A Practical, Typed Variant Object Model

Or, How to Stand On Your Head and Enjoy the View

Pottayil Harisanker Menon Zachary Palmer Alexander Rozenshteyn Scott Smith

The Johns Hopkins University

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

Abstract

Traditionally, typed objects have been encoded as records; the fields and methods of an object are stored in the fields of a record and projected when needed. While the dual approach of using variants instead of records has been explored, it is more challenging to type: the output type of a variant case match must depend on the input value; this is a form of dependent typing.

In this paper, we construct a variant-based encoding of objects which is statically typeable and which improves on the flexibility of typed object models in several dimensions: messages can be represented as simple first-class data, object extension is more generally typeable than in previous systems and, arguably, a better integration of objects and functions is obtained.

This encoding is possible due to the features of our new core language, TinyBang, which incorporates a subtype constraint type inference system with several novel extensions. We develop a generalized notion of first-class cases – functions with an inherent pattern match that are composable – and we extend previous notions of conditional constraint types to obtain accurate typings. For added flexibility, TinyBang's record-like structure is type-indexed, meaning data can be projected based on its type alone. Our record structure's concatenation operator is asymmetric by default, naturally supporting object extension. Finally, we develop a refined notion of parametric polymorphism which aims to achieve a good combination of flexibility and efficiency of inference.

1. Introduction

Typed objects today sit on a foundation of records and subtyping: object members are stored in record fields and are projected on access, update, or invocation [9]. While many formalizations of typed objects are direct and not defined by a translation to records [1], they are in spirit not far from this underlying record view.

While records have been the most common form of encoding, there is a well-known duality between records and variants which means tagged variants and a match/case over them can also be used to encode objects. This *variant encoding* has been effectively used in dynamically-typed settings: an object is a function which receives a message in the form of a variant, matches on that variant, and executes the appropriate branch of code. Instead of invoking a method by projecting a `foo` field from a record and passing arguments to the result, one passes a `Foo(...)` message to the object and allows the object to invoke the appropriate function. The variant-based encoding has the advantage of representing object messages as first-class data that can be directly manipulated. This view is the object model of actors [3] and is the most natural for distributed objects since the packet data being sent across the network to a remote object is directly realized as a variant.

Unfortunately, it is well-known that the variant-based encoding of objects is difficult to type; all languages currently supporting a variant-based view of objects are dynamically typed. The root of the typing problem is that a match expression's cases are generally required to each produce the same type, meaning all methods of the encoded objects would have to have the same return type to typecheck (ouch!). In the record-based encoding, record fields are heterogeneously typed and so are not afflicted in this way. So while the duality is perfect in the dynamically-typed setting, it is not perfect using standard type theory.

In this paper, we present a small core language, TinyBang, in which a variant-based encoding of objects is typeable. TinyBang improves on the flexibility of typed object models in several dimensions. We can represent messages as simple first-class data; existing work on first-class message typing generally requires messages to be "frozen" under a function abstraction whereas our messages can be direct tagged data. Also, object extension is more generally typeable than in previous type theories for extensible objects; this results in an arguably better integration of objects and functions.

In order to achieve these expressiveness advantages, TinyBang contains several novel approaches to record/variant syntax and typing. The variants and records are different enough from the standard variety that we elect to use different names: *onions* are our record-like constructor, and *scapes*[1] are the variant match/case destructor. Onions and scapes combine and improve on several language design ideas, including the concepts of *first-class cases*, *dependent pattern types*, *type-indexed records*, and *asymmetric concatenation*. A high-level overview of these features is as follows.

Scapes: generalizing first-class cases Traditional case/match statements are monolithic blocks that lack an explicit composition operator on case constructs. Such a composition is key to encoding objects as variants because, if objects are case matches on messages, object extension/subclassing/mixin is case composition.

The solution is a (typed) notion of first-class cases, which has been explored previously in [6]. There, a construct $\oplus$ is defined for extending a case with an additional single clause. Our scapes generalize first-class cases by supporting composition of arbitrary case expressions, not just the addition of one clause. In the record analogy, this is like the difference between extension of a record by a single field and a generalized record append. Since their case extension is only adding case clauses to the end of an existing case, it cannot model inheritance or mixins: the former requires added methods to be higher-priority than existing ones and the latter appends of two sets of methods rather than adding a single method to a set. So, we believe our scapes offer the first high-level algebraic case composition operator which can directly support object inheritance for the variant encoding of objects.

Less related is the Pattern Calculus [13], which is a more fundamental algebra in that patterns themselves are first-class entities separate from the code to be executed in case of a match. How-

[1] A scape is the green, leafless stem of an onion that grows above the ground.

ever, variable bindings in this calculus are extremely complex; we therefore elect to make the case clause our level of abstraction.

Dependent pattern types A weak form of dependent type is necessary to address the need for heterogeneity in case branches, the most fundamental problem existing type systems have with typing the variant object encoding. A case branching on multiple variants needs to return a type based on the particular *value* of the variant that arrives at runtime, in particular based on which tag the variant has. Our *dependent pattern types* solve this problem. They are *weakly* dependent since they depend only on the tag and not more detailed information from the value; the advantage of their weakness is that type inference is still decidable.

Our approach is a generalization of the conditional constraints originating in [5] and elaborated in [18]. Conditional constraints partly capture this dependency but only locally and so lose the dependency information in the presence of side effects. Our dependent pattern types fully capture the dependency.

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

Additionally, since (untagged) functions can be placed in our onions, we can re-use the onion record structure to hold our scape clauses and avoid the need for two different extension operations as found in [6]. As we alluded above, *asymmetric* concatenation is the key to composing scapes and thus properly defining inheritance and overriding; onions thus support asymmetric concatenation as the default. Asymmetric record concatenation was initially proposed by Wand for modeling inheritance [27], but difficulties in obtaining principal types in unification-based type inference [26] caused a switch in research focus to symmetric notions of record concatenation, which unfortunately are not amenable to modeling the overriding of methods. In standard record-based encodings of inheritance [9], there is no first-class record extension operation; inheritance requires the superclass to be known statically. This is implicitly a consequence of the difficulty of typing asymmetric record extension. Dynamically-typed OO languages have explored first-class object extenders to good benefit; the use of typed asymmetric record concatenation can bring this flexibility to the typed world.

There has been work developing type systems that support more flexible notions of object extension [8, 20] and our presented object encoding builds on this work. They define a phase transition where objects are initially open for extension but closed for messaging and can transition to *sealed* objects open for messaging but permanently closed for extension. This approach allows for more flexible programming patterns than standard class/inheritance object construction. Here we show for the first time how this approach may be generalized to support the (functional) extension of already-sealed objects, bringing fundamentally new dynamic-style object flexibility to a statically-typed domain.

Synergy Each of these features has some precedent in the literature and in each we have made improvements. The main claim of this paper is that the *combination* of this set of features yields a highly expressive language in which many paradigms of object-oriented programming can be encoded in an extremely lightweight fashion. Along with supporting the variant-based encoding of objects, it naturally supports functional object extension, mixins, as well as new programming patterns not even imaginable when wearing the straitjacket of a fixed object model.

Subtype constraint types TinyBang uses a subtype constraint inference based type system [5, 12, 18]; in particular, it is most closely related to [18]. Compared to [18], our system does not need row types or conditional constraints to typecheck record concatenation, incorporates type-indexed records and first-class cases, and contains a more general notion of conditional type and a more general model of parametric polymorphism. Our approach to parametric polymorphism is based on flow analysis [23, 28] and improves on previous work for expressive but efficient contour sharing.

While use of subtype constraint inference is a major thrust of our broader research agenda, the concepts of this paper should not fundamentally depend on use of a constraint inference type system. We believe it should be possible to build a more standard declarative polymorphic subtype system for TinyBang syntax; it would need to include a form of dependent pattern type for scapes, as well as an asymmetric record concatenation operation for onions.

Direct objects vs encoded objects plus translucent sugar While this paper discusses how to *encode* object features in a non-OO language [9] as opposed to directly giving types and runtime semantics for objects [1], the fundamental ideas here could also be used to give an object language a direct type system and runtime semantics. We focus on encodings rather than direct semantics because, given that our encodings are extremely lightweight, we believe it is possible to expose them directly to the programmer. In other words, sugar can be directly added to the language for objects, classes, inheritance, etc, but the sugar is so simple it is *translucent*: it is possible for the programmer to look under the hood of the encoding if needed and do some cool things not possible if the hood was permanently locked.

Record vs variant encoding We believe the variant encoding has been under-appreciated outside of the actor-like language community and a main focus of this paper is showing it has potential as a viable alternative in a typed context. The typed variant encoding was previously explored by our lab in [24], but that paper only showed how primitive objects could be encoded and was not focusing on a practical encoding of all object features such as self awareness and object extension.

Outline In the next section, we give a high-level overview of how TinyBang can encode objects. Section 3 contains the formal type system for MicroBang, a subset of TinyBang containing the key features but trimmed down for technical readability. We conclude in Section 4.

2. Overview

In this Section we informally present our object encoding and make the case that it is a good one.

2.1 Variant-Based Object Encodings

We begin with a very brief review of the standard record encoding, and then we outline the main problem with the natural dual variant encoding.

