# Types for Flexible Objects

Pottayil Harisanker Menon, Zachary Palmer, Alexander Rozenshteyn, and Scott Smith

Department of Computer Science
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.
We present TinyBang, a small typed language, in which flexible object operations can be encoded. Our encodings center around a novel, asymmetric data conjoiner which is used to concatenate records and to construct dependently-typed functions. TinyBang's subtype constraint system ensures that all types are inferred; there are no data declarations or type annotations. We show that the previously mentioned operations can be seamlessly encoded in TinyBang. TinyBang also solves an open problem in OO literature: objects can be extended after being messaged without compromising on the expressiveness of subtyping. We prove TinyBang's type system both sound and decidable; all examples 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 [BF98,RS02,BBV11], 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 $\lambda$`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; [BF98], 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 [Pot00] 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 (`‘a 5`, for example, labels `5` with `‘a`), concatenation (`‘a 5 & ‘b 1`), 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 [AWL94], 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 [SM01]. For example, consider the type-indexed record `{foo = 45; bar = 22; 13}`: the untagged element and `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/Haskell `Some(3)`. Type-indexed records are thus a universal data type and lend themselves to flexible programming patterns in the same spirit as what lists did for Lisp and objects did for Smalltalk.

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 (on `bar` and the `int` field here). Asymmetric concatenation is key for supporting flexible object concatenation, as well as for standard notions of inheritance. We term the `&` operation *onioning*. We need not include any record syntax in core TinyBang since we may simply write e.g. `‘foo 45 & ‘bar 22 & 13`, the onioning of a sequence of labeled data, to express the above record. Our type formalism is built around onioning and this leads to a simple formalization of concatenation.

*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. We refer to a single clause as a *simple function*, and the conjunction of multiple clauses is a *compound function*. TinyBang's first-class functions generalize the first-class cases of [BAC06].

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. The type theory remains simple because the dependency is only on the variant. Dependently typed first-class cases are critical for typing our object encodings, a topic we discuss later in this section.

*Alignment of expressions, patterns, and types* Programming languages typically have quite distinct syntax for expressions, pattern matching, and their types, but we have found it beneficial to align them. The core TinyBang syntax unifies the expression and pattern syntax. The difference is in the interpretation: expressions are read "forward" to construct values while patterns are read "backwards" to destruct them. We use a nested expression (and pattern) grammar in our examples for readability, but the TinyBang core language is in A-translated form; this is particularly important because it leads to a very strong alignment between the expression and type grammars. Consider for example a single function application in A-normal form, expressed as `x1 = x2 x3`; our corresponding subtype constraint is $\alpha_2\alpha_3 <: \alpha_1$ (also commonly notated as $\alpha_3 \rightarrow \alpha_1 <: \alpha_2$). A similar strong notational analogy holds for TinyBang's labeling and concatenation operations.

This alignment allows the formal definition of the operational semantics and type inference to be compact in spite of the expressiveness of the language. The alignment of expression and type grammars also allows for us to prove type soundness by an approach different from subject reduction: a simple functor from expressions to types can be shown to provably achieve a stepwise simulation between the operational semantics and the type inference closure process. In the language of category theory, we can establish an adjunction between this functor and its inverse.

## 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 would work in TinyBang, 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 [Agh85] use an untyped form of variant-based objects. Although, 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 dependent types we use for cases solves this problem. Second, case matching is a monolithic operation in most languages, making it difficult to compose objects or create extensions. But, as mentioned above, TinyBang supports concatenation of case clauses; this is the root feature needed to support object concatenation and subclassing is easily encoded in this way.

*Resealing objects to support object extension* A key idea of [BF98] 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. Capturing `self` in a positive position – as a value bound in the message dispatch function – makes it covariant and so subtyping is not compromised.

*A broad collection of supported features* TinyBang also supports simple encodings of other standard object features: in the next section we cover object extension, subclassing and inheritance, mixins, default arguments, and operator overloading.

*Outline* In the next section, we give an overview of how TinyBang 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. Section 4 then illustrates how this core can be extended to the full semantics of TinyBang; this includes state and primitive operators. We conclude in Section 6.

## 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 1. The core TinyBang that is formalized is an A-normalized version of this grammar. The expression grammar contains the key features mentioned above: simple functions (written $\phi$ `->` $e$), onioning (written $e$ `&` $e$), and labeled data (written `` `Foo `` $e$ for a label constructor `` `Foo ``). State is also explicit and ML-style syntax is used.

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

**Fig. 1.** TinyBang Syntax

Program types take the form of a set of subtype constraints [AWL94]. For the purposes of this Overview we will be informal about type syntax. Our 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 $\alpha$ with constraints `int` $<: \alpha$ and `bool` $<: \alpha$. So, as we do with the 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 $\phi$ `->` $e$, with $\phi$ 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 = (‘double x -> x + x) in obj (‘double 4)
```

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

Unlike a traditional `match` expression, a simple function is only capable of matching one pattern. To express general match expressions, functions are appended 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 = (‘double x -> x + x) & (‘isZero x -> x == 0) in obj ‘double 4
```

The above shows that traditional `match` expressions can be encoded using the `&` operator to join a number of simple functions: one for each case. Function conjunction generalizes the first-class cases of [BAC06]; the aforecited work only supports adding one clause to the end and so does not allow "override" of an existing clause or "mixing" of two arbitrary sets of clauses.

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

will not typecheck; OCaml match/case expressions must return the same type in all case branches.[1] Instead, we give the function a dependent pattern type that is informally (`‘double int → int) & (‘isZero int → 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 ∪ 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 in a dimension critical for typing objects. These dependent pattern types generalize conditional constraint types [AWL94,Pot00].

*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 [BF98], 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`:

```
let fixpoint = f -> (g -> x -> g g x) (h -> y -> f (h h) y) in
let seal = fixpoint (seal -> obj ->
```

[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.

```
                         (msg -> obj (msg & ‘self (seal obj))) & obj) in
let obj = (‘double x -> x + x) &
          (‘quad x & ‘self self -> self (‘double x) + self (‘double x))
let sObj = seal obj in ...
```

The `seal` function is an object transformer which adds `‘self ...` as an implicit additional argument to all `obj` methods, "sealing" the self within the object. Looking at the code, the result `msg -> ...` is a new message handler onioned onto the left of `obj`. This message handler matches *every* message sent to `obj`, adds a `‘self` component to the message, and sends it on to `obj`. This message handler also recursively ensures that the value in `‘self` is *itself* a sealed object; this is the subtle reason why `fixpoint` is needed.

