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 via compilation to a decision tree, but certain computations can only be expressed verbosely using pattern matching primitives. The problem can be solved by extensions, but these are not standardized, and no single popular programming language contains all the extensions to pattern matching. By contrast, equations in Verse are both expressive and succinct, with no obvious need for extensions; however, Verse's cost model is a challenge. To resolve these problems, I propose a new language, V^-, which uses Verse's equations with some restrictions. I show that V^- subsumes P^+, a core language with three popular extensions to pattern matching. I also show that V^- can be compiled to a decision tree.

1 INTRODUCTION

Pattern matching is a beloved tool among functional programmers for examining and deconstructing data. It is also an established and well-researched topic [Burton and Cameron 1993; MacQueen and Baudinet 1985; Maranget 2008; Palao Gostanza et al. 1996; Ramsey 2022; Wadler 1987]. Pattern matching 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 2.2).

Extensions strengthen pattern matching, but they are not standardized, so each popular programming language with pattern matching features its own unique suite of extensions.

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

This makes extensions less of a ubiquitous feature of pattern matching and more subject to the discretion of the individual language designer. Rather than continuing to extend pattern matching across various frontiers *ad hoc*, a worthwhile goal could be to find an alternative to pattern matching that doesn't need extensions to succinctly solve the problems programmers face. A tempting possibility for such an alternative was introduced last year by the programming language Verse [Augustsson et al. 2023]. In Verse, a programmer can implicitly deconstruct data without pattern matching 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.

My central contribution in this thesis is to 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"). In this thesis, I also show that V^- subsumes pattern matching with popular extensions.

To support my central contribution, I have formalized pattern matching in a core language P^+ ("P plus") with a big-step operational semantics (Section 3.1), I have formalized a subset of Verse into a core language V^- with a big-step operational semantics (Section 3.4), I have formalized decision trees into a core language D ("D") with a big-step operational semantics (Section 3.8), I have formalized a translation between the languages (Sections 4 and 5), and I have implemented each language in Standard ML.

2 PATTERN MATCHING AND EQUATIONS

In this section, I expand on the definitions, forms, and tradeoffs of pattern matching and equations. These tradeoffs inform the compromises I make in V^- down the line in Section 3.4.

2.1 Pattern matching

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

Why use pattern matching? What could programmers use instead? One tool a programmer might use to deconstruct data is *observers* [Liskov and Guttag 1986]: functions that explicitly inquire a piece of data's structure 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`. But many functional programmers favor pattern matching over observers. I demonstrate with an example and a claim.

Consider a *shape* datatype in Standard ML, which represents shapes by their dimensions:

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

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

```
area : 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
*)
```

Now consider two implementations of `area`, one with observers and one with pattern matching, in Figure 1.

```
fun 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 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.

If that prospect doesn't convince you that pattern matching avoids a lot of the problems that observers have, I'll show five general reasons why programmers prefer pattern matching over observes. I refer to these as Nice Properties for the rest of the paper. 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.

A. 1 With pattern matching, code more closely resembles algebraic laws.
2 With pattern matching, it's easier to avoid duplicating code.
3 With pattern matching, a compiler may be able to tell if the code omits an important case through *exhaustiveness analysis*.

B.
4 Pattern matching plays nicer with constructed data.
5 With pattern matching, important intermediate values are always given a name.

Nice Properties 1 and 2 are the most important of these: they allow programmers to write code that looks like what they write at the whiteboard, with flexible laws and minimal duplication. I show in Section 2.2 that extensions to pattern matching largely exist to uphold these two properties.

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

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

B.
4 Where did `isSquare`, `sqSide`, and all the other observers come from? To even *implement* 1a, a programmer has to define a whole new set of observers for `shapes`![1] Most programmers find this tiresome indeed. 1b did not have to do any of this.
5 To extract the internal values, 1b had to name them, and their names serve as documentation.

Having had the chance to compare Figures 1a and 1b, if you moderately prefer the latter, that's good: most functional programmers— in fact, most *programmers*— likely do as well.

Figure 1b provides an opportunity to introduce a few terms that are common in pattern matching. 1b is a classic example of where pattern matching most commonly occurs: within a *case* expression. A *case* expression tests a *scrutinee* (`sh` in 1b) against a list of

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

patterns (`SQUARE s`, etc.). When the result of evaluating the scrutinee matches with a pattern, the program evaluates the right-hand side of the respective branch (`s * s`, etc.) within the *case* expression.[2]

2.2 Popular extensions to pattern matching

Extensions to pattern matching simplify cases that are otherwise troublesome. Specifically, extensions help restore Nice Properties 1 and 2 in cases where pattern matching fails to satisfy them.

In this section, I illustrate several such instances of exactly this, and 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.

For the sake of comparison, I coin the term *bare pattern matching* to denote pattern matching *without* extensions: in bare pattern matching, the only syntactic forms of patterns are names and applications of value constructors to zero or more patterns.

2.2.1 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[3].

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

---

[2]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.

[3]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 2.2.2. Technically, Haskell *only* has guards, but they subsume side conditions, so I hand-wavingly say that it does have side conditions.

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

exclaimTall : shape -> string
(*
exclaimTall (Square s)              == "Wow! That's a tall 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
exclaimBigArea sh                   == "Your shape is small.",
                                        otherwise
*)
```

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

```
let exclaimTall sh =
match sh with
  | Square s -> if s > 100.0
                then "Wow! That's a tall 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 tall 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.

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 (and hopefully you, the reader) am not thrilled with this code. It gets the job done, but it fails to adhere to Nice Properties 1 and 2: 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 `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 minimize overhead, restoring Nice Properties 1 and 2. And a side condition can exploit bindings that emerge from the preceding pattern match. For instance, the `when` clauses in Figure 3 exploit 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 keeping Nice Property 3 becomes an NP-hard problem, because the compiler 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: minimal code duplication and ease of translation from laws to code (1, 2).

A side condition can incorporate an extra "check"- in this case, a boolean expression—within 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 it can't make a decision based off an arbitrary pattern match, and it can't bind names for the programmer to use in the right-hand side. In the next section, I showcase when this limitation matters, and how another extension addresses it.

2.2.2 *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 don't give that flexibility.

Side conditions don't appear to help here, so I'll 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 Nice Property 2: avoiding duplicated code.

Once again, the code works, but it's lost Nice Properties 2 and 1 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 from constructed data.

To restore the Nice Properties, I'll 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 `=`. On the left-hand side of the `<-` is a pattern, and on the right is an expression. At runtime, the program processes a guard by evaluating the expression and testing if it matches with the pattern. If it does, it processes the next guard with any bindings introduced by processing guards before it. If it 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 the program tests the same way it would a side condition, so guards subsume side conditions!

```
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.

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.

The power of pattern guards lies in the ability for 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.

2.2.3 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 a token in a video game and how much fun it is, and 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. Here are the `token` type and the type and algebraic laws for `game_of_token`.

```
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
```

I can write code for game_of_token in OCaml using bare patterns (Figure 6), but I'm dissatisfied with how it fails the first three (1, 2, 4) of the Nice Properties: it has many duplicated right-hand sides, it is visually different from the algebraic laws, and the function, even though it uses pattern matching, doesn't really play nice with my custom type; deconstructing it is tedious and redundant.

I could try to use a couple of helper functions to reduce clutter, and write something like Figure 7. It looks ok, but I'm still hurting for Nice Properties 2 and 4, and the additional calls hurt performance.

Thankfully, an extension once again comes to the rescue. *Or-patterns* condense multiple patterns which share a right-hand side, and when any one of the patterns matches with the scrutinee, the right-hand side is evaluated with the bindings created by that pattern. I

```
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. 6. game_of_token, with redundant right-hand sides, should raise a red flag.

```
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. 7. 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. 8. Or-patterns condense game_of_token significantly, and it is easier to read line-by-line.

exploit or-patterns in Figure 8 to restore the Nice Properties and eliminate much of the uninteresting code that appeared in 6 and 7.

In addition to the inherent appeal of brevity, or-patterns serve to concentrate complexity at a single juncture and create single points of truth. And by eliminating helper functions, like the ones in Figure 7, or-patterns actually boost performance.

2.2.4 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 in unison. When picking a language to do so, I am stuck! Indeed, 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; developers [n. d.]; Klabnik and Nichols 2023; Kokke et al. 2020; Leroy et al. 2023; Marlow et al. 2010; Syme et al. 2010; Team [n. d.]].

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.

2.3 Verse's 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 paper, I use "$\mathcal{VC}$ " and "Verse" interchangeably.)

Equations present a different, yet powerful, way to write code that makes decisions and deconstructs data. In this section, I will introduce you to equations similarly to I how I introduced to you pattern matching. Once you're familiar with equations, you'll be ready to compare their strengths and weaknesses with those of pattern matching, and judge the compromise I propose.

Even if you read Augustsson et al. [2023], $\mathcal{VC}$'s equations and the paradigms of functional logic programming might look unfamiliar. To help ease you into familiarity with equations, I'll ground explanations and examples in how they relate to pattern matching.

$\mathcal{VC}$ uses *equations* instead of pattern matching to 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.

```
∃ area. 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}
```

Fig. 9. `area` in $\mathcal{VC}$ uses existentials and equations rather than pattern matching.

Every equation in Verse takes the form $x = e$, where x is a name and e is an expression. During runtime, $\mathcal{VC}$ relies on a process called *unification* 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).