Record encodings Typed object encodings today generally encode objects into some form of record: an object consists of a record with the labels representing the method names and the label values are the functions which represent the method bodies, as in:

```
1 let o = { double = (fun x -> x + x);
2           inc = (fun x -> x + 1)       } in ...
```

Various different encoding methods are used for self-awareness of such records and for representing fields [9]; we momentarily

$$
\begin{array}{rcll}
e & ::= & x \mid \text{lbl}\, e \mid e \,\texttt{\&}\, e \mid e \,\texttt{\&-}\, \pi \mid e \,\texttt{\&.}\, \pi \mid e\ e & \textit{TinyBang expressions} \\
 & & \mid e \,\text{op}\, e \mid \texttt{def}\ x = e\ \texttt{in}\ e \mid x = e\ \texttt{in}\ e & \\
 & & \mid \texttt{()} \mid \mathbb{Z} \mid c \mid \texttt{(\&)} \mid \chi\ \texttt{->}\ e & \textit{TinyBang literals} \\
q & ::= & \epsilon \mid \texttt{final} \mid \texttt{immut} & \\
\text{op} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{==} \mid \texttt{<=} \mid \texttt{>=} & \\
x & ::= & \textit{(alphanumeric identifiers)} & \\
\text{lbl} & ::= & \texttt{`[a-zA-Z0-9\_]}^{+} & \\
c & ::= & \texttt{'[\^{}\textbackslash']'} \mid \texttt{'\textbackslash\textbackslash'} \mid \texttt{'\textbackslash''} & \textit{character literals} \\
\chi & ::= & x : \chi^{\text{P}} \mid x \mid \chi^{\text{P}} & \textit{patterns} \\
\chi^{\text{P}} & ::= & \text{tprim} \mid \text{lbl}\, \chi \mid (\chi^{\text{P}}\, \texttt{\&} \ldots \texttt{\&}\, \chi^{\text{P}}) \mid \texttt{fun} \mid \texttt{any} & \\
\pi & ::= & \text{tprim} \mid \text{lbl} \mid \texttt{fun} & \textit{projectors} \\
\text{tprim} & ::= & \texttt{char} \mid \texttt{int} \mid \texttt{unit} &
\end{array}
$$

Figure 1: TinyBang Syntax

ignore these issues for simplicity. Invoking a method on a record-encoded object involves projecting the function from the record and invoking that function, `o.double 4`.

Variant encodings In a variant-based encoding of objects, the object is instead a single dispatch function. Messages are passed directly to the dispatch function as variants and the dispatch function cases on the message to choose the code to execute. The following example shows a trivial variant-encoded object and a corresponding message dispatch.

```
let obj = fun msg -> match msg with
    | Say_Hello -> print_string "Hello!"
    | Calc_Double x -> x + x
in obj (Calc_Double 2)
```

Unfortunately, the above variant-encoded object does not typecheck in most languages with a `match` or similar expression because the branches are required to have the same result type. In the above, the `Say_Hello` branch has type `unit` while the `Calculate_Double` branch has type `int`. This property of the `match` expression imposes the debilitating requirement that all methods on an object have the same return type. We solve this problem by inferring more expressive *dependent pattern types* for match expressions; we discuss these types in the following Section.

2.2 Language Features for a Typeable Encoding

Our goal is to create a concise, typeable encoding for variant-based objects. By concise, we mean that the desugared, encoded version of objects should be legible and intuitive to a programmer. TinyBang's feature set is specifically selected to make such an encoding viable. The syntax for TinyBang appears in Figure 1. This figure should be used as a reference; we will discuss the semantics of each production as necessary throughout this section.

In order to typecheck our encoding, we will use a subtype constraint-based type system. The encoding we present is not intrinsically tied to such an approach, but subtype constraints provide a number of advantages. Principal typing, for instance, is trivial. Also, unlike when row types are used, it is not necessary to explicitly annotate type transition sites such as upcasts [21]. Most importantly, subtype constraint systems are handily modified to accommodate the more unusual language features we describe below. We defer detailed discussion of our type system until section 3; this section's use of types remains informal for presentation clarity.

Scapes as methods We begin by considering the oversimplified case of an object with a single method and no fields. This object is not self-aware; self-awareness is discussed in Section 2.3. In the variant encoding, such an object can be represented by a function which matches on a single case. In the TinyBang grammar, note that *all* functions are written χ `->` e, with χ being the pattern to match against the function's argument. Combining pattern match with function definition is also possible in ML and Haskell, but we can go further: there is no need for any `match` syntax in TinyBang since `match` can be encoded as a pattern and its application. We call such pattern-matching functions *scapes*. For instance, consider the following object and its invocation:

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

The syntax `` `double 4 `` is a label constructor similar to a OCaml polymorphic variant; as in OCaml, the expression `` `double 4 `` has type `` `double int ``. The scape `` `double x -> x + x `` is a function which matches on any argument containing a `` `double `` label and binds its contents to the variable `x`. As a result, the above code evaluates to `8`. Note that the expression `` `double 4 `` represents a *first-class message*; the entire object invocation is represented with its arguments as a variant.

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

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

The above shows that traditional `match` expressions can be encoded by using the `&` operator to join a number of scapes: one scape for each case. Our scape conjunction generalizes the first-class cases of [6] to support general appending of arbitrary cases; the aforecited work only supports adding one clause to the end and so does not allow "override" of an existing clause or "mixing" of two arbitrary sets of clauses. The above work does includes a construct $\ominus$ for removing a case clause containing a certain pattern and we plan to add similar functionality to our system in the near future.

Dependent pattern types Critical to our typed encoding is the fact that the scape extension is assigned a dependent pattern type which remembers the association between input types and output types. The two scapes above have the approximate[2] types `` `double int `` $\rightarrow$ `int` and `` `isZero int `` $\rightarrow$ `boolean`, respectively. While we could assign a type (`` `double int ``) $\cup$ (`` `isZero int ``) $\rightarrow$ `int` $\cup$ `boolean`, this type is imprecise; we want to retain the fact that, if a `` `double int `` is passed to the scape, an `int` will always be returned.

The dependent pattern type assigned to the above scape contains this relationship and loses no information; it is (`` `double int `` $\rightarrow$ `int`) `&` (`` `isZero int `` $\rightarrow$ `boolean`). If the scape is applied in the context where the type of message is known, the appropriate result type is inferred; the above method invocation, for instance, always has type `int` and not type `int` $\cup$ `boolean`. Because of this dependent typing, `match` expressions encoded in TinyBang may be heterogeneous; that is, each case branch may have a different type in a meaningful way. When we present our type system in Section 3, we show how these dependent pattern types extend the expressivity of conditional constraint types in a dimension critical for typing objects.

[2] This section uses simplified types which improve readability by avoiding constraint sets. The actual types used in TinyBang are more precise and are described in Section 3.

Onions as records We now show how our general onion data conjoiner, `&`, can act like a record constructor. For example, here is how we encode objects with methods that take multiple arguments:

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

In the last line, the object is invoked with the argument `‘sum (‘x 3 & ‘y 2)`. As above, the `‘sum` label merely wraps another value. In this case, it is an onion between two labels; this is pretty much a two-label record. This record-like onion is passed to the pattern `‘x x & ‘y y`; here, we use `&` to denote pattern conjunction, which requires that the value must match both subpatterns to match the overall pattern. When the argument `‘sum (‘x 3 & ‘y 2)` is passed to the object, the formal parameters `x` and `y` are bound to the values `3` and `2`. Observe from this example how there is no hard distinction in TinyBang between records and variants: there is only one class of label and a 1-ary record is the same as a 1-ary variant.

2.3 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 address this, each scape representing an object method now includes an additional parameter component `‘self` which binds a `self` variable. For instance, we can encode a simple self-aware object as:

```
def obj = (‘double x -> x + x)
        & (‘quad x & ‘self self ->
             self ‘double (self ‘double x)) in
...
```

Messaging such an object requires the more cumbersome `obj (‘quad 4 & ‘self obj)` to send message `‘quad 4` to `obj`. While this encoding is acceptable for simple uses of objects, it requires the full type of the object to be known at the call site; in a heterogeneous collection of objects, for instance, the exposed self-types are artificially forced to be the same and legal messagings may fail. This is a classic problem and we follow previous work [8] to address this issue. In [8], 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. This work was refined in [20] to increase the granularity to a per-method basis: an object can be proper in one method and prototypical in another, but a method must still be sealed before the object can receive any messages of that type, and sealed code may never again be extended.

Unlike the aforecited work, our encoding permits objects to be *resealed*: sealed objects may be extended and then sealed again. The flexibility of TinyBang allows the sharp phase distinction between prototypes and proper objects to be relaxed. All object extension below will be performed on sealed objects. The resulting language is, to our knowledge, the first static type system with general, flexible support for typing extensible objects. Object sealing in TinyBang is defined directly as a function `seal`:

```
def fix = f -> (g -> x -> g g x)
               (h -> y -> f (h h) y) in
def seal = fix (seal -> obj ->
            obj & (msg ->
              obj (‘self (seal obj) & msg))) in
def sObj = seal obj in
...
```

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

Extending previously sealed objects We now show how we can improve on the previous object extension models to support extension of previously-sealed objects. In the self binding scape, the value of `self` is onioned onto the *left* of the message rather than the right; this choice is purposeful. Because of this, any value of `‘self` which exists in a message passed to a sealed object takes priority over the `‘self` provided by the self binding scape. While we do not anticipate that programmers will want to manually specify a `self` value, this overriding is what permits an object to be resealed. Consider the following continuation of the previous code:

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

When this code is executed, the variable `sixteen` will hold the value `16`. Even after the `‘quad 4` message has been sent to `sObj`, we may extend it; in this case, `obj2` redefines how `‘double` messages are handled. `sObj2` represents the sealed version of this new object. When the `‘quad 4` message is sent to `sObj2`, it is passed to `obj2` with a `‘self` component; that is, `sObj2 (‘quad 4)` has the same effect as `obj2 (‘self sObj2 & ‘quad 4)`. Because `obj2` does not change how `‘quad` messages are handled, this has the same effect as `sObj (‘self sObj2 & ‘quad 4)`. `sObj` is also a sealed object which adds a `‘self` component to the *left*; thus this has the same effect as `obj (‘self sObj & ‘self sObj2 & ‘quad 4)`. Because any pattern match will always match the rightmost `‘self`, the latter-sealed object is provided in the `‘self` added to the message passed to the `‘quad`-handling method and so any messages sent from that method will be dispatched to an object which includes the extensions in `sObj2`.

In summary, this encoding works because, while we “tie the knot” on `self` using `seal`, we leave open the possibility of future *overriding* of `self`; it is merely a record element and we support record field override via asymmetric concatenation. Note that we could easily define a final, non-extensible object to restrict extension for encapsulation purposes; this is achieved simply by adding `‘self` to the *right* of the argument rather than the left in `seal`.

One limit of this sealing approach is that sealed objects cannot be extended to the *left*; thus, the following will not typecheck:

```
...
def obj = (‘foo _ -> 1) in
def sObj = seal obj in
def obj2 = (‘bar _ -> 2) & sObj in
def sObj2 = seal obj2 in
sObj2 ‘bar ()
```

The last line in the above code produces a type error. This is because the `‘bar` message is captured by the self-binding scape of `sObj` and is then sent only to `obj`, where it will fail to match.

