An Alternative to Pattern Matching, Inspired by Verse

ROGER BURTONPATEL, Tufts University, USA

Pattern matching appeals to functional programmers for its expressiveness and good cost model, but it fails to express certain computations without verbosity. Verbosity can be reduced by using extended forms of pattern matching, but extensions are not standardized, and the most desirable combination of extensions cannot be found in any one popular programming language. Alternatively to pattern matching, computations can be expressed succinctly by using equations, as has been demonstrated by the Verse programming language. But such computations may have time and space costs that can be hard to predict. As a compromise, I propose a new language, V^-, which uses equations in a limited way that makes time and space costs easy to predict. In comparative examples, V^- expresses computations as succinctly as pattern matching with popular extensions, and just like pattern matching, it can be compiled to a decision tree.

An implementation of V^- and its compiler can be found at https://github.com/rogerburtonpatel/vml.

Author's address: Roger Burtonpatel, roger.burtonpatel@tufts.edu, Tufts University, 419 Boston Ave, Medford, Massachusetts, USA, 02155.

Contents

1 INTRODUCTION

Perhaps the most beloved tool among functional programmers for examining and deconstructing data is pattern matching. Pattern matching is also an established and well-researched topic [Baudinet and MacQueen 1986; Burton and Cameron 1993; Maranget 2008; Palao Gostanza et al. 1996; Ramsey 2022; Wadler 1987]. It is appreciated by programmers and researchers alike for two main reasons: It enables *implicit* data deconstruction, and it has a desirable cost model. Specifically (regarding the latter), pattern matching can be compiled to a *decision tree*, a data structure that enforces linear runtime performance by guaranteeing no part of the data will be examined more than once. [Maranget 2008]

However, pattern matching cannot express certain common computations succinctly, forcing programmers who wish to express these computations to duplicate code, nest *case* expressions, and create multiple points of truth. To mitigate this, designers of popular programming languages have introduced *extensions* to pattern matching (Section 2.1).

Extensions strengthen pattern matching, but they are not standardized, so each popular programming language with pattern matching features its own unique suite of extensions. Extensions are subject to the discretion of the individual language designer, not ubiquitous. Rather than continuing to extend pattern matching *ad hoc*, a worthwhile goal could be to find an alternative that doesn't need extensions. A tempting possibility was introduced last year by the programming language Verse [Augustsson et al. 2023]. In Verse, a programmer can deconstruct data using a different tool the language offers: *equations*. Equations are expressive and flexible, and it appears that they can express everything that pattern matching can, including with popular extensions.

But a full implementation of Verse is complicated, cost-wise. Verse is a functional logic programming language, and expressions can backtrack at runtime and return multiple results, both of which are hard to predict in their costs.

In this thesis, I show that the expressive quality of Verse's equations and the decision-tree property of patterns can be combined in a single language. Since the language is a streamlined adaptation of Verse with a reduced feature set, I call it V^- ("V minus").

To support this claim, I have formalized V^- with a big-step operational semantics (Section 4.2), I have formalized decision trees into a core language D ("D") with a big-step

operational semantics (Section 5.1), I have formalized a translation from V^- to D (Sections 5), and I have implemented both languages in Standard ML.

2 PATTERN MATCHING AND ITS EXTENSIONS

In this section, I expand on the definitions, forms, and tradeoffs of pattern matching. These tradeoffs inform the compromises I make in V^- (Section 4.2).

Pattern matching lets programmers examine and deconstruct data by matching them against patterns. When a pattern p matches a value v, it can produce bindings for subvalues of v. For example, pattern $x :: xs$ matches any application of the value constructor *cons* (::), and it binds the first element of the cons cell to x and the second to xs.

Why use pattern matching? What could programmers use instead? Before pattern matching was invented, a programmer had to deconstruct data using *observers* [Liskov and Guttag 1986]: functions that explicitly examine a piece of data and extract its components. Examples of observers in functional programming languages include Scheme's `null?`, `car`, and `cdr`, and ML's `null`, `hd`, and `tl`. Given the option of pattern matching, however, many functional programmers favor it over observers. I demonstrate with an example and a claim.

Consider a `standard_shape` datatype in Standard ML, which represents shapes by their dimensions[1]:

```
datatype standard_shape = SQUARE    of real
                        | TRIANGLE  of real * real
                        | TRAPEZOID of real * real * real
```

I define an `area` function on *standard_shape*s, with this type and these algebraic laws:

```
area : standard_shape -> real
   area (SQUARE s)              == s * s
   area (TRIANGLE (w, h))       == 0.5 * w * h
   area (TRAPEZOID (b1, b2, h)) == 0.5 * (b1 + b2) * h
```

[1]For sake of simplicity, `standard_shapes` always have an area that can be obtained by standard area formulas: $\frac{1}{2} * base * height$ for triangles; $\frac{1}{2} * (base_1 + base_2) * height$ for trapezoids.

Now compare two implementations of `area`, one with observers and one with pattern matching (Figure 1).

```
fun observers_area sh =
  if isSquare sh
  then sqSide sh * sqSide sh
  else if isTriangle sh
  then 0.5 * triW sh * triH sh
  else 0.5 * traB1 sh * traB2 sh * traH sh
```

(a) area with observers

```
fun pm_area sh =
  case sh
    of SQUARE s              => s * s
     | TRIANGLE (w, h)       => 0.5 * w * h
     | TRAPEZOID (b1, b2, h) => 0.5 * (b1 + b2) * h
```

(b) area with pattern matching

Fig. 1. Implementing area using observers is tedious, and the code doesn't look like the algebraic laws. Using pattern matching makes an equivalent implementation more appealing.

Implementing the observers `isSquare`, `isTriangle`, `sqSide`, `triW`, `traB1`, `traB2`, and `traH` is left as an (excruciating) exercise to the reader.

In general, pattern matching is preferred over observers for five reasons.

A. 1 *Lawlike.* With pattern matching, code more closely resembles algebraic laws.
2 *Single copy.* With pattern matching, it's easier to avoid duplicating code.
3 *Exhaustiveness.* With pattern matching, a programmer can easily tell if they've covered all cases, and a compiler can verify this through *exhaustiveness analysis.*

B. 4 *Call-free.* Pattern matching does not need function calls to deconstruct data.

5 *Signposting*. With pattern matching, important intermediate values are always given names.

For the rest of the paper. I refer to these as Nice Properties. They are broken into two groups: Group A, which contains properties of pattern matching that programmers enjoy in general, and Group B, which contains properties strictly to do with pattern matching's specific strengths over observers.

The most important of the Nice Properties are Lawlike and Single copy: they allow programmers to write code that looks like what they write at the whiteboard, with flexible laws and minimal duplication. Upholding these two properties is the primary responsibility of *extensions* to pattern matching (Section 2.1).

Let's see how each of our Nice Properties holds up in `area`:

A.
1 *Lawlike.* `pm_area` more closely resembles the algebraic laws for `area`.
2 *Single copy.* `observers_area` had to call observers like`squareSide` multiple times, and each observer needs `sh` as an argument. `pm_area` was able to extract the `standard_shapes`' internal values with a single pattern, and the name `sh` is not duplicated anywhere in its body.
3 *Exhaustiveness.* If the user adds another value constructor to `standard_shape`—say, `CIRCLE`, the compiler will warn the user of the possibility of a `Match` exception in `pm_area`, and even tell them that they must add a pattern for `CIRCLE` to rule out this possibility. `observers_area` will not cause the compiler to complain, and if it's passed a `CIRCLE` at runtime, the program will likely crash!

B.
4 *Call-free.* Where did `isSquare`, `sqSide`, and all the other observers come from? To even *implement* `observers_area`, a programmer has to define a whole new set of observers for `standard_shapes`![2] Most programmers find this tiresome indeed. `pm_area` did not have to do any of this.
5 *Signposting.* To extract the internal values, `pm_area` had to name them, and their names serve as documentation.

[2]Sometimes the compiler throws programmers a bone: with some constructed data, i.e., Scheme's records, the compiler provides observers automatically. In others, i.e., algebraic datatypes in ML, it does not.

Having had the chance to compare pattern matching and observers, if you moderately prefer pattern matching, that's good: most functional programmers—in fact, most *programmers*—likely do as well.

`pm_area` provides an opportunity to introduce a few terms that are common in pattern matching. `pm_area` is a classic example of where pattern matching most commonly occurs: within a *case* expression. A *case* expression tests a *scrutinee* (`sh`) against a list of *branches*. Each branch contains a pattern on the left-hand side (`SQUARE s`, etc.) and an expression on the right-hand side (`s * s`, etc.). When a pattern matches the result of evaluating the scrutinee, the program evaluates the right-hand side of the respective branch.[3]

2.1 Popular extensions to pattern matching

Extensions to pattern matching simplify cases that are otherwise troublesome. Specifically, extensions help restore Nice Properties Lawlike and Single copy in cases where pattern matching fails to satisfy them.

In this section, I illustrate several such cases, and I demonstrate how extensions help programmers write code that adheres to the Nice Properties. The three extensions I describe are those commonly found in the literature and implemented in compilers: side conditions, pattern guards, and or-patterns.

To denote pattern matching *without* extensions, I coin the term *bare pattern matching*. In bare pattern matching, a pattern has one of two syntactic forms: a name or an application (of a value constructor to zero or more patterns).

2.1.1 Side conditions.

First, I illustrate why programmers want *side conditions*, an extension to pattern matching common in most popular functional programming languages, including OCaml, Erlang, Scala, and Haskell[4].

[3] OCaml, which you'll see in future sections, calls *case* `match`. Some literature [Erwig and Jones 2001] calls this a *head expression*. I follow the example of Ramsey [2022] and Maranget [2008] in calling the things *case* and *scrutinee*. Any of these terms does the job.

[4] I use the term *side conditions* to refer to a pattern followed by a Boolean expression. Some languages call this a *guard*, which I use to describe a different, more powerful extension to pattern matching in Section 4.4.3. Haskell has *only* guards, but a Boolean guard is a side condition.

I define a (rather silly) function `exclaimTall` in OCaml on `standard_shapes`. I have to translate our `standard_shape` datatype to OCaml, and while I'm at it, I write the type and algebraic laws for `exclaimTall`:

```
type standard_shape = Square of float
                    | Triangle of float * float
                    | Trapezoid of float * float * float

exclaimTall : standard_shape -> string

exclaimTall (Square s)              == "Wow! That's a sizeable square!",
                                       where s > 100.0
exclaimTall (Triangle (w, h))       == "Goodness! Towering triangle!",
                                       where h > 100.0
exclaimTall (Trapezoid (b1, b2, h)) == "Zounds! Tremendous trapezoid!",
                                       where h > 100.0
exclaimTall sh                      == "Your shape is small.",
                                       otherwise
```

Armed with pattern matching, I implement `exclaimTall` in OCaml (Figure 2).

Here, I'm using the special variable `_`—that's the underscore character, a wildcard pattern—to indicate that I don't care about the bindings of a pattern.

I am not thrilled with this code. It gets the job done, but it fails to adhere to the Nice Properties of Lawlike and Single copy: the code does not look like the algebraic laws, and it duplicates right-hand side, `"Your shape is small"`, three times. I find the code unpleasant to read, too: the actual "good" return values of the function, the exclamatory strings, are gummed up in the middle of the `if-then-else` expressions.

Fortunately, this code can be simplified by using the `standard_shape` patterns with a *side condition*, i.e., a syntactic form for "match a pattern *and* a Boolean condition." The `when` keyword in OCaml provides such a form, as seen in Figure 3.

A side condition streamlines pattern-and-Boolean cases and minimizes overhead, restoring Lawlike and Single copy. And a side condition can exploit bindings that emerge from the preceding pattern match. For instance, the `when` clauses in Figure 3 exploit

```
let exclaimTall sh =
match sh with
  | Square s -> if s > 100.0
                then "Wow! That's a sizeable square!"
                else "Your shape is small."
  | Triangle (_, h) ->
                if h > 100.0
                then "Goodness! Towering triangle!"
                else "Your shape is small."
  | Trapezoid (_, _, h) ->
                if h > 100.0
                then "Zounds! Tremendous trapezoid!"
                else "Your shape is small."
```

Fig. 2. An invented function `exclaimTall` in OCaml combines pattern matching with an `if` expression, and is not very pretty.

```
let exclaimTall sh =
  match sh with
    | Square s when s > 100.0 ->
            "Wow! That's a sizeable square!"
    | Triangle (_, h) when h > 100.0 ->
            "Goodness! Towering triangle!"
    | Trapezoid (_, _, h) when h > 100.0 ->
            "Zounds! Tremendous trapezoid!"
    | _ ->  "Your shape is small."
```

Fig. 3. With a side condition, `exclaimTall` in OCaml becomes simpler and more adherent to the Nice Properties.

names `s` and `h`, which are bound in the match of `sh` to `Square s`, `Triangle (_, h)`, and `Trapezoid (_, _, h)`, respectively.

