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. Here we present TinyBang, a small typed language in which flexible object operations can be encoded. We illustrate this flexibility by solving an open problem in OO literature: we give an encoding where objects can be extended after being messaged without compromising the expressiveness of subtyping. TinyBang's subtype constraint system ensures that all types are completely inferred; there are no data declarations or type annotations. We formalize TinyBang and prove the type system is sound and decidable; all examples in the paper run in our most recent implementation.

1 Introduction

Modern scripting languages such as Python and JavaScript have become popular in part due to the flexibility of their object semantics. In addition to supporting traditional OO operations such as inheritance and polymorphic dispatch, scripting programmers can add or remove members from existing objects, arbitrarily concatenate objects, represent messages as first-class data, and perform transformations on objects at any point during their lifecycle.

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 λ`self`... so that different "self" values may be used in the event of extension. But exposing "self" in this way puts it in a contravariant position; as a result, subtyping on objects is invalid. The above systems create compromises; [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, concatenation, higher-order functions, and pattern matching. Classes, objects, inheritance, object extension, overloading, and switch/case can be fully and faithfully encoded with these primitives. TinyBang also has full type inference for ease and brevity of programming. It is not intended to be a programming language for humans; instead, it aims to serve as a conceptual core for such a language.

1.1 Key Features of TinyBang

TinyBang's type system is grounded in subtype constraint type theory [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 `13` is implicitly tagged with type `int`, and projecting `int` from this record would yield `13`. Since records are type-indexed, we do not need to distinguish records from non-records; `22`, for example, is a type-indexed record of one (integer) field. Variants are also just a special case of 1-ary records of labeled data, so `‘Some 3` expresses the ML `Some(3)`. Type-indexed records are thus a universal data type and lend themselves to flexible programming patterns in the same spirit as Lisp lists and Smalltalk objects.

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

Dependently typed first-class cases TinyBang's first-class functions are written "*pattern* `->` *expression*". In this way, first-class functions are also first-class *case clauses*. We permit the concatenation of these clauses via `&` to give multiple dispatch possibilities. TinyBang's first-class functions generalize the first-class cases of [BAC06].

Additionally, we define a novel notion of union elimination, a *slice*, which allows the type of bindings in a case arm to be refined based on which pattern was matched. Dependently typed first-class cases are critical for typing our object encodings, a topic we discuss in the next section.

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

2 Overview

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

2.1 Language Features for Flexible Objects

The TinyBang expression syntax used in this section appears in Figure 2.1.

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

Fig. 2.1. 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. For example, the informal `int` ∪ `bool` is in fact be expressed via constraints `int` $<: \alpha$ and `bool` $<: \alpha$. 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 pattern-matches on the message identifying that single method. We see in Figure 2.1 that all TinyBang functions are written ϕ `->` e, with ϕ being a *pattern* to match against the function's argument. Each function has only one pattern; we call these one-clause pattern-matching functions *simple functions*. For instance, consider the following object and its invocation:

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

The syntax `` `twice 4`` is a label constructor similar to an OCaml polymorphic variant or Haskell `newtype`; like these languages, the expression `` `twice 4`` has type `` `twice int``. The simple function `` `twice x -> x + x`` is a function which matches on any argument containing a `` `twice`` label and binds its contents to the variable `x`. Note that the expression `` `twice 4`` is a first-class message.

A simple function is only capable of matching one pattern; to express general pattern matching, functions are concatenated via the *onion* operation `&` to give *compound functions*. Given two function expressions `e1` and `e2`, the expression `(e1 & e2)` conjoins them to make a compound function. `(e1 & e2) arg` will apply the function which has a pattern matching `arg`; if both patterns match `arg`, the leftmost function (e.g. `e1`) is given priority. For example:

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

The above shows that traditional `match` expressions can be encoded using the `&` operator to join a number of simple functions: one for each case. Because these functions are values, TinyBang's function conjunction generalizes the first-class cases of [BAC06]; that work does not support "override" of existing clauses or heterogeneously typed case branches.

Dependent pattern types The above shows the encoding of a simple object with two methods, no fields, and no self-awareness, but this function conjunction approach presents some typing challenges. Consider the analogous OCaml match/-case expression:

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

This will not typecheck, since the same type must be returned for all branches.[1] We resolve this in TinyBang by giving the function a dependent pattern type (`‘twice 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`. When we present the formal type system below, we show how these dependent pattern types extend the expressiveness of conditional constraint types [AWL94,Pot00] in a dimension critical for typing objects.

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

Onions are records There is no record syntax in TinyBang; instead, it suffices to use concatenation (&) on labeled values. We informally call these records *onions* to signify these properties. Here is an example of how multi-argument methods can be defined:

```
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` 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 no self-awareness: they cannot invoke their own methods. Encoding a runtime model of self-awareness is simple; for instance, dynamic dispatch can be accomplished simply by transforming each method

[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.

invocation (e.g. `obj.m(arg)`) into a function call in which the object is passed to itself (e.g. `obj (‘self obj & ‘m arg)`). But while this model exhibits appropriate runtime behavior, it does not typecheck properly in the presence of subtyping. Consider the code in Figure 2.2 and the Java statement `A obj = new B();`. The encoding of `obj.foo()` here would fail to typecheck; the `B` implementation of `foo` expects `this` to have a `bar` method, but the weakened `obj` cannot guarantee this. Although this example uses Java type annotations to weaken the variable's type, similar weakening eventually occurs with any decidable type inference algorithm. We also observe that the same problem arises if, rather than self-passing at the call site, we encode the object to carry itself in a field.

