Building a Typed Scripting Language

by

Zachary Palmer

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

May, 2015

Abstract

Since the 1990s, scripting languages (e.g. Python, Ruby, JavaScript, and many others) have gained widespread popularity. Features such as ad-hoc data manipulation, dynamic structural typing, and terse syntax permit rapid engineering and improve developer productivity. Unfortunately, programs written in scripting languages execute slower and are less scalable than those written in traditional languages (such as C or Java) due to the challenge of statically analyzing scripting languages' semantics. Although various research projects have made progress on this front, corner cases in the semantics of existing scripting languages continue to defy static analysis and software engineers must generally still choose between program performance and programmer performance when selecting a language.

We address that dichotomy in this dissertation by designing a scripting language with the intent of statically analyzing it. We select a set of core primitives in which common language features such as object-orientation and case analysis can be encoded and give a sound and decidable type inference system for it. Our type theory is based on subtype constraint systems but is also closely related to abstract interpretation; we use this connection to guide development of the type system and to employ a novel type soundness proof strategy based on simulation.

At the heart of our approach is a type indexed record we call the *onion* which supports asymmetric concatenation and dispatch; we use onions to formally encode a variety of features, including records, operator overloading, objects, and mixins. An optimistic call-site polymorphism model defined herein captures the ad-hoc, case-analysis-based reasoning often used in scripting languages. Although the language in this dissertation uses a particular set of core primitives, the strategy we use to design it is general: we demonstrate a simple, formulaic process for adding features such as integers and state.

Acknowledgements

First and foremost I thank my advisor, Dr. Scott Smith, who is one of the most patient and nurturing people I have ever met. The opportunity to study under him – a gift by which I had no means earned – is among the greatest experiences of my life. I could never have hoped for such a fine educator.

The other contributors to this work, direct and indirect, are many; I am but one author of this work and the beneficiary of their assistance and generosity. I give my thanks to the most influential here:

Rebekah Palmer: my closest friend, wife, and resident mathematician, whose company and understanding have made me whole and happy once again;

Alex Rozenstheyn and Hari Menon: my friends and peers, whose keen intellect and feedback were invaluable in my education;

Barbara and Staymon Palmer: my grandparents, who raised me to a sound mind and body and were there for me in my worst;

Grace Palmer: my sister, childhood defender, and lifetime deus ex machina;

Carolyn Jeffers: a dear friend from another time, who cultivated the best of me and has not been forgotten.

To the above and to all the others who have brought me to this point: thank you. I am deeply grateful.

Contents

List of Figures

List of Notations

List of Definitions

List of Theorems

Chapter 1

Introduction

Scripting languages such as Python, Ruby, and JavaScript have seen widespread adoption since the 1990s in large part because they improve developer productivity. Scripting languages provide a high level of abstraction and very terse syntax which allow the rapid expression of complex algorithms. Unfortunately, this comes at a cost: while there exist few broad, formal studies of the performance difference between scripting languages and their more traditional counterparts such as C and Java, informal benchmarks (e.g. [63]) suggest that scripts typically execute at least an order of magnitude slower and most researchers would agree that the gap is significant.

This performance difference is due in part to the fact that scripting languages are notoriously hard to statically analyze as a result of their abstract and flexible nature; it has been shown [58], for instance, that real world JavaScript programs are surprisingly dynamic, routinely modifying object definitions at runtime and relying heavily on complex contextual invariants. Several projects [69, 31, 66, 56] attempt to retrofit type systems and other static analyses onto existing scripting languages and a considerable body of research has been developed around the topic. This challenge of static analysis also presents scripting languages with scalability problems: with fewer detailed analyses available, debugging is more challenging and refactoring and feature implementation are more error prone.

This problem in static analysis is not pervasive or inherent in the nature of scripting languages; several of the projects cited above note that only a few features of scripting languages, such as the oft-incriminated `eval` function, are responsible. As a result, a partial typechecking of a script can be accomplished with gradual typing [64]. Such measures are only necessary, however, because the language contains semantics which were added to the language without static analysis in mind and, as it happens, are not strictly necessary to give a scripting language its abstract, flexible, and productive properties. In this dissertation, we show that *it is possible to construct a language which has the static analyzability of traditional languages and the flexibility of scripting languages.*

1.1 What is a "Scripting Language?"

This dissertation focuses on a process by which a typed scripting language may be developed. "Scripting language," much like "functional language" or "object-oriented language," is something of an amorphous phrase: while it is easy for most computer scientists to decide if a given language is a scripting language, the decision process itself is difficult to articulate. Nevertheless, the term is used at the time of this writing to denote at least the following informal properties:

- **Minimal redundancy.** The syntax of a scripting language is terse. Type signatures are rarely required (and often aren't permitted at all), variable declarations aren't typically necessary, and a wide variety of operators are available to express common non-trivial operations (such as sublist deletion). This terse nature extends to the semantics as well: interface declarations, for instance, are obviated by structural subtyping (often known as "duck typing") and common programming patterns are captured with functional abstractions or metaprogramming.

- **Flexible definitions.** Scripting languages treat definitions in a fluid fashion. Python, for instance, permits the dynamic addition of methods and fields to classes and objects. Code modules are typically first class and often allow their members to be redefined or decorated. Flow control is subject to this flexibility as well: core language semantics can be redefined by a programmer's code (such as exception handling in Ruby or failed method dispatch in numerous languages).

- **Dynamic typing.** Due in part to the above, scripting languages are challenging to statically type. Instead, scripting languages perform dynamic typing checks during evaluation (to preserve the sanity of the runtime environment). Simple static analysis tools ("delinters") are used to detect simple probable bugs and style errors. An emphasis on documentation and unit testing compensates for the requirement that programmers establish and maintain program invariants manually.

Although the above properties are far from a formal definition of scripting languages, they do allow us to pronounce many languages as either scripting languages or not. Python, Ruby, and JavaScript, for instance, are scripting languages: all three strongly demonstrate the properties listed above. Java and C++ are clearly not scripting languages: both require explicit interface declarations and type signatures and do not allow ad-hoc manipulation of data structures. Given the above, we would also say that Haskell is not a scripting language; although it demonstrates some of the functional abstraction properties described above and has type inference, it also lacks ad-hoc data manipulation.

In this document, we specify a "typed scripting language": a language which has the first two properties above but which is statically typed. Because we desire minimal redundancy, we require the system to provide *total type inference*: a programmer should never be required to write a type signature. Section 1.2 will briefly review existing work; Section 1.3 will discuss our strategy for defining such a language.

While the above properties are those on which we focus in this dissertation, the term "scripting language" is often used in another sense: to describe languages suitable for writing "glue code" which is largely responsible for controlling other applications or for wiring other applications together. VisualBasic and bash are examples of languages which fit into this category; other shell languages are also fine examples. While minimal redundancy and flexible definitions make languages like Python and Ruby more suited to application control and wiring that traditional languages like C and C++, they are still designed primarily as application programming languages. Although we recognize the "glue code" definition of the term "scripting language," this document does not use the term in that sense.

1.2 Existing Work

A considerable body of research has been devoted to statically typing scripting languages. That research has enjoyed a measure of success and has been instrumental in guiding some of the designs and decisions found here. We briefly discuss that research to provide context and to illustrate why it is insufficient for our purposes.

1.2.1 Retrofitting a Full Static Type System

Several early projects attempted to overlay a typechecker onto an existing dynamically-typed language [35, 11, 1, 29], but none of these approaches were widely adopted. This is in large part because the very design decisions which led to an elegant and flexible dynamic language also impede static typing. One obvious example is the scope of mutability. It is well-known that limiting the scope of mutation is critical to support flexible subtyping, but scripting languages allow mutation nearly everywhere.

Beyond the problem of language design is the fact that dynamically-typed languages in the hands of programmers can produce code that is far more convoluted than the language designers intended. In [58], the authors analyze a large set of real JavaScript programs and conclude that, contrary to common beliefs regarding the static analyzability of scripting languages, the dynamism in the language is fundamental and widely used: call sites are likely to be polymorphic, object properties are modified or deleted regularly, function types are variadic, and so on. Consider the following example from the Dojo JavaScript framework:

```
onto : function(arr, obj, f){
    if(!f){
        arr.push(obj);
    } else if(f) {
        var func = (typeof f == "string") ? obj[f] : f;
        arr.push(function(){ func.call(obj); });
    }
}
```

The meaning of this function is well-explained by the authors of System D [17], who derive a simplified example from it to showcase the difficulty of statically typing scripting languages. From that paper:

> The function `onto` is used to register callback functions to be called after the DOM and required library modules have finished loading. The author of `onto` went to great pains to make it extremely flexible in the kinds of arguments it takes. If the `obj` parameter is provided but `f` is not, then `obj` is the function to be called after loading. Otherwise, both `f` and `obj` are provided, and either: (a) `f` is a string, `obj` is a dictionary, and the (function) value corresponding to key `f` in `obj` is called with `obj` as a parameter after loading; or (b) `f` is a function to be called with `obj` as a parameter after loading.

Statically typing this form of case-analysis-driven logic proves to be difficult. The authors of [17] present a core calculus which is capable of typechecking code of similar dynamic complexity, but the resulting work yields an inference algorithm which is only local and which cannot infer polymorphic types.

At the time of this writing, probably the most successful attempt to statically type an existing scripting language is the DRuby project [31]. The DRuby system is a sophisticated application of subtype constraint theory which includes union and intersection types, local flow-sensitive analysis, and other techniques designed to capture the behavior of existing Ruby source code. Despite being the most complete result thus far, DRuby is neither sound nor complete: some unsound programs are admitted by the type checker and some valid programs are rejected. DRuby does not typecheck the Ruby API because some functions such as `flatten` (the type of the function which recursively collapses a list such that e.g. `[1,[[2,3],4]]` becomes `[1,2,3,4]`) do not have finite types in DRuby's type system. DRuby also requires type annotations on polymorphic functions. Although DRuby was capable of typechecking several open source Ruby libraries with little to no modification, the problems described here illustrate the fundamental challenges in retrofitting a type system onto a mature scripting language.

1.2.2 Gradual Typing

Another common strategy is to statically type only part of the scripting language, using gradual typing [64] to bridge the gap between typed code and untyped code. Typed Racket [66], for instance, distinguishes between Typed Racket modules and (untyped) Racket modules. The syntax of Typed Racket is largely compatible with Racket, but type signatures are required on top-level members. The compiler then inserts the appropriate dynamic checks at the boundary between typed and untyped modules, giving the guarantees and benefits of static typing to the modules which include static type information. While Typed Racket does relax the boundary between statically typed and dynamically typed code, its approach is not compatible with our objectives: it requires the programmer to write type signatures, which (other than for purposes of checking or documentation) script programmers perceive as redundant.

Some projects use gradual typing without requiring type signatures; TypeScript [19] is an example of such a language. TypeScript is a superset of JavaScript which permits (but does not require) type annotations, defines a structural subtyping relation on objects, and infers types wherever possible. These properties of TypeScript are desirable and the language has proven itself in practice to some degree, but TypeScript's type system is not sound. This unsoundness is intentional as, in some cases, it allows existing JavaScript programs and libraries to be annotated with TypeScript signatures without rewriting a considerable amount of code. By introducing this unsoundness, however, the designers of TypeScript were able to ignore subtle issues of parametricity which are non-trivial to solve. A deeper analysis of the TypeScript type system is difficult as, at the time of this writing, it is not formally specified.

Regardless of the particular strategy used, gradual typing presents a problem when our objectives are considered: fundamentally, gradual typing systems are designed to allow dynamically typed code to link with statically typed code in such a way that the invariants of the latter are preserved. While this is a useful strategy for dealing with existing languages, it does not provide a complete static analysis and so prohibits certain optimizations and runtime guarantees from being utilized. In a sense, gradual typing permits us to avoid static typing for the problematic portion of the language; while this allows us to statically type more of our program than we could with a purely dynamic system, it is not a complete solution.

1.3 Our Approach: The *Bang Family

The approaches discussed above all share a common trait: they are attempting to add static typing to an existing scripting language. These languages are designed for dynamic behavior with no consideration for static analysis in mind. There is considerable motivation to do so: as there is already a significant volume of code written in these languages, any improvement on the situation has immediate and widespread benefits. Unfortunately, it also serves as a metaphorical ball and chain, since the semantics of the scripting language may be impossible to statically capture in a desirable way.

With that in mind, we instead approach this problem by defining a scripting language from scratch. We include in this language only those semantics we can effectively typecheck, but we make sure to include those necessary to express common scripting idioms. For instance, our design does not support object mutation, but it permits object functional extension which achieves many of the same goals. This set of semantics was not selected *a priori*; it is the result of numerous iterations of the language development process we illustrate in this document and refinements based on practical observation of what was and was not possible in each variation.

In order to keep the language definition manageable, we do not simply define a single programming language and type system with all of the language features (classes, mixins, higher-order functions, etc.) that we desire. We instead develop a core scripting language called TinyBang over a small calculus in which the semantics we desire can be encoded. We then define LittleBang, a language including common scripting features and conforming to our requirements of minimal redundancy and flexibility, in terms of a desugaring procedure which yields TinyBang code.[1] Although encoding existing scripting language features into TinyBang is a significant motivator in our work, TinyBang's core semantics are interesting in their own right due to their expressiveness.

To give a sense of how this process works, we now give an outline of the semantics of TinyBang. We use informal, ML-like syntax here; we discuss the actual syntax of TinyBang in Section 2. We ask readers to bear in mind that the following semantics are statically

[1]These languages are part of the BigBang research project. The eventual goal of that project (which is beyond the scope of this dissertation) is to define a full-fledged production-ready scripting language which includes user-defined syntax extensions, efficient compilation tools, metaprogramming facilities, and a pony.

typable, which we demonstrate in Section 3.5.

1.3.1 Onions and Asymmetric Concatenation

Existing scripting languages, especially object-oriented scripting languages, rely heavily on the use of dictionary structures.[2] It is natural, then, to create a typed scripting language by beginning with a typed dictionary structure. In TinyBang, we satisfy this requirement with a primitive data structure we call an *onion*.

A TinyBang onion is a type-indexed record [61]. The elements of type-indexed records are labeled by their types rather than explicit names. Informally, one can consider the type-indexed record `{5, 'z', "hello"}` which contains three elements: an integer, a character, and a string. Projection from this record is by type; for instance, `{5, 'z', "hello"}.int` would evaluate to `5`. Traditional records are easily encoded using a label constructor (which functions much like a Haskell `newtype`): `{foo = 5, 7}` is a record containing an integer (`7`) and a `foo`-labeled integer (`foo = 5`). Because records are type-indexed, we can choose not to distinguish between records and non-records: we perceive `5` as an onion (type-indexed record) containing a single integer.

TinyBang, like other scripting languages, does not require variables to be fixed at a particular type. A single variable may contain an integer at one point and a string at another; the type system will infer an appropriate union type. A variant type can then be expressed as a variable to which multiple different types can be assigned. A normal variable (one which is assigned from only one type) is then essentially a 1-ary variant. As a result, a 1-ary variant and a 1-ary record in TinyBang have the same representation. In this way, onions serve as TinyBang's common data type (similar to the role of lists in LISP or objects in Ruby or Smalltalk); that is, everything is an onion.

Onions in TinyBang support *asymmetric* concatenation via the onioning operator `&`. Informally, evaluating `{foo = 54, bar = 3} & {bar = 12, 9}` yields `{foo = 54, bar = 3, 9}`. Because a `bar`-labeled integer element appears in both records, the onioning operator prefers the one on the left. This asymmetric concatenation property of onions allows them to function as dictionaries with a set of statically-known keys, satisfying our need for a typed dictionary structure. This serves as the foundation of a number of asymmetric features in the language, including overloading, subclassing, and mixins.

[2]These dictionary structures go by several names: dictionaries in Python, hashes in Perl, associative arrays in PHP, and so on. Even in Ruby, where "everything is an object" [60], objects are represented internally as dictionaries. To be clear, "dictionary" here means a finite mapping from identifiers to values.

1.3.2 Pattern-Matching Partial Functions

TinyBang, like most modern scripting languages, supports first-class functions. In TinyBang, however, all functions are written `fun` p `->` e where e is an expression and p is a single pattern (similar to an ML pattern). This grammar subsumes that of typical lambda expressions; the increment function, for instance, can be expressed as `fun x -> x + 1`. We can also (informally) write `fun {foo = x} -> x` as the function which projects the `foo` field from a record. If a record without a `foo` field were provided to that latter function, a compile-time type error would occur.

The above is relatively standard fare for a language with a static type system, subtyping, and first-class functions. Functions in TinyBang are interesting because they also function as first-class *case clauses* and a generalization of the cases in MLPolyR [8]. They operate in this fashion because an *onion of functions* exhibits asymmetric dispatch behavior. For instance, if `projfoo` is the above `foo`-projection function and `projbar` is a similar `bar`-projection function, then we can write `projfoo & projbar` to concatenate those functions. The resulting compound function projects either `foo` or `bar` from its argument. If the argument has neither, a type error results; if the argument has both, `foo` is projected (because onioning asymmetrically prefers the left for dispatch as well).

The TinyBang type system ascribes unusually precise types to these first-class functions (and concatenations thereof). In standard type systems, all case branches are constrained to have the same result type; this loses the dependency between the variant input and the output. In TinyBang, compound function types are not reduced; they are described as the type-level concatenation of their simple function types. As a result, application of a compound function can return a different type depending on the variant constructor of its argument. For instance, `f = (fun int -> 0) & (fun char -> 'z')` is a function which transforms any integer into `0` and any character into `'z'`. The type of `f 3` is `int` (and not, as is typically inferred, `int` $\cup$ `char`). In order to support this mechanism, our type system uses a notion of a *filtered type* which provides most of the desired behavior of positive intersection types without succumbing to the well-understood performance problems [4] that they imply.

1.3.3 Call-Site Polymorphism

The TinyBang type system uses a *call-site* polymorphism model (as opposed to the `let`-polymorphism models common in the ML family of languages). In `let`-polymorphism, functions are abstracted to polymorphic types when they are bound to variables in a `let` expression; they are then instantiated when those variables are used. Since polymorphic types are only assigned to variables in the type environment and not to expressions themselves, polymorphism is bounded by lexical scope. This has the advantage of simplifying compilation – polymorphic functions can only be used at types or type variables dictated by the lexical scope of the function – but the disadvantage of limiting expressiveness, especially with respect to contextual invariants.

In call-site polymorphism, function expressions themselves are inferred polymorphic types. Further, while `let` polymorphism instantiates functions when the function variable is named, call-site polymorphism delays this process until the function is *called*; thus, a function retains its polymorphism e.g. when returned by another function. The primary motivator for this decision is the manner in which we apply the compound functions (and their precise types) above. By using call-site polymorphism, we are able to capture the sort of case-based reasoning used in e.g. the example in Section 1.2.1. We formalize the TinyBang type system to be parametric in its polymorphism model but also present a polymorphism model which we use in practice. We design that polymorphism model using techniques similar to existing static analysis work [62, 1, 74, 43], which allows it to take advantage of compound function types to preserve type alignment during case analyses. Throughout most of this document, we invite the reader to assume that the type system is sufficiently polymorphic for the examples we present. The formalization of the polymorphism model appears in Chapter 6.

1.3.4 Contributions

This document is driven by the objective of building a typed scripting language from scratch. To that end, it makes the following contributions:

- The development of the filtered type (Section 1.3.2), a form of heterogeneous intersection type which captures precise contextual information during pattern matching without exponential blow-up.
- A novel call-site polymorphism model and an unusually precise notion of variable freshening required to adapt it to a (decidable) constraint theory.
- A strategy for proving type soundness which is inspired by abstract interpretation but novel in type system literature.
- A typed, variant-based object encoding which permits object extension at later stages than previous works [9] while retaining subtyping properties such as depth subtyping that other models do not [59].
- The synthesis of these ideas and underused type theory research to create a general process for the development of a from-scratch typed scripting language, of which

TinyBang is an example.

1.4 Outline

The remainder of this dissertation is organized as follows. First, Chapter 2 provides an informal overview of TinyBang; we demonstrate the semantics of the language and show how it captures several common scripting language features. Next, we give a formal definition of a reduced form of TinyBang's operational semantics and type system in Chapter 3. Chapters 4 and 5 give the soundness and decidability proofs for the TinyBang type system. Chapter 6 presents the polymorphism model we use with TinyBang in practice and demonstrates that it satisfies the requirements of our proofs.

Chapter 7 demonstrates how the formalism from Chapter 3 can be extended with common features such as reference cells and arithmetic. In Chapter 8, we then use the resulting system – the formalization, polymorphism model, and built-ins – to formally define the encodings we outline in Chapter 2. Chapter 9 gives some discussion of how the implementation of the TinyBang typechecker (and therefore the checking of LittleBang programs encoded to it) can be made efficient. Finally, Chapters 10, 11, and 12 discuss related work, future work, and conclusions for this material.

Readers need not consume this document in a linear fashion. Figure 1.1 shows a graph of recommended prerequisites for each chapter.

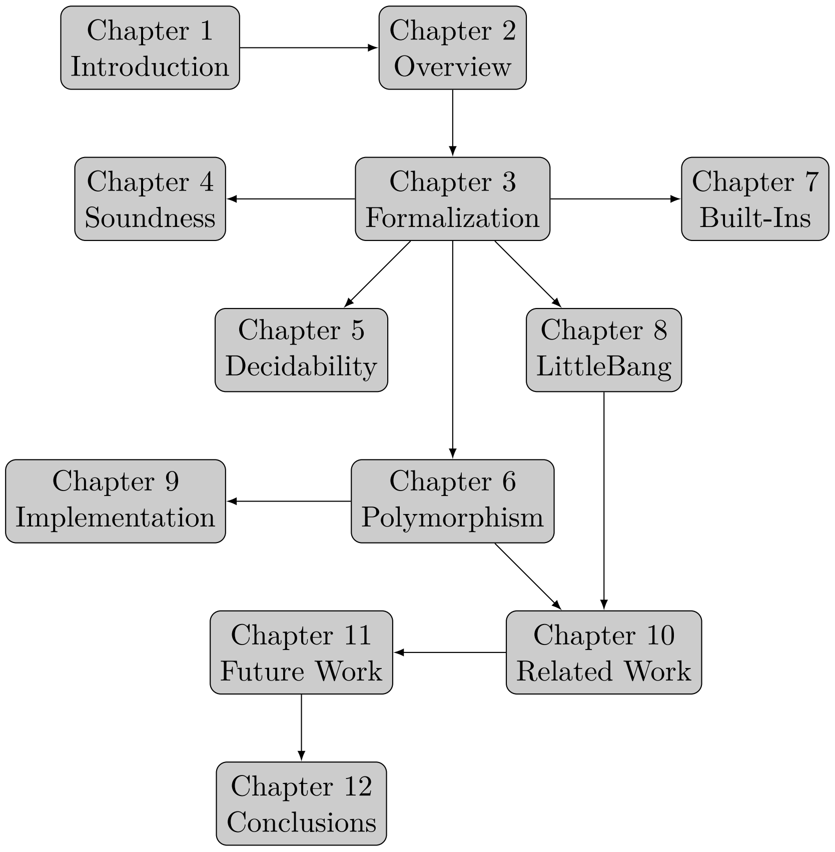

Figure 1.1: Chapter Dependencies

Chapter 2

Overview

This chapter provides an overview of the semantics of TinyBang. We illustrate these semantics by encoding several features, most of which are common in scripting languages. A more complete discussion of these encodings can be found in Chapter 8, but that presentation requires familiarity with the formal operational semantics and type system. To provide a quickly-accessible perspective on this work, this chapter uses informal notation for types and draws from the intuitions of ML-like languages.

2.1 TinyBang Syntax

There are three concrete languages presented in this dissertation. The first is LittleBang, the language containing syntax for the features we desire and which we formalize in Chapter 8. The second is nested TinyBang, which we present here as a target for encoding the features of scripting languages. The third concrete language appears in the formalization of the system (Chapter 3) because it is an easier target for demonstrating formal properties; we call it ANF (or A-normal form) TinyBang. We use the name TinyBang to refer to one of the latter two languages when it is clear which we mean from context. Throughout this chapter, for instance, we will use the unqualified term "TinyBang" to refer to the nested version. The syntax of the (nested) TinyBang language appears in Figure 2.1. We use $\mathbb{Z}$ to denote literal integers and $\mathbb{S}$ to denote literal strings.

In the examples below, we generally avoid using operator precedence. For legibility, though, some understood precedence rules are helpful. We consider labels ($l\ e$) to have the highest precedence of all operators; these labels function much like Haskell `newtype`s or

$$\begin{array}{rcll}
e & ::= & x \mid \texttt{()} \mid \mathbb{Z} \mid \mathbb{S} \mid l\ e \mid e\,\texttt{\&}\,e \mid \texttt{ref}\,e \mid \texttt{!}\,e \mid x\,\texttt{:=}\,e\,\texttt{in}\,e \mid & \textit{expressions} \\
 & & \texttt{let}\,x\,\texttt{=}\,e\,\texttt{in}\,e \mid \texttt{fun}\,p\,\texttt{->}\,e \mid e\ e \mid e \boxdot e & \\
p & ::= & x \mid \texttt{()} \mid \texttt{int} \mid \texttt{str} \mid l\ p \mid p\,\texttt{*}\,p & \textit{patterns} \\
\boxdot & ::= & \textit{(binary operators: equality, addition, etc.)} & \textit{operators} \\
l & ::= & \texttt{`}\,\textit{(alphanumeric)} & \textit{labels} \\
x & & & \textit{variables}
\end{array}$$

Figure 2.1: TinyBang Syntax

OCaml polymorphic variants in that they simply wrap an underlying value. For instance, the expression `‘A 5` has type `‘A int`. The onioning operator has lesser precedence than the label constructor; thus, `‘A 5 & ‘B 6` is grouped `(‘A 5) & (‘B 6)`. Finally, functions of the form $\texttt{fun}\,p\,\texttt{->}\,e$ have the least precedence; that is, `fun x -> x & 1` is the function which right-onions a `1` onto a value (and not an onion between the identity function and the integer `1`). Precedence for the remaining language components is similar to ML. Pattern precedence follows expression precedence: label patterns ($l\ p$) have higher priority than conjunction patterns ($p_1 \,\texttt{*}\, p_2$).

TinyBang program types take the form of a set of subtype constraints [4]. For the purposes of this overview, we will be informal about type syntax. Our formal type grammar can be viewed as an A-normalized form of a typical type grammar (which is a large motivator for the ANF TinyBang discussed in Chapter 3): each type variable is immediately bounded by a shallow type, such as $\texttt{int} <: \alpha$ or $l\ \alpha_1 <: \alpha_2$. Union types emerge naturally from this system by giving a variable multiple lower bounds (e.g. $\texttt{int} \cup \texttt{str}$ can be represented by the set $\{\texttt{int} <: \alpha, \texttt{str} <: \alpha\}$. As we do with the expression syntax, we will write types in nested form in this section for presentation clarity. The formal type system appears in Section 3.5.

2.2 Conditionals: A Simple Exercise in Compound Functions

We begin our discussion of TinyBang with a simple observation: the syntax in Figure 2.1 does not include a conditional expression (i.e. `if` e `then` e `else` e), nor does it include boolean values. In TinyBang, boolean values are encoded using the expressions `‘True ()` and `‘False ()`; here, `()` represents the empty onion, the 0-ary conjunction of other values. This encoding is first-order and trivial to type; the encoding of boolean operations is only slightly more involved.

As indicated in both Section 1.3.2 and Figure 2.1, all functions in TinyBang include a pattern; in a sense, this means that they are potentially partial. The function `fun int -> 0`, for instance, maps all integers to zero but cannot be applied to non-integer values; the expression `(fun int -> 0) "bad"`, for instance, fails to evaluate (and is rejected by the type system). Patterns can match the contents of labels as well; for instance, `(fun ‘A x -> x + 1) (‘A 3)` evaluates to `4`.

The introduction also indicated that functions can be combined using the onion operator `&` to produce *compound functions*. Dispatch on compound functions applies the leftmost function with a matching pattern. For example, suppose `f` is a variable containing the value `(fun int -> 0) & (fun str -> "")`. Then `f 4` evaluates to zero, `f "hello"` evaluates to the empty string, and `f ()` produces a type error (since `()` matches neither pattern).

The above shows we can branch on the type of an argument. Because each label implies a distinct type, this means we can branch based on whether the type of our argument is `‘True ()` or `‘False ()`. Comparison operators in TinyBang yield one of these two values. To express a condition (e.g. `if x == y then x + 1 else x - 1`), we can write the following:

```
( (fun ‘True _ -> x + 1)
& (fun ‘False _ -> x - 1) )  (x == y)
```

(Note here that the pattern `_` is used as an alias for `()` for readability. Because every type is a subtype of the empty onion, the pattern matching the empty onion matches everything else; that is, the empty onion is the top of the TinyBang type lattice.)

This expression will pass the evaluation of `x == y` to the compound function on the left. As this value is either a `‘True ()` or a `‘False ()` (depending on whether the values are equal or not), this is typesafe: all cases of the argument are exhaustively matched. Based on which value is produced, a different function body is invoked. In practice, of course, a compiler would not generate a function dispatch for every conditional; the runtime representation of `x == y` can clearly be reduced to a machine word as long as it only reaches the application site and is not used in some other disjunct. By using these semantics in TinyBang, however, we keep the core of the language (and thus the formal definitions in later chapters) quite small.

As mentioned in Section 1.3, we achieve our goal of a typed scripting language by developing TinyBang as a minimal core language and then building LittleBang, a language with scripting language features, through a series of formal encodings. The above is an example of such an encoding; although LittleBang includes booleans, they are encoded into TinyBang as labeled data and compound functions. As this chapter proceeds, we will use encodings from previous sections; in this case, for instance, we may use the syntax `if` e_1 `then` e_2 `else` e_3 as shorthand for the above encoding to improve readability. As this overview is merely an introduction to the semantics, formal encodings are not given for every feature until Chapter 8.

2.3 Overloading

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

```
let neg = fun x * int -> 0 - x in ...
```

To clarify, `x * int` is a conjunction pattern in the above for the function which subtracts its argument from zero; that is, we can also write `fun (x * int) -> (0 - x)`. A conjunction pattern matches only if both of its conjuncts match; the `int` pattern matches only integers while the `x` pattern matches everything (and binds whatever it matches to the variable `x`).

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

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

Negation is now overloaded: `neg 4` evaluates to `-4`, and `neg ‘True ()` evaluates to `‘False ()` due to how application matches against the patterns of functions. Note that this assignment is not an update to the `neg` variable but an assignment to a new one; this allows overloadings to be given a controlled scope in which they apply. In our object model presented below, overloaded methods are encoded using this same approach.

2.4 Onions as Function Arguments

TinyBang functions are reminiscent of ML family functions in that they take a single argument (and not an argument list). Most modern scripting languages (including Python, Perl, Ruby, and JavaScript) use a C-like function syntax: functions are declared and invoked with lists of parameters and arguments which are delimited by parentheses. Of course, the latter convention can be encoded in TinyBang in the usual way: by passing a single argument which contains all of the parameters. We use onions in this capacity, since they are the (only) data conjoiner in our core language.

To match on onions, we make use of the conjunction pattern operator *. For instance, to encode

```
let f(x,y) = x + y in f(5,3)
```

we can write

```
let f = fun ‘1 x * ‘2 y -> x + y in f (‘1 5 & ‘2 3)
```

As described above, the conjunction pattern ‘1 x * ‘2 y matches only if both of its conjuncts – ‘1 x and ‘2 y – match the argument (which is ‘1 5 & ‘2 3 here). They do, while binding x to 5 and binding y to 3. The body of the function is then executed, resulting in the value 8.

This encoding also creates appropriate type errors. The invocation f(7), which is missing an argument, decodes to f (‘1 7). Because the ‘2 y pattern cannot match this argument, the conjunction pattern cannot match it. Because f here is a simple function, there are no other patterns to match and so a type error occurs.[1]

2.4.1 Keyword Arguments

As we have demonstrated, TinyBang's compound functions enable a form of case analysis on their input. When that input is an onion of arguments, case analysis allows a form of argument preprocessing which is common in scripting languages. Python and Ruby 2, for instance, allow callers to provide arguments to functions by name rather than position. These *named* or *keyword* arguments are useful for calling functions which accept numerous parameters in order to make the call site more readable. In TinyBang, we can capture this behavior by matching the named arguments when they exist and constructing an argument record with those arguments in the appropriate positions. For instance, we may encode let f(x,y) = x + y in f(5,y=3) as

```
1 let f’ = fun ‘1 x * ‘2 y -> x + y in
2 let f = (fun (‘1 x * ‘2 y) * a -> f’ a)
3       & (fun (‘1 x * ‘y y) * a -> f’ (a & ‘2 y))
4       & (fun (‘x x * ‘y y) * a -> f’ (a & ‘1 x & ‘2 y)) in
5 f (‘1 5 & ‘y 3)
```

The invocation on line 5 begins dispatch by attempting to match the argument, (‘1 5 & ‘y 3), against the first pattern in f (‘1 x * ‘2 y, which appears on line 2). This conjunction does not match because the pattern ‘2 y cannot match an argument without a ‘2 component. Because f is a compound function, this is not a type error; application

[1]Using this simple encoding, of course, *extra* arguments (e.g f(5,3,2)) would simply be ignored. We discuss how this may be addressed in Section 8.5.3.

proceeds to the pattern on line 3 and this pattern does match the argument, so its body is invoked. The function body on line 3 appends to the right of the argument `a` the value `‘2 3`. The resulting value `‘1 5 & ‘y 3 & ‘2 3` is passed to the "real" function `f’`, which then extracts the `‘1` and `‘2` components.