Equations offer 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 you can get away with knowing that 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 Verse. I've written our `area` function in $\mathcal{VC}$ extended with a `float` type and a multiplication operator `*` in Figure 9.

Like in the pattern-matching example in 1b, the right-hand sides of `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 observers. This claim mirrors my argument for pattern matching, and to support it I appeal to the Nice Properties:

(1) `area` using equations looks like the algebraic laws, with the addition of the explicit ∃. It relies more on mathematical notation, but that might not be a bad thing:

though it doesn't resemble the algebraic laws a programmer would write, it likely resembles the equations that a mathematician would.

(2) `area` using equations does not duplicate virtually any code.

(3) `area` using equations deconstructs user-defined types as easily as 1b does with pattern matching.

(4) `area` using equations has all important internal values named very explicitly.

(5) This Property may not hold, because $\mathcal{VC}$ on its own is untyped. Without a type system, a compiler cannot help me with non-exhaustive cases. However, there is no published compiler, type system or no, for Verse, and only when one is made available can I make this assertion. For this reason, and for the sake of focusing on more important details, I choose to proceed in my analysis of equations in $\mathcal{VC}$ without considering this Property.

If you still believe these Properties to be desirable, you understand why I claim programmers prefer equations to observers. Now I'll 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 2.1, I demonstrated how pattern matching had to resort to extensions to regain the Properties when challenging examples stole them away. In Figure 10, I've implemented those examples in $\mathcal{VC}$ (this time extended with strings, floats, and $\ast$) using choice and equations. Take a look for yourself!

The code in Figure 10 has all the Nice Properties (again, disregarding the ambiguous 5th.) 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 a strong option for writing code!

2.4 $\mathcal{VC}$ has a challenging cost model

So what's the catch? Programmers everywhere have not thrown up their hands, renounced pattern matching, and adopted a puritanical dogma of equations. Sadly, this is not merely a matter of preference.

