Types for Flexible Objects

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

Abstract

Scripting languages are popular in part due to their extremely flexible objects. These languages support numerous object features, including dynamic extension, mixins, traits, and first-class messages. While some work has succeeded in typing these features individually, the solutions have limitations in some cases and no project has combined the results.

In this paper we define TinyBang, a small typed language containing only functions, labeled data, a data combinator, and pattern matching. We show how it can directly express all of the aforementioned flexible object features and still have sound typing. We use a subtype constraint type inference system with several novel extensions to ensure full type inference; our algorithm refines parametric polymorphism for both flexibility and efficiency. We also use TinyBang to solve an open problem in OO literature: objects can be extended after being messaged without loss of width or depth subtyping and without dedicated metatheory. A core subset of TinyBang is proven sound and a preliminary implementation has been constructed.

1. Introduction

Modern scripting languages such as Python and JavaScript have become popular in part due to the flexibility of their object semantics. In addition to supporting traditional OO operations such as inheritance and polymorphic dispatch, scripting programmers can add or remove members from existing objects, arbitrarily concatenate objects, represent messages as first-class data, and perform transformations on objects at any point during their lifecycle. This flexibility allows programs to be more concise and promotes a more separated design; in Ruby, for instance, refinements and other forms of "monkey patching" can be used to extend class definitions, allowing a user to add to the behavior of library classes without modifying the relevant library.

While a significant body of work has focused on typing these flexible object operations, the solutions proposed place restrictions on how the resulting objects can be used. The Bono-Fisher object calculus [8] defines a sound means by which objects may be functionally extended: functions within the calculus can define mixins and other higher-order object transformations. But objects in this calculus must be "sealed" before they are messaged and sealed objects cannot be extended. Later works define models which allow objects to be extended at any point during their lifecycle, but these models either do not admit depth subtyping [26] or limit the precision of types which result from object extensions [6]. As we will show, these restrictions are too prohibitive to properly model the philosophy of script programming.

In this paper, we present TinyBang, a typed core language which contains very few features: *onioning* (an asymmetric data combinator for record-like data structures), *scapes* (partial functions which pattern-match their arguments) and labeled data. TinyBang has no explicit syntax for classes, objects, inheritance, object extension, overloading, or switch/case. We show, however, that objects and flexible operations on them, including those listed above, can be tersely expressed with only scapes and onions. These operations preserve width and depth subtyping and TinyBang's type system infers correct types for them without any programmer annotation.

We do not expect programmers to write in TinyBang directly but instead in BigBang, a language including syntax for objects, classes, and so on that is not presented here; BigBang will desugar to TinyBang. While the fundamental ideas we present here could be used to give BigBang types and runtime semantics for objects directly, we focus on object encodings because they yield a few benefits. First: TinyBang's object calculus operations can be defined as in-language functions, avoiding the need for object-specific metatheory. Second: our encoding is simple enough to be *translucent* in that it will be possible, though atypical, to "work under the hood" to manipulate objects on a lower level when necessary. Third: this translucency ensures an elegant encoding and attests to the expressiveness of the underlying language constructs.

In order to preserve translucency in the face of flexible operations like method override and first-class messages, we design our object encoding with a *variant-based* philosophy. In contrast with record-based encodings, in which objects are records and method invocation occurs by calling the appropriate member, variant-based encodings encode objects as message-processing functions and represent method invocation as a simple function call. First-class messages are then simply the argument to that function call, and method

names are the (optional) tags placed on such arguments. The variant-based philosophy not new; Actor models [2] are untyped variant-based objects, but the variant-based encoding is more challenging to type than record-based encodings. This paper shows how, using onions and scapes, the variant-based encoding is more translucent than a traditional record-based one.

Scapes: generalizing first-class cases In our variant-based encoding, method dispatch is essentially matching on a message value. But traditional match/case expressions are monolithic blocks: they lack an explicit composition operator. We must have extensible match/case expressions to allow our encoded objects to be extensible. To accomplish this, we use a (typed) notion of first-class cases, which has been explored previously in [7]. There, a construct $\oplus$ is defined for extending a case with a single additional clause. Our scapes generalize first-class cases by supporting composition of arbitrary case expressions, not just the addition of one clause. Since the case extension of [7] is only adding case clauses to the end of an existing case, it cannot model inheritance or object concatenation: the former requires the subclass's methods to take priority over those of the superclass and the latter requires arbitrary composition of case expressions. We believe that scapes offer the first high-level algebraic case composition operator which can directly support object inheritance for a variant encoding of objects.

Scapes are in spirit similar to typed multimethods [10, 21, 22] but multimethods are a *nominal* dispatch on class names whereas scapes are a *structural* dispatch following ML/Haskell deep patterns; many of the scripting patterns we aim to model rely on structural dispatch.

Less related is the Pattern Calculus [15], which is a more fundamental algebra in that patterns themselves are first-class entities separate from the code to be executed in case of a match. However, variable bindings in this calculus are extremely complex, so we choose the case clause as our level of abstraction.

Dependent pattern types Another problem arises when using traditional match/case expressions in a typed, variant-based object encoding: all case branches must have the same output type. This essentially requires every method of an object to return the same type, which is clearly unacceptable. TinyBang solves this fundamental problem by including a weak form of dependent type, the *dependent pattern type*, to allow for heterogeneity in case branches. A dependent pattern type allows the type of a value to condition on the variant tag of the argument. We call these types *weakly* dependent because they depend only on this variant tag and not more detailed information from the value; the advantage of this weakness is that type inference is still decidable.

Our approach is a generalization of the conditional constraints originating in [4] and elaborated in [24]. Conditional constraints partly capture this dependency but only locally and so lose the dependency information in the presence of side effects. Our dependent pattern types fully capture this dependency. Dependent pattern types are related to types that can be given to multimethods [10], but again are structural instead of being nominal.

Onions: type-indexed records supporting asymmetric concatenation and object extension A type-indexed record is one for which contents are projected using types rather than labels [27]. For example, consider the type-indexed record `{foo = 45; bar = 22; (); 13}` which implicitly tags the untagged elements `()` and `13` with their types. Projecting `int` from this record would yield `13` (as the other integers are labeled). Similarly, one can project unlabeled functions from a type-indexed record. Our *onions* are a form of type-indexed record; this added flexibility is handy in many situations, as will be seen below.

Additionally, since (untagged) functions can be placed in our onions, we can re-use the onion record structure to hold our scape clauses and avoid needing two different extension operations as found in [7]. As we alluded above, *asymmetric* concatenation is the key to composing scapes and thus properly defining inheritance and overriding; onions thus support asymmetric concatenation as the default. Asymmetric record concatenation was initially proposed by Wand for modeling inheritance [29], but difficulties in obtaining principal types in unification-based type inference [30] caused a switch in research focus to symmetric notions of record concatenation. Unfortunately, these notions are not amenable to modeling method overriding. In standard record-based encodings of inheritance [9], there is no first-class record extension operation; inheritance requires the superclass to be known statically. This is implicitly a consequence of the difficulty of typing asymmetric record extension. Dynamically-typed OO languages have explored first-class object extenders to significant benefit; the use of typed asymmetric record concatenation can bring this flexibility to the typed world.

Subtype constraint types TinyBang uses a subtype constraint inference based type system [4, 14, 24]; in particular, it is most closely related to [24]. Compared to [24], our system does not need row types or conditional constraints to typecheck record concatenation, incorporates type-indexed records and first-class cases, and contains a more general notion of conditional type and a more general model of parametric polymorphism. Our approach to parametric polymorphism is based on flow analysis [28, 31] and improves on previous work for expressive but efficient contour sharing. It uses a restricted form of regular expressions to finitely abstract call strings, following program analyses [20].

While our subtype constraint system infers extremely precise and expressive types, these types are also by nature difficult to read and defy modularization. TinyBang's type system is therefore a whole-program analysis. Our broader research agenda includes acquiring the typical benefits of modularity by using other techniques. Lightweight, approximate type declarations, for instance, can be used to provide

$$
\begin{array}{rcll}
e & ::= & x \mid \text{lbl}\, e \mid e \,\texttt{\&}\, e \mid e \,\texttt{\&-}\, \pi \mid e \,\texttt{\&.}\, \pi \mid e \,\texttt{\&!}\, \pi & \textit{expressions} \\
 & & \mid e\ e \mid e\ \text{op}\ e \mid \texttt{def}\ x = e\ \texttt{in}\ e \mid x = e\ \texttt{in}\ e & \\
 & & \mid \mathbb{Z} \mid \textit{(characters)} \mid \texttt{()} \mid \phi\ \texttt{->}\ e & \textit{literals} \\
\text{op} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{==} \mid \texttt{<=} \mid \texttt{>=} & \\
x & ::= & \textit{(alphanumeric identifiers)} & \\
\text{lbl} & ::= & \texttt{`}\textit{(alphanumeric identifiers)} & \\
\phi & ::= & x : \varphi \mid x \mid \varphi & \textit{patterns} \\
\varphi & ::= & \text{tprim} \mid \text{lbl}\, \phi \mid (\varphi \,\texttt{\&}\, \ldots \,\texttt{\&}\, \varphi) \mid \texttt{fun} \mid \texttt{any} & \\
\pi & ::= & \text{tprim} \mid \text{lbl} \mid \texttt{fun} & \textit{projectors} \\
\text{tprim} & ::= & \texttt{int} \mid \texttt{char} &
\end{array}
$$

Figure 2.1. TinyBang Syntax

well-defined type interfaces and to support dynamic loading. While we believe whole-program typechecking can be made efficient and practical, the means to do so are well beyond the scope of this paper.

Outline In the next section, we give an overview of how TinyBang can encode object features. Section 3 contains the operational semantics for MicroBang, a subset of TinyBang trimmed to the key features for readability. In Section 4 the type system for MicroBang^0, a monomorphic restriction of the full MicroBang type system, is presented; we add polymorphism in Section 5 to yield the full MicroBang type system. (Type soundness of MicroBang is established in Appendix A.) Since MicroBang lacks several features of TinyBang, we then define a series of language extensions that close this gap: Section 6 adds fully wide and deep patterns and Section 7 describes how mutable state, onion filtering, and symmetric concatenation are added to MicroBang. We conclude in Section 8.

2. Overview

This section gives an overview of the TinyBang language and of how it supports flexible object operations and other scripting features. Section 2.1 describes the TinyBang language itself and shows how simple objects can be encoded. Section 2.2 describes how self-awareness is encoded and is more flexible in TinyBang than in previous work. Section 2.3 describes flexible operations that we can define over this object encoding, such as mixins and overloading. Section 2.4 compares this variant-based encoding with its record-based analogue.

2.1 Language Features for Flexible Objects

TinyBang's syntax appears in Figure 2.1. The expression grammar contains the key features mentioned above: scapes (written ϕ `->` e), onions (written e `&` e), and labeled data (written `` `Foo `` e for a label constructor `` `Foo ``). The remaining language features are orthogonal to our object encoding but are helpful simplifying the examples below.

In order to typecheck our encoding, we use a subtype constraint-based type system. The encoding we present is not intrinsically tied to such an approach, but subtype constraints provide numerous advantages. Compared to row typing systems, for instance, it is not necessary to explicitly annotate type transition sites such as upcasts [25]. Most importantly, subtype constraint systems are handily modified to accommodate the more unusual language features we describe below. We defer detailed discussion of our type system to Sections 4 through 7; this section's use of types remains informal for presentation clarity.

Scapes as methods We begin by considering the oversimplified case of an object with a single method and no fields or self-awareness. In the variant encoding, such an object is represented by a function which matches on a single case. Note that, in TinyBang, *all* functions are written ϕ `->` e, with ϕ being a *pattern* to match against the function's argument. Combining pattern match with function definition is also possible in ML and Haskell, but we go further: there is no need for any `match` syntax in TinyBang since `match` can be encoded as a pattern and its application. We call such pattern-matching functions *scapes*. For instance, consider the following object and its invocation:

```
def obj = (`double x -> x + x) in obj (`double 4)
```

The syntax `` `double 4 `` is a label constructor similar to an OCaml polymorphic variant; as in OCaml, the expression `` `double 4 `` has type `` `double int ``. The scape `` `double x -> x + x `` is a function which matches on any argument containing a `` `double `` label and binds its contents to the variable `x`. Note that the expression `` `double 4 `` represents a *first-class message*; the object invocation is represented with its arguments as a variant.

Unlike a traditional `match` expression, an individual scape is a function and is only capable of matching one pattern. To express general match expressions with scapes, individual scapes are appended via the *onion* operation `&`. (This conjoiner has many other uses which are discussed below.) Given two scape expressions `e1` and `e2`, the expression `(e1 & e2)` conjoins the patterns to make a scape with the conjoined pattern, and `(e1 & e2) a` will apply the scape which has a pattern matching `a`; if both patterns match `a`, the rightmost scape (`e2`) is given priority. We can thus write a dispatch on an object with two methods simply as:

```
def obj = (`double x -> x + x)
        & (`isZero x -> x == 0) in obj `double 4
```

The above shows that traditional `match` expressions can be encoded using the `&` operator to join a number of scapes: one scape for each case. Our scape conjunction generalizes the first-class cases of [7] to support general appending of arbitrary cases; the aforecited work only supports adding one clause to the end and so does not allow "override" of an existing clause or "mixing" of two arbitrary sets of clauses.

Dependent pattern types The above shows how to encode an object with multiple methods as an onion of scapes. But we must be careful not to type this encoding in the

way that match/case expressions are traditionally typed. The analogous OCaml match/case expression

```
let obj m = (match m with
  | `double x -> x + x
  | `isZero x -> x == 0) in ...
```

will not typecheck; OCaml match/case expressions must return the same type in all case branches. (The recent OCaml 4 GADT extension mitigates this difficulty but requires an explicit type declaration, type annotations, and only works under a closed world assumption). Instead, we give the scape a dependent pattern type (`` `double int → int) & (`isZero int → boolean ``).[1] If the scape is applied in the context where the type of message is known, the appropriate result type is inferred; for instance, invoking this method with `` `isZero 0 `` always produces type `int` and not type `int ∪ boolean`. Because of this dependent typing, `match` expressions encoded in TinyBang may be heterogeneous; that is, each case branch may have a different type in a meaningful way. When we present the formal type system below, we show how these dependent pattern types extend the expressiveness of conditional constraint types in a dimension critical for typing objects.

Onions as records We now show how our data conjoiner, `&`, can act like a record constructor. For example, here is how we encode objects with multi-argument methods:

```
1 def obj = (`sum (`x x & `y y) -> x + y)
2         & (`equal (`x x & `y y) -> x == y)
3 in obj (`sum (`x 3 & `y 2))
```

The `` `sum `` label on line 3 merely wraps another value, in this case an onion of two labels: our representation of a two-label record. This record-like onion is passed to the pattern `` `x x & `y y ``; here, we use `&` to also denote pattern conjunction, which requires that the value must match both subpatterns to match the overall pattern. Observe from this example how there is no hard distinction in TinyBang between records and variants: there is only one class of label and a 1-ary record is the same as a 1-ary variant.

2.2 Self-Awareness and Resealable Objects

Up to this point objects have not been able to invoke their own methods, so the encoding is incomplete. To model self-reference we build on the work of [8], where an object exists in one of two states: as a prototype, which can be extended but not messaged, or as a "proper" object, which can be messaged but not extended. A prototype may be "sealed" to transform it into a proper object, at which point it may never again be extended.

Unlike the aforecited work, our encoding permits sealed objects to be extended and then *resealed*. The flexibility of TinyBang allows the sharp phase distinction between prototypes and proper objects to be relaxed. All object extension below will be performed on sealed objects. Object sealing in TinyBang requires no special metatheory; it is defined directly as a function `seal`:

```
def fixpoint = f -> (g -> x -> g g x)
  (h -> y -> f (h h) y) in
def seal = fixpoint (seal -> obj ->
  obj & (msg -> obj (`self (seal obj) & msg))) in
def obj = (`double x -> x + x)
        & (`quad x & `self self ->
            self (`double x) + self (`double x))
def sObj = seal obj in ...
```

The `seal` function accepts an object as an argument and returns it with a new message handler onioned onto its right. This message handler matches *every* argument and will therefore be used for every message the returned object receives. We call this message handler the *self binding scape* because it adds a `` `self `` component to every message sent to the object using a reference to the object as it stood at the time the seal occurred. The self binding scape also ensures that the value in `` `self `` is a sealed object, allowing methods to message it normally. As a result of sealing the object, a `` `quad 4 `` message sent to `sObj` would produce the same effect as a `` `self sObj & `quad 4 `` message sent to `obj`.

Extending previously sealed objects Previous work [6, 26] allows an object to be extended after it has been messaged. In [26], this power comes at the cost of depth subtyping, which is fundamental to modern object-oriented languages; for instance, it is necessary for the Java statement `Set<? extends Number> s = new HashSet<Integer>();` to typecheck. In [6], depth subtyping is admitted but the type system is nominal rather than structural; one result of this is that extended objects have only the type of the extension and not the type of the original object. We now show how TinyBang improves on previous work without making either of these concessions.

In the self binding scape above, the value of `self` is onioned onto the *left* of the message rather than the right; this choice is purposeful given the right-precedence nature of onioning. Because of this, any value of `` `self `` which exists in a message passed to a sealed object takes priority over the `` `self `` provided by the self binding scape. Consider the following continuation of the previous code:

```
def sixteen = sObj `quad 4 in
def obj2 = sObj & (`double x -> x) in
def sObj2 = seal obj2 in
def four = sObj2 `quad 4 in ...
```

When this code is executed, the variable `sixteen` will hold the value `16`. Even after the `` `quad 4 `` message has been sent to `sObj`, we may extend it; in this case, `obj2` redefines how `` `double `` messages are handled. `sObj2` represents the sealed version of this new object. When the

[1] This section uses simplified types which improve readability by avoiding constraint sets. The actual types used in TinyBang are described in Sections 4-7.

`‘quad 4` message is sent to `sObj2`, it is passed to `obj2` with a `‘self` component; that is, `sObj2 (‘quad 4)` has the same effect as `obj2 (‘self sObj2 & ‘quad 4)`. Because `obj2` does not change how `‘quad` messages are handled, this has the same effect as `sObj (‘self sObj2 & ‘quad 4)`. `sObj` is also a sealed object which adds a `‘self` component to the *left*; thus this has the same effect as `obj (‘self sObj & ‘self sObj2 & ‘quad 4)`. Because any pattern match will always match the rightmost `‘self`, the latter-sealed object is provided in the `‘self` added to the message passed to the `‘quad`-handling method and so any messages sent from that method will be dispatched to an object which includes the extensions in `sObj2`.

This code successfully typechecks object extension because, while we "tie the knot" on `self` using `seal`, we leave open the possibility of future *overriding* of `self`; it is merely a record element and we support record field override via asymmetric concatenation.

For examples in the remainder of the paper, we will assume that `seal` has been defined as above.

Onioning it all together Onions also provide a natural mechanism for including fields; we simply concatenate them to the scapes that represent the methods. Consider the following object which stores and increments a counter:

```
def obj = seal (‘x 0 &
     (‘inc _ & ‘self self ->
        (‘x x -> x = x + 1 in x) self))
in obj ‘inc ()
```

Label construction implicitly creates a mutable cell[2], so the `‘x` label is used to store the counter's current value. The bottom line invokes the scape part of `obj`; the `‘x` label does not interfere because it is not a scape. (Note that syntax `()` here is the "empty onion", the 0-ary conjunction of data items.)

In this body, `self` is passed to the inner scape. Because `seal` onions the target object onto the left of the self binding method, all of the labels from the unsealed object are still visible; thus, `self` has the same `‘x` label as `obj` and the cell within `obj`'s `‘x` label is bound to the variable `x`. The code `x = x + 1 in x` is then executed, incrementing the label's contents.

It may seem unusual that `obj` is, at the top level, a heterogeneous "mash" of a record field (the `‘x`) and a function (the scape which handles `‘inc`). This is sensible because onions are *type-indexed* [27], meaning that they use the types of the values themselves to identify data. When matching on an onion, the rightmost value is projected; `(7 & 3) + 1` is just `4`. Scapes are a special case; instead of projecting the rightmost scape, all scapes are projected and application selects the rightmost scape which matches the argument. This type-indexed view is useful because it leads to more concise code as seen in Section 2.3 below.

The above counter object code is quite concise considering that it defines a self-referential, mutable counter object using no syntactic sugar whatsoever in a core language with no explicit object syntax. But as we said before, we do not expect programmers to write directly in TinyBang under normal circumstances. Here we define just a few sugarings which we use in the examples throughout the remainder of this section, although the larger BigBang language would include sugarings for each of the features we are about to mention as well. When reading the sugarings below, reflect that it is still translucent in the sense we describe above: looking at desugared code is not prohibitive for programmers desiring more control.

```
o.x                 ≅ (‘x x -> x) o
o.x = e1 in e2      ≅ (‘x x -> x = e1 in e2) o
if e1 then e2       ≅ ((‘True _ -> e2) &
     else e3          (‘False _ -> e3)) e1
e1 and e2           ≅ ((‘True _ -> e2) &
                       (‘False _ -> ‘False ())) e1
```

Using this sugar, the third line of the counter object above can be more concisely expressed as `self.x = self.x + 1 in self.x`.

2.3 Flexible Object Operations

The above shows both a typed, variant-based object encoding for TinyBang as well as two flexible object features: first-class messages are trivial to encode and objects may be extended after they are messaged without compromising expressiveness. We now focus on typed encodings of flexible object operations as discussed in Section 1. Traditionally, these abstractions are defined in a first-order sense; inheritance, for instance, can only be expressed if the type of the parent class is statically known. In contrast, TinyBang's encoding can type higher-order abstractions.

We show our encoding in terms of objects rather than classes for simplicity; applying these concepts to classes is straightforward, and working at the object layer also shows how TinyBang can express (functional) object extension. For clarity, we use the sugar defined above.

Default arguments Many scripting languages include a notion of a "default argument": an argument which may be specified to a method but which will take on a default value if it is missing. Onions encode this behavior quite easily. For instance, consider:

```
def obj = seal
    ( (‘add (‘x x & ‘y y) -> x + y)
    & (‘sub (‘x x & ‘y y) -> x - y) ) in
def dflt = obj -> obj &
    (‘add a -> obj (‘add (‘x 1 & a))) in
def obj2 = dflt obj in
obj2 (‘add (‘y 3));
```

[2] The core MicroBang language we formalize is immutable, but in Section 7.1 we show how the MicroBang type system may be extended to include mutable state.

```
obj2 (‘add (‘x 7 & ‘y 2))
```

Given an object which accepts a message `‘add`, `dflt` will, in all `‘add` messages sent without an `‘x` component, include `‘x 1`. Because the `‘x 1` is onioned onto the left of `a`, it will have no effect if an `‘x` is provided in the message. Thus, the invocations on the last lines will yield `4` and `9`, respectively. Note also that, while most languages only permit default arguments in a first-order sense, TinyBang can encode default arguments as a higher-order transformation.

Overloading The pattern-matching semantics of scapes also provide a translucent mechanism whereby function (and thus method and operator) overloading can be defined. We might originally define negation on the integers as

```
def neg = x:int -> 0 - x in ...
```

Later code could then extend the definition of negation to include boolean values. Because operator overloading assigns new meaning to an existing symbol, we redefine `neg` to include all of the behavior of the old `neg` as well as new cases for `‘True` and `‘False`:

```
def neg = neg &! (‘True _ -> ‘False ())
              &! (‘False _ -> ‘True ()) in ...
```

The `&!` operator is TinyBang's *symmetric* concatenation syntax, ensuring that new overloadings produce a type error if they override existing definitions; `&!` is not in our core MicroBang type system but is added as an extension in Section 7.3. Negation is now overloaded: `neg 4` evaluates to `-4`, and `neg ‘True ()` evaluates to `‘False ()` due to how scape application matches patterns. Overloading as defined above shares similarities with the $\lambda\&$-calculus [10] and typed multimethod-based language designs [21, 22]; these projects, however, dispatch on nominal (class) names and not on structural information as we do. Note that our type system assumes whole-program compilation. Many of the subtleties in implementations of overloading arise from attempts to modularize it [5, 21]; we get to live in a simpler universe.

Mixins The following example shows how a simple two-dimensional `point` object can be combined with a `mixin` providing extra methods:

```
def point = seal (‘x 0 & ‘y 0
  & (‘l1 _ & ‘self self -> self.x + self.y)
  & (‘isZero _ & ‘self self ->
        self.x == 0 and self.y == 0)) in
def mixin = ((‘near _ &
    ‘self self -> (self ‘l1 ()) < 4)) in
def mixedPoint = seal (point & mixin) in
mixedPoint ‘near ()
```

The `point` variable is our original point object. The mixin is merely a scape which calls the value passed as `self`. Because an object's methods are just scapes onioned together, onioning the mixin into the point object is sufficient to produce the resulting mixed point; the `mixedPoint` variable contains an onion with `x` and `y` fields as well as all three scapes.

The above example is well-typed in TinyBang; parametric polymorphism is used to allow `point`, `mixin`, and `mixedPoint` to have different self-types. The `mixin` variable, the interesting part of the above code, has roughly the type "`(‘near unit & ‘self` α`)` $\rightarrow$ `boolean` where α is an object capable of receiving the `‘l1` message and producing an `int`". `mixin` can be onioned with any object that satisfies these properties. If the object does not have these properties, a type error will result when the `‘near` message is passed; for instance, `(seal mixin) (‘near ())` is not typeable because `mixin`, the value of `self`, does not have a scape which can handle the `‘l1` message.

TinyBang mixins are first-class values; the actual mixing need not occur until runtime. For instance, the following code selects a weighting metric to mix into a point based on some runtime condition `cond`.

```
def cond = (runtime boolean) in
def point = (as above) in
def w1 = (‘weight _ & ‘self self ->
            self.x + self.y) in
def w2 = (‘weight _ & ‘self self ->
            self.x - self.y) in
def mixedPoint = seal
    (point & (if cond then w1 else w2)) in
mixedPoint ‘weight ()
```

Traits can be encoded in a similar manner: each trait is an onion and objects are created by their concatenation. The encoding becomes slightly more complex when resolving conflicting methods; we omit the full discussion for brevity.

Inheritance, classes, and subclasses Typical object-oriented constructs can be defined in almost the same way. Object inheritance is accomplished in a fashion similar to mixins but with one exception: a variable `super` is bound to the original object and captured in the closure of the inheriting objects methods, allowing it to be reached for static dispatch. This allows overriding methods to invoke the methods they have overridden. The flexibility of the previously-defined `seal` function permits us to ensure that the inheriting object is used for future dispatches even in calls to overridden methods. Classes are defined quite simply as objects which generate other objects; subclasses are merely extensions of those object generating objects. We forgo any examples here for brevity.

2.4 A Record-Based Comparison

We have shown how an object model amenable to flexible operations can be defined using onions and scapes. In our encoding, we chose a variant-based approach. Since records and variants are natural duals, however, we now offer a high-level comparison with a record-based encoding.

TinyBang's scapes and onions can also be used to build a pure record-based encoding of objects that would include support for first-class messages. Unlike previous encodings of first-class messages [23, 24], TinyBang provides a view which uses subtyping and dependent pattern types and does not require conditional constraints, row types, or a higher-kinded system. Such a record-based encoding is the true dual of the variant-based encoding presented above; the variant-encoded message `‘msg arg`, for example, dualizes to `obj -> obj.msg arg` in the record-encoded system.

It is also possible to use first-class labels to form a record-based encoding of objects; first-class messages are then easily encoded with first-class labels. Fluid object types [13] are a recent system which uses this approach to infer types in scripting languages such as JavaScript. (Note that we do intend to investigate in the future whether the addition of first-class labels to TinyBang will give us additional added flexibility; they should not be theoretically challenging to add.)

