PatBang

Flexible type-safe pattern matching

Pottayil Harisanker Menon Zachary Palmer Alexander Rozenshteyn Scott Smith
The Johns Hopkins University
{pharisa2, zachary.palmer, scott, arozens1}@jhu.edu

Abstract

Patterns provide an important dimension of expressiveness to functional programming languages because they describe a concise syntax for data destruction. However, most languages treat patterns as second-class citizens: case match expressions cannot be extended, patterns cannot be selected dynamically, and patterns cannot be composed or modified by program logic. In this paper, we present a rich, full-featured pattern language, PatBang, which treats patterns as first-class data, and additionally supports highly expressive patterns including recursive and disjunctive patterns, yet is still provably type-safe. We additionally show how such an expressive pattern language enables new programming patterns, including use of patterns for expressing *pytes*, a form of lightweight static interface declaration. Type soundness is proven and an implementation of the type inference algorithm and interpreter is provided.

1. Introduction

Patterns provide an important dimension of expressiveness to functional programming languages because they describe a concise syntax for data destruction. Pattern clauses such as `Tree(data,left,right) ->` ... are easy to read; they make immediately evident both the structure of the argument as well as the values to which the `data`, `left`, and `right` variables are bound. This is even more valuable when matching deep structure such as in `Node(Some(n),Empty,Empty) ->` .... Pattern matching also gives a terse syntax for exhaustively destructing on disjunctive data, e.g.

```
match t with Empty -> ...
           | Tree(data,left,right) -> ...
```

However, most languages treat patterns as second-class citizens: match expressions such as the above cannot be extended to work with wider data types; patterns cannot be selected dynamically; and patterns cannot be composed or modified by program logic.

In this paper, we present a rich, full-featured pattern semantics with the goal of writing even more concise and expressive code. Our presentation language, PatBang, supports these pattern semantics along with flexible data composition operations. Despite this flexibility, PatBang is statically typed via structural subtyping principles and types are fully inferred – no type annotations are necessary. The following is a brief outline of the kinds of patterns that PatBang supports.

Recursive patterns for shape invariant assertions While patterns can often be deep, standard pattern match syntax puts a fixed depth on pattern structure. But consider a pattern which can describe inductive invariants on data. If such a pattern is statically typechecked, it amounts to a compile-time shape assertion on the structure of the data. Consider the following ML-like pseudo-syntax:

```
match d with (rec l. Cons(Good(y),l) | Nil) -> ()
```

Informally, the `rec l` is defining a recursive pattern which constrains the simple list data to only contain `Good`-wrapped elements. The PatBang pattern grammar includes such recursive patterns and ensures that they are typechecked appropriately. Previous work [9] typechecks recursive patterns in the context of an ML-style type system, which requires the pattern to match a substructure of a fixed, declared data type. Other previous work [22] supports recursive patterns with subtyping but requires type declarations and lacks polymorphism. We aim to support recursive patterns over highly flexible data without need for declared data types.

First-class patterns for dynamic pattern composition While first-class functions are now universally accepted as a useful programming language feature, patterns remain fixed syntactic constructs rather than values. This prevents patterns from being constructed or chosen at runtime and limits how they can be reused. Composable, first-class patterns are a natural dual to first-class functions and have not been widely explored.

We design PatBang to include first-class patterns and define the type system to conservatively approximate pattern flow. We also define semantics that allow patterns to be composed, placing one pattern somewhere within another pattern's structure. This allows patterns to describe e.g. lists of pairs or balanced binary trees of lists through the use of generic, reusable pattern values. This also allows patterns to serve as data destruction strategies and, with sufficient polymorphism, witnesses to dynamic shape assertions. Some interesting forms of pattern abstraction were explored in [13] which allows for elegant generic programming patterns over structures; we aim to support patterns as fully first-class entities.

Powerfully typed patterns with a concise formalization We formalize PatBang as an extension of the TinyBang language [16], which uses a polymorphic subtype constraint inference type system [2, 12, 19, 23]. While the aforementioned novel pattern concepts are fundamental and could be realized in many forms of type system, PatBang's pattern semantics and TinyBang's type system benefit considerably from each other. The pattern semantics can be typechecked precisely due to the expressiveness of constraint-based typing and, because the patterns are powerful enough to describe static shapes, they can serve as a lightweight form of type assertion in PatBang.

Synergy of recursive and first-class patterns Our ultimate goal is not simply to extend patterns for their own sake; it is to produce a more powerful notion of pattern and pattern matching that can lead to a more expressive and useful programming language design. In the following section, we give numerous examples showing how the combination of features in PatBang leads to improved programmability. One particularly useful example we show is *pytes*, a novel form of type filter that can statically verify the shape of recursive data.

Validation The supplementary material supplied with this submission contains a complete proof of type soundness and an argument for the decidability of the type inference algorithm. It also contains an implementation of the PatBang language with a set of unit tests.

Paper outline The remainder of this paper proceeds as follows. Section 2 gives an overview of the semantics and expressiveness of PatBang by code example. Section 3 provides an operational semantics for PatBang, and the type system is presented in Section 4. Section 5 summarizes the work and draws conclusions.

2. Overview

The PatBang language extends TinyBang [16] to support patterns which are first-class, higher-order, recursive, and composable. We use these terms to describe different properties of these patterns. First-class indicates that patterns are values and so can be passed to functions. Higher-order indicates that the patterns interact; that is, patterns can be passed to *other patterns*.

The TinyBang language on which PatBang is based is designed as a compilation target for a scripting language. This motivation inspires a number of design decisions. First, TinyBang has no type declarations or type signatures; all types in TinyBang are inferred. Second, all data (including higher-order functions) can be arbitrarily conjoined, generalizing record concatenation. Third, case matches are heterogeneously typed: the output type of a given match is weakly dependent upon the match argument (up to the expressiveness of the pattern grammar).

To achieve these goals, TinyBang's type system uses a subtype constraint-based type system with several novel modifications. General data conjunction is asymmetric to allow overriding. And polymorphism is per call site rather than per binding use (as in `let`-bound polymorphism).

TinyBang's type system is quite powerful and already admits quite expressive pattern semantics; we describe them in Section 2.1. Building on top of this type system, PatBang is able to statically typecheck far more advanced patterns. The remainder of this section shows how these patterns can be used to test deep structural subtyping properties, describe compositions of data structures, and even serve a purpose similar to type signatures or contracts.

2.1 Extensible Cases

The pattern semantics we take from TinyBang already permit first-class, extensible case constructs; we summarize the ideas taken from TinyBang in this subsection. In most languages, a simple expression which extracts the contents of a variant data type might be written:

```
case arg of
  | ‘Dollars n -> n
  | ‘Euros n -> n
```

This expression extracts the units from a currency value. PatBang uses the syntax `‘Dollars` to describe a label: a single-argument data constructor similar to OCaml's polymorphic variants. The PatBang type system therefore requires that `arg` be either a `‘Dollars`- or `‘Euros`-labeled value.

But this traditional form of case expression is quite monolithic: it cannot be extended to support more forms of currency. In PatBang, this expression is syntactic sugar for the following:

```
def cases = ( (‘Dollars n -> n)
            & (‘Euros n -> n)   )
in cases arg
```

In the desugared form, each case branch is a distinct, function-like value called a *scape*. The `‘Dollars n -> n` scape, for instance, matches any `‘Dollars`-labeled argument and returns the contents of the label. The operator `&` is used to conjoin two scape values into a larger scape value which accepts any argument that either scape can accept. Finally, the expression `cases arg` applies this conjunction of scapes to the argument.

Scapes are distinct from functions in that the entire structure of the pattern is part of the scape's type. This is part of what distinguishes them from previous work on extensible cases [5]. That work permits cases to be extended, but only in the direction of lower priority and only by one case at a time. PatBang's scapes, on the other hand, can be used to encode behavior such as method override and can be arbitrarily concatenated.

Because scapes are first-class data and can be arbitrarily conjoined with other scapes, it is now possible to extend our case match with additional cases matching other currency patterns. For instance, one can now write

```
def cases' = cases & (‘Wampum n -> n) in cases' arg
```

in order to define a `cases'` which handles a new form of currency. In fact, the extension may itself be an arbitrary value (e.g. `cases & moreCases`). Because scapes are general and because patterns are permitted to match the outermost argument (e.g. `x -> x`), scapes supercede functions in PatBang.

Because the type of a scape includes its pattern, a scape conjunction can correctly compute the pattern describing the argument types it will accept. This ensures that type errors on application can be detected despite free-form scape conjunction. For instance, `cases ‘Dirt 3` would raise a type error because the type of `cases` is "$(‘\texttt{Dollars}\ \alpha_1 \to \alpha_1)$ `&` $(‘\texttt{Euros}\ \alpha_2 \to \alpha_2)$" and neither pattern in this type matches the `‘Dirt` label. This type for `cases` is also critical because it permits the heterogeneous typing of case branches described above; the mapping from input pattern to output type is retained for use at call sites, allowing the types of arguments to refine the types of the results.

As stated above, PatBang's data model uses structural subtyping. This means that the pattern `‘Dollars n` matches any datum which *includes* a `‘Dollars`-labeled value; it is possible for other data to be present as well. For instance,

```
cases (‘Dollars 4 & ‘Commission 0.07)
```

evaluates to `4`. Here, the `&` operator is being used to conjoin two labeled data together; in PatBang, this operator conjoins any data into a value known as an *onion*. An onion is a form of *type-indexed* record; the above argument can be equivalently viewed as the record `{Dollars = 4, Commission = 0.07}`. Type indexing means that scapes may also be present in the onion without being under a particular label name; in a sense, a special `scapes` label exists which contains all of the scapes that have been onioned into the record (so informally `5 & (v -> v) & (‘A x -> x)` is the record `{int = 5, scapes = [v -> v, {A = x} -> x]}`). Everything is an onion in PatBang, even primitive values such as `5` (it is informally the record `{int = 5}`); this perspective helps to simply PatBang's semantics and type system.

Because PatBang's type system is constraint-based, variant types are expressed as a type variable with multiple labeled lower bounds: a variant type with constructors `‘Foo` and `‘Bar` would have the type $\alpha_3 \setminus \{‘\texttt{Foo}\ \alpha_1 <: \alpha_3, ‘\texttt{Bar}\ \alpha_2 <: \alpha_3\}$. It is well-known that positive union types can be constructed in constraint-based type systems using this approach. One consequence of this is that unary variant types and unary record types are equivalent in PatBang.

2.2 Recursive Patterns

PatBang programs do not require type declarations or signatures; data can be composed arbitrarily and labels do not restrict their

contents. In contrast to more traditional systems, this means that matching a `‘Hd x & ‘Tl _` value is insufficient for establishing that a datum is a cons-cell style list: there is no guarantee that the contents of the `‘Tl` will have similar structure. For this reason (among others), it is useful to have a *recursive pattern*: a form of pattern which recursively constrains a data type. For instance, consider

```
def isList =
  ( (_ -> false)
  & (rec p: ‘Nil _ | (‘Hd _ & ‘Tl p) -> true) ) in ...
```

(Recall that the *rightmost* scape is given priority in application, hence the catch-all being on the left/top of the onion.) This scape includes pattern operators for pattern conjunction and disjunction: the `&` pattern matches only if both of its argument patterns match, while the `|` pattern matches whenever either of its arguments match. The recursive pattern above matches only data which has a list structure recursively under the `‘Tl` label. While it is still possible for other data to be present (e.g. `‘Hd 4 & ‘Tl (‘Nil ()) & ‘Note "clean"`), the pattern represents a least guarantee. This pattern can be refined as usual; that is

```
rec p: ‘Nil _ | (‘Hd ‘Dollars _ & ‘Tl p) -> true
```

will match only lists containing `‘Dollars`-labeled values; lists containing a non-`‘Dollars` element and values which are not lists will not match.

A naïve pattern matching algorithm at runtime simply descends into each level of the argument, expanding the recursive part of the pattern as it goes. Because PatBang is an eager, stateless language, all runtime data are finite and so matching a well-formed pattern (e.g. from `isList`) will terminate. It is possible however to do much better in practice: a sensible runtime could include a means by which data could be flagged as matching a given pattern. Information gathered by the type system can be used to determine which patterns a given datum will always (or never) match; the remaining cases would be handled by caching the results of the naïve match described above.

Our pattern grammar does introduce the subtle problem of ill-formed recursive patterns. Pattern matching would not terminate in the face of *non-contractive* patterns such as `rec p: p`. A non-contractive pattern is similar to a non-contractive recursive type [14]: it recurses without reaching an intervening constructor. Like non-contractive types, such patterns have no well-defined semantics; as a result, the PatBang type system prohibits them. Further discussion on detecting and prohibiting non-contractive patterns can be found in Section 4.2.4.

Well-defined recursive patterns are especially useful in matching deep, complex structure. Consider the following task: given a deep expression grammar (such as PatBang's) and a deep value grammar where value terms are also expression terms, determine whether a given expression is a value. For example, `‘A 3` is a value but `‘A (3 + 4)` is not. In particular, the nesting property is important; an expression is a value only if all of its subexpressions are values. The following scape solves this problem:

```
def isValue = (_ -> false) in ...
            & (rec p: ‘Int _ | ‘Label (‘Name string) p |
                      ‘Onion (‘L p & ‘R p) | ... -> true)
```

While a recursive function can solve this problem as well, recursive patterns are interesting for two reasons. First, all pattern matches are decidable (since non-contractive patterns are prohibited). Second, a pattern is more declarative; it indicates the author's intentions in analyzing the shape of data.

2.3 Patterns as Data

As mentioned above, patterns in PatBang are first-class data members. In fact, all scapes in PatBang are constructed from two data items – a pattern and a function – using the scape operator `><`. The syntax `‘A x & ‘B y -> x + y` we have been using up to now is in fact equivalent to the following:

```
def p = (x,y) <- ‘A x & ‘B y in
def f = (n,m) -> n + m in
p >< f
```

When the scape `p >< f` is applied to an argument (e.g. `‘A 9 & ‘B 5`), that argument is first matched against the pattern to yield bindings for the pattern's variables ($\texttt{x} \mapsto 9$, $\texttt{y} \mapsto 5$). These bindings are then aligned *positionally* to the function's parameters ($\texttt{n} \mapsto \texttt{x}$, $\texttt{m} \mapsto \texttt{y}$), yielding a final binding ($\texttt{n} \mapsto 9$, $\texttt{m} \mapsto 5$).

As a result of these semantics, patterns represent general strategies for destructing data. For instance, consider the following alternatives for the pattern `p` above:

```
def p = (x,y) <- ‘A y & ‘B x in ...
```

```
def p = (a,b,c) <- ‘A (a & b) & ‘B c in ...
```

```
def p = (y) <- ‘A y in ...
```

The first alternative binds the contents of `‘A` and `‘B` to opposite positions. This results in `n` taking on the value `5` and `m` taking on the value `9`, thus yielding `-4`. The second alternative binds both `a` and `b` to the contents of `‘A` and binds `c` to the contents of `‘B` but, since the function only consumes two arguments, `c` goes unused and the evaluation results in `0`. The last alternative will produce a type error at `p >< f` because the pattern does not bind enough variables to satisfy the requirements of the function.

2.4 Pattern Substitution

In addition to being first-class, patterns are also higher-order: they can be passed to other patterns to perform pattern composition. Because patterns describe shapes of data, such a pattern composition is the composition of these shapes. Consider the following patterns:

```
def list = (x) <- rec p: (‘Hd x & ‘Tl p) | ‘Nil _ in
def dlrs = () <- ‘Dollars _ in
def dlrsList = () <- list (dlrs ()) in ...
```

The `list` pattern generally describes the shape of lists. The `dlrsList` pattern uses *pattern substitution* to replace every occurrence of `x` in `list` with the pattern `dlrs`. The result is a pattern which will only match lists of elements which include `‘Dollars`-labeled data.

It should be noted that the variable `x` in the `list` pattern is not *fully* bound: in the case of an empty list, there would be no data for it to consume. It would therefore be a type error to pair the `list` pattern with any scape with a non-empty parameter list. But `x` still appears in the pattern's variable list so that substitutions can match pattern arguments with their substitution sites.

Because patterns are first-class data, pattern substitution also permits a programmer to write functionality that generalizes certain forms of patterns. For instance, consider the problem from Section 2.2 in which values are a subset of expressions. Consider the following:

```
def pats = [ (r) <- ‘Int _, (r) <- ‘Label (‘Name string) r
           , (r) <- ‘Onion (‘L r & ‘R r), ...] in
def rec disjunctPat =
      (‘Hd next & ‘Tl rest ->
        (x) <- next (x) | (disjunctPat rest) (x) )
    & (‘Nil _ -> (x) <- none) in
() <- rec p: (disjunctPat pats) p
```

This code creates a pattern similar to the one in the `isValue` scape: it matches only the values in a given expression grammar. The values it matches are defined by the list of `pats`. Each such pattern takes a single argument defining how it should recurse. The `disjunctPat` scape then creates a single-variable pattern which is the disjunction of all of these patterns; the special pattern `none` matches nothing and is therefore the base case of pattern disjunction. Finally, the pattern substitution of the recursion variable `p` ties the knot and allows each part of the disjunction to recurse back into the disjunction. In leaf cases such as `(r) <- 'Int _`, the recursive pattern is simply ignored. The key observation is that `pats` is an arbitrary list of patterns that could have been dynamically computed.

2.5 Pytes: Patterns as Type Signatures

PatBang has no type assertion syntax; all types are inferred. While this decision is motivated by the objectives of the language, it is also in part because patterns can be used in their stead: PatBang's patterns can fulfill the purpose that type signatures fulfill in many other languages. We use the term *pyte* to describe a pattern used in this way. A pyte is an identity scape restricted to a specific pattern; any value passed to the scape will be returned unchanged, but a type error will result if the value does not match the pyte's pattern. We use the syntactic sugar `:-` to denote application of a pyte. For example, `x :- 'A _` is equivalent to

```
def pyte = arg & 'A _ -> arg in pyte x
```

This code passes the value `x` through a scape which matches arguments of the form `'A _`. If the type of `x` has some form which does not match the pyte, a type error occurs; otherwise, the argument is returned unaffected. Dynamic pyte-checking can be expressed by the sugar `x :+ 'A _`, which de-sugars to

```
def dynPyteFail = (_ -> raise 'TypeError ()) in
def pyte = arg & 'A _ -> arg in
(dynPyteFail & pyte) x
```

Because `dynPyteFail` matches anything that `pyte` does not, this code will always typecheck. If the value fails the pattern match, an exception is raised at runtime.

In some sense, pytes are similar to contracts [10] in that they are programs that constrain values. Pytes, however, are statically verified because non-exhaustive matches are always treated as type errors. We further discuss how pytes relate to contracts below.

Because every pattern is a pyte, PatBang's pytes are capable of expressing recursive forms such as lists. In fact, the `list` pyte from Section 2.4 expresses a polymorphic list type. For instance, `y :- list` will (statically) verify that `y` is a list; `y :- dlrsList` will verify that `y` is a list of `'Dollars` values. (This is similar to e.g. Java's `List<T>`.)

While patterns are suitable for matching against first-order data, they are not suitable for constraining the types of scapes; patterns in PatBang cannot be precise about a scape's input type because it can be constrained in quite complex ways by the scape's body. For this reason, we also admit *arrow* pytes which restrict the input and output of a scape. Formally, we let pytes (ψ) range over patterns (φ) and arrow pytes (ψ `->` ψ). This too is syntactic sugar; a reasonable attempt at desugaring `scape :-` ψ_1 `->` ψ_2 is as follows:

```
arg -> (scape (arg :- ψ1)) :- ψ2
```

This code simply performs η-expansion on `scape`, and then annotates the input and output of that expansion with the pytes to the left and right of the arrow. Because non-exhaustive pattern matches are always a type error in PatBang, any program flow which does not conform to the expectations of the arrow pyte is *statically* detected. Pytes are similar to contracts in that this programmatic decoration only has an effect if the flow is actually used; `(x -> x + 1) :- char -> char` will not produce a type error unless it is called.

Programmatic type decoration on higher-order functions has a long history in programming languages, dating back to Dana Scott's retracts [20]. More recent work on higher-order contracts [10] has shown how dynamic constraints on a function domain and range can enforce assertions on higher-order functions at runtime. Higher-order contracts are weaker than pytes in that they are not static, and they are stronger in that dependent contracts may be expressed in analogy to dependent types, and conditions that could never be statically checked, such as being a member of a subrange of `int`, are expressible as dynamic checks. There have been several subsequent projects which add a layer of static verification for contracts [11, 17, 24], but the advantage of pytes is they come "for free" for us: no additional theorem prover, program analysis, or decision procedure (respectively) is needed. Additionally, none of these static contract frameworks support statically verifying recursive shape properties of data as is possible with recursive patterns used in a pyte.

Extracting patterns While this encoding seems correct at first glance, it is imperfect: the resulting scape matches *every* argument, meaning that `t & s` and `t & (s :-` ψ`)` may have different meanings. Fortunately, the *pattern-of* prefix operator `$` can solve this problem. When evaluated on a single scape, this operator retrieves that scape's pattern; that is, `$(p >< f)` is always `p`. When evaluated on an onion of scapes, the resulting pattern is the *disjunction* of the patterns of all of the scapes in the onion; that is, it is a pattern matching anything that the onion of scapes can match. This operator permits the implementation of lightweight proxy functions and other similar design patterns in the OO model described in [16].

Using the pattern-of operator, we can define a match-preserving η-conversion to use in the desugaring of the arrow pyte above:

```
arg & $scape -> (scape (arg :- ψ1)) :- ψ2
```

This desugaring allows arrow pytes to describe transparent interface specifications without affecting the semantics of the underlying value. Using the pattern-of operator, we can also intercept invocations in a proxy-like manner (e.g. `arg & $s -> log arg; s arg`); we can also improve upon the `seal` method used in the object model of [16] to allow the addition of default methods. Most importantly, this operator allows each of these applications to have match-preserving semantics; the full extensibility of scapes is unaffected in these encodings.

2.6 More Pattern Expressiveness

This subsection briefly discusses other forms of patterns which further expand upon the expressiveness of PatBang.

Match list variables The previous patterns which match deep data structures serve only as recognizers. Using *match list variables*, a class of pattern variables with different binding semantics, patterns are able to describe filtered iteration strategies. Typically, pattern variables observe the same asymmetric properties as onions: the rightmost binding is preferred. For instance, the pattern `(x) <- rec p: x & (int | 'A p)` will, for the argument `'A (3 & 'A 4)`, bind `x` to `4`; we prefer the rightmost bindings and so the deepest value is bound to `x`. But a match list variable (such as in `(*x) <- rec p: x & (int | 'A p)` would produce a `'Hd _ & 'Tl _` form of list containing all of the bindings it encounters: in this case, the list `[3,4]`. For a more complex example, consider:

```
def treeWalk = (*x) <- rec p: 'Leaf x | ('L p & 'R p) in
def red = (y) <- 'Red y in
def redWalk = (*z) <- treeWalk (red (z)) in
def redFilterWalk = (*z) <- treeWalk (_ | red (z)) in ...
```

The `treeWalk` pattern, if combined with a single-parameter function, would generate a depth-first walk of a binary tree, placing the list of leaf values in the variable `x`. The `redWalk` pattern uses pattern composition to generate such a walk but only over trees which exclusively contain `‘Red` leaves. The `redFilterWalk` pattern, on the other hand, matches every leaf but binds `z` to a list of the `‘Red`-labeled data encountered on the walk. In general, match list variables allow PatBang's patterns to tersely describe general strategies for processing the contents of data structures in a higher-order and composable fashion.

Patterns and data construction The `list` pattern shown in Section 2.2 is capable of recognizing lists but not binding its contents. In general, binding is only successful if both branches of a disjunct can bind a value and, in the case of empty lists, the `list` pattern cannot. This changes with the introduction of a `where` clause: a data construction clause which can be attached to a pattern to artificially construct a binding. For instance, consider the following pattern:

```
def listHead =
    (y) <- rec p: (‘Nil _ where y = ‘None ())
                | (‘Tl p & ‘Hd x where y = ‘Some x) in ...
```

The `listHead` pattern binds any list (including empty ones). It binds `y` to the head of the list when the list is not empty. In general, the contents of the `where` clause admits any expression; that expression is executed after the pattern match is considered successful. The `where` clause permits a pattern to act as an interpretation of a data structure similar to the `where` clause found in [9].

Label variables Every pattern shown thus far assumes that labels are first-order: the names of labels are statically known everywhere they are constructed or destructed. This view is taken as a default in PatBang to assist in efficient compilation. But it can be useful to process a data structure without prior knowledge of its entire shape. To accomplish this, PatBang uses *label variables*. Label variables ℓ are bound in a special form of pattern such as in the code below:

```
def rec s = (z -> z)
          & (n & int -> n + 1)
          & (ℓ x &! y -> ℓ (s x) & (s y)) in
s (‘A (‘B 7 & ‘C "yes") & ‘D 3)
```

(Recall that application prefers rightmost scapes; thus, the identity scape `(z -> z)` only applies if the other scapes do not.) The first step of evaluation in this code applies `s` to an onion of data, which matches the third scape. The pattern ℓ `x &!` `y` in that scape describes a *disjoint conjunction* which matches any onion with a label: the value `y` is bound to all of the argument *except* the part containing the label ℓ while `x` is bound to the contents of the label ℓ. In this case, we bind such that $\{$`x` $\mapsto$ `(‘B 7 & ‘C "yes")`, `y` $\mapsto$ `‘D 3`, $\ell \mapsto$ `‘A`$\}$. In the body of this branch, the code recurses on both `x` and `y`, reconstructing the onion once recursion is complete. When a recursion reaches an integer, it is incremented; other data is handled by the identity scape. As a result, evaluation yields the value `‘A (‘B 8 & ‘C "yes") & ‘D 4`.

This process is quite similar to the generic programming found in [13], although PatBang's `&!` pattern offers fewer guarantees on the order in which it disassembles the onion (i.e., ℓ is chosen arbitrarily from the labels in the onion). But PatBang's data structures use *label name* rather than *position* to describe construction. This permits PatBang to be more flexible regarding data composition; it also permits more precise invariants on the routine's recursion (e.g. don't recurse into `‘Salary` labels).

3. Operational Semantics

We now present the operational semantics for a restricted form of PatBang. This language does not include all of the features from Section 2; we leave out a number of pattern forms to reduce redundancy in the formalization. First, it does not include the expressiveness enhancements described in Section 2.6. Second, some simple patterns like `|` and `none` are elided; we discuss their formalism in Appendix C. Third, language features such as `if` and `def` are encoded (see Figure 3.1) to simplify the expression semantics. Encoding `def` is safe because it does not affect call-site polymorphism as it affects `let`-bound polymorphism. We can use an encoded `fix` because our constraint-based type system can express recursive types. Restricted PatBang does include all of the functionality of labels, onions, scapes, recursive patterns, and pattern substitution.

```
if e1 then e2 else e3  ≅  ( (‘True _ -> e1)
                          & (‘False _ -> e2)) e3
def x = e1 in e2       ≅  (x -> e2) e1
def rec x = e1 in e2   ≅  def x = fix (x -> e1) in e2
fix                    ≅  f -> (x -> f (v -> x x v))
                               (x -> f (v -> x x v))
```

Figure 3.1. Simplified PatBang Encodings

e	::=	$\overrightarrow{s}$	*expressions*
E	::=	$\overrightarrow{x = v}$	*environment*
s	::=	$x = v \mid x = x \mid x = x\ x \mid x = x \boxdot x$	*clauses*
v	::=	$\mathbb{Z} \mid$ `()` $\mid l\ x \mid x$ `&` $x \mid$	*values*
		$\overrightarrow{x}$ `->` $e \mid \overrightarrow{y}$ `<-` $\varphi \mid x$ `><` x	
l	::=	‘*(alphanumeric)*	*labels*
x, y	::=	*(alphanumeric)*	*value and pattern variables*
$\boxdot$	::=	`+` $\mid$ `==`	*integer operators*
φ	::=	`int` $\mid l\ \varphi \mid$ `fun` $\mid$ `pat` $\mid$ `scape` $\mid$	*pattern bodies*
		φ `&` $\varphi \mid x\ \overrightarrow{\varphi} \mid$ `rec` y `:` $\varphi \mid y$	
π	::=	`int` $\mid l \mid$ `fun` $\mid$ `pat` $\mid$ `scape`	*projectors*

Figure 3.2. PatBang ANF Language Grammar

Notation For a given grammatic construct $*$, we let $[*_1, \ldots, *_n]$ denote a list of $*$, often using the equivalent shorthand $\overrightarrow{{}^{n}*_{\square}}$; the "$_{\square}$" indicates which constructs are indexed, and the n and $_{\square}$ can be elided when they are obvious or not needed. Operator $\|$ denotes list concatenation. For sets, we use similar indexing notation: $\overbrace{{}^{n}*_{\square}}$ abbreviates $\{*_1, \ldots, *_n\}$ for some arbitrary ordering of the set.

