Types for Flexible Objects

Pottayil Harisanker Menon   Zachary Palmer   Alexander Rozenshteyn   Scott Smith

The Johns Hopkins University

{pharisa2, zachary.palmer, arozens1, scott}@jhu.edu

Abstract

Scripting languages are popular in part due to their extremely flexible objects. Features such as dynamic extension, mixins, and first-class messages improve programmability and lead to concise code. But attempts to statically type these features have met with limited success.

Here we present TinyBang, a small typed language in which flexible object operations can be encoded. We illustrate this flexibility by solving an open problem in OO literature: we give an encoding where objects can be extended after being messaged without compromising the expressiveness of subtyping.

TinyBang's subtype constraint system ensures that all types are completely inferred; there are no data declarations or type annotations. We formalize TinyBang and prove the type system is sound and decidable; all examples in the paper run in our most recent implementation.

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.

While a significant body of work has focused on statically typing flexible object operations [5, 7, 20], the solutions proposed place significant restrictions on how objects can be used. The fundamental tension lies in supporting self-referentiality. For an object to be extensible, "self" must be exposed in some manner equivalent to a function abstraction λ`self`... so that different "self" values may be used in the event of extension. But exposing "self" in this way puts it in a contravariant position; as a result, subtyping on objects is invalid. The above systems create compromises; [7], for instance, does not permit objects to be extended after they are messaged.

Along with the problem of contravariant self, it is challenging to define a fully first-class object concatenation operation with pleasing typeability properties. The aforementioned type systems do not support concatenation of arbitrary objects. In the related space of typed record concatenation, previous work [18] has shown that general record concatenation may be typed but requires considerable machinery including presence/absence types and conditional constraints.

In this paper, we present a new programming language calculus, TinyBang, which aims for significant flexibility in statically typing flexible object operations. In particular, we support object extension without restrictions, and we have simple type rules for a first-class concatenation operation.

TinyBang achieves its expressiveness with very few primitives: the core expressions include only labeled data, concatenation, higher-order functions, and pattern matching. Classes, objects, inheritance, object extension, overloading, and switch/case can be fully and faithfully encoded with these primitives. TinyBang also has full type inference for ease and brevity of programming. It is not intended to be a programming language for humans; instead, it aims to serve as a conceptual core for such a language.

1.1 Key Features of TinyBang

TinyBang's type system is grounded in subtype constraint type theory [3], with a series of improvements to both expression syntax and typing to achieve the expressiveness needed for flexible object encodings.

Type-indexed records supporting asymmetric concatenation TinyBang uses type-indexed records: records for which content can be projected based on its type [21]. For example, consider the type-indexed record `{foo = 45; bar = 22; 13}`: the untagged element `13` is implicitly tagged with type `int`, and projecting `int` from this record would yield `13`. Since records are type-indexed, we do not need to distinguish records from non-records; `22`, for example, is a type-indexed record of one (integer) field. Variants are also just a special case of 1-ary records of labeled data, so `` `Some 3`` expresses the ML `Some(3)`. Type-indexed records are thus a universal data type and lend themselves to flexible programming patterns in the same spirit as Lisp lists and Smalltalk objects.

TinyBang records support asymmetric concatenation via the `&` operator; informally, `{foo = 45; bar = 22; 13} & {baz = 45; bar = 10; 99}` results in `{foo = 45; bar = 22; baz = 45; 13}` since the left side is given priority for the overlap. Asymmetric concatenation is key for supporting flexible object concatenation, as well as for standard notions of inheritance. We term the `&` operation *onioning*.

Dependently typed first-class cases TinyBang's first-class functions are written "*pattern* `->` *expression*". In this way, first-class functions are also first-class *case clauses*. We permit the concatenation of these clauses via `&` to give multiple dispatch possibilities. TinyBang's first-class functions generalize the first-class cases of [6].

In standard type systems, all case branches are constrained to have the same result type, losing the dependency between the variant input and the output. This problem is solved in TinyBang by giving compound functions dependent types: application of a compound function can return a different type based on the variant constructor of its argument. Additionally, we define a novel notion of *slice* which allows the type of bindings in a case arm to be refined based on which pattern was matched. Dependently typed first-class cases are critical for typing our object encodings, a topic we discuss later in this section.

1.2 Object-oriented programming in TinyBang

Contributions of our object encodings include the following.

Objects as variants Objects are commonly encoded as records of functions; method invocation is achieved by invoking a function in the record. While such an encoding could be made to work in TinyBang, it is much more complex and so we opt to use a *variant-based* encoding: objects are message-processing functions which case on the form of message and method invocation is a simple function call. The variant-based view of objects is not new; classic Actor models [2] use an untyped form. But while these models are traditionally quite challenging to type, TinyBang's type system is well-suited to them. First, variant-based objects rely on case matching, a traditionally homogeneously-typed construct, to dispatch messages; the type-refining argument slicing and dependently-typed case result types makes them expressive enough to be useful for encoding objects. Second, case matching is a monolithic operation in most languages, making it difficult to compose objects or create extensions; TinyBang's first-class case clauses support object concatenation and subclassing.

Resealing objects to support object extension A key idea of [7] is a *sealing* transformation where an object template turns into a messageable (but non-extensible) object. Our encoding of objects extends that idea by adding a *resealing* operation: the type of `self` is captured in the closure of a message dispatch function that can be *overridden* due to the asymmetric nature of onioning in TinyBang. Our resealer solves an open problem of making a fully flexible model where objects can be both extended and messaged whilst

$$
\begin{array}{rcll}
e & ::= & x \mid () \mid \mathbb{Z} \mid l\ e \mid \mathtt{ref}\ e \mid \mathtt{!}\ e \mid e\,\&\,e \mid & \textit{expressions} \\
 & & \phi \mathbin{\mathtt{->}} e \mid e \boxdot e \mid e\ e \mid \mathtt{let}\ x = e\ \mathtt{in}\ e \mid x \mathbin{:=} e\ \mathtt{in}\ e & \\
\phi & ::= & x \mid () \mid \mathtt{int} \mid l\ \phi \mid \phi\,\&\,\phi & \textit{patterns} \\
\boxdot & ::= & \mathtt{+} \mid \mathtt{-} \mid \mathtt{==} \mid \mathtt{<=} \mid \mathtt{>=} & \textit{operators} \\
l & ::= & \text{‘}\textit{(alphanumeric)} & \textit{labels}
\end{array}
$$

Figure 2.1. TinyBang Syntax

being statically typed. Several of the novel features of TinyBang need to work together to make this encoding work.

Outline In the next section, we give an overview of TinyBang and how it can encode object features. In Section 3, we show the operational semantics and type system for a core subset of TinyBang, trimmed to the key features for readability; we prove soundness and decidability for this system in the appendices. Related work is in Section 4 and we conclude in Section 5.

2. Overview

This section gives an overview of the TinyBang language and of how it supports flexible object operations and other scripting features.

2.1 Language Features for Flexible Objects

The TinyBang syntax used in this section appears in Figure 2.1. The core TinyBang formalized in Section 3 uses a restricted A-normal form of this grammar. Operator precedence is as follows: labels (e.g. `‘Foo 0`) have highest parse precedence, onioning (e.g. `‘A 0 & ‘B 0`) has the next highest precedence, and function arrows (e.g. `x -> ‘A x`) have the least precedence. Patterns use the same precedence rules. State and variable binding uses ML-style syntax; reference constructors are parsed with the same precedence as labels.

Program types take the form of a set of subtype constraints [3]. For the purposes of this Overview we will be informal about type syntax. Our formal type constraint syntax can be viewed as an A-normalized type grammar, and this grammar implicitly supports (positive) union types by giving a type variable multiple lower bounds: for example, `int` $\cup$ `bool` is equivalently expressed as a type α with constraints `int` $<:\ \alpha$ and `bool` $<:\ \alpha$. So, as we do with the expression syntax, we will write types in a standard nested form in this section for presentation clarity. The details of the type system are presented in Section 3.3.

Simple functions 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 these one-clause pattern-matching functions *simple functions*. For instance, consider the following object and its invocation:

```
let obj = (`twice x -> x + x) in obj (`twice 4)
```

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

Unlike a traditional `match` expression, a simple function is only capable of matching one pattern. To express general match expressions, functions are concatenated via the *onion* operation `&` to give *compound functions*. Given two function expressions `e1` and `e2`, the expression `(e1 & e2)` conjoins them to make a compound function with the conjoined pattern, and `(e1 & e2) a` will apply the function which has a pattern matching `a`; if both patterns match `a`, the leftmost function (e.g. `e1`) is given priority. We can thus write a dispatch on an object with two methods simply as:

```
let obj = (`twice x -> x + x) & (`isZero x -> x == 0) in
obj `twice 4
```

The above shows that traditional `match` expressions can be encoded using the `&` operator to join a number of simple functions: one for each case. Function conjunction generalizes the first-class cases of [6]; that work does not support "override" of existing clauses or heterogeneously typed case branches.

Dependent pattern types The above shows how to encode an object with multiple methods as an onion of simple functions. 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
                 | `twice x -> x + x
                 | `isZero x -> x == 0) in ...
```

will not typecheck; OCaml match/case expressions must return the same type in all case branches.[1] Instead, we give the function a dependent pattern type that is informally (`` `twice int `` $\rightarrow$ `int`) & (`` `isZero int `` $\rightarrow$ `bool`). If the function 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 `bool` and not type `int` $\cup$ `bool`. 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 [3, 18] in a dimension critical for typing objects.

[1] 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.

This need for dependent typing arises largely from our desire to accurately type a variant-based object model; a record-based encoding of objects would not have this problem. We choose a variant-based encoding because it greatly simplifies the encodings such as self-passing and overloading, which we describe below.

Onions are records There is no record syntax in TinyBang; we only require the record concatenation operator `&` so we can append values into type-indexed records. We informally call these records *onions* to signify these properties. Here is an example of how objects can be encoded with multi-argument methods:

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

The `` `x 3 & `y 2 `` is an onion of two labels and amounts to a two-label record. This `` `sum ``-labeled onion is passed to the pattern `` `x x & `y y ``. (We highlight the pattern `&` differently than the onioning `&` because the former is a *pattern conjunction* operator: the value must match both subpatterns.) Also observe from this example how there is no hard distinction in TinyBang between records and variants: there is only one class of label. This means that the 1-ary record `` `sum 4 `` is the same as the 1-ary variant `` `sum 4 ``.

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 [7], 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*. This 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`, which takes an object `obj` and returns its sealed counterpart. We define `seal` as follows:

```
let fixpoint =
        f -> (g -> x -> g g x) (h -> y -> f (h h) y) in
let seal = fixpoint (seal -> obj ->
        (msg -> obj (msg & `self (seal obj))) & obj) in
let obj = (`twice x -> x + x) &
          (`quad x & `self self ->
               self (`twice x) + self (`twice x))
let sObj = seal obj in
let twenty = sObj `quad 5 in          // returns 20
```

The `seal` function operates by adding a message handler which captures *every* message sent to `obj`. (We still add

& obj to this message handler to preserve the non-function parts of the object.) The message handler adds a ‘self component (containing seal obj) to the right of the message and then passes it to the original object. We require fixpoint to ensure that this self-reference is also sealed. Thus, every message send to sObj will, in effect, be sent to obj with ‘self sObj attached to the right.

Extending previously sealed objects In the self binding function above, the value of self is onioned onto the *right* of the message; this gives any explicit value of ‘self in a message passed to a sealed object priority over the ‘self provided by the self binding function (onioning is asymmetric with left precedence). Consider the following continuation of the previous code:

```
let sixteen = sObj ‘quad 4 in          // returns 16
let obj2 = (‘twice x -> x) & sObj in
let sObj2 = seal obj2 in
let eight = sObj2 ‘quad 4 in ...       // returns 8
```

We can extend sObj after messaging it, here overriding the ‘twice message; sObj2 represents the (re-)sealed version of this new object. sObj2 properly knows its "new" self due to the resealing, evidenced here by how ‘quad invokes the new ‘twice. To see why this works let us trace the execution. Expanding the sealing of sObj2, sObj2 (‘quad 4) has the same effect as obj2 (‘quad 4 & ‘self sObj2), which has the same effect as sObj (‘quad 4 & ‘self sObj2). Recall sObj is also a sealed object which adds a ‘self component to the *right*; thus this has the same effect as obj (‘quad 4 & ‘self sObj2 & ‘self sObj). Because the leftmost ‘self has priority, the ‘self is properly sObj2 here. We see from the original definition of obj that it sends a ‘twice message to the contents of self (here, sObj2), which then follows the same pattern as above until obj (‘twice 4 & ‘self sObj2 & ‘self sObj) is invoked (two times – once for each side of +).

Sealed and resealed objects obey the desired object subtyping laws because we "tie the knot" on self using seal, meaning there is no contravariant self parameter on object method calls to invalidate object subtyping. Additionally, our type system includes parametric polymorphism and so sObj and the re-sealed sObj2 do not have to share the same self type, and the fact that & is a *functional* extension operation means that there will be no pollution between the two distinct self types. Key to the success of this encoding is the asymmetric nature of &: it allows us to override the default ‘self parameter. This self resealing is possible in the record model of objects, but is much more convoluted; this is a reason that we switched to a variant model.

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

```
let obj = seal (‘x (ref 0) &
               (‘inc _ & ‘self self ->
                     (‘x x -> x := !x + 1 in !x) self))
in obj ‘inc ()
```

Observe how obj is a heterogeneous "mash" of a record field (the ‘x) and a function (the handler for ‘inc). This is sound because onions are *type-indexed* [21], meaning that they use the types of the values themselves to identify data. For this particular example, invocation obj ‘inc () (note () is an empty onion, a 0-ary conjunction) correctly increments in spite of the presence of the ‘x label in obj.

Here we define a few sugarings which we use in the examples throughout the remainder of this section, although a "real" language built on these ideas would include sugarings for each of the features we are about to mention as well.

o.x	≅	(‘x x -> x) o
o.x = e_1 in e_2	≅	(‘x x -> x = e_1 in e_2) o
if e_1 then e_2 else e_3	≅	((‘True _ -> e_2) & (‘False _ -> e_3)) e_1
e_1 and e_2	≅	((‘True _ -> e_2) & (‘False _ -> ‘False ())) e_1

Using this sugar, the update in the counter object above could be expressed more succinctly as self.x = self.x + 1 in self.x.

2.3 Flexible Object Operations

Here we cover how TinyBang supports a broad family of flexible object operations, expanding on the first-class messages and flexible extension operations covered above. We show encodings in terms of objects rather than classes for simplicity; applying these concepts to classes is straightforward.

Default arguments TinyBang can easily encode default arguments, arguments which are optional and take on a default value if missing. For instance, consider:

```
let obj = seal ( (‘add (‘x x & ‘y y) -> x + y)
               & (‘sub (‘x x & ‘y y) -> x - y) ) in
let dflt = obj -> (‘add a -> obj (‘add (a & ‘x 1))) & obj in
let obj2 = dflt obj in
obj2 (‘add (‘y 3)) + obj2 (‘add (‘x 7 & ‘y 2))  // 4 + 9
```

Object dflt overrides obj's ‘add to make 1 the default value for ‘x. Because the ‘x 1 is onioned onto the right of a, it will have no effect if an ‘x is explicitly provided in the message. This example also shows how the addition of default behavior is a first-class operation and not just first-order syntax, giving TinyBang more flexibility than most implementations of default arguments.

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