$\mathcal{VC}$'s equations appear to be comparably pleasing to pattern matching in their brevity and expressiveness. However, full Verse allows computations that are problematic, costwise. In $\mathcal{VC}$, names (logical variables) are *values*, and can just as easily be the result of

```
∃ exclaimTall. exclaimTall = λ sh.
  one {
      ∃ s. sh = ⟨Square s⟩;
      s > 100.0; "Wow! That's a tall 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. 10. Code for the 2.1 functions with equations looks similar, and it doesn't need extensions.

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 at runtime classically requires backtracking, which can lead to exponential runtime cost [Clark 1982; Hanus 2013; Wadler 1985].

Pattern matching, by contrast, has a desirable cost model. Maranget [2008] showed that pattern matching 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. A decision tree, however, forbids backtracking by nature: once the program makes a decision based on the form of a value, it can't re-test it later with new information.

3 A COMPROMISE

To bridge the gap between pattern matching, equations, and decision trees, I have created and implemented a semantics for three core languages. To model pattern matching with extensions, I introduce P^+. To model programming with equations, I introduce V^-. And to provide an efficient cost model to which both P^+ and V^- can be compiled, I introduce D.

Of these three languages, the most unusual is V^-. It has equations and choice, like $\mathcal{VC}$, but without multiple results or backtracking. To eliminate multiple results, expressions in V^- evaluate to at most one result, and choice is not a valid form of expression in the language. To eliminate backtracking, the compiler rejects a V^- program that would need to backtrack at runtime.

In this section, I present the three languages. Their semantics appear in Sections 3.1, 3.7, and 4.2, respectively, and in Appendix E. In my design, I took inspiration from Verse: each of P^+, V^-, and D 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.

Because they share a core, and to facilitate comparisons and proofs, I present V^-, P^+, and D as three subsets of a single unifying language U, whose abstract syntax is given in Table 4. Column "Unique To" indicates which components of U belong to the sub-languages. For all intents and purposes, the three languages are distinct; it is because they all share the same core of lambdas, value constructors (K), names, and function application that I have decided to house them in U together.

Like in $\mathcal{VC}$, every lambda-calculus term is valid in our languages and has the same semantics. Also like the lambda calculus and $\mathcal{VC}$, all three languages are *strict*, meaning every expression is evaluated when it is bound to a variable, and they are all untyped. Creating a type system for P^+ and V^- is a worthy effort but is the subject of another paper. Typing for low-level languages like D is outside the scope of this thesis.

That the only form of constructed data in these languages is value-constructor application is an endeavor for simplicity. 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 in these core languages.

Using just value constructors, though, a programmer can simulate more primitive data like strings. For example, `Wow! That's A Tall Square` is a valid expression in any of the languages, 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 by the programs because their names begin with a capital letter. A programmer can use a similar trick to simulate integers: `One`, `Two`, etc. Because the languages all also have lambda, Church Numerals [Church 1985] are another viable[4] option.

In the subsections below, I discuss each language in more detail. Section 8 goes further in analyzing how P^+ and V^- relate to modern languages with pattern matching and to $\mathcal{VC}$ respectively.

3.1 Introducing P^+

P^+ offers a standardized core for pattern matching, enhanced by common extensions. In addition to bare pattern matching— names and applications of value constructors— the language includes pattern guards, or-patterns, and side conditions. Although pattern guards subsume side conditions, I include side conditions in P^+ and separate them from guards for purpose of study. Furthermore, P^+ introduces a new form of pattern: p_1, p_2. This allows a pattern in P^+ be a *sequence* of sub-patterns, allowing a programmer to stuff as many patterns as they want in the space of a single one. I discuss the implications of this in Section C.

3.2 Formal Semantics of P^+

In this section, I present a big-step operational semantics for P^+. The semantics describes how expressions in P^+ are evaluated and how pattern matching works in the language. Instead of a rewrite semantics that desugars extensions into *case* expressions, P^+ has a big-step semantics that directly describes how they are handled by the runtime core. Figure 11 contains the concrete syntax of P^+, Figure 1 provides the metavariables used in the judgement forms and rules of the semantics, and Section 3.3 contains the forms and rules. Since pattern matching is the heart of P^+, I also describe it in plain English.

3.2.1 Expressions in P^+. An expression in P^+ evaluates to produce a single value, shown by the Eval judgement form.

(Eval) $\langle \rho, e \rangle \Downarrow v$ Expression e evaluates in environment ρ to produce value v.

[4]Not to mention meditative.

Programs	P	::=	$\{d\}$	definition
Definitions	d	::=	`val` x `=` e	bind name to expression
Expressions	e	::=	v	literal values
		|	x, y, z	names
		|	$K\{e\}$	value constructor application
		|	`\`x`.` e	lambda declaration
		|	$e_1\ e_2$	function application
		|	`case` e $\{p$ `->` $e\}$	*case* expression
		|	`(`e`)`	
Patterns	p	::=	$p_1 \mid p_2$	or-pattern
		|	p `,` p'	pattern guard
		|	p `<-` e	pattern from explicit expression
		|	x	name
		|	`_`	wildcard
		|	$K\ \{p\}$	value constructor application
		|	`when` e	
		|	`(`p`)`	
Values	v	::=	$K\{v\}$	value constructor application
		|	`\`x`.` e	lambda value
Value Constructors	K	::=	`true` | `false`	booleans
		|	`A-Z`x	name beginning with capital letter

Fig. 11. P^+: Concrete syntax

3.2.2 *Pattern matching in P^+.*

Pattern matching in P^+ is represented by a single judgement form, with two possible outcomes: success (a refined environment ρ') and failure (†). The metavariable s, a *solution* to a pattern match, combines these outcomes.

$\langle \rho, p, v \rangle \rightarrowtail \rho'$ Pattern p matches value v in environment ρ, producing bindings ρ';

(Match-Success)

$\langle \rho, p, v \rangle \rightarrowtail \dagger$ Pattern p does not match value v in environment ρ.

(Match-Fail)

Pattern guards introduce a special case: if a pattern is bound to an expression in the form $\langle \rho, p \leftarrow e, v \rangle \rightarrowtail s$, it will match if the expression e evaluates to value v' and p matches

P^+ Metavariables	
e	expression
$v,\ v'$	value
K	value constructor
p	pattern
x, y	names
$\dagger$	pattern match failure
s	a solution, either v or $\dagger$
ρ	environment: $name \to \mathcal{V}$
$\rho + \rho'$	extension
$\rho \uplus \rho'$	disjoint union
$\{x \mapsto y\}$	environment with name x mapping to y

Table 1. P^+ metavariables and their meanings

with v'. When a pattern is standalone, as in all other cases, it will match on v, the *original* scrutinee of the case expression. For example, in the P^+ expression `case x of Square s, Big b <- mumble s -> b`, the result of evaluating `x` matches with `Square s`, and the result of evaluating `mumble s` matches with `Big b`. Generally: when $\langle \rho, p_1, p_2 \leftarrow e, v\rangle \rightarrowtail s$, $\langle \rho, p_1, v\rangle \rightarrowtail s_1$, $\langle \rho, e\rangle \Downarrow v'$, and $\langle \rho, p_2, v'\rangle \rightarrowtail s_2$.

Pattern matching is defined inductively:

- A name x matches any value v, and produces the environment $\{x \mapsto v\}$.
- A value constructor K applied to atoms matches a value v if v is an application of K to the same number of values, each of which matches the corresponding atom. Its match produces the disjoint union of matching all internal atoms to all internal values.
- A pattern *when* e matches when e evaluates to a value other than *false*, and produces the empty environment.
- A wildcard pattern _ matches any value v, and produces the empty environment.
- A pattern $p \leftarrow e$ matches when e evaluates to value v, and p matches v.
- A pattern p_1, p_2 matches if both p and p_2 match.
- A pattern $p_1 \mid p_2$ matches if either p_1 or p_2 matches.

When a pattern is of the form $K, p_1, \ldots, p_n$, each sub-pattern p_i may introduce new variables during pattern matching. Bindings for all these variables must be combined in a single environment. *Disjoint union* is an operation on two environments. Its definition borrowed, paraphrased, from Ramsey [2022]:

Disjoint union is a way to capture the aggregate environment of matching constructed data with a constructor-application pattern. The disjoint union of environments ρ_1 and ρ_2 is written $\rho_1 \uplus \rho_{12}$, and it is defined if and only if $\operatorname{dom} \rho_1 \cap \operatorname{dom} \rho_2 = \emptyset$:

$$\operatorname{dom}(\rho_1 \uplus \rho_2) = \operatorname{dom} \rho_1 \uplus \operatorname{dom} \rho_2,$$

$$(\rho_1 \uplus \rho_2)(x) = \begin{cases} \rho_1(x), & \text{if } x \in \operatorname{dom} \rho_1 \\ \rho_2(x), & \text{if } x \in \operatorname{dom} \rho_2 \end{cases}$$

For example, in the P^+ expression `case Pair 1 (Pair 2 3) of Pair x (Pair y z) -> z`, the right-hand size `z` is evaluated with the environment $\{x \mapsto 1\} \uplus \{y \mapsto 2\} \uplus \{z \mapsto 3\}$.

Disjoint union across multiple results, when any result is failure, still represents failure:

$$\dagger \uplus \rho = \dagger \quad \rho \uplus \dagger = \dagger \quad \dagger \uplus \dagger = \dagger$$

At runtime, disjoint union also fails if $\operatorname{dom} \rho_1 \cap \operatorname{dom} \rho_2 \neq \emptyset$, meaning a constructor-application pattern had duplicate names, like in `Pair x x`. This means P^+ has only *linear* patterns under value constructors, i.e., the same name cannot bind multiple components of a single instance of constructed data.

Finally, when a pattern in a branch in a *case* expression fully matches, its corresponding right-hand side is evaluated with top-level ρ extended with the ρ' produced by the pattern match. Environment extension is notated $\rho + \rho'$.

3.3 Rules (Big-step Operational Semantics) for P^+:

Some of these rules are a variation on the rules found in Ramsey [Ramsey 2022].

3.3.1 *Judgement forms for V^-.*

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

$$\langle \rho, p, v \rangle \rightarrowtail \rho' \quad (\textsc{Match-Success})$$

$$\langle \rho, p, v \rangle \rightarrowtail \dagger \quad (\textsc{Match-Fail})$$

3.3.2 *Evaluating General Expressions.*

$$(\textsc{Eval-VconEmpty}) \quad \frac{}{\langle \rho, K \rangle \Downarrow K}$$

$$(\textsc{Eval-VconMulti}) \quad \frac{\langle \rho, e_i \rangle \Downarrow v_i \qquad 1 \le i \le n}{\langle \rho, K(e_1, \ldots e_n) \rangle \Downarrow K(v_1, \ldots v_i)}$$

$$(\textsc{Eval-Name}) \quad \frac{\begin{array}{c} x \in \mathrm{dom}\, \rho \\ \rho(x) = v \end{array}}{\langle \rho, x \rangle \Downarrow v}$$

$$(\textsc{Eval-LambdaDecl}) \quad \frac{}{\langle \rho, \lambda x.e \rangle \Downarrow \lambda x.e}$$

$$(\textsc{Eval-FunApp}) \quad \frac{\begin{array}{c} \langle \rho, e_1 \rangle \Downarrow \lambda x.e \\ \langle \rho, e_2 \rangle \Downarrow v' \\ \langle \rho\{x \mapsto v'\}, e \rangle \Downarrow r \end{array}}{\langle \rho, e_1\ e_2 \rangle \Downarrow r}$$

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

$$(\textsc{Eval-CaseScrutinee}) \quad \frac{\langle \rho, e \rangle \Downarrow v \qquad \langle \rho, \mathit{case}\ v\ [p_1\ e_1], \ldots, [p_n\ e_n] \rangle \Downarrow v'}{\langle \rho, \mathit{case}\ e\ [p_1\ e_1], \ldots, [p_n\ e_n] \rangle \Downarrow v'}$$

$$(\textsc{Eval-CaseMatch})\quad \frac{\langle \rho, p_1, v\rangle \rightarrowtail \rho' \qquad \langle \rho + \rho', e_1\rangle \Downarrow v}{\langle \rho, \mathit{case}\ v\ [p_1\ e_1], \ldots, [p_n\ e_n]\rangle \Downarrow v'}$$

$$(\textsc{Eval-CaseFail})\quad \frac{\langle \rho, p_1, v\rangle \rightarrowtail \dagger \qquad \langle \rho, \mathit{case}\ v\ [p_2\ e_2], \ldots, [p_n\ e_n]\rangle \Downarrow v'}{\langle \rho, \mathit{case}\ v\ [p_1\ e_1], \ldots, [p_n\ e_n]\rangle \Downarrow v'}$$

3.3.3 Rules for pattern matching.

$$(\textsc{Pat-MatchVcon})\quad \frac{\begin{array}{c}\langle \rho, p_i, v_i\rangle \rightarrowtail s_i, \qquad 1 \le i \le m \\ s = s_1 \uplus \cdots \uplus s_m\end{array}}{\langle \rho, K[p_1 \ldots p_m], K[v'_1 \ldots v'_m]\rangle \rightarrowtail s}$$

$$(\textsc{Pat-FailVcon})\quad \frac{v \text{ does not have the form } K[v'_1, \ldots v'_m]}{\langle \rho, K\ p_1, \ldots p_m, v\rangle \rightarrowtail \dagger}$$

$$(\textsc{Pat-MatchBareVcon})\quad \frac{}{\langle \rho, K, K\rangle \rightarrowtail \{\}}$$

$$(\textsc{Pat-FailBareVcon})\quad \frac{v \ne K}{\langle \rho, K, v\rangle \rightarrowtail \dagger}$$

$$(\textsc{Pat-MatchVar})\quad \frac{}{\langle \rho, x, v\rangle \rightarrowtail \{x \mapsto v\}}$$

$$(\textsc{Pat-MatchWhen})\quad \frac{\langle \rho, e\rangle \Downarrow v' \qquad v' \ne \mathit{false}}{\langle \rho, \mathit{when}\ e, v\rangle \rightarrowtail \{\}}$$

$$(\textsc{Pat-MatchWildcard})\quad \frac{}{\langle \rho_, v\rangle \rightarrowtail \{\}}$$

$$\text{(Pat-FailWhen)}\ \frac{\langle \rho, e\rangle \Downarrow v' \qquad v' = \mathit{false}}{\langle \rho, \mathit{when}\ e, v\rangle \rightarrowtail \dagger}$$

$$\text{(Pat-ArrowExp)}\ \frac{\langle \rho, e\rangle \Downarrow v' \qquad \langle \rho, p, v'\rangle \rightarrowtail s}{\langle \rho, p \leftarrow e, v\rangle \rightarrowtail s}$$

$$\text{(Pat-MultiFail)}\ \frac{\langle \rho, p, v\rangle \rightarrowtail \dagger}{\langle \rho, p_1, p_2, v\rangle \rightarrowtail \dagger}$$

$$\text{(Pat-MultiSolution)}\ \frac{\begin{array}{c}\langle \rho, p_1, v\rangle \rightarrowtail \rho' \\ \langle \rho \uplus \rho', p_2, v\rangle \rightarrowtail s\end{array}}{\langle \rho, p_1, p_2, v\rangle \rightarrowtail s}$$

$$\text{(Pat-OrFst)}\ \frac{\langle \rho, p_1, v\rangle \rightarrowtail \rho'}{\langle \rho, p_1 \mid p_2, v\rangle \rightarrowtail \rho'}$$

$$\text{(Pat-OrSnd)}\ \frac{\langle \rho, p_1, v\rangle \rightarrowtail \dagger \qquad \langle \rho, p_2, v\rangle \rightarrowtail s}{\langle \rho, p_1 \mid p_2, v\rangle \rightarrowtail s}$$

I show how a programmer might utilize P^+ to solve the previous problems (Section 2.2) in Figure 12. The examples in the figure all compile with the `pplus` program.

As mentioned in Section 3, masquerading value constructors stand in for strings. P^+ has no infix operators, so some expressions are parenthesized.

3.4 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

```
val exclaimTall = \sh.
  case sh of Square s, when (> s) 100 ->
            Wow! That's A Tall Square!
    | Triangle w h, when (> s) 100 ->
            Goodness! Towering Triangle!
    | Trapezoid b1 b2 h, when (> s) 100 ->
            Zounds! Tremendous Trapezoid!
    | _ -> Your Shape Is Small
```

(a) `exclaimTall` in P^+

```
val tripleLookup = \ rho. \x.
  case x of
    Some w <- (lookup rho) x
  , Some y <- (lookup rho) w
  , Some z <- (lookup rho) y -> z
  | _ -> handleFailure x
```

(b) `tripleLookup` in P^+

