

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 *shapes*, 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.

```

124     fun area sh =
125         if isSquare sh
126         then sqSide sh * sqSide sh
127         else if isTriangle sh
128         then 0.5 * triW sh * triH sh
129         else 0.5 * traB1 sh * traB2 sh * traH sh
130
131
132
133                                     (a) area with observers
134     fun area sh =
135         case sh
136         of SQUARE s                => s * s
137          | TRIANGLE (w, h)         => 0.5 * w * h
138          | TRAPEZOID (b1, b2, h) => 0.5 * b1 * b2 * h
139
140
141
142
143                                     (b) area with pattern matching
144

```

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 observers. 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

¹Sometimes 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 Haskell³.

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`:

²OCaml, 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.

³I 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.

```

247 type shape = Square of float
248             | Triangle of float * float
249             | Trapezoid of float * float * float
250
251
252 exclaimTall : shape -> string
253 (*
254 exclaimTall (Square s)           == "Wow! That's a tall square!",
255                                 where s > 100.0
256 exclaimTall (Triangle (w, h))    == "Goodness! Towering triangle!",
257                                 where h > 100.0
258 exclaimTall (Trapezoid (b1, b2, h)) == "Zounds! Tremendous trapezoid!",
259                                 where h > 100.0
260
261 exclaimBigArea sh                == "Your shape is small.",
262                                 otherwise
263 *)
264
265

```

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

```

269 let exclaimTall sh =
270 match sh with
271 | Square s -> if s > 100.0
272               then "Wow! That's a tall square!"
273               else "Your shape is small."
274 | Triangle (_, h) ->
275   if h > 100.0
276   then "Goodness! Towering triangle!"
277   else "Your shape is small."
278 | Trapezoid (_, _, h) ->
279   if h > 100.0
280   then "Zounds! Tremendous trapezoid!"
281   else "Your shape is small."
282
283

```

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

```

288         let exclaimTall sh =
289             match sh with
290             | Square s when s > 100.0 ->
291                 "Wow! That's a tall square!"
292             | Triangle (_, h) when h > 100.0 ->
293                 "Goodness! Towering triangle!"
294             | Trapezoid (_, _, h) when h > 100.0 ->
295                 "Zounds! Tremendous trapezoid!"
296             | _ -> "Your shape is small."
297
298

```

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!

```

411         tripleLookup rho x
412         | Just w <- lookup rho x
413         , Just y <- lookup rho w
414         , Just z <- lookup rho y
415         = z
416     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`.

```

452   type funlevel = int
453
454   type token = BattlePass  of funlevel | ChugJug    of funlevel | TomatoTown of funlevel
455               | HuntersMark of funlevel | SawCleaver of funlevel
456               | MoghLordOfBlood of funlevel | PreatorRykard of funlevel
457               ... other tokens ...
458
459
460   game_of_token : token -> string * funlevel
461
462   game_of_token t == ("Fortnite", f), where t is any of
463                               BattlePass f,
464                               ChugJug f, or
465                               TomatoTown f
466   game_of_token t == ("Bloodborne", 2 * f),
467                               where t is any of
468                               HuntersMark f or
469                               SawCleaver f
470   game_of_token t == ("Elden Ring", 3 * f),
471                               where t is any of
472                               MoghLordOfBlood f or
473                               PreatorRykard f
474   game_of_token _ == ("Irrelevant", 0), otherwise
475
476

```

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

482
483 I could try to use a couple of helper functions to reduce clutter, and write something like
484 Figure 7. It looks ok, but I'm still hurting for Nice Properties 2 and 4, and the additional
485 calls hurt performance.

486
487 Thankfully, an extension once again comes to the rescue. *Or-patterns* condense multiple
488 patterns which share a right-hand side, and when any one of the patterns matches with
489 the scrutinee, the right-hand side is evaluated with the bindings created by that pattern. I

490
491
492

```

493     let game_of_token token = match token with
494     | BattlePass f      -> ("Fortnite", f)
495     | ChugJug f        -> ("Fortnite", f)
496     | TomatoTown f     -> ("Fortnite", f)
497     | HuntersMark f    -> ("Bloodborne", 2 * f)
498     | SawCleaver f     -> ("Bloodborne", 2 * f)
499     | MoghLordOfBlood f -> ("Elden Ring", 3 * f)
500     | PreatorRykard f  -> ("Elden Ring", 3 * f)
501     | _                -> ("Irrelevant", 0)
502
503

```

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

```

505
506     let fortnite f = ... complicated ... in
507     let bloodborne f = ... complicated' ... in
508     let eldenring f = ... complicated'' ... in
509     match token with
510     | BattlePass f -> fortnite f
511     ... and so on ...
512

```

Fig. 7. `game_of_token` with helpers is somewhat better, but I'm not satisfied with it.

```

515 let game_of_token token = match token with
516 | BattlePass f | ChugJug f | TomatoTown f -> ("Fortnite", f)
517 | HuntersMark f | SawCleaver f           -> ("Bloodborne", 2 * f)
518 | MoghLordOfBlood f | PreatorRykard f    -> ("Elden Ring", 3 * f)
519 | _                                         -> ("Irrelevant", 0)
520
521

```

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.

```

575      ∃ area. area = λ sh.
576      one {  ∃ s. sh = ⟨SQUARE, s⟩; s * s
577            | ∃ w h. sh = ⟨TRIANGLE, w, h⟩;
578              0.5 * w * h
579            | ∃ b1 b2 h. sh = ⟨TRAPEZOID, b1, b2, h⟩;
580              0.5 * b1 * b2 * h}
581

```

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 \exists . It relies more on mathematical notation, but that might not be a bad thing:

- 616 though it doesn't resemble the algebraic laws a programmer would write, it likely
 617 resembles the equations that a mathematician would.
- 618 (2) area using equations does not duplicate virtually any code.
 - 619 (3) area using equations deconstructs user-defined types as easily as **1b** does with
 620 pattern matching.
 - 621 (4) area using equations has all important internal values named very explicitly.
 - 622 (5) This Property may not hold, because \mathcal{VC} on its own is untyped. Without a type
 623 system, a compiler cannot help me with non-exhaustive cases. However, there is
 624 no published compiler, type system or no, for Verse, and only when one is made
 625 available can I make this assertion. For this reason, and for the sake of focusing
 626 on more important details, I choose to proceed in my analysis of equations in \mathcal{VC}
 627 without considering this Property.

631 If you still believe these Properties to be desirable, you understand why I claim program-
 632 mers prefer equations to observers. Now I'll make a stronger claim: equations are *at least*
 633 as good as pattern matching with popular extensions. How can I claim this? By appealing
 634 again to the Nice Properties! In Section 2.1, I demonstrated how pattern matching had to
 635 resort to extensions to regain the Properties when challenging examples stole them away.
 636 In Figure 10, I've implemented those examples in \mathcal{VC} (this time extended with strings,
 637 floats, and $*$) using choice and equations. Take a look for yourself!

638 The code in Figure 10 has all the Nice Properties (again, disregarding the ambiguous
 639 5th.) This is promising for \mathcal{VC} . If it rivals pattern matching with popular extensions in
 640 desirable properties, and \mathcal{VC} does everything using only equations and choice, it seems
 641 like the language is a strong option for writing code!

645 2.4 \mathcal{VC} has a challenging cost model

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

649 \mathcal{VC} 's equations appear to be comparably pleasing to pattern matching in their brevity
 650 and expressiveness. However, full Verse allows computations that are problematic, cost-
 651 wise. In \mathcal{VC} , names (logical variables) are *values*, and can just as easily be the result of
 652
 653
 654
 655
 656