*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 = (‘double x -> x) & sObj in
let sObj2 = seal obj2 in
let four = sObj2 ‘quad 4 in ...              // returns 4
```

We can extend `sObj` after messaging it, here overriding the `‘double` 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 `‘double`. 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.

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* [SM01], 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` | $\cong$ ``(`x v -> !v) o`` |
| `o.x =` $e_1$ `in` $e_2$ | $\cong$ ``(`x v -> v :=`` $e_1$ `in` $e_2$`) o` |
| `if` $e_1$ `then` $e_2$ `else` $e_3$ | $\cong$ ``((`True _ ->`` $e_2$``) & (`False _ ->`` $e_3$`))` $e_1$ |
| $e_1$ `and` $e_2$ | $\cong$ ``((`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))           // returns 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. Note that one may prefer to disallow override of an existing overloading; defining an operator `&!` with this behavior is simple since it can simply check whether the patterns overlap.

*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 mixedPoint = seal (point & mixin) in mixedPoint ‘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 `mixedPoint`.

The above example typechecks in TinyBang; parametric polymorphism is used to allow `point`, `mixin`, and `mixedPoint` to have different self-types. The `mixin` variable, has the approximate type “(`‘near unit & ‘self` $\alpha$) $\rightarrow$ `bool` where $\alpha$ 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 mixedPoint = seal (point & (if cond then w1 else w2)) in
mixedPoint ‘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.

## 3 Formalization

We now give formal semantics to a subset of TinyBang. This subset includes functions, onions, and labels as described in Section 2; state and primitive operators are added in Section 4. Here, 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.

*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 $\overset{n}{\overrightarrow{g}}$. We elide the $n$ when it is unnecessary. Operator $\|$ denotes list concatenation. For sets, we use similar notation: $\overset{n}{\overbracket{g}}$ 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, $\overset{n}{\overrightarrow{g_{\square}/g'}}$ 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. The grammar of our A-normalized language appears in Figure 2. 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 2 as the *ANF* language.

| | | | |
|---|---|---|---|
| $e ::= \overrightarrow{s}$ | *expressions* | $\phi ::= \overrightarrow{x = \mathring{v}}$ | *patterns* |
| $s ::= x = v \mid x = x \mid x = x\ x$ | *clauses* | | |
| $E ::= \overrightarrow{x = v}$ | *environment* | $B ::= \overrightarrow{x = x}$ | *bindings* |
| $v ::= \mathbb{Z} \mid \texttt{()} \mid l\ x \mid x \,\texttt{\&}\, x \mid \phi \texttt{ -> } e$ | *values* | $\mathring{v} ::= \texttt{int} \mid \texttt{()} \mid l\ \mathring{v} \mid \mathring{v} \,\texttt{\&}\, \mathring{v}$ | *pattern vals* |
| $l ::=$ \` *(alphanumeric)* | *labels* | $x ::=$ *(alphanumeric)* | *variables* |

**Fig. 2.** TinyBang ANF Grammar

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 $\alpha$-renamed such that they do.

We now define the A-translation function $\wr e \int_x$. 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 operate on patterns as well. 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 $z$. Using this notation, our definition of this translation function appears in Figure 3.

The A-translation of $e$ is defined as $\wr e \int_y$ for some fresh $y$. The definition is straightforward in most respects. Notice that A-translation of patterns is in perfect parallel with the expressions in the above; for example, the expression

| **Expressions** | **Patterns** |
|---|---|
| $\llparenthesis \mathbb{Z} \rrparenthesis_x = [x = \mathbb{Z}]$ | $\llparenthesis \texttt{int} \rrparenthesis_x = [x = \texttt{int}]$ |
| $\llparenthesis \texttt{()} \rrparenthesis_x = [x = \texttt{()}]$ | $\llparenthesis \texttt{()} \rrparenthesis_x = [x = \texttt{()}]$ |
| $\llparenthesis l\ e \rrparenthesis_x = \llparenthesis e \rrparenthesis_y \Vert [x = l\ y]$ | $\llparenthesis l\ \phi \rrparenthesis_x = \llparenthesis \phi \rrparenthesis_y \Vert [x = l\ y]$ |
| $\llparenthesis e_1 \,\&\, e_2 \rrparenthesis_x = \llparenthesis e_1 \rrparenthesis_y \Vert \llparenthesis e_2 \rrparenthesis_z \Vert [x = y \,\&\, z]$ | $\llparenthesis \phi_1 \,\&\, \phi_2 \rrparenthesis_x = \llparenthesis \phi_1 \rrparenthesis_y \Vert \llparenthesis \phi_2 \rrparenthesis_z \Vert [x = y \,\&\, z]$ |
| $\llparenthesis \phi \texttt{ -> } e \rrparenthesis_x = [x = \llparenthesis \phi \rrparenthesis_y \texttt{ -> } \llparenthesis e \rrparenthesis_z]$ | |
| $\llparenthesis e_1\ e_2 \rrparenthesis_x = \llparenthesis e_1 \rrparenthesis_y \Vert \llparenthesis e_2 \rrparenthesis_z \Vert [x = y\ z]$ | |
| $\llparenthesis \texttt{let}\ x_1 = e_1\ \texttt{in}\ e_2 \rrparenthesis_{x_2} = \llparenthesis e_1 \rrparenthesis_{x_1} \Vert \llparenthesis e_2 \rrparenthesis_{x_2}$ | |
| $\llparenthesis x_2 \rrparenthesis_{x_1} = [x_1 = x_2]$ | $\llparenthesis x_2 \rrparenthesis_{x_1} = [x_2 = \texttt{()}]$ |

**Fig. 3.** TinyBang A-Translation

`‘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 is taken to match the argument when the function is applied, 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 is pattern matching, for which several auxiliary definitions are needed.

**Compatibility** The first basic relation we define is *compatibility*: is a value accepted by a given pattern? We write $x\ {}_E{\preceq}^B_\phi\ x'$ to indicate that the value $x$ is compatible with the pattern variable $x'$. $E$ represents the environment in which to interpret the value $x$ while $\phi$ represents the environment in which to interpret the pattern $x'$. $B$ dictates how, upon this successful match, the values from $E$ will be bound to the pattern variables in $\phi$. Compatibility is the least relation satisfying the rules in Figure 4. $B$ additionally must be minimal; that is, $x\ {}_E{\preceq}^B_\phi\ x'$ only holds if there exists no strict sublist $B'$ such that $x\ {}_E{\preceq}^{B'}_\phi\ x'$. We write $x_0\ {}_E{\npreceq}_\phi\ x'_0$ to indicate that $x_0\ {}_E{\preceq}^B_\phi\ x'_0$ does not hold for any $B$.

To better understand this relation, 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` ${}_E{\preceq}^B_\phi$ `p3` holds with the bindings $B$ shown in the third column.

$$\textsc{Integer}\quad \frac{x_0 = n \in E, n \in \mathbb{Z} \qquad x'_0 = \texttt{int} \in \phi}{x_0 \;{}_E{\preceq}^B_\phi\; x'_0}
\qquad
\textsc{Empty Onion}\quad \frac{x_0 = v \in E \qquad x'_0 = \texttt{()} \in \phi \qquad x'_0 = x_0 \in B}{x_0 \;{}_E{\preceq}^B_\phi\; x'_0}$$

$$\textsc{Label}\quad \frac{x_0 = l\ x_1 \in E \qquad x'_0 = l\ x'_1 \in \phi \qquad x_1 \;{}_E{\preceq}^B_\phi\; x'_1}{x_0 \;{}_E{\preceq}^B_\phi\; x'_0}$$

$$\textsc{Conjunction Pattern}\quad \frac{x'_0 = x'_1 \,\&\, x'_2 \in \phi \qquad x_0 \;{}_E{\preceq}^B_\phi\; x'_1 \qquad x_0 \;{}_E{\preceq}^B_\phi\; x'_2}{x_0 \;{}_E{\preceq}^B_\phi\; x'_0}
\qquad
\textsc{Onion Value Left}\quad \frac{x_0 = x_1 \,\&\, x_2 \in E \qquad x_1 \;{}_E{\preceq}^B_\phi\; x'_0}{x_0 \;{}_E{\preceq}^B_\phi\; x'_0}$$

$$\textsc{Onion Value Right}\quad \frac{x_0 = x_1 \,\&\, x_2 \in E \qquad x'_0 = x'_1 \,\&\, x'_2 \notin \phi \qquad x_2 \;{}_E{\preceq}^B_\phi\; x'_0 \qquad x_1 \;{}_E{\not\preceq}_\phi\; x'_0}{x_0 \;{}_E{\preceq}^B_\phi\; x'_0}$$

**Fig. 4.** Pattern compatibility rules

| $E$ | $\phi$ | $B$ |
|---|---|---|
| `v1 = 0` | `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` | |

**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 $\phi \,\texttt{->}\, e$ and a series of pattern clauses are expressed by onioning together simple functions. We thus need to define an application matching relation $x_0\ x_1 \rightsquigarrow_E e$ to determine if an onion of pattern clauses $x_0$ can be applied to argument $x_1$. Matching is defined as the least relation satisfying the rules in Figure 5; helper function RV extracts the return variable from a value, formally defined as follows: $\mathrm{RV}(\vec{e} \,\|\, [x = v]) = x$, and $\mathrm{RV}(\vec{e} \,\|\, [x = \mathring{v}]) = x$. We use the notation $x_0\ x_1 \not\rightsquigarrow_E$ to mean that there exists no $e$ such that $x_0\ x \rightsquigarrow_E e$.

$$\textsc{Function}\quad \frac{x_0 = (\phi \,\texttt{->}\, e) \in E \qquad x_1 \;{}_E{\preceq}^B_\phi\; \mathrm{RV}(\phi)}{x_0\ x_1 \rightsquigarrow_E B \,\|\, e}
\qquad
\textsc{Onion Left}\quad \frac{x_0 = x_2 \,\&\, x_3 \in E \qquad x_2\ x_1 \rightsquigarrow_E e}{x_0\ x_1 \rightsquigarrow_E e}$$

$$\textsc{Onion Right}\quad \frac{x_0 = x_2 \,\&\, x_3 \in E \qquad x_2\ x_1 \not\rightsquigarrow_E \qquad x_3\ x_1 \rightsquigarrow_E e}{x_0\ x_1 \rightsquigarrow_E e}$$

**Fig. 5.** Application matching rules

The Function rule is the base case of a simple function application: the argument value $x_1$ must be compatible with the pattern $\phi$, 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 if the (higher priority) left side has failed to match.

**Operational Semantics** Using the compatibility and matching relations from above, we now define the operational semantics of restricted TinyBang as a small step relation $e \longrightarrow^1 e'$. 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 $\alpha$-renaming function which freshens 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 6.

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

$$\textsc{Application}\quad \frac{x_0\ x_1 \rightsquigarrow_E e' \qquad \boldsymbol{\alpha}(e') = e''}{E \,\|\, [x_2 = x_0\ x_1] \,\|\, e \longrightarrow^1 E \,\|\, e'' \,\|\, [x_2 = \mathrm{RV}(e'')] \,\|\, e}$$

**Fig. 6.** The operational semantics small step relation

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 bindings 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 [AW93] and suitable for complex pattern matching [Pot00] and object-orientation [WS01].