3.1 PatBang Syntax

In addition to the encodings described in Figure 3.1, we also simplify our formalization by using the A-normal form grammar appearing in Figure 3.1. The A-translation from the nested grammar of Section 2 is fairly straightforward and so is elided. For instance, the nested expression `(s 2) (‘B n) -> n + 1` would A-translate to

```
x1 = 2
p1 = s x1
p2 = (n) <- p1 (‘B n)
f = (n) -> x2 = 1
          r = n + x2
s = p2 >< f
```

We use indentation to indicate the two clauses which are in the body of the function `f`. Note that each clause is a variable assignment to a redex. Finally, we also require a guarantee of well-formedness: a PatBang ANF expression is *well-formed* iff it is closed and each variable is defined no more than once. For the remainder of this document, we assume that expressions are well-formed.

3.2 Supporting Relations

We prepare to define the operational semantics of PatBang by specifying the following supporting relations.

Lookup Evaluation of PatBang proceeds by reducing each non-value clause in an expression to a value form (or a series of new clauses in the case of application). A list E of value assignments is therefore an evaluation environment. For such an environment, we define $E\,(x) = v$ when $x = v \in E$. Because variable definitions are unique, this lookup function is deterministic.

Projection Projectors π in the grammar represent different outermost value sorts. We use the notation $v \in \pi$ to denote that the outermost sort of v matches π; for instance, `'A x` $\in$ `'A`. All outermost sorts are matched by some projector except for onions, which no projector matches. The cases in which projection holds are exactly the following:

Definition 3.3 (Projector Matching)**.**

$$\begin{array}{lcl@{\qquad}lcl} \overrightarrow{x}\texttt{ -> }e & \in & \texttt{fun} & \mathbb{Z} & \in & \texttt{int} \\ \overrightarrow{y}\texttt{ <- }\varphi & \in & \texttt{pat} & l\,x & \in & l \\ x\texttt{ >< }x' & \in & \texttt{scape} & & & \end{array}$$

We then define a simple projection function $E\downarrow^*_\pi(x)$ which projects from a given variable x all values which match the projector π. The result is a list because, when x is an onion, projection yields all of its components which match π.

Definition 3.4 (Value Projection)**.**

$$E\downarrow^*_\pi(x) = \begin{cases} [] & \text{if } E\,(x) = \texttt{()} \\ [] & \text{if } E\,(x) \text{ is non-onion}, E\,(x) \notin \pi \\ [E\,(x)] & \text{if } E\,(x) \text{ is non-onion}, E\,(x) \in \pi \\ E\downarrow^*_\pi(x_1) \mathbin{+\!\!+} E\downarrow^*_\pi(x_2) & \text{if } E\,(x) = x_1\texttt{ \& }x_2 \end{cases}$$

In most cases, we only need the highest priority (right-most) value from the onion. For those cases, we define the following:

Definition 3.5 (Single Value Projection)**.**

$$E\downarrow_\pi(x) = v_n \text{ where } E\downarrow^*_\pi(x) = \overrightarrow[n]{v}$$

Note that $E\downarrow_\pi(x)$ is partial; it is undefined when x contains no values which match the projector.

Compatibility We will also require a notion of *compatibility* between patterns and arguments. When an onion of scapes is called, the call is expanded in a series of steps. First, the argument is checked against each pattern in priority order to determine which scape to use. On a match, this pattern checking process yields a set of bindings from pattern variables to values within the argument. Second, these values are matched positionally with their respective function parameters and the body of the function is expanded.

In order to formalize this process, we begin by defining the pattern compatibility relation $x \preceq_E \overrightarrow{y}\texttt{ <- }\varphi \setminus \overrightarrow{v}$ as the least relation that satisfies the clauses in Figure 3.6. This relation receives an argument variable and a pattern in the context of some environment and yields a list of values for consumption by the corresponding function. Because this relation descends into the argument producing bindings as it goes, we overload it as $x \preceq_E \varphi \setminus B\,; \overrightarrow{\varphi'}$ to operate on pattern bodies to produce a set of bindings B of the form $\overrightarrow{y \mapsto v}$. The cases of pattern substitution in that figure (e.g. $\varphi_2\,[\varphi_1 \,/\, y]$) are capture-avoiding; substitution of e.g. y will not replace below a shadowing binding of y.

Here, the occurrence check driven by $\overrightarrow{\varphi'}$ is used to detect non-contractive patterns. Non-contractive patterns are those which refer to themselves without an intervening use of a label; for instance, `rec p: int & p` is a non-contractive pattern. Non-contractive patterns are treated as a match failure here for simplicity of presentation; the type system, however, treats them as type errors because

$$\begin{array}{lll} x \preceq_E \overrightarrow[n]{y}\texttt{ <- }\varphi \setminus \overrightarrow[n]{v} & \text{if} & x \preceq_E \varphi \setminus \overrightarrow[m]{y_\square \mapsto v_\square} \cup B\,; \emptyset, n \le m \\ x \preceq_E \varphi \setminus \emptyset; \overrightarrow{\varphi'} & \text{if} & \varphi \in \{\texttt{int}, \texttt{fun}, \texttt{pat}, \texttt{scape}\}, \\ & & E\downarrow_\varphi(x) \text{ is defined} \\ x \preceq_E l\,\varphi \setminus B\,; \overrightarrow{\varphi'} & \text{if} & E\downarrow_l(x) = l\,x', x' \preceq_E \varphi \setminus B\,; \emptyset \\ x \preceq_E \varphi_1\texttt{ \& }\varphi_2 \setminus B\,; \overrightarrow{\varphi'} & \text{if} & x \preceq_E \varphi_1 \setminus B_1\,; \overrightarrow{\varphi'}, x \preceq_E \varphi_2 \setminus B_2\,; \overrightarrow{\varphi'}, \\ & & B = B_2 \cup \{y \mapsto v \in B_1 \mid y \mapsto v' \notin B_2\} \\ x \preceq_E \varphi_1 \setminus B\,; \overrightarrow{\varphi'} & \text{if} & \varphi_1 = x'\overrightarrow[n]{\varphi''}, E\downarrow_{\texttt{pat}}(x') = \overrightarrow[n]{y}\texttt{ <- }\varphi_2\,, \\ & & \varphi_2 \notin \overrightarrow{\varphi'}, \varphi_3 = \varphi_2\,[\varphi''_1 \,/\, y_1] \ldots [\varphi''_n \,/\, y_n], \\ & & x \preceq_E \varphi_3 \setminus B\,; \overrightarrow{\varphi'} \cup \{\varphi_1\} \\ x \preceq_E \varphi_1 \setminus B\,; \overrightarrow{\varphi'} & \text{if} & \varphi_1 \notin \overrightarrow{\varphi'}, \varphi_1 = \texttt{rec}\,y : \varphi_2\,, \\ & & x \preceq_E \varphi_2\,[\varphi_1 \,/\, y] \setminus B\,; \overrightarrow{\varphi'} \cup \{\varphi_1\} \\ x \preceq_E y \setminus \{y \mapsto v\}; \overrightarrow{\varphi'} & \text{if} & E\,(x) = v \end{array}$$

Figure 3.6. PatBang Value Compatibility

they have no well-defined semantics. This is quite similar to the issue of non-contractive types [14] such as $\mu t.\, t$, which are also traditionally rejected. Note that this occurrence check does not trigger for contractive patterns because the set of visited patterns is "forgotten" by the label clause; the pattern matching the label's contents is related with an empty set of visits rather than the set with which the label relates.

For notational purposes, we write e.g. $x \not\preceq_E \varphi$ to indicate that there exists no B and $\overrightarrow{\varphi'}$ such that $x \preceq_E \varphi \setminus B\,; \overrightarrow{\varphi'}$.

The first clause in Figure 3.6 is of a different form than the other clauses. It describes how values bound in the pattern are arranged into a list to be used in a function. The notation $\overrightarrow[m]{y_\square \mapsto v_\square}$ implies that there is *some* ordering over the set which aligns with the variable list $\overrightarrow[n]{y}$. Note that m may be less than n. This allows patterns with some variables that are not always bound to be utilized by functions that do not require those variables.

We now require a mechanism which will apply an argument to an onion of scapes in priority order. Formally, we write $x \preceq\!\!\preceq_E \overrightarrow{v} \setminus e$ to indicate that the application of $\overrightarrow{v}$ to the argument x should result in the expression e being executed. Note that e does not contain fresh variables – its bound variables are freshened later – but it otherwise represents how the scape should be evaluated for the given argument x. We define this relation as follows:

Definition 3.7 (Value Application Compatibility)**.**

$$\begin{array}{lll} x_1 \preceq\!\!\preceq_E [] \setminus e & & \text{is false} \\ x_1 \preceq\!\!\preceq_E \overrightarrow{v} \mathbin{+\!\!+} [x_2\texttt{ >< }x_3] \setminus \overrightarrow[n]{x'_\square\texttt{ = }v'_\square} \mathbin{+\!\!+} e & \text{if} & E\downarrow_{\texttt{pat}}(x_2) = v'', \\ & & E\downarrow_{\texttt{fun}}(x_3) = \overrightarrow[n]{x'}\texttt{ -> }e\,, \\ & & x_1 \preceq_E v'' \setminus \overrightarrow[n]{v'} \\ x_1 \preceq\!\!\preceq_E \overrightarrow{v} \mathbin{+\!\!+} [x_2\texttt{ >< }x_3] \setminus e & \text{if} & E\downarrow_{\texttt{pat}}(x_2) = v'', x_1 \not\preceq_E v'', \\ & & E\downarrow_{\texttt{fun}}(x_3) = v', x_1 \preceq\!\!\preceq_E \overrightarrow{v} \setminus e \end{array}$$

As with compatibility, we write $x \not\preceq\!\!\preceq_E \overrightarrow{v}$ to signify that there is no e such that $x \preceq\!\!\preceq_E \overrightarrow{v} \setminus e$.

3.3 Small-Step Semantics

Using the above relations, we now define small-step semantics for PatBang. We take $\alpha(e)$ to be an α-conversion function which renames all bound variables in e to a fixed set of fresh names relative to the current context. We then define our small-step relation over closed e such that variable names are unique. This definition is as follows; an explanation of the rules appears below.

Definition 3.8 (PatBang Small-Step Semantics)**.**

$E \,\|[x = x']\quad \longrightarrow^1 E \,\|[x = v]$ when $E\,(x') = v$

$E \,\|[x = x_1 + x_2]\,\|\, e \;\longrightarrow^1 E \,\|[x = n]\,\|\, e$
when $E\downarrow_{\texttt{int}}(x_1) = n_1, E\downarrow_{\texttt{int}}(x_2) = n_2, n_1 + n_2 = n$

$E \,\|[x = x_1 == x_2]\,\|\, e \longrightarrow^1 E \,\|[x' = \texttt{()}\,, x = \texttt{`True}\ x']\,\|\, e$
when $E\downarrow_{\texttt{int}}(x_1) = n_1, E\downarrow_{\texttt{int}}(x_2) = n_2, n_1 = n_2, x'$ fresh

$E \,\|[x = x_1 == x_2]\,\|\, e \longrightarrow^1 E \,\|[x' = \texttt{()}\,, x = \texttt{`False}\ x']\,\|\, e$
when $E\downarrow_{\texttt{int}}(x_1) = n_1, E\downarrow_{\texttt{int}}(x_2) = n_2, n_1 \neq n_2, x'$ fresh

$E \,\|[x = x_1\ x_2]\,\|\, e'' \;\longrightarrow^1 E \,\|\, \alpha(e' \,\|[x' = r])\,\|[x = \alpha(x')]\,\|\, e''$
when $E\downarrow^*_{\texttt{scape}}(x_1) = \vec{v}, x_2 \preceq_E \vec{v} \backslash\, e' \,\|[x' = r]$

We then define small-step computation $e_0 \longrightarrow^* e_n$ to hold when $e_0 \longrightarrow^1 \ldots \longrightarrow^1 e_n$ for some $n \geq 0$. Note that $e \longrightarrow^* E$ means that computation has resulted in a final value. We write $e \not\longrightarrow^1$ iff there is no e' such that $e \longrightarrow^1 e'$; observe $E \not\longrightarrow^1$ for any E.

Here are some intuitions for the above operational semantics rules. In the second case, we observe that the first unevaluated redex is an addition: x_1 `+` x_2. We first use projection to obtain the integer from each of x_1 and x_2; we then replace the redex with the sum of the results. If e.g. x_1 is defined as an integer (or as a variable which is inductively defined as an integer) in E, then projection is straightforward. If x_1 is an onion containing integers, then the rightmost integer is used; that is, `1 & 2` projects the integer `2` in keeping with the examples in Section 2. If x_1 contains no integers, then projection is undefined and no small step can occur, leaving the evaluation "stuck" (that is, $e \not\longrightarrow^1$). The small step evaluation rules for equality use the same approach as addition.

In the final case, the first unevaluated redex is an application $x_1\ x_2$. We begin by projecting all scapes from x_1; we then use the compatibility relation to determine which scape matches the argument x_2 first. If no such scape exists, then the argument does not match any of the patterns and evaluation is stuck; otherwise, we determine the appropriate body $e' \,\|[x' = r]$ to execute (which includes the pattern's bindings). We describe the body of the scape in this manner because x', as the last variable in the body, describes the result of the scape. We then replace the original application expression $[x = x_1\ x_2]$ with $\alpha(e' \,\|[x' = r])\,\|[x = \alpha(x')]$: the body of the matched scape modified to copy the result of the scape into x. The variables in these terms have been freshened by α-conversion wherever they originated from the body of the scape in order to ensure that names in the new expression are still unique; note that both occurrences of x' freshen to the same variable here.

We have defined a small-step operational semantics for PatBang; we will now discuss how it may be typed.

4. Type System

In this section, we present a type system for the restricted form of PatBang shown in Section 3. The operational semantics of PatBang are highly flexible and we need a correspondingly flexible type system for this task. Data conjunction via the onion operator, for instance, means that we must support structural subtyping. For this reason, we base our type system on polymorphic subtype constraint type systems [2, 12, 19, 23] with a number of novel extensions as discussed below. In such systems, types are expressed as a pair between a type and a set of constraints on that type; for instance, boolean values have the type

$$\alpha_1 \backslash \{\texttt{`True}\,\alpha_2 <: \alpha_1, \texttt{`False}\,\alpha_3 <: \alpha_1, \texttt{()} <: \alpha_2, \texttt{()} <: \alpha_3\}$$

In short, this type indicates that α_1 has two lower bounds: `'True ()` and `'False ()`. As a result, we can view α_1 as the union between these two types. Quantifying over such sets gives a form of polymorphism generalizing F-bounded quantification [7] since arbitrary subtype constraints are allowed on the bound variables.

Because the constraints admitted by PatBang are so expressive, it is difficult to construct a traditional subject-reduction proof for its type system. While it is possible to compute constrained types for both expressions in $e \longrightarrow^1 e'$, it is in general difficult to align the constraints between those types to satisfy a (decidable) subtyping relation. It is also difficult to construct a regular tree or other form of denotational type model [3, 6, 22] given how flexible the typing of pattern matching needs to be[1]. Instead, we prove soundness by simulation: the type system conservatively approximates the operational semantics at each relation, ensuring that a program that "gets stuck" at runtime will have an identifiable inconsistency in the constraints of its type. In this view, it is helpful to view constraints as possible information flows in the PatBang program; for instance, we can take the constraint `()` $<:$ α_2 to indicate that, at runtime, a value of the type `()` may appear at the point in the program which we have designated α_2. In fact, every operational semantics construct has a type system analogue: clauses map to constraints, expressions map to constraint sets, the value-level pattern compatibility function maps to a type-level compatibility relation, and so on. Preserving this alignment makes the simulation proof relatively straightforward and is part of the reason we chose to formalize over an ANF grammar. As such, PatBang's type system can be viewed as a hybrid between a flow analysis [1, 18, 21] and a traditional type system.

$$\begin{array}{lcll}
C & ::= & \overbrace{c} & \textit{constraint sets} \\
c & ::= & \tau <: \alpha \mid \alpha <: \alpha \mid \alpha\,\alpha <: \alpha \mid \alpha \boxdot \alpha <: \alpha & \textit{constraints} \\
\tau & ::= & \texttt{int} \mid \texttt{()} \mid l\,\underrightarrow{\alpha} \mid \alpha\,\texttt{\&}\,\alpha \mid \overrightarrow{\alpha}\,\texttt{->}\,\alpha \backslash C \mid \beta\,\texttt{<-}\,\phi \mid \alpha\,\texttt{><}\,\alpha & \textit{types} \\
\phi & ::= & \texttt{int} \mid l\,\phi \mid \texttt{fun} \mid \texttt{pat} \mid \texttt{scape} \mid \phi\,\texttt{\&}\,\phi \mid \alpha\,\overrightarrow{\phi} \mid \texttt{rec}\,\beta : \phi \mid \beta & \textit{pattern body types} \\
\alpha & ::= & \langle x\,, \dot{\mathcal{C}} \rangle & \textit{type variables} \\
\beta & ::= & \langle y \rangle & \textit{pattern type variables}
\end{array}$$

Figure 4.1. PatBang Type Grammar

The grammar used by the PatBang type system appears in Figure 4.1. This grammar makes evident the simulation described above: the productions of constraints mirror the productions of clauses in Figure 3.1. The PatBang typechecking process consists of three phases. First, a constrained type is inferred for the expression being typechecked. Next, the constraints in that type are deductively closed over a logical system. Finally, this constraint set is checked for immediate inconsistencies which indicate a type error. The following sections describe each of these processes in detail.

4.1 Initial Derivation

Because the PatBang operational semantics and type system are so closely related, it is quite simple to lift an expression into a set of constraints; this process is called *initial derivation* and, for the most part, maps each evaluation-level construct onto its type system counterpart.

Simulation of most clauses in the operational semantics of PatBang is quite simple, but scape application presents challenges to the type system due to its need to freshen type variables in the body of the scape. Executing a program may yield unbounded calls, but the type system cannot freshen an unbounded number of type

[1] We have constructed a draft regular tree model and do believe it is possible, just not illuminative since the pattern matching sematics is very complex.

$$
\begin{aligned}
[\![[s]]\!]_{\text{E}} &= \langle \alpha \,, C \rangle \\
&\quad \text{where } [\![s]\!]_{\text{S}} = \langle \alpha \,, C \rangle \\
[\![[s' \mid\mid \vec{s}]]\!]_{\text{E}} &= \langle \alpha \,, C \cup C' \rangle \\
&\quad \text{where } [\![s']\!]_{\text{S}} = \langle \alpha' \,, C' \rangle, [\![\vec{s}]\!]_{\text{S}} = \langle \alpha' \,, C \rangle
\end{aligned}
$$

$$
\begin{aligned}
[\![x = \mathbb{Z}]\!]_{\text{S}} &= \langle \dot{\alpha}_x, \texttt{int} <: \dot{\alpha}_x \rangle \\
[\![x = \texttt{()}]\!]_{\text{S}} &= \langle \dot{\alpha}_x, \texttt{()} <: \dot{\alpha}_x \rangle \\
[\![x = l\ x_1]\!]_{\text{S}} &= \langle \dot{\alpha}_x, l\ \dot{\alpha}_{x_1} <: \dot{\alpha}_x \rangle \\
[\![x = x_1 \,\&\, x_2]\!]_{\text{S}} &= \langle \dot{\alpha}_x, \dot{\alpha}_{x_1} \,\&\, \dot{\alpha}_{x_2} <: \dot{\alpha}_x \rangle \\
[\![x = \overset{n}{\overrightarrow{x}} \texttt{->}\ e]\!]_{\text{S}} &= \langle \dot{\alpha}_x, \overset{n}{\overrightarrow{\dot{\alpha}_{x_\square}}} \texttt{->}\ x' \setminus C <: \dot{\alpha}_x \rangle \\
&\quad \text{where } [\![e]\!]_{\text{E}} = \langle \alpha' \,, C \rangle \\
[\![x = \overset{n}{\overrightarrow{y}} \texttt{<-}\ \varphi]\!]_{\text{S}} &= \langle \dot{\alpha}_x, \overset{n}{\overrightarrow{\langle y_\square \rangle}} \texttt{<-}\ [\![\varphi]\!]_{\text{P}} <: \dot{\alpha}_x \rangle \\
[\![x = x_1 \texttt{><} x_2]\!]_{\text{S}} &= \langle \dot{\alpha}_x, \dot{\alpha}_{x_1} \texttt{><} \dot{\alpha}_{x_2} <: \dot{\alpha}_x \rangle \\
[\![x = x_1]\!]_{\text{S}} &= \langle \dot{\alpha}_x, \dot{\alpha}_{x_1} <: \dot{\alpha}_x \rangle \\
[\![x = x_1\ x_2]\!]_{\text{S}} &= \langle \dot{\alpha}_x, \dot{\alpha}_{x_1}\ \dot{\alpha}_{x_2} <: \dot{\alpha}_x \rangle \\
[\![x = x_1 \boxdot x_2]\!]_{\text{S}} &= \langle \dot{\alpha}_x, \dot{\alpha}_{x_1} \boxdot \dot{\alpha}_{x_2} <: \dot{\alpha}_x \rangle
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{int}]\!]_{\text{P}} &= \texttt{int} \\
[\![l\ \varphi]\!]_{\text{P}} &= l\ [\![\varphi]\!]_{\text{P}} \\
[\![\texttt{fun}]\!]_{\text{P}} &= \texttt{fun} \\
[\![\texttt{pat}]\!]_{\text{P}} &= \texttt{pat} \\
[\![\texttt{scape}]\!]_{\text{P}} &= \texttt{scape} \\
[\![\varphi_1 \,\&\, \varphi_2]\!]_{\text{P}} &= [\![\varphi_1]\!]_{\text{P}} \,\&\, [\![\varphi_2]\!]_{\text{P}} \\
[\![x\ \overset{n}{\overrightarrow{\varphi}}]\!]_{\text{P}} &= \dot{\alpha}_x \overset{n}{\overrightarrow{[\![\varphi_\square]\!]_{\text{P}}}} \\
[\![\texttt{rec}\, y : \varphi]\!]_{\text{P}} &= \texttt{rec}\, \langle y \rangle : [\![\varphi]\!]_{\text{P}} \\
[\![y]\!]_{\text{P}} &= \langle y \rangle
\end{aligned}
$$

Figure 4.2. PatBang Initial Derivation

variables or it would not terminate. PatBang uses the polymorphism model of TinyBang [16]: type variables are freshened using a *contour* $\mathcal{C}$ which describes the context of the application that generated them; recursive calls reuse contours, preventing them from freshening variables indefinitely. We defer further discussion of contours to Section 4.3.

To model these contours, type variables in PatBang have structure (as opposed to being atomic as in most type systems). A PatBang type variable is defined as a pair between the value variable from which it was initially derived and a *possible contour* $\dot{\mathcal{C}}$, which is either a contour (if the variable has already been freshened) or the special symbol ❋ (if it is still contained within the set of constraints in its enclosing scape type). Type variables produced by initial derivation have no contour as they have not yet been freshened. For notational convenience, we write $\dot{\alpha}_x$ to represent $\langle x \,, ❋ \rangle$.

We define initial derivation using three functions: $[\![e]\!]_{\text{E}} = \langle \alpha \,, C \rangle$ (which produces a type variable and constraint set from an expression), $[\![s]\!]_{\text{S}} = \langle \alpha \,, c \rangle$ (which produces a type variable and single constraint for a single clause), and $[\![\varphi]\!]_{\text{P}} = \langle \phi \rangle$ (which produces a pattern type from a pattern). The definition of these functions appears in Figure 4.2. These functions are often written as type rules; we are using an inductive definition because there is no nondeterminism.

4.2 Constraint Closure

The second step of the typechecking process is where the hard work happens: the deductive closure over the set of constraints produced by initial derivation [2]. In essence, constraint closure abstractly models the computation performed by the operational semantics: where the operational semantics would replace clauses with other clauses, constraint closure uses constraints to add more constraints. Old constraints are never removed during closure. This is in part because the PatBang type system is flow-insensitive: constraint closure can happen in any order and it is not always clear when a constraint is no longer needed. Also, this makes the constraint closure monotonic (excepting type variable unification) and so simplifies the proof of soundness.

Because constraint closure models evaluation, we must have for each function used during evaluation some type system counterpart. We begin by defining a function $\text{FVAR}(\alpha)$ which precisely describes how fresh type variables are created to model the fresh variables required when integer comparisons are executed. (Fresh variables produced during application are discussed elsewhere; they require a more complex definition because the number of variables freshened per application is not constant.)

Definition 4.3 (Fresh Variables)**.**
For all α in the initial derivation, $\text{FVAR}(\alpha) = \alpha'$ where α' is a fresh type variable not in the initial derivation with the same possible contour as α.

4.2.1 Fibrations

Constraint closure requires type system analogues for the projection and compatibility functions appearing in Section 3.2. Before these relations can be defined, however, we need to define *fibrations*. We use fibrations to prevent a class of false positives which arise in the PatBang type system.

Recall that the evaluation system looks up the value for a given variable x simply by locating an assignment for that variable in the environment (e.g. $x = v \in E$). Likewise, a type can be found for a type variable α by searching for a bound in the constraint set (e.g. $\tau <: \alpha \in C$). In the case of a union, however, we might choose one of many lower bounds and a problem arises if these choices are made inconsistently. For instance, consider

```
1 ( (int -> 1) & (char -> 2) ) arg
```

Suppose that `arg` has the type $\alpha \setminus \{\texttt{int} <: \alpha \,, \texttt{char} <: \alpha\}$; that is, it is a union between `int` and `char`. If we naïvely check each pattern to see if *some* form of α fails to match it, we conclude that the onion does not exhaustively match `arg`; `char` fails to match the first pattern and `int` fails to match the second. But this is obviously the wrong conclusion; there exists no single type of α that fails to match both patterns.

The root of this problem is that we are not ensuring that we have a consistent view of α across pattern matches. This so-called *union alignment problem* is well-known and invariably is an issue that must be dealt with in union type systems; [8] for example is a recent paper reviewing the problem. Unfortunately, sound union elimination rules in traditional union typing systems usually end up being weaker than desired: the rule must be syntactically constrained to eliminate only on union-typed expressions which are known to be evaluated only once (and thus to have made a *single* choice for which union branch was taken at runtime).