We focus on a variant-based encoding because it leads to a more translucent and direct syntax. For instance, the variant first-class message `‘msg arg` may be directly pattern matched, but the record-based function representation is harder to deconstruct. The flexible object operations we define above are also more opaque in the record-based view. In the above encoding, one adds a method `mthd` to an object `obj` simply by writing `obj & mthd`. In the record-based view, where methods are labeled functions which are projected when they are called, one must write the considerably more cumbersome

```
1 obj & ‘mthd (( (_ -> mthd) &
2               (‘mthd old -> old & mthd)) obj)
```

to account for the fact that the original object may or may not already have a `‘mthd` component. The implementation of a `seal` method for the record-based view is very complex; we omit it here for brevity.

Much of the terseness in the TinyBang encoding arises due to the fact that it cleanly puts fields and methods into different syntactic sorts: labels and scapes, respectively. It is possible to get by with only one of these sorts much like it is possible to use only one of $\wedge$ or $\vee$ in Boolean logic: DeMorgan's law allows one to be written in terms of the other, but the resulting expressions are opaque and unintuitive. In this sense, the encoding that we provide is inspired by the variant philosophy but is in fact a hybrid encoding in which methods are on the variant side of the record-variant duality and fields are on the record side. This hybrid view provides a very natural encoding of self-awareness (in the form of the `seal` function) without complicating field access.

3. MicroBang Operational Semantics

We now begin formal discussion of the operational semantics and typing of the language features presented in Section 2. Because the language is reasonably complex, we begin our discussion with MicroBang, a simplified TinyBang lacking complex patterns, mutable state, and the advanced onion operators `&-`, `&.`, and `&!`. This section defines a small-step operational semantics for MicroBang and subsequent sections address typing.

$$
\begin{array}{rcll}
e & ::= & \overrightarrow{x = r} & \textit{expressions} \\
r & ::= & v \mid x\ x \mid x\ \mathrm{op}\ x & \textit{redices} \\
v & ::= & x \mid \mathbb{Z} \mid \mathtt{lbl}\ x \mid x\ \mathtt{\&}\ x \mid \mathtt{()} \mid \phi\ \mathtt{->}\ e & \textit{values} \\
\phi & ::= & x\ \mathtt{:}\ \varphi & \textit{patterns} \\
\varphi & ::= & \mathtt{int} \mid \mathtt{lbl}\ x \mid \mathtt{any} & \textit{primary patterns} \\
x & & & \textit{variables} \\
\mathrm{op} & ::= & \mathtt{+} \mid \mathtt{==} & \textit{operators} \\
\mathtt{lbl} & ::= & \mathtt{‘[a\text{-}zA\text{-}Z0\text{-}9\_]^+} & \textit{label names} \\
\pi & ::= & \mathtt{int} \mid \mathtt{lbl} \mid \mathtt{fun} & \textit{projectors}
\end{array}
$$

Figure 3.1. MicroBang Syntax

Notation For a given grammatic construct $*$, we let $[*_1, \ldots, *_n]$ denote a list of $*$'s, often using the equivalent shorthand $\overrightarrow{*_\square}^n$; 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: $\overbrace{*_\square}^n$ abbreviates $\{*_1, \ldots, *_n\}$. We only use this notation on *finite* lists and sets in this paper.

3.1 MicroBang Grammar

We present the grammar for MicroBang in Figure 3.1. This grammar is written in A-normal form to simplify the small-step operational semantics presented below. The A-translation from a more traditional recursive grammar is standard and is not defined here for sake of brevity. As an example, the expression `4 + 5 == 9` would translate to the A-normal expression $[$`x1 = 4, x2 = 5, x3 = x1+x2, x4 = 9, x5 = x3==x4`$]$: each program point is assigned its own variable and variable names are unique. For the remainder of the paper, we will assume that we work only over ANF programs e with unique variable names.

3.2 Operational Semantics

The operational semantics of MicroBang proceed by evaluating each redex in the expression from left to right. Let E be of the form $\overrightarrow{x = v}$; that is, E is the subset of e for which each redex is already a value. E constitutes both the environment and the evaluation context in our semantics. For such an $E = \overrightarrow{x = v}$, we define $E(x) = v$ to be a lookup function defined when $x = x_i$ for some $1 \leq i \leq n$. When v_i is a non-variable value, $v = v_i$; otherwise, $v = E(v_i)$ recursively.

Projection The projectors π in the grammar represent the different outermost value sorts. We use notation $v \in \pi$ to denote that the projector π matches the outermost structure of a given value v; for example, `‘A x` $\in$ `‘A`. The special projector `fun` matches all scapes (and only scapes). No projector

matches onions directly. We then define a simple function $E{\downarrow}^*_\pi(x)$ which projects from a given value x all values which match the projector π; there can be more than one result because if x is an onion, the result is the list of all values which match the projector.

Definition 3.1 (Value Projection)**.**

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

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

Definition 3.2 (Single Value Projection)**.**

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

Note that the function $E{\downarrow}_\pi(x)$ is partial because it is undefined for cases in which the value x does not contain any values which match the projector.

Compatibility The next relation we use in our definition of MicroBang's operational semantics is compatibility. Compatibility holds only when a given value matches a given pattern. If it does, then the match is with respect to a list of bindings E_0 which describes how the contents of the argument are assigned to the pattern's variables. For convenience, we define this relation on both patterns (ϕ) and primary patterns (φ). We define $x \preceq_E \phi \backslash E_0$ by cases on the structure of the pattern as follows:

Definition 3.3 (Value Compatibility)**.**

$x_1 \preceq_E x_2 : \varphi \backslash E_0 \parallel [x_2 = x_1]$	if	$x_1 \preceq_E \varphi \backslash E_0$
$x_1 \preceq_E \texttt{int} \backslash []$	if	$E{\downarrow}_{\texttt{int}}(x_1) \in \mathbb{Z}$
$x_1 \preceq_E \text{lbl}\, x_2 \backslash [x_2 = x_3]$	if	$E{\downarrow}_{\text{lbl}}(x_1) = \text{lbl}\, x_3$
$x_1 \preceq_E \texttt{any} \backslash []$	always	

We write $x \not\preceq_E \phi$ to indicate that there exists no E_0 such that $x \preceq_E \phi \backslash E_0$.

We then use the above compatibility relation to define an application-driven form of compatibility. We write $x \preceq_E \overrightarrow{\phi \texttt{ -> } e} \backslash E_0; e$ to denote that the argument x can be applied to one of the scapes in $\overrightarrow{\phi \texttt{ -> } e}$; in this case, E_0 is the set of bindings from x to the pattern and e is the body of the scape to be applied. We define this relation below by induction on the length of the list of scapes. This definition will always use the rightmost applicable scape.

Definition 3.4 (Value Application Compatibility)**.**

$x \preceq_E [] \backslash E_0; e$	is false
$x \preceq_E \overrightarrow{\phi \texttt{ -> } e} \parallel [\phi' \texttt{ -> } e'] \backslash E_0; e'$	if $x \preceq_E \phi' \backslash E_0$
$x \preceq_E \overrightarrow{\phi \texttt{ -> } e} \parallel [\phi' \texttt{ -> } e'] \backslash E_0; e''$	if $x \not\preceq_E \phi' \wedge x \preceq_E \overrightarrow{\phi \texttt{ -> } e} \backslash E_0; e''$

Small-Step Semantics Using the relations above, we now define small-step computation for MicroBang. 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. Our single-step relation is then defined over closed e with unique variable names as follows; we give intuitions below.

Definition 3.5 (MicroBang Small-Step Semantics)**.**

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

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

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

$$E \parallel [x = x_1\ x_2] \parallel e'' \longrightarrow^1 E \parallel \alpha(E_0 \parallel e' \parallel [x' = r]) \parallel [x = \alpha(x')] \parallel e''$$
when $E{\downarrow}^*_{\texttt{fun}}(x_1) = \overrightarrow{v}, x_2 \preceq_E \overrightarrow{v} \backslash E_0; e' \parallel [x' = r]$

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 first 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 find the pattern bindings E_0 and the body $e' \parallel [x' = r]$ of the matching scape. 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 $\alpha(E_0 \parallel e' \parallel [x' = r]) \parallel [x = \alpha(x')]$: the pattern bindings for the argument, the body of the matched scape, and a final definition 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 the freshening will freshen both occurrences of x' to the same fresh variable here.

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

4. A Monomorphic MicroBang Type System

In order to simplify the presentation of MicroBang's type system, we first present a monomorphic type system we call MicroBang^0. The MicroBang^0 type system includes all features of MicroBang syntax and types except polymorphism. This allows us to show how scape application is typed with-

$$
\begin{array}{rcll}
\alpha & & & \textit{type variables} \\
\boldsymbol{\alpha} & ::= & \overbrace{\alpha} & \textit{type variable sets} \\
\Gamma & ::= & \overbrace{x : \alpha} & \textit{contexts} \\
\tau & ::= & \mathtt{int} \mid \mathrm{lbl}\,\alpha \mid \alpha \,\&\, \alpha \mid () \mid \tau_\phi \to \alpha \setminus \mathrm{C} & \textit{types} \\
\tau_\phi & ::= & \alpha{\sim}\tau_\varphi & \textit{pattern types} \\
\tau_\varphi & ::= & \mathtt{int} \mid \mathrm{lbl}\,\alpha \mid \mathtt{any} & \textit{primary pattern types} \\
\mathrm{c} & ::= & \tau <: \alpha \mid \alpha <: \alpha \mid \alpha\ \alpha <: \alpha \mid \alpha\,\mathrm{op}\,\alpha <: \alpha & \textit{constraints} \\
\mathrm{C} & ::= & \overbrace{\mathrm{c}} & \textit{constraint sets}
\end{array}
$$

Figure 4.1. MicroBang^0 Type Grammar

out the distraction of the complex polymorphism model. Polymorphic MicroBang is presented in Section 5.

Typechecking in MicroBang^0 consists of two phases: initial derivation, in which the program is translated to a type variable and a set of initial constraints over that variable; and closure, during which the initial constraint set is deductively closed over a logical system and then checked for consistency. We begin our discussion with the type and constraint grammar shown in Figure 4.1.

The types τ in the type grammar represent concrete *lower* bounds in MicroBang^0's type system. They are, informally and respectively, primitives, labels of cells, the onion of two other types, the empty onion type, and functions. MicroBang^0's constraint grammar is restricted in that all subtype constraint upper bounds are type variables; this canonical form is nonetheless complete and facilitates reasoning. Note that we use $\alpha_1\ \alpha_2 <: \alpha_3$ which is equivalent to the more standard notation $\alpha_1 <: \alpha_2 \texttt{->} \alpha_3$.

4.1 Initial Derivation

Type checking in MicroBang^0 is performed against a fixed, closed expression e. The first step of typechecking is to translate e into a type variable and a set of initial constraints. In order to define this operation concisely, we first give some supplementary definitions.

Initial type derivation is performed with respect to a context Γ. We still operate under the assumption from Section 3.1 that variable bindings in our programs are unique; thus, new bindings can be safely introduced to a context with set union. Throughout type derivation, we also require fresh type variables. We write $\dot{\alpha}_i$ to denote the ith fresh variable relative to a given point in translation.

We define initial derivation as three functions: $[\![e]\!]^\Gamma_{\mathrm{E}}$, $[\![x = r]\!]^\Gamma_{\mathrm{R}}$, and $[\![\phi]\!]^\Gamma_{\mathrm{P}}$. These functions define derivation over expressions, redex definitions, and patterns, respectively. Note that the letters E, R, and P are not variables; they distinguish the sort of initial derivation which is being performed.

We write $[\![e]\!]^\Gamma_{\mathrm{E}} = \langle \alpha, \mathrm{C} \rangle$ to indicate that an expression e in context Γ translates to the type α constrained by C. Expressions contain redex definitions; we write $[\![x = r]\!]^\Gamma_{\mathrm{R}} = \mathrm{c}$ to indicate that the redex definition $x = r$ translates to the constraint c. Because these definitions may contain scapes

$$
\begin{array}{rl}
[\![[x = r]]\!]^\Gamma_{\mathrm{E}} & = \langle \dot{\alpha}_1, \{[\![x = r]\!]^{\Gamma \cup \Gamma'}_{\mathrm{R}}\} \rangle \\
& \quad \text{where } \Gamma' = \{x : \dot{\alpha}_1\} \\
[\![[x = r] \parallel e]\!]^\Gamma_{\mathrm{E}} & = \langle \alpha', \{[\![x = r]\!]^{\Gamma \cup \Gamma'}_{\mathrm{R}}\} \cup \mathrm{C} \rangle \\
& \quad \text{where } \Gamma' = \{x : \dot{\alpha}_1\}, [\![e]\!]^{\Gamma \cup \Gamma'}_{\mathrm{E}} = \langle \alpha', \mathrm{C} \rangle
\end{array}
$$

$$
\begin{array}{rl}
[\![x = x']\!]^\Gamma_{\mathrm{R}} & = \Gamma(x') <: \Gamma(x) \\
[\![x = \mathrm{lbl}\, x']\!]^\Gamma_{\mathrm{R}} & = \mathrm{lbl}\,\Gamma(x') <: \Gamma(x) \\
[\![x = x' \,\&\, x'']\!]^\Gamma_{\mathrm{R}} & = \Gamma(x') \,\&\, \Gamma(x'') <: \Gamma(x) \\
[\![x = \phi \texttt{->} e]\!]^\Gamma_{\mathrm{R}} & = \tau_\phi \to \alpha \setminus \mathrm{C} <: \Gamma(x) \\
& \quad \text{where } \begin{cases} [\![\phi]\!]^\Gamma_{\mathrm{P}} = \langle \Gamma', \tau_\phi \rangle, \\ [\![e]\!]^{\Gamma \cup \Gamma'}_{\mathrm{E}} = \langle \alpha, \mathrm{C} \rangle, \end{cases} \\
[\![x = ()]\!]^\Gamma_{\mathrm{R}} & = () <: \Gamma(x) \\
[\![x = \mathbb{Z}]\!]^\Gamma_{\mathrm{R}} & = \mathtt{int} <: \Gamma(x) \\
[\![x = x'\ x'']\!]^\Gamma_{\mathrm{R}} & = \Gamma(x')\ \Gamma(x'') <: \Gamma(x) \\
[\![x = x'\ \mathrm{op}\ x'']\!]^\Gamma_{\mathrm{R}} & = \Gamma(x')\,\mathrm{op}\,\Gamma(x'') <: \Gamma(x)
\end{array}
$$

$$
\begin{array}{rl}
[\![x_1 : \mathtt{int}]\!]^\Gamma_{\mathrm{P}} & = \langle \{x_1 : \dot{\alpha}_1\}, \dot{\alpha}_1 {\sim} \mathtt{int} \rangle \\
[\![x_1 : \mathrm{lbl}\, x_2]\!]^\Gamma_{\mathrm{P}} & = \langle \{x_1 : \dot{\alpha}_1, x_2 : \dot{\alpha}_2\}, \dot{\alpha}_1 {\sim} \mathrm{lbl}\,\dot{\alpha}_2 \rangle \\
[\![x_1 : \mathtt{any}]\!]^\Gamma_{\mathrm{P}} & = \langle \{x_1 : \dot{\alpha}_1\}, \dot{\alpha}_1 {\sim} \mathtt{any} \rangle
\end{array}
$$

Figure 4.2. MicroBang^0 Initial Type Derivation

and each scape has a pattern, we write $[\![\phi]\!]^\Gamma_{\mathrm{P}} = \langle \Gamma', \tau_\phi \rangle$ to indicate that a pattern ϕ in context Γ translates to another context Γ' which contains the bindings for the pattern as well as a pattern type τ_ϕ. Using this notation, we define the initial type derivation process in Figure 4.2.[3]

Much of this translation process is direct, substituting variables with their corresponding type variables and modeling value assignment as a lower bound. The translation of a scape creates a type which captures the constraints derived from its body. These constraints are introduced during closure only if the scape is used; otherwise, they have no effect on typechecking.

4.2 Constraint Closure

The second step in the typechecking process is to deductively close over the set of constraints produced by derivation. To define the closure relation, we must first specify other relations which mirror those used in the small-step semantics in Section 3. We begin by defining *concretization*, a simple relation which locates the lower bounds for a given type via transitivity chains in the constraint set.

Definition 4.1 (Concretization)**.** $\tau\ {}^{\mathrm{C}}{<:^*}\ \alpha_n$ iff $\{\tau <: \alpha_0, \alpha_0 <: \alpha_1, \ldots, \alpha_{n-1} <: \alpha_n\} \subseteq \mathrm{C}$ where $n \geq 0$.

Unlike the lookup function $E(x)$ from the operational semantics, concretization is a relation. In cases where the precise type is not statically known (such as a condition, the possible types of which are `'True ()` or `'False ()`), we approximate it by using a union of the possible types. As a

[3] Recall from the top of Section 3 that we use $\square$ for indexing positions. For instance, we would write $\overbrace{x_\square + 1}^{n}$ to denote $[x_1 + 1, \ldots, x_n + 1]$.

result, a given type variable may have many lower bounds and so there may be many concretizations for a single α_n.

Closure must also create boolean labels which contain the empty onion `()` similar to how the operational semantics create a fresh x' when a boolean is created. To address this, we define a mapping $\text{ETV}(\cdot)$ to provide fresh variables:

Definition 4.2 (Empty Onion Variables)**.** For all α in the initial derivation, $\text{ETV}(\alpha) = \alpha'$ where α' is a fresh type variable not in the initial derivation.

Projection Next we define projection, a relation which extracts from a type variable all type lower bounds matching a given projector. We write $\mathrm{C} \vdash \alpha \xrightarrow{\pi\triangleright} \vec{\tau}$ to denote that $\vec{\tau}$ are the non-onion components of the lower-bounding types of α which match the projector π. Note that this relation is not a function; if α has multiple lower bounds, for instance, it will relate multiple type lists. In our definition, we use a relation $\tau \sqsubseteq \pi$ to determine if a non-onion type matches a given projector. It is true in exactly the following cases:

Definition 4.3 (Projector Match)**.**

$$\begin{array}{rl} \texttt{int} \sqsubseteq \texttt{int} & \\ \text{lbl}\,\alpha \sqsubseteq \text{lbl} & \text{for any } \alpha \\ \tau_\phi \to \alpha \setminus \mathrm{C} \sqsubseteq \texttt{fun} & \text{for any } \tau_\phi, \alpha, \mathrm{C} \end{array}$$

Using this predicate, we define projection as follows:

Definition 4.4 (Projection)**.**

$$\begin{array}{lll} \mathrm{C} \vdash \alpha' \xrightarrow{\pi\triangleright} [] & \text{if} & \tau \overset{\mathrm{C}}{<:^*} \alpha', \tau \not\sqsubseteq \pi, \tau \text{ is non-onion} \\ \mathrm{C} \vdash \alpha' \xrightarrow{\pi\triangleright} [\tau] & \text{if} & \tau \overset{\mathrm{C}}{<:^*} \alpha', \tau \sqsubseteq \pi \\ \mathrm{C} \vdash \alpha' \xrightarrow{\pi\triangleright} \vec{\tau}_1 \,\|\, \vec{\tau}_2 & \text{if} & \alpha_1 \,\&\, \alpha_2 \overset{\mathrm{C}}{<:^*} \alpha', \\ & & \mathrm{C} \vdash \alpha_1 \xrightarrow{\pi\triangleright} \vec{\tau}_1, \mathrm{C} \vdash \alpha_2 \xrightarrow{\pi\triangleright} \vec{\tau}_2 \end{array}$$

Projection from a non-onion type is quite simple: produce a singleton list containing the type if it matches the projector and produce an empty list if it does not. Projection from an onion involves concatenating the projection from each side; the result is a priority-ordered list. This relation is decidable, although calculating it efficiently is technically challenging in some rare cases. We discuss this in Appendix B.

We often want only the highest priority (rightmost) type from an onion which matches a given projector; this reflects the rightmost priority of onioning. For this purpose, we overload projection syntax with a single projection relation:

Definition 4.5 (Single Projection)**.**

$\mathrm{C} \vdash \alpha \xrightarrow{\pi\triangleright} \tau'$ holds iff $\mathrm{C} \vdash \alpha \xrightarrow{\pi\triangleright} \vec{\tau} \,\|\, [\tau']$ holds for some $\vec{\tau}$.

Note that this relation is partial; it does not hold when there are no types in any concrete form of α matching the projector π.

Compatibility We now define a type-level compatibility relation to model the compatibility relation of operational semantics. To do so, we define $\dot{\mathrm{C}}$ to range over "possible constraint sets"; each $\dot{\mathrm{C}}$ is either a set of constraints C or the special symbol ❁. We define union over possible constraint sets as $\dot{\mathrm{C}}_1 \cup \dot{\mathrm{C}}_2 = \mathrm{C}_1 \cup \mathrm{C}_2$ when both $\dot{\mathrm{C}}_1 = \mathrm{C}_1$ and $\dot{\mathrm{C}}_2 = \mathrm{C}_2$; we define it as ❁ when either $\dot{\mathrm{C}}_1$ or $\dot{\mathrm{C}}_2$ is ❁.

We write $\alpha \preceq_{\mathrm{C}} \tau_\phi \backslash \dot{\mathrm{C}}$ to describe how a type variable α may match the pattern τ_ϕ. When $\dot{\mathrm{C}}$ is C, this relation indicates that those constraints allow the type information from α to flow into the variables in τ_ϕ (similar to how E_0 describes how data flows from an argument x into a pattern ϕ in the small-step semantics). When $\dot{\mathrm{C}}$ is ❁, this relation indicates that the argument does not match the pattern. The ❁ notation is helpful here because α may have multiple lower bounds; it is thus possible for one of the forms of α to match the pattern while another does not.

We define type-level compatibility by casing on the pattern type. As in the operational semantics, we define compatibility on both patterns and primary patterns.

Definition 4.6 (Compatibility)**.**

$$\begin{array}{lll} \alpha_1 \preceq_{\mathrm{C}} \alpha_2 \sim \tau_\varphi \backslash \{\alpha_1 <: \alpha_2\} \cup \dot{\mathrm{C}} & \text{if} & \alpha_1 \preceq_{\mathrm{C}} \tau_\varphi \backslash \dot{\mathrm{C}} \\ \alpha_1 \preceq_{\mathrm{C}} \texttt{int} \backslash \emptyset & \text{if} & \mathrm{C} \vdash \alpha_1 \xrightarrow{\texttt{int}\triangleright} \texttt{int} \\ \alpha_1 \preceq_{\mathrm{C}} \text{lbl}\,\alpha_2 \backslash \{\alpha_3 <: \alpha_2\} & \text{if} & \mathrm{C} \vdash \alpha_1 \xrightarrow{\text{lbl}\triangleright} \text{lbl}\,\alpha_3 \\ \alpha_1 \preceq_{\mathrm{C}} \texttt{any} \backslash \emptyset & \text{always} & \\ \alpha_1 \preceq_{\mathrm{C}} \texttt{int} \backslash ❁ & \text{if} & \mathrm{C} \vdash \alpha_1 \xrightarrow{\texttt{int}\triangleright} [] \\ \alpha_1 \preceq_{\mathrm{C}} \text{lbl}\,\alpha_2 \backslash ❁ & \text{if} & \mathrm{C} \vdash \alpha_1 \xrightarrow{\text{lbl}\triangleright} [] \end{array}$$

Just as in the small-step semantics in Section 3, we now use this compatibility relation to define an application-driven form of compatibility. We write $\alpha \preceq_{\mathrm{C}} \left(\overrightarrow{\tau\phi'_{\square} \to \alpha'_{\square} \setminus \mathrm{C}'_{\square}}\right) \backslash \alpha''; \mathrm{C}''$ to indicate that α, the type in the argument position of an application, is compatible with the pattern of one of the scape types in the specified scape list. α'' and C'' represent the return type and constraint set of the matched pattern's scape. We define this relation as follows:

Definition 4.7 (Application Compatibility)**.**

$$\begin{array}{ll} \alpha \preceq_{\mathrm{C}} []\backslash\alpha''; \mathrm{C}'' & \text{is false} \\ \alpha \preceq_{\mathrm{C}} \vec{\tau} \,\|\, [(\tau\phi' \to \alpha' \setminus \mathrm{C}')]\backslash\alpha'; \mathrm{C}'' \cup \mathrm{C}' & \text{if } \alpha \preceq_{\mathrm{C}} \tau\phi'\backslash\mathrm{C}'' \\ \alpha \preceq_{\mathrm{C}} \vec{\tau} \,\|\, [(\tau\phi' \to \alpha' \setminus \mathrm{C}')]\backslash\alpha''; \mathrm{C}'' & \text{if } \alpha \preceq_{\mathrm{C}} \tau\phi'\backslash❁, \\ & \quad \alpha \preceq_{\mathrm{C}} \vec{\tau}\backslash\alpha''; \mathrm{C}'' \end{array}$$

Again following the small-step semantics, this relation will always use the rightmost available scape. We write $\alpha \not\preceq_{\mathrm{C}} \overrightarrow{{}^{n}\tau\phi'_{\square} \to \alpha'_{\square} \setminus \mathrm{C}'_{\square}}$ to denote that $\forall i \in \{1..n\}.\ \alpha \preceq_{\mathrm{C}} \tau\phi'_i\backslash❁$: that is, that every pattern could fail to match α.[4]

Constraint Closure We are now equipped to define the constraint closure relation described above. We write $\mathrm{C}_{\text{IN}} \xrightarrow{\text{CL}1} \mathrm{C}_{\text{OUT}}$ to denote that a single step of closure maps C_{IN} onto C_{OUT}. The definition of this relation appears in Figure 4.3, which uses the following notational sugar. In general, each rule assumes constraint closure to be performed over some set of constraints C_{IN}. We write a constraint set C_{NEW} as the conclusion of a rule to abbreviate $\mathrm{C}_{\text{IN}} \xrightarrow{\text{CL}1} \mathrm{C}_{\text{IN}} \cup \mathrm{C}_{\text{NEW}}$. We may also write a single constraint c as the conclusion of a rule to abbreviate $\{\mathrm{c}\}$. If a constraint c appears in the premise of a rule, we take it to mean $\mathrm{c} \in \mathrm{C}_{\text{IN}}$.

[4] This definition of compatibility is sufficient for soundness but is overly conservative due to a form of union misalignment; it is possible for each pattern to fail to match α in a different way but for no single instance of α to fail to match all patterns. We solve this problem by modifying the relation to include a *fibration*, which describes concrete type structure, and then ensuring alignment between these fibrations. We defer presentation of this solution until Section 6 to simplify the initial type system we are describing here.