```
class A {                         class B extends A {
  int foo() { return 4; }           int foo() { return this.bar(); }
}                                   int bar() { return 8; }
                                  }
```

Fig. 2.2. Self-Encoding Example Code

The simple approach shown above fails because, informally, the object does not know its own type. To successfully typecheck dynamic dispatch, we must keep an *internal* type for the object separate from the *external* type it has in any particular context; that is, in the above example, the object must remember that it is a `B`-typed object even when it is generalized e.g. in an `A`-typed variable. One simple approach to this is to capture an appropriate `self`-type in closure, similar to how functions can recurse via self-passing:

```
let obj0 = self -> (‘foo _ & ‘self self -> self self (‘bar _ & ‘self self)) &
                   (‘bar _ & ‘self self -> 8) in
let obj = obj0 obj0 in ...
```

The initial passing of `obj0` to itself bootstraps the self-passing mechanism; `obj` becomes a function with `obj0` captured in closure as `self`; thus, any message passed to `obj` uses the type of `obj0` at the time `obj` was created rather than relying on its type in context. While this approach is successful in creating a self-aware object, it interferes with extensibility: the type of `self` is fixed, preventing additional methods from being added or overridden.

To create an extensible encoding of dynamically dispatched objects, we build on the work of [BF98]. In that work, 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. Prototypes cannot be messaged because they do not yet have a notion of their own type. A prototype may be "sealed" to transform it into a proper object, capturing the object's type in a fashion similar to the above. A sealed object may not be extended for the same reason as the above: there exists no mechanism to modify or specialize the captured `self`-type.

The work we present here extends [BF98] with two notable refinements. First, that work presents a calculus in which objects are primitives; in TinyBang, however, objects and the sealing process itself are encoded using simple functions and conjunction. Second, TinyBang admits a limited context in which sealed objects may be extended and then *resealed*, thus relaxing the sharp phase distinction between prototypes and proper objects. All object extension below will be performed on sealed objects. Object sealing is accomplished by a TinyBang function `seal`:

```
let fixpoint = f -> (w -> w w) (t -> a -> f (t t) a) in
let seal = fixpoint (seal -> obj ->
                (msg -> obj (msg & ‘self (seal obj))) & obj) in
let obj = (‘twice x -> x + x) &
          (‘quad x & ‘self self -> self (‘twice x) + self (‘twice x))
let sObj = seal obj in
let twenty = sObj ‘quad 5 in          // returns 20
```

Here, `fixpoint` is simply the Y-combinator. The `seal` function operates by adding a message handler which captures *every* message sent to `obj`. (We still add `& obj` to this message handler to preserve the non-function parts of the object.) This message handler captures the type of `obj` in the closure of a function; thus, later invocations of this message handler will continue to use the type at seal-time even if the types of variables containing the object are weakened. The message handler adds a `‘self` component containing the sealed object to the right of the message and then passes it to the original object. We require `fixpoint` to ensure that this self-reference is also sealed. So, every message sent to `sObj` will be sent on to `obj` with `‘self sObj` added to the right. Note that the `‘twice` method does not require a `‘self` pattern component: due to structural subtyping, any message with a `‘twice` label (whether it also includes a `‘self` or not) will do.

The key to preserving extensibility with this approach is the fact that, while the `seal` function has captured the type of `obj` in closure, the `self` value is still sent to the original object as an argument. We now show how this permits extension of sealed objects in certain contexts.

Extending previously sealed objects In the definition of `seal` above, the catch-all message handler adds a `‘self` label to the *right* of the message; thus, if any `‘self` label already existed in the message, it would be given priority over the one added by `seal`. We can take advantage of this behavior in order to extend a sealed object and then *reseal* it, refining its `self`-type. Consider the following continuation of the previous code:

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

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

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

It should be noted that this resealing approach is limited to contexts in which no type information has yet been lost regarding the sealed object. Using the example from Figure 2.2, typechecking would likely fail if one were to seal a `B`, weaken its type to an `A`, extend it, and then reseal the result: the new `self`-type would be derived from `A`, not `B`, and so would still lack knowledge of the `bar` method. The runtime behavior is always correct, and typechecking would also succeed if the extension provided its own override of the `bar` method for the newly-sealed object to use.

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`.

The above counter object code is quite concise: it defines a self-referential, mutable counter object using no syntactic sugar whatsoever in a core language with no explicit object syntax. But as we said before, we do not expect programmers to write directly in TinyBang under normal circumstances. Here are a few syntactic sugarings used in subsequent examples. (A "real" language built on these ideas would include sugarings for each of the features we are about to mention as well.)

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

2.3 Flexible Object Operations

We now cover how TinyBang supports a wealth 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 optional arguments that take on a default value if missing. For instance, consider:

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

Object `dflt` overrides `obj`'s `` `add `` to make `1` the default value for `` `x ``. Because the `` `x 1 `` is onioned onto the right of `a`, it will have no effect if an `` `x `` is explicitly provided in the message.

Overloading The pattern-matching semantics of functions also provide a simple mechanism whereby multi-functions can be defined to overload their behavior. 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 multi-functions assign new meaning to an existing symbol, we redefine `neg` to include all of the behavior of the old `neg` as well as new cases for `` `True `` and `` `False ``:

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

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

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

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

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

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

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

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

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

3 Formalization

Here we give formal semantics to TinyBang. For clarity, features which are not unique to our semantics – integers, state, etc. – are omitted. We first translate TinyBang programs to A-normal form; 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. The full technical report [MPRS14b] shows how the omitted features are handled.

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 $\Vert$ denotes list concatenation. For sets, we use similar notation: $\overset{n}{\overbrace{g}}$ abbreviates $\{g_1, \ldots, g_n\}$ for some arbitrary ordering of the set.

3.1 A-Translation

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

$$
\begin{array}{rll@{\qquad}rll}
e & ::= \vec{s} & \textit{expressions} & \phi & ::= \overrightarrow{x = \mathring{v}} & \textit{patterns} \\
s & ::= x = v \mid x = x \mid x = x\ x & \textit{clauses} & & & \\