In a subtype constraint context, where (positive) union types take the form of multiple lower bounds on a type variable, union eliminations can be viewed as occurring in canonical positions: each choice of a lower bound on a type variable implicitly performs union elimination. For example, we must process α *pointwise* first as `int` and then as `char`, fixing the choice throughout the overall check for application compatibility. To achieve this, we record the union elimination decision(s) in a fibration, formally described below. We add fibrations to the projection relation, allowing us to require that multiple projections use the same fibration. We also add fibrations to the compatibility relation. The grammar of fibrations is as follows:

$$\mathfrak{f} \quad ::= \quad \langle \tau \,, \vec{\mathfrak{f}} \rangle \mid ❋ \qquad \textit{fibrations}$$

We additionally require that, for all fibrations of the form $\langle \tau \,, \overset{n}{\vec{\mathfrak{f}}} \rangle$, there are n type variables in τ; that is, labels have $n = 1$, onions and scapes have $n = 2$, and all others have $n = 0$. Each fibration in the list corresponds directly to a type variable in τ, describing how its union elimination decisions were made. The pur-

$C \vdash \alpha \xrightarrow{\pi\triangleright} [\,]; \langle \tau\,, \overrightarrow[n]{\mathfrak{f}} \rangle$ if $\tau <: \alpha \in C, \tau$ is non-onion,
and one of $\pi = \texttt{int} \wedge \tau \neq \texttt{int} \wedge n = 0$
or $\pi = l \wedge \tau \neq l\,\alpha' \wedge n = 1$
or $\pi = \texttt{fun} \wedge \tau \neq \overrightarrow{\alpha'}\texttt{->}\alpha''\setminus C' \wedge n = 0$
or $\pi = \texttt{pat} \wedge \tau \neq \overrightarrow{\beta}\texttt{<-}\phi \wedge n = 0$
or $\pi = \texttt{scape} \wedge \tau \neq \alpha'\texttt{><}\alpha'' \wedge n = 2$

$C \vdash \alpha \xrightarrow{\pi\triangleright} [\,]; \langle \texttt{()}\,, [\,]\rangle$ if $\texttt{()} <: \alpha \in C$

$C \vdash \alpha \xrightarrow{\texttt{int}\triangleright} [\mathfrak{f}]; \mathfrak{f}$ if $\mathfrak{f} = \langle \texttt{int}\,, [\,]\rangle, \texttt{int} <: \alpha \in C$

$C \vdash \alpha \xrightarrow{l\triangleright} [\mathfrak{f}]; \mathfrak{f}$ if $l\,\alpha' <: \alpha \in C, \mathfrak{f} = \langle l\,\alpha'\,, [\mathfrak{f}']\rangle$

$C \vdash \alpha \xrightarrow{\texttt{fun}\triangleright} [\mathfrak{f}]; \mathfrak{f}$ if $\tau <: \alpha \in C, \tau = \overrightarrow{\alpha'}\texttt{->}\alpha''\setminus C', \mathfrak{f} = \langle \tau\,, [\,]\rangle$

$C \vdash \alpha \xrightarrow{\texttt{pat}\triangleright} [\mathfrak{f}]; \mathfrak{f}$ if $\tau <: \alpha \in C, \tau = \overrightarrow{\beta}\texttt{<-}\phi\,, \mathfrak{f} = \langle \tau\,, [\,]\rangle$

$C \vdash \alpha \xrightarrow{\texttt{scape}\triangleright} [\mathfrak{f}]; \mathfrak{f}$ if $\tau <: \alpha \in C, \tau = \alpha_1\texttt{><}\alpha_2\,, \mathfrak{f} = \langle \tau\,, [\mathfrak{f}_1, \mathfrak{f}_2]\rangle$

$C \vdash \alpha \xrightarrow{\pi\triangleright} \overrightarrow{\mathfrak{f}}_1 \parallel \overrightarrow{\mathfrak{f}}_2; \mathfrak{f}''$ if $\tau' <: \alpha \in C, \tau' = \alpha_1\,\&\,\alpha_2\,,$
$C \vdash \alpha_1 \xrightarrow{\pi\triangleright} \overrightarrow{\mathfrak{f}}_1; \mathfrak{f}'_1, C \vdash \alpha_2 \xrightarrow{\pi\triangleright} \overrightarrow{\mathfrak{f}}_2; \mathfrak{f}'_2,$
$\mathfrak{f}'' = \langle \tau'\,, [\mathfrak{f}'_1, \mathfrak{f}'_2]\rangle$

Figure 4.4. PatBang Projection

pose of the ❋ construction is to act as a simple base case for the fibration; it represents the lack of any further decisions being made.

For instance, consider the type $\alpha_1 \setminus \{\texttt{`A}\,\alpha_2 <: \alpha_1\,, \texttt{int} <: \alpha_2\,, \alpha_1 <: \alpha_2\}$. In a deep grammar with unions and $\mu$ types, this type is equivalent to $\mu\alpha.\ \texttt{`A}\,(\texttt{int} \cup \alpha)$: in other words, a sequence of `A labels (at least one) terminating in a an int. (Such a type may be inferred in PatBang as a result of type variable unification.) One legal fibration of this type is $\langle \texttt{`A}\,\alpha_2\,, [\langle \texttt{int}\,, [\,]\rangle]\rangle$. This fibration describes that the first decision (which, admittedly, is the only legal one) was to pick the $\texttt{`A}\,\alpha_2$ lower bound for $\alpha_1$. It also indicates that int was used as the lower bound for $\alpha_2$. Another legal fibration for this type is $\langle \texttt{`A}\,\alpha_2\,, [\divideontimes]\rangle$; this fibration may be valid in cases in which deeper decisions were not necessary, such as matching against the pattern `A _. The ❋ symbol exists in the fibration grammar to ensure that finite fibrations exist for corecursive types; while these types cannot be instantiated at runtime, type variable unification can create them in response to pathological code.

The construction of a fibration can be viewed as inference of a series of both wide and deep union elimination decisions. One problem with standard union type elimination is developing a complete type inference procedure over those unions; fibrations can be viewed as a solution to the union elimination type inference problem. Using fibrations, we can now define the type system analogues of the functions used during evaluation.

4.2.2 Projection

We define a relation to extract from a type variable all lower bounds matching a given projector. This relation corresponds to the $E{\downarrow}^*_\pi(x)$ function in the operational semantics. Unlike the operational semantics, however, the type system looses information; while evaluation has a single value for x, the type system must independently consider every type lower bound which could reach the type variable corresponding to x. For this reason, type projection is a relation and not a function.

We write $C \vdash \alpha \xrightarrow{\pi\triangleright} \overrightarrow{\mathfrak{f}}; \mathfrak{f}'$ to denote that we project from α all types which match π. These types arrive as the list of fibrations $\overrightarrow{\mathfrak{f}}$ (rather than a list of types) in order to track not only which types can be projected but which union elimination decisions have already been made for them. The $\mathfrak{f}'$ is the fibration describing the union decisions that were made in order to find this list; it is alignment with this fibration across multiple relations which preserves the union alignment property described above. Its definition appears in Figure 4.4.

The additional complexity in this projection definition when compared to Definition 3.4 is largely the result of fibrations: in the cases where rules could be simplified using Definition 3.3, the type system projection relation must create a fibration specific to the projector and so cannot easily generalize. But this is a matter of notation; type system projection is only more complex than evaluation projection in that it tracks the lower-bounding decisions made as it explores onion structure.

In cases in which only the types of the fibrations are necessary, we may write a list of types rather than the list of fibrations; that is, we write $C \vdash \alpha \xrightarrow{\pi\triangleright} \overrightarrow[n]{\tau}; \mathfrak{f}'$ to mean $C \vdash \alpha \xrightarrow{\pi\triangleright} \overrightarrow[n]{\langle \tau_\square\,, \ldots\rangle}; \mathfrak{f}'$. We may also elide $\mathfrak{f}'$ when it is unimportant; that is, $C \vdash \alpha \xrightarrow{\pi\triangleright} \overrightarrow{\mathfrak{f}}$ is an abbreviation for $\exists \mathfrak{f}'.\ C \vdash \alpha \xrightarrow{\pi\triangleright} \overrightarrow{\mathfrak{f}}; \mathfrak{f}'$. Finally, as in the evaluation system, we define a single projection relation for cases in which only the highest priority result is important:

Definition 4.5 (Single Projection)**.**
$C \vdash \alpha \xrightarrow{\pi\triangleright} \mathfrak{f}_n; \mathfrak{f}'$ holds iff $C \vdash \alpha \xrightarrow{\pi\triangleright} \overrightarrow[n]{\mathfrak{f}}; \mathfrak{f}'$ holds.

We use single projection in most cases; application is the only case which requires multiple projection. Note that single projection is partial; if there are no types in any concrete form of α matching π, it does not hold.

For an example, we consider the list type $\alpha_1 \setminus \{\texttt{`Nil}\,\alpha_4 <: \alpha_1\,, \alpha_2\,\&\,\alpha_3 <: \alpha_1\,, \texttt{`Hd}\,\alpha_5 <: \alpha_2\,, \texttt{`Tl}\,\alpha_1 <: \alpha_3\,, \texttt{()} <: \alpha_4\,, \texttt{int} <: \alpha_5\}$; intuitively, this type is $\mu t.\ \texttt{`Nil}\ \texttt{()} \cup \texttt{`Hd}\,\texttt{int}\,\&\,\texttt{`Tl}\,t$, the type of integer lists. One legal projection is $C \vdash \alpha_1 \xrightarrow{\texttt{`Hd}\triangleright} [\mathfrak{f}]; \mathfrak{f}'$ where $\mathfrak{f} = \langle \texttt{`Hd}\,\alpha_5\,, [\mathfrak{f}'']\rangle$ and $\mathfrak{f}' = \langle \alpha_2\,\&\,\alpha_3\,, [\mathfrak{f}, \langle \texttt{`Tl}\,\alpha_1\,, [\divideontimes]\rangle]\rangle$. The value of $\mathfrak{f}$ reflects the specific instance of `Hd which was found; in particular, this result delivers α_5 as a means by which the contents of that label can be accessed. The value of $\mathfrak{f}'$ contains the work necessary to reach this value of $\mathfrak{f}$: we must choose the onion for α_1, choose the $\texttt{`Tl}\,\alpha_1$ bound for $\alpha_3$ (to ensure that no `Hd value could appear there), and then use $\mathfrak{f}$ as our decision for α_2. The ❋ appears because decisions under the `Tl are irrelevant here. The separation between $\mathfrak{f}$ and $\mathfrak{f}'$ ensures that the specific work done to reach the label has been abstracted away by this projection relation. Note that there are no constraints on $\mathfrak{f}''$ – projection is shallow – but such limitations can be enforced where this relation is used.

4.2.3 Positive Compatibility

Next, we must define the type-level compatibility relations to model the pattern compatibility relations defined in the operational semantics. We begin by showing a simplified version of type compatibility that aligns quite closely with value compatibility; we then extend the relation because the compatibility failures need to be made explicit and are not just the negation of successes.

As with projection, type-level compatibility is a relation (but not a function) and includes fibrations to permit union alignment. We write $\alpha_1 \preceq_C \overrightarrow[n]{\beta}\texttt{<-}\phi \setminus \overrightarrow[n]{\tau}; \mathfrak{f}$ to indicate that an argument type variable α is compatible with the pattern $\overrightarrow[n]{\beta}\texttt{<-}\phi$, producing the argument types $\overrightarrow[n]{\tau}$. As with type projection, this is in the context of a fibration $\mathfrak{f}$ to address union alignment. Instead of using a mapping B from variables onto values, this relation uses a mapping $\Gamma = \overbrace{\beta \mapsto \tau}$ from pattern type variables onto types. Otherwise, type compatibility closely mirrors that of Figure 3.6.

The label clause of compatibility reveals why projection must relate to fibrations (as opposed to just types): compatibility of a pattern may need to explore beyond a specific projection, so we might need to constrain the fibration from projection to match the fibration from a recursive compatibility check. Using the constraint set from the example in Section 4.2.2, consider $\alpha_1 \preceq_C \texttt{`Hd}\,\texttt{int} \setminus \emptyset; \mathfrak{f}'; \emptyset$. The label clause shows the relation to hold by showing both that `Hd can be projected from α_1 and that its contents are compatible with int. In showing the latter, it is necessary to align the appropriate part of the fibration $\mathfrak{f}'$ with the

$$\alpha_1 \preceq_C \overset{n}{\overrightarrow{\beta}}\ \texttt{<-}\ \phi \setminus \overset{n}{\overrightarrow{\tau}}; \mathfrak{f} \quad \text{if} \quad \alpha_1 \preceq_C \phi \setminus \overset{m}{\overbrace{\beta_\square \mapsto \tau_\square}} \cup \Gamma; \mathfrak{f}; \emptyset, n \leq m$$

$$\alpha_1 \preceq_C \phi \setminus \emptyset; \mathfrak{f}'; \overline{\phi'} \quad \text{if} \quad \phi \in \{\texttt{int}, \texttt{fun}, \texttt{pat}, \texttt{scape}\}, \; C \vdash \alpha_1 \xrightarrow{\phi} \mathfrak{f}; \mathfrak{f}'$$

$$\alpha_1 \preceq_C l\, \phi \setminus \Gamma; \mathfrak{f}'; \overline{\phi'} \quad \text{if} \quad C \vdash \alpha_1 \xrightarrow{l} \langle l\, \alpha_2, [\mathfrak{f}] \rangle; \mathfrak{f}', \; \alpha_2 \preceq_C \phi \setminus \Gamma; \mathfrak{f}; \emptyset$$

$$\alpha_1 \preceq_C \phi_1 \,\&\, \phi_2 \setminus \Gamma; \mathfrak{f}; \overline{\phi'} \quad \text{if} \quad \alpha_1 \preceq_C \phi_1 \setminus \Gamma_1; \mathfrak{f}; \overline{\phi'}, \alpha_1 \preceq_C \phi_2 \setminus \Gamma_2; \mathfrak{f}; \overline{\phi'}, \; \Gamma = \Gamma_2 \cup \{\beta \mapsto \tau \in \Gamma_1 \mid \beta \mapsto \tau' \notin \Gamma_2\}$$

$$\alpha_1 \preceq_C \phi_1 \setminus \Gamma; \mathfrak{f}; \overline{\phi'} \quad \text{if} \quad \phi_1 = \alpha_2\, \overset{n}{\overrightarrow{\phi''}}, C \vdash \alpha_2 \xrightarrow{\texttt{pat}} \langle\, \overset{n}{\overrightarrow{\beta}}\ \texttt{<-}\ \phi_2, [] \rangle; \mathfrak{f}', \; \phi_2 \notin \overline{\phi'}, \phi_3 = \phi_2\,[\phi_1'' \,/\, \beta_1] \ldots [\phi_n'' \,/\, \beta_n], \; \alpha_1 \preceq_C \phi_3 \setminus \Gamma; \mathfrak{f}; \{\phi_1\} \cup \overline{\phi'}$$

$$\alpha_1 \preceq_C \phi_1 \setminus \Gamma; \mathfrak{f}; \overline{\phi'} \quad \text{if} \quad \alpha_1 \notin \overline{\phi'}, \phi_1 = \texttt{rec}\, \beta : \phi_2, \; \alpha_1 \preceq_C \phi_2\,[\phi_1 \,/\, \beta] \setminus \Gamma; \mathfrak{f}; \{\phi_1\} \cup \overline{\phi'}$$

$$\alpha_1 \preceq_C \beta_2 \setminus \{\beta_2 \mapsto \tau\}; \langle \tau, \overrightarrow{\mathfrak{f}} \rangle; \overline{\phi'} \quad \text{if} \quad \tau <: \alpha_1 \in C$$

Figure 4.6. PatBang Compatibility Successes

contents of the label. The example in Section 4.2.2 shows that $C \vdash \alpha_1 \xrightarrow{\text{`Hd}} \langle \text{`Hd}\, \alpha_5, [\mathfrak{f}''] \rangle; \mathfrak{f}'$ and it remains to show that $\alpha_5 \preceq_C \texttt{int} \setminus \emptyset; \mathfrak{f}''; \emptyset$; this is true when $\mathfrak{f}'' = \langle \texttt{int}, [] \rangle$.

The decidability of this relation is more subtle than its evaluation-level counterpart; nonetheless, a decidable algorithm exists to determine the type lists and fibrations for which a given compatibility will hold. We defer discussion of this decidability property to Appendix B.

4.2.4 Complete Compatibility

While the simple compatibility relation above provides intuitions about how type system compatibility works in PatBang, it is incomplete. Because compatibility is a relation, it is no longer sufficient to use the failure of the relation to signify compatibility failure. It is possible that compatibility succeeds for a given argument under one fibration but not under another; this does not allow us to determine whether it is because the fibration does not correctly describe the argument or if the argument does not match the pattern. Furthermore, we must observe the non-contractive case in Section 3.2: non-contractive patterns should generate a type failure for reasons of soundness.

To address this problem, we first define two additional grammar symbols: $\dot{\Gamma}$, which ranges over Γ and the special symbols ☹ and ☠; and $\dot{C}$, which ranges over C and the special symbol ☠. We use ☹ to designate a compatibility failure; in such cases, pattern matching should move on to the next available pattern. We use ☠ to designate that type checking should fail. This allows us to solve the above problem by making our failure modes explicit in the type system compatibility relation. We then define compatibility as follows:

Definition 4.8 (Compatibility)**.** Let relation R_{S} be the least relation satisfying the clauses in Figure 4.6. Let relation R_{F} be the least relation satisfying the clauses in Figure 4.7. We define PatBang's compatibility relation as the union between these two relations.

As with the operational semantics, substitution is capture-avoiding.

4.2.5 Application Compatibility

Just as in the small-step semantics, we use the compatibility relation to create an application compatibility relation. Because application compatibility is a type system relation, it makes failure explicit just as in Definition 4.8. Further, application compatibility must yield a fibration for union alignment purposes. Otherwise, however, the following definition is quite similar to Definition 3.7.

Using $\ddot{\Gamma}$ to range over ☹ and ☠,

$$\alpha_1 \preceq_C \overset{n}{\overrightarrow{\beta}}\ \texttt{<-}\ \phi \setminus \ddot{\Gamma}; \mathfrak{f} \quad \text{if} \quad \alpha_1 \preceq_C \phi \setminus \ddot{\Gamma}; \mathfrak{f}; \emptyset$$

$$\alpha_1 \preceq_C \phi \setminus ☹; \mathfrak{f}'; \overline{\phi'} \quad \text{if} \quad \phi \in \{\texttt{int}, \texttt{fun}, \texttt{pat}, \texttt{scape}\}, \; C \vdash \alpha_1 \xrightarrow{\phi} []; \mathfrak{f}'$$

$$\alpha_1 \preceq_C l\, \phi \setminus ☹; \mathfrak{f}; \overline{\phi'} \quad \text{if} \quad C \vdash \alpha_1 \xrightarrow{l} []; \mathfrak{f}$$

$$\alpha_1 \preceq_C l\, \phi \setminus \ddot{\Gamma}; \mathfrak{f}'; \overline{\phi'} \quad \text{if} \quad C \vdash \alpha_1 \xrightarrow{l} \langle l\, \alpha_2, [\mathfrak{f}] \rangle; \mathfrak{f}', \alpha_2 \preceq_C \phi \setminus \ddot{\Gamma}; \mathfrak{f}; \emptyset$$

$$\alpha_1 \preceq_C \phi_1 \,\&\, \phi_2 \setminus \ddot{\Gamma}; \mathfrak{f}; \overline{\phi'} \quad \text{if} \quad \alpha_1 \preceq_C \phi_i \setminus \ddot{\Gamma}; \mathfrak{f}; \overline{\phi'}, i \in \{1, 2\}$$

$$\alpha_1 \preceq_C \phi_1 \setminus ☹; \mathfrak{f}; \overline{\phi'} \quad \text{if} \quad \phi_1 \notin \overline{\phi'}, \phi_1 = \alpha_2\, \overset{n}{\overrightarrow{\phi''}}, \; C \vdash \alpha_2 \xrightarrow{\texttt{pat}} []; \mathfrak{f}'$$

$$\alpha_1 \preceq_C \phi_1 \setminus \ddot{\Gamma}; \mathfrak{f}; \overline{\phi'} \quad \text{if} \quad \phi_1 \notin \overline{\phi'}, \phi_1 = \alpha_2\, \overset{n}{\overrightarrow{\phi''}}, \; C \vdash \alpha_2 \xrightarrow{\texttt{pat}} \langle\, \overset{n}{\overrightarrow{\beta}}\ \texttt{<-}\ \phi_2, [] \rangle; \mathfrak{f}', \; \phi_3 = \phi_2\,[\phi_1'' \,/\, \beta_1] \ldots [\phi_n'' \,/\, \beta_n], \; \alpha_1 \preceq_C \phi_3 \setminus \ddot{\Gamma}; \mathfrak{f}; \{\phi_1\} \cup \overline{\phi'}$$

$$\alpha_1 \preceq_C \phi_1 \setminus \ddot{\Gamma}; \mathfrak{f}; \overline{\phi'} \quad \text{if} \quad \alpha_1 \notin \overline{\phi'}, \phi_1 = \texttt{rec}\, \beta : \phi_2, \; \alpha_1 \preceq_C \phi_2\,[\phi_1 \,/\, \beta] \setminus \ddot{\Gamma}; \mathfrak{f}; \{\phi_1\} \cup \overline{\phi'}$$

$$\alpha_1 \preceq_C \phi \setminus ☠; \mathfrak{f}; \overline{\phi'} \quad \text{if} \quad \phi \in \overline{\phi'}$$

Figure 4.7. PatBang Compatibility Failures

Definition 4.9 (Application Compatibility)**.**

$$\alpha_1 \mathrel{\overset{\preceq}{\ll}}_C \overrightarrow{\tau} \,\|[(\alpha_2 \prec\!\succ \alpha_3)] \setminus \alpha_4 ; C' \cup C''; \mathfrak{f}$$
if $C \vdash \alpha_2 \xrightarrow{\texttt{pat}} \tau'', \alpha_1 \preceq_C \tau'' \setminus \overset{n}{\overrightarrow{\tau'}}; \mathfrak{f}$, $C \vdash \alpha_3 \xrightarrow{\texttt{fun}} \left(\overset{n}{\overrightarrow{\alpha'}}\ \texttt{->}\ \alpha_4 \setminus C'\right), C'' = \overset{n}{\overbrace{\tau'_\square <: \alpha'_\square}}$

$$\alpha_1 \mathrel{\overset{\preceq}{\ll}}_C \overrightarrow{\tau} \,\|[(\alpha_2 \prec\!\succ \alpha_3)] \setminus \alpha_4 ; \dot{C}'; \mathfrak{f}$$
if $C \vdash \alpha_2 \xrightarrow{\texttt{pat}} \tau'', C \vdash \alpha_3 \xrightarrow{\texttt{fun}} \tau''', \alpha_1 \preceq_C \tau'' \setminus ☹; \mathfrak{f}$, $\alpha_1 \mathrel{\overset{\preceq}{\ll}}_C \overrightarrow{\tau} \setminus \alpha_4 ; \dot{C}'; \mathfrak{f}$

$$\alpha_1 \mathrel{\overset{\preceq}{\ll}}_C [] \setminus \alpha_2 ; ☠; \mathfrak{f} \quad \text{always}$$

$$\alpha_1 \mathrel{\overset{\preceq}{\ll}}_C \overrightarrow{\tau} \,\|[(\alpha_2 \prec\!\succ \alpha_3)] \setminus \alpha_4 ; ☠; \mathfrak{f}$$
if $C \vdash \alpha_2 \xrightarrow{\texttt{pat}} \tau'', C \vdash \alpha_3 \xrightarrow{\texttt{fun}} \left(\overset{n}{\overrightarrow{\alpha'}}\ \texttt{->}\ \alpha_4 \setminus C'\right)$, $\forall m.\ \alpha_1 \preceq_C \tau'' \setminus \overset{m}{\overrightarrow{\tau'}}; \mathfrak{f} \implies m < n$

$$\alpha_1 \mathrel{\overset{\preceq}{\ll}}_C \overrightarrow{\tau} \,\|[(\alpha_2 \prec\!\succ \alpha_3)] \setminus \alpha_4 ; ☠; \mathfrak{f} \quad \text{if } C \vdash \alpha_2 \xrightarrow{\texttt{pat}} \tau'', \alpha_1 \preceq_C \tau'' \setminus ☠; \mathfrak{f}$$

$$\alpha_1 \mathrel{\overset{\preceq}{\ll}}_C \overrightarrow{\tau} \,\|[(\alpha_2 \prec\!\succ \alpha_3)] \setminus \alpha_4 ; ☠; \mathfrak{f} \quad \text{if } C \vdash \alpha_2 \xrightarrow{\texttt{pat}} [] \text{ or } C \vdash \alpha_3 \xrightarrow{\texttt{fun}} []$$

As in projection, we elide the fibration when it is unimportant.

4.3 Contours

As a last step before defining PatBang's constraint closure relation, we now discuss contours. Contours are used during constraint closure in order to provide call-site polymorphism. PatBang uses the same polymorphism model as the one presented in Section 5.3 of [16] but, for clarity, we summarize it here.

Recall that type variables in PatBang are not atomic: a type variable is a pair between a single value variable in the original source program and a possible contour. Type variables which are captured in the body of a scape have no contour; type variables at the top level of a constraint set always have a contour. A contour describes the calling context in which the type variable was polyinstantiated. Our contour is a form of regular expression over call strings, where each character in the string is a call site in the program. For decidability and performance, we restrict these regular expressions such that (1) a given call site appears in only one place in the expression and (2) the only terms in the expression are literal call sites and Kleene closures over non-empty sets of call sites. Furthermore, we require that, during closure, no two contours overlap.

Consider the example call graph in Figure 4.10. Suppose that some scape `s` is called in three different places (which we name a, b, and c). Suppose also that call sites b and c are within another scape `t` which is called once at a site we name d. Let a and d be

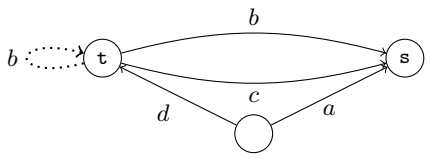

Figure 4.10. Example Call Graph

at top level. Finally, consider any value variable `x` which appears within the scape `s`. (We initially ignore the dotted edge.)

Constraint closure will polyinstantiate the contents of `s` everywhere it appears; this creates the type variables $\langle \texttt{x}, \text{“}a\text{”} \rangle$, $\langle \texttt{x}, \text{“}db\text{”} \rangle$, and $\langle \texttt{x}, \text{“}dc\text{”} \rangle$; these latter two are because call sites b and c are nested within the scape called from d. If no other calls exist, these are the only type variables for `x`. They are completely different type variables; each has a distinct set of lower bounds.

But suppose call site b from within scape `t` calls a variable which may either call `s` or recurse back into `t`; this latter case corresponds to the dotted edge in Figure 4.10. To prevent divergence, flows which differ only in the number of times they use a b-labeled are represented using the same contour. This produces a variable $\langle \texttt{x}, \text{“}db^*c\text{”} \rangle$. This variable's contour covers the contour dc (in that the former's regular expression subsumes the latter's); we would therefore unify those two variables, keeping the former contour.

We choose this model of polymorphism because of PatBang's scripting background: it can capture program-wide invariants while still being intuitive and robust to refactoring. The non-overlapping restriction and similar decisions allow constraint closure to be polynomial in usual cases; exponential cases exist but are similar to those which arise in ML-like systems.

For the purposes of constraint closure, we outline three functions which are defined in [16]:

- $\textsc{cnew}(\alpha\,, C)$, a function which creates a contour for the call site described by variable α (with C as the current constraint set);
- $\textsc{inst}(C, \mathcal{C})$, a function which polyinstantiates the free variables in a set of constraints C using contour $\mathcal{C}$; and
- $\textsc{repl}(C, \mathcal{C})$, a function which uses information in $\mathcal{C}$ to unify type variables in recursive call cycles in C.

4.3.1 Constraint Closure

PatBang's constraint closure is defined in terms of a relation $C \Longrightarrow^1 C'$ which describes single closure steps. Just as expressions are modeled by constraint sets, evaluation of expressions is modeled by constraint closure. For this reason, the constraint closure rules parallel the small-step semantics of PatBang shown in Section 3.3. The fresh variables needed by integer equality are modeled by the $\textsc{fvar}(\alpha)$ function. The variable freshening function $\alpha(*)$ used in application is modeled here by the contour functions described above.

Definition 4.12 (Constraint Closure)**.** Constraint closure is defined to be the least relation satisfying the clauses in Figure 4.11.

As in small-step evaluation, we define $C_0 \Longrightarrow^* C_n$ to indicate $C_0 \Longrightarrow^1 \ldots \Longrightarrow^1 C_n$.

After constraint closure is complete, the constraint set is checked for immediate contradictions. A constraint set containing a contradiction is *inconsistent* and represents a type error. We define inconsistency as follows:

Definition 4.13 (Inconsistency)**.** A constraint set C is inconsistent iff one of the following holds:
$\exists\, \alpha_1\, \alpha_2 <: \alpha_3 \in C$ such that $C \vdash \alpha_1 \xrightarrow{\texttt{fun}\triangleright}{}^{n}\overrightarrow{\tau}$ and $\alpha_2 \not\preccurlyeq_C {}^{n}\overrightarrow{\tau} \setminus \alpha_3$; ☠
$\exists\, \alpha_1 \boxdot \alpha_2 <: \alpha_3 \in C$ such that $C \vdash \alpha_i \xrightarrow{\texttt{int}\triangleright} []$ for $i \in \{1, 2\}$

Transitivity
$C \Longrightarrow^1 C \cup \{\tau <: \alpha_2\}$
when $\tau <: \alpha_1 \in C, \alpha_1 <: \alpha_2 \in C$

Addition
$C \Longrightarrow^1 C \cup \{\texttt{int} <: \alpha_1\}$
when $\alpha_2 + \alpha_3 <: \alpha_1 \in C, C \vdash \alpha_2 \xrightarrow{\texttt{int}\triangleright} \texttt{int}\,, C \vdash \alpha_3 \xrightarrow{\texttt{int}\triangleright} \texttt{int}$

Equality
$C \Longrightarrow^1 C \cup \{() <: \alpha'\,, \texttt{`True}\,\alpha' <: \alpha_1\,, \texttt{`False}\,\alpha' <: \alpha_1\}$
when $\textsc{fvar}(\alpha_1) = \alpha'\,, \alpha_2 == \alpha_3 <: \alpha_1 \in C,$
$C \vdash \alpha_2 \xrightarrow{\texttt{int}\triangleright} \texttt{int}\,, C \vdash \alpha_3 \xrightarrow{\texttt{int}\triangleright} \texttt{int}$

Application
$C \Longrightarrow^1 \textsc{repl}(C \cup C'', \mathcal{C})$
when $\alpha_1\, \alpha_2 <: \alpha_3 \in C, C \vdash \alpha_1 \xrightarrow{\texttt{scape}\triangleright} {}^{n}\overrightarrow{\tau}, \alpha_2 \not\preccurlyeq_C {}^{n}\overrightarrow{\tau} \setminus \alpha_4\,; C',$
$\mathcal{C} = \textsc{cnew}(\alpha_3\,, C), C'' = \textsc{inst}(C' \cup \{\alpha_4 <: \alpha_3\}, \mathcal{C})$

Figure 4.11. Constraint Closure Clauses

These conditions check for direct conflicts. In the latter condition, we detect non-integer arguments passed to integer operations. In the former, we detect cases in which an argument passed to an onion of scapes does not match any of the patterns on those scapes. Note that this also detects the application of non-scapes; in such a case, projection yields an empty list which trivially satisfies the failure condition.