APPLICATION

$$\frac{\alpha_1\ \alpha_2 <: \alpha_3 \qquad \mathrm{C}_{\text{IN}} \vdash \alpha_1 \xrightarrow{\texttt{fun}\triangleright} \vec{\tau} \qquad \alpha_2 \preceq_{\mathrm{C}_{\text{IN}}} \vec{\tau} \backslash \alpha_4; \mathrm{C}'}{\mathrm{C}' \cup \{\alpha_4 <: \alpha_3\}}$$

INTEGER ADDITION

$$\frac{\alpha_1 \texttt{ + } \alpha_2 <: \alpha_3 \qquad \mathrm{C}_{\text{IN}} \vdash \alpha_1 \xrightarrow{\texttt{int}\triangleright} \texttt{int} \qquad \mathrm{C}_{\text{IN}} \vdash \alpha_2 \xrightarrow{\texttt{int}\triangleright} \texttt{int}}{\texttt{int} <: \alpha_3}$$

INTEGER EQUALITY

$$\frac{\alpha_1 \texttt{ == } \alpha_2 <: \alpha_3 \qquad \alpha_3' = \mathrm{ETV}(\alpha_3) \qquad \mathrm{C}_{\text{IN}} \vdash \alpha_1 \xrightarrow{\texttt{int}\triangleright} \texttt{int} \qquad \mathrm{C}_{\text{IN}} \vdash \alpha_2 \xrightarrow{\texttt{int}\triangleright} \texttt{int}}{\{\texttt{`True}\ \alpha_3' <: \alpha_3, \texttt{`False}\ \alpha_3' <: \alpha_3, \texttt{()} <: \alpha_3'\}}$$

Figure 4.3. MicroBang^0 Constraint Closure

Each constraint closure rule's first premise requires the presence of a constraint for a particular grammar form. The Integer Addition rule, for instance, requires a constraint of the form $\alpha_1 \texttt{ + } \alpha_2 <: \alpha_3$; Figure 4.2 shows us that each redex of the form $e_1 \texttt{ + } e_2$ produces such a constraint. We then use projection to confirm that each type variable has `int` as a lower bound; if so, then we can conclude $\texttt{int} <: \alpha_3$, which indicates that the result also has a lower bound of `int`. (We consider type errors, such as when either side has a non-`int` lower bound, when we detect contradictions below.) The Integer Equality rule works in a similar fashion but produces two constraints; this is because the result is not known until runtime and a union of the types is a conservative approximation. In this rule, α' is the variable associated with α_3 which models the fresh x' in the operational semantics.

The application rule is also similar to the integer rules. Given a constraint that indicates that a function call occurred, we first use projection to determine which functions might have been available. We then use the compatibility relation to determine which of the scapes might have been suitable for the argument; for such a suitable scape, we conclude the constraints describing the body of that scape as well as an additional constraint $\alpha_3 <: \alpha_2$ which ensures that the result type of the scape's computation flows into the output point for the call site.

In parallel to the operational semantics, we define a relation $\mathrm{C}_0 \xrightarrow{\text{CL}}{}^* \mathrm{C}_n$ to hold when $\mathrm{C}_0 \xrightarrow{\text{CL}}{}^1 \ldots \xrightarrow{\text{CL}}{}^1 \mathrm{C}_n$ for $n \geq 0$.

Contradictions After constraint closure is complete, detecting contradictions simply involves finding inconsistencies in the constraint set. We define such inconsistencies as follows:

Definition 4.8 (Inconsistency)**.** A set of constraints C is *inconsistent* iff either

$\exists \alpha_1\ \alpha_2 <: \alpha_3 \in \mathrm{C}$ such that $\mathrm{C} \vdash \alpha_1 \xrightarrow{\texttt{fun}\triangleright} \vec{\tau}$ and $\alpha_2 \not\preceq_{\mathrm{C}} \vec{\tau}$, or
$\exists \alpha_1\ \mathrm{op}\ \alpha_2 <: \alpha_3 \in \mathrm{C}$ such that $\mathrm{C} \vdash \alpha_i \xrightarrow{\texttt{int}\triangleright} []$ for some $i \in \{1..2\}$

A set of constraints is *consistent* if it is not inconsistent.

These conditions check for direct conflicts. In the latter condition, we detect non-integer arguments passed to an integer binary operation. In the prior, we detect an onion of scapes α_0 for which some portion of its argument α_1 is incompatible with every scape in the onion. Note that this also detects the application of a non-scape: if α_0 contains no scapes, projection will produce an empty list which then trivially satisfies the incompatibility condition.

Typechecking is then simple. Given an expression e, we first calculate the set of constraints C such that $[\![e]\!]^{\emptyset}_{\text{E}} = \langle \alpha, \mathrm{C} \rangle$ (for some α). We then calculate the closure such that $\mathrm{C} \xrightarrow{\text{CL}}{}^* \mathrm{C}'$. Finally, we determine if C' is inconsistent. If it is, then we reject e; otherwise, e is declared type safe.

Because the polymorphic type system presented in the next section is a strict generalization of MicroBang^0, we defer discussion of type soundness to the next section.

5. Polymorphism

We now extend MicroBang^0 with parametric polymorphism to create a complete type system for MicroBang. Our polymorphism inference system is complex and so is placed in this separate section for readability. We are designing this system in the context of scripting languages, so we aim to realize the intuitions of types and subtyping held by programmers accustomed to dynamically-typed languages: a method call which can be understood using the philosophy of duck typing should simply work. For this reason, we base our system on expressive polymorphic set constraint type systems [3, 12, 14, 24, 31] adding several extensions for greater expressiveness.

Parametric polymorphism is important for expressiveness; we would like to avoid the arbitrary polymorphism cutoffs found in `let`-polymorphism and in local type inference. In order to obtain the maximal amount of polymorphism, every scape in MicroBang is inferred a polymorphic type; this approach is inspired by flow analyses [1, 28] and has previously been ported to a type constraint context [31]. In the ideal, each call site produces a fresh instantiation of the type variables. Unfortunately, this ideal cannot be implemented since a single program can have an unbounded number of function call sequences and so produce infinitely many instantiations. A standard solution to dealing with this case in the program analysis literature is to simply chop off call sequences at some fixed point. While such arbitrary cutoffs may work for program analyses, they work less well for type systems: they make the system hard for users to understand and potentially brittle to small refactorings.

We have developed an approach here starting from our previous work on this topic [16, 17, 31] to produce polymorphism on functions which is "nearly maximal" in that common programming patterns have expressive polymorphism, suffer no arbitrary cutoffs, and are not too inefficient. In this approach, a call to a non-recursive function is maximally polymorphic. Recursive call cycles are polymorphic only on the first traversal; type variables are reused when a recursive cycle is closed. Because scapes are higher-order in

MicroBang, it is not known at the start of closure which series of calls will be cyclic. Instead, the MicroBang type system optimistically instantiates fresh variables for each call sequence it encounters and then merges these variables when a recursive cycle is discovered.

This section makes adjustments to the MicroBang^0 type system at each step – initial derivation, constraint closure, and contradiction analysis – to introduce polymorphism. We explain the mechanism we use to recognize recursive call cycles and to unify type variables when appropriate. We then present formal statements about the properties of our type system and describe our strategy for proving them correct.

5.1 Initial Derivation

To define initial type derivation for the MicroBang type system, we must first adjust our view of type variables. Type variables in MicroBang^0, as in most type systems, are uninterpreted; they are terminal forms in the grammar. In the full MicroBang type system, however, we need to record how each variable is polyinstantiated. To do this, we use a *contour*, a structure which records a polyinstantiation context. The operations over contours are somewhat complex and quite difficult to motivate without a usage context; we therefore defer their presentation until Section 5.2. For now, we treat contours $\mathcal{C}$ as opaque, uninterpreted symbols.

The most direct way to associate a type variable with its contour information is to assign to it some structure. We begin by assuming that each redex and each pattern in the program is associated with some unique *identifier* ι. With this notation, our type variable grammar is as follows:

Definition 5.1 (Type Variable Grammar)**.**

$$\begin{array}{rcll} \dot{\mathcal{C}} & ::= & \mathcal{C} \mid \text{❁} & \textit{possible contours} \\ \alpha & ::= & \langle \iota, \mathbb{Z}, \dot{\mathcal{C}} \rangle & \textit{type variables} \end{array}$$

Each type variable is now a triple between an identifier, an index, and a possible contour. The identifier names the point in the program for which the type variable was originally inferred. The index allows multiple type variables to be used when inferring a type for a given expression. Type variables also have a *possible contour* $\dot{\mathcal{C}}$: either a contour $\mathcal{C}$ or the special symbol ❁ (which we describe here as "no contour"). The contour of a type variable describes how it was polyinstantiated; a type variable with no contour has not been polyinstantiated.

Initially, all type variables are inferred with no contour. In fact, the definition of initial type derivation given in Figure 4.2 is still largely suitable for the polymorphic system; we must make two small adjustments. First, rather than taking $\dot{\alpha}_i$ to represent an arbitrary fresh type variable, we give fresh type variables specific structure:

Definition 5.2 (Fresh Variables)**.** In initial expression type derivation, $\dot{\alpha}_i = \langle \iota, i, \text{❁} \rangle$ where ι is the identifier of the redex for which we are currently deriving a type. The same is true in initial pattern derivation, where ι is the identifier of the pattern.

Second, we must capture the free type variables of functions during derivation so that they may be correctly polyinstantiated later; this parallels the process of type generalization in more traditional type systems. To accomplish this, we first modify the type grammar of the language so that scape types take the following form:

Definition 5.3 (Concrete Type Grammar)**.**

$$\tau \ ::= \ \ldots \mid \forall \boldsymbol{\alpha}.\tau_\phi \to \alpha \setminus \mathrm{C}$$

The adjustments which appear in Definitions 5.1 and 5.3 are the only changes to the type grammar we must make for the polymorphic MicroBang type system. Otherwise, the type grammar we use is the same as in Figure 4.1.

Next, we define derivation over scape expressions, using $\textsc{ftv}(\cdot)$ as a function calculating free type variables:

Definition 5.4 (Initial Scape Type Derivation)**.**

$$[\![x \texttt{ = } \phi \texttt{ -> } e]\!]^{\Gamma}_{\mathrm{R}} \quad = \quad \forall \boldsymbol{\alpha}.\tau_\phi \to \alpha' \setminus \mathrm{C} <: \Gamma(x)$$

$$\text{where} \begin{cases} [\![\phi]\!]^{\Gamma}_{\mathrm{P}} = \langle \Gamma', \tau_\phi \rangle, \\ [\![e]\!]^{\Gamma \cup \Gamma'}_{\mathrm{E}} = \langle \alpha', \mathrm{C} \rangle, \\ \boldsymbol{\alpha} = (\textsc{ftv}(\tau_\phi) \cup \textsc{ftv}(\mathrm{C})) - \{\alpha'' \mid x : \alpha'' \in \Gamma\} \end{cases}$$

Initial type derivation in MicroBang is then all of the rules from Figure 4.2 amended with Definition 5.4 and interpreted using our new definition of $\dot{\alpha}_i$. The next step in defining our type system is constraint closure.

5.2 Constraint Closure

The MicroBang^0 type system defined several relations to support its constraint closure process; those relations can be used in the MicroBang type system with little modification. We first observe that, because of our new scape type, we must modify the projection matching of Definition 4.4 to match against the new scape type syntax:

Definition 5.5 (Polymorphic Projector Match)**.**

$$\forall \boldsymbol{\alpha}.\tau_\phi \to \alpha \setminus \mathrm{C} \sqsubseteq \texttt{fun} \quad \text{for any } \boldsymbol{\alpha}, \tau_\phi, \alpha, \mathrm{C}$$

We then can directly inherit from MicroBang^0 the concretization (Definition 4.1), projection (Definition 4.4), and compatibility (Definition 4.6) relations.

The empty onion variable mapping $\textsc{etv}(\alpha)$ must be updated to observe our new type variable model; we use previously unused index position 0 for this purpose.

Definition 5.6 (Empty Onion Variables)**.**

$\textsc{etv}(\langle \iota, n, \dot{c} \rangle) = \langle \iota, 0, \dot{c} \rangle$.

The application-driven form of compatibility is nearly identical but must be updated for the new form of scape type described above. We define this updated relation as follows:

Definition 5.7 (Application Compatibility)**.**

$$\begin{array}{ll} \alpha \preceq_{\mathrm{C}} [] \backslash \alpha''; \mathrm{C}'' & \text{is false} \\ \alpha \preceq_{\mathrm{C}} \vec{\tau} \,\|\, [(\forall \boldsymbol{\alpha}.\tau_\phi' \to \alpha' \setminus \mathrm{C}')] \backslash \alpha'; \mathrm{C}'' \cup \mathrm{C}' & \text{if } \alpha \preceq_{\mathrm{C}} \tau_\phi' \backslash \mathrm{C}'' \\ \alpha \preceq_{\mathrm{C}} \vec{\tau} \,\|\, [(\forall \boldsymbol{\alpha}.\tau_\phi' \to \alpha' \setminus \mathrm{C}')] \backslash \alpha''; \mathrm{C}'' & \text{if } \alpha \preceq_{\mathrm{C}} \tau_\phi' \backslash \text{❁}, \\ & \alpha \preceq_{\mathrm{C}} \vec{\tau} \backslash \alpha''; \mathrm{C}'' \end{array}$$

We write $\alpha \not\preceq_{\mathrm{C}} \overrightarrow{{}^{n} \forall \boldsymbol{\alpha}_\square.\tau_{\phi_\square} \to \alpha'_\square \setminus \mathrm{C}_\square}$ to denote that $\forall i \in \{1..n\}.\alpha \preceq_{\mathrm{C}} \tau_{\phi_i} \backslash \text{❁}$; that is, every pattern could fail to match the argument type α. Using this updated notation,

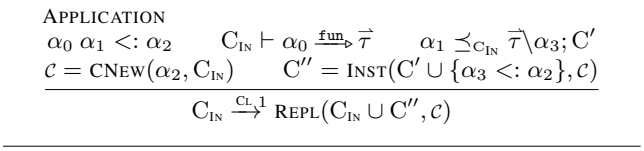

Figure 5.1. MicroBang Polymorphic Application Rule

inconsistency as in Definition 4.8 is suitable for the full MicroBang type system.

We are also nearly ready to define the full MicroBang constraint closure. But while the initial derivation step only dealt with non-contoured type variables, constraint closure interacts with type variables containing any form of $\dot{c}$. Fortunately, constraint closure only requires three contour-related functions. We describe them at a high level for now and defer formal definitions until Section 5.3. These functions are:

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

The Integer Addition and Integer Equality rules from Figure 4.3, when interpreted under these new definitions, are appropriate for use in the MicroBang type system; the Application rule needs to be modified; the revised rule appears in Figure 5.1. This rule begins much as it did in MicroBang^0: it conditions on the presence of a constraint indicating an application. It then projects all of the scapes from α_0 and checks their compatibility with the argument α_1. The rule then polyinstantiates the constraints (along with the output wiring constraint $\alpha_3 <: \alpha_2$) and unions them with the original constraint set C_{IN}. Finally, contour replacement is used to unify any cycles which were discovered by this process.

This constraint closure relation together with the initial derivation function described above provides the powerful call-site polymorphism that MicroBang requires. We formally define typechecking in Section 5.4. In order to complete the type system, however, we must first provide definitions for the three contour functions described above.

5.3 Contours

We now present the structure of contours and define the operations which closure requires of them. A contour is meant to track the polyinstantiation of a type variable. More specifically, it is meant to track the polyinstantiation contexts in which a given type variable is relevant. The type variable $\langle \iota, 1, \text{❁} \rangle$, for instance, is the first type variable inferred at redex (or pattern) ι; the type variable $\langle \iota, 1, \mathcal{C} \rangle$ is a polyinstantiation of that type variable which describes the type of the redex or pattern ι only in the context represented by $\mathcal{C}$.

We must therefore define contours such that they represent the contextual information relevant to our polymorphism model. A finite approximation of call strings is needed to avoid the unbounded computation of our operational semantics; to accomplish this, we use a restricted form of regular expressions over call strings to represent contours. We motivate our choice of restriction by example, using integers to represent identifiers and using lists of such identifiers to represent call strings (where each identifier represents a call site).

Contours by Example Using the above notation, consider the call string $[1, 2]$. This means that function call site 1 is called which then leads to a call at site 2. This call string can be represented as the simple regular expression $[1, 2]$ (we represent the concatenation of regular expressions as a list to fit with our eventual contour grammar).

When recursion happens, however, the call string will repeat a call site (e.g., $[1, 2, 3, 2]$). To guarantee termination, we restrict our regular expressions such that any identifier appears at most once. So, we will represent this call string using the contour expression $[1, \{2, 3\}]$, where a set represents the Kleene closure of its elements, i.e. "$1(2 \mid 3)^*$" in traditional regular expression notation. Note that this expression also matches the call strings $[1]$, $[1, 3, 2, 3]$, *etc*. This information loss only applies to recursive call cycles; we can still extend the call string and the expression with non-recursive calls without loss. For instance, we represent the call string $[1, 2, 3, 2, 4]$ with the expression $[1, \{2, 3\}, 4]$.

In order to keep our contour expressions manageable and prevent unnecessary complexity in constraint closure, we also do not permit their nesting. For instance, the call string $[1, 2, 3, 2, 4, 1]$ would not extend the expression above to $[1, \{2, 3\}, 4, 2]$, nor would it treat the set as an element of a larger set as in $[\{1, \{2, 3\}, 4\}]$. Instead, we simply merge all of the elements between the two identical call sites, yielding $[\{1, 2, 3, 4\}]$. While this does represent a loss of information, we believe the cases in which this information loss is relevant to the programmer are obscure.

Although the expressions here are general enough to describe each polyinstantiation, constraint closure will occasionally require two contours to be *merged*, unifying the variables which exist in each contour with the corresponding variables in the other. In order to support this with no information loss, we define contours as sets of the restricted regular expressions presented here; such a contour matches a given call string if that string is matched by any of the regular expressions in its set.

Program analyses have previously use regular expressions to abstractly represent arbitrary call strings; see *e.g.* ΔCFA [20] and citations therein. Our work differs from existing program analyses because our goals do not match those of program optimizers. Compared to [20], for example, we create contours for each call site (rather than each function) as this finer granularity adds expressiveness and limits arbitrary cutoffs. On the other hand, we are not in-

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

Figure 5.2. MicroBang Contour Grammar

terested in the precise invariants on variables in the calling context that are derivable from a ΔCFA analysis.

Contour Grammar We now formalize the grammar of MicroBang contours using the above examples as a guide. Each term of the restricted regular expressions above was either a single, literal call site (in the form of an identifier) or a set of such call sites. We use the term *contour part* $\mathcal{P}$ to refer to such a term. Each restricted regular expression was a list of these parts; we refer to these expressions as *contour strands* $\mathcal{S}$. As stated above, a contour $\mathcal{C}$ is a set of these strands. We define this grammar in Figure 5.2.

Because contours represent regular expressions over call strings, it is helpful to formalize the meaning of contours in terms of the sets of call strings they match:

Definition 5.8 (Contour Meaning)**.**

$$
\begin{array}{lrcl}
\text{(Contours)} & [\![\mathcal{C}]\!] & = & \bigcup\{[\![\mathcal{S}]\!] \mid \mathcal{S} \in \mathcal{C}\} \\
\text{(Strands)} & [\![[]]\!] & = & \{[]\} \\
 & [\![\overrightarrow{\mathcal{P}}]\!] & = & \{\vec{\iota_1} \,\|\, \ldots \,\|\, \vec{\iota_n} \mid \forall i \in \{1..n\}.\vec{\iota_i} \in [\![\mathcal{P}_i]\!]\} \\
\text{(Parts)} & [\![\overbrace{\iota}]\!] & = & \{\overrightarrow{\iota'} \mid \forall i \in \{1..n\}.\iota'_i \in \overbrace{\iota}\} \\
 & [\![\iota]\!] & = & \{[\iota]\}
\end{array}
$$

Using the examples from above, we would say that $[1,2,3,2,4] \in [\![\mathcal{C}]\!]$ where $\mathcal{C}$ is the contour containing just the strand $[1,\{2,3\},4]$.

Contour Relations In order to define the three contour functions used in closure above, 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 $(\!|\mathrm{C}|\!)$ 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 leaf cases are interesting for contours ($(\!|\mathcal{C}|\!) = \{\mathcal{C}\}$) and non-contours ($(\!|\text{✵}|\!) = \emptyset$); 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 5.9 (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.

Contour Creation The functions for creating new contours are described in Figure 5.3; we outline their behavior here.

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

Figure 5.3. MicroBang Contour Creation

The MAKE function gives a starting contour for a type variable representing a call site; the application rule in Figure 5.1 represents a call site by using the output type variable of the corresponding application constraint. This starting contour is the contour of the call site extended with the call site's identifier; in essence, the new contour represents pushing the call site onto a call stack.

The act of adding the call site to the starting contour may have introduced a cycle. To resolve this issue, the starting contour is then *collapsed*. COLLAPSE will find any cycles – that is, any sequence of contour parts which repeat a given ι, thus potentially representing call site recursion – and join them into a single Kleene closure set of ι. This enforces our invariant that the contour strands in the type system contain a given ι in at most one position.

By collapsing the contours we use in polyinstantiation, we guarantee termination; showing a worst case bound in the permutations of the call sites is trivial. Unfortunately, contour collapse does not guarantee a constraint set with disjoint contours. Closure over a constraint set in which contours overlap is quite involved and is exponentially complex in common cases. To remedy this problem, we *widen* the contour to subsume any contour it overlaps; this is as simple as unioning the strands of those contours with the strands of the new one. While the resulting contour still overlaps contours which appear in the constraint set, it can now be used to replace them. As a result, no overlapping contours exist after calculating the REPL of the constraint set.

Instantiation and Replacement The INST and REPL functions perform instantiation and replacement, respectively, for contours. We define here a three-argument version of contour instantiation, $\textsc{Inst}(*, \boldsymbol{\alpha}, \mathcal{C})$, where the set represents the bound variables which should not be instantiated; this is used when instantiation enters a scape type. The two-argument version of contour instantiation is defined in terms of this three-argument form: $\textsc{Inst}(\mathrm{C}, \mathcal{C}) = \textsc{Inst}(\mathrm{C}, \emptyset, \mathcal{C})$.

The INST and REPL functions are quite similar in structure; they are largely the natural homomorphisms over the type system constructs. For instance, $\textsc{Inst}(\tau <: \alpha, \boldsymbol{\alpha}, \mathcal{C}) =$

$\textsc{Inst}(\tau, \boldsymbol{\alpha}, \mathcal{C}) <: \textsc{Inst}(\alpha, \boldsymbol{\alpha}, \mathcal{C})$. The cases for which they are not naturally homomorphic appear below:

Definition 5.10 (Contour Function Cases)**.**

$$\textsc{Inst}(\forall \boldsymbol{\alpha}_1.\tau_\phi \to \alpha \setminus \mathrm{C}, \boldsymbol{\alpha}_2, \mathcal{C}) = \forall \boldsymbol{\alpha}_1.\tau_\phi \to \alpha \setminus \textsc{Inst}(\mathrm{C}, \boldsymbol{\alpha}_1 \cup \boldsymbol{\alpha}_2, \mathcal{C})$$

$$\textsc{Inst}(\langle \iota, n, \dot{\mathcal{C}} \rangle, \boldsymbol{\alpha}, \mathcal{C}) = \begin{cases} \langle \iota, n, \mathcal{C} \rangle & \text{if } \dot{\mathcal{C}} = \text{❁} \wedge \langle \iota, n, \dot{\mathcal{C}} \rangle \notin \boldsymbol{\alpha} \\ \langle \iota, n, \dot{\mathcal{C}} \rangle & \text{otherwise} \end{cases}$$

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

The Inst function is simply a capture-avoiding type variable substitution on the first argument. This allows the application rule to instantiate the set of constraints in a scape (which were all inferred with no contour) by giving them the contour provided by CNew.

The Repl function performs contour substitution using a given contour. The contour argument replaces contours appearing in the constraint set when they are subsumed by that contour argument; as a result, any two type variables which are both subsumed by this contour are now unified. This is the mechanism by which contours are merged to bound polyinstantiation.

The above definitions provide the three contour functions which were used in Section 5.2; this completes the definition of constraint closure.

5.4 Typechecking and Soundness

The relations above are sufficient to define a complete polymorphic type system for MicroBang. We formally define typechecking as follows:

Definition 5.11 (Typechecks)**.** A closed expression e *typechecks* iff $[\![e]\!]^{\emptyset}_{\mathrm{E}} = \langle \alpha, \mathrm{C} \rangle$, $\mathrm{C}' = \textsc{Inst}(\mathrm{C}, \{[]\})$, and for any C'', $\mathrm{C}' \xrightarrow{\mathrm{CL}}{}^{*} \mathrm{C}''$ implies C'' is consistent.

This definition of typechecking is similar to the description given in Section 4, but includes an additional step. Immediately after derivation, the resulting set of constraints is instantiated with the initial contour $\{[]\}$, which represents a global, top-level calling context. After this instantiation, we simply assert that no valid closure of C' contains inconsistencies.

Typechecking is both sound and decidable.

Theorem 5.1 (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 5.2 (Decidability)**.** Given a closed e, it is decidable whether e typechecks.

A full proof of this theorem appears in Appendix B.

This completes the MicroBang presentation. In the following two sections, we will extend MicroBang to include the features of TinyBang not present in MicroBang.

6. Deep and Wide Patterns

We begin extending the MicroBang type system toward TinyBang by adding deep and wide patterns. Deep patterns are those which can specify structure beyond a single label boundary; for instance, the TinyBang scape `‘A ‘B x -> x` is a deep pattern. Wide patterns are those which can match their argument in multiple ways. The only form of wide pattern in TinyBang is the conjunctive pattern such as `‘A x & ‘B y`. Conjunctive patterns only match an argument if all of their subpatterns match it. Here we extend MicroBang to MicroBang$^{\phi}$, the grammar of which is the same as Figure 3.1 except for its pattern grammar:

Definition 6.1 (MicroBang$^{\phi}$ Pattern Grammar)**.**

$$\begin{array}{rcll} \phi & ::= & x : \varphi & \textit{patterns} \\ \varphi & ::= & \texttt{int} \mid \mathrm{lbl}\, x : \varphi \mid \varphi \,\&\, \varphi \mid \texttt{any} & \textit{primary patterns} \end{array}$$

This grammar supports both the wide and the deep patterns of TinyBang. Because pattern conjunction is associative, we will write $\varphi_1 \,\&\, \ldots \,\&\, \varphi_n$ to denote $\varphi_1 \,\&\, (\ldots (\varphi_{n-1} \,\&\, \varphi_n) \ldots)$.

This section shows how the polymorphic type system for MicroBang can be extended to support both wide and deep patterns. We also provide a solution to the union alignment problem we mention in Section 4.2, as it is more pronounced when wide patterns are involved. For sake of brevity, we do not formally extend the operational semantics of MicroBang.

6.1 Derivation

In order to support the derivation step of type checking, we must extend derivation to cover our new pattern grammar. To do so, we must first extend the type grammar to provide types for the new patterns. The pattern types for MicroBang$^{\phi}$ appear below.

Definition 6.2 (MicroBang$^{\phi}$ Pattern Type Grammar)**.**

$$\begin{array}{rcll} \tau_\phi & ::= & \alpha \sim \tau_\varphi & \textit{pattern types} \\ \tau_\varphi & ::= & \texttt{int} \mid \mathrm{lbl}\, \alpha \sim \tau_\varphi \mid \tau_\varphi \,\&\, \tau_\varphi \mid \texttt{any} & \textit{primary pattern types} \end{array}$$

Overall, the type grammar for MicroBang$^{\phi}$ includes the changes from Definition 6.2 as well as the changes from the polymorphic model in Definitions 5.1 and 5.3. Otherwise, it has the same type grammar as MicroBang^{0} from Figure 4.1.

Using this type grammar, we can define initial type derivation. As with the type grammar, initial type derivation in MicroBang$^{\phi}$ is quite similar to the previous systems; it only changes for the new pattern grammar, the rules for which appear in Figure 6.1. The expression and redex rules of MicroBang$^{\phi}$ are the same as those shown in Figure 4.2 with the polymorphism adjustments from Definition 5.4.