Importantly, side conditions come at a cost: their inclusion means that a compiler enforcing the Exhaustiveness Nice Property becomes an NP-hard problem, because it must now perform exhaustiveness analysis not only on patterns, but on arbitrary expressions. Modern compilers give a weaker form of exhaustiveness that only deals with patterns,

and side conditions are worth the tradeoff for restoring the two most important of the Nice Properties: Lawlike and Single copy.

A side condition adds an extra "check"—in this case, a Boolean expression—to a pattern. But side conditions have a limitation: the check can make a decision based off of an expression evaluating to `true` or `false`, but not an expression evaluating to, say, `nil` or `::` (cons). In the next section, I use an example to showcase when this limitation matters, and how another extension addresses it.

2.1.2 Pattern guards. To highlight a common use of pattern guards to address such a limitation, I modify an example from Erwig and Jones [2001], the proposal for pattern guards in GHC. Suppose I have an OCaml abstract data type of finite maps, with a lookup operation:

```
lookup : finitemap -> int -> int option
```

Let's say I want to perform three successive lookups, and call a "fail" function if *any* of them returns `None`. Specifically, I want a function with this type and algebraic laws:

```
tripleLookup : finitemap -> int -> int

  tripleLookup rho x == z, where
                              lookup rho x == Some w
                              lookup rho w == Some y
                              lookup rho y == Some z
  tripleLookup rho x == handleFailure x, otherwise

  handleFailure : int -> int

  handleFailure's implementation is unimportant.
    handleFailure (x : int) = ... some error-handling ... -> x
```

To express this computation succinctly, the program needs to make decisions based on how successive computations match with patterns, but neither bare pattern matching nor side conditions give that flexibility.

Side conditions don't appear to help here, so I try with bare pattern matching. Figure 4 shows how I might implement `tripleLookup` as such.

```
let tripleLookup (rho : finitemap) (x : int) =
    match lookup rho x with Some w ->
      (match lookup rho w with Some y ->
        (match lookup rho y with Some z -> z
        | _ -> handleFailure x)
      | _ -> handleFailure x)
    | _ -> handleFailure x
```

Fig. 4. `tripleLookup` in OCaml with bare pattern matching breaks the Nice Property of Single copy: avoiding duplicated code.

Once again, the code works, but it's lost the Lawlike and Single copy Nice Properties by duplicating three calls to `handleFailure` and stuffing the screen full of syntax that distracts from the algebraic laws. Unfortunately, it's not obvious how a side condition could help us here, because we need pattern matching to extract and name internal values `w`, `y`, and `z` from constructed data.

To restore the Nice Properties, I introduce a more powerful extension to pattern matching: *pattern guards*, a form of "smart pattern" in which intermediate patterns bind to expressions within a single branch of a `case`. Pattern guards can make `tripleLookup` appear *much* simpler, as shown in Figure 5—which, since pattern guards aren't found in OCaml, is written in Haskell.

Guards appear as a comma-separated list between the `|` and the `=`. Each guard has a pattern, followed by `<-`, then an expression. The guard is evaluated by evaluating the expression and testing if the pattern matches with the result. If it does, the next guard is evaluated in an environment that includes the bindings introduced by evaluating guards before it. If the match fails, the program skips evaluating the rest of the branch and falls through to the next one. As a bonus, a guard can simply be a Boolean expression which

```
tripleLookup rho x
   | Just w <- lookup rho x
   , Just y <- lookup rho w
   , Just z <- lookup rho y
   = z
tripleLookup _ x = handleFailure x
```

Fig. 5. Pattern guards swoop in to restore the Nice Properties, and all is right again.

the program tests the same way it would a side condition, so guards subsume side conditions!

If you need further convincing on why programmers want for guards, look no further than Erwig & Peyton Jones's *Pattern Guards and Transformational Patterns* [Erwig and Jones 2001], the proposal for pattern guards in GHC: the authors show several other examples where guards drastically simplify otherwise-maddening code.

Pattern guards enable expressions within guards to utilize names bound in preceding guards, enabling imperative pattern-matched steps with expressive capabilities akin to Haskell's `do` notation. It should come as no surprise that pattern guards are built in to GHC.

2.1.3 Or-patterns. I conclude our tour of extensions to pattern matching with or-patterns, which are built in to OCaml. Let's consider a final example. I have a type `token` which represents an item or location in a video game and how much fun it is, and I need to quickly know what game it's from and how much fun I'd have playing it. To do so, I'm going to write a function `game_of_token` in OCaml. HThe `token` type and the type and algebraic laws for `game_of_token` are in Figure 6.

I can write code for `game_of_token` in OCaml using bare patterns (Figure 7), but I'm dissatisfied with how it fails the Lawlike and Single copy Nice Properties: it is visually different from the algebraic laws, and it has many duplicated right-hand sides.

I could try to use a couple of helper functions to reduce clutter, so I do so and come up with the code in Figure 8. It looks ok, but I'm still hurting for Nice Property Lawlike, and now I've lost the Call-free Property, as well.

```
type funlevel = int

type token = BattlePass  of funlevel | ChugJug    of funlevel | TomatoTown of funlevel
           | HuntersMark of funlevel | SawCleaver of funlevel
           | MoghLordOfBlood of funlevel | PreatorRykard of funlevel
           ... other tokens ...


game_of_token : token -> string * funlevel

game_of_token t == ("Fortnite", f),  where t is any of
                                       BattlePass f,
                                       ChugJug f, or
                                       TomatoTown f
game_of_token t == ("Bloodborne", 2 * f),
                                       where t is any of
                                         HuntersMark f or
                                         SawCleaver f
game_of_token t == ("Elden Ring", 3 * f),
                                       where t is any of
                                         MoghLordOfBlood f or
                                         PreatorRykard f
game_of_token _ == ("Irrelevant", 0), otherwise
```

Fig. 6. Type and laws for game_of_token, which make helpful use of ”where.”

```
let game_of_token token = match token with
    | BattlePass f      -> ("Fortnite", f)
    | ChugJug f         -> ("Fortnite", f)
    | TomatoTown f      -> ("Fortnite", f)
    | HuntersMark f     -> ("Bloodborne", 2 * f)
    | SawCleaver f      -> ("Bloodborne", 2 * f)
    | MoghLordOfBlood f -> ("Elden Ring", 3 * f)
    | PreatorRykard f   -> ("Elden Ring", 3 * f)
    | _                 -> ("Irrelevant", 0)
```

Fig. 7. game_of_token, with redundant right-hand sides, should raise a red flag.

Once again, an extension comes to the rescue. *Or-patterns* condense multiple patterns that share a right-hand side, and when any one of the patterns matches with the scrutinee,

```
let fortnite   f = ... complicated   ... in
let bloodborne f = ... complicated'  ... in
let eldenring  f = ... complicated'' ... in
match token with
| BattlePass f -> fortnite f
... and so on ...
```

Fig. 8. game_of_token with helpers is somewhat better, but I'm not satisfied with it.

```
let game_of_token token = match token with
    | BattlePass f | ChugJug f | TomatoTown f  -> ("Fortnite", f)
    | HuntersMark f | SawCleaver f            -> ("Bloodborne", 2 * f)
    | MoghLordOfBlood f | PreatorRykard f     -> ("Elden Ring", 3 * f)
    | _                                       -> ("Irrelevant", 0)
```

Fig. 9. Or-patterns condense game_of_token significantly, and it is easier to read line-by-line.

the right-hand side is evaluated with the bindings created by that pattern. I exploit or-patterns in Figure 9 to restore the Nice Properties and eliminate much of the uninteresting code that appeared in 7 and 8.

In addition to the inherent appeal of brevity, or-patterns serve to concentrate complexity at a single juncture and create single points of truth, restoring the Lawlike and Single copy properties.

2.1.4 Wrapping up pattern matching and extensions. I have presented three popular extensions that make pattern matching more expressive and how to use them effectively. Earlier, though, you might have noticed a problem. Say I face a decision-making problem that obliges me to use *all* of these extensions. When picking a language to do so, I am stuck! No major functional language has all three of these extensions. Remember when I had to switch from OCaml to Haskell to use guards, and back to OCaml for or-patterns? The two extensions are mutually exclusive in Haskell and OCaml, and also Scala, Erlang/Elixir, Rust, F#, and Agda [Barklund and Virding 1999; Klabnik and Nichols 2023; Kokke et al. 2020; Leroy et al. 2023; Marlow et al. 2010; Syme et al. 2010; The Elixir Team; École Polytechnique Fédérale].

I find the extension story somewhat unsatisfying. At the very least, I want to be able to use pattern matching, with the extensions I want, in a single language. Or, I want an alternative that gives me the expressive power of pattern matching with these extensions.

3 EQUATIONS

An intriguing alternative to pattern matching exists in *equations*, from the Verse Calculus ($\mathcal{VC}$), a core calculus for the functional logic programming language *Verse* [Antoy and Hanus 2010; Augustsson et al. 2023; Hanus 2013]. (For the remainder of this thesis, I use "$\mathcal{VC}$ " and "Verse" interchangeably.)

$\mathcal{VC}$'s *equations* test for structural equality and create bindings. Like pattern matching, equations scrutinize and deconstruct data at runtime by testing for structural equality and unifying names with values. Unlike pattern matching, $\mathcal{VC}$'s equations can unify names on both left- *and* right-hand sides.

Every equation in $\mathcal{VC}$ takes the form $x = e$, where x is a name and e is an expression[5]. During runtime, $\mathcal{VC}$ relies on a process called *unification* [Robinson 1965] to attempt to bind x and any unbound names in e to values. Unification is the process of finding a substitution that makes two different logical atomic expressions identical. Much like pattern matching, unification can fail if the runtime attempts to bind incompatible values or structures (i.e., finds no substitution).

An equation offers a form of binding that looks like a single pattern match. What about a list of many patterns and right-hand sides, as in a case expression? For this, $\mathcal{VC}$ has *choice* (`|`). The full semantics of choice are too complex to cover here, but choice, when combined with the **one** operator, has a very similar semantics to case; that is, "proceed and create bindings if any one of these computations succeed."

Let's look at what equations, **one**, and choice look like in $\mathcal{VC}$ (Figure 10). I've written the `area` function in $\mathcal{VC}$ extended with a `float` type and a multiplication operator `*`.

In the figure, the name `vc_area` is bound to a lambda (λ) that takes a single argument, `sh`. The body of the function is a **one** expression over three choices (separated by `|`). If any of the choices succeeds, **one** ensures evaluation of the other choices halts

[5]To make programmers happy, full Verse allows an equation to take the form $e_1 = e_2$, which desugars to $\exists x.x = e_1;\ x = e_2$, with x fresh.

```
∃ vc_area. vc_area = λ sh.
  one {  ∃ s. sh  = ⟨SQUARE, s⟩; s * s
       | ∃ w h. sh = ⟨TRIANGLE, w, h⟩;
              0.5 * w * h
       | ∃ b1 b2 h. sh = ⟨TRAPEZOID, b1, b2, h⟩;
              0.5 * (b1 + b2) * h}


   Laws for area:

   area (SQUARE s)              == s * s
   area (TRIANGLE (w, h))       == 0.5 * w * h
   area (TRAPEZOID (b1, b2, h)) == 0.5 * (b1 + b2) * h
```

Fig. 10. vc_area in $\mathcal{VC}$ uses existentials and equations rather than pattern matching. Below are the algebraic laws for the original area function.

and the succeeding expression's result is returned. In each branch, the existential ∃ introduces names `s, w, h, b1, b2`, and is followed by an *equation* that unifies them with `sh`, along with the familiar value constructors `SQUARE`, `TRIANGLE`, and `TRAPEZOID`. After the equation is a semicolon followed by an expression, which is evaluated if the equation succeeds. As in `pm_area`, the right-hand sides of `vc_area` are *guarded* by a "check;" now, the check is successful unification in an equation rather than a successful match on a pattern. Similarly, **one** with a list of choices represents matching on any *one* pattern to evaluate a single right-hand side.

Why use equations? I begin with a digestible claim: $\mathcal{VC}$'s equations are preferable to observer functions. This claim mirrors my argument for pattern matching, and to support it I appeal to the Nice Properties:

(1) *Lawlike.* `vc_area` makes only one addition to the algebraic laws: the explicit ∃. This makes `vc_area` look more like mathematical notation than pure algebraic laws, but that might not be a bad thing: while it less resembles the algebraic laws a programmer would write, it more resembles the equations that a mathematician would.

(2) *Single copy.* `vc_area` does not duplicate any code.

(3) *Exhaustiveness.* By writing the equations that unify `sh` with the value-constructor forms first, it is easy in this example to see that the code is exhaustive. Creating a static analysis tool for $\mathcal{VC}$ that ensures exhaustiveness on all expressions may or may not be a significant challenge; full Verse has a tool that can verify if a terminating expression on the right-hand side of a function will *always succeed* or not [Peyton-Jones 2024].

(4) *Call-free.* `vc_area` deconstructs user-defined types as easily as `pm_area` does with pattern matching.

(5) *Signposting.* `vc_area` has all important internal values named explicitly.