```

657  $\exists$  exclaimTall. exclaimTall =  $\lambda$  sh.
658   one {
659      $\exists$  s. sh =  $\langle$ Square s $\rangle$ ;
660     s > 100.0; "Wow! That's a tall square!"
661   |  $\exists$  w h. sh =  $\langle$ Triangle, w, h $\rangle$ ;
662     h > 100.0; "Goodness! Towering triangle!"
663   |  $\exists$  b1 b2 h. sh =  $\langle$ Trapezoid, b1, b2, h $\rangle$ ;
664     h > 100.0; "Zounds! Tremendous trapezoid!"
665   | "Your shape is small." }

```

(a) exclaimTall in \mathcal{VC}

```

 $\exists$  tripleLookup. tripleLookup =  $\lambda$  rho x.
  one {  $\exists$  w. lookup rho x =  $\langle$ Just w $\rangle$ ;
         $\exists$  y. lookup rho w =  $\langle$ Just y $\rangle$ ;
         $\exists$  z. lookup rho y =  $\langle$ Just z $\rangle$ ;
        z
      | handleFailure x }

```

(b) tripleLookup in \mathcal{VC}

```

 $\exists$  game_of_token. game_of_token =  $\lambda$  token.
   $\exists$  f. one {
667     token = one {  $\langle$ BattlePass, f $\rangle$  |  $\langle$ ChugJug, f $\rangle$  |  $\langle$ TomatoTown, f $\rangle$ };
668      $\langle$ "Fortnite", f $\rangle$ 
669   | token = one {  $\langle$ HuntersMark, f $\rangle$  |  $\langle$ SawCleaver, f $\rangle$ };
670      $\langle$ "Bloodborne", 2 * f $\rangle$ 
671   | token = one {  $\langle$ MoghLordOfBlood, f $\rangle$  |  $\langle$ PreatorRykard, f $\rangle$ };
672      $\langle$ "Elden Ring", 3 * f $\rangle$ 
673   |  $\langle$ "Irrelevant", 0 $\rangle$  }

```

(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 ??, 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 viable⁴ 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 .

⁴Not to mention meditative.

780	Programs	P	$::=$	$\{d\}$	definition
781	Definitions	d	$::=$	$\text{val } x = e$	bind name to expression
782	Expressions	e	$::=$	v	literal values
783				$ $ x, y, z	names
784				$ $ $K\{e\}$	value constructor application
785				$ $ $\lambda x. e$	lambda declaration
786				$ $ $e_1 e_2$	function application
787				$ $ $\text{case } e \{p \rightarrow e\}$	case expression
788				$ $ (e)	
789	Patterns	p	$::=$	$p_1 \mid p_2$	or-pattern
790				$ $ p, p'	pattern guard
791				$ $ $p \leftarrow e$	pattern from explicit expression
792				$ $ x	name
793				$ $ $-$	wildcard
794				$ $ $K\{p\}$	value constructor application
795				$ $ $\text{when } e$	
796				$ $ (p)	
797	Values	v	$::=$	$K\{v\}$	value constructor application
798				$ $ $\lambda x. e$	lambda value
799	Value Constructors	K	$::=$	$\text{true} \mid \text{false}$	booleans
800				$ $ $A\text{-}Zx$	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 (\dagger). The metavariable s , a *solution* to a pattern match, combines these outcomes.

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

(MATCH-SUCCESS)

$\langle \rho, p, v \rangle \mapsto \dagger$ 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 $\langle \rho, p \leftarrow e, v \rangle \mapsto s$, 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: $name \rightarrow \mathcal{V}$
$\rho + \rho'$	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 $b \leftarrow \text{mumble } s \rightarrow b$, the result of evaluating x matches with Square s , and the result of evaluating $\text{mumble } s$ matches with Big b . Generally: when $\langle \rho, p_1, p_2 \leftarrow e, v \rangle \mapsto s$, $\langle \rho, p_1, v \rangle \mapsto s_1$, $\langle \rho, e \rangle \Downarrow v'$, and $\langle \rho, p_2, v' \rangle \mapsto s_2$.

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_1 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, \dots, 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 $\rho_1 \uplus \rho_2$, and it is defined if and only if $\text{dom } \rho_1 \cap \text{dom } \rho_2 = \emptyset$:

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

For example, in the P^+ expression `case Pair 1 (Pair 2 3) of Pair x (Pair y z) -> z`, the right-hand side `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 $\text{dom } \rho_1 \cap \text{dom } \rho_2 \neq \emptyset$, 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^- .

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

$$\langle \rho, p, v \rangle \mapsto \rho' \text{ (MATCH-SUCCESS)}$$

$$\langle \rho, p, v \rangle \mapsto \dagger \text{ (MATCH-FAIL)}$$

3.3.2 Evaluating General Expressions.

$$\text{(EVAL-VCONEMPTY)} \frac{}{\langle \rho, K \rangle \Downarrow K}$$

$$\text{(EVAL-VCONMULTI)} \frac{\langle \rho, e_i \rangle \Downarrow v_i \quad 1 \leq i \leq n}{\langle \rho, K(e_1, \dots, e_n) \rangle \Downarrow K(v_1, \dots, v_i)}$$

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

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

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

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

$$\text{(EVAL-CASESCRUTINEE)} \frac{\langle \rho, e \rangle \Downarrow v \quad \langle \rho, \text{case } v \text{ } [p_1 \ e_1], \dots, [p_n \ e_n] \rangle \Downarrow v'}{\langle \rho, \text{case } e \text{ } [p_1 \ e_1], \dots, [p_n \ e_n] \rangle \Downarrow v'}$$

$$(EVAL-CASEMATCH) \frac{\langle \rho, p_1, v \rangle \rightarrow \rho' \quad \langle \rho + \rho', e_1 \rangle \Downarrow v}{\langle \rho, case\ v\ [p_1\ e_1], \dots, [p_n\ e_n] \rangle \Downarrow v'}$$

$$(EVAL-CASEFAIL) \frac{\langle \rho, p_1, v \rangle \rightarrow \dagger \quad \langle \rho, case\ v\ [p_2\ e_2], \dots, [p_n\ e_n] \rangle \Downarrow v'}{\langle \rho, case\ v\ [p_1\ e_1], \dots, [p_n\ e_n] \rangle \Downarrow v'}$$

3.3.3 Rules for pattern matching.

$$(PAT-MATCHVCON) \frac{\langle \rho, p_i, v_i \rangle \rightarrow s_i, \quad 1 \leq i \leq m \quad s = s_1 \uplus \dots \uplus s_m}{\langle \rho, K[p_1 \dots p_m], K[v'_1 \dots v'_m] \rangle \rightarrow s}$$

$$(PAT-FAILVCON) \frac{v \text{ does not have the form } K[v'_1, \dots v'_m]}{\langle \rho, K\ p_1, \dots p_m, v \rangle \rightarrow \dagger}$$

$$(PAT-MATCHBAREVCON) \frac{}{\langle \rho, K, K \rangle \rightarrow \{\}}$$

$$(PAT-FAILBAREVCON) \frac{v \neq K}{\langle \rho, K, v \rangle \rightarrow \dagger}$$

$$(PAT-MATCHVAR) \frac{}{\langle \rho, x, v \rangle \rightarrow \{x \mapsto v\}}$$

$$(PAT-MATCHWHEN) \frac{\langle \rho, e \rangle \Downarrow v' \quad v' \neq false}{\langle \rho, when\ e, v \rangle \rightarrow \{\}}$$

$$(PAT-MATCHWILDCARD) \frac{}{\langle \rho, v \rangle \rightarrow \{\}}$$

$$\begin{array}{l}
\text{(PAT-FAILWHEN)} \quad \frac{\langle \rho, e \rangle \Downarrow v' \quad v' = \text{false}}{\langle \rho, \text{when } e, v \rangle \mapsto \dagger} \\
\\
\text{(PAT-ARROWEXP)} \quad \frac{\langle \rho, e \rangle \Downarrow v' \quad \langle \rho, p, v' \rangle \mapsto s}{\langle \rho, p \leftarrow e, v \rangle \mapsto s} \\
\\
\text{(PAT-MULTIFAIL)} \quad \frac{\langle \rho, p, v \rangle \mapsto \dagger}{\langle \rho, p_1, p_2, v \rangle \mapsto \dagger} \\
\\
\text{(PAT-MULTISOLUTION)} \quad \frac{\langle \rho, p_1, v \rangle \mapsto \rho' \quad \langle \rho \uplus \rho', p_2, v \rangle \mapsto s}{\langle \rho, p_1, p_2, v \rangle \mapsto s} \\
\\
\text{(PAT-ORFST)} \quad \frac{\langle \rho, p_1, v \rangle \mapsto \rho'}{\langle \rho, p_1 \mid p_2, v \rangle \mapsto \rho'} \\
\\
\text{(PAT-ORSND)} \quad \frac{\langle \rho, p_1, v \rangle \mapsto \dagger \quad \langle \rho, p_2, v \rangle \mapsto s}{\langle \rho, p_1 \mid p_2, v \rangle \mapsto s}
\end{array}$$

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

