \text{(T-Stack)}$$

Traits have no structure in common, and there is no way to compose traits. This means that, while traits may be used in V to obtain a form of structural subtyping relationship between records, there are subtle distinctions between the two systems.

Type Inference For type inference, V uses a constraint-based approach, dividing the algorithm into three parts: constraint collection, in which the abstract syntax tree is traversed and both a type T and a list of type constraints C is generated; constraint solving, which returns a type substitution σ mapping type variables to types; and substitution application, which applies the type substitution σ to the type T to obtain a principal type. For the sake of clarity, only a brief overview will be given for each part of the algorithm.

Constraint Collection The constraint collection stage takes, as input, an expression e and a typing environment Γ, and produces, as outputs, a type T and a set of constraints C. Rules for constraint collection have the form

$$\Gamma \vdash e : T \mid C$$

The environment Γ used in constraint collection is a mapping from identifiers to type schemes. The definition of the environment, along with the two variations of type associations, is given below.

$$\begin{array}{lcll} \Gamma & ::= & \overline{x \mapsto assoc} & \\ \\ assoc & ::= & T & \text{(Simple Association)} \\ & \mid & \forall X_1, \ldots X_n \,.\, T & \\ & & & \text{(Universal Association)} \end{array}$$

Constraints are simply equations between two types. This allows creating exact match between two types (i.e. type T is equal to type S), as is necessary when making sure the first term of an application is a function, for example.

However, regular type constraints cannot be used to express to trait conformance. In order to adapt the constraint system to allow the creation of these different types of constraints, a slight workaround using type variables is employed. By default, types in the V language do not carry trait conformance information with them. The only exceptions to this are type variables, each one having a declared set of traits associated with it. By using this fact, it is possible to create an equality constraint that only enforces trait requirements.

As an example, we present the constraint collection rule for the equality function $=$.

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1^{\{Equatable\}}\ is\ new}{\Gamma \vdash e_1 = e_2 : \text{Bool} \mid C_1 \cup C_2 \cup \{T_1 = T_2; X_1^{\{Equatable\}} = T_2\}} \tag{T-=}$$

First, T_1 and T_2 are obtained as the types of e_1 and e_2, respectively. The assertion

$$X^{\{Equatable\}} is\ new$$

states the creation of a new type variable X associated with the *Equatable* trait. Finally, two new constraints are generated: the first guarantees that T_1 is equal to T_2; the second creates an equality between T_2 and X. Since X can represent any type that conforms to the $Equatable$ trait, this constraint will only be satisfied if T_2 is a type that conforms to $Equatable$.

To illustrate the constraints that accessors introduce, we present below some of the constraint collection rules for accessors.

$$\frac{X_1\ is\ new \qquad X_2^{\{\{l:T_1\}\}}\ is\ new}{\Gamma \vdash \#l : X_2^{\{\{l:T_1\}\}} \# X_1 \mid \{\}} \tag{T-Label}$$

$$\frac{X_1\ is\ new \qquad X_2\ is\ new \qquad X_3\ is\ new}{\Gamma \vdash \texttt{stack} : X_1 \# X_2 \to X_2 \# X_3 \to X_1 \# X_3 \mid \{\}} \tag{T-stack}$$

$$\frac{\begin{array}{c} X_0\ is\ new \\ \forall\, i \in [1,n]\,.\ X_i\ is\ new\ \wedge\ \Gamma \vdash e_i : T_i \mid C_i \end{array}}{\Gamma \vdash \#(e_1, \ldots e_n) : X_0 \#(X_i, \ldots X_n) \mid \bigcup_{i=1}^{n} C_i \cup \{T_i = X_0 \# X_i\}} \tag{T-Joined}$$

The `get` and `set` built-in functions do not create constraints by themselves, as their constraint collection rules, given below, show.

$$\frac{X_1 \; is \; new \qquad X_2 \; is \; new}{\Gamma \vdash \texttt{get} : X_2 \# X_1 \to X_2 \to X_1 \mid \{\}} \tag{T-GET}$$

$$\frac{X_1 \; is \; new \qquad X_2 \; is \; new}{\Gamma \vdash \texttt{set} : X_2 \# X_1 \to X_1 \to X_2 \to X_2 \mid \{\}} \tag{T-SET}$$

Constraint Solving Constraint solving is performed by the unification algorithm, which attempts to solve the set of equalities defined by the constraints collected in the previous step. The algorithm takes, as input, a set of constraints C, and produces a substitution σ as output. A substitution is a mapping from variable types to types.

$$\sigma \quad ::= \quad \{\} \mid \{X \mapsto T\} \cup \sigma$$

For most types, the unification process is straightforward: if the types are the same, they are considered unified. If they are composite types (such as tuples and records), constraints between their components are added to the end of the constraint list as new constraints. If the types do not match, the unification process stops and the expression is badly formed. When it comes to variable types, however, unification is more complex. Since a variable type comes with a set of traits associated with it, the conformance of the type to which it is being equaled must be checked. This process might create more constraints, and these must be added to the end of the list. Every iteration of the unification process might create more constraints but, since the types are reduced at each recursive call, the algorithm is guaranteed to terminate.

Application The last component of the type inference algorithm is the application of a type substitution. This replaces all variable types that are specified by the substitution, resulting in a new instance of the input type. Application takes, as input, a type T and a substitution σ, producing another type T' as output.

3.6 Comparing V to other Languages

V is similar to most functional languages in design, with most of the language differing from existing ones only in syntax. The aspects of V that diverge from the norm are its record system and type traits, and these are worth a closer inspection.

3.6.1 Incorporating the record accessor system in other languages

The record system of V could be replicated in other pure functional languages. The essential features that a language must have are parametric polymorphism, some kind of ad-hoc polymorphism and a way to declare records. Although V has records as anonymous constructions, this is not a necessity to support V's record system. However, the language should allow distinct record types to share the same field names.

As an example, let us look at how Haskell could go about incorporating V's record system. First, Haskell would need to allow the creation of different record types with the same field. This would be a necessity to allow the polymorphic behavior seen in V's accessors. Since the only reason Haskell currently does not allow this is because of the generated `getter` functions, implementing accessors would remove this restriction.

Another requirement for implementing accessors is creating the accessor functions themselves. One possible approach is the one taken in (HASKELLWIKI, 2018), in which laziness and pattern matching are used to allow both getting and setting the field of a record in a single function.

Finally, accessors must be generated automatically and be usable by multiple records. This can be done by generating a type class for every field in any declared record. By using pattern matching, multi-parameter type classes and functional dependency, these type classes are simple and easy to generate. In the example below, the compiler would need to generate code for creating an accessor for the field `code` in the `Record` type (the same would need to be done for the `age` field).

```haskell
type Accessor rec field = rec -> field -> (field, rec)

data Record = R {code :: String, age :: Int}

-- Generated automatically
class CodeLabel rec field | rec -> field where
  codeAcc :: Accessor rec field

instance CodeLabel Record String where
  codeAcc r@(R code _) string =
      (code, r { code = string })
```

Any new record types that are declared with the same field name (i.e `code`) would only require a new instance declaration, since the type class `CodeLabel` is already de-

clared. If desirable, it would be possible to allow the programmer to create custom instances of the generated type classes, granting them the ability to use accessors on data distinct from records.

Finally, implementing all accessor related functions is easy in Haskell. As an example, below would be the basic implementations for `get` and `set`. A more complete example, with implementations for `stack` and `distort` and some uses, can be found in Appendix E.

```
get :: Accessor rec field -> rec -> field
get acc rec = fst $ acc rec undefined

set :: Accessor rec field -> field -> rec -> rec
set acc value rec = snd $ acc rec value
```

3.6.2 Limitations

As already mentioned, while traits allow some form of ad-hoc subtyping, they are not a complete replacement for actual structural subtyping and all the flexibility it provides.

With structural subtyping, the following function, for instance, could be defined and applied to lists of records of different record types (as long as these record types have a lowest common record type).

```
let extractNames ls =
  map (get #name) ls
```

With the current system of traits, however, it is impossible to create such a list. This is because different record types aren't connected in any way, and a list requires that the type of every element be the same.

If the language had subtyping relations, it would be possible to find a lowest common type between every record type, thus allowing the creation of lists with distinct types for every element.

A more nuanced analysis of the differences between traits and subtyping (or Type Classes) could reveal more details about the advantages and disadvantages of each. Doing this kind of analysis was outside the scope of this work, but some exploration could

provide useful information in understanding how these different systems behave in relation to each other.

4 V IMPLEMENTATION

The formal semantics of V was developed in parallel with a functioning interpreter, which allowed experimenting in usability during the development process. This chapter will focus on this implementation, giving details of each component and decisions made along the way.

4.1 Overview

The implementation of V was done in F#, itself a functional language (although impure). This decision was made because of two main reasons. First, it was the language in which the class project was implemented, so there was already a codebase to work with. Second, functional languages lend themselves very well to implementing parsers and compilers because of their recursive nature and strong type systems.

This implementation of V is interpreted, meaning that one cannot compile V code into executable code. Instead, the interpreter takes an input syntax tree and executes the program directly inside the F# runtime. Although there is no reason that V could not be compiled, starting with an interpreted version of the language was more straightforward and served the purpose of demonstrating the capabilities of the language.

The V interpreter has 4 stages, all of which will be explained in detail later:

1. Parsing
   Takes source code and transforms it into an extended syntax tree
2. Translation (desugaring)
   Transforms an extended syntax tree into a core syntax tree
3. Type Inference
   Performs type inference and type checking for programs in V
4. Evaluation
   Executes a program in V

In order to facilitate programming, it became necessary to define a standard library of functions, such as function composition, transformation between strings and integers, etc. To do this, an ad-hoc library system was implemented, which allows programs to import external code into a V source file. This system by no means functions as a complete library system, but it serves the purpose of providing a simple way to use

functions and constants defined in another file.

4.2 Parser

The parser takes as input source code in V and produces, as output, an extended syntax tree. The parser was created using FParsec (TOLKSDORF, 2013), a parser combinator library for F#. This decision means that the parsing is done in one stage, without the need for tokenization beforehand.

For the most part, the parser analyzes the source code in a purely syntactic way. This means that it catches errors such as using a reserved keyword as an identifier, but not using an identifier before its declaration (this is done by the translator).

The only exception, and where the parser uses semantic information to perform parsing, is for infix operators. V allows the programmer to specify infix operators with a custom priority and associativity, and the only way for the parser to correctly parse expressions that use infix operators is to know these data.

The parser does this by maintaining a state while parsing. This state is simply a collection of available infix operator names and their priority and associativity. Whenever the parser encounters an infix operator declaration, it stores its name, priority and associativity and the operator can then be used later in the code.

The parser is the single component that has suffered the most changes throughout the development process. As the language evolved, several additions and changes to the definition of the language made the previous versions of the parser obsolete, and it had to be rewritten from scratch.

The first functioning version of the parser (the first attempt was using regular expressions, but that failed as soon as parentheses were introduced) was made by hand, cleaning the input string manually and looking at the first characters to decide what term it was parsing. This worked for correct programs and while the language was simple, but it soon became unwieldy to work with.

In this stage, there was no concept of a core and extended language, so the parser generated the same syntax tree that would be used by the type system. As I wanted to add expressions that were easier to use for programmers, such as multi-parameter functions and list syntax, the parser started to be responsible for translating these complex expressions into the language's core terms.

After having to deal with multiple different translations and parsing in the same

place, an extended language was created, making the parser deal only with creating this extended syntax tree. This reduced the complexity of the parser a lot, but it still had several drawbacks, such as no parsing environment and terrible error reporting.

After I decided to add custom infix operators to the language, the lack of a parsing environment and the complexities of manually doing parsing became untenable. I looked into parsing libraries, and FParsec was the most recommended for F#. It copies most of its API from Parsec, a Haskell library for parsing, and so is perfectly suited for functional programming.

By composing different parser components, it is simple to create a parser that expands easily whenever new structures or expressions are added. With the addition of an environment that follows the parsing, it became possible to allow the programmer to define custom infix operators with different priorities and have the parser correctly associate terms with operators.

Another benefit of changing the parser to use a parsing library was speed. The manual parser took seconds to parse the standard library, making it the slowest component in the interpreter. The new parser reduced this time to a few hundred milliseconds, and now it is no longer the bottleneck in the interpreter pipeline.

4.3 Translation

As was described in the previous section, the extended language was added to V as a way to allow creating easier to use programmer-facing API at the same time as maintaining a simple and compact language core. As the language grew, most of the new additions were made at the extended language level, and even some features that were once in the core could be generalized and moved into the extended language, such as conditionals, tuples and lists.

The translator works on two levels: first, it translates extended language expressions into their equivalent expressions in the core language. This means transforming a conditional into a match expression; changing a tuple into a series of function applications (with the tuple constructor); transforming a multi-parameter function into multiple single-parameter functions; etc.

Second, the translator validates certain characteristics of its source program. One of these is making sure that type aliases are declared before being used. Since the core language has no concept of a type alias, it is the translator's responsibility to find the

actual type to which the alias refers.

Because of this, the translator requires an environment. Besides keeping track of type aliases, this environment also maintains a list of identifier substitutions. When translating, any declared identifier is replaced by a newly generated, unique identifier, and these must be kept track of to guarantee correct translation.

One of the main reasons for replacing declared identifiers is that the translator must, on occasion, generate new identifiers. This happens mostly when translating functions. In the extended language, functions allow the use of patterns, while the core language only allows binding of identifiers in function arguments. This means that the translator must create new *unique* identifiers, and the easiest way to make sure that every identifier is unique is to always generate new identifiers, even for user-declared bindings.

4.4 Type Inference

This implementation uses the constraint-collection version of the type system defined before. This means that there are 3 stages for type inference: constraint collection, solving constraint and substitution application. The first two stages (collection and unification) each have environments that are constructed while the abstract syntax tree is traversed and constraints are collected.

The implementation of the type system is flexible enough to allow for the addition of new types, traits and trait conformances, even though these features do not yet exist in the language. This is done by placing most information about types in the constraint collection and unification environments, which allows the algorithms to react to new types or traits.

Although there are 3 separate stages for the typing algorithm, the constraint collection stage calls both the unification and application stages in one scenario: typing a `let` expression. Because of polymorphism, the constraint collection algorithm needs to find the principal type of the value being bound, so it relies on unification to do this.

4.5 Evaluation

Much like the type inference system, evaluation is implemented in such a way as to allow easy addition of new built-in functions and data constructors. Evaluation is done under an environment that contains, in addition to a mapping from identifiers to values, a list of available constructors and the number of arguments they take.

Since V has eager evaluation, all arguments are evaluated before being applied to a function. This poses a problem for functions that should have short-circuit behavior, such as the boolean `AND` and `OR`. For this reason, the implementation treats these two functions differently, postponing the evaluation of the second argument only if it is necessary. Unfortunately, this means that the user cannot create custom functions with short-circuit natively, and so must use workarounds such as functions that accept closures with a Void (or ignored) argument.

IO is implemented in a very rudimentary manner. When a read function is encountered, the F# runtime is called requesting a single character from the Console (stdin). The same is performed on write, converting a V character into an F# character and writing it to the Console (stdout). Reading and writing complete lines was easily implemented in the standard library.

4.6 REPL

The REPL (Read-Eval-Print Loop) is the best way to test the language and quickly experiment with its features and syntax. It allows a user to write expressions that are immediately evaluated and printed on the screen, while also declaring constants and functions that can be used during the REPL session.

The REPL works in a very straightforward manner. First, it tries to parse the code inserted by the user as an expression. If this succeeds, the code is evaluated and the resulted is printed on the screen. If the parsing fails, the REPL then attempts to parse the code as a library (i.e, as a declaration). If this succeeds, the resulting declaration is added to the running environment. If both attempts at parsing fail (or a typing/evaluation failure occurs), the error is printed on the screen and no changes are made to the environment.

Besides allowing basic evaluation of expressions, the REPL also provides two commands to the programmer. The first command, `<type>`, makes the REPL print, in-

stead of the result of evaluating an expression, the type of the expression. This is information that, normally, would not be accessible to the programmer, but can be useful to understand what the type system is doing for any specific expression.

The second command, `<clear>`, removes all declarations from the environment. Similar to this, the programmer can choose to create a REPL session that does not have the standard library preloaded. This means that functions such as `head`, `printInt` or `.` (function composition) will not be available to use.

4.7 Libraries

Libraries exist on two levels of the V language: parsing and translation. A library is, essentially, a list of declarations and a list of operators with their priority and associativity that can be imported into a source code. This system does not have the concept of namespaces and can cause overriding of functions and constants if multiple libraries are imported, but it serves the simple purpose of creating a set of functions that can be reused in multiple programs.

Parsing supports libraries by having a mode that parses a sequence of declarations, instead of parsing expressions. This means that the parser will reject source code that ends in a value, and will instead only accept a sequence of `let` declarations.

Furthermore, the parser, when encountering an `import` expression, will immediately search for a file with the requested name and insert the operators into the parsing environment. The result of parsing an `import` is just a sequence of declarations, acting as though the text of the library replaced the expression.

Although it serves the purpose of supporting a simple standard library, the whole system will need to be rethought in order to actually be useful in a production environment. As it stands, the lack of namespaces is a major hindrance to effectively using libraries, and this can only be done by a complete overhaul of the system.

Furthermore, the actual implementation of libraries also leaves a lot to be desired, especially when it comes to translation. Currently, information about unique identifiers is lost when importing a library, and this can bring severe issues when importing multiple libraries into a single source file. As the whole design of the library system needs to be changed, fixing these issues is not a priority, and the system is still usable is most cases.

4.7.1 Standard Library

The standard library is a collection of functions and operators intended to facilitate programming in V. Some features that were once in the core or extended language were moved to the library when the language became powerful enough to support their implementation, and many other useful functions were implemented as a way to test the language and provide sample code in V.

The standard library offers 64 definitions, most of them functions, a few operators, and a single type alias (from `[Char]` to `String`). These functions are diverse in nature, with functions for manipulating lists, integers, tuples and even other functions. Figure 4.1 shows a few functions that illustrate what the standard library has to offer.

Figure 4.1: Sample of functions in the standard library

```
// Swaps the components of a tuple
let swap (x, y) = (y, x);

// Composes two functions
let compose f g x = f (g x);

// Infix version of composition. Allows the syntax "f . g" to be used
let infixr 9 (.) = compose;

// Returns the first element of a list (is a partial function)
let head (x :: xs) = x;

// Applies a function to every element of a list
let rec map f ls =
    match ls with
    | [] -> []
    | x :: xs -> f x :: map f xs
;

// Writes a string to the output, followed by a new line
let rec writeLn line =
        match line with
        | [] -> write '\n'
        | char :: rest ->
                do {
                        write char;
                        writeLn rest
                }
;
```

4.8 Tests

In order to maintain sanity throughout the development process, a suite of tests had to be created. Some of these tests were made after bugs were found in an ad-hoc manner, while others were created to establish the expected behavior before any code was written. They not only serve to make sure that language features are kept functioning throughout development, but also provide a way to create sample programs and see how the language behaves in a number of scenarios.