6.2 Fibrations for Union Alignment

In order to support constraint closure, we must redefine compatibility in terms of these new patterns. However, conjunctive patterns present difficulty; the union alignment problem that we previously overlooked is considerably worse. For instance, consider `(‘A x & ‘B y -> x + y) z`

$$
\begin{aligned}
[\![x : \varphi]\!]^{\Gamma}_{\mathrm{P}} &= \langle \{x : \mathring{\alpha}_1\} \cup \Gamma', \mathring{\alpha}_1 \sim \tau_\varphi \rangle \\
&\quad \text{where } [\![\varphi]\!]^{\Gamma}_{\mathrm{P}} = \langle \Gamma', \tau_\varphi \rangle \\
[\![\mathtt{int}]\!]^{\Gamma}_{\mathrm{P}} &= \langle \emptyset, \mathtt{int} \rangle \\
[\![\mathrm{lbl}\, x : \varphi]\!]^{\Gamma}_{\mathrm{P}} &= \langle \{x : \mathring{\alpha}_1\} \cup \Gamma', \mathring{\alpha}_1 \sim \tau_\varphi \rangle \\
&\quad \text{where } [\![\varphi]\!]^{\Gamma}_{\mathrm{P}} = \langle \Gamma', \tau_\varphi \rangle \\
[\![\varphi_1 \,\&\, \varphi_2]\!]^{\Gamma}_{\mathrm{P}} &= \langle \Gamma_1 \cup \Gamma_2, \tau_{\varphi_1} \sim \tau_{\varphi_2} \rangle \\
&\quad \text{where } [\![\varphi_1]\!]^{\Gamma}_{\mathrm{P}} = \langle \Gamma_1, \tau_{\varphi_1} \rangle \text{ and } [\![\varphi_2]\!]^{\Gamma}_{\mathrm{P}} = \langle \Gamma_2, \tau_{\varphi_2} \rangle \\
[\![\mathtt{any}]\!]^{\Gamma}_{\mathrm{P}} &= \langle \emptyset, \mathtt{any} \rangle
\end{aligned}
$$

Figure 6.1. MicroBang$^{\phi}$ Initial Pattern Type Derivation

$$
\begin{array}{ll}
\mathrm{C} \vdash \alpha_1 \xrightarrow{\mathtt{int}\triangleright} [\mathtt{int}]; \langle \mathtt{int}, [] \rangle & \text{if } \mathtt{int} \overset{\mathrm{C}}{<:^*} \alpha_1 \\
\mathrm{C} \vdash \alpha_1 \xrightarrow{\mathrm{lbl}\triangleright} [\mathrm{lbl}\, \alpha_2]; \langle \mathrm{lbl}\, \alpha_2, [\mathfrak{f}] \rangle & \text{if } \mathrm{lbl}\, \alpha_2 \overset{\mathrm{C}}{<:^*} \alpha_1 \\
\mathrm{C} \vdash \alpha_1 \xrightarrow{\mathtt{fun}\triangleright} [\tau]; \langle \tau, [\mathfrak{f}] \rangle & \text{if } \tau \overset{\mathrm{C}}{<:^*} \alpha_1, \\
 & \quad \tau = \forall \boldsymbol{\alpha}. \tau_\phi \to \alpha_2 \setminus \mathrm{C}' \\
\mathrm{C} \vdash \alpha_1 \xrightarrow{\pi\triangleright} []; \langle \mathtt{int}, [] \rangle & \text{if } \pi \neq \mathtt{int}, \mathtt{int} \overset{\mathrm{C}}{<:^*} \alpha_1 \\
\mathrm{C} \vdash \alpha_1 \xrightarrow{\pi\triangleright} []; \langle \mathrm{lbl}\, \alpha_2, [\mathfrak{f}] \rangle & \text{if } \pi \neq \mathrm{lbl}, \mathrm{lbl}\, \alpha_2 \overset{\mathrm{C}}{<:^*} \alpha_1 \\
\mathrm{C} \vdash \alpha_1 \xrightarrow{\pi\triangleright} []; \langle \tau, [\mathfrak{f}] \rangle & \text{if } \pi \neq \mathtt{fun}, \tau \overset{\mathrm{C}}{<:^*} \alpha_1, \\
 & \quad \tau = \forall \boldsymbol{\alpha}. \tau_\phi \to \alpha_2 \setminus \mathrm{C}' \\
\mathrm{C} \vdash \alpha_1 \xrightarrow{\pi\triangleright} \vec{\tau}_1 \parallel \vec{\tau}_2; \langle \alpha_1' \,\&\, \alpha_2', [\mathfrak{f}_1, \mathfrak{f}_2] \rangle & \\
 & \quad \text{if } \alpha_1' \,\&\, \alpha_2' \overset{\mathrm{C}}{<:^*} \alpha_1, \\
 & \quad \mathrm{C} \vdash \alpha_1' \xrightarrow{\pi\triangleright} \vec{\tau}_1; \mathfrak{f}_1, \mathrm{C} \vdash \alpha_2' \xrightarrow{\pi\triangleright} \vec{\tau}_2; \mathfrak{f}_2
\end{array}
$$

Figure 6.2. MicroBang$^{\phi}$ Projection

where argument `z` has type α and the constraint set contains $\{\texttt{`A int} <: \alpha, \texttt{`B int} <: \alpha\}$. It can be shown that both `'A` and `'B` can be projected from α individually, but we do not know that a single value will contain both of them.

The root of this problem is that the type variable α represents a union type and we are not ensuring that we have a consistent view of it across multiple projections. This so-called *union alignment problem* is well-known and invariably arises in union type systems; [11], for example, is a recent paper reviewing the problem and presenting a sound union elimination type rule. 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, the union eliminations can be viewed as occurring in canonical positions: each closure rule propagates per concrete lower bound, and thus can be viewed as performing an implicit union elimination by choosing a particular lower bound to propagate. For example, the α above is processed pointwise first as `'A int`, and then as `'B int`, fixing the choice each time. In order to achieve pointwise processing, any union decisions made for a given expression are marked and we constrain to make the *same* choices every time when considering a single function application; this achieves alignment. To record the particular union elimination decision(s) made, we introduce type *fibrations*, 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:

Definition 6.3 (Fibration Grammar)**.**

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

Each fibration $\mathfrak{f}$ is used to describe decisions about a specific type variable at a specific point in the type. If the fibration is of the form ❋, it indicates that no decisions have been made about the type variable: it is unconstrained in context. If the fibration is of the form $\langle \tau, \vec{\mathfrak{f}} \rangle$, then a specific lower bound has already been chosen for that type variable at that location in the type. The associated list of fibrations describe the decisions for type variables in τ; it is a singleton list if τ is a label, for instance, and an empty list if τ is `int`.

Note that this deep structure is necessary: it is not sufficient to build a mapping between type variables and lower bounds. For example, consider the type α under $\{\texttt{int} <: \alpha, \texttt{`A}\ \alpha <: \alpha, \texttt{`B}\ \alpha <: \alpha\}$; each time the type recurses, the variable α may be given any of three lower bounds. When checking compatibility with the pattern `'A 'B _`, this type would use the fibration $\langle \texttt{`A}\ \alpha, [\langle \texttt{`B}\ \alpha, [\text{❋}] \rangle] \rangle$. This fibration indicates that we chose `'A` α for the first occurrence α, we chose `'B` α for the second occurrence of α, and we did not make any further decisions. A simple mapping from α to a single lower bound is unable to express this case.

6.3 Constraint Closure

Using fibrations, we can now adjust the relations used in constraint closure to integrate deep patterns into the type system described in Section 5.

Projection We begin by redefining projection to include fibrations. We now write $\mathrm{C} \vdash \alpha \xrightarrow{\pi\triangleright} \vec{\tau}; \mathfrak{f}$ to indicate projection; this new relation simply includes an additional place for the fibration. For convenience, we elide the fibration $\mathfrak{f}$ when it is unimportant. Given this definition, our new projection relation appears in Figure 6.2. This relation does not use the projection matching predicate from Definitions 4.3 and 5.5 because each case of projection must relate to a fibration based on the lower bounding type used.

The first two rules of this projection relation correspond to the first rule of MicroBang's projection relation from Definition 4.4; the next two rules correspond to the second rule of MicroBang's projection relation; and the last rule here corresponds to MicroBang projection's last rule. Note that the label fibrations above use the unconstrained fibration variable $\mathfrak{f}$; in these cases, any fibration may relate. This is relevant for deep patterns; if a label `'A` α' is successfully projected from a type variable α, for example, we might later discover that decisions must be made for α'.

$$\alpha_1 \preceq_{\mathrm{C}} \alpha_2 \sim \tau_\varphi \backslash \{\alpha_1 <: \alpha_2\} \cup \dot{\mathrm{C}}; \mathfrak{f} \quad \text{if} \quad \alpha_1 \preceq_{\mathrm{C}} \tau_\varphi \backslash \dot{\mathrm{C}}; \mathfrak{f}$$

$$\alpha_1 \preceq_{\mathrm{C}} \texttt{int} \backslash \emptyset; \mathfrak{f} \quad \text{if} \quad \mathrm{C} \vdash \alpha_1 \xrightarrow{\texttt{int}}_{\triangleright} \texttt{int}; \mathfrak{f}$$

$$\alpha_1 \preceq_{\mathrm{C}} \mathrm{lbl}\, \alpha_2 \sim \tau_\varphi \backslash \{\alpha_3 <: \alpha_2\} \cup \dot{\mathrm{C}}; \mathfrak{f}' \quad \text{if} \quad \alpha_3 \preceq_{\mathrm{C}} \tau_\varphi \backslash \dot{\mathrm{C}}; \mathfrak{f},\ \mathfrak{f}' = \langle \mathrm{lbl}\, \alpha_3, [\mathfrak{f}] \rangle, \mathrm{C} \vdash \alpha_1 \xrightarrow{\mathrm{lbl}}_{\triangleright} \mathrm{lbl}\, \alpha_3; \mathfrak{f}'$$

$$\alpha_1 \preceq_{\mathrm{C}} \tau_{\varphi_1} \,\&\, \tau_{\varphi_2} \backslash \dot{\mathrm{C}}_1 \cup \dot{\mathrm{C}}_2; \mathfrak{f} \quad \text{if} \quad \alpha_1 \preceq_{\mathrm{C}} \tau_{\varphi_1} \backslash \dot{\mathrm{C}}_1; \mathfrak{f},\ \alpha_1 \preceq_{\mathrm{C}} \tau_{\varphi_2} \backslash \dot{\mathrm{C}}_2; \mathfrak{f}$$

$$\alpha_1 \preceq_{\mathrm{C}} \texttt{any} \backslash \emptyset; \mathfrak{f} \quad \text{always}$$

$$\alpha_1 \preceq_{\mathrm{C}} \texttt{int} \backslash \text{❁}; \mathfrak{f} \quad \text{if} \quad \mathrm{C} \vdash \alpha_1 \xrightarrow{\texttt{int}}_{\triangleright} \text{❁}; \mathfrak{f}$$

$$\alpha_1 \preceq_{\mathrm{C}} \mathrm{lbl}\, \alpha_2 \backslash \text{❁}; \mathfrak{f} \quad \text{if} \quad \mathrm{C} \vdash \alpha_1 \xrightarrow{\mathrm{lbl}}_{\triangleright} \text{❁}; \mathfrak{f}$$

Figure 6.3. MicroBang$^\phi$ Compatibility

As with MicroBang, we often find it useful to discuss only the highest priority type which is projected from an onion. We also need to discuss the case in which no types can be projected from the onion with a given projector. We define notation for both cases below.

Definition 6.4 (MicroBang$^\phi$ Single Projection)**.** $\mathrm{C} \vdash \alpha \xrightarrow{\pi}_{\triangleright} \tau; \mathfrak{f}$ holds when $\mathrm{C} \vdash \alpha \xrightarrow{\pi}_{\triangleright} \vec{\tau} \,\|\, [\tau]; \mathfrak{f}$ for some $\vec{\tau}$. Further, $\mathrm{C} \vdash \alpha \xrightarrow{\pi}_{\triangleright} \text{❁}; \mathfrak{f}$ holds when $\mathrm{C} \vdash \alpha \xrightarrow{\pi}_{\triangleright} []; \mathfrak{f}$.

Compatibility We now define compatibility in MicroBang$^\phi$. As with projection, we simply extend the previous compatibility relation, writing $\alpha \preceq_{\mathrm{C}} \tau\phi \backslash \dot{\mathrm{C}}; \mathfrak{f}$. Compatibility is then defined in Figure 6.3.

The structure of this compatibility rule is slightly different from the one presented in Definition 4.6, but the intention is the same: each rule relates a type variable to a pattern either by producing a set of constraints (in the case that a type represented by the variable matches the pattern) or a ❁ (in the case that a type represented by the variable does not match the pattern). Because type variables may represent unions, it is possible for the same type variable and pattern to relate in multiple ways. The second rule in Figure 6.3, for instance, indicates that integers match the integer pattern under no constraints. The third rule indicates that labels match the pattern containing their label name by allowing the variable from the label type to flow to the variable in the label pattern. The last rule indicates that a label pattern does not match a variable if a label cannot be projected from that variable.

As with projection, we have redefined compatibility to use fibrations. In the case of compatibility with a conjunctive pattern (e.g., `'A x & 'B y -> ...`), note that each compatibility relation against one of the subpatterns uses the same fibration as the other; this ensures union alignment for wide patterns. Likewise, the rule for label pattern matching enforces alignment for deep patterns; the fibration used when projecting the label type must immediately contain the fibration used when recursing into the label pattern's subpattern.

A further advantage of adding fibrations to compatibility is that it allows us to force union alignment across multiple compatibility relations. We write the application-driven compatibility relation of MicroBang$^\phi$ as $\alpha \preceq_{\mathrm{C}} \vec{\tau} \backslash \alpha'; \mathrm{C}; \mathfrak{f}$ to indicate that α, fibrated as $\mathfrak{f}$, is compatible with some scape in $\vec{\tau}$. The $\boldsymbol{\alpha}$, α', and C are the captured variables, return type, and constraints of the matched scape. We define this relation as follows:

Definition 6.5 (Application Compatibility)**.**

$$\alpha \preceq_{\mathrm{C}} []\backslash\alpha''; \mathrm{C}''; \mathfrak{f} \quad \text{is false}$$

$$\alpha \preceq_{\mathrm{C}} \vec{\tau} \,\|\, [(\forall \boldsymbol{\alpha}.\tau\phi' \to \alpha' \setminus \mathrm{C}')] \backslash \alpha'; \mathrm{C}'' \cup \mathrm{C}'; \mathfrak{f} \quad \text{if } \alpha \preceq_{\mathrm{C}} \tau\phi' \backslash \mathrm{C}''; \mathfrak{f}$$

$$\alpha \preceq_{\mathrm{C}} \vec{\tau} \,\|\, [(\forall \boldsymbol{\alpha}.\tau\phi' \to \alpha' \setminus \mathrm{C}')] \backslash \alpha''; \mathrm{C}''; \mathfrak{f} \quad \text{if } \alpha \preceq_{\mathrm{C}} \tau\phi' \backslash \text{❁}; \mathfrak{f},\ \alpha \preceq_{\mathrm{C}} \vec{\tau} \backslash \alpha''; \mathrm{C}''; \mathfrak{f}$$

Observe that the same fibration is used for all pattern compatibility checks; this forces application compatibility to use a consistent representation of the argument type α. For notational convenience, we omit the fibration when it is not important which fibration is used.

Finally, we take $\alpha \not\preceq_{\mathrm{C}} \overrightarrow{{}^{n}\forall \boldsymbol{\alpha}.\tau\phi \to \alpha' \setminus \mathrm{C}}$ to mean that $\exists \mathfrak{f}.\, \forall i \in \{1..n\}.\, \alpha \preceq_{\mathrm{C}} \tau\phi_i \backslash \text{❁}; \mathfrak{f}$. Thus, a pattern match failure is only valid if it occurs for the same fibration at every pattern in the list of scapes.

Upgrade Complete! Using fibrations and the notational updates that appear above, the rest of the MicroBang$^\phi$ type system follows in the same fashion as the MicroBang type system. In fact, the definitions of closure, inconsistency, and typechecking are syntactically identical; we merely need to interpret Figure 5.1, Definition 4.8, and Definition 5.11 (respectively) in the context of the definitions presented in this section. We do not provide proofs of soundness and termination for this system, but there is little added complexity beyond fibration bookkeeping.

7. Bridging the Gap

While the previous sections have demonstrated how the most challenging features of TinyBang can be statically typed, there are still several language features which appear in TinyBang but not MicroBang$^\phi$. While we do not have the space to formalize the typing of each of these features, this section shows several features of interest and discusses them in varying degrees of formality. In each case, we extend MicroBang (and not MicroBang$^\phi$). There are no technical difficulties in extending MicroBang$^\phi$ with each of these features; we extend the simpler MicroBang type system for brevity.

7.1 State

The TinyBang language presented in Section 2 includes state: the construction of a label implies the creation of a cell, and variables bound to the contents of labels may be mutated. We now show how the operational semantics and type system for MicroBang can be amended to include these state semantics.

Syntax We begin by making changes to the MicroBang language grammar to accommodate the state semantics. Rather than treating an expression as a list of redex assignments, we treat an expression as a list of clauses, some of which are traditional redex assignments and some of which provide grammar for state operations. The resulting grammar changes appear below:

Definition 7.1 (Stateful MicroBang Grammar)**.**

$$
\begin{array}{rcll}
y & & & \textit{cell variables} \\
e & ::= & \overrightarrow{s} & \textit{expressions} \\
s & ::= & x = r \mid y := x \mid y \texttt{<-} x \mid x \texttt{<-} y \mid y = y & \textit{clauses} \\
v & ::= & x \mid \mathbb{Z} \mid \text{lbl}\, y \mid x \,\texttt{\&}\, x \mid \texttt{()} \mid \phi \texttt{->} e & \textit{values} \\
\varphi & ::= & \texttt{int} \mid \text{lbl}\, y \mid \texttt{any} & \textit{primary patterns}
\end{array}
$$

The full grammar of this stateful MicroBang is the one appearing in Definition 7.1 along with the original rules for r, ϕ, x, op, lbl, and π from Figure 3.1. Although this grammar admits any form of clause at the end of an expression e, we only consider non-empty e in which the last clause is of the form $x = r$. The cell variables y in the new grammar are the only variables which actually appear in the TinyBang syntax; the x variables are only necessary for intermediate names induced by our ANF grammar. The stateful TinyBang syntax translates to this ANF grammar by using $y := x, \ldots$ to create cells in place of `def` y `=` x `in` $\ldots$ and by using $y \texttt{<-} x, \ldots$ to update cells in place of y `=` x `in` $\ldots$. Further, all uses of a cell must be captured in an ANF variable; the TinyBang y_1 `+` y_2, for instance, is translated as $[x_1 \texttt{<-} y_1, x_2 \texttt{<-} y_2, x_3 = x_1 + x_2]$.

Semantics The new grammar clauses provide state semantics as described above. We also observe that although the value and pattern grammars are nearly identical to MicroBang, labels now contain cell variables instead of normal variables. As a result, label contents are always mutable.

Rather than interpreting E as a list of the form $\overrightarrow{x = v}$ as we did before, we now interpret it as a list of clauses where each clause is of the form $x = v$, $y := x$, or $y = y$. Note that E is thus both the environment and the (cyclic) store. Lookup on a cell variable $E(y)$ is now defined inductively; it is $E(x)$ when $y := x \in E$ and $E(y')$ when $y = y' \in E$.

The projection function $E{\downarrow}^*_\pi(x)$ from Definition 3.1 can be used verbatim in this system, but we must redefine the label clause of compatibility in terms of the above syntax. This new definition is as follows:

Definition 7.2 (Stateful Value Compatibility)**.**

$$x_1 \preceq_E \text{lbl}\, y_3 \backslash [y_3 = y_2] \quad \text{if} \quad E{\downarrow}_{\text{lbl}}(x_1) = \text{lbl}\, y_2$$

The definition of compatibility in stateful MicroBang is the one appearing in Definition 3.3 with the new clause from Definition 7.2.

Application Compatibility in Definition 3.4 can also be used verbatim in this system if interpreted under the new definition of compatibility. We can now define the new small step rules for stateful MicroBang:

Definition 7.3 (Stateful Small-Step Semantics)**.**

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

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

$$E_1 \,\|[y := x_1]\| \, E_2 \,\|[y \texttt{<-} x_2]\| \, e \longrightarrow^1 E_1 \,\|[y := x_2]\| \, E_2 \,\|\, e$$

$$E_1 \,\|[y := x_1]\| \, E_2 \,\|[x_2 \texttt{<-} y]\| \, e \longrightarrow^1 E_1 \,\|[y := x_1]\| \, E_2 \,\|[x_2 = x_1]\| \, e$$

$$
\begin{aligned}
[\![s]\!]^\Gamma_{\text{E}} &= \langle \dot\alpha_1, \{[\![s]\!]^{\Gamma\cup\Gamma'}_{\text{R}}\}\rangle \\
&\quad \text{where } \Gamma' = \{\textsc{Var}(s) : \dot\alpha_1\} \\
[\![[s] \,\|\, e]\!]^\Gamma_{\text{E}} &= \langle \alpha', \{[\![s]\!]^{\Gamma\cup\Gamma'}_{\text{R}}\} \cup \text{C}\rangle \\
&\quad \text{where } \Gamma' = \{\textsc{Var}(s) : \dot\alpha_1\}, [\![e]\!]^{\Gamma\cup\Gamma'}_{\text{E}} = \langle \alpha', \text{C}\rangle
\end{aligned}
$$

$$
\begin{aligned}
[\![x = \text{lbl}\, y]\!]^\Gamma_{\text{R}} &= \text{lbl}\, \Gamma(y) <: \Gamma(x) \\
[\![y := x]\!]^\Gamma_{\text{R}} &= \text{cell}(\Gamma(x)) <: \Gamma(y) \\
[\![y \texttt{<-} x]\!]^\Gamma_{\text{R}} &= \Gamma(y) <: \text{cellS}(\Gamma(x)) \\
[\![x \texttt{<-} y]\!]^\Gamma_{\text{R}} &= \Gamma(y) <: \text{cellG}(\Gamma(x)) \\
[\![y = y']\!]^\Gamma_{\text{R}} &= \text{cell}(\Gamma(y)) <: \text{cell}(\Gamma(y'))
\end{aligned}
$$

$$[\![x : \text{lbl}\, y]\!]^\Gamma_{\text{P}} = \langle \{x : \dot\alpha_1, y : \dot\alpha_2\}, \dot\alpha_1 \sim \text{lbl}\, \dot\alpha_2\rangle$$

Figure 7.1. Stateful Initial Derivation Rules

The small step semantics for stateful MicroBang are then defined by the application and addition rules from Definition 3.5 and all of the rules in Definition 7.3.

Initial Derivation The semantics above create a model of state in which cells cannot directly contain other cells; labels must be used to nest them. To model this directly in the type system, we treat cells as a new form of constraint rather than a type constructor. The cell lower-bounding constraint carries a type variable representing the type of its contents; the upper-bounding constraints carry the variables which are used to read from or write to the cell. This allows us to incrementally build the type of the cell by adding constraints to the type carried by the cell lower bound, which in turn supports our model of flow-insensitive cell types. We specify the additions to the constraint grammar below:

Definition 7.4 (State Constraint Grammar)**.**

$$
\begin{array}{rcll}
\text{c} & ::= & \ldots \mid \text{cell}(\alpha) <: \alpha \mid \text{cell}(\alpha) <: \text{cell}(\alpha) & \\
& & \mid \alpha <: \text{cellG}(\alpha) \mid \alpha <: \text{cellS}(\alpha) & \textit{constraints}
\end{array}
$$

The lower-bounding cell constraint is used to represent the creation of a cell; the two upper-bounding constraints represent, respectively, getting a value from and setting a value into a cell. We make use of these constraints by modifying the initial type derivation rules. The modified derivation rules (as well as the rule for assignment) appear in Figure 7.1. For these rules, we define $\textsc{Var}(s)$ to be the variable on the left side of a given s, which is either an x or y; we also allow bindings for both x and y to appear in Γ. Finally, we extend the function $[\![\cdot]\!]^\Gamma_{\text{R}}$ to operate on all clauses and we assume that all clauses are associated with a unique identifier ι (not just redices).

The initial type derivation for stateful MicroBang is defined by Figure 4.2 incorporating the changes from Definition 5.4 and Figure 7.1.

Constraint Closure Our next step in defining state is to define the operations in constraint closure. The projection, compatibility, and application compatibility definitions in Section 5 can be directly inherited for use in this type system.

INTEGER EQUALITY

$$\frac{\alpha_1 \texttt{ == } \alpha_2 <: \alpha_3 \qquad \alpha_3' = \text{ETV}(\alpha_3) \qquad \alpha_3'' = \text{CTV}(\alpha_3) \qquad \text{C}_{\text{IN}} \vdash \alpha_1 \xrightarrow{\texttt{int}}\triangleright \texttt{ int} \qquad \text{C}_{\text{IN}} \vdash \alpha_2 \xrightarrow{\texttt{int}}\triangleright \texttt{ int}}{\{\texttt{`True } \alpha_3'' <: \alpha_3, \texttt{`False } \alpha_3'' <: \alpha_3, \text{cell}(\alpha_3') <: \alpha_3'', \texttt{() } <: \alpha_3'\}}$$

FORWARD PROPAGATION

$$\frac{\text{cell}(\alpha_1) <: \alpha_2 \qquad \alpha_2 <: \text{cellG}(\alpha_3)}{\alpha_1 <: \alpha_3}$$

BACKWARD PROPAGATION

$$\frac{\text{cell}(\alpha_1) <: \alpha_2 \qquad \alpha_2 <: \text{cellS}(\alpha_3)}{\text{cell}(\alpha_3) <: \alpha_2}$$

ALIAS PROPAGATION

$$\frac{\text{cell}(\alpha_1) <: \text{cell}(\alpha_2) \qquad \text{cell}(\alpha_3) <: \alpha_4 \qquad \alpha_3 \in \{\alpha_1, \alpha_2\}}{\{\text{cell}(\alpha_1) <: \alpha_4, \text{cell}(\alpha_2) <: \alpha_4\}}$$

Figure 7.2. New Stateful Closure Rules

To define the constraint closure rules themselves, we first must define $\text{CTV}(\cdot)$, a type variable function similar to $\text{ETV}(\cdot)$, to model the new fresh variable for cells used in the operational semantics. This function uses -1, another as-of-yet unused index:

Definition 7.5 (Cell Variables)**.** $\text{CTV}(\langle \iota, n, \dot{c} \rangle) = \langle \iota, -1, \dot{c} \rangle$.

Using the above, we define our new constraint closure rules in Figure 7.2. These rules assume that concretization is extended to cell lower bounds. The Integer Equality rule is redefined to ensure that it properly creates cells for labels. We also add two rules for cell type propagation. The remaining closure rules can be inherited from Figures 4.3 and 5.1.

The propagation rules in this figure propagate types through cells in a flow-insensitive manner: the static type of a cell is the union of all of the types which may be assigned to it. Forward propagation ensures that accessing a cell variable produces the type of its contents. Backward propagation ensures that any assignment to a cell causes it to be treated as if it may have had that type all along. Note that this model of state solves the polymorphic reference problem without a value restriction or equivalent. Inconsistency and typechecking definitions are directly inherited from Definitions 4.8 and 5.11, respectively.

7.2 Onion Filtering

The TinyBang language includes onion subtraction (`&-`) and onion projection (`&.`), two features which functionally filter an onion structure. This filtering allows onions to be taken apart and recombined. We now introduce these operations as type constructors in the MicroBang language. We will use the type grammar from Figure 4.1 with changes from Definitions 5.1 and 5.3 as well as from Definition 7.6 below:

Definition 7.6 (Onion Filtering Types)**.**

$$\tau \ ::= \ \ldots \mid \alpha \texttt{ \&- } \pi \mid \alpha \texttt{ \&. } \pi \quad \textit{types}$$

Filtering is shallow, so it is described using a projector. The expressions $e \texttt{ \&. } \pi$ and $e \texttt{ \&- } \pi$ produce values that contain only those components which match (resp. do not match) the projector. Initial derivation over these expressions is as expected:

Definition 7.7 (Onion Filtering Initial Derivation)**.**

$$[\![x = x' \texttt{ \&. } \pi]\!]_{\text{R}}^{\Gamma} = \Gamma(x') \texttt{ \&. } \pi <: \Gamma(x)$$
$$[\![x = x' \texttt{ \&- } \pi]\!]_{\text{R}}^{\Gamma} = \Gamma(x') \texttt{ \&- } \pi <: \Gamma(x)$$

In the type system, we view `&.` and `&-` as onion constructors similar to $\alpha_1 \,\&\, \alpha_2$; as a result, they should not be included where a non-onion type is expected to appear. To ensure this, we add the following rules to our definition of projection from Definition 4.4.