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

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

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

Figure 2: Some Simple Syntactic Sugar

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

The above bears some description. Label construction implicitly creates a mutable cell, so the `‘x` label is used to store the counter's current value. The bottom line invokes the `obj` onion with an increment message. The label `‘x 0` is not considered as part of this invocation because it is not a scape; thus, the scape is considered next (and is executed because its pattern matches the message).

In this body, `self` is passed to a scape matching `‘x x`. Because `seal` onions the target object onto the left of the self binding method, all of the labels from the unsealed object are still visible in the sealed object. Thus, `self` has the same `‘x` label as `obj` and the cell within `obj`'s `‘x` label is bound to the variable `x`. The code `x = x + 1 in x` is then executed; this increments the contents of the `‘x` label and returns the value `1`. It should be noted that, in TinyBang, variable expressions implicitly dereference cells; this is the reason that `x + 1` successfully typechecks.

It may seem unusual that `obj` is, at the top level, a heterogeneous "mash" of a record field (the `‘x`) and a function (the scape which handles `‘inc`). This is in fact perfectly sensible; onions are *type-indexed* [22], meaning that they use the types of the values themselves to identify data. A simple case of type indexing is `4 & ()` which denotes the "onion" between the values `4` and `()`; such an onion would have the type `int & unit`. Values are projected when matched by a pattern, but they can also be implicitly projected; for example, `(4 & ()) + 1` evaluates to `5`. Note that, unlike label-indexed records, a value is equivalent to the 1-ary indexed record containing it; `5` can be viewed as both the integer five as well as the onion containing only the integer five.

In the case of overlap, the rightmost value is projected; `(7 & 3) + 1` is just `4` since the rightmost integer in the onion has precedence. Scapes are a special case; instead of projecting the rightmost scape, all scapes are projected and application selects the rightmost scape which matches the argument. For instance, the application `(‘x 0 & (int -> 4)) 1` implicitly projects the scape from the onion. We believe this type-indexed view is useful because it leads to more concise code; it also makes it trivial to define operator overloading, as we will discuss in Section 2.5 below.

The above counter object code is quite concise considering that it defines a self-referential, mutable counter object using no syntactic sugar whatsoever in a core language with no explicit object syntax. But programmers would benefit from syntactic sugar to capture common abstractions. We define a number of sugarings in Figure 2 which we use in the examples throughout the remainder of this section. It should be observed that this sugar is still translucent in the sense we describe above: looking at desugared code is not prohibitive for programmers desiring more control.

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

2.4 Typeable OO Abstractions

The previous section demonstrates a typed, variant-based encoding for objects. In this section, we focus on a typed, variant-based encoding for common object-oriented abstractions such as mixins and inheritance. Traditionally, these abstractions are defined in a first-order sense; inheritance, for instance, can only be expressed if the type of the parent class is statically known. In contrast, TinyBang's variant-based encoding can type higher-order abstractions.

We show our encoding in terms of objects rather than classes for simplicity; applying these concepts to classes is straightforward. We also work at the object and not class layer to show how TinyBang can easily express (functional) object extension. For clarity, we use the above-defined sugar for projection.

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

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

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

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

```
def mixin = seal (
    (‘nearZero _ & ‘self self ->
       (self ‘l1 ()) <= 4)) in
mixin ‘nearZero ()
```

As we would expect, this code is not typeable because `mixin`, the value of `self`, does not have a scape which can handle the `‘l1` message.

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

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

Inheritance Inheritance can be encoded with similar ease. As above, we show inheritance using objects rather than classes for simplicity; applying these concepts to classes provides no significant technical challenge. We consider the case in which we define an extension of the above point object which includes a concept of brightness in a field `k`:

```
def point = (as above) in
def brightPoint = seal (
  def super = point in
    point & `k 255 &
    (`isZero _ & `self self ->
      self.k == 0 and
        super (`isZero () & `self self)))
in brightPoint `isZero ()
```

As with mixins, we extend the object by onioning new scapes and fields onto the right. But recall that the onioning operator `&` is *asymmetric*; the rightmost scape matching a message is invoked. Because this extended object has a scape which handles `` `isZero `` messages, that scape will be invoked (as opposed to the original `` `isZero ``-matching scape from `point`). Because we have bound the parent object using the `super` variable, we may use it to statically invoke `point`'s scape from within our new `` `isZero ``-handling scape. To ensure that any messages it receives are potentially handled by `brightPoint` methods, we explicitly pass to `super` our *own* `self`.

Classes and subclasses The above examples show mixins and inheritance over objects. Classes are encoded by taking the view that they are merely objects with construction methods for other objects. The class for the point object above can be encoded as below. This class also includes a counter which tracks the number of points which have been created.

```
def Point = seal (
    `created 0 &
    (`new (`x x & `y y) & `self self ->
      self.created = self.created + 1 in
      seal (
        `x x & `y y &
        (`l1 _ & `self self -> self.x + self.y) &
        (`isZero _ & `self self ->
            self.x == 0 and self.y == 0)))) in
def point = Point `new (`x 1 & `y 2) in
...
```

Subclasses of classes can be defined in direct parallel to how extension of objects was defined above.

As previously stated, a language in practice would benefit from sugar to standardize and beautify class definitions. Nonetheless, users of the class may wish to examine the desugared form of the class definitions they create; as they are first-class values, these classes may be passed as arguments to other routines, pattern-matched, and so on. The lightweight encoding we have specified in this section ensures that the desugared form of a class is legible.

2.5 Programming Patterns with Scapes and Onions

Scapes and onions as defined above are motivated by our typeable, variant-based encoding for objects. However, this expressiveness also permits us to code in patterns not strictly in line with traditional object-oriented concepts such as objects or inheritance. This section provides a few examples of the emergent expressiveness of TinyBang's onions and scapes.

Operator overloading The pattern-matching semantics of scapes also provide a natural definition of operator overloading; we merely view an operator as an onion of scapes matching against the arguments. Operator overloading generally refers to infix operators; here we avoid complicating the discussion with matters of parsing and consider overloading on functions only. We might originally define negation on the integers as

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

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

```
...
def neg = neg
        & (`True unit -> `False ())
        & (`False unit -> `True ()) in ...
```

Negation is now overloaded: `neg 4` evaluates to `-4`, and `` neg `True () `` evaluates to `` `False () `` due to how scape application matches patterns. Furthermore, these operator overloadings can be defined in a lexically scoped manner; thus, operators may be defined to have different meanings in specific contexts.

The biggest problem with reasoning about operator overloading in this fashion is that it admits the same *overriding* properties that we described for objects. TinyBang can easily solve this problem by including a symmetric concatenation operator `&!` which produces a type error if its arguments overlap.

Data sharing TinyBang has two additional operations over onions: onion projection (`&.`) and onion subtraction (`&-`). Both of these operations use a projector to keep (or discard) components of an onion which match the projector. Projectors, represented in Figure 1 as π, are a shallow form of pattern with no variable bindings. For instance, consider the following:

```
def a = `A 1 & `B 2 & `C 3 in
def b = a &- `A in
def c = a &. `A in
b.B = 4 in
...
```

In the above code, `b` lacks the `` `A `` component of `a` but is otherwise structurally the same. `c` contains *only* the `` `A `` component of `a` and nothing else. Because `b` is a structural copy of `a` (including copies of all of the references in its labels), the assignment of a value to `b`'s `` `B `` component also affects `a`; by the end of the above code, `a` is `` `A 1 & `B 4 & `C 3 ``.

The object-oriented analogue of these operations is being able to strip away or extract fields from an object. This can result in a form of data sharing. Consider the following code (in which we leave objects unsealed for brevity):

```
def obj1 = seal (
        `x 0 &
        (`take _ & `self self ->
        self.x = self.x + 1 in self.x)) in
def obj2 = seal (
        (obj1 &. `x) & `y 2 &
        (`sum _ & `self self ->
            self.x + self.y)) in
...
```

In the above, `obj1` is a counter object increments an internal counter and returns that counter's current state. `obj2` is simply an object which sums two numbers. However, the `` `x `` label in `obj2` is drawn from `obj1`; they are, in essence, the same variable. This means that `obj2`'s response to the `` `sum `` message will change each time `obj1` is sent a `` `take `` message.

Exceptions While the TinyBang grammar in Figure 1 does not include a mechanism for exceptions, they are easy to incorporate. Typical exception semantics can be obtained by (1) extending the expression grammar with a `throw` expression and (2) extending the pattern grammar with an exception form. This observation is made in [7] and is a natural consequence of the first-class cases defined in [6]. TinyBang improves upon this behavior. Just as the aforecited extensible cases can extend a case with a single clause but not generally concatenate cases, their exception handling mechanism only permits extension by one exception handler at a

time. TinyBang onions permit general concatenation whether the scapes contained within are using exception patterns or not. The following is an example of how exception handling can be written using this extension of TinyBang:

```
def f = z -> throw `Bar z in
def handler = (n -> n)
            & (exn `Foo _ -> 0)
            & (exn `Bar x -> x) in
handler (f 4)
```

The above code would evaluate to `4`. When `f 4` is evaluated, an exception with the payload `` `Bar 4 `` is raised. When the argument to an application evaluates to an exception, the applied onion's exception-matching scapes are used to locate a match for the exception payload. If no such match is found, the entire application expression evaluates to the exception (which will then propagate upward). The identity function in the above `handler` is required to ensure that the argument will be passed on if no exception is raised.

2.6 Record vs variant encoding comparison

Now that we have overviewed the joys of standing on your head, let us step back to compare the objects-as-records and objects-as-variants approaches.

First, note that TinyBang's scapes and onions can also be used to build a pure record-based encoding of objects that would include support for first-class messages. Unlike previous encodings of first-class messages [17, 18], TinyBang provides a view which uses subtyping and dependent pattern types and does not require conditional constraints, row types, or a higher-kinded system. Such a record-based encoding is the true dual of the variant-based encoding presented above; the variant-encoded message `` `msg arg ``, for example, dualizes to `obj -> obj.msg arg` in the record-encoded system.