2.4.2 Default Arguments

Thus far, we have used TinyBang's onion operator to conjoin distinct types of data; we have not discussed the implications of overlapping data types. For instance, consider the expression `5 & 6`; what meaning does the resulting value have in addition? In TinyBang, onions are asymmetric, preferring the leftmost element; that is, `(5 & 6) + 3` evaluates to `8`. In fact, the expression `5 & 6` is operationally equivalent to the expression `5`; both operands have the same type and, as onions are typed-indexed records, only one value may appear at a particular type index. A similar property is true for labels – `‘A 5 & ‘A 6` is operationally equivalent to `‘A 5` – but only when their types are deeply indistinguishable; the expressions `‘A 5` and `‘A 5 & ‘A "six"`, for instance, are not operationally equivalent because the pattern `‘A str` matches only the latter.

Some early theories of record concatenation include asymmetry [71] but were later found to be unsound [72]. Asymmetry causes mathematical subtleties that complicate a type theory and, for this reason, research into record concatenation has focused primarily on symmetric operations [36, 57, 53, 51]. Symmetric concatenation is initially quite appealing as a basis for record concatenation as troublesome cases of field overlap cannot occur by definition. In the presence of polymorphism, however, sound theories of symmetric concatenation must enforce the presence or absence of labels e.g. via row types and their inference. We choose asymmetric concatenation in TinyBang because it enables us to encode a variety of fundamentally asymmetric language features.

As an example, we consider *default arguments.* The keyword arguments discussed above are often used to simplify invocations of highly configurable functions with numerous sensible defaults (e.g. the `Popen` constructor from Python's `subprocess` module, which includes more than a dozen default keyword arguments for controlling file descriptors, buffer sizes, and other subprocess properties). In this case, the function definition includes default values for the arguments which are not provided by the caller. We demonstrate how asymmetry can be used to capture this language feature through a simple example.

Consider the Python function `def inc(x,y=1) = x+y`. This function adds `y` to `x` if `y` is supplied; it adds `1` to `x` otherwise. The following TinyBang code accomplishes this task (which eschews the keyword argument functionality above for readability only):

```
let f’ = fun ‘1 x * ‘2 y -> x + y in
let f = fun (`1 x) * a -> f' (a & `2 1) in
...
```

If we call this function using an onion as a record of arguments as suggested at the top of Section 2.4, then the Python invocation `f(3)` would result in `4`. In TinyBang, this call would be written `` f (`1 3) ``, which has the effect of passing `` `1 3 & `2 1 `` to `f'`. If we instead evaluate `` f (`1 3 & `2 5) `` (the TinyBang equivalent of the Python `f(3,5)`), then `` `1 3 & `2 5 & `2 1 `` is passed to `f'`. As discussed above, the pattern in `f'` will match the leftmost `` `2 `` in the onion; thus, the right-onioning of `` `2 1 `` will have no effect unless the original argument included no `` `2 `` component.[2]

2.5 Sanitized Strings: Using Labels as Tags

TinyBang's pattern matching semantics do not require that the pattern match is exhaustive; the argument may include additional elements which are not matched and variables are bound against the whole structure of the argument (not just the pattern which was matched). As an example, `(fun int -> 0) (3 & "three")` is permissible; the additional string in the argument does not prevent it from matching the `int` pattern since the argument has an `int` component. Extra data of this form is only lost when we operate on the value: `(fun x -> x + 1) (3 & "three")` yields `4` (and not `4 & "three"`) because primitive integer addition constructs a new integer using the integer components of its operands.

We can use this property of onions to represent sanitation tracking, a practical safety feature of web frameworks like Rails and Django. Those frameworks feature webpage template languages which permit the insertion of strings into web pages. To prevent code injection attacks, inserted strings are automatically escaped unless they have been marked as "safe". This is the dual of the naïve (and error-prone) behavior: instead of explicitly escaping those strings which may cause problems, we escape every string which has not been explicitly blessed by the web application. Strings may be marked safe at the time they are inserted into the page or beforehand by a utility which has generated a known-safe string, but further operations on safe string do not yield safe strings and must be handled with care.

In TinyBang, marking a string as safe (or marking any data with such an annotation) is as simple as onioning a distinct label onto the value. For instance, we can consider the string `"<script src='foo.js'/>"` to be unsafe and we can consider the string `` "<script src='foo.js'/>" & `safe () `` to be safe.[3] We can then write the following

[2]This encoding also does not deal with extra arguments. Again, we discuss this in Section 8.5.3.

[3]At the time of this writing, many browsers bizarrely do not handle self-closing script tags correctly

function to ensure that a string is safe:

```
let ensureSafe = (fun s * str * ‘safe _ -> s)
               & (fun s * str -> htmlEscape s) in ...
```

Here, the `ensureSafe` function matches `‘safe`-tagged strings in the first branch and other strings[4] in the second branch. The variable `s` matches the entire argument in both cases; thus, in the first case, the argument is returned intact, including its `‘safe` tag.

2.6 Dynamic Object-Oriented Semantics

Object-orientation plays a significant role in modern scripting languages and it has received significant attention in the design of TinyBang. As with the above features, objects are not built into the semantics of TinyBang. Instead, we show in this section how an object model can be encoded using onions and compound functions. This object model supports statically-typed object extension properties (such as functional extension after use) which are not present in previous calculi and does so without the use of dedicated metatheory.

2.6.1 Variant-Based Objects

Widely used object-oriented languages such as C++ and Java view objects as record structures: each object is effectively a dispatch table and a collection of named fields, both of which are statically typed dictionaries. This record-object tradition extends to scripting languages such as JavaScript and Python. In JavaScript, for instance, methods are added to objects simply by assigning a function to the appropriate label. Method invocation is accomplished by looking up the appropriate function in the dispatch table or method dictionary and passing it the object as well as the other method arguments.

Dual to the record-based model of objects is the *variant-based* model. In this model, dispatch is not accomplished by examining the object to find the appropriate handler. Instead, the object is treated as a function and passed the message directly; it is then the responsibility of the object to determine how to process the message. This notion of variant-based objects is not new; it is used in an untyped form in classic actor models [2]. These models traditionally only appear in dynamic languages because they are difficult to type, but TinyBang's onions are well-suited to the task. We therefore draw from these variant-based designs when encoding objects in TinyBang.

anyway, so the “unsafe” string would be inadvertently inert. We use the self-closing tag here for presentation purposes.

[4]Recall that dispatch always takes the leftmost match. The second pattern matches *all* strings, but the first pattern captures all `‘safe`-tagged strings and only the remaining strings are compared with the second pattern.

2.6.1.1 Simple Functions as Methods

We begin by considering the oversimplified case of an object with a single method and no fields or self-awareness. In the variant encoding, such an object is represented by a function which matches on a single case. In TinyBang, this is the form of *all* simple functions. For instance, consider the following object and its invocation:

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

This “object” is just a simple function accepting a `‘twice`-labeled argument. Messaging this object is the same as invoking it as a function. In order to define an object with multiple methods, one need only onion the message handlers together:

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

We note that, because of this means of dispatch, TinyBang features *first-class messages*: the message sent to an object is a first-class value and can be stored or passed separate from the object that eventually receives it.

The manner in which we define objects as an conjoined set of dispatch functions is related to the λ&-calculus [13] and with typed multimethod-based language designs [46, 45]. These works perform dispatch nominally (that is, by declared class names); by contrast, TinyBang's dispatch is structural.

2.6.1.2 Heterogeneous Case Types

The above shows how to encode an object with multiple methods as an onion of simple functions. But we must be careful not to type onions in the way that match/case expressions are traditionally typed. The analogous OCaml match/case expression

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

will not typecheck; OCaml match/case expressions must return the same type in all case branches.[5] Instead, the types of onions preserve the types of their components; we would give the above a type which is, informally,

$$(\texttt{‘twice int} \rightarrow \texttt{int}) \;\&\; (\texttt{‘isZero int} \rightarrow \texttt{‘True ()} \cup \texttt{‘False ()})$$

If the compound 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

[5]OCaml 4 GADTs mitigate this difficulty but require an explicit type declaration, type annotations, and only work under a closed world assumption.

`‘twice 4` always produces type `int` and not type `int` ∪ `‘True ()` ∪ `‘False ()`. This captures a form of associated type [16] similar to Haskell type families or C++ traits classes. As a result, `match` expressions encoded in TinyBang may be heterogeneous; that is, each case branch may have a different type in a meaningful way. When we present the formal type system below, we show how these heterogeneous case types extend the expressiveness of conditional constraint types [4, 54] in a dimension critical for typing objects.

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

2.6.2 Self-Awareness and Resealable Objects

The above objects have not been able to invoke their own methods, so our encoding is incomplete. Dynamic languages often address this problem simply by adding the object as a parameter to each of its methods. This is a suitable approach in dynamic languages, as the runtime is the only system of concern. In a statically-typed language, however, this results in subtle typing problems. To see why, consider the following Python pseudo-code:

```
class A(object):
  def foo(self, x): x + 1
class B(A):
  def foo(self, x): self.bar(x) + 1
  def bar(self, y): y + 1
```

If we view the above as collections of message-handling functions, we must be able to assign types to those functions. What is the type of the `self` parameter in the `foo` methods above? The intuitive conclusion is to give `self` the type of the class in which it appears: `A` for the first and `B` for the second two. But the type of the first method is then $(\texttt{A}, \texttt{int}) \rightarrow \texttt{int}$ and the type of the second method is $(\texttt{B}, \texttt{int}) \rightarrow \texttt{int}$. This presents a problem: because the `B` implementation of `foo` overrides the `A` implementation, we need to use it in the latter's place. This would require that $(\texttt{B}, \texttt{int}) \rightarrow \texttt{int}$ be a subtype of $(\texttt{A}, \texttt{int}) \rightarrow \texttt{int}$, but that cannot be: function subtypes are contravariant on the input of the function. The only reason this is okay in practice is that we have knowledge (which is not expressed by these types) that the `B.foo` method is only used in conjunction with `B` objects. Existing object calculi deal with this incoherence by giving more sophisticated types to objects; we avoid this approach as we desire to keep the TinyBang core as small as possible.

The above incoherence arises because the type of `self` is captured in a contravariant position. We can therefore solve this problem by capturing the appropriate type of `self` in closure when it is known (rather than leaving it exposed to variance). We describe this capturing operation as *sealing* the object, taking that vocabulary from previous work [9] on extensible object calculi. 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. A prototype may be "sealed" to transform it into a proper object, at which point it may never again be extended. We improve on that situation shortly, allowing sealed objects to be extended and resealed, but first present our sealing as follows. Unlike [9], our model requires no special metatheory: it is defined directly as a function `seal` using onioning semantics to capture types appropriately.

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

A more in-depth discussion of the typing of this function appears in Section 8.6.2. We can summarize its behavior by saying that it adds a message handler which captures *every* message sent to `obj`. (We still add `& obj` to this message handler to preserve the non-function parts of the object.) The message handler adds a `‘self` component (containing `seal obj`) to the right of the message and then passes it to the original object. We require `fixpoint` to ensure that this self-reference is also sealed. Thus, every message sent to `sObj` will, in effect, be sent to `obj` with `‘self sObj` attached to the right. Because the type of `obj` is captured here in closure in a positive position, it is not subject to the same contravariance problems we witnessed in the example above.

2.6.2.1 Extending and Resealing Objects

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

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

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

The above resealing types correctly because our type system includes parametric polymorphism and so `sObj` and the resealed `sObj2` do not have to share the same `self` types. Because this is a *functional* extension, there is 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 present a variant model here.

Although `seal` provides an in-theory means of extending and resealing objects, its precision is bounded by the accuracy of the type system and polymorphism model we use. It is not possible, for instance, to extend and reseal an object beyond the point where significant information about its type has been lost. A discussion of these limitations appears in Section 8.6.2 after the type system and polymorphism model have been presented.