Definition 7.8 (Onion Filtering Projection)**.**

$$\begin{array}{lll}
\text{C} \vdash \alpha_1 \xrightarrow{\pi}\triangleright \vec{\tau} & \text{if} & \alpha_2 \texttt{ \&- } \pi' \overset{\text{C}}{<:}{}^{*} \alpha_1, \pi \neq \pi', \text{C} \vdash \alpha_2 \xrightarrow{\pi}\triangleright \vec{\tau} \\
\text{C} \vdash \alpha_1 \xrightarrow{\pi}\triangleright [] & \text{if} & \alpha_2 \texttt{ \&- } \pi' \overset{\text{C}}{<:}{}^{*} \alpha_1, \pi = \pi' \\
\text{C} \vdash \alpha_1 \xrightarrow{\pi}\triangleright \vec{\tau} & \text{if} & \alpha_2 \texttt{ \&. } \pi' \overset{\text{C}}{<:}{}^{*} \alpha_1, \pi = \pi', \text{C} \vdash \alpha_2 \xrightarrow{\pi}\triangleright \vec{\tau} \\
\text{C} \vdash \alpha_1 \xrightarrow{\pi}\triangleright [] & \text{if} & \alpha_2 \texttt{ \&. } \pi' \overset{\text{C}}{<:}{}^{*} \alpha_1, \pi \neq \pi'
\end{array}$$

The first two rules deal with onion subtraction. The first rule requires that the projectors are not equal; this means that, in this case, the subtraction should not affect the result of projection. In such a case, projection continues into the type variable of the onion subtraction type. The second rule requires that the projector being used is equal to the projector being subtracted; in this case, we know that any type projected from the variable in the onion subtraction type would be subtracted here and so we simply return an empty list. The third and fourth rule are of the same form but with the projector conditions reversed.

In this way, we use projection to hide the structure introduced by the filtering types. Every other step in the system uses projection rather than destructing types directly, so we can interpret all of the remaining definitions in Section 5 in terms of this new projection relation in order to provide the onion filtering features.

7.3 Symmetric Onions

While we have shown that the asymmetric properties of onions concisely describe overriding behavior, it is sometimes useful to guarantee an override has not occurred. In Section 2.3, we highlight TinyBang's symmetric onioning operator: `&!`. This operator creates an onion symmetrically, using the constraint-based nature of our type system to raise a type error if an override has occurred.

This symmetry requirement is represented by a new form of constraint – $\alpha_1 \bowtie \alpha_2$ – which indicates that two variables do not share a common component. This constraint is introduced whenever a symmetric onion is used; that is, $[\![x = x_1 \texttt{ \&! } x_2]\!]_{\text{R}}^{\Gamma} = \{\Gamma(x_1) \bowtie \Gamma(x_2), \Gamma(x_1) \,\&\, \Gamma(x_2) <: \Gamma(x)\}$. We then complete type derivation and perform constraint closure as before. Implementing the symmetry guarantee is as simple as ensuring that no single projector succeeds in projecting a type on both sides of the symmetric onion. To accomplish this, we use the following definition of constraint set inconsistency:

Definition 7.9 (Symmetric Onion Inconsistency)**.** C is inconsistent iff it is inconsistent by Definition 4.8 or the following is true:

$\exists \alpha_1 \not\bowtie \alpha_2 \in \mathrm{C}$ such that $\exists \pi.\ \mathrm{C} \vdash \alpha_1 \xrightarrow{\pi}_{\triangleright} \tau_1 \wedge \mathrm{C} \vdash \alpha_2 \xrightarrow{\pi}_{\triangleright} \tau_2$

8. Conclusions

We have presented TinyBang, a core language in which a broad class of flexible object-oriented operations may be easily encoded and statically typed. Encodable operations include first-class messages, functional object extension, mixins, default arguments, and overloading. In particular, TinyBang's object extension is more general than in previous works; an object can be extended even after it has been messaged without loss of depth subtyping or other critical properties.

No single feature of the TinyBang language is responsible for this flexibility. Instead, TinyBang's novelty lies in the *combination* of improvements to the state of the art in type system design: a more general notion of first-class cases, simpler typing for asymmetric record concatenation, a more expressive dependent typing of case/match, concise syntax via type-indexed records, and a highly flexible method for inference of parametric polymorphism.

We defined the operational semantics, a type inference algorithm, and established type soundness for MicroBang, a subset of TinyBang with the most key features. To make a useful type theory, we needed to address how to maximally polyinstantiate functions yet preserve termination; for this, we defined a notion of polyinstantiation *contour* denoted by a form of regular expression. For wide and deep patterns, we had to introduce *fibrations* to solve a subtle question of union type alignment. To prove decidability of type inference, the issue of non-contractive recursive types had to be addressed.

The TinyBang type inference algorithm and a TinyBang interpreter have been implemented in Haskell, allowing us to confirm correctness of the type inference algorithm and operational semantics on code examples. All examples in this paper typecheck and run in the current implementation. The type inference implementation still needs significant tuning for improved efficiency.

The larger picture In practice, programmers need more than just the expressiveness of TinyBang; they also need syntactic sugar, an efficient compiler, and a rich set of debugging tools. We are presently working on a realistic language design and implementation for BigBang [19]; TinyBang is the core for BigBang. BigBang should be appealing because it will preserve most of the versatile spirit of dynamically-typed scripting languages while reaping the scalability and runtime performance made possible by static types.

Acknowledgments

We thank the ECOOP 2013 reviewers; our effort to improve the technical presentation was inspired by their thorough reviews. We also thank Nathaniel (Wes) Filardo for helpful comments.

References

[1] O. Agesen. The cartesian product algorithm. In *Proceedings ECOOP'95*, volume 952 of *Lecture Notes in Computer Science*, 1995.

[2] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1985.

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

[4] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *POPL 21*, pages 163–173, 1994.

[5] E. Allen, D. Chase, J. Hallet, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification, version 1.0. Technical report, Oracle Labs, 2008.

[6] L. Bettini, V. Bono, and B. Venneri. Delegation by object composition. *Science of Computer Programming*, 76:992–1014, 2011.

[7] M. Blume, U. A. Acar, and W. Chae. Extensible programming with first-class cases. In *ICFP '06*, pages 239–250, 2006.

[8] V. Bono and K. Fisher. An imperative, first-order calculus with object extension. In *ECOOP'98*, pages 462–497. Springer Verlag, 1998.

[9] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. *Information and Computation*, 155(1-2):108–133, 1999.

[10] G. Castagna, G. Ghelli, , and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.

[11] J. Dunfield. Elaborating intersection and union types. In *Int'l Conf. Functional Programming*, 2012.

[12] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. In *MFPS*, Electronic Notes in Theoretical Computer Science. Elsevier, 1995.

[13] A. Guha, J. G. Politz, and S. Krishnamurthi. Fluid object types. Technical Report CS-11-04, Brown University, 2011.

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

[15] B. Jay and D. Kesner. First-class patterns. *J. Funct. Program.*, 19(2): 191–225, 2009.

[16] A. Kulkarni, Y. D. Liu, and S. F. Smith. Task types for pervasive atomicity. In *OOPSLA*, pages 671–690. ACM, 2010.

[17] Y. D. Liu and S. Smith. Pedigree types. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2008.

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

[19] P. H. Menon, Z. Palmer, A. Rozenshteyn, and S. Smith. Big Bang: Designing a statically-typed scripting language. In *International Workshop on Scripts to Programs (STOP)*, Beijing, China, 2012.

[20] M. Might and O. Shivers. Environment analysis via ΔCFA. In *Proceedings of the 33rd Annual ACM Symposium on the Principles of Programming Languages (POPL 2006)*, pages 127–140, Charleston, South Carolina, January 2006.

[21] T. D. Millstein and C. Chambers. Modular statically typed multimethods. In *ECOOP '99*, pages 279–303. Springer-Verlag, 1999.

[22] W. B. Mugridge, J. Hamer, and J. G. Hosking. Multi-methods in a statically-typed programming language. In *ECOOP '91*, 1991.

[23] S. Nishimura. Static typing for dynamic messages. In *POPL'98*, 1998.

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

[25] D. Rémy and J. Vouillon. Objective ML: a simple object-oriented extension of ML. In *POPL '97*, pages 40–53. ACM, 1997.

[26] J. G. Riecke and C. A. Stone. Privacy via subsumption. *Inf. Comput.*, 172(1):2–28, Feb. 2002.

[27] M. Shields and E. Meijer. Type-indexed rows. In *POPL*, pages 261–275, 2001.

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

[29] M. Wand. Complete type inference for simple objects. In *Proc. 2nd IEEE Symposium on Logic in Computer Science*, 1987.

[30] M. Wand. Corrigendum: Complete type inference for simple objects. In *LICS*. IEEE Computer Society, 1988.

[31] T. Wang and S. F. Smith. Precise constraint-based type inference for Java. In *ECOOP'01*, pages 99–117, 2001. ISBN 3-540-42206-4.

A. MicroBang Type Soundness Proof

We present here a proof for Theorem 5.1, the type soundness theorem for MicroBang. 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.

Throughout this proof, the following notational sugar is used. This notation does not add meaning to the operational semantics or type system; it merely simplifies the presentation of some of the definitions below.

Definition A.1 (Proof Notation)**.**

- We will write $\boldsymbol{x}$ in lieu of $\overbrace{x}$ to represent a set of variables.
- We will write $\boldsymbol{c}$ in lieu of $\overbrace{c}$ to represent a set of contours.

A.1 Variables

In order to define simulation, we must establish a convention for naming variables similar to the one given in Definition 5.1 for type variables. We are only concerned with programs for which small-step semantics are defined; we thus know that the left-hand side of all variable definitions in the program will be unique. We therefore define a bijection of the form $x \leftrightarrow \langle \iota, \mathbb{Z}, \dot{c} \rangle$; that is, this bijection maps each variable to a triple between an identifier, an integer, and a possible contour. This bijection is implicitly parametric in terms of the expression e that we are typechecking.

Definition A.2 (Variable Bijection)**.** Let $\boldsymbol{x} = \{x'_i \mid e = \overbrace{x'_\square = r'_\square}^{m}, 1 \le i \le m\}$. For each redex clause $x = r$ identified by ι in the program we are typechecking, let $x \leftrightarrow \langle \iota, 1, \dot{c} \rangle$ where $\dot{c} = \{[]\}$ when $x \in \boldsymbol{x}$ and $\dot{c} = \bullet$ otherwise. In a pattern $x : \varphi$ identified by ι, we let $x \leftrightarrow \langle \iota, 1, \bullet \rangle$. In a pattern $x : \mathrm{lbl}\, x'$ identified by ι, we let $x' \leftrightarrow \langle \iota, 2, \bullet \rangle$.

We leave the rest of this bijection unconstrained. We choose this bijection because it allows us to align fresh variables with the type variables that represent them. Throughout this proof, we will occasionally use a variable's triple in place of the variable itself; for instance, we may write $\{\langle \iota, 1, \bullet \rangle : \langle \iota, 1, \bullet \rangle\}$ to mean $\{x : \langle \iota, 1, \bullet \rangle\}$ such that $x \leftrightarrow \langle \iota, 1, \bullet \rangle$.

We will also need to be more precise in our definition of the α-conversion function discussed in Section 3.2. We begin by defining a function $\textsc{bv}(e)$ which produces the set of variables bound within a given expression e; this function is only defined over expressions with unique variable names. We also define this function over patterns and scape values for convenience.

Definition A.3 (Bound Variables)**.** We define $\textsc{bv}(\overbrace{x_\square = r_\square}^{n}) = \overbrace{x_\square}^{n}$. We let $\textsc{bv}(x : \mathrm{lbl}\, x') = \{x, x'\}$ and we let $\textsc{bv}(x : \varphi) = \{x\}$ in all other cases of φ. We let $\textsc{bv}(\phi \texttt{->} e) = \textsc{bv}(\phi) \cup \textsc{bv}(e)$.

This bound variables function produces a set which mirrors the purpose of the set of type variables captured by a function type, but it is easier to compute given the structure of expressions in the operational semantics.

Next we define a function $\textsc{einst}(*, \boldsymbol{x}, \mathcal{C})$ which freshens the free variables in the language construct $*$ except those appearing in $\boldsymbol{x}$. This function is the analogue of $\textsc{inst}(*, \boldsymbol{\alpha}, \mathcal{C})$ in Definition 5.10 and, as such, is largely the natural homomorphism over the language constructs. The only case in which this function deviates is shown below:

Definition A.4 (Evaluation Instantiation)**.**

$$\textsc{einst}(\phi \texttt{->} e, \boldsymbol{x}, \mathcal{C}) = \phi \texttt{->} (\textsc{einst}(e, \boldsymbol{x} \cup \textsc{bv}(\phi \texttt{->} e), \mathcal{C}))$$

$$\textsc{einst}(\langle \iota, n, \dot{\mathcal{C}} \rangle, \boldsymbol{x}, \mathcal{C}) = \begin{cases} \langle \iota, n, \mathcal{C} \rangle & \text{if } \dot{\mathcal{C}} = \bullet \wedge \langle \iota, n, \dot{\mathcal{C}} \rangle \notin \boldsymbol{x} \\ \langle \iota, n, \dot{\mathcal{C}} \rangle & \text{otherwise} \end{cases}$$

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.5 (Evaluation Freshening)**.**

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

In this function, $\mathcal{C}'$ is analogous to the result of $\textsc{make}$ in Figure 5.3. We overload $\textsc{efresh}$ to be defined over lists of scapes, which is calculated by applying $\textsc{efresh}$ pointwise on each scape.

Using the freshening function as defined above, we can now clarify its use in the small step semantics in Definition 3.5. We do so in the following definition:

Definition A.6 (Clarified Application Semantics)**.**

$$E \,\|[x = x_1\ x_2]\,\| e'' \longrightarrow^1 E \,\|\, \textsc{efresh}(x, E_0 \,\|\, e' \,\|[x' = r]) \,\|[x = \textsc{efresh}(x, x')]\,\| e''$$
$$\text{when } E{\downarrow^*_{\texttt{fun}}}(x_1) = \overbrace{\phi \texttt{->} e}^{n}, x_2 \preceq_E \overbrace{\phi \texttt{->} e}^{n} \backslash E_0; e' \,\|[x' = r]$$

Observe that Definition A.6 merely replaces the use of $\alpha(\cdot)$ with $\textsc{efresh}(x, \cdot)$. For this proof, we take the small-step semantics of MicroBang to be those in Definition 3.5 except the application case, which is taken from Definition A.6. In light of this precise definition of freshening, it is now possible to define the simulation relation for the soundness proof.

A.2 Simulation

We now define the simulation relation as a two place relation defined between a language construct and its corresponding type system construct. Values, for example, are simulated by types; redex assignments are simulated by constraints. We write $*_1 \preccurlyeq *_2$ to denote that language construct $*_1$ is simulated by type system construct $*_2$. We define this relation as follows:

Definition A.7 (Simulation)**.** Simulation is defined by the rules in Figure A.1.

A.3 Contours

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

Lemma A.8. Given any contour $\mathcal{C} \neq \emptyset$, $[\![\mathcal{C}]\!] \neq \emptyset$.

variables:			
❁	$\preccurlyeq$	❁	
$\mathcal{C}$	$\preccurlyeq$	$\mathcal{C}'$	iff $\mathcal{C} \leq \mathcal{C}'$
$\langle \iota, n, \dot{c} \rangle$	$\preccurlyeq$	$\langle \iota, n, \dot{c}' \rangle$	iff $\dot{c} \preccurlyeq \dot{c}'$
$\boldsymbol{x}$	$\preccurlyeq$	$\boldsymbol{\alpha}$	iff $\forall x' \in \boldsymbol{x}.\ \exists \alpha \in \boldsymbol{\alpha}.\ x' \preccurlyeq \alpha$
patterns:			
$x : \varphi$	$\preccurlyeq$	$\alpha \sim \tau_\varphi$	iff $x \preccurlyeq \alpha \wedge \varphi \preccurlyeq \tau_\varphi$
`int`	$\preccurlyeq$	`int`	
$\mathrm{lbl}\, x$	$\preccurlyeq$	$\mathrm{lbl}\, \alpha$	iff $x \preccurlyeq \alpha$
`any`	$\preccurlyeq$	`any`	
values:			
x	$\preccurlyeq$	α	iff $x \preccurlyeq \alpha$
$\boldsymbol{x}$	$\preccurlyeq$	$\boldsymbol{\alpha}$	iff $\forall x \in \boldsymbol{x}.\ \exists \alpha \in \boldsymbol{\alpha}.\ x \preccurlyeq \alpha$
n	$\preccurlyeq$	`int`	iff $n \in \mathbb{Z}$
$\mathrm{lbl}\, x$	$\preccurlyeq$	$\mathrm{lbl}\, \alpha$	iff $x \preccurlyeq \alpha$
x_1 `&` x_2	$\preccurlyeq$	α_1 `&` α_2	iff $x_1 \preccurlyeq \alpha_1 \wedge x_2 \preccurlyeq \alpha_2$
`()`	$\preccurlyeq$	`()`	
ϕ `->` e	$\preccurlyeq$	$\forall \boldsymbol{\alpha}.\tau_\phi \to \alpha \setminus \mathrm{C}$	iff $\textsc{bv}(\phi \texttt{->} e) \preccurlyeq \boldsymbol{\alpha} \wedge \phi \preccurlyeq \tau_\phi \wedge e \preccurlyeq \langle \alpha, \mathrm{C} \rangle$
redexes:			
x `=` v	$\preccurlyeq$	$\tau <: \alpha$	iff $x \preccurlyeq \alpha \wedge v \preccurlyeq \tau$
x_3 `=` $x_1\ x_2$	$\preccurlyeq$	$\alpha_1\ \alpha_2 <: \alpha_3$	iff $\forall i \in \{1..3\}.x_i \preccurlyeq \alpha_i$
x_3 `=` $x_1\ \mathrm{op}\ x_2$	$\preccurlyeq$	$\alpha_1\ \mathrm{op}\ \alpha_2 <: \alpha_3$	iff $\forall i \in \{1..3\}.x_i \preccurlyeq \alpha_i$
expressions:			
$[]$	$\preccurlyeq$	C	
$e \,\|\|[x = r]$	$\preccurlyeq$	$\mathrm{C} \cup \{\mathrm{c}\}$	iff $e \preccurlyeq \mathrm{C} \wedge x = r \preccurlyeq \mathrm{c}$
$e \,\|\|[x = r]$	$\preccurlyeq$	$\langle \alpha, \mathrm{C} \rangle$	iff $e \,\|\|[x = r] \preccurlyeq \mathrm{C} \wedge x \preccurlyeq \alpha$
environments:			
$[]$	$\preccurlyeq$	$\emptyset$	
$E \,\|\|[x = v]$	$\preccurlyeq$	$\Gamma \cup \{x : \tau\}$	iff $E \preccurlyeq \Gamma \wedge v \preccurlyeq \tau$

Figure A.1. Simulation Relation Definition

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.9. 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.8, we have that $[\![\mathcal{C}_1]\!] \neq \emptyset$ and so $\mathcal{C}_2 \between \mathcal{C}_3$ by Definition 5.9. □

Lemma A.10. Contour subsumption is transitive.

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

Lemma A.11. $(\!|\mathrm{C}_1 \cup \mathrm{C}_2|\!) = (\!|\mathrm{C}_1|\!) \cup (\!|\mathrm{C}_2|\!)$

Proof. Trivial by inspection of Definition 5.10. □

Lemma A.12. If τ appears in C, then $(\!|\tau|\!) \subseteq (\!|\mathrm{C}|\!)$. Similar statements are true about other constructs appearing in C.

Proof. Trivial by inspection of Definition 5.10. □

Lemma A.13. $[\![\mathcal{S}_1]\!] \subseteq [\![\mathcal{S}_2]\!]$ 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. For all of the below, let $\mathcal{S}_1 = \overset{n}{\rightharpoonup} \mathcal{P}$ and let $\mathcal{S}_2 = \overset{n}{\rightharpoonup} \mathcal{P}'$.

First, assume $[\![\mathcal{S}_1]\!] \subseteq [\![\mathcal{S}_2]\!]$. We intend to show that $[\![\mathcal{S}_1 \,\|[\mathcal{P}]]\!] \subseteq [\![\mathcal{S}_2 \,\|[\mathcal{P}]]\!]$. By Definition 5.10, every element in $[\![\mathcal{S}_1 \,\|[\mathcal{P}]]\!]$ is of the form $\vec{\iota}_1 \,\|\, \vec{\iota}_3$ where $\vec{\iota}_1 \in [\![\mathcal{S}_1]\!]$ and $\vec{\iota}_3 \in [\![\mathcal{P}]\!]$; we also know that every element of $[\![\mathcal{S}_2 \,\|[\mathcal{P}]]\!]$ is of the form $\vec{\iota}_2 \,\|\, \vec{\iota}_3$ where $\vec{\iota}_2 \in [\![\mathcal{S}_2]\!]$. Because $[\![\mathcal{S}_1]\!] \subseteq [\![\mathcal{S}_2]\!]$, we have that any $\vec{\iota}_1 \,\|\, \vec{\iota}_3 = \vec{\iota}_2 \,\|\, \vec{\iota}_3$ for some $\vec{\iota}_2$; thus, $[\![\mathcal{S}_1 \,\|[\mathcal{P}]]\!] \subseteq [\![\mathcal{S}_2 \,\|[\mathcal{P}]]\!]$.

We next assume $[\![\mathcal{S}_1 \,\|[\mathcal{P}]]\!] \subseteq [\![\mathcal{S}_2 \,\|[\mathcal{P}]]\!]$ with the intent of proving that $[\![\mathcal{S}_1]\!] \subseteq [\![\mathcal{S}_2]\!]$. By Definition 5.10, every element in $[\![\mathcal{S}_1 \,\|[\mathcal{P}]]\!]$ is of the form $\vec{\iota}_1 \,\|\, \vec{\iota}_3$ where $\vec{\iota}_1 \in [\![\mathcal{S}_1]\!]$ and $\vec{\iota}_3 \in [\![\mathcal{P}]\!]$; we also know that every element of $[\![\mathcal{S}_2 \,\|[\mathcal{P}]]\!]$ is of the form $\vec{\iota}_2 \,\|\, \vec{\iota}_3$ where $\vec{\iota}_2 \in [\![\mathcal{S}_2]\!]$. For the first set, we can create a relation defining each element as equivalent if their values of $\vec{\iota}_1$ are equivalent; we can do likewise for the second set and $\vec{\iota}_2$. Each equivalence class is of the same size and contains a number of elements equal to the possible forms of $\vec{\iota}_3$, which is the cardinality of $[\![\mathcal{P}]\!]$. Since $[\![\mathcal{S}_1 \,\|[\mathcal{P}]]\!] \subseteq [\![\mathcal{S}_2 \,\|[\mathcal{P}]]\!]$, we know that there are at least as many elements in the second set as there are in the first; since all equivalence classes are the same size, we know that there are at least as many equivalence classes in the second set as there are in the first. Because equivalence classes from these sets can be indexed by $\vec{\iota}_1$ and $\vec{\iota}_2$ respectively, we know that the set of all $\vec{\iota}_1$ is a subset of the set of all $\vec{\iota}_2$. By definition, $\vec{\iota}_1$ is an arbitrary member of $[\![\mathcal{S}_1]\!]$ and $\vec{\iota}_2$ is an arbitrary member of $[\![\mathcal{S}_2]\!]$; thus, $[\![\mathcal{S}_1]\!] \subseteq [\![\mathcal{S}_2]\!]$.

To prove that $[\![\mathcal{S}_1]\!] \subseteq [\![\mathcal{S}_2]\!]$ iff $[\![[\mathcal{P}] \,\|\, \mathcal{S}_1]\!] \subseteq [\![[\mathcal{P}] \,\|\, \mathcal{S}_2]\!]$, we use the same approach as above. □

Lemma A.14. $[\![\mathcal{C}_1]\!] \subseteq [\![\mathcal{C}_2]\!]$ 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.14. In particular, the semantics of contours simply expands the number of forms of $\vec{\iota}_1$ and $\vec{\iota}_2$; the rest of the proof structure is identical. □

A.4 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.15.

Definition A.15 (Well-Formed Contour Set)**.** A contour set $\boldsymbol{\mathcal{C}}$ is well-formed when $\forall \mathcal{c}_1 \in \boldsymbol{\mathcal{C}}.\mathcal{c}_1 \neq \emptyset \wedge \forall \mathcal{c}_2 \in \boldsymbol{\mathcal{C}}.\mathcal{c}_1 \neq \mathcal{c}_2 \implies \mathcal{c}_1 \not\between \mathcal{c}_2$. A constraint set C is well-formed when $(\!|\mathrm{C}|\!)$ is well-formed.

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

Lemma A.16. Let $\mathrm{C}' = \mathrm{C}_1 \cup \mathrm{C}_2$ where C_1 and C_2 are well-formed. If $(\!|\mathrm{C}_1|\!) \subseteq (\!|\mathrm{C}_2|\!)$, then C' is also well-formed.

Proof. By Lemma A.11 and because $(\!|\mathrm{C}_1|\!) \subseteq (\!|\mathrm{C}_2|\!)$, we thus have $(\!|\mathrm{C}'|\!) = (\!|\mathrm{C}_1 \cup \mathrm{C}_2|\!) = (\!|\mathrm{C}_1|\!) \cup (\!|\mathrm{C}_2|\!) = (\!|\mathrm{C}_2|\!)$. If C_2 is well-formed, we have that $(\!|\mathrm{C}_2|\!)$ is well-formed; thus, $(\!|\mathrm{C}'|\!)$ is well-formed and so C' is well-formed. □

Lemma A.17. 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 $\{\mathcal{c}_2, \mathcal{c}_3\} \subseteq (\!|\mathrm{C}|\!)$. Then $\mathcal{c}_2 = \mathcal{c}_3$.

Proof. Because C is well-formed, we know that $\mathcal{c}_1 \neq \emptyset$. By Lemma A.9, we have that $\mathcal{c}_2 \between \mathcal{c}_3$. Again because C is well-formed, we have that $\forall \mathcal{c}, \mathcal{c}' \in \mathrm{C}.\mathcal{c} \neq \mathcal{c}' \implies \mathcal{c} \not\between \mathcal{c}'$. Since $\mathcal{c}_2 \between \mathcal{c}_3$ we must therefore have $\mathcal{c}_2 = \mathcal{c}_3$. □

Lemma A.18. Let $\boldsymbol{\mathcal{C}}$ be well-formed and let $\boldsymbol{\mathcal{C}}' \subseteq \boldsymbol{\mathcal{C}}$; then $\boldsymbol{\mathcal{C}}'$ is also well-formed. If C and C' are well-formed and $(\!|\mathrm{C}'|\!) \subseteq (\!|\mathrm{C}|\!)$, then $\mathrm{C} \cup \mathrm{C}'$ is well-formed. If C is well-formed and $\mathrm{C}' \subseteq \mathrm{C}$, then C' is also well-formed.