```
val game_of_token = \t.
  case t of
    BattlePass f   | (ChugJug f | TomatoTown f) -> P (Fortnite f)
  | HuntersMark f | SawCleaver f                -> P (Bloodborne ((* 2) f))
  | MoghLordOfBlood f | PreatorRykard f         -> P (EldenRing ((* 3) f))
  | _                                           -> P (Irrelevant 0)
```

(c) `game_of_token` in P^+

Fig. 12. The functions in P^+ have the same desirable implementation as before. All the example compile with the `pplus` program.

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^-.

V^-'s concrete syntax is defined in Figure 13. V^- takes several key concepts from $\mathcal{VC}$, with several key differences, illustrated in Table 2.

Like $\mathcal{VC}$	**Unlike $\mathcal{VC}$**
V^- uses equations to guard computation.	V^- solves an equation at most once at run-time and never backtracks.
V^- uses choice.	V^-'s choice can only guard computation and its result is never returned.
V^- uses *success* and *failure* to make decisions.	An expression in V^- returns at most one value.

Table 2. Key similarities and differences between V^- and $\mathcal{VC}$

Lemma 3.1. *An expression in V^- can return at most one result.*

Proof. The syntax of V^- (Figure 13) forbids any expression to return the result of evaluating choice. Let e_v be a V^- expression. If choice appears in e_v, it may only appear in a guard. Guards may only precede expressions and are never returned. Therefore, e_v may never return the result of evaluating choice, and there is no other syntactic form in V^- that allows for multiple results. □

Lemma 3.2. *An expression in V^- can never backtrack at runtime.*

Proof. In progress. □

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
		|	`\`x`.` e	lambda declaration
Guarded Expressions	G	::=	`[E {`x`}.]{`g`} ->` e	names, guards, and body
Guards	g	::=	e	intermediate expression
		|	x `=` e	equation
		|	g`{;`g`}` **|** g`{;`g`}`	choice
Values	v	::=	$K\{v\}$	value constructor application
		|	`\`x`.` e	lambda value

Fig. 13. V^-: Concrete syntax

3.5 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 3.7).

Figure 14 provides an example of how a programmer can utilize V^- to solve the previous problems (Section 2.2):

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

(a) `exclaimTall` in V^-

```
val tripleLookup = \ rho. \x.
  if E x 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. 14. The functions in V^- have a desirably concise implementation, as before.

V^- looks satisfyingly similar to both P^+ and $\mathcal{VC}$. The V^- examples in Figure 14 have the same number of cases as the P^+ examples in Figure 12, and share the existential and equations with the $\mathcal{VC}$ examples in Figure 10.

3.6 Formal Semantics of V^-:

In this section, I present a big-step operational semantics for P^+. The semantics describes how expressions in P^+ are evaluated and how pattern matching works in the language.

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

Table 3. V^- metavariables and their meanings

Instead of a rewrite semantics that desugars extensions into *case* expressions, P^+ has a big-step semantics that directly describes how they are handled by the runtime core. Figure 11 contains the concrete syntax of P^+, Figure 1 provides the metavariables used in the judgement forms and rules of the semantics, and Section 3.3 contains the forms and rules. Since pattern matching is the heart of P^+, I also describe it in plain English.

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

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

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

3.6.2 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 produces **fail**.

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 our semantics, this process occurs over a *list* of guards in a context $\mathcal{T}$: the program picks a guard from $\mathcal{T}$, attempts to solve it to refine the environment or fail, and repeats. V^- can only pick a guard out of the context $\mathcal{T}$ that it "knows" it can solve. "Knowing" a guard can be solved can be determined in V^- at compile time. If V^- can't pick a guard and there are guards left over, the semantics gets stuck at compile time.

$$\rho; \mathcal{T} \vdash gs \rightarrowtail s \quad (\textsc{Solve-Guards})$$

The environment ρ maps from a name to a value vs 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 will either refine ρ (ρ') or be **rejected**. We use the metavariable † to represent rejection, and a guard producing † will cause the top-level list of guards to also produce †.

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

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

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

3.6.3 Refinement ordering on environments.

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

Success only refines the environment; that is, when $\langle \rho, e \rangle \rightarrowtail \rho'$, we expect $\rho \subseteq \rho'$.

3.7 Rules (Big-step Operational Semantics) for V^-:

3.7.1 *Judgement forms for V^-.*

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

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

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

3.7.2 *Shifting a guard to the context.*

$$(\textsc{Move-Guard-To-Ctx}) \; \frac{\rho; g \cdot \mathcal{T} \vdash gs \cdot gs' \rightarrowtail s}{\rho; \mathcal{T} \vdash gs \cdot g \cdot gs' \rightarrowtail s}$$

3.7.3 *Choosing and processing a guard.*

$$(\textsc{Solve-Guard-Refine}) \; \frac{\rho \vdash g \rightarrowtail \rho' \qquad \rho';\ \mathcal{T} \cdot \mathcal{T}' \vdash gs \rightarrowtail s}{\rho; \mathcal{T} \cdot g \cdot \mathcal{T}' \vdash gs \rightarrowtail s}$$

$$(\textsc{Solve-Guard-Reject}) \; \frac{\rho \vdash g \rightarrowtail \dagger}{\rho; \mathcal{T} \cdot g \cdot \mathcal{T}' \vdash gs \rightarrowtail \dagger}$$

3.7.4 *Properties of guards.*

$$(\textsc{Multi-Guard-Assoc}) \; \frac{\rho; \mathcal{T} \cdot g_1 \cdot g_2 \cdot \mathcal{T}' \vdash gs \rightarrowtail s}{\rho; \mathcal{T} \cdot g_2 \cdot g_1 \cdot \mathcal{T}' \vdash gs \rightarrowtail s}$$

3.7.5 *Refinement with different types of guards.*

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

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

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

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

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

$$(\textsc{Guard-VCon-Fail})\ \frac{\begin{array}{c} x \in \operatorname{dom} \rho \\ \rho x = v \\ v \text{ does not have the form } K[v'_1, \ldots v'_m] \end{array}}{\rho \vdash x = K\ e_1, \ldots e_m \rightarrowtail \dagger}$$

$$(\textsc{Guard-ExpSeq-Succ})\ \frac{\langle \rho, e \rangle \Downarrow v}{\rho \vdash e \rightarrowtail \{\}}$$

$$(\textsc{Guard-ExpSeq-Fail})\ \frac{\langle \rho, e \rangle \Downarrow \textbf{fail}}{\rho \vdash e \rightarrowtail \dagger}$$

$$(\textsc{Guard-Choice-First})\ \frac{\rho; \varepsilon \vdash gs \rightarrowtail \rho'}{\rho \vdash gs \mathbin{|} gs' \rightarrowtail \rho'}$$

$$(\textsc{Guard-Choice-Second})\ \frac{\rho; \varepsilon \vdash gs \rightarrowtail \dagger \qquad \rho; \varepsilon \vdash gs' \rightarrowtail s}{\rho \vdash gs \mathbin{|} gs' \rightarrowtail s}$$

3.7.6 *Evaluating General Expressions.*

$$(\textsc{Eval-IfFi-Fail})\ \frac{}{\langle \rho, \textsc{if}\ [\quad]\ \textsc{fi} \rangle \Downarrow \mathbf{fail}}$$

$$(\textsc{Eval-IfFi-Success})\ \frac{\begin{array}{c}\rho' = \rho\{x_1 \mapsto \bot \ldots x_n \mapsto \bot\} \\ \rho'; \varepsilon \vdash gs \rightarrowtail \rho'' \\ \langle \rho'', e \rangle \Downarrow r\end{array}}{\langle \rho, \textsc{if}\ [\ \exists x_1 \ldots x_n.\ gs \longrightarrow e \ \square\ \ldots\ ]\ \textsc{fi} \rangle \Downarrow r}$$

$$(\textsc{Eval-IfFi-Reject})\ \frac{\begin{array}{c}\rho' = \rho\{x_1 \mapsto \bot \ldots x_n \mapsto \bot\} \\ \rho'; \varepsilon \vdash gs \rightarrowtail \dagger \\ \langle \rho, \textsc{if}\ [\ \ldots\ ]\ \textsc{fi} \rangle \Downarrow r\end{array}}{\langle \rho, \textsc{if}\ [\ \exists x_1 \ldots x_n.\ gs \longrightarrow e \ \square\ \ldots\ ]\ \textsc{fi} \rangle \Downarrow r}$$

$$(\textsc{Eval-VconEmpty})\ \frac{}{\langle \rho, K \rangle \Downarrow K}$$

$$(\textsc{Eval-VconMulti})\ \frac{\langle \rho, e_i \rangle \Downarrow v_i \qquad 1 \leq i \leq n}{\langle \rho, K(e_1, \ldots e_n) \rangle \Downarrow K(v_1, \ldots v_i)}$$

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

$$\text{(EVAL-NAME)}\ \frac{x \in \operatorname{dom} \rho \quad \rho(x) = v}{\langle \rho, x \rangle \Downarrow v}$$

$$\text{(EVAL-NAME-FAIL)}\ \frac{x \notin \operatorname{dom} \rho}{\langle \rho, x \rangle \Downarrow \textbf{fail}}$$

$$\text{(EVAL-LAMBDADECL)}\ \frac{}{\langle \rho, \lambda x.e \rangle \Downarrow \lambda x.e}$$

$$\text{(EVAL-FUNAPP)}\ \frac{\langle \rho, e_1 \rangle \Downarrow \lambda x.e \quad \langle \rho, e_2 \rangle \Downarrow v' \quad \langle \rho\{x \mapsto v'\}, e \rangle \Downarrow r}{\langle \rho, e_1\ e_2 \rangle \Downarrow r}$$

$$\text{(EVAL-LITERAL)}\ \frac{}{\langle \rho, v \rangle \Downarrow v}$$

3.8 Introducing D

While P^+ and V^- exist for writing programs, D exists as the target of translation.