You understand why I claim programmers prefer equations to observer functions. Now I make a stronger claim: equations are *at least as good as* pattern matching with popular extensions. How can I claim this? By appealing again to the Nice Properties! In Section 2, I demonstrated how pattern matching had to resort to extensions to regain the Properties when challenging examples stole them away. In Figure 11, I've implemented those examples in $\mathcal{VC}$ (this time extended with strings, floats, and $*$) using choice and equations. Take a look for yourself!

The code in Figure 11 has all the Nice Properties. This is promising for $\mathcal{VC}$. If it rivals pattern matching with popular extensions in desirable properties, and $\mathcal{VC}$ does everything using only equations and choice, it seems like the language is an intriguing option for writing code!

3.1 $\mathcal{VC}$ has a challenging cost model

So what's the catch? In $\mathcal{VC}$, names (logical variables) are *values*, and they can just as easily be the result of evaluating an expression as an integer or tuple. To bind these names, $\mathcal{VC}$, like other functional logic languages, relies on *unification* of its logical variables and *search* at runtime to meet a set of program constraints [Antoy and Hanus 2010; Hanus 2013]. Combining unifying logical variables with search requires backtracking, which can lead to exponential runtime cost [Clark 1982; Hanus 2013; Wadler 1985].

Pattern matching, by contrast, can be compiled to a *decision tree*, a data structure that enforces linear runtime performance by guaranteeing no part of the scrutinee will be examined more than once [Maranget 2008]. A decision tree does not backtrack: once a

```
∃ exclaimTall. exclaimTall = λ sh.
  one {
      ∃ s. sh = ⟨Square s⟩;
      s > 100.0; "Wow! That's a sizeable square!
          "
    | ∃ w h. sh = ⟨Triangle, w, h⟩;
      h > 100.0; "Goodness! Towering triangle!"
    | ∃ b1 b2 h. sh = ⟨Trapezoid, b1, b2, h⟩;
      h > 100.0; "Zounds! Tremendous trapezoid!"
    | "Your shape is small." }
```

(a) `exclaimTall` in $\mathcal{VC}$

```
∃ tripleLookup. tripleLookup = λ rho x.
    one { ∃ w. lookup rho x = ⟨Just w⟩;
          ∃ y. lookup rho w = ⟨Just y⟩;
          ∃ z. lookup rho y = ⟨Just z⟩;
          z
        | handleFailure x }
```

(b) `tripleLookup` in $\mathcal{VC}$

```
∃ game_of_token. game_of_token = λ token.
  ∃ f. one {
         token = one { ⟨BattlePass, f⟩ | ⟨ChugJug, f⟩ | ⟨TomatoTown, f⟩};
           ⟨"Fortnite", f⟩
       | token = one { ⟨HuntersMark, f⟩ | ⟨SawCleaver, f⟩};
           ⟨"Bloodborne", 2 * f⟩
       | token = one { ⟨MoghLordOfBlood, f⟩ | ⟨PreatorRykard, f⟩};
           ⟨"Elden Ring", 3 * f⟩
       |   ⟨"Irrelevant", 0⟩ }
```

(c) `game_of_token` in $\mathcal{VC}$

Fig. 11. Code for the 2 functions with equations looks similar, and it doesn't need extensions.

program makes a decision based on the form of a value, it does not re-test it later with new information.

4 A COMPROMISE

To bridge the gap between pattern matching, equations, and decision trees, I have created and implemented a semantics for a new core language: V^- ("V-minus").

V^- has equations and choice, like $\mathcal{VC}$, but it does not have multiple results or backtracking. To eliminate multiple results, expressions in V^- evaluate to at most one result, and choice only *guards* computation; it is not a valid form of expression. To eliminate backtracking, the compiler rejects a V^- program that would need to backtrack at runtime. To provide an efficient and backtracking-free cost model to which V^- can be compiled, I introduce a core language of decision trees, D, in Section 5.1.

4.1 Two languages of a kind

In this section, I present V^-. Its semantics appears in Section 4.8, and for reference in Appendix B. In my design, I took inspiration from Verse: V^- has a conventional sub-language that is the lambda calculus extended with named value constructors K applied to zero or more values. I chose named value constructors over $\mathcal{VC}$'s tuples because they look more like patterns. D, the language of decision trees and target of translation from V^-, has the same lambda-calculus-plus-value-constructors core, with decision trees substituted for V^-'s **if-fi**. Because they share a core, and to facilitate comparisons and proofs, I present V^- and D as two subsets of a single unifying language U, whose abstract syntax appears in Table 1. Forms in black are present in both languages, forms in red are specific to V^-, and forms in blue are specific to D.

As in $\mathcal{VC}$, every lambda-calculus term is valid in V^- and D has the same semantics. Also like the lambda calculus and $\mathcal{VC}$, V^- and D are *strict*, meaning every expression is evaluated when it is bound to a variable. V^- and D are also untyped.

The only form of constructed data in V^- and D is value-constructor application, represented by the metavariable K. In full languages, other forms of data like numbers and strings have a similar status to value constructors, but their presence would complicate the development of semantics and code.

Using just value constructors, though, a programmer can simulate more primitive data like strings. For example, `Wow! That's A Sizeable Square` is a valid expression in V^- and D, because it is an application of constructor `Wow!` to the arguments `That's`, `A`, `Tall`, and `Square`, all of which are value constructors themselves. Each name in this "sentence" is considered a value constructor because it begins with a capital letter. To simulate integers, a programmer can use Peano numbers, they can use value constructors to implement binary numbers, or they can cheat with singletons: `One`, `Two`, etc. Because the languages all also have lambda, Church Numerals [Church 1985] are another option.

In the subsection below, I discuss V^- in more detail. I discuss D in more detail in Section 5.1. In Section 8, I discuss how V^- relates to $\mathcal{VC}$.

Syntactic Forms	**Cases**
P : Programs	$\{d\}$
d : Definitions	**val** $x = e$
v: Values	$K\{v\}$
	$\lambda x.\ e$
e : Expressions	v
	x, y, z
	$K\{e\}$
	$\lambda x.\ e$
	$e_1\ e_2$
	if $\overline{G}$ **fi**
	t
G : Guarded Expressions	$[\exists \overline{x}.]\ \overline{g} \rightarrow e$
g : Guards	$x = e$
	e
	$\overline{g}$ **|** $\overline{g}'$
t : Decision Trees	$test\ x\ \{K_i/\overline{y}_i\ \Rightarrow\ t\}\ [else\ t]$
	e
	$\exists x.\ t$
	$let\ x = e\ in\ t[;\ unless\ \textbf{fail} \Rightarrow\ t]$
	$if\ x == e\ then\ t\ else\ t$
	fail

Table 1. Abstract Syntax of V^- and D. Forms in black are present in both languages, forms in red are specific to V^-, and forms in blue are specific to D.

4.2 Introducing V^-

To fuel the pursuit of smarter decision-making, I now draw inspiration from $\mathcal{VC}$. Equations in $\mathcal{VC}$ look attractive, but the cost model of $\mathcal{VC}$ is a challenge.

The elements of $\mathcal{VC}$ that lead to unpredictable or costly run times are backtracking and multiple results. So, I begin with a subset of $\mathcal{VC}$, which I call V^- ("V minus"), with these elements removed. Removing them strips much of the identity of $\mathcal{VC}$, but it leaves its *equations* to build on top of in an otherwise-typical programming context of single results and no backtracking at runtime.

Having stripped out the functional logic programming elements of $\mathcal{VC}$ (backtracking and multiple results), only the decision-making bits are left over. To wrap these, I add a classic decision-making construct: guarded commands [Dijkstra 1976] The result is V^-.

Programs	P	::=	$\{d\}$	definition
Definitions	d	::=	$val\ x = e$	bind name to expression
Expressions	e	::=	v	literal values
		|	x, y, z	names
		|	**if** $[\ G\ \{[\!]\ G\}\ ]$ **fi**	if-fi
		|	$K\ \{e\}$	value constructor application
		|	$e_1\ e_2$	function application
		|	$\lambda x.\ e$	lambda declaration
Guarded Expressions	G	::=	$[\exists\ \{x\}.]\ \{g\}\ \rightarrow e$	names, guards, and body
Guards	g	::=	e	intermediate expression
		|	$e_1 = e_2$	equation
		|	$g\ \{;g\}\ \ \mathbf{|}\ g\ \{;g\}$	choice
Values	v	::=	$K\{v\}$	value constructor application
		|	$\lambda x.\ e$	lambda value

* Desugars to $x = e_1;\ x = e_2$, x fresh.

Fig. 12. V^-: Concrete syntax

V^- takes several key concepts from $\mathcal{VC}$—namely, equations and choice—with several key modifications. Like $\mathcal{VC}$, V^- has equations and choice, but unlike in $\mathcal{VC}$, choice only guards computations, there are no multiple results, and all decision-making appears in the **if-fi** construct, inspired by Dijkstra [Dijkstra 1976].

4.3 Programming in V^-

Even with multiple modifications, V^- still allows for many of the same pleasing computations as full Verse. A programmer can…

(1) Introduce a set of equations, to be solved in a nondeterministic order
(2) Guard expressions with those equations

(3) Flexibly express "proceed when any of these operations succeeds" with the new semantics of choice (Section 4.8).

Figure 13 provides an example of how a programmer can utilize V^- to solve the previous problems (Section 2.1):

```
val exclaimTall =~\sh.
  if E s. sh = Square s; (> s) 100 ->
           Wow! That's A Sizeable Square!
    [] E w h. sh = Triangle w h; (> h) 100 ->
           Goodness! Towering Triangle!
    [] E b1 b2 h. sh = Trapezoid b1 b2 h;
     (> h) 100 -> Zounds! Tremendous Trapezoid!
    [] ->  Your Shape Is Small
  fi
```

(a) `exclaimTall` in V^-

```
val tripleLookup =~\rho.~\x.
  if E w y z v1 v2 v3.
    v1 = (lookup rho) x; v1 = Some w;
    v2 = (lookup rho) w; v2 = Some y;
    v3 = (lookup rho) y; v3 = Some z
      -> z
    [] -> handleFailure x
  fi
```

(b) `tripleLookup` in V^-

```
val game_of_token =~\t.
  if E f. (t = BattlePass f  | (t = ChugJug f | t = TomatoTown f)) ->
       P (Fortnite f)
  [] E f. (t = HuntersMark f | t = SawCleaver f)                   ->
       P (Bloodborne ((* 2) f))
  [] E f. (t = MoghLordOfBlood f | t = PreatorRykard f)            ->
       P (EldenRing ((* 3) f))
  [] -> P (Irrelevant 0)
  fi
```

(c) `game_of_token` in V^-

Fig. 13. The functions in V^- have a desirably concise implementation, as before.

V^- looks satisfyingly similar to pattern matching and to $\mathcal{VC}$. The V^- examples in Figure 13 have the same number of cases as the pattern-matching examples, and share the existential and equations with the $\mathcal{VC}$ examples in Figure 11.

V^- *Metavariables*	
e	An expression
$v,\ v'$	A value
fail	An expression failure
r	v | **fail** : expressions produce *results*: values or failure.
ρ	An environment: $name \rightarrow \mathcal{V}_\perp$
$\rho\{x \mapsto y\}$	An environment extended with name x mapping to y
g	A guard
$\overline{g}$	Zero or more guards, separated by *;*
G	A guarded expression
$\overline{G}$	Zero or more guarded expressions, separated by $\square$
eq	An equation
$\dagger$	when solving guards is rejected
s	$\hat{\rho}$ | $\dagger$: guards produce *solutions*: a refined environment $\hat{\rho}$ or rejection
$\mathcal{T}$	A context of all temporarily stuck guards (a sequence)

Table 2. V^- metavariables and their meanings

4.4 Formal Semantics of V^-

In this section, I present a big-step operational semantics for V^-. The semantics describes how an expression in V^- is evaluated and how an equation (and more generally, a guard) works in the language. Instead of a rewrite semantics that makes substitutions within guards, V^- has a big-step semantics that directly describes how they are handled by the runtime core. Figure 12 contains the syntax of V^-, Figure 2 provides the metavariables used in the judgement forms and rules of the semantics, and Section 4.8 contains the forms and rules. Since solving guards is the heart of V^-, I also describe it in plain English.

4.4.1 Expressions. An expression in $\mathcal{VC}$ evaluates to produce possibly-empty sequence of values, where an empty sequence of values is identical to the syntactic form **fail**. In V^-, an expression never returns multiple values, but it can **fail**. Specifically, in V^-, an expression evaluates to produce a single *result*. A result is either a single value v or **fail**.

$$r ::= v \mid \textbf{fail}$$

$$\langle \rho, e \rangle \Downarrow r \quad \text{(Eval)}$$

4.4.2 Names and refinement of environments. In V^-, like in $\mathcal{VC}$, names can be introduced with the existential $\exists$ before they are given a binding. Bindings are given by equations in guards (4.4.3). When a name is introduced with $\exists$, it is bound in the environment ρ to $\bot$ (pronounced "bottom"). Success of a guard only *refines* the environment; that is, when $\rho \vdash g \rightarrowtail \rho'$, we expect $\rho \subseteq \rho'$. The definition of $\subseteq$ on environments is given below.

$$\rho \subseteq \rho' \text{ when } \operatorname{dom}\rho \subseteq \operatorname{dom}\rho' \text{ and } \forall x \in \operatorname{dom}\rho : \rho(x) \subseteq \rho'(x)$$

4.4.3 Guards. For example, the V^- expression `((\x. x) K)` succeeds and returns the value `K`. The V^- expression `if fi`, the empty `if-fi`, always **fail**s.