```

1026 val exclaimTall = \sh.
1027   case sh of Square s, when (> s) 100 ->
1028     Wow! That's A Tall Square!
1029   | Triangle w h, when (> s) 100 ->
1030     Goodness! Towering Triangle!
1031   | Trapezoid b1 b2 h, when (> s) 100 ->
1032     Zounds! Tremendous Trapezoid!
1033   | _ -> 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 runtime and never backtracks.
V^- uses choice.	V^- 's choice can only guard computation and its result is never returned.
V^- uses <i>success</i> and <i>failure</i> 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	$::=$	$\text{val } x = e$	bind name to expression
Expressions	e	$::=$	v	literal values
			$ \quad x, y, z$	names
			$ \quad \text{if } [G \{[] G\}] \text{ fi}$	if-fi
			$ \quad K\{e\}$	value constructor application
			$ \quad e_1 e_2$	function application
Guarded Expressions	G	$::=$	$ \quad \lambda x. e$	lambda declaration
			$ \quad [E \{x\}.] \{g\} \rightarrow e$	names, guards, and body
			$ \quad x = e$	equation
Guards	g	$::=$	$ \quad g\{;g\} \mid g\{;g\}$	choice
Values	v	$::=$	$ \quad K\{v\}$	value constructor application
			$ \quad \lambda 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	$v \mathbf{fail}$: expressions produce <i>results</i> : 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 <i>solutions</i> : 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**.

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

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

3.6.2 Guards. For example, the V^- expression $((\backslash x. x) K)$ 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.

$$\rho; \mathcal{T} \vdash gs \mapsto s \text{ (SOLVE-GUARDS)}$$

The environment ρ maps from a name to a value vs or \perp . \perp 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 \dagger to represent rejection, and a guard producing \dagger will cause the top-level list of guards to also produce \dagger .

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

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

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

3.6.3 Refinement ordering on environments.

$$\rho \subseteq \rho' \text{ when } \text{dom } \rho \subseteq \text{dom } \rho'$$

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

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

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

3.7.1 Judgement forms for V^- .

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

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

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

3.7.2 Shifting a guard to the context.

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

3.7.3 Choosing and processing a guard.

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

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

3.7.4 Properties of guards.

$$\text{(MULTI-GUARD-ASSOC)} \frac{\rho; \mathcal{T} \cdot g_1 \cdot g_2 \cdot \mathcal{T}' \vdash gs \rightsquigarrow s}{\rho; \mathcal{T} \cdot g_2 \cdot g_1 \cdot \mathcal{T}' \vdash gs \rightsquigarrow s}$$

3.7.5 Refinement with different types of guards.

$$\text{(GUARD-NAMEEXP-BOT)} \frac{\begin{array}{c} x \in \text{dom } \rho \\ \langle \rho, e \rangle \Downarrow v \\ \rho(x) = \perp \end{array}}{\rho \vdash x = e \rightsquigarrow \rho\{x \mapsto v\}}$$

1272	
1273	$x \in \text{dom } \rho$
1274	$\langle \rho, e \rangle \Downarrow v$
1275	$\rho(x) = v$
1276	(GUARD-NAMEEXP-EQ) $\frac{\rho(x) = v}{\rho \vdash x = e \rightarrow \rho}$
1277	
1278	
1279	$x \in \text{dom } \rho$
1280	$\langle \rho, e \rangle \Downarrow v$
1281	$\rho(x) = v'$
1282	$v \neq v'$
1283	(GUARD-NAMEEXP-FAIL) $\frac{v \neq v'}{\rho \vdash x = e \rightarrow \dagger}$
1284	
1285	
1286	
1287	$x, y \in \text{dom } \rho$
1288	$\rho(x) = \perp, \rho(y) = v$
1289	(GUARD-NAMES-BOT-SUCC) $\frac{\rho(x) = \perp, \rho(y) = v}{\rho \vdash x = y \rightarrow \rho\{x \mapsto v\}}$
1290	
1291	
1292	$x, y \in \text{dom } \rho$
1293	$\rho(x) = v, \rho(y) = \perp$
1294	(GUARD-NAMES-BOT-SUCC-REV) $\frac{\rho(x) = v, \rho(y) = \perp}{\rho \vdash x = y \rightarrow \rho\{y \mapsto v\}}$
1295	
1296	
1297	
1298	$x \in \text{dom } \rho$
1299	$\rho x = v$
1300	v does not have the form $K[v'_1, \dots, v'_m]$
1301	(GUARD-VCON-FAIL) $\frac{v \text{ does not have the form } K[v'_1, \dots, v'_m]}{\rho \vdash x = K e_1, \dots e_m \rightarrow \dagger}$
1302	
1303	
1304	$\langle \rho, e \rangle \Downarrow v$
1305	(GUARD-EXPSEQ-SUCC) $\frac{\langle \rho, e \rangle \Downarrow v}{\rho \vdash e \rightarrow \{\}}$
1306	
1307	
1308	$\langle \rho, e \rangle \Downarrow \mathbf{fail}$
1309	(GUARD-EXPSEQ-FAIL) $\frac{\langle \rho, e \rangle \Downarrow \mathbf{fail}}{\rho \vdash e \rightarrow \dagger}$
1310	
1311	
1312	

$$\text{(GUARD-CHOICE-FIRST)} \quad \frac{\rho; \varepsilon \vdash gs \multimap \rho'}{\rho \vdash gs \mid gs' \multimap \rho'}$$

$$\text{(GUARD-CHOICE-SECOND)} \quad \frac{\rho; \varepsilon \vdash gs \multimap \dagger \quad \rho; \varepsilon \vdash gs' \multimap s}{\rho \vdash gs \mid gs' \multimap s}$$

