Revista de Informática Teórica e Aplicada - RITA - ISSN 2175-2745
Vol. 25, Num. 3 (2018) 89-101

RESEARCH ARTICLE

V: a language with extensible record accessors and a trait-based type system

V: uma linguagem com acessores de registro extensíveis e um sistema de tipos baseado em traits

Arthur G. Vedana[1]*, Rodrigo Machado[1], Álvaro Moreira[1]

Abstract: This article introduces the V language, a purely functional programming language with a novel approach to records. Based on a system of type traits, V attempts to solve issues commonly found when manipulating records in purely functional programming languages.

Keywords: Functional Programming Languages — Records — Traits

Resumo: Este artigo introduz a linguagem V, uma linguagem de programação puramente funcional com uma nova abordagem a registros. Usando um sistema de *traits* de tipos, V tenta resolver problemas comumente encontrados ao manipular registros em linguagens puramente funcionais.

Palavras-Chave: Linguagens de Programação Funcionais — Registros — Traits

[1] *Instituto de Informática, UFRGS, Brasil*
*__Corresponding author__: agieselvedana@gmail.com
DOI: https://doi.org/10.22456/2175-2745.82772 • **Received:** 11/05/2018 • **Accepted:** 26/08/2018

1. Introduction

Functional programming languages, or simply functional languages, are the ones in which programs are primarily defined by means of the definition and combination of functions. Haskell[1], Ocaml[2], Scheme[3], F#[4], Elm[5], and Scala[6] are examples of such languages. Using features such as higher-order functions and currying, functional languages can express complex operations in (generally) fewer lines of code than imperative or object-oriented ones. Also, a strong type system, featured in many functional languages, allows the development of code that is free from some kinds of errors.

Purely functional languages are the ones in which referential transparency holds, meaning that the result of a function application is uniquely defined by the its arguments. There are fewer languages with this property, from which we can mention Haskell and Elm. A consequence of referential transparency is the absence of implicit side effects during function evaluation, which means, among other things, that one cannot modify values stored in memory. Because of this, purely functional languages are known for not being as convenient for dealing with records as languages that allow side effects. This issue is so serious that, at least in Haskell, the use of a library to work with large nested records is practically required (details are presented in Section 2). Although many different approaches have been suggested to mitigate this problem, these are usually limited. For instance, the popular Lenses[7] 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#. Both specification and implementation can be found on the project page on Github[1]. This paper presents the current status of the language and, in particular, the record subsystem and how traits are used in the type system.

The structure of this article is as follows. In Section 2, the current state of records in purely functional languages is introduced, showing what can be improved by the approach taken in V. In Section 3, records and accessors in the V language are introduced. In Section 4, the mechanism that supports polymorphic accessors, Traits, is defined. In Section 5, the V language as a whole is introduced, explaining its characteristics, type system and operational semantics. In Section 6, we discuss how V's accessor system could be incorporated into other languages. In Section 7, we discuss related work. Finally, in Section 8, the current state of the language is exposed, along with known limitations and future work.

2. Motivation: Existing Record Systems

Purely functional languages are known for not providing an easy way to manipulate records. To exemplify these difficulties, we present a brief description of how Haskell and Elm deal with record creation and manipulation.

Haskell In the Haskell language, the following code declares a new named record type `A` with two fields (`a` and `b`) of type `Int`, together with a value `x` of type `A`. Based on the data declaration, the compiler creates two *getter* functions `a :: A→Int` and `b :: A→Int` in the same scope.

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.

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.

This syntax is not first-class, meaning that updates to records cannot be bound to identifiers, passed as arguments or returned as functions. This restriction limits the use of the syntax, since one cannot compose updates or perform multiple updates directly.

Another issue in Haskell is the cumbersome notation 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.

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

Elm Elm's records are anonymous, meaning that both record values and record types are defined without an explicit name. That being said, it is possible to associate a name to a type (known as a type alias) for legibility and documentation purposes.

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` for the getter function `.a` applied to 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:

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 update fields by a pipe (`|`).

This syntax, however, has a limitation that makes updating nested fields even more cumbersome than Haskell: it only accepts names before the pipe (in other words, one cannot write

in Elm). This means that, to update an inner field, it is necessary to first bind the inner record to a name.