Constraint sets have been used previously to reason about program variables as sets of potential runtime values [Hei94]. We adopt this strategy and take it a few steps further. First, we build in a close syntactic correspondence between program expressions and types. Second, we do not take the standard approach of presenting a declarative ruleset for typing and an independent type inference algorithm for that ruleset; instead, we only present the inference algorithm, in the form of a constraint closure process which deductively closes on a set of subtyping constraints. Operational semantics single-stepping then has a close alignment to a single step of this constraint closure process, so close that a formal simulation relationship is not hard to establish: there is a functor from expressions to types that is a simulation.

$$
\begin{array}{lll}
C ::= \overbrace{c} & & \textit{constraint sets} \\
c ::= \tau <: \alpha \mid \alpha <: \alpha \mid \alpha\ \alpha <: \alpha & & \textit{constraint} \\
V ::= \overbrace{\tau <: \alpha} & & \textit{constraint value sets} \\
F ::= \overbrace{\alpha <: \alpha} & & \textit{constraint value sets} \\
\tau ::= \texttt{int} \mid \texttt{()} \mid l\ \alpha \mid \alpha \,\texttt{\&}\, \alpha \mid \alpha\backslash V \to \alpha\backslash C & & \textit{types} \\
\alpha & & \textit{type variables}
\end{array}
$$

**Fig. 7.** The TinyBang type grammar

**Initial Alignment** Figure 7 provides a grammar of the type system.

There is a very close alignment between expression and type grammar elements; $E$ has type $V$, and $B$ has type $F$. Expressions $e$ have types $\alpha\backslash C$; these are less different than they appear, since the expression clauses are a list the last variable contains the final value implicitly whereas in the type the final type location $\alpha$ must be explicit. Both $v$ and $\mathring{v}$ have type $\tau$; this is because, while the correspondence from $v$ is lossy (all $\mathbb{Z}$ map to `int`, for example), the correspondence from $\mathring{v}$ is not: all patterns are singleton-typed.

We formalize this initial alignment step as a function $\llbracket e \rrbracket_{\mathrm{E}}$ which produces a constrained type $\alpha\backslash C$; see Figure 8. 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 corresponding constraint $\ldots <: \overset{\star}{\alpha}_0$. Note that functions $\phi \texttt{->} e$ produce types $\alpha'\backslash V \to \alpha\backslash C$, patterns only contain value constraints $V \subseteq C$.

$$
\begin{aligned}
\llbracket \overrightarrow{s}^{\,n} \rrbracket_{\mathrm{E}} &= \alpha_n \backslash \overbrace{c}^{n} \text{ where } \forall i \in \{1..n\}. \llbracket s_i \rrbracket_{\mathrm{S}} = \alpha_i \backslash c_i \\
\llbracket \overrightarrow{x \texttt{=} \mathring{v}}^{\,n} \rrbracket_{\mathrm{P}} &= \alpha_n \backslash \overbrace{c}^{n} \text{ where } \forall i \in \{1..n\}. \llbracket x_i \texttt{=} \mathring{v}_i \rrbracket_{\mathring{\mathrm{S}}} = \alpha_i \backslash c_i
\end{aligned}
$$

$$
\begin{aligned}
\llbracket x_0 \texttt{=} \mathbb{Z} \rrbracket_{\mathrm{S}} &= \overset{\star}{\alpha}_0 \backslash\ \texttt{int} <: \overset{\star}{\alpha}_0 &
\llbracket x_0 \texttt{=} \texttt{int} \rrbracket_{\mathring{\mathrm{S}}} &= \overset{\star}{\alpha}_0 \backslash\ \texttt{int} <: \overset{\star}{\alpha}_0 \\
\llbracket x_0 \texttt{=} \texttt{()} \rrbracket_{\mathrm{S}} &= \overset{\star}{\alpha}_0 \backslash\ \texttt{()} <: \overset{\star}{\alpha}_0 &
\llbracket x_0 \texttt{=} \texttt{()} \rrbracket_{\mathring{\mathrm{S}}} &= \overset{\star}{\alpha}_0 \backslash\ \texttt{()} <: \overset{\star}{\alpha}_0 \\
\llbracket x_0 \texttt{=} l\ x_1 \rrbracket_{\mathrm{S}} &= \overset{\star}{\alpha}_0 \backslash\ l\ \overset{\star}{\alpha}_1 <: \overset{\star}{\alpha}_0 &
\llbracket x_0 \texttt{=} l\ x_1 \rrbracket_{\mathring{\mathrm{S}}} &= \overset{\star}{\alpha}_0 \backslash\ l\ \overset{\star}{\alpha}_1 <: \overset{\star}{\alpha}_0 \\
\llbracket x_0 \texttt{=} x_1 \,\texttt{\&}\, x_2 \rrbracket_{\mathrm{S}} &= \overset{\star}{\alpha}_0 \backslash\ \overset{\star}{\alpha}_1 \,\texttt{\&}\, \overset{\star}{\alpha}_2 <: \overset{\star}{\alpha}_0 &
\llbracket x_0 \texttt{=} x_1 \,\texttt{\&}\, x_2 \rrbracket_{\mathring{\mathrm{S}}} &= \overset{\star}{\alpha}_0 \backslash\ \overset{\star}{\alpha}_1 \,\texttt{\&}\, \overset{\star}{\alpha}_2 <: \overset{\star}{\alpha}_0 \\
\llbracket x_0 \texttt{=} \phi \texttt{->} e \rrbracket_{\mathrm{S}} &= \overset{\star}{\alpha}_0 \backslash\ \llbracket \phi \rrbracket_{\mathrm{P}} \to \llbracket e \rrbracket_{\mathrm{E}} <: \overset{\star}{\alpha}_0 \\
\llbracket x_0 \texttt{=} x_1 \rrbracket_{\mathrm{S}} &= \overset{\star}{\alpha}_0 \backslash\ \overset{\star}{\alpha}_1 <: \overset{\star}{\alpha}_0 \\
\llbracket x_0 \texttt{=} x_1\ x_2 \rrbracket_{\mathrm{S}} &= \overset{\star}{\alpha}_0 \backslash\ \overset{\star}{\alpha}_1\ \overset{\star}{\alpha}_2 <: \overset{\star}{\alpha}_0
\end{aligned}
$$

**Fig. 8.** Initial alignment

Next, we must define the type system constraint closure relation. At a high level, the type system formalization contains relations paralleling the operational semantics: below we define compatibility, matching, and single-step closure relations which are close analogues of the operational semantics versions of the previous section. There is however one novel relation not found in the operational semantics, the *slicing* relation; we now discuss the need for that relation and present its definition.

**Slicing** Any type system must conservatively approximate runtime behavior for typechecking to be decidable. The particular places our system loses information is individual integers are approximated by `int` in the type system; and, the call stack must somehow be finitely approximated. A result of these imprecisions is that some type variables will have more than one lower bound. For example, execution of clause `x1 = x2 == x3` in the operational semantics will store only one of `‘True` or `‘False` in `x1`, whereas the corresponding constraint set is the union $\alpha_1\backslash\{$`‘True` $\alpha_2 <: \alpha_1,$ `‘False` $\alpha_2 <: \alpha_1,$ `()` $<: \alpha_2\}$. Now, consider the pattern `‘True _ & ‘False _`, which should match values with *both* labels. It seems like the type variable $\alpha_1$ should match this pattern if it has both a `‘True` lower bound and a `‘False` lower bound, but a constraint set like the one above represents a *union*, not an intersection. So, a naïve porting of the definitions in the operational semantics will fail on this account.

The solution we take is to perform union elimination before making the compatibility check. This way, compatibility is guaranteed union-free input. We call these union-free subsets a *slice* of the original type. Slices solve the above problem because we would first consider the slice of the $\alpha_1$ type containing just the `‘True` part of the union, and then just the `‘False` part.

Slices are for the most part easy to define, but there is a subtlety for the case that the value has been collapsed to a recursive data type (an effect of the finitization of the call stack as alluded to above); for instance, $\alpha\backslash\{$`‘A` $\alpha <: \alpha, \texttt{int} <: \alpha\}$ expresses the union of the types `int`, `‘A` $\alpha$, `‘A ‘A` $\alpha$, and so on. Fortunately, there is always a limit to how deeply a type must be sliced in order to determine if a pattern matches it.

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 defined as follows:

$$\textsc{Leaf}\quad \frac{\alpha \notin V}{\alpha\backslash V \ll \alpha\backslash C} \qquad \textsc{Atomic}\quad \frac{\tau \in \{\texttt{int}\,,\texttt{()}\,,\alpha_1\backslash V' \rightarrow \alpha_2\backslash C'\} \qquad \tau <: \alpha \in V \qquad \tau <: \alpha' \in C}{\alpha\backslash V \ll \alpha'\backslash C}$$

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

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

**Fig. 9.** The slice relation for union elimination

**Definition 1.** *$\alpha\backslash V \ll \alpha'\backslash C$ is the least relation defined by the rules in Figure 9. Slices are*

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

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.

**Compatibility** With slicing defined, it is now possible to define the type system compatibility relation in parallel with how expression compatibility was defined. Type compatibility relates a slice $\alpha\backslash V$ to a pattern $\alpha'\backslash V'$ in the context of some bindings $F$. Slicing imposes one more detail on compatibility: slices may be variable in depth, and it is possible for the slice to be too shallow. For instance, consider the type $\alpha_0\backslash\{\texttt{`A}\ \alpha_1 <: \alpha_0, \texttt{`B}\ \alpha_2 <: \alpha_1, \texttt{`C}\ \alpha_2 <: \alpha_1, \texttt{int} <: \alpha_2\}$ (which is informally `` `A (`B `` $\texttt{int} \cup$ `` `C int)``). When matching the nested pattern `` `A (`B int & `C int)``, it is possible that a slice of $\alpha_0$ may not be deep enough to eliminate the union; for instance, the slice $\alpha'_0\backslash\{\texttt{`A}\ \alpha_1 <: \alpha'_0\}$ has picked a lower bound for $\alpha_0$ but not for $\alpha_1$. There is no problem of insufficient depth in the expression evaluation system; expression function application is only attempted when the function and argument are values already. But the constraint closure (below) is not so linearly ordered and the value types may not yet be fully formed.

To address this problem, the compatibility relation here has an additional place indicating whether the match was "complete" (represented $\bullet$) or "partial" (represented $\bigcirc$). We use metavariable $\LEFTcircle$ to range over these two values. The relation of compatibility is only partial when the slice is insufficiently deep. We define type system compatibility $\alpha\ {}^{\bullet}_{V}{\preceq}^{F}_{V'}\ \alpha'$ to be the least relation such that (1) it satisfies the rules appearing in Figure 10 and (2) the binding set $F$ is minimal. In the case of conjunction, we let $\LEFTcircle_1 \cap \LEFTcircle_2$ be $\bullet$ when $\LEFTcircle_1 = \LEFTcircle_2 = \bullet$ and let it be $\bigcirc$ otherwise.