3.7.6 Evaluating General Expressions.

$$\text{(EVAL-IFFI-FAIL)} \quad \frac{}{\langle \rho, \text{IF } [\] \text{ FI} \rangle \Downarrow \mathbf{fail}}$$

$$\text{(EVAL-IFFI-SUCCESS)} \quad \frac{\begin{array}{c} \rho' = \rho\{x_1 \mapsto \perp \dots x_n \mapsto \perp\} \\ \rho'; \varepsilon \vdash gs \multimap \rho'' \\ \langle \rho'', e \rangle \Downarrow r \end{array}}{\langle \rho, \text{IF } [\exists x_1 \dots x_n. gs \rightarrow e \square \dots] \text{ FI} \rangle \Downarrow r}$$

$$\text{(EVAL-IFFI-REJECT)} \quad \frac{\begin{array}{c} \rho' = \rho\{x_1 \mapsto \perp \dots x_n \mapsto \perp\} \\ \rho'; \varepsilon \vdash gs \multimap \dagger \\ \langle \rho, \text{IF } [\dots] \text{ FI} \rangle \Downarrow r \end{array}}{\langle \rho, \text{IF } [\exists x_1 \dots x_n. gs \rightarrow e \square \dots] \text{ FI} \rangle \Downarrow r}$$

$$\text{(EVAL-VCONEMPTY)} \quad \frac{}{\langle \rho, K \rangle \Downarrow K}$$

$$\text{(EVAL-VCONMULTI)} \quad \frac{\langle \rho, e_i \rangle \Downarrow v_i \quad 1 \leq i \leq n}{\langle \rho, K(e_1, \dots e_n) \rangle \Downarrow K(v_1, \dots v_i)}$$

$$\text{(EVAL-VCONMULTI-FAIL)} \quad \frac{\exists e_i. 1 \leq i \leq n : \langle \rho, e_i \rangle \Downarrow \mathbf{fail}}{\langle \rho, K(e_1, \dots e_n) \rangle \Downarrow \mathbf{fail}}$$

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

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

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

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

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

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 ???. 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. Maranget uses vectors and matrices to express multiple simultaneous matches of values to patterns as a single match of a vector with a matrix row. This allows him to run a *specialization* pass that reduces the number of rows in the matrix, ultimately leading to smaller trees.

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 begins 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:

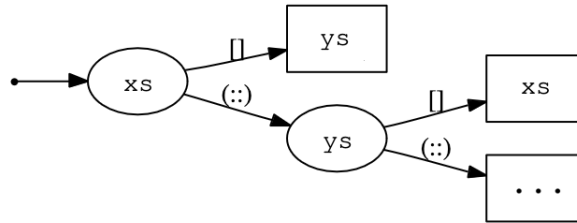


Fig. 16. The final compiled decision tree for `merge`, right-to-left

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

1477	Programs	P	$::=$	$\{d\}$	definition
1478	Definitions	d	$::=$	$\text{val } x = e$	bind name to expression
1479	Expressions	e	$::=$	v	literal values
1480				$ $ x, y, z	names
1481				$ $ $K\{e\}$	value constructor application
1482				$ $ $\lambda x. e$	lambda declaration
1483				$ $ $e_1 e_2$	function application
1484				$ $ t	decision tree
1485				$ $ (e)	
1486	Decision Trees	t	$::=$	$\text{test } x \{K/i\} \rightarrow t [x \rightarrow t]$	test node
1487				$ $ $\text{try-let } x = e \text{ in } t [; \text{otherwise } t]$	try-let node
1488				$ $ $\text{cmp } x = e \rightarrow t, t, t$	comparison node
1489				$ $ $\text{extract } x \{y\} e \text{ in } t$	extract node
1490				$ $ fail	fail node
1491	Values	v	$::=$	$K\{v\}$	value constructor application
1492				$ $ $\lambda x. e$	lambda value
1493	Value Constructors	K	$::=$	$\text{true} \mid \text{false}$	booleans
1494				$ $ $A-Zx$	name beginning with capital letter
1495					
1496					

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: $V^- \rightarrow D$

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} :

$$\begin{aligned} \mathcal{I} [[\text{if } \dots \square gs_1; gs_2 \mid gs_3; gs_4 \rightarrow e \square \dots fi]] \\ == \\ \text{if } \dots \square gs_1; gs_2 \rightarrow e \square gs_3; gs_4 \rightarrow e \square \dots fi \end{aligned}$$

1518 With the new context and desugared *if-fi*, *compile* then repeatedly chooses a guarded
 1519 expression G and attempts the following while translating G 's internal expressions with
 1520 \mathcal{D} :
 1521

- 1522 (1) If there are no guards in G , insert a *match* node with the right-hand side of G .
- 1523 (2) Otherwise, choose an equation in G of the form $x = K \dots$ s.t. x is *known*.
- 1524 (3) If one is found, use it to generate a *test* node, building each subtree of the *test* by
 1525 pruning all branches in *all* guarded expressions of the *if-fi* in which $x = e$ and
 1526 $e \neq K \dots$ and invoking *compile* on the remaining ones with a context where x is
 1527 known. Each of these remainders is the child of an *extract* node which extracts
 1528 the internals of each possible value constructor into a list of names which are
 1529 introduced to the context. The algorithm builds the default tree of the *test* by finding
 1530 any branches of the in the *if-fi* in which x is not bound to a value constructor.
 1531
- 1532 (4) If no equation of the form $x = K \dots$ is found, try to find an equation $x = e$ s.t. either
 1533 x is unknown and all names in e are known.
 1534
- 1535 (5) If one is found, use it to generate a *try-let* node with two children: one if binding
 1536 succeeds, and one if e fails. *compile* prunes each subtree of the *try-let* accordingly.
 1537
- 1538 (6) If no equation is found, try to find an equation $x = y$ s.t. x is known and y is
 1539 unknown.
 1540
- 1541 (7) If one is found, use it to generate a *try-let* node with one child for when binding
 1542 succeeds. *compile* prunes the subtree of all instance of $x = y$.
 1543
- 1544 (8) If no such equation is found, try to find a condition e s.t. all names in e are known.
- 1545 (9) If one is found, generate a fresh name x' and use it to generate a *try-let* node for
 1546 the equation $x = e'$, pruning each subtree of e with a substitution of $[x/e]$.
 1547
- 1548 (10) If no condition e is found, try to find an equation $x = e$ s.t. both x and all names in
 1549 e are *known*.
 1550
- 1551 (11) If one is found, generate a *cmp* node, prune the *if-fi* of all duplicate instances of
 1552 that $x = e$, and invoke *compile* again.
 1553
- 1554 (12) If none is found, the *if-fi* cannot be compiled to a decision tree. The algorithm halts
 1555 with an error.
 1556