Like $\mathcal{VC}$, V^- has a nondeterministic semantics. Guards are solved in V^- similarly to how equations are solved in Verse: the program nondeterministically picks one out of a context ($\mathcal{T}$), attempts to solve it, and moves on.

In my semantics, this process occurs over a *list* of guards $\overline{g}$ in a guarded expression G: the program picks a guard from G, attempts to solve it to refine the environment or fail, and repeats. V^- can only pick a guard out of G that it *knows* it can solve. *Knowing* a guard can be solved can be determined in V^- before code is executed. If V^- can't pick a guard and there are guards left over, the semantics gets stuck before code is executed.

$$\rho \vdash \overline{g} \rightarrowtail s \quad \text{(Solve-Guards)}$$

The environment ρ maps from a name to a value v or $\bot$. $\bot$ means a name has been introduced with the existential, $\exists$, but is not yet bound to a value. Given any such ρ, a guard g eithers refine ρ (ρ') or is **rejected**. We use the metavariable $\dagger$ to represent rejection, and if any guard in a list is rejected, the entire list of guards is rejected.

$$\rho \vdash g \rightarrowtail \rho' \quad \text{(Guard-Refine)}$$

$$\rho \vdash g \rightarrowtail \dagger \quad \text{(Guard-reject)}$$

For example, in the V^- expression `if E x. x = K; x = K2 -> x fi`, the existential (in concrete syntax, `E`) introduces `x` to ρ bound to $\bot$, producing the environment $\{x \mapsto \bot\}$. The guard `x = K` successfully unifies `x` with `K`, producing the environment $\{x \mapsto \texttt{K}\}$. The guard `x = K2` attempts to unify `K` with `K2` and is rejected with †.

4.5 Notable rules in the V^- semantics

Worth discussing in the V^- semantics are those in which a name in ρ is bound to a value or to $\bot$. Notable among these include Guard-Name-Exp-Bot, Guard-Name-Exp-Eq, Guard-Name-Exp-Fail, Guard-Names-Bot-Succ, and Guard-Names-Bot-Succ-Rev. This section discusses each of these rules in short detail. The full set of rules for V^- is in Section 4.8.

In this section, I use the terms *known* and *unknown* to denote a name's status in ρ. If a name x is bound to a value v in ρ, then x is *known*. If x is bound to $\bot$ in ρ, then x is *unknown*. I use this terminology again when compiling V^- to D, since it describes the same status in both evaluation in compilation.

$$\text{(Guard-NameExp-Bot)}\ \frac{\begin{array}{c} x \in \operatorname{dom}\rho \\ \langle \rho, e\rangle \Downarrow v \\ \rho(x) = \bot \end{array}}{\rho \vdash x = e \rightarrowtail \rho\{x \mapsto v\}}$$

When $\rho(x) = \bot$ (x is *unknown*) and $\langle \rho, e\rangle \Downarrow r$, the program refines ρ by making a binding of x to v. Here, $=$ is treated like a let-binding in ML-like languages.

$$\frac{\begin{array}{c} x \in \operatorname{dom}\rho \\ \langle \rho, e\rangle \Downarrow v \\ \rho(x) = v' \\ v \neq v' \end{array}}{\rho \vdash x = e \rightarrowtail \dagger}\ \text{(Guard-NameExp-Fail)}$$

When $\rho(x) = v$ (x is *known*) and $\langle \rho, e\rangle \Downarrow r$, the program proceeds without refining the environment, since no new information is gained. Here, $=$ is treated like a == in C-like languages.

$$\frac{\begin{array}{c} x \in \operatorname{dom} \rho \\ \langle \rho, e \rangle \Downarrow v \\ \rho(x) = v' \\ v \neq v' \end{array}}{\rho \vdash x = e \rightarrowtail \dagger} \text{ (Guard-NameExp-Fail)}$$

When $\rho(x) = v'$ (x is *known*) and $\langle \rho, e \rangle \Downarrow r$ and $v \neq v'$, unification fails, so the guarded expression fails. Here, $=$ is treated like a == in C-like languages.

$$\frac{\begin{array}{c} x,\ y \in \operatorname{dom} \rho \\ \rho(x) = v,\ \rho(y) = \bot \end{array}}{\rho \vdash x = y \rightarrowtail \rho\{y \mapsto v\}} \text{ (Guard-Names-Bot-Succ)}$$

When $\rho(x) = v$ (x is *known*) and $rho(y) = \bot$ (y is *unknown*), the program can always bind y to v. Here, $=$ is treated like a let-binding in ML-like languages.

$$\text{(Guard-Names-Bot-Succ-Rev)} \ \frac{\begin{array}{c} x,\ y \in \operatorname{dom} \rho \\ \rho(x) = \bot,\ \rho(y) = v \end{array}}{\rho \vdash x = y \rightarrowtail \rho\{x \mapsto v\}}$$

This rule is simply an application of Guard-Names-Bot-Succ in reverse.

4.6 Rules (Big-step Operational Semantics) for U, shared by V^- and D

4.6.1 Judgement forms for U. In both V^- and D, an expression evaluates to produce a single *result*. A result is either a single value v or **fail**.

$$\langle \rho, e \rangle \Downarrow r \quad \textsc{(Eval)}$$

4.7 Evaluating general expressions in U

Here, I show all the rules which are shared by V^- and D under U.

$$\textsc{(Eval-Vcon)}\ \frac{\langle \rho, e_i \rangle \Downarrow v_i \qquad 1 \leq i \leq n}{\langle \rho, K(e_1, \ldots e_n) \rangle \Downarrow K(v_1, \ldots v_i)} \qquad \frac{\exists e_i.\ 1 \leq i \leq n : \langle \rho, e_i \rangle \Downarrow \textbf{fail}}{\langle \rho, K(e_1, \ldots e_n) \rangle \Downarrow \textbf{fail}}\ \textsc{(Eval-Vcon-Fail)}$$

$$\textsc{(Eval-Name)}\ \frac{\begin{array}{c} x \in \operatorname{dom} \rho \\ \rho(x) = v \end{array}}{\langle \rho, x \rangle \Downarrow v} \qquad \frac{}{\langle \rho, \lambda x.e \rangle \Downarrow (\!| \lambda x.e, \rho |\!)}\ \textsc{(Eval-LambdaDecl)}$$

$$\textsc{(Eval-Funapp)}\ \frac{\begin{array}{c} \langle \rho, e_1 \rangle \Downarrow (\!| \lambda x.e, \rho' |\!) \\ \langle \rho, e_2 \rangle \Downarrow v \\ \langle (\rho + \rho')\{x \mapsto v\}, e \rangle \Downarrow r \end{array}}{\langle \rho, e_1\ e_2 \rangle \Downarrow r} \qquad \frac{\begin{array}{c} \langle \rho, e_1 \rangle \Downarrow (\!| \lambda x.e, \rho' |\!) \\ \langle \rho, e_2 \rangle \Downarrow \textbf{fail} \end{array}}{\langle \rho, e_1\ e_2 \rangle \Downarrow \textbf{fail}}\ \textsc{(Eval-Funapp-Fail)}$$

$$\textsc{(Eval-Literal)}\ \frac{}{\langle \rho, v \rangle \Downarrow v}$$

4.8 Rules (Big-step Operational Semantics) specific to V^-

These judgement forms and rules are specific to V^-.

4.8.1 Judgement forms for V^-.

$$\rho \vdash g \rightarrowtail \rho' \quad \textsc{(Guard-Refine)}$$

$$\rho \vdash g \rightarrowtail \dagger \quad \textsc{(Guard-reject)}$$

4.8.2 *Choosing and solving a guard.*

$$\text{(Solve-Guard-Refine)}\ \frac{\rho \vdash g \rightarrowtail \rho' \qquad \rho' \vdash\ \overline{g} \cdot \overline{g}\,' \rightarrowtail s}{\rho \vdash \overline{g} \cdot g \cdot \overline{g}\,' \rightarrowtail s}$$

$$\text{(Solve-Guard-Reject)}\ \frac{\rho \vdash g \rightarrowtail \dagger}{\rho \vdash \overline{g} \cdot g \cdot \overline{g}\,' \rightarrowtail \dagger}$$

4.8.3 *Properties of guards.*

$$\text{(Multi-Guard-Commut)}\ \frac{\rho \vdash \overline{g} \cdot g_1 \cdot g_2 \cdot \overline{g}\,' \rightarrowtail s}{\rho \vdash \overline{g} \cdot g_2 \cdot g_1 \cdot \overline{g}\,' \rightarrowtail s}$$

4.8.4 *Desugaring of Complex Equations.*

$$\text{(Desugar-EqExps)}\ \frac{\begin{array}{c} x \text{ fresh} \\ \rho\{x \mapsto \bot\}; \mathcal{T} \vdash g \Downarrow s \end{array}}{\rho \vdash \overline{g} \cdot e_1 = e_2 \cdot \overline{g}\,' \rightarrowtail s}$$

4.8.5 *Refinement with different types of guards.*

$$\text{(Guard-NameExp-Bot)}\ \frac{\begin{array}{c} x \in \operatorname{dom} \rho \\ \langle \rho, e \rangle \Downarrow v \\ \rho(x) = \bot \end{array}}{\rho \vdash x = e \rightarrowtail \rho\{x \mapsto v\}} \qquad \text{(Guard-NameExp-Eq)}\ \frac{\begin{array}{c} x \in \operatorname{dom} \rho \\ \langle \rho, e \rangle \Downarrow v \\ \rho(x) = v \end{array}}{\rho \vdash x = e \rightarrowtail \rho}$$

$$\frac{\begin{array}{c} x \in \operatorname{dom} \rho \\ \langle \rho, e \rangle \Downarrow v \\ \rho(x) = v' \\ v \neq v' \end{array}}{\rho \vdash x = e \rightarrowtail \dagger}\ \text{(Guard-NameExp-Fail)} \qquad \frac{\begin{array}{c} x,\ y \in \operatorname{dom} \rho \\ \rho(x) = v,\ \rho(y) = \bot \end{array}}{\rho \vdash x = y \rightarrowtail \rho\{y \mapsto v\}}\ \text{(Guard-Names-Bot-Succ)}$$

$$\text{(Guard-Names-Bot-Succ-Rev)}\ \frac{\begin{array}{c} x,\ y \in \operatorname{dom} \rho \\ \rho(x) = \bot,\ \rho(y) = v \end{array}}{\rho \vdash x = y \rightarrowtail \rho\{x \mapsto v\}}$$

$$\text{(Guard-Vcon-Succ)}\ \frac{\begin{array}{c} x \in \operatorname{dom} \rho \\ \rho(x) = K\ v_1, \ldots v_n \\ \rho \vdash y_1 = e_1; \cdots \cdot y_n = e_n \rightarrowtail s, \text{ where} \\ y_i \text{ fresh}, \rho(y_i) = v_i \qquad 1 \leq i \leq n \end{array}}{\rho \vdash x = K\ e_1 \ldots e_n \rightarrowtail s}$$

$$\text{(Guard-Vcon-Fail)}\ \frac{\begin{array}{c} x \in \operatorname{dom} \rho \\ \rho(x) = v \\ v \text{ does not have the form } K[v_1', \ldots v_n'] \end{array}}{\rho \vdash x = K\ e_1, \ldots e_n \rightarrowtail \dagger}$$

$$\text{(Guard-Expseq-Succ)}\ \frac{\langle \rho, e \rangle \Downarrow v}{\rho \vdash e \rightarrowtail \rho} \qquad \frac{\langle \rho, e \rangle \Downarrow \textbf{fail}}{\rho \vdash e \rightarrowtail \dagger}\ \text{(Guard-Expseq-Fail)}$$

$$\text{(Guard-Choice-First)}\ \frac{\rho \vdash \overline{g} \rightarrowtail \rho'}{\rho \vdash \overline{g} \mid \overline{g}' \rightarrowtail \rho'} \qquad \text{(Guard-Choice-Second)}\ \frac{\rho \vdash \overline{g} \rightarrowtail \dagger \qquad \rho \vdash \overline{g}' \rightarrowtail s}{\rho \vdash \overline{g} \mid \overline{g}' \rightarrowtail s}$$

4.8.6 *Evaluating **if-fi**.*

$$\text{(Eval-IfFi-Fail)}\ \frac{}{\langle \rho, \textbf{if}\ \ \textbf{fi} \rangle \Downarrow \textbf{fail}} \qquad \frac{\begin{array}{c} \rho' = \rho\{x_1 \mapsto \bot \ldots x_n \mapsto \bot\} \\ \rho' \vdash \overline{g} \rightarrowtail \rho'' \\ \langle \rho'', e \rangle \Downarrow r \end{array}}{\langle \rho, \textbf{if } \exists\, x_1 \ldots x_n.\ \overline{g} \rightarrow e\ \square; \overline{G}\ \textbf{fi} \rangle \Downarrow r}\ \text{(Eval-IfFi-Success)}$$

$$\text{(Eval-IfFi-Reject)}\ \frac{\begin{array}{c} \rho' = \rho\{x_1 \mapsto \bot \ldots x_n \mapsto \bot\} \\ \rho' \vdash \overline{g} \rightarrowtail \dagger \\ \langle \rho, \textbf{if } \overline{G}\ \textbf{fi} \rangle \Downarrow r \end{array}}{\langle \rho, \textbf{if } \exists\, x_1 \ldots x_n.\ \overline{g} \rightarrow e\ \square\ \overline{G}\ \textbf{fi} \rangle \Downarrow r}$$

5 V^- CAN BE COMPILED TO A DECISION TREE

5.1 Introducing D

While V^- exists for writing programs, D exists as the target of translation and provides a means by which to demonstrate V^-'s efficient cost model. This is because the decision-making construct itself in D, the *decision tree*, has an efficient cost model. A decision tree can implement either pattern matching or **if-fi** while guaranteeing never to repeat a test. The worst-case cost of evaluating a decision tree is linear in its depth, which itself linear in the size of the code. This desirable property of decision trees is half of a space-time tradeoff: when a decision tree is produced by compiling a *case* expression, there are pathological cases in which the total size of the tree is exponential in the size of the source code (from *case*). Run time remains linear, but code size may not be.