E & ::= \overrightarrow{x = v} & \textit{environment} & B & ::= \overrightarrow{x = x} & \textit{bindings} \\
v & ::= \mathbb{Z} \mid \texttt{()} \mid l\ x \mid x\ \texttt{\&}\ x \mid \phi \texttt{ -> } e & \textit{values} & \mathring{v} & ::= \texttt{int} \mid \texttt{()} \mid l\ x \mid x\ \texttt{\&}\ x & \textit{pattern vals} \\
l & ::= \texttt{`}\textit{(alphanumeric)} & \textit{labels} & x & ::= \textit{(alphanumeric)} & \textit{variables}
\end{array}
$$

Fig. 3.1. 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 α-renamed such that they do. The constructions E and B are not directly used in the A-translation; they define the environment and bindings in the semantics.

We define the A-translation function $\wr e \int_x$ in Figure 3.2. Here, e is a nested TinyBang expression; the result is the A-normalized form $\vec{s}$ in which the final declared variable is x. We overload this notation to patterns as well. We use y and z to range over fresh variables unique to that invocation of $\wr - \int_-$; different recursive invocations use different fresh variables y and z.

$$
\begin{array}{rcl@{\qquad}rcl}
\multicolumn{3}{c}{\textbf{Expressions}} & \multicolumn{3}{c}{\textbf{Patterns}} \\
\wr \texttt{()} \int_x & = & [x = \texttt{()}] & \wr \texttt{()} \int_x & = & [x = \texttt{()}] \\
\wr l\ e \int_x & = & \wr e \int_y \Vert [x = l\ y] & \wr l\ \phi \int_x & = & \wr \phi \int_y \Vert [x = l\ y] \\
\wr e_1 \texttt{ \& } e_2 \int_x & = & \wr e_1 \int_y \Vert \wr e_2 \int_z \Vert [x = y \texttt{ \& } z] & \wr \phi_1 \texttt{ \& } \phi_2 \int_x & = & \wr \phi_1 \int_y \Vert \wr \phi_2 \int_z \Vert [x = y \texttt{ \& } z] \\
\wr \phi \texttt{ -> } e \int_x & = & [x = \wr \phi \int_y \texttt{ -> } \wr e \int_z] & \wr x_2 \int_{x_1} & = & [x_2 = \texttt{()}] \\
\wr e_1\ e_2 \int_x & = & \wr e_1 \int_y \Vert \wr e_2 \int_z \Vert [x = y\ z] & & & \\
\wr \texttt{let}\ x_1 = e_1\ \texttt{in}\ e_2 \int_{x_2} & = & \wr e_1 \int_{x_1} \Vert \wr e_2 \int_{x_2} & & & \\
\wr x_2 \int_{x_1} & = & [x_1 = x_2] & & &
\end{array}
$$

Fig. 3.2. TinyBang A-Translation

Notice that A-translation of patterns is in perfect parallel with the expressions in the above; for example, the expression `‘A x -> x` translates to $[$`y`$_1$ `=` `(`$[$`x = ()`, `y`$_3$ `= ‘A x`$]$ `->` $[$`y`$_2$ `= x`$]$`)`$]$. Using the same A-translation for patterns and expressions greatly aids the formal development, but it takes some practice to read these A-translated patterns. Variables matched against empty onion (`x = ()` here) are unconstrained and represent bindings that can be used in the body. Variables matched against other pattern clauses (such as `y`$_3$) are not bindings; clause `y`$_3$ `= ‘A x` constrains the argument to match a `‘A`-labeled value. The last binding in the pattern, here `y`$_3$ `= ‘A x`, is taken to match the argument when the function is applied; this is in analogy to how the variable in the last clause of an expression is the final value. Variable binding takes place on `()` (wildcard) definitions: every pattern clause `x = ()` binds `x` in the function body. 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 noted.

3.2 Operational Semantics

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

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

We write $x \;{}^{\odot}_{E}\!\preceq^{B}_{\phi}\; x'$ to indicate that the value x is compatible (if $\odot = \bullet$) or incompatible (if $\odot = \bigcirc$) with the pattern x'. E represents the environment in which to interpret the value x while ϕ represents the environment in which to interpret the pattern x'. B dictates how, upon a successful match, the values from E will be bound to the pattern variables in ϕ. Compatibility is the least relation satisfying the rules in Figure 3.3.

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

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

Conjunction Pattern
$$\frac{x'_0 = x'_1 \,\&\, x'_2 \in \phi \quad x_0 \;{}^{\odot_1}_{E}\!\preceq^{B_1}_{\phi}\; x'_1 \quad x_0 \;{}^{\odot_2}_{E}\!\preceq^{B_2}_{\phi}\; x'_2}{x_0 \;{}^{\min(\odot_1,\odot_2)}_{E}\!\preceq^{B_1 \,\|\, B_2}_{\phi}\; x'_0}$$

Onion Value Left
$$\frac{x_0 = x_1 \,\&\, x_2 \in E \quad x_1 \;{}^{\bullet}_{E}\!\preceq^{B}_{\phi}\; x'_0}{x_0 \;{}^{\bullet}_{E}\!\preceq^{B}_{\phi}\; x'_0}$$

Onion Value Right
$$\frac{x_0 = x_1 \,\&\, x_2 \in E \quad x'_0 = x'_1 \,\&\, x'_2 \notin \phi \quad x_1 \;{}^{\bigcirc}_{E}\!\preceq^{B'}_{\phi}\; x'_0 \quad x_2 \;{}^{\odot}_{E}\!\preceq^{B}_{\phi}\; x'_0}{x_0 \;{}^{\odot}_{E}\!\preceq^{B}_{\phi}\; x'_0}$$

Label Mismatch
$$\frac{x_0 = l\ x_1 \in E \quad x'_0 = \mathring{v} \in \phi \quad \mathring{v} = l'\ x_2 \text{ only if } l \neq l' \quad \mathring{v} \text{ not of the form } x' \,\&\, x'' \text{ or } \texttt{()}}{x_0 \;{}^{\bigcirc}_{E}\!\preceq^{[]}_{\phi}\; x'_0}$$

Fig. 3.3. Pattern compatibility rules

The compatibility relation is key to TinyBang's semantics and bears some explanation. As mentioned above, every clause $x = \texttt{()}$ appearing in the pattern binds the variable x; the Empty Onion rule ensures this by adding a binding clause to B. The Label rule simply recurses when the value is a label and the pattern matches that label; the Label Mismatch rule (which is the base case for failure) applies when the pattern does not match that label. Conjunction is relatively self-evident; min is used here as a logical "and" over the two recursive premises. The onion rules reflect TinyBang's asymmetric concatenation semantics. Given a value $x_1 \,\&\, x_2$, it is possible that both x_1 and x_2 match the pattern. If so, we must ensure that we take the bindings from the compatibility of x_1. The Onion Value Left rule applies when the left side matches. The Onion Value Right rule only applies only if the left side *doesn't* match; that is, a proof of compatibility may recursively depend on a proof of *incompatibility.* This is the reason that our relation is defined to be constructive for both success and failure: it is necessary to show that the relation is inductively well-founded.