Since the test suite was created in parallel with the implementation, the tests grew organically as they were needed. This means that no metrics were used to evaluate the tests or to decide which tests to implement, leading to possible duplication of tests or lack of coverage. This lack of formalism did not have any noticeable downsides, however, as the suite of tests was able to adequately serve its purpose.

The language has a suite of more than 400 tests, ranging from simple parser tests, such as ensuring proper associativity and nested expressions, to complete system tests that parse, translate, type check and evaluate whole programs. These tests help provide confidence when making changes to the language, as many bugs have been caught by a single failing test in supposedly unrelated parts of the language.

Table 4.1 shows the distribution of these tests. The standard library is the component with by far the most number of tests, comprising more than 60% of the test suite. These tests exercise the high-level functioning of the language, as they test the parser, type inference and evaluation at the same time.

The other tests are more directed, and so are useful for defining the expected behavior of a single component in a specific circumstance. These include tests such as wrong type declaration for the parser, repeated variables for the translation and the correct application of polymorphism in the type inference.

Table 4.1: Test Suite

Type of Test	Number of Tests
Evaluation	26
Type Inference	52
Translation	22
Parsing	45
Standard Library	254
Records	23
Total	422

4.9 Statistics

This section is a collection of statistics about the current implementation of V.

Table 4.2 shows the amount of lines of code for every component in the interpreter. The core components (parser, translator, type inference and evaluator) comprise only about 36% of the total line count, while tests are almost half of the entire codebase. The rest of the code is comprised of miscellaneous files, such as data definitions and the code for the REPL.

Table 4.2: Lines of Code per Component

Component	Lines of Code
Parser	726
Translator	368
Type Inference	856
Evaluator	591
Tests	3247
Others	1114
Total	6902

Although efficiency was not a goal when implementing the interpreter, it is still useful to illustrate how it behaves in varied scenarios. The tables below show execution times for multiple programs, and all times shown were obtained by averaging 3 separate executions of the same code to remove any variations. All of the tests were compiled using Microsoft® F# Interactive version 10.1.0 for F# 4.1 and executed on a machine running Windows 10 Pro (Version 1803), with an Intel® Core™ i5-4460 processor and 16 GB of RAM.

The first program to be tested is the smallest possible program in V:

```
return 0
```

This code was executed both with and without a standard library, and the time taken for each stage of the interpreter is shown in Table 4.3. Without the standard library, the stage that takes the most time is the parser, comprising 70% of the total runtime. Adding the standard library completely changes the picture, however. Although every stage takes longer to execute, type inference stands out with a 12x increase.

This is most likely caused by the fact that every function in the standard library is implicitly typed. The type checker must then infer the type of every expression in the

program, and this process grows with a time complexity larger than linear to the amount of values. The runtime would possibly be reduced if the functions were explicitly typed, but a better solution would be to improve the efficiency of the implementation of the type checker.

Table 4.3: Execution Time - Small Program

Stage	Time (ms)	
	with stdlib	without stdlib
Parser	186	154
Translator	13	5
Type Inference	590	44
Evaluator	21	17
Total	810	220

The next example is a program on the other extreme: 3000 lines of code. This program was made by replicating the standard library 8 times, and the results of execution are shown in Table 4.4. As can be seen, the increase in time for most stages grows at a manageable rate. The type inference stage is the single largest contributor to the increase in execution times for larger programs.

Table 4.4: Execution Time - Large Program

Stage	**Time (ms)**
Parser	386
Translator	41
Type Inference	33.600
Evaluator	21
Total	34.048

Although the program shown above was large, it did not actually compute any values. The following examples are small programs to compute values, and so the most time-consuming stage is the evaluation. Because of this, I have chosen to omit the times for the other stages, focusing only on the **Evaluator**.

The first program is a naive implementation of the fibonacci numbers. The program, shown below, is then executed with a number of different input values. Table 4.5 shows the execution time for these inputs and. As expected, the time taken to execute the code increases exponentially.

```
let rec fib n =
```

```
        match n with
        | 0 -> 1
        | 1 -> 1
        | n -> fib (n - 1) + fib (n - 2)
;
```

Table 4.5: Execution Time - Fibonacci

Input	Evaluation Time (ms)
1	32
2	32
4	32
8	32
16	46
18	68
20	130
22	280
24	690
26	1800
28	4677
30	12366

The last example is sorting a list. V implements a very naive version of quicksort in its standard library, and the code for it is shown below:

```
  let rec sort ls =
    match ls with
    | [] -> []
    | pivot :: xs ->
        (sort $ filter ((>) pivot) xs)
        @ [pivot] @
        (sort $ filter ((<=) pivot) xs)
;
```

Table 4.6 shows the execution time for sorting lists of different sizes. Two variations of these lists were used: one that is already sorted, and one that is in reverse order. As can be seen, these two variations produce different execution times, as expected of the quicksort algorithm.

As was said in the beginning of this section, efficiency was not a goal of the current implementation of the interpreter. That being said, the results shown here give a general picture of the state of the current implementation and indicate a few areas that could be improved.

Table 4.6: Execution Time - Quicksort

Size	Time (ms)	
	Ordered	Reverse Ordered
10	30	30
50	51	58
100	140	155
150	243	335
200	430	590

The type inference system is one of the main bottlenecks when it comes to execution times, and so improving its implementation would yield the most benefit. Furthermore, a few tests ran into stack-space issues during evaluation, and so reducing usage of the stack (and any other memory space) could alleviate these problems.

5 CONCLUSION

This work presented the V language, describing its creation and giving motivation for its existence. Then, an informal overview of the language's record system was given, focusing on accessors (and their manipulators) through the use of code examples.

A formal definition of the language followed, describing the syntax, semantics and type system of V. The syntax is divided into a core language and sugar, with inspiration from Haskell and F#. The semantics is strict, inspired mainly by F#, with the concept of *paths* for evaluating accessors. The type system uses a constraint collection and unification algorithm for type inference, modifying the basic Hindley-Milner to add support for traits.

A functioning interpreter was implemented for the language, complete with parser, type-inference, evaluation and a REPL. This implementation was created alongside the definition of the language, so many parts of it evolved throughout the work. Almost half the codebase of the implementation is comprised of tests, ranging from testing single components to the whole program.

Although efficiency was not a goal of this work, some preliminary tests were made to analyze the runtime of the interpreter. These tests show that the implementation of type-inference system was the main bottleneck, specially as the input program grows, so it is one the main candidates for optimization work in the future.

5.1 Status and Future Work

The V language is completely open-source, both its definition and its implementation. There is extensive documentation available describing its syntax, semantics, type system, and even its standard library. These projects can be found in the following repositories:

https://github.com/AvatarHurden/V
https://github.com/AvatarHurden/V-Documentation

The language is still under development, with many possible fronts still to be explored. Among these are language features such as list comprehensions and allowing the declaration of new traits and types; and other meta-language features such as a complete library system and more comprehensive error reporting.

Currently, the language has support only for basic IO with the foundation for a Monad system in place. The next step is to fully implement the Monad system, creating a Monad trait and generic `return` and `bind` functions. This will also require some changes to the evaluation system, since it currently only supports polymorphism based on the input values.

One important aspect of language design is proving properties such as type safety. Although the language was designed with type safety in mind — and, since the implementation went in stride with the definition, many tests were created to test the design — proofs of type safety were neglected. Usually, these kinds of proofs are done using small-step semantics, which would pose a problem given that V's semantics is defined with big-step semantics. However, work done in (SIEK, 2018) shows a technique that could be used to prove type safety while using big-step semantics.

Finally, it would be necessary to implement a compiler in order to make V a complete language. The interpreter serves as a way to test the language and showcase its features, but, as was shown in Section 4.9, its efficiency is subpar (especially the type-inference system). By implementing a compiler, not only would the language be faster, but certain issues, such as stack overflow for non-terminating programs, would not occur.

5.2 Publications

A paper about V and its record system was submitted to and presented at WEIT 2017, receiving the award of second best paper among 47 other works. This paper focused mostly on V's approach to records, with only brief mentions of the language design as a whole.

Because of the award, the author was invited to submit an extended version of the paper to RITA (VEDANA; MACHADO; MOREIRA, 2018). This extended version gave more details about V as a language, while still focusing heavily on the benefits of traits and its the record system.

REFERENCES

BACKUS, J. W. et al. Report on the algorithmic language algol 60. **Numerische Mathematik**, Springer, v. 2, n. 1, p. 106–136, 1960.

CZAPLICKI, E.; CHONG, S. Asynchronous functional reactive programming for guis. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 48, n. 6, p. 411–422, jun. 2013. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/2499370.2462161>.

FLOYD, R. W. Assigning meanings to programs. **Mathematical aspects of computer science**, v. 19, n. 19-32, p. 1, 1967.

GOSLING, J. et al. **The Java Language Specification, Java SE 8 Edition**. 1st. ed. [S.l.]: Addison-Wesley Professional, 2014. ISBN 013390069X, 9780133900699.

HASKELLWIKI. **Record Access**. 2018. <https://wiki.haskell.org/Record_access>. Accessed: 2018-08-06.

HENDERSON, P.; MORRIS JR., J. H. A lazy evaluator. In: **Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages**. New York, NY, USA: ACM, 1976. (POPL '76), p. 95–103. Disponível em: <http://doi.acm.org/10.1145/800168.811543>.

HINDLEY, R. The principal type-scheme of an object in combinatory logic. **Transactions of the american mathematical society**, JSTOR, v. 146, p. 29–60, 1969.

HUDAK, P.; FASEL, J. H. A gentle introduction to haskell. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 27, n. 5, p. 1–52, maio 1992. ISSN 0362-1340.

HUDAK, P. et al. A history of haskell: Being lazy with class. In: **Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages**. New York, NY, USA: ACM, 2007. (HOPL III), p. 12–1–12–55. ISBN 978-1-59593-766-7. Disponível em: <http://doi.acm.org/10.1145/1238844.1238856>.

IERUSALIMSCHY, R.; FIGUEIREDO, L. H. D.; FILHO, W. C. Lua—an extensible extension language. **Software: Practice and Experience**, Wiley Online Library, v. 26, n. 6, p. 635–652, 1996.

KAHN, G. Natural semantics. In: SPRINGER. **Annual Symposium on Theoretical Aspects of Computer Science**. [S.l.], 1987. p. 22–39.

KMETT, E. A. **lens: Lenses, Folds and Traversals**. 2012. <https://hackage.haskell.org/package/lens>. Accessed: 2018-05-07.

LIANG, S.; HUDAK, P.; JONES, M. Monad transformers and modular interpreters. In: **Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. New York, NY, USA: ACM, 1995. (POPL '95), p. 333–343. ISBN 0-89791-692-1. Disponível em: <http://doi.acm.org/10.1145/199448.199528>.

MARLOW, S. et al. Haskell 2010 language report. **Available online http://www.haskell. org/(May 2011)**, 2010.

MILNER, R. A theory of type polymorphism in programming. **Journal of computer and system sciences**, Elsevier, v. 17, n. 3, p. 348–375, 1978.

NIERSTRASZ, O.; DUCASSE, S.; POLLET, D. **Squeak by Example**. [S.l.]: Square Bracket Associates, 2009. ISBN 3952334103, 9783952334102.

O'CONNOR, R. Functor is to lens as applicative is to biplate: Introducing multiplate. **arXiv preprint arXiv:1103.2841**, 2011.

ODERSKY, M.; OTHERS. The Scala Language Specification. 2004.

PIERCE, B. C. **Types and Programming Languages**. 1st. ed. [S.l.]: The MIT Press, 2002. ISBN 0262162091, 9780262162098.

ROSSUM, G. V.; JR, F. L. D. **Python reference manual**. [S.l.]: Centrum voor Wiskunde en Informatica Amsterdam, 1995.

SABRY, A. What is a purely functional language? **Journal of Functional Programming**, Cambridge University Press, v. 8, n. 1, p. 1–22, 1998.

SABRY, A. What is a purely functional language? **Journal of Functional Programming**, Cambridge University Press, v. 8, n. 1, p. 1–22, 1998.

SIEK, J. **Type Safety in Three Easy Lemmas**. 2018. <http://siek.blogspot.com/2013/05/type-safety-in-three-easy-lemmas.html>. Accessed: 2018-09-26.

STACKOVERFLOW. **Stack Overflow 2018 Statistics**. 2018. <https://insights.stackoverflow.com/survey/2018/#top-paying-technologies>. Accessed: 2018-06-27.

STRACHEY, C. Fundamental concepts in programming languages. **Higher-order and symbolic computation**, Springer, v. 13, n. 1-2, p. 11–49, 2000.

SYME, D. et al. The f# 2.0 language specification. **Microsoft, August**, 2010.

TAFT, T. S.; DUFF, R. A. **Ada 95 reference manual. language and standard libraries: International standard ISO/IEC 8652: 1995 (E)**. [S.l.]: Springer Science & Business Media, 1997. v. 8652.

TOLKSDORF, S. **FParsec–a parser combinator library for F#**. 2013.

UNGAR, D. et al. Organizing programs without classes. **Lisp Symb. Comput.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 4, n. 3, p. 223–242, jul. 1991. ISSN 0892-4635.

VEDANA, A. G.; MACHADO, R.; MOREIRA, Á. F. V: a language with extensible record accessors and a trait-based type system. **Revista de Informática Teórica e Aplicada**, v. 25, n. 3, p. 89–101, 2018.

WADLER, P.; BLOTT, S. How to make ad-hoc polymorphism less ad hoc. In: **Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. New York, NY, USA: ACM, 1989. (POPL '89), p. 60–76. ISBN 0-89791-294-2.

APPENDIX A — ABSTRACT SYNTAX

A.1 Expressions

Programs in V are expressions. Each expression is a member of the abstract syntax tree defined below. The syntax tree will be constructed in parts, with an explanation of what each expression means and their uses. The full syntax tree can be obtained by simply joining all the separate sections.

Functions V, as a functional language, treats functions as first class constructions. This means that functions are expressions, and can be passed as arguments, bound to identifiers, etc. Below are the function expressions available in V, along with function application.

$$
\begin{array}{lcl}
e & ::= & \cdots \\
& | & func \\
& | & e_1\ e_2 \\
& | & x
\end{array}
$$

$$
\begin{array}{lcl}
func & ::= & \texttt{fn}\ x \Rightarrow e \\
& | & \texttt{rec}\ f\ x \Rightarrow e \\
& | & \texttt{rec}\ f : T\ \ x \Rightarrow e
\end{array}
$$

$$
\begin{array}{lcl}
x & ::= & \{x_0, x_1, \ldots\}
\end{array}
$$

All functions in V take exactly one parameter, and so function application evaluates the function by providing a single argument to it.

The first type of function ($\texttt{fn}\ x \Rightarrow e$) defines a simple unnamed function with a parameter x. x is an identifier from a set of name identifiers.

The body of the function is the expression e, which may or may not contain occurrences of the identifier passed as the argument of the function.

The other two types of functions are both variants of recursive functions available in V. These functions have a name, f, which is also a member of the set of name identifiers, and this name is used to allow recursive calls from within their body (e). Like unnamed functions, they take exactly one parameter, x, which may or may not be present in their bodies.

The difference between these variants is in their type declaration: the first variant is implicitly typed, while the second is explicitly typed. In the second variant, the programmer specifies the return type of the function as T (types will be shown later).

There are also expressions to use functions.

The first of them is function application: the first of the expressions is a function, and the second is the argument being passed to the function.

The last expression is simply to allow the use of the parameter defined in a function. An identifier x is only considered valid if it has been bound before (either by a function or, as will be seen later, by let declarations or `match` expressions).

Built-in Functions V has a few built-in functions that provide basic behavior.

These are:

$$
\begin{array}{lcll}
e & ::= & \cdots & \\
 & | & Builtin & \\
 & & & \\
Builtin & ::= & + & \text{(add, binary)} \\
 & | & - & \text{(subtract, binary)} \\
 & | & * & \text{(multiply, binary)} \\
 & | & \div & \text{(divide, binary)} \\
 & | & - & \text{(negate, unary)} \\
 & & & \\
 & | & < & \text{(less than, binary)} \\
 & | & \leq & \text{(less than or equal, binary)} \\
 & | & > & \text{(more than, binary)} \\
 & | & \geq & \text{(more than or equal, binary)} \\
 & | & = & \text{(equal, binary)} \\
 & | & \neq & \text{(not equal, binary)} \\
 & & & \\
 & | & \vee & \text{(Or, binary)} \\
 & | & \wedge & \text{(And, binary)} \\
 & & & \\
 & | & \text{get} & \text{(binary)} \\
 & | & \text{set} & \text{(ternary)} \\
 & | & \text{stack} & \text{(binary)}
\end{array}
$$

	distort	(ternary)

Every function has its arity declared. The arity of a function defines how many parameters it needs before it can be fully evaluated.

This behavior does not change the fact that functions receive only one parameter at a time. These particular cases can be thought of as nested functions, each taking a single parameter, until all the necessary parameters have been received.

This means that partially applied built-in functions are also treated as functions, and, therefore, can be passed as arguments, bound to identifiers, etc.

Most of these functions are basic and require no explanation. The last 4, however, will only be explained later, after records and accessors have been introduced.

The boolean functions $\vee$ and $\wedge$ are treated differently than others. Even though they are binary, they have a short-circuit mechanism. This means that, if the result of the application can be known by the first parameter (True in the case of $\vee$ or False in the case of $\wedge$), the second parameter is not evaluated. This is in contrast to all other functions in V, which evaluate all of their arguments before trying to evaluate themselves (This will be explained in more detail in Appendix B).

Constructors V has another type of special function: Data Constructors. Data Constructors are, as their name suggests, functions that construct values (data).

When fully evaluated, constructors define a form of structured data, storing the values passed to them as arguments. To access these values, it is possible to pattern match (see Appendix A.1) on the constructor name when fully evaluated.

Like built-in functions, constructors can be in a partially evaluated state. Partially evaluated constructors are treated as normal functions and, therefore, do not allow matching on their name.

e	::=	con	
con	::=	n	(arity 0)
	|	b	(arity 0)
	|	c	(arity 0)
	|	nil	(arity 0)
	|	::	(arity 2)
	|	Tuple n	(arity n, $n \geq 2$)

$$
\begin{array}{lll}
b & ::= & true \mid false \\
n & ::= & \mathbb{Z} \\
c & ::= & `char' \\
char & ::= & \texttt{ASCII characters}
\end{array}
$$

These functions are *extra* special, however, because they can have arity zero. This means that they "construct" values as soon as they are declared. The basic zero-arity constructors defined in the language are integers, booleans and characters.

The following two constructors (nil and ::) are related to lists. The first (with arity 0) is the empty list. The second constructor has arity 2, and extends a list (its second argument) by adding a new value (its first argument) to its head.

The last constructor is actually a family of constructors describing tuples. A constructor Tuple n defines a constructor that has n parameters and evaluates to a tuple with n elements. It is also important to note that $n \geq 2$. This means that only tuple constructors with 2 or more parameters are valid.

Records and Accessors The record system in V is composed of two parts: records and accessors.