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

2.6.3 A Hybrid Object Model

Although the design of message handling in TinyBang is heavily inspired by variant-based object models, onions provide a natural record-like mechanism for including fields: we simply concatenate them to the functions that represent the methods. This results in a hybrid variant-for-methods, record-for-fields object model. Consider the following object which stores and increments a counter:

```
let obj = seal (‘x (ref 0) &
                (fun ‘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. When dispatching on an onion which includes non-function values, those values are ignored; they are treated as if they have a pattern which matches no values. So here, the invocation `obj ‘inc ()` correctly increments in spite of the presence of the `‘x` label in `obj`. We note that the above counter object code uses no syntactic sugar or previously-defined encodings; despite this, it is quite concise considering that it defines a self-referential, mutable counter object.

Because fields are encoded here as the contents of labels, we have no way to access them except by pattern matching. That said, we can define a simple projection sugar: we write $e.x$ to mean `(fun` ‘x x' `->` x'`)` e for some fresh x'. Using this sugar, the update in the counter object above could be expressed more succinctly as `self.x := self.x + 1 in self.x`.

2.6.4 First-Class Mixins

Given the above, we can also use onions to define mixins [10] for the objects above. 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)
    & (fun ‘l1 _ * ‘self self -> self.x + self.y)
    & (fun ‘isZero _ * ‘self self ->
            self.x == 0 and self.y == 0)) in
let mixin = fun ‘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 () * ‘self` α`)` $\rightarrow$ `bool` where α is an object capable of receiving the `‘l1` message and producing an `int`". `mixin` can be onioned with any object that satisfies these properties. If the object does not have these properties, a type error will result when the `‘near` message is passed; for instance, `(seal mixin) (‘near ())` is not typeable because `mixin`, the value of `self`, does not have a function which can handle the `‘l1` message.

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

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

2.6.5 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 generators.

Chapter 3

Formalization

We now give formal semantics to a subset of TinyBang. For clarity, features which are largely orthogonal to the conjunction semantics – primitives, state, etc. – are omitted.

We begin by translating the TinyBang program to A-normal form; this gives a more direct alignment between expressions and types and considerably simplifies the soundness and decidability proofs. Section 3.3 describes some well-formedness properties on the resulting grammar which are used in the rest of the formalization. In Section 3.4, we define the operational semantics of this A-normalized TinyBang language. Section 3.5 defines the type system and soundness and decidability properties. We show how the omitted features can be reintroduced to this formal system in Chapter 7.

3.1 Notation

The formal presentation throughout the rest of this document makes frequent use of lists, sets, and simple operations on those structures. For consistency as well as legibility, we define here a number of notational conventions we will use.

Notation 3.1 (Algebraic Notation)**.** Using g to represent an arbitrary grammar construct, this document observes the following notational conventions:

- We let $[g_1, \ldots, g_n]$ denote an n-ary list of g. We often use the equivalent shorthand $\overset{n}{\overrightarrow{g}}$. We elide the n when it is unnecessary, as in $\overrightarrow{g}$.
- We use the operator $||$ for list concatenation, as in $\overrightarrow{g'} \,||\, \overrightarrow{g''}$.

- We write e.g. $[g|g \in Q]$ for list comprehensions. In such a comprehension, the ordering of elements is nondeterministic.
- For sets, we use the notation $\{g_1, \ldots, g_n\}$ (similar to lists but with brace delimiters). The shorthand ${}^{n}\overbracket{g}$ abbreviates this (for some arbitrary ordering of the set). Again, we elide the n when unnecessary, as in $\overbracket{g}$.
- We sometimes use $\square$ to indicate index positions for clarity. For example, ${}^{n}\overrightarrow{g_\square/g'}$ is shorthand for $[g_1/g', \ldots, g_n/g']$.

3.2 A-Normalized Form

We use a constraint-based type theory to define the type system of TinyBang because such theories are especially applicable to structural subtyping. In this theory, constraints can be viewed as simple statements of fact regarding points in the program (e.g. "at this point, the type `‘A ()` may appear"). Using this approach to typecheck a nested grammar such as the one appearing in Chapter 2 is inconvenient and complicates the formalization.

In order to avoid these complications, we convert TinyBang into A-normal form. In this form, a TinyBang program is a sequence of clauses and each clause represents a single unit of work (such as constructing a value or calling a function). This allows the form of the TinyBang program to correspond exactly with the form of the constraints used in the type system. We use this correspondence in Chapter 4 to prove the soundness of the type system by *simulation* rather than subject reduction.

The grammar of our A-normalized language appears in Figure 3.1. As mentioned in Chapter 2, we refer to the language presented in that chapter as *nested* TinyBang; the language presented here is *ANF* TinyBang. Throughout this chapter, we use the unqualified term TinyBang to refer to ANF TinyBang (as the nested form is less relevant to the formalization).

Expressions in ANF TinyBang have been reduced from their usual deep form into an ordered list of simple clauses. Because clauses are not deep – they consist of a simple constructor over variables or values – each clause represents a single step of evaluation. Patterns have likewise been reduced to a simple collection of instructions, although the filter rules which comprise patterns are unordered. Patterns are expressed as a pattern variable and a set of decomposition rules; pattern matching starts at this variable to determine how to decompose the function's argument.

$$\begin{array}{rcll}
e & ::= & \vec{s} & \textit{expressions} \\
s & ::= & x = v \mid x = x \mid x = x\ x & \textit{clauses} \\
v & ::= & \texttt{()} \mid l\ x \mid x\ \texttt{\&}\ x \mid p\ \texttt{->}\ e & \textit{values} \\
p & ::= & x \backslash F & \textit{patterns} \\
F & ::= & \overset{\sqcup}{f} & \textit{filter rule sets} \\
f & ::= & x = \varphi & \textit{filter rules} \\
\varphi & ::= & \texttt{()} \mid l\ x \mid x * x & \textit{filters}
\end{array}$$

Figure 3.1: TinyBang ANF Grammar

Expressions

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

Patterns

$$\begin{array}{rcl}
\llparenthesis \texttt{()} \rrparenthesis_x & = & \{x = \texttt{()}\} \\
\llparenthesis l\ p \rrparenthesis_x & = & \llparenthesis p \rrparenthesis_y \cup \{x = l\ y\} \\
\llparenthesis p_1 * p_2 \rrparenthesis_x & = & \llparenthesis p_1 \rrparenthesis_y \cup \llparenthesis p_2 \rrparenthesis_z \cup \{x = y * z\} \\
\llparenthesis x_2 \rrparenthesis_{x_1} & = & \{x_1 = y * x_2, y = \texttt{()}, x_2 = \texttt{()}\}
\end{array}$$

Figure 3.2: TinyBang A-Translation

3.2.1 A-Translation

We define the A-translation function $\llparenthesis e \rrparenthesis_x$ in Figure 3.2. This function accepts a nested TinyBang expression and produces the A-normalized form $\vec{s}$ in which the final declared variable is x. We overload this notation to produce a set of filter rules for a given pattern. We use y and z to range over fresh variables unique to that invocation of the translation function.

For instance, consider the nested TinyBang expression `‘A ‘B ()`. We have from Figure 3.2 that $\llparenthesis \texttt{‘A ‘B ()} \rrparenthesis_x = \llparenthesis \texttt{‘B ()} \rrparenthesis_{y_1} \mid\mid [x = \texttt{‘A}\ y_1]$. This in turn reduces to $\llparenthesis \texttt{()} \rrparenthesis_{y_2} \mid\mid [y_1 = \texttt{‘B}\,y_2] \mid\mid [x = \texttt{‘A}\,y_1]$, so the final ANF TinyBang expression is $[y_2 = \texttt{()}, y_1 = \texttt{‘B}\,y_2, x = \texttt{‘A}\,y_1]$. Because the label form of the A-translation was used twice, we use distinct variables y_1 and y_2 in those cases. The A-translated expression lists its clauses in the same order that a depth-first left-to-right evaluation of a nested expression would evaluate.

3.3 Well-Formed Expressions

Given the A-normalized TinyBang grammar, the operational semantics and type system below require a notion of *well-formedness* over expressions; we define this notion here. Briefly, we require that expressions be closed, that variable names be uniquely defined, and that patterns be well-founded. Any closed nested TinyBang expression can be α-renamed such that its A-translation satisfies these properties; we formalize them here to ensure a thorough treatment in our definitions and proofs. For this reason, some readers may wish to skim this section, continue on to the operational semantics in Section 3.4, and then return here when the definitions in this section become relevant.

3.3.1 Pattern Notation

The syntax of patterns in the ANF language is very flexible due to the fact that a variable name identifies each point in a given pattern. That said, the proliferation of these variables clutters notation. To reduce the effect of variable clutter, we introduce the following notational sugar to be used in the following discussion and throughout the rest of the document. We note that this notational sugar does *not* apply to Definition 3.6 above.

Notation 3.2 (Pattern Root Sugar)**.** For legibility, we sometimes write $\varphi\backslash F$ to mean $x\backslash F$ such that $x = \varphi \in F$.

Notation 3.3 (Pattern Construction Sugar)**.** For legibility, we sometimes write $l\ P$ to mean $\{l\ x\backslash F \mid x\backslash F \in P\}$. Similar notation applies for other pattern constructors, taking a Cartesian product in cases where a pattern constructor has more than one type variable argument.

3.3.2 Pattern Well-Formedness

In defining well-formedness on expressions, it is useful to have a concept of well-formedness on patterns. Due to the flexibility of TinyBang's pattern grammar, it is possible to express non-sensical patterns such as those containing multiple bindings for the same variable (e.g. $x\backslash\{x = \texttt{()}\,, x = \texttt{`A}\ x', x' = \texttt{()}\}$) or patterns which are cyclic (e.g. $x\backslash\{x = x\}$. These patterns are not meaningful in matching or destructing data; pattern well-formedness rules them out. We begin by giving terminology to the variables appearing within pattern clauses.

Definition 3.4 (Filter Rule Variable Appearances)**.** A variable x *appears on the left of* a filter rule f iff $f = x = \varphi$. A variable x *appears on the right of* a filter rule f iff $f = x' = \varphi$ and φ is one of $l\ x$, $x * x''$, or $x'' * x$.

Using the above definition, we can provide a concept of a well-formed pattern:

Definition 3.5 (Pattern Well-Formedness)**.** A pattern $x \backslash F$ is *ill-formed* iff:

- There exist two filter rules $f_1 \neq f_2$ such that $\{f_1, f_2\} \subseteq F$, x' appears on the left of f_1, and x' appears on the left of f_2; or
- There exists some $\overset{n}{\overbrace{f}} \subseteq F$ such that x' appears on the left of f_1 iff x' appears on the right of f_n and, for each $i \in \{2..n\}$, x' appears on the left of f_i iff x' appears on the right of f_{i-1}; or
- There exists some $f \in F$ such that x' appears on the right of f and, for all $f' \in F$, x' does not appear on the left of f'; or
- For all $f \in F$, x does not appear on the left of f.

A pattern which is not ill-formed is *well-formed*.

The first rule above indicates that two filter rules have the same variable on the left-hand side. The second rule above declares ill-formed any pattern which contains a cycle in its constraints. The last two rules exclude patterns which are incomplete (e.g. $x \backslash \emptyset$).

By using subtype constraints to represent patterns, it is quite possible to admit cyclic (i.e. recursive) patterns in TinyBang. Recursive patterns are not novel; for instance, [27] provides an ML-like language with recursive pattern matching on nominal data types. We require acyclicity to simplify the discussion of TinyBang (especially the soundness proof), but we briefly discuss how recursive patterns may be added to the language in Section 11.1.

Unless otherwise mentioned, we assume for the rest of this document that the patterns we discuss are well-formed.

3.3.3 Free and Bound Variables

We also require that a well-formed expression is closed. In order to define what it means for an ANF expression to be closed, we must be precise about when variables are bound. We accomplish this by providing a definition of two functions, BV and FV, which represent the bound and free variables (respectively) of a given expression. We overload these functions to cover the bound and free variables of patterns and clauses and the free variables of values.

Definition 3.6 (Bound and Free Variables)**.** BV and FV are defined as follows:

Expressions

$$\begin{aligned}
\textsc{bv}([]) &= \emptyset & \textsc{fv}([]) &= \emptyset \\
\textsc{bv}({}^{n}\overrightarrow{s}) &= \textsc{bv}({}^{n-1}\overrightarrow{s}) \cup \textsc{bv}(s_n) & \textsc{fv}({}^{n}\overrightarrow{s}) &= \textsc{fv}({}^{n-1}\overrightarrow{s}) \cup (\textsc{fv}(s_n) - \textsc{bv}({}^{n-1}\overrightarrow{s}))
\end{aligned}$$

Clauses

$$\begin{aligned}
\textsc{bv}(x_1 \texttt{=} x_2) &= \{x_1\} & \textsc{fv}(x_1 \texttt{=} x_2) &= \{x_2\} \\
\textsc{bv}(x_1 \texttt{=} x_2\ x_3) &= \{x_1\} & \textsc{fv}(x_1 \texttt{=} x_2\ x_3) &= \{x_2, x_3\} \\
\textsc{bv}(x_1 \texttt{=} v) &= \{x_1\} & \textsc{fv}(x_1 \texttt{=} v) &= \textsc{fv}(v)
\end{aligned}$$

Values

$$\begin{aligned}
\textsc{fv}(\texttt{()}) &= \emptyset \\
\textsc{fv}(l\ x) &= \{x\} \\
\textsc{fv}(x_1 \,\texttt{\&}\, x_2) &= \{x_1, x_2\} \\
\textsc{fv}(p \texttt{->} e) &= \textsc{fv}(e) - \textsc{bv}(p)
\end{aligned}$$

Patterns

$$\begin{aligned}
\textsc{bv}(x\backslash F \cup \{x \texttt{=} \varphi\}) &= \{x\} \cup \textsc{bv}(\varphi\backslash F) & \textsc{fv}(x\backslash F \cup \{x \texttt{=} \varphi\}) &= \textsc{fv}(\varphi\backslash F) - \textsc{bv}(\varphi\backslash F) \\
\textsc{bv}(\texttt{()}\ \backslash F) &= \emptyset & \textsc{fv}(\texttt{()}\ \backslash F) &= \emptyset \\
\textsc{bv}(l\ x\backslash F) &= \textsc{bv}(x\backslash F) & \textsc{fv}(l\ x\backslash F) &= \textsc{fv}(x\backslash F) \\
\textsc{bv}(x_1 \texttt{*} x_2\backslash F) &= \textsc{bv}(x_1\backslash F) \cup \textsc{bv}(x_2\backslash F) & \textsc{fv}(x_1 \texttt{*} x_2\backslash F) &= \textsc{fv}(x_1\backslash F) \cup \textsc{fv}(x_2\backslash F)
\end{aligned}$$

An expression e is *closed* iff $\textsc{fv}(e) = \emptyset$.

Note that the above functions are not well-defined when patterns contain multiple filter rules for the same variable, as in $x\backslash\{x \texttt{=} \texttt{()}\ , x \texttt{=} l\ x', x' \texttt{=} \texttt{()}\}$. As a result, expressions which include such patterns are not closed by this definition.

3.3.4 Well-Formedness

Using the above, we can give a definition for well-formed ANF expressions:

Definition 3.7 (Well-Formed Expression)**.** An expression e is *well-formed* iff all of the following are true:

- Each variable is bound in at most one pattern or clause; that is, no two patterns or clauses appearing anywhere within e have a non-empty intersection of bound variable sets.
- e is closed.
- Each pattern p appearing in e is well-formed.

Fortunately, any sensible nested TinyBang expression will be well-formed when it is appropriately α-renamed and A-translated. Throughout the rest of this document,

$$\begin{array}{rcll}
E & ::= & \overrightarrow{x = v} & \textit{environments} \\
P, P^{+}, P^{-}, P^{\mathrlap{\updownarrow}\phi} & ::= & \overbrace{p} & \textit{pattern sets} \\
\mathrlap{\Updownarrow}\phi & ::= & \mathrlap{\uparrow}\phi \mid \mathrlap{\downarrow}\phi & \textit{binding states} \\
\odot & ::= & \bullet \mid \circ & \textit{match result}
\end{array}$$

Figure 3.3: TinyBang Operational Semantics Grammar

we assume we are working with well-formed ANF expressions unless otherwise expressly indicated.

3.4 Operational Semantics

Next, we define a small-step operational semantics for ANF TinyBang. The primary complexity of these semantics is the process of pattern matching, so much of this section focuses on the auxiliary definitions which are eventually used by the operational semantics. In addition to the grammar of ANF TinyBang presented in Figure 3.1, we also require some non-terminals used exclusively in the operational semantics' algebra; we give this extra notation in Figure 3.3.

It is helpful to recall that our soundness proof strategy is to establish a simulation between the operational semantics and the type system (Section 3.2). In some of the supporting definitions below, we choose relational forms that will be particularly amenable to alignment with the corresponding type system relations we will define in Section 3.5.

3.4.1 Compatibility

The first auxiliary relation we define is *compatibility*, which is at the heart of the pattern matching process. We must be able to determine whether an argument matches some patterns and fails to match others. In particular, match failures are important due to the manner in which dispatch is handled on compound functions: an argument is only dispatched to the second function in a compound function if it matches the second pattern *and* fails to match the first.

Although the process of pattern matching is fairly intuitive, formally defining it requires care. In addition to determining whether or not a value matches a pattern, we must be able to obtain the bindings which occur as a result of the match. Onions require special handling as they are pattern matched in a non-trivial way; a similar statement is true about conjunction patterns. Finally, in order to simplify the soundness proof in Chapter 4, we need our proofs to express that a given value simultaneously matches a *set* of patterns. All of these steps introduce (largely orthogonal) complexity to the relation.

Empty Onion Pattern

$$\frac{}{x\backslash E \overset{1}{\sim} \texttt{()}\,\backslash F}$$

Conjunction Pattern

$$\frac{x\backslash E \overset{1}{\sim} x'_1\backslash F \qquad x\backslash E \overset{1}{\sim} x'_2\backslash F}{x\backslash E \overset{1}{\sim} x'_1 * x'_2\backslash F}$$

Onion

$$\frac{x_0 = x_1 \mathbin{\&} x_2 \in E \qquad x_1\backslash E \overset{1}{\sim} p \vee x_2\backslash E \overset{1}{\sim} p}{x_0\backslash E \overset{1}{\sim} p}$$

Label

$$\frac{x_0 = l\ x_1 \in E \qquad x_1\backslash E \overset{1}{\sim} x'\backslash F}{x_0\backslash E \overset{1}{\sim} l\ x'\backslash F}$$

Figure 3.4: Value Compatibility (Revision 1)

This section presents our compatibility relation by starting from the definition of a relatively simple (but lacking) relation and incrementally adding the properties we require. We show the relation after each modification and discuss the changes. The finished relation appears in Section 3.4.1.5.

3.4.1.1 An Initial, Intuitive Definition

At its core, our compatibility relation must answer a simple question: does this value match that pattern? We define our first revision of the compatibility relation to model this question, writing the relation as $x\backslash E \overset{1}{\sim} p$. We use the terminal symbol $\overset{1}{\sim}$ to indicate that this is the first revision of our compatibility relation. The definition of this relation is fairly intuitive: the relation holds if a proof exists in the proof system shown in Figure 3.4. Recall Notation 3.2 in reading this figure: $\texttt{()}\,\backslash F$ is a synonym for $x\backslash F \cup \{x = \texttt{()}\}$ for some x.

The rationale behind each of these rules is quite simple. A conjunction pattern matches a value if each of its conjuncts does. An pattern matches an onion if it matches either side. Label patterns match only those values which have the same label.

While appealingly simple, this relation lacks several properties which we will require to use it effectively. The following sections will augment and modify this relation incrementally.

3.4.1.2 Explicit Failure

One weakness of our first revision of compatibility is that it does not explicitly acknowledge failure. As a result, it would be difficult to use that relation to formally define the behavior of compound functions. To address this issue, we modify compatibility to admit constructive proofs of compatibility failure as well.

The definition of explicit failure requires careful handling of some patterns and

$$\textsc{Empty Onion Pattern}\quad \dfrac{}{x\backslash E \overset{2}{\precsim}_{\bullet} \texttt{()} \backslash F} \qquad \textsc{Conjunction Pattern}\quad \dfrac{x\backslash E \overset{2}{\precsim}_{\odot_1} x_1'\backslash F \qquad x\backslash E \overset{2}{\precsim}_{\odot_2} x_2'\backslash F}{x\backslash E \overset{2}{\precsim}_{\min(\odot_1,\odot_2)} x_1' \,\texttt{*}\, x_2'\backslash F}$$

$$\textsc{Empty Onion}\quad \dfrac{\textsc{CtrPat}(p) \qquad x \,\texttt{=}\, \texttt{()} \in E}{x\backslash E \overset{2}{\precsim}_{\circ} p} \qquad \textsc{Function}\quad \dfrac{\textsc{CtrPat}(p) \qquad x \,\texttt{=}\, (p' \,\texttt{->}\, e') \in E}{x\backslash E \overset{2}{\precsim}_{\circ} p}$$

$$\textsc{Onion}\quad \dfrac{\textsc{CtrPat}(p) \qquad x_0 \,\texttt{=}\, x_1 \,\texttt{\&}\, x_2 \in E \qquad x_1\backslash E \overset{2}{\precsim}_{\odot_1} p \qquad x_2\backslash E \overset{2}{\precsim}_{\odot_2} p}{x_0\backslash E \overset{2}{\precsim}_{\max(\odot_1,\odot_2)} p}$$

$$\textsc{Label}\quad \dfrac{x_0 \,\texttt{=}\, l\ x_1 \in E \qquad x_1\backslash E \overset{2}{\precsim}_{\odot} x'\backslash F}{x_0\backslash E \overset{2}{\precsim}_{\odot} l\ x'\backslash F} \qquad \textsc{Label Failure}\quad \dfrac{x_0 \,\texttt{=}\, v \in E \qquad v \text{ of the form } l'\ x_1 \text{ only if } l \neq l'}{x_0\backslash E \overset{2}{\precsim}_{\circ} l\ x'\backslash F}$$

Figure 3.5: Value Compatibility (Revision 2)

values. The empty onion pattern must match the empty onion value, but that value must match nothing else. To aid in these definitions, it is helpful to distinguish between *constructor patterns* (which are immediate eliminators of value constructors) and *non-constructor patterns* (such as conjunction patterns, which do not eliminate value constructors directly). Formally, we define that function as follows:

Definition 3.8 (Constructor Pattern)**.** We define *constructor patterns* to be those patterns which immediately match a particular data constructor. A constructor pattern is a pattern for which the following predicate, Ctrpat, returns true.

$$\textsc{CtrPat}(p) = \begin{cases} \text{true} & p = l\ x\backslash F \\ \text{false} & \text{otherwise} \end{cases} \qquad\qquad \textsc{CtrPat}(P) = \bigwedge\{\textsc{CtrPat}(p) \mid p \in P\}$$

A pattern which is not a constructor pattern is a *non-constructor pattern.*

In a language with primitive types (e.g. $\mathbb{Z}$), the corresponding simple patterns (e.g. `int`) would also be constructor patterns.

With this in mind, we add a place to the relation which is $\bullet$ when compatibility holds and $\circ$ when it doesn't. We also define a simple ordering – $\circ < \bullet$ – to simplify our presentation. The second revision of our compatibility relation is written $x\backslash E \overset{2}{\precsim}_{\odot} p$ and is taken as the fixed point of the rules appearing in Figure 3.5.

Note the new Empty Onion and Function rules, which indicate explicit failure for the two value forms with no corresponding pattern. The only pattern which matches these values is the empty onion pattern, which is not a constructor pattern (and thus is handled by the Empty Onion Pattern rule instead). A similar failure case exists for label values, which only match patterns with precisely the same label.

The use of the constructor pattern predicate is also important in the Onion rule, where it is used to ensure that conjunctions are separated before onions. Consider a case in which the nested pattern `‘A () * ‘B ()` is matched against the nested value `‘A () & ‘B ()`. If the Onion rule could be used here, we could prove a match failure; to do so, we would need only to show e.g. that `‘A ()` is incompatible with `‘A () * ‘B ()` (which is true, since it has no `‘B` component). By preventing the Onion rule from acting on conjunction patterns, we force the use of the Conjunction Pattern rule, splitting the question of compatibility into two cases: whether `‘A ()` matches the value and whether `‘B ()` does.

3.4.1.3 Value Selection

The second compatibility revision frequently includes predicates of the form $x = v \in E$. This duplication becomes problematic when attempting to add a notion of binding to compatibility, as there are specific cases in which the rule selecting the value cannot make all of the decisions about binding it. The replication of these predicates also becomes problematic for alignment with the type system, since the process of type selection is more complicated than simply selecting a definition from the environment. In preparation for adding binding to compatibility, we refactor the value selection process into a separate, common rule. This means that, in a given rule, the selection of a value may or may not have happened.

We formally model this refactoring as *two* relations, mutually defined, of the forms $x \backslash E \overset{3}{\sim}_{\odot} p$ and $v \backslash E \overset{3}{\sim}_{\odot} p$. We define these relations to hold if a proof exists in the system appearing in Figure 3.6.

In addition to preparing the relation for the addition of bindings, this adjustment also helps to simplify some of the other rules.

3.4.1.4 Bindings

The various revisions of the compatibility relation defined thus far do not provide variable bindings. When we apply a function to an argument, we must not only know *if* the pattern matches but, if so, how the argument binds to the variables appearing within the pattern. We represent those bindings as an additional environment as each binding is of the form $x = v$.

$$\textsc{Value Selection}\quad \frac{x = v \in E \qquad v\backslash E \overset{3}{\sim}_{\odot} p}{x\backslash E \overset{3}{\sim}_{\odot} p} \qquad \textsc{Empty Onion Pattern}\quad \frac{}{v\backslash E \overset{3}{\sim}_{\bullet} \texttt{()}\backslash F} \qquad \textsc{Conjunction Pattern}\quad \frac{v\backslash E \overset{3}{\sim}_{\odot_1} x_1'\backslash F \qquad v\backslash E \overset{3}{\sim}_{\odot_2} x_2'\backslash F}{v\backslash E \overset{3}{\sim}_{\min(\odot_1,\odot_2)} x_1' \,\texttt{*}\, x_2'\backslash F}$$

$$\textsc{Empty Onion}\quad \frac{\textsc{CtrPat}(p)}{\texttt{()}\backslash E \overset{3}{\sim}_{\bigcirc} p} \qquad \textsc{Function}\quad \frac{\textsc{CtrPat}(p)}{(p' \texttt{->} e')\backslash E \overset{3}{\sim}_{\bigcirc} p} \qquad \textsc{Onion}\quad \frac{\textsc{CtrPat}(p) \qquad x_1\backslash E \overset{3}{\sim}_{\odot_1} p \qquad x_2\backslash E \overset{3}{\sim}_{\odot_2} p}{x_1 \,\texttt{\&}\, x_2\backslash E \overset{3}{\sim}_{\max(\odot_1,\odot_2)} p}$$

$$\textsc{Label}\quad \frac{x\backslash E \overset{3}{\sim}_{\odot} x'\backslash F}{l\ x\backslash E \overset{3}{\sim}_{\odot} l\ x'\backslash F} \qquad \textsc{Label Failure}\quad \frac{v \text{ of the form } l'\ x \text{ only if } l \neq l'}{v\backslash E \overset{3}{\sim}_{\bigcirc} l\ x'\backslash F}$$

Figure 3.6: Value Compatibility (Revision 3)

This binding task is somewhat more complex than it first appears. A naïve approach to matching would be to add to the bindings at each proof step, but this can lead to multiple bindings for the same variable. Consider the pattern `p \ [p = ‘A q, q = ()]`. When matching the nested value `‘A () & ‘B ()` to this pattern, we expect the entire value to be bound to the variable `p`; after all, `p` is the top of the pattern. Since the pattern `p \ [p = ‘A q, q = ()]` appears in uses of both the Label rule and the Onion rule, simply adding to the bindings in both of those rules will result in multiple bindings for `p`. The desired behavior is for the *largest* of those values to be bound; that is, we want to bind the pattern variable as near to the root of the proof tree as possible.

We solve this problem by adding another place to the relation. The binding state symbol $\updownarrow$ ranges over $\downarrow$ (which indicates that all binding is complete) and $\uparrow$ (which indicates that the current pattern has not been bound). We can then force specific binding states in certain rules. We define a new revision to our compatibility relation, written $x\backslash E \overset{4}{\sim}_{\odot} p \updownarrow E'$ and $v\backslash E \overset{4}{\sim}_{\odot} p \updownarrow E'$, to hold when a proof exists in the system appearing in Figure 3.7.

The addition of bindings to the compatibility relation is a pervasive change and interacts with the existing rules in several interesting ways. First and foremost, we observe how the binding state position ($\updownarrow$) ensures that each pattern variable is bound exactly once. The only rule which transitions a proof tree from the binding-needed state ($\uparrow$) to the binding-complete state ($\downarrow$) is the Binding rule. The only way to transition back is to build upon the pattern, which causes a different pattern variable to be used. Because the Onion rule can only use premises which are in the binding-needed state, the Onion rule can only act on patterns which have not yet been bound.

Each of the failure rules (e.g. Empty Onion or Label Failure) produce an empty binding list; since bindings are meaningless when compatibility fails, this choice is arbitrary.

$$\textsc{Value Selection}\quad \frac{x = v \in E \qquad v\backslash E \overset{4}{\sim}_{\odot} p \updownarrow E'}{x\backslash E \overset{4}{\sim}_{\odot} p \updownarrow E'}$$

$$\textsc{Binding}\quad \frac{v\backslash E \overset{4}{\sim}_{\odot} x\backslash F \uparrow E'}{v\backslash E \overset{4}{\sim}_{\odot} x\backslash F \downarrow E' \,||\, [x = v]}$$

$$\textsc{Empty Onion Pattern}\quad \frac{}{v\backslash E \overset{4}{\sim}_{\bullet} \texttt{()}\, \backslash F \uparrow []}$$

$$\textsc{Conjunction Pattern}\quad \frac{v\backslash E \overset{4}{\sim}_{\odot_1} x_1'\backslash F \downarrow E_1' \qquad v\backslash E \overset{4}{\sim}_{\odot_2} x_2'\backslash F \downarrow E_2'}{v\backslash E \overset{4}{\sim}_{\min(\odot_1,\odot_2)} x_1' \,\texttt{*}\, x_2'\backslash F \uparrow E_1' \,||\, E_2'}$$

$$\textsc{Empty Onion}\quad \frac{\textsc{CtrPat}(p)}{\texttt{()}\,\backslash E \overset{4}{\sim}_{\bigcirc} p \uparrow []}$$

$$\textsc{Function}\quad \frac{\textsc{CtrPat}(p)}{(p' \texttt{ -> } e')\backslash E \overset{4}{\sim}_{\bigcirc} p \uparrow []}$$

$$\textsc{Onion}\quad \frac{\textsc{CtrPat}(p) \qquad x_1\backslash E \overset{4}{\sim}_{\odot_1} p \uparrow E_1' \qquad x_2\backslash E \overset{4}{\sim}_{\odot_2} p \uparrow E_2' \qquad E_3' = \begin{cases} E_1' & \text{if } \odot_1 = \bullet \\ E_2' & \text{otherwise} \end{cases}}{x_1 \,\texttt{\&}\, x_2\backslash E \overset{4}{\sim}_{\max(\odot_1,\odot_2)} p \uparrow E_3'}$$

$$\textsc{Label}\quad \frac{x\backslash E \overset{4}{\sim}_{\odot} x'\backslash F \downarrow E'}{l\ x\backslash E \overset{4}{\sim}_{\odot} l\ x'\backslash F \uparrow E'}$$

$$\textsc{Label Failure}\quad \frac{v \text{ of the form } l'\ x \text{ only if } l \neq l'}{v\backslash E \overset{4}{\sim}_{\bigcirc} l\ x'\backslash F \uparrow []}$$

Figure 3.7: Value Compatibility (Revision 4)

But the terminal success rule (Empty Onion Pattern) also produces no bindings and the inductive success rules simply pass the bindings along. This is sensible because these rules also yield proofs in the binding-needed state, which ensures that the Binding rule will generate the appropriate bindings. In fact, the only place the bindings are augmented is in the Binding rule, which simply attaches the value currently being considered to the current pattern variable.

One interesting result of bindings is that the Onion rule requires conditional analysis of its results. A pattern matches an onion value if it matches either side. In order to preserve the left precedence semantics of onions, however, we must prefer the left bindings in the event that the pattern matches *both* sides. This way, matching e.g. the nested pattern `int` with the nested value `4 & 5` will deterministically bind the `4`.

3.4.1.5 Simultaneous Matching

The previous revision of the value compatibility relation is sufficient to use in defining the operational semantics of the ANF TinyBang language. Unfortunately, this relation is somewhat difficult to handle in the type system due to a subtle issue of union alignment, which we discuss in Section 3.5.3.5. In order to ensure a direct correspondence with the type system, our compatibility proofs must be able to express multiple simultaneous facts about the value's pattern matching properties; that is, in a single proof, we must be able to indicate that value v matches pattern p but does not match pattern p'. Although this can be expressed in the value system with multiple proofs of the single-pattern variety, each type variable may have multiple lower bounds. This causes a form of non-determinism which is most easily resolved by ensuring that each variable is selected at most once, thus ensuring a consistent view of the type.

To accommodate this type system property, we revise the compatibility relation one final time to be able to express multiple simultaneous matching operations. Instead of a single pattern p, the relation includes three sets of patterns: $P^{\oplus}$ and P^{+} (which are patterns that match the value) and P^{-} (which are patterns that do not). As an additional degree of expressiveness (also motivated by the type system), the distinction between $P^{\oplus}$ and P^{+} is that the patterns in $P^{\oplus}$ will produce bindings whereas the patterns in P^{+} will not. This use of sets of matching and non-matching patterns obviates the need for a $\odot$ position, so we write this value compatibility relation as $x\backslash E \sim \frac{P^{\oplus}/P^{+}}{P^{-}} \Phi\, E'$. We formally define compatibility follows:

Definition 3.9 (Value Compatibility)**.** We mutually define the relations $v\backslash E \sim \frac{P^{\oplus}/P^{+}}{P^{-}} \Phi\, E'$ and $x\backslash E \sim \frac{P^{\oplus}/P^{+}}{P^{-}} \Phi\, E'$ to hold when a proof exists in the proof system appearing in Figure 3.8.

The final value compatibility relation looks considerably more complicated than the previous revisions, but this apparent complexity is almost entirely the result of the set operations introduced by the simultaneous matching property. Value Selection, for instance, is identical in form. Binding now binds the root variables of all patterns in $P^{\oplus}$ (rather than binding the root variable of a single pattern as in previous revisions). The Empty Onion rule generalizes over its previous forms: it states that an empty onion value can be shown to fail to match any set of constructor patterns. The Label Failure rule is obsolete; it is expressed as the set P_2^{-} of failing patterns in the Label rule. The only compelling changes appear in the Empty Onion Pattern, Conjunction Pattern, and Onion rules.

Unlike previous revisions, the Empty Onion Pattern rule is not a terminal rule. Essentially, it states that any proof of compatibility can be extended with any positive match on an empty onion pattern. In previous revisions, this rule was axiomatic due to the fact that only one pattern could be considered at a time (and using that rule forced said pattern to be an empty onion pattern). In this final revision, multiple patterns are considered simultaneously and the Empty Onion Pattern is merely providing additional positive matches.

A similar realization arises from the Conjunction Pattern rule. How do we know if a value matches a conjunction pattern? Simply put, the value in question must match both of the conjuncts. If we know, for instance, that a value v matches all of the patterns

Leaf

$$\frac{}{x\backslash E \sim \frac{\emptyset/\emptyset}{\emptyset} \mathrel{\overset{\uparrow}{\underset{\downarrow}{\phi}}} []}$$

Value Selection

$$\frac{x = v \in E \qquad v\backslash E \sim \frac{P^{\phi}/P^{+}}{P^{-}} \mathrel{\overset{\uparrow}{\underset{\downarrow}{\phi}}} E'}{x\backslash E \sim \frac{P^{\phi}/P^{+}}{P^{-}} \mathrel{\overset{\uparrow}{\underset{\downarrow}{\phi}}} E'}$$

Binding

$$\frac{v\backslash E \sim \frac{P^{\phi}/P^{+}}{P^{-}} \mathrel{\overset{\uparrow}{\phi}} E' \qquad E'' = [x = v \mid x\backslash F \in P^{\phi}]}{v\backslash E \sim \frac{P^{\phi}/P^{+}}{P^{-}} \mathrel{\underset{\downarrow}{\phi}} E' \,||\, E''}$$

Empty Onion Pattern

$$\frac{v\backslash E \sim \frac{P_1^{\phi}/P_1^{+}}{P^{-}} \mathrel{\overset{\uparrow}{\phi}} E' \qquad P_2^{\phi} \cup P_2^{+} = \{\,()\,\backslash F\}}{v\backslash E \sim \frac{P_1^{\phi} \cup P_2^{\phi}/P_1^{+} \cup P_2^{+}}{P^{-}} \mathrel{\overset{\uparrow}{\phi}} E'}$$

Conjunction Pattern

$$\frac{\begin{gathered} v\backslash E \sim \frac{P_1^{\phi}/P_1^{+}}{P_1^{-}} \mathrel{\overset{\uparrow}{\phi}} E' \qquad P_2^{\phi} \subseteq \{x_1' * x_2'\backslash F \mid x_1'\backslash F \in P_1^{\phi} \wedge x_2'\backslash F \in P_1^{\phi}\} \\ P_2^{+} \subseteq \{x_1' * x_2'\backslash F \mid x_1'\backslash F \in P_1^{+} \wedge x_2'\backslash F \in P_1^{+}\} \\ P_2^{-} \subseteq \{x_1' * x_2'\backslash F \mid x_1'\backslash F \in P_1^{-} \vee x_2'\backslash F \in P_1^{-}\} \end{gathered}}{v\backslash E \sim \frac{P_1^{\phi} \cup P_2^{\phi}/P_1^{+} \cup P_2^{+}}{P_1^{-} \cup P_2^{-}} \mathrel{\overset{\uparrow}{\phi}} E'}$$

Conjunction Weakening

$$\frac{\begin{gathered} v\backslash E \sim \frac{P_1^{\phi} \cup P_2^{\phi}/P_1^{+} \cup P_2^{+}}{P_1^{-} \cup P_2^{-}} \mathrel{\underset{\downarrow}{\phi}} E' \\ P_2^{\phi} \subseteq \{x_i\backslash F \mid i \in \{1,2\} \wedge x_1 * x_2\backslash F \in P_1^{\phi}\} \qquad P_2^{+} \subseteq \{x_i\backslash F \mid i \in \{1,2\} \wedge x_1 * x_2\backslash F \in P_1^{+}\} \\ P_2^{-} \subseteq \{x_i\backslash F \mid i \in \{1,2\} \wedge x_1 * x_2\backslash F \in P_1^{-}\} \end{gathered}}{v\backslash E \sim \frac{P_1^{\phi}/P_1^{+}}{P_1^{-}} \mathrel{\underset{\downarrow}{\phi}} E'}$$

Empty Onion

$$\frac{\textsc{CtrPat}(P^{-})}{()\,\backslash E \sim \frac{\emptyset/\emptyset}{P^{-}} \mathrel{\overset{\uparrow}{\phi}} []}$$

Function

$$\frac{\textsc{CtrPat}(P^{-})}{(p \texttt{->} e)\backslash E \sim \frac{\emptyset/\emptyset}{P^{-}} \mathrel{\overset{\uparrow}{\phi}} []}$$

Onion

$$\frac{\begin{gathered} x_1\backslash E \sim \frac{P_1^{\phi}/P_1^{+}}{P^{-} \cup P_2^{\phi} \cup P_2^{+}} \mathrel{\overset{\uparrow}{\phi}} E_1' \\ x_2\backslash E \sim \frac{P_2^{\phi}/P_2^{+}}{P^{-}} \mathrel{\overset{\uparrow}{\phi}} E_2' \qquad \textsc{CtrPat}(P_1^{\phi} \cup P_2^{\phi} \cup P_1^{+} \cup P_2^{+} \cup P^{-}) \end{gathered}}{x_1 \,\&\, x_2\backslash E \sim \frac{P_1^{\phi} \cup P_2^{\phi}/P_1^{+} \cup P_2^{+}}{P^{-}} \mathrel{\overset{\uparrow}{\phi}} E_1' \,||\, E_2'}$$

Label

$$\frac{x\backslash E \sim \frac{P^{\phi}/P^{+}}{P_1^{-}} \mathrel{\underset{\downarrow}{\phi}} E' \qquad \textsc{CtrPat}(P_2^{-}) \qquad \{l\ x'\backslash F \in P_2^{-}\} = \emptyset}{l\ x\backslash E \sim \frac{l\,P^{\phi}/l\,P^{+}}{(l\,P_1^{-}) \cup P_2^{-}} \mathrel{\overset{\uparrow}{\phi}} E'}$$

Figure 3.8: Value Compatibility

appearing in a set P_1^+, then we can prove that v also matches any pattern which is a conjunct between two patterns appearing in P_1^+ (in addition to anything else we already know). This explains the set comprehensions in the premises for both the binding and non-binding pattern matches.

A dual intuition can be used to understand the negative pattern premise: a conjunction pattern fails to match a value if *either* of its conjuncts do. Thus, in the case of P_1^-, the set comprehension includes any conjunction pattern for which either conjunct has been shown not to match. At initial glance, it may appear that this set is infinite – one of the variables appears unconstrained. Recall, however, that $x_1 * x_2 \backslash F$ is notation for $x_0 \backslash F \wedge x_0 = x_1 * x_2 \in F$. Since F is finite, there are finitely many $x_0 = x_1 * x_2$ appearing within it.

As a result of the Conjunction Pattern rule, child patterns may appear in a set of patterns when a conjunction would suffice; that is, if a conjunction pattern appears in P^+, then know that its conjuncts do as well. As it is often necessary to make a weaker statement (asserting only the match of the conjunction pattern itself), the Conjunction Weakening rule permits us to discard any knowledge of success or failure in pattern matching conjuncts (so long as we maintain knowledge of the conjunction itself). It should be noted that this is only possible in the binding-complete ($\bigcirc\mkern-14mu\downarrow$) state; it's never possible to discard a pattern before its variables have been bound. This is the primary reason for a separate rule; if we were to discard these patterns in the Conjunction Pattern rule, the conjuncts' variables may not be bound properly.

The particulars of bindings are somewhat subtle in this relation. In the previous revision of the Onion rule, for example, needed case analysis to determine which side of the onion was matched by the pattern so that the correct bindings could be provided. Here, the Onion rule accomplishes this same task using pattern sets. We insist that the set of patterns $P_2^{\bigcirc\mkern-14mu\uparrow} \cup P_2^+$ that we claim match the right side of the onion must *not* match the left side of the onion; this way, the bindings E_2' cannot define any variables which are also defined by E_1'. More importantly, preferring the bindings obtained from matching the left side of the onion preserves the left precedence semantics described in Chapter 2.

We also observe that the Conjunction Pattern and Empty Onion Pattern rules only accept and produce proofs in the bind-needed state. This is in contrast to the previous revision, in which these rules produced binding-complete ($\bigcirc\mkern-14mu\downarrow$) proofs. In the previous revision, each pattern was bound by an independent use of the Binding rule; in this relation, the Binding rule binds numerous patterns simultaneously. By ensuring that Conjunction Pattern and Empty Onion Pattern do not accept $\bigcirc\mkern-14mu\downarrow$ proofs, we prevent duplicate binding of patterns. The only rule which accepts $\bigcirc\mkern-14mu\uparrow$ and produces $\bigcirc\mkern-14mu\downarrow$ is the Binding rule; the only rule which accepts $\bigcirc\mkern-14mu\downarrow$ and produces $\bigcirc\mkern-14mu\uparrow$ is the Label rule. In the Label rule, every root pattern variable is replaced by a new variable; this ensures that no two uses of the binding rule bind the same pattern variable in a well-formed set of patterns.

3.4.1.6 Example

In order to demonstrate the behavior of the compatibility relation, we present in this section an example based on the following nested TinyBang code:

```
let f = (‘Z () -> ())
      & (‘A () * ‘B x -> x)
in f (‘A () & ‘B ‘C ())
```

In particular, we focus on the compatibility proof necessary to demonstrate that the argument matches the second pattern in the compound function `f`. In this example, we use variables of the form $\texttt{p}n$ as the fresh pattern variables chosen by A-translation and variables of the form $\texttt{y}n$ as the fresh value variables. Thus, the A-translation of the first pattern is

$$p_1 = x'_1 \backslash F_1 = \texttt{p1}\backslash \left\{\begin{array}{l}\texttt{p1 = ‘Z p2,}\\ \texttt{p2 = ()}\end{array}\right\}$$

The A-translation of the second pattern is

$$p_2 = x'_2 \backslash F_2 = \texttt{p3}\backslash \left\{\begin{array}{l}\texttt{p3 = p4 * p6,}\\ \texttt{p4 = ‘A p5,}\\ \texttt{p5 = (),}\\ \texttt{p6 = ‘B x,}\\ \texttt{x = ()}\end{array}\right\}$$

Finally, the A-translation of the argument is

$$E = \left[\begin{array}{l}\texttt{y1 = (),}\\ \texttt{y2 = ‘A y1,}\\ \texttt{y3 = (),}\\ \texttt{y4 = ‘C y3,}\\ \texttt{y5 = ‘B y4,}\\ \texttt{y6 = y2 \& y5}\end{array}\right]$$

for the variable `y6`.

According to the informal semantics from Chapter 2, the second branch of the compound function should be applied as long as (1) the argument matches the second pattern and (2) the argument does *not* match the first pattern. Upon matching, the contents of the `‘B` label – the value `‘C ()`, represented by the variable `y4` in the ANF – should be bound to `x`. Let $E' \ni$ [`x = ‘C y3`]; then this is to say that we expect to prove `y6`$\backslash E \sim \frac{\{p_2\}/\emptyset}{\{p_1\}} \Phi\!\!\downarrow$ E' (where the only other members of E' bind variables of the form `y`n).

We give a natural deduction proof of this expectation in Figure 3.9. For readability, we take several notational shortcuts in this diagram:

- We elide variable definitions and logical assertions in some places. For instance, we do not show the Label rule premise $\{l\ x'\backslash F \in P_2^-\} = \emptyset$ or the definition of the set comprehensions (e.g. P_2^{Φ}) in the Conjunction Pattern rule.
- We write ... in the list of bindings for all bindings for pattern variables of the form `p`n (since they are unreachable by the nested form of the program).
- We write CtrPat(...) when the set of patterns asserted as constructor patterns is evident from the rule being used.
- We use ... in the negative pattern set for conjunction patterns to stand in for the set of all conjunction patterns with `p1` as one of its conjunctions (to avoid writing the set comprehension).

3.4.2 Matching

The previous section defined a compatibility relation which can be used to demonstrate that a value matches and/or does not match given sets of patterns. In this section, we provide the next supporting relation for the operational semantics, the *application matching* (or just *matching*) relation. This relation is used to match the appropriate part of a compound function with a given argument based on which pattern the argument matches. This relation is much simpler than compatibility, so we give it in a single figure below (rather than the incremental approach used in the previous subsection).

In TinyBang, recall that there is no `case`/`match` syntax for pattern matching; individual pattern clauses *pattern* `->` *body* are expressed as simple functions p `->` e and a series of pattern clauses are expressed by onioning together simple functions. We thus define our application matching relation to determine which of an onion x_0 of simple functions can be applied to a given argument x_1 (or if none of them can). Like compatibility, this relation includes a notion of constructive failure. It is more similar to the earlier revisions of compatibility, however, in that it only considers one compound function and one argument at a time.

We write the matching relation as $x_0\ x_1\ {}^{P_1}_{E}\!\!\rightsquigarrow^{P_2}_{\odot}\ e\ @\ E'$. As stated above, x_0 and x_1 are the function and the argument, respectively, which are defined in the environment E.

$$\text{\textsc{Value Selection}}\;\dfrac{\text{\textsc{Label}}\;\dfrac{\text{\textsc{Value Selection}}\;\dfrac{\texttt{y4 = `C y3} \in E \qquad \text{\textsc{Binding}}\;\dfrac{\text{\textsc{Empty Onion Pattern}}\;\dfrac{\text{\textsc{Label}}\;\dfrac{\text{\textsc{Leaf}}\;\dfrac{}{\texttt{y3}\backslash E \sim \frac{\emptyset/\emptyset}{\emptyset} \phi^{\uparrow} []} \qquad \text{\textsc{CtrPat}}(\emptyset)}{\texttt{`C y3}\backslash E \sim \frac{\emptyset/\emptyset}{\emptyset} \phi^{\uparrow} []}}{\texttt{`C y3}\backslash E \sim \frac{\{\texttt{x}\backslash F_2\}/\emptyset}{\emptyset} \phi^{\uparrow} []}}{\texttt{`C y3}\backslash E \sim \frac{\{\texttt{x}\backslash F_2\}/\emptyset}{\emptyset} \phi^{\downarrow} [\texttt{x = `C y3}]}}{\texttt{y4}\backslash E \sim \frac{\{\texttt{x}\backslash F_2\}/\emptyset}{\emptyset} \phi^{\downarrow} [\texttt{x = `C y3}]} \qquad \text{\textsc{CtrPat}}(\ldots)}{\texttt{`B y4}\backslash E \sim \frac{\{\texttt{p6}\backslash F_2\}/\emptyset}{\{\texttt{p1}\backslash F_1\}} \phi^{\uparrow} [\texttt{x = `C y3}]}}{\texttt{y5}\backslash E \sim \frac{\{\texttt{p6}\backslash F_2\}/\emptyset}{\{\texttt{p1}\backslash F_1\}} \phi^{\uparrow} [\texttt{x = `C y3}]}$$

$$\text{\textsc{Value Selection}}\;\dfrac{\texttt{y6 = y2 \& y5} \in E \qquad \text{\textsc{Conjunction Weakening}}\;\dfrac{\text{\textsc{Binding}}\;\dfrac{\text{\textsc{Conjunction}}\;\dfrac{\text{\textsc{Onion}}\;\dfrac{\text{\textsc{Value Selection}}\;\dfrac{\text{\textsc{Label}}\;\dfrac{\text{\textsc{Value Selection}}\;\dfrac{\texttt{y1 = ()} \in E \qquad \text{\textsc{Binding}}\;\dfrac{\text{\textsc{Empty Onion Pattern}}\;\dfrac{\text{\textsc{Empty Onion}}\;\dfrac{\text{\textsc{CtrPat}}(\emptyset)}{\texttt{()}\,\backslash E \sim \frac{\emptyset/\emptyset}{\emptyset} \phi^{\uparrow} []}}{\texttt{()}\,\backslash E \sim \frac{\{\texttt{p5}\backslash F_2\}/\emptyset}{\emptyset} \phi^{\uparrow} []}}{\texttt{()}\,\backslash E \sim \frac{\{\texttt{p5}\backslash F_2\}/\emptyset}{\emptyset} \phi^{\downarrow} [\ldots]}}{\texttt{y1}\backslash E \sim \frac{\{\texttt{p5}\backslash F_2\}/\emptyset}{\emptyset} \phi^{\downarrow} [\ldots]} \qquad \text{\textsc{CtrPat}}(\ldots)}{\texttt{`A y1}\backslash E \sim \frac{\{\texttt{p4}\backslash F_2\}/\emptyset}{\{\texttt{p1}\backslash F_1,\texttt{p6}\backslash F_2\}} \phi^{\uparrow} [\ldots]}}{\texttt{y2}\backslash E \sim \frac{\{\texttt{p4}\backslash F_2\}/\emptyset}{\{\texttt{p1}\backslash F_1,\texttt{p6}\backslash F_2\}} \phi^{\uparrow} [\ldots]} \qquad \vdots \qquad \text{\textsc{CtrPat}}(\ldots)}{\texttt{y2 \& y5}\backslash E \sim \frac{\{\texttt{p4}\backslash F_2,\texttt{p6}\backslash F_2\}/\emptyset}{\{\texttt{p1}\backslash F_1\}} \phi^{\uparrow} [\texttt{x = `C y3},\ldots]}}{\texttt{y2 \& y5}\backslash E \sim \frac{\{\texttt{p3}\backslash F_2,\texttt{p4}\backslash F_2,\texttt{p6}\backslash F_2\}/\emptyset}{\{\texttt{p1}\backslash F_1,\ldots\}} \phi^{\uparrow} [\texttt{x = `C y3},\ldots]}}{\texttt{y2 \& y5}\backslash E \sim \frac{\{\texttt{p3}\backslash F_2,\texttt{p4}\backslash F_2,\texttt{p6}\backslash F_2\}/\emptyset}{\{\texttt{p1}\backslash F_1,\ldots\}} \phi^{\downarrow} [\texttt{x = `C y3},\ldots]}}{\texttt{y2 \& y5}\backslash E \sim \frac{\{\texttt{p3}\backslash F_2\}/\emptyset}{\{\texttt{p1}\backslash F_1\}} \phi^{\downarrow} [\texttt{x = `C y3},\ldots]}}{\texttt{y6}\backslash E \sim \frac{\{\texttt{p3}\backslash F_2\}/\emptyset}{\{\texttt{p1}\backslash F_1\}} \phi^{\downarrow} [\texttt{x = `C y3},\ldots]}$$

Figure 3.9: Example Proof of Value Compatibility

$$\textsc{Function Match}\quad \frac{x_0 = p \texttt{->} e \in E \qquad x_1 \backslash E \sim \frac{\{p\}/\emptyset}{P} \Phi\, E'}{x_0\ x_1\ {}^{P}_{E}\!\rightsquigarrow^{\{p\}}_{\bullet}\ e\ @\ E'}$$

$$\textsc{Function Mismatch}\quad \frac{x_0 = p \texttt{->} e \in E \qquad x_1 \backslash E \sim \frac{\emptyset/\emptyset}{P\cup\{p\}} \Phi\, []}{x_0\ x_1\ {}^{P}_{E}\!\rightsquigarrow^{\{p\}}_{\bigcirc}\ []\ @\ []}$$

$$\textsc{Non-Function}\quad \frac{x_0 = v \in E \qquad v \text{ not of the form } p \texttt{->} e \text{ or } x_1' \,\&\, x_2'}{x_0\ x_1\ {}^{P}_{E}\!\rightsquigarrow^{\emptyset}_{\bigcirc}\ []\ @\ []}$$

$$\textsc{Onion Left}\quad \frac{x_0 = x_2 \,\&\, x_3 \in E \qquad x_2\ x_1\ {}^{P_1}_{E}\!\rightsquigarrow^{P_2}_{\bullet}\ e\ @\ E'}{x_0\ x_1\ {}^{P_1}_{E}\!\rightsquigarrow^{P_2}_{\bullet}\ e\ @\ E'}$$

$$\textsc{Onion Right}\quad \frac{x_0 = x_2 \,\&\, x_3 \in E \qquad x_2\ x_1\ {}^{P_1}_{E}\!\rightsquigarrow^{P_2}_{\bigcirc}\ e_2\ @\ E_2' \qquad x_3\ x_1\ {}^{P_1\cup P_2}_{E}\!\rightsquigarrow^{P_3}_{\odot}\ e_3\ @\ E_3'}{x_0\ x_1\ {}^{P_1}_{E}\!\rightsquigarrow^{P_2\cup P_3}_{\odot}\ e_3\ @\ E_3'}$$

Figure 3.10: Value Application Relation

The $\odot$ position is used to indicate success ($\bullet$) or failure ($\bigcirc$). In the event of success, e is the body of the successfully-matching simple function and E' contains the appropriate bindings which resulted from matching the argument to its pattern. (Otherwise, these positions are of no semantic value.) The pattern set positions P_1 and P_2 are discussed below but, like the positive pattern set P^+ in the compatibility relation, they are present to aid in the alignment with the type system and can be safely ignored when considering only the operational semantics. We discuss the purpose of the pattern sets below.

We define the application compatibility relation formally as follows:

Definition 3.10 (Value Application Matching)**.** We define $x_0\ x_1\ {}^{P_1}_{E}\!\rightsquigarrow^{P_2}_{\odot}\ e\ @\ E'$ to hold when a proof exists in the system appearing in Figure 3.10.

Consider a case in which a single simple function is applied to an argument. If the pattern of the function matches the argument, then the Function Match rule of application matching will apply (with the set of previously-visited patterns P_1 being the empty set): the compatibility premise has $P^{\Phi} = \{p\}$ (where p is the function's pattern) and $P^+ = P^- = \emptyset$. In this case, the e and E' positions are the body of the function and the bindings produced by pattern matching (respectively). If instead the function's pattern does not match the argument, then the Function Mismatch rule applies (since the function's pattern is in the negative pattern set P^-). For any case in which the argument is a simple function, this is exhaustive: the function's pattern either matches the argument or it does not.

If the function position is a compound function, then we have three cases: (1) the left side of the compound function matches the argument, (2) the left side does not match but the right side does, or (3) neither side matches. The first case is trivially handled by the Onion Left rule. The second two cases are addressed by the Onion Right rule: as long as the left side does not match, the result is determined by the right side. This how the left-priority dispatch of compound functions is ensured.

As stated above, the pattern set positions of this relation are not strictly necessary to express the operational semantics; they are present to ease the alignment with the type system. Thinking of the application matching process in a procedural, chronological sense, the purpose of the left pattern set P_1 is to indicate which patterns have failed to match the argument thus far. This is the reason that we selected $P_1 = \emptyset$ in our simple function example above: if no patterns have yet been considered, no patterns have failed to match. The right pattern set P_2 specifies a set of patterns which have been considered by previous compatibility proofs; hence, the only rules which add a new pattern to that set are the Function Match and Function Mismatch rules. (The remaining rules with non-empty P_2 build their failed pattern sets from other application matching proofs.) This place in the application matching relation is redundant – the Onion Right rule ensures that the right side of a compound function is only considered if its left side failed – but it is only redundant because the operational semantics are deterministic. In the type system, union elimination on types is modeled non-deterministically, requiring this left-prioritized exploration to be explicitly represented. By explicitly representing it in the evaluation relation as well, we simplify the soundness proof.

3.4.3 Small Step Evaluation

Given the above definitions of compatibility and matching, we can define the small-step semantics of ANF TinyBang as a relation $e \longrightarrow^1 e'$. Our definition uses an environment-based semantics; it proceeds 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.

The expressions in ANF TinyBang uniquely define each variable. To maintain this invariant throughout evaluation, we must ensure that newly introduced variables have not yet been encountered; in particular, as function bodies are expanded, we must freshen them. We do so using a function $\boldsymbol{\alpha}(x, x') = x''$ which takes a call site x and a function body variable x' and returns a new variable x'' meant to replace x' in the body of the function. We give here a formal notion of this freshness.

Definition 3.11 (Variable Freshening)**.** We define $\boldsymbol{\alpha}(x, x')$ as a function with the following properties:

1. This function is injective.
2. The codomain of this function is disjoint from the set of variables appearing in the initial expression e.
3. There exists a strict partial ordering on variables such that $x < x'$ if x' is in the codomain of $\boldsymbol{\alpha}(x, -)$ (i.e. there are no cycles in any variable's freshening lineage).

Because no call site is visited more than once (as call sites within loops are subjected to variable substitution), this ensures that every new variable is fresh.[1] As we often consider variable substitution for a fixed call site, we introduce the following notational convenience:

Notation 3.12 (Variable Replacement)**.** We use ς to range over variable replacement functions (e.g. $\varsigma(x) = x'$). We also introduce some notational sugar for clarity:

- We overload the application of a replacement function to operate homomorphically over expressions (e.g. $\varsigma(\texttt{x} = \texttt{y}) = \varsigma(\texttt{x}) = \varsigma(\texttt{y})$.
- We write $\varsigma\big|_e$ to indicate the function which replaces all variables bound by e in the same manner as ς and which replaces all other variables with themselves. Taken with the above, we have $\varsigma\big|_{\texttt{x=y}}(\texttt{x=y}) = \varsigma(\texttt{x})\texttt{=y}$ (since `x` is bound by the expression `x=y` but `y` is not).

In addition to the above, we must be able to freshen only the pattern side of variables in the bindings produced by compatibility. To do so, we provide the following definition:

Definition 3.13 (Binding Freshening)**.** We let $\textsc{BFresh}(\varsigma, \overset{n}{\overrightarrow{x_\square = x'_\square}}) = \overset{n}{\overrightarrow{\varsigma(x_\square) = x'_\square}}$.

We also require the ability to identify the last bound variable in an expression. The last variable in a function's body, for instance, contains the result of that function's evaluation. We define a simple function for this purpose:

Definition 3.14 (Return Variable)**.** We let $\textsc{rv}(\overset{n}{\overrightarrow{s}})$ be the variable bound by s_n.

Using this notation, we define the small step relation as follows:

Definition 3.15 (Small Step Operational Semantics)**.** Let $e \longrightarrow^1 e'$ be the relation satisfying the rules given by Figure 3.11. We write $e \not\longrightarrow^1 e'$ if it is not true that $e \longrightarrow^1 e'$.

As stated above, small-step evaluation proceeds by operating on the first unevaluated clause. This is expressed by the use of concatenation with E; for instance, the only expressions of the form $E \,||\, [x_1 = x_2] \,||\, e$ are those expressions for which the first unevaluated clause is $x_1 = x_2$ (because the grammar of E only admits evaluated clauses). The Variable Lookup rule above operates on such clauses, replacing them with evaluated clauses and

[1]Section 4.7.1 contains a formalization of these properties which is used by the type system's proof of soundness.

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

$$\textsc{Application}\quad \frac{x_0\ x_1 \ {}_{E}^{\emptyset}\!\!\rightsquigarrow^{P}_{\bullet}\ e' \mathbin{@} E' \qquad \varsigma = \boldsymbol{\alpha}(x_2, -) \qquad \varsigma' = \varsigma\big|_{E' \,||\, e'} \qquad e'' = \varsigma'(e') \qquad E'' = \textsc{BFresh}(\varsigma', E')}{E \,||\, [x_2 = x_0\ x_1] \,||\, e \longrightarrow^1 E \,||\, E'' \,||\, e'' \,||\, [x_2 = \textsc{rv}(e'')] \,||\, e}$$

Figure 3.11: Small Step Evaluation

otherwise leaving the expression unchanged.

The Application Rule proceeds first by finding a match for the provided argument x_1 in the function x_0. If the application matching relation yields a successful match, e' is the body of the simple function whose pattern matched the argument and E' contains the bindings to its pattern's variables. Both E' and e' are then freshened and inserted into the expression in place of the origin application clause. Finally, the additional clause $x_2 = \textsc{rv}(e'')$ copies the result of the function (stored in its last defined variable) to the variable which was originally assigned from the application.

We can then formally define the process of evaluation as a sequence of steps with the following relation:

Definition 3.16 (Multiple Small Steps)**.** Let $e_0 \longrightarrow^* e_n$ iff $e_0 \longrightarrow^1 e_1 \longrightarrow^1 \ldots \longrightarrow^1 e_n$.

When evaluation is complete, the last expression in the sequence will be of the form E. We observe that $E \not\longrightarrow^1 e$ for any e. In any other case, however, some unevaluated clause cannot be evaluated and the expression is "stuck." We define this term formally as follows:

Definition 3.17 (Stuck)**.** If $e \not\longrightarrow^1 e'$ and e is not of the form E, then e is *stuck*.

In the next section, we present the TinyBang type system, which conservatively attempts to determine whether an expression can become stuck.

3.5 Type System

This section formalizes a type system for ANF TinyBang based on previous work on subtype constraint systems [3, 4, 53, 54, 74]. A subtype constraint system is suitable for typing ANF TinyBang for several reasons. First, previous subtype constraint systems [4, 53] capture control flow information through the use of conditional constraints; this helps in providing the heterogeneous case type behavior described in Section 2.2, although the complexity of the types used in TinyBang prevent these previous approaches from being directly applied. Second, subtype constraint systems have been shown [53] to be suitable for complex pattern matching and record concatenation operations (although those systems use row typing [71] while this type system uses onion types and a weakened form of lower bound intersection). Combining subtype constraints with a suitable polymorphism model [74] has also shown such systems capable of inferring precise types for objects.

The fact that subtype constraints are so suitable for our evaluation system is not simply good fortune. As stated in Section 1.3, we developed the evaluation and type systems concurrently with the objective of keeping them as similar as possible. For instance, the choice of our ANF expression grammar (which represents all in-progress evaluation as a list of shallow clauses rather than with a recursive expression grammar) was made for more direct alignment with our type system (which represents types as a set of shallow constraints rather than a recursive type grammar). In addition to aligning the expression and type grammars, we design the evaluation and type system *relations* to be quite similar as well. We do not take the standard approach of presenting a declarative set of type checking rules and a separate type inference algorithm. Type checking in TinyBang instead proceeds by inferring subtype constraints for a given expression and then deductively closing over those constraints to determine if they are consistent. This constraint closure corresponds to evaluation in the same way that the type grammar corresponds to the expression grammar.

TinyBang's type checking is quite similar to approaches used in the field of abstract interpretation [21]. The connection between abstract interpretation and subtype constraint systems is well-understood ([50] among others). For instance, we begin typechecking by inferring constraints for the expression to be typechecked. This inference function forms the left adjoint of a Galois connection from the expression grammar to the type grammar and the constraint closure process can be viewed as the execution of the abstract interpretation. Unlike abstract interpretations, however, our closure process is monotonic and produces only invariant facts; we discuss this in greater detail when we use this connection to structure the soundness proof in Chapter 4.

Our formal presentation of the type system appears below. We begin by defining the type grammar and the initial alignment function. We then define relations corresponding to those which appeared in the evaluation system; as a result of decisions we made when defining the evaluation relations, the alignment between them and the corresponding type relations is quite intuitive. We then give a formal definition of typechecking in terms of these definitions.

$$
\begin{array}{rcll}
C & ::= & \overbracket{c} & \textit{constraint sets} \\
c & ::= & \tau|_{\Pi^-}^{\Pi^+} <: \alpha \mid \alpha <: \alpha \mid \alpha\ \alpha <: \alpha & \textit{constraints} \\
\tau & ::= & () \mid l\ \alpha \mid \alpha \,\&\, \alpha \mid \pi \rightarrow t & \textit{types} \\
t & ::= & \alpha \backslash C & \textit{constrained types} \\
\Pi, \Pi^+, \Pi^-, \Pi^{\oplus} & ::= & \overbracket{\pi} & \textit{pattern sets} \\
\pi & ::= & \alpha \backslash \Psi & \textit{patterns} \\
\Psi & ::= & \overbracket{\psi} & \textit{filter constraint sets} \\
\psi & ::= & \phi <: \alpha & \textit{filter constraints} \\
\phi & ::= & () \mid l\ \alpha \mid \alpha * \alpha & \textit{filters} \\
\alpha & & & \textit{type variables}
\end{array}
$$

Figure 3.12: Type Grammar

3.5.1 Initial Alignment

Figure 3.12 provides a grammar of the TinyBang type system. The close alignment between the expression and type grammars can be seen here: τ corresponds to v, π corresponds to p, and so forth. In fact, these grammars are more similar than they are different. Of note, however, is that C does *not* directly correspond to e; instead, we represent the type of e with the pair $\alpha\backslash C$ (a constrained type). This is necessary to model the behavior of the RV function: expressions are ordered and so the last clause of an expression (or the variable thereof) is easy to determine, but constraint sets are unordered and so the corresponding type variable must be explicitly represented.

Another difference between the expression grammar and the type grammar arises in value clauses. A value clause in the expression grammar is of the form $x = v$, but the corresponding type system constraint is written $\tau|_{\Pi^-}^{\Pi^+} <: \alpha$. In this case, the type system carries additional information in the form of a restriction which is described as two pattern sets: those which match and those which do not. The meaning of $\tau|_{\Pi^-}^{\Pi^+}$ is the set of values which are members of τ that match all patterns in Π^+ and do not match patterns in Π^-. This is, in essence, a type intersection. Although lower-bounding intersection types are known to introduce challenges in subtype constraint systems [3], the associated problems do not arise because the restriction here is limited to the pattern grammar (which is not as expressive as the type grammar). The process of initial alignment does not introduce restrictions – all lower-bounding constraints have empty pattern sets – so a reader can safely ignore the restriction sets for now.

Given an expression, we need to be able to produce a set of constraints representing it which we can then subject to logical closure. We formalize this initial alignment step as a function $[\![e]\!]_{\mathrm{E}}$ taking an expression and yielding a constrained type $\alpha\backslash C$. To support this

$$
\begin{array}{rcl}
[\![{}^{n}\overrightarrow{s}]\!]_{\mathrm{E}} & = & \alpha_n \backslash {}^{n}\overline{c} \text{ where } \forall i \in \{1..n\}.\, [\![s_i]\!]_{\mathrm{E}} = \alpha_i \backslash c_i \\[1ex]
[\![x_1 \mathtt{=} x_2]\!]_{\mathrm{E}} & = & [\![x_1]\!]_{\mathrm{V}} \backslash [\![x_2]\!]_{\mathrm{V}} <: [\![x_1]\!]_{\mathrm{V}} \\
[\![x_1 \mathtt{=} x_2 \; x_3]\!]_{\mathrm{E}} & = & [\![x_1]\!]_{\mathrm{V}} \backslash [\![x_2]\!]_{\mathrm{V}} \; [\![x_3]\!]_{\mathrm{V}} <: [\![x_1]\!]_{\mathrm{V}} \\
[\![x \mathtt{=} v]\!]_{\mathrm{E}} & = & [\![x]\!]_{\mathrm{V}} \backslash \left([\![v]\!]_{\mathrm{E}} \right)\Big|_{\emptyset}^{\emptyset} <: [\![x]\!]_{\mathrm{V}} \\[1ex]
[\![\mathtt{()}]\!]_{\mathrm{E}} & = & \mathtt{()} \\
[\![l \; x]\!]_{\mathrm{E}} & = & l \; [\![x]\!]_{\mathrm{V}} \\
[\![x_1 \,\mathtt{\&}\, x_2]\!]_{\mathrm{E}} & = & [\![x_1]\!]_{\mathrm{V}} \,\mathtt{\&}\, [\![x_2]\!]_{\mathrm{V}} \\
[\![p \,\mathtt{->}\, e]\!]_{\mathrm{E}} & = & [\![p]\!]_{\mathrm{P}} \rightarrow [\![e]\!]_{\mathrm{E}} \\[1ex]
[\![x \backslash F]\!]_{\mathrm{P}} & = & [\![x]\!]_{\mathrm{V}} \backslash \{ [\![f]\!]_{\mathrm{P}} \mid f \in F \} \\
[\![x \mathtt{=} \varphi]\!]_{\mathrm{P}} & = & [\![\varphi]\!]_{\mathrm{P}} <: [\![x]\!]_{\mathrm{V}} \\[1ex]
[\![\mathtt{()}]\!]_{\mathrm{P}} & = & \mathtt{()} \\
[\![l \; x]\!]_{\mathrm{P}} & = & l \; [\![x]\!]_{\mathrm{V}} \\
[\![x_1 \,\mathtt{*}\, x_2]\!]_{\mathrm{P}} & = & [\![x_1]\!]_{\mathrm{V}} \,\mathtt{*}\, [\![x_2]\!]_{\mathrm{V}}
\end{array}
$$

Figure 3.13: Initial Alignment

alignment, we first define the alignment on the variables themselves:

Definition 3.18 (Initial Alignment Variables)**.** Let $[\![x]\!]_{\mathrm{V}} = \alpha$ be an injective function. Type variables in the codomain of $[\![-]\!]_{\mathrm{V}}$ are termed *initial type variables*.

Given this function, we then define $[\![e]\!]_{\mathrm{E}}$ below. This definition also overloads $[\![-]\!]_{\mathrm{E}}$ to operate on individual clauses and defines a function $[\![-]\!]_{\mathrm{P}}$ which operates on patterns.

Definition 3.19 (Initial Alignment)**.** Let the functions $[\![-]\!]_{\mathrm{E}}$ and $[\![-]\!]_{\mathrm{P}}$ be defined as in Figure 3.13.

The alignment itself is quite simple, but it is worth nothing that functions $p \,\mathtt{->}\, e$ produce types $\alpha\backslash\Psi \rightarrow \alpha'\backslash C$: the entire body of the function is preserved in the form of a constrained type.

3.5.2 Definitions for Alignment

Above, we have defined a type grammar which parallels that of the evaluation system. Below, we will do the same for each relation appearing in the evaluation system, ensuring that the type checking process closely follows the behavior of evaluation. In order to do so, however, we require several definitions to parallel basic statements made about the evaluation system and the well-formedness of expressions. We give those simple definitions below.

3.5.2.1 Bound and Free Variables

In Section 3.3.3, we give a definition of two functions: BV, which determines the variables bound in a given expression, clause, and so on; and FV, which determines the variables free in such an argument. Although it is uncommon to use that terminology in this way with type variables – bound and free type variables are usually determined by explicit quantifiers present on types – we define those parallels here as they pertain to the constraint sets themselves. Just as a clause binds the variable on its left, for instance, a constraint binds the variable appearing on its right. We formally define these functions below. Note that this definition carefully follows the same alignment as the initial alignment function above.

Definition 3.20 (Bound and Free Type Variables)**.** BV and FV are defined on types and constraints as follows:

Constraint Sets

$$\textsc{bv}(\emptyset) = \emptyset \qquad \textsc{fv}(\emptyset) = \emptyset$$

$$\textsc{bv}(C) = \{\textsc{bv}(c) \mid c \in C\} \qquad \textsc{fv}(C) = \{\textsc{fv}(c) \mid c \in C\} - \textsc{bv}(C)$$

Constraints

$$\textsc{bv}(\alpha_1 <: \alpha_2) = \{\alpha_2\} \qquad \textsc{fv}(\alpha_1 <: \alpha_2) = \{\alpha_1\}$$

$$\textsc{bv}(\alpha_1\ \alpha_2 <: \alpha_3) = \{\alpha_3\} \qquad \textsc{fv}(\alpha_1\ \alpha_2 <: \alpha_3) = \{\alpha_1, \alpha_2\}$$

$$\textsc{bv}(\tau\big|^{\Pi^+}_{\Pi^-} <: \alpha_1) = \{\alpha_1\} \qquad \textsc{fv}(\tau\big|^{\Pi^+}_{\Pi^-} <: \alpha_1) = \textsc{fv}(\tau)$$

Types

$$\textsc{fv}(\texttt{()}) = \emptyset$$

$$\textsc{fv}(l\ \alpha) = \{\alpha\}$$

$$\textsc{fv}(\alpha_1 \,\&\, \alpha_2) = \{\alpha_1, \alpha_2\}$$

$$\textsc{fv}(\pi \texttt{->} \alpha\backslash C) = \textsc{fv}(C) - \textsc{bv}(\pi)$$

Pattern Types

$$\textsc{bv}(\alpha\backslash\Psi \cup \{\phi <: \alpha\}) = \{\alpha\} \cup \textsc{bv}(\phi\backslash\Psi) \qquad \textsc{fv}(\alpha\backslash\Psi \cup \{\phi <: \alpha\}) = \textsc{fv}(\phi\backslash\Psi) - \textsc{bv}(\phi\backslash\Psi)$$

$$\textsc{bv}(\texttt{()}\,\backslash\Psi) = \emptyset \qquad \textsc{fv}(\texttt{()}\,\backslash\Psi) = \emptyset$$

$$\textsc{bv}(l\ \alpha\backslash\Psi) = \textsc{bv}(\alpha\backslash\Psi) \qquad \textsc{fv}(l\ \alpha\backslash\Psi) = \textsc{fv}(\alpha\backslash\Psi)$$

$$\textsc{bv}(\alpha_1 * \alpha_2\backslash\Psi) = \textsc{bv}(\alpha_1\backslash\Psi) \cup \textsc{bv}(\alpha_2\backslash\Psi) \qquad \textsc{fv}(\alpha_1 * \alpha_2\backslash\Psi) = \textsc{fv}(\alpha_1\backslash\Psi) \cup \textsc{fv}(\alpha_2\backslash\Psi)$$

As in the evaluation system, the above function is not well-defined when a pattern type contains multiple filter rules for the same type variable. This is intentional; it mirrors the evaluation system and prevents such patterns from being present in (the types of) closed expressions.

3.5.2.2 Type System Patterns

Next, we provide definitions relating to patterns which correspond to some basic definitions used by the evaluation system. Notation 3.2, for instance, describes the meaning of a value appearing in a pattern where a variable is expected. We provide a similar definition for pattern types here; as constrained types have syntax similar to patterns and pattern types, we extend this notation to constrained types as well.

Notation 3.21 (Type Root Sugar)**.** For legibility, we sometimes write $\tau\backslash C$ to mean $\alpha\backslash C$ such that $\tau <: \alpha \in C$. Likewise, we write $\phi\backslash\Psi$ to mean $\alpha\backslash\Psi$ where $\phi <: \alpha \in \Psi$.

We also give the same meaning to the use of a pattern set within a label constructor:

Notation 3.22 (Pattern Type Constructor Sugar)**.** For legibility, we sometimes write $l\ \Pi$ to mean $\{l\ \alpha\backslash\Psi \mid \alpha\backslash\Psi \in \Pi\}$. Similar notation applies for other pattern constructors, taking a Cartesian product in cases where a pattern constructor has more than one type variable argument.

Further, Section 3.4.1.2 gives a notion of a constructor pattern. The type system's patterns correspond directly; this gives rise to the idea of a *constructor pattern type*. This definition is unsurprising and is given as follows:

Definition 3.23 (Constructor Pattern Type)**.** We define *constructor pattern types* to be those pattern types which immediately match a particular data constructor type. We overload the predicate function $\textsc{CtrPat}$ as follows:

$$\textsc{CtrPat}(\pi) = \begin{cases} \text{true} & \pi = l\ \alpha\backslash\Psi \\ \text{false} & \text{otherwise} \end{cases} \qquad\qquad \textsc{CtrPat}(\Pi) = \bigwedge\{\textsc{CtrPat}(\pi) \mid \pi \in \Pi\}$$

As in the evaluation system, this definition of constructor pattern types would include integers, strings, or any other concrete primitive types if they were added to the language.

Finally, Section 3.3.2 defines a notion of pattern well-formedness. We extend that notion to include pattern types with the following definitions:

Definition 3.24 (Filter Constraint Variable Appearances)**.** A type variable α *is the upper bound of* a filter constraint ψ iff $\psi = \phi <: \alpha$. A type variable α *appears in the lower bound of* a filter constraint ψ iff $\psi = \phi <: \alpha'$ and ϕ is one of $l\ \alpha$, $\alpha * \alpha''$, or $\alpha'' * \alpha$.

Definition 3.25 (Pattern Type Well-Formedness)**.** A pattern type $\alpha\backslash\Psi$ is *ill-formed* iff:

- There exist two filter constraints $\psi_1 \neq \psi_2$ such that $\{\psi_1, \psi_2\} \subseteq \Psi$, α' is the upper bound of ψ_1, and α' is the upper bound of ψ_2; or

- There exists some $\overbrace{\psi}^{n}$ $\subseteq \Psi$ such that α' is the upper bound of ψ_1 iff α' appears in the lower bound of ψ_n and, for each $i \in \{2..n\}$, α' is the upper bound of ψ_i iff α' appears in the lower bound of ψ_{i-1}; or
- There exists some $\psi \in \Psi$ such that α' appears in the lower bound of ψ and, for all $\psi' \in \Psi$, α' is not the upper bound of ψ'; or
- For all $\psi \in \Psi$, α is not the upper bound of ψ.

A pattern type which is not ill-formed is *well-formed*.

It is trivial to show that the initial alignment of a well-formed pattern is a well-formed pattern type.

With these definitions, we are ready to proceed to the formalization of type system relations which parallel those in the evaluation system.

3.5.3 Compatibility

In Section 3.4.1, we provided a definition of the evaluation compatibility relation. That relation answers a significant question: with what bindings does a given value match a pattern specification (a set of positive binding patterns, a set of positive non-binding patterns, and a set of negative (non-binding) patterns)? The answer to this question is core to the semantics of the TinyBang language. A parallel question arises during type checking: with what bindings does a given type match a pattern type specification? We present the type system equivalent of compatibility here.

Recall that the design of the evaluation compatibility relation was selected with the intent of simplifying the alignment between that relation and its type system equivalent; we reap the rewards of this effort now. We define type compatibility as follows:

Definition 3.26 (Type Compatibility)**.** Taking $\updownarrow$ to range over the set $\{\uparrow, \downarrow\}$, we mutually define the relations $\tau\backslash C \sim \frac{\Pi^{\updownarrow}/\Pi^{+}}{\Pi^{-}} \updownarrow C'$ and $\alpha\backslash C \sim \frac{\Pi^{\updownarrow}/\Pi^{+}}{\Pi^{-}} \updownarrow C'$ to hold when a proof exists in the system appearing in Figure 3.14.

The type compatibility relation is very nearly a syntactic substitution of the value compatibility relation from Section 3.4.1. In fact, the only two rules which are not syntactic substitutions are the Binding and Type Selection rules. The Binding rule introduces filtered lower bounds to the constraint set; the Type Selection rule makes use of them. We discuss those rules below and provide examples. Before this discussion, however, we introduce some notation which is used exclusively in examples for the sake of legibility.

$$\textsc{Leaf}\quad \frac{}{\alpha\backslash C \sim \frac{\emptyset/\emptyset}{\emptyset} \Updownarrow \emptyset}$$

$$\textsc{Type Selection}\quad \frac{\tau|^{\Pi_1^+}_{\Pi_1^-} <: \alpha \in C \qquad \tau\backslash C \sim \frac{\Pi^{\phi}/\Pi_1^+\cup\Pi_2^+}{\Pi_1^-\cup\Pi_2^-} \Updownarrow C'}{\alpha\backslash C \sim \frac{\Pi^{\phi}/\Pi_2^+}{\Pi_2^-} \Updownarrow C'}$$

$$\textsc{Binding}\quad \frac{\tau\backslash C \sim \frac{\Pi^{\phi}/\Pi^+}{\Pi^-} \Uparrow C' \qquad C'' = \{\tau|^{\Pi^{\phi}\cup\Pi^+}_{\Pi^-} <: \alpha \mid \alpha\backslash\Psi \in \Pi^{\phi}\}}{\tau\backslash C \sim \frac{\Pi^{\phi}/\Pi^+}{\Pi^-} \Downarrow C' \cup C''}$$

$$\textsc{Empty Onion Pattern}\quad \frac{\tau\backslash C \sim \frac{\Pi_1^{\phi}/\Pi_1^+}{\Pi^-} \Downarrow C' \qquad \Pi_2^{\phi} \cup \Pi_2^+ = \{\,()\,\backslash\Psi\}}{\tau\backslash C \sim \frac{\Pi_1^{\phi}\cup\Pi_2^{\phi}/\Pi_1^+\cup\Pi_2^+}{\Pi^-} \Uparrow C'}$$

$$\textsc{Conjunction Pattern}\quad \frac{\begin{array}{c}\tau\backslash C \sim \frac{\Pi_1^{\phi}/\Pi_1^+}{\Pi_1^-} \Downarrow C' \qquad \Pi_2^{\phi} \subseteq \{\alpha_1' * \alpha_2'\backslash\Psi \mid \alpha_1'\backslash\Psi \in \Pi_1^{\phi} \wedge \alpha_2'\backslash\Psi \in \Pi_1^{\phi}\} \\ \Pi_2^+ \subseteq \{\alpha_1' * \alpha_2'\backslash\Psi \mid \alpha_1'\backslash\Psi \in \Pi_1^+ \wedge \alpha_2'\backslash\Psi \in \Pi_1^+\} \\ \Pi_2^- \subseteq \{\alpha_1' * \alpha_2'\backslash\Psi \mid \alpha_1'\backslash\Psi \in \Pi_1^- \vee \alpha_2'\backslash\Psi \in \Pi_1^-\}\end{array}}{\tau\backslash C \sim \frac{\Pi_1^{\phi}\cup\Pi_2^{\phi}/\Pi_1^+\cup\Pi_2^+}{\Pi_1^-\cup\Pi_2^-} \Uparrow C'}$$

$$\textsc{Conjunction Weakening}\quad \frac{\begin{array}{c}\tau\backslash C \sim \frac{\Pi_1^{\phi}\cup\Pi_2^{\phi}/\Pi_1^+\cup\Pi_2^+}{\Pi_1^-\cup\Pi_2^-} \Downarrow C' \qquad \Pi_2^{\phi} \subseteq \{\alpha_i\backslash\Psi \mid i \in \{1,2\} \wedge \alpha_1 * \alpha_2\backslash\Psi \in \Pi_1^{\phi}\} \\ \Pi_2^+ \subseteq \{\alpha_i\backslash\Psi \mid i \in \{1,2\} \wedge \alpha_1 * \alpha_2\backslash\Psi \in \Pi_1^+\} \\ \Pi_2^- \subseteq \{\alpha_i\backslash\Psi \mid i \in \{1,2\} \wedge \alpha_1 * \alpha_2\backslash\Psi \in \Pi_1^-\}\end{array}}{\tau\backslash C \sim \frac{\Pi_1^{\phi}/\Pi_1^+}{\Pi_1^-} \Downarrow C'}$$

$$\textsc{Empty Onion}\quad \frac{\textsc{CtrPat}(\Pi^-)}{()\,\backslash C \sim \frac{\emptyset/\emptyset}{\Pi^-} \Uparrow \emptyset} \qquad\qquad \textsc{Function}\quad \frac{\textsc{CtrPat}(\Pi^-)}{(\pi \to t)\backslash C \sim \frac{\emptyset/\emptyset}{\Pi^-} \Uparrow \emptyset}$$

$$\textsc{Onion}\quad \frac{\begin{array}{c}\alpha_1\backslash C \sim \frac{\Pi_1^{\phi}/\Pi_1^+}{\Pi^-\cup\Pi_2^{\phi}\cup\Pi_2^+} \Uparrow C_1' \\ \alpha_2\backslash C \sim \frac{\Pi_2^{\phi}/\Pi_2^+}{\Pi^-} \Uparrow C_2' \qquad \textsc{CtrPat}(\Pi_1^{\phi} \cup \Pi_2^{\phi} \cup \Pi_1^+ \cup \Pi_2^+ \cup \Pi^-)\end{array}}{\alpha_1 \,\&\, \alpha_2\backslash C \sim \frac{\Pi_1^{\phi}\cup\Pi_2^{\phi}/\Pi_1^+\cup\Pi_2^+}{\Pi^-} \Uparrow C_1' \cup C_2'}$$

$$\textsc{Label}\quad \frac{\alpha\backslash C \sim \frac{\Pi^{\phi}/\Pi^+}{\Pi_1^-} \Downarrow C' \qquad \textsc{CtrPat}(\Pi_2^-) \qquad \{l\ \alpha'\backslash\Psi \in \Pi_2^-\} = \emptyset}{l\ \alpha\backslash C \sim \frac{l\,\Pi^{\phi}/l\,\Pi^+}{(l\,\Pi_1^-)\cup\Pi_2^-} \Uparrow C'}$$

Figure 3.14: Type Compatibility

3.5.3.1 Example Notation

As stated above, the shallow nature of type constraints aligns very well with the ANF grammar of the evaluation system. It provides several advantages in a formal system: simplification of decidability and soundness proofs, an ease of naming (since every point in any tree in either system is uniquely named by a variable), and so forth. Unfortunately, the introduction of so many unique variable names is detrimental to readability. It is much more intuitive to discuss the type `‘A ‘B ()` rather than the type $\alpha_0\backslash\{\texttt{‘A}\ \alpha_1 <: \alpha_0, \texttt{‘B}\ \alpha_2 <: \alpha_1, \texttt{()} <: \alpha_2\}$ so long as the particular variable names α_0, α_1, and α_2 have no special relevance to the topic at hand. As types become more complex, this only becomes more significant.

Although we formalize the type system in terms of the more verbose and algebraically convenient constraint sets, the nested type form is useful in discussions. As we wish to ensure that the discussion remains formal and precise, we now introduce notation for using these nested types in a formal manner to refer to their set constraint equivalents.

Notation 3.27 (Nested Types)**.** Throughout the remainder of this document, the following notation applies:

- If a type appears as the lower bound of a constraint without pattern sets, those sets are assumed to be empty. For instance, we take $\texttt{()} <: \alpha$ to mean $\texttt{()}\,\Big|^{\emptyset}_{\emptyset} <: \alpha$.

- If a type appears where a type variable was expected within a constraint in a set, we take this to mean that there exists an additional variable in the set which is bounded from below only by that type and that this additional variable was intended in place of the type. For instance, we take $\alpha_0\backslash\{\texttt{‘A ()} <: \alpha_0\}$ to mean $\alpha_0\backslash\{\texttt{‘A}\ \alpha_1 <: \alpha_0, \texttt{()} <: \alpha_1\}$ for some $\alpha_1 \notin \{\alpha_0\}$.

- If a type τ appears where a constrained type $\alpha\backslash C$ is expected, we take α_0 to be some otherwise-unused variable and assume that τ appears as its only lower bound in C. For instance, we take `()` to mean $\alpha_0\backslash\{\texttt{()} <: \alpha_0\}$ when a constrained type is expected.

- We interpret patterns in a fashion analogous to the previous two points. For instance, we take `‘A ‘B ()` to mean $\alpha_0\backslash\{\texttt{‘A}\ \alpha_1 <: \alpha_0, \texttt{‘B}\ \alpha_2 <: \alpha_1, \texttt{()} <: \alpha_2\}$ for some otherwise-unused α_0, α_1, and α_2 when a pattern is expected.

- If the term $\alpha_1 \cup \alpha_2$ appears where a type variable was expected, we take this to mean that there exists an additional variable in the set which is bounded from below only by those two type variables and that this additional variable was intended in place of the union expression.[2] This may (and often will) be combined with other interpretations of certain terms in place of type variables. For instance, we take `‘A ()` $\cup$ `‘B ()` to mean $\alpha_0\backslash\{$`‘A ()` $<:\alpha_0$, `‘B ()` $<:\alpha_0\}$ (which is in turn interpreted as above).

Using the above example notation, we now discuss the two rules of significance in the type compatibility relation.

3.5.3.2 The Binding Rule

In both systems, the Binding rule is the only rule which adds to the bindings for a given compatibility relation; all other rules either create an empty binding set or faithfully relay the bindings from their premises. In the evaluation system, all pattern variables are bound to the value currently under consideration (the first place in the compatibility relation). These bindings are represented as value clauses and later used to represent values arriving at the variables defined by the pattern.

In the type system, however, the constraint corresponding to the value clause includes the type restriction; thus, type system bindings are represented by constraints of the form $\tau|^{\Pi^+}_{\Pi^-} <: \alpha$. The pattern sets Π^+ and Π^- restrict the type τ by limiting it to the cases in which it matches the patterns Π^+ and does not match the patterns Π^-. This is unnecessary in the evaluation system because the value under consideration only represents a single value: it either matches a pattern or it does not. A type, however, may match a pattern in some cases but not others because types may represent multiple values.

As an example, consider the constrained type[3] `‘A (‘B ()` $\cup$ `‘C ())`. Also consider the following function:

```
p3 \ { p3 = ‘A p4
     , p4 = ‘B p5
     , p5 = ()
     } ->
     { x6 = p4
     }
```

This function has type (`‘A` α_4)$\backslash\{$`‘B ()` $<: \alpha_4\} \rightarrow \alpha_6\backslash\{\alpha_4 <: \alpha_6\}$; in other words, it accepts anything of the form `‘A ‘B` ... and its return type is exactly the type of the `‘B` label and its contents. Suppose that the above type – `‘A (‘B ()` $\cup$ `‘C ())` – is passed to this function. Within the body of the function, we know we are only dealing with the cases of this type which can match the `‘A ‘B ()` pattern; thus, we know that α_4 is bound

[2]Recall that, in a subtype constraint system, union types can be described with multiple lower bounds on a type variable.

[3]Here, we are using the notation from Section 3.5.3.1.

to a `‘B` label only. The compatibility relation represents this with a binding set such as $\{$`‘B ()`$|^{\{\texttt{‘B ()}\}}_{\emptyset} <: \alpha_4,$ `‘C ()`$|^{\{\texttt{‘B ()}\}}_{\emptyset} <: \alpha_4\}$, which directly expresses the previous intuitions: the filtered type `‘C ()`$|^{\{\texttt{‘B ()}\}}_{\emptyset}$ has no inhabitants because `‘C ()` does not match the pattern `‘B ()`. The resulting bindings are, for all intents and purposes, equivalent to the type `‘B ()`, but this sort of reduction is not always possible in the general case.[4] Rather than eagerly reducing the type now, we opt to defer the work to the point at which the filtered type becomes relevant: the Type Selection rule.

3.5.3.3 The Type Selection Rule

As the Binding rule of type compatibility introduces new lower-bounding filtered type constraints, the Type Selection rule consumes them. Like the Value Selection rule in the evaluation semantics, the purpose of the Type Selection rule is to construct a variable-based compatibility proof using a type-based compatibility proof. But for the reasons describe above, the Type Selection rule receives a type as well as a restriction in the form of two pattern sets; any statement we make regarding the compatibility of the variable must apply to the type under these restrictions. Conveniently, we can simply incorporate those restrictions as part of the compatibility proof burden! The form of the compatibility relation was chosen specifically to make this possible, thus easing the process of handling filtered types. We drive our explanation by example.

Consider the case of passing the `x6` return value from the above example to a compound function of type (`‘C ()` $\rightarrow \ldots$) & (`‘B ()` $\rightarrow \ldots$). (We elide the body types here because they are unimportant.) Because the argument type is equivalent to `‘B ()`, we would expect it to fail to match the first pattern and to succeed in matching the second. This is, in fact, the behavior we observe.

When checking compatibility with the first pattern, `‘C ()`, we have two cases: one for each of the lower bounds of α_4. In the case of `‘B ()`$|^{\{\texttt{‘B ()}\}}_{\emptyset} <: \alpha_4$, the Type Selection rule dictates that we must proceed on the type `‘B ()` with the additional burden that it matches the pattern `‘B ()` (which is added to the P^+ position). With or without this additional proof burden, the type `‘B ()` can be shown incompatible with the pattern `‘C ()` and not compatible with it. In the other case – `‘C ()`$|^{\{\texttt{‘B ()}\}}_{\emptyset} <: \alpha_4$ – we proceed likewise but with the type `‘C ()`. In this case, the additional proof burden becomes crucial. It is, of course, possible to show that the type `‘C ()` is compatible with the pattern `‘C ()`, but the

[4]Previous drafts of this type system [49] used a *slicing* technique to construct union-eliminated bindings at this point (rather than restrict the type using patterns); a new type was actively constructed which represented only the portion of the original type that matched the patterns. While appealing, this technique was subject to subtle decidability issues, the solutions to which were not elegant. This type restriction-based model is a refinement of that legacy.

Type Selection rule also adds the burden of the restriction – `‘B ()` – to the proof. Since it cannot be shown that the type `‘C ()` matches the pattern `‘B ()`, it cannot be shown that α_4 matches the pattern `‘C ()` for the first function. In this way, empty types are lazily filtered out during pattern matching.

When checking compatibility with the second pattern, `‘B ()`, the process is much the same. In this case, the first lower bound successfully matches (since its restriction is essentially redundant here) and pattern matching succeeds. The second case proceeds the same way and will, in fact, continue to do so in any pattern matching case, since the type `‘C ()` will never match the pattern `‘B ()` regardless of which other patterns are involved.

3.5.3.4 Positive Non-Binding Patterns

In designing the value compatibility relation in Section 3.4.1, we included a position which is never used: the P^{+} position, which we call the “positive non-binding patterns.” Although the operational semantics can do without this position, we include it in order to simplify the alignment between the evaluation system and the type system. The type system uses that position to store the proof obligations incurred by restrictions which have been encountered in the argument.

The distinction between the positions $\Pi^{\ddagger}$ and Π^{+} is that the former creates variable bindings while the latter does not. In practice, we are attempting to apply a function to an argument and we wish to receive bindings for that function’s pattern. It would not make sense to include bindings for the other functions that the argument has passed through before arriving here; thus, those patterns are stored in the latter set.

3.5.3.5 Pattern Sets and Union Alignment

In addition to allowing the restrictions to be incorporated into the compatibility proof as positive non-binding patterns, our use of pattern sets in the compatibility relation serves another purpose: it solves the subtle problem of *union alignment* [22]. Consider checking the compatibility of the argument `‘A ()` $\cup$ `‘B ()` with the pattern `‘A ()` $*$ `‘B ()`. This pattern match should fail: there are values in the type (e.g. `‘A ()`) which do not match the pattern. Intuitively, an argument matches a conjunction pattern if it matches both conjuncts, so we should be able to reduce this problem to two smaller problems: whether `‘A ()` $\cup$ `‘B ()` matches the pattern `‘A ()` and whether it matches the pattern `‘B ()`. The difficulty is these union types neither entirely match nor entirely fail to match those patterns: some values in the type `‘A ()` $\cup$ `‘B ()` will match the pattern `‘A ()` and some values will not.

This problem results from the existence of union types, which are represented in a set of type constraints by multiple lower bounds on a single type variable. Recall from Notation 3.27 that the type $\texttt{`A ()} \cup \texttt{`B ()}$ is more properly written $\alpha\backslash\{\texttt{`A ()} <: \alpha, \texttt{`B ()} <: \alpha\}$. In the evaluation system, the corresponding environment – $[x = \texttt{`A } \ldots, x = \texttt{`B } \ldots]$ – is ill-formed: only one value is assigned to each variable at runtime. During static analysis, however, we often only know that the value appearing at a given point in the program is drawn for a particular set; representing this nondeterminism requires union types.

Intuitively, however, we can proceed by case analysis: consider each disjunct as a separate case. This leads to a rule like the Type Selection rule, where a lower bound is selected nondeterministically from the set: proofs of the relation can now be constructed using either disjunct in the type. But these nondeterministic union eliminations must be done *consistently*. In the above example, consider the two smaller tasks: matching $\texttt{`A ()} \cup \texttt{`B ()}$ with $\texttt{`A ()}$ and matching $\texttt{`A ()} \cup \texttt{`B ()}$ with $\texttt{`B ()}$. We can show a positive match in the first task by selecting the left disjunct and we can show a positive match in the second task by selecting the right disjunct, but this doesn't mean that a value of the type will match the pattern. To avoid false positives, we must ensure that the same decision is made about each union at each point in the type throughout a given proof.

Previous type compatibility relations in this theory [49] have more closely resembled one of the draft versions presented in the operational semantics section: specifically, Figure 3.7. In that relation, a single value type is checked against a single pattern. In order to avoid the above union alignment problem, it was necessary to ensure that unions had been previously eliminated from the type; otherwise, the handling of conjunction patterns would induce two positions in the proof tree where the same part of the type was being union eliminated and the aforementioned union elimination problem could not be prevented without adding ungainly alignment constraints to the relation.

In this type theory, we instead ensure that each part of the type is union-eliminated by at most one point in the proof tree. In the Conjunction Pattern rule, for instance, there is only one premise relying on a compatibility proof. The only case in which multiple compatibility premises are permitted is the Onion rule, where each premise holds for a disjoint part of the type we are examining. In this way, the union elimination problem is solved implicitly by limiting the form of compatibility proofs rather than by explicit alignment or argument type preprocessing.

3.5.4 Application Matching

The type system's application matching relation directly parallels the evaluation system's relation in Section 3.4.2. As in the evaluation system, this relation propagates the bindings from compatibility, adds the body of the matched function, and enforces left precedence on dispatch. In a fashion similar to the type compatibility relation (though to a lesser extent), the presence of union types and restrictions complicates matters; here, we discuss these issues, give a formal definition of the relation, and then highlight the differences between the evaluation and type systems.

Because filtered types are present, we must accommodate them in the application matching relation. In the evaluation system, for instance, it is enough that a value is assigned to the function variable x_0 and that it has certain properties. In the type system, however, it is possible to have constraints of the form $(\pi \rightarrow t)|_{\emptyset}^{\{\texttt{`A ()}\}} <: \alpha_0$. In analogy with the above examples of filtered types in compatibility, this type has no inhabitants – there are no function values which match the pattern `` `A () `` – and so this type is spurious. In this case, acting on that type would not be unsound as it would simply cause the type system to overapproximate the type of the application. To limit false positives, however, it is desirable to make use of this information.

To utilize restrictions within application matching, we give a predicate which conservatively determines whether a type is empty. We say a type is *sensible* if it meets this predicate; all types which are not sensible are empty (though the converse is not true). We define our sensibility predicate as follows:

Definition 3.28 (Sensible Types)**.** We define $\textsc{Sensible}(\tau, \Pi^+, \Pi^-, C)$ to be true iff $\tau\backslash C \sim \frac{\emptyset/\Pi^+}{\Pi^-} \not\Downarrow \emptyset$.

We observe that, for instance, $\textsc{Sensible}(\pi \rightarrow t, \{\texttt{`A ()}\}, \emptyset, C)$ is never true for the reasons outlined above. In some cases, this sensibility predicate still overapproximates; for instance, the type $\alpha_0\backslash\{\texttt{`A}\ \alpha_1|_{\emptyset}^{\emptyset} <: \alpha_0, \texttt{()}\ |_{\emptyset}^{\{\texttt{`B ()}\}} <: \alpha_1\}$ appears sensible (despite the fact that $\texttt{()}\ |_{\emptyset}^{\{\texttt{`B ()}\}}$ is not) because our exploration into the type is initially motivated by the filtering pattern sets on the root type variable α_0 (which are empty). Solving this problem perfectly is expensive if not impossible [38, 67]; this somewhat shallow sensibility predicate acts as a good tradeoff between precision and performance.

Using this predicate, we formally define application matching as follows:

Definition 3.29 (Type Application Matching)**.** We define the relation $\alpha_0\ \alpha_1 \overset{\Pi}{\underset{C}{\rightsquigarrow}}{}^{\Pi'}_{\odot}\ \alpha'\backslash C'\ @\ C''$ to hold when a proof exists in the system appearing in Figure 3.15.

As in the compatibility relation, the application matching relation of the evaluation system is designed primarily to align well with this relation; in order to meet this goal, it contains two redundant pattern set positions: one for patterns which have been shown elsewhere in the proof to fail to match the argument and another for patterns which have been shown in this proof subtree to fail to match the argument. While these positions are redundant in the evaluation system, they are necessary in the type system: the nondeterministic selection of constraints in the type application matching relation makes it subject

Function Match

$$\frac{\tau|^{\Pi^+}_{\Pi^-} <: \alpha_0 \in C \qquad \tau = \pi \rightarrow \alpha'\backslash C' \qquad \textsc{Sensible}(\tau, \Pi^+, \Pi^-, C) \qquad \alpha_1\backslash C \sim \frac{\{\pi\}/\emptyset}{\Pi} \oint C''}{\alpha_0\ \alpha_1\ {}^{\Pi}_{C}{\rightsquigarrow}^{\{\pi\}}_{\bullet}\ \alpha'\backslash C' \mathbin{@} C''}$$

Function Mismatch

$$\frac{\tau|^{\Pi^+}_{\Pi^-} <: \alpha_0 \in C \qquad \tau = \pi \rightarrow \alpha'\backslash C' \qquad \textsc{Sensible}(\tau, \Pi^+, \Pi^-, C) \qquad \alpha_1\backslash C \sim \frac{\emptyset/\emptyset}{\Pi\cup\{\pi\}} \oint \emptyset}{\alpha_0\ \alpha_1\ {}^{\Pi}_{C}{\rightsquigarrow}^{\{\pi\}}_{\circ}\ \alpha'\backslash \emptyset \mathbin{@} \emptyset}$$

Non-Function

$$\frac{\tau|^{\Pi^+}_{\Pi^-} <: \alpha_0 \in C \qquad \tau \text{ not of the form } \pi \rightarrow t \text{ or } \alpha'_1 \mathbin{\&} \alpha'_2 \qquad \textsc{Sensible}(\tau, \Pi^+, \Pi^-, C)}{\alpha_0\ \alpha_1\ {}^{\Pi_1}_{C}{\rightsquigarrow}^{\emptyset}_{\circ}\ \alpha'\backslash \emptyset \mathbin{@} \emptyset}$$

Onion Left

$$\frac{\begin{array}{c}\tau|^{\Pi^+}_{\Pi^-} <: \alpha_0 \in C \\ \tau = \alpha_2 \mathbin{\&} \alpha_3 \qquad \textsc{Sensible}(\tau, \Pi^+, \Pi^-, C) \qquad \alpha_2\ \alpha_1\ {}^{\Pi_1}_{C}{\rightsquigarrow}^{\Pi_2}_{\bullet}\ \alpha'\backslash C' \mathbin{@} C''\end{array}}{\alpha_0\ \alpha_1\ {}^{\Pi_1}_{C}{\rightsquigarrow}^{\Pi_2}_{\bullet}\ \alpha'\backslash C' \mathbin{@} C''}$$

Onion Right

$$\frac{\begin{array}{c}\tau|^{\Pi^+}_{\Pi^-} <: \alpha_0 \in C \qquad \tau = \alpha_2 \mathbin{\&} \alpha_3 \qquad \textsc{Sensible}(\tau, \Pi^+, \Pi^-, C) \\ \alpha_2\ \alpha_1\ {}^{\Pi_1}_{C}{\rightsquigarrow}^{\Pi_2}_{\circ}\ \alpha'_2\backslash C'_2 \mathbin{@} C''_2 \qquad \alpha_3\ \alpha_1\ {}^{\Pi_1\cup\Pi_2}_{C}{\rightsquigarrow}^{\Pi_3}_{\odot}\ \alpha'_3\backslash C'_3 \mathbin{@} C''_3\end{array}}{\alpha_0\ \alpha_1\ {}^{\Pi_1}_{C}{\rightsquigarrow}^{\Pi_2\cup\Pi_3}_{\odot}\ \alpha'_3\backslash C'_3 \mathbin{@} C''_3}$$

Figure 3.15: Type Application Matching Relation

to the same form of union alignment concerns as the type compatibility relation.

For instance, consider the case in which a function of type $(\texttt{`Some ()} \rightarrow \ldots) \mathbin{\&} (\texttt{`None ()} \rightarrow \ldots)$ is applied to an argument of type $(\texttt{`Some ()}) \cup (\texttt{`None ()})$. We would expect this to be successful: there should exist two proofs of compatibility (one for each side of the disjunct) which successfully match the appropriate function bodies; further, it should be impossible to construct a proof of failure ($\odot = \circ$) since the disjuncts are exhaustively matched.

For the same reasons as in the compatibility relation, it is insufficient simply to prove compatibility for each side of the disjunct: we must make the union elimination decision in a consistent way. In the compatibility rules, we forced the union elimination decision into a single point in the proof tree. We do not apply that strategy here, however, because such an approach would require matching on a *list* of functions simultaneously, which becomes tremendously burdensome. Instead, we create each proof of compatibility with the appropriate context, including all of the patterns which came before it. Taking

α_0 to be the function type and α_1 to be the argument type, we can prove (again using Notation 3.27) that $\alpha_1\backslash C \sim \frac{\{\texttt{`Some ()}\}/\emptyset}{\emptyset} \phi \ldots$. Then, to consider all cases of the argument which are not part of that outcome, we proceed by adding `‘Some ()` to the claimed pattern set. It can then be shown that $\alpha_1\backslash C \sim \frac{\{\texttt{`None ()}\}/\emptyset}{\{\texttt{`Some ()}\}} \phi \ldots$. Using these as the premises to the Function Match rule, one can then build two proofs of type application matching, each corresponding to one of the ways in which the compound function may match its union-typed argument.

Suppose we wanted to show that applying a function of type (`‘Some ()` $\rightarrow \ldots$) & (`‘None ()` $\rightarrow \ldots$) to an argument of type (`‘Some ()`) $\cup$ (`‘None ()`) might fail. (This should not be possible – as stated above, the patterns of this compound function exhaustively cover the argument – but it is helpful to illustrate why.) We must begin by showing via the Function Mismatch rule that the argument does not match the pattern `‘Some ()` (which can be proven, as it is how the second successful match described above is constructed.) But because the Function Mismatch rule “claims” that pattern, the next use of Function Mismatch is obligated to show $\alpha_1\backslash C \sim \frac{\emptyset/\emptyset}{\{\texttt{`Some ()}, \texttt{`None ()}\}} \phi \ldots$ – that is, that the argument can fail to match the `‘Some ()` and `‘None ()` patterns *simultaneously*. Although the argument can fail to match either of these patterns, it does not fail to match both of them simultaneously. As a result, we cannot prove a match failure; this is desirable, since the application is an exhaustive match.

With this application matching relation defined, we can now go own to formalize the process of constraint closure.

3.5.5 Constraint Closure

In this section, we define the constraint closure relation. Each step of constraint closure propagates constraint information and abstractly models a single step of the operational semantics. This constraint closure is defined in terms of an abstract polymorphism framework; we first discuss that framework and then present the constraint closure relation.

3.5.5.1 Polymorphism

The TinyBang type system is parametric in its polymorphism model, which consists of two functions. The first, Φ, is analogous to the $\boldsymbol{\alpha}(-,-)$ freshening function of the operational semantics: $\Phi(\alpha, \alpha')$ gives the type variable which should be used at call site α to represent the variable α'. For clarity in discussing this function, we give formal notation similar to that of the operational semantics:

Notation 3.30 (Type Variable Replacement)**.** We use σ to range over type variable replacement functions (e.g. $\sigma(\alpha) = \alpha'$). We also introduce some notational sugar for clarity:

- We overload the application of a replacement function to operate homomorphically over constraint sets and other type system constructs (e.g. $\sigma(\alpha_1 <: \alpha_2) = \sigma(\alpha_1) <: \sigma(\alpha_2)$.
- We write $\sigma\big|_C$ to indicate the function which replaces all type variables bound[5] by C in the same manner as σ and which replaces all other variables with themselves. Taken with the above, we have $\sigma\big|_{\{\alpha_1 <: \alpha_2\}}(\{\alpha_1 <: \alpha_2\}) = \{\alpha_1 <: \sigma(\alpha_2)\}$ (since α_2 is bound by the constraint set but α_1 is not).

Corresponding with the evaluation system, we also need a function which freshens only the variable part of bindings:

Definition 3.31 (Binding Type Freshening)**.** We let $\textsc{TFresh}(\sigma, \overset{n}{\overbracket{\alpha_{\square} <: \alpha'_{\square}}}) = \overset{n}{\overbracket{\alpha_{\square} <: \sigma(\alpha'_{\square})}}$.

Note that these definitions correspond quite directly to those provided in Section 3.4.3. Unlike in evaluation, however, we do not require $\Phi(\alpha, -)$ to be an injective function. For decidability, Φ must not freshen *every* variable uniquely; it may sometimes map a variable to itself or onto other variables which have already appeared during closure. We formally define the polymorphism model properties required for decidability in Chapter 5. The abstraction of this polyinstantiation function is connected to *a posteriori* soundness in abstract interpretation [42], which we discuss in Chapter 4.

The second function required by a TinyBang polymorphism model is the merging function Υ. This function is used to merge type variables by producing for a given set of type variables $\overbracket{\alpha}$ a set of type variable replacement functions $\overbracket{\sigma}$. In concept, any replacement expressed by Υ represents a type variable unification. This merging function is used exclusively to *weaken* the type system in order to improve performance in practice. When ignoring concerns of performance, readers may safely imagine this function to return an empty set. For notational convenience, we write $\Upsilon(C)$ to mean $\Upsilon(\overbracket{\alpha})$ where $\overbracket{\alpha}$ is the set of all type variables appearing anywhere within C.

The simplest model is $\Phi_{\textsc{Mono}}/\Upsilon_{\textsc{Mono}}$, the canonical monomorphic model:

Definition 3.32 (Monomorphic Model)**.** We define the TinyBang monomorphic model as follows:

- $\Phi_{\textsc{Mono}}(\alpha, \alpha') = \alpha'$
- $\Upsilon_{\textsc{Mono}}(C) = \emptyset$

We present the monomorphic system here because, while it is relatively uninteresting, it provides a good intuition when considering the rules below. The implementation of the TinyBang typechecker uses a polymorphism model called CR which we discuss in

[5]Recall from Section 3.5.2 that we refer to variables as bound and free not by explicit quantifiers but in terms of the constraint set itself; although the terminology is somewhat unusual, the analogy is helpful in discussing the alignment between the systems.

$$\textsc{Transitivity}\quad \frac{\{\tau\big|_{\Pi^-}^{\Pi^+} <: \alpha_1, \alpha_1 <: \alpha_2\} \subseteq C}{C \Longrightarrow^1 C \cup \{\tau\big|_{\Pi^-}^{\Pi^+} <: \alpha_2\}}$$

$$\textsc{Application}\quad \frac{\begin{array}{cccc} & \alpha_0\ \alpha_1 <: \alpha_2 \in C & \multicolumn{2}{c}{\alpha_0\ \alpha_1 \overset{\emptyset}{\underset{C}{\rightsquigarrow}}{}^{\Pi}_{\bullet}\ \alpha_1' \backslash C_1' \ @\ C_1''} \\ \sigma = \Phi(\alpha_2, -) & \sigma' = \sigma\big|_{C_1' \cup C_1''} & \alpha_2' \backslash C_2' = \sigma'(\alpha_1' \backslash C_1') & C_2'' = \textsc{TFresh}(\sigma', C_1'') \end{array}}{C \Longrightarrow^1 C \cup C_2' \cup C_2'' \cup \{\alpha_2' <: \alpha_2\}}$$

$$\textsc{Merging}\quad \frac{\sigma \in \Upsilon(C)}{C \Longrightarrow^1 C \cup \sigma(C)}$$

Figure 3.16: Type Constraint Closure

Chapter 6.

3.5.5.2 Constraint Closure Relation

We can now formalize constraint closure in terms of a polymorphism model; this formalism closely follows that of the small-step operational semantics relation in Section 3.4.3. We write $C \Longrightarrow^1 C'$ to indicate a single step of constraint closure, formally defining it as follows:

Definition 3.33 (Constraint Closure)**.** We define $C \Longrightarrow^1 C'$ to hold when a proof exists in the proof system appearing in Figure 3.16.

We note that this definition of the constraint closure relation is implicitly parametric in a polymorphism model Φ/Υ. We assume when it is used that it is evident from context whether the polymorphism model is fixed or not.

It is also convenient to parallel the operational semantics by giving notation for multiple constraint closure steps:

Definition 3.34 (Multiple Constraint Closure)**.** Let $C_0 \Longrightarrow^* C_n$ iff $C_0 \Longrightarrow^1 \ldots \Longrightarrow^1 C_n$ for some $n \geq 0$.

Each operational semantics rule corresponds to a single constraint closure rule. The Variable Lookup rule of the operational semantics small step relation, for instance, corresponds directly to the Transitivity rule of constraint closure. The same is true of the Application rules; although the similarity is obscured slightly by the different algebraic notation used in the rules, the type system Application rule has premises equivalent to those in the evaluation system Application rule (except that these premises are mapped across the alignment which, as previously discussed, forms a Galois connection). The most significant difference between the two relations is the use of $\Phi(-,-)$ rather than $\boldsymbol{\alpha}(-,-)$; in practice, the former will only partly freshen variables so that the overall number of constraints remains finite. The only oddity in this rule set is the Merging rule, which we discuss below in Section 3.5.5.2.1.

The operational semantics has a definition for a "stuck" expression (Definition 3.17). The type system analogue is the *inconsistent* constraint set: a constraint set which contains contradictory or otherwise problematic constraints. In traditional constraint theory, such constraint sets typically include immediately false typing statements (such as $\top \leq \bot$). In TinyBang, there are no upper bounding constraints, but it is possible for a function to be applied to an incorrect argument type.

Definition 3.35 (Inconsistency)**.** A constraint set C is *inconsistent* iff there exists some $\alpha_0\ \alpha_1 <: \alpha_2 \in C$ such that $\alpha_0\ \alpha_1 \ {}^{\emptyset}_{C}\!\!\rightsquigarrow^{\Pi}_{\bigcirc}\ \alpha'_1\backslash C'_1 \ @\ C''_1$. A constraint set which is not inconsistent is *consistent*.

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

Definition 3.36 (Typechecking)**.** A closed expression e *typechecks* iff $[\![e]\!]_{\mathrm{E}} = \alpha\backslash C$ and $C \Longrightarrow^{*} C'$ implies that C' is consistent.

The definition of typechecking is also implicitly parametric in a polymorphism model Φ/Υ.

3.5.5.2.1 Merging In addition to the rules aligning with the operational semantics, the constraint closure rule includes a Merging rule. This rule is effectively used for type variable unification; it adds constraints by using variable replacements provided by Υ on the constraints we already have. This rule is not necessary for soundness (since the other rules already simulate the behavior of the operational semantics), but it also does not interfere with soundness. We provide an intuition for this fact as follows. Observe that both the compatibility and matching relations are monotonic in the number of constraints in the provided constraint set; that is, neither of them relies on the absence of a constraint from the set in any way. Thus, adding more constraints to a set will only increase the ways in which compatibility and matching will hold (both in terms of e.g. successful matches and failed matches). Inconsistency is also a monotonic property on constraint sets: since it relies only on the presence of constraints and matching relations (and never on their absence), any inconsistent constraint set remains inconsistent regardless of how many additional constraints are added to it. Thus, as long as the constraint closure relation is a sound way of statically modeling the small step semantics *without* the Merging rule, the addition of the

Merging rule cannot make an otherwise inconsistent constraint set appear to be consistent.

The value of the Merging rule is that it simplifies the computational complexity of the constraint closure process. In practice, we use a polymorphism model with TinyBang which we call $\Phi_{\mathrm{CR}}/\Upsilon_{\mathrm{CR}}$. We define that polymorphism model in Section 6.2 and show how it simplifies the constraint closure process in Section 9.2.

Chapter 4

Proof of Soundness

The previous chapter formalized an operational semantics and type system for the TinyBang language. In this chapter, we show that the provided type system is sound: it does not admit any programs which become stuck. We begin by presenting our theorem statement[1]:

Theorem 1 (Soundness)**.** For any closed e, if $e \longrightarrow^{*} e'$ and e' is stuck then e does not typecheck.

This chapter proceeds first by discussing the overall proof strategy. We then group the required lemmas by section and prove the above theorem statement at the end of the chapter.

4.1 Proof Strategy

Proving a subtyping system sound is often accomplished via *progress and preservation* [75], a technique which requires the demonstration of two key properties. First, one must show that the type system never assigns valid types to stuck expressions. Second, one must show that, when $e \longrightarrow^{1} e'$, the type assigned to e is a supertype of the type assigned to e'. It is then possible to use an inductive argument to demonstrate that the type system will never assign a valid type to a program which will become stuck.

While subject reduction is an effective technique for traditional type systems, it is not easily applied to TinyBang's type system. Demonstrating the first property is non-trivial but feasible: one must show that the initial alignment of any stuck expression

[1]Recall that the terms "closed" (Definition 3.6) and "stuck" (Definition 3.17) are provided in Section 3.4.