4.4 Typechecking

Using the above, we then define typechecking as:

Definition 4.14 (Typechecks)**.** A closed expression e *typechecks* iff $[\![e]\!]_{\text{E}} = \langle \alpha\,, C \rangle$, $C' = \textsc{inst}(C, \{[]\})$, and for any C'', $C' \Longrightarrow^* C''$ implies C'' is consistent.

This algorithm first derives a set of constraints for the expression. It then performs an instantiation with the initial contour $\{[]\}$; this represents the top-level calling context and maintains the invariant that all top-level constraints have a contour. A program then typechecks if constraint closure cannot find an inconsistent constraint set.

Typechecking is both sound and decidable.

Theorem 4.15 (Type Soundness)**.** For any closed e such that $e \longrightarrow^* e'$, $e' \not\longrightarrow^1$, and e' is not of the form E, we have that e does not typecheck.

A full proof of this theorem appears in Appendix A.

Theorem 4.16 (Decidability)**.** Given a closed e, it is decidable whether e typechecks.

A full proof of this theorem appears in Appendix B.

5. Conclusions

In this paper the flexibility and usability of type-safe pattern matching is extended in the context of the toy language PatBang. PatBang is built on top of TinyBang [16] which itself includes flexible notions of data, including statically typable record append operations and first-class case constructs, all built on a highly polymorphic subtype constraint inference type system. PatBang extends pattern matching in two primary directions. First, patterns are fully first-class: they can be passed directly as parameters, composed, and hooked up to functions. Second, recursive patterns are supported and allow for static or dynamic determination that values match a recursive shape. These and other features such as disjunctive patterns and pattern extraction enable a new class of expressive programming patterns.

The paper here includes some new technical results of interest. We develop a novel notation of a *fibration* which is used to align “union type elimination choices” in a pattern match with multiple

branches. Since at runtime any union type is realized by a datum inhabiting exactly one side of the union, the particular choice of data made in the first pattern match will be the same made for the second, and fibrations are structures that record the "proof trace" as to which union choices were taken so they can be replayed. Fibrations are somewhat related to path-sensitive program analyses, where multiple conditional branches on the same (immutable) expression can be inferred to always branch the same way. A path-sensitive analysis is a shallow notion of union alignment, whereas fibrations align union choices over arbitrary recursive datatypes. Fibrations were briefly defined in [16] but are fully defined and first proven sound here.

In addition to the above theoretical contributions, the supplementary material includes a PatBang interpreter and typechecker, written in Haskell. This implementation includes unit tests for all the examples of Section 2, excepting the extended operators of Section 2.6 and the `dynPyteFail` example that uses runtime exceptions.

5.1 Related Work

Recursive patterns were explored in [9, 22]. [9] is less general than PatBang: recursive patterns there are defined over a language requiring ML-style declared datatypes and every pattern is thus a subcomponent of a single fixed ML datatype. PatBang includes subtyping, bounded quantification, and inferred variant types. [22] has recursive patterns in the presence of subtyping but lacks bounded polymorphism, type inference, first-class patterns or conjunctive patterns (which are not useful in their context because there is no record concatenation operator). A built-in notion of function pattern is included but it matches by matching the static type, and cannot be encoded like pytes; a pattern disjunction operator is included.

Some forms of first-class pattern were explored in [13]. This book contains many different formal systems that include more generic forms of pattern matching as well as an implemented programming language, bondi. The system is capable of expressing a certain style of higher-order pattern and describing data shapes using patterns, but it does not support fully programmatic composition of patterns, data types, or case branches. For example, the # operator in this calculus composes cases in a first-class manner but requires each case branch to return the same type, and there is no programmatic operator for composing two pattern values to make a new pattern. But bondi does admit a form of generic programming: algorithms can be written to destruct and operate on data while being agnostic to its overall structure. The disjoint conjunction operator described in Section 2.6 permits similar behavior in PatBang, although differences in the languages' data models prevent either language's generic programming from being strictly more expressive than the other's.

In the dynamically-typed world there have been several extensions to patterns, of which a recent extension to the Thorn scripting language [4] appears to be the most ambitious. The current paper was partly inspired by the expressiveness of the patterns in the aforecited paper. While Thorn lacks recursive patterns, it includes a related iterative pattern; it also includes conjunctive, disjunctive, negation, and higher-order patterns. Many of the patterns in [4] are purely dynamic; in fact arbitrary dynamic predicates can be asserted in a pattern. Such patterns could easily be added to PatBang, but the type system would have to conservatively assume that each dynamic predicate could either hold or fail.

5.2 Future work

While PatBang is a toy language, the longer-term aim is to incorporate these flexible patterns into a practical programming language. While PatBang's patterns are already very flexible, we believe more exploration is necessary to find the optimal set of pattern matching operators. At a simpler level, negation patterns are also a planned extension; they are not difficult to add to the specification but require a more powerful form of fibration to support a decidable type inference algorithm.

References

[1] O. Agesen. The cartesian product algorithm. In *ECOOP*, volume 952 of *Lecture Notes in Computer Science*, 1995.

[2] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *FPCA*, pages 31–41, 1993.

[3] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Trans. Program. Lang. Syst.*, pages 575–631, Sept. 1993.

[4] B. Bloom and M. J. Hirzel. Robust scripting via patterns. In *DLS*, pages 29–40. ACM, 2012.

[5] M. Blume, U. A. Acar, and W. Chae. Extensible programming with first-class cases. In *ICFP*, pages 239–250, 2006.

[6] G. Castagna and Z. Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *ICFP*, 2011.

[7] W. R. Cook, W. Hill, and P. S. Canning. Inheritance is not subtyping. In *POPL*, 1990.

[8] J. Dunfield. Elaborating intersection and union types. In *ICFP*, 2012.

[9] M. Fähndrich and J. Boyland. Statically checkable pattern abstractions. In *ICFP*, pages 75–84. ACM Press, 1997.

[10] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, 2002.

[11] J. Gronski and C. Flanagan. Unifying hybrid types and contracts. In *Eighth Symposium on Trends in Functional Programming*, 2007.

[12] N. Heintze. Set-based analysis of ML programs. In *LFP*, pages 306–317. ACM, 1994.

[13] B. Jay. *Pattern Calculus: Computing with Functions and Structures*. Springer Verlag, 2009.

[14] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. In *POPL*, 1984.

[15] D. B. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

[16] P. H. Menon, Z. Palmer, A. Rozenshteyn, and S. Smith. Types for flexible objects. Technical report, The Johns Hopkins University, May 2013.

[17] P. Meunier, R. B. Findler, and M. Felleisen. Modular set-based analysis from contracts. In *POPL*, 2006.

[18] M. Might and O. Shivers. Environment analysis via ΔCFA. In *POPL*, 2006.

[19] F. Pottier. A versatile constraint-based type inference system. *Nordic J. of Computing*, 7(4):312–347, 2000.

[20] D. Scott. Data types as lattices. *SIAM Journal of Computing*, 5(3): 522–586, 1976.

[21] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, 1991. TR CMU-CS-91-145.

[22] J. Vouillon. Polymorphic regular tree types and patterns. In *POPL*, 2006.

[23] T. Wang and S. F. Smith. Precise constraint-based type inference for Java. In *ECOOP*, pages 99–117, 2001. ISBN 3-540-42206-4.

[24] D. N. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for Haskell. In *POPL*, 2009.

A. Proof of Soundness

We present here a proof for Theorem 4.15, the type soundness theorem for PatBang. Our proof strategy is to build a simulation relation between the small step semantics and the type system. We show that the initial derivation of a program simulates that program and that the simulation is preserved as the program executes. Finally, we use this simulation to show that every stuck program is simulated by an inconsistent constraint set.

In order to complete the PatBang type system, we first provide definitions for the contours described in Section 4.3. We then give the simulation relation which demonstrates alignment of the evaluation and type systems. Next, we show that corresponding relations in the evaluation and type systems preserve simulation. Finally, we use this simulation to prove soundness.

A.1 Contour Definitions

Before the proof can proceed, we must precisely define the contours from Section 4.3 as well as relations over them. Contours themselves are defined using the following grammar:

Definition A.1 (Contour Grammar)**.**

$$\begin{array}{rcll} \mathcal{P} & ::= & x \mid \overline{x} & \textit{contour parts} \\ \mathcal{S} & ::= & \vec{\mathcal{P}} & \textit{contour strands} \\ \mathcal{C} & ::= & \overline{\mathcal{S}} & \textit{contours} \end{array}$$

Because contours represent regular expressions over call strings, we define operations over contours in terms of the sets of call strings they match. We therefore define *contour meaning* as an operation $[\![\mathcal{C}]\!]$ which produces the (potentially infinite) set of call strings matched by a contour.

Definition A.2 (Contour Meaning)**.**

$$\begin{array}{lrcl} \text{(Contours)} & [\![\mathcal{C}]\!] & = & \bigcup\{[\![\mathcal{S}]\!] \mid \mathcal{S} \in \mathcal{C}\} \\ \text{(Strands)} & [\![[]]\!] & = & \{[]\} \\ & [\![\overrightarrow[n]{\mathcal{P}}]\!] & = & \{\vec{x}_1 \,\|\, \ldots \,\|\, \vec{x}_n \mid \forall i \in \{1..n\}.\vec{x}_i \in [\![\mathcal{P}_i]\!]\} \\ \text{(Parts)} & [\![\overline{x}]\!] & = & \{\overrightarrow[n]{x'} \mid \forall i \in \{1..n\}.\, x'_i \in \overline{x}\} \\ & [\![x]\!] & = & \{[x]\} \end{array}$$

A.1.1 Contour Relations

In order to define the three contour functions used in closure, we require some auxiliary definitions. The first and simplest of these is contour extraction, a relation which produces the set of contours found in a type system grammatic construct such as a constraint set; we use $(\!|C|\!)$ to denote this. In most cases, this is just the natural catamorphism which unions results; for instance, $(\!|\alpha_1 <: \alpha_2|\!) = (\!|\alpha_1|\!) \cup (\!|\alpha_2|\!)$. The interesting leaf cases are defined below:

Definition A.3 (Contour Extraction)**.**

$$\begin{array}{rcl} (\!|\mathcal{C}|\!) & = & \{\mathcal{C}\} \\ (\!|*|\!) & = & \emptyset \end{array}$$

For all other leaf cases, the extraction yields an empty set.

We next define contour subsumption: a contour $\mathcal{C}_2$ subsumes another contour $\mathcal{C}_1$ (written $\mathcal{C}_1 \leq \mathcal{C}_2$) if every call sequence described by $\mathcal{C}_1$ is also described by $\mathcal{C}_2$. We also define contour overlap: a contour $\mathcal{C}_1$ overlaps a contour $\mathcal{C}_2$ (written $\mathcal{C}_1 \between \mathcal{C}_2$) if there is any call sequence represented by both of them. We define these relations below:

Definition A.4 (Contour Relations)**.**

$$\begin{array}{rcl} \mathcal{C}_1 \leq \mathcal{C}_2 & \text{iff} & [\![\mathcal{C}_1]\!] \subseteq [\![\mathcal{C}_2]\!] \\ \mathcal{C}_1 \between \mathcal{C}_2 & \text{iff} & [\![\mathcal{C}_1]\!] \cap [\![\mathcal{C}_2]\!] \neq \emptyset \end{array}$$

Because contours are a restricted subset of regular expressions, subsumption and overlap can be computed efficiently.

A.1.2 Contour Creation

We now define one of the three contour functions used during constraint closure. This function, $\textsc{cnew}(\alpha\,, C)$, generates a new contour for a given type variable representing a call site; C represent the constraint set with which the resulting contour must be compatible.

Definition A.5 (Contour Creation)**.**

$$\begin{array}{rcl} \textsc{Make}(\langle x\,, \mathcal{C}\rangle) & = & \{\mathcal{S} \,\|[x] \mid \mathcal{S} \in \mathcal{C}\} \\ \textsc{Sites}(x) & = & \{x\} \\ \textsc{Sites}(\overline{x}) & = & \overline{x} \\ \textsc{Collapse}(\mathcal{C}) & = & \{\textsc{Collapse}(\mathcal{S}) \mid \mathcal{S} \in \mathcal{C}\} \\ \textsc{Collapse}(\mathcal{S}) & = & \begin{cases} \textsc{Collapse}(\mathcal{S}_1 \,\|\, \left(\bigcup \overrightarrow[n]{\textsc{Sites}(\mathcal{P}_\square)}\right) \,\|\, \mathcal{S}_2) & \\ \qquad \text{when } \begin{cases} \mathcal{S} = \mathcal{S}_1 \,\|\, \overrightarrow[n]{\mathcal{P}} \,\|\, \mathcal{S}_2\,, n > 1, \\ \textsc{Sites}(\mathcal{P}_1) \cap \textsc{Sites}(\mathcal{P}_n) \neq \emptyset \end{cases} & \\ \mathcal{S} \qquad \text{otherwise} & \end{cases} \\ \textsc{Widen}(\mathcal{C}\,, C) & = & \mathcal{C} \cup (\bigcup\{\mathcal{C}' \mid \mathcal{C}' \in (\!|C|\!) \wedge \mathcal{C} \between \mathcal{C}'\}) \\ \textsc{CNew}(\alpha\,, C) & = & \textsc{Widen}(\textsc{Collapse}(\textsc{Make}(\alpha)), C) \end{array}$$

A.1.3 Contour Replacement

The contour replacement function $\textsc{Repl}(C, \mathcal{C})$ performs contour replacement on the provided set of constraints. This function is largely the natural homomorphisms on the type system constructs; for instance, $\textsc{Repl}(\tau <: \alpha\,, \mathcal{C}) = \textsc{Repl}(\tau\,, \mathcal{C}) <: \textsc{Repl}(\alpha\,, \mathcal{C})$. The only case for which this is not true is as follows:

Definition A.6 (Contour Replacement)**.**

$$\textsc{Repl}(\langle x\,, \dot{\mathcal{C}}\rangle, \mathcal{C}) = \begin{cases} \langle x\,, \mathcal{C}\rangle & \text{if } \dot{\mathcal{C}} \leq \mathcal{C} \\ \langle x\,, \dot{\mathcal{C}}\rangle & \text{otherwise} \end{cases}$$

A.1.4 Contour Instantiation

Contour instantiation is written $\textsc{Inst}(C, \mathcal{C})$ and instantiates free, non-contoured variables in a given constraint set. Because this function is recursively defined and because of the free variable requirement, we define instantiation as a function $\textsc{Inst}(C, \mathcal{C}\,, \overline{\alpha})$ and take $\textsc{Inst}(C, \mathcal{C})$ to be an alias for $\textsc{Inst}(C, \mathcal{C}\,, \emptyset)$. The third argument is used to track the set of variables which are presently bound and so should not be instantiated.

We also require a function $\textsc{btv}(\cdot)$ to determine the set of variables bound by a given construct. We define this function as follows:

Definition A.7 (Bound Type Variables)**.**

$$\begin{array}{rcl} \textsc{btv}(\ldots <: \alpha) & = & \{\alpha\} \\ \textsc{btv}(C) & = & \bigcup_{c \in C} \textsc{btv}(c) \end{array}$$

We now define $\textsc{Inst}(C, \mathcal{C}\,, \overline{\alpha})$. This is largely the natural homomorphisms over the type system constructs. There are two cases in which this differs:

Definition A.8 (Contour Instantiation)**.**

$$\textsc{Inst}(\langle x\,, \dot{\mathcal{C}}\rangle, \mathcal{C}\,, \overline{\alpha}) = \begin{cases} \langle x\,, \mathcal{C}\rangle & \text{if } \dot{\mathcal{C}} = * \wedge \langle x\,, \dot{\mathcal{C}}\rangle \notin \overline{\alpha} \\ \langle x\,, \dot{\mathcal{C}}\rangle & \text{otherwise} \end{cases}$$

$$\textsc{Inst}(\overrightarrow[n]{\alpha} \text{->} \alpha' \setminus C, \dot{\mathcal{C}}\,, \overline{\alpha''}) = \overrightarrow{\alpha} \text{->} \alpha' \setminus \textsc{Inst}(C, \dot{\mathcal{C}}\,, \overline{\alpha''} \cup \overline{\alpha} \cup \textsc{btv}(C))$$

A.2 Variables

Before defining simulation, we must be more precise with respect to the variable freshening function $\alpha(\cdot)$ used in Definition 3.8. Simulation must show that, when variables are freshened during small-step evaluation of application, the correponding type variables are freshened in a way that maintains simulation. To do this, we define a bijection with elements of the form $x \leftrightarrow \langle x'\,, \dot{\mathcal{C}}\rangle$; that is, the bijection maps each variable to another variable and a possible contour. The intuition behind this bijection is to track how each value variable x is freshened so that its pair $\langle x'\,, \dot{\mathcal{C}}\rangle$ can be simulated by

the type variable corresponding to x. This bijection is implicitly parametric in the expression e that we are typechecking.

Definition A.9 (Variable Bijection)**.**
Let $\overset{n}{\overrightarrow{x}}$ be all variables in the expression e. For all $i \in \{1..n\}$, we let $x_i \leftrightarrow \langle x_i \,, \divideontimes \rangle$. The remainder of the bijection for e is unconstrained.

We choose this bijection because it allows us to align fresh variables with the type variables that represent them. Throughout this proof, we occasionally write a pair in the place of the variable it represents; for instance, we may write $\langle x' \,, \mathcal{C} \rangle$ = v in place of x = v if $x \leftrightarrow \langle x' \,, \mathcal{C} \rangle$.

We will also need to be more precise in our definition of the α-conversion function discussed in Section 3.3. We begin by defining a function $\textsc{bev}(e)$ which produces the set of variables bound within a given expression e; this parallels Definition A.7.

Definition A.10 (Bound Variables)**.**

$$\begin{aligned} \textsc{bev}(x \texttt{ = } \ldots) &= \{x\} \\ \textsc{bev}(\overrightarrow{s}) \quad &= \textstyle\bigcup_{s' \in \overrightarrow{s}} \textsc{bev}(s') \end{aligned}$$

Next we define a function $\textsc{einst}(*, \underbrace{\overrightarrow{x}}, \mathcal{C})$ which freshens the free variables in the language construct $*$ except those appearing in $\underbrace{\overrightarrow{x}}$. This definition is the analogue of Definition A.8 and, as such, is largely the natural homomorphism over the language constructs. The only case in which this function deviates is shown below. Recall that, as specified above, we use pairs in place of variables according to the bijection in Definition A.9.

Definition A.11 (Evaluation Instantiation)**.**

$$\textsc{einst}(\langle x \,, \dot{\mathcal{C}} \rangle, \mathcal{C} \,, \underbrace{\overrightarrow{x'}}) = \begin{cases} \langle x \,, \mathcal{C} \rangle & \text{if } \dot{\mathcal{C}} = \divideontimes \wedge \langle x \,, \dot{\mathcal{C}} \rangle \notin \underbrace{\overrightarrow{x'}} \\ \langle x \,, \dot{\mathcal{C}} \rangle & \text{otherwise} \end{cases}$$

$$\textsc{einst}(\overrightarrow{x} \texttt{ -> } e \,, \dot{\mathcal{C}} \,, \underbrace{\overrightarrow{x'}}) = \overrightarrow{x} \texttt{ -> } \textsc{einst}(e \,, \dot{\mathcal{C}} \,, \underbrace{\overrightarrow{x'}} \cup \underbrace{\overrightarrow{x}} \cup \textsc{bev}(e))$$

Given the above, we define an α-conversion function $\textsc{efresh}(x \,, e)$ which freshens variables in the given expression in the context of call site x. We define this function as:

Definition A.12 (Evaluation Freshening)**.**

$$\begin{aligned} \textsc{efresh}(\langle x \,, \mathcal{C} \rangle, e) &= \textsc{einst}(e \,, \emptyset, \mathcal{C}') \\ \textsc{efresh}(\langle x \,, \mathcal{C} \rangle, x') &= \textsc{einst}(x' \,, \emptyset, \mathcal{C}') \\ \text{where} \quad \mathcal{C}' &= \{\mathcal{S} \,\|[x] \mid \mathcal{S} \in \mathcal{C}\} \end{aligned}$$

In this function, $\mathcal{C}'$ is analogous to the result of MAKE from Definition A.5. We overload EFRESH to be defined over lists of scapes, which is calculated by applying pointwise on each scape.

Using the freshening function as defined above, we can now clarify its use in the small step semantics in Definition 3.8. We do so in the following definition:

Definition A.13 (Clarified Application Semantics)**.**

$$E \,\|[x \texttt{ = } x_1 \;\; x_2] \,\|\, e'' \longrightarrow^1 E \,\|\, \textsc{efresh}(x \,, e' \,\|[x' \texttt{ = } r]) \,\|[x \texttt{ = } \textsc{efresh}(x \,, x')] \,\|\, e'$$
$$\text{when } E \downarrow^*_{\texttt{scape}}(x_1) = \overrightarrow{v}, x_2 \preceq_E \overrightarrow{v} \backslash\, e' \,\|[x' \texttt{ = } r]$$

Observe that Definition A.13 merely replaces the use of $\alpha(\cdot)$ with $\textsc{efresh}(x \,, \cdot)$. For this proof, we take the small-step semantics of PatBang to be those in Definition 3.8 except the application case, which is taken from Definition A.13. In light of this precise definition of freshening, it is now possible to define the simulation relation for the soundness proof.

A.3 Simulation

We now define the simulation relation for the soundness proof as a two-place relation $*_1 \preccurlyeq *_2$ between an evaluation-level construct $*_1$ and its corresponding type-level construct $*_2$. Values, for example, are simulated by types; clauses are simulated by constraints. We define this relation as follows:

Definition A.14 (Simulation)**.**
The simulation relation $*_1 \preccurlyeq *_2$ is defined by the rules in Figure A.15.

variables:			
$\divideontimes$	$\preccurlyeq$	$\divideontimes$	
$\mathcal{C}$	$\preccurlyeq$	$\mathcal{C}'$	iff $\mathcal{C} \leq \mathcal{C}'$
$\langle x \,, \dot{\mathcal{C}} \rangle$	$\preccurlyeq$	$\langle x \,, \dot{\mathcal{C}}' \rangle$	iff $\dot{\mathcal{C}} \preccurlyeq \dot{\mathcal{C}}'$
$\underbrace{\overrightarrow{x}}$	$\preccurlyeq$	$\underbrace{\overrightarrow{\alpha}}$	iff $\forall\, x' \in \underbrace{\overrightarrow{x}}.\, \exists\, \alpha' \in \underbrace{\overrightarrow{\alpha}}.\; x' \preccurlyeq \alpha'$
patterns:			
y	$\preccurlyeq$	$\langle y \rangle$	
`int`	$\preccurlyeq$	`int`	
$l\, \varphi$	$\preccurlyeq$	$l\, \phi$	iff $\varphi \preccurlyeq \phi$
`fun`	$\preccurlyeq$	`fun`	
`pat`	$\preccurlyeq$	`pat`	
`scape`	$\preccurlyeq$	`scape`	
φ_1 `&` φ_2	$\preccurlyeq$	ϕ_1 `&` ϕ_2	iff $\varphi_1 \preccurlyeq \phi_1 \,, \varphi_2 \preccurlyeq \phi_2$
$x \overset{n}{\overrightarrow{y}}$	$\preccurlyeq$	$\alpha \overset{n}{\overrightarrow{\beta}}$	iff $x \preccurlyeq \alpha \,, \forall i \in \{1..n\}.\; y_i \preccurlyeq \beta_i$
`rec` y `:` φ	$\preccurlyeq$	`rec` β `:` ϕ	iff $y \preccurlyeq \beta \,, \varphi \preccurlyeq \phi$
$\overset{n}{\underbrace{y \mapsto v}}$	$\preccurlyeq$	$\overset{n}{\underbrace{\beta \mapsto \tau}}$	iff $\forall i \in \{1..n\}.\; y_i \preccurlyeq \beta_i \wedge v_i \preccurlyeq \tau_i$
values:			
n	$\preccurlyeq$	`int`	iff $n \in \mathbb{Z}$
`()`	$\preccurlyeq$	`()`	
$l\, x$	$\preccurlyeq$	$l\, \alpha$	iff $x \preccurlyeq \alpha$
x_1 `&` x_2	$\preccurlyeq$	α_1 `&` α_2	iff $x_1 \preccurlyeq \alpha_1 \wedge x_2 \preccurlyeq \alpha_2$
$\overset{n}{\overrightarrow{x}}$ `->` e	$\preccurlyeq$	$\overset{n}{\overrightarrow{\alpha}}$ `->` $\alpha' \backslash C$	iff $e \preccurlyeq \langle \alpha' , C \rangle, \forall i \in \{1..n\}.\; x_i \preccurlyeq \alpha_i$
$\overset{n}{\overrightarrow{y}}$ `<-` φ	$\preccurlyeq$	$\overset{n}{\overrightarrow{\beta}}$ `<-` ϕ	iff $\varphi \preccurlyeq \phi \,, \forall i \in \{1..n\}.\; y_i \preccurlyeq \beta_i$
x_1 `><` x_2	$\preccurlyeq$	α_1 `><` α_2	iff $x_1 \preccurlyeq \alpha_1 \,, x_2 \preccurlyeq \alpha_2$
clauses:			
x `=` v	$\preccurlyeq$	$\tau <: \alpha$	iff $x \preccurlyeq \alpha \,, v \preccurlyeq \tau$
x `=` x'	$\preccurlyeq$	$\alpha' <: \alpha$	iff $x \preccurlyeq \alpha \,, x' \preccurlyeq \alpha'$
x_1 `=` $x_2\ x_3$	$\preccurlyeq$	$\alpha_2\ \alpha_3 <: \alpha_1$	iff $\forall i \in \{1..3\}.\, x_i \preccurlyeq \alpha_i$
x_1 `=` $x_2 \boxdot x_3$	$\preccurlyeq$	$\alpha_2 \boxdot \alpha_3 <: \alpha_1$	iff $\forall i \in \{1..3\}.\, x_i \preccurlyeq \alpha_i$
expressions:			
$[]$	$\preccurlyeq$	C	
$e \,\|[s]$	$\preccurlyeq$	$C \cup \{c\}$	iff $e \preccurlyeq C \wedge s \preccurlyeq c$
$e \,\|[s]$	$\preccurlyeq$	$\langle \alpha \,, C \rangle$	iff $s = x$ `=` $\ldots, e \,\|[s] \preccurlyeq C \wedge x \preccurlyeq \alpha$

Figure A.15. Simulation Relation Definition

A.4 Contours

We begin constructing the type soundness proof by showing some properties of contours.

Lemma A.16. Given any contour $\mathcal{C} \neq \emptyset$, $[\![\mathcal{C}]\!] \neq \emptyset$.

Proof. Because $\mathcal{C} \neq \emptyset$, it is sufficient to show that $[\![\mathcal{S}]\!] \neq \emptyset$ for any $\mathcal{S}$ in $\mathcal{C}$. When $\mathcal{S}$ is the empty list, the result is $\{[]\}$ and so non-empty. Otherwise, it suffices to show that $[\![\mathcal{P}]\!] \neq \emptyset$ for any $\mathcal{P}$, which is immediate by inspection. □

Lemma A.17. Suppose $\mathcal{C}_1 \leq \mathcal{C}_2$ and $\mathcal{C}_1 \leq \mathcal{C}_3$ and $\mathcal{C}_1 \neq \emptyset$. Then $\mathcal{C}_2 \between \mathcal{C}_3$.