Compatibility failure $\alpha_0\ {}_{V}{\not\preceq}_{V'}\ \alpha'_0$ holds when neither partial nor full compatibility holds for any set of bindings: $\forall F, \LEFTcircle. \neg(\alpha_0\ {}^{\LEFTcircle}_{V}{\preceq}^{F}_{V'}\ \alpha'_0)$. We let $\alpha_0\ {}_{V}{\preceq}^{F}_{V'}\ \alpha'_0$ abbreviate $\alpha_0\ {}^{\bullet}_{V}{\preceq}^{F}_{V'}\ \alpha'_0$. The failure case is subtle: we only want to fail when we have a deep enough slice to know we completely checked all possibilities. Partial matching was introduced to ensure that too-shallow slices are considered neither a failure nor a full match.

**Matching** Type matching directly parallels expression matching. A match is either partial or full based on whether the underlying compatibility check was partial or full. Matching $\alpha_0\ \alpha_1\ {}^{\LEFTcircle}_{V_0}{\leadsto}_{V_1}\ \alpha_2\backslash C'$ is defined as the least relation satisfying the clauses in Figure 11. We write $\alpha_0\ \alpha_1\ {}_{V_0}{\not\leadsto}_{V_1}$ when there exist no $\LEFTcircle$, $\alpha$, and $C'$ such that $\alpha_0\ \alpha_1\ {}^{\LEFTcircle}_{V_0}{\leadsto}_{V_1}\ \alpha'\backslash C'$.

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

$$\textsc{Partial}\quad \frac{\nexists \tau . \tau <: \alpha_0 \in V}{\alpha_0 \; {}^{\circ}_{V}{\preceq}^{F}_{V'} \; \alpha_0'}$$

$$\textsc{Integer}\quad \frac{\texttt{int} <: \alpha_0 \in V \qquad \texttt{int} <: \alpha_0' \in V'}{\alpha_0 \; {}^{\bullet}_{V}{\preceq}^{F}_{V'} \; \alpha_0'}$$

$$\textsc{Empty Onion}\quad \frac{\tau <: \alpha_0 \in V \qquad \texttt{()} <: \alpha_0' \in V' \qquad \alpha_0 <: \alpha_0' \in F}{\alpha_0 \; {}^{\bullet}_{V}{\preceq}^{F}_{V'} \; \alpha_0'}$$

$$\textsc{Label}\quad \frac{l\ \alpha_1 <: \alpha_0 \in V \qquad l\ \alpha_1' <: \alpha_0' \in V' \qquad \alpha_1 \; {}^{\LEFTcircle}_{V}{\preceq}^{F}_{V'} \; \alpha_1'}{\alpha_0 \; {}^{\LEFTcircle}_{V}{\preceq}^{F}_{V'} \; \alpha_0'}$$

$$\textsc{Conjunction Pattern}\quad \frac{\alpha_1' \mathbin{\&} \alpha_2' <: \alpha_0' \in V' \qquad \alpha_0 \; {}^{\LEFTcircle_1}_{V}{\preceq}^{F}_{V'} \; \alpha_1' \qquad \alpha_0 \; {}^{\LEFTcircle_2}_{V}{\preceq}^{F}_{V'} \; \alpha_2'}{\alpha_0 \; {}^{\LEFTcircle_1 \cap \LEFTcircle_2}_{V}{\preceq}^{F}_{V'} \; \alpha_0'}$$

$$\textsc{Onion Value Left}\quad \frac{\alpha_1 \mathbin{\&} \alpha_2 <: \alpha_0 \in V \qquad \alpha_1 \; {}^{\LEFTcircle}_{V}{\preceq}^{F}_{V'} \; \alpha_0'}{\alpha_0 \; {}^{\LEFTcircle}_{V}{\preceq}^{F}_{V'} \; \alpha_0'}$$

$$\textsc{Onion Value Right}\quad \frac{\alpha_1 \mathbin{\&} \alpha_2 <: \alpha_0 \in V \qquad \alpha_1' \mathbin{\&} \alpha_2' <: \alpha_0' \notin V' \qquad \alpha_2 \; {}^{\LEFTcircle}_{V}{\preceq}^{F}_{V'} \; \alpha_0' \qquad \alpha_1 \; {}_{V}{\not\preceq}_{V'} \; \alpha_0'}{\alpha_0 \; {}^{\LEFTcircle}_{V}{\preceq}^{F}_{V'} \; \alpha_0'}$$

**Fig. 10.** Type compatibility: does a value type match a pattern?

to indicate the freshening of the variables in $C$. The additional parameter $\alpha$ describes the call site at which the polyinstantiation took place; this is useful for some polymorphism models. The second function, $\Upsilon$, is a merging function; it is used to unify type variables from different polyinstantiations when recursion is detected. Our closure is parametric in the pair of functions $\Phi/\Upsilon$; the simplest model would be a monomorphic type system given by $\Phi_{\text{Mono}}(C, \alpha) = C$ and $\Upsilon_{\text{Mono}}(C) = C$. 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 12 are satisfied. 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:

**Definition 2 (Inconsistency).** *A constraint set $C$ is* inconsistent *iff 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 $\alpha_0'\ \alpha_1'\ {}_{V_0}{\not\succ}_{V_1}$. A constraint set which is not inconsistent is* consistent.

Like constraint closure, inconsistency is also implicitly parametric in $\Phi/\Upsilon$. The intention of this definition is to capture the cases in which an expression can get stuck. The only case in which application will not evaluate is when the compound function cannot match the argument. Since a slice represents a set of

$$\textsc{Function}\quad \dfrac{(\alpha'\backslash V' \to \alpha\backslash C) <: \alpha_0 \in V_0 \qquad \alpha_1 \overset{\text{◐}}{{}_{V_1}\!\preceq}{}^{F}_{V'}\ \alpha'}{\alpha_0\ \alpha_1 \overset{\text{◐}}{{}_{V_0}\!\rightsquigarrow}{}_{V_1}\ \alpha\backslash C \cup F}$$

$$\textsc{Onion Left}\quad \dfrac{\alpha_2 \,\&\, \alpha_3 <: \alpha_0 \in V_0 \qquad \alpha_2\ \alpha_1 \overset{\text{◐}}{{}_{V_0}\!\rightsquigarrow}{}_{V_1}\ \alpha\backslash C}{\alpha_0\ \alpha_1 \overset{\text{◐}}{{}_{V_0}\!\rightsquigarrow}{}_{V_1}\ \alpha\backslash C}$$

$$\textsc{Onion Right}\quad \dfrac{\alpha_2 \,\&\, \alpha_3 <: \alpha_0 \in V_0 \qquad \alpha_2\ \alpha_1\ {}_{V_0}\!\not\rightsquigarrow_{V_1} \qquad \alpha_3\ \alpha_1 \overset{\text{◐}}{{}_{V_0}\!\rightsquigarrow}{}_{V_1}\ \alpha\backslash C}{\alpha_0\ \alpha_1 \overset{\text{◐}}{{}_{V_0}\!\rightsquigarrow}{}_{V_1}\ \alpha\backslash C}$$

**Fig. 11.** Type application matching

$$\textsc{Transitivity}\quad \dfrac{\{\tau <: \alpha_1, \alpha_1 <: \alpha_2\} \subseteq C}{C \Longrightarrow^1 C \cup \{\tau <: \alpha_2\}}$$

$$\textsc{Application}\quad \dfrac{\alpha_0\ \alpha_1 <: \alpha_2 \in C \qquad \alpha_0'\backslash V_0 \ll \alpha_0\backslash C \qquad \alpha_1'\backslash V_1 \ll \alpha_1\backslash C \qquad \alpha_0'\ \alpha_1' \overset{\bullet}{{}_{V_0}\!\rightsquigarrow}{}_{V_1}\ \alpha'\backslash C'}{C \Longrightarrow^1 \Upsilon(\Phi(C' \cup \{\alpha' <: \alpha_2\}, \alpha_2) \cup C \cup V_1)}$$

**Fig. 12.** Type constraint closure single-step relation

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

This definition is also, of course, implicitly parametric in $\Phi/\Upsilon$.

**A worked example** Here we work through a small example to clarify how the relations interact.[2] 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. See Figure 13 for the A-translation and initial alignment. This means that the constraint set corresponding to the program is $C = \big\{\alpha_{\texttt{neg}}\ \alpha_{\texttt{arg}} <: \alpha_{\texttt{r}}, \texttt{int} <: \alpha_{\texttt{arg}}, \texttt{`True}\ \alpha_{eo} <: \alpha_{\texttt{arg}}, \texttt{()} <: \alpha_{eo}, \texttt{`False}\ \texttt{()} <: \alpha_{\texttt{arg}}, \alpha_0\backslash V_0 \to \alpha_1\backslash C_1 <: \alpha_{\texttt{iId}}, \alpha_2\backslash\{\texttt{`True}\ \texttt{()} <: \alpha_2\} \to \alpha_3\backslash\{\texttt{`False}\ \texttt{()} <: \alpha_3\} <: \alpha_4, \alpha_5\backslash\{\texttt{`False}\ \texttt{()} <: \alpha_5\} \to \alpha_6\backslash\{\texttt{`True}\ \texttt{()} <: \alpha_6\} <: \alpha_7, \alpha_4 \,\&\, \alpha_7 <: \alpha_8, \alpha_8 \,\&\, \alpha_{\texttt{iId}} <: \alpha_{\texttt{neg}}\big\}$. We also take $V_f$ to be the largest set such that $\alpha_f\backslash V_f \ll \alpha_{\texttt{neg}}\backslash C$ (our system will consider

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

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

**Fig. 13.** Worked example

other such sets, 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 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 can demonstrate that $\alpha_4\ \alpha\ {}_{V_f}\!\not\twoheadrightarrow_{V}$ because we can demonstrate that $\alpha\ {}_{V}\!\not\preceq_{V'}\ \alpha_2$ (where $V' = \{\texttt{`True ()} <: \alpha_2\}$) by inspection. This allows us to apply the onion right match rule (and once again by similar logic on $\alpha_5$) If we can show that $\alpha\ {}_{V}\!\overset{\bullet}{\preceq}{}^{F}_{V_0}\ \alpha_0$, 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 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.

**Soundness and Decidability of Typing** We now give the formal assertions of type soundness and the decidability of type inference. Proofs of these theorems appear in the appendices of our supplementary material. As discussed above, we prove soundness 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. Informally, we write $e \preccurlyeq C$ to indicate that $e$ is simulated by $C$. This simulation is the alignment we have discussed throughout this section; for instance, $\texttt{5} \preccurlyeq \texttt{int}$ and, for initial alignment, $x_0 \preccurlyeq \overset{\star}{\alpha}_0$. Given this relation, soundness is asserted as follows.

**Theorem 1 (Soundness).** *Given simulation-preserving functions $\Phi$ and $\Upsilon$, if $e \longrightarrow^* e'$ where $e'$ is stuck then $e$ does not typecheck.*

We say that $\Phi$ is simulation-preserving if $e \preccurlyeq C$ implies $e \preccurlyeq \Phi(C, \alpha)$ for any $\alpha$. We define simulation preservation over $\Upsilon$ similarly.

The proof strategy is direct. We show the result of initial alignment is a simulation, and that simulation is preserved through each of the relations in the evaluation and type systems, in particular through single-step operational semantics and constraint closure. A full proof appears in Appendix A.

The second formal property is decidability. This theorem requires that $\Phi$ and $\Upsilon$ be constrained to introduce a fixed number of variables into any produced constraint set given an initial constraint set. We define $\Phi/\Upsilon$ with this property to be *finitely freshening*; the full definition appears in Appendix C. We state our decidability theorem as follows:

**Theorem 2 (Decidability).** *For any fixed, finitely freshening, computable $\Phi$ and $\Upsilon$, typechecking is decidable.*

A proof of this theorem appears in Appendix B.

### 3.4 A Useful Polymorphism Model: $\boldsymbol{\Phi}_{\mathrm{CR}}/\boldsymbol{\Upsilon}_{\mathrm{CR}}$

Here we take a brief detour to describe our 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
```