This is because the decision-making construct in D, the *decision tree*, has a deterministic and efficient cost model. Specifically, a decision tree is an automaton that implements pattern matching with the key property that no part of the scrutinee of a *case* expression is examined more than once at runtime. This means that the worst-case cost of evaluating a decision tree is linear in its depth. 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.

Decision trees are classically used as an intermediate representation for compiling *case* expressions. In this work, I will use them as a target for compiling *if-fi* in V^-. I do so to prove that equations, at least given their restrictions in V^-, can be compiled to a decision tree.

D is a generalization of the trees found in Maranget [2008]. I discuss the specifics of D's decision trees in the following section.

4 V^- CAN BE COMPILED TO A DECISION TREE

To demonstrate that V^- has the same desirable cost model as 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 a word of the *scrutinee*— the value being tested— more than once. It is the compilation from V^- to D that establishes this property by ensuring that no *test* node T has any proper ancestor T' such that T and T' both test the same location in memory.

This compilation algorithm serves to demonstrate that V^- is a viable alternative to pattern matching on the grounds that they have equivalent cost models: pattern matching can be compiled to a decision tree, which MacQueen and Baudinet [1985] built the foundation for and Maranget [2008] expanded on.

The algorithm runs during $\mathcal{D}$, the transformation from V^- to D. Its domain, instead of a *case* expression, is V^-'s *if-fi*.

4.1 D is a generalization of Maranget's trees

D's concrete syntax is given in Figure 17. Decision trees in D are engineered to look like Maranget's trees. 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. Maragnet 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.

The underlying structures of Maranget's and D's decision trees, however, are analogous. The same operation is the heart of their evaluation: they take a value, examine it, and choose a branch based on its form (Maranget calls the operation Switch; we call it *test*).

Let's look at an example from *Compiling Pattern Matching to Good Decision Trees* which shows the structure of a simple pattern-matching function and its corresponding decision tree.

Maranget beings with the function, `merge`, which merges two lists:

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

Fig. 15. The skeleton of Maranget's `merge`

He compiles the function to this decision tree:

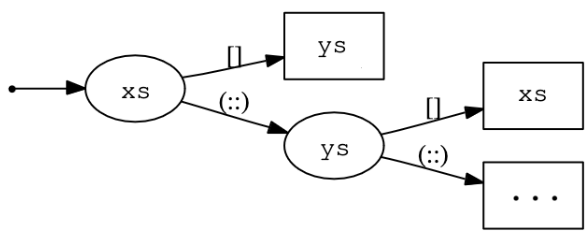

Fig. 16. The final compiled decision tree for `merge`, right-to-left

When presented with the values `xs` and `ys`, the tree first tests `xs` against its two known possible forms: the nullary list constructor `[]`, and the 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 the same list of possible constructors, and returns a value according to the result of the match. Each time the tree goes down a `::` branch, it can extract 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, an *extract* node extract all from a value constructor at once for use in subtrees. The compiler is responsible for introducing the fresh names used in *extract*.

In D, like in V^{-}, expressions can *fail*, meaning some of D's syntactic forms like *try-let* and *cmp* need an additional

Programs	P	::=	$\{d\}$	definition
Definitions	d	::=	`val` x = e	bind name to expression
Expressions	e	::=	v	literal values
		|	x, y, z	names
		|	$K\{e\}$	value constructor application
		|	`\`x. e	lambda declaration
		|	$e_1\ e_2$	function application
		|	t	decision tree
		|	(e)	
Decision Trees	t	::=	`test` x {K/i} -> t [x ->t]	test node
		|	`try-let` x = e `in` t[; `otherwise` t]	try-let node
		|	`cmp` x = e -> t, t, t	comparison node
		|	`extract` x $\{y\}$ e `in` t	extract node
		|	`fail`	fail node
Values	v	::=	$K\{v\}$	value constructor application
		|	`\`x. e	lambda value
Value Constructors	K	::=	`true` | `false`	booleans
		|	`A-Z`x	name beginning with capital letter

Fig. 17. D: Concrete syntax

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

rab: The semantics for D are being revised.

4.3 The $\mathcal{D}$ algorithm: $V^- \to D$

When presented with an *if-fi*, $\mathcal{D}$ calls the function *compile*, whole algorithm is presented in Figure 18. $\mathcal{D}$ first introduces all the names under the all existential $\exists$'s to a context which determines if a name is *known* or *unknown*. At the start, each name introduced by $\exists$ is *unknown*. Since all names are unique at this stage, there are no clashes. $\mathcal{D}$ also desugars choice to multiple *if-fi* branches with a desugaring function $\mathcal{I}$:

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

$$==$$

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

With the new context and desugared *if-fi*, *compile* then repeatedly chooses a guarded expression G and attempts the following while translating G's internal expressions with $\mathcal{D}$:

(1) If there are no guards in G, insert a *match* node with the right-hand side of G.
(2) Otherwise, choose an equation in G of the form $x = K \ldots$ s.t. x is *known*.
(3) If one is found, use it to generate a *test* node, building each subtree of the *test* by pruning all branches in *all* guarded expressions of the *if-fi* in which $x = e$ and $e \neq K \ldots$ and invoking *compile* on the remaining ones with a context where x is known. Each of these remainders is the child of an *extract* node which extracts the internals of each possible value constructor into a list of names which are introduced to the context. The algorithm builds the default tree of the *test* by finding any branches of the in the *if-fi* in which x is not bound to a value constructor.
(4) If no equation of the form $x = K \ldots$ is found, try to find an equation $x = e$ s.t. either x is unknown and all names in e are known.
(5) If one is found, use it to generate a *try-let* node with two children: one if binding succeeds, and one if e fails. *compile*prunes each subtree of the *try-let* accordingly.
(6) If no equation is found, try to find an equation $x = y$ s.t. x is known and y is unknown.
(7) If one is found, use it to generate a *try-let* node with one child for when binding succeeds. *compile*prunes the subtree of all instance of $x = y$.
(8) If no such equation is found, try to find a condition e s.t. all names in e are known.
(9) If one is found, generate a fresh name x' and use it to generate a *try-let* node for the equation $x = e'$, pruning each subtree of e with a substitution of $[x/e]$.
(10) If no condition e is found, try to find an equation $x = e$ s.t. both x and all names in e are *known*.
(11) If one is found, generate a *cmp* node, prune the *if-fi* of all duplicate instances of that $x = e$, and invoke *compile*again.
(12) If none is found, the *if-fi* cannot be compiled to a decision tree. The algorithm halts with an error.

The algorithm terminates when inserts a final *match* node 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 $\mathcal{D}$ is 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.

In the figure, I use the notation $x@b$ to denote the bag of all guards and expressions in branch b in which name x appears. A branch is a list of guards followed by a terminal expression; it is a branch of an *if-fi* stripped of the existential since $\mathcal{D}$ introduces all names at the top level. I use the notation $\mathrm{dom}(b)$ to describe the set of all names that appear in a branch. I use the notation $branches - -g$ to mean "*branches* pruned of guard g" and the notation $branches - - - g$ to mean "*branches* pruned of all *branches* containing g." I use the standard substitution notation $branches[x/e]$ to mean "*branches* with name x substituted for expression e." I use a shorthand $(context + \{n_1 \ldots n_i \mapsto known\})$ to mean "*context* extended with each of n_x bound to *known.*

Not show in the algorithm is the case where *compile* cannot choose a g of one of the valid forms; in this case, *compile* 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 lambda.