Records are a type of structured data composed of associations between labels and values called fields. Each label is part of an ordered set of labels l, and can only appear once in every record. The order in which the labels are declared in a record is not important, as any operation that is done on a record first orders the fields according to the order int the set of labels.

Accessors are terms that allow access to fields within a record. Accessors view records as trees, where each non-leaf node is a record, and each edge has a name (the label of the field). Accessors, then, define a path on this tree, extracting the node at the end of the path.

$$
\begin{array}{lll}
e & ::= & \cdots \\
 & \mid & \{l_1 : e_1, \ldots l_n : e_n\} \quad (n \geq 1) \\
 & \mid & \#l \\
 & \mid & \#(e_1, \ldots e_n) \quad (n \geq 2) \\
 & & \\
l & ::= & \{l_1, l_2, ...\}
\end{array}
$$

The most basic type of path is a simple label $\#l$. An accessor can also be made by combining multiple accessors. The $\#(e_1, \ldots e_n)$ expression creates an accessor for multiple fields of the same record. Another type of composition is vertical, and is obtained by using the built-in function "stack".

Furthermore, paths can be distorted with the "distort" built-in function, specifying a pair of functions to be applied when extracting or updating values in the field.

More details about how accessors (and paths) work will be provided in Appendix B.

Let and Patterns The `let` expression is used to bind values to identifiers, allowing them to be reused in a sub-expression. A `let` expression is divided into 2 parts: the binding and the sub-expression. The binding itself also has 2 parts: the left-hand side, which is a pattern; and the right-hand side, which will be the value to be bound.

Patterns are used to "unpack" values, and can be either explicitly or implicitly typed.

$$
\begin{array}{lcll}
e & ::= & \cdots & \\
 & | & \texttt{let}\ p = e_1\ \texttt{in}\ e_2 & \\
 & & & \\
p & ::= & patt & \\
 & | & patt : T & \\
 & & & \\
patt & ::= & x & \\
 & | & \_ & \\
 & | & con\ p_1, \ldots p_n & (\text{constructor pattern}, n = \text{arity } con) \\
 & | & \{l_1 : p_1, \ldots l_n : p_n\} & (n \geq 1) \\
 & | & \{l_1 : p_1, \ldots l_n : p_n, \ldots\} & (\text{partial record}, n \geq 1)
\end{array}
$$

Much like functions, `let` expressions allow the use of identifiers in an expression by attaching values to the identifiers. Differently from functions, however, a single `let` expression can bind multiple identifiers to values by using patterns. Patterns are matched against the values in the right-hand side of the binding, and can, depending on their structure, create any number of identifier bindings.

The pattern x is a simple identifier pattern. It matches any value in the right-hand side, and binds it to the identifier.

The _ also matches any value, but it does not create any new bindings (this is called the ignore pattern).

The con pattern matches a completely applied constructor. This pattern is a compound pattern, with the same number of components as the arity of the constructor. The con pattern itself does not create any bindings, but its components might, since they are themselves patterns and, as such, will be matched against the components of the constructor.

The next pattern is a record pattern. This matches a record with *exactly* the same fields as the pattern. Since all labels in a record are ordered, the fields do not need to be reordered for the matching.

For matching any record with *at least* the fields $l_1, \ldots l_n$, one can use the pattern $\{l_1 : p_1, \ldots l_n : p_n, \ldots\}$. This pattern will match any record whose set of labels is a superset of $l_1, \ldots l_n$.

Match Expression A `match` expression attempts to match a value against a list of patterns. Every pattern is paired with a resulting expression to be evaluated if the pattern matches. Furthermore, it is possible to specify a boolean condition to be tested alongside the pattern matching. This condition will only be tested if the match succeeds, so it can use any identifier bound by the pattern. The matching stops at the first pattern that successfully matches (and any condition is satisfied), and its paired expression is then evaluated.

$$
\begin{array}{lcll}
e & ::= & \cdots & \\
 & | & \texttt{match}\ e\ \texttt{with}\ match_1, \ldots match_n & (n \geq 1) \\
 & & & \\
match & ::= & p \rightarrow e & \\
 & | & p\ \texttt{when}\ e_1 \rightarrow e_2 &
\end{array}
$$

IO The language supports simple input and output through the `read` and `write` built-in functions. These functions operate on values constructed on the `IO` constructor, and a new empty value (`Void`) is introduced to represent the result of evaluating the `write` function.

$$
\begin{array}{lcll}
\mathit{Builtin} & ::= & \cdots & \\
& | & \text{read} & (\text{unary}) \\
& | & \text{write} & (\text{unary})
\end{array}
$$

$$
\begin{array}{lcll}
\mathit{con} & ::= & \cdots & \\
& | & \text{IO} & (\text{arity } 1) \\
& | & \text{Void} & (\text{arity } 0)
\end{array}
$$

Both `IO` functions operate on single characters; `read` receives one character from the standard system IO (console), while `write` prints a single character to the system IO.

To allow composition of these functions, two monadic operations were introduced to the language: `return` and `bind`.

$$
\begin{array}{lcll}
\mathit{Builtin} & ::= & \cdots & \\
& | & \text{return} & (\text{unary}) \\
& | & \text{bind} & (\text{binary})
\end{array}
$$

`return` is a function that takes any value and returns that value encapsulated in the `IO` constructor. `bind` takes an `IO` value and a function, returning the result of applying the function to the value that was encapsulated in the `IO`.

Exceptions This expression always evaluates to a runtime error.

$$
\begin{array}{lcl}
e & ::= & \cdots \\
& | & \mathit{raise}
\end{array}
$$

Runtime errors usually happen when an expression cannot be correctly evaluated, such as division by zero, accessing an empty list, etc.

Sometimes, however, it can be necessary to directly cause an error. The raise expression serves this purpose.

A.2 Types

Since V is strongly typed, every (valid) expression has exactly one type associated with it. Some expressions allow the programmer to explicitly declare types, such as patterns and recursive functions. Other expressions, such as $e_1 = e_2$, or even constants, such as 1 or true, have types implicitly associated with them. These types are used by the type system (see Appendix C) to check whether an expression is valid or not, avoiding run-time errors that can be detected at compile time.

Types Below are all the types available in V. The first type is a fully applied constructor type. The second type is a function type. The third type is a record type and, finally, the last type is an accessor type.

$$
\begin{array}{rcll}
T & ::= & \cdots & \\
& | & conT\ T_1,\ \ldots\ T_n & (n = \text{arity}\ conT) \\
& | & T_1 \to T_2 & \\
& | & \{l_1 : T_1,\ \ldots\ l_n : T_n\} & (n \geq 1) \\
& | & T_1 \# T_2 & Accessor
\end{array}
$$

Most of the types are compound types, and the only scenario in which a type is not compound is for constructor types with arity 0. Function types specify the type of the single parameter (T_1) and the type of output (T_2).

Record types are also compound types, but they associate every component to its corresponding label. Just like record expressions, the fields are ordered according to the order of the labels.

Finally, accessor types define the types of accessor expressions. They have two components: T_1, which is the type of the record being accessed; and T_2, the type of the value being accessed. It is read as T_1 *accesses* T_2.

Constructor Types These are types associated with constructors. Much like constructors, they can take any amount of arguments to be fully applied, and the number of arguments they take is described by their arity. Instead of taking values as arguments, however, constructor types take types as arguments.

$$
\begin{array}{rcll}
conT & ::= & \text{Int} & (\text{arity } 0) \\
& | & \text{Bool} & (\text{arity } 0) \\
& | & \text{Char} & (\text{arity } 0) \\
& | & \text{List} & (\text{arity } 1) \\
& | & \text{Tuple}_T & (\text{arity } n,\ n \geq 2) \\
& | & \text{IO}_T & (\text{arity } 1) \\
& | & \text{Void}_T & (\text{arity } 0)
\end{array}
$$

Variable Types These types represent an unknown constant type. Explicitly typed expressions cannot be given variable types, but they are used by the type system for implicitly typed expressions. In the course of the type inference, the type system can replace variable types for their type.

It is important to realize that variable types already represent a unique type with

an unknown identity. This means that a variable type may only be replaced by the specific type which it represents and not any other type. This distinction becomes important when talking about polymorphism, which uses variable types, along with universal quantifiers, to represent a placeholder for any possible type.

$$\begin{aligned} T \quad &::= \quad \cdots \\ &\quad | \quad X^{Traits} \end{aligned}$$

$$X \quad ::= \quad X_1, X_2, ...$$

A.3 Traits

Types can conform to traits, which define certain behaviors that are expected of said type. Regular types always have their trait information implicitly defined, since this information is included in the language. Variable types, on the other hand, can explicitly state which traits they possess, restricting the set of possible types they can represent (this is represented by the superscript $Traits$ in a variable type X).

$$\begin{aligned} Traits \quad &::= \quad \varnothing \\ &\quad | \quad \{Trait\} \cup Traits \end{aligned}$$

$$\begin{aligned} Trait \quad &::= \quad Equatable \\ &\quad | \quad Orderable \\ &\quad | \quad \{l : Type\} \quad \text{(Record Label)} \end{aligned}$$

The information on which types conform to which traits is defined in the unification environment (see Appendix C.2). When a type T conforms to a trait $Trait$, the notation used is $T \in Trait$. The same notation can be used to describe when a type conforms to a set of traits $Traits$ (i.e. $T \in Traits$).

By default, the following rules hold for conformance:

Equatable If a type T is $Equatable$, expressions of type T can use the equality operators ($=, \neq$).

To define the set of types that belong to $Equatable$, the following rules are used:

$$\{\text{Int}, \text{Bool}, \text{Char}\} \subset Equatable$$

$$T \in Equatable \implies List\ T \in Equatable$$

$$X^{Traits} \in Equatable \implies Equatable \in Traits$$

Orderable If a type T is $Orderable$, expressions of type T can use the inequality operators ($<, \leq, >, \geq$). Any type that is $Orderable$ is also $Equatable$.

To define the set of types that belong to $Orderable$, the following rules are used:

$$\{\text{Int}, \text{Char}\} \subset Orderable$$
$$T \in Orderable \implies List\, T \in Orderable$$
$$X^{Traits} \in Orderable \implies Orderable \in Traits$$

Record Label A type T_1 conforms to a Record Label Trait $\{l : T_2\}$ if it is a record that contains a field with the label l and the type T_2.

If the type conforms to the trait $\{l : T_2\}$, it can then use the accessor $\#l$. This ensures that accessor for a field can only be used on records that have that field.

To define the set of types that belong to a record label $\{l : T\}$, the following rules are used:

$$\{l_1 : T_1, \ldots\, l_n : T_n, \ldots\, T_k\} \in \{l : T\} \iff l_n = l \wedge T_n = T \quad (1 \leq n \leq k)$$
$$X^{Traits} \in \{l : T\} \implies \{l : T\} \in Traits$$

APPENDIX B — OPERATIONAL SEMANTICS

The V language is evaluated using a big-step evaluation with environments. This evaluation reduces an expression into a value directly, not necessarily having a rule of evaluation for every possible expression. To stop programmers from creating programs that cannot be evaluated, a type inference system will be specified later.

Value A value is the result of the evaluation of an expression in big-step. This set of values is different from the set of expressions of V, even though they share many similarities.

$$
\begin{array}{lcll}
v & ::= & con\ v_1,\ \ldots\ v_n & (n = \text{arity } con) \\
& | & \{l_1 : v_1,\ \ldots\ l_n : v_n\} & (n \geq 1) \\
& | & \#path & \\
& | & \langle func, env \rangle & \\
& | & \ll Builtin\ .\ v_1,\ \ldots v_n \gg & (n < \text{arity } Builtin) \\
& | & \ll con\ .\ v_1,\ \ldots v_n \gg & (n < \text{arity } con)
\end{array}
$$

$$
\begin{array}{lcll}
path & ::= & l & \\
& | & path\ .\ path & \\
& | & (path_1,\ \ldots\ path_n) & (n \geq 2) \\
& | & path\ [v_1,\ v_2] &
\end{array}
$$

The value $\langle func, env \rangle$ defines closures. They represent the result of evaluating functions (and recursive functions) and store the environment at the moment of evaluation. This means that V has static scope, since closures capture the environment at the moment of evaluation and V has eager evaluation.

The values $\ll Builtin\ .\ v_1,\ \ldots v_n \gg$ and $\ll con\ .\ v_1,\ \ldots v_n \gg$ are partial applications of built-in functions and constructors, respectively.

Once all the parameters have been defined, they evaluate either to the result of the function (in the case of $Builtin$) or to a fully applied constructor $con\ v_1,\ \ldots\ v_n$.

Environment An evaluation environment is a 2-tuple which contains the following information:

1. Arity of constructors

   If a constructor has arity n, it requires n arguments to become fully evaluated.

2. Associations between identifiers and values

A new association is created every time a value is bound. This happens in `let` declarations, function application and `match` expressions.

$$
\begin{array}{lcll}
env & ::= & (arities, vars) & \\
 & & & \\
arities & ::= & \{\} \mid \{con \mapsto n\} \cup arities & (n \in \mathbb{N}) \\
vars & ::= & \{\} \mid \{x \mapsto v\} \cup vars &
\end{array}
$$

B.1 Paths

Accessors possess structure when treated as values. This structure is built through use of the operations available on accessors, such as the compose expression or the built-in functions. Since accessors view records as trees, this structure is a *path* along a tree.

The most basic structure of a path is a single label. This path describes a field immediately below the root of the tree (the root is viewed as the record itself, and every child is a field of the record). This path is created by using the simple accessor expression $\#l$.

Two paths can be composed vertically, allowing access to subfields of a record. In this scenario, the tree must have at least the same depth as the path (and along the correct field names). Vertical composition is achieved by using the "stack" built-in function, and always combines two paths.

Paths can also be composed horizontally, described as a tuple of paths. When composed horizontally, all paths are used on the root of the record, and the end points of the paths are joined in a tuple for extraction or updating.

Finally, paths can be distorted. This means that the path has two values (functions) associated with it. These functions are then used to transform the values stored in the field of the record (one for extraction, another for updating).

Path Traversal Rules As previously stated, accessors describe paths along a record tree. To use these paths, an auxiliary $traverse$ function is used. This functions receives 3 arguments: a path, a record and an update value. The function returns 2 values: the old value associated with the field specified by the path; and an updated record.

This updated record uses the value provided as input to the function to update the field specified by the path. This last argument of the $traverse$ function can be omitted and, in such a case, no update is done (that is, the updated record is the same as the input

record).

The first rule is for a simple label path. The label must be present in the provided record. A new record is created, associating the provided value with the label specified by the path.

$$\frac{1 \leq \|k\| \leq \|n\| \qquad r = \{l_1 : v_1,\ \ldots\ l_k : v,\ \ldots\ l_n : v_n\}}{traverse(l_k, \{l_1 : v_1, ...l_n : v_n\}, v) = v_k, r}$$

For vertically composed paths, three calls to $traverse$ are needed.

$$\frac{traverse(path_1, \{l_1 : v_1, ...l_n : v_n\}) = rec, r \qquad traverse(path_2, rec, v) = v', rec' \qquad traverse(path_1, \{l_1 : v_1, ...l_n : v_n\}, rec') = rec, r'}{traverse(path_1\ .\ path_2, \{l_1 : v_1, ...l_n : v_n\}, v) = v', r'}$$

The first call omits the update value, and is used to extract a record associated with the first component of the path. This record is then passed, along with the second component of the path and the update value, to the second call of $traverse$. Finally, the third call uses the return of the second call to update the internal record, returning a new updated record.

Joined paths also require multiple calls to $traverse$, but the exact number depends on the amount of paths joined.

$$\frac{r_0 = \{l_1 : v_1, ...l_n : v_n\} \qquad \forall i \in [1,\ n]\ .\ traverse(path_i, r_{i-1}, v_i) = v'_i, r_i}{traverse((path_1,\ \ldots\ path_n), \{l_1 : v_1, ...l_n : v_n\}, (v_1,\ \ldots\ v_n)) = (v'_1,\ \ldots\ v'_n), r_n}$$

Pairing the paths with the components of the tuple provided as the update value, each pair is passed as input to a call to $traverse$. This happens from left to right, and each call returns a part of the old value and a partially updated record. Every call uses the partially updated record provided, and the last call to $traverse$ returns the fully updated record.

Distorted paths require two calls to $traverse$, one before and one after applying the distortions.

$$\frac{\begin{array}{c} traverse(path, \{l_1 : v_1, ...l_n : v_n\}) = v_{old}, r \\ \{\} \vdash v_2\ v\ v_{old} \Downarrow v' \qquad \{\} \vdash v_1\ v_{old} \Downarrow v'_{old} \\ traverse(path, \{l_1 : v_1, ...l_n : v_n\}, v') = v_{old}, r \end{array}}{traverse(path\ [v_1,\ v_2], \{l_1 : v_1, ...l_n : v_n\}, v) = v'_{old}, r}$$

First, the current value of the field is extracted. This value is then passed to the first component (v_1) of the accessor, returning the distorted current value. Then, the new distorted value v, along with the current value of the field, is passed to the second component (v_2) of the distorted accessor. This value is then provided as the new update value for a call to $traverse$, returning the updated record.

B.2 Pattern Matching

For `let` and `match` expressions, it is necessary to match a pattern p to a value v. This process, if successful, creates a mapping of identifiers to their corresponding elements of v. If v does not match the pattern p, the process fails.

In the case of a `let` expression, failing to match means the whole expression evaluates to $raise$. For `match` expressions, a failed pattern causes the next pattern to be attempted. If there are no more patterns, the expression evaluates to $raise$.

To aid in this matching, a auxiliary "match" function is defined. The function takes a pattern p and a value v, returning a mapping of identifiers to values (the $vars$ of an environment). If the matching fails, the function will return nothing.

The following are the rules for the match function:

$$match(x, v) = \{x \mapsto v\}$$

$$match(\_, v) = \{\}$$

$$\frac{con_1 = con_2 \qquad \forall\, i \in [1, n]\ \ match(p_i, v_i) = vars_i}{match(con_1\ v_1,\ \ldots v_n, con_2\ p_1,\ \ldots p_n) = \bigcup_{i=1}^{n} vars_i}$$

$$\frac{k \geq n \qquad \forall\, i \in [1, n]\ \ \exists\, j \in [1, k]\ \ l_i^1 = l_j^2 \wedge match(p_i, v_j) = vars_i}{match(\{l_1^1 : p_1, \ldots, l_n^1 : p_n, \ldots\}, \{l_1^2 : v_1, ...l_k^2 : v_k\}) = \bigcup_{i=1}^{n} vars_i}$$

$$\frac{\forall\, i \in [1,n]\ \ l_i^1 = l_i^2 \wedge match(p_i, v_i) = vars_i}{match(\{l_1^1 : p_1, \ldots, l_n^1 : p_n\}, \{l_1^2 : v_1, \ldots l_n^2 : v_n\}) = \bigcup_{i=1}^{n} vars_i}$$

Any other inputs provided to the match function will result in a failed matching. This is represented by:

$$\neg\, match(p, v)$$

B.3 Big-Step Rules

Function Expressions Every function evaluates to a closure. This basically stores the function definition and the current environment in a value, allowing the evaluation environment to be restored on function application.

$$\text{env} \vdash func \Downarrow \langle func, env \rangle \tag{BS-Fn}$$

Built-in functions evaluate to a partially applied built-in without any arguments.

$$\text{env} \vdash Builtin \Downarrow \ll Builtin\ .\ \gg \tag{BS-Builtin}$$

Similarly, constructors, if they need at least one argument, evaluate to a partially applied constructor. If, however, they do not take any arguments, they immediately evaluate to a fully applied constructor.

$$\frac{\text{env.arities}(con) > 0}{\text{env} \vdash con \Downarrow \ll con\ .\ \gg} \tag{BS-Con}$$

$$\frac{\text{env.arities}(con) = 0}{\text{env} \vdash con \Downarrow con} \tag{BS-Con0}$$

Application An application expression requires either a closure, a partially applied built-in function or a partially applied constructor for its left-hand operand.