1557 The algorithm terminates when inserts a final *match* node for a right-hand side expres-
 1558 sion 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 $\text{dom}(b)$ to describe the set of all names that appear in a branch. I use the notation $\text{branches} - -g$ to mean “branches pruned of guard g ” and the notation $\text{branches} - - - g$ to mean “branches pruned of all branches containing g .” I use the standard substitution notation $\text{branches}[x/e]$ to mean “branches with name x substituted for expression e .” I use a shorthand $(\text{context} + \{n_1 \dots n_i \mapsto \text{known}\})$ 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.

```

1600 Require:  $\forall (n : \text{name}) \in \text{branches}, n \text{ unique}$ 
1601   compile context branches =
1602   if null(branches) then fail
1603   else if  $\neg \exists g \in \text{fst}(\text{hd}(\text{branches}))$  then
1604     match (hd branches)
1605   else
1606     let  $g$  be a guard such that  $g \in \text{branches}$  in
1607     if  $g$  has the form  $x = K(e_1 \dots e_i)$  then let
1608        $KS = \{K \mid \exists b \in \text{branches} : x \in \text{dom}(b) \wedge K(\dots) \in x@b\}$ 
1609        $\text{edges} = \text{map}(\lambda K. (\text{let } ns = n_1 \dots n_i, n_x \text{ fresh in}$ 
1610          $(K, \text{extract}(x, ns, \text{compile}(\text{context} + \{n_1 \dots n_i \mapsto \text{known}\}))$ 
1611          $(\text{mapPartial}(\text{refine } x (K(ns))) \text{branches}))$ 
1612       end)
1613        $\text{defaults} = \text{filter}(\lambda b.x \notin \text{dom}(b) \vee K(\dots) \notin x@b)$ 
1614       in  $\text{test}(x, \text{edges}, \text{SOME}(\text{compile defaults}))$ 
1615     end
1616     else if  $g$  has the form  $x = e : x \text{ unknown}, e \text{ known}$  then
1617        $\text{try\_let}(x, \mathcal{D}(\text{context } e),$ 
1618        $\text{compile}(\text{context}\{x \mapsto \text{known}\})((\text{branches} - \text{eq})[x/e]),$ 
1619        $\text{SOME}(\text{compile context } (\text{branches} - \text{eq} - e)))$ 
1620     else if  $g$  has the form  $x = y : x \text{ known}, y \text{ unknown}$  then
1621        $\text{try\_let}(x, y, \text{compile}(\text{context}\{y \mapsto \text{known}\})(\text{branches} - \text{eq}), \text{NONE})$ 
1622     else if  $g$  has the form  $x = e : x \text{ known}, e \text{ known}$  then
1623        $\text{cmp}(x, \mathcal{D}(\text{context } e),$ 
1624        $\text{compile context } ((\text{branches} - \text{eq})[x/e]),$ 
1625        $\text{compile context } (\text{branches} - \text{eq} - e),$ 
1626        $\text{SOME}(\text{compile context } (\text{branches} - \text{eq} - e)))$ 
1627     else if  $g$  has the form  $e : e \text{ known}$  then
1628        $\text{try\_let}(x, \mathcal{D}(\text{context } e),$ 
1629        $\text{compile}(\text{context}\{x \mapsto \text{known}\})((\text{branches} - \text{eq})[x/e]),$ 
1630        $\text{SOME}(\text{compile context } (\text{branches} - \text{eq} - e)))$ 
1631     ,  $x$  fresh
1632   end
1633 where
1634    $\text{snd}(a, b) = b$ 
1635    $\text{refine}(x, K(ns)b) =$ 
1636   if  $K'(es) \in x@b \wedge (K \neq K' \vee \text{length}(es) \neq \text{length}(ns))$  then NONE
1637   else 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

$$\begin{aligned}
& \mathcal{E}[[x]] \rightsquigarrow x \\
& \mathcal{E}[[K\ e_1 \dots e_n]] \rightsquigarrow K\ \mathcal{E}[[e_1]] \dots \mathcal{E}[[e_n]] \\
& \mathcal{E}[[\lambda x. e]] \rightsquigarrow \lambda x. \mathcal{E}[[e]] \\
& \mathcal{E}[[e_1\ e_2]] \rightsquigarrow \mathcal{E}[[e_1]]\ \mathcal{E}[[e_2]] \\
& \mathcal{E}[[\text{case } e\ p_1\ e_1 \mid \dots \mid p_n\ e_n]] \rightsquigarrow \\
& \quad \forall i. 1 \leq i \leq n : \\
& \quad \text{if } \exists x. x = \mathcal{E}[[e]]; \\
& \quad \text{let } (ns_1, gs_1) \dots (ns_i, gs_i) = \mathcal{P}[[p_1]]x \cdot \dots \cdot \mathcal{P}[[p_i]]x \text{ in} \\
& \quad \text{if } \exists ns_1. gs_1 \rightarrow \mathcal{E}[[e_1]]; \square \dots \square \exists ns_i. gs_i \rightarrow \mathcal{E}[[e_i]] \text{ fi} \\
& \quad \text{fi} \\
& \quad , x \text{ fresh}
\end{aligned}$$

5.3 Translating Patterns

$$\begin{aligned}
& \mathcal{P}[[y]]x \rightsquigarrow (y, [x = y]) \\
& \mathcal{P}[[K]]x \rightsquigarrow ([], [x = K]) \\
& \mathcal{P}[[K\ p_1 \dots p_n]]x \rightsquigarrow \\
& \quad \forall i. 1 \leq i \leq n : \\
& \quad \text{let } y_i \text{ be a fresh name,} \\
& \quad (ns_1, gs_1) \dots (ns_i, gs_i) = \mathcal{P}[[p_1]]y_1 \cdot \dots \cdot \mathcal{P}[[p_i]]y_i \\
& \quad \text{in} \\
& \quad (ns_1 \cdot \dots \cdot ns_i \cdot y_1 \dots y_i, x = K\ y_1 \dots y_i \cdot gs_1 \cdot \dots \cdot gs_i) \\
& \mathcal{P}[[\text{when } e]]x \rightsquigarrow ([], [\mathcal{E}[[e]]]) \\
& \mathcal{P}[[p_1, p_2]]x \rightsquigarrow \text{let } (ns_1, gs_1) = \mathcal{P}[[p]]x, (ns_2, gs_2) = \mathcal{P}[[p']]x \text{ in } (ns_1 \cdot ns_2, gs_1 \cdot gs_2) \\
& \mathcal{P}[[p_1 \mid p_2]]x \rightsquigarrow \text{let } (ns_1, gs_1) = \mathcal{P}[[p]]x, (ns_2, gs_2) = \mathcal{P}[[p']]x \text{ in } (ns_1 \cdot ns_2, [gs_1 \mid gs_2])
\end{aligned}$$

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 $(\mathcal{D} \circ \mathcal{E})$.

Finally, I claim that the translation from *case* expressions to decision trees, $(\mathcal{D} \circ \mathcal{E})$, 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. Maranget’s formalism was the foundation off of which I built D .

Extensions to pattern matching, and how they appeal to language designers, find an excellent example in Erwig & 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 eliminate 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.