Require: $\forall(n : \text{name}) \in \mathit{branches},\ n$ unique

compile context branches =
if *null(branches)* **then** *fail*
else if $\neg\exists g \in$ *fst(hd(branches))* **then**
  match (hd branches)
else
  let g be a guard such that $g \in \mathit{branches}$ **in**
  if g has the form $x = K(e_1 \dots e_i)$ **then let**
    $KS = \{K \mid \exists\ b \in \mathit{branches} : x \in \mathrm{dom}(b) \wedge K(\dots) \in x@b\}$
    $\mathit{edges} = \mathit{map}($
    $\lambda K.$ (**let** $\mathit{ns} = n_1 \dots n_i,\ n_x$ fresh **in**
    $(K, \mathit{extract}(x, \mathit{ns}, \mathit{compile}(\mathit{context} + \{n_1 \dots n_i \mapsto \mathit{known}\})$
    $(\mathit{mapPartial}(\mathit{refine}\ x\ (K(\mathit{ns})))\mathit{branches}))$
    end)
    $\mathit{defaults} = \mathit{filter}(\lambda b.x \notin \mathrm{dom}(b) \vee K(\dots) \notin x@b)$
    in $\mathit{test}(x, \mathit{edges}, \mathit{SOME}(\mathit{compile}\ \mathit{defaults}))$
    end
  else if g has the form $x = e : x\ \mathit{unknown},\ e\ \mathit{known}$ **then**
    $\mathit{try\_let}(x, \mathcal{D}\ (\mathit{context}\ e),$
    $\mathit{compile}(\mathit{context}\{x \mapsto \mathit{known}\})((\mathit{branches} - -\mathit{eq})[x/e]),$
    $\mathit{SOME}(\mathit{compile}\ \mathit{context}\ (\mathit{branches} - - -\ \mathit{eq} - - -\ e)))$
  else if g has the form $x = y : x\ \mathit{known},\ y\ \mathit{unknown}$ **then**
    $\mathit{try\_let}(x, y, \mathit{compile}(\mathit{context}\{y \mapsto \mathit{known}\})(\mathit{branches} - -\mathit{eq}), \mathit{NONE})$
  else if g has the form $x = e : x\ \mathit{known},\ e\ \mathit{known}$ **then**
    $\mathit{cmp}(x, \mathcal{D}\ (\mathit{context}\ e),$
    $\mathit{compile}\ \mathit{context}\ ((\mathit{branches} - -\mathit{eq})[x/e]),$
    $\mathit{compile}\ \mathit{context}\ (\mathit{branches} - - -\ \mathit{eq} - -e),$
    $\mathit{SOME}(\mathit{compile}\ \mathit{context}\ (\mathit{branches} - - -\ \mathit{eq} - - -\ e)))$
  else if g has the form $e : e\ \mathit{known}$ **then**
    $\mathit{try\_let}(x, \mathcal{D}\ (\mathit{context}\ e),$
    $\mathit{compile}(\mathit{context}\{x \mapsto \mathit{known}\})((\mathit{branches} - -\mathit{eq})[x/e]),$
    $\mathit{SOME}(\mathit{compile}\ \mathit{context}\ (\mathit{branches} - - -\ \mathit{eq} - - -\ e)))$
    $, x$ fresh
    end
where
$\mathit{snd}(a, b) = b$
$\mathit{refine}(x, K(\mathit{ns})b) =$
if $K'(\mathit{es}) \in x@b \wedge (K \neq K' \vee \mathit{length}(\mathit{es}) \neq \mathit{length}(\mathit{ns}))$ **then** NONE
else $\mathit{SOME}\ b$

Fig. 18. The $\mathcal{D}$ algorithm.

4.4 Translation from V^- to D preserves semantics

Translating *if-fi* to a decision tree should preserve semantics. The following theorem formalizes this claim:

Proof. See appendix A. □

5 EQUATIONS SUBSUME PATTERN MATCHING WITH POPULAR EXTENSIONS

In my introduction I stated that V^- can be compiled to a decision tree, and that V^- subsumes pattern matching with popular extensions. Having shown the former, I now show the latter. I do so by presenting an algorithm $\mathcal{E}$ which translates P^+ to V^-.

5.1 Domains

I give the names and domains of the translation functions:

$$\mathcal{E} : P^+Exp \;\rightarrow\; V^-Exp$$

$$\mathcal{P} : Pattern \;\rightarrow\; Name \;\rightarrow\; Name\ list \;*\; Guard\ list$$

The translation functions $\mathcal{E}$ and $\mathcal{P}$ are defined case by case:

5.2 Translating Expressions

$$\mathcal{E}[\![x]\!] \rightsquigarrow x$$

$$\mathcal{E}[\![K\ e_1 \ldots e_n]\!] \rightsquigarrow K\ \mathcal{E}[\![e_1]\!] \ldots \mathcal{E}[\![e_n]\!]$$

$$\mathcal{E}[\![\lambda x.\ e]\!] \rightsquigarrow \lambda x.\ \mathcal{E}[\![e]\!]$$

$$\mathcal{E}[\![e_1\ e_2]\!] \rightsquigarrow \mathcal{E}[\![e_1]\!]\ \mathcal{E}[\![e_2]\!]$$

$$\mathcal{E}[\![\texttt{case}\ e\ p_1\ e_1 | \ldots | p_n\ e_n]\!] \rightsquigarrow$$

$$\forall i.\ 1 \leq i \leq n :$$

$$\texttt{if}\ \exists x.\ x\ =\ \mathcal{E}[\![e]\!];$$

$$\text{let } (ns_1, gs_1) \ldots (ns_i, gs_i) = \mathcal{P}[\![p_1]\!]x\ \cdot\ \ldots \cdot\ \mathcal{P}[\![p_i]\!]x \text{ in}$$

$$\texttt{if}\ \exists\, ns_1.\ gs_1 \rightarrow \mathcal{E}[\![e_1]\!];\ \Box \ldots \Box\ \exists\, ns_i.\ gs_i \rightarrow \mathcal{E}[\![e_i]\!]\ \texttt{fi}$$

$$\texttt{fi}$$

$$,\ x \text{ fresh}$$

5.3 Translating Patterns

$$\mathcal{P}[\![y]\!]x \rightsquigarrow (y, [x = y])$$

$$\mathcal{P}[\![K]\!]x \rightsquigarrow ([], [x = K])$$

$$\mathcal{P}[\![K\ p_1\ \ldots\ p_n]\!]x \rightsquigarrow$$

$$\forall i.\ 1 \leq i \leq n :$$

$$\text{let } y_i \text{ be a fresh name,}$$

$$(ns_1, gs_1) \ldots (ns_i, gs_i) = \mathcal{P}[\![p_1]\!]y_1 \cdot \ldots \cdot \mathcal{P}[\![p_i]\!]y_i$$

$$\text{in}$$

$$(ns_1 \cdot \ldots \cdot ns_i \cdot y_1 \ldots y_i, x = K\ y_1\ \ldots\ y_i \cdot\ gs_1 \cdot \ldots \cdot gs_i)$$

$$\mathcal{P}[\![\mathit{when}\ e]\!]x \rightsquigarrow ([], [\mathcal{E}[\![e]\!]])$$

$$\mathcal{P}[\![p_1, p_2]\!]x \rightsquigarrow \text{let } (ns_1, gs_1) = \mathcal{P}[\![p]\!]x\ , (ns_2, gs_2) = \mathcal{P}[\![p']\!]x \text{ in } (ns_1 \cdot ns_2, gs_1 \cdot gs_2)$$

$$\mathcal{P}[\![p_1 \mid p_2]\!]x \rightsquigarrow \text{let } (ns_1, gs_1) = \mathcal{P}[\![p]\!]x\ , (ns_2, gs_2) = \mathcal{P}[\![p']\!]x \text{ in } (ns_1 \cdot ns_2, [gs_1 \mid gs_2])$$

5.3.1 Significance of the translation. In Section 2.2, I showed how extensions to pattern matching uphold Nice Properties 1 and 2, and how with them, programmers can write more concise code. **The translation aims to show that if a programmer can code with desirable properties in P^+, they can write code with the same properties in V^-.** Proving this claim formally is a goal for future work. Informally, $\mathcal{P}$ does not duplicate code except for introducing new names when translating a constructor-application pattern, and I believe eliminating this redundancy is possible through a desugaring optimization based off of the laws presented in Section C.

$\mathcal{E}$ is largely uninteresting, except for the translation from *case* to *if-fi*.

To compile *case* expressions to decision trees like Maranget does, translate P^+ to D using $(\mathcal{D}\ o\ \mathcal{E})$.

Finally, I claim that the translation from *case* expressions to decision trees, $(\mathcal{D}\ o\ \mathcal{E})$, is consistent with Maranget and others [Maranget 2008; Scott and Ramsey 2000]. Proving this claim is a good goal for future work; it is not the main focus of this paper.

6 IMPLEMENTATIONS

I have full implementations of P^+, V^-, and D at https://github.com/rogerburtonpatel/vml. The languages are complete, from parsers to evaluation to unparsers. In the same repository lives the `dtran` program, which translates from P^+ to V^- and V^- to D. Translating P^+ to D is also possible by composing these two translations. With the implementations, I have been able to gather satisfying empirical evidence of the functionality of the translations, and that they (empirically) preserve semantics.

7 RELATED WORK

The dual foundations of this paper are Augustsson et al.'s Verse Calculus [Augustsson et al. 2023] and Maranget's decision trees [Maranget 2008]. Augustsson et al. give the formal rewrite semantics for the Verse Calculus; Maranget gives an elegant formalism of decision trees. The big-step semantics in this paper is based off of the rewrite semantics; proving their equivalence 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 facilitated writing the code. Maragnet'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 & Peyton Jones [Erwig and Jones 2001]. The authors describe pattern guards and transformational patterns, 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].

Compiling Pattern Matching [Augustsson 1985] by Augustsson gives a foundation in exactly what it says. Ramsey and Scott have a crisp example of a match-compilation algorithm (pattern matching to decision trees) in When Do Match-Compilation Heuristics Matter? [Scott and Ramsey 2000]. Scott and Ramsey's algorithm structurally inspired mine, and I was privy to source code from the algorithm whose study aided my implementation.

8 FUTURE WORK

8.0.1 Desirable formalisms about translations. First and foremost, to flush out the proofs of semantics preservation of $\mathcal{P}$ and $\mathcal{D}$ is my top priority. Making these as airtight as possible will greatly strengthen the argument that V^- is not only a viable alternative to P^+ syntactically; it is also formally equivalent.

8.0.2 Desirable formalisms about V^-. Several formalisms would strengthen the viability of V^-, and are good targets for future work:

(1) Proving V^- is deterministic
(2) Proving the big-step semantics of V^- is consistent with the published semantics of $\mathcal{VC}$.

8.0.3 Exhaustiveness analysis P^+ and V^-. Exhaustiveness analysis can help restore Nice Property 5: with it, the compiler can warn programmers of a missing or extraneous match condition in a *case* expression, and potentially an *if-fi*. Owing to its significantly more flexible structure, however, *if-fi* may prove trickier to analyze for missing match conditions than *case*.