It is also possible to use first-class labels to form a record-based encoding of objects; first-class messages are then easily encoded with first-class labels. Fluid object types [11] are a recent system which uses this approach to infer types in scripting languages such as JavaScript. (Note that we do intend to investigate in the future whether the addition of first-class labels to TinyBang will give us additional added flexibility; they should not be theoretically challenging to add.)

We focus on the variant-based encoding of messages because we believe they are a more simple and direct syntax; for example, the variant form `` `msg arg `` may be directly pattern-matched. The TinyBang variant encoding also cleanly puts fields and methods into different syntactic sorts (labels and scapes, respectively). By analogy with boolean logic, it is possible to use only $\wedge$ (or only $\vee$) since DeMorgan's Laws allow one to be defined in terms of the other; in practice, however, it is far more pleasant to live on both sides of the duality. We believe the most natural view is to put fields on the record ($\wedge$) side of the duality and methods on the variant ($\vee$) side. We were motivated to switch from a record-based encoding to the variant-record hybrid presented here due to how pleasantly simple the self-reference encoding of objects becomes with the simple `seal` function. A record-based encoding which declares `seal` as a function and which supports object extensibility as outlined in Section 2.5 is either very complex or impossible to define (we tried, not very hard, and failed).

Overall, the use of variants as messages is more suitable for our uses and, more generally, is an approach that is woefully neglected in the current literature.

3. Type System

In this section, we outline the type system for TinyBang in terms of a smaller language: MicroBang. MicroBang has a simpler syntax and lacks both state and complex patterns. We have developed the full TinyBang type rules and omit features here only for conciseness of presentation. The syntax of MicroBang appears in Figure 3.

$$
\begin{array}{rcll}
e & ::= & x \mid \mathrm{lbl}\, e \mid e \,\texttt{\&}\, e \mid e \,\texttt{\&-}\, \pi \mid e \,\texttt{\&.}\, \pi & \textit{MicroBang expressions} \\
 & & \mid e\ e \mid e \,\texttt{+}\, e \mid e \,\texttt{-}\, e & \\
 & & \mid \texttt{()} \mid \mathbb{Z} \mid \texttt{(\&)} \mid \chi \,\texttt{->}\, e & \textit{MicroBang literals} \\
x & ::= & \texttt{[a-zA-Z\_][a-zA-Z0-9\_]}^* & \\
\mathrm{lbl} & ::= & \texttt{`[a-zA-Z0-9\_]}^+ & \\
v & ::= & \texttt{()} \mid \mathbb{Z} \mid \mathrm{lbl}\, v \mid v \,\texttt{\&}\, v \mid \texttt{(\&)} \mid \chi \,\texttt{->}\, e & \\
\mathbb{Z} & ::= & \texttt{[0123456789]}^+ \mid \texttt{-[0123456789]}^+ & \\
\chi & ::= & x : \chi^{\mathrm{P}} & \textit{patterns} \\
\chi^{\mathrm{P}} & ::= & \mathrm{tprim} \mid \mathrm{lbl}\, x \mid \texttt{fun} \mid \texttt{any} & \\
\pi & ::= & \mathrm{tprim} \mid \mathrm{lbl} \mid \texttt{fun} & \textit{projectors}
\end{array}
$$

Figure 3: MicroBang Grammar

The TinyBang type system is designed to realize an intuition of types and subtyping held by programmers accustomed to dynamically-typed languages: a method call which can be understood using the philosophy of duck typing should simply work. For this reason, we base our system on expressive polymorphic set constraint type systems [4, 10, 12, 18, 28] adding several extensions for greater expressiveness.

Parametric polymorphism is important for expressiveness and we would like to avoid the arbitrary cutoffs of polymorphism that are found with `let`-polymorphism or local type inference. In order to obtain the maximal amount of polymorphism, every scape in MicroBang is inferred a polymorphic type; this approach is inspired by flow analyses [2, 23] and has previously been ported to a type constraint context [28]. In the ideal, each call site then produces a fresh instantiation of the type variables. Unfortunately, this ideal cannot be implemented since a single program can have an unbounded number of function call sequences and so produce infinitely many instantiations. A standard solution to dealing with this case in the program analysis literature is to simply chop off call sequences at some fixed point. While arbitrary cutoffs may work for program analyses, they work less well for type systems: they make the system hard for users to understand and potentially brittle to small refactorings.

We have developed an approach here starting from our previous work on this topic [14, 15, 28] to produce polymorphism on functions which is "nearly maximal" in that common programming patterns have expressive polymorphism, suffer no arbitrary cutoffs, and are not too inefficient. In this approach, non-recursive functions are maximally polymorphic. Recursive call cycles are polymorphic only on the first traversal; type variables are reused when a recursive cycle is closed. We discuss our polymorphism model in more detail in Sections 3.3 and 3.5.

3.1 Type Grammar

We now present the mechanics of the MicroBang type system. Typechecking consists of two phases: derivation, in which the expression is assigned a type variable and a set of constraints based on its structure; and closure, during which the constraint set is deductively closed over a logical system in a monotonically increasing fashion, and then checked for consistency. We begin our discussion by presenting the grammar of types and constraints for MicroBang, shown in Figure 4. In that figure, we use the notation $[R, \ldots, R]$ to denote a list of R. Throughout this paper, we use $\oplus$ to denote list concatenation.

Type derivation occurs over a fixed program. We assign each subexpression of that program a unique expression identifier; we use ι to range over all expression identifiers. Program points are named by an expression identifier ι and an index. Type variables are named by a program point and a *contour identifier* c. Contour

$$
\begin{array}{rcll}
\ell & ::= & \langle \iota, \mathbb{Z} \rangle & \\
\alpha & ::= & {}^{\ell}\alpha^{\mathcal{C}} & \\
\mathcal{C} & ::= & [\ell, \ldots, \ell] & \textit{contour identifiers} \\
\tau & ::= & \mathtt{tprim} \mid \mathtt{lbl}\,\alpha \mid \alpha \,\mathtt{\&}\, \alpha \mid \alpha \,\mathtt{\&-}\, \pi \mid \alpha \,\mathtt{\&.}\, \pi \mid \mathtt{(\&)} & \\
 & & \mid \forall \vec{\alpha}.\tau^{\chi} \to \alpha \setminus \mathrm{C} & \\
\vec{\tau} & ::= & [\tau, \ldots, \tau] & \\
\tau^{\chi} & ::= & \alpha \sim \tau^{\chi_{\mathrm{P}}} & \\
\tau^{\chi_{\mathrm{P}}} & ::= & \mathtt{tprim} \mid \mathtt{lbl}\,\alpha \mid \mathtt{fun} \mid \mathtt{any} & \\
\mathtt{tprim} & ::= & \mathtt{int} \mid \mathtt{unit} & \\
\mathrm{c} & ::= & \tau <: \alpha \mid \alpha <: \alpha \mid \alpha <: \alpha \to \alpha \mid \alpha \,\mathrm{op}\, \alpha <: \alpha \mid \lightning & \\
\mathrm{C} & ::= & \mathrm{c}^* & \\
\dot{\mathrm{C}} & ::= & \mathrm{C} \mid \text{❁} &
\end{array}
$$

Figure 4: MicroBang Type Grammar

identifiers are defined by lists of program points and are used for tracking polymorphism. We defer discussion of contours until Section 3.3. For now, a reader may assume that all contour identifiers are equal; indeed, this is the case until constraint closure.

The types τ in the type grammar represent concrete *lower* bounds in MicroBang's type system. They are, informally and respectively, primitives, labels of cells, the onion concatenation of two other types, the subtraction and projection of a term from an onion, the empty onion type, and polymorphic functions. Functions are associated with a set of constraints which are expanded whenever that function is applied.

MicroBang's constraint grammar is inspired by [25] in that the lower-bounding types τ are not used as upper bounds. Instead, upper bounds define uses of types; for instance, the constraint $\alpha_1 <: \alpha_2 \to \alpha_3$ describes the use of α_1 as a function where input is a subtype of α_2 and output is a supertype of α_3. This allows MicroBang to be precise about how types are used.

The grammar uses a symbol ❁ to represent a special "none" value which is used in the relations discussed in Section 3.4. We also include a special constraint $\lightning$ which indicates that a contradiction has occurred and typechecking should fail.

3.2 Type Derivation

Type derivation is described in terms of a four place relation $\Gamma \vdash e : \alpha \setminus \mathrm{C}$ which relates a type context, an expression, a type variable, and a set of constraints over that type variable.

A type context in derivation is a set Γ of mappings $x : \alpha$ from a variable name x onto a type variable α. We define a context Γ to be well formed if $\forall x : \alpha, x' : \alpha'.x = x' \implies \alpha = \alpha'$; that is, no two mappings exist which have different values but the same key. We assume that we will only operate over well-formed contexts.

Given the nature of contexts, the disjoint union $\Gamma_1 \uplus \Gamma_2$ of two well-formed contexts Γ_1 and Γ_2 is well-formed; however, the union $\Gamma_1 \cup \Gamma_2$ is not necessarily well-formed. We use the notation $\Gamma_1 \cup\!\!\!\!\cdot\, \Gamma_2$ to denote $\Gamma_2 \uplus \{x : \alpha | x : \alpha \in \Gamma_1, \forall \alpha'.x : \alpha' \notin \Gamma_2\}$.

We use the notation ${}^{n}\alpha$ to select type variables as follows:

$$ {}^{n}\alpha = {}^{\ell}\alpha^{[]} \text{ where } \ell = \langle \iota, n \rangle $$

Thus an expression with identity ι would have ${}^{2}\alpha = {}^{\langle \iota, 2 \rangle}\alpha^{[]}$.

We define a distinguished type variable, $\alpha_{\mathtt{unit}}$, to represent a unit type. We take some $\iota_{\mathtt{unit}}$ used in evaluation which is *not* associated with any expression in e and define $\alpha_{\mathtt{unit}} = {}^{\langle \iota_{\mathtt{unit}}, 0 \rangle}\alpha^{[]}$. In addition to the constraints below, derivation is always assumed to include the constraint $\mathtt{unit} <: \alpha_{\mathtt{unit}}$.