In the case of a closure, there are two different behaviors, depending on whether the function is recursive or not.

When applying non recursive functions, a new association between the parameter identifier (x) and the argument (v_2) is added to the environment stored in the closure (env_1). The body of the function (e) is then evaluated using this new environment.

$$\frac{\text{env} \vdash e_1 \Downarrow \langle \texttt{fn}\ x \Rightarrow e, \text{env}_1 \rangle \qquad \text{env} \vdash e_2 \Downarrow v_2 \qquad \{x \mapsto v_2\} \cup \text{env}_1 \vdash e \Downarrow v}{\text{env} \vdash e_1\ e_2 \Downarrow v} \quad \text{(BS-AppFn)}$$

Recursive functions, besides associating the identifier to the argument, also create an association between the function name and its value (i.e the closure itself). This allows the body of the function to call itself, creating a recursive structure.

For operational semantics, there is no difference between the typed and untyped versions of recursive functions, so both have the same evaluation rules.

$$\frac{\text{env} \vdash e_1 \Downarrow \langle \texttt{rec}\ f\ x \Rightarrow e, \text{env}_1 \rangle \qquad \text{env} \vdash e_2 \Downarrow v_2 \qquad \{x \mapsto v_2, f \mapsto \langle \texttt{rec}\ f\ x \Rightarrow e, \text{env}_1 \rangle\} \cup \text{env}_1 \vdash e \Downarrow v}{\text{env} \vdash e_1\ e_2 \Downarrow v} \quad \text{(BS-AppFnRec)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \langle \texttt{rec}\ f : T\ x \Rightarrow e, \text{env}_1 \rangle \qquad \text{env} \vdash e_2 \Downarrow v_2 \qquad \{x \mapsto v_2, f \mapsto \langle \texttt{rec}\ f : T\ x \Rightarrow e, \text{env}_1 \rangle\} \cup \text{env}_1 \vdash e \Downarrow v}{\text{env} \vdash e_1\ e_2 \Downarrow v} \quad \text{(BS-AppFnRec2)}$$

Application on partially applied constructors can behave in two different ways, depending on how many arguments have been already applied.

If the arity of the constructor is larger than the number of arguments already applied (plus the new one being applied), the result of the application is a partially applied constructor with the new value added to the end.

$$\frac{\text{env} \vdash e_1 \Downarrow \ll con\ .\ v_1,\ \ldots v_n \gg \qquad n+1 < \text{env.arities}(con) \qquad \text{env} \vdash e_2 \Downarrow v}{\text{env} \vdash e_1\ e_2 \Downarrow \ll con\ .\ v_1,\ \ldots v_n,\ v \gg} \quad \text{(BS-AppCon)}$$

If the arity of the constructor is equal to 1 more than the number of already applied arguments, the application results in a completely applied constructor.

$$\frac{\text{env} \vdash e_1 \Downarrow \ll con\ .\ v_1,\ \ldots v_n \gg \qquad n+1 = \text{env.arities}(con) \qquad \text{env} \vdash e_2 \Downarrow v}{\text{env} \vdash e_1\ e_2 \Downarrow con\ v_1,\ \ldots v_n,\ v} \quad \text{(BS-AppConTotal)}$$

Application on partially applied built-in functions works similarly, having different rules depending on the number of arguments.

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \ll Builtin\ .\ v_1,\ \ldots v_n \gg \\ n+1 < \text{arity}\ Builtin \qquad \text{env} \vdash e_2 \Downarrow v\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow \ll Builtin\ .\ v_1,\ \ldots v_n,\ v \gg} \quad \text{(BS-AppBuiltin)}$$

The result of applying the last argument of a built-in function varies depending on what the function does (and what kind of arguments it accepts). These rules will be provided later.

Application propagates exceptions ($raise$). If the first sub-expression of an application evaluates to $raise$, the whole expression evaluates to $raise$. This is true for the second expression in most scenarios, but there are a couple of exceptions (see B.3) that do not necessarily evaluate this sub-expression for complete evaluation.

$$\frac{\text{env} \vdash e_1 \Downarrow raise}{\text{env} \vdash e_1\ e_2 \Downarrow raise} \quad \text{(BS-AppRaise)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v \qquad \text{env} \vdash e_2 \Downarrow raise}{\text{env} \vdash e_1\ e_2 \Downarrow raise} \quad \text{(BS-AppRaise2)}$$

Identifier The evaluation of an identifier depends on the environment in which it is evaluated. If the environment has an association between the identifier and a value, the value is returned. If it does not, the program is malformed and cannot be evaluated (this will be caught in the type system).

$$\frac{\text{env.vars}(x) = v}{\text{env} \vdash x \Downarrow v} \quad \text{(BS-Ident)}$$

Records A record construction expression $\{l_1 : e_1,\ \ldots\ l_n : e_n\}$ evaluates each of its sub-expressions individually, resulting in a record value. The order of evaluation is defined by the order of the labels and is done from smallest to largest.

$$\frac{\forall\, k \in [1,n] \ \ \text{env} \vdash e_k \Downarrow v_k}{\text{env} \vdash \{l_1 : e_1,\ \ldots\ l_n : e_n\} \Downarrow \{l_1 : v_1,\ \ldots\ l_n : v_n\}} \quad \text{(BS-Record)}$$

If any of the sub-expressions evaluate to raise, the whole record also evaluates to raise.

$$\frac{\exists\, k \in [1,n] \ \ \text{env} \vdash e_k \Downarrow raise}{\text{env} \vdash \{l_1 : e_1,\ \ldots\ l_n : e_n\} \Downarrow raise} \quad \text{(BS-RecordRaise)}$$

Accessors There is a different evaluation rule for each type of path available to accessors.

The simplest rule is for a label accessor, which is in itself a value.

$$\text{env} \vdash \#l \Downarrow \#l \qquad \text{(BS-Label)}$$

Joined accessors evaluate each of their sub-expressions, expecting an accessor value as a result.

$$\frac{\forall\, k \in [1,n] \;\; \text{env} \vdash e_k \Downarrow \#path_k}{\text{env} \vdash \#(e_1,\, \ldots\, e_n) \Downarrow \#(path_1,\, \ldots\, path_n)} \qquad \text{(BS-Joined)}$$

To create a stacked accessor, the built-in function "stack" must be used. This function has arity 2, and requires both arguments to be accessors. The paths of the accessors are then composed in a stacked accessor, which is the result of the evaluation.

$$\frac{\text{env} \vdash e_1 \Downarrow \ll \text{stack} \,.\, \#path_1 \gg \qquad \text{env} \vdash e_2 \Downarrow \#path_2}{\text{env} \vdash e_1\; e_2 \Downarrow \#path_1 \,.\, path_2} \qquad \text{(BS-Stacked)}$$

Similarly, creating distorted accessors requires the built-in function "distort". This function takes 3 arguments, the first being an accessor, and the remaining two being functions. When fully evaluated, the path of the accessor is combined with the function values, creating a distorted accessor.

$$\frac{\text{env} \vdash e_1 \Downarrow \ll \text{distort} \,.\, \#path,\, v_1 \gg \qquad \text{env} \vdash e_2 \Downarrow v_2}{\text{env} \vdash e_1\; e_2 \Downarrow \#path\; [v_1,\, v_2]} \qquad \text{(BS-Distorted)}$$

Using Accessors There are two built-in functions that take accessors as arguments.

Get takes 2 arguments: an accessor and a record. The $traverse$ function is then called with the accessor's path and the record (the third argument is omitted), and the first return (i.e. the value associated with the path) is used as the result of the evaluation.

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \ll \text{get} \,.\, \#path \gg \qquad \text{env} \vdash e_2 \Downarrow \{l_1 : v_1,\, \ldots\, l_n : v_n\} \\ traverse(path, \{l_1 : v_1,\, \ldots\, l_n : v_n\}) = v', r'\end{array}}{\text{env} \vdash e_1\; e_2 \Downarrow v'} \qquad \text{(BS-Get)}$$

Set takes 3 arguments: an accessor, a generic value and a record. The $traverse$ function is then called with the arguments, using the generic value as the update value of the call. The result of the evaluation is the second return of the $traverse$ function (i.e. the updated record).

$$\frac{\text{env} \vdash e_1 \Downarrow \ll \text{set} \,.\, \#path,\, v \gg \qquad \text{env} \vdash e_2 \Downarrow \{l_1 : v_1,\, \ldots\, l_n : v_n\} \qquad traverse(path, \{l_1 : v_1,\, \ldots\, l_n : v_n\}, v) = v', r'}{\text{env} \vdash e_1\ e_2 \Downarrow r'} \text{ (BS-Set)}$$

Numerical Operations The V language only supports integers, so all operations are done on integer numbers. This means that the division always results in a whole number, truncated towards zero.

$$\frac{\text{env} \vdash e_1 \Downarrow \ll + \,.\, n_1 \gg \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n\| = \|n_1\| + \|n_2\|}{\text{env} \vdash e_1\ e_2 \Downarrow n} \text{ (BS-+)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \ll - \,.\, n_1 \gg \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n\| = \|n_1\| - \|n_2\|}{\text{env} \vdash e_1\ e_2 \Downarrow n} \text{ (BS–)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \ll - \,.\, \gg \qquad \text{env} \vdash e_2 \Downarrow n_1 \qquad \|n\| = -\|n_1\|}{\text{env} \vdash e_1\ e_2 \Downarrow n} \text{ (BS– (unary))}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \ll * \,.\, n_1 \gg \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n\| = \|n_1\| * \|n_2\|}{\text{env} \vdash e_1\ e_2 \Downarrow n} \text{ (BS-*)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \ll \div \,.\, n_1 \gg \qquad \text{env} \vdash e_2 \Downarrow 0}{\text{env} \vdash e_1\ e_2 \Downarrow raise} \text{ (BS-÷Zero)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \ll \div \,.\, n_1 \gg \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n_2\| \neq 0 \qquad \|n\| = \|n_1\| \div \|n_2\|}{\text{env} \vdash e_1\ e_2 \Downarrow n} \text{ (BS-÷)}$$

Equality Operations The equality operators ($=$ and $\neq$) test the equality of fully applied constructors and records.

$$\frac{\text{env} \vdash e_1 \Downarrow \ll = \,.\, (con\ v_1^1,\, \ldots v_n^1) \gg \qquad \text{env} \vdash e_2 \Downarrow (con\ v_1^2,\, \ldots v_n^2) \qquad \forall\, k \in [1,n] \;\; \text{env} \vdash (=\ v_k^1)\ v_k^2 \Downarrow true}{\text{env} \vdash e_1\ e_2 \Downarrow true} \text{ (BS-=ConTrue)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \ll = \,.\, (con\ v_1^1,\, \ldots v_n^1) \gg \qquad \text{env} \vdash e_2 \Downarrow (con\ v_1^2,\, \ldots v_n^2) \qquad \exists\, k \in [1,n] \;\; \text{env} \vdash (=\ v_k^1)\ v_k^2 \Downarrow false}{\text{env} \vdash e_1\ e_2 \Downarrow false} \text{ (BS-=ConFalse)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \ll= \ .\ (con\ v_1^1,\ \ldots v_n^1) \gg \qquad \text{env} \vdash e_2 \Downarrow (con'\ v_1^2,\ \ldots v_k^2) \qquad con' \neq con}{\text{env} \vdash e_1\ e_2 \Downarrow false}$$

(BS-=ConFalse2)

$$\frac{\text{env} \vdash e_1 \Downarrow \ll= \ .\ \{l_1^1 : v_1^1,\ \ldots\ l_n^1 : v_n^1\} \gg \qquad \text{env} \vdash e_2 \Downarrow \{l_1^2 : v_1^2,\ \ldots\ l_n^2 : v_n^2\} \qquad \forall\, k \in [1,n]\ \ l_k^1 = l_k^2 \wedge \text{env} \vdash (=\ v_k^1)\ v_k^2 \Downarrow true}{\text{env} \vdash e_1\ e_2 \Downarrow true}$$

(BS-=RecordTrue)

$$\frac{\text{env} \vdash e_1 \Downarrow \ll= \ .\ \{l_1^1 : v_1^1,\ \ldots\ l_n^1 : v_n^1\} \gg \qquad \text{env} \vdash e_2 \Downarrow \{l_1^2 : v_1^2,\ \ldots\ l_n^2 : v_n^2\} \qquad \exists\, k \in [1,n]\ \ l_k^1 = l_k^2 \wedge \text{env} \vdash (=\ v_k^1)\ v_k^2 \Downarrow false \qquad \forall\, j \in [1,k)\ \ \text{env} \vdash v_j^1 = v_j^2 \Downarrow true}{\text{env} \vdash e_1\ e_2 \Downarrow false}$$

(BS-=RecordFalse)

$$\frac{\text{env} \vdash e_1 \Downarrow \ll \neq \ .\ v_1 \gg \qquad \text{env} \vdash e_2 \Downarrow v_2 \qquad \text{env} \vdash (=\ v_1)\ v_2 \Downarrow false}{\text{env} \vdash e_1\ e_2 \Downarrow true}$$

(BS-≠True)

$$\frac{\text{env} \vdash e_1 \Downarrow \ll \neq \ .\ v_1 \gg \qquad \text{env} \vdash e_2 \Downarrow v_2 \qquad \text{env} \vdash (=\ v_1)\ v_2 \Downarrow true}{\text{env} \vdash e_1\ e_2 \Downarrow false}$$

(BS-≠False)

Inequality Operations The inequality operators function much in the same way as the equality operators. The only difference is that they do not allow comparison of certain kinds of expressions (such as booleans) when such expressions do not have a clear ordering to them.

To reduce the number of rules, some rules are condensed for all inequality operators ($<, \leq, >, \geq$). The comparison done on numbers is the ordinary numerical comparison. For characters, the ASCII values are compared numerically.

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n_1\|\, opIneq\, \|n_2\|}{\text{env} \vdash e_1\ opIneq\ e_2 \Downarrow true}$$

(BS-IneqNumTrue)

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \neg \, \|n_1\| \, opIneq \, \|n_2\|}{\text{env} \vdash e_1 \; opIneq \; e_2 \Downarrow true} \text{ (BS-IneqNumFalse)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow c_1 \qquad \text{env} \vdash e_2 \Downarrow c_2 \qquad \|c_1\| \, opIneq \, \|c_2\|}{\text{env} \vdash e_1 \; opIneq \; e_2 \Downarrow true} \text{ (BS-IneqCharTrue)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow c_1 \qquad \text{env} \vdash e_2 \Downarrow c_2 \qquad \neg \, \|c_1\| \, opIneq \, \|c_2\|}{\text{env} \vdash e_1 \; opIneq \; e_2 \Downarrow true} \text{ (BS-IneqCharFalse)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 < e_2 \Downarrow false} \text{ (BS-<Nil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 \leq e_2 \Downarrow true} \text{ (BS-$\leq$Nil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 > e_2 \Downarrow false} \text{ (BS->Nil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 \geq e_2 \Downarrow true} \text{ (BS-$\geq$Nil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 < e_2 \Downarrow false} \text{ (BS-<ListNil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 \leq e_2 \Downarrow false} \text{ (BS-$\leq$ListNil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 > e_2 \Downarrow true} \text{ (BS->ListNil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 \geq e_2 \Downarrow true} \text{ (BS-$\geq$ListNil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow v_1 :: v_2}{\text{env} \vdash e_1 < e_2 \Downarrow true} \text{ (BS-<NilList)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow v_1 :: v_2}{\text{env} \vdash e_1 \leq e_2 \Downarrow true} \quad \text{(BS-}\leq\text{NilList)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow v_1 :: v_2}{\text{env} \vdash e_1 > e_2 \Downarrow false} \quad \text{(BS->NilList)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow v_1 :: v_2}{\text{env} \vdash e_1 \geq e_2 \Downarrow false} \quad \text{(BS-}\geq\text{NilList)}$$

$$\frac{\begin{array}{cc}\text{env} \vdash e_1 \Downarrow v_1 :: v_2 & \text{env} \vdash e_2 \Downarrow v_3 :: v_4 \\ \text{env} \vdash v_1 = v_3 \Downarrow false & \text{env} \vdash v_1 \; opIneq \; v_3 \Downarrow b\end{array}}{\text{env} \vdash e_1 \; opIneq \; e_2 \Downarrow b} \quad \text{(BS-IneqListHead)}$$

$$\frac{\begin{array}{cc}\text{env} \vdash e_1 \Downarrow v_1 :: v_2 & \text{env} \vdash e_2 \Downarrow v_3 :: v_4 \\ \text{env} \vdash v_1 = v_3 \Downarrow true & \text{env} \vdash v_2 \; opIneq \; v_4 \Downarrow b\end{array}}{\text{env} \vdash e_1 \; opIneq \; e_2 \Downarrow b} \quad \text{(BS-IneqListTail)}$$

Boolean Operations The built-in functions $\vee$ (OR) and $\wedge$ (AND) are treated differently from all other functions in V. They are binary functions, but they only evaluate their second argument if strictly necessary. This is done to provide them a short-circuit behavior, keeping in line with expectations from other programming languages.

$$\frac{\text{env} \vdash e_1 \Downarrow \ll \wedge \, . \, false \gg}{\text{env} \vdash e_1 \; e_2 \Downarrow false} \quad \text{(BS-}\wedge\text{False)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \ll \wedge \, . \, true \gg \qquad \text{env} \vdash e_2 \Downarrow b}{\text{env} \vdash e_1 \; e_2 \Downarrow b} \quad \text{(BS-}\wedge\text{True)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \ll \vee \, . \, true \gg}{\text{env} \vdash e_1 \; e_2 \Downarrow true} \quad \text{(BS-}\vee\text{True)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \ll \vee \, . \, false \gg \qquad \text{env} \vdash e_2 \Downarrow b}{\text{env} \vdash e_1 \; e_2 \Downarrow b} \quad \text{(BS-}\vee\text{False)}$$

Let Expressions These expressions are used to associate an identifier with a specific value, allowing the value to be reused throughout the program. Since V is a functional

language, these are not variables, and the values assigned to an identifier will be constant (unless the same identifier is used in a new let expression).