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

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

If $\mathrm{C}' \subseteq \mathrm{C}$, then $(\!|\mathrm{C}'|\!) \subseteq (\!|\mathrm{C}|\!)$; this can be established from Definition 5.9 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.5 Simulation Properties

We next show useful properties of the simulation relation in Definition A.7.

Lemma A.19. Suppose $\overrightarrow{x_\square^{\,n} = r_\square} \preccurlyeq \mathrm{C}$. Then $[x_1 = r_1, \ldots, x_{i-1} = r_{i-1}, x_{i+1} = r_{i+1}, \ldots, x_n = r_n] \preccurlyeq \mathrm{C}$.

Proof. Immediate from Definition A.7. □

Lemma A.20. Suppose $\overrightarrow{x'^{\,n'}_\square = r'_\square} \preccurlyeq \mathrm{C}'$ and $\overrightarrow{x''^{\,n''}_\square = r''_\square} \preccurlyeq \mathrm{C}''$. Let $\overrightarrow{x^{\,n+m}_\square = r_\square}$ 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 $x_j = x'_j$ for all $j \leq k$, $x_j = x'_{j-m}$ for all $j > k + m$, and $x_j = x''_{j-k}$ otherwise (and likewise for r). Let $\mathrm{C} = \mathrm{C}' \cup \mathrm{C}''$. Then $\overrightarrow{x^{\,n}_\square = r_\square} \preccurlyeq \mathrm{C}$.

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

Proof. By repeated application of Definition A.7, we have that $[] \,\|\, [x'_1 = r'_1] \,\|\, \ldots \,\|\, [x'_n = r'_n] \preccurlyeq \mathrm{C}''' \cup \{\mathrm{c}'_1\} \cup \ldots \cup \{\mathrm{c}'_n\}$ for some C''' such that $\mathrm{C}''' \cup \{\mathrm{c}'_1\} \cup \ldots \cup \{\mathrm{c}'_n\} = \mathrm{C}'$. Likewise, $[] \,\|\, [x''_1 = r''_1] \,\|\, \ldots \,\|\, [x''_n = r''_n] \preccurlyeq \mathrm{C}'''' \cup \{\mathrm{c}''_1\} \cup \ldots \cup \{\mathrm{c}''_n\}$ for some C'''' such that $\mathrm{C}'''' \cup \{\mathrm{c}''_1\} \cup \ldots \cup \{\mathrm{c}''_n\} = \mathrm{C}''$. We choose $\mathrm{C}''' = \mathrm{C}'$ and $\mathrm{C}'''' = \mathrm{C}''$ for convenience.

We aim to show that $\overrightarrow{x^{\,n+m}_\square = r_\square} \preccurlyeq \mathrm{C}$. Because $\mathrm{C} = \mathrm{C}' \cup \mathrm{C}''$, we have that $\mathrm{C} = \mathrm{C}' \cup \mathrm{C}'' \cup \underbrace{\mathrm{c}'_\square}_{n} \cup \underbrace{\mathrm{c}''_\square}_{m}$. We can thus show that $[] \,\|\, [x_1 = r_1] \,\|\, \ldots \,\|\, [x_{n+m} = r_{n+m}] \preccurlyeq \mathrm{C} \cup \{\mathrm{c}_1\} \cup \ldots \cup \{\mathrm{c}_{n+m}\}$ by choosing each c as we chose x: we let $\mathrm{c}_j = \mathrm{c}'_j$ for all $j \leq k$, $\mathrm{c}_j = \mathrm{c}'_{j-m}$ for all $j > k + m$, and $\mathrm{c}_j = \mathrm{c}''_{j-k}$ otherwise. By Definition A.7, we are finished.

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

Lemma A.21. Suppose $\boldsymbol{x} \preccurlyeq \boldsymbol{\alpha}$ and $\boldsymbol{x}' \preccurlyeq \boldsymbol{\alpha}'$. Then $\boldsymbol{x} \cup \boldsymbol{x}' \preccurlyeq \boldsymbol{\alpha} \cup \boldsymbol{\alpha}'$.

Proof. From $\boldsymbol{x} \preccurlyeq \boldsymbol{\alpha}$ we have that for each x in $\boldsymbol{x}$ there exists some α in $\boldsymbol{\alpha}$ such that $x \preccurlyeq \alpha$; thus, there exists some α in $\boldsymbol{\alpha} \cup \boldsymbol{\alpha}'$ such that $x \preccurlyeq \alpha$. The same argument can be made regarding $\boldsymbol{x}'$. As a result, every x in $\boldsymbol{x} \cup \boldsymbol{x}'$ has some α in $\boldsymbol{\alpha} \cup \boldsymbol{\alpha}'$ such that $x \preccurlyeq \alpha$. By Definition A.7, we are finished. □

Lemma A.22. If $e \preccurlyeq \mathrm{C}$, then $x_2 = x_1 \in e \implies \exists \alpha_1, \alpha_2.\ \alpha_1 <: \alpha_2 \in \mathrm{C} \wedge x_2 = x_1 \preccurlyeq \alpha_1 <: \alpha_2$. Corresponding statements hold true for the remaining expression and constraint forms.

Proof. Immediate from Definition A.7. □

Lemma A.23. If $e \preccurlyeq \mathrm{C}$ and C is well-formed, then $x_2 = x_1 \in e \wedge x_3 = x_2 \in e \implies \exists \alpha_1, \alpha_2, \alpha_3 \in \mathrm{C}.\ \{\alpha_1 <: \alpha_2, \alpha_2 <: \alpha_3\} \in \mathrm{C} \wedge \forall i \in \{1..3\}.x_i \preccurlyeq \alpha_i$. Similar statements hold for other sequences of constraints.

Proof. From Lemma A.22, we have that there exist some α_1, α_2, α'_2, and α_3 such that $\{\alpha_1 <: \alpha_2, \alpha'_2 <: \alpha_3\} \in \mathrm{C}$, $x_1 \preccurlyeq \alpha_1$, $x_2 \preccurlyeq \alpha_2$, $x_2 \preccurlyeq \alpha'_2$, and $x_3 \preccurlyeq \alpha_3$. It remains to show that $\alpha_2 = \alpha'_2$.

Let $x_2 \leftrightarrow \langle \iota_1, n_1, \dot{\mathcal{c}}_1 \rangle$, let $\alpha_2 = \langle \iota_2, n_2, \dot{\mathcal{c}}_2 \rangle$, and let $\alpha'_2 = \langle \iota'_2, n'_2, \dot{\mathcal{c}}'_2 \rangle$. From $x_2 \preccurlyeq \alpha_2$ and $x_2 \preccurlyeq \alpha'_2$, we have that $\iota_1 = \iota_2 = \iota'_2$ and that $n_1 = n_2 = n'_2$. We have two cases: one in which $\dot{\mathcal{c}}_1 = \bullet$ and one in which it does not. If $\dot{\mathcal{c}}_1 = \bullet$, then $x_2 \preccurlyeq \alpha_2$ gives us by Definition A.7 that $\dot{\mathcal{c}}_2 = \bullet$; similarly, $x_2 \preccurlyeq \alpha'_2$ gives us that $\dot{\mathcal{c}}'_2 = \bullet$. 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 $\dot{\mathcal{c}}'_2 = \mathcal{c}'_2$ such that $\mathcal{c}_1 \leq \mathcal{c}_2$ and $\mathcal{c}_1 \leq \mathcal{c}'_2$ Because C is well-formed, we know by Lemma A.17 that $\mathcal{c}_2 = \mathcal{c}'_2$; thus, $\alpha_2 = \alpha'_2$. □

A.6 Initial Derivation Simulation

We now show that the initial type derivation of an expression simulates that expression. To do so, we begin by defining the notion of a *canonical* context.

Definition A.24 (Canonical Context)**.** A context Γ is *canonical* iff for all mappings $\langle \iota, n, \dot{c} \rangle : \langle \iota', n', \dot{c}' \rangle$ in Γ, we have $\iota = \iota'$, $n = n'$, and $\dot{c} = c \leq c' = \dot{c}'$.

We now show some lemmas over canonical contexts.

Lemma A.25. If Γ and Γ' are canonical, then $\Gamma \cup \Gamma'$ is canonical.

Proof. Trivial by definition of set union. □

Lemma A.26. Given $\{x : \alpha\} \in \Gamma$ where Γ is canonical, $x \preccurlyeq \alpha$.

Proof. Let $x \leftrightarrow \langle \iota, n, \dot{c} \rangle$. By Definition A.24, $\dot{c} = c$ and $\alpha = \langle \iota, n, c' \rangle$ such that $c \leq c'$. By Definition A.7, $x \preccurlyeq \alpha$. □

Our next objective is to show that the type and constraint set given by initial type derivation simulates the input expression. We begin by showing the simulation of initial pattern derivation.

Lemma A.27. For any ϕ and canonical Γ such that $[\![\phi]\!]^{\Gamma}_{\mathrm{P}} = \langle \Gamma', \tau_\phi \rangle$, we have that Γ' is canonical, every variable in Γ' has the non-contour $\clubsuit$, and that $\phi \preccurlyeq \tau_\phi$.

Proof. By case analysis.

If $\phi = x : \mathtt{int}$, then $[\![\phi]\!]^{\Gamma}_{\mathrm{P}} = \langle \{x : \dot{\alpha}_1\}, \dot{\alpha}_1 \sim \mathtt{int} \rangle$. By Definition 5.2, we have that $\dot{\alpha}_1 = \langle \iota, 1, \clubsuit \rangle$ where ι identifies this pattern. By Definition A.2, we have that $x \leftrightarrow \langle \iota, 1, \clubsuit \rangle$. Thus $\{x : \dot{\alpha}_1\}$ is canonical by Definition A.24. By Definition A.7, we have that $x \preccurlyeq \dot{\alpha}_1$; thus, we have $x : \mathtt{int} \preccurlyeq \dot{\alpha}_1 \sim \mathtt{int}$ and we are finished. Because the only variable appearing in the context is of the form $\dot{\alpha}_i$ and all such variables have the non-contour $\clubsuit$, the context has only variables with the non-contour.

Similar arguments hold for the $x : \mathrm{lbl}\, x'$ and $x : \mathtt{any}$ cases. □

In order to support the lemmas over initial derivation, we will need to show some properties that align the bound variable set of a scape with the bound variable set of its type. We start by defining a function to retrieve the upper bound of a constraint; it is also overloaded to find the upper bounds of a constraint set.

Definition A.28 (Constraint Upper Bounds)**.**

$$\mathrm{UB}(\mathrm{c}) = \begin{cases} \alpha & \text{when } \mathrm{c} = \tau <: \alpha \\ \alpha & \text{when } \mathrm{c} = \alpha' <: \alpha \\ \alpha & \text{when } \mathrm{c} = \alpha'\ \alpha'' <: \alpha \\ \alpha & \text{when } \mathrm{c} = \alpha'\ \mathrm{op}\ \alpha'' <: \alpha \end{cases}$$

$$\mathrm{UB}(\mathrm{C}) = \{\mathrm{UB}(\mathrm{c}) \mid \mathrm{c} \in \mathrm{C}\}$$

We then show the simple property that the upper bounds of scape type's constraint set simulate the bound variables of its scape.

Lemma A.29. For any e such that $[\![e]\!]^{\Gamma}_{\mathrm{E}} = \langle \alpha', \mathrm{C} \rangle$, we have $\mathrm{BV}(e) \preccurlyeq \mathrm{UB}(\mathrm{C})$.

Proof. Let $e = \overset{n}{\overrightarrow{x = r}}$. For all $1 \leq i \leq n$, let $\alpha_i = \langle \iota_i, 1, \clubsuit \rangle$ where ι_i is the identity of $x_i = r_i$. Let $\Gamma' = \{x_i : \alpha_i \mid 1 \leq i \leq n\}$. Then by Figure 4.2, we have that $\alpha' = \alpha_n$ and that $\mathrm{C} = \overset{n}{\llcorner \mathrm{c}}$ where for all $1 \leq i \leq n$ we have $\mathrm{c}_i = [\![x_i = r_i]\!]^{\Gamma \cup \Gamma'}_{\mathrm{R}}$.

By Definition A.3, we have that $\mathrm{BV}(e) = \{x_i \mid 1 \leq i \leq n\}$. By Definition A.28, we have that $\mathrm{UB}(\mathrm{C}) = \{\mathrm{UB}(\mathrm{c}_i) \mid 1 \leq i \leq n\}$. By Definition A.7, we have that $\{x_i \mid 1 \leq i \leq n\} \preccurlyeq \{\mathrm{UB}(\mathrm{c}_i) \mid 1 \leq i \leq n\}$ if $\forall x \in \{x_i \mid 1 \leq i \leq n\}.\ \exists \alpha \in \{\mathrm{UB}(\mathrm{c}_i) \mid 1 \leq i \leq n\}.\ x \preccurlyeq \alpha$. It is therefore sufficient to show that $\forall i \in \{1..n\}. x_i \preccurlyeq \mathrm{UB}(\mathrm{c}_i)$. More generally, it is sufficient to show that for all $[\![x' = r']\!]^{\Gamma}_{\mathrm{R}} = \mathrm{c}'$ we have $x' \preccurlyeq \mathrm{UB}(\mathrm{c}')$. This is trivial by case analysis. □

We next show some supporting lemmas.

Lemma A.30. If $\phi \preccurlyeq \tau_\phi$ then $\mathrm{BV}(\phi) \preccurlyeq \mathrm{FTV}(\tau_\phi)$.

Proof. By case analysis. If $\phi = x : \mathtt{int}$ then by Definition A.7 we have $\tau_\phi = \alpha \sim \mathtt{int}$ such that $x \preccurlyeq \alpha$. We have that $\mathrm{BV}(\phi) = \{x\}$ and that $\mathrm{FTV}(\tau_\phi) = \{\alpha\}$ and so this case is finished. A similar argument is made when $\phi = x : \mathtt{any}$.

Otherwise, $\phi = x : \mathrm{lbl}\, x'$ and by Definition A.7 we have $\tau_\phi = \alpha \sim \mathrm{lbl}\, \alpha'$ such that $x \preccurlyeq \alpha$ and $x' \preccurlyeq \alpha'$. We have that $\mathrm{BV}(\phi) = \{x, x'\}$ and that $\mathrm{FTV}(\tau_\phi) = \{\alpha, \alpha'\}$ and so we are finished. □

We now present the proof that initial derivation simulates as expected.

Lemma A.31. Let e be an expression $\overset{n}{\overrightarrow{x_\square = r_\square}}$ with unique bindings closed over a canonical context Γ_1 such that no variable is bound in both e and Γ_1 and all variables in Γ_1 have the non-contour $\clubsuit$. If $[\![e]\!]^{\Gamma_1}_{\mathrm{E}} = \langle \alpha, \mathrm{C} \rangle$, then $e \preccurlyeq \langle \alpha, \mathrm{C} \rangle$. Further, C is well-formed and every type variable in $\mathrm{FTV}(\mathrm{C})$ is either in Γ_1 or simulates some variable in $\overset{n}{\overrightarrow{x}}$.

Proof. By structural induction on the size of e.

For all $1 \leq i \leq n$, let $\alpha_i = \langle \iota_i, 1, \clubsuit \rangle$ where ι_i is the identity of $x_i = r_i$. Let $\Gamma_2 = \{x_i : \alpha_i \mid 1 \leq i \leq n\}$. Observe that Γ_2 is canonical by Definition A.24. Let $\Gamma_3 = \Gamma_1 \cup \Gamma_2$. We observe that this context contains only variables with the non-contour. Also, by Lemma A.25 this context is canonical. For all $1 \leq i \leq n$, let $\mathrm{c}_i = [\![x_i = r_i]\!]^{\Gamma_3}_{\mathrm{R}}$; then $\mathrm{C} = \overset{n}{\llcorner \mathrm{c}}$ and $\alpha = \alpha_n$.

To show the property that $e \preccurlyeq \langle \alpha, \mathrm{C} \rangle$, it is thus sufficient to show that $e \preccurlyeq \mathrm{C}$. By induction on the length of e using Definition A.7, it is sufficient to show that $x_i = r_i \preccurlyeq \mathrm{c}_i$ for all $1 \leq i \leq n$ to prove this property. To show the property that every type variable in $\mathrm{FTV}(\mathrm{C})$ is either in Γ_1 or simulates some variable in $\overset{n}{\overrightarrow{x}}$, it is sufficient to show that every type variable in $\mathrm{FTV}(\mathrm{c}_i)$ is in Γ_3 for all $1 \leq i \leq n$. We defer showing the property that C is well-formed until the end of

the proof. We now show each of these properties for every c_i by case analysis on the redex r_i for each $1 \leq i \leq n$.

If r_i is x', then $[\![x_i = r_i]\!]_{\mathrm{R}}^{\Gamma_3} = \Gamma_3(x') <: \Gamma_3(x)$. By Lemma A.26, $x \preccurlyeq \Gamma_3(x)$ and $x' \preccurlyeq \Gamma_3(x')$. It is trivial that the free type variables for this constraint are in Γ_3; they are retrieved directly from that context. Thus by Definition A.7, this case is finished. Similar arguments hold when r_i is $\mathsf{lbl}\, x'$, $x' \,\&\, x''$, $x'\ x''$, or $x' \,\mathsf{op}\, x''$.

If r_i is $()$, then $[\![x_i = r_i]\!]_{\mathrm{R}}^{\Gamma_3} = () <: \Gamma_3(x)$. By Lemma A.26, we have $x \preccurlyeq \Gamma_3(x)$; by Definition A.7, we have $() \preccurlyeq ()$. Thus by Definition A.7, this case is finished. A similar argument holds when r_i is some $n \in \mathbb{Z}$.

If r_i is of the form $\phi \texttt{->} e'$, then let $\overset{m}{\longrightarrow}{}'_{\square} = r'_{\square} = e'$. We have that $[\![x_i = r_i]\!]_{\mathrm{R}}^{\Gamma_3} = \forall \boldsymbol{\alpha}.\tau_\phi \to \alpha' \setminus \mathrm{C}'$ such that (1) $[\![\phi]\!]_{\mathrm{P}}^{\Gamma_3} = \langle \Gamma_4, \tau_\phi \rangle$ for some Γ_4, (2) $[\![e']\!]_{\mathrm{E}}^{\Gamma_3 \cup \Gamma_4} = \langle \alpha', \mathrm{C}' \rangle$, and (3) $\boldsymbol{\alpha} = (\mathrm{FTV}(\tau_\phi) \cup \mathrm{FTV}(\mathrm{C}')) - \boldsymbol{\alpha}'$ where $\boldsymbol{\alpha}' = \{\alpha'' \mid x : \alpha'' \in \Gamma_3\}$. By Lemma A.27, we have that Γ_4 is canonical and only contains variables with the non-contour.

Let $\Gamma_5 = \Gamma_3 \cup \Gamma_4$; observe that this context contains only variables with the non-contour and, by Lemma A.25, is canonical. Because (1) Γ_1 contains no variables bound in e, (2) Γ_2 contains only variables bound directly by e, (3) Γ_4 contains only variables bound by τ_ϕ, and (4) variable bindings are unique in e, we know that Γ_5 does not contain any bindings that appear in e'. Thus by the induction hypothesis we have that $e' \preccurlyeq \langle \alpha', \mathrm{C}' \rangle$ and C' is well-formed and every type variable in $\mathrm{FTV}(\mathrm{C}')$ is either in Γ_5 or simulates some variable x'_j where $1 \leq j \leq m$. Suppose we were to show that $\mathrm{BV}(\phi \texttt{->} e') \preccurlyeq \boldsymbol{\alpha}$; then Definition A.7 gives us that $r_i \preccurlyeq \forall \boldsymbol{\alpha}.\tau_\phi \to \alpha' \setminus \mathrm{C}'$, by Lemma A.26 we have $x \preccurlyeq \Gamma_3(x)$, and so by Definition A.7 we would have $x_i = r_i \preccurlyeq c_i$. So it remains to show that $\mathrm{BV}(\phi \texttt{->} e') \preccurlyeq \boldsymbol{\alpha}$ and that every free type variable in c_i is in Γ_3.

We have from above that free type variable in C' is either in Γ_5 or simulates some variable x'_j. This is to say that the variable is either in Γ_3, in Γ_4, or simulates some variable x'_j. The constraint c_i contains a type variable known to be in Γ_3 and the type of the scape. The type of the scape captures every free variable in τ_ϕ (those appearing in Γ_4) and every free variable in C'; thus, the scape type has no free variables and so the only free variable in c_i is in Γ_3. It remains to show that $\mathrm{BV}(\phi \texttt{->} e') \preccurlyeq \boldsymbol{\alpha}$.

By Definition A.3 and the above, $\mathrm{BV}(\phi \texttt{->} e') \preccurlyeq \boldsymbol{\alpha}$ is equivalent to $\mathrm{BV}(\phi) \cup \mathrm{BV}(e') \preccurlyeq (\mathrm{FTV}(\tau_\phi) \cup \mathrm{FTV}(\mathrm{C}')) - \boldsymbol{\alpha}'$. By Lemma A.21, it is sufficient to show that $\mathrm{BV}(\phi) \preccurlyeq \mathrm{FTV}(\tau_\phi) - \boldsymbol{\alpha}'$ and that $\mathrm{BV}(e') \preccurlyeq \mathrm{FTV}(\mathrm{C}') - \boldsymbol{\alpha}'$. Because Γ_3 does not contain any variables in ϕ and because Γ_3 is canonical, it does not contain any variables in τ_ϕ; therefore, the prior condition reduces to $\mathrm{BV}(\phi) \preccurlyeq \mathrm{FTV}(\tau_\phi)$. Because $\phi \preccurlyeq \tau_\phi$, Lemma A.30 gives us this property.

It thus remains to show that $\mathrm{BV}(e') \preccurlyeq \mathrm{FTV}(\mathrm{C}') - \boldsymbol{\alpha}'$. By Lemma A.29, it is sufficient to show that $\mathrm{FTV}(\mathrm{C}') - \boldsymbol{\alpha}' \subseteq \mathrm{UB}(\mathrm{C}')$. From above, we know that every type variable in $\mathrm{FTV}(\mathrm{C}')$ is either in $\boldsymbol{\alpha}'$ or simulates a variable in $\overset{m}{\rightarrow}{}'$, which is the definition of $\mathrm{BV}(e')$; thus, this case is finished.

We now show the deferred condition that C is well-formed. Every variable appearing in that constraint set is either from the above Γ_2 or Γ_4 (which have only variables with the non-contour), from Γ_1 (in which every variable has the non-contour), or is from some induction on this lemma. Since every variable in C has the non-contour, $(\!|\mathrm{C}|\!) = \emptyset$ and so C is well-formed. □

A.7 Simulation Preservation

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

Lemma A.32. If $E \preccurlyeq \mathrm{C}$, then $E(x) = v \wedge x \preccurlyeq \alpha \implies \tau \overset{\mathrm{C}}{<:}{}^* \alpha \wedge v \preccurlyeq \tau$.

Proof. For $E(x) = v$ to be defined, there must exist some $x = v' \in E$. We have two cases: either v' is of the form x' or it is not. If it is not, then $v = v'$ by the definition of $E(x)$. By $E \preccurlyeq \mathrm{C}$ and Lemma A.22, we know that $\tau <: \alpha \in \mathrm{C}$ such that $x \preccurlyeq \alpha$ and $v \preccurlyeq \tau$. By Definition 4.1, we have that $\tau \overset{\mathrm{C}}{<:}{}^* \alpha$.

In the case that v' is of the form x', then there exists some $x = x' \in E$. By $E \preccurlyeq \mathrm{C}$ and Lemma A.22, we have that there exists some $\alpha' <: \alpha \in \mathrm{C}$ such that $x \preccurlyeq \alpha$ and $x' \preccurlyeq \alpha'$. By definition of $E(x)$, we know that $E(x) = E(x')$. By the induction hypothesis, we have that $\tau \overset{\mathrm{C}}{<:}{}^* \alpha' \wedge v \preccurlyeq \tau$; such induction is sound because we only consider closed e with unique variable names. Because $\tau \overset{\mathrm{C}}{<:}{}^* \alpha'$ and $\alpha' <: \alpha \in \mathrm{C}$, we have by Definition 4.1 that $\tau \overset{\mathrm{C}}{<:}{}^* \alpha$. □

Lemma A.33. If $E \preccurlyeq \mathrm{C}$, $E{\downarrow}^*_\pi(x) = \overset{n}{\rightarrow}{}'$, and $x \preccurlyeq \alpha$, then $\mathrm{C} \vdash \alpha \overset{\pi}{\triangleright} \overset{n}{\rightarrow}{}' \wedge \forall i \in \{1..n\}.\ v_i \preccurlyeq \tau_i$.

Proof. First, we observe that the projection matching operation in Definition 4.3 properly simulates projector matching in the operational semantics; that is, $v \in \pi \wedge v \preccurlyeq \tau$ iff $\tau \sqsubseteq \pi$; this is shown by simple case analysis on π. We then show this lemma by induction on the size of the E preceding the definition of x (which is sound because e is closed and has unique variable names).

If $E(x)$ is a non-onion value v' and $v' \in \pi$, then by Lemma A.32 we have $\tau' \overset{\mathrm{C}}{<:}{}^* \alpha$ and $v' \preccurlyeq \tau'$ for some non-onion type τ'. We also have from the above that $\pi \sqsubseteq \alpha'$. By Definition 3.1, $\overset{n}{\rightarrow}{}' = [v']$; by Definition 4.4, $\overset{n}{\rightarrow}{}' = [\tau']$. Because $v_1 \preccurlyeq \tau_1$ and $n = 1$, this case is finished.

If $E(x)$ is a non-onion value v' and $v' \notin \pi$, then again by Lemma A.32 we have $\tau' \overset{\mathrm{C}}{<:}{}^* \alpha$ and $v' \preccurlyeq \tau'$ for some non-onion type τ'. We also have from the above that $\pi \not\sqsubseteq \alpha'$. By Definition 3.1, $\overset{n}{\rightarrow}{}' = []$; by Definition 4.4, $\overset{n}{\rightarrow}{}' = []$. Therefore this case is finished.

If $E(x)$ is an onion value $x_1 \,\&\, x_2$, then $\overset{n}{\rightarrow}{}' = \overset{n_1}{\rightarrow}{}' \parallel \overset{n_2}{\rightarrow}{}''$ where $n_1 + n_2 = n$, $E{\downarrow}^*_\pi(x_1) = \overset{n_1}{\rightarrow}{}'$, and $E{\downarrow}^*_{x_2}(=) \overset{n_2}{\rightarrow}{}''$. By Lemma A.32 we have $\alpha_1 \,\&\, \alpha_2 \overset{\mathrm{C}}{<:}{}^* \alpha$ and $x_1 \,\&\, x_2 \preccurlyeq \alpha_1 \,\&\, \alpha_2$.

By Definition A.7, we have that $x_1 \preccurlyeq \alpha_1$ and $x_2 \preccurlyeq \alpha_2$. We therefore induct on this lemma for each side of the onion; the left induction gives $\mathrm{C} \vdash \alpha_1 \xrightarrow{\pi}\triangleright \overrightarrow[n_1]{\tau'}$ and $\forall i \in \{1..n_1\}.\ v'_i \preccurlyeq \tau'_i$ while the right induction gives $\mathrm{C} \vdash \alpha_2 \xrightarrow{\pi}\triangleright \overrightarrow[n_2]{\tau''}$ and $\forall i \in \{1..n_1\}.\ v''_i \preccurlyeq \tau''_i$. By Definition 3.1, $\overrightarrow[n]{v} = \overrightarrow[n_1]{v'} \parallel \overrightarrow[n_2]{v''}$; by Definition 4.4, $\overrightarrow[n]{\tau} = \overrightarrow[n_1]{\tau'} \parallel \overrightarrow[n_2]{\tau''}$. We have from above that $\forall i \in \{1..n\}.v \preccurlyeq \tau$; we are therefore finished. □

Lemma A.34. If $E \preccurlyeq \mathrm{C}$, $E{\downarrow_\pi}(x) = v$, and $x \preccurlyeq \alpha$, then $\mathrm{C} \vdash \alpha \xrightarrow{\pi}\triangleright \tau \wedge v \preccurlyeq \tau$.