```
let neg = x & int -> 0 - x in ...
```

Here, the conjunction pattern x & int will match the argument with int and also bind it to the variable x. 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`:

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

Negation is now overloaded: `neg 4` evaluates to `-4`, and `neg ‘True ()` evaluates to `‘False ()` due to how application matches function patterns.

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

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

Here `mixin` is a function which invokes the value passed as `self`. Because an object's methods are just functions onioned together, onioning `mixin` into `point` is sufficient to produce a properly functioning `mixPt`.

The above example typechecks in TinyBang; parametric polymorphism is used to allow `point`, `mixin`, and `mixPt` to have different self-types. The `mixin` variable has the approximate type "(`‘near unit & ‘self` α) $\rightarrow$ `bool` 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 function 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`.

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

Inheritance, classes, and subclasses Typical object-oriented constructs can be defined similarly to the above. Object inheritance is similar to mixins, but a variable `super` is also bound to the original object and captured in the closure of the inheriting objects methods, allowing it to be reached for static dispatch. The flexibility of the `seal` function permits us to ensure that the inheriting object is used for future dispatches even in calls to overridden methods. Classes are simply objects that generate other objects, and subclasses are extensions of those object generating objects. We forgo examples here for brevity.

$$
\begin{array}{rcll}
e & ::= & \overrightarrow{s} & \textit{expressions} \\
E & ::= & \overrightarrow{x = v} & \textit{environment} \\
s & ::= & x = v \mid x = x \mid x = x\ x & \textit{clauses} \\
v & ::= & () \mid l\ x \mid x \,\&\, x \mid \phi \texttt{->} e & \textit{values} \\
\phi & ::= & \overrightarrow{x = \mathring{v}} & \textit{patterns} \\
\mathring{v} & ::= & () \mid l\ x \mid x \,\&\, x & \textit{pattern vals} \\
B & ::= & \overrightarrow{x = x} & \textit{bindings} \\
l & ::= & \text{‘}\textit{(alphanumeric)} & \textit{labels} \\
x & ::= & \textit{(alphanumeric)} & \textit{variables}
\end{array}
$$

Figure 3.1. TinyBang ANF Grammar

3. Formalization

We now give formal semantics to a subset of TinyBang. For clarity, features which are not directly related to our conjunction semantics – integers, state, etc. – are omitted. We will first translate the TinyBang program to A-normal form; this allows us to use much simpler definitions and provides a better alignment between expressions and types. Section 3.2 defines the operational semantics of the A-normalized version of restricted TinyBang. Section 3.3 defines the type system and soundness and decidability properties. We show how the omitted features can be reintroduced to this formal system in Appendix D.

Notation For a given construct g, we let $[g_1, \ldots, g_n]$ denote an n-ary list of g, often using the equivalent shorthand ${}^{n}\overrightarrow{g}$. We elide the n when it is unnecessary. Operator $\|$ denotes list concatenation. For sets, we use similar notation: ${}^{n}\overbracket{g}$ abbreviates $\{g_1, \ldots, g_n\}$ for some arbitrary ordering of the set. We sometimes use $\scriptstyle\square$ to indicate index positions for clarity. For example, ${}^{n}\overrightarrow{g_{\square}/g'}$ is shorthand for $[g_1/g', \ldots, g_n/g']$.

3.1 A-Translation

In order to simplify our formal presentation, we convert TinyBang into A-normal form; this brings expressions, patterns, and types into close syntactic alignment which greatly simplifies the proofs. The grammar of our A-normalized language appears in Figure 3.1. For the purposes of discussion, we will refer to the restriction of the language presented in Section 2 as the *nested* language and to the language appearing in Figure 3.1 as the *ANF* language.

Observe how expression and pattern grammars are nearly identical. We require that both expressions and patterns declare each variable at most once; expressions and patterns which do not have this property must be α-renamed such that they do. The constructions E and B are not directly used in the A-translation; they define the environment and bindings for the operational semantics below.

We define the A-translation function $\llparenthesis e \rrparenthesis_x$ in Figure 3.2. This function accepts a nested TinyBang expression and produces the A-normalized form $\overrightarrow{s}$ in which the final declared variable is x. We overload this notation to patterns as well.

Expressions

$$\begin{array}{rcl}
\llparenthesis\, () \,\rrparenthesis_x & = & [x = ()] \\
\llparenthesis\, l\ e \,\rrparenthesis_x & = & \llparenthesis e \rrparenthesis_y \,\|\, [x = l\ y] \\
\llparenthesis\, e_1 \mathbin{\&} e_2 \,\rrparenthesis_x & = & \llparenthesis e_1 \rrparenthesis_y \,\|\, \llparenthesis e_2 \rrparenthesis_z \,\|\, [x = y \mathbin{\&} z] \\
\llparenthesis\, \phi \texttt{->} e \,\rrparenthesis_x & = & [x = \llparenthesis \phi \rrparenthesis_y \texttt{->} \llparenthesis e \rrparenthesis_z] \\
\llparenthesis\, e_1\ e_2 \,\rrparenthesis_x & = & \llparenthesis e_1 \rrparenthesis_y \,\|\, \llparenthesis e_2 \rrparenthesis_z \,\|\, [x = y\ z] \\
\llparenthesis\, \texttt{let}\ x_1 = e_1\ \texttt{in}\ e_2 \,\rrparenthesis_{x_2} & = & \llparenthesis e_1 \rrparenthesis_{x_1} \,\|\, \llparenthesis e_2 \rrparenthesis_{x_2} \\
\llparenthesis\, x_2 \,\rrparenthesis_{x_1} & = & [x_1 = x_2]
\end{array}$$

Patterns

$$\begin{array}{rcl}
\llparenthesis\, () \,\rrparenthesis_x & = & [x = ()] \\
\llparenthesis\, l\ \phi \,\rrparenthesis_x & = & \llparenthesis \phi \rrparenthesis_y \,\|\, [x = l\ y] \\
\llparenthesis\, \phi_1 \mathbin{\&} \phi_2 \,\rrparenthesis_x & = & \llparenthesis \phi_1 \rrparenthesis_y \,\|\, \llparenthesis \phi_2 \rrparenthesis_z \,\|\, [x = y \mathbin{\&} z] \\
\llparenthesis\, x_2 \,\rrparenthesis_{x_1} & = & [x_2 = ()]
\end{array}$$

Figure 3.2. TinyBang A-Translation

We use y and z to range over fresh variables unique to that invocation of the translation function; different recursive invocations use different fresh variables y and z.

Notice that A-translation of patterns is in perfect parallel with the expressions in the above; for example, the expression `‘A x -> x` translates to $[$`y`$_1$ `=` $($`[x = ()`$,$ `y`$_3$ `= ‘A x]` `->` `[y`$_2$ `= x]`$)]$. Using the same A-translation for patterns and expressions greatly aids the formal development, but it takes some practice to read these A-translated patterns. Variables matched against empty onion (`x = ()` here) are unconstrained and represent bindings that can be used in the body. Other variables such as the `y`$_3$ are not bindings; clause `y`$_3$ `= ‘A x` constrains the argument to match a `‘A`-labeled value. The last binding in the pattern, here `y`$_3$ `= ‘A x`, is taken to match the argument when the function is applied; this is in analogy to how the variable in the last clause of an expression is the final value. Variable binding is mostly standard: every clause `x = ()` appearing in the function's pattern binds `x` in the body of the function. In clause lists, each clause binds the defining variable for all clauses appearing after it; nested function clauses follow the usual lexical scoping rules. For the remainder of the paper, we assume expressions are closed unless otherwise noted.

3.2 Operational Semantics

Next, we define an operational semantics for ANF TinyBang. The primary complexity of these semantics is pattern matching, for which several auxiliary definitions are needed.

3.2.1 Compatibility

The first basic relation we define is *compatibility*: is a value accepted by a given pattern? We define compatibility using a constructive failure model for reasons of well-foundedness which are discussed below. We use the symbol $\odot$ to range over the two symbols $\bullet$ and $\circ$, which indicate compatibility and incompatibility, respectively. We define a simple ordering with $\circ < \bullet$ on these symbols.

We write $x \ {}^{\odot}_{E}\!\preceq^{B}_{\phi}\ x'$ to indicate that the value x is compatible (if $\odot = \bullet$) or incompatible (if $\odot = \circ$) with

Empty Onion

$$\frac{x_0 = v \in E \qquad x'_0 = () \in \phi \qquad B = [x'_0 = x_0]}{x_0 \ {}^{\bullet}_{E}\!\preceq^{B}_{\phi}\ x'_0}$$

Label

$$\frac{x_0 = l\ x_1 \in E \qquad x'_0 = l\ x'_1 \in \phi \qquad x_1 \ {}^{\odot}_{E}\!\preceq^{B}_{\phi}\ x'_1}{x_0 \ {}^{\odot}_{E}\!\preceq^{B}_{\phi}\ x'_0}$$

Conjunction Pattern

$$\frac{\begin{array}{c} x'_0 = x'_1 \mathbin{\&} x'_2 \in \phi \\ x_0 \ {}^{\odot_1}_{E}\!\preceq^{B_1}_{\phi}\ x'_1 \qquad x_0 \ {}^{\odot_2}_{E}\!\preceq^{B_2}_{\phi}\ x'_2 \end{array}}{x_0 \ {}^{\min(\odot_1,\odot_2)}_{E}\!\preceq^{B_1 \,\|\, B_2}_{\phi}\ x'_0}$$

Onion Value Left

$$\frac{x_0 = x_1 \mathbin{\&} x_2 \in E \qquad x_1 \ {}^{\bullet}_{E}\!\preceq^{B}_{\phi}\ x'_0}{x_0 \ {}^{\bullet}_{E}\!\preceq^{B}_{\phi}\ x'_0}$$

Onion Value Right

$$\frac{\begin{array}{c} x_0 = x_1 \mathbin{\&} x_2 \in E \\ x'_0 = x'_1 \mathbin{\&} x'_2 \notin \phi \qquad x_1 \ {}^{\circ}_{E}\!\preceq^{B'}_{\phi}\ x'_0 \qquad x_2 \ {}^{\odot}_{E}\!\preceq^{B}_{\phi}\ x'_0 \end{array}}{x_0 \ {}^{\odot}_{E}\!\preceq^{B}_{\phi}\ x'_0}$$

Label Mismatch

$$\frac{\begin{array}{c} x_0 = l\ x_1 \in E \qquad x'_0 = \mathring{v} \in \phi \qquad \mathring{v} = l'\ x_2 \text{ only if } l \neq l' \\ \mathring{v} \text{ not of the form } x' \mathbin{\&} x'' \text{ or } () \end{array}}{x_0 \ {}^{\circ}_{E}\!\preceq^{[]}_{\phi}\ x'_0}$$

Figure 3.3. Pattern compatibility rules

the pattern x'. E represents the environment in which to interpret the value x while ϕ represents the environment in which to interpret the pattern x'. B dictates how, upon a successful match, the values from E will be bound to the pattern variables in ϕ. Compatibility is the least relation satisfying the rules in Figure 3.3.

The compatibility relation is key to TinyBang's semantics and bears some explanation. As mentioned above, every clause $x = ()$ appearing in the pattern binds the variable x; the Empty Onion rule ensures this by adding a binding clause to B. The Label rule simply recurses when the value is a label and the pattern matches that label; the Label Mismatch rule (which is the base case for failure) applies when the pattern does not match that label. Conjunction is relatively self-evident; $\min$ is used here as a logical "and" over the two recursive premises.

The onion rules require special attention due to TinyBang's asymmetric concatenation semantics. Given a value $x_1 \mathbin{\&} x_2$, it is possible that both x_1 and x_2 match the pattern. If so, we must ensure that we take the bindings from the compatibility of x_1 and *not* those from x_2. The Onion Value Left rule applies when the left side matches; in such a case, whether the right side matches is irrelevant. The Onion Value Right rule only applies if the left side *doesn't* match;

that is, a proof of compatibility may recursively depend on a proof of *incompatibility*. This is the reason that our relation is defined to be constructive for both success and failure: it is necessary to show that the relation is inductively well-founded. The premise $x'_0 = x'_1 \,\&\, x'_2 \notin \phi$ is used to ensure that e.g. when matching the value `‘A () & ‘B ()` to the pattern `‘A () & ‘B ()`, we decompose the pattern first.[2]

This relation is designed with a particular goal: to make compatibility between a value and a pattern unambiguous. We assert that we have done so in the following lemma:

Lemma 3.1. For any x, E, x', and ϕ, there exist unique B and $\odot$ such that $x \overset{\odot}{\underset{E}{}}\!\preccurlyeq^{B}_{\phi}\ x'$.

The proof is direct by induction: B and $\odot$ are constructed from the leaves of the tree up to the root. Similar lemmas apply for the other relations throughout this section.

For an example, consider matching the pattern `‘A a & ‘B b` against the value `‘A 0 & ‘B ()`. The A-translations of these expressions are, respectively, the first and second columns below. Compatibility `v5` $\overset{\bullet}{\underset{E}{}}\!\preccurlyeq^{B}_{\phi}$ `p3` holds with the bindings B shown in the third column.

E	ϕ	B
`v1 = ()`	`a = ()`	`a = v1`
`v2 = ‘A v1`	`p1 = ‘A a`	
`v3 = ()`	`b = ()`	`b = v3`
`v4 = ‘B v3`	`p2 = ‘B b`	
`v5 = v2 & v4`	`p3 = p1 & p2`	

3.2.2 Matching

Compatibility determines if a value matches a *single* pattern; we next define a matching relation to check if a series of pattern clauses match. In TinyBang, recall there is no `case`/`match` syntax for pattern matching; individual pattern clauses *pattern* `->` *body* are expressed as simple functions $\phi \mathrel{\texttt{->}} e$ and a series of pattern clauses are expressed by onioning together simple functions. We thus need to define an application matching relation $x_0\ x_1 \overset{\odot}{\rightsquigarrow}_E\ e$ to determine if an onion of pattern clauses x_0 can be applied to argument x_1. This relation is constructive on failure in the same fashion as compatibility, this time to ensure that left precedence on compound function invocation is correctly enforced.

We define matching as the least relation satisfying the rules in Figure 3.4. The helper function RV extracts the return variable from a value, formally defined as follows: $\textsc{rv}(e \,\|\, [x = v]) = x$ and $\textsc{rv}(e \,\|\, [x = \mathring{v}]) = x$.

The Function rule is the base case of a simple function application: the argument value x_1 must be compatible with the pattern ϕ, and if so insert the resulting bindings B at the top of the function body e. The Onion Left/Right rules are the inductive cases; notice that the Onion Right rule can only match successfully if the (higher priority) left side has failed to match. The Non-Function rule is the base case for application of a non-function, which fails but in a way which permits dispatch to continue through the onion.

Function
$$\frac{x_0 = (\phi \mathrel{\texttt{->}} e) \in E \qquad x_1 \overset{\odot}{\underset{E}{}}\!\preccurlyeq^{B}_{\phi}\ \textsc{rv}(\phi)}{x_0\ x_1 \overset{\odot}{\rightsquigarrow}_E\ B \,\|\, e}$$

Onion Left
$$\frac{x_0 = x_2 \,\&\, x_3 \in E \qquad x_2\ x_1 \overset{\bullet}{\rightsquigarrow}_E\ e}{x_0\ x_1 \overset{\bullet}{\rightsquigarrow}_E\ e}$$

Onion Right
$$\frac{x_0 = x_2 \,\&\, x_3 \in E \qquad x_2\ x_1 \overset{\circ}{\rightsquigarrow}_E\ e' \qquad x_3\ x_1 \overset{\odot}{\rightsquigarrow}_E\ e}{x_0\ x_1 \overset{\odot}{\rightsquigarrow}_E\ e}$$

Non-Function
$$\frac{x_0 = (\phi \mathrel{\texttt{->}} e) \notin E \qquad x_0 = x_2 \,\&\, x_3 \notin E}{x_0\ x_1 \overset{\circ}{\rightsquigarrow}_E\ e}$$

Figure 3.4. Application matching rules

Variable Lookup
$$\frac{x_1 = v \in E}{E \,\|\, [x_2 = x_1] \,\|\, e \longrightarrow^1 E \,\|\, [x_2 = v] \,\|\, e}$$

Application
$$\frac{x_0\ x_1 \overset{\bullet}{\rightsquigarrow}_E\ e' \qquad \boldsymbol{\alpha}(e') = e''}{E \,\|\, [x_2 = x_0\ x_1] \,\|\, e \longrightarrow^1 E \,\|\, e'' \,\|\, [x_2 = \textsc{rv}(e'')] \,\|\, e}$$

Figure 3.5. The operational semantics small step relation

3.2.3 Operational Semantics

Using the compatibility and matching relations from above, we now define the operational semantics of restricted TinyBang as a small step relation $e \longrightarrow^1 e'$. Our definition uses an environment-based semantics; it proceeds by acting on the first unevaluated clause of e. We use an environment-based semantics (rather than a substitution-based semantics) due to its suitability to ANF and because it aligns well with the type system presented in the next section.

We must freshen variables as they are introduced to the expression to preserve the invariant that the ANF TinyBang expression uniquely defines each variable; to do so, we take $\boldsymbol{\alpha}(e)$ to be an α-renaming function which freshens (with respect to the operational semantics derivation) all variables in e which are not free. We then define the small step relation as the least relation satisfying the rules given by Figure 3.5.

The application rule simply inlines the (freshened) function body e'' in the event that the matching relation reports that it matched; recall that e' returned by matching is the body of the matched function with argument variable bind-

[2] Interesting semantics arise from these decisions. While it is possible to add to an onion a label it already contains, this doesn't necessarily override that label in the shallow sense; that is, `‘A (‘B ()) & ‘A (‘C ())` still matches the pattern `‘A ‘C ()`. There is subtle expressiveness in these semantics; we do not explore them here for reasons of space.

$$
\begin{array}{rcll}
C & ::= & \overbracket{c} & \textit{constraint sets} \\
c & ::= & \tau <: \alpha \mid \alpha <: \alpha \mid \alpha\ \alpha <: \alpha & \textit{constraint} \\
V & ::= & \overbracket{\tau <: \alpha} & \textit{constraint value sets} \\
F & ::= & \overbracket{\alpha <: \alpha} & \textit{constraint flow sets} \\
\tau & ::= & \texttt{()} \mid l\ \alpha \mid \alpha \,\&\, \alpha \mid \alpha\backslash V \rightarrow \alpha\backslash C & \textit{types} \\
\alpha & & & \textit{type variables}
\end{array}
$$

Figure 3.6. The TinyBang type grammar

ings prepended. The variable x_2 needing the result of the call is assigned the return value of the inlined function.

We define $e_0 \longrightarrow^* e_n$ 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. When $e \not\longrightarrow^1$ for some e not of the form E, we say that e is *stuck*.

3.3 Type System

We base TinyBang's type system on subtype constraint systems, which have been shown to be quite expressive [3] and suitable for complex pattern matching [18] and object-orientation [24]. We begin by establishing a close syntactic correspondence between expressions and types (which is made easy by our choice of ANF expression syntax); we then define type system relations which parallel those from the operational semantics, including a deductive constraint closure which parallels the small step relation itself. Our proof of soundness proceeds by showing that, due to this alignment, programs which get stuck always correspond to inconsistent constraint sets.

3.3.1 Initial Alignment

Figure 3.6 provides a grammar of the type system. Note the very close alignment between expression and type grammar elements; E has type V, and variable bindings B have the type analog F. Expressions e have types $\alpha\backslash C$; expressions and type grammars are less different than they appear, since the expression clauses are a list where the last variable contains the final value implicitly whereas in the type the final type location α must be explicit. Both v and $\mathring{v}$ have type τ.

We formalize this initial alignment step as a function $\llbracket e \rrbracket_{\text{E}}$ which produces a constrained type $\alpha\backslash C$; see Figure 3.7. Initial alignment over a given e picks a single fresh type variable for each program variable in e; for the variable x_0, we denote this fresh type variable as $\overset{\star}{\alpha}_0$. The function is simple: each clause $x_0 \texttt{=} \ldots$ is mapped to a constraint $\ldots <: \overset{\star}{\alpha}_0$. Note that functions $\phi \texttt{->} e$ produce types $\alpha'\backslash V \rightarrow \alpha\backslash C$: patterns only contain value constraints $V \subseteq C$.

3.3.2 Equivalence

Below, we define the type system's constraint closure relation. As with the grammars, the type system relations closely parallel the relations in the evaluation system: there is a type system analogue for compatibility, matching, and small-step operational semantics (the latter of which is paralleled by the constraint closure relation itself). In order to keep the

$$
\begin{array}{rcl}
\llbracket \overset{n}{\overrightarrow{s}} \rrbracket_{\text{E}} & = & \alpha_n \backslash \overset{n}{\overbracket{c}} \text{ where} \\
 & & \quad \forall i \in \{1..n\}. \llbracket s_i \rrbracket_{\text{S}} = \alpha_i \backslash c_i \\
\llbracket \overset{n}{\overrightarrow{x \texttt{=} \mathring{v}}} \rrbracket_{\text{P}} & = & \alpha_n \backslash \overset{n}{\overbracket{c}} \text{ where} \\
 & & \quad \forall i \in \{1..n\}. \llbracket x_i \texttt{=} \mathring{v}_i \rrbracket_{\mathring{\text{S}}} = \alpha_i \backslash c_i \\[1ex]
\llbracket x_0 \texttt{=} \texttt{()} \rrbracket_{\text{S}} & = & \overset{\star}{\alpha}_0 \backslash\ \texttt{()} <: \overset{\star}{\alpha}_0 \\
\llbracket x_0 \texttt{=} l\ x_1 \rrbracket_{\text{S}} & = & \overset{\star}{\alpha}_0 \backslash\ l\ \overset{\star}{\alpha}_1 <: \overset{\star}{\alpha}_0 \\
\llbracket x_0 \texttt{=} x_1 \,\&\, x_2 \rrbracket_{\text{S}} & = & \overset{\star}{\alpha}_0 \backslash\ \overset{\star}{\alpha}_1 \,\&\, \overset{\star}{\alpha}_2 <: \overset{\star}{\alpha}_0 \\
\llbracket x_0 \texttt{=} \phi \texttt{->} e \rrbracket_{\text{S}} & = & \overset{\star}{\alpha}_0 \backslash\ \llbracket \phi \rrbracket_{\text{P}} \rightarrow \llbracket e \rrbracket_{\text{E}} <: \overset{\star}{\alpha}_0 \\
\llbracket x_0 \texttt{=} x_1 \rrbracket_{\text{S}} & = & \overset{\star}{\alpha}_0 \backslash\ \overset{\star}{\alpha}_1 <: \overset{\star}{\alpha}_0 \\
\llbracket x_0 \texttt{=} x_1\ x_2 \rrbracket_{\text{S}} & = & \overset{\star}{\alpha}_0 \backslash\ \overset{\star}{\alpha}_1\ \overset{\star}{\alpha}_2 <: \overset{\star}{\alpha}_0 \\[1ex]
\llbracket x_0 \texttt{=} \texttt{()} \rrbracket_{\mathring{\text{S}}} & = & \overset{\star}{\alpha}_0 \backslash\ \texttt{()} <: \overset{\star}{\alpha}_0 \\
\llbracket x_0 \texttt{=} l\ x_1 \rrbracket_{\mathring{\text{S}}} & = & \overset{\star}{\alpha}_0 \backslash\ l\ \overset{\star}{\alpha}_1 <: \overset{\star}{\alpha}_0 \\
\llbracket x_0 \texttt{=} x_1 \,\&\, x_2 \rrbracket_{\mathring{\text{S}}} & = & \overset{\star}{\alpha}_0 \backslash\ \overset{\star}{\alpha}_1 \,\&\, \overset{\star}{\alpha}_2 <: \overset{\star}{\alpha}_0
\end{array}
$$

Figure 3.7. Initial alignment

type system decidable, we define these relations in terms of a notion of type variable equivalence which represents type variable unifications. This relation grows monotonically throughout closure as new unifications are performed.

We use the symbol $\cong$ to range over type variable equivalences. For notational purposes, we overload these relations to operate homomorphically on constraint sets. We may also write $\cong$ in a context where a set is expected (e.g. $\langle \alpha, \alpha' \rangle \in \cong$ or $\cong \subseteq \cong'$); in this case, it is interpreted as the set of pairs of variables for which the relation holds.

Most of the relations below are explicitly defined as *equivalence-parametric*, meaning that they take an equivalence relation $\cong$ as an implicit parameter. These relations are defined in a way that respects equivalence: in any place of the relation other than the equivalence parameter, replacing a type variable with another, equivalent type variable will not change whether or not the relation holds. In the definition of equivalence-parametric relations, we assume that $\in$ and $\notin$ are also equivalence parametric; $\texttt{()} <: \alpha \notin C$, for instance, indicates that no constraint $\texttt{()} <: \alpha'$ appears in C such that $\alpha \cong \alpha'$.

3.3.3 Slicing

When defining type compatibility in TinyBang, two subtle problems arise which did not appear in the evaluation system. Here, we describe those problems and define a new relation, *slicing* to solve them.

The first problem has to do with soundness. In the evaluation system, each variable is guaranteed to have a single assignment; thus, a premise of value compatibility in Figure 3.3 such as "$x_1 \texttt{=} l\ x_0 \in E$" is unambiguous. This is not so in the type system: a single type variable may have multiple lower bounds. Such type variables represent union types in constraint-based type systems and present subtle challenges. For instance, consider a direct translation of Figure 3.3 to the type system (replacing all x with α, all B with F, all ϕ with V, and so on). In the resulting relation, the (informally written) type `‘A` $\alpha \cup$ `‘B` α would appear to match the pattern `‘A _ &‘B _`: we can prove that the argument type variable has a `‘A` lower bound and, independently, we can prove that it has a `‘B` lower bound. This is an instance of the well-known *union elimination* problem: we must be consistent in our view of how case analysis on unions is performed. Because TinyBang's dispatch on functions has weak dependent typing properties, this imprecision is actually unsound and must be addressed: if the type system erroneously concludes that the argument matches the `‘A _ &‘B _` pattern, this imprecision may cause it not to consider a lower priority function which is actually invoked at runtime.

The second challenge in defining type compatibility has to do with precision. Recall that a key feature of TinyBang is preserving the dependency between the input and the output of a function; this is critical to support encodings such as overloading as presented in Section 2. In particular, pattern matching in TinyBang does not just bind portions of the argument; it also *refines* the types of those bindings. In essence, TinyBang is resolving application through case analysis. Consider the following TinyBang code:

```
( (‘CB f -> f (‘Foo ()))
& (g -> g ())) arg
```

In this example, we take `arg` to be either a callback function accepting `()` or the label `‘CB` around a callback function accepting `‘Foo ()`; in other words, `arg` is union-typed. The callback function will be invoked based on whether or not the label is present. We must be sure, however, that the second clause only considers the case in which `arg` is a function; this is, after all, the only case in which the second clause can be invoked at runtime. But if we do not perform this refinement at the type level, `g` takes on the full union type and a spurious type error results when we consider a `‘CB` label applied to `()`. (Note that this is different from a legitimate type error, such as if `arg` were to have an `‘Oops` type as a third disjunct.)

Both of these problems are caused by the presence of union types, so we solve them by defining a slicing relation to eliminate unions before checking compatibility. This ensures that our union eliminations are consistent (because they are performed before, not during, compatibility checking) and also provides us with a refined form of the argument to use in typechecking the function body. In the first problem above, slicing eliminates the soundness concern because the argument is first separated into the distinct `‘A` α and `‘B` α slices and compatibility is checked for each of them separately. The second problem above is addressed in the same way; this process is illustrated in Part A of Figure 3.8. Note that this union elimination *must* be complete up to the depth of the pattern; otherwise, union alignment problems will begin where the elimination stopped.

We write $\alpha\backslash V \ll \alpha'\backslash C$ to indicate that $\alpha\backslash V$ is a slice of the constrained type $\alpha'\backslash C$. The relation is equivalence-parametric and is defined as follows:

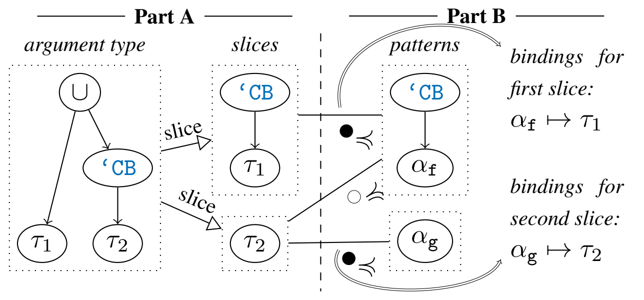
Part A
argument type
slices
∪
‘CB
τ_1
τ_2
slice
slice
Part B
patterns
‘CB
$\alpha_{\mathtt{f}}$
$\alpha_{\mathtt{g}}$
bindings for first slice:
$\alpha_{\mathtt{f}} \mapsto \tau_1$
bindings for second slice:
$\alpha_{\mathtt{g}} \mapsto \tau_2$

Figure 3.8. How slices refine types

Leaf
$$\frac{\alpha \notin V}{\alpha\backslash V \ll \alpha\backslash C}$$

Atomic
$$\frac{\tau \text{ not of the form } l\ \alpha_1 \text{ or } \alpha_1 \mathbin{\&} \alpha_2 \qquad \tau <: \alpha \in V \qquad \tau <: \alpha' \in C}{\alpha\backslash V \ll \alpha'\backslash C}$$

Label
$$\frac{l\ \alpha_1 <: \alpha_0 \in V \qquad l\ \alpha'_1 <: \alpha'_0 \in C \qquad \alpha_1\backslash V \ll \alpha'_1\backslash C}{\alpha_0\backslash V \ll \alpha'_0\backslash C}$$

Onion
$$\frac{\alpha_1 \mathbin{\&} \alpha_2 <: \alpha_0 \in V \qquad \alpha'_1 \mathbin{\&} \alpha'_2 <: \alpha'_0 \in C \qquad \alpha_1\backslash V \ll \alpha'_1\backslash C \qquad \alpha_2\backslash V \ll \alpha'_2\backslash C}{\alpha_0\backslash V \ll \alpha'_0\backslash C}$$

Figure 3.9. The slice relation for union elimination

Definition 3.2. $\alpha\backslash V \ll \alpha'\backslash C$ is the least relation defined by the rules in Figure 3.9. Slicing is equivalence-parametric. Slices are

- *well-formed* iff each α'' in V has at most one lower bound in V,
- *disjoint* iff $\tau <: \alpha'' \in V$ implies that α'' does not appear in C, and
- *minimal* iff every upper-bounding α'' in V is either α or appears in a lower bound in V.
- *acyclic* iff there exists a preorder on type variables in V s.t. $\alpha_1 < \alpha_2$ when $\tau <: \alpha_2 \in V$ and α_1 appears in τ

Unless otherwise explicitly noted, we assume slices are well-formed, disjoint, minimal, and acyclic. Since the set V is minimal, the rules enforce that only one of a choice of unions in C will be in V, achieving the desired union elimination.

One nuance of slicing is that we require the Leaf rule, which allows slicing to stop at an arbitrary point, to handle recursive data types. For instance, the recursive type $\alpha\backslash\{$`‘A` $\alpha <: \alpha, () <: \alpha\}$ expresses the union of the types `()`, `‘A ()`, `‘A ‘A ()`, and so on. The above relation can slice this type into $\alpha'\backslash\{() <: \alpha'\}$, $\alpha'\backslash\{$`‘A` $\alpha'' <: \alpha', () <: \alpha''\}$, and so on. As there are infinitely many such slices, the Leaf rule allows us to create "partial" slices which represent (perhaps infinitely) many complete slices; for instance, the slice $\alpha'\backslash\{$`‘A` $\alpha\}$ (where α is from the original type above) represents all slices other than the simple `()` case. The use of α, unbounded within the set, indicates a point at which union elimination has stopped. It is the responsibility of compatibility, defined below, to handle partial slices correctly.

3.3.4 Compatibility

Using the above slicing relation, we can now define the type compatibility relation. Because a slice is a union-free representation of a type up to some depth, we can define compatibility in much the same way as we did in the evaluation system. As mentioned above, however, slicing may stop at an arbitrary point. In fact, it may not perform any real slicing at all; $\alpha\backslash\emptyset$ is always a legal slice of α!

To handle partial slices, we define the type compatibility relation to filter out slices which are too shallow. In addition to being compatible (●) or incompatible (○), type compatibility may show that a slice and a pattern are *partially* compatible (◐), meaning that the slice lines up with the pattern correctly but is insufficiently deep. We use the metavariable $\circledcirc$ to range over this extension to the $\odot$ grammar. As in value compatibility, we view these symbols as ordered: ○ < ◐ < ●. We define type system compatibility as the least (equivalence-parametric) relation satisfying the rules appearing in Figure 3.10.

Figure 3.11 shows an example of slicing and compatibility on a recursive type: Peano integers. Here, α is a union type between a successor and a zero. We consider matching Peano integers against two patterns (written here in nested form): `‘Z ()` and `‘S ‘S ()`. (Recall that `()` in patterns means “match anything.”) The topmost slice matches the first pattern directly. The middle slice is partial after a single `‘S`: it fails to match the `‘Z ()` pattern (we elide that arrow for visual clarity) and *partially* matches the second pattern. The middle slice is insufficiently deep, indicating neither success nor failure. The bottommost slice matches the second pattern completely; although it is also a partial slice, it is deep enough that we can assert it would match that pattern regardless of how it might be further expanded. In general, there is always a point at which we can stop slicing: patterns are of fixed, finite depth, so every slice fixes as either compatible or incompatible after a certain depth.

Other than the additional concern of partial slices, type compatibility is much like value compatibility. It uses a constructive failure model, allowing for the incompatibility premise in Onion Value Right and ensuring well-foundedness; the handling of onions and pattern conjunctions is also the same. Due to this similarity, readers may find a review of Section 3.2.1 helpful at this point.

3.3.5 Matching

Type matching directly parallels expression matching. As in the evaluation system, type matching propagates the result of compatibility and uses the $\circledcirc$ place to enforce left precedence of dispatch. Matching $\alpha_0\ \alpha_1\ {}^{\circledcirc}_{V_0}\!\rightsquigarrow_{V_1}\ \alpha_2\backslash C'$ is defined as the least relation satisfying the clauses in Figure 3.12; matching is also equivalence-parametric.

Partial

$$\dfrac{\nexists\tau.\tau <: \alpha_0 \in V}{\alpha_0\ {}^{◐}_{V}\!\preceq^{\emptyset}_{V'}\ \alpha'_0}$$

Empty Onion

$$\dfrac{\tau <: \alpha_0 \in V \qquad () <: \alpha'_0 \in V' \qquad F = \{\alpha_0 <: \alpha'_0\}}{\alpha_0\ {}^{●}_{V}\!\preceq^{F}_{V'}\ \alpha'_0}$$

Label

$$\dfrac{l\ \alpha_1 <: \alpha_0 \in V \qquad l\ \alpha'_1 <: \alpha'_0 \in V' \qquad \alpha_1\ {}^{\circledcirc}_{V}\!\preceq^{F}_{V'}\ \alpha'_1}{\alpha_0\ {}^{\circledcirc}_{V}\!\preceq^{F}_{V'}\ \alpha'_0}$$

Conjunction Pattern

$$\dfrac{\alpha'_1 \,\&\, \alpha'_2 <: \alpha'_0 \in V' \qquad \alpha_0\ {}^{\circledcirc_1}_{V}\!\preceq^{F_1}_{V'}\ \alpha'_1 \qquad \alpha_0\ {}^{\circledcirc_2}_{V}\!\preceq^{F_2}_{V'}\ \alpha'_2}{\alpha_0\ {}^{\min(\circledcirc_1,\circledcirc_2)}_{V}\!\preceq^{F_1\cup F_2}_{V'}\ \alpha'_0}$$

Onion Value Left

$$\dfrac{\alpha_1 \,\&\, \alpha_2 <: \alpha_0 \in V \qquad \alpha_1\ {}^{\circledcirc}_{V}\!\preceq^{F}_{V'}\ \alpha'_0 \qquad \circledcirc \neq ○}{\alpha_0\ {}^{\circledcirc}_{V}\!\preceq^{F}_{V'}\ \alpha'_0}$$

Onion Value Right

$$\dfrac{\alpha_1 \,\&\, \alpha_2 <: \alpha_0 \in V \qquad \alpha'_1 \,\&\, \alpha'_2 <: \alpha'_0 \notin V' \qquad \alpha_1\ {}^{○}_{V}\!\preceq^{F'}_{V'}\ \alpha'_0 \qquad \alpha_2\ {}^{\circledcirc}_{V}\!\preceq^{F}_{V'}\ \alpha'_0}{\alpha_0\ {}^{\circledcirc}_{V}\!\preceq^{F}_{V'}\ \alpha'_0}$$

Label Mismatch

$$\dfrac{l\ \alpha_1 <: \alpha_0 \in V \qquad \tau <: \alpha'_0 \in V' \qquad \tau = l'\ \alpha_2 \text{ only if } l \neq l' \qquad \tau \text{ not of the form } \alpha' \,\&\, \alpha'' \text{ or } ()}{\alpha_0\ {}^{○}_{V}\!\preceq^{\emptyset}_{V'}\ \alpha'_0}$$

Figure 3.10. Type compatibility: does a type match a pattern?

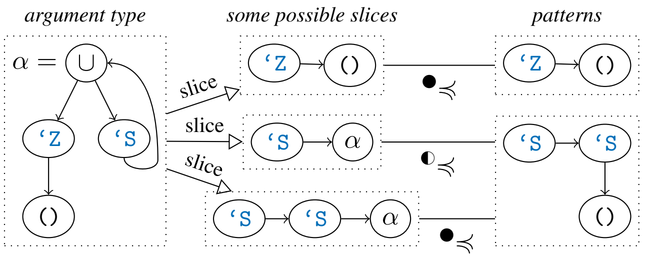

Figure 3.11. Type compatibility example

3.3.6 Constraint Closure

Constraint closure can now be defined; each step of closure represents one forward propagation of constraint information and abstractly models a single step of the operational semantics. This closure is implicitly defined in terms of an abstract polymorphism framework defined by two functions. The first, Φ, is analogous to the $\boldsymbol{\alpha}(-)$ freshening function of the operational semantics. For decidability, however, we do not want Φ to freshen *every* variable uniquely; it only performs *some* α-substitution on the constrained type. We write $\Phi(C, \alpha)$ to indicate the freshening of the variables in C. The

FUNCTION

$$\frac{(\alpha'\backslash V' \to \alpha\backslash C) <: \alpha_0 \in V_0 \qquad \alpha_1 \;{}^{\circledcirc}_{V_1}\!\!\preceq^{F}_{V'}\; \alpha'}{\alpha_0\ \alpha_1 \;{}^{\circledcirc}_{V_0}\!\!\rightsquigarrow_{V_1}\; \alpha\backslash C \cup F}$$

ONION LEFT

$$\frac{\alpha_2 \mathbin{\&} \alpha_3 <: \alpha_0 \in V_0 \qquad \alpha_2\ \alpha_1 \;{}^{\circledcirc}_{V_0}\!\!\rightsquigarrow_{V_1}\; \alpha\backslash C \qquad \circledcirc \neq \bigcirc}{\alpha_0\ \alpha_1 \;{}^{\circledcirc}_{V_0}\!\!\rightsquigarrow_{V_1}\; \alpha\backslash C}$$

ONION RIGHT

$$\frac{\begin{array}{c}\alpha_2 \mathbin{\&} \alpha_3 <: \alpha_0 \in V_0 \\ \alpha_2\ \alpha_1 \;{}^{\bigcirc}_{V_0}\!\!\rightsquigarrow_{V_1}\; \alpha'\backslash C' \qquad \alpha_3\ \alpha_1 \;{}^{\circledcirc}_{V_0}\!\!\rightsquigarrow_{V_1}\; \alpha\backslash C\end{array}}{\alpha_0\ \alpha_1 \;{}^{\circledcirc}_{V_0}\!\!\rightsquigarrow_{V_1}\; \alpha\backslash C}$$

NON-FUNCTION

$$\frac{(\alpha'\backslash V' \to \alpha\backslash C) <: \alpha_0 \notin V_0 \qquad \alpha_2 \mathbin{\&} \alpha_3 <: \alpha_0 \notin V_0}{\alpha_0\ \alpha_1 \;{}^{\bigcirc}_{V_0}\!\!\rightsquigarrow_{V_1}\; \alpha\backslash C}$$

Figure 3.12. Type application matching

TRANSITIVITY

$$\frac{\{\tau <: \alpha_1, \alpha_1 <: \alpha_2\} \subseteq C}{C \Longrightarrow^1 C \cup \{\tau <: \alpha_2\}}$$

APPLICATION

$$\frac{\begin{array}{c}\alpha_0\ \alpha_1 <: \alpha_2 \in C \\ \alpha'_0\backslash V_0 \ll \alpha_0\backslash C \qquad \alpha'_1\backslash V_1 \ll \alpha_1\backslash C \\ \alpha'_0\ \alpha'_1 \;{}^{\bullet}_{V_0}\!\!\rightsquigarrow_{V_1}\; \alpha'\backslash C' \qquad \alpha''\backslash C'' = \Phi(\alpha'\backslash C', \alpha_2)\end{array}}{C \Longrightarrow^1 C \cup V_1 \cup C'' \cup \{\alpha'' <: \alpha_2\}}$$

Figure 3.13. Type constraint closure single-step relation

additional parameter α describes the call site at which the polyinstantiation took place; this is useful for some polymorphism models. The second function, Υ, unifies type variables by producing type variable equivalences as described in Section 3.3.2. In particular, the equivalence is used to unify variables in certain cases of recursion to avoid arbitrary blow-up. The simplest polymorphism model would be a monomorphic type system given by $\Phi_{\text{MONO}}(C, \alpha) = C$ and where the relation given by $\Upsilon_{\text{MONO}}(-)$ relates each type variable to itself and no others. We discuss our concrete choice $\Phi_{\text{CR}}/\Upsilon_{\text{CR}}$ in Section 3.4.

We write $C \Longrightarrow^1 C'$ to indicate a single step of constraint closure. This relation is defined as the least such that the rules in Figure 3.13 are satisfied. This relation defines the equivalence to be used by the relations in its premises (and is not equivalence-parametric): at each constraint closure step $C \Longrightarrow^1 C'$, we take the type variable equivalence relation to be $\Upsilon(C)$. We write $C_0 \Longrightarrow^* C_n$ to indicate $C_0 \Longrightarrow^1 \ldots \Longrightarrow^1 C_n$ for some $n \geq 0$.

The operational semantics has a definition for a "stuck" expression; the type system analogue is the inconsistent constraint set, which we define as follows:

$[\![\texttt{r = neg arg}]\!]_{\text{E}} = \alpha_{\texttt{neg}}\, \alpha_{\texttt{arg}} <: \alpha_{\texttt{r}}$
$[\![\wr\texttt{x \& int -> x}\int_{\texttt{iId}}]\!]_{\text{E}} = \{\alpha_0\backslash V_0 \to \alpha_1\backslash C_1 <: \alpha_{\texttt{iId}}\}$
    where $V_0 = \{\texttt{()} <: \alpha_{\texttt{x}}, \texttt{int} <: \alpha_{13}, \alpha_{\texttt{x}} \mathbin{\&} \alpha_{13} <: \alpha_0\}$
    and $C_1 = \{\alpha_{\texttt{x}} <: \alpha_1\}$
$[\![\wr(\texttt{`True \_ -> `False ()}) \mathbin{\&} (\texttt{`False \_ -> `True ()}) \mathbin{\&} \texttt{iId}\int_{\texttt{neg}}]\!]_{\text{E}} =$
    $\{\alpha_2\backslash\{\texttt{`True ()} <: \alpha_2\} \to \alpha_3\backslash\{\texttt{`False ()} <: \alpha_3\} <: \alpha_4,$
    $\alpha_5\backslash\{\texttt{`False ()} <: \alpha_5\} \to \alpha_6\backslash\{\texttt{`True ()} <: \alpha_6\} <: \alpha_7,$
    $\alpha_4 \mathbin{\&} \alpha_7 <: \alpha_8, \alpha_8 \mathbin{\&} \alpha_{\texttt{iId}} <: \alpha_{\texttt{neg}}\}$

Figure 3.14. Worked example

Definition 3.3 (Inconsistency)**.** A constraint set C is *inconsistent* iff, under the equivalence $\Upsilon(C)$, there exists some $\alpha_0\ \alpha_1 <: \alpha_2 \in C$ and $\alpha'_0\backslash V_0 \ll \alpha_1\backslash C$ and $\alpha'_1\backslash V_1 \ll \alpha_1\backslash C$ such that $\alpha'_0\ \alpha'_1 \;{}^{\bullet}_{V_0}\!\!\rightsquigarrow_{V_1}\; \alpha'\backslash C'$. A constraint set which is not inconsistent is *consistent*.

Informally, inconsistency captures the cases in which an expression can get stuck. Here, the only such case is when a compound function cannot match its argument. Since a slice represents a set of values which may arrive at runtime, we declare a constraint set inconsistent if the type of a compound function does not match some slice of its argument.

Given the above, we can now define what it means for a program to be type correct:

Definition 3.4 (Typechecking)**.** A closed expression e *typechecks* iff $[\![e]\!]_{\text{E}} = \alpha\backslash C$ and $C \Longrightarrow^* C'$ implies that C' is *consistent*.

This definition is also implicitly parametric in Φ/Υ.

3.3.7 A worked example

Here we work through a small example to clarify how the relations interact.[3] Recall the overloaded negation example from Section 2.3. Consider the code `r = neg arg` where `arg` may be either an int or a boolean, depending on user input. Rather than redefine `neg` as in that example, we use `iId`, the integer identity function, in place of integer negation for simplicity. (To present this simple example, we extend the above type system with the atomic type `int`.) See Figure 3.14 for the A-translation and initial alignment. This means that the constraint set corresponding to the program is $C = \{\alpha_{\texttt{neg}}\, \alpha_{\texttt{arg}} <: \alpha_{\texttt{r}}, \texttt{int} <: \alpha_{\texttt{arg}}, \texttt{`True}\ \alpha_{eo} <: \alpha_{\texttt{arg}}, \texttt{()} <: \alpha_{eo}, \texttt{`False ()} <: \alpha_{\texttt{arg}}, \alpha_0\backslash V_0 \to \alpha_1\backslash C_1 <: \alpha_{\texttt{iId}}, \alpha_2\backslash\{\texttt{`True ()} <: \alpha_2\} \to \alpha_3\backslash\{\texttt{`False ()} <: \alpha_3\} <: \alpha_4, \alpha_5\backslash\{\texttt{`False ()} <: \alpha_5\} \to \alpha_6\backslash\{\texttt{`True ()} <: \alpha_6\} <: \alpha_7, \alpha_4 \mathbin{\&} \alpha_7 <: \alpha_8, \alpha_8 \mathbin{\&} \alpha_{\texttt{iId}} <: \alpha_{\texttt{neg}}\}$. We also take V_f to be the largest set such that $\alpha_f\backslash V_f \ll \alpha_{\texttt{neg}}\backslash C$ (other slices exist, but we do not need them for this presentation), and we will use the same names to refer to the variables in the constraint set and in the slice.

To perform closure on this constraint set, we apply the Application closure rule on $\{\alpha_{\texttt{neg}}\, \alpha_{\texttt{arg}} <: \alpha_{\texttt{r}}\}$, for which we need to find a slice $\alpha'\backslash V' \ll \alpha_{\texttt{arg}}\backslash C$. There are five such slices (up to alpha equivalence), and we will focus on

[3] For ease of presentation, we will use l `()` $<: \alpha$ as shorthand for the constraints $\{l\ \alpha' <: \alpha, \texttt{()} <: \alpha'\}$ where α' does not occur elsewhere in the overall constraint set.

three of them: $\alpha\backslash\{\texttt{int} <: \alpha\}$, $\alpha\backslash\{\texttt{`True ()} <: \alpha'\}$, and $\alpha\backslash\{\texttt{`True}\ \alpha_{eo} <: \alpha\}$. The cases for `False are analogous.

Letting $V = \{\texttt{int} <: \alpha\}$, we have $\alpha_4\ \alpha \underset{V_f}{\bullet} \rightsquigarrow_V \alpha''\backslash C''$ because we can show $\alpha \underset{V}{\circ} \preceq^{F'}_{V'} \alpha_2$ (where $V' = \{\texttt{`True ()} <: \alpha_2\}$) by inspection. Thus we can apply the Onion Value Right rule (and again by similar logic on α_5). If we can show that $\alpha \underset{V}{\bullet} \preceq^{F}_{V_0} \alpha_0$, we can satisfy the second premise of the Function match rule and then the third (and last) premise of the Application closure rule; this is straightforward using the Conjunction Pattern, Empty Onion, and (hypothetical) Integer compatibility rules (in that order).

Letting $V = \{\texttt{`True ()} <: \alpha'\}$, we can apply Onion Left and Function match rules, followed by the Label and Empty Onion compatibility rules, to satisfy the third premise of the Application closure rule. Letting $\{\texttt{`True}\ \alpha_{eo} <: \alpha\}$, however, there is only a partial match and we do not satisfy the Application closure rule.

3.3.8 Formal Properties

We now state formal assertions regarding our type system. Proofs of these statements appear in the supplementary appendices. We begin with soundness which, as discussed above, we prove by simulation rather than subject reduction; the A-translation of the program and the careful alignment between each of the relations in the system makes simulation a natural choice. We require the polyinstantiation function Φ to be *simulation-preserving* in that it maintains this simulation and henceforth discuss only simulation-preserving Φ. We then state soundness as follows:

Theorem 1 (Soundness)**.** If $e \longrightarrow^* e'$ where e' is stuck then e does not typecheck.

A proof appears in Appendix A.

The remainder of this section shows that typechecking is decidable. Our strategy is to demonstrate that the constraint closure of a finite constraint set C forms a subset inclusion lattice and that it is sufficient (and computable) to check the consistency of the top of that lattice. Because constraint closure is parametric in the polymorphism model, we must limit Φ to introduce finitely many variables into any produced constraint set given an initial constraint set. We define Φ with this property to be *finitely freshening*; the full definition appears in Appendix C. Henceforth, we discuss only finitely freshening Φ. We now state a critical finiteness lemma:

Lemma 3.5. For any e, there exists a finite set of constraints C'' such that, if $[\![e]\!]_{\mathrm{E}} = \alpha\backslash C$ and $C \Longrightarrow^* C'$, then $C' \subseteq C''$.

A proof appears in Appendix B.

Because of this finiteness and because constraint closure is monotonic on the constraint set (by inspection of the constraint closure rules), we can show that every constraint closure path is of finite length. We write $C \Longrightarrow^! C'$ to indicate that $C \Longrightarrow^* C'$ and that, for every C'' such that $C' \Longrightarrow^1 C''$, we have $C' \cong C''$ (where $\cong$ is defined by $\Upsilon(C')$). We then state:

Lemma 3.6 (Convergence)**.** For any finite C, there exists some C' such that $C \Longrightarrow^! C'$.

This lemma is also proven in Appendix B.

The above shows that, for a given program, constraint closure will converge after finitely many steps. For this result to be applicable, however, we would like every path to converge to the *same* set: the top of our closure lattice. This proof requires that Υ is *equivalence monotone*, which is true when $C \subseteq C'$ implies that $\Upsilon(C) \subseteq \Upsilon(C')$ – that is, a constraint superset induces monotonically more equivalences. Henceforth, we consider only equivalence monotone Υ. We assert this confluence property as follows:

Lemma 3.7 (Confluence)**.** If $C \Longrightarrow^* C'$ and $C \Longrightarrow^* C''$ then there exists some C''' such that $C' \Longrightarrow^* C'''$ and $C'' \Longrightarrow^* C'''$.

This is to say that, although multiple constraint closure paths exist, any two constraint closure paths with a common origin can meet again in the future. The proof of this lemma appears in Appendix B and is relatively simple to show; the key observation is that the set of constraint closure premises that hold increases monotonically as the constraint set grows.

These assertions show that constraint closure for a given program forms a subset inclusion lattice. Because inconsistency is also monotonic with constraint set growth, we need only check the top constraint set in this lattice for inconsistency. It must be possible to decide the top of that lattice:

Lemma 3.8. $C \Longrightarrow C'$ is decidable for finite C.

This decidability is proven in Appendix B and largely follows from Lemma 3.5 above.

Using the above, we can give an equivalent formulation of typechecking:

Lemma 3.9. An expression e typechecks if and only if $[\![e]\!]_{\mathrm{E}} = \alpha\backslash C$, $C \Longrightarrow^! C'$, and C' is consistent.

This lemma follows immediately from the above definitions and statements. We then use this alternate formulation to state our second theorem:

Theorem 2 (Decidability)**.** Typechecking is decidable.

Appendix B contains a proof of the above which follows by inspection of each step of the equivalent formulation.

In summary, TinyBang's type system is sound and decidable if the polymorphism model is finitely freshening, simulation-preserving, and equivalence monotone. We call such a polymorphism model *well-behaved*. The proof that $\Phi_{\mathrm{MONO}}/\Upsilon_{\mathrm{MONO}}$ is well-behaved is simple, but the model is not very useful. We now discuss a much more expressive well-behaved polymorphism model we use in practice.

3.4 A Useful Polymorphism Model: $\Phi_{\mathrm{CR}}/\Upsilon_{\mathrm{CR}}$

Our implementation of TinyBang uses a polymorphism model $\Phi_{\mathrm{CR}}/\Upsilon_{\mathrm{CR}}$ which provides a good trade-off between expressiveness and efficiency. We choose this model over more traditional approaches such as `let`-bound polymorphism because such models lack the needed expressiveness needed for common object-oriented programming patterns. For example, consider:

```
let factory x = (‘get _ -> x) in
let mkPair f = ‘A (f 0) & ‘B (f ()) in
mkPair factory
```

Here, `factory` creates simple polymorphic value container. In a `let`-bound model, the type given to `f` within `mkPair` is monomorphic; thus, the best type we can assign to the argument type of `f` where it is used is the empty onion (and not `int`). The novel `seal` function in Section 2 also fails to typecheck using `let`-bound polymorphism because `obj` would be mono-typed from within the catch-all message handler added by the sealing routine.

To provide sufficient precision, the TinyBang type system uses call site polymorphism: rather than polyinstantiating variables where they are named, we polyinstantiate functions when the function is invoked. We thus view every function as having a polymorphic type. In order to ensure maximal polymorphism, every variable bound by the body of the function should be polyinstantiated. This approach is inspired by flow analyses [1, 22] and has previously been ported to a type constraint context [24]. A program can have an unbounded number of function call sequences so some compromise must be made for recursive programs; a standard solution to this problem is to 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.

Our approach is to conservatively model the runtime stack (as in [14]). In particular, we abstract call sequences as regular expressions, and a single type variable associated with a regular expression will represent all runtime variables whose call strings match the regular expression. This model is powerful enough to assign useful types to the above code as well as all of the code in the overview. The full definition of our $\Phi_{\mathrm{CR}}/\Upsilon_{\mathrm{CR}}$ function appears in Appendix C, along with a proof that this polymorphism model is well-behaved as discussed in the previous section.

4. Related Work

We believe that TinyBang is the first language with sufficient features for encoding unrestricted typed object extension and messaging. Several features of TinyBang are critical to the success of this encoding, including general asymmetric record append, dependently-typed case results, slices to refine the types of case inputs, and a polymorphism model that generalizes let-polymorphism to support polymorphic object factories.

TinyBang's object resealing is inspired by the Bono-Fisher object calculus [7], in which mixins and other higher-order object transformations are written as functions. Objects in this calculus must be "sealed" before they are messaged; unlike our resealing, sealed objects cannot be extended. Some related works relax this restriction but add others. In [20], for instance, the power to extend a messaged object comes at the cost of depth subtyping. In [5], depth subtyping is admitted but the type system is nominal rather than structural.

Typed multimethods [15] perform a dispatch similar to TinyBang's dispatch on compound functions, but multimethod dispatch is *nominally* typed while compound function dispatch is *structurally* typed. First-class cases [6] allow composition of case branches much like TinyBang. In [6], however, general case concatenation requires case branches to be written in CPS and requires a phase distinction between constructing a case and matching with it. TinyBang has a form of dependent type which allows different case branches to return different types; this generalizes the expressiveness of conditional constraints [3, 18] and is related to support for typed first-class messages *a la* [17, 18] – first-class messages are just labeled data in our encoding.

TinyBang shares several features and goals with CDuce [10]: both aim to be flexible languages built around constraint subtyping. CDuce supports dependently-typed case results as we do, but it does not have slicing on the pattern input side because the pattern grammar cannot bind refined types. CDuce lacks a typed record append operation and so the object encodings of this paper are not possible there. CDuce takes a local type inference approach; this has the advantage of being modular but the disadvantage of not being complete, requiring type annotations in some cases, and the decidability of their inference algorithm remains open.

TinyBang's onions are unusual in that they are a form of record supporting typed asymmetric concatenation. The bulk of work on typing record extension addresses symmetric concatenation only [19]. Standard typed record-based encodings of inheritance [8] avoid the problem of typing first-class concatenation by reconstructing records rather than extending them, but this requires the superclass to be fixed statically. A combination of conditional constraints and row types can be used to type record extension [18]; TinyBang uses a different approach that does not need row types.

Our approach to parametric polymorphism in Section 3.4 is based on flow analysis [22, 24]. In the aforecited, polymorphic contours are permanently distinct once instantiated. In TinyBang, contours are optimistically generated and then merged when a call cycle is detected; this allows more expressive polymorphism since call cycles don't need to be conservatively approximated up front. Contours are merged by injection into a restricted space of regular expressions over call strings, an approach that follows program analyses [14].

We believe the technical development of this paper is novel in several interesting ways. Proofs of type soundness usually proceed by subject reduction, but we can directly align the operational semantics with the type inference algorithm in a simulation relation to produce a direct type soundness proof. As such, TinyBang's type system can be viewed as a hybrid between a flow analysis [22] or abstract interpretation [11] and a traditional type system. Without such a technique, type soundness would be very challenging given the expressiveness needs of our type system. There is no reasonable regular tree or other form of denotational type model [4, 9, 23] of TinyBang given the flexibility of pattern matching. For example, polymorphism in TinyBang cannot be convex in the sense of [9] since our pattern matching syntax is expressive enough to distinguish all top-level forms. Another novel technical contribution is the nearly perfect unification of expression, pattern, and type syntax, putting them all in very similar A-normal forms. This syntax unification greatly cleans up the metatheory.

One novelty of the type system formalization is how we deal with the well-known *union alignment problem*. The root of this problem is that there may not be a consistent view of which branch of a union type was taken in different pattern match clauses. All standard union type systems must make a compromise union elimination rule; [12] contains a review of existing approaches. Our novel solution is the notion of a *slice* in Section 3.3.3 which can be viewed as a (locally) complete notion of union elimination.

5. Conclusions

Mixins, dynamic extensions, and other flexible object operations are making an impact in dynamically-typed object-oriented scripting languages, and this flexibility is important to bring to statically-typed languages. To this end, we presented TinyBang, a core language with a static type inference system that types such flexible operations without onerous false type errors or the need for manual programmer annotation. We believe TinyBang solves a longstanding open problem: it infers types for object-oriented programs without compromising the expressiveness of object subtyping or of object extension. This is possible due to a combination of novel features in TinyBang: asymmetric concatenation, first-class dependently-typed cases, slicing for type refinement, and a flexible non-let-based polymorphism model.

TinyBang is proved type sound; the proofs are found in the supplementary appendices. We have implemented the type inference algorithm and interpreter for TinyBang to provide a cross-check on the soundness of our ideas; the implementation can be downloaded from [13]. Type inference is decidable but not provably polynomial for the same reason that let-polymorphism is not: artificial programs exist which will exhibit exponential runtimes. However, all of the examples in Section 2 typecheck in our implementation without exponential blowup, and we have designed the polymorphism model with practical performance in mind.

We do not expect programmers to write in TinyBang. Instead, programmers would write in BigBang, a language we are developing which includes syntax for objects, classes, and so on, and which will de-sugar to TinyBang.

TinyBang infers extremely precise types, especially in conjunction with the context-sensitive polymorphism model described in Section 3.4. While powerful, these types are by nature difficult to read and defy modularization. Addressing the problem of readability is beyond the scope of this paper but is part of our broader research agenda.

References

[1] O. Agesen. The cartesian product algorithm. In *ECOOP*, 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, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *POPL 21*, pages 163–173, 1994.

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

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

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

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

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

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

[10] G. Castagna, K. Nguyen, Z. Xu, H. Im, S. Lenglet, and L. Padovani. Polymorphic functions with set-theoretic types. part 1: Syntax, semantics, and evaluation. In *POPL*, 2014.

[11] P. Cousot. Types as abstract interpretations, invited paper. In *POPL*, Jan. 1997.

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

[13] P. H. Menon, Z. Palmer, A. Rozenshteyn, and S. Smith. Tinybang implementation, Jan 2014. `http://pl.cs.jhu.edu/big-bang/tiny-bang_2014-03-01.tgz`.

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

[15] T. D. Millstein and C. Chambers. Modular statically typed multimethods. In *ECOOP*, pages 279–303. Springer-Verlag, 1999.

[16] M. H. A. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43(2), 1942.

[17] S. Nishimura. Static typing for dynamic messages. In *POPL*, 1998.

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

[19] D. Rémy. Type inference for records in a natural extension of ML. In *Theoretical Aspects Of Object-Oriented Programming*. MIT Press, 1994.

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

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

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

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

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

$$
\begin{array}{cccll}
() & \preccurlyeq_M & () & & \\
l\ x & \preccurlyeq_M & l\ \alpha & \text{iff} & x \preccurlyeq_M \alpha \\
x_1 \,\&\, x_2 & \preccurlyeq_M & \alpha_1 \,\&\, \alpha_2 & \text{iff} & x_1 \preccurlyeq_M \alpha_1 \text{ and } x_2 \preccurlyeq_M \alpha_2 \\
\phi \texttt{ -> } e & \preccurlyeq_M & \alpha'\backslash V \rightarrow \alpha\backslash C & \text{iff} & \phi \preccurlyeq_M \alpha'\backslash V \text{ and } e \preccurlyeq_M \alpha\backslash C \\
\hline
x & \preccurlyeq_M & \alpha & \text{iff} & M(x) = \alpha \\
\hline
x = v & \preccurlyeq_M & \tau <: \alpha & \text{iff} & x \preccurlyeq_M \alpha \text{ and } v \preccurlyeq_M \tau \\
x_2 = x_1 & \preccurlyeq_M & \alpha_1 <: \alpha_2 & \text{iff} & x_1 \preccurlyeq_M \alpha_1 \text{ and } x_2 \preccurlyeq_M \alpha_2 \\
x_3 = x_1\ x_2 & \preccurlyeq_M & \alpha_1\ \alpha_2 <: \alpha_3 & \text{iff} & \forall i \in \{1..3\}.\, x_i \preccurlyeq_M \alpha_i \\
\hline
x = \mathring{v} & \preccurlyeq_M & \tau <: \alpha & \text{iff} & x \preccurlyeq_M \alpha \text{ and } \mathring{v} \preccurlyeq_M \tau \\
\hline
e & \preccurlyeq_M & C & \text{iff} & \forall s \in e.\, \exists c \in C.\, s \preccurlyeq_M c \\
e & \preccurlyeq_M & \alpha\backslash C & \text{iff} & \text{RV}(e) \preccurlyeq_M \alpha \text{ and } e \preccurlyeq_M C \\
\hline
\phi & \preccurlyeq_M & V & \text{iff} & \forall x = \mathring{v} \in e.\, \exists c \in V.\, x = \mathring{v} \preccurlyeq_M c \\
\phi & \preccurlyeq_M & \alpha\backslash V & \text{iff} & \text{RV}(\phi) \preccurlyeq_M \alpha \text{ and } \phi \preccurlyeq_M V
\end{array}
$$

Figure A.1. Simulation relation

A. Proof of Soundness

In this appendix, we give the proof for Theorem 1. As stated in Section 3.3, our strategy is to establish a simulation relation between the original program and the constraint set obtained by initial alignment. We then show that this simulation is preserved throughout constraint closure and observe that stuck programs are simulated by inconsistent constraint sets.

A.1 Initial Alignment

We begin by defining this simulation relation over each construct in the evaluation grammar and its corresponding construct in the type grammar. We add a third place to this relation to represent the variable mapping $x \rightarrow \alpha$ which aligns value variables to their representative type variables; we use M to range over these functions. The simulation relation is defined as follows:

Definition A.1. The simulation relation $\preccurlyeq_M$ is defined as the least relation satisfying the rules in Figure A.1. The notation $x \preccurlyeq \alpha$ is used to denote $\exists M.\, x \preccurlyeq_M \alpha$; similar notation holds for other grammatical constructs.

Using this relation, we can now demonstrate that the initial alignment of an expression produces a constrained type which simulates it.

Lemma A.2. If e is closed then $e \preccurlyeq [\![e]\!]_{\text{E}}$.

Proof. This is to say that, for some M, $e \preccurlyeq_M [\![e]\!]_{\text{E}}$. Suppose that, for all s, we have $s \preccurlyeq_M \alpha\backslash\{c\}$ when $[\![s]\!]_{\text{S}} = \alpha\backslash c$. Then this lemma can be proven by induction on the length of e. So it suffices to show simulation for clauses.

Consider the case where we must show $\mathbb{Z} \preccurlyeq$ `int` and $x_0 \preccurlyeq_M \overset{\star}{\alpha}_0$. The former is true by Definition A.1. The latter is true by construction: we let M be the function mapping each value variable x_i to its corresponding fresh variable $\overset{\star}{\alpha}_i$. The same argument holds for all non-application clauses.

For functions, we must show that $\phi \preccurlyeq_M [\![\phi]\!]_{\text{P}}$ and that $e' \preccurlyeq_M [\![e']\!]_{\text{E}}$. The latter is true by induction on the size of e'. The former is shown in much the same fashion as the above. By induction on the length of ϕ, it suffices to show that for all $x = \mathring{v}$ we have $x = \mathring{v} \preccurlyeq_M \alpha\backslash\{\tau <: \alpha\}$ when $[\![x = \mathring{v}]\!]_{\mathring{\text{S}}} = \alpha\backslash\tau <: \alpha$. This can be shown in the same fashion as the non-application clauses above. □

Here, *closed* has the usual meaning: a variable is not used before it is defined (either by a clause or by a pattern).

ATOMIC
$$\frac{x' = v \in E' \qquad v \text{ not of the form } l\ x_1 \text{ or } x_1 \,\&\, x_2}{x\backslash[x = v] \lll x'\backslash E'}$$

LABEL
$$\frac{x_0 = l\ x_1 \in E' \qquad x_1\backslash E \lll x_1'\backslash E'}{x_0\backslash E \,\|\, [x_0 = l\ x_1] \lll x_0'\backslash E'}$$

ONION
$$\frac{\begin{array}{c} x_0 = x_1 \,\&\, x_2 \in E' \\ x_1\backslash E_1 \lll x_1'\backslash E_1' \qquad x_2\backslash E_2 \lll x_2'\backslash E_2' \end{array}}{x_0\backslash E_1 \,\|\, E_2 \,\|\, [x_0 = x_1 \,\&\, x_2] \lll x_0'\backslash E'}$$

Figure A.2. Deep copy relation

A.2 Deep Copying

In proving soundness, we want to show that each small step of the expression has a corresponding constraint closure rule which preserves the simulation. We accomplish this by showing a one-to-one correspondence between steps of the evaluation system and steps of the type system. We are already prepared to do this for most relations – each value compatibility premise corresponds to a type compatibility premise, for instance – but there is no evaluation system relation corresponding to the slicing relation in the type system. We define such a corresponding relation here to considerably simplify the overall proof.

Slicing refines a type by union elimination, essentially copying a union-free subtype into fresh variables. We require an evaluation system relation which aligns with this behavior as closely as possible. Because values during evaluation are already union-free – variables may be defined only once in an expression – this evaluation system relation need only perform the copying step. We therefore define a *deep copy* relation; this relation is later inserted into the operational semantics to make it better align with the constraint closure. Deep copying is written $x\backslash E \lll x'\backslash E'$ when x in environment E is a deep copy of x' in environment E'. We define this relation as the least relation satisfying the rules in Figure A.2.

Definition A.3. $x'\backslash E' \lll x\backslash E$ is the least relation defined by the rules in Figure A.2. Deep copies are are *disjoint* iff $x = v \in E'$ implies that x does not appear in E.

Henceforth we only consider disjoint, minimal deep copies. It should be noted that E' may not be closed – variables appearing within a function are not copied – but the openness of E' is not an impediment to this soundness proof.

$$
\begin{array}{cccll}
\texttt{()} & \approx_M & \texttt{()} & & \\
l\ x & \approx_M & l\ \alpha & \text{iff} & x \approx_M \alpha \\
x_1 \,\&\, x_2 & \approx_M & \alpha_1 \,\&\, \alpha_2 & \text{iff} & x_1 \approx_M \alpha_1 \text{ and } x_2 \approx_M \alpha_2 \\
\phi \texttt{ -> } e & \approx_M & \alpha'\backslash V \rightarrow \alpha\backslash C & \text{iff} & \phi \texttt{ -> } e \preccurlyeq_M \alpha'\backslash V \rightarrow \alpha\backslash C \\
\hline
x & \approx_M & \alpha & \text{iff} & M(x) = \alpha \text{ and } x \neq x' \\
 & & & & \qquad \iff M(x') \neq \alpha \\
\hline
x \texttt{=} v & \approx_M & \tau <: \alpha & \text{iff} & x \approx_M \alpha \text{ and } v \approx_M \tau \\
x_2 \texttt{=} x_1 & \approx_M & \alpha_1 <: \alpha_2 & \text{iff} & x_1 \approx_M \alpha_1 \text{ and } x_2 \approx_M \alpha_2 \\
x_3 \texttt{=} x_1\ x_2 & \approx_M & \alpha_1\ \alpha_2 <: \alpha_3 & \text{iff} & \forall i \in \{1..3\}.\, x_i \approx_M \alpha_i \\
\hline
x \texttt{=} \mathring{v} & \approx_M & \tau <: \alpha & \text{iff} & x \approx_M \alpha \text{ and } \mathring{v} \approx_M \tau \\
\hline
\overset{n}{\overrightarrow{s}} & \approx_M & \overset{n\sqcup}{c} & \text{iff} & \forall 1 \leq i \leq n.\, s_i \approx_M c_i \\
e & \approx_M & \alpha\backslash C & \text{iff} & \mathrm{RV}(e) \approx_M \alpha \text{ and } e \approx_M C \\
\hline
\overset{n}{\overrightarrow{x_\square \texttt{=} \mathring{v}_\square}} & \approx_M & \overset{n\sqcup}{c} & \text{iff} & \forall 1 \leq i \leq n.\, x_i \texttt{=} \mathring{v}_i \approx_M c_i \\