In the above code, `factory` creates simple polymorphic value container. In a `let`-bound model, the type given to `f` in the scope of `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 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 [Shi91,Age95] and has previously been ported to a type constraint context [WS01]. 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 regular expressions [MS06]. 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. We call the regular expressions *contours*. The full definition of our $\Phi_{\mathrm{CR}}/\Upsilon_{\mathrm{CR}}$ function appear in Appendix C, along with a proof that this polymorphism model is finitely freshening as discussed in the previous section.

## 4 Extended Features

Here, we present a few straightforward extensions to the TinyBang ANF which support primitive operators and state.

## 4.1 Builtins

We begin with builtin operators for primitive data. These operators are introduced to the language via a generic framework; we introduce that framework and then give an example of its use by defining the builtin for integer addition. The framework itself involves small modifications to the language grammar, the A-translation process, and the operational semantics as depicted in Figure 14.

In particular, the operational semantics merely evaluates the newly added builtin clause by deferring to a metatheoretic function ($\boxdot(-,-,-)$) that defines the each operator's behavior. Rather than A-translating builtin clauses directly, we translate them to applications of a curried binary function bound to the variable $x^{\boxdot}$, which is expected to pattern match on the arguments before evaluating the builtin with them.

$$\begin{array}{lll} s ::= x = \boxdot(x, x) \mid \ldots & & \textit{clauses} \\ \boxdot ::= + \mid \ldots & & \textit{builtins} \end{array}$$

$$\wr e_1 \boxdot e_2 \wr_x = \wr x^{\boxdot}\ e_1\ e_2 \wr_x$$

$$\textsc{Builtin}\quad \frac{x_1 = v_1 \in E \qquad x_2 = v_2 \in E \qquad \boxdot(E, v_1, v_2) = \langle E', v\rangle}{E \,\|\, x_0 = \boxdot(x_1, x_2) \,\|\, e \longrightarrow^1 E' \,\|\, x_0 = v \,\|\, e}$$

**Fig. 14.** Generic TinyBang builtin extensions

As a concrete case, we now consider extending the system to support addition on integers; this is defined in Figure 15 in three parts. The first part defines the metatheoretic function for the behavior of this operation; a redex `x = y + z` will be reduced by the Builtin rule above using this definition. We choose to "hide" the binary operation inside of a simple function definition (the second part of Figure 15) which we assume to be bound in the environment; we can then model this function in the type system using a fabricated type (the third part of Figure 15) which we ensure is part of the initial constraint set in closure.

$+(E, v_1, v_2) = \langle E, v_3\rangle$ where $v_3$ is the sum of the integer projections of $v_1$ and $v_2$

$$x^{+} = [x_1^{+} = \texttt{int}]\ \texttt{->}\ [x_2^{+} = [x_3^{+} = \texttt{int}]\ \texttt{->}\ [x_4^{+} = +(x_1^{+}, x_3^{+})]]$$

$$\alpha_1^{+}\backslash\{\texttt{int} <: \alpha_1^{+}\} \to \alpha_2^{+}\backslash\{\alpha_3^{+}\backslash\{\texttt{int} <: \alpha_3^{+}\} \to \alpha_4^{+}\backslash\{\texttt{int} <: \alpha_4^{+}\} <: \alpha_2^{+}\} <: \alpha^{+}$$

**Fig. 15.** Addition-specific TinyBang builtin extensions

## 4.2 State

Adding state to TinyBang is only slightly more involved than adding addition. As seen in Figure 16, we use the label `ref` to create cells, and we add assignment as a builtin operator following Section 4.1. We do not need builtins for reading from or creating a ref; labels and pattern matching do this. Note that the constraint $\alpha_4^{\texttt{<-}} <: \alpha_2^{\texttt{<-}}$ corresponds to the addition of the new binding to the environment.

| | |
|---|---|
| Grammar | $l ::= \texttt{ref} \mid \ldots$ *labels* <br> $\boxdot ::= \texttt{<-} \mid \ldots$ *builtins* |
| Behavior | $\texttt{<-}(E, v_1, v_2) = \langle E', \texttt{()} \rangle$ where $v_1$ has a projection $\texttt{ref}\ x$, <br> $E = E_1 \parallel x \,\texttt{=}\, v \parallel E_2$, and $E' = E_1 \parallel E_2 \parallel x \,\texttt{=}\, v_2$ |
| Function | $x^{\texttt{<-}} \texttt{ = } \left[x_1^{\texttt{<-}} \texttt{ = ()}\, , x_2^{\texttt{<-}} \texttt{ = ref } x_1^{\texttt{<-}}\right] \texttt{ -> } \left[x_3^{\texttt{<-}} \texttt{ = } [x_4^{\texttt{<-}} \texttt{ = ()}] \texttt{ -> } [x_5^{\texttt{<-}} \texttt{ = <-}(x_2^{\texttt{<-}}, x_4^{\texttt{<-}})]\right]$ |
| Type | $\alpha_1^{\texttt{<-}} \backslash \{ \texttt{()} <: \alpha_2^{\texttt{<-}}, \texttt{ref}\ \alpha_2^{\texttt{<-}} <: \alpha_1^{\texttt{<-}} \} \rightarrow \alpha_3^{\texttt{<-}} \backslash C <: \alpha^{\texttt{<-}}$ <br> where $C = \alpha_4^{\texttt{<-}} \backslash \{ \texttt{()} <: \alpha_4^{\texttt{<-}} \} \rightarrow \alpha_5^{\texttt{<-}} \backslash \{ \texttt{()} <: \alpha_5^{\texttt{<-}}, \alpha_4^{\texttt{<-}} <: \alpha_2^{\texttt{<-}} \} <: \alpha_3^{\texttt{<-}}$ |

**Fig. 16.** Ref grammar and semantics

We need to make one more change to the system. We amend the slice definition from Figure 9 to include the rule in Figure 17, which triggers *instead* of the existing label rule. This rule only permits trivial pattern matching under a ref; less restrictive models exist, but we choose this one to simplify presentation.

$$\textsc{Ref}\quad \frac{\texttt{ref}\ \alpha_1 <: \alpha_0 \in V \qquad \texttt{ref}\ \alpha_1 <: \alpha_0' \in C \qquad \alpha_1 \backslash V \ll \alpha_1 \backslash C}{\alpha_0 \backslash V \ll \alpha_0' \backslash C}$$

**Fig. 17.** Ref slice rule extending Figure 9

To see why this rule is necessary, consider the code below on the left. If this program were run, it would get stuck evaluating the fourth line on an addition of 0 to a non-integer value.

Without the slice rule, the constraint that $x$ can contain `()` will not be properly added to the global constraint set; instead, the constraint set would only observe that $x$ contains `()` within the body of the set function on line 3. The slice rule effectively mandates that constraints applied to local ref variables also apply to their external sources.

With the slice rule, our state model would correctly reject this program, because we would eventually conclude that $\{\texttt{ref}\ \alpha_1 <: \alpha_x, \texttt{()} <: \alpha_1, \texttt{int} <: \alpha_1, \alpha^{\texttt{+}}\ \alpha_1 <: \alpha_2\}$ That is to say that there is a call to the addition function with an empty onion argument, which results in a failed pattern match. On the other hand, consider the code on the right in the following example:

```
1 let x = ref 0 in
2 let f = (() -> !x + 1) in
3 let _ = x <- () in
4 f ()
```

```
1 let x = ref () in
2 let f = (() -> !x + 1) in
3 let _ = x <- 0 in
4 f ()
```

While this code would execute correctly, we have chosen a flow-insensitive model of state and the code is rejected because we reach the same conclusion as above.

## 5 Related Work

We believe that TinyBang offers the first high-level algebraic case composition operator which can directly support object inheritance for a variant encoding of objects. The key type system requirement is for different branches of a conditional or case clause to be typed differently; this property is called path-sensitivity in program analyses. Our approach is a generalization of conditional constraints [AWL94,Pot00]; conditional constraints can partly capture path sensitivity, but fail to track the dependencies through side effects.

TinyBang's object resealing is inspired by the Bono-Fisher object calculus [BF98], 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 [RS02], for instance, the power to extend a messaged object comes at the cost of depth subtyping. In [BBV11], depth subtyping is admitted but the type system is nominal rather than structural.

Typed multimethods [CGL95,MC99] perform a dispatch similar to TinyBang's dispatch on compound functions, but multimethod dispatch is *nominally* typed while compound function dispatch is *structurally* typed. The first-class cases in [BAC06] allow composition of case branches much in the same fashion as TinyBang. In [BAC06], however, cases may only be extended by adding a clause to the end of the list; it is not possible to concatenate two arbitrary case blocks or to prepend a case onto the beginning of a block. TinyBang generalizes this work by making the previous two operations possible; these operations are necessary for the encoding of e.g. method override, which must intercept a call and so must be a prepended case.

TinyBang's onions are unusual in that they are a form of record supporting typed asymmetric concatenation. Wand initially proposed asymmetric record concatenation for modeling inheritance [Wan91] but the inference algorithm was fundamentally exponential. Standard typed record-based encodings of inheritance [BCP99] 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 [Pot00]; TinyBang uses a different approach that does not need row typing and that we believe is simpler.