1846 From the beginning, Milod provided me with encouraging mentorship that kept me
1847 enthusiastic and determined to complete the project. He was exceptionally patient as I
1848 gave him whirlwind tour after whirlwind tour of the changing codebase and problems, and
1849 kept me grounded in the problems at hand. He sent me helpful examples of his research
1850 to aid me in my proofs, and gave me some particularly encouraging words towards the
1851 end of the project that I will not soon forget.

1853 My undergraduate advisor, Mark Sheldon, has always been both supportive and kind.
1854 I have enjoyed many long talks in his office, and I am deeply grateful that he is on my
1855 committee.

1857 Alva Couch was the advisor of my original thesis idea, which was to compile program-
1858 ming languages with music. Ultimately, we decided that I should pursue this project
1859 instead, and I am grateful to him for his mentorship in that moment and onwards.

1861 My family— my mother, Jennifer Burton, my father, Aniruddh Patel, and my sister, Lilia
1862 Burtonpatel, have all given me support, encouragement, and (arguably most importantly)
1863 food. My gratitude for them is immeasurable.

1864 My gratitude towards my friends is also without limit. In particular, and in no order,
1865 Liam Strand, Annika Tanner, Max Stein, Cecelia Crumlish, and Charlie Bohnsack all
1866 showed specific interest in the work and encouraged me as I moved forward. Rachael
1867 Clawson was subjected to much rubber-ducking, and endured valiantly. Aliénor Rice and
1868 Marie Kazibwe were as steadfast thesis buddies as I could ever hope for. Jasper Geer, my
1870 PL partner in crime, was always one of my favorite people to talk to about my thesis. His
1871 pursuits in research inspire mine.

1873 Skylar Gilfeather, my unbelievable friend. Yours is support that goes beyond words;
1874 care, food, silent and spoken friendship, late night rides to anywhere, laughs and tears are
1875 some that can try to capture it. I am so, so grateful for how ceaselessly you've encouraged
1876 me on this journey. Every one of your friends is lucky to have you in their life, and I am
1877 blessed that you are such a core part of mine.

1879 Anna Quirós, I am writing these words as you sleep behind me. Your support and love
1880 have been immeasurable. I will always be grateful to you for this year *incréible*. I could
1881 not have smiled through it all without you.

1883 Thank you all.

1884
1885
1886

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 Núñez. 1996. A new look at pattern matching in abstract data types. *ACM SIGPLAN Notices* 31, 6 (1996), 110–121.

- 1928 Norman Ramsey. 2022. *Programming Languages: Build, Prove, and Compare*. Cambridge University Press.
- 1929 Kevin Scott and Norman Ramsey. 2000. When do match-compilation heuristics matter. *University of Virginia, Charlottesville, VA* (2000).
- 1930 Don Syme, Luke Hoban, Tao Liu, Dmitry Lomov, James Margetson, Brian McNamara, Joe Pamer, Penny
- 1931 Orwick, Daniel Quirk, Chris Smith, et al. 2010. The F# 2.0 language specification. *Microsoft, August*
- 1932 (2010).
- 1933 The Elixir Team. [n. d.]. *Elixir Documentation*. Hexdocs.pm. <https://hexdocs.pm/elixir/index.html>
- 1934 Philip Wadler. 1985. How to replace failure by a list of successes a method for exception handling, back-
- 1935 tracking, and pattern matching in lazy functional languages. In *Conference on Functional Programming*
- 1936 *Languages and Computer Architecture*. Springer, 113–128.
- 1937 Philip Wadler. 1987. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of*
- 1938 *the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 307–313.
- 1939
- 1940
- 1941

1942 A PROOFS

1943 • **Proof: Translation from V^- to D preserves semantics**

1944 PROOF. In progress.

□

1947 • **Proof: Translation from P^+ to V^- preserves semantics**

1948 PROOF. In progress.

□

1951 B DISCUSSION

1952 Both backtracking and multiple results often manifest within \mathcal{VC} 's choice operator, which

1953 tempt its removal from V^- altogether. However, I am drawn to harnessing the expressive

1954 potential of this operator, particularly when paired with \mathcal{VC} 's 'one' keyword. When

1955 employed with choice as a condition, 'one' elegantly signifies 'proceed if any branch of

1956 the choice succeeds.

1957 To this end, in V^- , choice is permitted with several modifications:

- 1960 (1) Choice may only appear as a condition or 'guard', not as a result or the right-hand
- 1961 side of a binding.
- 1962 (2) If any branch of the choice succeeds, the choice succeeds, producing any bindings
- 1963 found in that branch. The program examines the branches in a left-to-right order.
- 1964 (3) The existential \exists may not appear under choice.
- 1965
- 1966
- 1967
- 1968

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:

2010

2011

2012

$$K(\text{when } e) p2 \dots \equiv K _ p2 \dots, \text{when } e \quad (1)$$

2013

2014

$$K(\text{when } e, p2) p3 \dots \equiv K p2 p3 \dots, \text{when } e \quad (2)$$

2015

2016

$$K(p1, \text{when } e) p3 \dots \equiv K p1 p3 \dots, \text{when } e \quad (3)$$

2017

$$K(\text{when } e \mid p2) p3 \dots \equiv (K _ p3 \dots, \text{when } e) \mid (K p2 p3 \dots) \quad (4)$$

2018

2019

$$K(p1 \mid \text{when } e) p3 \dots \equiv (K p2 p3 \dots) \mid (K _ p3 \dots, \text{when } e) \quad (5)$$

2020

$$\text{when } e \leftarrow e \equiv _ < -e, \text{when } e \quad (6)$$

2021

2022

2023

2024

2025

2026

2027