Proof. By the definition of subsumption, we have $[\![\mathcal{C}_1]\!] \subseteq [\![\mathcal{C}_2]\!]$ and $[\![\mathcal{C}_1]\!] \subseteq [\![\mathcal{C}_3]\!]$. Thus, $[\![\mathcal{C}_2]\!] \cap [\![\mathcal{C}_3]\!] \supseteq [\![\mathcal{C}_1]\!]$. By Lemma A.16, we have that $[\![\mathcal{C}_1]\!] \neq \emptyset$ and so $\mathcal{C}_2 \between \mathcal{C}_3$ by Definition A.4. □

Lemma A.18. Contour subsumption is transitive.

Proof. Trivial by definition of contour subsumption: $\subseteq$ is transitive. □

Lemma A.19. $(\!|C_1 \cup C_2|\!) = (\!|C_1|\!) \cup (\!|C_2|\!)$

Proof. Trivial by inspection of Definition A.3. □

Lemma A.20. If τ appears in C, then $(\!|\tau|\!) \subseteq (\!|C|\!)$. Similar statements are true about other constructs appearing in C.

Proof. Trivial by inspection of Definition A.3. □

Lemma A.21. If two contour strands have a meaning subset relationship, this relationship is preserved by prepending or appending a new contour part. This is to say that $[\![\mathcal{S}_1]\!] \subseteq [\![\mathcal{S}_2]\!]$ iff $[\![\mathcal{S}_1 \,\|\, [\mathcal{P}]]\!] \subseteq [\![\mathcal{S}_2 \,\|\, [\mathcal{P}]]\!]$; likewise, $[\![\mathcal{S}_1]\!] \subseteq [\![\mathcal{S}_2]\!]$ iff $[\![[\mathcal{P}] \,\|\, \mathcal{S}_1]\!] \subseteq [\![[\mathcal{P}] \,\|\, \mathcal{S}_2]\!]$.

Proof. By Definition A.2, $[\![\cdot]\!]$ assigns to contours a subset of the semantics of regular expressions. This property therefore reduces to showing that subsumption between regular expressions is preserved by concatenation, which is true. □

Lemma A.22. If two contours have a meaning subset relationship, this relationship is preserved by prepending or appending a new contour part to each strand in both contours. This is to say that $[\![\mathcal{C}_1]\!] \subseteq [\![\mathcal{C}_2]\!]$ iff $[\![\{\mathcal{S} \,\|\, [\mathcal{P}] \mid \mathcal{S} \in \mathcal{C}_1\}]\!] \subseteq [\![\{\mathcal{S} \,\|\, [\mathcal{P}] \mid \mathcal{S} \in \mathcal{C}_2\}]\!]$; likewise, $[\![\mathcal{C}_1]\!] \subseteq [\![\mathcal{C}_2]\!]$ iff $[\![\{[\mathcal{P}] \,\|\, \mathcal{S} \mid \mathcal{S} \in \mathcal{C}_1\}]\!] \subseteq [\![\{[\mathcal{P}] \,\|\, \mathcal{S} \mid \mathcal{S} \in \mathcal{C}_2\}]\!]$.

Proof. In a fashion similar to Lemma A.22. In particular, contours simply introduce the semantics of regular expression alternatives; preservation of subsumption by concatenation is still valid in the presence of these alternatives. □

A.5 Constraint Set Well-Formedness

The constraint closure process relies on the assumption that every contour in the constraint set is disjoint; that is, no two contours found in the constraint set overlap. This is necessary to ensure that closure is efficient; otherwise, two premises which share a term might apply at some intersection of their terms' contours. Furthermore, it is necessary to ensure that every constraint is meaningful; therefore, no constraint may contain the empty contour ($\emptyset$). We describe a contour set as *well-formed* when all of its contours are disjoint and none are the empty contour. A constraint set is well-formed when the contours which appear within it constitute a well-formed set. We formalize this as Definition A.23.

Definition A.23 (Well-Formed Contour Set)**.** A contour set $\overline{\mathcal{C}'}$ is well-formed when $\forall \mathcal{C}_1 \in \overline{\mathcal{C}'}. \mathcal{C}_1 \neq \emptyset \wedge \forall \mathcal{C}_2 \in \overline{\mathcal{C}'}. \mathcal{C}_1 \neq \mathcal{C}_2 \implies \mathcal{C}_1 \between \mathcal{C}_2$. A constraint set C is well-formed when $(\!|C|\!)$ is well-formed.

We now show some important properties of well-formed constraint sets.

Lemma A.24. Let $C' = C_1 \cup C_2$ where C_1 and C_2 are well-formed. If $(\!|C_1|\!) \subseteq (\!|C_2|\!)$, then C' is also well-formed.

Proof. By Lemma A.19 and because $(\!|C_1|\!) \subseteq (\!|C_2|\!)$, we thus have $(\!|C'|\!) = (\!|C_1 \cup C_2|\!) = (\!|C_1|\!) \cup (\!|C_2|\!) = (\!|C_2|\!)$. If C_2 is well-formed, we have that $(\!|C_2|\!)$ is well-formed; thus, $(\!|C'|\!)$ is well-formed and so C' is well-formed. □

Lemma A.25. Let C be a well-formed constraint set and let $\mathcal{C}_1$ be a contour such that $\mathcal{C}_1 \leq \mathcal{C}_2, \mathcal{C}_1 \leq \mathcal{C}_3$, and $\{\mathcal{C}_2, \mathcal{C}_3\} \subseteq (\!|C|\!)$. Then $\mathcal{C}_2 = \mathcal{C}_3$.

Proof. Because C is well-formed, we know that $\mathcal{C}_1 \neq \emptyset$. By Lemma A.17, we have that $\mathcal{C}_2 \between \mathcal{C}_3$. Again because C is well-formed, we have that $\forall \mathcal{C}, \mathcal{C}' \in C. \mathcal{C} \neq \mathcal{C}' \implies \mathcal{C} \between \mathcal{C}'$. Since $\mathcal{C}_2 \between \mathcal{C}_3$ we must therefore have $\mathcal{C}_2 = \mathcal{C}_3$. □

Lemma A.26. Let $\overline{\mathcal{C}}$ be well-formed and let $\overline{\mathcal{C}'} \subseteq \overline{\mathcal{C}}$; then $\overline{\mathcal{C}'}$ is also well-formed. If C and C' are well-formed and $(\!|C'|\!) \subseteq (\!|C|\!)$, then $C \cup C'$ is well-formed. If C is well-formed and $C' \subseteq C$, then C' is also well-formed.

Proof. Because $\overline{\mathcal{C}}$ is well-formed, we have that $\emptyset \notin \overline{\mathcal{C}}$ and that $\forall \mathcal{C}_1, \mathcal{C}_2 \in \overline{\mathcal{C}}. [\![\mathcal{C}_1]\!] \cap [\![\mathcal{C}_2]\!] = \emptyset$. Because $\overline{\mathcal{C}'} \subseteq \overline{\mathcal{C}}$, we have that $\emptyset \notin \overline{\mathcal{C}'}$. Likewise, we have that $\forall \mathcal{C}_1, \mathcal{C}_2 \in \overline{\mathcal{C}'}. [\![\mathcal{C}_1]\!] \cap [\![\mathcal{C}_2]\!] = \emptyset$. Thus, $\overline{\mathcal{C}'}$ is well-formed.

If $(\!|C'|\!) \subseteq (\!|C|\!)$, the fact that $C \cup C'$ is well-formed reduces to the above.

If $C' \subseteq C$, then $(\!|C'|\!) \subseteq (\!|C|\!)$; this can be established from Definition A.3 because $(\!|\cdot|\!)$ is defined pointwise on constraint sets due to being a natural homomorphism. As a result, that C' is well-formed reduces to the above. □

A.6 Evaluation Properties

We briefly show some properties of the evaluation system which become important in later proofs.

Lemma A.27. All values represented in a closed environment E are finite.

Proof. By induction on the number of clauses to the left of a given variable definition. For any variable x in a closed E, every variable x' in its value must appear to the left of the definition of x; thus, fewer variables appear to the left of the definition of x' than appear to the left of the definition of x. □

Lemma A.28. Value compatibility is deterministic: given a value variable and a pattern, only one definition of compatibility can exist for a given x and φ.

Proof. Immediate by inspection of Figure 3.6. □

A.7 Simulation Properties

We next show useful properties of the simulation relation in Definition A.14.

Lemma A.29. Suppose $\overrightarrow{{}^{n}s} \,\|\, [s'] \,\|\, \overrightarrow{{}^{m}s''} \preccurlyeq C$. Then $\overrightarrow{{}^{n}s} \,\|\, \overrightarrow{{}^{m}s''} \preccurlyeq C$.

Proof. Immediate from Definition A.14. □

Lemma A.30. Suppose $\overrightarrow{{}^{n}s'} \preccurlyeq C'$ and $\overrightarrow{{}^{m}s''} \preccurlyeq C''$. Let $\overrightarrow{{}^{n+m}s}$ be defined such that the latter list has been interjected into the former at some position k; that is, for some k in $\{0..n\}$, we have $s_j = s'_j$ for all $j \leq k$, $s_j = s'_{j-m}$ for all $j > k + m$, and $s_j = s''_{j-k}$ otherwise. Let $C = C' \cup C''$. Then $\overrightarrow{{}^{n+m}s} \preccurlyeq C$.

As a corollary, if $e \preccurlyeq C$ for some e and C, then $e \preccurlyeq C \cup C'$ for any C'.

Proof. By repeated application of Definition A.14, we have that $[] \,\|\, [s'_1] \,\|\, \ldots \,\|\, [s'_n] \preccurlyeq C' \cup \{c'_1\} \cup \ldots \cup \{c'_n\}$. Likewise, $[] \,\|\, [s''_1] \,\|\, \ldots \,\|\, [s''_m] \preccurlyeq C'' \cup \{c''_1\} \cup \ldots \cup \{c''_m\}$.

We aim to show that $\overrightarrow{{}^{n+m}s} \preccurlyeq C$. Because $C = C' \cup C''$, we have that $C = C' \cup C'' \cup \underbrace{c' \cup \ldots \cup c'}_{n} \cup \underbrace{c'' \cup \ldots \cup c''}_{m}$. We can thus show that $[] \,\|\, [s_1] \,\|\, \ldots \,\|\, [s_{n+m}] \preccurlyeq C \cup \{c_1\} \cup \ldots \cup \{c_{n+m}\}$ by choosing each c as we chose s: we let $c_j = c'_j$ for all $j \leq k$, $c_j = c'_{j-m}$ for all $j > k + m$, and $c_j = c''_{j-k}$ otherwise. By Definition A.14, we are finished.

The corollary can be shown by the degenerate case in which $m = 0$. □

Lemma A.31. Suppose $\overline{x} \preccurlyeq \overline{\alpha}$ and $\overline{x'} \preccurlyeq \overline{\alpha'}$. Then $\overline{x} \cup \overline{x'} \preccurlyeq \overline{\alpha} \cup \overline{\alpha'}$.

Proof. From $\overline{x} \preccurlyeq \overline{\alpha}$ we have that for each x in $\overline{x}$ there exists some α in $\overline{\alpha}$ such that $x \preccurlyeq \alpha$; thus, there exists some α in $\overline{\alpha} \cup \overline{\alpha'}$ such that $x \preccurlyeq \alpha$. The same argument can be made regarding $\overline{\alpha'}$. As a result, every x in $\overline{x} \cup \overline{x'}$ has some α in $\overline{\alpha} \cup \overline{\alpha'}$ such that $x \preccurlyeq \alpha$. By Definition A.14, we are finished. □

Lemma A.32. If $e \preccurlyeq C$, then $x_2 = x_1 \in e \implies \exists\, \alpha_1\,, \alpha_2\,.\ \alpha_1 <: \alpha_2 \in C \wedge x_2 = x_1 \preccurlyeq \alpha_1 <: \alpha_2$. Corresponding statements hold true for the remaining expression and constraint forms.

Proof. Immediate from Definition A.14. □

Lemma A.33. If $e \preccurlyeq C$ and C is well-formed, then $x_2 = x_1 \in e \wedge x_3 = x_2 \in e \implies \exists\, \alpha_1\,, \alpha_2\,, \alpha_3 \in C.\ \{\alpha_1 <: \alpha_2\,, \alpha_2 <: \alpha_3\} \in C \wedge \forall i \in \{1..3\}.\, x_i \preccurlyeq \alpha_i$. Similar statements hold for other sequences of constraints.

Proof. From Lemma A.32, we have that there exist some α_1, α_2, α_2', and α_3 such that $\{\alpha_1 <: \alpha_2\,, \alpha_2' <: \alpha_3\} \in C$, $x_1 \preccurlyeq \alpha_1$, $x_2 \preccurlyeq \alpha_2$, $x_2 \preccurlyeq \alpha_2'$, and $x_3 \preccurlyeq \alpha_3$. It remains to show that $\alpha_2 = \alpha_2'$.

Let $x_2 \leftrightarrow \langle x_1''\,, \dot{\mathcal{C}}_1\rangle$, let $\alpha_2 = \langle x_2''\,, \dot{\mathcal{C}}_2\rangle$, and let $\alpha_2' = \langle x_3''\,, \dot{\mathcal{C}}_3\rangle$. From $x_2 \preccurlyeq \alpha_2$ and $x_2 \preccurlyeq \alpha_2'$, we have that $x_1'' = x_2'' = x_3''$. We have two cases: one in which $\dot{\mathcal{C}}_1 = \divideontimes$ and one in which it does not. If $\dot{\mathcal{C}}_1 = \divideontimes$, then $x_2 \preccurlyeq \alpha_2$ gives us by Definition A.14 that $\dot{\mathcal{C}}_2 = \divideontimes$; similarly, $\dot{\mathcal{C}}_3 = \divideontimes$. Thus, $\alpha_2 = \alpha_2'$.

Otherwise, $\dot{\mathcal{C}}_1 = \mathcal{C}_1$. By $x_2 \preccurlyeq \alpha_2$ and $x_2 \preccurlyeq \alpha_2$, we have that $\dot{\mathcal{C}}_2 = \mathcal{C}_2$ and $\dot{\mathcal{C}}_3 = \mathcal{C}_3$ such that $\mathcal{C}_1 \leq \mathcal{C}_2$ and $\mathcal{C}_1 \leq \mathcal{C}_3$ Because C is well-formed, we know by Lemma A.25 that $\mathcal{C}_2 = \mathcal{C}_3$; thus, $\alpha_2 = \alpha_2'$. □

A.8 Initial Derivation Simulation

Our next step is to show that the initial derivation of a program simulates that program. We begin by showing the simulation of initial pattern derivation.

Lemma A.34. For any φ such that $[\![\varphi]\!]_{\text{P}} = \phi$, we have that $\varphi \preccurlyeq \phi$.

Proof. By induction on the definition of initial derivation in Figure 4.2, every case is immediate from Definition A.14. □

We next show that initial derivation over clauses and expressions simulates as expected.

Lemma A.35. Let e be a well-formed expression $\overset{n}{\overrightarrow{s}}$. If $[\![e]\!]_{\text{E}} = \langle \alpha\,, C\rangle$, then $e \preccurlyeq \langle \alpha\,, C\rangle$ where C is well-formed.

Proof. By structural induction on the size of e.

For all $1 \leq i \leq n$, suppose $s_i = x_i = \ldots$, let $\alpha_i = \dot{\alpha}_{x_i}$, and let $c_i = [\![s_i]\!]_{\text{S}}$. Then by the definition of initial derivation, $C = \overset{n}{\underbrace{c}}$ and $\alpha = \alpha_n$. By Definition A.14, it is thus sufficient to show that $e \preccurlyeq C$. By induction on the length of e, it is sufficient to show that $s_i \preccurlyeq c_i$ for all $1 \leq i \leq n$. For all s_i which are not assignments of patterns or functions, this property is immediate by inspection of Figure 4.2. Otherwise, we have two cases: when s_i is a pattern assignment or when s_i is a function assignment.

Suppose $s_i = x = \overrightarrow{y} \texttt{<-} \varphi$. By Figure 4.2, $c_i = \overrightarrow{\beta} \texttt{<-} \phi$. By inspection, each y simulates the corresponding β. We have that $\varphi \preccurlyeq \phi$ by Lemma A.34. Thus, $s_i \preccurlyeq c_i$.

Suppose $s_i = x = \overset{m}{\overrightarrow{x'}} \texttt{->} e'$. By Figure 4.2, $c_i = \overset{m}{\overrightarrow{\alpha'}} \texttt{->} \alpha'' \backslash C'$. To show that $s_i \preccurlyeq c_i$, we show that for all $1 \leq j \leq m$ we have $x_j' \preccurlyeq \alpha_j'$; we must also show that $e' \preccurlyeq \langle \alpha''\,, C'\rangle$. The former is trivial by inspection; the latter is proven by induction on this lemma (because e' is smaller than e).

It remains to show that C is well-formed. This is trivial because every type variable appearing within it resulted from Figure 4.2 and all such type variables have no contour. □

A.9 Simulation Preservation

The next step in this proof is to demonstrate that each pair of relations in the evaluation and type systems preserve the simulation relation shown above. Because these relations make use of fibrations, it is now necessary to discuss how fibrations relate to variables. We inductively define the *canonical fibration* for a given value variable as follows:

Definition A.36 (Canonical Fibration)**.** Suppose variable x exists in expression e. Suppose also that, for some C and α appearing in that C, $e \preccurlyeq C$ and $x \preccurlyeq \alpha$. A fibration $\mathfrak{f}$ is *canonical* for x and α under e and C (equiv $x \preccurlyeq \alpha \mathbin{@} \mathfrak{f}$) iff for some row in the following table, we have $E(x) = v$ and $\tau <: \alpha \in C$ and $\mathfrak{f} = \langle \tau\,, \overrightarrow{\mathfrak{f}'}\rangle$ and P is true. (This definition is built by induction on the structure of the fiber.)

v	τ	$\overrightarrow{\mathfrak{f}'}$	P
n	`int`	$[]$	$n \in \mathbb{Z}$
`()`	`()`	$[]$	
$l\, x_1$	$l\, \alpha_1$	$[\mathfrak{f}_1]$	$x_1 \preccurlyeq \alpha_1 \mathbin{@} \mathfrak{f}_1$
x_1 `&` x_2	α_1 `&` α_2	$[\mathfrak{f}_1, \mathfrak{f}_2]$	$x_1 \preccurlyeq \alpha_1 \mathbin{@} \mathfrak{f}_1, x_2 \preccurlyeq \alpha_2 \mathbin{@} \mathfrak{f}_2$
$\overrightarrow{x'}$ `->` e'	$\overrightarrow{\alpha'}$ `->` $\alpha'' \backslash C'$	$[]$	
$\overrightarrow{y}$ `<-` φ	$\overrightarrow{\beta}$ `<-` ϕ	$[]$	
x_1 `><` x_2	α_1 `><` α_2	$[\mathfrak{f}_1, \mathfrak{f}_2]$	$x_1 \preccurlyeq \alpha_1 \mathbin{@} \mathfrak{f}_1, x_2 \preccurlyeq \alpha_2 \mathbin{@} \mathfrak{f}_2$

We can now show that various relations used during constraint closure preserve constraint set simulation.

Lemma A.37. Suppose that $E \preccurlyeq C$, $E{\downarrow}^*_\pi(x) = \overset{n}{\overrightarrow{v}}$, and $x \preccurlyeq \alpha \mathbin{@} \mathfrak{f}^*$. Then $C \vdash \alpha \overset{\pi}{\triangleright} \overset{n}{\overrightarrow{\mathfrak{f}}}; \mathfrak{f}'$ and for all $1 \leq i \leq n$ we have $v_i \preccurlyeq \tau_i$ (where $\mathfrak{f}_i = \langle \tau_i\,, \ldots\rangle$).

Proof. By induction on the depth of x (which is sound by Lemma A.27). We proceed by cases on $v' = E(x)$. Because $E \preccurlyeq C$ and $v' \in E$, we have by Lemma A.32 that $\tau' <: \alpha \in C$ for some τ' such that $v' \preccurlyeq \tau'$.

If v' is non-onion, we have two cases. If $v' \in \pi$, then $n = 1$ and $v_1 = v'$. By Figure 4.4, we have that $\mathfrak{f}_1 = \mathfrak{f}'$ for all such cases of τ'. In each case, we can let $\mathfrak{f}_1 = \mathfrak{f}^*$; because $\mathfrak{f}^*$ is canonical, Definition A.36 gives us that this satisfies $C \vdash \alpha \overset{\pi}{\triangleright} \overset{n}{\overrightarrow{\mathfrak{f}}}; \mathfrak{f}'$.

Otherwise, suppose v' is non-onion but $v' \notin \pi$; then $n = 0$. In all such cases, Figure 4.4 gives us that $C \vdash \alpha \overset{\pi}{\triangleright} []; \mathfrak{f}'$ where $\mathfrak{f}' = \langle \tau'\,, \overset{m}{\overrightarrow{\mathfrak{f}''}}\rangle$. Inspection of Definition A.36 shows us that m is always the length of the canonical fibration's inner list; that is, a canonical fibration can always be used for $\mathfrak{f}'$. Because $\mathfrak{f}^*$ is canonical, we let $\mathfrak{f}' = \mathfrak{f}^*$ and this case is complete.

Otherwise, v' is an onion. If $v' =$ `()` then $\tau' =$ `()` and this result is trivial to show using the above approach.

Otherwise, $v' = x_1$ `&` x_2 and we know that $E{\downarrow}^*_\pi(x_1) = \overset{n''}{\overrightarrow{v''}}$ and $E{\downarrow}^*_\pi(x_2) = \overset{n'''}{\overrightarrow{v'''}}$ such that $\overset{n}{\overrightarrow{v}} = \overset{n''}{\overrightarrow{v''}} \parallel \overset{n'''}{\overrightarrow{v'''}}$. We also have that $\tau' = \alpha_1$ `&` α_2. Next, we induct on x_1 and x_2 to learn $C \vdash \alpha_1 \overset{\pi}{\triangleright} \overset{n''}{\overrightarrow{\langle \tau_\square''\,, \mathfrak{f}_\square''\rangle}}; \mathfrak{f}_1''''$ and $C \vdash \alpha_1 \overset{\pi}{\triangleright} \overset{n'''}{\overrightarrow{\langle \tau_\square'''\,, \mathfrak{f}_\square'''\rangle}}; \mathfrak{f}_2''''$ such that $\overset{n''}{\overrightarrow{\tau''}} \preccurlyeq \overset{n'''}{\overrightarrow{\tau'''}}$. By Definition A.36, $\langle \tau'\,, [\mathfrak{f}_1''''\,, \mathfrak{f}_2'''']\rangle$ is canonical for x and α as long as $\mathfrak{f}_1''''$ is canonical for x_1 and α_1 (and similarly for $\mathfrak{f}_2''''$). Thus, we are finished. □

Lemma A.38. Suppose that $E \preccurlyeq C$, $E{\downarrow}_\pi(x) = v$, and $x \preccurlyeq \alpha \mathbin{@} \mathfrak{f}'$. Then $C \vdash \alpha \overset{\pi}{\triangleright} \mathfrak{f}; \mathfrak{f}'$ and $v \preccurlyeq \tau$ (where $\mathfrak{f} = \langle \tau\,, \ldots\rangle$).

Proof. Immediate from Lemma A.37. □

Lemma A.39. Suppose $E \preccurlyeq C$, $\varphi \preccurlyeq \underbrace{\phi}$, and $x \preccurlyeq \alpha \mathbin{@} \mathfrak{f}$. If $\underbrace{x} \preceq_E \underbrace{\varphi} \setminus B\,; \underbrace{\varphi'}$, then $\alpha \preceq_C \phi \backslash \Gamma; \mathfrak{f}; \underbrace{\phi'}$ such that $B \preccurlyeq \Gamma$ and $\underbrace{\varphi'} \preccurlyeq \underbrace{\phi'}$.

$$
\begin{array}{ll}
x_1 \preceq_E^{\Circledcirc} {}^{n}\overrightarrow{y} \texttt{<-}\, \varphi \setminus \Circledcirc & \text{if } x_1 \preceq_E^{\Circledcirc} \varphi \setminus \Circledcirc; \emptyset \\
x_1 \preceq_E^{\Circledcirc} \varphi \setminus \Circledcirc; \overrightarrow{\varphi'} & \text{if } \varphi \in \{\texttt{int}, \texttt{fun}, \texttt{pat}, \texttt{scape}\}, \\
 & \quad E{\downarrow_{\varphi}}(x_1) = [] \\
x_1 \preceq_E^{\Circledcirc} l\,\varphi \setminus \Circledcirc; \overrightarrow{\varphi'} & \text{if } E{\downarrow_l^*}(x_1) = [] \\
x_1 \preceq_E^{\Circledcirc} l\,\varphi \setminus \Circledcirc; \overrightarrow{\varphi'} & \text{if } E{\downarrow_l}(x_1) = l\,x_2, x_2 \preceq_E^{\Circledcirc} \varphi \setminus \Circledcirc; \emptyset \\
x_1 \preceq_E^{\Circledcirc} \varphi_1 \,\&\, \varphi_2 \setminus \Circledcirc; \overrightarrow{\varphi'} & \text{if } x_1 \preceq_E^{\Circledcirc} \varphi_i \setminus \Circledcirc; \overrightarrow{\varphi'}, i \in \{1,2\} \\
x_1 \preceq_E^{\Circledcirc} \varphi_1 \setminus \Circledcirc; \overrightarrow{\varphi'} & \text{if } \varphi_1 \notin \overrightarrow{\varphi'}, \varphi_1 = x_2\, {}^{n}\overrightarrow{\varphi''}, \\
 & \quad E{\downarrow_{\texttt{pat}}^*}(x_2) = [] \\
x_1 \preceq_E^{\Circledcirc} \varphi_1 \setminus \Circledcirc; \overrightarrow{\varphi'} & \text{if } \varphi_1 \notin \overrightarrow{\varphi'}, \varphi_1 = x_2\, {}^{n}\overrightarrow{\varphi''}, \\
 & \quad E{\downarrow_{\texttt{pat}}}(x_2) = {}^{n}\overrightarrow{y} \texttt{<-}\, \varphi_2, \\
 & \quad \varphi_3 = \varphi_2\,[\varphi_1'' \,/\, y_1] \ldots [\varphi_n'' \,/\, y_n], \\
 & \quad x_1 \preceq_E^{\Circledcirc} \varphi_3 \setminus \Circledcirc; \{\varphi_1\} \cup \overrightarrow{\varphi'} \\
x_1 \preceq_E^{\Circledcirc} \varphi_1 \setminus \Circledcirc; \overrightarrow{\varphi'} & \text{if } x_1 \notin \overrightarrow{\varphi'}, \varphi_1 = \texttt{rec}\, y : \varphi_2, \\
 & \quad x_1 \preceq_E^{\Circledcirc} \varphi_2\,[\varphi_1 \,/\, y] \setminus \Circledcirc; \{\varphi_1\} \cup \overrightarrow{\varphi'} \\
x_1 \preceq_E^{\Circledcirc} \varphi \setminus \Circledcirc; \overrightarrow{\varphi'} & \text{if } \varphi \in \overrightarrow{\varphi'}
\end{array}
$$

Figure A.40. PatBang Value Compatibility Failures

Proof. By induction on the structure of the definition of $x \preceq_E \varphi \setminus B ; \overrightarrow{\varphi'}$.

Primitive patterns: If $\varphi \in \{\texttt{int}, \texttt{fun}, \texttt{pat}, \texttt{scape}\}$, then we know that $\varphi = \phi$, $B = \emptyset$, and $E{\downarrow_{\varphi}}(x) = v$ for some v. By Lemma A.38, we have that $C \vdash \alpha \mathrel{\not\sim\!\!\triangleright} \tau ; \mathfrak{f}$. This gives us that $\Gamma = \emptyset$ and so $B \preccurlyeq \Gamma$. A similar argument is made in the case of variable patterns (i.e. patterns of the form y).

Label patterns: If $\varphi = l\,\varphi''$, then we know that $E{\downarrow_l}(x) = l\,x'$ and that $x' \preceq_E \varphi'' \setminus B ; \emptyset$. By Definition A.14, we know that $\phi = l\,\phi''$ such that $\varphi'' \preccurlyeq \phi''$. By Lemma A.38, we have that $C \vdash \alpha \perp\!\!\!\triangleright \langle l\,\alpha', [\mathfrak{f}'] \rangle ; \mathfrak{f}''$ such that $x' \preccurlyeq \alpha'$. Definition A.36 gives us that $x' \preccurlyeq \alpha' \mathbin{@} \mathfrak{f}'$. We learn by induction that $\alpha' \preceq_C \phi'' \setminus \Gamma ; \mathfrak{f} ; \emptyset$. A similar argument is made in the case of conjunctive patterns.