Type derivation uses a function TPATTYPE to type patterns. That function is defined as follows:

$$ \textsc{TPatType}(x : \chi^{\mathrm{P}}, \iota) = \langle \Gamma \cup\!\!\!\!\cdot\, \{x : {}^{3}\alpha\}, {}^{3}\alpha \sim \tau^{\chi_{\mathrm{P}}} \rangle $$

where $\textsc{TPatPriType}(\chi^{\mathrm{P}}, \iota) = \langle \Gamma, \tau^{\chi_{\mathrm{P}}} \rangle$ and

$$
\begin{array}{rcl}
\textsc{TPatPriType}(\ \mathtt{tprim}\ , \iota\ ) & = & \langle\ [] \qquad\ , \mathtt{tprim}\ \rangle \\
\textsc{TPatPriType}(\ \mathtt{lbl}\, x\ , \iota\ ) & = & \langle\ \{x : {}^{4}\alpha\}\ , \mathtt{lbl}\, {}^{4}\alpha\ \rangle \\
\textsc{TPatPriType}(\ \mathtt{fun}\ , \iota\ ) & = & \langle\ [] \qquad\ , \mathtt{fun}\ \rangle \\
\textsc{TPatPriType}(\ \mathtt{any}\ , \iota\ ) & = & \langle\ [] \qquad\ , \mathtt{any}\ \rangle
\end{array}
$$

Using this function, the type derivation rules appear in Figure 5. It should be noted that these rules are not subtyping judgments; as is the standard in the constraint subtyping literature [5], we need not present judgments for a subtyping relation. Instead, the subtyping relation is implicit in whether or not closure over the constraints inferred from Figure 5's derivation will produce a contradiction [10]. We defer discussion of constraint closure until Section 3.5 since it is defined in terms of a number of supporting relations.

We begin by considering the integer literal rule. In any context, an integer literal expression is assigned a type variable ${}^{1}\alpha$ into which the concrete lower bound `int` will flow. Similar rules apply for other literal forms. These are the means by which ground types enter the flow graph.

The onion derivation rule for an expression `() & 5` first performs derivations on the subexpressions `()` and `5`; it then creates a constraint which indicates that an onion of these two values flows into the type variable representing the entire expression. Using natural numbers to represent points in the program, such an expression might result in the overall type ${}^{1}\alpha^{[]} \setminus \{\mathtt{unit} <: {}^{2}\alpha^{[]}, \mathtt{int} <: {}^{3}\alpha^{[]}, {}^{2}\alpha^{[]} \,\mathtt{\&}\, {}^{3}\alpha^{[]} <: {}^{1}\alpha^{[]}\}$.

Scape types are particularly important in MicroBang; they provide both polymorphism and dependent pattern typing. Scape types are always polymorphic; the $\vec{\alpha}$ represents those type variables which are free in the scape's body. As in [25], we polyinstantiate functions during *closure* and not derivation; thus, the application rule merely adds an upper-bounding constraint.

INTEGER LITERAL

$$ \frac{}{\Gamma \vdash \mathtt{[0\text{-}9]^{+}} : {}^{1}\alpha \setminus \{\mathtt{int} <: {}^{1}\alpha\}} $$

UNIT LITERAL

$$ \frac{}{\Gamma \vdash \mathtt{()} : {}^{1}\alpha \setminus \{\mathtt{unit} <: {}^{1}\alpha\}} $$

EMPTY ONION LITERAL

$$ \frac{}{\Gamma \vdash \mathtt{(\&)} : {}^{1}\alpha \setminus \{\mathtt{(\&)} <: {}^{1}\alpha\}} $$

LABEL

$$ \frac{\Gamma \vdash e : \alpha_2 \setminus \mathrm{C}}{\Gamma \vdash \mathtt{lbl}\, e : {}^{1}\alpha \setminus \{\mathtt{lbl}\, \alpha_2 <: {}^{1}\alpha\} \cup \mathrm{C}} $$

ONION

$$ \frac{\Gamma \vdash e_1 : \alpha_1 \setminus \mathrm{C}_1 \qquad \Gamma \vdash e_2 : \alpha_2 \setminus \mathrm{C}_2}{\Gamma \vdash e_1 \,\mathtt{\&}\, e_2 : {}^{0}\alpha \setminus \{\alpha_1 \,\mathtt{\&}\, \alpha_2 <: {}^{0}\alpha\} \cup \mathrm{C}_1 \cup \mathrm{C}_2} $$

ONION SUBTRACTION

$$ \frac{\Gamma \vdash e : \alpha_2 \setminus \mathrm{C}}{\Gamma \vdash e \,\mathtt{\&-}\, \pi : {}^{1}\alpha \setminus \{\alpha_2 \,\mathtt{\&-}\, \pi <: {}^{1}\alpha\} \cup \mathrm{C}} $$

ONION PROJECTION

$$ \frac{\Gamma \vdash e : \alpha_2 \setminus \mathrm{C}}{\Gamma \vdash e \,\mathtt{\&.}\, \pi : {}^{1}\alpha \setminus \{\alpha_2 \,\mathtt{\&.}\, \pi <: {}^{1}\alpha\} \cup \mathrm{C}} $$

HYPOTHESIS

$$ \frac{x : \alpha_1 \in \Gamma}{\Gamma \vdash x : \alpha_1 \setminus []} $$

SCAPE

$$ \frac{\textsc{TPatType}(\chi, \iota) = \langle \Gamma', \tau^{\chi} \rangle \qquad \Gamma \cup\!\!\!\!\cdot\, \Gamma' \vdash e : \alpha_2 \setminus \mathrm{C}}{\Gamma \vdash \chi \,\mathtt{->}\, e : {}^{1}\alpha \setminus \{(\forall \vec{\alpha}.\tau^{\chi} \to \alpha_2 \setminus \mathrm{C}) <: {}^{1}\alpha\}} $$

APPLICATION

$$ \frac{\Gamma \vdash e_1 : \alpha_1 \setminus \mathrm{C}_1 \qquad \Gamma \vdash e_2 : \alpha_2 \setminus \mathrm{C}_2}{\Gamma \vdash e_1\ e_2 : {}^{2}\alpha' \setminus \{\alpha_1 <: {}^{1}\alpha' \to {}^{2}\alpha', \alpha_2 <: {}^{1}\alpha'\} \cup \mathrm{C}_1 \cup \mathrm{C}_2} $$

LAZY OPERATION

$$ \frac{\Gamma \vdash e_1 : \alpha_1 \setminus \mathrm{C}_1 \qquad \Gamma \vdash e_2 : \alpha_2 \setminus \mathrm{C}_2}{\Gamma \vdash e_1 \,\mathrm{op}\, e_2 : {}^{0}\alpha \setminus \{\alpha_1 \,\mathrm{op}\, \alpha_2 <: {}^{0}\alpha\} \cup \mathrm{C}_1 \cup \mathrm{C}_2} $$

Figure 5: MicroBang Type Derivation

$$
\begin{array}{rcll}
\mathcal{V} & ::= & & \textit{vertices} \\
\boldsymbol{\mathcal{V}} & ::= & \{\mathcal{V}, \ldots, \mathcal{V}\} & \\
\mathcal{E} & ::= & \langle \mathcal{V}, \mathcal{V}, \ell \rangle & \textit{edges} \\
\boldsymbol{\mathcal{E}} & ::= & \{\mathcal{E}, \ldots, \mathcal{E}\} & \\
\mathcal{T} & ::= & \langle \boldsymbol{\mathcal{V}}, \boldsymbol{\mathcal{E}} \rangle & \textit{contour trees}
\end{array}
$$

Figure 6: Contour Tree Grammar

The input for a scape is represented as a pattern type τ^{χ}; this is critical to dependent pattern typing. When an onion of scapes is applied, each scape type includes a set of constraints representing its behavior. Because the pattern type is available when application closure occurs, we can use it to select only the scape (and corresponding constraint set) which will actually be used. MicroBang's dependent pattern typing has the same expressiveness to conditional constraints [5, 12, 19]. But in TinyBang, dependent pattern typing also captures the constraints for state modifications; conditional constraints do not model this behavior. For instance, we statically know that the expression

```
(    ('A int & 'B x -> x = 2 in x)
  & ('A unit & 'B x -> x = () in x) )
('A 1 & 'B 0)
```

has the type `int` because the latter scape will never be invoked. Using dependent pattern types, the TinyBang type system can correctly infer this behavior; the constraints in the latter scape which allow `unit` to flow into the `'B` cell are never introduced into the global constraint set.

3.3 Contours and Contour Trees

We take a brief aside to discuss type contours. Each time application is processed in the constraint closure described below, the type variables in the body of the applied scape are instantiated with a new contour; in essence, the flow graph representing the function's body is duplicated. This is the means by which the MicroBang type system achieves polymorphism; these new variables are only used to represent the flow of a specific scape application. Decidability is achieved by reusing old contours when recursion is detected.

In order to determine when and which contours are reused, we make use of a *contour tree*. A contour tree is, more precisely, a rooted multi-tree with self loops. Edges in this tree are annotated with a single program label ℓ representing the site of the application which created them; vertices are unannotated. The grammar describing this system appears in Figure 6.

A contour tree $\mathcal{T} = \langle \boldsymbol{\mathcal{V}}, \boldsymbol{\mathcal{E}} \rangle$ is *well-formed* if (1) it is non-empty and (2) for all $\langle \mathcal{V}_1, \mathcal{V}_2, \ell_1 \rangle$ and $\langle \mathcal{V}_1, \mathcal{V}_3, \ell_2 \rangle$ in $\boldsymbol{\mathcal{E}}$, $\ell_1 \neq \ell_2$; that is, no vertex has more than one outgoing edge with a given ℓ. In the following discussion, we only describe well-formed contour trees.

Extension When new contours are created, they are represented in the contour tree by an *extension*. In Figure 7, we define a function CExtend over contour trees which ensures that a given tree contains a specified path. CExtend accepts as input a contour tree and a path in the form of a contour identifier; it produces a resulting tree containing that path.

Reduction When a contour tree contains a path which repeats a call site, the type system has detected recursion. In order to ensure termination, the contour tree is *reduced*; this creates equivalence classes of contours, effectively assigning only a single contour to each recursive call cycle. We define contour tree reduction as a relation $\circledcirc$; we write $\mathcal{T}_1 \circledcirc \mathcal{T}_2$ to indicate that $\mathcal{T}_1$ reduces to $\mathcal{T}_2$. This relation is defined in terms of a tree rewriting function CIdentify.