2028

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.

2029

2030

2031

2032

2033

2034

2035

2036

2037

2038

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.

2039

2040

2041

2042

2043

2044

2045

2046

2047

2048

2049

2050

E FORMAL DEFINITIONS OF ALL LANGUAGES

E.0.1 Judgement forms for V^- .

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

$$\langle \rho, p, v \rangle \mapsto \rho' \text{ (MATCH-SUCCESS)}$$

$$\langle \rho, p, v \rangle \mapsto \dagger \text{ (MATCH-FAIL)}$$

E.0.2 Evaluating General Expressions.

$$\text{(EVAL-VCONEMPTY)} \frac{}{\langle \rho, K \rangle \Downarrow K}$$

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\ \{\square G\}\]\ fi$	V^-
	t	D
v : Values	$K\{v\}$	
	$\lambda x. e$	
p : Patterns	x	P^+
	$-$	P^+
	$\bar{K}\ \{p\}$	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**

$$(EVAL-VCONMULTI) \frac{\langle \rho, e_i \rangle \Downarrow v_i \quad 1 \leq i \leq n}{\langle \rho, K(e_1, \dots, e_n) \rangle \Downarrow K(v_1, \dots, v_i)}$$

2092		
2093		$x \in \text{dom } \rho$
2094		$\rho(x) = v$
2095	(EVAL-NAME)	$\frac{\rho(x) = v}{\langle \rho, x \rangle \Downarrow v}$
2096		
2097		
2098		
2099	(EVAL-LAMBDADECL)	$\frac{}{\langle \rho, \lambda x. e \rangle \Downarrow \lambda x. e}$
2100		
2101		
2102		$\langle \rho, e_1 \rangle \Downarrow \lambda x. e$
2103		$\langle \rho, e_2 \rangle \Downarrow v'$
2104		$\langle \rho\{x \mapsto v'\}, e \rangle \Downarrow r$
2105	(EVAL-FUNAPP)	$\frac{\langle \rho\{x \mapsto v'\}, e \rangle \Downarrow r}{\langle \rho, e_1 \ e_2 \rangle \Downarrow r}$
2106		
2107		
2108		
2109	(EVAL-LITERAL)	$\frac{}{\langle \rho, v \rangle \Downarrow v}$
2110		
2111		
2112		
2113	(EVAL-CASESCRUTINEE)	$\frac{\langle \rho, e \rangle \Downarrow v \quad \langle \rho, \text{case } v \ [p_1 \ e_1], \dots, [p_n \ e_n] \rangle \Downarrow v'}{\langle \rho, \text{case } e \ [p_1 \ e_1], \dots, [p_n \ e_n] \rangle \Downarrow v'}$
2114		
2115		
2116		
2117	(EVAL-CASEMATCH)	$\frac{\langle \rho, p_1, v \rangle \mapsto \rho' \quad \langle \rho + \rho', e_1 \rangle \Downarrow v}{\langle \rho, \text{case } v \ [p_1 \ e_1], \dots, [p_n \ e_n] \rangle \Downarrow v'}$
2118		
2119		
2120		
2121	(EVAL-CASEFAIL)	$\frac{\langle \rho, p_1, v \rangle \mapsto \dagger \quad \langle \rho, \text{case } v \ [p_2 \ e_2], \dots, [p_n \ e_n] \rangle \Downarrow v'}{\langle \rho, \text{case } v \ [p_1 \ e_1], \dots, [p_n \ e_n] \rangle \Downarrow v'}$
2122		
2123		
2124		
2125	<i>E.0.3 Rules for pattern matching.</i>	
2126		
2127		$\langle \rho, p_i, v_i \rangle \mapsto s_i, \quad 1 \leq i \leq m$
2128		$s = s_1 \uplus \dots \uplus s_m$
2129	(PAT-MATCHVCON)	$\frac{s = s_1 \uplus \dots \uplus s_m}{\langle \rho, K[p_1 \dots p_m], K[v'_1 \dots v'_m] \rangle \mapsto s}$
2130		
2131		
2132		

$$\text{(PAT-FAILVCON)} \frac{v \text{ does not have the form } K[v'_1, \dots, v'_m]}{\langle \rho, K \ p_1, \dots, p_m, v \rangle \rightsquigarrow \dagger}$$

$$\text{(PAT-MATCHBAREVCON)} \frac{}{\langle \rho, K, K \rangle \rightsquigarrow \{\}}$$

$$\text{(PAT-FAILBAREVCON)} \frac{v \neq K}{\langle \rho, K, v \rangle \rightsquigarrow \dagger}$$

$$\text{(PAT-MATCHVAR)} \frac{}{\langle \rho, x, v \rangle \rightsquigarrow \{x \mapsto v\}}$$

$$\text{(PAT-MATCHWHEN)} \frac{\langle \rho, e \rangle \Downarrow v' \quad v' \neq \text{false}}{\langle \rho, \text{when } e, v \rangle \rightsquigarrow \{\}}$$

$$\text{(PAT-MATCHWILDCARD)} \frac{}{\langle \rho, v \rangle \rightsquigarrow \{\}}$$

$$\text{(PAT-FAILWHEN)} \frac{\langle \rho, e \rangle \Downarrow v' \quad v' = \text{false}}{\langle \rho, \text{when } e, v \rangle \rightsquigarrow \dagger}$$

$$\text{(PAT-ARROWEXP)} \frac{\langle \rho, e \rangle \Downarrow v' \quad \langle \rho, p, v' \rangle \rightsquigarrow s}{\langle \rho, p \leftarrow e, v \rangle \rightsquigarrow s}$$

$$\text{(PAT-MULTIFAIL)} \frac{\langle \rho, p, v \rangle \rightsquigarrow \dagger}{\langle \rho, p_1, p_2, v \rangle \rightsquigarrow \dagger}$$

$$\begin{array}{c}
2174 \\
2175 \\
2176 \\
2177 \quad (\text{PAT-MULTISOLUTION}) \quad \frac{\langle \rho, p_1, v \rangle \multimap \rho' \quad \langle \rho \uplus \rho', p_2, v \rangle \multimap s}{\langle \rho, p_1, p_2, v \rangle \multimap s} \\
2178 \\
2179
\end{array}$$

$$\begin{array}{c}
2180 \\
2181 \quad (\text{PAT-ORFST}) \quad \frac{\langle \rho, p_1, v \rangle \multimap \rho'}{\langle \rho, p_1 \mid p_2, v \rangle \multimap \rho'} \\
2182 \\
2183
\end{array}$$

$$\begin{array}{c}
2184 \\
2185 \quad (\text{PAT-ORSND}) \quad \frac{\langle \rho, p_1, v \rangle \multimap \dagger \quad \langle \rho, p_2, v \rangle \multimap s}{\langle \rho, p_1 \mid p_2, v \rangle \multimap s} \\
2186 \\
2187
\end{array}$$

2188
2189 *E.0.4 Judgement forms for V^- .*

$$\begin{array}{c}
2190 \quad \langle \rho, e \rangle \Downarrow r \text{ (EVAL)} \\
2191 \\
2192
\end{array}$$

$$\begin{array}{c}
2193 \quad \rho \vdash g \multimap \rho' \text{ (GUARD-REFINE)} \\
2194 \\
2195
\end{array}$$

$$\begin{array}{c}
2196 \quad \rho \vdash g \multimap \dagger \text{ (GUARD-REJECT)} \\
2197
\end{array}$$

2198 *E.0.5 Shifting a guard to the context.*

$$\begin{array}{c}
2199 \\
2200 \quad (\text{MOVE-GUARD-TO-CTX}) \quad \frac{\rho; g \cdot \mathcal{T} \vdash gs \cdot gs' \multimap s}{\rho; \mathcal{T} \vdash gs \cdot g \cdot gs' \multimap s} \\
2201 \\
2202 \\
2203
\end{array}$$

2204 *E.0.6 Choosing and processing a guard.*

$$\begin{array}{c}
2205 \\
2206 \quad (\text{SOLVE-GUARD-REFINE}) \quad \frac{\rho \vdash g \multimap \rho' \quad \rho'; \mathcal{T} \cdot \mathcal{T}' \vdash gs \multimap s}{\rho; \mathcal{T} \cdot g \cdot \mathcal{T}' \vdash gs \multimap s} \\