Our approach to parametric polymorphism in Section 3.4 is based on flow analysis [Shi91,WS01]. 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 for more expressive polymorphism since call cycles don't need to be conservatively approximated up front. Contours are merged by their injection into a restricted space of regular expressions over call strings, an approach that follows program analyses [MS06].

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 [Shi91,Age95,MS06] and a traditional type system. Without such a technique, type soundness can be very challenging given how expressive the type constraints are; subject reduction would require significant machinery and there is no reason-able regular tree or other form of denotational type model [AC93,Vou06,CX11] of TinyBang given the flexibility of pattern matching. For example, polymorphism in TinyBang cannot be convex in the sense of [CX11] since our pattern matching syntax is expressive enough to distinguish between all top-level forms.

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; [Dun12] contains a review of existing approaches. Our novel solution is the notion of a *slice* in Section 3.3 which can be viewed as a (locally) complete notion of union elimination; since each slice embodies only *one* disjunct relative to the patterns being matched, there is nothing to mis-align.

While there are many programming language designs to which we could compare TinyBang, we offer TinyBang more as a calculus to showcase novel typing ideas (such as the object extension typing problem we address here). A comparison of TinyBang with languages that have real-world implementations is well outside of the scope and focus of this paper.

## 6 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 the semantics of the onion operator `&` which permits both asymmetric concatenation (for default parameters) and generalized extensible case matching (for capturing messages as they are dispatched).

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 we would de-sugar that code to TinyBang. While the fundamental ideas in this paper could be used to give types and runtime semantics to BigBang directly, we focus on encodings for objects and other features because they yield a few benefits. First: TinyBang's object calculus operations can be defined as in-language functions, avoiding the need for object-specific metatheory. Second: our encoding is simple enough to be *translucent* in that it will be possible, though atypical, to "work under the hood" to manipulate objects on a lower level when necessary. Third: this translucency ensures an elegant encoding and attests to the expressiveness of the underlying language constructs.

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. We are currently developing a means to support dynamic loading and separate compilation when lightweight, approximate type signatures are placed on module boundaries.

Our appendices include proofs for the soundness (Appendix A) and decidability (Appendix B) of our type inference system as well as our polymorphism model (Appendix C). We have implemented the type inference algorithm and interpreter for an older version of the TinyBang formalism to provide a cross-check on the soundness of our ideas. We believe that inference is not only decidable but, in practice, polynomial based on our experience to date. It is not provably polynomial for the same reason that let-polymorphism is not: artificial programs exist which will exhibit exponential runtimes. All of the examples in Section 2 typecheck in our implementation without exponential blowup.

## References

AC93. Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Trans. Program. Lang. Syst.*, pages 575–631, September 1993.

Age95. Ole Agesen. The cartesian product algorithm. In *ECOOP*, volume 952 of *Lecture Notes in Computer Science*, 1995.

Agh85. G.A. Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, 1985.

AW93. A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *FPCA*, pages 31–41, 1993.

AWL94. A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *POPL 21*, pages 163–173, 1994.

BAC06. Matthias Blume, Umut A. Acar, and Wonseok Chae. Extensible programming with first-class cases. In *ICFP*, pages 239–250, 2006.

BBV11. Lorenzo Bettini, Viviana Bono, and Betti Venneri. Delegation by object composition. *Science of Computer Programming*, 76:992–1014, 2011.

BCP99. Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1-2):108–133, 1999.

BF98. Viviana Bono and Kathleen Fisher. An imperative, first-order calculus with object extension. In *ECOOP'98*, pages 462–497. Springer Verlag, 1998.

CGL95. G. Castagna, G. Ghelli, , and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.

CX11. Giuseppe Castagna and Zhiwu Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *ICFP*, 2011.

Dun12. Joshua Dunfield. Elaborating intersection and union types. In *ICFP*, 2012.

Hei94. Nevin Heintze. Set-based analysis of ML programs. In *LFP*, pages 306–317. ACM, 1994.

MC99. Todd D. Millstein and Craig Chambers. Modular statically typed multimethods. In *ECOOP*, pages 279–303. Springer-Verlag, 1999.

MS06. Matthew Might and Olin Shivers. Environment analysis via $\Delta$CFA. In *POPL*, 2006.

Pot00. François Pottier. A versatile constraint-based type inference system. *Nordic J. of Computing*, 7(4):312–347, 2000.

RS02. Jon G. Riecke and Christopher A. Stone. Privacy via subsumption. *Inf. Comput.*, 172(1):2–28, February 2002.

Shi91. Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie-Mellon University, 1991. TR CMU-CS-91-145.

SM01. Mark Shields and Erik Meijer. Type-indexed rows. In *POPL*, pages 261–275, 2001.

Vou06. Jérôme Vouillon. Polymorphic regular tree types and patterns. In *POPL*, 2006.

Wan91. Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, July 1991.

WS01. Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. In *ECOOP*, pages 99–117, 2001.

## 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 $x \to \alpha$ 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 4.** *The simulation relation $\preccurlyeq_M$ is defined as the least relation satisfying the rules in Figure 18. The notation $x \preccurlyeq \alpha$ is used to denote $\exists M.\, x \preccurlyeq_M \alpha$; similar notation holds for other grammatical constructs.*

| | | | |
|---|---|---|---|
| $\mathbb{Z}$ | $\preccurlyeq_M$ | `int` | |
| `()` | $\preccurlyeq_M$ | `()` | |
| $l\ x$ | $\preccurlyeq_M$ | $l\ \alpha$ | iff $x \preccurlyeq_M \alpha$ |
| $x_1$ `&` $x_2$ | $\preccurlyeq_M$ | $\alpha_1$ `&` $\alpha_2$ | iff $x_1 \preccurlyeq_M \alpha_1$ and $x_2 \preccurlyeq_M \alpha_2$ |
| $\phi$ `->` $e$ | $\preccurlyeq_M$ | $\alpha'\backslash V \to \alpha\backslash C$ | iff $\phi \preccurlyeq_M \alpha'\backslash V$ and $e \preccurlyeq_M \alpha\backslash C$ |
| $x$ | $\preccurlyeq_M$ | $\alpha$ | iff $M(x) = \alpha$ |
| `int` | $\preccurlyeq_M$ | `int` | |
| $x$ `=` $v$ | $\preccurlyeq_M$ | $\tau <: \alpha$ | iff $x \preccurlyeq_M \alpha$ and $v \preccurlyeq_M \tau$ |
| $x_2$ `=` $x_1$ | $\preccurlyeq_M$ | $\alpha_1 <: \alpha_2$ | iff $x_1 \preccurlyeq_M \alpha_1$ and $x_2 \preccurlyeq_M \alpha_2$ |
| $x_3$ `=` $x_1\ x_2$ | $\preccurlyeq_M$ | $\alpha_1\ \alpha_2 <: \alpha_3$ | iff $x_1 \preccurlyeq_M \alpha_1$ and $x_2 \preccurlyeq_M \alpha_2$ and $x_3 \preccurlyeq_M \alpha_3$ |
| $x$ `=` $\mathring{v}$ | $\preccurlyeq_M$ | $\tau <: \alpha$ | iff $x \preccurlyeq_M \alpha$ and $\mathring{v} \preccurlyeq_M \tau$ |
| $e$ | $\preccurlyeq_M$ | $C$ | iff $\forall s \in e.\, \exists c \in C.\, s \preccurlyeq_M c$ |
| $e$ | $\preccurlyeq_M$ | $\alpha\backslash C$ | iff $\textsc{rv}(e) \preccurlyeq_M \alpha$ and $e \preccurlyeq_M C$ |
| $\phi$ | $\preccurlyeq_M$ | $V$ | iff $\forall x = \mathring{v} \in e.\, \exists c \in V.\, x = \mathring{v} \preccurlyeq_M c$ |
| $\phi$ | $\preccurlyeq_M$ | $\alpha\backslash V$ | iff $\textsc{rv}(\phi) \preccurlyeq_M \alpha$ and $\phi \preccurlyeq_M V$ |

**Fig. 18.** Simulation relation

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

**Lemma 1.** *If $e$ is closed then $e \preccurlyeq [\![e]\!]_{\mathrm{E}}$.*

*Proof.* This is to say that, for some $M$, $e \preccurlyeq_M [\![e]\!]_{\mathrm{E}}$. Suppose that, for all $s$, we have $s \preccurlyeq_M \alpha\backslash\{c\}$ when $[\![s]\!]_{\mathrm{S}} = \alpha\backslash c$. 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 4. 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 $\phi \preccurlyeq_M [\![\phi]\!]_{\mathrm{P}}$ and that $e' \preccurlyeq_M [\![e']\!]_{\mathrm{E}}$. 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 $\phi$, it suffices to show that for all $x = \mathring{v}$ we have $x = \mathring{v} \preccurlyeq_M \alpha\backslash\{\tau <: \alpha\}$ when $[\![x = \mathring{v}]\!]_{\mathring{\mathrm{S}}} = \alpha\backslash\tau <: \alpha$. 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).

## 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 19.

$$\textsc{Atomic}\quad \frac{x' = v \in E' \qquad v \in \{\mathbb{Z}, \texttt{()}, \phi \texttt{->} e\}}{x\backslash[x = v] \lll x'\backslash E'} \qquad\qquad \textsc{Label}\quad \frac{x_0 = l\ x_1 \in E' \qquad x_1\backslash E \lll x_1'\backslash E'}{x_0\backslash E \,\|\, [x_0 = l\ x_1] \lll x_0'\backslash E'}$$

$$\textsc{Onion}\quad \frac{x_0 = x_1 \,\&\, x_2 \in E' \qquad x_1\backslash E_1 \lll x_1'\backslash E_1' \qquad x_2\backslash E_2 \lll x_2'\backslash E_2'}{x_0\backslash E_1 \,\|\, E_2 \,\|\, [x_0 = x_1 \,\&\, x_2] \lll x_0'\backslash E'}$$

**Fig. 19.** Deep copy relation

**Definition 5.** *$x'\backslash E' \lll x\backslash E$ is the least relation defined by the rules in Figure 19. 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.