The function CIdentify shown in Figure 8 models vertex identification over contour trees; given a set of vertices and a tree, it produces the tree in which those vertices are merged. The identity

$$
\begin{array}{l}
\textsc{CExtend}(\mathcal{T}, \mathcal{C}) = \textsc{CExtHelp}(\mathcal{T}, \mathcal{C}, \mathcal{V}) \text{ where } \mathcal{V} \text{ is the root of } \mathcal{T} \\
\textsc{CExtHelp}(\langle \boldsymbol{\mathcal{V}}, \boldsymbol{\mathcal{E}} \rangle, \mathcal{C}, \mathcal{V}) = \\
\quad \begin{cases}
\mathcal{T} \text{ when } \mathcal{C} = [] \\
\textsc{CExtHelp}(\langle \boldsymbol{\mathcal{V}}, \boldsymbol{\mathcal{E}} \rangle, \mathcal{C}', \mathcal{V}') \text{ when } \mathcal{C} = [\ell] \oplus \mathcal{C}', \langle \mathcal{V}, \mathcal{V}', \ell \rangle \in \boldsymbol{\mathcal{E}} \\
\textsc{CExtHelp}(\langle \boldsymbol{\mathcal{V}} \cup \{\mathcal{V}'\}, \boldsymbol{\mathcal{E}} \cup \{\mathcal{V}, \mathcal{V}', \ell\} \rangle, \mathcal{C}', \mathcal{V}') \\
\quad \text{when } \mathcal{C} = [\ell] \oplus \mathcal{C}', \langle \mathcal{V}, \mathcal{V}', \ell \rangle \notin \boldsymbol{\mathcal{E}}
\end{cases}
\end{array}
$$

Figure 7: CExtend Definition

For a given $\boldsymbol{\mathcal{V}}_{\text{Merge}}$ and $\mathcal{T}_{\text{In}} = \langle \boldsymbol{\mathcal{V}}_{\text{In}}, \boldsymbol{\mathcal{E}}_{\text{In}} \rangle$, let

- $\mathcal{V}_{\text{Repl}} \notin \boldsymbol{\mathcal{V}}_{\text{In}}$
- $\textsc{CRename}(\mathcal{V}) = \begin{cases} \mathcal{V}_{\text{Repl}} & \text{when } \mathcal{V} \in \boldsymbol{\mathcal{V}}_{\text{Merge}} \\ \mathcal{V} & \text{otherwise} \end{cases}$
- $\boldsymbol{\mathcal{V}}_{\text{Out}} = \{\mathcal{V}_{\text{Repl}}\} \cup \boldsymbol{\mathcal{V}}_{\text{In}} - \boldsymbol{\mathcal{V}}_{\text{Merge}}$
- $\boldsymbol{\mathcal{E}}_{\text{Out}} = \{\langle \textsc{CRename}(\mathcal{V}_{\text{Src}}), \textsc{CRename}(\mathcal{V}_{\text{Dst}}), \ell \rangle | \langle \mathcal{V}_{\text{Src}}, \mathcal{V}_{\text{Dst}}, \ell \rangle \in \boldsymbol{\mathcal{E}}_{\text{In}}\}$

Then $\textsc{CIdentify}(\mathcal{T}_{\text{In}}, \boldsymbol{\mathcal{V}}_{\text{Merge}}) = \mathcal{T}_{\text{Out}} = \langle \boldsymbol{\mathcal{V}}_{\text{Out}}, \boldsymbol{\mathcal{E}}_{\text{Out}} \rangle$.

Figure 8: CIdentify Definition

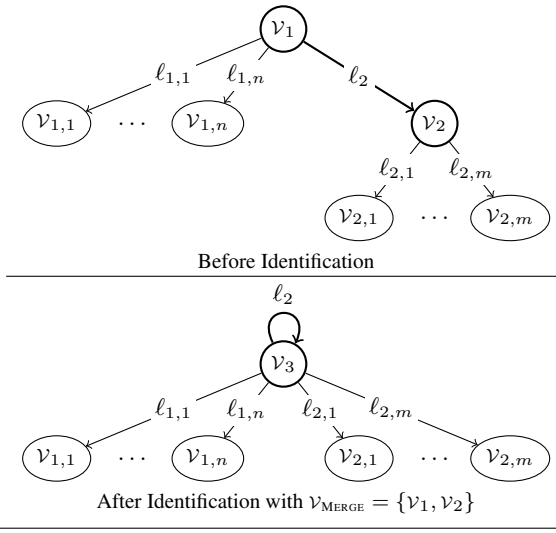

Figure 9: Vertex Identification Example

of the new vertex is the union of the old vertices' identities. This operation does not eliminate any edges between the merged vertices; any edges between the set of vertices being merged become self-loops on the new vertex in the resulting tree.

The CIdentify function will always output a tree when the input set $\boldsymbol{\mathcal{V}}_{\text{Merge}}$ forms a path. This is always the case in the tree reduction definition. The diagram in Figure 9 gives an example of a vertex identification between two nodes.

The reduction relation We define a relation $\circledcirc$ which describes reduction on contour trees; we write $\mathcal{T}_1 \circledcirc \mathcal{T}_2$ to indicate that $\mathcal{T}_1$ reduces to $\mathcal{T}_2$. This operation is defined as follows:

$\mathcal{T}_1 \circledcirc \mathcal{T}_2$ iff there exists some sequence of edges $\langle \mathcal{V}_0, \mathcal{V}_1, \ell_1 \rangle, \ldots, \langle \mathcal{V}_{n-1}, \mathcal{V}_n, \ell_n \rangle$ such that $n \geq 2$, $\ell_1 = \ell_n$, and $\textsc{CIdentify}(\mathcal{T}_1, \{\mathcal{V}_0, \ldots, \mathcal{V}_n\}) = \mathcal{T}_2$

We use the notation $\mathcal{T}_1 \circledcirc\!\!\!\!= \mathcal{T}_2$ to signify both of the following:

- Either $\mathcal{T}_1 = \mathcal{T}_2$ or there exists some $\mathcal{T}_3$ such that $\mathcal{T}_1 \circledcirc \mathcal{T}_3$ and $\mathcal{T}_3 \circledcirc \mathcal{T}_2$ (that is, $\mathcal{T}_1$ transitively reduces to $\mathcal{T}_2$) and
- There exists no $\mathcal{T}_4$ such that $\mathcal{T}_2 \circledcirc \mathcal{T}_4$

Contour equivalence When scape applications are expanded in closure, they are initially assumed to be non-recursive. When a given application is discovered to be part of a recursive cycle, a contour tree reduction occurs. For this reduction to affect constraint closure, we must define equivalence between contours. We then use this equivalence definition to define constraint equivalence.

Because contour identifiers are lists of program labels and because edges in the contour tree are annotated with program labels, a contour identifier describes a path (or vertex) in a contour tree. Because there may exist multiple edges from one node to another and because self-loops exist, multiple paths in a given contour tree may refer to the same vertex. We define an equivalence relation $\simeq^{\mathcal{T}}$ such that two contour identifiers are equivalent if, as paths from the root of a contour tree $\mathcal{T}$, they arrive at the same vertex. We write $\simeq$ when $\mathcal{T}$ is evident from context. This relation is defined as follows:

For a given $\mathcal{T} = \langle \boldsymbol{\mathcal{V}}, \boldsymbol{\mathcal{E}} \rangle$, $[\ell_1, \ldots, \ell_n] \simeq^{\mathcal{T}} [\ell'_1, \ldots, \ell'_m]$ if all of the following are true:

- $\{\langle \mathcal{V}_0, \mathcal{V}_1, \ell_1 \rangle, \ldots, \langle \mathcal{V}_{n-1}, \mathcal{V}_n, \ell_n \rangle\} \subseteq \boldsymbol{\mathcal{E}}$
- $\{\langle \mathcal{V}_0, \mathcal{V}'_1, \ell'_1 \rangle, \ldots, \langle \mathcal{V}'_{m-1}, \mathcal{V}'_m, \ell'_m \rangle\} \subseteq \boldsymbol{\mathcal{E}}$
- $\mathcal{V}_0$ is the root of $\mathcal{T}$ and $\mathcal{V}_n = \mathcal{V}'_m$

This equivalence definition allows us to define equivalence classes on contour identifiers; we use the notation $[\![\mathcal{C}]\!]^{\mathcal{T}}$ to denote the equivalence class containing $\mathcal{C}$ under the equivalence relation $\simeq^{\mathcal{T}}$. Again, we elide $\mathcal{T}$ when it is evident from context.

Constraint equivalence We also overload the above equivalence relation notation with a natural homomorphism over other grammatical constructs. For instance:

$$\begin{array}{rcll} \alpha_1 <: \alpha_2 & \simeq & \alpha_3 <: \alpha_4 & \text{iff } \alpha_1 \simeq \alpha_3 \text{ and } \alpha_2 \simeq \alpha_4 \\ \alpha_1 \,\&\, \alpha_2 & \simeq & \alpha_3 \,\&\, \alpha_4 & \text{iff } \alpha_1 \simeq \alpha_3 \text{ and } \alpha_2 \simeq \alpha_4 \\ \alpha_1 \sim \tau_1^{\chi_\mathrm{P}} & \simeq & \alpha_2 \sim \tau_2^{\chi_\mathrm{P}} & \text{iff } \alpha_1 \simeq \alpha_2 \text{ and } \tau_1^{\chi_\mathrm{P}} \simeq \tau_2^{\chi_\mathrm{P}} \\ {}^{\ell}\alpha^{\mathcal{C}} & \simeq & {}^{\ell}\alpha^{\mathcal{C}'} & \text{iff } \mathcal{C} \simeq \mathcal{C}' \\ & \vdots & & \end{array}$$

We likewise extend the equivalence class notation to cover other grammatical constructs. For example, if ${}^{\ell}\alpha^{\mathcal{C}_1} \simeq {}^{\ell}\alpha^{\mathcal{C}_2}$, then we would write $[\![\texttt{`A}\ {}^{\ell}\alpha^{\mathcal{C}_1}]\!]$ to denote at least the set $\{\texttt{`A}\ {}^{\ell}\alpha^{\mathcal{C}_1}, \texttt{`A}\ {}^{\ell}\alpha^{\mathcal{C}_2}\}$.