is inconsistent. To show the second property, however, one must demonstrate that the initial alignment of e is a supertype of the initial alignment of e'. This poses a question of subtyping between two polymorphic constrained types, which has been shown to be at least NP-hard and potentially undecidable [38]. While previous work [67] has provided a decidable approximation for this question, it is somewhat complex and difficult to integrate with the TinyBang type system.

Instead of using progress and preservation, we opt for a proof of soundness by simulation. We define a simulation relation between expressions and constraint sets (and respectively between evaluation system constructs and their corresponding type system constructs). We demonstrate that the initial alignment of an expression always produces a constraint set which establishes this relation. We then demonstrate that each corresponding pair of relations in the two systems – value compatibility and type compatibility, for instance – maintain this simulation relation. Ultimately, this is used to show preservation of simulation between the operational semantics and the constraint closure relation. Finally, we show that any stuck expression is simulated only by inconsistent constraint sets; thus, if the initial alignment of an expression closes only to consistent sets, the expression will never become stuck.

As mentioned in Section 3.5, our approach is inspired in part by work in abstract interpretation [21]. In a sense, one can view the constraint closure relation as an abstract interpretation of an ANF TinyBang program. The initial alignment function $[\![-]\!]_{\mathrm{E}}$ can be viewed as an abstraction function α[2] (or a degenerate form thereof); the simulation relation we preserve throughout type checking is related to the left adjoint of a Galois connection between the evaluation system and the type system. Readers familiar with abstract interpretations should feel free to use this intuition while reading the proof below.

Unlike abstract interpretations, however, the closure of the constraint set is *monotonic*. Abstract interpretations involve the execution of an abstracted form of the underlying program; both the concrete interpreter and the abstract interpreter are transition systems which deduce properties relevant to program states. Our constraint closure, on the other hand, monotonically gathers a set of time-invariant facts about the program. We utilize this monotonicity to simplify this proof as well as the decidability proof in the next chapter while still providing the flexibility and type precision offered by existing subtype constraint theory.

[2]For readers unfamiliar with abstract interpretation, α is the traditional name of an abstraction function from sets of concrete values to sets of abstract values (in contrast to the concretization function from sets of abstract values to sets of concrete values, which is usually called γ). We are not referring to a type variable here.

4.2 Simulation

This section defines the simulation relation we will use to show this alignment between evaluation and constraint closure. Simulation ($\preccurlyeq$) relates an evaluation construct, such as an expression or a clause, to a type system construct, such as a constraint set or an individual constraint. This relation is largely a homomorphism; information is only lost in the following places:

1. When relating variables to type variables
   (a) because this mapping may not be injective
2. When relating expressions to constraint sets
   (a) because the ordering of clauses is not preserved
   (b) and because the constraint set may be larger than otherwise necessary (due to conservative approximation)

One subtlety arises in defining this otherwise straightforward relation. Some information loss (namely points 1a and 2b above) is acceptable at the top level of constraint closure but becomes a problem during polyinstantiation. For instance, the presence of "extra" constraints (point 2b above) is admissible at top level during constraint closure because it allows us to construct union types and thus conservatively approximate runtime behavior. But during polyinstantiation, the alignment must be perfect in this respect to ensure that the implicit set of bound variables is properly simulated. We begin our formal definitions by specifying a *bisimulation* relation ($\approx$) which is perfect with respect to points 1a and 2b (but still lossy in point 2a above) and then define our simulation relation in terms of it.

4.2.1 The Bisimulation Relation

Bisimulation, like simulation, relates an evaluation construct to a type system construct. Bisimulation is a three place relation; the third place contains a variable mapping M which is a function mapping each value variable x to the type variable α which represents it. We define bisimulation formally as follows:

Definition 4.1 (Bisimulation)**.** The bisimulation relation ${}_{E}{\approx}_{M}$ is defined as the least relation satisfying the rules in Figure 4.1 such that M is injective.

The bisimulation relation is a fairly straightforward formalization of the alignment discussed intuitively in Section 3.5.1: variables align with type variables, expressions align with constraint sets, and so on. The alignment between clauses and constraints is somewhat

$$
\begin{array}{cccl}
() & {}_E{\approx}_M & () & \\
l\ x & {}_E{\approx}_M & l\ \alpha & \text{iff } x\ {}_E{\approx}_M\ \alpha \\
x_1\,\&\,x_2 & {}_E{\approx}_M & \alpha_1\,\&\,\alpha_2 & \text{iff } x_1\ {}_E{\approx}_M\ \alpha_1 \text{ and } x_2\ {}_E{\approx}_M\ \alpha_2 \\
p\texttt{->}e & {}_E{\approx}_M & \pi \rightarrow \alpha\backslash C & \text{iff } p\ {}_E{\approx}_M\ \pi \text{ and } e\ {}_E{\approx}_M\ \alpha\backslash C \\
\hline
x & {}_E{\approx}_M & \alpha & \text{iff } M(x) = \alpha \\
\hline
x\texttt{=}v & {}_E{\approx}_M & \tau\big|_{\Pi^-}^{\Pi^+} <: \alpha & \text{iff } x\ {}_E{\approx}_M\ \alpha \text{ and } v\ {}_E{\approx}_M\ \tau \text{ and} \\
 & & & \quad P^+\ {}_E{\approx}_M\ \Pi^+ \text{ and } P^-\ {}_E{\approx}_M\ \Pi^- \text{ and } v\backslash E \sim \frac{\emptyset/P^+}{P^-} \Downarrow [] \\
x_2\texttt{=}x_1 & {}_E{\approx}_M & \alpha_1 <: \alpha_2 & \text{iff } x_1\ {}_E{\approx}_M\ \alpha_1 \text{ and } x_2\ {}_E{\approx}_M\ \alpha_2 \\
x_3\texttt{=}x_1\ x_2 & {}_E{\approx}_M & \alpha_1\ \alpha_2 <: \alpha_3 & \text{iff } \forall i \in \{1..3\}.\, x_i\ {}_E{\approx}_M\ \alpha_i \\
\hline
() & {}_E{\approx}_M & () & \\
l\ x & {}_E{\approx}_M & l\ \alpha & \text{iff } x\ {}_E{\approx}_M\ \alpha \\
x_1 \ast x_2 & {}_E{\approx}_M & \alpha_1 \ast \alpha_2 & \text{iff } x_1\ {}_E{\approx}_M\ \alpha_1 \text{ and } x_2\ {}_E{\approx}_M\ \alpha_2 \\
\hline
x\texttt{=}\varphi & {}_E{\approx}_M & \phi <: \alpha & \text{iff } x\ {}_E{\approx}_M\ \alpha \text{ and } \varphi\ {}_E{\approx}_M\ \phi \\
\overset{n\sqcup}{f} & {}_E{\approx}_M & \overset{n\sqcup}{\psi} & \text{iff } \forall 1 \le i \le n.\, f_i\ {}_E{\approx}_M\ \psi_i \\
x\backslash F & {}_E{\approx}_M & \alpha\backslash\Psi & \text{iff } x\ {}_E{\approx}_M\ \alpha \text{ and } F\ {}_E{\approx}_M\ \Psi \\
\hline
\overset{n}{\overrightarrow{s}} & {}_E{\approx}_M & \overset{n\sqcup}{c} & \text{iff } \forall 1 \le i \le n.\, s_i\ {}_E{\approx}_M\ c_i \\
e & {}_E{\approx}_M & \alpha\backslash C & \text{iff } \mathrm{RV}(e)\ {}_E{\approx}_M\ \alpha \text{ and } e\ {}_E{\approx}_M\ C
\end{array}
$$

Figure 4.1: Bisimulation relation

different from the rest of the relation. Here, the constraint includes patterns which the clause does not; this is included in order to support the simulation relation, which we discuss in the next section.

Readers familiar with abstract interpretation may have encountered a parallel to this approach in Might and Manolios's *a posteriori* approach to soundness in abstract interpretation [42]. In that work, an allocation policy (represented there as $\hat{\pi}$) is used to perform abstraction of memory store locations. The resulting abstraction function (represented there as α_L) induced by this policy is demonstrated to be sound on a per-analysis basis. The strategy we use here is quite similar: the allocation policy $\hat{\pi}$ is similar to the polyinstantiation function Φ and the induced abstraction α_L is similar to the mapping M from the bisimulation relation here.

For notational convenience, we may alternately view a variable mapping function M as a set of pairs of the form $x \mapsto \alpha$ where x is unique throughout the set. This is particularly relevant when creating new mappings, as in $M' = M \cup \{x \mapsto \alpha\}$. When creating new variable maps such as M', we will demonstrate that the uniqueness of x is preserved.

4.2.2 The Simulation Relation

Bisimulation is a strong property; it is impossible to maintain throughout constraint closure. Because well-formed expressions define each variable exactly once, for in-

$$
\begin{array}{cccll}
\texttt{()} & {}_E{\preccurlyeq}_M & \texttt{()} & & \\
l\ x & {}_E{\preccurlyeq}_M & l\ \alpha & \text{iff} & x\ {}_E{\preccurlyeq}_M\ \alpha \\
x_1\,\texttt{\&}\,x_2 & {}_E{\preccurlyeq}_M & \alpha_1\,\texttt{\&}\,\alpha_2 & \text{iff} & x_1\ {}_E{\preccurlyeq}_M\ \alpha_1 \text{ and } x_2\ {}_E{\preccurlyeq}_M\ \alpha_2 \\
p\,\texttt{->}\,e & {}_E{\preccurlyeq}_M & \pi \rightarrow \alpha\backslash C & \text{iff} & p\,\texttt{->}\,e\ {}_E{\approx}_M\ \pi \rightarrow \alpha\backslash C \\
\hline
x & {}_E{\preccurlyeq}_M & \alpha & \text{iff} & M(x) = \alpha \\
\hline
x\,\texttt{=}\,v & {}_E{\preccurlyeq}_M & \tau\big|^{\Pi^+}_{\Pi^-} <: \alpha & \text{iff} & x\ {}_E{\preccurlyeq}_M\ \alpha \text{ and } v\ {}_E{\preccurlyeq}_M\ \tau \text{ and} \\
 & & & & P^+\ {}_E{\approx}_M\ \Pi^+ \text{ and } P^-\ {}_E{\approx}_M\ \Pi^- \text{ and } v\backslash E \sim \frac{\emptyset / P^+}{P^-} \between [] \\
x_2\,\texttt{=}\,x_1 & {}_E{\preccurlyeq}_M & \alpha_1 <: \alpha_2 & \text{iff} & x_1\ {}_E{\preccurlyeq}_M\ \alpha_1 \text{ and } x_2\ {}_E{\preccurlyeq}_M\ \alpha_2 \\
x_3\,\texttt{=}\,x_1\ x_2 & {}_E{\preccurlyeq}_M & \alpha_1\ \alpha_2 <: \alpha_3 & \text{iff} & \forall i \in \{1..3\}.\, x_i\ {}_E{\preccurlyeq}_M\ \alpha_i \\
\hline
e & {}_E{\preccurlyeq}_M & C & \text{iff} & \forall s \in e.\, \exists c \in C.\, s\ {}_E{\preccurlyeq}_M\ c \\
e & {}_E{\preccurlyeq}_M & \alpha\backslash C & \text{iff} & \textsc{rv}(e)\ {}_E{\preccurlyeq}_M\ \alpha \text{ and } e\ {}_E{\preccurlyeq}_M\ C
\end{array}
$$

Figure 4.2: Simulation relation

stance, any constraint set bisimulating a well-formed expression contains no union types. Using this property, however, we can define the weaker simulation relation which we do maintain throughout closure:

Definition 4.2 (Simulation)**.** The simulation relation ${}_E{\preccurlyeq}_M$ is defined as the least relation satisfying the rules in Figure 4.2.

Note that, in the simulation relation, constraint sets are permitted to overapproximate expressions: it is only necessary that a constraint exists for each clause, not that there is a one-to-one relationship. We preserve the stronger relation over function bodies, however: simulation between functions and their types is the same as bisimulation.

Now let us consider the alignment between clauses and constraints. Unlike the other alignments, value clauses and lower-bounding constraints do not align perfectly: lower-bounding constraints have type restrictions and the alignment requires a proof of compatibility. Such restricted types are the "filter types" discussed in Section 3.5.1: we use them to retain information gathered about arguments by pattern matching. But we cannot use these filters without justification: it is always sound to overapproximate the type of values which may appear at runtime, but any reduction in our approximation must be supported. We accomplish this by attaching the compatibility proof obligation to the simulation relation: the use of a filtered type constraint to simulate a value is only justified if the value meets the conditions expressed by the filters.

4.2.3 Minimal Variable Mappings

In some cases, it is useful to ensure that the variable mapping used in a simulation is minimal. We maintain a minimal mapping throughout the simulation proof so that, as new variables enter the expression (as a result of freshening during evaluation), we preserve the uniqueness property of the keys in the mapping; spurious mappings would make this difficult. To support the proofs later in this section, we introduce the following lemma:

Lemma 4.3. If $e\ {}_{E}{\preccurlyeq}_{M}\ C$, then $e\ {}_{E}{\preccurlyeq}_{M'}\ C$ such that the set of variables appearing in e is exactly the domain of M'.

Proof. If a variable appears in e but not in M, then this simulation could not hold; it therefore suffices to induct on the number of variables in the domain of M not appearing in e. A proof of the inductive step is immediate from the fact that simulation only uses M to examine variables which appear somewhere within e. □

Likewise, we can arbitrarily extend the mappings used in a simulation proof:

Lemma 4.4. If $e\ {}_{E}{\preccurlyeq}_{M}\ C$ and $M' \supseteq M$ then $e\ {}_{E}{\preccurlyeq}_{M'}\ C$.

Proof. By induction on the size of the proof. □

4.2.4 Simulation from Bisimulation

Because bisimulation is a stronger property than simulation, we also have the following straightforward lemma.

Lemma 4.5. If $e\ {}_{E}{\approx}_{M}\ C$ then $e\ {}_{E}{\preccurlyeq}_{M}\ C$.

Proof. By induction on the size of the proof of $e\ {}_{E}{\approx}_{M}\ C$. □

The inclusion of the compatibility proof requirement in bisimulation exists largely to make this proof trivial.

4.2.5 Expanding the Bisimulation Environment

The environment E used in the bisimulation relation ${}_{E}{\approx}_{M}$ is used exclusively for maintaining proofs of compatibility, ensuring that filtered types properly describe the values they bisimulate. Throughout evaluation, this environment grows and the compatibility proofs are constructed over different environments. Fortunately, value compatibility is monotonic in its environment, so we can demonstrate that the environment of a bisimulation grows freely. We begin by formally stating the property that value compatibility is monotonic:

Lemma 4.6. If $v \backslash E_1 \sim \frac{P^{\oplus}/P^{+}}{P^{-}} \updownarrow E'$ then $v \backslash E_1 \,||\, E_2 \sim \frac{P^{\oplus}/P^{+}}{P^{-}} \updownarrow E'$.

Proof. By inspection of the Value Compatibility relation (Figure 3.8), proofs may only rely on the presence of individual clauses appearing in the environment. Because each clause appearing in E_1 also appears in $E_1 \,||\, E_2$, this proof proceeds by induction on the size of the proof of $v \backslash E_1 \sim \frac{P^{\oplus}/P^{+}}{P^{-}} \updownarrow E'$ and then by case analysis on the proof rule used. □

Given this lemma, it is possible to prove our expansion property:

Lemma 4.7. If $e \;{}_{E_1}\!\!\approx_M C$ then $e \;{}_{E_1 \,||\, E_2}\!\!\approx_M C$. Similar statements hold for the other pairs at which bisimulation is defined.

Proof. By the same strategy as Lemma 4.6 – induction on the size of the proof followed by case analysis on the proof rule – using Lemma 4.6 to show the same property on compatibility proofs. □

Of course, a similar lemma holds for simulation:

Lemma 4.8. If $e \;{}_{E_1}\!\!\preccurlyeq_M C$ then $e \;{}_{E_1 \,||\, E_2}\!\!\preccurlyeq_M C$. Similar statements hold for the other pairs at which simulation is defined.

Proof. By the same strategy as Lemma 4.7, using Lemma 4.7 to show the same property on bisimulation proofs. □

4.2.6 Combining Simulation Proofs

As evaluation and constraint closure proceed, new clauses and constraints are added. To show simulation over those new constraints and clauses, we simply demonstrate that the concatenation of expressions and union of constraint sets is simulation-preserving. We formally state this relatively simple lemma as follows for purposes of reference:

Lemma 4.9. If $e_1 \;{}_{E}\!\!\preccurlyeq_M C_1$ and $e_2 \;{}_{E}\!\!\preccurlyeq_M C_2$ then $e_1 \,||\, e_2 \;{}_{E}\!\!\preccurlyeq_M C_1 \cup C_2$.

Proof. Immediate from Definition 4.2. □

4.3 Initial Alignment

Our overall proof strategy is to use the simulation to show that the type system conservatively approximates stuck states of the evaluation system. At the beginning of typechecking, we must establish the simulation which we will preserve throughout the constraint closure process. This is accomplished by the initial alignment function (Definition 3.19); in fact, this function establishes bisimulation as described above. To prove this, we first require a concept of the *depth* of a pattern:

Definition 4.10 (Pattern Depth)**.** We define the *depth* of a well-formed pattern $x\backslash F$ as follows:

$$\begin{aligned}
\textsc{pd}(()\,\backslash F) &= 0 \\
\textsc{pd}(l\ x\backslash F) &= \textsc{pd}(x\backslash F) + 1 \\
\textsc{pd}(x_1 * x_2\backslash F) &= \max(\textsc{pd}(x_1\backslash F), \textsc{pd}(x_2\backslash F)) + 1
\end{aligned}$$

Quite simply, the depth of a pattern is the maximum number of steps it takes to reach a leaf of the pattern from its root. Using this definition, we can show that the initial alignment function establishes a bisimulation.

Lemma 4.11. If e is well-formed then $e \;{}_E{\approx}_M\; [\![e]\!]_{\mathrm{E}}$.

Proof. By taking E to be $[]$ and M to be $[\![-]\!]_{\mathrm{V}}$. By Definition 4.2, we must show that there exists some M such that $e \;{}_E{\approx}_M\; [\![e]\!]_{\mathrm{E}}$. By induction on the length of e, it suffices to show that there is a one-to-one mapping between clauses and constraints.

Consider a clause of the form $x_1 = l\ x_2$. The initial alignment of this clause is $[\![x_1]\!]_{\mathrm{V}} \backslash ([\![l\ x_2]\!]_{\mathrm{E}})|^{\emptyset}_{\emptyset} <: [\![x_1]\!]_{\mathrm{V}}$; assuming for notation that $[\![x_i]\!]_{\mathrm{V}} = \alpha_i$ for each i, we have $\alpha_1 \backslash l\ \alpha_2|^{\emptyset}_{\emptyset} <: \alpha_1$; thus, it suffices in this case to show that $x_1 = l\ x_2 \;{}_E{\approx}_M\; l\ \alpha_2|^{\emptyset}_{\emptyset} <: \alpha_1$. By Definition 4.1, this requires us to show that $x_1 \;{}_E{\approx}_M\; \alpha_1$ and that $x_2 \;{}_E{\approx}_M\; \alpha_2$ (both of which are satisfied by $M = [\![-]\!]_{\mathrm{V}}$) and also that $x \backslash E \sim \frac{\emptyset/\emptyset}{\emptyset} \Downarrow []$. This latter proof is immediate from the Leaf rule of Figure 3.8. Each other clause of the form $x = v$ is simulated in a similar fashion.

The above demonstrates that each clause is simulated by some constraint. To show that this simulation is one-to-one, we recall that no two clauses in a well-formed expression bind the same variable. Because all clauses bind a variable and all variable bindings must be unique, each clause is *only* simulated by its corresponding constraint; that is, it is impossible for two constraints to simulate the same clause.

To show simulation for functions, we must show that $p \;{}_E{\approx}_M\; [\![p]\!]_{\mathrm{P}}$ and that $e' \;{}_E{\approx}_M\; [\![e']\!]_{\mathrm{E}}$. The latter is true by induction on the size of e'. The former is shown in much the same fashion as the above. By induction on the depth of p, it suffices to show simulation for each pattern filter. This can be shown in the same fashion as the non-application clauses above. □

4.4 Compatibility

Once an initial simulation is established, we must demonstrate that each step of evaluation preserves simulation. We accomplish this by showing simulation preserving or simulation establishing properties for each relation used in typechecking, starting with compatibility. In particular we need to show that, for a given proof of value compatibility and a simulation on certain positions of the compatibility relation, a type compatibility proof exists such that the remaining places in the relation are also in some form of simulation. Using proofs of this form, we can eventually show the same properties for the operational semantics and constraint closure rules.

Our argument proceeds by induction on the size of the value compatibility proof, demonstrating that each rule that might've been used in the construction of the proof has a corresponding rule in the type system whose premises can be satisfied by the premises of the value compatibility rule and the given simulation. In most cases, this is quite simple, but we do require some supporting lemmas. We give those lemmas below followed by the establishing proof for the compatibility relations.

4.4.1 Combining Value Compatibility Proofs

The evaluation and type system have been carefully designed to parallel each other; this similarity helps to simplify the proof of soundness. But in order to allow for type refinement, lower-bounding constraints include sets of matching and non-matching patterns; no corresponding patterns exist on the value side. As a result, the Value Selection rule of value compatibility simply selects the value for a given variable; the corresponding Type Selection rule of type compatibility selects a type for a given type variable and then adds the restrictions of that type to the obligations of the proof. To accommodate this, we have defined the simulation relation in Figure 4.2 to include a proof that a corresponding value compatibility proof holds. This way, we can make use of this information when we consider these two rules in our inductive preservation proof below.

For this information to be useful, though, we require a specific property of value compatibility: that two proofs can, in a sense, be combined. The Type Selection rule requires a single proof of compatibility in its premises which simultaneously demonstrates the existing proof obligation as well as the restrictions imposed by the selected lower-bounding type. The Value Selection rule provides a proof of value compatibility showing the equivalent of the prior; the latter is shown by the proof of value compatibility given by the simulation. To be able to combine these proofs, we first need a simple supporting lemma:

Lemma 4.12. If $x\backslash E \sim \frac{\emptyset / P^+}{P^-} \Updownarrow E'$ then $E' = []$.

Proof. By induction on the size of the proof. □

We then show the property of proof combination (and thus are able to satisfy the premise of the Type Selection rule) in following lemma:

Lemma 4.13. Suppose $x\backslash E \sim \frac{P_1^{\diamond} / P_1^+}{P_1^-} \Updownarrow E_1'$ and $x\backslash E \sim \frac{P_2^{\diamond} / P_2^+}{P_2^-} \Updownarrow E_2'$ such that one of $P_1^{\diamond}$ or $P_2^{\diamond}$ is empty. Then $x\backslash E \sim \frac{P_1^{\diamond} \cup P_2^{\diamond} / P_1^+ \cup P_2^+}{P_1^- \cup P_2^-} \Updownarrow E_1' \,||\, E_2'$.

Proof. By induction on the sum of the sizes of the two proofs. We begin by strengthening the inductive hypothesis to include values: if $v\backslash E \sim \frac{P_1^{\diamond} / P_1^+}{P_1^-} \Updownarrow E_1'$ and $v\backslash E \sim \frac{P_2^{\diamond} / P_2^+}{P_2^-} \Updownarrow E_2'$

such that one of P_1^{ϕ} or P_2^{ϕ} is empty, then $v\backslash E \sim \frac{P_1^{\phi} \cup P_2^{\phi} / P_1^{+} \cup P_2^{+}}{P_1^{-} \cup P_2^{-}} \mathrel{\Phi} E_1' \,||\, E_2'$. We proceed by case analysis on the rules used, beginning with the cases for the variable form of the relation.

Leaf. Suppose wlog that the proof of $x\backslash E \sim \frac{P_1^{\phi} / P_1^{+}}{P_1^{-}} \mathrel{\Phi} E_1'$ uses the Leaf rule. Then $P_1^{\phi} = P_1^{+} = P_1^{-} = \emptyset$ and $E_1' = []$. Then a proof of $x\backslash E \sim \frac{P_1^{\phi} \cup P_2^{\phi} / P_1^{+} \cup P_2^{+}}{P_1^{-} \cup P_2^{-}} \mathrel{\Phi} E_1' \,||\, E_2'$ can be stated as a proof of $x\backslash E \sim \frac{P_2^{\phi} / P_2^{+}}{P_2^{-}} \mathrel{\Phi} E_2'$, which we are given by assumption; thus, this case is complete.

Variable Selection. Otherwise, both proofs must use the Variable Selection rule. We thus have proofs of $x = v \in E$, $v\backslash E \sim \frac{P_1^{\phi} / P_1^{+}}{P_1^{-}} \mathrel{\Phi} E_1'$, and $v\backslash E \sim \frac{P_2^{\phi} / P_2^{+}}{P_2^{-}} \mathrel{\Phi} E_2'$. We note that the premise $x = v \in E$ satisfied in both original proofs must be the same because E is well-formed. We have by assumption that one of P_1^{ϕ} or P_2^{ϕ} is $\emptyset$; thus, by the inductive hypothesis we have $v\backslash E \sim \frac{P_1^{\phi} \cup P_2^{\phi} / P_1^{+} \cup P_2^{+}}{P_1^{-} \cup P_2^{-}} \mathrel{\Phi} E_1' \,||\, E_2'$. This together with $x = v \in E$ above gives us $x\backslash E \sim \frac{P_1^{\phi} \cup P_2^{\phi} / P_1^{+} \cup P_2^{+}}{P_1^{-} \cup P_2^{-}} \mathrel{\Phi} E_1' \,||\, E_2'$ and this case is finished.

Given the above, it suffices to show the second part of this lemma: that the combination property can be shown for value proofs.

Binding. Suppose wlog that the proof of $v\backslash E \sim \frac{P_1^{\phi} / P_1^{+}}{P_1^{-}} \mathrel{\Phi} E_1'$ uses the Binding rule; then $\Phi = \phi$. As a result, case analysis on Figure 3.8 reveals that the proof of $v\backslash E \sim \frac{P_2^{\phi} / P_2^{+}}{P_2^{-}} \mathrel{\Phi} E_2'$ must also use the Binding rule. We therefore know that $x\backslash E \sim \frac{P_1^{\phi} / P_1^{+}}{P_1^{-}} \mathrel{\phi} E_1''$, $x\backslash E \sim \frac{P_2^{\phi} / P_2^{+}}{P_2^{-}} \mathrel{\phi} E_2''$, $E_1''' = [x = v \mid x\backslash F \in P_1^{\phi}]$, $E_2''' = [x = v \mid x\backslash F \in P_2^{\phi}]$, $E_1' = E_1'' \,||\, E_1'''$, and $E_2' = E_2'' \,||\, E_2'''$.

By the inductive hypothesis, we have that $v\backslash E \sim \frac{P_1^{\phi} \cup P_2^{\phi} / P_1^{+} \cup P_2^{+}}{P_1^{-} \cup P_2^{-}} \mathrel{\phi} E_1'' \,||\, E_2''$. For notation, let $j \in \{1, 2\}$ be the index of the proof such that $P_j^{\phi} = \emptyset$ and let $i \in \{1, 2\}$ such that $i \neq j$. Then by Lemma 4.12, we have $E_j'' = []$; thus, $v\backslash E \sim \frac{P_1^{\phi} \cup P_2^{\phi} / P_1^{+} \cup P_2^{+}}{P_1^{-} \cup P_2^{-}} \mathrel{\phi} E_i''$. Because $P_j^{\phi} = \emptyset$, we have that $E_j''' = []$ and so $E_j' = []$. Therefore, by the Binding rule, we have $v\backslash E \sim \frac{P_1^{\phi} \cup P_2^{\phi} / P_1^{+} \cup P_2^{+}}{P_1^{-} \cup P_2^{-}} \mathrel{\phi} E_i'$. Since $E_j' = []$, we have $E_1' \,||\, E_2' = E_i'$ and so this case is complete.

(Note that this requirement – that one of P_1^{ϕ} or P_2^{ϕ} be empty – is not solely to avoid concerns of ordering in E'. It is necessary to ensure that E', although possibly open, is otherwise well-formed.)

Empty Onion Pattern. Suppose wlog that the proof of $v\backslash E \sim \frac{P_1^{\phi} / P_1^{+}}{P_1^{-}} \mathrel{\Phi} E_1'$ uses the Empty Onion Pattern rule. Then, for some $P_1^{\phi\prime} \cup P_1^{+\prime} = \{ () \backslash F \}$ such that $P_1^{\phi} = P_1^{\phi\prime} \cup P_1^{\phi\prime\prime}$ and $P_1^{+} = P_1^{+\prime} \cup P_1^{+\prime\prime}$, we have $x\backslash E \sim \frac{P_1^{\phi\prime\prime} / P_1^{+\prime\prime}}{P_1^{-\prime\prime}} \mathrel{\Phi} E_1'$. By the inductive hypothesis, we know $v\backslash E \sim \frac{P_1^{\phi\prime\prime} \cup P_2^{\phi} / P_1^{+\prime\prime} \cup P_2^{+}}{P_1^{-} \cup P_2^{-}} \mathrel{\Phi} E_1' \,||\, E_2'$. Then by the Empty Onion Pattern rule, we know

$v\backslash E \sim \frac{P_1^{\phi} \cup P_2^{\phi} / P_1^{+} \cup P_2^{+}}{P_1^{-} \cup P_2^{-}} \updownarrow E_1' \,||\, E_2'$ and this case is complete.

Conjunction Pattern. Suppose wlog that the proof of $v\backslash E \sim \frac{P_1^{\phi} / P_1^{+}}{P_1^{-}} \updownarrow E_1'$ uses the Conjunction Pattern rule. This case proceeds exactly as in the case of the Empty Onion Pattern rule. In particular, the rule adds a subset of a set comprehension for each of the binding, positive, and negative patterns. Because, for instance, $P_1^+ \cup P_2^+$ is strictly larger than P_1^+ and because of the form of the set comprehensions, the combined proof may always select the same subset.

Conjunction Weakening. Suppose wlog that the proof of $v\backslash E \sim \frac{P_1^{\phi} / P_1^{+}}{P_1^{-}} \updownarrow E_1'$ uses the Conjunction Weakening rule; then $\updownarrow = \downarrow$ and $v\backslash E \sim \frac{P_1^{\phi} \cup P_3^{\phi} / P_1^{+} \cup P_3^{+}}{P_1^{-} \cup P_3^{-}} \updownarrow E_1'$ for some P_3^{ϕ}, P_3^{+}, and P_3^{-}. By the inductive hypothesis, we have $v\backslash E \sim \frac{P_1^{\phi} \cup P_2^{\phi} \cup P_3^{\phi} / P_1^{+} \cup P_2^{+} \cup P_3^{+}}{P_1^{-} \cup P_2^{-} \cup P_3^{-}} \updownarrow E_1' \,||\, E_2'$. By the Conjunction Weakening rule, we have $v\backslash E \sim \frac{P_1^{\phi} \cup P_2^{\phi} / P_1^{+} \cup P_2^{+}}{P_1^{-} \cup P_2^{-}} \updownarrow E_1' \,||\, E_2'$. The premises which originally held on the Conjunction Weakening rule in the first proof still hold in this use in the combined proof, so this case is complete.

Empty Onion. Suppose wlog that the proof of $v\backslash E \sim \frac{P_1^{\phi} / P_1^{+}}{P_1^{-}} \updownarrow E_1'$ uses the Empty Onion rule. Then $v = \texttt{()}$, $P_1^{\phi} = P_1^{+} = \emptyset$, $E_1' = []$, $\updownarrow = \uparrow$, and $\textsc{CtrPat}(P_1^-)$. By case analysis on Figure 3.8, we see that this is the only remaining case which acts on the value $\texttt{()}$, so the proof of $v\backslash E \sim \frac{P_2^{\phi} / P_2^{+}}{P_2^{-}} \updownarrow E_2'$ must also use the Empty Onion rule; thus, $P_2^{\phi} = P_2^{+} = \emptyset$, $E_2' = []$, and $\textsc{CtrPat}(P_2^-)$. By Definition 3.8, we have $\textsc{CtrPat}(P_1^- \cup P_2^-)$; thus, by the Empty Onion rule, we have $\texttt{()}\,\backslash E \sim \frac{\emptyset / \emptyset}{P_1^{-} \cup P_2^{-}} \uparrow []$ and this case is finished.