Although decision trees are classically used as an intermediate representation for compiling *case* expressions, in this work, I use them as a target for compiling **if-fi** in V^-. In particular, I show that equations in V^- can be compiled to a decision tree.

D is a generalization of the trees found in Maranget [2008].

5.2 D is a generalization of Maranget's trees

D's syntax is given in Figure 16. Decision trees in D are engineered to look like Maranget's trees.

The heart of a decision tree is the *test* form: it takes a value, examines it, and chooses a branch based on its form (Maranget calls the operation Switch).

Let's look at an example from Maranget [2008] which shows the structure of a simple pattern-matching function and its corresponding decision tree. I show Maranget's example for two reasons: First, the example serves to bolster your understanding of decision trees with a classic, well-established model. Second, since I currently have no visualization generator for D (D can currently only be visualized as plain text), I use Maranget's example to have a reasonable visual representation of decision trees. The example beings with the function, `merge`, which merges two lists:

```
let rec merge = match xs,ys with
| [],_ -> ys
| _,[] -> xs
| x::rx,y::ry -> ...
```

Fig. 14. The skeleton of Maranget's `merge`

The function is compiled to this decision tree:

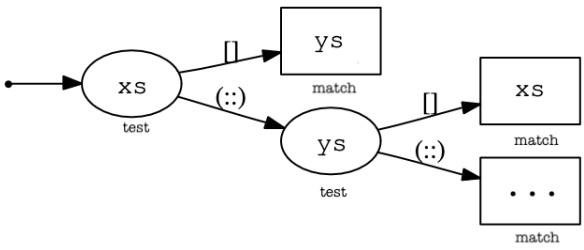

Fig. 15. The final compiled decision tree for `merge`, right-to-left

The decision tree for `merge`, like the original function, tests values and makes decisions. When presented with the values `xs` and `ys`, the tree first tests `xs` against its two known possible forms: the nullary list constructor `[]`, and an application of the *cons* constructor `::`. If `xs` is equal to `[]`, the tree immediately returns `ys`. If `xs` is an application of `::`, the tree then tests `ys` against `[]` and `::`, and it returns a value according to the result of the match. Each time the tree goes down a `::` branch, it extracts the arguments of the `::` for later use: these are `x`, `y`, `xr`, and `yr`, which are used in the `...` branch. This process of extracting arguments generalizes to all value constructors with one or more arguments.

In D, as in Maranget's trees, the *test* node extracts all names from a value constructor at once for use in subtrees. The compiler is responsible for introducing the fresh names used in *test*. The compiler alpha-renames all necessary terms before it translates an **if-fi** to a decision tree to ensure all names are unique.

In D, like in V^-, expressions can *fail*, meaning some of D's syntactic forms like *try-let* and *cmp* have an extra branch which is executed if the examined expression fails.

Programs	P	::=	$\{d\}$	definition
Definitions	d	::=	$val\ x = e$	bind name to expression
Expressions	e	::=	v	literal values
		|	x, y, z	names
		|	t	decision tree
		|	$K\ \{e\}$	value constructor application
		|	$e_1\ e_2$	function application
		|	$\lambda x.\ e$	lambda declaration
Decision Trees	t	::=	$test\ x\ \{K\{y\}\ \Rightarrow\ t\}[else\ t]$	test node
		|	$let\ x = e\ in\ t[\ unless\ fail\ =>\ t]$	let-unless node
		|	$if\ x = e\ then\ t\ else\ t$	comparison node
		|	$\exists\ x.\ t$	exists node
		|	**fail**	fail node
Values	v	::=	$K\{v\}$	value constructor application
		|	$\lambda x.\ e$	lambda value

Fig. 16. D: Concrete syntax

5.3 Rules (Big-step Operational Semantics) for D:

5.3.1 *Evaluating Decision Trees.*

$$\text{(Eval-Test-Fail)}\ \frac{}{\rho \vdash test(x, []) \Downarrow \textbf{fail}}$$

$$\text{(Eval-Test-Succeed)}\ \frac{\rho(x) = K\ \overline{v} \qquad len\ \overline{v} = i \qquad \rho \vdash t \Downarrow r}{\rho \vdash test(x, (K\ \overline{y}, t) \cdot ts) \Downarrow r}$$

$$\text{(Eval-Test-Recurse)}\ \frac{\rho(x) = v \qquad v \text{ does not have the form } K\ \overline{v} \text{ s.t. } len\ \overline{v} = len\ \overline{y} \qquad \rho \vdash test(x, ts), result = r \Downarrow r}{\rho \vdash test(x, (K\ \overline{y}, t) \cdot ts) \Downarrow r}$$

$$\text{(If-First)}\ \frac{\rho \vdash e \Downarrow v \quad \rho(x) = v \quad \rho \vdash t_1 \Downarrow r}{\rho \vdash \mathit{if}\ x = e\ \mathit{then}\ t_1\ \mathit{else}\ t_2\ \mathit{unless}\ \mathbf{fail} \Rightarrow t_3 \Downarrow r}$$

$$\text{(If-Second)}\ \frac{\rho \vdash e \Downarrow v \quad \rho(x) \neq v \quad \rho \vdash t_2 \Downarrow r}{\rho \vdash \mathit{if}\ x = e\ \mathit{then}\ t_1\ \mathit{else}\ t_2\ \mathit{unless}\ \mathbf{fail} \Rightarrow t_3 \Downarrow r}$$

$$\text{(If-Unless)}\ \frac{\rho \vdash e \Downarrow \mathbf{fail} \quad \rho \vdash t_3 \Downarrow r}{\rho \vdash \mathit{if}\ x = e\ \mathit{then}\ t_1\ \mathit{else}\ t_2\ \mathit{unless}\ \mathbf{fail} \Rightarrow t_3 \Downarrow r}$$

$$\text{(Let-Succeed)}\ \frac{\rho \vdash e \Downarrow v \quad \rho\{x \mapsto v\} \vdash t_1 \Downarrow r}{\rho \vdash \mathit{let}\ x = e\ \mathit{in}\ t_1\ \mathit{unless}\ \mathbf{fail} \Rightarrow t_2 \Downarrow r}$$

$$\text{(Let-Unless)}\ \frac{\rho \vdash e \Downarrow \mathbf{fail} \quad \rho \vdash t_2 \Downarrow r}{\rho \vdash \mathit{let}\ x = e\ \mathit{in}\ t_1\ \mathit{unless}\ \mathbf{fail} \Rightarrow t_2 \Downarrow r}$$

$$\text{(Fail)}\ \frac{}{\rho \vdash \mathbf{fail} \Downarrow \mathbf{fail}}$$

$$\text{(Exists)}\ \frac{\rho\{x \mapsto \bot\} \vdash t \Downarrow r}{\rho \vdash \exists x.\ t \Downarrow r}$$

5.4 The $\mathcal{D}$ algorithm: $V^- \rightarrow D$

To demonstrate that V^- has a similarly-desirable cost model to pattern matching, I present an algorithm for compiling V^- to a decision tree. I choose the decision tree as a target for compilation for the simple reason of its appealing cost model. A decision tree can be exponential in size but never examines any word of the *scrutinee*—the value being tested—more than once. This property is established by the compilation from V^- to D, which ensuring that no *test* node T has any proper ancestor T' such that T and T' both test the same location in memory.

There are a few minor differences in the algorithm I use and Maranget's: his compilation algorithm is more complex than the one in this paper, and involves an intermediate representation of occurrence vectors and clause matrices which the algorithm I present does not use. Maranget uses vectors and matrices to express multiple simultaneous matches of values to patterns as a single match of a vector with a matrix row. This allows him to run a *specialization* pass that reduces the number of rows in the matrix, ultimately leading to smaller trees. Because the trees produced by my algorithm still have the linear-in-code-size property, I find them acceptable for the current work.

The algorithm runs during $\mathcal{D}$, the transformation from V^- to D. Its domain, instead of a *case* expression, is V^-'s **if-fi**. $\mathcal{D}$ propagates a context C which maps each defined name to *known* or *unknown*. The context is used when determining the form of a guard.

When presented with an **if-fi**, $\mathcal{D}$ ĩnvokes *compile* with context C. *compile* first desugars the **if-fi** by expanding choice to multiple **if-fi** branches with a desugaring function $\mathcal{I}$:

$\mathcal{I}[\![if\ \ldots\ \square\ gs_1;\ gs_2\ \mathbf{|}\ gs_3;\ gs_4\ \rightarrow e\ \square\ \ldots\ fi]\!]$

==

$if\ \ldots\ \square\ gs_1;\ gs_2\ \rightarrow e\ \square\ gs_3;\ gs_4\ \rightarrow e\ \square\ \ldots\ fi$

With the desugared **if-fi**, *compile* then repeatedly chooses a guarded expression G and applies one of the compilation rules below. The rules are applied in a nondeterministic order.

The algorithm terminates when it inserts a final *match* node (with rule Match). A *match* node is inserted for a right-hand side expression e when the list of guards preceding e is empty or a list of assignments from names to unbound names. Termination of $\mathcal{D}$ ĩs guaranteed because each recursive call passes a list of guarded expressions in which the

number of guards is strictly smaller, so eventually the algorithm reaches a state in which the first unmatched branch is all trivially-satisfied guards.

If *compile* cannot choose a g of one of the valid forms, it halts with an error. This can happen when no g is currently solvable in the context, as determined by the same algorithm that V^- uses to pick a guard to solve, or when the program would be forced to unify incompatible values, such as any value with a closure.

5.5 Big-step rules for *compile*

The judgement form for the compiler is:

$$C \vdash \textbf{if } \overline{G} \textbf{ fi} \hookrightarrow\!\!\!\!\rightarrow e \quad \textsc{(Compile)}$$

5.6 Rules (Big-step Translation) for compiling if-fi

The rules are nondeterministic: the structure of the final result e depends on the order in which rules are applied by the compiler.

$$\textsc{(Exists)}\ \frac{C\{x_1 \mapsto \mathit{unknown}\} \vdash \textbf{if } \overline{G} \ \square\ \exists \ldots x_n.\ \overline{g} \to e' \ \square\ \overline{G}' \textbf{ fi} \hookrightarrow\!\!\!\!\rightarrow e}{C \vdash \textbf{if } \overline{G} \ \square\ \exists x_1 \ldots x_n.\ \overline{g} \to e' \ \square\ \overline{G}' \textbf{ fi} \hookrightarrow\!\!\!\!\rightarrow e}$$

If the compiler finds one or more names introduced by $\exists$, it strips a name from the $\exists$ list and adds it to Cas *unknown*. This is the rule that grows the context with new names, along with lambda, which introduces its argument as *known*.

$$\textsc{(Fail)}\ \frac{C \vdash \textbf{if } \overline{G} \ \square\ \overline{G}' \textbf{ fi} \hookrightarrow\!\!\!\!\rightarrow e}{C \vdash \ \textbf{if } \overline{G} \ \square\ \overline{g};\ \textbf{fail};\ \overline{g}' \ \to e' \ \square\ \overline{G}' \textbf{ fi} \hookrightarrow\!\!\!\!\rightarrow e}$$

If the compiler finds a guarded expression G which contains **fail**, it stops compiling G altogether.

$$\textsc{(Match)}\ \frac{}{C \vdash \textbf{if } \overline{G} \ \square\ \ \square\ \overline{G}' \textbf{ fi} \hookrightarrow\!\!\!\!\rightarrow e'}$$

If the compiler finds an ungarded right-hand side expression e', it inserts a *match* node containing e'.