After evaluating the expression that is to be associated to the identifier (that is, e_1), resulting in v, the let expression evaluates e_2. For this evaluation, the association of p to v is added to the environment. The result of this evaluation (that is, v_2) is the final result of the evaluation of the entire let expression.

$$\frac{\text{env} \vdash e_1 \Downarrow v \qquad match(p, v) = \text{env}_1 \qquad \text{env}_1 \cup \text{env} \vdash e_2 \Downarrow v_2}{\text{env} \vdash \texttt{let}\ p = e_1\ \texttt{in}\ e_2 \Downarrow v_2} \quad \text{(BS-Let)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v \qquad \neg match(p, v)}{\text{env} \vdash \texttt{let}\ p = e_1\ \texttt{in}\ e_2 \Downarrow raise} \quad \text{(BS-Let2)}$$

If the sub-expression e_1 evaluates to $raise$, the whole expression also evaluates to $raise$.

$$\frac{\text{env} \vdash e_1 \Downarrow raise}{\text{env} \vdash \texttt{let}\ p = e_1\ \texttt{in}\ e_2 \Downarrow raise} \quad \text{(BS-LetRaise)}$$

Match Expression The match expression receives a input value and a list of $matches$, attempting to pattern match against each one. The first $match$ which correctly matches terminates the processing, and its corresponding expression is evaluated as the result of the whole expression.

If no $match$ returns a valid result, the whole expression evaluates to $raise$.

$$\frac{\text{env} \vdash e \Downarrow v \qquad \exists j \in [1..n]\, multiMatch(v, \text{env}, match_j) = v_j \qquad \forall k \in [1..j) \neg\, multiMatch(v, \text{env}, match_k)}{\text{env} \vdash \texttt{match}\ e\ \texttt{with}\ match_1, ... match_n \Downarrow v_j} \quad \text{(BS-Match)}$$

$$\frac{\text{env} \vdash e \Downarrow v \qquad \forall j \in [1..n] \neg\, multiMatch(v, \text{env}, match_j)}{\text{env} \vdash \texttt{match}\ e\ \texttt{with}\ match_1, ... match_n \Downarrow raise} \quad \text{(BS-Match2)}$$

In order to properly evaluate a match expression, it is necessary to define an auxiliary function, here called $multiMatch$. This function receives an input value, an environment and a $match$.

If the $match$ has a conditional expression, it must evaluate to $true$ for the match to be considered valid.

$$\frac{\neg\, match(p, v)}{\neg\, multiMatch(v, \text{env}, p \to e)}$$

$$\frac{\neg\, match(p, v)}{\neg\, multiMatch(v, \text{env}, p \text{ when } e_1 \to e_2)}$$

$$\frac{match(p, v) = \text{env}_1 \qquad \text{env} \cup \text{env}_1 \vdash e_1 \Downarrow false}{\neg\, multiMatch(v, \text{env}, p \text{ when } e_1 \to e_2)}$$

$$\frac{match(p, v) = \text{env}_1 \qquad \text{env} \cup \text{env}_1 \vdash e \Downarrow v_2}{multiMatch(v, \text{env}, p \to e) = v_2}$$

$$\frac{match(p, v) = \text{env}_1 \qquad \text{env} \cup \text{env}_1 \vdash e_1 \Downarrow true \quad \text{env} \cup env_1 \vdash e_2 \Downarrow v_2}{multiMatch(v, \text{env}, p \text{ when } e_1 \to e_2) = v_2}$$

IO Expressions Evaluating any IO expression requires interaction with an operating system, and how this is done is left as a matter of implementation. The only requirement imposed by the operational rules is that reading and writing be done one character at a time.

$$\frac{\text{env} \vdash e_1 \Downarrow \ll \text{read} \,.\, \gg \qquad \text{env} \vdash e_2 \Downarrow \text{Void} \quad c \text{ is the next character in the standard input}}{\text{env} \vdash e_1\ e_2 \Downarrow \text{IO}\ c} \qquad \text{(BS-Read)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \ \ll \text{write} \,.\, \gg \qquad \text{env} \vdash e_2 \Downarrow c \quad c \text{ is written to the standard input}}{\text{env} \vdash e_1\ e_2 \Downarrow \text{IO Void}} \qquad \text{(BS-Write)}$$

Composing IO In order to compose IO operations, two monadic functions were introduced in the language: return and bind. Currently, the only kind of value that can be manipulated with these functions is IO, but this system can be extended to support any monadic type in the future.

$$\frac{\text{env} \vdash e_1 \Downarrow \ll \text{return} \,.\, \gg \qquad \text{env} \vdash e_1 \Downarrow v}{\text{env} \vdash e_1\ e_2 \Downarrow \text{IO}\ v} \tag{BS-Return}$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \ll \text{bind} \,.\, \text{IO}\ v_1 \gg \qquad \text{env} \vdash e_1 \Downarrow v_2 \\ \text{env} \vdash v_2\ v_1 \Downarrow v_3\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow v_3} \tag{BS-Bind}$$

Exceptions Some programs can be syntactically correct but still violate the semantics of the V language, such as a dividing by zero or trying to access the head of an empty list. In these scenarios, the expression is evaluated as the $raise$ value.

Besides violation of semantic rules, the only other expression that evaluates to the $raise$ value is the $raise$ expression, using the following rule:

$$\text{env} \vdash raise \Downarrow raise \tag{BS-Raise}$$

This value propagates upwards through the evaluation tree if a "regular" value is expected. This means that expressions that need well-defined sub-expressions, such as numerical and equality operations, evaluate to $raise$ if any of these sub-expressions evaluate to $raise$.

APPENDIX C — TYPE SYSTEM

Like many programming languages, V has a type system. A type system is a way of statically analyzing programs to decide whether they are well-formed or not. To do this, every expression in the abstract syntax tree has, associated with it, type information.

Typing rules are then used to check that a program is correctly constructed. If a program passes the type check, guarantees are made about its execution. V's type system is not *secure* in the sense that correctly typed programs will always have correct execution.

Examples of programs that pass the type system but fail to execute are division by zero and accessing the head of an empty list. In general, however, these errors are caused by incomplete pattern matching on algebraic data types, and most errors are still caught by the type system.

V's type system is a Hindley-Milner style type system, with support for implicit type annotations and let polymorphism. Furthermore, *traits* allow ad-hoc polymorphism and, used with records, a kind of structural subtyping (more details will be provided later).

The Type Inference Algorithm Since V allows implicitly typed expressions (that is, expressions without any type annotations provided by the programmer), it is necessary to infer, and not only check, the type of expressions. A constraint-based inference system is used, which divides the algorithm into three parts: constraint collection, in which the abstract syntax tree is traversed and both a type and a list of type equality constraints is generated; constraint unification, in which the list of constraints is condensed into a type substitution; and substitution application, which applies the substitution to the type to obtain a principle type.

C.1 Constraint Collection

The first step of the type inference algorithm is the collection of type constraints. A type constraint is an equality between two types, and so the result of the constraint collection is a system of equations.

The constraint collection algorithm takes, as input, an expression e and a typing

environment Γ (defined below), and produces, as an output, a type T, a set of constraints C and a unification environment γ (defined in C.2). The algorithm is given as a set of rules of the form:

$$\Gamma \vdash e : T \mid C \mid \gamma \tag{T}$$

Constraints As described above, a set of constraints C is composed of equalities between two types.

$$C \quad ::= \quad \emptyset \mid \{T_1 = T_2\} \cup C$$

Environment The type inference environment is a 2-tuple with the following components:

1. Mapping between constructors and their type
   Every constructor has a type associated with it. This type will be a function if the constructor has arity greater than 0, and can contain variable types.
2. Mapping between identifiers and type associations
   An identifier can be either simply or universally bound to a type. The difference between these associations will be explained later.

$$\begin{array}{lcll}
\Gamma & ::= & (constructors, vars) & \\
 & & & \\
constructors & ::= & \{\} \mid \{con \mapsto T\} \cup constructors & \\
vars & ::= & \{\} \mid \{x \mapsto assoc\} \cup vars & \\
 & & & \\
assoc & ::= & T & \text{(Simple Association)} \\
 & \mid & \forall X_1, \ldots X_n \,.\, T & \text{(Universal Association)}
\end{array}$$

Type Associations When identifiers are bound in a program (with pattern matching, for example), an association between the identifier and its type is added to the typing environment. Depending on the type of binding, however, this association can be one of two kinds: simple or universal.

A simple association binds a name to a monomorphic type. This type can be "simple", such as Int, or it can contain variable types, such as $X \rightarrow$ Bool. In either case, however, the type is "constant", and it is returned unchanged from the environment.

A universal association binds a name to a type scheme. A type scheme is composed of a type T and any number of type variables $X_1, \cdots X_n$. These variables are free

in the type T, and a new instance of them is generated every time the association is returned from the environment.

Universal associations allow for polymorphic functions in the language, so each use of the function does not add constraints to other uses. The only expressions that create universal associations are let-expressions. This means that function parameters cannot be polymorphic, since function parameters are bound to simple type associations.

Free Variables Type variables can either be bound or free relative to an environment. Bound variable types are those that are associated to an identifier. This association must be a simple association, but the variable type can occur anywhere in the type tree. As an example, the type variable X_1 is bound in the environment below:

$$\Gamma = \{x \mapsto (Int, X_1, Char)\}$$

Inversely, free type variables are all those that do not occur bound in the environment.

A helper function, $\Gamma \vdash free(T)$, returns the set of all free type variables in the type T. Another function, $\Gamma \vdash fresh(T)$, returns a type T' in which all free type variables in T are replaced by new, unbound type variables. Both of these functions require an environment Γ with which to judge whether type variables are free or not.

Pattern Matching One way to bind identifiers is by pattern matching. When a pattern is encountered (such as a `let` expression), it is necessary to match the type of the pattern with the type of the value.

To do this, two auxiliary *match* functions are defined. Both take, as input, a pattern p and a type T, returning a list of constraints and a modified typing environment.

The first of the functions, $match$, only creates simple type associations and is used in `match` expressions. The second, $match_U$, can create both simple and universal associations, being used in `let` expressions.

The following are the rules for the $match$ function:

$$\Gamma \vdash match(x, T) = \{\}, \{x \mapsto T\} \cup \Gamma$$

$$\Gamma \vdash match(x : T_{pat}, T) = \{T_{pat} = T\}, \{x \mapsto T\} \cup \Gamma$$

$$\Gamma \vdash match(\_, T) = \{\}, \Gamma$$

$$\Gamma \vdash match(\_ : T_{pat}, T) = \{T_{pat} = T\}, \Gamma$$

$$\frac{\Gamma(con) = T' \qquad \Gamma \vdash fresh(T') = T'' \qquad T'' = T_1 \to \cdots \to T_n \qquad \Gamma_0 = \Gamma \qquad \forall i \in [1,n] \;\; \Gamma_{i-1} \vdash match(p_i, T_i) = C_i, \Gamma_i}{\Gamma \vdash match(con\, p_1, \ldots\, p_n, T) = \{T_n = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\Gamma(con) = T' \qquad \Gamma \vdash fresh(T') = T'' \qquad T'' = T_1 \to \cdots \to T_n \qquad \Gamma_0 = \Gamma \qquad \forall i \in [1,n] \;\; \Gamma_{i-1} \vdash match(p_i, T_i) = C_i, \Gamma_i}{\Gamma \vdash match(con\, p_1, \ldots\, p_n : T_{pat}, T) = \{T_{pat} = T, T_n = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\forall i \in [1,n] \;\; \Gamma_{i-1} \vdash fresh(X) = X_i \wedge \Gamma_{i-1} \vdash match(p_i, X_i) = C_i, \Gamma_i}{match(\{l_1 : p_1, \ldots, l_n : p_n\}, T) = \{\{l_1 : X_1, \ldots, l_n : X_n\} = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\forall i \in [1,n] \;\; \Gamma_{i-1} \vdash fresh(X) = X_i \wedge \Gamma_{i-1} \vdash match(p_i, X_i) = C_i, \Gamma_i}{match(\{l_1 : p_1, \ldots, l_n : p_n\} : T_{pat}, T) = \{\{l_1 : X_1, \ldots, l_n : X_n\} = T, T_{pat} = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\forall i \in [1,n] \;\; \Gamma_{i-1} \vdash fresh(X) = X_i \wedge \Gamma_{i-1} \vdash match(p_i, X_i) = C_i, \Gamma_i \qquad X_0^{\{\{l_i : X_i\}, \ldots \{l_n : X_n\}\}}}{match(\{l_1 : p_1, \ldots, l_n : p_n, \ldots\}, T) = \{X_0 = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\forall i \in [1,n] \;\; \Gamma_{i-1} \vdash fresh(X) = X_i \wedge \Gamma_{i-1} \vdash match(p_i, X_i) = C_i, \Gamma_i \qquad X_0^{\{\{l_i : X_i\}, \ldots \{l_n : X_n\}\}}}{match(\{l_1 : p_1, \ldots, l_n : p_n, \ldots\} : T_{pat}, T) = \{X_0 = T, T_{pat} = T\} \cup \bigcup_{i=1}^{n} c_i, \Gamma_n}$$

The rules for the $match_U$ are similar, but with a few key differences. Since universal matching is always called with a completely unified type in a `let` expression, certain concessions can be made. Furthermore, other differences arise due to the ability for the $match_U$ function to create universal type associations.

Because $match_U$ can create universal associations, it checks for any free variable types in the type T. If free variable types are found, then a universal association is made based on these variable types. If there are no free variable types, a simple association is created instead.

$$\frac{\Gamma \vdash free(T) = \{\}}{\Gamma \vdash match_U(x, T) = \{\}, \{x \mapsto T\} \cup \Gamma}$$

$$\frac{\Gamma \vdash free(T) = \{X_1, \ldots, X_n\}}{\Gamma \vdash match_U(x, T) = \{\}, \{x \mapsto \forall X_1, \ldots X_n \,.\, T\} \cup \Gamma}$$

$$\frac{\Gamma \vdash free(T) = \{\}}{\Gamma \vdash match_U(x : T_{pat}, T) = \{T_{pat} = T\}, \{x \mapsto T\} \cup \Gamma}$$

$$\frac{\Gamma \vdash free(T) = \{X_1, \ldots, X_n\}}{\Gamma \vdash match_U(x : T_{pat}, T) = \{T_{pat} = T\}, \{x \mapsto \forall X_1, \ldots X_n \,.\, T\} \cup \Gamma}$$

$$\Gamma \vdash match_U(\_, T) = \{\}, \Gamma$$

$$\Gamma \vdash match_U(\_ : T_{pat}, T) = \{T_{pat} = T\}, \Gamma$$

The next difference comes when matching against constructor patterns. Instead of creating a fresh instance of the type associated with the constructor by replacing all variable types with fresh variable types, the type T (passed as paramenter to $match_U$) is used.

This function is called $\Gamma \vdash rebase(T_1, T_2)$, creating a new instance of T_1 based on T_2. Informally, this means that both the type T_1 and T_2 are traversed simultaneously and, when a variable type is encountered in T_1, it is replaced by the equivalent type in T_2.

$$\frac{\begin{array}{c}\Gamma(con) = T' \qquad \Gamma \vdash rebase(T', T) = T'' \\ T'' = T_1 \to \cdots \to T_n \qquad \Gamma_0 = \Gamma \\ \forall i \in [1, n] \;\; \Gamma_{i-1} \vdash match_U(p_i, T_i) = C_i, \Gamma_i\end{array}}{\Gamma \vdash match_U(con\ p_1, \ \ldots\ p_n, T) = \{T_n = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\begin{array}{c}\Gamma(con) = T' \qquad \Gamma \vdash rebase(T', T) = T'' \\ T'' = T_1 \to \cdots \to T_n \qquad \Gamma_0 = \Gamma \\ \forall i \in [1, n] \;\; \Gamma_{i-1} \vdash match_U(p_i, T_i) = C_i, \Gamma_i\end{array}}{\Gamma \vdash match_U(con\ p_1, \ \ldots\ p_n : T_{pat}, T) = \{T_{pat} = T, T_n = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

Finally, pattern matching on records in also slightly different. For complete record patterns, we know that the type T is a record type with the necessary fields, and so the matching rule becomes much smaller.

$$\frac{T = \{l_1 : T_1, \ldots, l_n : T_n\} \quad \forall i \in [1,n] \;\; \Gamma_{i-1} \vdash match_U(p_i, T_i) = C_i, \Gamma_i}{match(\{l_1 : p_1, \ldots, l_n : p_n\}, T) = \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{T = \{l_1 : T_1, \ldots, l_n : T_n\} \quad \forall i \in [1,n] \;\; \Gamma_{i-1} \vdash match(p_i, T_i) = C_i, \Gamma_i}{match(\{l_1 : p_1, \ldots, l_n : p_n\} : T_{pat}, T) = \{T_{pat} = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

For partial record patterns, however, a little more care must be taken. Since there may be fewer fields in the pattern than the type, a search must be done to match the correct sub-patterns with the sub-types.

$$\frac{T = \{l'_1 : T_1, \ldots, l'_k : T_k\} \quad k \geq n \qquad \forall\, i \in [1,n] \;\; \exists\, j \in [1,k] \;\; l_i = l'_j \wedge \Gamma_{i-1} \vdash match(p_i, T_j) = C_i, \Gamma_i}{match(\{l_1 : p_1, \ldots, l_n : p_n, \ldots\}, T) = \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{T = \{l'_1 : T_1, \ldots, l'_k : T_k\} \quad k \geq n \qquad \forall\, i \in [1,n] \;\; \exists\, j \in [1,k] \;\; l_i = l'_j \wedge \Gamma_{i-1} \vdash match(p_i, T_j) = C_i, \Gamma_i}{match(\{l_1 : p_1, \ldots, l_n : p_n, \ldots\} : T_{pat}, T) = \{T_{pat} = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

Constraint Collection Rules Every expression in V has a rule for constraint collection, similar to how every expression has a rule for its semantic evaluation.

If a rule does not create any constraints or unification environment (i.e. they are both empty), then these will be omited to improve readability. As an example, the following rule:

$$\Gamma \vdash + : \text{Int} \to \text{Int} \to \text{Int} \mid \{\} \mid \{\} \tag{T-+}$$

will be written as:

$$\Gamma \vdash + : \text{Int} \to \text{Int} \to \text{Int} \tag{T-+}$$

Functions The rules for function expressions are all similar, though with a few differences between them. All of them create a fresh type variable X_1 to represent the type

of their argument, and the resulting type is always $X_1 \to T_1$, where T_1 is the type of the body of the function.