\phi & \approx_M & \alpha\backslash V & \text{iff} & \mathrm{RV}(\phi) \approx_M \alpha \text{ and } \phi \approx_M V
\end{array}
$$

Figure A.3. Shallow bismulation relation

Note that in contrast with slicing, deep copies do not have a Leaf rule; the deep copy relation always copies the entire value. This is possible because values in TinyBang cannot be cyclic. More formally, we say a variable is *well-founded* iff either it has an atomic value (an integer, empty onion, or simple function) or if it refers only to other well-founded variables. This leads us to the following lemma.

Lemma A.4. Every variable x in a closed expression e has a finite deep copy.

Proof. By induction on the length of e, we know that x is well-founded. By inspection, deep copy is defined for every form of value. Thus, a finite deep copy proof can be constructed for any such x. □

A.3 Bisimulation of Deep Copying and Slicing

Our objective is to show that each evaluation step has a corresponding type system step; we can now show how deep copying and slicing correspond to each other. But we must show a stronger property than preservation of the simulation relation from Definition A.1. To understand why, recall from Section 3.3 that type compatibility is unsound if the input type contains unions. The simulation relation does not guarantee that the type side is union-free, so it will be insufficient in our proofs regarding simulation. We therefore define a *shallow bisimulation* relation to capture the union-free nature of the type. The shallow bisimulation is quite powerful: it guarantees that there is a one-to-one correspondence between the two systems (except under function bodies: hence "shallow"). This relation is as given as follows:

Definition A.5. The shallow bisimulation relation $\approx_M$ is defined as the least relation satisfying the rules in Figure A.3. The notation $x \approx \alpha$ is used to denote $\exists M.\, x \approx_M \alpha$; similar notation holds for other grammatical constructs.

The shallow bisimulation essentially preserves the type refinement performed by a slice. While we cannot keep this refinement indefinitely – such a type system would be undecidable – we use the shallow bisimulation to show that, for our purposes, compatibility holds in the evaluation system iff compatibility holds in the type system. This condition is necessary both for soundness and for the dependent type dispatch of TinyBang functions. Of course, shallow bisimulation implies simulation, allowing us to shed this information once it is no longer necessary:

Lemma A.6. If $e \approx C$ then $e \preccurlyeq C$.

Proof. By induction on the size of the proof of $e \approx C$. □

We can now show that deep copying and slicing together not only preserve simulation but also create a shallow bisimulation:

Lemma A.7. Let $x' \preccurlyeq_{M'} \alpha'$ and $E' \preccurlyeq_{M'} C'$. Suppose that $x\backslash E \lll x'\backslash E'$ for some (potentially open) E. Then there exists some α and V such that $x \approx_M \alpha$, $E \approx_M V$, and $\alpha\backslash V \ll \alpha'\backslash C'$. Furthermore, $M \supseteq M'$.

Proof. By induction on the size of the proof of $x\backslash E \lll x'\backslash E'$. This induction is well-founded for closed E' by Lemma A.4. For each deep copy proof rule, we select the corresponding slice proof rule (e.g. the deep copy label rule and the slice label rule). Because $E' \preccurlyeq_M C'$, each premise on E' (e.g. $x'_0 \texttt{=} l\ x'_1 \in E'$) is simulated by a premise on C' (e.g. $\alpha'_0 \texttt{=} l\ \alpha'_1 \in C'$). Subproofs (e.g. $\alpha_1\backslash V \ll \alpha'_1\backslash C'$ are proven by induction on the size of the proof.

Fresh type variables are selected to correspond to those variables used in E (e.g. $x_0 \preccurlyeq_M \alpha_0$); the resulting mapping M is thus exactly M' with additional entries for these variables. Because both x_0 and α_0 are fresh, these new entries respects the bijection requirement on M. We then select a V with the necessary property (e.g. $l\alpha_1 <: \alpha_0 \in V$) to shallowly bisimulate the extension to the deep copy's environment (e.g. $E = [\ldots, x_0 \texttt{=} l\ x_1]$) and show by Definition A.5 that bisimulation holds under M. In particular, the bisimulation holds because we select exactly one new constraint at each step and because we do not descend into functions. This process applies to each of the deep copy proof rules and is applied inductively to construct a proof of slicing.

Due to our selection of fresh type variables, the slice is well-formed. The slice is minimal because the atomic case of deep copies does not admit superfluous clauses and V simulates those clauses. The slice is well-formed because well-formed environments E' only define each variable once and the deep copy E is well-formed if E' is well-formed. □

A.4 Compatibility and Matching

We now show that the compatibility relation preserves this shallow bisimulation. In one way, this proof is somewhat simpler: compatibility introduces no fresh variables, so a single variable map M is sufficient. But compatibility introduces another challenge. The Onion Right rule relies on a premise of *incompatibility* of all potential bindings; we must not only show that compatibility preserves bisimulation but that incompatibility does as well. This was our motivation for constructing the bisimulation: because the slice models the value so precisely, we have a perfect alignment between the proof trees.

Lemma A.8. Suppose for some (possibly open) E that $E \approx_M \alpha_0 \backslash V$, $\phi \approx_M \alpha'_0 \backslash V'$, $x_0 \approx_M \alpha_0$, and $x'_0 \approx_M \alpha'_0$. Then for all B and $\odot$, $x_0 \; {}^{\odot}_{E}{\preceq}^{B}_{\phi} \; x'_0$ if and only if there exists some F such that $B \approx_M F$ and $\alpha_0 \; {}^{\odot}_{V_0}{\preceq}^{F}_{V'_0} \; \alpha'_0$.

Proof. Inspection of Figures 3.3 and 3.10 reveal that value compatibility and full type compatibility are isomorphic up to the shallow bisimulation. In particular, whenever a value compatibility rule applies to evaluation constructs, a type compatibility rule applies to the bisimulation of those constructs (and vice versa).

We begin by showing the forward implication for some fixed $\odot$. In the case where the Label rule of value compatibility applies, $x_2 = \texttt{`A}\ x_1 \in E$ and $x'_2 = \texttt{`A}\ x'_1 \in \phi$ and $x_1 \; {}^{\odot}_{E}{\preceq}^{B}_{\phi} \; x'_1$. By shallow bisimulation, we have that $\alpha = \alpha_2 \approx_M x_2$, that $\texttt{`A}\ \alpha_1 <: \alpha_2 \in V$, and corresponding facts hold for the pattern type. This and induction on the size of the proof of value compatibility give us that the label rule of type compatibility applies. A similar approach is used for each other rule in the relation, completing the forward implication.

The reverse implication – that a proof of type compatibility evidences a proof of value compatibility – is shown in the same fashion: using a fixed $\odot$, we proceed in each rule by shallow bisimulation and induction on the size of the proof tree. Note that nothing is shown for type compatibility proofs using ◐ as this is not necessary to satisfy the lemma statement. □

We can likewise demonstrate preservation of bisimulation by the respective matching relations. We state this property as follows:

Lemma A.9. Suppose for some (possibly open) E_0 and E_1 such that $E_0 \approx_M \alpha_0 \backslash V_0$, $E_1 \approx_M \alpha_1 \backslash V_1$, $x_0 \approx_M \alpha_0$, and $x_1 \approx_M \alpha_1$. Let $x_0 = \text{RV}(E_0)$ and $x_1 = \text{RV}(E_1)$. Let $E = E_0 \,\|\, E_1$. Then for any $\odot$, $x_0\ x_1 \overset{\odot}{\rightsquigarrow}_E\ e$ if and only if there exists some $\alpha' \backslash C$ such that $e \approx_M \alpha' \backslash C$ and $x_0\ x_1 \; {}^{\odot}_{V_0}{\rightsquigarrow}_{V_1} \; \alpha' \backslash C$.

Proof. By the same strategy as Lemma A.8, using Lemma A.8 to show the compatibility premises. □

A.4.1 An Alternate Operational Semantics

The above lemmas show that we can establish and preserve a bisimulation starting with a deep copy and continuing through matching. But the operational semantics of TinyBang are not defined to make use of deep copying. We construct here an alternate operational semantics which makes use of the deep copy operation above and argue that it is equivalent to the operational semantics presented in Section 3.2. We define our operational semantics as the least satisfying the rules in Figure A.4.

VARIABLE LOOKUP
$$\frac{x_1 = v \in E}{E \,\|\, [x_2 = x_1] \,\|\, e \overset{\cdot}{\longrightarrow}^1 E \,\|\, [x_2 = v] \,\|\, e}$$

APPLICATION
$$\frac{x'_1 \backslash E'_1 \lll x_1 \backslash E \qquad \begin{array}{c} x'_0 \backslash E'_0 \lll x_0 \backslash E \\ x'_0\ x'_1 \overset{\bullet}{\rightsquigarrow}_{E'_0 \,\|\, E'_1}\ e' \end{array} \qquad \boldsymbol{\alpha}(e') = e''}{E \,\|\, [x_2 = x_0\ x_1] \,\|\, e \overset{\cdot}{\longrightarrow}^1 E \,\|\, E'_1 \,\|\, e'' \,\|\, [x_2 = \text{RV}(e'')] \,\|\, e}$$

Figure A.4. Alternative Small Step Relation

We define $e_0 \overset{\cdot}{\longrightarrow}^* e_n$ when $\forall 1 \leq i \leq n.\, e_{i-1} \overset{\cdot}{\longrightarrow}^1 e_i$. We say $e \overset{\cdot}{\not\longrightarrow}^1$ when there exists no e' such that $e \overset{\cdot}{\not\longrightarrow}^1 e'$.

The Variable Lookup rule in this alternate relation is identical to that of the original relation. The Application rule differs only in that it creates a deep copy of the function and the argument before performing application. This alternate relation is "equi-stuck" with the original; each gets stuck only when the other does. We formalize this property as follows:

Lemma A.10. Iff $e \longrightarrow^* e'$ and $e' \not\longrightarrow^1$ where e' not of form E, then $e \overset{\cdot}{\longrightarrow}^* e''$ and $e'' \overset{\cdot}{\not\longrightarrow}^1$ where e'' not of form E.

Proof. By inspection of the rule sets, there are only two differences, both in the Application rule. First, $\overset{\cdot}{\longrightarrow}^1$ performs a deep copy while $\longrightarrow^1$ does not; a deep copy is an α-renaming and no operations condition on the specific name of a variable. Second, $\overset{\cdot}{\longrightarrow}^1$ uses $E'_0 \,\|\, E'_1$ in the matching relation whereas $\longrightarrow^1$ uses the entire environment E. But because $E'_0 \,\|\, E'_1$ is a deep copy of x_0 and x_1 and because neither the matching relation nor any relation it uses depends on the specific name of a variable, any variables which are not x'_0, x'_1, or transitively reachable from those variables are extraneous and do not affect the relations. □

We now show our preservation lemma: for any small step, there is a corresponding closure step which preserves simulation.

Lemma A.11 (Preservation)**.** Suppose that $e \overset{\cdot}{\longrightarrow}^1 e'''$ and that $e \preccurlyeq C$. Further, suppose that Φ and Υ are simulation-preserving. Then there exists some C''' such that $C \Longrightarrow C'$ and $e''' \preccurlyeq C'''$.

Proof. In a fashion similar to Lemma A.7, we show that, given a simulation, each premise of the operational semantics implies a corresponding premise of the constraint closure. The Transitivity rule demonstrates this immediately from simulation.

The only other case is when the Application rule applies. For clarity, we use variable names matching those chosen in the rules in Figures 3.13 and A.4. Then we know that we have an expression $E \,\|\, [x_2 \texttt{=} x_0\ x_1] \,\|\, e$; we also have each premise of small step application. Because $e \preccurlyeq_M C$, we know that $\alpha_0\ \alpha_1 <: \alpha_2 \in C$. By Lemmas A.7 and A.9, we have that $e' \preccurlyeq \alpha' \backslash C'$.

Recall that $e'' = \boldsymbol{\alpha}(e')$ and that $\alpha'' \backslash C'' = \Phi(\alpha' \backslash C', \alpha_2)$. By the simulation preservation of Φ, we have $e'' \preccurlyeq_{M'} \alpha'' \backslash C''$ for some M'. All variables in e'' and $\alpha'' \backslash C''$ are either fresh or already appear in M, so we can partition M' into two disjoint submaps: M'_1 (which contains only fresh mappings) and M'_2 (which contains all non-fresh mappings and is a submap of M). We let $M'' = M \cup M'_1$ and thus have that $e'' \,\|\, [x_2 \texttt{=} \mathrm{RV}(e'')] \preccurlyeq_{M''} C'' \cup \{\alpha_2 <: \alpha''\}$. We union the original C and the argument slice V_1 with this result, which corresponds to the concatenation of the original environment E and the deep copy variables E'_1 onto the above. This concatenation and respective union also preserve simulation, so we have $e''' \preccurlyeq_{M''} C'''$ and so we are finished. □

A.5 Stuck Simulation

We require one final lemma before we can prove our soundness theorem: that stuck programs are simulated by inconsistent constraint sets:

Lemma A.12. If $e \preccurlyeq C$ and e is stuck, then C is inconsistent.

Proof. By definition, e of form E is not stuck; therefore, $e = E \,\|\, [s] \,\|\, e'$. By case analysis, s must be of the form $x_2 \texttt{=} x_0\ x_1$; otherwise, the Variable Lookup rule always applies (because e is closed and s is of the form $x_2 \texttt{=} x_1$). By Lemma A.4, the deep copy premises always hold for closed E; furthermore, α-renaming is always possible. Therefore, because e is stuck, we must have that $x'_0\ x'_1 \overset{\bigcirc}{\rightsquigarrow}_{E'_0 \,\|\, E'_1}\ e''$. By Lemma A.9 we have that $\alpha'_0\ \alpha'_1\ {}^{\bigcirc}_{V_0}\!\rightsquigarrow_{V_1}\ \alpha' \backslash C'$. By Definition 3.3, C is inconsistent. □

A.6 Proof of Soundness

Using the above, we can now give a proof of Theorem 1.

Proof. Let $[\![e]\!]_{\mathrm{E}} = \alpha \backslash C$. Because e is closed, Lemma A.2 gives us that $e \preccurlyeq \alpha \backslash C$. We have that $e \longrightarrow^* e'$ where e' is stuck; then by Lemma A.10, we have that $e \stackrel{\cdot}{\longrightarrow}^* e'$ where $e' \stackrel{\cdot}{\not\longrightarrow}^1$ and e' not of the form E. By induction on the number of steps in the proof of $e \stackrel{\cdot}{\longrightarrow}^* e'$, Lemma A.11 gives us that that there exists a C' such that $C \Longrightarrow^* C'$ and $e' \preccurlyeq C'$. Lemma A.12 gives us that C' is inconsistent. Therefore C closes to an inconsistent constraint set and, by Definition 3.4, e does not typecheck. □

B. Proof of Decidability

We show here that TinyBang typechecking is a computable function. In particular, we provide a detailed sketch of Theorem 2. The primary challenge of this proof is showing that constraint closure is bounded by a finite set (Lemma 3.5 in Section 3.3.8). We prove this property by a high level counting argument: we define a finite set of constraint forms and a set of type variables; we then show that closure states C are always a set of substitutions of the (finite) type variables into the (finite) forms, of which there are only finitely many possibilities.

For this appendix we assume we are working over a fixed expression e, and we take C_{00} to be the set of *all* subtype constraints found in $[\![e]\!]_{\mathrm{E}}$. C_{00} includes all constraints under function types; it is not a sensible set in terms of the closure process and is only used to bound the size of other sets. Set C_{00} is our fixed set of constraint forms; there are only finitely many because finitely many different labels could be used in the original program.

B.1 Finite Slices

Intuitively, a slice creating during typechecking only needs to be deep enough to cover the pattern. The definition of slicing, however, places no limit on the size of a slice and also requires variables within a slice to be fresh. Here, we define a modified form of type inference to restrict the depth of slices by adjusting the application rule; we then show that the soundness proof remains valid with this modified rule.

First we characterize the set of all potential slice forms. Since slices are well-formed, minimal, and acyclic as per Definition 3.2, and we are only concerned about the structure of a slice and not the particular type variables used, we can view a slice $\alpha \backslash V$ as a tree with α at the root, with label constraint nodes having one child, and onion nodes have two children, and tree edges being subtyping. We hereafter write a slice form $\alpha \backslash V$ more simply as V since the head α will be the unique type variable in V without an upper bound. The type variables in the leaves of a slice, if any, are informally considered *free*; the type variables internal to the slice (including the root) are considered *bound*. This follows the intuition that slice forms are owned by the pattern that will be used to wire them in.

Viewing slices as trees, we can define the *depth* of slice V to be the length n of the longest chain $\overset{n}{\overrightarrow{(\tau <: \alpha)}}$ in V such that for each τ_i for $i > 1$, α_{i+1} occurs in τ_i. We first assert that given an initial expression e we can place a fixed bound n_{maxslice} on the depth of any slice we need to consider in closure without loss of generality. By inspection of the type compatibility relation in Figure 3.10, if a slice is not deep enough the Partial rule will fire and partial compatibility will never lead to a closure step. By inspection of the Empty Onion rule (the leaf case of the pattern V'), any deeper component of the slice V is ignored. So intuitively, if the slice reaches the bottom of the pattern, it will be sufficient.

Thus, taking the set of all pattern types in C_{00} to be $\overset{\sqcup}{V'}$, we would like to define n_{maxslice} to be $n_{\mathit{max}\text{-}\phi} = \mathit{max}(\{\mathit{depth}(V') \mid V' \in \overset{\sqcup}{V'}\})$, the depth of the deepest pattern. This is close to serving as our global slice depth bound, but it is not correct: slices do not line up with patterns "one-to-one" because the appearance of a conjunction in a pattern and the appearance of an onion within a slice do not necessarily coincide. In particular, the Onion Value Left and Onion Value Right compatibility rules in Figure 3.10 descend deeper into the slice without descending deeper into the pattern. So, we must slice deeply enough so that all underlying label/atom/function structure is reached through the various data onioning constraints $\alpha_0 \,\&\, \alpha_1 <: \alpha_2$. In the worst case, we may need to go through every onion value constraint in C_{00} to find the next deepest structure in the pattern; we may also have to visit the same onion structure numerous times (in the case of non-contractive onions like $\alpha_1 \,\&\, \alpha_2 <: \alpha_2$ where each visit allows another lower bound to be added). But each onion can only yield at most every lower bound in the constraint set; after this, additional lower bounds are redundant. In order words, while non-contractive onions can in principle be "infinitely wide", there are finitely many non-onion types to onion together and, at some point, the data must repeat; since only the left-most copy matters, the repetitions need not be considered. So, given that there are $n_{\&}$ onion constraints in C_{00} and n_{τ} lower bounds in C_{00}, we then need to slice to depth at most $n_{\mathit{maxslice}} = n_{\mathit{max}\text{-}\phi} * n_{\&} * n_{\tau}$: between each level of pattern we have $n_{\&} * n_{\tau}$ levels of onioning in the extreme worst case.