$$\text{(Test)}\ \frac{\begin{array}{c}\rho(x) = known \\ C + \overline{\{y_i \mapsto known\}} \vdash e\left[K_i\ \overline{y}_i/x\right] \hookrightarrow\!\!\!\!\rightarrow e_i \\ \overline{t} = \sum_i K_i\ \overline{y}_i \Rightarrow e_i \\ e_0 = e\left[K_0/x\right] \qquad K_0 \text{ does not appear in } e\end{array}}{C \vdash\ e \text{ as } \mathbf{if}\ \overline{G} \square \overline{g};\ x = K\ e_1 \ldots e_n;\ \overline{g}'\ \rightarrow e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow test(x, \overline{t}, e_0)}$$

If the compiler finds a known name x equated to the application of a value constructor K, it assembles a set of branches that correspond to varying value constructors. The only requirement of the set is that it contain K. With substiution of $\left[K_i\overline{y}_i/x\right]$, the compiler ensures that x will never be tested again, and also allows for rules Elim-Vcon and Expand-Vcon to make progress.

$$\text{(Elim-Vcon)}\ \frac{\begin{array}{c}C \vdash \mathbf{if}\ \overline{G} \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e \\ e_{lhs} \text{ does not have the form } K\ e'[1] \ldots e'[n]\end{array}}{C \vdash\ \mathbf{if}\ \overline{G} \square \overline{g};\ e_{lhs} = K\ e_1 \ldots e_n;\ \overline{g}'\ \rightarrow e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e}$$

If the compiler finds two equated expressions where one is an application of a value constructor to n arguments and the other isn't, the compiler skips over that branch, like in Fail.

$$\frac{C \vdash\ \mathbf{if}\ \overline{G} \square \overline{g};\ e_1' = e_1;\ \ldots;\ e_n' = e_n;\ \overline{g}'\ \rightarrow e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e}{C \vdash\ \mathbf{if}\ \overline{G} \square \overline{g};\ K\ e_1' \ldots e_n' = K\ e_1 \ldots e_n;\ \overline{g}'\ \rightarrow e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e}\ \text{(Expand-Vcon)}$$

If the compiler finds two equated expressions where each is an application of a value constructor to n arguments, the compiler expands the single equation into a list of equations between the arguments. This rule takes advante of V^-'s sugared equation form $e_1 = e_2$, which desugars to $x = e_1 \cdot x = e_2$, with x fresh.

$$\frac{\begin{array}{c}\rho(x) = unknown \qquad C \vdash e'\ :\ known \\ C\{x \mapsto known\} \vdash e\left[x/e'\right] \hookrightarrow\!\!\!\!\rightarrow t_1 \\ t_2 = e\left[\mathbf{fail}/e'\right]\end{array}}{C \vdash\ e \text{ as } \mathbf{if}\ \overline{G} \square \overline{g};\ x = e;\ \overline{g}'\ \rightarrow e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow let\ x = e'\ in\ t_1\ unless\ \mathbf{fail}\ \Rightarrow\ t_2}\ \text{(Let-Unless)}$$

If the compiler finds an opportunity to bind a name to an expression e', it creates a *let-unless* node and substitutes x for all instances of e', making x known in the new context. In the *unless* ***fail*** branch, **fail** is substituted for e' so that Fail can eliminate guarded expressions containing it.

$$\frac{\begin{array}{c}\rho(x) = known \qquad C \vdash e' \ : \ known \\ t_1 = e\left[x/e'\right] \\ C\{y \mapsto known\} \vdash e\left[\mathbf{fail}/x = e'\right]\left[y/e'\right] \hookrightarrow\!\!\!\!\rightarrow t_2 \qquad y \text{ fresh} \\ t_3 = e\left[\mathbf{fail}/e'\right] \\ t_{if} = if\ x = y\ then\ t_1\ else\ t_2\end{array}}{C \vdash\ e \text{ as } \mathbf{if}\ \overline{G} \ [] \ \overline{g};\ y = e;\ \overline{g}'\ \rightarrow e'\ [] \ \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow let\ x = e'\ in\ t_{if}\ unless\ \mathbf{fail}\ \Rightarrow\ t_3} \text{ (Let-If)}$$

If the compiler finds two expressions being compared e', it creates an *if* node within a *let-unless* node, binding a fresh name y to e' and using it for substitution to ensure e' is never evaluated again. Like in Let-Unless, in the *unless* ***fail*** branch, **fail** is substituted for e' so that Fail can eliminate guarded expressions containing it.

5.7 Reduction Strategies

In my implementation, I apply the Test rule before other rules, having found in my experiments that this heuristic leads to the smallest trees. A risk of inserting a *test* node before a *let-unless* node is that if there are many shared names across branches, a *test* node will introduce those names in *let-unless* nodes in each subtree. It is possible to insert *let-unless* nodes before *test* nodes, in which case those names will be bound only once. The risk of the *let*-first strategy is that if there are many names used in only one subtree of a *test*, these names will be introduced to *all* branches. The extra bindings may harm performance by bloating an environment in an interpreter or thrashing the icache if the tree further is compiled to machine code.

In the implementation, $\mathcal{D}$ first introduces all the names under the all existential $\exists$'s to a context which determines if a name is *known* or *unknown*, applying the Exists rule is applied all at the start. At the start of compilation, each name introduced by $\exists$ is *unknown* in a context C. Since all names in the program are unique at this stage, there are no clashes.

5.8 Full Big-step rules for *compile*, with no descriptions

The judgement form for compilation is Compile:

$$C \vdash \mathbf{if}\ \overline{G}\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e \quad \text{(Compile)}$$

The rules are nondeterministic: the structure of the final result e depends on the order in which rules are applied by the compiler.

$$\text{(Exists)}\ \frac{C\{x_1 \mapsto \mathit{unknown}\} \vdash \mathbf{if}\ \overline{G} \square \exists \ldots x_n.\ \overline{g} \rightarrow e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e}{C \vdash \mathbf{if}\ \overline{G} \square \exists x_1 \ldots x_n.\ \overline{g} \rightarrow e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e}$$

$$\text{(Fail)}\ \frac{C \vdash \mathbf{if}\ \overline{G} \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e}{C \vdash \ \mathbf{if}\ \overline{G} \square \overline{g};\ \mathbf{fail};\ \overline{g}'\ \rightarrow e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e}$$

$$\text{(Match)}\ \frac{}{C \vdash \mathbf{if}\ \overline{G} \square\ \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e}$$

$$\text{(Test)}\ \frac{\begin{array}{c}\rho(x) = \mathit{known} \\ C + \overline{\{y_i \mapsto \mathit{known}\}} \vdash e\left[K_i\ \overline{y}_i/x\right] \hookrightarrow\!\!\!\!\rightarrow e_i \\ \overline{t} = \sum_i K_i\ \overline{y}_i \Rightarrow e_i \\ e_0 = e\left[K_0/x\right] \qquad K_0 \text{ does not appear in } e\end{array}}{C \vdash\ e \text{ as } \mathbf{if}\ \overline{G} \square \overline{g};\ x = K\ e_1 \ldots e_n;\ \overline{g}'\ \rightarrow e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow \mathit{test}(x, \overline{t}, e_0)}$$

$$\text{(Elim-Vcon)}\ \frac{\begin{array}{c}C \vdash \mathbf{if}\ \overline{G} \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e \\ e_{lhs} \text{ does not have the form } K\ e'[1] \ldots e'[n]\end{array}}{C \vdash\ \mathbf{if}\ \overline{G} \square \overline{g};\ e_{lhs} = K\ e_1 \ldots e_n;\ \overline{g}'\ \rightarrow e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e}$$

$$\frac{C \vdash\ \mathbf{if}\ \overline{G} \square \overline{g};\ e_1' = e_1;\ \ldots;\ e_n' = e_n;\ \overline{g}'\ \rightarrow e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e}{C \vdash\ \mathbf{if}\ \overline{G} \square \overline{g};\ K\ e_1' \ldots e_n' = K\ e_1 \ldots e_n;\ \overline{g}'\ \rightarrow e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e}\ \text{(Expand-Vcon)}$$

$$\frac{\begin{array}{c}\rho(x) = \mathit{unknown} \qquad C \vdash e' \ : \ \mathit{known} \\ C\{x \mapsto \mathit{known}\} \vdash e\left[x/e'\right] \hookrightarrow\!\!\!\!\rightarrow\ t_1 \\ t_2 = e\left[\mathbf{fail}/e'\right]\end{array}}{C \vdash \ e \text{ as } \ \mathbf{if}\ \overline{G} \ \square\ \overline{g};\ x = e;\ \overline{g}'\ \rightarrow e' \ \square\ \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow\ \mathit{let}\ x = e'\ \mathit{in}\ t_1\ \mathit{unless}\ \mathbf{fail}\ \Rightarrow\ t_2} \ \text{(Let-Unless)}$$

$$\frac{\begin{array}{c}\rho(x) = \mathit{known} \qquad C \vdash e' \ : \ \mathit{known} \\ t_1 = e\left[x/e'\right] \\ C\{y \mapsto \mathit{known}\} \vdash e\left[\mathbf{fail}/x = e'\right]\left[y/e'\right] \hookrightarrow\ t_2 \qquad y \text{ fresh} \\ t_3 = e\left[\mathbf{fail}/e'\right] \\ t_{if} = \mathit{if}\ x = y\ \mathit{then}\ t_1\ \mathit{else}\ t_2\end{array}}{C \vdash \ e \text{ as } \ \mathbf{if}\ \overline{G} \ \square\ \overline{g};\ y = e;\ \overline{g}'\ \rightarrow e' \ \square\ \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow\ \mathit{let}\ x = e'\ \mathit{in}\ t_{if}\ \mathit{unless}\ \mathbf{fail}\ \Rightarrow\ t_3} \ \text{(Let-If)}$$

5.9 Translation from V^- to D preserves semantics

Translating **if-fi** to a decision tree should preserve semantics:

Conjecture 5.1. *Given a specific but arbitrarily chosen environment ρ and context C, when $\langle \rho, e\rangle \Downarrow r_1$ and $C \vdash e \hookrightarrow\!\!\!\!\rightarrow\ e'$ and $\langle \rho, e'\rangle \Downarrow r_2$, then $r_1 = r_2$.*

Proving this conjecture is the subject of future work.

6 IMPLEMENTATIONS

I have placed implementations of V^- and D at https://github.com/rogerburtonpatel/vml. The implementations are complete, from parsers to evaluation to unparsers. In the same repository lives the `dtran` program, which translates from V^- to D. With the implementations, I include test cases for evaluation of V^-, evaluation of D, and the translation between the two. In each of these test cases, the translation preserves semantics.

7 RELATED AND FUTURE WORK

This paper builds on Augustsson et al.'s Verse Calculus [Augustsson et al. 2023] and decision trees [Maranget 2008]. Augustsson et al. give the formal rewrite semantics for the

Verse Calculus; Maranget gives an elegant formalism of decision trees and a translation algorithm from patterns to decision trees. I attempted to imitate the behavior of the rewrite semantics of $\mathcal{VC}$ in the big-step semantics of V^- by manually rewriting terms in $\mathcal{VC}$ and creating rules that would imitate the ultimate result of term-rewriting. Proving the equivalence between the two semantics is the subject of future work. I chose a big-step semantics because it is the style of semantics I am most comfortable with; writing the formalisms this way helped me write the code. Using a rewrite semantics instead would more closely relate V^- and $\mathcal{VC}$, and is a likely future project. Maranget's formalism was the foundation off of which I built D.

Extensions to pattern matching, and how they appeal to language designers, find an excellent example in Erwig and Jones [2001]. The authors describe pattern guards and transformational patterns (another extension to pattern matching), both of which allow a Haskell programmer to write more concise code using pattern matching. Or-patterns are documented in the OCaml Language Reference Manual [Leroy et al. 2023].

Augustsson [1985] gives a foundation in compiling pattern matching. Scott and Ramsey [2000] have a crisp example of a match-compilation algorithm (pattern matching to decision trees). Scott and Ramsey's algorithm structurally inspired mine, and studying the source code from the paper aided my implementation.

For future work, my top priority is to prove that $\mathcal{D}$ preserves semantics. Next, I plan to prove that V^- is deterministic, and that **the big-step semantics of V^- is consistent with the published semantics of $\mathcal{VC}$.** As the authors of the Verse paper proved that the rewrite semantics of Verse is skew-confluent, I plan to prove that big-step semantics of V^- is deterministic, despite the nondeterminism of choosing a guard. Second, V^- is designed to be Verse-like, and formalizing the relationship between the two would strengthen V^-'s viability as a language that compromises between pattern matching and equations.

Finally, I would like to explore exhaustiveness analysis of V^-. Exhaustiveness analysis can warn programmers of a missing or extraneous alternative in a *case* expression. Owing to its significantly more flexible structure, however, **if-fi** may prove trickier to analyze.

8 DISCUSSION: THE DESIGN OF V^-

This section is aimed at audience familiar with $\mathcal{VC}$; in particular, with the concept of *names as values*.