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

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

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 that individual pattern clauses *pattern* `->` *body* are simple functions $\phi \texttt{->} e$ and a series simple functions onioned together expresses a multi-clause pattern match. So, we define an application matching relation $x_0\ x_1\ {}^{\odot}{\rightsquigarrow}_E\ e$ to determine if an onion of pattern clauses x_0 can be applied to argument x_1. This relation is constructive on failure in the same fashion as compatibility.We define matching as the least relation satisfying the rules in Figure 3.4. The helper function RV extracts the return variable from a value, defined as follows: $\mathrm{RV}(e \,\|\, [x = \ldots]) = x$ and $\mathrm{RV}(e \,\|\, [x = \mathring{v}]) = x$.

Function
$$\frac{x_0 = (\phi \texttt{->} e) \in E \qquad x_1\ {}^{\odot}_{E}{\preceq}^{B}_{\phi}\ \mathrm{RV}(\phi)}{x_0\ x_1\ {}^{\odot}{\rightsquigarrow}_E\ B \,\|\, e}$$

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

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

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

Fig. 3.4. 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 ϕ, and if so insert the resulting bindings B at the top of the function body e. The Onion Left/Right rules are the inductive cases; notice that the Onion Right rule can only match successfully if the (higher priority) left side has failed to match. The Non-Function rule is the base case for application of a non-function, which fails but in a way which permits dispatch to continue through the onion.

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

We must freshen variables as they are introduced to the expression to preserve the invariant that the ANF TinyBang expression uniquely defines each variable;

to do so, we take $\boldsymbol{\alpha}(e)$ to be an α-renaming function which freshens all variables in e which are not free. We then define the small step relation as the least relation satisfying the rules given by Figure 3.5.

$$\textsc{Variable Lookup}\quad \frac{x_1 = v \in E}{E \,\|\, [x_2 = x_1] \,\|\, e \longrightarrow^1 E \,\|\, [x_2 = v] \,\|\, e} \qquad \textsc{Application}\quad \frac{x_0\ x_1 \overset{\bullet}{\leadsto}_E e' \quad \boldsymbol{\alpha}(e') = e''}{E \,\|\, [x_2 = x_0\ x_1] \,\|\, e \longrightarrow^1 E \,\|\, e'' \,\|\, [x_2 = \mathrm{RV}(e'')] \,\|\, e}$$

Fig. 3.5. The operational semantics small step relation

The application rule simply inlines the freshened function body e'' in the event there was a match. 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 expressive [AWL94] and suitable for complex pattern matching [Pot00] and object-orientation [WS01]. We begin by aligning expressions and types (an operation made easy by our choice of ANF expression syntax); we then define type system relations which parallel those from the operational semantics, including a deductive constraint closure which parallels the small step relation itself. Our proof of soundness proceeds by showing a simulation property that programs stay aligned with their types as they execute, and stuck programs correspond to inconsistent constraint sets.

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

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

Fig. 3.6. The TinyBang type grammar

We formalize this initial alignment step as a function $[\![e]\!]_{\mathrm{E}}$ which produces a constrained type $\alpha\backslash C$; see Figure 3.7. Initial alignment over a given e picks a single fresh type variable for each program variable in e; for the variable x_0, we denote this fresh type variable as $\mathring{\alpha}_0$.

Slicing When defining type compatibility in TinyBang, a subtle problem arises which did not appear in the evaluation system. In the evaluation system, each variable is guaranteed to have a single assignment; thus, a premise of value

$$\llbracket \overset{n}{\overrightarrow{s}} \rrbracket_{\mathrm{E}} = \alpha_n \backslash \overset{n}{\sqcup} c \text{ where } \forall i \in \{1..n\}.\llbracket s_i \rrbracket_{\mathrm{S}} = \alpha_i \backslash c_i$$

$$\llbracket \overset{n}{\overrightarrow{x \texttt{=} \mathring{v}}} \rrbracket_{\mathrm{P}} = \alpha_n \backslash \overset{n}{\sqcup} c \text{ where } \forall i \in \{1..n\}.\llbracket x_i \texttt{=} \mathring{v}_i \rrbracket_{\mathring{\mathrm{S}}} = \alpha_i \backslash c_i$$

$$\begin{array}{rclcrcl}
\llbracket x_0 \texttt{=} \mathbb{Z} \rrbracket_{\mathrm{S}} &=& \overset{\star}{\alpha}_0 \backslash\ \texttt{int} <: \overset{\star}{\alpha}_0 & \quad & \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}} \rightarrow \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{array}$$

Fig. 3.7. Initial alignment

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

We solve this problem by defining a *slicing* relation to eliminate unions before checking compatibility. This ensures that our union eliminations are consistent (because they are performed before, not during, compatibility checking) and additionally provides us with a refined form of the argument to use in typechecking the function body. Slicing eliminates the above soundness concern because the argument is first separated into the distinct $\texttt{`A}\,\alpha$ and $\texttt{`B}\,\alpha$ slices and compatibility is checked for each of them separately. Note that union elimination must be complete up to the depth of the pattern; otherwise, union alignment problems will begin where the elimination stopped.