8.0.4 Using V^- to inform programming in Verse. $\mathcal{D}$ and the proof that $\mathcal{D}$ preserves semantics help show that certain computations that use equations for decision-making can be compiled to efficient code. A future project could be to extend V^- to include *all* of $\mathcal{VC}$, and use $\mathcal{D}$to eliminates as much backtracking at runtime as possible, falling back to the VC's fully general evaluation mechanism only when necessary. My hope is that, using these ideas, both the Verse programmer and language designer might make any discovery that allows them to increase the efficiency of full-Verse programs.

9 CONCLUSION

I have introduced the languages P^+ and V^- to demonstrate the viability of equations as an alternative to pattern matching, and D, $\mathcal{E}$, and $\mathcal{D}$ to show how both languages can be compiled to efficient code. I have shown with equivalent examples how programs written in V^- has the same desirable properties as equivalent programs written in P^+, and I have also shown that translating from pattern matching to equations preserves the desirable properties. Finally, I have shown how V^-, like pattern matching, can be compiled to efficient code. 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.

Finally, and in particular with V^-, I hope to provide a stepping stone between pattern matching and equations that a new programmer to Verse will find illuminating.

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 undergraduate 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 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.

11 REFERENCES

REFERENCES

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

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

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

Jonas Barklund and Robert Virding. 1999. Erlang 4.7. 3 Reference Manual DRAFT (0.7). *Ericsson AB* (1999), 79.

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

Alonzo Church. 1985. *The calculi of lambda-conversion*. Number 6. Princeton University Press.

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

Scala developers. [n. d.]. Pattern Matching - The Scala Programming Language. https://docs.scala-lang.org/tour/pattern-matching.html.

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

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

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

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

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

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

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

David MacQueen and M Baudinet. 1985. Tree Pattern matching for ML. *Unpublished manuscript* (1985).

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

Simon Marlow et al. 2010. Haskell 2010 Language Report: Chapter 3. (2010).

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

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

Kevin Scott and Norman Ramsey. 2000. When do match-compilation heuristics matter. *University of Virginia, Charlottesville, VA* (2000).

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

The Elixir Team. [n. d.]. *Elixir Documentation.* Hexdocs.pm. https://hexdocs.pm/elixir/index.html

Philip Wadler. 1985. 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.* Springer, 113–128.

Philip Wadler. 1987. 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.* 307–313.

A PROOFS

- **Proof: Translation from V^- to D preserves semantics**

  Proof. In progress. □

- **Proof: Translation from P^+ to V^- preserves semantics**

  Proof. In progress. □

B DISCUSSION

Both backtracking and multiple results often manifest within $\mathcal{VC}$'s choice operator, which tempt its removal from V^- altogether. However, I am drawn to harnessing the expressive potential of this operator, particularly when paired with $\mathcal{VC}$'s 'one' keyword. When employed with choice as a condition, '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 the $\mathcal{VC}$ runtime: a name in V^- is an *expression* rather than a *value*. This alteration, coupled with my adjustments to choice, eradicates backtracking. Our rationale behind this is straightforward: if an expression returns a name, and subsequently, the program imposes a new constraint on that name, it may necessitate the reevaluation of the earlier expression—- a scenario I strive to avoid.

B.0.1 V^- and P^+, side by side. I compare V^- with P^+ as an exercise in comparing equations with pattern matching. They certainly look similar, which hints that V^- might be as expressive as pattern matching with the three extensions. Proving this claim is the topic of Section 5.

When might a programmer prefer V^- over P^+, or vice versa? After programming in both, I have come up with three empirical observations:

(1) *The scrutinee:* When there is no obvious single scrutinee, V^- is more succinct. When there is a scrutinee, P^+ is more succinct.
(2) *Binding and decision-making:* Binding and decision-making are joined in a single construct in V^-: `=`. P^+ needs different kinds of syntax, like `<-`, to express different kinds of binding. And a programmer wanting for `let` in P^+ will never feel this need in V^-: `=` subsumes that, too.
(3) *Names:* In V^-, names are explicitly introduced. In practice, this helps prevent a common mistake in pattern matching in which a programmer attempts to match a value v on an in-scope name x, expecting the match to succeed iff x evaluates to v at runtime, only to see the match always succeed.

Indeed, having stripped out the functional logic programming elements of $\mathcal{VC}$, 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^-. V^-'s concrete syntax is defined in Figure 13.

C ADDRESSING HOW P^+ HANDLES UNUSUAL PATTERN COMBINATIONS

P^+ admits of strange-looking patterns: consider `Cons (when true) zs`. But these should not be alarming, because such syntactic forms reduce to normal forms by (direct) application of algebraic laws:

$$K(\mathit{when}\ e)\ p2\ \ldots === K\ \_\ p2\ \ldots, \mathit{when}\ e \tag{1}$$

$$K(\mathit{when}\ e,\ p2\ )\ p3\ \ldots === K\ p2\ \ p3\ \ldots, \mathit{when}\ e \tag{2}$$

$$K(p1\ , \mathit{when}\ e)\ p3\ \ldots === K\ p1\ \ p3\ \ldots, \mathit{when}\ e \tag{3}$$

$$K(\mathit{when}\ e \,|\, p2\ )\ p3\ \ldots === (K\ \_\ p3\ \ldots, \mathit{when}\ e) \,|\, (K\ p2\ \ p3\ \ldots) \tag{4}$$

$$K(p1 \,|\, \mathit{when}\ e)\ p3\ \ldots === (K\ p2\ \ p3\ \ldots) \,|\, (K\ \_\ p3\ \ldots, \mathit{when}\ e) \tag{5}$$

$$\mathit{when}\ e \leftarrow e === \ \_ < -e, \mathit{when}\ e \tag{6}$$

Repeatedly applying these laws until the program reaches a fixed point normalizes placements of *when*. Laws (2) and (3) work because P^+ has no side effects and the laws assume all names are unique (the compiler takes care of this), so changing the order in which patterns match has no effect on semantics.

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

V^- certainly appears to relate to $\mathcal{VC}$ semantically. 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 our semantics of V^- are consistent with Augutsson et al.'s $\mathcal{VC}$ is an excellent goal for future work.

E FORMAL DEFINITIONS OF ALL LANGUAGES

E.0.1 Judgement forms for V^-.

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

$$\langle \rho, p, v \rangle \rightarrowtail \rho' \ \ (\textsc{Match-Success})$$

$$\langle \rho, p, v \rangle \rightarrowtail \dagger \ \ (\textsc{Match-Fail})$$

E.0.2 Evaluating General Expressions.

$$(\textsc{Eval-VconEmpty})\ \frac{}{\langle \rho, K \rangle \Downarrow K}$$

Syntactic Forms	**Cases**	**Unique to**
P : Programs	$\{d\}$	
d : Definitions	$val\ x = e$	
e : Expressions	v	
	x, y, z	
	$K\{e\}$	
	$\lambda x.\ e$	
	$e_1\ e_2$	
	$case\ e\ \{p\ \rightarrow\ e\}$	P^+
	$if\ [\ G\ \{[]G\}\ ]\ fi$	V^-
	t	D
v : Values	$K\{v\}$	
	$\lambda x.\ e$	
p : Patterns	x	P^+
	$\_$	P^+
	$K\ \{p\}$	P^+
	$when\ e$	P^+
	p_1, p_2	P^+
	$p \leftarrow e$	P^+
	$p_1 \mid p_2$	P^+
G : Guarded Expressions	$[\exists\ \{x\}.]\{g\} \rightarrow e$	V^-
g : Guards	$x = e$	V^-
	e	V^-
	$g\{;g\} \mid g\{;g\}$	V^-
t : Decision Trees	$test\ x\ \{K/i\}\ \rightarrow\ t[x\ \rightarrow t]$	D
	e	D
$try - let\ x = e\ in\ t[;\ otherwise\ t]$	D	
$cmp\ x = e : t, t, t$	D	
$extract\ x\ \{y\}\ e\ in\ t$	D	
$fail$	D	

Table 4. Abstract Syntax of all languages, with deliniations in column **Unique To**

$$(\textsc{Eval-VconMulti})\quad \frac{\langle \rho, e_i \rangle \Downarrow v_i \qquad 1 \leq i \leq n}{\langle \rho, K(e_1, \ldots e_n) \rangle \Downarrow K(v_1, \ldots v_i)}$$

$$\text{(Eval-Name)}\ \frac{x \in \operatorname{dom} \rho \qquad \rho(x) = v}{\langle \rho, x \rangle \Downarrow v}$$

$$\text{(Eval-LambdaDecl)}\ \frac{}{\langle \rho, \lambda x.e \rangle \Downarrow \lambda x.e}$$

$$\text{(Eval-FunApp)}\ \frac{\langle \rho, e_1 \rangle \Downarrow \lambda x.e \qquad \langle \rho, e_2 \rangle \Downarrow v' \qquad \langle \rho\{x \mapsto v'\}, e \rangle \Downarrow r}{\langle \rho, e_1\ e_2 \rangle \Downarrow r}$$