The V language attempts to improve the record access and update mechanism by introducing polymorphic record accessors. These accessors are used for both getting and setting the value of a field, removing the necessity of a custom syntax for record updates. Accessors in V are polymorphic, meaning that the same accessor can be used with multiple records (and, unlike Haskell, there is no issue of namespaces). Furthermore, V accessors are extensible in the sense that it is possible to combine them by means of operations named `stack`, `join` and `distort`, which will be subsequently introduced. Finally, since accessors in V are first-class (unlike the update syntax in Elm and Haskell), they can be passed as arguments and returned from functions, which allows great flexibility in their use.

3. Record System in V

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.

[1] ⟨https://github.com/AvatarHurden/V⟩

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, such as is required 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 and requires exposing intermediate updates, as illustrated by the following example, in which the inner field `name` is updated to `"Savior"`.

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

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

let oldPlayer = get #player game;

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

By using 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. Using this function with accessors `acc1` and `acc2` means that `acc1` is an accessor that refers to a field which itself contains a record, and this record must have a field with the label specified by `acc2`.

The following code presents how a stacked accessor is used to perform the same update shown in the previous example. As can be seen, the record `game` has a field `player`, which is a record. This means that the `#player` accessor can be stacked with the `#name` accessor to create a new accessor that goes directly to the `name` subfield of the `player` field.

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

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

let playerName = stack #player #name;

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

The `stack` function can be used as many times as is required, allowing to focus on an inner field no matter how deep it is located within a nested record structure. Each application of the `stack` operations indicates that the structure being manipulated has one extra level of inner records.

As an example, suppose that the `name` field of the `player` record is changed into a record that separates first names from surnames. In this scenario, using the `playerName` accessor would give us the full record of the hero's name. We can, however, `stack playerName` with the `#surname` accessor to

access the `surname` field, which is nested three levels deep in the `game` record.

```
let heroName = { firstName: "Mighty",
                 surname: "Hero"};

let player = { name: heroName,
               level: 6,
               health: 100 };

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

let playerName = stack #player #name;
let playerSurname = stack playerName #surname;

get playerSurname game
// "Hero"
```

As a side-note, `stacking` is associative, meaning that it does not matter how `stack` operations are grouped. This means that `playerSurname` and `playerSurname'`, defined below, are exactly the same, and `stacking` can be done either from inner to outer fields or vice-versa.

```
let playerSurname = stack #player (stack #name #surname);
let playerSurname' = stack (stack #player #name) #surname;
```

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 the current 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 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. The `getter` function is defined as taking the head (first element) of the list. This means that, when using the `get` function with the distorted accessor, only the first enemy of the list will be retrieved.

The `modifier` function is defined in terms of the existing list of enemies and the new enemy being provided (`enemy'`). Since the accessor is supposed to replace the first enemy of the list, it first removes the first element of the existing enemy list (`tail enemies`), and then adds the new enemy to the front of this list. This makes it so that, when using the `set` function with a single enemy, it can be transformed into the correct list of enemies to be stored in the `enemies` field.

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

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

let getter enemies     = head enemies;

let modifier enemy' enemies = enemy' :: (tail enemies);

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 };
```

Infix Operators Although the functions and expressions shown above provide all the functionality needed to manipulate records and accessors, using prefixed functions is not ideal. For this reason, infix versions of each expression are provided, aiming at a simpler and easier to understand syntax. The following table presents the current infix syntax for the operations responsible for manipulating accessors.

Prefix	Infix
`get #field record`	`record .^ #field`
`set #field v record`	`record & #field ^= v`
`modify #field f record`	`record & #field ^~ f`
`stack #outer #inner`	`#outer :. #inner`
`distort #field f g`	`#field ~. (f, g)`

In the code above, the `&` operator allows inverse application (i.e `x & f` is equivalent to `f x`). This allows composing multiple setting or modifying operations on the same record, as the example below shows:

```
let player' = player
                & #level   ^= 8
                & #health  ^~ (\x -> x + 30);

// player' = { name: "Hero",
//             level: 8,
//             health: 130}
```

This allows much cleaner code when using multiple accessors on a single record, removing the necessity of creating intermediate bindings for every accessor used.

Another positive aspect is the flexibility obtained by allowing arbitrary functions to manipulate accessors. As an example, below are functions to increase the `level` field by one and set the `health` field to 100:

```
let increaseLevel = #level ^~ (\x -> x + 1);

let restoreHealth = #health ^= 100;
```

These functions can then be joined by using simple function composition (. operator), allowing the creation of composable operations easily.