When calling the collection algorithm on the body of the function (i.e. e), the typing environment Γ is modified by addind a new association between the identifier x and the type X_1.

$$\frac{\Gamma \vdash fresh(X) = X_1 \qquad \{x \mapsto X_1\} \cup \Gamma \vdash e : T_1 \mid C_1 \mid \gamma_1}{\Gamma \vdash \texttt{fn}\ x \Rightarrow e : X_1 \to T_1 \mid C_1 \mid \gamma_1} \quad \text{(T-Fn)}$$

Recursive functions add the same association between x and X_1, but they also create a new association for the name of the function, f. If the function is implicitly typed, a new type variable, X_2, is used to represent the type of the function. Thus, f is associated to X_2, and a new constraint between X_2 and $X_1 \to T_1$ is created.

$$\frac{\begin{array}{c}\Gamma \vdash fresh(X) = X_1 \qquad \Gamma \vdash fresh(X) = X_2 \\ \{f \mapsto X_2, x \mapsto X_1\} \cup \Gamma \vdash e : T_1 | C_1 \mid \gamma_1\end{array}}{\Gamma \vdash \texttt{rec}\ f\ x \Rightarrow e : X_1 \to T_1 \mid C_1 \cup \{X_2 = X_1 \to T_1\} \mid \gamma_1} \quad \text{(T-Rec)}$$

If the function is explicitly typed, however, no new type variables are created. Instead, f is associated directly to $X_1 \to T$, and a constraint to guarantee that the provided type is correct is created (that is, that T is equal to T_1).

$$\frac{\Gamma \vdash fresh(X) = X_1 \qquad \{f \mapsto (X_1 \to T), x \mapsto X_1\} \cup \Gamma \vdash e : T_1 \mid C_1 \mid \gamma_1}{\Gamma \vdash \texttt{rec}\ f : T\ \ x \Rightarrow e : X \to T_1 \mid C_1 \cup \{T_1 = T\} \mid \gamma_1} \quad \text{(T-Rec2)}$$

Built-in Functions None of the built-in functions create any constraints or unification environment, nor do their types depend on a typing environment. Some functions, because of their polymorphic nature, require creation of fresh type variables.

Numerical Functions These functions all manipulate Int values, with negate ($-$) being the only function with a single argument.

$$\Gamma \vdash + : \text{Int} \to \text{Int} \to \text{Int} \quad \text{(T-+)}$$

$$\Gamma \vdash - : \text{Int} \to \text{Int} \to \text{Int} \quad \text{(T–)}$$

$$\Gamma \vdash * : \text{Int} \to \text{Int} \to \text{Int} \quad \text{(T-*)}$$

$$\Gamma \vdash \div : \text{Int} \to \text{Int} \to \text{Int} \quad \text{(T-÷)}$$

$$\Gamma \vdash - : \text{Int} \to \text{Int} \qquad \text{(T-Negate)}$$

Equality Functions These functions do not require a specific type for their arguments, but they both must be equal and conform to the $Equatable$ trait.

$$\frac{\Gamma \vdash fresh(X^{\{Equatable\}}) = T}{\Gamma \vdash =: T \to T \to \text{Bool}} \qquad \text{(T-=)}$$

$$\frac{\Gamma \vdash fresh(X^{\{Equatable\}}) = T}{\Gamma \vdash \neq: T \to T \to \text{Bool}} \qquad \text{(T-}\neq\text{)}$$

Inequality Functions Similar to equality, both arguments must have the same type and conform to the $Orderable$ trait.

$$\frac{\Gamma \vdash fresh(X^{\{Orderable\}}) = T}{\Gamma \vdash <: T \to T \to \text{Bool}} \qquad \text{(T-<)}$$

$$\frac{\Gamma \vdash fresh(X^{\{Orderable\}}) = T}{\Gamma \vdash \leq: T \to T \to \text{Bool}} \qquad \text{(T-}\leq\text{)}$$

$$\frac{\Gamma \vdash fresh(X^{\{Orderable\}}) = T}{\Gamma \vdash >: T \to T \to \text{Bool}} \qquad \text{(T->)}$$

$$\frac{\Gamma \vdash fresh(X^{\{Orderable\}}) = T}{\Gamma \vdash \geq: T \to T \to \text{Bool}} \qquad \text{(T-}\geq\text{)}$$

Boolean Functions Both functions require two Bool arguments, returning another Bool.

$$\Gamma \vdash \vee : \text{Bool} \to \text{Bool} \to \text{Bool} \qquad \text{(T-}\vee\text{)}$$

$$\Gamma \vdash \wedge : \text{Bool} \to \text{Bool} \to \text{Bool} \qquad \text{(T-}\wedge\text{)}$$

Accessor Functions These functions manipulate accessors, creating fresh type variables to represent all necessary types.

$$\frac{\Gamma \vdash fresh(X) = T_1 \qquad \Gamma \vdash fresh(X) = T_2}{\Gamma \vdash \text{get} : T_2 \# T_1 \to T_2 \to T_1} \qquad \text{(T-get)}$$

$$\frac{\Gamma \vdash fresh(X) = T_1 \qquad \Gamma \vdash fresh(X) = T_2}{\Gamma \vdash \text{set} : T_2 \# T_1 \to T_1 \to T_2 \to T_2} \quad \text{(T-SET)}$$

$$\frac{\Gamma \vdash fresh(X) = T_1 \qquad \Gamma \vdash fresh(X) = T_2 \qquad \Gamma \vdash fresh(X) = T_3}{\Gamma \vdash \text{stack} : T_2 \# T_1 \to T_1 \# T_3 \to T_2 \# T_3} \quad \text{(T-STACK)}$$

$$\frac{\Gamma \vdash fresh(X) = T_1 \qquad \Gamma \vdash fresh(X) = T_2 \qquad \Gamma \vdash fresh(X) = T_3}{\Gamma \vdash \text{distort} : T_2 \# T_1 \to (T_1 \to T_3) \to (T_3 \to T_1 \to T_1) \to T_2 \# T_3} \quad \text{(T-DISTORT)}$$

IO Functions The typing rules for both IO related functions are very simple, relying on Void_T and IO_T to define their inputs and outputs.

$$\Gamma \vdash \text{read} : \text{Void}_T \to \text{IO}_T\ \text{Char} \quad \text{(T-READ)}$$

$$\Gamma \vdash \text{write} : \text{Char} \to \text{IO}_T\ \text{Void}_T \quad \text{(T-GET)}$$

For IO composition, the return and bind functions also have straightforward type signatures.

$$\frac{\Gamma \vdash fresh(X) = T_1}{\Gamma \vdash \text{return} : T_1 \to \text{IO}_T\ T_1} \quad \text{(T-RETURN)}$$

$$\frac{\Gamma \vdash fresh(X) = T_1 \qquad \Gamma \vdash fresh(X) = T_2}{\Gamma \vdash \text{write} : \text{IO}_T\ T_1 \to (T_1 \to \text{IO}_T\ T_2) \to \text{IO}_T\ T_2} \quad \text{(T-BIND)}$$

Constructors The rule for typing constructors is very simple. The type is extracted from the environment, and then a fresh instance is generated from that type.

$$\frac{\Gamma(con) = T \qquad \Gamma \vdash fresh(T) = T'}{\Gamma \vdash con : T'} \quad \text{(T-CON)}$$

Application The constraint collection rule for an application is simple, creating just one fresh type variable and one new constraint. The type variable X_1 represents the type of the result of the application, and, therefore, is the return type of the collection. Furthermore, the type of e_1, T_1, must be equal to a function that takes T_2 (the type of e_2) as an argument and returns X_1.

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \mid \gamma_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \mid \gamma_2 \qquad \Gamma \vdash fresh(X) = X_1}{\Gamma \vdash e_1\ e_2 : X_1 \mid C_1 \cup C_2 \cup \{T_1 = T_2 \to X_1\} \mid \gamma_1 \cup \gamma_2} \quad \text{(T-APP)}$$

Identifiers The type of an identifier is, like for constructors, completely defined by its typing association in the environment. The typing rule does not create a fresh instance of this type, since the environment already does this when returning types that are universally bound.

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \text{(T-Ident)}$$

Records The constraint collection rule for a record is relatively straightforward. Each field of the record is passed through the collection algorithm, and the resulting types are combined into a single record type with their matching labels. Similarly, the resulting constraints and unification environments are combined by union.

$$\frac{\forall\, k \in [1,n] \;\; \Gamma \vdash e_k : T_k \mid C_k \mid \gamma_k}{\Gamma \vdash \{l_1 : e_1,\, \ldots\, l_n : e_n\} : \{l_1 : T_1,\, \ldots\, l_n : T_n\} \mid \bigcup_{i=1}^{n} C_i \mid \bigcup_{i=1}^{n} \gamma_i} \quad \text{(T-Record)}$$

Accessors The constraint rules for simple label accessors relies on type variables and record label traits. A new fresh type variable, T_1, is generated, representing the type of the field being accessed. Another new fresh type variable, T_2, which must conform to the record label trait associating the label l to the type T_1, is generated, representing the type of the record that is being accessed.

$$\frac{\Gamma \vdash fresh(X) = T_1 \qquad \Gamma \vdash fresh(X^{\{l:T_1\}}) = T_2}{\Gamma \vdash \#l : T_2 \# T_1} \quad \text{(T-Label)}$$

A joined accessor does not use record label traits, but instead relies on type variables and constraints to guarantee the correct type information.

A single type variable X_0 represents the record being accessed. For every component e_i of the accessor, a new type variable X_i is generated, along with the resulting type T_i of calling the constraint collection algorithm. The type T_i is then constrained to be equal to $X_0 \# X_i$, indicating that all components refer the same record, but access fields with (possibly) different types.

Finally, the resulting type is an accessor that returns a tuple composed of all X_i when accessing a record of type X_0.

$$\frac{fresh(X) = X_0 \qquad \forall\, i \in [1,n] \;\; \Gamma \vdash fresh(X) = X_i \wedge \Gamma \vdash e_i : T_i \mid C_i \mid \gamma_i}{\Gamma \vdash \#(e_1,\, \ldots\, e_n) : X_0 \#(X_i,\, \ldots\, X_n) \mid \bigcup_{i=1}^{n} C_i \cup \{T_i = X_0 \# X_i\} \mid \bigcup_{i=1}^{n} \gamma_i} \quad \text{(T-Joined)}$$

Let Expression The constraint collection rule for a `let` expression depends on both the unification and the application algorithms.

The expression e_1 is passed through the constraint collection algorithm, resulting in a type T_1, a set of constraints C_1 and a unification environment γ_1. The constraints C_1 are then unified (see Appendix C.2) under the unification environment γ_1, resulting in a substitution σ.

The substitution σ is then applied (see Appendix C.3) to the type T_1, resulting in a principle type T_1'. The substitution is also applied to the environment Γ, and the result of this application is used to evaluate a universal match between p and T_1', resulting in a new set of constraints C_1' and a new typing environment Γ'.

Finally, the type of the expression e_2 is obtained under the environment Γ'.

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_1 \mid C_1 \mid \gamma_1 \qquad \gamma_1 \vdash \mathcal{U}(C_1) = \sigma \qquad \sigma(T_1) = T_1' \\ \sigma(\Gamma) \vdash match_U(p, T_1') = C_1', \Gamma' \qquad \Gamma' \vdash e_2 : T_2 \mid C_2 \mid \gamma_2\end{array}}{\Gamma \vdash \texttt{let}\ p = e_1\ \texttt{in}\ e_2 : T_2 \mid C_1' \cup C_1 \cup C_2 \mid \gamma_2 \cup \gamma_1} \quad \text{(T-Let)}$$

Match Expression The constraint collection rule for a match expression requires an auxiliary function, much like its operational semantic rule. A fresh type variable X_1 is created, representing the output type of the expression and, along with the type T of the expression e, is used to validate every $match_i$ in the expression.

$$\frac{\begin{array}{c}\Gamma \vdash e : T \mid C \mid \gamma \qquad \Gamma \vdash fresh(X) = X_1 \\ \forall i \in [1..n] \Gamma \vdash validate(match_i, T, X_1) = C_i \mid \gamma_i\end{array}}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ match_1, \ldots\ match_n : X_1 \mid C \cup \bigcup_{i=1}^{n} C_i \mid \gamma \cup \bigcup_{i=1}^{n} \gamma_i} \quad \text{(T-Match)}$$

The $validate$ function takes a $match$ expression, a type T_{in}, representing the type of the pattern, and a T_{out}, representing the result of evaluating the $match$ expression, as inputs. The function outputs a set of constraints and a unification environment if successful.

For an unconditional $match$, the pattern in matched against the provided input type T_{in} and the type of the expression e is constrained to equal the provided type T_{out}. It is important to realize that the typing environment returned by the match (i.e. Γ') is used only to obtain the type of e, since any identifiers bound in the pattern p can only be used inside a single $match$ expression.

$$\frac{\Gamma \vdash match(p, T_{in}) = C, \Gamma' \qquad \Gamma' \vdash e : T_1 \mid C_1 \mid \gamma_1}{\Gamma \vdash validate(p \rightarrow e, T_{in}, T_{out}) = C \cup C_1 \cup \{T_1 = T_{out}\} \mid \gamma_1}$$

The same holds true for a conditional $match$, with the added verification that the type of e_1 must be equal to $Bool$.

$$\frac{\Gamma \vdash match(p, T_{in}) = C, \Gamma' \qquad \Gamma' \vdash e_1 : T_1 \mid C_1 \mid \gamma_1 \qquad \Gamma' \vdash e_2 : T_2 \mid C_2 \mid \gamma_2}{\begin{array}{c}\Gamma \vdash validate(p \text{ when } e_1 \rightarrow e_2, T_{in}, T_{out}) = \\ C \cup C_1 \cup C_2 \cup \{T_1 = Bool, T_2 = T_{out}\} \mid \gamma_1 \cup \gamma_2\end{array}}$$

Exception The $raise$ expression simply creates and returns a new fresh type variable.

$$\frac{\Gamma \vdash fresh(X) = X_1}{\Gamma \vdash raise : X_1} \qquad \text{(T-Raise)}$$

C.2 Unification

After constraint collection, the second step in the type inference algorithm is unification. Unification attempts to solve the set of equalities defined by the constraints collected in the previous step.

The algorithm takes, as input, a set of constraints C and a unification environment γ, and produces a substitution σ as output. The algorithm is given as a set of rules following the form:

$$\gamma \vdash \mathcal{U}(C) = \sigma \qquad \text{(U)}$$

Unification Environment The unification environment is a set of trait specifications. Trait specifications are 3-tuples that define the requirements for a specific type to conform to a specific trait.

$$\gamma \quad ::= \quad \{\} \mid \{\text{trtSpec}\} \cup \gamma$$

$$\text{trtSpec} \quad ::= \quad (conT, Trait, [Traits_1, \ldots Traits_n]) \quad (n = \text{arity } conT)$$

A trait specification describes conformance to a trait. Some types, such as records and functions, have their trait conformance built into the language, and it is not necessary to use the unification environment to decide conformance. For a constructor type $conT\ T_1, \ldots\ T_n$ to conform to a trait $Trait$, however, the following criteria must hold:

1. There exists a trait specification $(conT, Trait, [Traits_1, \ldots Traits_n])$ in the unification environment

2. For all $i \in [1, n]$, $T_i \in Traits_i$

This will be formally defined in the unification algorithm itself, but these are the general rules that govern trait conformance.

Unification Rules Unification iterates through the list of constraints, always operating on the first constraint in the list and recursing on the remaining constraints. Because of this format, it can be defined as a base case and recursion rules.

Base Cases The basic case, with an empty constraint list, returns an empty substitution.

$$\gamma \vdash \mathcal{U}(\emptyset) = \emptyset \tag{U-Empty}$$

If both types of a constraint are equal, they are discarded and the recursion is called.

$$\frac{T_1 = T_2}{\gamma \vdash \mathcal{U}(\{T_1 = T_2\} \cup C) = \gamma \vdash \mathcal{U}(C)} \tag{U-Equals}$$

Compound Types When encountered, compound types are destructured and equality constraints with their corresponding components are added to the end of the constraint list.

$$\frac{\begin{array}{c} T_1 = T_1^1 \# T_1^2 \qquad T_2 = T_2^1 \# T_2^2 \\ C' = \{T_1^1 = T_2^1, T_1^2 = T_2^2\} \end{array}}{\gamma \vdash \mathcal{U}(\{T_1 = T_2\} \cup C) = \gamma \vdash \mathcal{U}(C \cup C')} \tag{U-Accessor}$$

$$\frac{\begin{array}{c} T_1 = T_1^1 \to T_1^2 \qquad T_2 = T_2^1 \to T_2^2 \\ C' = \{T_1^1 = T_2^1, T_1^2 = T_2^2\} \end{array}}{\gamma \vdash \mathcal{U}(\{T_1 = T_2\} \cup C) = \gamma \vdash \mathcal{U}(C \cup C')} \tag{U-Fn}$$

$$\frac{\begin{array}{c} T_1 = \{l_1 : T_1^1, \ldots l_n : T_1^n\} \qquad T_2 = \{l_1 : T_2^1, \ldots l_n : T_2^n\} \\ C' = \{T_1^1 = T_2^1, \ldots T_1^n = T_2^n\} \end{array}}{\gamma \vdash \mathcal{U}(\{T_1 = T_2\} \cup C) = \gamma \vdash \mathcal{U}(C \cup C')} \tag{U-Record}$$

For applied constructor types, it is also necessary to verify that the constructor types themselves are equal. If not, the unification fails.

$$\frac{T_1 = conT_1\ T_1^1,\ \ldots\ T_1^n \qquad T_2 = conT_2\ T_2^1,\ \ldots\ T_2^n \qquad conT_1 = conT_2 \qquad C' = \{T_1^1 = T_2^1,\ \ldots\ T_1^n = T_2^n\}}{\gamma \vdash \mathcal{U}(\{T_1 = T_2\} \cup C) = \gamma \vdash \mathcal{U}(C \cup C')} \quad \text{(U-Cons)}$$

Type Variables When unifying a constraint that contains a type variable, a more complicated process of unification is necessary.

First, the type variable must not be contained in the free variables of its paired type, ensuring that the types are not circular. Then, the type T_2 must conform to the type variable's traits, and this process might create additional constraints that must be unified. These newly created constraints, along with the remaining constraints, are then unified after having every occurrence of X replaced with the type T_2.

$$\frac{T_1 = X^{Traits} \qquad X \notin free(T_2) \qquad T_2 \in Traits \to C_T \qquad C' = C \cup C_T \qquad \gamma \vdash \mathcal{U}(C'[X \mapsto T_2]) = \sigma}{\gamma \vdash \mathcal{U}(\{T_1 = T_2\} \cup C) = \{X \mapsto T_2\} \cup \sigma} \quad \text{(U-Var)}$$

Substitution The result of applying the unification algorithm is a substitution σ. A substitution is a mapping from type variables to types.

$$\sigma \quad ::= \quad \emptyset \mid \{X \mapsto T\} \cup \sigma$$

C.3 Application

The last component of the type inference algorithm is the application of a type substitution. This replaces all type variables that are specified by the substitution, resulting in a new instance of the input type.

Application takes, as input, a type T and a substitution σ, producing another type T' as output. It is defined with rules of the form:

$$\sigma(T) = T' \quad \text{(A)}$$

APPENDIX D — EXTENDED LANGUAGE

In order to facilitate programming, it is useful to define an extended language. A program is first parsed into this language, and the resulting tree is translated into the regular abstract syntax.

This allows the core language to be concise, reducing the complexity of type inference and evaluation. Complex constructs (such as comprehensions and multi-parameter functions) can be included only in the extended language, and it suffices to provide a translation into the core language.