We write $\alpha\backslash V \ll \alpha'\backslash C$ to indicate that $\alpha\backslash V$ is a slice of the constrained type $\alpha'\backslash C$; this relation is defined as follows:

Definition 1. *$\alpha\backslash V \ll \alpha'\backslash C$ is the least relation defined by the rules in Figure 3.8. Slices $\alpha\backslash V$ of $\alpha'\backslash C$ additionally must always be*

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

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

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

$$\textsc{Onion}\quad \frac{\alpha_1 \mathbin{\&} \alpha_2 <: \alpha_0 \in V \quad \alpha_1' \mathbin{\&} \alpha_2' <: \alpha_0' \in C \quad \alpha_1\backslash V \ll \alpha_1'\backslash C \quad \alpha_2\backslash V \ll \alpha_2'\backslash C}{\alpha_0\backslash V \ll \alpha_0'\backslash C}$$

Fig. 3.8. The slice relation for union elimination

One nuance of slicing is that the Leaf rule can allow slicing to stop at an arbitrary point - in practice the data is sliced as deep as the pattern requires it, but it is possible to slice too shallowly by this relation, in which case a *partial* match is all that is obtained, a topic discussed below.

Compatibility Using the above slicing relation, we can now define the type compatibility relation. Because a slice is a union-free representation of a type up to some depth, we can define compatibility in much the same way as we did in the evaluation system. As mentioned above, however, slicing may stop at an arbitrary point.

To handle partial slices, we define the type compatibility relation to filter out slices which are too shallow. In addition to being compatible (●) or incompatible (○), type compatibility may show that a slice and a pattern are *partially* compatible (◐), meaning that the slice lines up with the pattern correctly but is insufficiently deep. We use the metavariable ⊚ to range over this extension to the ⊙ grammar. As in value compatibility, we view these symbols as ordered: ○ < ◐ < ●. We define type system compatibility as the least relation satisfying the rules appearing in Figure 3.9.

Figure 3.10 shows an example of slicing and compatibility on a recursive type: Peano integers. Here, α is a union type between a successor and a zero. We consider matching Peano integers against two patterns (written here in nested form): `‘Z ()` and `‘S ‘S ()`. (Recall that `()` in patterns means "match anything.") The topmost slice matches the first pattern directly. The middle slice is partial after a single `‘S`: it fails to match the `‘Z ()` pattern (we elide that arrow for visual clarity) and *partially* matches the second pattern. The middle slice is insufficiently deep, indicating neither success nor failure. The bottommost slice matches the second pattern completely; although it is also a partial slice, it is deep enough that we can assert it would match that pattern regardless of how it might be further expanded. In general, there is always a point at which we can stop slicing: patterns are of fixed, finite depth, so every slice fixes as either compatible or incompatible after a certain depth.

Other than the additional concern of partial slices, type compatibility is much like value compatibility; so, see Section 3.2 for more explanation of compatibility.

Partial

$$\frac{\nexists\tau.\tau <: \alpha_0 \in V}{\alpha_0 \;{}^{\LEFTcircle}_{V}{\preceq}^{\emptyset}_{V'}\; \alpha_0'}$$

Empty Onion

$$\frac{\tau <: \alpha_0 \in V \quad \texttt{()} <: \alpha_0' \in V' \quad F = \{\alpha_0 <: \alpha_0'\}}{\alpha_0 \;{}^{\CIRCLE}_{V}{\preceq}^{F}_{V'}\; \alpha_0'}$$

Label

$$\frac{l\ \alpha_1 <: \alpha_0 \in V \quad l\ \alpha_1' <: \alpha_0' \in V' \quad \alpha_1 \;{}^{\circledcirc}_{V}{\preceq}^{F}_{V'}\; \alpha_1'}{\alpha_0 \;{}^{\circledcirc}_{V}{\preceq}^{F}_{V'}\; \alpha_0'}$$

Conjunction Pattern

$$\frac{\alpha_1' \mathbin{\&} \alpha_2' <: \alpha_0' \in V' \quad \alpha_0 \;{}^{\circledcirc_1}_{V}{\preceq}^{F_1}_{V'}\; \alpha_1' \quad \alpha_0 \;{}^{\circledcirc_2}_{V}{\preceq}^{F_2}_{V'}\; \alpha_2'}{\alpha_0 \;{}^{\min(\circledcirc_1,\circledcirc_2)}_{V}{\preceq}^{F_1 \cup F_2}_{V'}\; \alpha_0'}$$

Onion Value Left

$$\frac{\alpha_1 \mathbin{\&} \alpha_2 <: \alpha_0 \in V \quad \alpha_1 \;{}^{\circledcirc}_{V}{\preceq}^{F}_{V'}\; \alpha_0' \quad \circledcirc \neq \bigcirc}{\alpha_0 \;{}^{\circledcirc}_{V}{\preceq}^{F}_{V'}\; \alpha_0'}$$

Onion Value Right

$$\frac{\alpha_1 \mathbin{\&} \alpha_2 <: \alpha_0 \in V \quad \alpha_1' \mathbin{\&} \alpha_2' <: \alpha_0' \notin V' \quad \alpha_1 \;{}^{\bigcirc}_{V}{\preceq}^{F'}_{V'}\; \alpha_0' \quad \alpha_2 \;{}^{\circledcirc}_{V}{\preceq}^{F}_{V'}\; \alpha_0'}{\alpha_0 \;{}^{\circledcirc}_{V}{\preceq}^{F}_{V'}\; \alpha_0'}$$

Label Mismatch

$$\frac{\begin{array}{c} l\ \alpha_1 <: \alpha_0 \in V \\ \tau <: \alpha_0' \in V' \quad \tau = l'\ \alpha_2 \text{ only if } l \neq l' \quad \tau \text{ not of the form } \alpha' \mathbin{\&} \alpha'' \text{ or } \texttt{()} \end{array}}{\alpha_0 \;{}^{\bigcirc}_{V}{\preceq}^{\emptyset}_{V'}\; \alpha_0'}$$

Fig. 3.9. Type compatibility: does a type match a pattern?

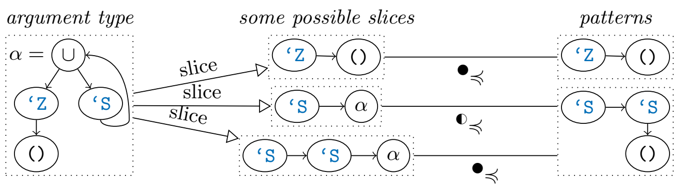

Fig. 3.10. Type compatibility example