Label. Suppose wlog that the proof of $v\backslash E \sim \frac{P_1^{\phi} / P_1^{+}}{P_1^{-}} \updownarrow E_1'$ uses the Label rule. Then $v = l\ x$, $P_1^{\phi} = l\ P_1^{\phi\prime}$, $P_1^{+} = l\ P_1^{+\prime}$, $P_1^{-} = (l\ P_1^{-\prime}) \cup P_1^{-\prime\prime}$, $\updownarrow = \uparrow$, $x\backslash E \sim \frac{P_1^{\phi\prime} / P_1^{+\prime}}{P_1^{-\prime}} \downarrow E_1'$, $\textsc{CtrPat}(P_1^{-\prime\prime})$, and $\{l\ x'\backslash F \in P_1^{-\prime\prime}\} = \emptyset$. As with the Empty Onion case, case analysis on Figure 3.8 reveals that this the only remaining case which acts on the value $l\ x$, so the proof of $v\backslash E \sim \frac{P_2^{\phi} / P_2^{+}}{P_2^{-}} \updownarrow E_2'$ must also use the Label rule; thus, $P_2^{\phi} = l\ P_2^{\phi\prime}$, $P_2^{+} = l\ P_2^{+\prime}$, $P_2^{-} = (l\ P_2^{-\prime}) \cup P_2^{-\prime\prime}$, $x\backslash E \sim \frac{P_2^{\phi\prime} / P_2^{+\prime}}{P_2^{-\prime}} \downarrow E_2'$, $\textsc{CtrPat}(P_2^{-\prime\prime})$, and $\{l\ x'\backslash F \in P_2^{-\prime\prime}\} = \emptyset$.

Given the above, we aim to use the Label rule to show the combined proof. By the inductive hypothesis, we have $x\backslash E \sim \frac{P_1^{\phi\prime} \cup P_2^{\phi\prime} / P_1^{+\prime} \cup P_2^{+\prime}}{P_1^{-\prime} \cup P_2^{-\prime}} \downarrow E_1' \,||\, E_2'$. By Definition 3.8, we have that $\textsc{CtrPat}(P_1^{-\prime\prime} \cup P_2^{-\prime\prime})$. Because $\{l\ x'\backslash F \in P_1^{-\prime\prime}\} = \emptyset$ and $\{l\ x'\backslash F \in P_2^{-\prime\prime}\} = \emptyset$, we have $\{l\ x'\backslash F \in P_1^{-\prime\prime} \cup P_2^{-\prime\prime}\} = \emptyset$. Thus, by the Label rule, we have $l\ x\backslash E \sim \frac{l\,(P_1^{\phi\prime} \cup P_2^{\phi\prime}) / l\,(P_1^{+\prime} \cup P_2^{+\prime})}{(l\,(P_1^{-\prime} \cup P_2^{-\prime}) \cup P_1^{-\prime\prime} \cup P_2^{-\prime\prime}} \uparrow$ $E_1' \,||\, E_2'$ and this case is complete.

Onion. Suppose wlog that the proof of $v\backslash E \sim \frac{P_1^{\phi} / P_1^{+}}{P_1^{-}} \updownarrow E_1'$ uses the Onion rule. This case proceeds exactly as in the Label case, using the inductive hypothesis on the pairs of proofs representing each side of the onion.

□

4.4.2 Discarding Bindings

The above combination lemma allows us to use the compatibility proof implied by simulation in order to show that the two compatibility relations preserve simulation. For this to be useful, we must be able to satisfy the simulation relation's requirement of a compatibility proof. As shown in Lemma 4.11, the base case (when a value's corresponding type has never been filtered) is trivial. As a value proceeds through pattern matches, however, it is simulated by non-trivial filtered types. Value compatibility creates bindings in the Binding rule and the selected values must be shown to be compatible with patterns corresponding to these filtered types.

Fortunately, a sufficient proof is already present as the premise of the Binding rule. To use it in simulation, however, we must discard its binding information. This leads to the following supporting lemma:

Lemma 4.14. If $x\backslash E \sim \frac{P^{\Phi}/P^{+}}{P^{-}} \Updownarrow E'$ then $x\backslash E \sim \frac{\emptyset/P^{\Phi}\cup P^{+}}{P^{-}} \Updownarrow []$.

Proof. By induction on the size of the proof. □

4.4.3 Establishing the Binding Simulation

Using the above supporting lemmas, we can now show the critical property of the compatibility relations: that they establish a relationship between the value bindings and the type bindings which we use later in the application process. The nature of this relationship is somewhat subtle. The variable side of each binding must have properties quite similar to bisimulation – a one-to-one correspondence – in order for the polyinstantiation lemmas (discussed below in Section 4.6) to yield usable conclusions. But establishing a full bisimulation over the binding sets is impossible; there's no way to guarantee that the value argument to compatibility (which, in practice, arrives from an argument during application) is union-free and bisimilar to its type. For this reason, we require a sort of "one-and-a-half similar" relation to address the special case of bindings; we call this the *binding simulation*. Thankfully, this definition is comparatively terse:

Definition 4.15 (Binding Simulation)**.** We write $E' \;{}_{E}\overset{\preccurlyeq}{\approx}_{M}\; C$ to indicate that both of the following are true:

- $\forall s \in E'.\, \exists c \in C.\, s \;{}_{M}\!\preccurlyeq_{E}\; c$
- $\forall c \in C.\, \exists s \in E'.\, s \;{}_{M}\!\preccurlyeq_{E}\; c.$

Using this definition and the previously-given supporting lemmas, we can show that the compatibility relations establish this binding simulation over their respective bindings.

Lemma 4.16. Suppose that $E \;{}_{E}{\preccurlyeq}_{M}\; C$, $P^{\oplus} \;{}_{E}{\approx}_{M}\; \Pi^{\oplus}$, $P^{+} \;{}_{E}{\approx}_{M}\; \Pi^{+}$, and $P^{-} \;{}_{E}{\approx}_{M}\; \Pi^{-}$. Then $x\backslash E \sim \frac{P^{\oplus}/P^{+}}{P^{-}} \phi\, E'$ and $x \;{}_{E}{\preccurlyeq}_{M}\; \alpha$ together imply that $\alpha\backslash C \sim \frac{\Pi^{\oplus}/\Pi^{+}}{\Pi^{-}} \phi\, C'$ such that $E' \;{}_{E}{\precapprox}_{M}\; C'$.

Proof. By induction on the size of the proof. We begin by strengthening the inductive hypothesis to include values: $v\backslash E \sim \frac{P^{\oplus}/P^{+}}{P^{-}} \phi\, E'$ and $v \;{}_{E}{\preccurlyeq}_{M}\; \tau$ together imply that $\tau\backslash C \sim \frac{\Pi^{\oplus}/\Pi^{+}}{\Pi^{-}} \phi\, C'$ such that $E' \;{}_{E}{\precapprox}_{M}\; C'$. We then proceed by case analysis on the proof rule used.

Leaf. If the Leaf rule is used, then this property is immediate: $\emptyset \;{}_{E}{\precapprox}_{M}\; \emptyset$ and $[] \;{}_{E}{\precapprox}_{M}\; \emptyset$ for all E and M.

Variable Selection. If the Variable Selection rule is used, then $x = v \in E$ and $v\backslash E \sim \frac{P^{\oplus}/P^{+}}{P^{-}} \phi\, E'$. Because $E \;{}_{E}{\preccurlyeq}_{M}\; C$ and $x = v \in E$, Definition 4.2 gives us that $\tau|^{\Pi^{+\prime}}_{\Pi^{-\prime}} <: \alpha \in C$ such that $v \;{}_{E}{\preccurlyeq}_{M}\; \tau$, $P^{+\prime} \;{}_{E}{\approx}_{M}\; \Pi^{+\prime}$, $P^{-\prime} \;{}_{E}{\approx}_{M}\; \Pi^{-\prime}$, and $v\backslash E \sim \frac{\emptyset/P^{+\prime}}{P^{-\prime}} \phi\, []$. By Lemma 4.13, we have $v\backslash E \sim \frac{P^{\oplus}/P^{+}\cup P^{+\prime}}{P^{-}\cup P^{-\prime}} \phi\, E'$. By the inductive hypothesis, this gives us $\tau\backslash C \sim \frac{\Pi^{\oplus}/\Pi^{+}\cup\Pi^{+\prime}}{\Pi^{-}\cup\Pi^{-\prime}} \phi\, C'$ such that $E' \;{}_{E}{\precapprox}_{M}\; C'$. We thus satisfy the premises for the Type Selection rule of type compatibility (Figure 3.14) and this case is finished.

Conjunction Weakening. If the Conjunction Weakening rule is used, then $v\backslash E \sim \frac{P^{\oplus}\cup P^{\oplus\prime}/P^{+}\cup P^{+\prime}}{P^{-}\cup P^{-\prime}} \phi\, E'$. By the inductive hypothesis, we have $\tau\backslash C \sim \frac{\Pi^{\oplus}\cup\Pi^{\oplus\prime}/\Pi^{+}\cup\Pi^{+\prime}}{\Pi^{-}\cup\Pi^{-\prime}} \phi\, C'$ such that corresponding places simulate. By the Conjunction Weakening rule of the type compatibility relation, $\tau\backslash C \sim \frac{\Pi^{\oplus}/\Pi^{+}}{\Pi^{-}} \phi\, C'$ and this case is finished.

Binding. If the Binding rule is used, then $v\backslash E \sim \frac{P^{\oplus}/P^{+}}{P^{-}} \phi\, E''_1$, $E''_2 = [x = v \mid x\backslash F \in P^{\oplus}]$, and $E' = E''_1 \,||\, E''_2$. We have $v \;{}_{E}{\preccurlyeq}_{M}\; \tau$ by premise. By the inductive hypothesis, we have $\tau\backslash C \sim \frac{\Pi^{\oplus}/\Pi^{+}}{\Pi^{-}} \phi\, C''_1$ such that $E''_1 \;{}_{E}{\precapprox}_{M}\; C''_1$. Let $C' = C''_1 \cup C''_2$ where $C''_2 = \{\tau|^{\Pi^{\oplus}\cup\Pi^{+}}_{\Pi^{-}} <: \alpha \mid \alpha\backslash F \in \Pi^{\oplus}\}$; then, by the Binding rule of type compatibility, we have $\tau\backslash C \sim \frac{\Pi^{\oplus}/\Pi^{+}}{\Pi^{-}} \phi\, C'$. It remains in this case to show that $E' \;{}_{E}{\precapprox}_{M}\; C'$.

We next aim to demonstrate that $E''_2 \;{}_{E}{\precapprox}_{M}\; C''_2$. By $P^{\oplus} \;{}_{E}{\approx}_{M}\; \Pi^{\oplus}$ we have that there exists orderings $P^{\oplus} = \overset{n}{\overbrace{p}}$ and $\Pi^{\oplus} = \overset{n}{\overbrace{\pi}}$ such that, for all i $\underbrace{\text{in } \{1..n\}, \text{ we}}_{n}$ have $p_i = x_i\backslash F_i \;{}_{E}{\approx}_{M}\; \alpha_i\backslash\Psi_i = \pi_i$. Observe that $E''_2 = \overset{n}{\overrightarrow{x_i = v}}$ and that $C''_2 = \tau|^{\Pi^{\oplus}\cup\Pi^{+}}_{\Pi^{-}} <: \alpha_i$. So we need to show that, for all i in $\{1..n\}$, we have $x_i = v \;{}_{E}{\preccurlyeq}_{M}\; \tau|^{\Pi^{\oplus}\cup\Pi^{+}}_{\Pi^{-}} <: \alpha_i$.

Select wlog some $i \in \{1..n\}$. We have $x_i \;{}_{E}{\preccurlyeq}_{M}\; \alpha_i$ from $P^{\oplus} \;{}_{E}{\approx}_{M}\; \Pi^{\oplus}$ above. We have $v \;{}_{E}{\preccurlyeq}_{M}\; \tau$ by premise. From $v\backslash E \sim \frac{P^{\oplus}/P^{+}}{P^{-}} \phi\, E''_1$, Lemma 4.14 gives us $v\backslash E \sim \frac{\emptyset/P^{\oplus}\cup P^{+}}{P^{-}} \phi\, []$. These facts together give us that $x_i = v \;{}_{E}{\preccurlyeq}_{M}\; \tau|^{\Pi^{\oplus}\cup\Pi^{+}}_{\Pi^{-}} <: \alpha_i$ for any i in $\{1..n\}$, so $E''_2 \;{}_{E}{\precapprox}_{M}\; C''_2$.

The above demonstrates that $E''_1 \;{}_{E}{\precapprox}_{M}\; C''_1$ and $E''_2 \;{}_{E}{\precapprox}_{M}\; C''_2$. It is then trivial by

Definition 4.15 that $E_1 \,||\, E_2'' \;{}_{E}{\precapprox}_{M}\; C_1'' \cup C_2''$; restated, we have $E' \;{}_{E}{\precapprox}_{M}\; C'$ and this case is complete.

Remaining Cases. The remaining cases proceed much as in the above. For each rule used by the value compatibility proof, we use its premises together with the simulations and bisimulations taken as premises of this lemma to demonstrate the premises of the corresponding type compatibility rule. These cases are simpler than those above; they do not require any supporting lemmas regarding bindings. □

4.5 Matching

Next, we demonstrate that the application matching relations preserve simulation. This proceeds similar to the argument for compatibility above; indeed, it's somewhat simpler since the relations are more similar. The only caveat is the use of the $\textsc{Sensible}(-,-,-,-)$ function in each type application matching rule, which we address with the following supporting lemma:

Lemma 4.17. If $x = v \;{}_{E}{\preccurlyeq}_{M}\; \tau|^{\Pi^+}_{\Pi^-} <: \alpha$ and $E \;{}_{E}{\preccurlyeq}_{M}\; C$ then $\textsc{Sensible}(\tau, \Pi^+, \Pi^-, C)$.

Proof. By Definition 4.2, we have $v \backslash E \sim^{\emptyset / P^+}_{P^-} \mathrel{\circ\!\!\!\downarrow} []$ for some $P^+ \;{}_{E}{\approx}_{M}\; \Pi^+$ and $P^- \;{}_{E}{\approx}_{M}\; \Pi^-$. By Lemma 4.16, this gives us that $\tau \backslash C \sim^{\emptyset / \Pi^+}_{\Pi^-} \mathrel{\circ\!\!\!\downarrow} \emptyset$, so by Definition 3.28 we have $\textsc{Sensible}(\tau, \Pi^+, \Pi^-, C)$. □

The alignment of the application matching relations is then shown as follows:

Lemma 4.18. Suppose $x_0 \;{}_{E}{\preccurlyeq}_{M}\; \alpha_0$, $x_1 \;{}_{E}{\preccurlyeq}_{M}\; \alpha_1$, $E \;{}_{E}{\preccurlyeq}_{M}\; C$, and $P_1 \;{}_{E}{\approx}_{M}\; \Pi_1$. Then $x_0\ x_1 \;{}^{P_1}_{E}{\rightsquigarrow}^{P_2}_{\odot}\; e \mathbin{@} E''$ implies that $\alpha_0\ \alpha_1 \;{}^{\Pi_1}_{C}{\rightsquigarrow}^{\Pi_2}_{\odot}\; \alpha' \backslash C' \mathbin{@} C''$ such that $e \;{}_{E}{\approx}_{M}\; \alpha' \backslash C'$, $E'' \;{}_{E}{\precapprox}_{M}\; C''$, and $P_2 \;{}_{E}{\approx}_{M}\; \Pi_2$.

Proof. By the same strategy as Lemma 4.16, using the simulation and the premises of value application matching to show the premises of type application matching. The only premises which are not immediately satisfied in this manner are those premises of the form $\textsc{Sensible}(\tau, \Pi^+, \Pi^-, C)$. In each rule, that premise appears alongside a premise of the form $\tau|^{\Pi^+}_{\Pi^-} <: \alpha \in C$, which we have previously shown from a $x = v \in E$ which it simulates. As a result, Lemma 4.17 satisfies this additional sensibility predicate. □

4.6 Polyinstantiation

During constraint closure, we must be able to show alignment between the freshening of variables in the evaluation system and the polyinstantiation of type variables in the type system. It is not possible to show a simulation-preserving alignment (since bisimulation is necessary) nor is it possible to show a bisimulation-preserving alignment (since type variable freshening is not injective). For our purposes, however, it is sufficient to show that a *bisimulating* pair is reduced to a *simulating* pair by these α-substitution functions.

To understand why bisimulation is a necessary precondition of this alignment, we consider the definition of $\textsc{bv}$ and $\textsc{fv}$ (Definition 3.6). A variable is bound in an expression if it appears on the left-hand side of a clause; these are the variables which are freshened by $\boldsymbol{\alpha}(x, -)$. In bisimulation, these clauses are guaranteed to be one-to-one with the constraints that bisimulate them; that is, bisimulation establishes an injective variable mapping as well as the one-to-one correspondence: upper-bounding type variables map injectively to and from the value variables appearing on the left side of clauses. Due to this injective mapping, we can be sure that the restricted variable replacement $\sigma|\_$ will replace variables in the same way as $\boldsymbol{\alpha}(-,-)$. Simulation by itself is insufficient because two value variables might be mapped onto the same type variable. If one of these value variables is bound and the other is free, there is no polyinstantiation for the type variable which preserves alignment.

We formalize the above property of bisimulation as follows:

Lemma 4.19. If $e \;{}_E{\approx}_M\; C$, x appears in e, and $M(x) = \alpha$, then $x \in \textsc{bv}(e)$ if and only if $\alpha \in \textsc{bv}(C)$.

Proof. Define $e = \overset{n}{\overrightarrow{s}}$ and $C = \overset{m}{\overbrace{c}}$. If some $x \in \textsc{bv}(e)$, then x appears on the left of some s_i. Because $e \;{}_E{\approx}_M\; C$, we have that some $s_i \;{}_E{\approx}_M\; c_j$. Further, we have that $\alpha = M(x)$ where α appears on the right of c_j. So by Definition 4.1, if $x \in \textsc{bv}(e)$ then $\alpha \in \textsc{bv}(C)$.

So it remains to show that, for any x appearing in e such that $M(x) = \alpha$, $x \notin \textsc{bv}(e)$ implies that $\alpha \notin \textsc{bv}(C)$. Suppose for contradiction that $\alpha \in \textsc{bv}(C)$. Then by Definition 3.20, some $c \in C$ is of the form $\ldots <: c$. Because $e \;{}_E{\approx}_M\; C$, we have that there is some $s \in e$ of the form $x' = \ldots$ such that $s \;{}_E{\approx}_M\; c$; thus, $x' \in \textsc{bv}(e)$ and also $M(x') = \alpha$. Because M is injective, $x = x'$. But $x \notin \textsc{bv}(e)$ and $x' \in \textsc{bv}(e)$, which is a contradiction; thus, $\alpha \notin \textsc{bv}(C)$ and we are finished. $\square$

Ultimately, however, the freshening of type variables may map two value variables onto the same type variable; in this circumstance, it is possible to show simulation of the result but not bisimulation. We therefore state the (slightly lossy) alignment between the two functions as follows:

Lemma 4.20. Suppose $e \;{}_E{\approx}_{M_1}\; \alpha_1 \backslash C$ and $E' \;{}_E\overset{\precsim}{\approx}_{M_1}\; C'$ and $x_0 \;{}_E{\approx}_{M_1}\; \alpha_0$. Let $\boldsymbol{\alpha}(x_0, -) = \varsigma$ and let $\Phi(\alpha_0, -) = \sigma$. Let $e'' = E' \,||\, e$ and let $C'' = C' \cup C$. Finally, suppose that the codomain of ς is disjoint from the domain of M_1. Then $\varsigma\big|_{e''}(e) \;{}_E{\precsim}_{M_3}\; \sigma\big|_{C''}(\alpha_1 \backslash C)$ and $\textsc{BFresh}(\varsigma\big|_{e''}, E') \;{}_E{\precsim}_{M_3}\; \textsc{TFresh}(\sigma\big|_{C''}, C')$ for some $M_3 \supseteq M_1$.

Proof. By showing that it suffices to construct M_3 by extending it with the same replacements performed on e and C. We define M_2 such that $M_2 \circ \varsigma|_{e''} = \sigma|_{C''} \circ M_1$; this is to say, let $M_2(x) = \sigma|_{C''}(M_1((\varsigma|_{e''})^{-1}(x)))$. (The replacement function $(\varsigma|_{e'})^{-1}$ is a well-defined partial function because ς is injective.)

We begin by showing that $\varsigma|_{e''}(e)\ {}_{E}{\preccurlyeq}_{M_2}\ \sigma|_{C''}(\alpha_1\backslash C)$; we simultaneously show that, for any x in the domain of both M_1 and M_2, $M_1(x) = M_2(x)$. By Definition 4.1, bisimulation preserves all structure except the ordering of expressions and the uniqueness of variables. α-substitutions do not affect expression ordering, so it suffices to show that, for any x and α in e and C (respectively) such that $x\ {}_{E}{\preccurlyeq}_{M_1}\ \alpha$, we have $\varsigma|_{e''}(x)\ {}_{E}{\preccurlyeq}_{M_2}\ \sigma|_{C''}(\alpha)$. Let $\varsigma|_{e''}(x) = x'$ and let $\sigma|_{C''}(\alpha) = \alpha'$.

For any x appearing in e and its corresponding α, there are two cases: either x is bound by e'' or it is not. If it is, then Lemma 4.19 gives us that α is bound by C''. We have $M_2(x') = \sigma|_{C''}(M_1((\varsigma|_{e''})^{-1}(x'))) = \sigma|_{C''}(M_1(x)) = \sigma|_{C''}(\alpha) = \alpha'$ and so $x'\ {}_{E}{\preccurlyeq}_{M_2}\ \alpha'$. Because x' is in the codomain of ς, we have by assumption that it is not part of the domain of M_1; thus, this case is finished.

Otherwise, x is not bound by e'' and so, by Lemma 4.19, α is not bound by C''. Thus, we have $x = x'$ and $\alpha = \alpha'$. We proceed as above and thus have $x'\ {}_{E}{\preccurlyeq}_{M_2}\ \alpha'$. In this case, $x = x'$ is in the domain of both M_1 and M_2, but $M_1(x) = \alpha = \alpha' = M_2(x')$ and so this case is finished.

Given the above, we have that $\varsigma|_{e''}(e)\ {}_{E}{\preccurlyeq}_{M_2}\ \sigma|_{C''}(\alpha_1\backslash C)$ and that $M_1(x) = M_2(x)$ for any x in both M_1 and M_2. Let $M_3 = M_1 \cup M_2$; by Lemma 4.4, we thus have $\varsigma|_{e''}(e)\ {}_{E}{\preccurlyeq}_{M_3}\ \sigma|_{C''}(\alpha_1\backslash C)$.

It remains to show that $\textsc{BFresh}(\varsigma|_{e''}, E')\ {}_{E}{\preccurlyeq}_{M_3}\ \textsc{TFresh}(\sigma|_{C''}, C')$. In this case, we are using the binding simulation $E'\ {}_{E}{\precsim}_{M_1}\ C'$ instead of bisimulation, but the freshening functions do not freshen the value/type side of their arguments. Because those parts are untouched and because binding simulation already requires each type and value to be in simulation, it only remains to show that the value and type variables are in simulation after they are freshened and polyinstantiated. This is demonstrated via the same strategy as in e and C above. □

4.7 Preservation

The previous sections demonstrate that pairs of relations in the evaluation and type systems (e.g. the pair of value compatibility and type compatibility) preserve certain simulation properties. One pair of relations remains: the (alternate) small step relation and the constraint closure relation. We begin by providing a set of supporting lemmas regarding variable freshness and then show our preservation lemma: that each small step is simulated by some constraint closure step.

4.7.1 Freshness

In order to prove our preservation lemma, we must first show some properties about the freshness of variables in the operational semantics. This is necessary to satisfy a precondition of Lemma 4.20: that the simulation mapping M does not contain the variables produced by $\boldsymbol{\alpha}(x, -)$ by the time we reach the call site x.

The definition of $\boldsymbol{\alpha}(-, -)$ requires that a strict partial order exists which represents the lineage of the resulting variables. We begin by defining some notation for that partial order:

Definition 4.21 (Fresh Variable Ordering)**.** We write $x \lessdot x'$ if (1) x' is in the codomain of $\boldsymbol{\alpha}(x, -)$ or (2) there exists some x'' such that $x \lessdot x''$ and x' is in the codomain of $\boldsymbol{\alpha}(x'', -)$. We write $x \not\lessdot x'$ if this is not the case.

We then make some simple observations regarding this partial order; these lemmas are helpful in simplifying our freshness lemma. The first lemma is quite similar to the contrapositive of transitivity:

Lemma 4.22. If $x_1 \lessdot x_2$ and $x_1 \not\lessdot x_3$, then $x_2 \not\lessdot x_3$.

Proof. Trivial by contradiction. □

Our second observation regarding the partial order is that, due to its injectivity, two variables freshened from the same call site cannot ultimately be freshened from each other.

Lemma 4.23. If x_1 and x_2 both appear in the codomain of $\boldsymbol{\alpha}(x_0, -)$ then w.l.o.g. $x_1 \not\lessdot x_2$.

Proof. Assume for contradiction that $x_1 \lessdot x_2$. By Definition 4.21, this implies one of two cases: (1) that x_2 is in the codomain of $\boldsymbol{\alpha}(x_1, -)$ or (2) that there exists some x' such that $x_1 \lessdot x'$ and x_2 is in the codomain of $\boldsymbol{\alpha}(x', -)$. We show a contradiction in each case.

In the first case, we know that $\boldsymbol{\alpha}(-, -)$ is injective. Because the codomain of $\boldsymbol{\alpha}(x_0, -)$ includes x_2, the codomain of $\boldsymbol{\alpha}(x_1, -)$ cannot. By contradiction, this case is complete.

In the second case, we use the same injectivity. Because the codomain of $\boldsymbol{\alpha}(x_0, -)$ includes x_2, we must conclude that $x' = x_0$. Then we conclude that $x_1 \lessdot x_0$. But $x_0 \lessdot x_1$ and $\lessdot$ is a strict partial order, which is a contradiction; we are therefore finished. □

We are now prepared to give our freshness lemma: that, given any unevaluated call site x_0 in the expression, none of the variables reserved for that call site appear in the expression.

Lemma 4.24. Given an initial expression e_0, let $e_0 \longrightarrow^1 e_1 \longrightarrow^1 \ldots \longrightarrow^1 e_n$. For any clause $x_0 \,\hat{=}\, x_1\ x_2$ appearing in e_n and any variable x'' appearing in e_n, we have $x_0 \not\lessdot x''$.

Proof. By induction on the length of the small step sequence. The base case is satisfied by definition: $\boldsymbol{\alpha}(-,-)$ is defined to have a codomain which is disjoint from the variables appearing in e_0.

For the inductive step, we consider $e_i \longrightarrow^1 e_j$ where $j = i + 1$. If the Variable Lookup rule applies, then all variables and application clauses appearing in e_j appear in e_i. Thus, this property holds.

If the Application rule applies, then $e_i = E \,||\, [x_0 \,\hat{=}\, x_1\ x_2] \,||\, e''$ and $e_j = E \,||\, e' \,||\, e''$. The rule gives us that the subexpression e' is the body of a function and some associated bindings which have been freshened by $\boldsymbol{\alpha}(x_0, -)$. To demonstrate that e_j meets our induction predicate, it suffices to show that, for each application clause binding some x'_0 and each variable x'' in e_j, we have $x'_0 \not\lessdot x''$. Because E by construction does not contain application clauses, it suffices to consider the following three cases:

1. The application clause appears in e'' and the variable appears in either E or e'',
2. The application clause appears in e' and the variable appears in either E or e'', or
3. The variable appears in e'.

Case 1: Choose w.l.o.g. an application clause binding some x'_0 appearing in e'' and a variable x'' appearing in E or e''. Because E and e'' are subexpressions of e_i, $x'_0 \not\lessdot x''$ is given by the inductive hypothesis.

Case 2: Choose w.l.o.g. an application clause binding some x'_0 appearing in e' and a variable x'' appearing in E or e''. Because x'_0 is bound by a clause in e', we know it was replaced by this small step and so $x_0 \lessdot x'_0$. Because x'' appears in either E or e'' and an application clause appears in e_i which defines x_0, we have by the induction hypothesis that $x_0 \not\lessdot x''$. By Lemma 4.22, we have $x'_0 \not\lessdot x''$.

Case 3: Choose w.l.o.g. an application clause binding some x'_0 appearing in e'' and a variable x'' appearing in e'. We have two subcases: either (1) x'' is bound by e' or (2) it is not. In subcase 1, x'' is bound by e' and so was replaced in this small step; thus, we know x'' appears immediately in the codomain of $\boldsymbol{\alpha}(x_0, -)$. We know x'_0 appears as the variable bound by an application clause in e' and so was replaced in this small step; thus, x'_0 also appears immediately in the codomain of $\boldsymbol{\alpha}(x_0, -)$. By Lemma 4.23, we therefore have $x'_0 \not\lessdot x''$.

In subcase 2, x'' is not bound by e' and so it was not replaced in this small step. Because it appears free in the original (pre-replacement) expression and because e_i is closed, the definition of x'' must appear in E. We therefore have that both x'' and the application clause binding x'_0 appear in e_i and so, by the induction hypothesis, $x'_0 \not\leqslant x''$. $\square$

4.7.2 Preservation Lemma

Here, we provide the preservation lemma itself. This lemma shows that each small step of evaluation is aligned with a constraint closure step such that simulation is preserved. In order to do this, the simulation itself must be using the same environment as the evaluation step: the largest prefix of the current expression which is an environment. We therefore provide the following definition for the purpose of stating the lemma:

Definition 4.25 (Maximal Environment)**.** Let $e' = E_0 \,||\, e_0$. Then E_0 is *maximal for* e' iff $e' = E_0 \,||\, E_1 \,||\, e_1$ only when $E_1 = []$. (That is, E_0 is the largest prefix of e' of the form E.) We may write that "E_0 is *maximal*" if e' is understood from context.

Using this definition of maximality, we can now give our preservation lemma: for any small step, there is a corresponding closure step which preserves simulation.

Lemma 4.26 (Preservation)**.** Suppose that $E_0 \,||\, e_0 \longrightarrow^1 E_3 \,||\, e_3$ (for maximal E_0 and E_3) and that $E_0 \,||\, e_0 \;{}_{E_0}{\preccurlyeq}_{M_0}\; C_0$. Then there exists some C_3 and M_3 such that $C_0 \Longrightarrow C_3$ and $E_3 \,||\, e_3 \;{}_{E_3}{\preccurlyeq}_{M_3}\; C_3$.

Proof. By showing that, given a simulation, each premise of an operational semantics rule proves a corresponding premise of a constraint closure rule. We proceed by case analysis on the operational semantics rule used.

Variable Lookup. If the Variable Lookup rule is used, then $e_0 = [x_2 = x_1] \,||\, e_3$ and $E_3 = E_0 \,||\, [x_2 = v]$ where $x_1 = v \in E_0$. By Definition 4.2 and $E_0 \,||\, e_0 \;{}_{E_0}{\preccurlyeq}_{M_0}\; C_0$, the fact that $x_2 = x_1 \in E_0 \,||\, e_0$ gives us that some $\alpha_1 <: \alpha_2 \in C_0$ such that $x_1 \;{}_{E_0}{\preccurlyeq}_{M_0}\; \alpha_1$ and $x_2 \;{}_{E_0}{\preccurlyeq}_{M_0}\; \alpha_2$. Likewise, because $x_1 = v \in E_0 \,||\, e_0$, there is some $\tau|^{\Pi^+}_{\Pi^-} <: \alpha_1 \in C_0$ for which we know the following: $x_1 \;{}_{E_0}{\preccurlyeq}_{M_0}\; \alpha_1$, $v \;{}_{E_0}{\preccurlyeq}_{M_0}\; \tau$, $P^+ \;{}_{E_0}{\approx}_{M_0}\; \Pi^+$, $P^- \;{}_{E_0}{\approx}_{M_0}\; \Pi^-$, and $v \backslash E_0 \sim \frac{\emptyset / P^+}{P^-} \oint []$.

Let $C_3 = C_0 \cup \{\tau|^{\Pi^+}_{\Pi^-} <: \alpha_2\}$. By the Transitivity rule in Figure 3.16, we have that $C_0 \Longrightarrow C_3$. If we let $M_3 = M_0$, then it suffices to show that $E_3 \,||\, e_3 \;{}_{E_3}{\preccurlyeq}_{M_0}\; C_3$. Because $E_3 = E_0 \,||\, [x_2 = v]$, it suffices by Lemma 4.8 to show that $E_0 \,||\, [x_2 = v] \,||\, e_3 \;{}_{E_0}{\preccurlyeq}_{M_0}\; C_3$. By $E_0 \,||\, e_0 \;{}_{E_0}{\preccurlyeq}_{M_0}\; C_0$, we already have proofs of simulation for all clauses except the new clause: $x_2 = v$. So it suffices to show that there exists some constraint $c \in C_3$ such that $x_2 = v \;{}_{E_0}{\preccurlyeq}_{M_0}\; c$.

We choose $c = \tau|^{\Pi^+}_{\Pi^-} <: \alpha_2$. By Definition 4.2, the premises of $x_2 = v \;{}_{E_0}{\preccurlyeq}_{M_0}\; \tau|^{\Pi^+}_{\Pi^-} <: \alpha_2$ are the same as the premises of $x_1 = v \;{}_{E_0}{\preccurlyeq}_{M_0}\; \tau|^{\Pi^+}_{\Pi^-} <: \alpha_1$ (which we have from above) except that we must show $x_2 \;{}_{E_0}{\preccurlyeq}_{M_0}\; \alpha_2$; we have this from $x_2 = x_1 \in E_0 \,||\, e_0$ and the original simulation. Therefore, $E_3 \,||\, e_3 \;{}_{E_0}{\preccurlyeq}_{M_3}\; C_3$ and we are finished.

Application. The only other case is when the Application rule applies. Then $e_0 = [x_2 = x_0\ x_1] \,||\, e_1$, $e_3 = e'' \,||\, [x_2 = \mathrm{RV}(e'')] \,||\, e_1$, and we know the following from the premises: $x_0\ x_1\ {}_{E_0}\!\overset{\emptyset}{\rightsquigarrow}{}^{P}_{\bullet}\ e'\ @\ E'$, $\varsigma = \boldsymbol{\alpha}(x_2, -)$, $\varsigma' = \varsigma|_{E' \,||\, e'}$, $e'' = \varsigma'(e')$, and $E'' = \textsc{BFresh}(\varsigma', E')$. As above, we proceed by showing that the premises of the Application closure rule apply and by showing that the resulting expression and constraint set preserve the simulation.

By Lemma 4.3, there exists some M_1 such that $E_0 \,||\, e_0\ {}_{E_0}\!\preccurlyeq_{M_1} C_0$ and the domain of M_1 is exactly the set of variables appearing in $E_0 \,||\, e_0$. We will proceed using M_1 and make use of this domain property shortly.

By Definition 4.2 and because $x_2 = x_0\ x_1 \in e_0$, we have that $\alpha_0\ \alpha_1 <: \alpha_2 \in C_0$ such that $x_i\ {}_{E_0}\!\preccurlyeq_{M_1} \alpha_i$ for $i \in \{0, 1, 2\}$. By Lemma 4.18, we have that $\alpha_0\ \alpha_1\ {}_{C_0}\!\overset{\emptyset}{\rightsquigarrow}{}^{\Pi}_{\bullet}\ \alpha'_1 \backslash C'_1\ @\ C''_1$ such that $e'\ {}_{E_0}\!\approx_{M_1} \alpha'_1 \backslash C'_1$ and $E'\ {}_{E_0}\!\precapprox_{M_1} C''_1$.