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 2.** *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 4. 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 6.** *The shallow bisimulation relation $\approx_M$ is defined as the least relation satisfying the rules in Figure 20. 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 3.** *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:

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

**Fig. 20.** Shallow bismulation relation

**Lemma 4.** *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 $\alpha$ 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 2. 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 $\alpha_0$ 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 6 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 5.** *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 $B$, $x_0 \ {}_{E}{\preceq}^{B}_{\phi}\ x'_0$ if and only if there exists some $F$ such that $B \approx_M F$ and $\alpha_0 \ \overset{\bullet}{{}_{V}}{\preceq}^{F}_{V'}\ \alpha'_0$*

*Proof.* Inspection of Figures 4 and 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).

In the case where the Label rule of value compatibility applies, $x_2 = \texttt{`A} \in x_1$ and $x'_2 = \texttt{`A}\ x'_1 \in \phi$ and $x_1 \ {}_{E}{\preceq}^{B}_{\phi}\ x'_1$. 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 rule in the relation. The same approach is used in the contrapositive to show that a proof of full type compatibility evidences a proof of value compatibility, thus completing the equivalence.

The remaining cases of compatibility are proven likewise.

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

**Lemma 6.** *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 $x_0 = \mathrm{RV}(E_0)$ and $x_1 = \mathrm{RV}(E_1)$. Let $E = E_0 \,\|\, E_1$. Then $x_0\ x_1 \leadsto_E e$ if and only if there exists some $\alpha' \backslash C$ such that $e \approx_M \alpha' \backslash C$ and $\alpha_0\ \alpha_1 \ \overset{\bullet}{{}_{V_0}}{\leadsto}_{V_1}\ \alpha' \backslash C$.*

*Proof.* By the same strategy as Lemma 5, using Lemma 5 to show the compatibility premises.

**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 21.
We define $e_0 \ \dot{\longrightarrow}^{*}\ e_n$ when $\forall 1 \leq i \leq n.\, e_{i-1} \ \dot{\longrightarrow}^{1}\ e_i$. We say $e \ \dot{\not\longrightarrow}^{1}$ when there exists no $e'$ such that $e \ \dot{\not\longrightarrow}^{1}\ e'$.

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

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

$$\textsc{Application}\quad \frac{x_0'\backslash E_0' \lll x_0\backslash E \qquad x_1'\backslash E_1' \lll x_1\backslash E \qquad x_0'\ x_1' \rightsquigarrow_{E_0' \,\|\, E_1'} e' \qquad \boldsymbol{\alpha}(e') = e''}{E \,\|\, [x_2 = x_0\ x_1] \,\|\, e \overset{\cdot}{\longrightarrow}{}^1 E \,\|\, E_1' \,\|\, e'' \,\|\, [x_2 = \mathrm{RV}(e'')] \,\|\, e}$$

**Fig. 21.** Alternative Small Step Relation

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 7.** *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 $\alpha$-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 8 (Preservation).** *Suppose that $e \overset{\cdot}{\longrightarrow}{}^1 e'$ and that $e \preccurlyeq C$. Further, suppose that $\Phi$ and $\Upsilon$ are simulation-preserving. Then there exists some $C'$ such that $C \Longrightarrow C'$ and $e' \preccurlyeq C'$.*

*Proof.* In a fashion similar to Lemma 4, 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.

Otherwise, the Application rule applies. For clarity, we use variable names matching those chosen in the rules in Figures 12 and 21. Then we know that we have an expression $E \,\|\, [x_2 = x_0\ x_1] \,\|\, e$; 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 4 and 6, we have that $e' \preccurlyeq \alpha'\backslash C'$.

Because $\alpha$-renaming does not prevent simulation, we have that $e'' \preccurlyeq_{M'} \alpha'\backslash C'$ and so $e'' \preccurlyeq_{M'} C'$. Because all variables in $e''$ are fresh, we know that $M$ and $M'$ are disjoint; thus, we can let $M'' = M \cup M'$ and all previous simulations are in terms of this new mapping. We also have that $x_2 = \mathrm{RV}(e'') \preccurlyeq_{M''} \alpha' <: \alpha_2$ from previous simulations. Because of this and because $\Phi$ is simulation-preserving,

$e'' \,\|\, [x_2 = \mathrm{RV}(e'')] \preccurlyeq_{M''} \Phi(C' \cup \{\alpha' <: \alpha_2\}, \alpha_2)$. Because $\Upsilon$ is simulation preserving, we union original $C$ and the argument slice $V_1$ with this result; this corresponds with the concatenation of the original environment $E$ and the deep copy variables $E'_1$ onto the above. This concatenation and union are also simulation preserving 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 9.** *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 $x_2 = x_0\ x_1$; otherwise, the Variable Lookup rule always applies (because $e$ is closed and $s$ is of the form $x_2 = x_1$). By Lemma 2, the deep copy premises always hold for closed $E$; furthermore, $\alpha$-renaming is always possible. Therefore, because $e$ is stuck, we must have that $x'_0\ x'_1 \not\rightsquigarrow_{E'_0 \,\|\, E'_1}$. By Lemma 6 we have that $\alpha'_0\ \alpha'_1\ {}_{V_0}\!\not\twoheadrightarrow_{V_1}$ . By Definition 2, $C$ 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 1 gives us that $e \preccurlyeq \alpha\backslash C$. We have that $e \longrightarrow^* e'$ where $e'$ is stuck; then by Lemma 7, we have that $e \overset{\cdot}{\longrightarrow}{}^* e'$ where $e' \overset{\cdot}{\not\longrightarrow}{}^1$ and $e'$ not of the form $E$. By induction on the number of steps in the proof of $e \overset{\cdot}{\longrightarrow}{}^* e'$, Lemma 8 gives us that that there exists a $C'$ such that $C \Longrightarrow^* C'$ and $e' \preccurlyeq C'$. Lemma 9 gives us that $C'$ is inconsistent. Therefore $C$ closes to an inconsistent constraint set and, by Definition 3, $e$ does not typecheck.

## B Proof of Decidability

We show here that TinyBang typechecking is decidable, in particular we provide a detailed sketch of Theorem 2.

The primary challenge of this proof is showing type closure is finite, more precisely that there are only finitely many constraint sets $C$ which can be the global closure state. We prove this property by a high level counting argument: we define a finite set of constraint forms and a set of type variables, and 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.

Slices intuitively need to be only so deep as to cover the pattern, but the definition of slicing placed no limit on the size of a slice and also required all variables in a slice to be fresh. So, we need to modify type inference to restrict the depth of slices. These changes will be made to the application closure rule, and then we need to verify 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 1, 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 $\alpha$ 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 $\alpha$ will be the unique type variable in $V$ without an upper bound. In a slice the type variables in the leaves, if any, are informally considered *free*, and the type variables internal to the slice as well as 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 $\tau_i$ for $i > 1$, $\alpha_{i+1}$ occurs in $\tau_i$. 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 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 and Integer rules (the leaf cases 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 makes sense that it will be sufficient.

Thus, taking the set of all pattern types in $C_{00}$ to be $\overbracket{V'}$, we would like to define $n_{maxslice}$ to be $n_{max\text{-}\phi} = max(\{depth(V') \mid V' \in \overbracket{V'}\})$, 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 structurally 1-1 with patterns because patterns have conjunctions in them which are not directly matching with onions. In particular, the Onion Value Left and Onion Value Right compatibility rules in Figure 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 leftmost copy matters, the repetitions need not be considered. So, given that there are $n_{\&}$ onion constraints in $C_{00}$ and $n_{\tau}$ lower bounds in $C_{00}$, we then need to slice to depth at most $n_{maxslice} = n_{max\text{-}\phi} * n_{\&} * n_{\tau}$: 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 $\alpha$-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 $\alpha$-conversion.

**Lemma 10.** *Set $VV_{allslice}$ is of finite cardinality.*

*Proof.* Each slice $V \in VV_{allslice}$ 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 $\alpha$-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.1 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, so they may appear in the global constraint set $C$. In our type inference implementation, we directly wire in the argument without adding any new type variables to the set, but we use slices here for a more elegant theory. While it 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.

The assertion of decidability, Theorem 2, assumes that our renaming function $\Phi_{\mathrm{CR}}/\Upsilon_{\mathrm{CR}}$ is finitely freshening. This property is formally defined in Definition 13 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 $\overbracket{\alpha}$ 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 any constraint set $C$ being fed into $\Phi_{\mathrm{CR}}/\Upsilon_{\mathrm{CR}}$ is a subset of $\overbracket{\alpha}$, thus the renaming and merging will also produce variables all in $\overbracket{\alpha}$. Thus, $\overbracket{\alpha}$ constitutes all type variables that may ever appear in any constraint set and that set is finite, proving finiteness of all paths.

## B.2 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 7 (application closure).** *Revise the Application closure rule of Figure 12 to insert the following additional preconditions:*

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

*where* $\mathit{unify}(V_1, V_2)$ *requires* $V_1$ *and* $V_2$ *to have the same tree structure and only be* $\alpha$*-variants of one another, and replaces leaf type variables of* $V_2$ *with the corresponding leaf type variables in* $V_1$*. In the Application rule conclusion we additionally replace slice* $V_1$ *with the (equivalent)* $V_3$*. (Note that* $V'$ *is the pattern from the function successfully matched on, it formally needs to be added as an additional argument to that relation so it can propagate to the Application rule.)*

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 19 is modified to remove the restrictions on $v$ in the Atomic rule: the deep copy need not stop only at atoms, it can stop at any point, meaning it could be no copy at all or it could be a full deep copy, or it could be any degree in between.

The bisimulation relation of Figure 20 can remain 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 21 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 none): an analogy of Lemma 7 holds for the revised small-step relation for *all* possible nondeterministic paths chosen.

Lastly, Preservation (Lemma 8) 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.3 Decidability

Given the above reformulation of application, we can now show that there are only finitely many closure states, and closure is thus a finite state machine.

We first show that construction of the initial derivation, slicing, compatibility, and matching are all easily decidable.