Recursive patterns: The case for recursive and substitution patterns follows an argument similar to that of label patterns with the additional observation that structural substitution on pattern variables preserves simulation. □

Our next lemma is similar to Lemma A.39 but for the case of a match failure. But Figure 3.6 does not contain any failure cases analogous to those in Figure 4.7. We first define the value-level failure cases in Figure A.40 and then argue that the resulting relation is equivalent to the original. In keeping with notation presented in Section 4, we use $\dot{B}$ to range over B and the symbol $\Circledcirc$. We write $x_1 \preceq_E^{\Circledcirc} {}^{n}\overrightarrow{y} \texttt{<-}\, \varphi \setminus \dot{B}$ to distinguish this new relation from the old one.

Definition A.41 (Complete Value Compatibility)**.** We define value incompatibility as the least relation satisfying all clauses in Figure A.40. We define extended value compatibility $x_1 \preceq_E^* {}^{n}\overrightarrow{y} \texttt{<-}\, \varphi \setminus B$ as the union between this value incompatibility relation and the normal value compatibility relation from Section 3.2.

Lemma A.42. If $x_1 \preceq_E {}^{n}\overrightarrow{y} \texttt{<-}\, \varphi \setminus B$ then $x_1 \preceq_E^* {}^{n}\overrightarrow{y} \texttt{<-}\, \varphi \setminus B$. If no such B exists, then $x_1 \preceq_E^* {}^{n}\overrightarrow{y} \texttt{<-}\, \varphi \setminus \Circledcirc$.

Proof. By case analysis on the different forms of pattern. In each case, the original value compatibility relation relies on a set of premises. A clause exists in Figure 4.7 to address each of these failures by relating to $\Circledcirc$. □

We now prove alignment of incompatibility using this extended relation.

Lemma A.43. Suppose that $E \preccurlyeq C$ for some closed, well-formed E, that $\varphi \preccurlyeq \phi$, and that $x \preccurlyeq \alpha \mathbin{@} \mathfrak{f}$. If $x \preceq_E^* \varphi \setminus \Circledcirc; \overrightarrow{\varphi'}$ then $\alpha \preceq_C \phi \setminus \dot{\Gamma} ; \mathfrak{f} ; \overrightarrow{\phi'}$ such that $\overrightarrow{\varphi'} \preccurlyeq \overrightarrow{\phi'}$ and $\dot{\Gamma}$ is either $\Circledcirc$ or ☠.

Proof. By induction on the structure of $x \preceq_E^* \varphi \setminus \Circledcirc; \overrightarrow{\varphi'}$.

Primitive patterns: If $\varphi \in \{\texttt{int}, \texttt{fun}, \texttt{pat}, \texttt{scape}\}$, then we know that $\varphi = \phi$ by Definition A.14. Then we have that $E{\downarrow_{\varphi}}(x)$ is undefined and, in turn, that $E{\downarrow_{\varphi}^*}(x) = []$. By Lemma A.37, we have that $C \vdash \alpha \mathrel{\not\sim\!\!\triangleright} [] ; \mathfrak{f}$. This gives us $\alpha \preceq_C \phi \setminus \Circledcirc ; \mathfrak{f} ; \overrightarrow{\phi'}$ for any $\overrightarrow{\phi'}$ and this case is complete.

Label patterns: If $\varphi = l\,\varphi''$, then we know $\phi = l\,\phi''$. There are two cases. In the first case, $E{\downarrow_l}(x)$ is undefined. By Lemma A.37, this gives us that $C \vdash \alpha \perp\!\!\!\triangleright [] ; \mathfrak{f}$ and so, by Definition 4.8, that $\alpha \preceq_C \phi \setminus \Circledcirc ; \mathfrak{f} ; \overrightarrow{\phi'}$ for any $\overrightarrow{\phi'}$. Otherwise, $E{\downarrow_l}(x) = l\,x'$ and, by Lemma A.38, we have $C \vdash \alpha \perp\!\!\!\triangleright \langle l\,\alpha', [\mathfrak{f}'] \rangle ; \mathfrak{f}$. If so, then it must be that $x' \preceq_E^* \varphi' \setminus \Circledcirc; \overrightarrow{\varphi'}$; otherwise, the premise $x \preceq_E^* \varphi \setminus \Circledcirc; \overrightarrow{\varphi'}$ would be contradicted. By induction, we learn that $\alpha' \preceq_C \phi' \setminus \Circledcirc ; \mathfrak{f}' ; \overrightarrow{\phi'}$ such that $\overrightarrow{\varphi'} \preccurlyeq \overrightarrow{\phi'}$. We thus have that $\alpha \preceq_C \phi \setminus \Circledcirc ; \mathfrak{f} ; \overrightarrow{\phi'}$ and this case is complete.

The remaining cases follow the same structure as the above in parallel to Lemma A.39. □

Lemma A.44. Suppose that $E \preccurlyeq C$ for some closed, well-formed E, that ${}^{n}\overrightarrow{y} \texttt{<-}\, \varphi \preccurlyeq {}^{n}\overrightarrow{\beta} \texttt{<-}\, \phi$, and that $x \preccurlyeq \alpha \mathbin{@} \mathfrak{f}$. If $x \preceq_E {}^{n}\overrightarrow{y} \texttt{<-}\, \varphi \setminus {}^{m}\overrightarrow{v}$, then $\alpha \preceq_C {}^{n}\overrightarrow{\beta} \texttt{<-}\, \phi \setminus {}^{m}\overrightarrow{\tau} ; \mathfrak{f}$. If there exists no such ${}^{m}\overrightarrow{v}$, then either $\alpha \preceq_C {}^{n}\overrightarrow{\beta} \texttt{<-}\, \phi \setminus \Circledcirc ; \mathfrak{f}$ or $\alpha \preceq_C {}^{n}\overrightarrow{\beta} \texttt{<-}\, \phi \setminus \text{☠} ; \mathfrak{f}$.

Proof. Immediate from Lemmas A.39 and A.43 and Definition A.14. □

Lemma A.45. Suppose that $E \preccurlyeq C$ for some closed, well-formed E, that $x_1 \preccurlyeq \alpha_1 \mathbin{@} \mathfrak{f}$ and that ${}^{n}\overrightarrow{v} \preccurlyeq {}^{n}\overrightarrow{\tau}$. If $x_1 \not\preceq_E {}^{n}\overrightarrow{v} \setminus e_1$ then $\alpha_1 \not\preceq_C {}^{n}\overrightarrow{\tau} \setminus \alpha_4 ; C_1 ; \mathfrak{f}$ such that $e_1 \preccurlyeq \langle \alpha_4, C_1 \rangle$ or $\alpha_1 \not\preceq_C {}^{n}\overrightarrow{\tau} \setminus \alpha_4 ; \text{☠} ; \mathfrak{f}$. If no such e_1 exists, then $\alpha_1 \not\preceq_C \overrightarrow{\tau} \setminus \alpha' ; \text{☠} ; \mathfrak{f}$.

Proof. By induction on the length of the value list.

If $n = 0$ then $x_1 \not\preceq_E []$ and so this property is trivial.

Otherwise, let $v_n = x_2 \texttt{><} x_3$; thus, $\tau_n = \alpha_2 \texttt{><} \alpha_3$ with the appropriate simulation. If $E{\downarrow_{\texttt{pat}}}(x_2)$ is undefined or if $E{\downarrow_{\texttt{fun}}}(x_3)$ is undefined, then $x_1 \not\preceq_E {}^{n}\overrightarrow{v}$. In either case, $\alpha_1 \not\preceq_C \overrightarrow{\tau} \setminus \alpha' ; \text{☠} ; \mathfrak{f}$.

Otherwise, $E{\downarrow_{\texttt{pat}}}(x_2) = v'$ and $E{\downarrow_{\texttt{fun}}}(x_3) = {}^{m}\overrightarrow{x'} \texttt{->}\, e'$. By Lemma A.38, this gives us $C \vdash \alpha_2 \mathrel{\overset{\texttt{pat}}{\not\sim\!\!\triangleright}} \tau'$ such that $v' \preccurlyeq \tau'$ and also $C \vdash \alpha_3 \mathrel{\overset{\texttt{fun}}{\not\sim\!\!\triangleright}} {}^{m}\overrightarrow{\alpha'} ; \alpha_4\, C'$ such that $e' \preccurlyeq \langle \alpha_4, C' \rangle$. Then we have two cases: either the argument is compatible with that pattern or it is not.

If there exists no ${}^{m}\overrightarrow{v''}$ such that $x_1 \preceq_E v_n \setminus {}^{m}\overrightarrow{v''}$, then Lemma A.44 gives us the either $\alpha \preceq_C \tau_n \setminus \text{☠} ; \mathfrak{f}$ or $\alpha \preceq_C \tau_n \setminus \Circledcirc ; \mathfrak{f}$. In the former case, have that $\alpha_1 \not\preceq_C \overrightarrow{\tau} \setminus \alpha' ; \text{☠} ; \mathfrak{f}$. In the latter case, we proceed by induction using the first $n-1$ elements of each list.

Otherwise, $x_1 \preceq_E v_n \setminus {}^{m}\overrightarrow{v''}$. Then Lemma A.44 gives us that $\alpha_1 \preceq_C \tau_n \setminus {}^{m}\overrightarrow{\tau''} ; \mathfrak{f}$ such that simulation holds between these lists. Let $e'' = {}^{m}\overrightarrow{x'_{\square} = v''_{\square}}$ and let $C'' = {}^{m}\overrightarrow{\tau''_{\square} <: \alpha'_{\square}}$. We then observe that $e'' \preccurlyeq C''$ by Definition A.14 and by the simulations granted from the theorems above. By Lemma A.30, we have that $e'' \parallel e' \preccurlyeq C'' \cup C'$. Because the last variable of $e'' \parallel e'$ is the last variable of e', we also have $e'' \parallel e' \preccurlyeq \langle \alpha_4, C'' \cup C' \rangle$. We let $e_1 = e'' \parallel e'$ and $C_1 = C'' \cup C'$ and we are finished. □

Contour instantiation preserves the simulation relation. To demonstrate this property, we first prove a supplementary lemma.

Lemma A.46. Suppose that $e \preccurlyeq C$, that $\overline{x} \preccurlyeq \overline{\alpha}$, and that $\mathcal{C}_1 \preccurlyeq \mathcal{C}_2$. Then for any x' appearing in e, there is a corresponding α' appearing in C such that $\textsc{EInst}(x', \mathcal{C}_1, \overline{x}) \preccurlyeq \textsc{Inst}(\alpha', \mathcal{C}_2, \overline{\alpha})$.

Proof. For any x' in e, Lemma A.32 gives us that there is a corresponding α' in C such that $x' \preccurlyeq \alpha'$. By Definition A.9, we have that $x' \leftrightarrow \langle x'', \dot{\mathcal{C}}_1' \rangle$ and, by Definition A.14, that $\alpha' = \langle x'', \dot{\mathcal{C}}_2' \rangle$ such that $\dot{\mathcal{C}}_1' \preccurlyeq \dot{\mathcal{C}}_2'$.

Suppose that $x' \in \overline{x}$ and that $\dot{\mathcal{C}}_1' = \divideontimes$. By Definition A.11, $\textsc{EInst}(x', \mathcal{C}_1, \overline{x}) = x'$. Because $\dot{\mathcal{C}}_1' \preccurlyeq \dot{\mathcal{C}}_2'$, we know that $\dot{\mathcal{C}}_2' = \divideontimes$. Because $\overline{x} \preccurlyeq \overline{\alpha}$, we know that $\alpha' \in \overline{\alpha}$. By Definition A.8, $\textsc{Inst}(\alpha', \mathcal{C}_2, \overline{\alpha}) = \alpha'$. Since $x' \preccurlyeq \alpha'$, this case is complete.

Suppose that $x' \notin \overline{x}$ and that $\dot{\mathcal{C}}_1' = \divideontimes$. Then $\textsc{EInst}(x', \overline{x}, \mathcal{C}_1) \leftrightarrow \langle x'', \mathcal{C}_1 \rangle$. Because $\dot{\mathcal{C}}_1' \preccurlyeq \dot{\mathcal{C}}_2'$, we know that $\dot{\mathcal{C}}_2' = \divideontimes$. Because $\overline{x} \preccurlyeq \overline{\alpha}$, we know that $\alpha' \notin \overline{\alpha}$. Therefore, $\textsc{Inst}(\alpha', \overline{\alpha}, \mathcal{C}_2) = \langle x'', \mathcal{C}_2 \rangle$ and, by Definition A.11, we have $\langle x'', \mathcal{C}_1 \rangle \preccurlyeq \langle x'', \mathcal{C}_2 \rangle$ so this case is finished.

Otherwise, $\dot{\mathcal{C}}_1' \neq \divideontimes$. Then $\textsc{EInst}(x', \mathcal{C}_1, \overline{x}) = x'$. Because $x' \preccurlyeq \alpha'$, we know $\dot{\mathcal{C}}_2' \neq \divideontimes$. As a result, $\textsc{Inst}(\alpha', \mathcal{C}_2, \overline{\alpha}) = \alpha'$ and we are finished. □

Lemma A.47. Suppose that $e \preccurlyeq C$, that $\overline{x} \preccurlyeq \overline{\alpha}$, and that $\mathcal{C}_1 \preccurlyeq \mathcal{C}_2$. Then $\textsc{EInst}(e, \mathcal{C}_1, \overline{x}) \preccurlyeq \textsc{Inst}(C, \mathcal{C}_2, \overline{\alpha})$.

Proof. By induction on Definition A.14.

For variables, simulation is demonstrated by Lemma A.46. For all lower bound types other than functions and all constraints, simulation is parametric in variables and so is also demonstrated by Lemma A.46.

For function lower bound types, we must show that for any $v = \overrightarrow[n]{x'} \texttt{->} e'$ appearing in e, there is a corresponding $\tau = \overrightarrow[n]{\alpha'} \texttt{->} \alpha'' \backslash C'$ appearing in C such that $\textsc{EInst}(v, \mathcal{C}_1, \overline{x}) \preccurlyeq \textsc{Inst}(\tau, \mathcal{C}_2, \overline{\alpha})$. For any such v in e, Lemma A.32 gives us that such a τ exists in C and that $v \preccurlyeq C$. By Definition A.14, this means that $\overrightarrow[n]{x'} \preccurlyeq \overrightarrow[n]{\alpha'}$ and $e' \preccurlyeq \langle \alpha'', C' \rangle$. Since each of these terms is smaller than v or τ accordingly, we induct on the definition of this lemma. Thus, function lower bound types are also simulated.

By Definition A.14, we know that for each $s' \in e$, there is a corresponding $c' \in C$ such that $s' \preccurlyeq c'$. Suppose that $\textsc{EInst}(s', \mathcal{C}_1)\overline{x} = s''$, that $\textsc{Inst}(c', \mathcal{C}_2, \overline{\alpha}) = c''$, that $\textsc{EInst}(e, \mathcal{C}_1, \overline{x}) = e'$, and that $\textsc{Inst}(C, \mathcal{C}_2)\overline{\alpha} = C'$. Because we have that simulation of each constraint is preserved by instantiation, we have that $s'' \preccurlyeq c''$. Since this is true for each s and c, we have that each s in e' is simulated by some c in C' and so we are finished. □

Lemma A.48. If $e \preccurlyeq C$ and $\mathcal{C}_1 \preccurlyeq \mathcal{C}_2$, then $\textsc{EFresh}(e, \mathcal{C}_1) \preccurlyeq \textsc{Inst}(C, \mathcal{C}_2)$.

Proof. Immediate from Lemma A.47. □

Contour replacement also preserves simulation. We use a strategy similar to that above, first proving over variables.

Lemma A.49. If $x \preccurlyeq \alpha$ then $x \preccurlyeq \textsc{Repl}(\alpha, \mathcal{C})$.

Proof. Let $\textsc{Repl}(\alpha, \mathcal{C}) = \alpha'$. Then, let $x \leftrightarrow \langle x', \dot{\mathcal{C}}_1 \rangle$, let $\alpha = \langle x', \dot{\mathcal{C}}_2 \rangle$, and let $\alpha' = \langle x'', \dot{\mathcal{C}}_3 \rangle$. By Definition A.6, contour replacement is naturally homomorphic down to (but not including) contours and so we know that $x' = x''$. It remains to show that $\dot{\mathcal{C}}_1 \preccurlyeq \dot{\mathcal{C}}_3$.

Because $x \preccurlyeq \alpha$, we know that $\dot{\mathcal{C}}_1 = \divideontimes$ iff $\dot{\mathcal{C}}_2 = \divideontimes$. If $\dot{\mathcal{C}}_1 = \divideontimes$, then by this and Definition A.6 we have $\dot{\mathcal{C}}_3 = \divideontimes$ and this case is complete.

Otherwise, $\dot{\mathcal{C}}_1 = \mathcal{C}_1$ and $\dot{\mathcal{C}}_2 = \mathcal{C}_2$. By Definition A.6, there are only two possible values for $\dot{\mathcal{C}}_3$: $\mathcal{C}$ (if $\mathcal{C}_2 \leq \mathcal{C}$) or $\mathcal{C}_2$ (if $\mathcal{C}_2 \not\leq \mathcal{C}$). In the latter case, $\alpha = \alpha'$ and so $x \preccurlyeq \alpha'$.

Otherwise, $\dot{\mathcal{C}}_3 = \mathcal{C}$ and $\mathcal{C}_2 \leq \mathcal{C}$ and so it suffices to show that $\mathcal{C}_1 \leq \mathcal{C}$. By Lemma A.18, it suffices to show that $\mathcal{C}_1 \leq \mathcal{C}_2$ and that $\mathcal{C}_2 \leq \mathcal{C}$. The latter is shown from this case's premise; the prior is shown from Definition A.14 because we know $x \preccurlyeq \alpha$. □

Lemma A.50. If $E \preccurlyeq C$ then $E \preccurlyeq \textsc{Repl}(C, \mathcal{C})$.

Proof. Because contour replacement is a natural homomorphism in all cases not proven by Lemma A.49, this argument follows in parallel to that of Lemma A.47. □

A.10 Contour Properties of Functions and Relations

We now prove contour-related properties about functions and relations which are necessary to show simulation of constraint closure.

Lemma A.51. $\mathcal{C} \leq \textsc{Collapse}(\mathcal{C})$

Proof. This is to show that $[\![\mathcal{C}]\!] \subseteq [\![\{\textsc{Collapse}(\mathcal{S}) \mid \mathcal{S} \in \mathcal{C}\}]\!]$. By Definition A.5, this is to show that $\bigcup\{[\![\mathcal{S}]\!] \mid \mathcal{S} \in \mathcal{C}\} \subseteq \bigcup\{[\![\mathcal{S}]\!] \mid \mathcal{S} \in \textsc{Collapse}(\mathcal{C})\}$, which is equivalent to $\bigcup\{[\![\mathcal{S}]\!] \mid \mathcal{S} \in \mathcal{C}\} \subseteq \bigcup\{[\![\mathcal{S}]\!] \mid \mathcal{S} \in \{\textsc{Collapse}(\mathcal{S}) \mid \mathcal{S} \in \mathcal{C}\}\}$, which reduces to $\bigcup\{[\![\mathcal{S}]\!] \mid \mathcal{S} \in \mathcal{C}\} \subseteq \bigcup\{[\![\textsc{Collapse}(\mathcal{S})]\!] \mid \mathcal{S} \in \mathcal{C}\}$. It is therefore sufficient to prove that $[\![\mathcal{S}]\!] \subseteq [\![\textsc{Collapse}(\mathcal{S})]\!]$ for all $\mathcal{S}$.

We proceed by strong induction on the number of cycles in $\mathcal{S}$; by "cycle", we mean a pair of distinct parts $\mathcal{P}_1$ and $\mathcal{P}_2$ in $\mathcal{S}$ such that $\textsc{Sites}(\mathcal{P}_1) \cap \textsc{Sites}(\mathcal{P}_2) \neq \emptyset$. The base case is trivial, since in the absence of a cycle we have $\textsc{Collapse}(\mathcal{S}) = \mathcal{S}$. In the inductive case, let $\mathcal{S} = \overrightarrow[n]{\mathcal{P}}$. Let cycles in $\mathcal{S}$ be represented by pairs of indices (i, j) where $i < j$ and $\textsc{Sites}(\mathcal{P}_i) \cap \textsc{Sites}(\mathcal{P}_j) \neq \emptyset$; let $\overrightarrow[m]{(i_\square, j_\square)}$ be the set of all cycles in $\mathcal{S}$. Finally, let $\mathcal{S}' = \overrightarrow[m']{\mathcal{P}'} = [\mathcal{P}_1, \ldots, \mathcal{P}_{i_k - 1}, (\textsc{Sites}(\mathcal{P}_{i_k}) \cup \ldots \cup \textsc{Sites}(\mathcal{P}_{j_k})), \mathcal{P}_{j_k + 1}, \ldots, \mathcal{P}_n]$ for some $1 \leq k \leq m$. We can construct the set of cycles in $\mathcal{S}'$ by forming a mapping M from each index in $\mathcal{S}$ to indices in $\mathcal{S}'$ such that all $1 \leq i'' < i_k$ map to themselves, all $i_k \leq i'' \leq j_k$ map to i_k, and all $j_k < i'' \leq n$ map to themselves minus $j_k - i_k$. For each (i, j) in the cycles of $\mathcal{S}$, either $M[i] = M[j]$ or $(M[i], M[j])$ is a cycle in $\mathcal{S}'$; in particular, we know that $M[i_k] = M[j_k]$, which indicates that cycle k has been eliminated. From this, we have that the set of cycles in $\mathcal{S}'$ is strictly smaller than the set of cycles in $\mathcal{S}$; thus, it is sufficient by the induction hypothesis to show that $[\![\mathcal{S}]\!] \subseteq [\![\mathcal{S}']\!]$. By repeated application of Lemma A.21, it remains to show that $[\![[\mathcal{P}_{i_k}, \ldots, \mathcal{P}_{j_k}]]\!] \subseteq [\![[\textsc{Sites}(\mathcal{P}_{i_k}) \cup \ldots \cup \textsc{Sites}(\mathcal{P}_{j_k})]]\!]$.

It is sufficient to show that $[\![\overrightarrow[n]{\mathcal{P}}]\!] \subseteq [\![[\overrightarrow[n]{\bigcup \textsc{Sites}(\mathcal{P}_\square)}]]\!]$. This is immediate; the latter contains every call sequence composed of any elements in $\overrightarrow[n]{\mathcal{P}}$ and the prior contains no call sequences containing elements not in $\overrightarrow[n]{\mathcal{P}}$. □

Lemma A.52. $\forall C.\, \mathcal{C} \leq \textsc{Widen}(\mathcal{C}, C)$

Proof. For any C, we have that $\textsc{Widen}(\mathcal{C}, C) = \mathcal{C}' \supseteq \mathcal{C}$ by inspection. By definition, $\mathcal{C} \leq \mathcal{C}'$ iff $[\![\mathcal{C}]\!] \subseteq [\![\mathcal{C}']\!]$. This reduces to $(\bigcup\{[\![\mathcal{S}]\!] \mid \mathcal{S} \in \mathcal{C}\}) \subseteq (\bigcup\{[\![\mathcal{S}]\!] \mid \mathcal{S} \in \mathcal{C}'\})$. Because $\mathcal{C} \subseteq \mathcal{C}'$, every element in the comprehension on the left also appears in the comprehension on the right; thus, $\mathcal{C} \leq \mathcal{C}'$. □

Lemma A.53. If $\textsc{Widen}(\mathcal{C}, C) = \mathcal{C}'$, then $\forall\, \mathcal{C}'' \in (\!|C|\!).\, \mathcal{C}'' \between \mathcal{C}' \implies \mathcal{C}'' \leq \mathcal{C}'$.

Proof. By definition, $\mathcal{C}' = \mathcal{C} \cup (\bigcup \{\mathcal{C}'' \mid \mathcal{C}'' \in (\!|C|\!) \wedge \mathcal{C} \between \mathcal{C}''\})$. We therefore have for any $\mathcal{C}''$ in $(\!|C|\!)$ that $\mathcal{C}'' \subseteq \mathcal{C}'$. By Definition A.2, we thus have that $[\![\mathcal{C}'']\!] \subseteq [\![\mathcal{C}']\!]$ and so $\mathcal{C}'' \leq \mathcal{C}'$. □

Lemma A.54. If $C \vdash \alpha \xrightarrow{\pi}\triangleright \overrightarrow{\tau}$ for some well-formed C, then $(\!|\overrightarrow{\tau}|\!) \subseteq (\!|\alpha|\!) \cup (\!|C|\!)$.

Proof. By inspection. Each rule produces a type which is either (1) devoid of contours, (2) constructed from τ, (3) constructed from constraints found in C, or (4) derived from another projection. □

Lemma A.55. If $\alpha \preceq_C \overrightarrow{\beta}$ `<-` $\phi \setminus \overrightarrow{\tau}$; `f`, then $(\!|\overrightarrow{\tau}|\!) \subseteq (\!|C|\!)$.

Proof. By inspection. The types in $\overrightarrow{\tau}$ are a subset of those in the bindings set. The only case in which a binding is added to the bindings set is the variable pattern case, which obtains its type directly from the constraint set C. □

Lemma A.56. If $\textsc{Inst}(C, \mathcal{C}, \overline{\alpha}) = C'$, then $(\!|C'|\!) \subseteq (\!|C|\!) \cup \{\mathcal{C}\}$.

Proof. Let $\alpha' = \textsc{Inst}(\alpha'', \mathcal{C}, \overline{\alpha})$. We observe that $(\!|\alpha'|\!) \subseteq (\!|\alpha''|\!) \cup \{\mathcal{C}\}$ by case analysis. Because contour instantiation is largely naturally homomorphic, the remainder of this argument proceeds in parallel to Lemma A.47. □

Lemma A.57. If $\textsc{Repl}(C, \mathcal{C}) = C'$, then $(\!|C'|\!) \subseteq \{\mathcal{C}\} \cup \{\mathcal{C}' \mid \mathcal{C}' \in (\!|C|\!) \wedge \mathcal{C}' \not\leq \mathcal{C}\}$.

Proof. Let $\alpha' = \textsc{Repl}(\alpha, \mathcal{C})$. We first prove that $(\!|\alpha'|\!) \subseteq \{\mathcal{C}\} \cup \{\mathcal{C}' \mid \mathcal{C}' \in (\!|\alpha|\!) \wedge \mathcal{C}' \not\leq \mathcal{C}\}$. This is done by simple case analysis. If the contour of α is $\divideontimes$ then contour replacement is a natural homomorphism and so $\alpha' = \alpha$; thus, $(\!|\alpha'|\!) = \{\mathcal{C}\}$. Otherwise, let the contour of α be $\mathcal{C}'$ and let the contour of α be $\mathcal{C}''$. If $\mathcal{C}' \leq \mathcal{C}$, then $\mathcal{C}'' = \mathcal{C}$; thus we have $(\!|\alpha'|\!) = \{\mathcal{C}\}$. If $\mathcal{C}' \not\leq \mathcal{C}$, then $\mathcal{C}'' = \mathcal{C}'$; thus we have $(\!|\alpha'|\!) = \{\mathcal{C}'\} \subseteq \{\mathcal{C}\} \cup \{\mathcal{C}'\}$.

Because contour replacement is largely naturally homomorphic, the remainder of this argument proceeds in parallel to Lemma A.47. □

A.11 Closure Simulation

We now show our Preservation Lemma. Informally, this lemma says that, for any expression e simulated by a constraint set C, every legal small step corresponds to some legal type closure step such that simulation still holds over the new expression and closure set. This is formalized as follows:

Lemma A.58 (Preservation)**.** Suppose $e \preccurlyeq C$, that C is well-formed, and that $e \longrightarrow^1 e'$. Then either (1) C is inconsistent or (2) there exists some C' such that $C \Longrightarrow^1 C'$, $e' \preccurlyeq C'$, and C' is well-formed.

We now give a proof of Lemma A.58.

Proof. By case analysis on the operational semantics small step rule. Our strategy for each case is first to show that at least one type closure rule applies for each small step. We then show that the resulting expression is simulated by the resulting constraint set.

Assignment: If $e = E \,\|[x = x']\,\| \, e_{\textsc{Rest}}$, then $e' = E \,\|[x = v]\,\| \, e_{\textsc{Rest}}$ where $E(x') = v$. By Lemma A.32, we have that $\tau <: \alpha' \in C$ where $v \preccurlyeq \tau$ and $x' \preccurlyeq \alpha'$. The same lemma gives us that $\alpha' <: \alpha \in C$ such that $x \preccurlyeq \alpha$ (because $x = x' \in e$). We thus have that $C' = C \cup \{\tau <: \alpha\}$.