```
let levelUp = increaseLevel . restoreHealth;

let r = { health: 20,
          level: 6 };

levelUp r
// { health: 100,
//   level:  7 }
```

Maintaining the thematic of a game, Figure 1 presents a more complete code example. A game state with one player and multiple enemies is defined, together with functions to update it. 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,...`}.

4. Types and Traits

This section introduces the types and traits available in V.

Types A fragment of the type language is shown below:

$$
\begin{array}{lcll}
T & ::= & \text{Int} & \\
 & | & \text{Bool} & \\
 & | & (T_1, \ldots T_n) & (n \geq 2) \\
 & | & T_1 \to T_2 & \\
 & | & \{l_1 : T_1, \ldots l_n : T_n\} & (n \geq 1) \\
 & | & T_1 \# T_2 & \textit{Accessor} \\
 & | & \ldots & \\
 & & & \\
l, l_1, l_2, \ldots & \in & \textit{Label} &
\end{array}
$$

Types include integers, booleans, tuples, functions, and record types, as they are usually presented in other functional languages. The set *Label* consists of a countable collection of field names for records. 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` :: $T_1 \# T_2 \to T_2 \to (T_1 \to T_1)$

For the sake of readability, the fragment above does not include several primitive types. One important omission, type variables, will be addressed in Section 5.4.

Traits The term trait was first introduced by [8] to denote a parent 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[9] and Scala[6]. Other languages have different names for similar tools, such as interfaces in Java[10] 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[11][12].

In this article, traits are used to support ad-hoc polymorphism (or overloading). A trait defines one or more behavior