So, we have a bound on the depth of any slice. We also have a fixed bound on the number of distinct nodes that can occur in any slice tree, each node must an α-variant of a constraint in C_{00}. (Note there can be multiple occurrences of the same constraint form in a given slice, but the restriction to finitely many forms will still keep the number of possible slices finite.) Define VV_{allslice} to be the set of all such slice tree forms that can be constructed of depth at most n_{maxslice}, modulo α-conversion.

Lemma B.1. Set VV_{allslice} is of finite cardinality.

Proof. Each slice $V \in VV_{\mathit{allslice}}$ is an at most n_{maxslice} depth tree of nodes drawn from the finite set C_{00} of node types; there are only finitely many finite trees that can be constructed over a finite set of nodes. □

Now that we have isolated all possible slices that we need to consider, we can address the issue of the fresh variables created by the current slicing definition. Before closure begins, we create a personal library of slices for each simple function; each simple function has a library of slices not sharing any bound variables with any other function's slice library. More precisely, we define function *sliceLib*(V') which takes as argument the pattern V' on any simple function occurring in C_{00}, and returns an α-renaming of slice set VV_{allslice} such that no two libraries share any type variables. Since there are finite patterns and each library contains finitely many slices needing finitely many type variables each, the total number of type variables needed to "stock" the slice libraries is finite. Then during closure, we pull a slice from the library when a function is matched.

B.2 Finite type variables

Given that all constraints in any closure state C must be drawn from the forms in C_{00}, the number of states will be finite if we can restrict ourselves to drawing all type variables from some fixed finite set. A particular challenge here is the slice V_1 in the Application closure rule. Slices of the function are discarded, but slices of the argument value are used to "wire" the argument into the pattern and so they may appear in the global constraint set C. (Our implementation uses deep, tree-shaped types rather than type variables in a slice, but we use slices here for a more elegant theory.) While this makes the presentation of soundness simpler, it complicates the decidability proof since we need to limit the variables used in these slices so as not to have the constraint set grow arbitrarily large.

Theorem 2, our decidability theorem, assumes that our polyinstantiation function Φ_{CR} is finitely freshening. This property is formally defined in Definition C.8 of Appendix C. Finite freshening requires the definition of a fixed finite set of type variables given the original program expression (this is the output of function f in the definition); with such a set defined, the definition guarantees there is a fixed finite set of distinct type variables $\overset{\sqcup}{\alpha}$ that is a superset of the type variables used in any constraint set C.

So, all that is needed is to define the fixed initial set of type variables. We define this set to include the initial type variables $\alpha \in C_{00}$, as well as all variables occurring in *sliceLib*, which is also finite as justified above. No other type variables are needed; additional type variables may be used by the polymorphism model but Appendix C addresses how those are also finite. With this restriction we have that the type variables appearing in any constraint set C being fed into Φ_{CR} is a subset of $\overset{\sqcup}{\alpha}$; thus, polyinstantiation will also produce variables all in $\overset{\sqcup}{\alpha}$. Thus, $\overset{\sqcup}{\alpha}$ constitutes all type variables that may ever appear in any constraint set and that set is finite, proving finiteness of all paths.

B.3 A restricted Application rule

Two changes are needed to the Application closure rule: we need to place the aforementioned restriction on accepting slices of at most a fixed depth n_{maxslice}, and we cannot simply place the function argument slice V_1 in the constraint set since it may have arbitrary fresh variables in it. Instead, we will use one of our fixed slice templates.

Definition B.2 (Modified Application Closure)**.** Revise the Application closure rule of Figure 3.13 to insert the following additional preconditions:

- *depth*$(V_0) \leq n_{\mathit{maxslice}}$
- *depth*$(V_1) \leq n_{\mathit{maxslice}}$.
- $V_2 \in$ *sliceLib*(V')
- *unify*$(V_1, V_2) = V_3$

where

- *unify*(V_1, V_2) requires V_1 and V_2 to have the same tree structure and only be α-variants of one another and operates by replacing leaf type variables of V_2 with the corresponding leaf type variables in V_1
- V' is the pattern from the successfully matched function (which needs to be added as a formal argument to the matching relation so it can propagate to the Application rule)

In the Application rule conclusion we replace slice V_1 with the (equivalent) V_3.

Hereafter we will assume restricted closure is used. Since a rule has changed we now revisit the soundness proof of Section A to sketch how it may be adapted to restricted closure.

First, the deep copy relation of Figure A.2 is modified to remove the restrictions on v in the Atomic rule: the deep copy may stop at any point (not just atoms) meaning that it could be no copy at all, a full deep copy, or any degree in between.

The bisimulation relation of Figure A.3 remains unchanged; for a slice and a (partial) deep copy they both must stop at the same depth for bisimulation to hold. This is the key to soundness: the exact aligning of the operational semantics deep copy degree with the slice depth.

The alternative small-step relation in Figure A.4 now is taken to use the modified (partial) deep copy relation; since there is no restriction on how deep to copy, reduction becomes nondeterministic. Fortunately, there is no real difference in how much copying is done (or not done): an analogy of Lemma A.10 holds for the revised small-step relation for *all* possible nondeterministic paths chosen.

Lastly, Preservation (Lemma A.11) can be shown to hold by a variant of our previous proof: given a fixed step $e \overset{\cdot}{\longrightarrow}^1 e'$, we perform the analogous $C \Longrightarrow^1 C'$ in the type constraint system. However, we cannot let the size of the deep copy dictate the depth of the slice in the case of Application; instead we let the slice dictate the depth of the deep copy. In particular choose the slice which picks lower bounds which line up with the application step taken by e and such that it is deep enough to be full-compatible. We know such a slice V exists: by the above reasoning our bound n_{maxslice} will suffice to reach the bottom of any pattern. Now, given slice V we can pick a corresponding e'' such that $e \overset{\cdot}{\longrightarrow}^1 e''$ and $e'' \approx V$. Thus, we may establish that $e'' \preccurlyeq C'$ in analogous fashion.

This is a slightly weaker version of preservation since it changes e' to e'' in the result. However, as mentioned above, all nondeterministic paths are aligned and are equi-stuck, and so the particular choice of nondeterministic path is irrelevant. Soundness then follows as before modulo this change of path.

B.4 Finiteness of Closure

Given the above reformulation of application, we can now show that closure is finite as stated in Lemma 3.8. We first show that construction of the initial derivation, slicing, compatibility, and matching are all easily decidable.

Lemma B.3. Given well-formed inputs the following functions/relations/properties are computable/decidable:

1. $[\![e]\!]_{\mathrm{E}}$
2. $\alpha\backslash V \ll \alpha'\backslash C$
3. Given potential slice V and pattern type V', $\exists F, \circledcirc .\ \alpha_0 \overset{\circledcirc}{\underset{V}{}}\!\preceq^{F}_{V'}\ \alpha'_0$
4. Given potential slices V_0, V_1, $\exists \alpha_2, C, \circledcirc .\ \alpha_0\ \alpha_1 \overset{\circledcirc}{\underset{V_0}{}}\!\leadsto_{V_1} \alpha_2\backslash C$

Proof. 1. is trivially computable since the definition of Figure 3.7 is a simple inductive definition.

2.: Since V is required to be an acyclic constraint set and each subgoal works over a subtype of one of the variables in V, the potential proof tree structure corresponds to the structure of V, which is finite; thus the proof search is finite.

3. Inspecting the compatibility rules of Figure 3.10, each rule subgoal *either* works on a type variable in a subtype of a constraint in the pattern, or a type variable in a subtype of a constraint in the slice. For example, Onion Value Left works on a variable in a subtype of the slice, and Conjunction pattern works on a variable in a subtype of the pattern. Thus, the proof structure is fixed based on the V/V' input and so proof search is finite. Both F and $\circledcirc$ are deterministically synthesized up the proof tree, so are easy to compute.

4. follows similarly to 3.; observe here that $\alpha_2\backslash C$ is deterministically synthesized up the proof tree. □

We now prove part of our finiteness lemma: that there are finitely many closure steps which may be taken from a given set.

Lemma B.4. Given $[\![e]\!]_{\mathrm{E}} \Longrightarrow^* C$, it is possible to compute the set $\{C' \mid C \Longrightarrow^1 C'\}$ (which is thus a finite set of finite C').

Proof. Since C is inductively finite, there are finitely many different matches of the constraints to the Transitivity and Application closure rule preconditions. Transitivity produces one possible output per matching input so only finitely many distinct transitivity steps are possible from C. Application is additionally parameterized by slices V_0 and V_1; given such slices, matching will produce a single output $\alpha'\backslash C'$ since these slices bisimulate expressions which are deterministic. Freshening function $\Phi_{\mathrm{CR}}/\Upsilon_{\mathrm{CR}}$ is a function so it will also produce a single output. So, the only nondeterminism in Application is in the slices. By the above construction we argued it suffices to only consider slices in the (finite) set VV_{allslice}; each such pair of potential slices can decidably be verified to be slices and checked for matching by the previous Lemma, leading to computation of a finite set of match outputs for each application rule and thus finitely many results overall. ☐

Taking the above, we can now show our finiteness lemma (restated here for convenience).

Lemma 3.5. For any e, there exists a finite set of constraints C'' such that, if $[\![e]\!]_{\mathrm{E}} = \alpha\backslash C$ and $C \Longrightarrow^* C'$, then $C' \subseteq C''$.

Proof. All constraints introduced by constraint closure have a form appearing in the finite set C_{00} and are restricted to variables from the finite set $\overleftrightarrow{\alpha}$. Thus, the set of all permutations of these variables in each of these constraint forms (which is finite) is the upper bound of any constraint set reachable by constraint closure. The remainder of this lemma is immediate from Lemma B.4 ☐

The above lemmas allow us to show two of the lemmas appearing in Section 3.3.8 with minimal effort.

Lemma 3.6 (Convergence)**.** For any finite C, there exists some C' such that $C \Longrightarrow^! C'$.

Proof. By inspection of Figure 3.13, constraint closure is monotonic; if $C \Longrightarrow^1 C'$ then $C \subseteq C'$. Because subset inclusion is transitive, this holds for $\Longrightarrow^*$ as well. Ignoring superfluous closure steps ($C \Longrightarrow^1 C$), we can thus bound every constraint closure sequence by the number of constraints constructable from C_{00} and the variables from the finite set $\overleftrightarrow{\alpha}$, which is shown to be finite in Lemma 3.5. ☐

Lemma 3.8. $C \Longrightarrow C'$ is decidable for finite C.

Proof. Immediate from Lemmas B.3 and B.4. ☐

B.5 Confluence

While not strictly necessary to show the *decidability* of typechecking, constraint closure confluence is a necessary property for the TinyBang type system to be usable in any sense. The above lemmas show that constraint closure is finite, but we must show that the results are always the same (up to the equivalence of Υ). We use the usual diamond property-driven approach to proving confluence of constraint closure, beginning with a proof of the single reduction diamond. We begin by showing a supplementary lemma:

Lemma B.5. If $C \Longrightarrow^1 C\cup C'$ then $C\cup C'' \Longrightarrow^1 C\cup C'\cup C''$ for any C''; that is, the number of ways in which to satisfy the constraint closure rules increases monotonically with the size of the constraint set.

Proof. If the closure step involved is transitivity, this property is trivial. For application, we show that the ways in which a premise can be satisfied increases monotonically with the size of the constraint set (either directly or indirectly). For clarity, we describe a relation as monotonic in some set if the number of ways to satisfy it increases monotonically with the size of that set; we describe a relation as monotonic in another relation if the number of ways to satisfy the relation increases monotonically with the number ways in which that other relation holds.

The first premise of application is a simple set containment check, which is trivially monotonic in C.

The second and third premises are slicing. The number of slices (e.g. $\alpha_0'\backslash V$) which allow this relation to hold increases with the number of constraints in C; thus, slicing is monotonic in C.

The fourth premise (matching) is monotonic in full compatibility, incompatibility, and in the number of constraints in V_0. (Note that partial compatibility is irrelevant here.) Since V_0 is a slice and slicing is monotonic in C, it suffices to show for this case that both full compatibility and incompatibility are monotonic in C. By inspection of Figure 3.10, we observe that every rule for full compatibility and for incompatibility tests for the presence (not absence) of a constraint in V.

The fifth premise (polyinstantiation) is the result of invoking a total function on the result of matching; this premise always holds.

Because each premise of application is monotonic in C, closure is monotonic in C; additional constraints do not prevent a constraint closure rule from being applicable (although they may cause it to be redundant, which is why Lemma 3.6 is not a contradiction). ☐

This allows us to prove the single reduction diamond property on constraint closure, which we state as follows:

Lemma B.6. If $C \Longrightarrow^1 C\cup C'$ and $C \Longrightarrow^1 C\cup C''$ then $C\cup C' \Longrightarrow^1 C\cup C'\cup C''$ and $C\cup C'' \Longrightarrow^1 C\cup C'\cup C''$.

Proof. We observe that there is some conjunction of premises P which holds true of C such that $C \Longrightarrow^1 C\cup C'$; we also observe that there is some conjunction of premises Q which holds true of C such that $C \Longrightarrow^1 C''$. By Lemma B.5, P is still true of $C\cup C''$ and Q is still true of $C\cup C'$, so both of these sets will close to $C\cup C'\cup C''$. ☐

We can then prove our general confluence property.

Lemma 3.7 (Confluence)**.** If $C \Longrightarrow^* C'$ and $C \Longrightarrow^* C''$ then there exists some C''' such that $C' \Longrightarrow^* C'''$ and $C'' \Longrightarrow^* C'''$.

Proof. By Lemma B.6, Lemma 3.6, and Newman's Lemma [16]. ☐

B.6 Computable Typechecking

We are nearly prepared to prove that the typechecking of an expression is computable. We must first show an equivalence between our original formulation of typechecking (in Definition 3.4) and our alternative formulation (in Lemma 3.9). This in turn depends upon a property of inconsistency:

Lemma B.7. If C is inconsistent, then $C\cup C'$ is inconsistent for any C'.

Proof. By the same argument as Lemma B.5. □

We then show our alternative typechecking formulation to be correct:

Lemma 3.9. An expression e typechecks if and only if $[\![e]\!]_{\text{E}} = \alpha\backslash C$, $C \Longrightarrow^! C'$, and C' is consistent.

Proof. Let $[\![e]\!]_{\text{E}} = \alpha\backslash C$. This is to say that $C \Longrightarrow^! C'$ with inconsistent C' if and only if there exists some inconsistent C'' such that $C \Longrightarrow^* C''$. If the former holds, then the latter clearly holds – we select $C'' = C'$. If the latter holds, then the former holds by Lemma B.7. □

Finally, we show that typechecking is computable.

Theorem 2 (Decidability)**.** Typechecking is decidable.

Proof. Initial alignment is a computable function by induction on the size of e. Complete closure ($\Longrightarrow^!$) is a computable function because it is both convergent (Lemma 3.6) and confluent (Lemma 3.7) and because closure itself is decidable (Lemma 3.8). Inconsistency is computable by the same argument. Therefore, whether a program e typechecks is a computable function. □

C. Polymorphism in TinyBang

This appendix details properties of polymorphism in TinyBang. We give a definition of the polymorphism model summarized in Section 3.4. We then prove both that it preserves the simulation relation defined in Appendix A and that it produces finitely many type variables as required by Appendix B.

C.1 Definition of $\Phi_{\text{CR}}/\Upsilon_{\text{CR}}$

The $\Phi_{\text{CR}}/\Upsilon_{\text{CR}}$ polymorphism model functions by creating fresh polyinstantiations of every variable at every call site and then merging these variables when recursion is detected. More precisely, a type variable merging occurs if (1) two type variables represent the type of the same variable from the original program and (2) they do so in the context of the same call site in the original program. In order to track this information, we define a bijection $\Leftrightarrow$ between type variables and type variable *descriptors* for a given typechecking pass.

A type variable descriptor represents two pieces of information. The first is the type variable $\mathring{\alpha}$ that was decided for the program variable during initial alignment. The second is a *contour* representing the set of call strings at which this type variable applies. (We give a precise definition of contours below.) For instance, suppose $\langle\alpha_{\mathtt{x}}, \alpha_{\mathtt{y}}\rangle \Leftrightarrow \alpha$ where $\alpha_{\mathtt{y}}$ is a contour representing just the call string "$\alpha_{\mathtt{y}}$"; then α is the polyinstantiation of $\alpha_{\mathtt{x}}$ when it is reached by a call site $\alpha_{\mathtt{y}}$ appearing at the top level of the program. For example, in the program