**Lemma 11.** *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 $V'$,* $\exists F.\alpha_0 \; {}^{\LEFTcircle}_{V}\!\preceq^{F}_{V'} \; \alpha'_0$
4. *Given potential slice $V$ and pattern type $V'$,* $\alpha_0 \; {}_{V}\!\not\preceq_{V'} \; \alpha'_0$
5. *Given potential slices $V, V'$,* $\exists \alpha_2, C'.\alpha_0 \; \alpha_1 \; {}^{\LEFTcircle}_{V}\!\rightsquigarrow_{V'} \alpha_2\backslash C'$
6. $\alpha_0 \; \alpha_1 \; {}_{V}\!\not\rightsquigarrow_{V'}$

*Proof.* 1. is trivially computable since the definition of Figure 8 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. and 4. are mutually referencing and so must simultaneously be proven decidable. Inspecting the compatibility rules of Figure 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. (Observe also that the subgoal of not-compatible in Onion Value Right is on $\alpha_1$, a type variable in a slice subtype, so the negation proof search is similarly bounded). The mapping $F$ is deterministically synthesized up the proof tree so is easy to compute.

5. and 6. follow similarly to 3. and 4.; observe here that $\alpha_2\backslash C'$ is deterministically synthesized up the proof tree.

**Lemma 12.** *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.

Since all the above relations in closure are decidable/computable, and there are only finitely many total constraint set states $C$ to consider as potential states in closure (due to the constraint forms all coming from $C_{00}$ and the type variables restricted to the finite set $\overbrace{\alpha}$ produced by the finitely freshening assumption), all potential closure paths can be exhaustively searched and Theorem 2 has been established.

## 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 $\Phi_{\mathrm{CR}}/\Upsilon_{\mathrm{CR}}$

The $\Phi_{\mathrm{CR}}/\Upsilon_{\mathrm{CR}}$ 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 $\alpha$ 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 $\langle \alpha_{\texttt{x}}, \alpha_{\texttt{y}} \rangle \Leftrightarrow \alpha$ 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 8.** *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 $x_2 \texttt{=} x_0\ x_1$)*

*We use $\epsilon$ 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 9.** *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 $\langle \alpha, \varnothing \rangle \Leftrightarrow \alpha'$".

**Polyinstantiation and Merging** 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 $\alpha$ 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 the polymorphism model. 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 $\alpha_1'$ for a variable $\alpha_1$ and $\langle \alpha_2, \mathcal{C} \rangle \Leftrightarrow \alpha_1$, then $\langle \alpha_2', \mathcal{C} \rangle \Leftrightarrow \alpha_2$ for some $\alpha_2'$ 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 $\Phi_{\mathrm{CR}}$ and $\Upsilon_{\mathrm{CR}}$ functions. We define $\Phi_{\mathrm{CR}}$ to instantiate variables appearing in a constraint set as follows:

**Definition 10.** *Let $\mathcal{C}$ be a well-formed contour. Then $\Phi_{\mathrm{CR}}(C, \mathcal{C})$ is the capture-avoiding substitution of type variables in $C$ such that, for any $\alpha$, each free type variable $\langle \alpha, \varnothing \rangle$ is replaced with $\langle \alpha, \mathcal{C} \rangle$ and all other type variables remain unchanged.*

Calling $\Phi_{\mathrm{CR}}$ 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.

The second function, $\Upsilon_{\mathrm{CR}}$, is used to relate variables already appearing in the constraint set with the newly polyinstantiated ones from $\Phi_{\mathrm{CR}}$. In particular, $\Upsilon_{\mathrm{CR}}$ 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 $\alpha_3$ for the call stack is composed of call site $\alpha_1$ followed by any number of recursive calls to call site $\alpha_2$. But when $\alpha_2$ 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.

Before defining our merge operation, we rely on some simple supporting definitions:

**Definition 11.** *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 $\alpha$ and $\alpha'$ are* connected *in $C$ iff either $\alpha$ and $\alpha'$ overlap or there exists some $\alpha''$ appearing in $C$ such that $\alpha$ overlaps with $\alpha''$ and $\alpha''$ and $\alpha'$ 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 $\alpha$ with a non-empty contour overlaps itself and connects with itself.) Using the above, we can give the following definition of $\Upsilon_{\mathrm{CR}}$:

**Definition 12.** *For a set of constraints $C$, $\Upsilon_{\mathrm{CR}}(C)$ is the $\alpha$-substitution of type variables in $C$ such that each $\alpha$ is replaced by $\langle \alpha', \mathcal{C}' \rangle$ where $\overbrace{\langle \alpha', \mathcal{C}_{\square} \rangle}^{n}$ is the set of variables connected to $\alpha$ in $C$ and $\mathcal{C}'$ is the least well-formed contour greater than all $\overbrace{\mathcal{C}}^{n}$.*

In our above example, the least well-formed contour $\mathcal{C}'$ is $\alpha_1(\alpha_2)^*$; thus the type variable $\langle\alpha_3, \alpha_1\alpha_2\rangle$ would be replaced by $\langle\alpha_3, \alpha_1(\alpha_2)^*\rangle$.

## C.2 Simulation Preservation

The TinyBang type system is sound when using $\Phi_{\mathrm{CR}}/\Upsilon_{\mathrm{CR}}$ as a polymorphism model. To prove this, we must show that both $\Phi_{\mathrm{CR}}$ and $\Upsilon_{\mathrm{CR}}$ are simulation-preserving. We begin with the latter function because it is somewhat simpler.

**Theorem 3.** *If $e \preccurlyeq C$ then $e \preccurlyeq \Upsilon_{\mathrm{CR}}(C)$.*

*Proof.* Let $C' = \Upsilon_{\mathrm{CR}}(C)$. We have that $e \preccurlyeq_M C$. We also have that $C'$ is an $\alpha$-substitution of type variables in $C$; thus, if e.g. $\tau <: \alpha \in C$ then $\tau <: \alpha' \in C'$ such that $\alpha$ was replaced by $\alpha'$. If we subject $M$ to the same $\alpha$-substitution as $C$, we can show by induction on the size of $e \preccurlyeq_M C$ that $e \preccurlyeq_{M'} C'$.

We must make a similar argument for polyinstantiation, but this argument is somewhat complicated by the capture-avoiding substitution. This is because, when capture is avoided, it is possible that some instances of a type variable have been replaced and others have not; if that type variable were representing a single evaluation variable, simulation would no longer hold. Fortunately, we can show that this case does not arise:

**Theorem 4.** *Suppose we are typechecking an expression containing $e$. If $e \preccurlyeq C$ then $e \preccurlyeq \Phi_{\mathrm{CR}}(C, \alpha)$.*

*Proof.* Let $C' = \Phi_{\mathrm{CR}}(C, \alpha)$. We have that $e \preccurlyeq_M C$ and that $C'$ is a capture avoiding substitution of variables in $C$. We consider each type variable $\alpha$ appearing in $C$ being replaced by some other $\alpha'$. If $\alpha$ only appears free or only appears captured in $C$, then this is an $\alpha$-substitution of $\alpha$ and the argument proceeds as in Theorem 3.

If $\alpha$ 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 $\alpha$ 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.3 Decidability

The decidability proof requires that the polymorphism model 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 13 (Finitely Freshening $\Phi/\Upsilon$).** *A polymorphism model $\Phi/\Upsilon$ is* finitely freshening *iff* $\forall f : \overset{\sqcup}{e} \to \overset{\sqcup}{\overset{\sqcup}{\alpha}}.\, \exists g : \overset{\sqcup}{e} \to \overset{\sqcup}{I}.\, \forall e.$ *we have all of the following:*

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

By this definition, we argue that the polymorphism model $\Phi_{\mathrm{CR}}/\Upsilon_{\mathrm{CR}}$ is finitely freshening using contours as indices and the type variable bijection as the basis for $\overset{\sqcup}{\alpha}$. 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 14.** *Let* $\textsc{AllContours}(-)$ *be a function taking a set of type variables $\overset{\sqcup}{\alpha}$ and producing the set $\overset{\sqcup}{\mathcal{C}}$ 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 13.** *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.

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

**Theorem 5.** *The polymorphism model $\Phi_{\mathrm{CR}}/\Upsilon_{\mathrm{CR}}$ is finitely freshening.*

*Proof.* We are given a function $f$ as described in Definition 13. 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 $\epsilon$ – and leave the remainder of the bijection unconstrained. Since neither $\varnothing$ and $\epsilon$ require literals, they are in $\textsc{AllContours}(f(e))$. We therefore define $\overset{\sqcup}{\alpha} = \{\langle \alpha', \mathcal{C} \rangle \mid \alpha' \in f(e) \wedge \mathcal{C} \in g(e)\}$.

It remains to show that each of $\Phi_{\mathrm{CR}}$ and $\Upsilon_{\mathrm{CR}}$ has the desired quality: that, given $C$ with variables in $\overset{\sqcup}{\alpha}$, the resulting $C'$ has only variables in $\overset{\sqcup}{\alpha}$. For $\Phi_{\mathrm{CR}}$, we observe that the function 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 $\overset{\sqcup}{\alpha}$ and so this property holds. The argument for $\Upsilon_{\mathrm{CR}}$ proceeds likewise.

Finally, we must show that the functions are themselves computable.

**Theorem 6.** *Both $\mathit{\Phi}_{\mathrm{CR}}$ and $\mathit{\Upsilon}_{\mathrm{CR}}$ are computable.*

*Proof.* $\mathit{\Phi}_{\mathrm{CR}}$ is a capture-avoiding substitution on a finite set and so is trivially decidable. Whether two variables $\alpha$ and $\alpha'$ overlap in a finite constraint set $C$ is decidable by enumeration of all possible paths between those variables. We have a finite set of well-formed contours for a given $e$ 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. $\mathit{\Upsilon}_{\mathrm{CR}}$ is an $\alpha$-substitution on a finite set using these least well-formed contours; therefore, $\mathit{\Upsilon}_{\mathrm{CR}}$ is also decidable.