```
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 =
  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 game =
  game & #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 accessor = #(#player, #enemies ~. (getter, modifier));
  game & accessor ^~ attack;
```

Figure 1. A simple game in V.

or characteristic (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, indicated by the $\in$ symbol.

The formation of traits is presented by the following syntax:

$$\begin{array}{rcll} Trait & ::= & Equatable & \\ & | & Orderable & \\ & | & \{l : T\} & (Record\ Label) \end{array}$$

The traits *Equatable* and *Orderable* are used for equality (= and $\neq$) and comparison ($\geq, >, <, \leq$) operations, respectively. The other traits, named *Record Label*, specifies a single 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 tested for equality, we have the following axiom.

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

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 $(T_1 \to T_2) \notin Equatable$ 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.

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

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).

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

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. We discuss one of the distinctions in Section 8.

5. The V Language

A reduced version of the V language is used is this article, containing all the basic elements necessary to define and use traits and records. This section presents aspects of the syntax, operational semantics and type system of the reduced V language. For information on the full language, see Section 8.

5.1 General Characteristics

V is a purely functional strict general purpose programming language with strong inspirations from Haskell. It has a strong and static type system, with support for parametric polymorphism through a Hindley-Milner style let-polymorphism[13]; and ad-hoc polymorphism, or overloading, through traits. Complete type inference is supported for every expression, although a few expressions allow explicit type annotations.

5.2 Syntax

Figure 2 shows the abstract syntax of the reduced V language. The first few expressions are common to most functional programming language. In order, we have integers 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 [13]. The *builtin* meta-variable refers to any function that is directly built into the language. This reduced language 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.

$$
\begin{array}{lcll}
e & ::= & n & \\
 & | & (e_1, \ldots e_n) & (n \geq 2) \\
 & | & x & \\
 & | & \texttt{let}\ p = e_1\ \texttt{in}\ e_2 & \\
 & | & \texttt{fn}\ x \Rightarrow e & \\
 & | & \texttt{rec}\ f\ x \Rightarrow e & \\
 & | & e_1\ e_2 & \\
 & | & \texttt{match}\ e\ \texttt{with} & \\
 & & match_1, \ldots, match_n & (n \geq 1) \\
 & | & error & \\
 & | & builtin & \\
 & | & \{l_1 : e_1, \ldots l_n : e_n\} & (n \geq 1) \\
 & | & \#l & \\
 & | & \#(e_1, \ldots e_n) & (n \geq 2) \\
x, x_0, x_1, \ldots & \in & Ident & \text{(set of identifiers)} \\
l, l_1, l_2, \ldots & \in & Label & \text{(set of record labels)}
\end{array}
$$

$$
\begin{array}{lcll}
match & ::= & p \rightarrow e & \\
 & | & p\ \texttt{when}\ e_1 \rightarrow e_2 & \\
 & & & \\
builtin & ::= & \texttt{get} & \text{(binary)} \\
 & | & \texttt{set} & \text{(ternary)} \\
 & | & \texttt{stack} & \text{(binary)} \\
 & | & \texttt{distort} & \text{(ternary)} \\
 & | & = & \text{(binary)} \\
 & & & \\
p & ::= & x & \\
 & | & \_ & \\
 & | & n & \\
 & | & (p_1, \ldots p_n) & (n \geq 2) \\
 & | & \{l_1 : p_1, \ldots l_n : p_n\} & (n \geq 1) \\
 & | & \{l_1 : p_1, \ldots l_n : p_n, \ldots\} & \\
 & & & \text{(partial record}, n \geq 1)
\end{array}
$$

Figure 2. The V syntax.

Patterns Both `match` and `let` expressions use patterns, which are denoted by the letter p in figure 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 third is an integer pattern; and the fourth is the tuple pattern, which needs to be matched recursively. The last two patterns, record and partial record, are noteworthy. Both match record expressions, 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.

5.3 Operational 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, tuples and records), closures and accessors are also considered values.

Environment and Values The definitions for environments and values are depicted in Figure 3. We employ the notation $\bar{x}$ to refer to an unordered collection of elements x. Since V has static scope, closures $\langle x, e, env \rangle$ represent a function $\texttt{fn}\ x \Rightarrow e$ 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 $\langle\!\langle\, builtin\ .\ v_1, \ldots v_n \,\rangle\!\rangle$ 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.

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

Figure 3. Environment and Values

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. Section 8 contains a

link to 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.

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

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

Pattern Matching A new identifier association 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 the link available in Section 8.

Below is the matching rule for the simple x pattern.

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

If a match fails, it is represented by the negation of a match.

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

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*.

$$\frac{\text{env} \vdash e_1 \Downarrow v \qquad match(p, v) = env_1 \qquad env_1 \cup \text{env} \vdash e_2 \Downarrow v_2}{\text{env} \vdash \texttt{let}\ p = e_1\ \texttt{in}\ e_2 \Downarrow v_2} \text{(BS-LET)}$$

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

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

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

$$\frac{\text{env} \vdash e_1 \Downarrow \langle\!\langle\, \text{stack}\ .\ \#path_1 \,\rangle\!\rangle \qquad \text{env} \vdash e_2 \Downarrow \#path_2}{\text{env} \vdash e_1\ e_2 \Downarrow \#path_1\ .\ path_2} \text{(BS-STACKED)}$$