By Lemma A.29, we have that $E \,\| \, e_{\textsc{Rest}} \preccurlyeq C$. Lemma A.30 gives us that $e' \preccurlyeq C'$. To show that C' is well-formed, we observe that the only contour appearing in $\{\tau <: \alpha\}$ is the one in α, which is a type variable already appearing in C; by Lemma A.26, C' is well-formed.

Integer Addition: If $e = E \,\|[x = x_1 \texttt{ + } x_2]\,\| \, e_{\textsc{Rest}}$, then $e' = E \,\|[x = n]\,\| \, e_{\textsc{Rest}}$ for some n. We also have that $E\downarrow_{\texttt{int}}(x_1) = n_1$, that $E\downarrow_{\texttt{int}}(x_2) = n_2$, and that $n = n_1 + n_2$. Because addition is well-defined, we further have that both n_1 and n_2 are elements of $\mathbb{Z}$.

We now show that the Integer Addition closure rule applies in this case. Because $x = x_1 \texttt{ + } x_2$ appears in e and because $e \preccurlyeq C$, Lemma A.32 gives us that there exists some $\alpha_1 \texttt{ + } \alpha_2 <: \alpha$ in C such that $x_1 \preccurlyeq \alpha_1$, $x_2 \preccurlyeq \alpha_2$, and $x \preccurlyeq \alpha$. Because $\pi\, \texttt{int} x_1 = n_1$ and $x_1 \preccurlyeq \alpha_1$, Lemma A.37 gives us that $C \vdash \alpha_1 \xrightarrow{\texttt{int}}\triangleright \tau_1$ and $n_1 \preccurlyeq \tau_1$; the same statement can be made regarding x_2, α_2, n_2, and τ_2.

Because we have shown that the Integer Addition closure rule applies, we know that $C \Longrightarrow^1 C'$ where $C' = C \cup \{\texttt{int} <: \alpha\}$. It remains to show that $e' \preccurlyeq C'$ and that C' is well-formed. We show both of these properties in the same fashion as the Assignment case above.

True Integer Equality: Suppose $e = E \,\|[x = x_1 \texttt{ == } x_2]\,\| \, e_{\textsc{Rest}}$. If $E\downarrow_{\texttt{int}}(x_1) = n$ and that $E\downarrow_{\texttt{int}}(x_2) = n'$ for some $n, n' \in \mathbb{Z}$ where $n = n'$, then $e' = E \,\|[x' = \texttt{()}, x = \texttt{`True } x']\,\| \, e$ for a fresh x' chosen such that, when $x \preccurlyeq \alpha$, we have $x' \preccurlyeq \textsc{fvar}(\alpha)$.

We now show that the Integer Equality closure rule applies in this case. Because $x = x_1 \texttt{ == } x_2$ appears in e and because $e \preccurlyeq C$, Lemma A.32 gives us that there exists some $\alpha_1 \texttt{ == } \alpha_2 <: \alpha$ in C such that $x_1 \preccurlyeq \alpha_1$, $x_2 \preccurlyeq \alpha_2$, and $x \preccurlyeq \alpha$. Because $E\downarrow_{\texttt{int}}(x_1) = n_1$ and $x_1 \preccurlyeq \alpha_1$, Lemma A.37 gives us that $C \vdash \alpha_1 \xrightarrow{\texttt{int}}\triangleright \tau_1$ and $n_1 \preccurlyeq \tau_1$; the same statement can be made regarding x_2, α_2, n_2, and τ_2.

Because we have shown that the Integer Equality closure rule applies, we know that $C \Longrightarrow^1 C'$ where $C' = C \cup \{\texttt{`True } \alpha' <: \alpha, \texttt{`False } \alpha' <: \alpha, \texttt{()} <: \alpha'\}$ where $\alpha' = \textsc{fvar}(\alpha)$. By Definition A.14 and because $x' \preccurlyeq \textsc{fvar}(\alpha)$, we have that $[x' = \texttt{()}, x = \texttt{`True } x'] \preccurlyeq \{\texttt{`True } \alpha' <: \alpha, \texttt{()} <: \alpha'\}$; by Lemma A.30, we have that $[x' = \texttt{()}, x = \texttt{`True } x'] \preccurlyeq \{\texttt{`True } \alpha' <: \alpha, \texttt{`False } \alpha' <: \alpha, \texttt{()} <: \alpha'\}$. By Lemma A.30, we therefore have that $e' \preccurlyeq C'$. It remains to show that C' is well-formed; this is demonstrated by Lemma A.26 as above.

A similar argument is made when $n \neq n'$ using `'False` rather than `'True`.

Application: If $e = E \,\|[x = x_1 \;\; x_2]\,\| \, e_{\textsc{Rest}}$, then $e' = E \,\|\, \textsc{efresh}(x, e'' \,\|[x' = r])\,\|[x = \textsc{efresh}(x, x')]\,\| \, e_{\textsc{Rest}}$ where $E\downarrow^*_{\texttt{scape}}(x_1) = \overrightarrow{v}$ and $x_2 \preceq_E \overrightarrow{v} \setminus e' \,\|[x' = r]$.

We now show that the Application closure rule applies in this case. Because $x = x_1 \;\; x_2$ appears in e and because $e \preccurlyeq C$, Lemma A.32 gives us that there exists some $\alpha_1 \;\; \alpha_2 <: \alpha$ in C such that $x_1 \preccurlyeq \alpha_1$, $x_2 \preccurlyeq \alpha_2$, and $x \preccurlyeq \alpha$. Because $E\downarrow^*_{\texttt{fun}}(x_1) = \overrightarrow{v}$ and $x_1 \preccurlyeq \alpha_1$ and by Lemma A.37, we have that there exists some $\overrightarrow{\tau}$ such that $\overrightarrow{v} \preccurlyeq \overrightarrow{\tau}$ and $C \vdash \alpha \xrightarrow{\texttt{fun}}\triangleright \overrightarrow{\tau}$. Let $e_{\textsc{New}} = e' \,\|[x' = r]$.

Because $x_2 \preceq_E \overrightarrow{v} \setminus e_{\textsc{New}}$, Lemma A.45 gives us that either (1) $\alpha_2 \preceq_C \overrightarrow{\tau} \setminus \alpha_0$; ☠ or (2) there exists some $\alpha_{\textsc{Last}}$ and $C_{\textsc{New}}$ such that $\alpha_2 \preceq_C \overrightarrow{\tau} \setminus \alpha_{\textsc{Last}}$; $C_{\textsc{New}}$ and $e_{\textsc{New}} \preccurlyeq \langle \alpha_{\textsc{Last}}, C_{\textsc{New}} \rangle$. In the prior case, we know by Definition 4.13 that C is inconsistent and we are finished. In the latter case, because both contour creation and contour instantiation are total functions, the Application closure rule applies.

Let $x_{\textsc{Last}} = r_{\textsc{Last}}$ be the last clause of $e_{\textsc{New}}$. Then $e_{\textsc{New}} \preccurlyeq C_{\textsc{New}}$ and $x_{\textsc{Last}} \preccurlyeq \alpha_{\textsc{Last}}$. Because $x \preccurlyeq \alpha$, we have by Definition A.14 that $[x = x_{\textsc{Last}}] \preccurlyeq \{\alpha_{\textsc{Last}} <: \alpha\}$.

Let $\mathcal{C} = \textsc{cnew}(\alpha_2, C)$, let $e'_{\textsc{New}} = \textsc{efresh}(e_{\textsc{New}}, \mathcal{C})$, and let $C'_{\textsc{New}} = \textsc{Inst}(C_{\textsc{New}}, \mathcal{C})$. By Lemma A.48, we have $e'_{\textsc{New}} \preccurlyeq C'_{\textsc{New}}$. Let $\textsc{efresh}(x_{\textsc{Last}}, \mathcal{C}) = x'_{\textsc{Last}}$. Let $C'' = \textsc{Inst}(\{\alpha_{\textsc{Last}} <: \alpha\}, \mathcal{C})$. By Definition A.8, we have that $C'' = \{\textsc{Inst}(\alpha_{\textsc{Last}}, \emptyset, \mathcal{C}) <: \textsc{Inst}(\alpha, \emptyset, \mathcal{C})\}$. The contour of α is not $\divideontimes$ as it appears at top level in C ; thus, $C'' = \{\alpha'_{\textsc{Last}} <: \alpha\}$ where $\alpha'_{\textsc{Last}} = \textsc{Inst}(\alpha_{\textsc{Last}}, \emptyset, \mathcal{C})$.

By Lemma A.48, we thus have that $[x = x'_{\text{LAST}}] \preccurlyeq C''$; thus by Lemma A.30 we have that $e'_{\text{NEW}} \,\|\, [x = x'_{\text{LAST}}] \preccurlyeq C'_{\text{NEW}} \cup C''$. It should be observed that the prior represents the expression which replaces the application at the call site in the operational semantics; the latter represents the set of constraints after instantiation in the Application Closure rule. We let $C_{\text{NEW}} \cup C'' = C'''$ and we let $e'_{\text{NEW}} \,\|\, [x = x'_{\text{LAST}}] = e'''$.

Finally, we show simulation by proceeding with an argument similar to the Assignment case above. By Lemma A.29, we have that $E \,\|\, e_{\text{REST}} \preccurlyeq C$; thus by Lemma A.30 we have that $E \,\|\, \text{EFRESH}(x, e''') \,\|\, e_{\text{REST}} \preccurlyeq C \cup C'''$. Finally, we let $C' = \text{REPL}(C \cup C''', \mathcal{C})$ and by Lemma A.50, we have $e \,\|\, e''' \,\|\, e_{\text{REST}} \preccurlyeq C'$.

It remains to show that $\text{REPL}(C \cup C'', \mathcal{C})$ is well-formed. We know from premises that C is well-formed. Our strategy is to use the extractions of the contour sets involved in the application rule to show that each step in this process is also well-formed. Because α_1 appears in C, we have from Lemma A.54 that $(\!|\vec{\tau}|\!) \subseteq (\!|C|\!)$. From this, because α_2 appears in C and by Lemma A.55, we have that $(\!|C_{\text{NEW}}|\!) \subseteq (\!|C|\!)$.

By Lemma A.56, we have that $(\!|C'_{\text{NEW}}|\!) \subseteq (\!|C|\!) \cup \{\mathcal{C}\}$. We also have that $(\!|C''|\!) \subseteq (\!|C|\!) \cup \{\mathcal{C}\}$. This gives us $(\!|C'_{\text{NEW}} \cup C''|\!) \subseteq (\!|C|\!) \cup \{\mathcal{C}\}$ by Lemma A.19, which is equivalent to $(\!|C'''|\!) \subseteq (\!|C|\!) \cup \{\mathcal{C}\}$. Again by Lemma A.19, we have $(\!|C \cup C'''|\!) \subseteq (\!|C|\!) \cup \{\mathcal{C}\}$, which is equivalent to $(\!|C'|\!) \subseteq (\!|C|\!) \cup \{\mathcal{C}\}$. By Lemma A.57, we have that $(\!|C'|\!) \subseteq \{\mathcal{C}\} \cup \{\mathcal{C}' \mid \mathcal{C}' \in (\!|C \cup C'''|\!) \wedge \mathcal{C}' \not\leq \mathcal{C}\}$. Because $(\!|C \cup C'''|\!) \subseteq (\!|C|\!) \cup \{\mathcal{C}\}$, this can be weakened to $(\!|C'|\!) \subseteq \{\mathcal{C}\} \cup \{\mathcal{C}' \mid \mathcal{C}' \in (\!|C|\!) \cup \{\mathcal{C}\} \wedge \mathcal{C}' \not\leq \mathcal{C}\}$; since $\mathcal{C} \leq \mathcal{C}$, this simplifies to $(\!|C'|\!) \subseteq \{\mathcal{C}\} \cup \{\mathcal{C}' \mid \mathcal{C}' \in (\!|C|\!) \wedge \mathcal{C}' \not\leq \mathcal{C}\}$. For notational convenience, let $\overline{\mathcal{C}} = \{\mathcal{C}' \mid \mathcal{C}' \in (\!|C|\!) \wedge \mathcal{C}' \not\leq \mathcal{C}\}$. It remains to show that $(\!|C'|\!)$ is well-formed; by Lemma A.26 it is sufficient to show that $\{\mathcal{C}\} \cup \overline{\mathcal{C}}$ is well-formed.

Since $\overline{\mathcal{C}}$ is a subset of $(\!|C|\!)$ and C is well-formed, it is sufficient to show that $\forall\, \mathcal{C}' \in \overline{\mathcal{C}}\,.\, \mathcal{C}' \not\between \mathcal{C}$. By Lemma A.53, we have that for every $\mathcal{C}'$ in $(\!|C|\!)$, we have either $\mathcal{C}' \not\between \mathcal{C}$ or $\mathcal{C}' \leq \mathcal{C}$. By definition, $\overline{\mathcal{C}}$ contains only those $\mathcal{C}'$ such that $\mathcal{C}' \not\leq \mathcal{C}$; thus, we know that $\forall\, \mathcal{C}' \in \overline{\mathcal{C}}\,.\, \mathcal{C}' \not\between \mathcal{C}$. Therefore, $\overline{\mathcal{C}} \cup \{\mathcal{C}\}$ is well-formed and so C' is well-formed. □

To complete our simulation of closure, we show stuck expressions are modeled by simulation as inconsistencies in the constraint set.

Lemma A.59. If $e \preccurlyeq C$, there exists no e' such that $e \longrightarrow^1 e'$, and e is not of the form E, then C is inconsistent.

Proof. We begin by observing that every e not of the form E must be factorable into some $E \,\|\, [x = r] \,\|\, e''$ where r is not of the form v. We proceed by case analysis on r and the conditions of Definition 3.8 which lead to a stuck e. In each case, we show that the corresponding constraint set is inconsistent.

If r is of the form $x_1 \circledcirc x_2$, we observe that e is only stuck when $E \downarrow^*_{\texttt{int}}(x_i) = []$ for $i = 1$ or $i = 2$. By Definition 3.5, this implies that $E \downarrow^*_{\texttt{int}}(x_i) = []$. Because $[x = x_1 \circledcirc x_2]$ is in e, Lemma A.32 gives us that $\alpha_1 \circledcirc \alpha_2 <: \alpha \in C$ for $x_1 \preccurlyeq \alpha_1$, $x_2 \preccurlyeq \alpha_2$, and $x \preccurlyeq \alpha$. Because $x_i \preccurlyeq \alpha_i$, Lemma A.37 gives us that $C \vdash \alpha_i \xrightarrow{\texttt{int}\triangleright} []$. The presence of the constraint $\alpha_1 \circledcirc \alpha_2 <: \alpha$ in C such that $C \vdash \alpha_i \xrightarrow{\texttt{int}\triangleright} []$ is inconsistent by Definition 4.13.

Otherwise, r is of the form $x_1\ x_2$. Let $E \downarrow^*_{\texttt{scape}}(x_1) = \vec{v}$. In this case, e is only stuck if $x_2 \not\in_E \vec{v}$. By Lemma A.32, we know that $\alpha_1\ \alpha_2 <: \alpha \in C$ with the appropriate simulations. By Lemma A.45, we have that $\alpha_2 \preccurlyeq_C \vec{\tau} \setminus \alpha_3$; 💀 where $\vec{v} \preccurlyeq \vec{\tau}$. By Definition 4.13, this is inconsistent. □

A.12 Proof of Soundness

We now prove Theorem 4.15. Our strategy for doing so is to show that simulation holds after the initial derivation and after each small step evaluation. Once small step evaluation is complete, we use this simulation to show that evaluation can only be stuck if the constraint set is inconsistent.

Proof. Given a closed, well-formed e, we have by Lemma A.35 that, if $[\![e]\!]_{\text{E}} = \langle \alpha\,, C \rangle$, then C is well-formed and $e \preccurlyeq \langle \alpha\,, C \rangle$. Since every type variable created by initial derivation has no contour, $(\!|C|\!) = \emptyset$. Let $e' = \text{EFRESH}(e\,, \{[]\})$ and let $C' = \text{INST}(C, \{[]\})$; by Lemma A.48, we have that $e' \preccurlyeq C'$. Observe that, by Definition A.12, e' is α-equivalent to e and so gets stuck iff e gets stuck.

We next induct on the length of constraint closure. Let $e_0 = e'$ and let $C_0 = C'$. Further, let $e_0 \longrightarrow^* e_n$ for some $n \geq 0$. We start with the base case that $e_0 \preccurlyeq C_0$ and that C_0 is well-formed. We use Lemma A.58 to prove our inductive step: if $e_i \preccurlyeq C_i$ and C_i is well-formed, then either C_i is inconsistent (and we are finished) or there exists some $C_i \Longrightarrow^1 C_{i+1}$ such that $e_{i+1} \preccurlyeq C_{i+1}$ and C_{i+1} is well-formed. We therefore have by induction that $e_n \preccurlyeq C_n$ and that C_n is well-formed.

We have by premise that $e_n \not\longrightarrow^1$ and that e_n is not of the form E. By Lemma A.59, we have that C_n is inconsistent. By Definition 4.14, e does *not* typecheck. □

B. Proof of Termination

We show here that the PatBang typechecking definition presented in Section 4 is decidable. We first present a proof of decidability via a counting argument. We then informally discuss how typechecking can be made computationally feasible.

B.1 Proof of Termination

We begin by showing each relation used in defining the type system is decidable, and each function is computable. We then show that the number of constraints which may appear in any closure is bounded in terms of the size of the original program and that the number of closure steps necessary to decide typechecking is finite.

Lemma B.1. Given a closed e, $[\![e]\!]_{\text{E}}$ is a computable function.

Proof. By direct induction on the definition in Figure 4.2. □

The decidability of the projection relation and computability of a function which finds all $\vec{\mathfrak{f}}$ and $\mathfrak{f}'$ that can be projected are complex. This is due to the (rare in practice) case where onions can directly recurse without an intervening label. Since onioning is a form of type intersection, such recursions are non-contractive and are usually syntactically ruled out as they have no well-defined semantics [15], but we support non-contractive recursions. The following series of Definitions and Lemmas give us this decidability and computability result.

We begin by defining a relation similar to the projection shown in Figure 4.4 and then align this new definition with the old version to show the decision procedure. The definition below uses the notation $\epsilon_\pi(\alpha)$ to indicate that a type variable α is *nullable*: it may produce an empty list for the given projector π.

Definition B.2 (Nullable)**.** $\epsilon_\pi(\alpha)$ if and only if either (1) there exists a τ not of the form $\alpha_1 \,\&\, \alpha_2$ such that $\tau <: \alpha$ appears in the constraint set and τ does not match π or (2) $\alpha_1 \,\&\, \alpha_2 <: \alpha$ appears in the constraint set and inductively both $\epsilon_\pi(\alpha_1)$ and $\epsilon_\pi(\alpha_2)$ hold.

Lemma B.3. Relation $\epsilon_\pi(\alpha)$ is decidable.

Proof. There are finitely many & constraints in C and it suffices never to visit the same & constraint twice. This latter is because

$$
\begin{array}{lll}
C \vdash^{\epsilon}_{0} \alpha \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}; \mathfrak{f}' & \text{if} & C \vdash \alpha \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}; \mathfrak{f}', \\
 & & \mathfrak{f}' = \langle \tau, \overrightarrow{\mathfrak{f}''} \rangle, \tau \text{ is non-onion} \\
C \vdash^{\epsilon}_{0} \alpha \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}_1 \parallel \overrightarrow{\mathfrak{f}}_2; \mathfrak{f}'' & \text{if} & \tau' <: \alpha \in C, \tau' = \alpha_1 \,\&\, \alpha_2, \\
 & & C \vdash^{\epsilon}_{0} \alpha_1 \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}_1; \mathfrak{f}'_1, C \vdash^{\epsilon}_{0} \alpha_2 \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}_2; \mathfrak{f}'_2, \\
 & & \mathfrak{f}'' = \langle \tau', [\mathfrak{f}'_1, \mathfrak{f}'_2] \rangle \\
C \vdash^{\epsilon}_{0} \alpha \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}_2; \mathfrak{f}'' & \text{if} & \tau' <: \alpha \in C, \tau' = \alpha_1 \,\&\, \alpha_2, \epsilon_\pi(\alpha_1), \\
 & & C \vdash^{\epsilon}_{0} \alpha_2 \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}_2; \mathfrak{f}'_2, \\
 & & \mathfrak{f}'' = \langle \tau', [\mathfrak{f}'_1, \mathfrak{f}'_2] \rangle \\
C \vdash^{\epsilon}_{0} \alpha \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}_1 \parallel \overrightarrow{\mathfrak{f}}_2; \mathfrak{f}'' & \text{if} & \tau' <: \alpha \in C, \tau' = \alpha_1 \,\&\, \alpha_2, \epsilon_\pi(\alpha_2), \\
 & & C \vdash^{\epsilon}_{0} \alpha_1 \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}_1; \mathfrak{f}'_1, \mathfrak{f}'' = \langle \tau', [\mathfrak{f}'_1, \mathfrak{f}'_2] \rangle \\
C \vdash^{\epsilon} \alpha \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}; \mathfrak{f}' & \text{if} & C \vdash^{\epsilon}_{0} \alpha \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}; \mathfrak{f}' \text{ or both } \epsilon_\pi(\alpha) \text{ and } \overrightarrow{\tau} = []
\end{array}
$$

Figure B.4. PatBang Modified Projection

each cyclic sequence of such constraints gives rise to the same premises as some set of acyclic sequences. It therefore suffices to consider all acyclic sequences and, because there is a fixed, finite number of constraints in C, there is a fixed, finite set of such acyclic sequences. □

B.1.1 Projection

Lemma B.5. Relation $C \vdash \alpha \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}; \mathfrak{f}'$ is decidable.

Proof. We first show that the modified projection relation $C \vdash^{\epsilon} \alpha \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}; \mathfrak{f}'$ at the bottom of Figure B.4 is equivalent to the original projection relation $C \vdash \alpha \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}; \mathfrak{f}'$ in Figure 4.4. The modified definition $C \vdash^{\epsilon}_{0} \alpha \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}; \mathfrak{f}'$ at the top of Figure B.4 by inspection inlines the empty list clause (the first clause) of the original definition into the & clause, and that constructively produces a non-empty list in each case. Thus, the only case where $C \vdash \alpha \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}; \mathfrak{f}'$ / $C \vdash^{\epsilon}_{0} \alpha \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}; \mathfrak{f}'$ differ is where the [] clause is the top-level result, and $C \vdash^{\epsilon} \alpha \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}; \mathfrak{f}'$ explicitly aligns that top-level behavior.

The modified relation $C \vdash^{\epsilon} \alpha \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}; \mathfrak{f}'$ is now shown to be decidable, and by the above equivalence, the original relation is decidable. Observe that the list $\overrightarrow{\mathfrak{f}}$ is divided into two non-empty sublists in any use of the & clause, so it is possible to exhaustively test each of the n possible factorings of a length n list, and $\epsilon_\pi(\alpha)$ was shown decidable in Lemma B.3. At the leaf case we rely on projection matching, which is trivially decidable. □

The notational sugar $C \vdash \alpha \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}$ implies a quite significant implication; it is equivalent to the statement $\exists \mathfrak{f}'. C \vdash \alpha \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}}; \mathfrak{f}'$. For decidability, it is critical to ensure that such a fibration can be constructed if it exists. We therefore present the following lemma:

Now we show that the set of all possible $\overrightarrow{\mathfrak{f}}$ and $\mathfrak{f}'$ pairs that project from a given type variable is computable. We show this by showing how the equivalent set of lists with duplicate types in each list removed, is computable. $\textsc{Dedup}(\cdot, \cdot)$ is a function which removes all duplicate types in a list of types, preserving the rightmost occurrence in the list only. Formally:

Definition B.6 (Deduplication)**.**

$$
\begin{array}{rcl}
\textsc{Dedup}(\overrightarrow{\mathfrak{f}}) & = & \textsc{Dedup}(\emptyset, \overrightarrow{\mathfrak{f}}) \\
\textsc{Dedup}(\overleftarrow{\mathfrak{f}'}, []) & = & [] \\
\textsc{Dedup}(\overleftarrow{\mathfrak{f}'}, \overrightarrow{\mathfrak{f}} \parallel [\mathfrak{f}'']) & = & \begin{cases} \textsc{Dedup}(\overleftarrow{\mathfrak{f}'}, \overrightarrow{\mathfrak{f}}) \text{ when } \mathfrak{f}'' \in \overleftarrow{\mathfrak{f}'} \\ \textsc{Dedup}(\overleftarrow{\mathfrak{f}'} \cup \{\mathfrak{f}''\}, \overrightarrow{\mathfrak{f}} \parallel [\mathfrak{f}'']) \text{ otherwise} \end{cases}
\end{array}
$$

Lemma B.7. There is a computable function which takes a C, π, and α as arguments and returns a set $\overbrace{\langle \overrightarrow{\mathfrak{f}}, \mathfrak{f}' \rangle}$ such that for all $\overrightarrow{\mathfrak{f}''}$ we have $\textsc{Dedup}(\overrightarrow{\mathfrak{f}''}) \in \overbrace{\overrightarrow{\mathfrak{f}}}$ if and only if $C \vdash \alpha \ \overset{\pi}{\triangleright} \overrightarrow{\mathfrak{f}''}; \mathfrak{f}'$.

Proof. The core of the algorithm is to perform non-deterministic disjunctive computation on each lower bound encountered (subject to some restrictions for termination). If we encounter a non-onion type under no other restrictions, computation is simple: we produce the singleton list containing that type and $\mathfrak{f}'$ is immediate.

If we encounter an onion type, we first explore the right side and then explore the left, concatenating the resulting type list sets pointwise. As we explore the left side, we maintain knowledge of the types which appeared in the right side's list and do not select proof subtrees which include them; this prevents generation of duplicates and is also key to termination: every non-onion expansion makes some form of progress. As with non-onion cases, construction of $\mathfrak{f}'$ is immediate.

For the case onions recurse into themselves without an intervening label, special care is needed. For instance, consider the constraint $\alpha_1 \,\&\, \alpha_2 <: \alpha_2$. In order to prevent divergence in such a case, we track each type variable and *the number of times* we have expanded it to an onion lower bound in this proof tree branch. After this number of passes reaches a certain threshold, we replace all expansions of onion lower bounds on that variable with [], pruning off that branch. We assert that the number of times a given onion is expanded can be conservatively be set to the number of distinct (onion and non-onion) lower-bounding types in C.

We first observe that this choice ensures decidability and then defend that it is sufficient for correctness. Because of this decision, there is now a maximum size to the proof tree; each variable with an onion lower bound is only expanded a fixed number of times and there are finitely many such variables. Because non-onion cases are always leaves and because each variable has finitely many lower bounds, the number of proof trees is finite and can be exhaustively checked.

We next assert that it is sufficient to build any proof tree described by the recursive onions to be large enough to hold any legal permutation of the lower-bounding types in C. This is because additional onion structure exploration will never yield more legal variable positions; there are only finitely many types that can be placed in any position. □

Lemma B.8. The function which given some C, α, and π returns the set of all $\mathfrak{f}$ and $\mathfrak{f}'$ in C such that $C \vdash \alpha \ \overset{\pi}{\triangleright} \mathfrak{f}; \mathfrak{f}'$ is a computable function.

Proof. By Lemma B.7, there is a computable function which produces the set of deduplicated lists such that the original lists were the result of projection. Because deduplication never removes the rightmost element of a list (trivial by inspection of Definition B.6), the set of rightmost types from the lists is equal to the set of rightmost types from the lists produced by projection. □

B.1.2 Compatibility

We must now demonstrate decidability and computable algorithms for the compatibility relations. This is quite subtle in comparison to projection, in particular due to the need to align different compatibility proofs under the same fibration. We begin by showing that compatibility is decidable if the fibration is known:

Lemma B.9. The relation $\alpha \preceq_C \phi \backslash \Gamma; \mathfrak{f}; \overleftarrow{\phi'}$ is decidable.

Proof. We first observe that there are finitely many patterns that can be visited by a compatibility relation before reaching a contractive

boundary. The occurrence check $\overbrace{\phi'}$ ensures that each substitution pattern and each recursive pattern are visited only once. There are finitely many substitution patterns because they are taken from the lower bounds of the (finite) constraint set. There are finitely many recursive patterns because they are either (1) part of a pattern already in the constraint set or (2) finite substitutions on those patterns.

Because there are finitely many patterns that can be visited, there is a fixed, finite upper bound on the size of $\overbrace{\phi'}$ in any given instance of compatibility. For this reason, it is sound to induct first on the structure of $\mathfrak{f}$, then on the number of visitable patterns *not* contained in $\overbrace{\phi'}$, and then on the size of ϕ. We proceed by case analysis on ϕ.