This translation does have the drawback of reducing the formality of evaluation. Since there are no evaluation rules for the additional constructs, it is impossible to prove the correctness of the translation rules. This does not in any way affect the correctness of the core language type inference and evaluation, and the advantages gained by this method far outweigh the drawbacks, so it is still a net positive to the language.

The following two sections will describe the abstract syntax tree for the extended language and how it translates to a syntax tree in the core language.

D.1 Abstract Syntax

The extended language has terms which are similar to (if not exactly the same as) ones existing in the core language. These terms are presented in their entirety here, and, since they are directly extracted from the core language, no explanation will be given for them.

$$
\begin{array}{lcll}
e' & ::= & func' & \\
 & | & e'_1\ e'_2 & \\
 & | & x & \\
 & | & Builtin & \\
 & | & con & \\
 & | & \{l_1 : e'_1,\ \ldots\ l_n : e'_n\} & (n \geq 1) \\
 & | & \#l & \\
 & | & \#(e'_1,\ \ldots\ e'_n) & (n \geq 2) \\
 & | & raise & \\
 & | & \texttt{match}\ e'\ \texttt{with}\ match'_1, \ldots\ match'_n & (n \geq 1)
\end{array}
$$

$$
\begin{array}{lcl}
match' & ::= & p' \to e' \\
& | & p' \ \mathsf{when}\ e'_1 \to e'_2
\end{array}
$$

Most patterns are extracted from the core language without any differences, and are defined below.

$$
\begin{array}{lcl}
p' & ::= & patt' \\
& | & patt' : T'
\end{array}
$$

$$
\begin{array}{lcll}
patt' & ::= & x & \\
& | & \_ & \\
& | & con\ p'_1, \ \ldots\ p'_n & (\text{constructor pattern}, n = \text{arity } con) \\
& | & \{l_1 : p'_1, \ \ldots\ l_n : p'_n\} & (n \geq 1) \\
& | & \{l_1 : p'_1, \ \ldots\ l_n : p'_n, \ldots\} & (\text{partial record}, n \geq 1)
\end{array}
$$

Most types are, like patterns, extracted from the language.

$$
\begin{array}{lcll}
T' & ::= & X^{Traits} & \\
& | & conT\ T'_1, \ \ldots\ T'_n & (n = \text{arity } conT) \\
& | & T'_1 \to T'_2 & \\
& | & \{l_1 : T'_1, \ \ldots\ l_n : T'_n\} & (n \geq 1) \\
& | & T'_1 \# T'_2 & Accessor
\end{array}
$$

D.1.1 Additions

The extended language provides a number of additions to the base expressions and types. These are presented below.

Type Aliases The first addition to the extended language is the concept of *type aliases*. These types are simple renames of existing types, and can be used in programs as a way to simplify type declarations.

$$
\begin{array}{lcl}
T' & ::= & \cdots \\
& | & \tau
\end{array}
$$

$$
\begin{array}{lcl}
\tau & ::= & \{\tau_0, \tau_1, \ldots\}
\end{array}
$$

Conditional Expression A conditional expression, which translates to a `match` expression on the patterns $true$ and $false$, has been added.

$$
\begin{array}{lcll}
e' & ::= & \cdots & \\
 & | & \texttt{if}\ e'_1\ \texttt{then}\ e'_2\ \texttt{else}\ e'_3 &
\end{array}
$$

Multi-Parameter and Pattern Matching Functions Functions have been extended to allow multiple parameters, removing the necessity of declaring nested functions. These functions still require at least one parameter.

Furthermore, patterns are allowed as the definition of parameters.

$$
\begin{array}{lcll}
func' & ::= & \cdots & \\
 & | & \texttt{fn}\ p'_1, \ldots\ p'_n \Rightarrow e' & (n \geq 1) \\
 & | & \texttt{rec}\ f : T\ \ p'_1, \ldots\ p'_n \Rightarrow e' & (n \geq 1) \\
 & | & \texttt{rec}\ f\ p'_1, \ldots\ p'_n \Rightarrow e' & (n \geq 1)
\end{array}
$$

Declarations The `let` expression is also extended, and a new construction ($decl'$) is needed. Besides the basic value binding, 4 new function bindings are allowed. These correspond to all combinations of typed, untyped, recursive and non-recursive functions, with at least one parameter.

Along with value and function bindings, a new type alias binding was added. This binding creates a new type alias that can be used further down in the syntax tree.

$$
\begin{array}{lcll}
e' & ::= & \cdots & \\
 & | & decl'\ \texttt{in}\ e & \\
 & & & \\
decl' & ::= & \texttt{let}\ p' = e' & \\
 & | & \texttt{let}\ f\ p'_1, \ldots\ p'_n = e' & (n \geq 1) \\
 & | & \texttt{let rec}\ f\ p'_1, \ldots\ p'_n = e' & (n \geq 1) \\
 & | & \texttt{let}\ f : T'\ p'_1, \ldots\ p'_n = e' & (n \geq 1) \\
 & | & \texttt{let rec}\ f : T'\ p'_1, \ldots\ p'_n = e' & (n \geq 1) \\
 & | & \texttt{type alias}\ \tau = T' &
\end{array}
$$

Lists Although lists are supported by the base language with the :: (cons) and nil data constructors, they are not easy to use with only these terms.

The extended language provides a term to implicitly define a list, specifying all of its components between square brackets.

$$
\begin{array}{lcll}
e' & ::= & \cdots & \\
 & | & [e'_1, \ldots\ e'_n] & (n \geq 0)
\end{array}
$$

In a similar fashion, a specific pattern for lists is added.

$$\begin{array}{lcll} patt' & ::= & \cdots & \\ & | & [p'_1, \ldots p'_n] & (n \geq 0) \end{array}$$

Range Using a similar construction to basic lists, *ranges* allow the programmer to specify a list of numbers without having to declare all of them explicitly.

There are two variations on ranges. The first is a simple range, providing the start and end values. This range creates a list with all integers starting from the first value, incrementing by one until the last value.

The second variation provides, along with the start and end values, the second value of the list. This allows the language to know what the increment of the range is. Besides allowing increments greater than 1, this also allows ranges that decrement from the start value until the end value.

$$\begin{array}{lcl} e' & ::= & \cdots \\ & | & [e'_1 \mathinner{..} e'_2] \\ & | & [e'_1, e'_2 \mathinner{..} e'_3] \end{array}$$

Comprehension V provides a very basic list comprehension syntax. This allows evaluating an expression for every value in an existing list, returning a list with the results of every evaluation.

$$\begin{array}{lcl} e' & ::= & \cdots \\ & | & [e'_1 \text{ for } p' \text{ in } e'_2] \end{array}$$

Tuple Like lists, tuples are supported by the language through the Tuple n constructor. To allow easier creation of tuples, a new term is added to the extended language.

$$\begin{array}{lcll} e' & ::= & \cdots & \\ & | & (e'_1, \ldots e'_n) & (n \geq 2) \end{array}$$

Do Notation For composing multiple IO operations, using the basic bind and return operations is cumbersome. Because of this, the following notation was introduced:

$$
\begin{array}{lcl}
e' & ::= & \cdots \\
& | & \mathsf{do}\ doTerms \\
\\
doTerms & ::= & e' \\
& | & doTerm :: doTerms \\
\\
doTerm & ::= & p' \leftarrow e' \\
& | & e' \\
& | & decl'
\end{array}
$$

Accessors and Records The extended language provides easier syntax for creating and using accessors, both in accessing and updating fields of records.

The basis for this syntax is the dot, which simplifies the construction of the most common use cases for accessors. A dot is composed of a stack of acc, which are individual components in an accessor. accs can be field labels, arbitrary identifiers or joined dots.

$$
\begin{array}{lcl}
e' & ::= & \cdots \\
& | & \#dot \\
\\
dot & ::= & acc\ .\ dot \\
& | & acc \\
\\
acc & ::= & l \\
& | & 'x \\
& | & (dot_1, \ldots\ dot_n) \quad (n \geq 2)
\end{array}
$$

Dot Access Another use of the dot is to allow a simple syntax for getting a field of a record, being very similar to many object-oriented languages.

$$
\begin{array}{lcl}
e' & ::= & \cdots \\
& | & x\ .\ dot
\end{array}
$$

Record Update Finally, the dot syntax is used to allow the easy update of a record.

$$
\begin{array}{lcl}
e' & ::= & \cdots \\
& | & \texttt{update}\ updates \\
\\
updates & ::= & \varnothing \\
& | & update :: updates \\
\\
update & ::= & dot \leftarrow e' \\
& | & dot \rightsquigarrow e' \\
& | & decl'
\end{array}
$$

D.2 Translation

To actually evaluate or type check a program in the extended language, it must first be translated into the core language. This is done by a translation algorithm which, besides converting extended terms into core terms, also performs some additional safety checks.

A translation rule is of the form:

$$\gamma \vdash e' \Rightarrow e$$

where γ is the translation environment.

Besides translating expressions, the translation algorithm also translates types (T'), functions ($func'$), etc. All these translations will be described using the same format, and also use the same environment.

Enviroment Like evaluation and type inference, the translation algorithm requires an environment to properly function. This environment contains the following information:

1. Type aliases

   A mapping of type aliases to core types

2. Mapping of generated identifiers

   A mapping of identifiers to other identifiers. This is used because the translation algorithm can create new identifiers, and so it maps identifiers from the input expression to new identifiers in the output expression.

$$
\begin{array}{lll}
\gamma & ::= & (aliases, ids) \\
\\
aliases & ::= & \{\} \mid \{\tau \mapsto T\} \cup aliases \quad (n \in \mathbb{N}) \\
ids & ::= & \{\} \mid \{x_1 \mapsto x_2\} \cup ids
\end{array}
$$

Below will be sections describing the translation algorithms for the different types of expressions in the extended language. As to avoid clutter, only the rules that perform some sort of computation or modification on the expression will be displayed. This means that rules such as:

$$\gamma \vdash \text{Int} \Rightarrow \text{Int} \tag{Tr-T-Int}$$

will not be provided.

Similarly, composite expressions that simply call the translation algorithm recursively on their sub-expressions, without any modification to structure, will be omitted. This includes rules such as:

$$\frac{\gamma \vdash T_1' \Rightarrow T_1 \qquad \gamma \vdash T_2' \Rightarrow T_2}{\gamma \vdash (T_1' \to T_2') \Rightarrow (T_1 \to T_2)} \tag{Tr-T-Func}$$

Type Translation Given the fact that trivial translations are not provided, there is only one translation rule that governs type translations.

$$\frac{\gamma.aliases(\tau) = T}{\gamma \vdash \tau \Rightarrow T} \tag{Tr-T-Alias}$$

Pattern Translation The algorithm for translating patterns works on lists of patterns instead of single patterns. This is done to allow verification of repeated identifiers in a list of patterns.

Since functions allow multiple patterns as parameters, a verification is done to ensure that there are no repeated identifiers in any of the parameters. Furthermore, composite patterns, such as a list or tuple pattern, also cannot repeat identifiers in their sub-patterns, as this would cause ambiguous binding.

This verification is done from left to right, composing a set of identifiers already used in the pattern. Each verification uses the set of identifiers from the previous verification, and so the complete set of used identifiers is created.

The translation also returns a modified translation environment. This environment has new identifier mappings, one for each x pattern found.

$$\frac{\mathrm{id}_0 = \emptyset \qquad \gamma_0 = \gamma \qquad \forall i \in [1,\, n]\,.\,\mathrm{collectPatterns}(p'_i, \mathrm{id}_{i-1}, \gamma_{i-1}) = p_i, \mathrm{id}_i, \gamma_i}{\gamma \vdash [p'_1,\, \ldots\, p'_n] \Rightarrow [p_1,\, \ldots\, p_n], \gamma_n} \quad \text{(Tr-P)}$$

To translate a single pattern, a list with only that pattern is created and then translated. This ensures that, even within a single pattern, no identifiers are allowed to repeat.

$$\frac{\gamma \vdash [p'] \Rightarrow [p], \gamma'}{\gamma \vdash p' \Rightarrow p, \gamma'} \quad \text{(Tr-P2)}$$

An auxiliary function, called "collectPatterns", is used in pattern translation. This function takes an extended pattern, a set of already used identifiers and a translation environment; and returns a core pattern and a new set of used identifiers.

This is the core of the pattern translation algorithm. If an identifier has already been used in a pattern (or list of patterns), the translation algorithm fails. If the identifier has not been used, a fresh identifier (guaranteed not to be in the environment) is generated. The original identifier is then associated to this new identifier and added to the environment.

$$\frac{x \notin \mathrm{id} \qquad x' \; is \; new}{\mathrm{collectPatterns}(x, \mathrm{id}, \gamma) = x', \mathrm{id} \cup \{x\}, \gamma \cup \{x \mapsto x'\}} \quad \text{(Tr-P-X)}$$

$$\frac{x \notin \mathrm{id} \qquad \gamma \vdash T' \Rightarrow T \qquad x' \; is \; new}{\mathrm{collectPatterns}(x : T', \mathrm{id}, \gamma) = x' : T, \mathrm{id} \cup \{x\}, \gamma \cup \{x \mapsto x'\}} \quad \text{(Tr-P-X2)}$$

Every other translation rule is a variation on iterating on sub-patterns to construct the list of used identifiers and final environment.

$$\mathrm{collectPatterns}(\_, \mathrm{id}, \gamma) = \_, \mathrm{id}, \gamma \quad \text{(Tr-P-Ignore)}$$

$$\frac{\gamma \vdash T' \Rightarrow T}{\mathrm{collectPatterns}(\_ : T', \mathrm{id}, \gamma) = \_ : T, \mathrm{id}, \gamma} \quad \text{(Tr-P-Ignore2)}$$

$$\frac{\mathrm{id}_0 = \mathrm{id} \qquad \forall i \in [1,\, n]\,.\,\mathrm{collectPatterns}(p'_i, \mathrm{id}_{i-1}, \gamma_{i-1}) = p_i, \mathrm{id}_i, \gamma_i}{\mathrm{collectPatterns}(con\; p'_1,\, \ldots\, p'_n, \mathrm{id}, \gamma) = con\; p_1,\, \ldots\, p_n, \mathrm{id}_n, \gamma_n} \quad \text{(Tr-P-Con)}$$

$$\frac{\mathrm{id}_0 = \mathrm{id} \qquad \gamma \vdash T' \Rightarrow T \qquad \forall i \in [1,\, n]\,.\,\mathrm{collectPatterns}(p'_i, \mathrm{id}_{i-1}, \gamma_{i-1}) = p_i, \mathrm{id}_i, \gamma_i}{\mathrm{collectPatterns}(con\ p'_1,\ \ldots\ p'_n : T', \mathrm{id}, \gamma) = con\ p_1,\ \ldots\ p_n : T, \mathrm{id}_n, \gamma_n} \text{ (Tr-P-Con2)}$$

$$\frac{\mathrm{id}_0 = \mathrm{id} \qquad \forall i \in [1,\, n]\,.\,\mathrm{collectPatterns}(p'_i, \mathrm{id}_{i-1}, \gamma_{i-1}) = p_i, \mathrm{id}_i, \gamma_i}{\mathrm{collectPatterns}(\{l_1 : p'_1,\ \ldots\ l_n : p'_n\}, \mathrm{id}, \gamma) = \{l_1 : p_1,\ \ldots\ l_n : p_n\}, \mathrm{id}_n, \gamma_n} \text{ (Tr-P-Record)}$$

$$\frac{\mathrm{id}_0 = \mathrm{id} \qquad \gamma \vdash T' \Rightarrow T \qquad \forall i \in [1,\, n]\,.\,\mathrm{collectPatterns}(p'_i, \mathrm{id}_{i-1}, \gamma_{i-1}) = p_i, \mathrm{id}_i, \gamma_i}{\mathrm{collectPatterns}(\{l_1 : p'_1,\ \ldots\ l_n : p'_n\} : T', \mathrm{id}, \gamma) = \{l_1 : p_1,\ \ldots\ l_n : p_n\} : T, \mathrm{id}_n, \gamma_n} \text{ (Tr-P-Record2)}$$

$$\frac{\mathrm{id}_0 = \mathrm{id} \qquad \forall i \in [1,\, n]\,.\,\mathrm{collectPatterns}(p'_i, \mathrm{id}_{i-1}, \gamma_{i-1}) = p_i, \mathrm{id}_i, \gamma_i}{\mathrm{collectPatterns}(\{l_1 : p'_1,\ \ldots\ l_n : p'_n, \ldots\}, \mathrm{id}, \gamma) = \{l_1 : p_1,\ \ldots\ l_n : p_n, \ldots\}, \mathrm{id}_n \gamma_n} \text{ (Tr-P-PartRec)}$$

$$\frac{\mathrm{id}_0 = \mathrm{id} \qquad \gamma \vdash T' \Rightarrow T \qquad \forall i \in [1,\, n]\,.\,\mathrm{collectPatterns}(p'_i, \mathrm{id}_{i-1}, \gamma_{i-1}) = p_i, \mathrm{id}_i, \gamma_i}{\mathrm{collectPatterns}(\{l_1 : p'_1,\ \ldots\ l_n : p'_n, \ldots\} : T', \mathrm{id}, \gamma) = \{l_1 : p_1,\ \ldots\ l_n : p_n, \ldots\} : T, \mathrm{id}_n, \gamma_n} \text{ (Tr-P-PartRec2)}$$

$$\frac{\mathrm{id}_0 = \mathrm{id} \qquad \forall i \in [1,\, n]\,.\,\mathrm{collectPatterns}(p'_i, \mathrm{id}_{i-1}, \gamma_{i-1}) = p_i, \mathrm{id}_i, \gamma_i \qquad p = ::\ p_1\ (::\ p_2\ \ldots (::\ p_n\ nil)\ldots)}{\mathrm{collectPatterns}([p'_1,\ \ldots\ p'_n], \mathrm{id}, \gamma) = p, \mathrm{id}_n, \gamma_n} \text{ (Tr-P-List)}$$

$$\frac{\mathrm{id}_0 = \mathrm{id} \qquad \gamma \vdash T' \Rightarrow T \qquad \forall i \in [1,\, n]\,.\,\mathrm{collectPatterns}(p'_i, \mathrm{id}_{i-1}, \gamma_{i-1}) = p_i, \mathrm{id}_i, \gamma_i \qquad p = ::\ p_1\ (::\ p_2\ \ldots (::\ p_n\ nil)\ldots)}{\mathrm{collectPatterns}([p'_1,\ \ldots\ p'_n] : T', \mathrm{id}, \gamma) = p : T, \mathrm{id}_n, \gamma_n} \text{ (Tr-P-List2)}$$

Typing Patterns Some expressions require that untyped patterns be transformed into typed patterns. For this, fresh variable types are generated.

$$\mathrm{typeP}(patt' : T', \gamma) = patt' : T' \qquad \text{(Typ-Patt)}$$

$$\frac{X \; is \; new}{\text{typeP}(patt', \gamma) = patt' : X} \quad \text{(Typ-Patt2)}$$