```
i = { p = () } -> { x = p };
v = 4;
y = i i;
z = i 4;
```

the function `i` is the identity function. The type variable $\langle\alpha_{\mathtt{x}}, \alpha_{\mathtt{y}}\rangle \Leftrightarrow \alpha$ represents the type of `x` when it is expanded into the call site at `y`; it does *not* represent the type of `x` when it is expanded into the call site at `z`.

More formally, we define contours as follows:

Definition C.1. A contour $\mathcal{C}$ is a regular expression with the following restrictions:

- The contour $\mathcal{C}$ is composed of a disjunction of *contour strands* $\mathcal{S}$,
- Each strand is a concatenation only of literals and Kleene closures over disjunctions of literals, and
- Each literal is the initial type variable of a call site declaration variable (e.g. the initial type of x_2 in the clause $x_2 = x_0\ x_1$)

We use ϵ to denote the contour matching only the empty call string. We use $\varnothing$ to denote the contour matching no call strings.

As stated above, contours could be arbitrarily long. For decidability, we provide the following refinement on contours which we maintain inductively through each closure step:

Definition C.2. A contour $\mathcal{C}$ is *well-formed* if each call site declaration variable appears at most once within it.

For example, the contour $\alpha_0(\alpha_1\alpha_2)^*$ is well-formed but the contour $\alpha_0\alpha_1\alpha_0$ is not. The bijection only admits descriptors with well-formed contours.

For the purposes of the definitions below, we define simple notation on contours. We write $\mathcal{C} \,\|\, \mathcal{C}'$ to indicate the concatenation on contours in the sense of regular expressions; note that while contours are closed under this operation (by cross product over the outermost union), well-formed contours are not. We also write $\mathcal{C} \leq \mathcal{C}'$ to denote that the set of call strings matched by $\mathcal{C}$ is a subset of the call strings matched by $\mathcal{C}'$.

For notational convenience, the remainder of this appendix often uses a type variable's descriptor in place of the type variable. For instance, we use `int` $<:\langle\alpha, \varnothing\rangle$ as shorthand for "`int` $<: \alpha'$ such that $\langle\alpha, \varnothing\rangle \Leftrightarrow \alpha'$".

C.1.1 Polyinstantiation

We begin our management of polymorphism with initial alignment. In particular, we always choose the type variable bijection such that each fresh type variable $\mathring{\alpha}$ chosen by initial alignment has a fixed mapping. If the variable x represented by $\mathring{\alpha}$ is defined within a pattern or the body of a function, then $\langle\mathring{\alpha}, \varnothing\rangle \Leftrightarrow \mathring{\alpha}$. Otherwise, $\langle\mathring{\alpha}, \epsilon\rangle \Leftrightarrow \mathring{\alpha}$ (to reflect the fact that α represents x at top level, in the empty call string).

Other than this initial set, new type variables enter closure in one of two ways: by being introduced by a slice or by polyinstantiation. We view variables introduced by slicing in the same way that we view the variables introduced by initial alignment; they are "original" variables in a sense and not part of polymorphism. That is, if a slice introduces a fresh variable α_1' for a variable α_1 and $\langle \alpha_2, \mathcal{C}\rangle \Leftrightarrow \alpha_1$, then $\langle \alpha_2', \mathcal{C}\rangle \Leftrightarrow \alpha_2$ for some α_2' describing the original variable of that slice. This is not a problem for our system because, as illustrated in Appendix B, only finitely many original slice variables will be used during a given typecheck.

The other manner in which type variables can be introduced is through the Φ_{CR} function. We define Φ_{CR} to instantiate variables appearing in a constraint set as follows:

Definition C.3. For some $\langle \alpha', \mathcal{C}'\rangle$, let $\mathcal{C}$ be the least well-formed contour such that, if some call string s is accepted by $\mathcal{C}'$, then $s \parallel \alpha'$ is accepted by $\mathcal{C}$. Then $\Phi_{\text{CR}}(\alpha\backslash C, \langle \alpha', \mathcal{C}'\rangle)$ is the substitution of type variables in $\alpha\backslash C$ such that (1) each $\langle \alpha'', \varnothing\rangle$ which is the upper bound of some $c \in C$ is replaced with $\langle \alpha'', \mathcal{C}\rangle$ everywhere it appears in C and (2) all other type variables remain unchanged.

Calling Φ_{CR} creates a fresh version of each uninstantiated type variable free in the constraint set. Note that type variables which already have a contour – which have been previously instantiated – are unaffected.

C.1.2 Unification

The second function, Υ_{CR}, is used to unify variables already appearing in the constraint set with the newly polyinstantiated ones from Φ_{CR}. In particular, Υ_{CR} is used to unify type variables whose responsibilities overlap. This is necessary to ensure that type variables are unified correctly when recursion is detected. For instance, imagine that $\mathcal{C}$ in the definition above was $\alpha_1(\alpha_2)^*$. Then a type variable $\langle \alpha_3, \mathcal{C}\rangle$ represents the polyinstantiation of α_3 for the call stack is composed of call site α_1 followed by any number of recursive calls to call site α_2. But when α_2 was first reached, it may not be known to be recursive; there may be variables such as $\langle \alpha_3, \alpha_1\alpha_2\rangle$ already in the constraint set. These variables must then be unified.

Before defining our unification function, we rely on some simple supporting definitions:

Definition C.4. Two variables $\langle \alpha, \mathcal{C}\rangle$ and $\langle \alpha', \mathcal{C}'\rangle$ *overlap* iff $\alpha = \alpha'$ and there exists a call string matched by both $\mathcal{C}$ and $\mathcal{C}'$. For some constraint set C, two variables α and α' are *connected in* C iff either α and α' overlap or there exists some α'' appearing in C such that α overlaps with α'' and α'' and α' are connected in C.

The purpose of this definition is to cluster type variables which have overlapping contours for the same program variable. These are the variables which should be unified under a contour which covers all of the contours in that cluster. (Observe that every α with a non-empty contour overlaps itself and connects with itself.) Using the above, we can give the following definition of Υ_{CR}:

Definition C.5. For a set of constraints C, $\Upsilon_{\text{CR}}(C)$ is the least equivalence relation conforming to the following requirements:

- Reflexivity, and
- If any two variables $\langle \alpha, \mathcal{C}\rangle$ and $\langle \alpha, \mathcal{C}'\rangle$ are connected in C then all variables $\langle \alpha', \mathcal{C}\rangle$ and $\langle \alpha', \mathcal{C}'\rangle$ are equivalent.

C.2 Soundness of $\Phi_{\text{CR}}/\Upsilon_{\text{CR}}$

To show that the TinyBang type system is sound when using $\Phi_{\text{CR}}/\Upsilon_{\text{CR}}$ as a polymorphism model, we must show that it is well-behaved. This entails showing three properties: the simulation preservation of Φ_{CR}, the finite freshening of Φ_{CR}, and the monotonicity of Υ_{CR}. We show each of those properties below.

C.2.1 Simulation Preservation

An informal discussion of simulation preservation is given in Section 3.3.6. We formalize it here as follows:

Definition C.6 (Simulation Preservation)**.** A polyinstantiation function Φ is simulation-preserving iff $e \preccurlyeq \alpha\backslash C$ implies that $\boldsymbol{\alpha}(e) \preccurlyeq \Phi(\alpha\backslash C, \alpha)$ for any α in C.

We then assert that our polymorphism model has this property:

Lemma C.7. Φ_{CR} is simulation-preserving.

Proof. Let $C' = \Phi_{\text{CR}}(C, \alpha)$. We have that $e \preccurlyeq_M C$ and that C' is a capture avoiding substitution of variables in C. We consider each type variable α appearing in C being replaced by some other α'. If α only appears free or only appears captured in C, then this is an α-substitution of α; thus, if e.g. $\tau <: \alpha \in C$ then $\tau <: \alpha' \in C'$ such that α was replaced by α'. If we subject M to the same α-substitution as C, we can show by induction on the size of $e \preccurlyeq_M C$ that $e \preccurlyeq_{M'} C'$.

If α appears both free and captured in C, then suppose that each variable x in e for which $M(x) = \alpha$ appears either only captured or only free in e. In this case, the corresponding instances of α can be substituted (or not) and an alternate mapping M' can be defined as $M'(x) = \alpha'$ (resp. $M'(x) = \alpha$) such that simulation still holds.

The only remaining case is that some x appears both free and captured in e such that $M(x) = \alpha$. But if this is the case, then any expression in which e is a subexpression is either not closed or contains a duplicate definition of x; in either case, this violates the assumptions made either by the typechecking algorithm or by the well-formedness of ANF expressions. Therefore, this case does not exist and we are finished. □

C.2.2 Finite Freshening

The decidability proof requires that the polyinstantiation function used with the TinyBang type system is *finitely freshening*. This property is outlined in Section 3.3. Informally, the property we want is that the polymorphism model will only incur finitely many polyinstantiations (even when presented with variables it has polyinstantiated). Formally, we state this property as follows:

Definition C.8 (Finitely Freshening Φ)**.** A polyinstantiation function Φ is *finitely freshening* iff $\forall f : \overbracket{e} \to \overbracket{\overbracket{\alpha}}. \exists g : \overbracket{e} \to \overbracket{I}. \forall e.$ we have all of the following:

- $\overbracket{\alpha}$ is a finite set of type variables isomorphic to $f(e) \times g(e)$,
- $f(e) \subseteq \overbracket{\alpha}$,
- For any α appearing in C, if the set of variables appearing in C is a subset of $\overbracket{\alpha}$ then the set of variables appearing in $\Phi(C, \alpha)$ is a subset of $\overbracket{\alpha}$, and

By this definition, we argue that Φ_{CR} is finitely freshening using contours as indices and the type variable bijection as the basis for $\overbracket{\alpha}$. We begin by defining a function to determine all of the contours which will be used in a typechecking pass; this function will be used as the basis for the g function required by the above definition.

Definition C.9. Let $\textsc{AllContours}(-)$ be a function taking a set of type variables $\overbracket{\alpha}$ and producing the set $\overbracket{\mathcal{C}}$ of all well-formed contours which can be constructed using only those type variables as literals.

We then state the following simple lemma regarding this function:

Lemma C.10. For a finite set of type variables, the $\textsc{AllContours}(-)$ function is computable and produces a finite set of contours.

Proof. Trivial by the grammar of regular expressions and because well-formed contours have a length bounded on the order of the number of type variables in the set. $\square$

Using this property, we can easily show our polymorphism model to be finitely freshening:

Lemma C.11. The polymorphism model Φ_{CR} is finitely freshening.

Proof. We are given a function f as described in Definition C.8. We choose the corresponding $g(e) = \textsc{AllContours}(f(e))$. For a given e, we select the bijection $\Leftrightarrow$ as described above – $\alpha \in f(e) \implies \alpha \Leftrightarrow \langle \alpha, \mathcal{C} \rangle$ where $\mathcal{C}$ is either $\varnothing$ or ϵ – and leave the remainder of the bijection unconstrained. Since neither $\varnothing$ and ϵ require literals, they are in $\textsc{AllContours}(f(e))$. We therefore define $\overbracket{\alpha} = \{\langle \alpha', \mathcal{C} \rangle \mid \alpha' \in f(e) \land \mathcal{C} \in g(e)\}$.

It remains to show that, given C with variables in $\overbracket{\alpha}$, the resulting C' has only variables in $\overbracket{\alpha}$. We observe that Φ_{CR} performs a substitution on some variables in the set. This substitution is such that a variable $\langle \alpha', \mathcal{C} \rangle$ is always substituted with another variable $\langle \alpha', \mathcal{C}' \rangle$ such that $\mathcal{C}'$ is a well-formed contour. Because $\langle \alpha', \mathcal{C}' \rangle$ was in C, we know that $\alpha' \in f(e)$; because $\mathcal{C}'$ is a well-formed contour, we know that $\mathcal{C}' \in g(e)$. Each variable is therefore in $\overbracket{\alpha}$ and so this property holds. $\square$

We must also show that Φ_{CR} itself is computable.

Lemma C.12. Φ_{CR} is computable.

Proof. Whether two variables α and α' overlap in a finite constraint set C is decidable by enumeration of all possible paths between those variables. For a given e, we have a finite set of well-formed contours and the inclusion problem on regular expressions is decidable, so finding a least well-formed contour which subsumes a set of other contours is decidable. Finally, Φ_{CR} is an α-substitution on a finite set and so is trivially decidable. $\square$

C.2.3 Monotonicity

Our next step is to show that Υ_{CR} is monotonic in its equivalences: that each additional constraint only increases the number of equivalences which hold over the set. This is stated as follows (recalling that we view equivalences as sets of equivalent pairs):

Lemma C.13. If $C \subseteq C'$ then $\Upsilon_{\text{CR}}(C) \subseteq \Upsilon_{\text{CR}}(C')$.