3.4 Closure Relations

The next step of typechecking is closure over the set of constraints obtained by type derivation. This closure is defined in terms of a number of relations, the definitions for which appear in this section. Each of these relations is implicitly indexed by a constraint set $\hat{\mathrm{C}}$ which is typically the current global constraint set. We sometimes write constraints in place of predicates; writing the constraint c in place of a predicate is syntactic sugar for $\mathrm{c} \in \hat{\mathrm{C}}$.

Concretization The first and simplest relation we define is concretization; see Figure 10. This relation determines those concrete types in a constraint set which may flow to a given type variable. While similar to a transitive subtype closure rule, concretization guarantees us that only concrete lower bounds are propagated.

Projection Next, we define type-based projection; this definition appears in Figure 11. This is a three place relation between a type, a projector, and another type. The projection relation determines

$$\begin{array}{rl} \tau \blacktriangleleft: \alpha_n \stackrel{\text{def}}{=} \exists \alpha_1, \ldots, \alpha_{n-1}. & (\tau <: \alpha_1) \quad \wedge (\alpha_1 \simeq \alpha_2) \\ & \wedge (\alpha_2 <: \alpha_3) \quad \wedge (\alpha_3 \simeq \alpha_4) \\ & \wedge \ldots \\ & \wedge (\alpha_{n-2} <: \alpha_{n-1}) \wedge (\alpha_{n-1} \simeq \alpha_n) \end{array}$$

Figure 10: TinyBang Concretization Relation

PRIMITIVE PROJECTION

$$\frac{}{\text{tprim} \xrightarrow{\text{tprim}_\triangleright} [\text{tprim}]}$$

LABEL PROJECTION

$$\frac{}{\text{lbl}\,\alpha \xrightarrow{\text{lbl}_\triangleright} [\text{lbl}\,\alpha]}$$

ONION PROJECTION

$$\frac{\tau_1 \blacktriangleleft: \alpha_1 \qquad \tau_2 \blacktriangleleft: \alpha_2 \qquad \tau_1 \xrightarrow{\pi_\triangleright} \vec{\tau}_1 \qquad \tau_2 \xrightarrow{\pi_\triangleright} \vec{\tau}_2}{\alpha_1 \,\&\, \alpha_2 \xrightarrow{\pi_\triangleright} \vec{\tau}_1 \oplus \vec{\tau}_2}$$

ONION SUB. PROJECTION

$$\frac{\pi \neq \pi' \qquad \tau \blacktriangleleft: \alpha \qquad \tau \xrightarrow{\pi_\triangleright} \vec{\tau}}{\alpha \,\&\text{-}\, \pi' \xrightarrow{\pi_\triangleright} \vec{\tau}}$$

ONION PROJ. PROJECTION

$$\frac{\pi = \pi' \qquad \tau \blacktriangleleft: \alpha \qquad \tau \xrightarrow{\pi_\triangleright} \vec{\tau}}{\alpha \,\&.\, \pi' \xrightarrow{\pi_\triangleright} \vec{\tau}}$$

SCAPE PROJECTION

$$\frac{}{\forall \vec{\alpha}.\tau^{\chi} \to \alpha \setminus \mathrm{C} \xrightarrow{\text{fun}_\triangleright} [\forall \vec{\alpha}.\tau^{\chi} \to \alpha \setminus \mathrm{C}]}$$

PRIM. FAIL

$$\frac{\tau \neq \text{tprim}}{\tau \xrightarrow{\text{tprim}_\triangleright} []}$$

LABEL FAIL

$$\frac{\tau \neq \text{lbl}\,\alpha}{\tau \xrightarrow{\text{lbl}_\triangleright} []}$$

ONION SUB. FAIL

$$\frac{\pi = \pi'}{\alpha \,\&\text{-}\, \pi' \xrightarrow{\pi_\triangleright} []}$$

ONION PROJ. FAIL

$$\frac{\pi \neq \pi'}{\alpha \,\&.\, \pi' \xrightarrow{\pi_\triangleright} []}$$

EMPTY FAIL

$$\frac{}{(\&) \xrightarrow{\pi_\triangleright} []}$$

SCAPE FAIL

$$\frac{\tau \neq \forall \vec{\alpha}.\tau^{\chi} \to \alpha \setminus \mathrm{C}}{\tau \xrightarrow{\text{fun}_\triangleright} []}$$

Figure 11: MicroBang Projection Rules

a priority-ordered list of ways the original type can be used as the form of type described by the type projector. For example, $\texttt{int}\,\&\,\texttt{char} \xrightarrow{\texttt{int}_\triangleright} [\texttt{int}]$ but $\texttt{unit} \xrightarrow{\texttt{fun}_\triangleright} []$. Projection never produces an onion; instead, onions are concretized and explored in a fashion which gives right precedence. This is necessary due to the fact that projection of scapes must produce every available scape so that they may be pattern-matched in order.

Compatibility We define a type/pattern compatibility relation in Figure 12. This is a three-place relation between a type, a pattern type form (τ^{χ} or τ^{χ_P}), and a set of constraints (or the ❁ symbol). If a set of constraints is related by compatibility with a type and a pattern type, then that type can be bound to the pattern type in question using the specified constraints. This relation is used to ensure that arguments arriving at a call site will flow correctly into the variables in the scape's pattern. If a given type and pattern type relate to ❁, then that type does not match the pattern in question. In the following, we use the notation $\dot{\mathrm{C}}_1 \cup \dot{\mathrm{C}}_2$ to refer to $\mathrm{C}_1 \cup \mathrm{C}_2$ (if $\dot{\mathrm{C}}_1 = \mathrm{C}_1$ and $\dot{\mathrm{C}}_2 = \mathrm{C}_2$) or to ❁ (if either $\dot{\mathrm{C}}_1$ or $\dot{\mathrm{C}}_2$ is ❁).

Application substitution The application substitution relation defines how function application builds new type variables for polyinstantiation. It is a three place relation written $\zeta(*, \vec{\alpha}, \mathcal{C})$ where $*$ is a type grammar construct such as τ or α. This relation is used to perform deep structural replacement of type variables in a given set of constraints. We thus define application substitution on each type grammar construct. Most of these definitions are simply the natural homomorphisms. For instance:

$$\begin{array}{rcl} \zeta(\tau <: \alpha_1, \vec{\alpha}, \mathcal{C}) & = & \zeta(\tau, \vec{\alpha}, \mathcal{C}) <: \zeta(\alpha_1, \vec{\alpha}, \mathcal{C}) \\ \zeta(\alpha_1 \,\&\, \alpha_2, \vec{\alpha}, \mathcal{C}) & = & \zeta(\alpha_1, \vec{\alpha}, \mathcal{C}) \,\&\, \zeta(\alpha_2, \vec{\alpha}, \mathcal{C}) \\ & \vdots & \end{array}$$

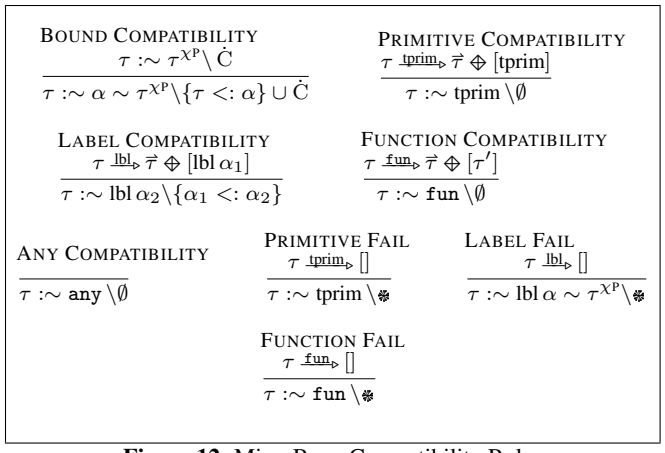

Figure 12: MicroBang Compatibility Rules

The only cases which are not natural homomorphisms are:

$$\zeta(\forall\vec{\alpha}_1.\alpha_1 \to \alpha_2 \setminus \mathrm{C}, \vec{\alpha}_2, \mathcal{C}) = \forall\vec{\alpha}_1.\alpha_1 \to \alpha_2 \setminus \zeta(\mathrm{C}, \vec{\alpha}_2 - \vec{\alpha}_1, \mathcal{C})$$

$$\zeta({}^{\ell}\alpha^{\mathcal{C}'}, \vec{\alpha}, \mathcal{C}) = \begin{cases} {}^{\ell}\alpha^{\mathcal{C}} & \text{when } {}^{\ell}\alpha^{\mathcal{C}'} \in \vec{\alpha} \\ {}^{\ell}\alpha^{\mathcal{C}'} & \text{otherwise} \end{cases}$$

Substitution on type variables will only apply a new contour ($\mathcal{C}$) if the old contour ($\mathcal{C}'$) is the initial contour ($[]$). This is the case in the use of ζ because it is only used on free type variables captured during derivation and such variables always use the initial contour.

Constraint set extension The constraint set extension function is used to ensure that, during constraint closure, constraints are only added to the global constraint set in each closure step if no equivalent constraint is already present. We denote constraint set extension by the symbol $\circ\!\!+$ and define it as follows:

$$\mathrm{C}_1 \circ\!\!+ \mathrm{C}_2 = \mathrm{C}_1 \cup \{\mathrm{c} \mid \mathrm{c} \in \mathrm{C}_2 \wedge \forall\, \mathrm{c}' \in \mathrm{C}_1\,.\,\mathrm{c} \notin [\mathrm{c}']\}$$

3.5 Constraint Closure

Using the above relations, we now specify the constraint closure step as the relation $\langle \mathrm{C}, \mathcal{T}\rangle \xrightarrow{\mathcal{C}} \langle \mathrm{C}, \mathcal{T}\rangle$ defined in Figure 13.