Proof. Immediate from Lemma A.33. □

Lemma A.35. If $E \preccurlyeq \mathrm{C}$, $x \preccurlyeq \alpha$, $\varphi \preccurlyeq \tau_\varphi$, and $x \preceq_E \varphi \backslash E_0$, then $\alpha \preceq_\mathrm{C} \tau_\varphi \backslash \mathrm{C}'$ for some C' such that $E_0 \preccurlyeq \mathrm{C}'$.

Proof. By case analysis.

If $\varphi = \mathtt{int}$, then because $\varphi \preccurlyeq \tau_\varphi$ and by Definition A.7 we have that $\tau_\varphi = \mathtt{int}$. By Definition 3.3, the only case in which $\varphi = \mathtt{int}$ results in compatibility is when $E{\downarrow_\mathtt{int}}(x) = n$ and $n \in \mathbb{Z}$; in this case, $E_0 = []$. Because $x \preccurlyeq \alpha$, Lemma A.34 gives us that $\mathrm{C} \vdash \alpha \xrightarrow{\mathtt{int}}\triangleright \tau$ such that $n \preccurlyeq \tau$. Because of this and $n \in \mathbb{Z}$, we have by Definition A.7 that $\tau = \mathtt{int}$. By Definition 4.6, we have that $\alpha \preceq_\mathrm{C} \mathtt{int} \backslash \emptyset$. Because Definition A.7 gives us that $[] \preccurlyeq \emptyset$, this case is finished.

If $\varphi = \mathrm{lbl}\, x'$, then because $\varphi \preccurlyeq \tau_\varphi$ and by Definition A.7 we have that $\tau_\varphi = \mathrm{lbl}\, \alpha'$ such that $x' \preccurlyeq \alpha'$. By Definition 3.3, the only case in which $\varphi = \mathrm{lbl}\, x'$ results in compatibility is when $E{\downarrow_\mathrm{lbl}}(x) = \mathrm{lbl}\, x''$; in this case, $E_0 = [x' = x'']$. Because $x \preccurlyeq \alpha$, Lemma A.34 gives us that $\mathrm{C} \vdash \alpha \xrightarrow{\mathrm{lbl}}\triangleright \tau$ such that $\mathrm{lbl}\, x'' \preccurlyeq \tau$. By Definition A.7, this gives us that $\tau = \mathrm{lbl}\, \alpha''$ such that $x'' \preccurlyeq \alpha''$. By Definition 4.6, we have that $\alpha \preceq_\mathrm{C} \mathrm{lbl} \backslash \{\alpha'' <: \alpha'\}$. Because $x'' \preccurlyeq \alpha''$ and $x' \preccurlyeq \alpha'$, we have by Definition A.7 that $[x' = x''] \preccurlyeq \{\alpha'' <: \alpha'\}$; thus, this case is finished.

If $\varphi = \mathtt{any}$, then because $\varphi \preccurlyeq \tau_\varphi$ and by Definition A.7 we have that $\tau_\varphi = \mathtt{any}$. By Definition 3.3, we have that $E_0 = []$; by Definition 4.6, we have that $\mathrm{C}' = \emptyset$. Because Definition A.7 gives us that $[] \preccurlyeq \emptyset$, we are finished. □

Lemma A.36. If $E \preccurlyeq \mathrm{C}$, $x \preccurlyeq \alpha$, $\phi \preccurlyeq \tau_\phi$, and $x \preceq_E \phi \backslash E_0$, then $\alpha \preceq_\mathrm{C} \tau_\phi \backslash \mathrm{C}''$ for some C'' such that $E_0 \preccurlyeq \mathrm{C}''$.

Proof. By the grammar of patterns in Figure 3.1, we have that $\phi = x' : \varphi$ for some φ. Because $\phi \preccurlyeq \tau_\phi$, we have that $\tau_\phi = \alpha' \sim \tau_\varphi$ for some τ_φ such that $\varphi \preccurlyeq \tau_\varphi$ and $x' \preccurlyeq \alpha'$. By Definition 3.3, the only case in which compatibility holds over ϕ is when $x' \preceq_E \varphi \backslash E'_0$ for some E'_0; in this case, $E_0 = E'_0 \,\|[x' = x]$. By Lemma A.35, we have that $\alpha \preceq_\mathrm{C} \tau_\varphi \backslash \mathrm{C}'$ such that $E'_0 \preccurlyeq \mathrm{C}'$. By Definition 4.6, this gives us that $\alpha \preceq_\mathrm{C} \alpha' \sim \tau_\varphi \backslash \mathrm{C}' \cup \{\alpha <: \alpha'\}$.

It remains to show that $E'_0 \,\|[x' = x] \preccurlyeq \mathrm{C}' \cup \{\alpha <: \alpha'\}$. By Lemma A.20 it is sufficient to show that $E'_0 \preccurlyeq \mathrm{C}'$ (which we already know from above) and that $[x' = x] \preccurlyeq \{\alpha <: \alpha'\}$. By Definition A.7, this is equivalent to showing that $x \preccurlyeq \alpha$ (which is a premise of this lemma) and that $x' \preccurlyeq \alpha'$ (which we know from the above). □

Lemma A.37. If $E \preccurlyeq \mathrm{C}$, $x \preccurlyeq \alpha$, $\overrightarrow[n]{\phi_\square \texttt{->} e_\square} \preccurlyeq \vec{\tau}$, and $x \preceq_E \overrightarrow[n]{\phi_\square \texttt{->} e_\square} \backslash E_0; e'$, then there exist some α' and C' such that $\alpha \preceq_\mathrm{C} \vec{\tau} \backslash \alpha'; \mathrm{C}'$ and $E_0 \,\|\, e' \preccurlyeq \langle \alpha', \mathrm{C}' \rangle$.

Proof. By induction on n.

Let $\vec{\tau} = \overrightarrow[n]{\forall \boldsymbol{\alpha}_\square . \tau_{\phi_\square} \to \alpha''_\square \setminus \mathrm{C}''_\square}$. If $x \preceq_E \phi_n \texttt{->} e_n \backslash E_0$, then by Lemma A.36 we have that $\alpha \preceq_\mathrm{C} \tau_{\phi_n} \backslash \mathrm{C}'''$ such that $E_0 \preccurlyeq \mathrm{C}'''$. We have from $\overrightarrow[n]{\phi_\square \texttt{->} e_\square} \preccurlyeq \vec{\tau}$ and Definition A.7 that $e_n \preccurlyeq \langle \alpha''_n, \mathrm{C}''_n \rangle$. By Definition 5.7 we have that $\alpha \preceq_\mathrm{C} \vec{\tau} \backslash \alpha''_n; \mathrm{C}''_n \cup \mathrm{C}'''$; thus, $\alpha' = \alpha''_n$ and $\mathrm{C}' = \mathrm{C}''_n \cup \mathrm{C}'''$. Because $e_n \preccurlyeq \langle \alpha''_n, \mathrm{C}''_n \rangle$, we know that $e_n \preccurlyeq \mathrm{C}''_n$; by Lemma A.20, we have that $E_0 \,\|\, e_n \preccurlyeq \mathrm{C}''_n \cup \mathrm{C}'''$. Because E_0 was prepended (rather than appended) to e_n, the last variable in the list is still the last variable in e_n; thus, we have $E_0 \,\|\, e_n \preccurlyeq \langle \alpha''_n, \mathrm{C}''_n \cup \mathrm{C}''' \rangle$ and this case is finished.

Otherwise, we simply induct on this lemma. We know that induction is possible; if $n = 0$ then the statement that $x \preceq_E \overrightarrow[n]{\phi_\square \texttt{->} e_\square} \backslash E_0; e'$ is a contradiction. □

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

Lemma A.38. Suppose that $e \preccurlyeq \mathrm{C}$, that $\boldsymbol{x} \preccurlyeq \boldsymbol{\alpha}$, 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 $\textsc{Einst}(x', \boldsymbol{x}, \mathcal{C}_1) \preccurlyeq \textsc{Inst}(\alpha', \boldsymbol{\alpha}, \mathcal{C}_2)$.

Proof. For any x' in e, Lemma A.22 gives us that there is a corresponding α' in C such that $x' \preccurlyeq \alpha'$. By Definition A.2, we have that $x' \leftrightarrow \langle \iota, n, \dot{\mathcal{C}}_1 \rangle$ and $\alpha' = \langle \iota, n, \dot{\mathcal{C}}_2 \rangle$ such that $\dot{\mathcal{C}}_1 \preccurlyeq \dot{\mathcal{C}}_2$.

Suppose that $x' \in \boldsymbol{x}$ and $\dot{\mathcal{C}}_1 = \bullet$. In this case, $\textsc{Einst}(x', \boldsymbol{x}, \mathcal{C}_1) = x'$. Because $x' \preccurlyeq \alpha'$, we know that $\dot{\mathcal{C}}_2$ is $\bullet$. Because of this and $\boldsymbol{x} \preccurlyeq \boldsymbol{\alpha}$, we also know that $\alpha' \in \boldsymbol{\alpha}$. Thus, $\textsc{Inst}(\alpha', \boldsymbol{\alpha}, \mathcal{C}_2) = \alpha'$. Because $x' \preccurlyeq \alpha'$, this case is finished.

Otherwise, suppose that $\dot{\mathcal{C}}_1 \neq \bullet$. In that case, $\textsc{Einst}(x', \boldsymbol{x}, \mathcal{C}_1) = x'$. Because $x' \preccurlyeq \alpha'$, we also know that $\dot{\mathcal{C}}_2 \neq \bullet$; thus, $\textsc{Inst}(\alpha', \boldsymbol{\alpha}, \mathcal{C}_2) = \alpha'$. Thus this case is finished.

Finally, suppose that $\dot{\mathcal{C}}_1 = \bullet$ but $x' \notin \boldsymbol{x}$. In this case, $\textsc{Einst}(x', \boldsymbol{x}, \mathcal{C}_1) = \langle \iota, n, \mathcal{C}_1 \rangle$. Because $x' \preccurlyeq \alpha'$, we know that $\dot{\mathcal{C}}_2$ is $\bullet$. By Definition A.7, the fact that $\boldsymbol{x} \preccurlyeq \boldsymbol{\alpha}$ means that every $\alpha'' \in \boldsymbol{\alpha}$ has some $x'' \in \boldsymbol{x}$ such that $x'' \preccurlyeq \alpha''$. For variables without a contour (such as α'), this implies that $x'' \leftrightarrow \langle \iota', n', \bullet \rangle$ such that $\alpha'' = \langle \iota', n', \bullet \rangle$. Thus, because $\dot{\mathcal{C}}_1 = \dot{\mathcal{C}}_2 = \bullet$, $x' \in \boldsymbol{x}$ implies $\alpha' \in \boldsymbol{\alpha}$. Using this, we conclude that $\textsc{Inst}(\alpha', \boldsymbol{\alpha}, \mathcal{C}_2) = \langle \iota, n, \mathcal{C}_2 \rangle$. Because $\mathcal{C}_1 \preccurlyeq \mathcal{C}_2$ and by Definition A.7, we are finished. □

We now show that simulation is preserved through instantiation.

Lemma A.39. Suppose that $e \preccurlyeq \mathrm{C}$, that $\boldsymbol{x} \preccurlyeq \boldsymbol{\alpha}$, and that $\mathcal{C}_1 \preccurlyeq \mathcal{C}_2$. Then $\textsc{EInst}(e, \boldsymbol{x}, \mathcal{C}_1) \preccurlyeq \textsc{Inst}(\mathrm{C}, \boldsymbol{\alpha}, \mathcal{C}_2)$.

Proof. By induction on Definition A.7.

For variables, simulation is demonstrated by Lemma A.38. For all lower bound types other than functions and all upper bound types, patterns, constraints and contexts, simulation is parametric in variables and so is also demonstrated by Lemma A.38.

For function lower bound types, we must show that for any $v = \phi \texttt{->} e'$ appearing in e, there is a corresponding $\tau = \forall \boldsymbol{\alpha}'.\tau_\phi \to \alpha \setminus \mathrm{C}'$ appearing in C such that $\textsc{EInst}(v, \boldsymbol{x}, \mathcal{C}_1) \preccurlyeq \textsc{Inst}(\tau, \boldsymbol{\alpha}, \mathcal{C}_2)$. For any such v in e, Lemma A.22 gives us that such an τ exists and that $v \preccurlyeq \tau$. By Definition A.7, this means that $\phi \preccurlyeq \tau_\phi$, $e' \preccurlyeq \mathrm{C}'$, and that $\textsc{bv}(v) \preccurlyeq \boldsymbol{\alpha}$. Since each of these terms are smaller than v or τ accordingly, we induct on the definition of this lemma and so function lower bound types are also simulated.

By Definition A.7, we know that for each $x \texttt{=} v \in e$, there is a corresponding $\mathrm{c} \in \mathrm{C}$ such that $x \texttt{=} v \preccurlyeq \mathrm{c}$. Suppose that $\textsc{EInst}(x \texttt{=} v, \boldsymbol{x}, \mathcal{C}_1) = x' \texttt{=} v'$, $\textsc{Inst}(\mathrm{c}, \boldsymbol{\alpha}, \mathcal{C}_2) = \mathrm{c}'$, $\textsc{EInst}(e, \boldsymbol{x}, \mathcal{C}_1) = e'$, and $\textsc{Inst}(\mathrm{C}, \boldsymbol{\alpha}, \mathcal{C}_2) = \mathrm{C}'$. Because we have that simulation of each constraint is preserved by instantiation, we have that $x' \texttt{=} v' \preccurlyeq \mathrm{c}'$. Since this is true for each $x' \texttt{=} v'$ and c', we have that each $x' \texttt{=} v'$ in e' is simulated by some c' in C'; thus $\textsc{EInst}(E, \boldsymbol{x}, \mathcal{C}_1) \preccurlyeq \textsc{Inst}(\mathrm{C}, \boldsymbol{\alpha}, \mathcal{C}_2)$. □

Lemma A.40. If $e \preccurlyeq \mathrm{C}$ and $\mathcal{C}_1 \preccurlyeq \mathcal{C}_2$, then $\textsc{EFresh}(e, \mathcal{C}_1) \preccurlyeq \textsc{Inst}(\mathrm{C}, \mathcal{C}_2)$.

Proof. Immediate from Lemma A.39. □

Contour replacement also preserves simulation. We use a strategy similar to that above.

Lemma A.41. If $x \preccurlyeq \alpha$ then $x \preccurlyeq \textsc{Repl}(\alpha, \mathcal{C})$.

Proof. Let $\textsc{Repl}(\alpha, \mathcal{C}) = \alpha'$. Then, let $x \leftrightarrow \langle \iota_1, n_1, \dot{\mathcal{C}}_1 \rangle$ and $\alpha = \langle \iota_2, n_2, \dot{\mathcal{C}}_2 \rangle$ and $\alpha' = \langle \iota_3, n_3, \dot{\mathcal{C}}_3 \rangle$. Because $x \preccurlyeq \alpha$, we know that $\iota_1 = \iota_2$ and $n_1 = n_2$; by Definition 5.10, contour replacement is naturally homomorphic down to (but not including) contours and so we know that $\iota_2 = \iota_3$ and $n_2 = n_3$. It remains to show that $\dot{\mathcal{C}}_1 \preccurlyeq \dot{\mathcal{C}}_3$.

$x \preccurlyeq \alpha$ also gives us that $\dot{\mathcal{C}}_1 = \text{❁}$ iff $\dot{\mathcal{C}}_2 = \text{❁}$. If $\dot{\mathcal{C}}_1 = \text{❁}$, then $\dot{\mathcal{C}}_2 = \text{❁}$ and, by Definition 5.10, $\dot{\mathcal{C}}_3 = \text{❁}$; this case is finished.

If $\dot{\mathcal{C}}_1 \neq \text{❁}$ then $\dot{\mathcal{C}}_1 = \mathcal{C}_1$ and $\dot{\mathcal{C}}_2 = \mathcal{C}_2$. By Definition A.7, it suffices to show that $\mathcal{C}_1 \leq \mathcal{C}_3$ where $\mathcal{C}_3 = \dot{\mathcal{C}}_3$. If $\mathcal{C}_2 \not\leq \mathcal{C}$, then $\dot{\mathcal{C}}_3 = \mathcal{C}_2$ and so $\dot{\mathcal{C}}_1 \leq \dot{\mathcal{C}}_3$ and this case is finished.

Otherwise, $\mathcal{C}_2 \leq \mathcal{C}$. Then $\dot{\mathcal{C}}_3 = \mathcal{C}$ and it suffices to show that $\mathcal{C}_1 \leq \mathcal{C}$. By Lemma A.10, it suffices to show that $\mathcal{C}_1 \leq \mathcal{C}_2$ and that $\mathcal{C}_2 \leq \mathcal{C}$; the latter we have from this case. We prove $\mathcal{C}_1 \leq \mathcal{C}_2$ by Definition A.7 because $x \preccurlyeq \alpha$ and so $\mathcal{C}_1 \preccurlyeq \mathcal{C}_2$. □

Lemma A.42. If $E \preccurlyeq \mathrm{C}$ then $E \preccurlyeq \textsc{Repl}(\mathrm{C}, \mathcal{C})$.

Proof. Because contour replacement is a natural homomorphism in all cases not proven by Lemma A.41, this argument follows in parallel to that of Lemma A.39. □

A.8 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.43. $\mathcal{C} \leq \textsc{Collapse}(\mathcal{C})$

Proof. This is to show that $[\![\mathcal{C}]\!] \subseteq [\![\{\textsc{Collapse}(\mathcal{S}) \mid \mathcal{S} \in \mathcal{C}\}]\!]$. By Definition 5.10, 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 $\underbrace{(i, j)}_{m}$ 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.13, 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 $[\![\overrightarrow[n]{\mathcal{P}}]\!] \subseteq [\![[\underbrace{\bigcup \textsc{Sites}(\mathcal{P}_\square)}_{n}]]\!]$. 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.44. $\forall \mathrm{C}.\mathcal{C} \leq \textsc{Widen}(\mathcal{C}, \mathrm{C})$

Proof. For any C, we have that $\textsc{Widen}(\mathcal{C}, \mathrm{C}) = \mathcal{C}' \supseteq \mathcal{C}$ 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.45. If $\textsc{Widen}(\mathcal{C}, \mathrm{C}) = \mathcal{C}'$, then $\forall \mathcal{C}'' \in (\!|\mathrm{C}|\!).\mathcal{C}'' \between \mathcal{C}' \implies \mathcal{C}'' \leq \mathcal{C}'$.

Proof. By definition, $\mathcal{C}' = \mathcal{C} \cup (\bigcup\{\mathcal{C}'' \mid \mathcal{C}'' \in (\!|\mathrm{C}|\!) \wedge \mathcal{C} \between \mathcal{C}''\})$. We therefore have for any $\mathcal{C}''$ in $(\!|\mathrm{C}|\!)$ that $\mathcal{C}'' \subseteq \mathcal{C}'$. By Definition 5.10, we thus have that $[\![\mathcal{C}'']\!] \subseteq [\![\mathcal{C}']\!]$ and so $\mathcal{C}'' \leq \mathcal{C}'$. □

Lemma A.46. If $\mathrm{C} \vdash \alpha \xrightarrow{\pi}\triangleright \vec{\tau}$ for some well-formed C, then $(\!|\vec{\tau}|\!) \subseteq (\!|\alpha|\!) \cup (\!|\mathrm{C}|\!)$.

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.47. If $\alpha \preceq_{\mathrm{C}} \tau_\phi \backslash \mathrm{C}'$, then $(\!|\mathrm{C}'|\!) \subseteq (\!|\mathrm{C}|\!) \cup (\!|\tau_\phi|\!) \cup (\!|\alpha|\!)$.

Proof. By inspection. The only cases in which constraints are added to the constraint set C′ in the compatibility relation are the cases for $\alpha_1' \sim \tau_\varphi$ and $\mathrm{lbl}\,\alpha_2'$. In the prior case, the added constraint is comprised of α and α_1'; in the latter, it is comprised of α_2' and some α_3' which resulted from projection. By Lemma A.46, we know that $(\!|\alpha_3'|\!) \subseteq (\!|\alpha|\!) \cup (\!|\mathrm{C}|\!)$. □

Lemma A.48. If $\textsc{Inst}(\mathrm{C}, \boldsymbol{\alpha}, \mathcal{C}) = \mathrm{C}'$, then $(\!|\mathrm{C}'|\!) \subseteq (\!|\mathrm{C}|\!) \cup \{\mathcal{C}\}$.

Proof. Let $\alpha' = \textsc{Inst}(\alpha, \boldsymbol{\alpha}, \mathcal{C})$. We first prove that $(\!|\alpha'|\!) \subseteq (\!|\alpha|\!) \cup \{\mathcal{C}\}$ by case analysis. If the contour of α is $\bullet$ and $\alpha \notin \boldsymbol{\alpha}$, then the contour of α' is $\mathcal{C}$ and so $(\!|\alpha'|\!) = \{\mathcal{C}\}$. If the contour of α is $\bullet$ but $\alpha \in \boldsymbol{\alpha}$, then the contour of α' is $\bullet$ and so $(\!|\alpha'|\!) = \emptyset$. Finally, if the contour of α is $\mathcal{C}'$, then the contour of α' is also $\mathcal{C}'$ and so $(\!|\alpha'|\!) = \mathcal{C}' = (\!|\alpha|\!)$. Thus, $(\!|\alpha'|\!) \subseteq (\!|\alpha|\!) \cup \{\mathcal{C}\}$.

Because contour instantiation is largely naturally homomorphic, the remainder of this argument proceeds in parallel to Lemma A.39. □

Lemma A.49. If $\textsc{Repl}(\mathrm{C}, \mathcal{C}) = \mathrm{C}'$, then $(\!|\mathrm{C}'|\!) \subseteq \{\mathcal{C}\} \cup \{\mathcal{C}' \mid \mathcal{C}' \in (\!|\mathrm{C}|\!) \wedge \mathcal{C}' \not\leq \mathcal{C}\}$.

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 $\bullet$ then contour replacement is a natural homomorphism and so $\alpha' = \alpha$; thus, $(\!|\alpha'|\!) = \emptyset \subseteq \{\mathcal{C}\} \cup \emptyset$. 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 $(\!|\alpha'|\!) = \{\mathcal{C}\} \subseteq \{\mathcal{C}\} \cup \emptyset$. 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.39. □

A.9 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.50 (Preservation)**.** If $e \preccurlyeq \mathrm{C}$, C is well-formed, and $e \longrightarrow^1 e'$, then there exists some C′ such that $\mathrm{C} \xrightarrow{\mathrm{CL}}{}^1 \mathrm{C}'$, $e' \preccurlyeq \mathrm{C}'$, and C′ is well-formed.

We now give a proof of Lemma A.50.

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.

Integer Addition: If $e = E \,\|\, [x = x_1 + x_2] \,\|\, e_{\text{REST}}$, then $e' = E \,\|\, [x = n] \,\|\, e_{\text{REST}}$ 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 $x = x_1 + x_2$ appears in e and because $e \preccurlyeq \mathrm{C}$, Lemma A.22 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 $E{\downarrow_{\texttt{int}}}(x_1) = n_1$ and $x_1 \preccurlyeq \alpha_1$, Lemma A.33 gives us that $\mathrm{C} \vdash \alpha_1 \xrightarrow{\texttt{int}}\triangleright \tau_1$ 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 $\mathrm{C} \xrightarrow{\mathrm{CL}}{}^1 \mathrm{C}'$ where $\mathrm{C}' = \mathrm{C} \cup \{\texttt{int} <: \alpha\}$. It remains to show that $e' \preccurlyeq \mathrm{C}'$ and that C′ is well-formed. We first observe that by Lemma A.19 we have $E \,\|\, e_{\text{REST}} \preccurlyeq \mathrm{C}$. By Lemma A.20 and $[x = n] \preccurlyeq \{\texttt{int} <: \alpha\}$ (which is immediate from the above and Definition A.7), we have that $e' \preccurlyeq \mathrm{C}'$. To show that C′ is well-formed, we observe that the only contour appearing in $\{\texttt{int} <: \alpha\}$ is that in α, which is a type variable appearing in C; thus $(\!|\{\texttt{int} <: \alpha\}|\!) \subseteq (\!|\mathrm{C}|\!)$. By Lemma A.18, we therefore have that C′ is well-formed.

True Integer Equality: If $e = E \,\|\, [x = x_1 \texttt{ == } x_2] \,\|\, e_{\text{REST}}$, then $e' = E \,\|\, [x' = \texttt{()}, x = \texttt{`True } x'] \,\|\, e$ for a fresh x' chosen such that $x' \leftrightarrow \langle \iota, 0, \dot{\mathcal{C}} \rangle$ where $x \leftrightarrow \langle \iota, n, \dot{\mathcal{C}} \rangle$. We also have that $E{\downarrow_{\texttt{int}}}(x_1) = n$ and that $E{\downarrow_{\texttt{int}}}(x_2) = n$ for some $n \in \mathbb{Z}$.

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 $e \preccurlyeq \mathrm{C}$, Lemma A.22 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.33 gives us that $\mathrm{C} \vdash \alpha_1 \xrightarrow{\texttt{int}}\triangleright \tau_1$ 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 $\mathrm{C} \xrightarrow{\mathrm{CL}}{}^1 \mathrm{C}'$ where $\mathrm{C}' = \mathrm{C} \cup \{\texttt{`True } \alpha' <: \alpha, \texttt{`False } \alpha' <: \alpha, \texttt{()} <: \alpha'\}$ where $\alpha' = \textsc{etv}(\alpha)$. Because $x \preccurlyeq \alpha$, we know that $\alpha = \langle \iota, n, \dot{\mathcal{C}}' \rangle$ where $\dot{\mathcal{C}}' = \bullet$ if $\dot{\mathcal{C}} = \bullet$ and $\dot{\mathcal{C}} \leq \dot{\mathcal{C}}'$ otherwise. By Definition 5.6, we have that $\alpha' = \langle \iota, 0, \dot{\mathcal{C}}' \rangle$; thus, $x' \preccurlyeq \alpha'$. By Definition A.7 and the above, we have that $[x' = \texttt{()}, x = \texttt{`True } x'] \preccurlyeq \{\texttt{`True } \alpha' <: \alpha, \texttt{()} <: \alpha'\}$; by Lemma A.20, we have that $[x' = \texttt{()}, x = \texttt{`True } x'] \preccurlyeq$

$\{\texttt{`True}\ \alpha' <: \alpha, \texttt{`False}\ \alpha' <: \alpha, () <: \alpha'\}$. By Lemma A.20, we therefore have that $e' \preccurlyeq \mathrm{C}'$. It remains to show that C' is well-formed; this is demonstrated by Lemma A.18 as above.

A similar argument is made when $e' = E \,\|\, [x' = (), x = \texttt{`False}\ x'] \,\|\, e$.

Application: If $e = E \,\|\, [x = x_1\ x_2] \,\|\, e_{\text{REST}}$, then $e' = E \,\|\, \textsc{Efresh}(x, E_0 \,\|\, e''' \,\|\, [x' = r]) \,\|\, [x = \textsc{Efresh}(x, x')] \,\|\, e_{\text{REST}}$ where $E{\downarrow}^*_{\texttt{fun}}(x_1) = \overset{n}{\overrightarrow{\phi \texttt{ -> } e''}}$ and $x_2 \preceq_E \overset{n}{\overrightarrow{\phi \texttt{ -> } e''}} \backslash E_0; e''' \,\|\, [x' = r]$.