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

`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.

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

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

These rules use a helper 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.

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

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.

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

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.

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

5.4 Type System

Most of the basic characteristics of the type system, such as the available types, traits and their relationships, have already been described in Section 4. This section will give more specific characteristics of the system, such as the environment and some typing rules.

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). As was explained in section 4, the same system of traits creates the possibility of makeshift structural subtyping.

Type Variables In section 4, a fragment of the syntax of types was shown. However, a type needed for the type inference system was omitted: the type variable, 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

$$\begin{array}{lcll} T & ::= & \cdots & \\ & | & X^{\overline{Trait}} & \\ & & & \\ X, X_1, X_2, \ldots & \in & TypeVar & \text{(set of type variables)} \end{array}$$

Type Inference For the type inference system, 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 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. 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 between identifiers and type associations. The definition of the environment, along with the two variations of type associations, is given below.

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

Constraints are simply equations between two types. This allows creating exact match between two types, as is necessary when making sure the first term of an application is a function, for example.

When it comes to trait conformance, however, exact matching would not give the expect result. 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 exception to this is 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 $=$. First, T_1 and T_2 are obtained as the types of e_1 and e_2, respectively. The assertion

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

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*.

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

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

$$\frac{X_1 \textit{ is new} \qquad X_2^{\{\{l:T_1\}\}} \textit{ is new}}{\Gamma \vdash \#l : X_2^{\{\{l:T_1\}\}} \# X_1 \mid \{\}} \text{(T-LABEL)}$$

$$\frac{X_1 \textit{ is new} \qquad X_2 \textit{ is new} \qquad X_3 \textit{ is new}}{\Gamma \vdash \texttt{stack} : X_1 \# X_2 \to X_2 \# X_3 \to X_1 \# X_3 \mid \{\}} \text{(T-STACK)}$$

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

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

$$\frac{X_1 \textit{ is new} \qquad X_2 \textit{ is new}}{\Gamma \vdash \texttt{get} : X_2 \# X_1 \to X_2 \to X_1 \mid \{\}} \text{(T-GET)}$$

$$\frac{X_1 \textit{ is new} \qquad X_2 \textit{ is new}}{\Gamma \vdash \texttt{set} : X_2 \# X_1 \to X_1 \to X_2 \to X_2 \mid \{\}} \text{(T-SET)}$$

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 produces a substitution σ as output. A substitution is a mapping from variable types to types.

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

For most types, the unification process is straightforward: if the types are the same, they are considered unified. If they are composed types (such as tuples and records), constraints between their components are added to the end of the constraint list, and they are considered unified. 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 step, 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. It is defined with rules of the form:

$$\sigma(T) = T'$$

5.5 Extended Language

As a way to facilitate actual programming in V, an extended language (syntactic sugar) is defined on top of the basic 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.

Some of the features that this extended language provides are type aliases (renaming existing types to simplify type declarations), multi-parameter functions with pattern matching and multiple kinds of declarations. More details about what expressions are available, along with the translation algorithm from the extended to the core language, can be found in the online documentation[2].

6. Incorporating the accessor system in other languages

We believe that the record system of V is interesting and 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 does not allow this currently 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 [14], 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, we show that the compiler would need to generate for creating an accessor for the field `code` in the

[2] https://github.com/AvatarHurden/V-Documentation

`Record` type (the same would need to be done for the `age` field).

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 declared. 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`, and other functions could be implemented just as easily.

7. Related Work

There are a few proposals to improve record manipulation in Haskell. The most well-known of these is the Lenses library, which provides easily composable functions to access and modify fields of records. Another proposal is the Record Access approach [14]. These approaches provide most of the same functionalities of V's accessors, but they are still built upon the limitations imposed by the Haskell language.

MLSub [15] supports structural subtyping for record types. Structural subtyping and V's traits achieve the same results when it comes to records, and our option for traits was due to its simplicity when compared to subtyping.

Elm Records are extensible by default. This means that functions that receive records only ever care about the fields they specify and will always accept records with extra fields.