Primitive patterns: When the pattern is one of `int`, `fun`, `pat`, or `scape`, compatibility relies on projection which is decidable by Lemma B.7.

Structural patterns: When the pattern is a label pattern, conjunctive pattern, or variable pattern, compatibility relies on at most three types of operations: (1) trivially decidable algebraic operations (e.g. finite set containment), (2) projection (decidable from above), and (3) other compatibility relations. In the label case, compatibility is over a smaller $\mathfrak{f}$; in the conjunctive pattern case, the $\mathfrak{f}$ and $\overbrace{\phi'}$ are the same and the patterns ϕ_1 and ϕ_2 are smaller. Thus, these cases are decidable by induction.

Recursive patterns: For recursive and substitution patterns, there are two cases: either the pattern exists in the set of visited patterns $\overbrace{\phi'}$ or it does not. If it does, compatibility is trivially decidable. Otherwise, compatibility relies on (1) projection (Lemma B.7), (2) trivially decidable operations such as pattern substitution, or (3) compatibility. For both types of pattern, the compatibility is over a set $\overbrace{\phi'}$ that contains a visited pattern it did not contain before; thus, by induction, this operation is decidable. □

Next, we must show that appropriate fibrations can be discovered as necessary. To proceed, we rely on fibrations as a means by which we can describe finite constraints on the regular trees represented by the constrained type variables in PatBang's type system. While the ❋ fibration is not assigned any special meaning in the theory, we use it here to represent a sort of wild card, admitting any fibration in its place. We show the validity of this strategy in the following lemma:

Lemma B.10. In projection and compatibility: if the relation holds with a fibration containing a ❋, then that relation will also hold for a fibration containing a different sub-fibration at that position.

Proof. By inspection of the definition of each relation. No clause of either relation requires a fibration to contain a ❋; therefore, the only case in which a fibration can contain a ❋ and still allow the relation to hold is when that part of the fibration is unconstrained. □

With the above in mind, we can also define a notion of *fibration subsumption*. A fibration $\mathfrak{f}$ subsumes another fibration $\mathfrak{f}'$ if every relation which holds with $\mathfrak{f}'$ also holds with $\mathfrak{f}$ (up to equivalence with Lemma B.10). We define fibration subsumption as follows:

Definition B.11 (Fibration Subsumption)**.**

$$\begin{array}{lll} \mathfrak{f} & \leq & ❋ \\ \langle \tau\,,\, \overrightarrow[n]{\mathfrak{f}} \rangle & \leq & \langle \tau\,,\, \overrightarrow[n]{\mathfrak{f}'} \rangle \quad \text{when } \forall i \in \{1..n\}.\, \mathfrak{f}_i \leq \mathfrak{f}'_i \end{array}$$

Lemma B.12. Fibration subsumption is decidable.

Proof. Trivial by induction on the size of the fibrations. □

We now describe properties of a computable function which models compatibility. This function generates the possible bindings and fibration and takes as input the other positions of the relation. It also takes as an input a *filtering* fibration. This argument ensures that all generated fibrations are restricted to a specific context; all fibrations produced in the output are subsumed by the filtering fibration.

Lemma B.13. A computable function exists from C, α, ϕ, $\overbrace{\phi'}$, and a filtering fibration $\mathfrak{f}'$ to sets $\overbrace{\langle \dot\Gamma, \mathfrak{f} \rangle}$ of possible bindings and the fibrations which lead to them. Each $\mathfrak{f}$ in the output is subsumed by $\mathfrak{f}'$. For all $\dot\Gamma$ and $\alpha \preceq_C \phi\backslash\dot\Gamma; \mathfrak{f}''; \overbrace{\phi'}$ such that $\mathfrak{f}'' \leq \mathfrak{f}'$, there exists some pair $\langle \dot\Gamma, \mathfrak{f} \rangle$ in the output such that $\alpha \preceq_C \phi\backslash\dot\Gamma; \mathfrak{f}; \overbrace{\phi'}$. Additionally, if $\dot\Gamma = ☹$, then such a pair exists where $\mathfrak{f}'' \leq \mathfrak{f}$.

Proof. Constructability: We first observe that, given the other four places of the compatibility relation, each set of bindings and some fibration for that set can be constructed. This process proceeds as in projection via the aggregation non-deterministic disjunctive computation at each choice of a type lower bound and at each single projection. We also add an additional occurrence check for patterns which is *not* cleared when passing through a label. This occurrence check is between *pairs* of the form $\langle \alpha\,, \phi \rangle$, tracking the argument type as well as the pattern. For substitution patterns, it tracks the substitution pattern itself, not the projection from the type variable. This occurrence check permits us to prevent divergence in certain recursive match scenarios.

Simple patterns (`int`, `fun`, etc.) rely on the computability of fibrations for projection (Lemma B.7). Label patterns are somewhat more complex; when the projection of the label type is computed, the projection function must mark the relevant unconstrained position of the resulting fibration so that the fibration obtained from computing compatibility on the label's contents can be placed in that position. Conjunctive patterns proceed by first computing compatibility on both subpatterns and then, for each pair of fibrations $\mathfrak{f}_1$ and $\mathfrak{f}_2$ obtained in this fashion, only admitting $\mathfrak{f}_1$ if $\mathfrak{f}_1 \leq \mathfrak{f}_2$ (and vice versa). Variable patterns are trivially computable.

In the case of recursive patterns, we rely on our second occurrence check. If the same pattern is seen twice on the same definition branch at the same type variable, we know that further expansions of that same structure will always succeed. We also know that such expansions will yield no more bindings that those obtained by the first expansion. Finally, we observe that, due to the non-deterministic disjunctive nature of this algorithm, any future expansions from α which could generate a novel set of bindings can also generate such bindings immediately after the first visit to the recursive pattern. For instance, consider the constraints $\{\texttt{`A}\,\alpha <: \alpha\}$ and the pattern `rec p: 'A p & int`. While it is true that this pattern can find a ☹ result by exploring into the label `'A`, that label can be discovered from the same variable by expanding in that direction immediately. Our computable function therefore stops with the binding of $\emptyset$ the second time it witnesses a given recursive pattern, relying on the matches between that point in the definition the point where the pattern was originally visited to gather the appropriate bindings. In order to prevent this occurrence check from giving false positives in the case of coinductive types, an implementation can verify that at least one branch of the exploration reached a leaf case which did not rely on the occurrence check.

Filtering: To ensure that all $\mathfrak{f}$ in the output is subsumed by $\mathfrak{f}'$, we allow $\mathfrak{f}'$ to dictate union decisions as long as it is not ❋. As a result, the only fibrations which are explored are those which agree with $\mathfrak{f}'$ at each step. The only additional change to this strategy is with respect to recursive pattern forms. The second occurrence check does not begin to accumulate patterns until the filtering fibration is $\mathfrak{f}'$. The decidability of the recursive case is unchanged by this

approach; the filtering fibration is finite, so we will eventually resume the occurrence check strategy described above.

Representation of Failure: Finally, we must show that, for any failure of compatibility, the set of fibrations $\underbrace{\mathfrak{f}}$ produced by this algorithm which relate to failure are representative in that any other fibration $\mathfrak{f}''$ that relates to failure is subsumed by a fibration in that set. The clauses in Figure 4.7 allow the algorithm to form a frontier of failure results: no clause constrains $\mathfrak{f}$ any more than is necessary for the proof to hold. The algorithm uses $\divideontimes$ in each unconstrained position in order to ensure that its results are as general as possible. If some $\mathfrak{f}''$ has a non-$\divideontimes$ where a produced $\mathfrak{f}$ has a $\divideontimes$, Lemma B.10 gives us that $\mathfrak{f}$ is strictly more general and subsumes $\mathfrak{f}''$. If some $\mathfrak{f}''$ has a $\divideontimes$ where a produced $\mathfrak{f}$ has a non-$\divideontimes$, then either it does not allow compatibility to relate or it is not subsumed by the filtering fibration $\mathfrak{f}'$. □

Lemma B.14. A computable function exists from C, α, $\overset{n}{\overrightarrow{\beta}}$ <- ϕ, and a filtering fibration $\mathfrak{f}'$ to a pair of sets $\underbrace{\langle \overrightarrow{\tau}, \mathfrak{f} \rangle}$ and $\underbrace{\mathfrak{f}}$ indicating the success and failure, respectively, of the compatibility relation. For all $\overrightarrow{\tau}$ such that $\alpha \preceq_C \overset{n}{\overrightarrow{\beta}}$ <- $\phi \setminus \overrightarrow{\tau}; \mathfrak{f}'$, there exists some $\langle \overrightarrow{\tau}, \mathfrak{f} \rangle$ in the first set. For all $\mathfrak{f}'$ such that $\alpha \preceq_C \overset{n}{\overrightarrow{\beta}}$ <- $\phi \setminus ☹; \mathfrak{f}'$, there exists some $\mathfrak{f}$ in the second set such that $\mathfrak{f}' \leq \mathfrak{f}$.

Proof. Immediate from Lemma B.13. The only remaining computation is to order the results of the binding set according to $\overset{n}{\overrightarrow{\beta}}$, which is trivially computable. □

Lemma B.15. Relation $\alpha \preceq_C \overrightarrow{\tau} \setminus \alpha; \dot{C}'; \mathfrak{f}$ is decidable. A computable function exists from C, α, and $\overrightarrow{\tau}$ to a set of valid $\dot{C}'$ and $\mathfrak{f}$.

Proof. First we show that the relation is decidable by induction on the length of $\overrightarrow{\tau}$. Inspection of Definition 4.9 reveals that the truth of this term is either immediate (when the length of $\overrightarrow{\tau}$ is zero), reliant upon decidable projection failure (shown to be decidable in Lemma B.5, reliant upon compatibility (shown to be decidable in Lemma B.9), or defined in terms of a smaller list.

Next, we show that a function exists to compute this relation. By Lemma B.14, we can compute the set of $\overrightarrow{\tau'}$ for which compatibility holds (and one $\mathfrak{f}$ for each such list) and the set of $\mathfrak{f}'$ for which compatibility relates to ☹ such that any other $\mathfrak{f}''$ relating to ☹ is subsumed by some $\mathfrak{f}'$. We say that this latter set of fibrations is *representative* of failure.

We proceed with the intuition of *partitioning* the (potentially infinitely many) different sequences of union decisions made for the argument type, using fibrations to represent classes of such decision sequences. (The fibration $\langle l\,\alpha', [\divideontimes] \rangle$, for instance, represents all decision sequences which begin with choosing the lower bound $l\,\alpha'$.) In the positive case, the $\overrightarrow{\tau'}$ represents all fibrations (and thus decision sequences) which can relate to this list of types. Although we cannot determine a set of fibrations which is representative for each $\overrightarrow{\tau'}$ as we can for ☹ – the fibration grammar is insufficiently expressive – no further union decisions are made once a binding set is discovered and so computation is finished.

In the case of failure, on the other hand, we have a set of $\mathfrak{f}'$ which is representative of failure. We proceed by using each of these fibrations in non-deterministic disjunctive computation to filter compatibility with the next scape in the list. This ensures that we only find compatible cases for union decision sequences which failed for the previous scape. Because the set of $\mathfrak{f}'$ is representative of failure, we know that no $\mathfrak{f}''$ exists which fails in the previous scape but permits compatibility to hold for the second scape unless some $\mathfrak{f}'$ also permits compatibility to hold (by Lemma B.10). This is to say that we know that we have not "missed" any union decision sequences for the argument type.

Finally, we observe that the list of scapes is necessarily finite. Because compatibility is computable (Lemma B.14) and we use it finitely many times (at most once for each of finitely many fibrations for each scape), we have that application compatibility is computable. □

B.1.3 Contours

Lemma B.16. Relations $\mathcal{C}_1 \leq \mathcal{C}_2$ and $\mathcal{C}_1 \between \mathcal{C}_2$ are decidable.

Proof. The grammar and meaning of contours is a subset of regular expressions and subsumption and overlap on regular expressions is well-known to be computable. □

Lemma B.17. The contour extraction ($(\!|\cdot|\!)$), instantiation ($\textsc{Inst}(\cdot,\cdot)$), and replacement ($\textsc{Repl}(\cdot,\cdot)$) are computable.

Proof. These functions are trivially computable by induction on their respective first arguments. Each step not defined as a natural homomorphism is trivially computable. For instantiation, the scape case calculates a finite union and the variable case performs a presence test on a finite set. For replacement, the variable case performs subsumption testing (decidable by Lemma B.16). The leaf cases of extraction are constant. □

Lemma B.18. The functions used in contour creation – $\textsc{CNew}(\cdot,\cdot)$, $\textsc{Collapse}(\cdot)$, and $\textsc{Widen}(\cdot,\cdot)$ – are decidable.

Proof. $\textsc{CNew}(\cdot,\cdot)$ is trivially computable. $\textsc{Collapse}(\mathcal{C})$ is trivially computable if each $\textsc{Collapse}(\mathcal{S})$ is computable for each $\mathcal{S} \in \mathcal{C}$. $\textsc{Collapse}(\mathcal{S})$ is computable given some $\mathcal{S}$ by induction on the number of cycles in $\mathcal{S}$. $\textsc{Widen}(\mathcal{C}, C)$ is computable given $\mathcal{C}$ and C because contour extraction and overlap are known to be decidable from Lemmas B.16 and B.17. □

B.1.4 Typechecking

Lemma B.19. Given C and C', $C \Longrightarrow^1 C'$ is decidable.

Proof. We proceed by showing that there is a computable function from C to $\overset{m}{\underbrace{C''}}$ such that $C \Longrightarrow^1 C_i''$ for all $i \in \{1..m\}$ and that there exists no other C''' such that $C \Longrightarrow^1 C'''$; that is, the set of possible next steps is computable and finite. Inspection of the closure rules reveals that the first premise of each rule tests for the presence of a constraint in C of a certain form; because C is finite, we can enumerate all such constraints. For each such constraint, the remainder of the premises in each rule are computable either trivially or by one of the above lemmas. For application, we use Lemma B.7 to produce the set of possible $\overrightarrow{\tau}$ scapes that may apply; this Lemma produces a list without duplicates, and it is clear that the presence of duplicates is irrelevant to the result of application compatibility, Definition 4.9. Because each rule has finitely many preconditions, this process is computable. Thus we can compute the set of constraints $\overset{m}{\underbrace{C''}}$; because each rule only adds finitely many constraints, each C_i'' is finite. This relation is then determined simply by $C' \in \overset{m}{\underbrace{C''}}$. □

Lemma B.20. It is decidable whether a given C is inconsistent.

Proof. By inspection of Definition 4.13, each condition of inconsistency first checks for a constraint of a given form; this check can be achieved by enumeration because C is finite. For each such constraint, it then performs a series of checks known to be decidable either trivially or from the above lemmas. □

We now prove Theorem 4.16.

Proof. The proof proceeds by showing that each closure rule can only introduce constraints drawn from an initial fixed finite set of possible constraints.

Given a closed e, $[\![e]\!]_{\text{E}} = \langle \alpha, C \rangle$ is computable from Lemma B.1.

New contours are only created by $\textsc{cnew}(\cdot, \cdot)$. This function guarantees that (1) all contour strands contain a given identifier only once and that (2) no contour strands contain an empty set as a part. As a result, there is a fixed, finite number of unique contour strands that could ever occur in any closure sequence of C and thus there is a fixed bound on the number of distinct contours possible in any closure. The only new value variables x introduced during closure are those produced by $\textsc{fvar}(\alpha)$ and there exists one such variable for each x appearing in the initial e. These facts together give us that, given a C, we can define a fixed, finite set of type variables which is a superset of any variable that could arise during any closure sequence.

Since the possible contours in any closure sequence are fixed in advance, there exists a fixed finite set can be defined which bounds all of the τ that could occur in any constraint over any closure sequence: by inspection of the closure rules, no new non-scape types τ are created in new constraints which are not just substitutions on types already occurring in C. Finally, function types contain a constraint set and all substitutions on such a set are fixed and finite since the number of possible substitutions is bounded by the above and so the number of constraints that could be added to closure via application have a fixed bound.

Since given initial e the possible type variables and types have a fixed bounded size, there is also a fixed, finite set bounding the constraints which may appear in any closure; let C^* be that set. Therefore, any C' such that $C \Longrightarrow^1 C'$ must be in the power set of C^*; because C^* is finite, its power set is also finite. Let this power set be denoted $\underset{m}{\underbrace{2^{C^*}}}$ and let n be the size of this set.

The set $\underset{m}{\underbrace{C'}}$ such that for all $i \in \{1..m\}$ we have $C \Longrightarrow^* C'_i$ is also decidable. We know this set to be a subset of 2^{C^*} and that any acyclic chain of closures $C \Longrightarrow^1 C'_{j_1} \Longrightarrow^1 \ldots \Longrightarrow^1 C'_i$ (for $i \in \{1..m\}$ and j_k all in $\{1..n\}$) has at most n steps (since there are only n such sets in 2^{C^*}). Because the relation holds over the first and last sets and is not based on the intervening sets, any cyclic chain of closures need not be considered as it is equivalent to some acyclic chain. Because these acyclic chains are all composed of elements of 2^{C^*}, the number of such acyclic chains is at most the sum of the permutations of the subsets of 2^{C^*}, which is finite. By enumeration of these permutations, we can exhaustively determine which elements of 2^{C^*} are in $\underset{m}{\underbrace{C'}}$ and which are not; therefore, $\underset{m}{\underbrace{C'}}$ is computable.

Finally, for each such C'_i, Lemma B.20 gives us that it is decidable whether or not that set is inconsistent. Since it is decidable whether any of these sets is inconsistent, it is decidable whether or not e typechecks. □

B.2 Efficient Typechecking

The proof of Theorem 4.16 in the previous section used a counting argument for sake of simplicity. While this shows that the PatBang type system is decidable, it does not show that it is computationally feasible. Because our objective is to produce a usable, highly flexible scripting language, we will now informally discuss how the complexity of typechecking is tractable.

Constraint Closure Confluence The first reduction in complexity starts with the way typechecking is phrased. A program e is typesafe only if, given its initial constraint C, *every* C' for which $C \Longrightarrow^* C'$ is consistent; computing every such constraint set would require considerable effort. This could be avoided if constraint closure were confluent; then, the closure work would be reduced to a computable, deterministic function. The constraint closure relation is not trivially confluent; the contours which it produces will vary based upon the order in which the constraint closure rules are used. This is, in particular, due to the unioning of contours in the widening step of contour creation. Up to contour meaning ($[\![\cdot]\!]$), however, these contours are equivalent. It is therefore sufficient to close over the initial constraint set to *any* of its fixed points and simply determine if that constraint set is inconsistent.

Projection Lemma B.7 shows that the set of orderings of scapes which will project from a type variable is computable, but it uses an excessively complex algorithm. Algorithms for projecting the single highest priority element from a type variable are quite simple, but – due to onion types which recurse directly and not under a label – the task of projecting multiple types in priority order is difficult in some corner cases. This is because immediately recursive onions are a form of *non-contractive* type because `&` shares enough similarity with intersection type, and intersection is not a contractive type operator [15]. It is well known that working with non-contractive recursive types is difficult [15] and, for that reason, it is standard to work only with contractive types. While eliminating non-contractive types from PatBang would be as simple as a syntactic check prohibiting immediately recursive onion types, we believe that non-contractive types will make it possible to statically type certain higher-order programming patterns.

There are only two places that our priority-based projection is used. The first is in the definition of single projection which, as mentioned above, can be implemented with simple, efficient algorithms. The second is in the Application closure rule, where it is used to determine the order in which scapes are applied. Observe that the use of priority-based projection here is inefficient; the projection tries to model each and every concrete type tree and individually determine the priority list which results from it, and there are many more such type trees than there are priority lists in every case. Furthermore, the set of priority-ordered lists contain significant redundancies, since the ordering only matters in cases where the types matched by scape patterns overlap.

In order to correct both sources of inefficiency, we plan to specialize an operation to address the behavior of the application rule over an onion of scapes. We rely on the fact that all types expressible in our constraint-based system can be expressed in a regular tree algebra and we can use operations over regular trees to calculate how much of the argument type is matched at any given point during the process. By defining the operation to handle all lower bounds at a given call site simultaneously, we effectively group the application cases into equivalence classes and process each equivalence class only once. This approach also allows us to consider each recursive onion type only once; because it considers all lower bounds of the recursive type variable simultaneously, we know that no further information can be gained by exploring it again.

Contours The only remaining concern regarding the complexity of the PatBang type system is the generation of type contours. As with any polymorphic system, the polyinstantiation of variables gains expressiveness at the cost of complexity and, as with any interesting polymorphic type system (such as those in the ML family of languages), typechecking PatBang is exponentially complex in pathological scenarios. We have attempted to ensure, however, that PatBang typechecking will be polynomial in practice.

The manner in which contours are created, for instance, is intended to prevent exponential complexity. Since contours on type

variables represent (often infinite) sets of calling contexts, it is not impossible to imagine that a closure rule may fire for some intersection of two variables' calling contexts. In that case, constraints would need to be constructed over the intersection of these calling contexts and propagated appropriately. This deep reasoning about type propagation is only helpful in very specific cases of overlapping recursive call cycles where the relevant input types are statically known; such reasoning also generates exponentially many constraints in the size of the program. Because this precision is costly and not very useful, we choose to completely unify the variables involved in any call cycle; this is accomplished at the end of the contour creation process by the $\textsc{Widen}(\cdot,\cdot)$ function, which ensures that any new contour will be unified with a contour it overlaps (even partially). This ensures that each contour in constraint closure represents a disjoint calling context, significantly reducing the complexity of closure.

In general, we expect that a polynomial-in-practice implementation of the PatBang typechecker is feasible, but it will take more effort to achieve than other, more traditional typecheckers due to subtle issues like those mentioned above.

C. Formalism of Other Patterns

In this section, we formalize the patterns which were elided from Sections 3 and 4 for simplicity: disjunction patterns, `none` patterns, and the pattern-of operator. Conveniently, this formal treatment only requires us to extend existing definitions. We begin by extending the existing syntax of patterns and pattern types:

Definition C.1 (Extended Pattern Grammars)**.**

$$\begin{array}{lcll} \varphi & ::= & \ldots \mid \varphi \mid \varphi \mid \$\, x \mid \texttt{none} & \textit{pattern bodies} \\ \phi & ::= & \ldots \mid \phi \mid \phi \mid \$\, \alpha \mid \texttt{none} & \textit{pattern body types} \end{array}$$

We then define the operational semantics of value compatibility as the least relation satisfying all clauses in Figure 3.6 as well as those in the following definition:

Definition C.2 (Extended Value Compatibility)**.**

$$\begin{array}{lll} x \preceq_E \varphi_1 \mid \varphi_2 \setminus B_2 ; \overline{\varphi'} & \text{if} & x \preceq_E \varphi_2 \setminus B_2 ; \overline{\varphi'} \\ x \preceq_E \varphi_1 \mid \varphi_2 \setminus B_1 ; \varphi' & \text{if} & x \preceq_E \varphi_2 \setminus \textcircled{\scriptsize :}; \varphi', x \preceq_E \varphi_1 \setminus B_1 ; \overline{\varphi'} \\ x_1 \preceq_E \$\, x_2 \setminus \emptyset; \overline{\varphi'} & \text{if} & E \downarrow^*_{\texttt{scape}}(x_2) = x'_\square \texttt{><} \overrightarrow{x''_\square}, \\ & & \forall i \in \{1..n\}.\ E \downarrow_{\texttt{pat}}(x'_i) = \overrightarrow{y}_i \texttt{<-} \varphi_i\,, \\ & & \varphi'' = \texttt{none} \,\&\, \varphi_1 \mid \ldots \mid \varphi_n\,, \\ & & x_1 \preceq_E \varphi'' \setminus B ; \overline{\varphi'} \end{array}$$

Here, a disjunction pattern attempts to match its right subpattern first and matches its left subpattern if that fails. The pattern-of operator creates a disjunction pattern from all of the scapes in the provided variable and matches against it. The `none` pattern has no clause; no values can ever match it.

The above augmentation is sufficient to define these three additional patterns; the remaining relations (projection, application compatibility, closure, etc.) are unaffected. In keeping with the type system presented in Section 4, the type compatibility clauses are quite similar but include failure cases. We define type compatibility to be the least relation satisfying all clauses from Figure 4.6, Figure 4.7, and Figure C.3.

As in the basic type system from Section 4, each clause either (1) directly mirrors an evaluation rule or (2) recognizes or propagates failure cases. No further modification to the type system is necessary.

The manner in which disjunction patterns bind variables is somewhat unusual and merits consideration. For a given argument, the left side is only considered when the right side fails to match. Consider the pattern `(x) <- 'None _ | 'Some x`. In general, this pattern does not always bind the pattern variable `x`; it is therefore often a type error to use that binding (e.g. pairing this pattern with `(y) -> y`), although the pattern can be used as a recognizer with

$$\begin{array}{lll} \alpha \preceq_C \phi_1 \mid \phi_2 \setminus \Gamma_2; \mathfrak{f}; \overline{\phi'} & \text{if} & \alpha \preceq_C \phi_2 \setminus \Gamma_2; \mathfrak{f}; \overline{\phi'} \\ \alpha \preceq_C \phi_1 \mid \phi_2 \setminus \dot{\Gamma}; \mathfrak{f}; \overline{\phi'} & \text{if} & \alpha \preceq_C \phi_2 \setminus \textcircled{\scriptsize :}; \mathfrak{f}; \overline{\phi'}\ \alpha \preceq_C \phi_1 \setminus \dot{\Gamma}; \mathfrak{f}; \overline{\phi'} \\ \alpha_1 \preceq_C \$\, \alpha_2 \setminus \emptyset; \mathfrak{f}; \overline{\phi'} & \text{if} & C \vdash \alpha_2 \xrightarrow{\texttt{scape}} \overrightarrow{\alpha'_\square \texttt{><} \alpha''_\square}, \\ & & \forall i \in \{1..n\}.\, C \vdash \alpha'_i \xrightarrow{\texttt{pat}} \overrightarrow{\beta}_i \texttt{<-} \phi_i\,, \\ & & \phi'' = \texttt{none} \,\&\, \phi_1 \mid \ldots \mid \phi_n\,, \\ & & \alpha_1 \preceq_C \phi'' \setminus \Gamma; \mathfrak{f}; \overline{\phi'} \\ \alpha \preceq_C \phi_1 \mid \phi_2 \setminus \text{\Lightning}; \mathfrak{f}; \overline{\phi'} & \text{if} & \alpha \preceq_C \phi_2 \setminus \text{\Lightning}; \mathfrak{f}; \overline{\phi'} \\ \alpha_1 \preceq_C \$\, \alpha_2 \setminus \textcircled{\scriptsize :}; \mathfrak{f}; \overline{\phi'} & \text{if} & C \vdash \alpha_2 \xrightarrow{\texttt{scape}} [] \\ \alpha_1 \preceq_C \$\, \alpha_2 \setminus \textcircled{\scriptsize :}; \mathfrak{f}; \overline{\phi'} & \text{if} & C \vdash \alpha_2 \xrightarrow{\texttt{scape}} \overrightarrow{\alpha'_\square \texttt{><} \alpha''_\square}, \\ & & 1 \le i \le n, C \vdash \alpha'_i \xrightarrow{\texttt{pat}} [] \\ \alpha_1 \preceq_C \$\, \alpha_2 \setminus \dot{\Gamma}; \mathfrak{f}; \overline{\phi'} & \text{if} & C \vdash \alpha_2 \xrightarrow{\texttt{scape}} \overrightarrow{\alpha'_\square \texttt{><} \alpha''_\square}, \\ & & \forall i \in \{1..n\}.\, C \vdash \alpha'_i \xrightarrow{\texttt{pat}} \overrightarrow{\beta}_i \texttt{<-} \phi_i\,, \\ & & \phi'' = \texttt{none} \,\&\, \phi_1 \mid \ldots \mid \phi_n\,, \\ & & \alpha_1 \preceq_C \phi'' \setminus \dot{\Gamma}; \mathfrak{f}; \overline{\phi'}, \dot{\Gamma} \in \{\textcircled{\scriptsize :}, \text{\Lightning}\} \\ \alpha_1 \preceq_C \texttt{none} \setminus \textcircled{\scriptsize :}; \mathfrak{f}; \overline{\phi'} & \text{always} & \end{array}$$

Figure C.3. PatBang Extended Compatibility

zero-argument functions. But in the event that the provided argument is *statically* known to have only `'Some`-labeled types as lower bounds, use of this binding will be successful; the compatibility check will always produce a sufficiently large bindings set and application compatibility will hold. This allows a pattern such as `(x) <- rec p: 'Nil _ | ('Hd x & 'Tl p)` to match the last element in a list successfully as long as it is always passed (statically-known) non-empty lists.