2207 \\
2208 \\
2209
\end{array}$$

$$\begin{array}{c}
2210 \\
2211 \quad (\text{SOLVE-GUARD-REJECT}) \quad \frac{\rho \vdash g \multimap \dagger}{\rho; \mathcal{T} \cdot g \cdot \mathcal{T}' \vdash gs \multimap \dagger} \\
2212 \\
2213 \\
2214
\end{array}$$

2215 *E.0.7 Properties of guards.*

2216

$$2217 \quad (\text{MULTI-GUARD-ASSOC}) \quad \frac{\rho; \mathcal{T} \cdot g_1 \cdot g_2 \cdot \mathcal{T}' \vdash gs \rightarrow s}{\rho; \mathcal{T} \cdot g_2 \cdot g_1 \cdot \mathcal{T}' \vdash gs \rightarrow s}$$

2219

2220

2221

2222 *E.0.8 Refinement with different types of guards.*

2223

2224

2225

2226

2227

2228

2229

2230

2231

2232

2233

2234

2235

2236

2237

2238

2239

2240

2241

2242

2243

2244

2245

2246

2247

2248

2249

2250

2251

2252

2253

2254

2255

$$(\text{GUARD-NAMEEXP-BOT}) \quad \frac{\begin{array}{c} x \in \text{dom } \rho \\ \langle \rho, e \rangle \Downarrow v \\ \rho(x) = \perp \end{array}}{\rho \vdash x = e \rightarrow \rho\{x \mapsto v\}}$$

$$(\text{GUARD-NAMEEXP-EQ}) \quad \frac{\begin{array}{c} x \in \text{dom } \rho \\ \langle \rho, e \rangle \Downarrow v \\ \rho(x) = v \end{array}}{\rho \vdash x = e \rightarrow \rho}$$

$$(\text{GUARD-NAMEEXP-FAIL}) \quad \frac{\begin{array}{c} x \in \text{dom } \rho \\ \langle \rho, e \rangle \Downarrow v \\ \rho(x) = v' \\ v \neq v' \end{array}}{\rho \vdash x = e \rightarrow \dagger}$$

$$(\text{GUARD-NAMES-BOT-SUCC}) \quad \frac{\begin{array}{c} x, y \in \text{dom } \rho \\ \rho(x) = \perp, \rho(y) = v \end{array}}{\rho \vdash x = y \rightarrow \rho\{x \mapsto v\}}$$

$$(\text{GUARD-NAMES-BOT-SUCC-REV}) \quad \frac{\begin{array}{c} x, y \in \text{dom } \rho \\ \rho(x) = v, \rho(y) = \perp \end{array}}{\rho \vdash x = y \rightarrow \rho\{y \mapsto v\}}$$

2256

2257

2258

2259

2260

2261

2262

2263

2264

2265

2266

2267

2268

2269

2270

2271

2272

2273

2274

2275

2276

2277

2278

2279

2280

2281

2282

2283

2284

2285

2286

2287

2288

2289

2290

2291

2292

2293

2294

2295

2296

$$\begin{array}{c}
x \in \text{dom } \rho \\
\rho x = v \\
\text{(GUARD-VCON-FAIL)} \frac{v \text{ does not have the form } K[v'_1, \dots, v'_m]}{\rho \vdash x = K \ e_1, \dots, e_m \rightsquigarrow \dagger} \\
\\
\text{(GUARD-EXPSEQ-SUCC)} \frac{\langle \rho, e \rangle \Downarrow v}{\rho \vdash e \rightsquigarrow \{\}} \\
\\
\text{(GUARD-EXPSEQ-FAIL)} \frac{\langle \rho, e \rangle \Downarrow \mathbf{fail}}{\rho \vdash e \rightsquigarrow \dagger} \\
\\
\text{(GUARD-CHOICE-FIRST)} \frac{\rho; \varepsilon \vdash gs \rightsquigarrow \rho'}{\rho \vdash gs \mid gs' \rightsquigarrow \rho'} \\
\\
\text{(GUARD-CHOICE-SECOND)} \frac{\rho; \varepsilon \vdash gs \rightsquigarrow \dagger \quad \rho; \varepsilon \vdash gs' \rightsquigarrow s}{\rho \vdash gs \mid gs' \rightsquigarrow s}
\end{array}$$

E.0.9 Evaluating General Expressions.

2285

2286

2287

2288

2289

2290

2291

2292

2293

2294

2295

2296

$$\begin{array}{c}
\text{(EVAL-IFFI-FAIL)} \frac{}{\langle \rho, \text{IF } [\] \text{ FI} \rangle \Downarrow \mathbf{fail}} \\
\\
\rho' = \rho \{x_1 \mapsto \perp \dots x_n \mapsto \perp\} \\
\rho'; \varepsilon \vdash gs \rightsquigarrow \rho'' \\
\langle \rho'', e \rangle \Downarrow r \\
\text{(EVAL-IFFI-SUCCESS)} \frac{}{\langle \rho, \text{IF } [\exists x_1 \dots x_n. gs \rightarrow e \square \dots] \text{ FI} \rangle \Downarrow r}
\end{array}$$

$$\begin{array}{c}
2297 \\
2298 \quad \rho' = \rho\{x_1 \mapsto \perp \dots x_n \mapsto \perp\} \\
2299 \quad \rho'; \varepsilon \vdash gs \mapsto \dagger \\
2300 \quad \langle \rho, \text{IF } [\dots] \text{ FI} \rangle \Downarrow r \\
2301 \quad (\text{EVAL-IFFI-REJECT}) \frac{}{\langle \rho, \text{IF } [\exists x_1 \dots x_n. gs \rightarrow e \square \dots] \text{ FI} \rangle \Downarrow r} \\
2302 \\
2303
\end{array}$$

$$\begin{array}{c}
2304 \\
2305 \quad (\text{EVAL-VCONEMPTY}) \frac{}{\langle \rho, K \rangle \Downarrow K} \\
2306 \\
2307 \\
2308 \quad (\text{EVAL-VCONMULTI}) \frac{\langle \rho, e_i \rangle \Downarrow v_i \quad 1 \leq i \leq n}{\langle \rho, K(e_1, \dots e_n) \rangle \Downarrow K(v_1, \dots v_i)} \\
2309 \\
2310 \\
2311
\end{array}$$

$$\begin{array}{c}
2312 \\
2313 \quad (\text{EVAL-VCONMULTI-FAIL}) \frac{\exists e_i. 1 \leq i \leq n : \langle \rho, e_i \rangle \Downarrow \mathbf{fail}}{\langle \rho, K(e_1, \dots e_n) \rangle \Downarrow \mathbf{fail}} \\
2314 \\
2315
\end{array}$$

$$\begin{array}{c}
2316 \\
2317 \quad x \in \text{dom } \rho \\
2318 \quad \rho(x) = v \\
2319 \quad (\text{EVAL-NAME}) \frac{}{\langle \rho, x \rangle \Downarrow v} \\
2320
\end{array}$$

$$\begin{array}{c}
2321 \\
2322 \quad (\text{EVAL-NAME-FAIL}) \frac{x \notin \text{dom } \rho}{\langle \rho, x \rangle \Downarrow \mathbf{fail}} \\
2323 \\
2324
\end{array}$$

$$\begin{array}{c}
2325 \\
2326 \quad (\text{EVAL-LAMBDADECL}) \frac{}{\langle \rho, \lambda x. e \rangle \Downarrow \lambda x. e} \\
2327 \\
2328
\end{array}$$

$$\begin{array}{c}
2329 \\
2330 \quad \langle \rho, e_1 \rangle \Downarrow \lambda x. e \\
2331 \quad \langle \rho, e_2 \rangle \Downarrow v' \\
2332 \quad \langle \rho\{x \mapsto v'\}, e \rangle \Downarrow r \\
2333 \quad (\text{EVAL-FUNAPP}) \frac{}{\langle \rho, e_1 \ e_2 \rangle \Downarrow r} \\
2334 \\
2335
\end{array}$$

2337

2338

2339

2340

2341

2342

2343

2344

2345

2346

2347

2348

2349

2350

2351

2352

2353

2354

2355

2356

2357

2358

2359

2360

2361

2362

2363

2364

2365

2366

2367

2368

2369

2370

2371

2372

2373

2374

2375

2376

2377

2378

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