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

$$\text{(Eval-CaseScrutinee)}\ \frac{\langle \rho, e \rangle \Downarrow v \qquad \langle \rho, \mathit{case}\ v\ [p_1\ e_1], \ldots, [p_n\ e_n] \rangle \Downarrow v'}{\langle \rho, \mathit{case}\ e\ [p_1\ e_1], \ldots, [p_n\ e_n] \rangle \Downarrow v'}$$

$$\text{(Eval-CaseMatch)}\ \frac{\langle \rho, p_1, v \rangle \rightarrowtail \rho' \qquad \langle \rho + \rho', e_1 \rangle \Downarrow v}{\langle \rho, \mathit{case}\ v\ [p_1\ e_1], \ldots, [p_n\ e_n] \rangle \Downarrow v'}$$

$$\text{(Eval-CaseFail)}\ \frac{\langle \rho, p_1, v \rangle \rightarrowtail \dagger \qquad \langle \rho, \mathit{case}\ v\ [p_2\ e_2], \ldots, [p_n\ e_n] \rangle \Downarrow v'}{\langle \rho, \mathit{case}\ v\ [p_1\ e_1], \ldots, [p_n\ e_n] \rangle \Downarrow v'}$$

E.0.3 Rules for pattern matching.

$$\text{(Pat-MatchVCon)}\ \frac{\langle \rho, p_i, v_i \rangle \rightarrowtail s_i, \quad 1 \le i \le m \qquad s = s_1 \uplus \cdots \uplus s_m}{\langle \rho, K[p_1 \ldots p_m], K[v_1' \ldots v_m'] \rangle \rightarrowtail s}$$

$$\text{(Pat-FailVcon)}\ \frac{v \text{ does not have the form } K[v'_1, \ldots v'_m]}{\langle \rho, K\ p_1, \ldots p_m, v\rangle \rightarrowtail \dagger}$$

$$\text{(Pat-MatchBareVcon)}\ \frac{}{\langle \rho, K, K\rangle \rightarrowtail \{\}}$$

$$\text{(Pat-FailBareVcon)}\ \frac{v \neq K}{\langle \rho, K, v\rangle \rightarrowtail \dagger}$$

$$\text{(Pat-MatchVar)}\ \frac{}{\langle \rho, x, v\rangle \rightarrowtail \{x \mapsto v\}}$$

$$\text{(Pat-MatchWhen)}\ \frac{\langle \rho, e\rangle \Downarrow v' \qquad v' \neq \mathit{false}}{\langle \rho, \mathit{when}\ e, v\rangle \rightarrowtail \{\}}$$

$$\text{(Pat-MatchWildcard)}\ \frac{}{\langle \rho,, v\rangle \rightarrowtail \{\}}$$

$$\text{(Pat-FailWhen)}\ \frac{\langle \rho, e\rangle \Downarrow v' \qquad v' = \mathit{false}}{\langle \rho, \mathit{when}\ e, v\rangle \rightarrowtail \dagger}$$

$$\text{(Pat-ArrowExp)}\ \frac{\langle \rho, e\rangle \Downarrow v' \qquad \langle \rho, p, v'\rangle \rightarrowtail s}{\langle \rho, p \leftarrow e, v\rangle \rightarrowtail s}$$

$$\text{(Pat-MultiFail)}\ \frac{\langle \rho, p, v\rangle \rightarrowtail \dagger}{\langle \rho, p_1, p_2, v\rangle \rightarrowtail \dagger}$$

$$\text{(PAT-MULTISOLUTION)}\ \frac{\langle \rho, p_1, v\rangle \rightarrowtail \rho' \qquad \langle \rho \uplus \rho', p_2, v\rangle \rightarrowtail s}{\langle \rho, p_1, p_2, v\rangle \rightarrowtail s}$$

$$\text{(PAT-ORFST)}\ \frac{\langle \rho, p_1, v\rangle \rightarrowtail \rho'}{\langle \rho, p_1 \mid p_2, v\rangle \rightarrowtail \rho'}$$

$$\text{(PAT-ORSND)}\ \frac{\langle \rho, p_1, v\rangle \rightarrowtail \dagger \qquad \langle \rho, p_2, v\rangle \rightarrowtail s}{\langle \rho, p_1 \mid p_2, v\rangle \rightarrowtail s}$$

E.0.4 *Judgement forms for V^-.*

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

$$\rho \vdash g \rightarrowtail \rho' \ \ \text{(GUARD-REFINE)}$$

$$\rho \vdash g \rightarrowtail \dagger \ \ \text{(GUARD-REJECT)}$$

E.0.5 *Shifting a guard to the context.*

$$\text{(MOVE-GUARD-TO-CTX)}\ \frac{\rho; g \cdot \mathcal{T} \vdash gs \cdot gs' \rightarrowtail s}{\rho; \mathcal{T} \vdash gs \cdot g \cdot gs' \rightarrowtail s}$$

E.0.6 *Choosing and processing a guard.*

$$\text{(SOLVE-GUARD-REFINE)}\ \frac{\rho \vdash g \rightarrowtail \rho' \qquad \rho';\ \mathcal{T} \cdot \mathcal{T}' \vdash gs \rightarrowtail s}{\rho; \mathcal{T} \cdot g \cdot \mathcal{T}' \vdash gs \rightarrowtail s}$$

$$\text{(SOLVE-GUARD-REJECT)}\ \frac{\rho \vdash g \rightarrowtail \dagger}{\rho; \mathcal{T} \cdot g \cdot \mathcal{T}' \vdash gs \rightarrowtail \dagger}$$

E.0.7 Properties of guards.

$$(\textsc{Multi-Guard-Assoc})\ \frac{\rho; \mathcal{T} \cdot g_1 \cdot g_2 \cdot \mathcal{T}' \vdash gs \rightarrowtail s}{\rho; \mathcal{T} \cdot g_2 \cdot g_1 \cdot \mathcal{T}' \vdash gs \rightarrowtail s}$$

E.0.8 Refinement with different types of guards.

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

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

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

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

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

$$(\textsc{Guard-VCon-Fail})\ \frac{x \in \operatorname{dom}\rho \qquad \rho x = v \qquad v \text{ does not have the form } K[v'_1, \ldots v'_m]}{\rho \vdash x = K\ e_1, \ldots e_m \rightarrowtail \dagger}$$

$$(\textsc{Guard-Expseq-Succ})\ \frac{\langle \rho, e \rangle \Downarrow v}{\rho \vdash e \rightarrowtail \{\}}$$

$$(\textsc{Guard-Expseq-Fail})\ \frac{\langle \rho, e \rangle \Downarrow \mathbf{fail}}{\rho \vdash e \rightarrowtail \dagger}$$

$$(\textsc{Guard-Choice-First})\ \frac{\rho; \varepsilon \vdash gs \rightarrowtail \rho'}{\rho \vdash gs \mid gs' \rightarrowtail \rho'}$$

$$(\textsc{Guard-Choice-Second})\ \frac{\rho; \varepsilon \vdash gs \rightarrowtail \dagger \qquad \rho; \varepsilon \vdash gs' \rightarrowtail s}{\rho \vdash gs \mid gs' \rightarrowtail s}$$

E.0.9 Evaluating General Expressions.

$$(\textsc{Eval-IfFi-Fail})\ \frac{}{\langle \rho, \textsc{if}\ [\ \ ]\ \textsc{fi} \rangle \Downarrow \mathbf{fail}}$$

$$(\textsc{Eval-IfFi-Success})\ \frac{\rho' = \rho\{x_1 \mapsto \bot \ldots x_n \mapsto \bot\} \qquad \rho'; \varepsilon \vdash gs \rightarrowtail \rho'' \qquad \langle \rho'', e \rangle \Downarrow r}{\langle \rho, \textsc{if}\ [\ \exists x_1 \ldots x_n.\ gs \rightarrow e \ \square\ \ldots\ ]\ \textsc{fi} \rangle \Downarrow r}$$

$$\text{(Eval-IfFi-Reject)}\ \frac{\begin{array}{c}\rho' = \rho\{x_1 \mapsto \bot \ldots x_n \mapsto \bot\} \\ \rho'; \varepsilon \vdash gs \rightarrowtail \dagger \\ \langle \rho, \textsc{if}\ [\ \ldots\ ]\ \textsc{fi}\rangle \Downarrow r\end{array}}{\langle \rho, \textsc{if}\ [\ \exists x_1 \ldots x_n.\ gs \rightarrow e\ \Box\ \ \ldots\ ]\ \textsc{fi}\rangle \Downarrow r}$$

$$\text{(Eval-VconEmpty)}\ \frac{}{\langle \rho, K\rangle \Downarrow K}$$

$$\text{(Eval-VconMulti)}\ \frac{\langle \rho, e_i\rangle \Downarrow v_i \qquad 1 \leq i \leq n}{\langle \rho, K(e_1, \ldots e_n)\rangle \Downarrow K(v_1, \ldots v_i)}$$

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

$$\text{(Eval-Name)}\ \frac{\begin{array}{c}x \in \operatorname{dom} \rho \\ \rho(x) = v\end{array}}{\langle \rho, x\rangle \Downarrow v}$$

$$\text{(Eval-Name-Fail)}\ \frac{x \notin \operatorname{dom} \rho}{\langle \rho, x\rangle \Downarrow \textbf{fail}}$$

$$\text{(Eval-LambdaDecl)}\ \frac{}{\langle \rho, \lambda x.e\rangle \Downarrow \lambda x.e}$$

$$\text{(Eval-FunApp)}\ \frac{\begin{array}{c}\langle \rho, e_1\rangle \Downarrow \lambda x.e \\ \langle \rho, e_2\rangle \Downarrow v' \\ \langle \rho\{x \mapsto v'\}, e\rangle \Downarrow r\end{array}}{\langle \rho, e_1\ e_2\rangle \Downarrow r}$$

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