Matching Type matching directly parallels expression matching. As in the evaluation system, type matching propagates the result of compatibility and uses the $\circledcirc$ place to enforce left precedence of dispatch. Matching $\alpha_0\ \alpha_1 \;{}^{\circledcirc}_{V_0}{\rightsquigarrow}_{V_1}\; \alpha_2 \backslash C'$ is defined as the least relation satisfying the clauses in Figure 3.11.

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

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

$$\textsc{Non-Function}\quad \frac{(\alpha'\backslash V' \to \alpha\backslash C) <: \alpha_0 \notin V_0 \quad \alpha_2 \,\&\, \alpha_3 <: \alpha_0 \notin V_0}{\alpha_0\ \alpha_1\ {}_{V_0}\!\overset{\circ}{\rightsquigarrow}_{V_1}\ \alpha\backslash C}$$

$$\textsc{Onion Left}\quad \frac{\alpha_2 \,\&\, \alpha_3 <: \alpha_0 \in V_0 \quad \alpha_2\ \alpha_1\ {}_{V_0}\!\overset{\circledcirc}{\rightsquigarrow}_{V_1}\ \alpha\backslash C \quad \circledcirc \neq \circ}{\alpha_0\ \alpha_1\ {}_{V_0}\!\overset{\circledcirc}{\rightsquigarrow}_{V_1}\ \alpha\backslash C}$$

$$\textsc{Onion Right}\quad \frac{\alpha_2 \,\&\, \alpha_3 <: \alpha_0 \in V_0 \quad \alpha_2\ \alpha_1\ {}_{V_0}\!\overset{\circ}{\rightsquigarrow}_{V_1}\ \alpha'\backslash C' \quad \alpha_3\ \alpha_1\ {}_{V_0}\!\overset{\circledcirc}{\rightsquigarrow}_{V_1}\ \alpha\backslash C}{\alpha_0\ \alpha_1\ {}_{V_0}\!\overset{\circledcirc}{\rightsquigarrow}_{V_1}\ \alpha\backslash C}$$

Fig. 3.11. Type application matching

The second function, Υ, unifies type variables by producing type variable equivalence relations. In particular, each of the above relations – slicing, compatibility, and matching – is actually defined to take an equivalence relation as an implicit parameter. In their definitions, we consider type variables in sets up to their equivalences by this relation; for instance, we read $\tau <: \alpha \notin V$ as $\nexists \alpha'.\, \tau <: \alpha' \in V \wedge \alpha \cong \alpha'$. For simplicity, readers may consider the monomorphic system given by $\Phi_{\text{Mono}}(C, \alpha) = C$ and where the equivalence relation given by $\Upsilon_{\text{Mono}}(-)$ is always equality; in this case, the definitions above can be read as they are presented. We discuss our concrete choice of polymorphism model for TinyBang in a Technical Report [MPRS14b].

We write $C \Longrightarrow^1 C'$ to indicate a single step of constraint closure. This relation is defined as the least such that the rules in Figure 3.12 are satisfied. This relation defines the equivalence to be used by the relations in its premises: at each constraint closure step $C \Longrightarrow^1 C'$, we take the type variable equivalence relation to be $\Upsilon(C)$. We write $C_0 \Longrightarrow^* C_n$ to indicate $C_0 \Longrightarrow^1 \ldots \Longrightarrow^1 C_n$.

$$\textsc{Transitivity}\quad \frac{\{\tau <: \alpha_1, \alpha_1 <: \alpha_2\} \subseteq C}{C \Longrightarrow^1 C \cup \{\tau <: \alpha_2\}} \qquad \textsc{Application}\quad \frac{\begin{array}{c}\alpha_0\ \alpha_1 <: \alpha_2 \in C \quad \alpha'_0\backslash V_0 \ll \alpha_0\backslash C \\ \alpha'_1\backslash V_1 \ll \alpha_1\backslash C \quad \alpha'_0\ \alpha'_1\ {}_{V_0}\!\overset{\bullet}{\rightsquigarrow}_{V_1}\ \alpha'\backslash C' \quad \alpha''\backslash C'' = \Phi(\alpha'\backslash C', \alpha_2)\end{array}}{C \Longrightarrow^1 C \cup V_1 \cup C'' \cup \{\alpha'' <: \alpha_2\}}$$

Fig. 3.12. Type constraint closure single-step relation

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

Informally, inconsistency captures the cases in which an expression can get stuck.

Given the above, we 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.*

Formal Properties We now state formal assertions regarding our type system. For reasons of space, we give proofs for each of these statements in [MPRS14b]. We begin with soundness which we prove by simulation: the A-translation of the program and the careful alignment between each of the relations in the system makes simulation a natural choice. We may state soundness as follows:

Theorem 1 (Soundness). *If* $e \longrightarrow^* e'$ *for stuck* e' *then* e *doesn't typecheck.*

This result is proven in Appendix A of [MPRS14b].

We must also show typechecking to be decidable. Our strategy is to demonstrate that the constraint closure of a finite constraint set C forms a subset inclusion lattice and that it is sufficient (and computable) to check the consistency of the top of that lattice. Because constraint closure is parametric in the polymorphism model, we must impose some requirements on the model. First, it must be *finitely freshening*: Φ must introduce finitely many variables into any produced constraint set given an initial constraint set. Second, to ensure convergence, polymorphism must be *equivalence monotone*: a constraint superset must induce monotonically more equivalences in Υ. See [MPRS14b] for the definition of a flexible polymorphism model with these properties.

Although the above restricts the polymorphism model to introduce finitely many variables to constraint closure, we have not bounded the number of variables introduced by slicing. The type system presented in this paper is oversimplified for legibility and does not bound the number of slice variables introduced; it is therefore undecidable. The solution to this problem is to introduce an occurrence check in slicing to prevent a single slice from making the same decision for the same variable at the same pattern multiple times. This approach is similar to determining whether the intersection of two regular trees is empty. While the type system is very slightly weakened by this modiciation, the examples in this paper are unaffected and the resulting system is intuitive and decidable.