Furthermore, Elm allows creating type aliases for "partial records", i.e. records with *at least* the specified fields. This is interesting, because it allows a clean composition for specifying the fields that a parameter must have (or simply to specify the type of a value).

Even though Haskell is not ideal with nested records, Elm does not to seem to be much better in this regard, as seen in Section 2. The update syntax does not allow arbitrary expressions in the "source" record, only simple variables. This means that, when updating subfields, it is necessary to create a new variable for every level of the hierarchy.

V accessors are inspired by the more general concept of views in databases. As far as we know, the first reference to views as a way to view one type as another was made by [16]. In that paper, the author introduces a mechanism to create an abstract type that, when used, is converted to a real type. This mechanism is comparable to the "distort" function on V accessors, allowing a field value to be viewed as a different value. Both mechanisms use a pair of functions to convert from one value to the other. In the case of Wadler, however, the type defined by the view is abstract, and any use of a value of that type is automatically converted to a value of the real type.

Bidirectional programming [17][18] works with the same principles as V accessors, allowing secure manipulating of the internal structure of data. The data and possible manipulations are limited to database records [19] and textual data [20]. Given the limited scope of the types a value can take, it is possible to design accessor manipulators that always maintain certain properties, such as complete bijection of every manipulation. In a general-purpose programming language, however, these properties are harder, if not impossible, to guarantee. With the "distort" accessor manipulator, for example, it would be necessary to determine that two arbitrary functions are inverses of each other. V limits itself to guaranteeing the correct type for these functions, passing the responsibility of creating inverse functions to the programmer.

8. Current Status and Future Work

The complete language and documentation can be found in the following repositories.

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

V is still under development. We believe that there is still room for improvements in the syntax adopted for stacking, distorting, getting, setting, and updating. Some option to be considered are: dot-syntax for accessing fields of a record; do-style notation for editing multiple fields; and left-arrow (←) notation for setting field values.

Currently, the language has no support for IO of any kind. Different approaches are being investigated to add input and output while still maintaining the purity of the language. The current Haskell approach of using an IO Monad is a good candidate.

As already mentioned before, our choice of traits to allow polymorphic accessors was due to our desire to keep things as simple as possible when focusing on records. 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 are not 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.

Author contributions

Arthur Vedana has contributed with the original conception of the language V, including the idea of a new approach for

record manipulation with extensible accessors and the adoption of polymorphism based on traits, along with the implementation of its interpreter.

The design and implementation of the language evolved into a graduation project advised by Rodrigo Machado and Alvaro Moreira. Besides contributing to the design of the language, they helped to improve the writing and presentation of this text.

References

[1] MARLOW, S. *Haskell 2010 Language Report*. 2012.

[2] REMY, D.; VOUILLON, J. Objective ML: A simple object-oriented extension of ML. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 1997. (POPL, '97).

[3] ABELSON, H. et al. Revised^5 report on the algorithmic language scheme. *Higher-order and symbolic computation*, v. 11, n. 1, p. 7–105, 1998.

[4] SYME, D. et al. *The F# 2.0 language specification*. Redmond, USA, 2010.

[5] CZAPLICKI, E.; CHONG, S. Asynchronous functional reactive programming for GUIs. *SIGPLAN Not.*, v. 48, n. 6, p. 411–422, 2013.

[6] ODERSKY, M.; OTHERS. The scala language specification. Citeulike article id: 104333. 2004.

[7] KMETT, E. *lens: Lenses, Folds and Traversals*. ⟨https://hackage.haskell.org/package/lens⟩. Accessed: 2018-05-07.

[8] UNGAR, D. et al. Organizing programs without classes. *Lisp Symb. Comput.*, v. 4, n. 3, p. 223–242, 1991.

[9] NIERSTRASZ, O.; DUCASSE, S.; POLLET, D. *Squeak by Example*. 1. ed. [S.l.]: Square Bracket Associates, 2009. v. 1.

[10] GOSLING, J. et al. *The Java Language Specification, Java SE 8 Edition*. 1st. ed. Boston, USA: Addison-Wesley Professional, 2014. v. 1.

[11] HUDAK, P.; FASEL, J. H. A gentle introduction to Haskell. *SIGPLAN Not.*, v. 27, n. 5, p. 1–52, 1992.

[12] 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).

[13] PIERCE, B. C. *Types and Programming Languages*. 1st. ed. Cambridge, USA: The MIT Press, 2002. v. 1.

[14] HASKELLWIKI. *Record Access*. ⟨https://wiki.haskell.org/Record_access⟩. Accessed: 2018-08-06.

[15] DOLAN, S.; MYCROFT, A. Polymorphism, subtyping, and type inference in MLsub. In: CASTAGNA, G. (Ed.). *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. New York, USA: ACM, 2017. (POPL, '17).

[16] WADLER, P. Views: A way for pattern matching to cohabit with data abstraction. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. New York, USA: ACM, 1987. (POPL, '87).

[17] GOTTLOB, G.; PAOLINI, P.; ZICARI, R. Properties and update semantics of consistent views. *ACM Trans. Database Syst.*, v. 13, n. 4, p. 486–524, 1988.

[18] BOHANNON, A.; PIERCE, B. C.; VAUGHAN, J. A. Relational lenses: A language for updatable views. In: *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. New York, USA: ACM, 2006. (PODS, '06).

[19] FOSTER, J. N. et al. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. *SIGPLAN Not.*, v. 40, n. 1, p. 233–246, 2005.

[20] BOHANNON, A. et al. Boomerang: Resourceful lenses for string data. In: NECULA, G. (Ed.). *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, USA: ACM, 2008. (POPL, '08).