We begin illustration of the constraint closure process by analyzing the Integer Addition rule. Given an expression `4 + 3`, we would acquire from derivation the type $\alpha_3 \setminus \{\texttt{int} <: \alpha_1, \texttt{int} <: \alpha_2, \alpha_1 \texttt{+} \alpha_2 <: \alpha_3\}$. Concretizing α_1 yields the type `int` which we can project using the projector `int`; the same is true for α_2. We can thus conclude that the result of the addition is `int`, which we then constrain as a lower bound of α_3. The contour tree is largely unused by this rule; it merely provides a notion of constraint equivalence. Integer Equality and Integer Operation Failure work similarly.

The Application rule bears explanation; it is responsible for the usual complexity of application in a type system as well as for pattern matching and precedence. At any call site $\alpha_1' \to \alpha_2'$, a value α_0 may arrive; this is the onion of scapes being invoked. We concretize it as τ_1 and project from it all scapes it contains. We then concretize any argument which could arrive at that call site as τ_2. For that argument, we determine the rightmost scape which is compatible with the provided argument. The compatibility relation provides a set of constraints C' which, when added to the constraint set, cause the contents of the argument to flow into the variables in the pattern type of the scape. In this way, the input type flows into the body of the function. We also introduce a constraint, $\alpha_i <: \alpha_2'$, to connect the function body to the output of the call site.

To achieve polymorphism, we extend the contour tree using the call site to generate a new contour. To achieve termination, we then transitively reduce the resulting contour tree; this is relevant if the extension we just performed reveals a recursive call. We then use the application substitution function ζ to polyinstantiate the type variables in the set of constraints describing the input, output, and body flow, granting polymorphism. Finally, we subject this substituted set to the constraint set extension operator $\circ\!\!+$ to eliminate constraints for which equivalents already exist in $\hat{\mathrm{C}}$; this is key to ensuring termination and preventing unnecessary exponential complexity in closure.

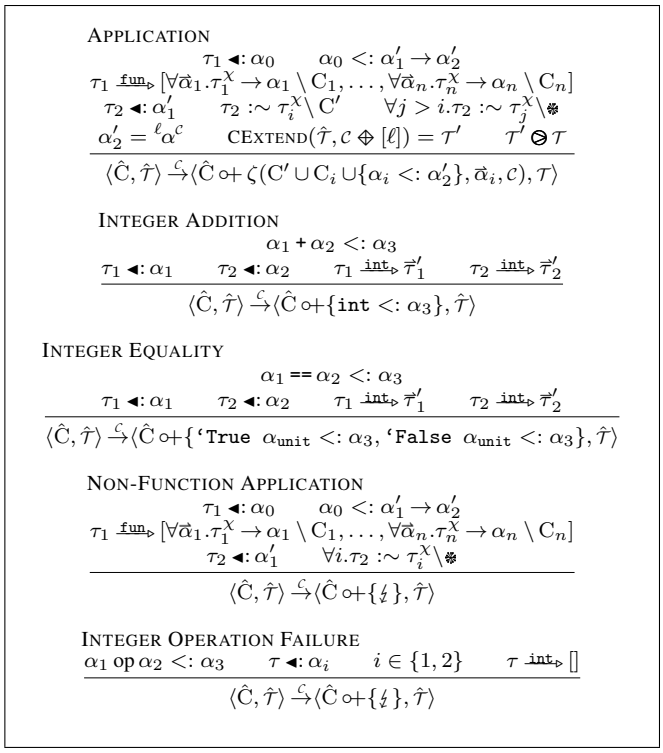

Figure 13: Closure Rules

Typechecking Given the above definition of a constraint closure step, we can provide the following definitions:

Definition 1. *Given a constraint set* C_1 *and a contour tree* $\mathcal{T}_1$*,*

1. *We write* $\langle \mathrm{C}_1, \mathcal{T}_1\rangle \xrightarrow{\mathcal{C}}{}^{*} \langle \mathrm{C}_2, \mathcal{T}_2\rangle$ *when either*
   (a) $\mathrm{C}_1 = \mathrm{C}_2$ *and* $\mathcal{T}_1 = \mathcal{T}_2$*, or*
   (b) $\langle \mathrm{C}_1, \mathcal{T}_1\rangle \xrightarrow{\mathcal{C}} \langle \mathrm{C}_3, \mathcal{T}_3\rangle$ *and* $\langle \mathrm{C}_3, \mathcal{T}_3\rangle \xrightarrow{\mathcal{C}}{}^{*} \langle \mathrm{C}_2, \mathcal{T}_2\rangle$
2. *We write* $\langle \mathrm{C}_1, \mathcal{T}_1\rangle \xrightarrow{\mathcal{C}}\!\!\!\!\!\!\not\quad$ *to indicate that there exists no* C_2 *and* $\mathcal{T}_2$ *such that* $\langle \mathrm{C}_1, \mathcal{T}_1\rangle \xrightarrow{\mathcal{C}} \langle \mathrm{C}_2, \mathcal{T}_2\rangle$*.*

We can now define typechecking as follows:

Definition 2. *Given a closed* e*,*

1. $\textsc{TypeInfer}(e) = \langle \mathrm{C}_n, \mathcal{T}_n\rangle$ *such that* $\emptyset \vdash e : \alpha \setminus \mathrm{C}_0$ *for some* α *and* C_0*,* $\mathcal{T}_0$ *is the initial contour tree containing one vertex and no edges,* $\langle \mathrm{C}_0, \mathcal{T}_0\rangle \xrightarrow{\mathcal{C}}{}^{*} \langle \mathrm{C}_n, \mathcal{T}_n\rangle$*, and* $\langle \mathrm{C}_n, \mathcal{T}_n\rangle \xrightarrow{\mathcal{C}}\!\!\!\!\!\!\not\quad$*.*
2. $\textsc{TypeCheck}(e)$ *holds if and only if* $\textsc{TypeInfer}(e) = \langle \mathrm{C}, \mathcal{T}\rangle$ *and* $\lightning \notin \mathrm{C}$*.*

Algorithmic properties The above typechecking algorithm is sound and decidable. We present these statements below:

Theorem 1 (Soundness)**.** *Let* $e_1 \xrightarrow{S} e_2$ *be a small-step evaluation relation for MicroBang; let* $e_1 \xrightarrow{S}{}^{*} e_2$ *be its transitive closure. Let* $\mathcal{T}_0$ *be the contour tree with one vertex and no edges. Then*

$\forall e.\ e \xrightarrow{S}{}^{*} \lightning$ *implies* $(\emptyset \vdash e : \alpha \setminus \mathrm{C}_0)$, $\langle \mathrm{C}_0, \mathcal{T}_0 \rangle \xrightarrow{C}{}^{*} \langle \mathrm{C}_n, \mathcal{T}_n \rangle$, *and* $\lightning \in \mathrm{C}_n$.

Proof of this Theorem is not trivial; we provide here a very high-level sketch of our proof technique. We elect to use a technique different than subject-reduction since it is difficult to re-build type derivations after a single step. Instead we create a so-called *constraint evaluator* which is a formal system lying at the rough midpoint between a type constraint system and a small-step operational semantics: it has actual integer values and does no approximation, but programs are expressed as a set of constraints that are structurally very similar to type constraints. Contours are never merged; the constraint evaluator may run forever. We can show our type system here is simulated by the constraint evaluator and the constraint evaluator is simulated by the small-step operational semantics, giving us the above Theorem by transitivity.

Theorem 2 (Decidability)**.** *The predicate* TYPECHECK *is decidable.*

This proof proceeds by demonstrating that there are finitely many closure steps which may occur on any derived constraint set and that each of these closure steps is computable. The latter is achieved by analysis of the operations used in closure; the prior is achieved by demonstrating that only finitely many constraints can be added during closure. In fact, this algorithm takes polynomial time under normal circumstances; worst-case exponential complexity is possible for pathological code similar to the exponential case of let-polymorphism.

4. Conclusions

In this paper, we have shown how it is possible to define a flexible variant encoding for typed objects in a novel core language, TinyBang. Particular advantages of programming in the encoding beyond the usual OO features include support for a more general notion of typed object extension than previous works, including support of fundamentally dynamic extension and extension of objects already actively being messaged; direct expressibility of first-class messages as simple labeled data; a trivial encoding of operator overloading; and enabling of other novel patterns that break the traditional object straitjacket.

Such a flexible object model is possible due to the flexibility of TinyBang, the underlying typed language into which we encode. TinyBang does not have one "blockbuster" extension but gets its significant expressiveness from a combination of a number of improvements to the state of the art in type system design: a more general notion of first-class case, simpler typing for asymmetric record append, a more expressive dependent typing of case/match, concise syntax via type-indexed records, and a highly flexible method for inference of parametric polymorphism.

The complete TinyBang type inference algorithm and a TinyBang interpreter have been implemented in Haskell; with this implementation, we have confirmed correctness of the type inference algorithm and operational semantics on code examples. All examples in this paper typecheck and run in the current implementation[3].

The larger picture In this paper we focus on fundamental questions of typing objects. It is not, however, a realistic language design proposal; TinyBang lacks syntactic sugar and other features that a real language needs. We are presently working on a realistic language design and implementation for BigBang [16]; TinyBang constitutes the proposed core for BigBang. We believe BigBang will be appealing because of the potential for great flexibility in coding, coming close to the spirit of dynamically-typed languages, but with the advantage of full static typechecking and more efficient running times compared to dynamically-typed languages.

[3] With the exception of the exceptions example in Section 2.5 – exceptions are currently being implemented.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

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

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

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

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

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

[7] M. Blume, U. A. Acar, and W. Chae. Exception handlers as extensible cases. In *APLAS '08*, pages 273–289. Springer-Verlag, 2008.

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

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

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

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

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

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

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

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

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

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

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

[19] F. Pottier. A 3-part type inference engine. In *ESOP'00*, pages 320–335. Springer Verlag, 2000.

[20] D. Rémy. From classes to objects via subtyping. In *ESOP*, pages 200–220. Springer-Verlag, 1998.

[21] D. Rémy and J. Vouillon. Objective ml: a simple object-oriented extension of ml. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 40–53. ACM, 1997.

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

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

[24] P. Shroff and S. F. Smith. Type inference for first-class messages with match-functions. In *FOOL 11*, 2004.

[25] S. F. Smith and T. Wang. Polyvariant flow analysis with constrained types. In *ESOP '00*. Springer Verlag, 2000.

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

[27] M. Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, July 1991.

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