8.1 Forms in $\mathcal{VC}$ and V^-

When designing V^-, I wanted the language to capture the expressiveness of $\mathcal{VC}$'s equations while retaining a similar decision-making construct to pattern matching. In pattern matching, the only decision-making construct is *case*; other forms of decision-making like *if* desugar to it. In V^-, the only decision-making construct is **if-fi**. This design differs from $\mathcal{VC}$, which features numerous ways to make decisions by combining **one**, **all**, equations, intermediate expressions, and choice. I did not want multiple results in V^-, so I eliminated anything that looked like **all**, and I combined all of the above constructs into the singular form **if-fi**. Like *case*, there is only one way to use **if-fi**- unlike **one**, equations, intermediate expressions, and choice in $\mathcal{VC}$. By restricting all decision-making to **if-fi**, programmers cannot "misuse" any of the forms in ways that might lead to problematic computations, such as multiple results by returning choice as an expression. Furthermore, as I've shown in the examples, the way in which programmers use **if-fi** mirrors the way they use *case*.

8.2 Choice in V^- vs. $\mathcal{VC}$

$\mathcal{VC}$'s choice operator is often the culprit behind both backtracking and multiple results, which tempted me to remove choice from V^- altogether. However, I want to harness the expressive potential of choice, particularly when paired with $\mathcal{VC}$'s **one** keyword. When combined with choice, **one** elegantly signifies "proceed if any branch of the choice succeeds."

To this end, in V^-, choice is permitted with several modifications:

(1) Choice may only appear as a condition or 'guard', not as a result or the right-hand side of a binding.
(2) If any branch of the choice succeeds, the choice succeeds, producing any bindings found in that branch. The program examines the branches in a left-to-right order.
(3) The existential $\exists$ may not appear under choice.

I introduce one more crucial modification to $\mathcal{VC}$: a name in V^- is an *expression*, not a *value*. This modification, coupled with my adjustments to choice, eradicates backtracking. My rationale is straightforward: if an expression returns a name, and if the program later imposes a new constraint on that name, it may necessitate the reevaluation of the earlier expression—a scenario I insist on avoiding.

9 CONCLUSION

I have introduced V^-, a language that makes decisions using equations. By example, I have shown that programs written in V^- can have the same desirable properties as equivalent programs written using pattern matching. And to show that equations can be compiled to efficient decision trees, I have introduced D and $\mathcal{D}$. In doing so, I have demonstrated that programming with equations is a promising alternative to pattern matching.

I have also fully implemented the languages. They exist for use and experimentation: they are syntactically simple and have conceptually accessible operational semantics. I hope that programmers will explore and develop their own opinions of these languages, which are publically available at https://github.com/rogerburtonpatel/vml.

10 ACKNOWLEDGEMENTS

This thesis would not have been possible without the infinitely generous time and support of my advisors, Norman Ramsey and Milod Kazerounian. Norman's offhand comment of "I wonder if Verse's equations subsume pattern matching" was the entire basis of this work, and his generosity in agreeing to advise a full thesis to answer his question will always be profoundly appreciated. During the academic year, Norman provided me with materials on both the technical story and on how to write about it well. He gave me regular feedback and has helped improve my research skills, my technical writing, and my understanding of programming languages in general tremendously. I especially appreciate how he has guided me in-person at the end of my undergrad when his book [Ramsey 2022] got me started down the path of PL at the beginning of it. Finally, Norman is also fantastically fun to pair program with.

From the beginning, Milod provided me with encouraging mentorship that kept me enthusiastic and determined to complete the project. He was exceptionally patient as I gave him whirlwind tour after whirlwind tour of the changing codebase and problems, and he

kept me grounded in the problems at hand. He sent me helpful examples of his research to aid me in my proofs, and gave me some particularly encouraging words towards the end of the project that I will not soon forget.

My undergraduate advisor, Mark Sheldon, has always been both supportive and kind. I have enjoyed many long talks in his office, and I am deeply grateful that he is on my committee.

Alva Couch was the advisor of my original thesis idea, which was to compile programming languages with music. Ultimately, we decided that I should pursue this project instead, and I am grateful to him for his mentorship in that moment and onwards.

My family—my mother, Jennifer Burton, my father, Aniruddh Patel, and my sister, Lilia Burtonpatel, have all given me support, encouragement, and (arguably most importantly) food. My gratitude for them is immeasurable.

My gratitude towards my friends is also without limit. In particular, and in no order, Liam Strand, Annika Tanner, Max Stein, Cecelia Crumlish, and Charlie Bohnsack all showed specific interest in the work and encouraged me as I moved forward. Rachael Clawson was subjected to much rubber-ducking, and endured valiantly. Aliénor Rice and Marie Kazibwe were as steadfast thesis buddies as I could ever hope for. Jasper Geer, my PL partner in crime, was always one of my favorite people to talk to about my thesis. His pursuits in research inspire mine.

Skylar Gilfeather, my unbelievable friend. Yours is support that goes beyond words; care, food, silent and spoken friendship, late night rides to anywhere, laughs and tears are some that can try to capture it. I am so, so grateful for how ceaselessly you've encouraged me on this journey. Every one of your friends is lucky to have you in their life, and I am blessed that you are such a core part of mine.

Anna Quirós, I am writing these words as you sleep behind me. Your support and love have been immeasurable. I will always be grateful to you for this year *increíble*. I could not have smiled through it all without you.

Thank you all.

REFERENCES

Sergio Antoy and Michael Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.

Lennart Augustsson. Compiling pattern matching. In *Conference on Functional Programming Languages and Computer Architecture*, pages 368–381. Springer, 1985.

Lennart Augustsson, Joachim Breitner, Koen Claessen, Ranjit Jhala, Simon Peyton Jones, Olin Shivers, Guy L Steele Jr, and Tim Sweeney. The verse calculus: A core calculus for deterministic functional logic programming. *Proceedings of the ACM on Programming Languages*, 7(ICFP):417–447, 2023.

Jonas Barklund and Robert Virding. Erlang 4.7. 3 reference manual draft (0.7). *Ericsson AB*, page 79, 1999.

Marianne Baudinet and David MacQueen. Tree pattern matching for ML. Available from David MacQueen, AT&T Bell Laboratories, 600 Mountain Avenue, Murray, Hill, NJ 07974, 1986.

F Warren Burton and Robert D Cameron. Pattern matching with abstract data types1. *Journal of Functional Programming*, 3(2):171–190, 1993.

Alonzo Church. *The calculi of lambda-conversion*. Princeton University Press, 1985.

KL Clark. An introduction to logic programming. *Introductory Readings in Expert Systems, ed. D. Michie*, pages 93–112, 1982.

Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

Martin Erwig and Simon Peyton Jones. Pattern guards and transformational patterns. *Electronic Notes in Theoretical Computer Science*, 41(1):3, 2001.

Michael Hanus. Functional logic programming: From theory to curry. *Programming Logics: Essays in Memory of Harald Ganzinger*, pages 123–168, 2013.

Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.

Wen Kokke, Jeremy G Siek, and Philip Wadler. Programming language foundations in agda. *Science of Computer Programming*, 194:102440, 2020.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, KC Sivaramakrishnan, and Jérôme Vouillon. The ocaml system release 5.1: Documentation and user's manual, 2023.

Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press/McGraw-Hill, Cambridge, MA, 1986.

Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 35–46, 2008.

Simon Marlow et al. Haskell 2010 language report: Chapter 3. 2010. URL https://www.haskell.org/onlinereport/haskell2010/.

Pedro Palao Gostanza, Ricardo Pena, and Manuel Núnez. A new look at pattern matching in abstract data types. *ACM SIGPLAN Notices*, 31(6):110–121, 1996.

Simon Peyton-Jones. Verification in verse, April 2024.

Norman Ramsey. *Programming Languages: Build, Prove, and Compare*. Cambridge University Press, 2022.

J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, jan 1965. ISSN 0004-5411. doi: 10.1145/321250.321253. URL https://doi.org/10.1145/321250.321253.

Kevin Scott and Norman Ramsey. When do match-compilation heuristics matter? Technical Report CS-2000-13, Department of Computer Science, University of Virginia, May 2000.

Don Syme, Luke Hoban, Tao Liu, Dmitry Lomov, James Margetson, Brian McNamara, Joe Pamer, Penny Orwick, Daniel Quirk, Chris Smith, et al. The f# 2.0 language specification. *Microsoft, August*, 2010.

The Elixir Team. Elixir documentation. URL https://hexdocs.pm/elixir/index.html.

Philip Wadler. How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Conference on Functional Programming Languages and Computer Architecture*, pages 113–128. Springer, 1985.

Philip Wadler. 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*, pages 307–313, 1987.

École Polytechnique Fédérale. Pattern matching - the scala programming language. https://docs.scala-lang.org/tour/pattern-matching.html.

A IS V^- A TRUE SUBSET OF $\mathcal{VC}$?

V^- certainly appears to relate to $\mathcal{VC}$ semantically. If they are formally related, V^- may inform programming in $\mathcal{VC}$, which could help new Verse programmers learn the language if they are familiar with more traditional decision-making constructs like **if-fi**. By starting with V^-, they could move from subset to full set as they learn the full Verse language.

Translating **if-fi** and choice in V^- to **one** and choice in $\mathcal{VC}$ is likely a sufficient embedding. Formalizing this translation and, more importantly, proving that my semantics of V^- are consistent with Augutsson et al.'s $\mathcal{VC}$ is an excellent goal for future work.

B FORMAL DEFINITIONS OF ALL LANGUAGES

Syntactic Forms	Cases
P : Programs	$\{d\}$
d : Definitions	**val** $x = e$
v: Values	$K\{v\}$
	$\lambda x.\ e$
e : Expressions	v
	x, y, z
	$K\{e\}$
	$\lambda x.\ e$
	$e_1\ e_2$
	if $\overline{G}$ **fi**
	t
G : Guarded Expressions	$[\exists \overline{x}.]\ \overline{g} \rightarrow e$
g : Guards	$x = e$
	e
	$\overline{g}$ **|** $\overline{g}'$
t : Decision Trees	$test\ x\ \{K_i/\overline{y}_i\ \Rightarrow\ t\}\ [else\ t]$
	e
	$\exists x.\ t$
	$let\ x = e\ in\ t[;\ unless\ \textbf{fail} \Rightarrow\ t]$
	$if\ x == e\ then\ t\ else\ t$
	fail

Table 3. Abstract Syntax of V^- and D. Forms in black are present in both languages, forms in red are specific to V^-, and forms in blue are specific to D.

B.1 Rules (Big-step Operational Semantics) for U, shared by V^- and D

B.1.1 Judgement forms for U. In both V^- and D, an expression evaluates to produce a single *result*. A result is either a single value v or **fail**.

$$\langle \rho, e \rangle \Downarrow r \quad \text{(Eval)}$$

B.2 Evaluating general expressions in U

Here, I show all the rules which are shared by V^- and D under U.

$$\text{(Eval-Vcon)}\ \frac{\langle \rho, e_i \rangle \Downarrow v_i \qquad 1 \leq i \leq n}{\langle \rho, K(e_1, \ldots e_n) \rangle \Downarrow K(v_1, \ldots v_i)} \qquad \frac{\exists e_i.\ 1 \leq i \leq n : \langle \rho, e_i \rangle \Downarrow \textbf{fail}}{\langle \rho, K(e_1, \ldots e_n) \rangle \Downarrow \textbf{fail}}\ \text{(Eval-Vcon-Fail)}$$

$$\text{(Eval-Name)}\ \frac{x \in \operatorname{dom} \rho \quad \rho(x) = v}{\langle \rho, x \rangle \Downarrow v} \qquad \frac{}{\langle \rho, \lambda x.e \rangle \Downarrow (\!|\lambda x.e, \rho|\!)}\ \text{(Eval-LambdaDecl)}$$

$$\text{(Eval-Funapp)}\ \frac{\langle \rho, e_1 \rangle \Downarrow (\!|\lambda x.e, \rho'|\!) \quad \langle \rho, e_2 \rangle \Downarrow v \quad \langle (\rho + \rho')\{x \mapsto v\}, e \rangle \Downarrow r}{\langle \rho, e_1\ e_2 \rangle \Downarrow r} \qquad \frac{\langle \rho, e_1 \rangle \Downarrow (\!|\lambda x.e, \rho'|\!) \quad \langle \rho, e_2 \rangle \Downarrow \textbf{fail}}{\langle \rho, e_1\ e_2 \rangle \Downarrow \textbf{fail}}\ \text{(Eval-Funapp-Fail)}$$

$$\text{(Eval-Literal)}\ \frac{}{\langle \rho, v \rangle \Downarrow v}$$

B.3 Rules (Big-step Operational Semantics) specific to V^-

These judgement forms and rules are specific to V^-.