Theorem 2 (Decidability). *Typechecking with the above modification and a finitely freshening, equivalence monotone polymorphism model is decidable.*

4 Related Work

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 [RS02,BBV11].

Typed multimethods [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. First-class cases [BAC06] allow composition of case branches much like TinyBang. In [BAC06], however, general case concatenation requires case branches to be written in CPS and requires a phase distinction between constructing a case and matching with it. TinyBang has a form of dependent type which allows different case branches to return different types; this generalizes the expressiveness of conditional constraints [AWL94,Pot00] and is related to support for typed first-class messages *a la* [Nis98,Pot00] – first-class messages are just labeled data in our encoding.

TinyBang shares several features and goals with CDuce [CNX$^+$14]: both aim to be flexible languages built around constraint subtyping. CDuce supports dependently-typed case results as we do, but it does not slice on the pattern side and so does not bind refined types. CDuce lacks a typed record append operation and so the object encodings of this paper are not possible there. CDuce takes a local type inference approach; this has the advantage of being modular but the disadvantage of not being complete, requiring type annotations in some cases, and the decidability of their inference algorithm remains open.

TinyBang's onions are unusual in that they are a form of record supporting typed asymmetric concatenation. The bulk of work on typing record extension addresses symmetric concatenation only [Rémy94]. 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 types.

There have been many attempts to bring static typing to existing scripting languages; two of the more recent systems include [CRJ12,FAFH09]. The majority of such systems require some explicit type annotations, and invariably are incomplete since an uncomputable problem is being solved: the languages contain too many fundamentally dynamic operations (e.g. mutable object extension). Once a type system loses information on a dynamic operation, it is difficult to recover. One primary tenant of this project's design philosophy is to build a language from the beginning with flexible but fully static typing; this way, there will never be a need to attempt such recovery.

5 Conclusions

We presented TinyBang, a core language with a static type inference system that types such flexible operations without onerous false type errors or the need for manual programmer annotation. We believe TinyBang solves a longstanding open problem: it infers types for object-oriented programs without compromising the expressiveness of object subtyping or of object extension. This is possible due to a combination of novel features in TinyBang: asymmetric concatenation, first-class dependently-typed cases, slicing for type refinement, and a flexible non-let-based polymorphism model.

TinyBang is proved type sound; the proofs are found in the supplementary appendices. We have implemented the type inference algorithm and interpreter for TinyBang to provide a cross-check on the soundness of our ideas; the implementation can be downloaded from [MPRS14a]. Type inference is decidable but not provably polynomial for the same reason that let-polymorphism is not:

artificial programs exist which will exhibit exponential runtimes. However, all of the examples in Section 2 typecheck in our implementation without exponential blowup, and we have designed the polymorphism model with practical performance in mind. We do not expect programmers to write in TinyBang. Instead, programmers would write in BigBang, a language we are developing which includes syntax for objects, classes, and so on, and which desugars to TinyBang.

TinyBang infers extremely precise types, especially in conjunction with the context-sensitive polymorphism model described in [MPRS14b]. While powerful, these types are by nature difficult to read and defy modularization. Addressing the problem of readability is part of our broader research agenda.

References

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*, pages 462–497. Springer Verlag, 1998.

CNX$^+$14. Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. Polymorphic functions with set-theoretic types. part 1: Syntax, semantics, and evaluation. In *POPL*, 2014.

CRJ12. Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. Nested refinements: A logic for duck typing. In *POPL*, 2012.

FAFH09. Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *SAC*, 2009.

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

MPRS14a. Pottayil Harisanker Menon, Zachary Palmer, Alexander Rozenshteyn, and Scott Smith. Tinybang implementation, Mar 2014. `http://pl.cs.jhu.edu/big-bang/tiny-bang_2014-03-01.tgz`.

MPRS14b. Pottayil Harisanker Menon, Zachary Palmer, Alexander Rozenshteyn, and Scott Smith. Types for flexible objects. Technical report, The Johns Hopkins University Programming Languages Laboratory, Mar 2014. `http://pl.cs.jhu.edu/big-bang/types-for-flexible-objects_2014-03-25.pdf`.

Nis98. Susumu Nishimura. Static typing for dynamic messages. In *POPL*, 1998.

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

Rémy94. Didier Rémy. Type inference for records in a natural extension of ML. In *Theoretical Aspects Of Object-Oriented Programming.* MIT Press, 1994.

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

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

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