We now show that the Application closure rule applies in this case. Because $x = x_1\ x_2$ appears in e and because $e \preccurlyeq \mathrm{C}$, Lemma A.22 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 $E{\downarrow}^*_{\texttt{fun}}(x_1) = \overset{n}{\overrightarrow{\phi \texttt{ -> } e''}}$ and $x_1 \preccurlyeq \alpha_1$ and by Lemma A.33, we have that there exists some $\vec{\tau}$ such that $\overset{n}{\overrightarrow{\phi \texttt{ -> } e''}} \preccurlyeq \vec{\tau}$ and $\mathrm{C} \vdash \alpha_1 \xrightarrow{\texttt{fun}\triangleright} \vec{\tau}$. Because $x_2 \preceq_E \overset{n}{\overrightarrow{\phi \texttt{ -> } e''}} \backslash E_0; e''' \,\|\, [x' = r]$, the previous, and by Lemma A.37, we have that there exist some α_{LAST} and C_{NEW} such that $\alpha_2 \preceq_{\mathrm{C}} \vec{\tau} \backslash \alpha_{\text{LAST}}; \mathrm{C}_{\text{NEW}}$ such that $E_0 \,\|\, e''' \preccurlyeq \langle \alpha_{\text{LAST}}, \mathrm{C}_{\text{NEW}} \rangle$. Because CNEW and INST are total functions, we have shown that the Application closure rule applies.

Let $e_{\text{NEW}} = E_0 \,\|\, e'''$ and let $x_{\text{LAST}} = r_{\text{LAST}}$ be the last element in e_{NEW}. Then $e_{\text{NEW}} \preccurlyeq \mathrm{C}_{\text{NEW}}$ and $x_{\text{LAST}} \preccurlyeq \alpha_{\text{LAST}}$. Because $x \preccurlyeq \alpha$, we have by Definition A.7 that $[x = x_{\text{LAST}}] \preccurlyeq \{\alpha_{\text{LAST}} <: \alpha\}$.

Let $\mathcal{C} = \textsc{CNew}(\alpha_2, \mathrm{C})$, let $e'_{\text{NEW}} = \textsc{Efresh}(e_{\text{NEW}}, \mathcal{C})$, and let $\mathrm{C}'_{\text{NEW}} = \textsc{Inst}(\mathrm{C}_{\text{NEW}}, \mathcal{C})$. By Lemma A.40, we have $e'_{\text{NEW}} \preccurlyeq \mathrm{C}'_{\text{NEW}}$. Let $\textsc{Efresh}(x_{\text{LAST}}, \mathcal{C}) = x'_{\text{LAST}}$. Let $\mathrm{C}'' = \textsc{Inst}(\{\alpha_{\text{LAST}} <: \alpha\}, \mathcal{C})$. By Definition 5.10, we have that $\mathrm{C}'' = \{\textsc{Inst}(\alpha_{\text{LAST}}, \emptyset, \mathcal{C}) <: \textsc{Inst}(\alpha, \emptyset, \mathcal{C})\}$. The contour of α is not $\maltese$ as it appears at top level in C ; thus, $\mathrm{C}'' = \{\alpha'_{\text{LAST}} <: \alpha\}$ where $\alpha'_{\text{LAST}} = \textsc{Inst}(\alpha_{\text{LAST}}, \emptyset, \mathcal{C})$. By Lemma A.40, we thus have that $[x = x'_{\text{LAST}}] \preccurlyeq \mathrm{C}''$; thus by Lemma A.20 we have that $e'_{\text{NEW}} \,\|\, [x = x'_{\text{LAST}}] \preccurlyeq \mathrm{C}'_{\text{NEW}} \cup \mathrm{C}''$. 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 $\mathrm{C}_{\text{NEW}} \cup \mathrm{C}'' = \mathrm{C}'''$.

By Lemma A.19, we have that $E \,\|\, e_{\text{REST}} \preccurlyeq \mathrm{C}$; thus by Lemma A.20 we have that $E \,\|\, \textsc{Efresh}(x, E_0 \,\|\, e''' \,\|\, [x' = r]) \,\|\, [x = \textsc{Efresh}(x, x')] \,\|\, e_{\text{REST}} \preccurlyeq \mathrm{C} \cup \mathrm{C}'''$. Finally, we let $\mathrm{C}' = \textsc{Repl}(\mathrm{C} \cup \mathrm{C}''', \mathcal{C})$ and by Lemma A.42, we have $E \,\|\, \textsc{Efresh}(x, E_0 \,\|\, e''' \,\|\, [x' = r]) \,\|\, [x = \textsc{Efresh}(x, x')] \,\|\, e_{\text{REST}} \preccurlyeq \mathrm{C}'$.

It remains to show that $\textsc{Repl}(\mathrm{C} \cup \mathrm{C}'', \mathcal{C})$ 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.46 that $(\!|\vec{\tau}|\!) \subseteq (\!|\mathrm{C}|\!)$. From this, because α_2 appears in C and by Lemma A.47, we have that $(\!|\mathrm{C}_{\text{NEW}}|\!) \subseteq (\!|\mathrm{C}|\!)$.

By Lemma A.48, we have that $(\!|\mathrm{C}'_{\text{NEW}}|\!) \subseteq (\!|\mathrm{C}|\!) \cup \{\mathcal{C}\}$. We also have that $(\!|\mathrm{C}''|\!) \subseteq (\!|\mathrm{C}|\!) \cup \{\mathcal{C}\}$. This gives us $(\!|\mathrm{C}'_{\text{NEW}} \cup \mathrm{C}''|\!) \subseteq (\!|\mathrm{C}|\!) \cup \{\mathcal{C}\}$ by Lemma A.11, which is equivalent to $(\!|\mathrm{C}'''|\!) \subseteq (\!|\mathrm{C}|\!) \cup \{\mathcal{C}\}$. Again by Lemma A.11, we have $(\!|\mathrm{C} \cup \mathrm{C}'''|\!) \subseteq (\!|\mathrm{C}|\!) \cup \{\mathcal{C}\}$, which is equivalent to $(\!|\mathrm{C}'|\!) \subseteq (\!|\mathrm{C}|\!) \cup \{\mathcal{C}\}$. By Lemma A.49, we have that $(\!|\mathrm{C}'|\!) \subseteq \{\mathcal{C}\} \cup \{\mathcal{C}' \mid \mathcal{C}' \in (\!|\mathrm{C} \cup \mathrm{C}'''|\!) \wedge \mathcal{C}' \not\leq \mathcal{C}\}$. Because $(\!|\mathrm{C} \cup \mathrm{C}'''|\!) \subseteq (\!|\mathrm{C}|\!) \cup \{\mathcal{C}\}$, this can be weakened to $(\!|\mathrm{C}'|\!) \subseteq \{\mathcal{C}\} \cup \{\mathcal{C}' \mid \mathcal{C}' \in (\!|\mathrm{C}|\!) \cup \{\mathcal{C}\} \wedge \mathcal{C}' \not\leq \mathcal{C}\}$; since $\mathcal{C} \leq \mathcal{C}$, this simplifies to $(\!|\mathrm{C}'|\!) \subseteq \{\mathcal{C}\} \cup \{\mathcal{C}' \mid \mathcal{C}' \in (\!|\mathrm{C}|\!) \wedge \mathcal{C}' \not\leq \mathcal{C}\}$. For notational convenience, let $\boldsymbol{\mathcal{C}} = \{\mathcal{C}' \mid \mathcal{C}' \in (\!|\mathrm{C}|\!) \wedge \mathcal{C}' \not\leq \mathcal{C}\}$. It remains to show that $(\!|\mathrm{C}'|\!)$ is well-formed; by Lemma A.18 it is sufficient to show that $\{\mathcal{C}\} \cup \boldsymbol{\mathcal{C}}$ is well-formed.

Since $\boldsymbol{\mathcal{C}}$ is a subset of $(\!|\mathrm{C}|\!)$ and C is well-formed, it is sufficient to show that $\forall \mathcal{C}' \in \boldsymbol{\mathcal{C}}.\mathcal{C}' \not\between \mathcal{C}$. By Lemma A.45, we have that for every $\mathcal{C}'$ in $(\!|\mathrm{C}|\!)$, we have either $\mathcal{C}' \not\between \mathcal{C}$ or $\mathcal{C}' \leq \mathcal{C}$. By definition, $\boldsymbol{\mathcal{C}}$ contains only those $\mathcal{C}'$ such that $\mathcal{C}' \not\leq \mathcal{C}$; thus, we know that $\forall \mathcal{C}' \in \boldsymbol{\mathcal{C}}.\mathcal{C}' \not\between \mathcal{C}$. Therefore, $\boldsymbol{\mathcal{C}} \cup \{\mathcal{C}\}$ 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.51. If $e \preccurlyeq \mathrm{C}$, 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 3.5 which lead to a stuck e. In each case, we show that the corresponding constraint set is inconsistent.

If r is of the form $x_1 \,\mathrm{op}\, x_2$, we observe that e is only stuck when $E{\downarrow}_{\texttt{int}}(x_i)$ is undefined for $i = 1$ or $i = 2$. By Definition 3.2, this implies that $E{\downarrow}^*_{\texttt{int}}(x_i) = []$. Because $[x = x_1 \,\mathrm{op}\, x_2]$ is in e, Lemma A.22 gives us that $\{\alpha_1 \,\mathrm{op}\, \alpha_2 <: \alpha\}$ for $x_1 \preccurlyeq \alpha_1$, $x_2 \preccurlyeq \alpha_2$, and $x \preccurlyeq \alpha$. Because $x_i \preccurlyeq \alpha_i$, Lemma A.33 gives us that $\mathrm{C} \vdash \alpha_i \xrightarrow{\texttt{int}\triangleright} []$. The presence of the constraint $\alpha_1 + \alpha_2 <: \alpha$ in C such that $\mathrm{C} \vdash \alpha_i \xrightarrow{\texttt{int}\triangleright} []$ is inconsistent by Definition 4.8.

Otherwise, r is of the form $x_1\ x_2$. Let $E{\downarrow}^*_{\texttt{fun}}(x_1) = \overset{n}{\overrightarrow{\phi_\square \texttt{ -> } e'_\square}}$. In this case, e is only stuck if there exists no E_0 and e'' such that $x_2 \preceq_E \overset{n}{\overrightarrow{\phi_\square \texttt{ -> } e'_\square}} \backslash E_0; e''$. Inspection of Definition 3.4 reveals that this is only the case if for all $1 \leq j \leq n$ there is no E_0 such that $x_2 \preceq_E \phi_j \backslash E_0$. Inspection of Definition 3.3 reveals that, for each such ϕ_j, this is only the case under two conditions: either (1) $\phi_j = x_3 : \texttt{int}$ and $E{\downarrow}_{\texttt{int}}(x_2)$ is undefined, or (2) $\phi_j = x_3 : \mathrm{lbl}\, x_4$ and $E{\downarrow}_{\mathrm{lbl}}(x_2)$ is undefined. Thus, one of these two conditions must be true for each j in $\{1..n\}$.

Because $[x = x_1\ x_2]$ is in e, Lemma A.22 gives us that $\alpha_1\ \alpha_2 <: \alpha$ exists in C such that $x_1 \preccurlyeq \alpha_1$, $x_2 \preccurlyeq \alpha_2$, and $x \preccurlyeq \alpha$. Because $x_1 \preccurlyeq \alpha_1$, Lemma A.33 gives us that $\mathrm{C} \vdash \alpha_1 \xrightarrow{\texttt{fun}} \overbrace{\forall \boldsymbol{\alpha}_{\square}.\tau\phi_{\square} \to \alpha'_{\square}}^{n} \setminus \mathrm{C}_{\square}$ and, among other things, that $\phi_j \preccurlyeq \tau\phi_j$ for each j in $\{1..n\}$.

For each j in $\{1..n\}$, we now show that $\alpha_2 \preceq_{\mathrm{C}} \tau\phi_j \backslash \text{❁}$. For such a j, one of the two conditions above holds. If $\phi_j = x_3 : \texttt{int}$, then $\phi_j \preccurlyeq \tau\phi_j$ gives us that, for some α_3 such that $x_3 \preccurlyeq \alpha_3$, $\tau\phi_j = \alpha_3 \sim \texttt{int}$. We also have that $E{\downarrow_{\texttt{int}}}(x_2)$ is undefined, which by Definition 3.2 gives us that $E{\downarrow^*_{\texttt{int}}}(x_2) = []$. Because $x_2 \preccurlyeq \alpha_2$, Lemma A.33 gives us that $\mathrm{C} \vdash \alpha_2 \xrightarrow{\texttt{int}} []$; as a result, $\alpha_2 \preceq_{\mathrm{C}} \tau\phi_j \backslash \text{❁}$.

Otherwise, $\phi_j = x_3 : \mathrm{lbl}\, x_4$; in this case, $\phi_j \preccurlyeq \tau\phi_j$ gives us that, for some α_3 and α_4 such that $x_3 \preccurlyeq \alpha_3$ and $x_4 \preccurlyeq \alpha_4$, $\tau\phi_j = \alpha_3 \sim \mathrm{lbl}\, \alpha_4$. We also have that $E{\downarrow_{\mathrm{lbl}}}(x_2)$ is undefined, which by Definition 3.2 gives us that $E{\downarrow^*_{\mathrm{lbl}}}(x_2) = []$. Because $x_2 \preccurlyeq \alpha_2$, Lemma A.33 gives us that $\mathrm{C} \vdash \alpha_2 \xrightarrow{\mathrm{lbl}} []$; as a result, $\alpha_2 \preceq_{\mathrm{C}} \tau\phi_j \backslash \text{❁}$.

We have thus shown that for all j in $\{1..n\}$, $\alpha_2 \preceq_{\mathrm{C}} \tau\phi_j \backslash \text{❁}$. Equivalently by notational sugar, we have shown that $\alpha_2 \not\preceq_{\mathrm{C}} \vec{\tau}$ given $\mathrm{C} \vdash \alpha_1 \xrightarrow{\texttt{fun}} \vec{\tau}$ for a constraint $\alpha_1\ \alpha_2 <: \alpha$ in C; by Definition 4.8, C is inconsistent. □

A.10 Proof of Soundness

We now prove Theorem 5.1. 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 e with unique variable bindings, we have by Lemma A.31 that, if $[\![e]\!]^{\emptyset}_{\mathrm{E}} = \langle \alpha, \mathrm{C} \rangle$, then C is well-formed and $e \preccurlyeq \langle \alpha, \mathrm{C} \rangle$; we also have that every type variable in C has the non-contour ❁. As a result, we have that $(\!|\mathrm{C}|\!) = \emptyset$. By Definition A.7, we have that $e \preccurlyeq \mathrm{C}$. Let $e' = \textsc{Efresh}(e, \{[]\})$ and let $\mathrm{C}' = \textsc{Inst}(\mathrm{C}, \{[]\})$; by Lemma A.40, we have that $e' \preccurlyeq \mathrm{C}'$. Observe that, by Definition A.5, e' is α-equivalent to e.

We next induct on the length of constraint closure. Let $e_0 = e'$ and let $\mathrm{C}_0 = \mathrm{C}'$. Further, let $e_0 \longrightarrow^1 \ldots \longrightarrow^1 e_n$ for some $n \geq 0$. We start with the base case that $e_0 \preccurlyeq \mathrm{C}_0$ and that C_0 is well-formed. We use Lemma A.50 to prove our inductive step: if $e_i \preccurlyeq \mathrm{C}_i$ and C_i is well-formed, then there exists some $\mathrm{C}_i \xrightarrow{\mathrm{CL}} \mathrm{C}_{i+1}$ such that $e_{i+1} \preccurlyeq \mathrm{C}_{i+1}$ and C_{i+1} is well-formed. We therefore have by induction that $e_n \preccurlyeq \mathrm{C}_n$ 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.51, we have that C_n is inconsistent. By Definition 5.11, e does *not* typecheck. □

B. MicroBang Termination

We show here that the MicroBang typechecking definition presented in Section 5 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, $[\![e]\!]^{\emptyset}_{\mathrm{E}}$ is a computable function.

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

Lemma B.2. Relation $\tau\ {}^{\mathrm{C}}\!<:^*\ \alpha$ is decidable.

Proof. By enumeration of the possible chains $\tau <: \alpha_0, \ldots, \alpha_{n-1} <: \alpha$. Each cyclic chain (a chain where the same type variable occurs more than once) has the same lower bounds as the acyclic chain obtained by pruning the cycle, and so it suffices to consider acyclic chains only. Because there are is a fixed, finite number of constraints in C, there is a fixed number of acyclic chains which means the fixed set of all possible lower bounds can be computed and the relation holds iff τ is in that set. □

The decidability of the projection relation and computability of a function which finds all $\vec{\tau}$ 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 [18], 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 Definition 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.3 (Nullable)**.** $\epsilon_\pi(\alpha)$ if and only if either (1) there exists a non-onion τ such that $\tau\ {}^{\mathrm{C}}\!<:^*\ \alpha$ and $\tau \not\sqsubseteq \pi$ or (2) $\alpha_1\ \&\ \alpha_2\ {}^{\mathrm{C}}\!<:^*\ \alpha$ and inductively both $\epsilon_\pi(\alpha_1)$ and $\epsilon_\pi(\alpha_2)$ hold.

Lemma B.4. Relation $\epsilon_\pi(\alpha)$ is decidable.

Proof. By Lemma B.2, we have that concretization is decidable. There are finitely many & constraints in C and it suffices never to visit the same & constraint twice by analogy with the cycle pruning in Lemma B.2. □

Definition B.5 (Modified Projection)**.**

$$
\begin{array}{lll}
\mathrm{C} \vdash_0^\epsilon \alpha' \xrightarrow{\pi\triangleright} [\tau] & \text{if} & \tau \overset{\mathrm{C}}{<:^*} \alpha', \tau \sqsubseteq \pi \\
\mathrm{C} \vdash_0^\epsilon \alpha' \xrightarrow{\pi\triangleright} \vec{\tau}_1 \,\|\, \vec{\tau}_2 & \text{if} & \alpha_1 \,\&\, \alpha_2 \overset{\mathrm{C}}{<:^*} \alpha', \\
 & & \mathrm{C} \vdash_0^\epsilon \alpha_1 \xrightarrow{\pi\triangleright} \vec{\tau}_1, \mathrm{C} \vdash_0^\epsilon \alpha_2 \xrightarrow{\pi\triangleright} \vec{\tau}_2 \\
\mathrm{C} \vdash_0^\epsilon \alpha' \xrightarrow{\pi\triangleright} \vec{\tau}_2 & \text{if} & \alpha_1 \,\&\, \alpha_2 \overset{\mathrm{C}}{<:^*} \alpha', \epsilon_\pi(\alpha_1), \\
 & & \mathrm{C} \vdash_0^\epsilon \alpha_2 \xrightarrow{\pi\triangleright} \vec{\tau}_2 \\
\mathrm{C} \vdash_0^\epsilon \alpha' \xrightarrow{\pi\triangleright} \vec{\tau}_1 & \text{if} & \alpha_1 \,\&\, \alpha_2 \overset{\mathrm{C}}{<:^*} \alpha', \epsilon_\pi(\alpha_2), \\
 & & \mathrm{C} \vdash_0^\epsilon \alpha_1 \xrightarrow{\pi\triangleright} \vec{\tau}_1 \\
\mathrm{C} \vdash^\epsilon \alpha \xrightarrow{\pi\triangleright} \vec{\tau} & \text{if} & \mathrm{C} \vdash_0^\epsilon \alpha \xrightarrow{\pi\triangleright} \vec{\tau} \text{ or } \epsilon_\pi(\alpha) \text{ and } \vec{\tau} = []
\end{array}
$$

Lemma B.6. Relation $\mathrm{C} \vdash \alpha \xrightarrow{\pi\triangleright} \vec{\tau}$ is decidable.

Proof. We first show that the modified projection relation $\mathrm{C} \vdash^\epsilon \alpha \xrightarrow{\pi\triangleright} \vec{\tau}$ at the bottom of Definition B.5 is equivalent to the original projection relation $\mathrm{C} \vdash \alpha \xrightarrow{\pi\triangleright} \vec{\tau}$ in Definition 4.4. The modified definition $\mathrm{C} \vdash_0^\epsilon \alpha \xrightarrow{\pi\triangleright} \vec{\tau}$ at the top of Definition B.5 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 $\mathrm{C} \vdash \alpha \xrightarrow{\pi\triangleright} \vec{\tau}$ / $\mathrm{C} \vdash_0^\epsilon \alpha \xrightarrow{\pi\triangleright} \vec{\tau}$ differ is where the [] clause is the top-level result, and $\mathrm{C} \vdash^\epsilon \alpha \xrightarrow{\pi\triangleright} \vec{\tau}$ explicitly aligns that top-level behavior.

The modified relation $\mathrm{C} \vdash^\epsilon \alpha \xrightarrow{\pi\triangleright} \vec{\tau}$ is now shown to be decidable, and by the above equivalence, the original relation is decidable. Observe that the list $\vec{\tau}$ 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.4. At the leaf case, we rely on concretization and projection matching; the latter is trivially decidable and the prior was shown to be decidable via Lemma B.2. □

Now we show that the set of all possible $\vec{\tau}$ 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. DEDUP is a function which removes all duplicate types in a list of types, preserving the rightmost occurrence in the list only. Formally:

Definition B.7 (Deduplication)**.**

$$
\begin{array}{rcl}
\textsc{Dedup}(\vec{\tau}) & = & \textsc{Dedup}(\emptyset, \vec{\tau}) \\
\textsc{Dedup}(\overline{\vec{\tau}}, []) & = & [] \\
\textsc{Dedup}(\overline{\vec{\tau}}, \vec{\tau} \,\|\, [\tau']) & = & \begin{cases} \textsc{Dedup}(\overline{\vec{\tau}}, \vec{\tau}) \text{ when } \tau' \in \overline{\vec{\tau}} \\ \textsc{Dedup}(\overline{\vec{\tau}} \cup \{\tau'\}, \vec{\tau} \,\|\, [\tau']) \text{ otherwise} \end{cases}
\end{array}
$$

Lemma B.8. There is a computable function which takes a C, π, and α as argument and returns a set $\overline{\vec{\tau}}$ such that for all $\vec{\tau}'$ we have $\textsc{Dedup}(\vec{\tau}') \in \overline{\vec{\tau}}$ if and only if $\mathrm{C} \vdash \alpha \xrightarrow{\pi\triangleright} \vec{\tau}'$.

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.

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.

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 conservatiely 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.9. The function which given some C, α, and π returns the set of all τ in C such that $\mathrm{C} \vdash \alpha \xrightarrow{\pi\triangleright} \tau$ is a computable function.

Proof. By Lemma B.8, 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.7), the set of rightmost types from the lists is equal to the set of rightmost types from the lists produced by projection. □

Lemma B.10. Relation $\alpha \preceq_\mathrm{C} \tau\phi \backslash \dot{\mathrm{C}}$ is decidable.

Proof. Inspection of Definition 4.6 reveals that compatibility is syntax-directed with leaf cases being either axiomatic or relying on single projection. By Lemma B.9, we have that single projection is decidable. □

Lemma B.11. Relation $\alpha \preceq_\mathrm{C} \vec{\tau} \backslash \alpha'; \mathrm{C}'$ is decidable.

Proof. By induction on the length of $\vec{\tau}$. Inspection of Definition 4.7 reveals that the truth of this term is either axiomatic (when the length of $\vec{\tau}$ is zero), reliant upon compatibility (shown to be decidable in Lemma B.10), or defined in terms of a smaller list. □

Lemma B.12. 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.13. The contour extraction ($(\!|\cdot|\!)$), instantiation (INST), and replacement (REPL) are decidable.

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.12). The leaf cases of extraction are constant. □

Lemma B.14. The functions used in contour creation – CNEW, COLLAPSE, and WIDEN – are decidable.

Proof. CNEW is trivially decidable. $\textsc{Collapse}(\mathcal{C})$ is trivially decidable if each $\textsc{Collapse}(\mathcal{S})$ is decidable 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 $\mathcal{S}$. $\textsc{Widen}(\mathcal{C}, \mathrm{C})$ is decidable given $\mathcal{C}$ and C because contour extraction and overlap are known to be decidable from Lemmas B.12 and B.13. □

Lemma B.15. Given C and C′, $\mathrm{C} \xrightarrow{\text{CL}}{}^{1} \mathrm{C}'$ is decidable.

Proof. We proceed by showing that there is a computable function from C to $\overbrace{\mathrm{C}''}^{m}$ such that $\mathrm{C} \xrightarrow{\text{CL}}{}^{1} \mathrm{C}''_i$ for all $i \in \{1..m\}$ and that there exists no other C‴ such that $\mathrm{C} \xrightarrow{\text{CL}}{}^{1} \mathrm{C}'''$; 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 by one of the above lemmas. For application, we use Lemma B.8 to produce the set of possible $\vec{\tau}$ 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 5.7. Because each rule has finitely many preconditions, this process is computable. Thus we can compute the set of constraints $\overbrace{\mathrm{C}''}^{m}$; because each rule only adds finitely many constraints, each C''_i is finite. This relation is then determined simply by $\mathrm{C}' \in \overbrace{\mathrm{C}''}^{m}$. □

Lemma B.16. It is decidable whether a given C is inconsistent.

Proof. By inspection of Definition 4.8, 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 from the above lemmas. □

We now prove Theorem 5.2.

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, computing $[\![e]\!]^{\emptyset}_{\mathrm{E}} = \langle \alpha, \mathrm{C} \rangle$ is decidable from Lemma B.1.

Clearly a fixed set of identifiers ι exist in e (and thus in C) and no type system rule introduces new identifiers. Additionally, new contours are only created by CNEW. 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. Because the initial expression e is fixed, there is also a fixed number of variable indices that could be used in a given closure sequence. 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 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. The number of possible patterns τ_ϕ is also fixed because new constraints added in closure contain no new patterns. 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 $\mathrm{C} \xrightarrow{\text{CL}}{}^{1} \mathrm{C}'$ must be in the power set of C^*; because C^* is finite, its power set is also finite. Let this power set be denoted 2^{C^*} and let n be the size of this set.

The set $\overbrace{\mathrm{C}'}^{m}$ such that for all $i \in \{1..m\}$ we have $\mathrm{C} \xrightarrow{\text{CL}}{}^{*} \mathrm{C}'_i$ is also decidable. We know this set to be a subset of 2^{C^*} and that any acyclic chain of closures $\mathrm{C} \xrightarrow{\text{CL}}{}^{1} \mathrm{C}'_{j_1} \xrightarrow{\text{CL}}{}^{1} \ldots \xrightarrow{\text{CL}}{}^{1} \mathrm{C}'_i$ (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 $\overset{m}{\sqcup}\mathrm{C}'$ and which are not; therefore, $\overset{m}{\sqcup}\mathrm{C}'$ is decidable.

Finally, for each such C'_i, Lemma B.16 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 5.2 in the previous section used a counting argument for sake of simplicity. While this shows that the MicroBang 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 $\mathrm{C} \xrightarrow{\mathrm{CL}}{}^{*} \mathrm{C}'$ 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.8 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 [18]. It is well known that working with non-contractive recursive types is difficult [18] and, for that reason, it is standard to work only with contractive types. While eliminating non-contractive types from TinyBang 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 object 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 TinyBang 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 TinyBang is exponentially complex in pathological scenarios. We have attempted to ensure, however, that TinyBang 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 WIDEN 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 TinyBang typechecker is feasible, but we concede that it will take more theoretical effort than other, more traditional typecheckers due to subtle issues like those mentioned above.