Proof. By Definition C.4, if two variables are connected in C then they are also connected in C'. Definition C.5 defines equivalence directly in terms of (1) the variables appearing in C (all of which appear in C') and (2) the connections between variables in C (all of which appear in C'), so we are finished. $\square$

We must then show Υ_{CR} to be computable.

Lemma C.14. Υ_{CR} is computable.

Proof. The regular expression intersection problem is decidable, so the connectedness of two variables α and α' in a finite C is decidable. Determining disjoint sets of connected variables is decidable by enumeration of all possible pairs. By the argument in Lemma C.12, finding a least well-formed contour is decidable. The resulting equivalence performs tests based either on (1) identity or (2) regular expression inclusion, which is also decidable. $\square$

C.2.4 Well-Behaved Polymorphism

Finally, we can formally state the well-behaved nature of our polymorphism model:

Theorem 3. The polymorphism model $\Phi_{\text{CR}}/\Upsilon_{\text{CR}}$ is well-behaved.

Proof. Immediate from Lemmas C.7, C.11, C.12, C.13, and C.14. $\square$

D. Built-Ins

In this appendix, we extend TinyBang with a generic framework for introducing built-in operations. We also show how to use the framework to introduce integer operations and reference cells.

s	$::=$	$x = \boxdot\, \overrightarrow{x} \mid \ldots$	*clauses*
$\boxdot$	$::=$	$+ \mid \ldots$	*built-ins*

BUILTIN
$$\frac{\boxdot(E, \overset{n}{\overrightarrow{x'}}) = \langle E', v\rangle}{\langle E, [x_0 = \boxdot\ \overset{n}{\overrightarrow{x'}}\,] \parallel e\rangle \longrightarrow^1 \langle E', [x_0 = v] \parallel e\rangle}$$

Figure D.1. Built-In Framework Operational Semantics Additions

c	$::=$	$\boxdot\overrightarrow{\alpha} <: \alpha \mid \ldots$	*constraints*

$$[\![x_0 = \boxdot\ \overset{n}{\overrightarrow{x}}]\!]_s = \overset{\star}{\alpha}_0 \backslash \boxdot\ \overset{n}{\overrightarrow{\overset{\star}{\alpha}}} <: \overset{\star}{\alpha}_0$$

BUILTIN
$$\frac{\boxdot\overset{n}{\overrightarrow{\alpha}} <: \alpha' \qquad \boxdot(C, \overset{n}{\overrightarrow{\alpha}}) = \langle C', \overleftrightarrow{\tau}\rangle}{C \Longrightarrow^1 C' \cup \{\tau' <: \alpha' \mid \tau' \in \overleftrightarrow{\tau}\}}$$

Figure D.2. Built-In Framework Type System Additions

D.1 Environment

To ensure that the framework is sufficiently general to properly handle state, which makes it possible to create cyclic data, we need an environment that supports cyclic scope. To this end, we amend the scoping rules such that an expression e is closed iff $\exists E.\ e = E \parallel e_0$ and every variable occurring in e is bound either by some clause before it or by some clause in E. We represent such closed e as $\langle E, e_0\rangle$.

D.2 Framework

The built-in framework makes additions to the language grammar and operational semantics; these additions appear in Figure D.1. Each individual built-in operator is represented by a symbol in the grammar $\boxdot$. We define for each builtin a metatheoretic function $\boxdot(-,-)$ that defines the operator's behavior; this function is from an environment and a list of operand variables onto a pair between a resulting environment and value.

The contents of Figure D.2 parallel these modifications at the type system level. We also overload our operator function: we define a metatheoretic function $\boxdot(-,-)$ from a constraint set and list of operand type variables onto a pair between a resulting constraint set and a *set* of resulting types. This set represents all possible resulting types; any runtime result must be in the union of these types.

To create a built-in operator, it is then sufficient to define the value and type level metatheoretic functions. Note that these functions may be partial; not every built-in can accept every variable. If the value level built-in function is partial, then evaluation becomes stuck. Correspondingly, we modify the definition of inconsistency in the following way:

v	$::=$	$\mathbb{Z} \mid \ldots$	*values*
$\mathring{v}$	$::=$	$\texttt{int} \mid \ldots$	*pattern vals*
τ	$::=$	$\texttt{int} \mid \ldots$	*types*

Figure D.3. Integer Additions

π	$::=$	$\texttt{int} \mid l \mid \texttt{ref}$	*projectors*

Figure D.4. Projectors

$$E_\pi(x) = \begin{cases} E(x) & \text{when } E(x) \text{ is non-onion}, E(x) \in \mathbb{Z}, \pi = \texttt{int} \\ E(x) & \text{when } E(x) \text{ is non-onion}, E(x) = l\ x', \pi = l \\ E(x) & \text{when } E(x) \text{ is non-onion}, E(x) = \texttt{ref}\ x', \pi = \texttt{ref} \\ E_\pi(x_1) & \text{when } E(x) = x_1 \,\&\, x_2, E_\pi(x_1) \text{ is defined} \\ E_\pi(x_2) & \text{when } E(x) = x_1 \,\&\, x_2, E_\pi(x_1) \text{ is undefined} \end{cases}$$

Figure D.5. Value Projection

Definition D.1 (Inconsistency)**.** A constraint set C is *inconsistent* either as indicated in Definition 3.3 or iff there exists some $\boxdot\overset{n}{\overrightarrow{\alpha}} <: \alpha' \in C$ such that $\boxdot(C, \overset{n}{\overrightarrow{\alpha}})$ is undefined.

D.3 Integers

For simplicity, we left integers out of the main body of the paper; we introduce integer values, patterns, and types (but not operations) here to help illustrate built-ins. The changes to the operational semantics and the type system are shown in Figure D.3; note that they are entirely syntactic.

D.4 Projection

Built-in operations commonly act as destructors, extracting values from onions to act upon them. To formalize this process, we define a notion of *projection* to extract the highest priority value of a given form from an onion. In the evaluation system, this is written as a function $E_\pi(x) = v$ where π is a projector from the grammar in Figure D.4. The function itself is defined in Figure D.5. Note that this function is undefined when the projection is not possible (e.g. when `int` is projected from a variable containing the empty onion).

We also define projection in the type system. Because the type system may have multiple lower bounds on a given type variable due to information loss, we must define a type projection *relation*. Further, we must explicitly address the cases in which projection may fail. For type projection, we write $C \vdash \alpha \overset{\pi}{\rightarrowtriangle} \dot{\tau}$ where $\dot{\tau}$ is either a τ or the special symbol $\circledast$. We elide C when it is understood. This relation is defined as the least relation satisfying the rules in Figure D.6. The decidability of this relation is subtle in the non-contractive case (e.g. $\{\texttt{`A}\,\alpha_1 \,\&\, \alpha_2 <: \alpha_2\} \cup C \vdash \alpha_2 \overset{\texttt{ref}}{\rightarrowtriangle} \dot{\tau}$), and an occurrence check is required.

Using this relation, we then define the partial function $C_\pi(\alpha) = \overleftrightarrow{\tau}$ such that $\alpha \overset{\pi}{\rightarrowtriangle} \tau' \Leftrightarrow \tau' \in \overleftrightarrow{\tau}$ and also that

$$\textsc{Integer Projection}\quad \frac{\texttt{int} <: \alpha \in C}{\alpha \xrightarrow{\texttt{int}} \texttt{int}} \qquad \textsc{Label Projection}\quad \frac{l\ \alpha' <: \alpha \in C}{\alpha \xrightarrow{l} l\alpha'}$$

$$\textsc{Reference Projection}\quad \frac{\texttt{ref}\ \alpha' <: \alpha \in C}{\alpha \xrightarrow{\texttt{ref}} \texttt{ref}\ \alpha'}$$

$$\textsc{Integer Failure}\quad \frac{\tau <: \alpha \in C \qquad \tau \notin \{\texttt{int}\ , \alpha'_1 \,\&\, \alpha'_2\}}{\alpha \xrightarrow{\texttt{int}} \ast}$$

$$\textsc{Label Failure}\quad \frac{\tau <: \alpha \in C \qquad \tau \notin \{l\ \alpha', \alpha'_1 \,\&\, \alpha'_2\}}{\alpha \xrightarrow{l} \ast}$$

$$\textsc{Reference Failure}\quad \frac{\tau <: \alpha \in C \qquad \tau \notin \{\texttt{ref}\ \alpha', \alpha'_1 \,\&\, \alpha'_2\}}{\alpha \xrightarrow{\texttt{ref}} \ast}$$

$$\textsc{Onion Left Projection}\quad \frac{\alpha_1 \,\&\, \alpha_2 <: \alpha \in C \qquad \alpha_1 \xrightarrow{\pi} \tau}{\alpha \xrightarrow{\pi} \tau}$$

$$\textsc{Onion Right Projection}\quad \frac{\alpha_1 \,\&\, \alpha_2 <: \alpha \in C \qquad \alpha_1 \xrightarrow{\pi} \ast \qquad \alpha_2 \xrightarrow{\pi} \dot{\tau}}{\alpha \xrightarrow{\pi} \dot{\tau}}$$

Figure D.6. Type Projection

$\alpha \xrightarrow{\pi} \ast$ does not hold. This is to say that $C_\pi(\alpha)$ is defined as the set of types which may be projected by π from α only when all such projections are successful. We use this function in the specification of the built-ins below to ensure that failed projections result in undefined type functions (and thus are recognized as type errors).

D.5 Arithmetic Built-In Example

We define two arithmetic operations, addition and integer comparison, in TinyBang as follows:

D.5.1 Addition

- $\texttt{+}(E, [x_1, x_2]) = \langle E, v_3 \rangle$ where v_3 is the sum of $E_{\texttt{int}}(x_1)$ and $E_{\texttt{int}}(x_2)$
- $\texttt{+}(C, [\alpha_1, \alpha_2]) = \langle C, \{\texttt{int}\} \rangle$ when $C_{\texttt{int}}(\alpha_1) = \{\texttt{int}\}$ and $C_{\texttt{int}}(\alpha_2) = \{\texttt{int}\}$

D.5.2 Integer Less Than or Equal To

- $\texttt{<=}(E, [x_1, x_2]) = \langle E', v_3 \rangle$ where
  - $x^{\texttt{<=}}$ is globally unique,
  - $E' = E \,\|\, [x^{\texttt{<=}} \texttt{=} \texttt{()}]$, and
  - v_3 is $\texttt{`True}\ x^{\texttt{<=}}$ when $E_{\texttt{int}}(x_1) \leq E_{\texttt{int}}(x_2)$ and $\texttt{`False}\ x^{\texttt{<=}}$ otherwise

$$\begin{array}{rcll} v & ::= & \ldots \mid \texttt{ref}\ x & \textit{values} \\ \mathring{v} & ::= & \ldots \mid \texttt{ref}\ x & \textit{pattern vals} \\ \\ \tau & ::= & \ldots \mid \texttt{ref}\ \alpha & \textit{types} \end{array}$$

Figure D.7. State Grammar Modifications

$$\textsc{Reference Cell}\quad \frac{E(x_0) = \texttt{ref}\ x_1 \qquad x'_0 \texttt{=} \texttt{ref}\ x'_1 \in \phi \qquad x_1 \ {}^{\odot}_{E}{\preceq}^{B}_{\phi}\ x'_1}{x_0\ {}^{\odot}_{E}{\preceq}^{B}_{\phi}\ x'_0}$$

$$[\![x_0 \texttt{=} \texttt{ref}\ x_1]\!]_{\mathrm{s}} = \mathring{\alpha}_0 \backslash \texttt{ref}\ \mathring{\alpha}_1 <: \mathring{\alpha}_0$$

$$\textsc{Reference Cell}\quad \frac{\texttt{ref}\ \alpha'_1 <: \alpha_0 \in V \qquad \texttt{ref}\ \alpha'_1 <: \alpha'_0 \in C}{\alpha_0 \backslash V \ll \alpha'_0 \backslash C}$$

$$\textsc{Reference Cell}\quad \frac{\begin{array}{c}\texttt{ref}\ \alpha_1 <: \alpha_0 \in V \\ \texttt{ref}\ \alpha'_1 <: \alpha'_0 \in V' \qquad \alpha_1\ {}^{\circledcirc}_{V}{\preceq}^{F}_{V'}\ \alpha'_1\end{array}}{\alpha_0\ {}^{\circledcirc}_{V}{\preceq}^{F}_{V'}\ \alpha'_0}$$

Figure D.8. State Rule Modifications

- $\texttt{<=}(C, [\alpha_1, \alpha_2]) = \langle C', \{\texttt{`True}\ \alpha^{\texttt{<=}}, \texttt{`False}\ \alpha^{\texttt{<=}}\} \rangle$ where
  - $\alpha^{\texttt{<=}}$ is globally unique,
  - $C' = C \cup \{\texttt{()} <: \alpha^{\texttt{<=}}\}$, and
  - $C_{\texttt{int}}(\alpha_1) = \{\texttt{int}\}$ and $C_{\texttt{int}}(\alpha_2) = \{\texttt{int}\}$

Other operations like subtraction and equality can be defined analagously.

D.6 State

We add state to TinyBang via explicit reference cells. This involves modifying the language and type grammars to include the reference cell construct and then modifying the appropriate relations accordingly. The grammar modifications appear in Figure D.7. Grammatically, reference cells are very similar to labels.

We also restrict the definition of a well-formed expression: to be well-formed, pattern matches beneath `ref`s must be trivial; that is, if $x_2 \texttt{=} \texttt{ref}\ x_1$ appears in a pattern, then $x_1 \texttt{=} \texttt{()}$ must also appear in that pattern.

Because state defines additional forms of value and type, it must also augment the definitions of slicing and pattern matching. Pattern matching on reference cells is quite like pattern matching on labels. Upon matching a reference cell variable, its contents are read and bound to the pattern variable. The corresponding rules appear in Figure D.8.

The restriction on the grammar of reference pattern matching is a way of avoiding the aliasing problem which arises on reference cells. If slicing were permitted to act on a reference cell variable, there would be two mechanisms for accessing it: the sliced variable and the pre-slice variable. By closure capture, the scape could access both; if their types are not identical (e.g. due to slicing), then an unsoundness can arise by writing to one cell and then reading from the other. The restriction we choose to impose is somewhat draconian and not entirely necessary, but we choose it for its simplicity and its clear soundness properties.

Because pattern matching provides a mechanism for reading from cells, we only need to define a builtin to update them. That builtin is as follows:

- $\texttt{:=}(E, [x_1, x_2]) = \langle E_1 \,\|\, [x' \texttt{=} v_2] \,\|\, E_2, \texttt{()}\rangle$ where
  - $E_1 \,\|[x' \texttt{=} v'] \,\|\, E_2 = E$
  - $E_{\texttt{ref}}(x_1) = \texttt{ref}\ x'$, and
  - $E(x_2) = v_2$
- $\texttt{:=}(C, [\alpha_1, \alpha_2]) = \langle C', \{\texttt{()}\}\rangle$ where
  - $C' = C \cup \{\alpha_2 <: \alpha' \mid \texttt{ref}\,\alpha' \in C_{\texttt{ref}}(\alpha_1)\}$

Observe that the type-level metatheoretic function $\texttt{:=}(-,-)$ introduces a transitivity constraint for each `ref` lower bound, at least one of which corresponds to the assignment in the expression rule. This means that each set expression (transitively) introduces a lower bound on the `ref` type variable. Every pattern match on the same `ref` will need to handle the type of *anything* that could have been assigned to it.