Function Translation Since functions have undergone massive changes from the core language, they need complex translation rules.

$$\frac{\begin{array}{c}\gamma \vdash [p'_1, \ldots p'_n] \Rightarrow [p_1, \ldots p_n], \gamma' \\ \forall i \in [1, n] \,.\, x_i \; is \; new \\ \gamma' \vdash e' \Rightarrow e \\ m = \texttt{match}\ (x_1, \ldots x_n)\ \texttt{with}\ (p_1, \ldots p_n) \rightarrow e \\ f = \texttt{fn}\ x_1 \Rightarrow \cdots \texttt{fn}\ x_n \Rightarrow m\end{array}}{\gamma \vdash (\texttt{fn}\ p'_1, \ldots p'_n \Rightarrow e') \Rightarrow f} \quad \text{(Tr-F-Fn)}$$

$$\frac{\begin{array}{c}\gamma \vdash [p'_1, \ldots p'_n] \Rightarrow [p_1, \ldots p_n], \gamma' \\ f' \; is \; new \qquad \forall i \in [1, n] \,.\, x_i \; is \; new \\ \gamma' \cup \{f \mapsto f'\} \vdash e' \Rightarrow e \\ m = \texttt{match}\ (x_1, \ldots x_n)\ \texttt{with}\ (p_1, \ldots p_n) \rightarrow e \\ fn = \texttt{rec}\ f'\ x_1 \Rightarrow \cdots \texttt{fn}\ x_n \Rightarrow m\end{array}}{\gamma \vdash (\texttt{rec}\ f\ \ p'_1, \ldots p'_n \Rightarrow e') \Rightarrow fn} \quad \text{(Tr-F-Rec)}$$

$$\frac{\begin{array}{c}\forall i \in [1,n] \,.\, \text{typeP}(p'_i, \gamma) = patt'_i : T'_i \\ T'' = T'_2 \rightarrow T'_3 \rightarrow \cdots \rightarrow T'_n \\ \gamma \vdash [patt'_1 : T'_1, \ldots p'_n : T'_n] \Rightarrow [p_1, \ldots p_n], \gamma' \\ f' \; is \; new \qquad \forall i \in [1, n] \,.\, x_i \; is \; new \\ \gamma' \cup \{f \mapsto f'\} \vdash e' \Rightarrow e \qquad \gamma' \vdash T'' \Rightarrow T \\ m = \texttt{match}\ (x_1, \ldots x_n)\ \texttt{with}\ (p_1, \ldots p_n) \rightarrow e \\ fn = \texttt{rec}\ f : T\ x_1 \Rightarrow \cdots \texttt{fn}\ x_n \Rightarrow m\end{array}}{\gamma \vdash (\texttt{rec}\ f : T'\ \ p'_1, \ldots p'_n \Rightarrow e') \Rightarrow fn} \quad \text{(Tr-F-RecT)}$$

When functions only have one parameter, the `match` expression does not use a tuple, since tuples must have at least 2 components.

$$\frac{\begin{array}{c}\gamma \vdash p' \Rightarrow p, \gamma' \qquad x \; is \; new \\ \gamma' \vdash e' \Rightarrow e \\ m = \texttt{match}\ x\ \texttt{with}\ p \rightarrow e \\ f = \texttt{fn}\ x \Rightarrow m\end{array}}{\gamma \vdash (\texttt{fn}\ p' \Rightarrow e') \Rightarrow f} \quad \text{(Tr-F-Fn1)}$$

$$\frac{\gamma \vdash p' \Rightarrow p, \gamma' \qquad x\ is\ new \qquad \gamma' \vdash e' \Rightarrow e \qquad m = \texttt{match}\ x\ \texttt{with}\ p \rightarrow e \qquad f = \texttt{rec}\ x \Rightarrow m}{\gamma \vdash (\texttt{rec}\ p' \Rightarrow e') \Rightarrow f} \quad \text{(Tr-F-Rec1)}$$

$$\frac{\text{typeP}(p', \gamma) = p'' \qquad \gamma \vdash p'' \Rightarrow p, \gamma' \qquad x\ is\ new \qquad \gamma' \vdash e' \Rightarrow e \qquad \gamma' \vdash T' \Rightarrow T \qquad m = \texttt{match}\ x\ \texttt{with}\ p \rightarrow e \qquad f = \texttt{rec}\ x : T \Rightarrow m}{\gamma \vdash (\texttt{rec}\ p' : T' \Rightarrow e') \Rightarrow f} \quad \text{(Tr-F-RecT1)}$$

As a further efficiency improvement, that `match` expression is only created when the parameters of the function are patterns. If every parameter of the function is a regular identifier (without type information), then no `match` expression is necessary. This is only done if the function does not specify a return type.

$$\frac{\gamma \vdash [x'_1, \ldots x'_n] \Rightarrow [x_1, \ldots x_n], \gamma' \qquad \gamma' \vdash e' \Rightarrow e \qquad f = \texttt{fn}\ x_1 \Rightarrow \cdots \texttt{fn}\ x_n \Rightarrow e}{\gamma \vdash (\texttt{fn}\ x'_1, \ldots x'_n \Rightarrow e') \Rightarrow f} \quad \text{(Tr-F-Fn2)}$$

$$\frac{\gamma \vdash [x'_1, \ldots x'_n] \Rightarrow [x_1, \ldots x_n], \gamma' \qquad f'\ is\ new \qquad \gamma' \cup \{f \mapsto f'\} \vdash e' \Rightarrow e \qquad fn = \texttt{rec}\ f'\ x_1 \Rightarrow \cdots \texttt{fn}\ x_n \Rightarrow e}{\gamma \vdash (\texttt{rec}\ f\ \ x'_1, \ldots x'_n \Rightarrow e') \Rightarrow fn} \quad \text{(Tr-F-Rec2)}$$

Declaration Translation The translation of declarations works differently from other translations. The result of translating a declaration is a set of associations between patterns and expressions; along with an updated translation environment.

$$\frac{\gamma \vdash p' \Rightarrow p, \gamma' \qquad \gamma \vdash e' \Rightarrow e}{\gamma \vdash (p' = e') \Rightarrow \{p \mapsto e\}, \gamma'} \quad \text{(Tr-Decl)}$$

$$\frac{\gamma \vdash T' \Rightarrow T}{\gamma \vdash \texttt{type alias}\ \tau = T' \Rightarrow \{\}, \gamma \cup \{\tau \mapsto T\}} \quad \text{(Tr-Decl-Alias)}$$

$$\frac{\begin{array}{c}\gamma \vdash (\texttt{fn}\ p'_1, \ldots\ p'_n \Rightarrow e') \Rightarrow e \\ f' \ is \ new\end{array}}{\gamma \vdash f\ p'_1, \ldots\ p'_n = e' \Rightarrow \{f' \mapsto e\}, \gamma \cup \{f \mapsto f'\}} \quad \text{(Tr-Decl-Func)}$$

$$\frac{\begin{array}{c}\gamma \vdash (\texttt{fn}\ p'_1, \ldots\ p'_n \Rightarrow e') \Rightarrow e \\ f' \ is \ new\end{array}}{\gamma \vdash f : T'\ p'_1, \ldots\ p'_n = e' \Rightarrow \{f' \mapsto e\}, \gamma \cup \{f \mapsto f'\}} \quad \text{(Tr-Decl-Func2)}$$

$$\frac{\begin{array}{c}\gamma \vdash (\texttt{rec}\ f\ p'_1, \ldots\ p'_n \Rightarrow e') \Rightarrow e \\ f' \ is \ new\end{array}}{\gamma \vdash \texttt{rec}\ f\ p'_1, \ldots\ p'_n = e' \Rightarrow \{f' \mapsto e\}, \gamma \cup \{f \mapsto f'\}} \quad \text{(Tr-Decl-Rec)}$$

$$\frac{\begin{array}{c}\forall i \in [1, n]\ .\ \text{typeP}(p'_i, \gamma) = patt'_i : T'_i \\ T'' = T'_1 \to T'_2 \to \cdots \to T'_n \\ \gamma \vdash (\texttt{rec}\ f : T'\ patt'_1 : T'_1, \ldots\ patt'_n : T'_n \Rightarrow e') \Rightarrow e \\ f' \ is \ new\end{array}}{\gamma \vdash \texttt{rec}\ f : T'\ p'_1, \ldots\ p'_n = e' \Rightarrow \{f' : T'' \mapsto e\}, \gamma \cup \{f \mapsto f'\}} \quad \text{(Tr-Decl-Rec2)}$$

Dot Translation To properly translate the accessor and dot syntax expressions introduced in the extended language, it is necessary to define translations for both acc and dot expressions.

When translating an acc, the result is an expression e. The rules governing this translation are given below.

$$\gamma \vdash l \Rightarrow \#l \quad \text{(Tr-Acc-Label)}$$

$$\gamma \vdash\ 'x \Rightarrow x \quad \text{(Tr-Acc-'x)}$$

$$\frac{\forall i \in [1, n]\ .\ \gamma \vdash dot_i \Rightarrow e_i}{\gamma \vdash (dot_1, \ldots\ dot_n) \Rightarrow \#(e_1, \ldots\ e_n)} \quad \text{(Tr-Acc-Joined)}$$

Similarly, translating a dot also yields an expression e. The two rules defining this translation are given below.

$$\frac{\gamma \vdash acc \Rightarrow e}{\gamma \vdash acc \Rightarrow e} \quad \text{(Tr-Dot-Stacked)}$$

$$\frac{\gamma \vdash acc \Rightarrow e_1 \qquad \gamma \vdash dot \Rightarrow e_2}{\gamma \vdash acc\ .\ dot \Rightarrow \texttt{stack}\ e_1\ e_2} \quad \text{(Tr-Dot-Stacked)}$$

Using the translations defined above, defining the translation of the accessor and dot access expressions is trivial:

$$\frac{\gamma \vdash dot \Rightarrow e}{\gamma \vdash \#dot \Rightarrow e} \quad \text{(Tr-E-Acc)}$$

$$\frac{\gamma \vdash dot \Rightarrow e}{\gamma \vdash x\ .\ dot \Rightarrow \texttt{get}\ e\ x} \quad \text{(Tr-E-Dot)}$$

Record Update Translation

$$\frac{x\ is\ new \qquad f' = (\texttt{fn}\ x \Rightarrow x)}{\text{concat}(\varnothing, \gamma) = f', \gamma} \quad \text{(Tr-Updates-Empty)}$$

$$\frac{x\ is\ new \qquad f' = (\texttt{fn}\ x \Rightarrow \texttt{set}\ \#dot\ e'\ x) \qquad \text{concat}(updates, \gamma) = g', \gamma'}{\text{concat}(dot \leftarrow e' :: updates, \gamma) = g'\ .\ f', \gamma'} \quad \text{(Tr-Updates-Set)}$$

$$\frac{x\ is\ new \qquad f' = (\texttt{fn}\ x \Rightarrow \texttt{modify}\ \#dot\ e'\ x) \qquad \text{concat}(updates, \gamma) = g', \gamma'}{\text{concat}(dot \leftsquigarrow e' :: updates, \gamma) = g\ .\ f', \gamma'} \quad \text{(Tr-Updates-Modify)}$$

$$\frac{\text{concat}(updates, \gamma) = g', \gamma'}{\text{concat}(decl' :: updates, \gamma) = \texttt{let}\ decl'\ \texttt{in}\ g', \gamma'} \quad \text{(Tr-Updates-Decl)}$$

$$\frac{\text{concat}(updates) = f', \gamma' \qquad \gamma' \vdash f' \Rightarrow f}{\gamma \vdash \texttt{update}\ updates \Rightarrow f} \quad \text{(Tr-E-Update)}$$

Expression Translation The conditional expression translates into a `match` expression, testing whether the first sub-expression (e_1) is $true$ or $false$ and evaluation e_2 or e_3, respectively.

$$\frac{\gamma \vdash e'_1 \Rightarrow e_1 \qquad \gamma \vdash e'_2 \Rightarrow e_2 \qquad \gamma \vdash e'_3 \Rightarrow e_3 \qquad e = \texttt{match}\ e_1\ \texttt{with}\ true \rightarrow e_2,\ false \rightarrow e_3}{\gamma \vdash \texttt{if}\ e'_1\ \texttt{then}\ e'_2\ \texttt{else}\ e'_3 \Rightarrow e} \quad \text{(Tr-E-Cond)}$$

The list expression translates into nested applications of the :: constructor, ending with the nil (empty list) constructor.

$$\frac{\forall i \in [1,\ n]\ .\ \gamma \vdash e'_i \Rightarrow e_i \qquad e = ::\ e_1\ (::\ e_2\ \ldots (::\ e_n\ nil)\ldots)}{\gamma \vdash [e'_1,\ \ldots\ e'_n] \Rightarrow e} \quad \text{(Tr-E-List)}$$

Similarly, the tuple expression translates into a complete application of a tuple (Tuple n) constructor, where n is the number of elements in the expression.

$$\frac{\forall i \in [1,\ n]\ .\ \gamma \vdash e'_i \Rightarrow e_i \qquad e = (\ \ldots\ (\text{Tuple}\ n\ e_1)\ e_2)\ \ldots e_n)}{\gamma \vdash (e'_1,\ \ldots\ e'_n) \Rightarrow e} \quad \text{(Tr-E-Tuple)}$$

When translating `let` expressions, the declaration is translated into an ordered set of associations between patterns and expressions. Nested `let` expressions are then created with these associations.

$$\frac{\gamma \vdash decl \Rightarrow \{p_1 \mapsto e_1,\ \ldots\ p_n \mapsto e_n\}, \gamma' \qquad \gamma' \vdash e' \Rightarrow e \qquad ret = (\texttt{let}\ p_1 = e_1\ \texttt{in} \cdots \texttt{let}\ p_n = e_n\ \texttt{in}\ e)}{\gamma \vdash \texttt{let}\ decl\ \texttt{in}\ e' \Rightarrow ret} \quad \text{(Tr-E-Let)}$$

There are two variations of ranges: one with an implicit step and one with an explicit step. Both of these rely on the existing of the function "range", which, when given a starting number, ending number and step, returns a list with the numbers.

The first variation uses a fixed step value of 1, while the second variation calculates its step by sutracting the second element of the range (e_2) from the first element (e_1).

$$\frac{\gamma \vdash e'_1 \Rightarrow e_1 \qquad \gamma \vdash e'_2 \Rightarrow e_2}{\gamma \vdash [e'_1\ ..\ e'_2] \Rightarrow \text{range}\ e_1\ e_2\ 1} \quad \text{(Tr-E-Range)}$$

$$\frac{\gamma \vdash e'_1 \Rightarrow e_1 \qquad \gamma \vdash e'_2 \Rightarrow e_2 \qquad \gamma \vdash e'_3 \Rightarrow e_3 \qquad i = -e_2\ e_1}{\gamma \vdash [e'_1,\ e'_2\ ..\ e'_3] \Rightarrow \text{range}\ e_1\ e_2\ i} \quad \text{(Tr-E-Range2)}$$

Comprehensions, similarly to ranges, rely on the function "map". A function is created with the pattern p' and the body e'_1. This function is then translated and passed as the first argument of the "map" function. The second argument of the function is the translation of the expression e'_2, which will eventually evaluate into a list.

$$\frac{\gamma \vdash (\texttt{fn}\ p' \Rightarrow e'_1) \Rightarrow f \qquad \gamma \vdash e'_2 \Rightarrow e_2}{\gamma \vdash [e'_1\ \text{for}\ p'\ \text{in}\ e'_2] \Rightarrow \text{map}\ f\ e_2} \quad \text{(Tr-E-Comprehension)}$$

Do Notation Translation

$$\text{concat}(e', \gamma) = e', \gamma \quad \text{(Tr-Do-Base)}$$

$$\frac{\text{concat}(doTerms, \gamma) = f', \gamma'}{\text{concat}(p' \leftarrow e' :: doTerms, \gamma) = \text{bind}\ e'\ (\texttt{fn}\ p' \Rightarrow f'), \gamma'} \quad \text{(Tr-Do-Bind)}$$

$$\frac{\text{concat}(doTerms, \gamma) = f', \gamma'}{\text{concat}(e' :: doTerms, \gamma) = \text{bind}\ e'\ (\texttt{fn}\ \_ \Rightarrow f'), \gamma'} \quad \text{(Tr-Do-Term)}$$

$$\frac{\text{concat}(doTerms, \gamma) = f', \gamma'}{\text{concat}(decl' :: doTerms, \gamma) = \texttt{let}\ decl'\ \texttt{in}\ f', \gamma'} \quad \text{(Tr-Do-Decl)}$$

$$\frac{\text{concat}(doTerms) = f', \gamma' \qquad \gamma' \vdash f' \Rightarrow f}{\gamma \vdash \texttt{do}\ doTerms \Rightarrow f} \quad \text{(Tr-E-Do)}$$

APPENDIX E — ACCESSORS IN HASKELL

```haskell
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FunctionalDependencies #-}

-- This would be included in an accessor library (or be built-in)
type Accessor rec field = rec -> field -> (field, rec)

get :: Accessor rec field -> rec -> field
get acc rec = fst $ acc rec undefined

set :: Accessor rec field -> field -> rec -> rec
set acc value rec = snd $ acc rec value

stack :: Accessor rec field1 -> Accessor field1 field2 -> Accessor rec field2
stack acc1 acc2 recOuter field =
  let (recInner, recOuter') = acc1 recOuter recInner'
      (value, recInner') = acc2 recInner field
  in (value, recOuter')

distort :: Accessor r f -> (f -> t) -> (t -> f -> f) -> Accessor r t
distort acc getter setter rOld tNew =
  let (fOld, rNew) = acc rOld fNew
      fNew = setter tNew fOld
  in (getter fOld, rNew)
-- Here ends the accessor library
-- The code below is to illustrate how it would be used

-- A simple record
data Record = R {code :: String, age :: Int} deriving Show

-- This code would be generated
class CodeLabel rec field | rec -> field where
  codeAcc :: Accessor rec field

instance CodeLabel Record String where
  codeAcc r@(R code _) string = (code, r { code = string })
-- End of generated code
```

```haskell
-- Another record with the same field name
-- (the 2 is added because of Haskell's current restriction on field names)
data Record2 = R2 {code2 :: String, age2 :: Double} deriving Show

-- This code would be generated
instance CodeLabel Record2 String where
  codeAcc r@(R2 code _) string = (code, r { code2 = string })
-- End of generated code

-- Record with another record as a field
data Outer = O { r :: Record, name :: String } deriving Show

-- This would be generated
class RLabel rec field | rec -> field where
    rAcc :: Accessor rec field

instance RLabel Outer Record where
  rAcc o@(O r _) record = (r, o { r = record })
-- End of generated code

-- Example values
r1 = R "1234" 22
r2 = R2 "1234" 22.0
o1 = O r1 "name"

-- Examples
-- get (stack rAcc codeAcc) o1
-- set codeAcc "3" $ get rAcc o1
-- codeAcc' = distort codeAcc read (\t _ -> show t)
-- set (stack rAcc codeAcc') 3 o1
```