B.3.1 Judgement forms for V^-.

$$\rho \vdash g \rightarrowtail \rho' \quad \text{(Guard-Refine)}$$

$$\rho \vdash g \rightarrowtail \dagger \quad \text{(Guard-reject)}$$

B.3.2 Choosing and solving a guard.

$$\text{(Solve-Guard-Refine)}\ \frac{\rho \vdash g \rightarrowtail \rho' \qquad \rho' \vdash\ \overline{g} \cdot \overline{g}\,' \rightarrowtail s}{\rho \vdash \overline{g} \cdot g \cdot \overline{g}\,' \rightarrowtail s}$$

$$\text{(Solve-Guard-Reject)}\ \frac{\rho \vdash g \rightarrowtail \dagger}{\rho \vdash \overline{g} \cdot g \cdot \overline{g}\,' \rightarrowtail \dagger}$$

B.3.3 Properties of guards.

$$\text{(Multi-Guard-Commut)}\ \frac{\rho \vdash \overline{g} \cdot g_1 \cdot g_2 \cdot \overline{g}\,' \rightarrowtail s}{\rho \vdash \overline{g} \cdot g_2 \cdot g_1 \cdot \overline{g}\,' \rightarrowtail s}$$

B.3.4 Desugaring of Complex Equations.

$$\text{(Desugar-EqExps)}\ \frac{\begin{array}{c} x \text{ fresh} \\ \rho\{x \mapsto \bot\}; \mathcal{T} \vdash g \Downarrow s \end{array}}{\rho \vdash \overline{g} \cdot e_1 = e_2 \cdot \overline{g}\,' \rightarrowtail s}$$

B.3.5 Refinement with different types of guards.

$$\text{(Guard-NameExp-Bot)}\ \frac{\begin{array}{c} x \in \operatorname{dom} \rho \\ \langle \rho, e \rangle \Downarrow v \\ \rho(x) = \bot \end{array}}{\rho \vdash x = e \rightarrowtail \rho\{x \mapsto v\}} \qquad \text{(Guard-NameExp-Eq)}\ \frac{\begin{array}{c} x \in \operatorname{dom} \rho \\ \langle \rho, e \rangle \Downarrow v \\ \rho(x) = v \end{array}}{\rho \vdash x = e \rightarrowtail \rho}$$

$$\frac{\begin{array}{c} x \in \operatorname{dom} \rho \\ \langle \rho, e \rangle \Downarrow v \\ \rho(x) = v' \\ v \neq v' \end{array}}{\rho \vdash x = e \rightarrowtail \dagger}\ \text{(Guard-NameExp-Fail)} \qquad \frac{\begin{array}{c} x,\ y \in \operatorname{dom} \rho \\ \rho(x) = v,\ \rho(y) = \bot \end{array}}{\rho \vdash x = y \rightarrowtail \rho\{y \mapsto v\}}\ \text{(Guard-Names-Bot-Succ)}$$

$$\text{(Guard-Names-Bot-Succ-Rev)}\ \frac{\begin{array}{c} x,\ y \in \operatorname{dom} \rho \\ \rho(x) = \bot,\ \rho(y) = v \end{array}}{\rho \vdash x = y \rightarrowtail \rho\{x \mapsto v\}}$$

$$\text{(Guard-Vcon-Succ)}\ \frac{\begin{array}{c} x \in \operatorname{dom}\rho \\ \rho(x) = K\ v_1, \ldots v_n \\ \rho \vdash y_1 = e_1; \cdots \cdot y_n = e_n \rightarrowtail s, \text{ where} \\ y_i \text{ fresh}, \rho(y_i) = v_i \qquad 1 \leq i \leq n \end{array}}{\rho \vdash x = K\ e_1 \ldots e_n \rightarrowtail s}$$

$$\text{(Guard-Vcon-Fail)}\ \frac{\begin{array}{c} x \in \operatorname{dom}\rho \\ \rho(x) = v \\ v \text{ does not have the form } K[v_1', \ldots v_n'] \end{array}}{\rho \vdash x = K\ e_1, \ldots e_n \rightarrowtail \dagger}$$

$$\text{(Guard-Expseq-Succ)}\ \frac{\langle \rho, e \rangle \Downarrow v}{\rho \vdash e \rightarrowtail \rho} \qquad \frac{\langle \rho, e \rangle \Downarrow \textbf{fail}}{\rho \vdash e \rightarrowtail \dagger}\ \text{(Guard-Expseq-Fail)}$$

$$\text{(Guard-Choice-First)}\ \frac{\rho \vdash \overline{g} \rightarrowtail \rho'}{\rho \vdash \overline{g}\ \mathbf{|}\ \overline{g}' \rightarrowtail \rho'} \qquad \text{(Guard-Choice-Second)}\ \frac{\rho \vdash \overline{g} \rightarrowtail \dagger \qquad \rho \vdash \overline{g}' \rightarrowtail s}{\rho \vdash \overline{g}\ \mathbf{|}\ \overline{g}' \rightarrowtail s}$$

*B.3.6 Evaluating **if-fi**.*

$$\text{(Eval-IfFi-Fail)}\ \frac{}{\langle \rho, \textbf{if}\quad \textbf{fi} \rangle \Downarrow \textbf{fail}} \qquad \frac{\begin{array}{c} \rho' = \rho\{x_1 \mapsto \bot \ldots x_n \mapsto \bot\} \\ \rho' \vdash \overline{g} \rightarrowtail \rho'' \\ \langle \rho'', e \rangle \Downarrow r \end{array}}{\langle \rho, \textbf{if } \exists\, x_1 \ldots x_n.\ \overline{g} \rightarrow e\ \square; \overline{G}\ \textbf{fi} \rangle \Downarrow r}\ \text{(Eval-IfFi-Success)}$$

$$\text{(Eval-IfFi-Reject)}\ \frac{\begin{array}{c} \rho' = \rho\{x_1 \mapsto \bot \ldots x_n \mapsto \bot\} \\ \rho' \vdash \overline{g} \rightarrowtail \dagger \\ \langle \rho, \textbf{if } \overline{G}\ \textbf{fi} \rangle \Downarrow r \end{array}}{\langle \rho, \textbf{if } \exists\, x_1 \ldots x_n.\ \overline{g} \rightarrow e\ \square\ \overline{G}\ \textbf{fi} \rangle \Downarrow r}$$

B.4 Rules (Big-step Operational Semantics) for D:

B.4.1 Evaluating Decision Trees.

$$\text{(Eval-Test-Fail)}\ \frac{}{\rho \vdash test(x, []) \Downarrow \textbf{fail}}$$

$$\text{(Eval-Test-Succeed)}\ \frac{\rho(x) = K\ \overline{v} \qquad len\ \overline{v} = i \qquad \rho \vdash t \Downarrow r}{\rho \vdash test(x, (K\ \overline{y}, t) \cdot ts) \Downarrow r}$$

$$\text{(Eval-Test-Recurse)}\ \frac{\rho(x) = v \qquad v \text{ does not have the form } K\ \overline{v} \text{ s.t. } len\ \overline{v} = len\ \overline{y} \qquad \rho \vdash test(x, ts), result = r \Downarrow r}{\rho \vdash test(x, (K\ \overline{y}, t) \cdot ts) \Downarrow r}$$

$$\text{(If-First)}\ \frac{\rho \vdash e \Downarrow v \qquad \rho(x) = v \qquad \rho \vdash t_1 \Downarrow r}{\rho \vdash if\ x = e\ then\ t_1\ else\ t_2\ unless\ \textbf{fail} \Rightarrow t_3 \Downarrow r}$$

$$\text{(If-Second)}\ \frac{\rho \vdash e \Downarrow v \qquad \rho(x) \neq v \qquad \rho \vdash t_2 \Downarrow r}{\rho \vdash if\ x = e\ then\ t_1\ else\ t_2\ unless\ \textbf{fail} \Rightarrow t_3 \Downarrow r}$$

$$\text{(If-Unless)}\ \frac{\rho \vdash e \Downarrow \textbf{fail} \qquad \rho \vdash t_3 \Downarrow r}{\rho \vdash if\ x = e\ then\ t_1\ else\ t_2\ unless\ \textbf{fail} \Rightarrow t_3 \Downarrow r}$$

$$\text{(Let-Succeed)}\ \frac{\rho \vdash e \Downarrow v \qquad \rho\{x \mapsto v\} \vdash t_1 \Downarrow r}{\rho \vdash \mathit{let}\ x = e\ \mathit{in}\ t_1\ \mathit{unless}\ \mathbf{fail} \Rightarrow t_2 \Downarrow r}$$

$$\text{(Let-Unless)}\ \frac{\rho \vdash e \Downarrow \mathbf{fail} \qquad \rho \vdash t_2 \Downarrow r}{\rho \vdash \mathit{let}\ x = e\ \mathit{in}\ t_1\ \mathit{unless}\ \mathbf{fail} \Rightarrow t_2 \Downarrow r}$$

$$\text{(Fail)}\ \frac{}{\rho \vdash \mathbf{fail} \Downarrow \mathbf{fail}}$$

$$\text{(Exists)}\ \frac{\rho\{x \mapsto \bot\} \vdash t \Downarrow r}{\rho \vdash \exists x.\ t \Downarrow r}$$

B.5 Full Big-step rules for *compile*, with no descriptions

$$C \vdash \mathbf{if}\ \overline{G}\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e\ \ \text{(Compile)}$$

B.6 Rules (Big-step Translation) for compiling if-fi

These rules in a nondeterministic order by the compiler.

$$\text{(Exists)}\ \frac{C\{x_1 \mapsto \mathit{unknown}\} \vdash \mathbf{if}\ \overline{G} \square \exists \ldots x_n.\ \overline{g} \to e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e}{C \vdash \mathbf{if}\ \overline{G} \square \exists x_1 \ldots x_n.\ \overline{g} \to e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e}$$

$$\text{(Fail)}\ \frac{C \vdash \mathbf{if}\ \overline{G} \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e}{C \vdash \ \mathbf{if}\ \overline{G} \square \overline{g};\ \mathbf{fail};\ \overline{g}'\ \to e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e}$$

$$\text{(Match)}\ \frac{}{C \vdash \mathbf{if}\ \overline{G} \square\ \ \square\ \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e}$$

$$\text{(Test)}\ \frac{\begin{array}{c}\rho(x) = known \\ C + \overline{\{y_i \mapsto known\}} \vdash e\left[K_i\ \overline{y}_i/x\right] \hookrightarrow\!\!\!\!\rightarrow e_i \\ \overline{t} = \sum_i K_i\ \overline{y}_i \Rightarrow e_i \\ e_0 = e\left[K_0/x\right] \qquad K_0 \text{ does not appear in } e\end{array}}{C \vdash\ e \text{ as } \mathbf{if}\ \overline{G} \square \overline{g};\ x = K\ e_1 \ldots e_n;\ \overline{g}'\ \rightarrow e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow test(x, \overline{t}, e_0)}$$

$$\text{(Elim-Vcon)}\ \frac{\begin{array}{c}C \vdash \mathbf{if}\ \overline{G} \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e \\ e_{lhs} \text{ does not have the form } K\ e'[1] \ldots e'[n]\end{array}}{C \vdash\ \mathbf{if}\ \overline{G} \square \overline{g};\ e_{lhs} = K\ e_1 \ldots e_n;\ \overline{g}'\ \rightarrow e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e}$$

$$\frac{C \vdash\ \mathbf{if}\ \overline{G} \square \overline{g};\ e_1' = e_1;\ \ldots;\ e_n' = e_n;\ \overline{g}'\ \rightarrow e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e}{C \vdash\ \mathbf{if}\ \overline{G} \square \overline{g};\ K\ e_1' \ldots e_n' = K\ e_1 \ldots e_n;\ \overline{g}'\ \rightarrow e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow e}\ \text{(Expand-Vcon)}$$

$$\frac{\begin{array}{c}\rho(x) = unknown \qquad C \vdash e'\ :\ known \\ C\{x \mapsto known\} \vdash e\left[x/e'\right] \hookrightarrow\!\!\!\!\rightarrow t_1 \\ t_2 = e\left[\mathbf{fail}/e'\right]\end{array}}{C \vdash\ e \text{ as } \mathbf{if}\ \overline{G} \square \overline{g};\ x = e;\ \overline{g}'\ \rightarrow e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow let\ x = e'\ in\ t_1\ unless\ \mathbf{fail}\ \Rightarrow\ t_2}\ \text{(Let-Unless)}$$

$$\frac{\begin{array}{c}\rho(x) = known \qquad C \vdash e'\ :\ known \\ t_1 = e\left[x/e'\right] \\ C\{y \mapsto known\} \vdash e\left[\mathbf{fail}/x = e'\right]\left[y/e'\right] \hookrightarrow\!\!\!\!\rightarrow t_2 \qquad y \text{ fresh} \\ t_3 = e\left[\mathbf{fail}/e'\right] \\ t_{if} = if\ x = y\ then\ t_1\ else\ t_2\end{array}}{C \vdash\ e \text{ as } \mathbf{if}\ \overline{G} \square \overline{g};\ y = e;\ \overline{g}'\ \rightarrow e' \square \overline{G}'\ \mathbf{fi} \hookrightarrow\!\!\!\!\rightarrow let\ x = e'\ in\ t_{if}\ unless\ \mathbf{fail}\ \Rightarrow\ t_3}\ \text{(Let-If)}$$