Let $\sigma = \Phi(\alpha_2, -)$, $\sigma' = \sigma|_{C'_1 \cup C''_1}$, $\alpha'_2 \backslash C'_2 = \sigma'(\alpha'_1 \backslash C'_1)$, and $C''_2 = \textsc{TFresh}(\sigma', C''_1)$. Given these definitions, the Application rule of constraint closure (Figure 3.16) gives us that $C_0 \Longrightarrow C_0 \cup C'_2 \cup C''_2 \cup \{\alpha'_2 <: \alpha_2\}$. We define C_3 to be the latter and so it suffices to show that $E_3 \,||\, e_3\ {}_{E_3}\!\preccurlyeq_{M_3} C_3$ for some M_3 and E_3.

By Lemma 4.24 and because we know that $x_2 = x_0\ x_1 \in E_0 \,||\, e_0$, we know that any variable appearing in either E_0 or e_0 is not in the codomain of ς; thus the domain of M_1 and the codomain of ς are disjoint. This together with the property above that the domain of M_1 is exactly the set of variables appearing in $E_0 \,||\, e_0$ satisfies the premises of Lemma 4.20.

By Lemma 4.20, we have $e''\ {}_{E_0}\!\preccurlyeq_{M_2} \alpha'_2 \backslash C'_2$ and $E''\ {}_{E_0}\!\preccurlyeq_{M_3} C''_2$ for some $M_3 \supseteq M_1$. By Lemma 4.4, we can extend our previous simulations from M_1 to M_3. Thus, by Lemma 4.9, we add the original environment and constraints to obtain $E_0 \,||\, E'' \,||\, e'' \,||\, [x_2 = \mathrm{RV}(e'')] \,||\, e_1\ {}_{E_0}\!\preccurlyeq_{M_3} C_0 \cup C''_2 \cup C'_2 \cup \{\alpha'_2 <: \alpha_2\}$. We let $E_3 = E_0 \,||\, E''$; then the previous can be restated as $E_3 \,||\, e_3\ {}_{E_0}\!\preccurlyeq_{M_3} C_3$. By this and Lemma 4.8, we have $E_3 \,||\, e_3\ {}_{E_3}\!\preccurlyeq_{M_3} C_3$ and we are finished. □

4.8 Stuck Simulation

The initial alignment lemma (Section 4.3) allows us to establish a simulation between an expression and its initial constraint set. The preservation lemma (Section 4.7.2) allows us to retain this simulation throughout evaluation. To prove soundness, we require one final lemma: that stuck programs are always simulated by inconsistent constraint sets. We present supporting lemmas for this property and show that the property holds.

4.8.1 Totality of Compatibility

To show that stuck programs are simulated by inconsistent constraint sets, we need the property that every application matching step during evaluation either explicitly succeeds or fails; this allows us to argue about the nature of stuck programs, specifically when the Application rule does not apply. To demonstrate that application matching is total in this sense, however, we also require that value compatibility is similarly total. We state this property as follows:

Lemma 4.27. Suppose $E = \overset{n}{\overrightarrow{s}}$ is well-formed and x_0 appears in E. For any pattern p, either $x\backslash E \sim \tfrac{\emptyset/\emptyset}{\{p\}} \updownarrow []$ or, for some E', $x\backslash E \sim \tfrac{\{p\}/\emptyset}{\emptyset} \updownarrow E'$.

Proof. By induction first on the index i of the clause s_i which defines x and then on the depth (PD) of p. Because x appears in E and E is well-formed (and thus closed), there must exist some clause $s_i = x_0 = v$. Because we know this clause exists, it is sufficient by the value selection rule to show that either $v\backslash E \sim \tfrac{\emptyset/\emptyset}{\{p\}} \updownarrow []$ or that, for some $P^{\uparrow} \ni p$ and some E', $v\backslash E \sim \tfrac{\{p\}/\emptyset}{\emptyset} \updownarrow E'$.

If $\updownarrow = \downarrow$, then it suffices by the Binding rule to show either that $v\backslash E \sim \tfrac{\{p\}/\emptyset}{\emptyset} \uparrow E'$ for some E' or that $v\backslash E \sim \tfrac{\emptyset/\emptyset}{\{p\}} \uparrow []$. Otherwise, $\updownarrow = \uparrow$ and the same conditions suffice.

Conjunction Patterns. We have three cases: p is a conjunction pattern, an empty pattern, or a constructor pattern. If p is a conjunction pattern, then let $p = x'_1 * x'_2\backslash F$, let $p'_1 = x'_1\backslash F$, and let $p'_2 = x'_2\backslash F$. By the Conjunction Pattern rule, it suffices to show that one of three conditions is always true: (1) $v\backslash E \sim \tfrac{\{p'_1,p'_2\}/\emptyset}{\emptyset} \uparrow E'$ for some E', (2) $v\backslash E \sim \tfrac{\emptyset/\emptyset}{\{p'_1\}} \uparrow []$, or (3) $v\backslash E \sim \tfrac{\emptyset/\emptyset}{\{p'_2\}} \uparrow []$. By Definition 4.10, $\text{PD}(p'_1) < \text{PD}(p)$ and $\text{PD}(p'_2) < \text{PD}(p)$, so the inductive hypothesis gives us either that $v\backslash E \sim \tfrac{\{p'_1\}/\emptyset}{\emptyset} \uparrow E'$ for some E' or that $v\backslash E \sim \tfrac{\emptyset/\emptyset}{\{p'_1\}} \uparrow []$; we have likewise for p'_2. If the latter case is true for p'_1, then (regardless of p'_2) condition 2 above holds. If the latter case is true for p'_2, then (regardless of p'_1) condition 3 above holds. If both p'_1 and p'_2 are compatible, then Lemma 4.13 together with the proofs for p'_1 and p'_2 satisfy condition 1 above. Thus, if p is a conjunction pattern, this lemma holds.

Empty Pattern. If p is an empty pattern $()\backslash\Psi$, then it suffices by the Empty Onion Pattern rule to show that $v\backslash E \sim \tfrac{\emptyset/\emptyset}{\emptyset} \downarrow E'$. This process is syntax directed by v. For instance, if v is a label $l\ x'$, then the Label rule shows that it is suffices to show $x'\backslash E \sim \tfrac{\emptyset/\emptyset}{\emptyset} \downarrow E'$, which is true by the Leaf rule. The remaining premises of the Label rule are trivial because all three pattern sets are empty. In general, each form of v has a corresponding rule which reduces to trivial conditions on empty sets and, in the cases of variables, uses of the Leaf rule.

The above demonstrates that this lemma holds when CtrPat(p) is false. We address the cases in which p is a constructor pattern by case analysis on v. Again, it suffices to show either that $v\backslash E \sim \frac{\{p\}/\emptyset}{\emptyset} \Phi E'$ for some E' or that $v\backslash E \sim \frac{\emptyset/\emptyset}{\{p\}} \Phi []$.

Empty Onion. Suppose $v = ()$; then the Empty Onion rule applies. We let $P^- = \{p\}$ and this case is finished.

Onion. Suppose $v = x_1 \,\&\, x_2$. Because E is closed and $x_1 \,\&\, x_2$ appears in E, we have that some clause $s_{j_1} = x_1 = v_1' \in E$; likewise, some clause $s_{j_2} = x_2 = v_2' \in E$. Again because E is closed, we have that $j_1 < i$ and $j_2 < i$. By induction, we have that, for each $k \in \{1, 2\}$, either $x_{j_k}\backslash E \sim \frac{\{p\}/\emptyset}{\emptyset} \Phi E'$ for some E' or that $x_{j_k}\backslash E \sim \frac{\emptyset/\emptyset}{\{p\}} \Phi []$. If either $k = 1$ or $k = 2$ is the former case, then the Onion rule gives us that $v\backslash E \sim \frac{\{p\}/\emptyset}{\emptyset} \Phi E'$ for some E'; if both $k = 1$ and $k = 2$ are the latter case, then the Onion rule gives us that $v\backslash E \sim \frac{\emptyset/\emptyset}{\{p\}} \Phi []$. Either way, this case is complete.

Label. Suppose $v = l\ x'$. Either p is of the form $l\ p'$ or it is not. If not, then observe that by the Leaf rule we have $x'\backslash E \sim \frac{\emptyset/\emptyset}{\emptyset} \phi []$. By the Label rule, this demonstrates that $v\backslash E \sim \frac{\emptyset/\emptyset}{\{p\}} \Phi []$ and that subcase is finished.

Otherwise, $p = l\ p'$. Because E is closed and $l\ x'$ appears in E, we have by the same argument as in the Onion rule that the inductive hypothesis applies to x'. Thus, either $x'\backslash E \sim \frac{\{p'\}/\emptyset}{\emptyset} \Phi E'$ for some E' or $x'\backslash E \sim \frac{\emptyset/\emptyset}{\{p'\}} \Phi []$. In the prior case, the Label rule demonstrates that $v\backslash E \sim \frac{\{p\}/\emptyset}{\emptyset} \Phi E'$; in the latter case, the Label rule demonstrates that $v\backslash E \sim \frac{\emptyset/\emptyset}{\{p\}} \Phi []$. Thus, the case in which $v = lx'$ is complete and we are finished. $\square$

4.8.2 Totality of Matching

Using the above, we can demonstrate that matching can be phrased as a function: for a given environment and application site, matching as it is used in the operational semantics either succeeds or fails. To demonstrate the proof by induction, we explicitly identify the semantics of the pattern positions in the lemma statement:

Lemma 4.28. Suppose $E = \overset{n}{\overrightarrow{s}}$ is well-formed and x_0 and x_1 appear in E. For some P_1, suppose that $\forall p \in P_1 .\, x_1\backslash E \sim \frac{\emptyset/\emptyset}{\{p\}} \phi []$. Then $x_0\ x_1 \;{}^{P_1}_{E}\!\!\rightsquigarrow^{P_2}_{\odot}\; e \mathbin{@} E'$ for some e, E', P_2, and $\odot$. Further, when $\odot = \bigcirc$, then we have for each p in P_2 that $x_1\backslash E \sim \frac{\emptyset/\emptyset}{\{p\}} \phi []$.

Proof. By induction on the index i of the clause s_i which defines x_0. Because x_0 appears in E and E is well-formed (and thus closed), there must exist some clause $s_i = x_0 = v$. We proceed by case analysis on v.

Function. Suppose $v = p \mathbin{\texttt{->}} e$. By Lemma 4.28, either $x_1\backslash E \sim \frac{\{p\}/\emptyset}{\emptyset} \phi E'$ for some E' or $x_1\backslash E \sim \frac{\emptyset/\emptyset}{\{p\}} \phi []$. In the prior case, we use Lemma 4.13 to show that $x_1\backslash E \sim \frac{\{p\}/\emptyset}{P_1} \phi E'$ for some E' and the Function Match rule applies.

In the latter case, we use Lemma 4.13 to show that $x_1\backslash E \sim \frac{\emptyset/\emptyset}{P_1 \cup \{p\}} \phi []$ and the Function Mismatch rule applies. Because $\odot = \bigcirc$, we are under the additional obligation to show that all patterns in P_2 are incompatible with x_1; namely, we must show $x_1 \backslash E \sim \frac{\emptyset/\emptyset}{\{p\}} \phi []$, which we have from the original case analysis which led us to the latter case.

Onion. Suppose $v = x_2 \,\&\, x_3$. Because E is closed and $x_2 \,\&\, x_3$ appears in E, there must be clauses $s_j = x_2 = v'_2$ and $s_k = x_3 = v'_3$ both appearing in E. Further, because E is closed, we have $j < i$ and $k < i$. By induction, we have $x_2\ x_1\ {}^{P_1}_{E}{\rightsquigarrow}^{P_2}_{\odot_2}\ e$ @ E' for some $\odot_2$. Also, if $\odot_2 = \bigcirc$, then we have for each p in P_2 that $x_1 \backslash E \sim \frac{\emptyset/\emptyset}{\{p\}} \phi []$. If $\odot_2 = \bullet$, then the Onion Left rule applies and we are finished.

Otherwise, $\odot_2 = \bigcirc$. By induction, we have $x_3\ x_1\ {}^{P_1}_{E}{\rightsquigarrow}^{P_2}_{\odot_3}\ e$ @ E'; if $\odot_3 = \bigcirc$ then we also have for each p in P_3 that $x_1 \backslash E \sim \frac{\emptyset/\emptyset}{\{p\}} \phi []$. The Onion Right rule applies and, if $\odot_3 = \bullet$, this case is finished. If $\odot_3 = \bigcirc$, then the additional obligation that the patterns in $P_2 \cup P_3$ are shown incompatible is satisfied by the previous inductive conclusions and the distributivity properties of set union with regards to universal quantifiers. Thus, this case is finished.

Non-Function, Non-Onion. Suppose v is neither an onion nor a function; then the Non-Function rule applies. Because $P_2 = \emptyset$ in this case, we are finished. $\square$

We then formally state and prove that stuck programs are simulated by stuck constraint sets.

Lemma 4.29. For any e, if $e\ {}_{E_0}{\preccurlyeq}_M\ C$ and e is stuck, then C is inconsistent.

Proof. By definition, e of the form E is not stuck; therefore, $e = E \,||\, [s] \,||\, e_1$ such that s is not of the form $x = v$. We also observe that s cannot be of the form $x = x'$ because e is closed; thus, there must be some clause appearing in E which binds x' and so the Value Lookup rule applies. By elimination, s must be of the form $x_2 = x_0\ x_1$.

The premises of the Application rule of the operational semantics are total α-renamings, function restrictions, or the condition $x_0\ x_1\ {}^{\emptyset}_{E}{\rightsquigarrow}^{P}_{\bullet}\ e'$ @ E'; thus, the latter must be false for e to be stuck. The value application matching relation is total for closed e (Lemma 4.27), so we must have that $x_0\ x_1\ {}^{\emptyset}_{E}{\rightsquigarrow}^{P}_{\bigcirc}\ e'$ @ E'. By Lemma 4.18 gives us that $\alpha_0\ \alpha_1\ {}^{\emptyset}_{C}{\rightsquigarrow}^{\Pi}_{\bigcirc}\ \alpha'_1 \backslash C'_1$ @ C''_1. By Definition 3.35, C is inconsistent. $\square$

4.9 Proof of Soundness

The above lemmas are sufficient to prove the soundness of the TinyBang type system. We restate that theorem here for convenience and then give its proof.

Theorem 1 (Soundness)**.** For any closed e, if $e \longrightarrow^* e'$ and e' is stuck then e does not typecheck.

Proof. Let $[\![e]\!]_{\mathrm{E}} = \alpha\backslash C$. Because e is closed, Lemma 4.11 gives us that $e\ {}_{[]}{\preccurlyeq_M}\ \alpha\backslash C$. Let E be maximal for e; Lemma 4.8 gives us $e\ {}_{E}{\preccurlyeq_M}\ \alpha\backslash C$.

We have that $e \longrightarrow^{*} e'$ where e' is stuck. By induction on the number of steps in the proof of $e \longrightarrow^{*} e'$, Lemma 4.26 gives us that that there exists a C' such that $C \Longrightarrow^{*} C'$ and $e'\ {}_{E'}{\preccurlyeq_{M'}}\ C'$. Lemma 4.29 gives us that C' is inconsistent. Therefore C closes to an inconsistent constraint set and, by Definition 3.36, e does not typecheck. $\square$

Chapter 5

Proof of Decidability

This chapter proves that the type system presented in Section 3.5 is decidable. Throughout this chapter, we assume the existence of some well-formed expression e_{INIT} which is to be typechecked. As the chapter proceeds, we define a series of values based upon e_{INIT} and then demonstrate that various relations are decidable with respect to them. We then show that we can use these decidable relations to determine whether e_{INIT} typechecks. Although e_{INIT} is fixed, it is also arbitrary; thus, this is sufficient to demonstrate an effective method for deciding whether any well-formed expression typechecks.

Recall from Section 3.5.5.1 that the type system is parametric in a polymorphism model. Whether the TinyBang type system is decidable depends on certain properties of the polymorphism model, specifically with regards to bounding the number of type variables it introduces to constraint closure. (For example, the perfectly precise polymorphism model, in which $\Phi(-,-)$ behaves exactly like $\boldsymbol{\alpha}(-,-)$, creates a type system which is undecidable on any program containing recursion.) We begin this chapter by introducing some useful notation. We then define properties of polymorphism models which allow us to constrain our proof to those models which yield decidable type systems. Next, we state our decidability theorem in Section 5.2. The remainder of the chapter is dedicated to the proof of that theorem, which proceeds much as outlined above. Because the proof of decidability proceeds largely by selecting sets which bound the values which can appear during closure, it is helpful to introduce the additional type grammar terms in Figure 5.1.

$$\begin{array}{rcll} A & ::= & \overset{\sqcup}{\alpha} & \textit{type variable sets} \\ T & ::= & \overset{\sqcup}{\tau} & \textit{type sets} \end{array}$$

Figure 5.1: Decidability Type Grammar Extensions

5.1 Polymorphism

The decidability proof relies upon demonstrating several type system relations decidable, most notably the constraint closure relation. It accomplishes this by demonstrating a finite upper bound on the set of constraints which can be closed from the initial alignment of e_{INIT}. For this to be possible, we must have a finite bound on the type variables produced by the polymorphism model for a given program. If a polymorphism model has this property, we call it a *finitely freshening* model. We formalize this property as follows:

Definition 5.1 (Finitely Freshening)**.** A polymorphism model Φ/Υ is *finitely freshening* iff there exists a *freshness bounding function* ϑ from expressions e onto type variable sets A such that, for any e, all of the following are true:

- $\vartheta(e)$ is a finite superset of the type variables appearing in $[\![e]\!]_{\text{E}}$.
- For every α and α' in $\vartheta(e)$, we have that $\Phi(\alpha, \alpha') \in \vartheta(e)$.
- For every $\alpha \in \vartheta(e)$, every $A \subseteq \vartheta(e)$, and every $\sigma \in \Upsilon(A)$, we have that $\sigma(\alpha) \in \vartheta(e)$.

For the monomorphic model, for instance, $\vartheta(e)$ is exactly the set of type variables appearing in $[\![e]\!]_{\text{E}}$. The remaining conditions of Definition 5.1 are immediate by Definition 3.32. It is still sufficient for a reader to consider the monomorphic model in this chapter; Chapter 6 introduces the polymorphism model we use in practice; we show it to be finitely freshening in Section 6.3.

Throughout the remainder of this chapter, we assume that some finitely freshening polymorphism model Φ/Υ is being used. We also assume that both Φ and Υ are computable functions.

5.2 Decidability Theorem

Given the definitions from the previous sections, we now state our decidability theorem as follows:

Theorem 2 (Decidability)**.** With a finitely freshening polymorphism model, typechecking is decidable.

The remainder of this chapter works toward proving this theorem. Recall from the introduction of this chapter that we assume that we are considering (without loss of generality) some well-formed expression e_{INIT}. We write all lemma statements in context of this assumption.

Much of the strategy involves demonstrating the finiteness of certain sets, such as the type variables in a constraint set under closure or the patterns used in the compatibility relation, and using that finiteness to show the relations used during typechecking to be decidable. While this is sufficient for decidability, we do not show complexity bounds on the type checking process. We do discuss the properties of typechecking which make it computationally feasible in practice in Chapter 9.

5.3 Conventions

As described above, the proof of decidability proceeds by finitely bounding sets of terms which appear at various points throughout the typechecking process. The notation consistently used throughout this chapter is as follows:

- When finitely bounding a set for the purposes of the entire constraint closure, we annotate the set with the subscript $_{\text{MAX}}$. For instance, the set of all type variables appearing during closure will be written A_{MAX}.
- When finitely bounding the set of values which may appear in a given place of a relation during a proof search, we annotate the set with a wide hat. For instance, the set of all type variables which may appear in the search for a compatibility proof will be written $\widehat{A}$.
- When discussing a proof search for a particular instance of a relation, the arbitrarily-chosen places of that relation are denoted with a wide tilde. For instance, the set of constraints chosen when discussing an arbitrary instance of the compatibility relation is written $\widetilde{C}$.

The above provides a summary of the notational form we use. These definitions appear at different points throughout the chapter: A_{MAX}, for instance, is defined in the next section whereas Π_{MAX} is defined in Section 5.9. The definitions are ordered in this fashion due to their dependency on the machinery presented in the intervening sections.

5.4 Finite Variables

We begin by demonstrating that the typechecking of e_{INIT} involves only finitely many type variables. We show this by demonstrating that each constraint closure step only introduces type variables from the finitely freshening polymorphism model. More formally,

Definition 5.2 (Type Variable Bounding Set)**.** Let $A_{\text{MAX}} = \vartheta(e_{\text{INIT}})$ where ϑ is the freshness bounding function of the finitely freshening polymorphism model.

We first demonstrate that the type compatibility relation does not generate type variables outside of the bounds of that set.

Lemma 5.3. Suppose either $\alpha\backslash C \sim \frac{\Pi^{\Diamond}/\Pi^{+}}{\Pi^{-}} \mathrel{\Diamond} C'$ or $\tau\backslash C \sim \frac{\Pi^{\Diamond}/\Pi^{+}}{\Pi^{-}} \mathrel{\Diamond} C'$ such that all variables appearing within C, $\Pi^{\Diamond}$, Π^{+}, Π^{-}, and either τ or α itself also appear within A_{MAX}. Then all variables appearing within C' also appear within A_{MAX}.

Proof. By induction on the size of the proof of compatibility. By inspection of the rules in Figure 3.14, it is immediate that the only rule which adds type variables to C' is the Binding rule; all other rules either specify an empty set in that location or provide a union of sets obtained from the C' position of compatibility premises and therefore are trivially addressed by their inductive premises. So it suffices to show that the C' position of the Binding rule includes only variables which appear within A_{MAX}. By inspection, this includes τ, $\Pi^{\Diamond}$, Π^{+}, Π^{-}, and a type variable from a pattern within $\Pi^{\Diamond}$. Since we already have by assumption that the variables appearing within these are in A_{MAX}, we are finished. □

Demonstrating a similar conclusion for the type application matching relation is fairly straightforward:

Lemma 5.4. Suppose that $\alpha_0\ \alpha_1 \overset{\Pi_1}{\underset{C}{}}\!\rightsquigarrow\!\overset{\Pi_2}{\underset{\odot}{}}\ \alpha'\backslash C'\ @\ C''$ such that all variables appearing within $\{\alpha_0, \alpha_1\}$, C, and Π_1 also appear in the set A_{MAX}. Then all variables appearing within Π_2 also appear in A_{MAX}. Further, if $\odot = \bullet$, then all variables appearing within $\{\alpha'\}$, C', and C'' also appear in A_{MAX}.

Proof. By induction on the size of the proof of application matching. We proceed by case analysis on the rule set.

Function Match. In the Function Match rule, π, α', and C' are taken from a function type appearing in C and therefore contain only variables appearing in A_{MAX}. Since $\Pi_2 = \{\pi\}$, it now suffices to show this property for C'. This is immediate from Lemma 5.3, so this case is finished.

Function Mismatch. In this case, $\odot = \bigcirc$, so it suffices to show this property only of Π_2. This is shown as above; Π_2 contains a single pattern taken from a function type appearing in C.

Non-Function. In this case, $\odot = \bigcirc$, so it suffices to show this property only of Π_2. This is trivial, since $\Pi_2 = \emptyset$.

Onion Left. For the Onion Left rule, this property is trivially satisfied by the inductive hypothesis.

Onion Right. By induction, we have from the first matching premise that Π_2 contains only variables appearing in A_{MAX}. By this and the second matching premise, we have this property for α'_3, C'_3, C''_3, and Π_3. From this, the conclusion is immediately satisfied. □

We finally demonstrate this finiteness of variables on the constraint closure itself.

Lemma 5.5. Suppose that $C \Longrightarrow C'$. If C contains only type variables appearing in A_{MAX}, then C' contains only type variables appearing in A_{MAX}.

Proof. By case analysis on the set of rules in Figure 3.16.

Transitivity. For the Transitivity rule, this is trivial; the only constraints which appear in C' but not C are those which are created from types, variables, and patterns taken directly from constraints in C.

Application. In the Application rule, α_0 and α_1 are taken from a constraint appearing in C. This (together with the empty set of patterns in the Π_1 position of the application matching premise) gives us by Lemma 5.4 that α'_1, C'_1, and C''_1 contain only variables appearing in A_{MAX}. By Definition 5.1, we have that σ will produce from those variables only other variables appearing in A_{MAX}; thus, α'_2, C'_2, and C''_2 contain only variables appearing in A_{MAX}. Because C' is comprised of constraints containing only those variables and the contents of C, this case is finished.

Merging. This case is trivial by Definition 5.1. □

5.5 Finite Types

In this section, we demonstrate that only finitely many types may appear during closure. This follows much the same process as the previous lemmas. We begin by defining a maximal, finite set of types and show that each relation respects it.

Definition 5.6 (Type Bounding Set)**.** Suppose T to be the set of all types appearing within $[\![e_{\text{INIT}}]\!]_{\text{E}}$ (including types nested within functions). Let $T_{\text{MAX}} = \bigcup\{f(\tau) \mid \tau \in T\}$ where f is a function accepting a type τ and producing a set of all types which can be derived by replacing the variables appearing in τ with variables appearing in A_{MAX}.

Of course, T_{MAX} is an extreme overapproximation of the types which will actually appear in closure; even in the most pathological program, it is impossible for every variable to be inserted into every type position. But T_{MAX} is finite (since $[\![e_{\text{INIT}}]\!]_{\text{E}}$ is finite, A_{MAX} is finite, and there are finitely many type variables in each type) and serves as an upper bound on the types appearing during constraint closure. We now show the appropriate properties for each relation, starting with compatibility.

Lemma 5.7. Suppose either $\alpha\backslash C \sim \frac{\Pi^{\Phi}/\Pi^{+}}{\Pi^{-}} \between C'$ or $\tau\backslash C \sim \frac{\Pi^{\Phi}/\Pi^{+}}{\Pi^{-}} \between C'$ such that all types appearing in C also appear in T_{MAX} (and, if appropriate, $\tau \in T_{\text{MAX}}$). Then all types appearing within C' also appear within T_{MAX}.

Proof. By the same strategy as Lemma 5.3. We begin by induction on the size of the proof. If it is a type variable compatibility proof, there are two cases: Leaf (which is trivially satisfied) or Type Selection. In the latter case, the type provided is taken from C, so the inductive premises are satisfied; the resulting C' is taken directly from the inductive conclusions and so the type variable compatibility proof case is finished.

Otherwise, the proof is a type compatibility proof. By inspection of the rules, all cases other than Binding are trivially satisfied by the inductive hypothesis. In the Binding rule, the types added to the C' position of the compatibility relation are taken directly from C, so we are finished. □

We then proceed to matching:

Lemma 5.8. Suppose $\alpha_0\ \alpha_1 \ {}^{\Pi_1}_{C}\!\rightsquigarrow^{\Pi_2}_{\odot}\ \alpha'\backslash C' \ @\ C''$. If all types appearing within C also appear within T_{MAX} then all types appearing within C' and C'' also appear in T_{MAX}.

Proof. By induction on the size of the proof tree. The Function Match base case is addressed by Lemma 5.7; the Function Mismatch and Non-Function base cases are trivial, as $C' = C'' = \emptyset$. The Onion Left and Onion Right rules both take their C' and C'' directly from inductive conclusions, so we are finished. □

Finally, we show this property on constraint closure as well.

Lemma 5.9. Suppose $C \Longrightarrow C'$ where all types appearing in C also appear in T_{MAX}. Then all types appearing in C' also appear in T_{MAX}.

Proof. By case analysis on the rule used.

Transitivity. This case is trivial exactly as in Lemma 5.5.

Application. In the Application rule, Lemma 5.8 gives us that C'_1 and C''_1 contain only types appearing in T_{MAX}. By Definition 5.1, the replacement function σ can only yield variables which appear in A_{MAX}; thus, by Definition 5.6, C'_2 and C''_2 contain only types appearing in T_{MAX}. These two sets contain the only new types added to C to form C', so this case is finished.

Merging. By Definition 5.1, the replacement function σ can only yield variables which appear in A_{MAX}; thus, by Definition 5.6, $\sigma(C)$ contains only types appearing in T_{MAX} and we are finished. □

5.6 Decidable Compatibility

Next, we demonstrate that the type compatibility relation is decidable. We accomplish this by demonstrating that there are only finitely many ways to build a proof tree of a particular judgment from the rules in Figure 3.14. For the purposes of this section, we select arbitrarily a compatibility judgment of discourse of either the form $\widetilde{\alpha}\backslash\widetilde{C} \sim \frac{\widetilde{\Pi^{\phi}}/\widetilde{\Pi^{+}}}{\widetilde{\Pi^{-}}} \widetilde{\Phi}\, \widetilde{C}'$ or the form $\widetilde{\tau}\backslash\widetilde{C} \sim \frac{\widetilde{\Pi^{\phi}}/\widetilde{\Pi^{+}}}{\widetilde{\Pi^{-}}} \widetilde{\Phi}\, \widetilde{C}'$ and show an algorithm for determining whether or not it holds (much in the fashion that we have done for e_{INIT}).

5.6.1 Finitely-Bounded Judgments

We begin by demonstrating that we can finitely bound the judgments used in any proof tree of our judgment of discourse. In order to do so, we show a finite bound for the values which appear at each place in the relation. To give these finite bounds, we begin by providing some auxilliary definitions.

5.6.1.1 Pattern Type Depth

Definition 4.10 gives a notion of the *depth* of a well-formed pattern; in the following discussion, a corresponding definition is necessary for pattern types as well:

Definition 5.10 (Pattern Type Depth)**.** We define the *depth* of a well-formed pattern type $\alpha\backslash\Psi$ as follows:

$$\begin{aligned}
\text{PD}(()\,\backslash\Psi) &= 0 \\
\text{PD}(l\ \alpha\backslash\Psi) &= \text{PD}(\alpha\backslash\Psi) + 1 \\
\text{PD}(\alpha_1 * \alpha_2\backslash\Psi) &= \max(\text{PD}(\alpha_1\backslash\Psi), \text{PD}(\alpha_2\backslash\Psi)) + 1
\end{aligned}$$

5.6.1.2 The All-Points Function

In finitely bounding the judgments which appear within a proof tree of our judgment of discourse, we must bound the patterns which can appear in such judgments. The filter constraint sets of the patterns do not change throughout compatibility and the only new patterns introduced come from $\widetilde{C}$; since all of these sources are finite, any pattern constructed from them (that doesn't create new filter constraints) will be as well. We formalize this intuition by giving an *all-points* function on patterns which yields the set of patterns which occur at all points within it.

Definition 5.11 (All-Points Function)**.** We define the *all-points function* over patterns to be $\text{AllPoints}(\alpha\backslash\Psi) = \{\alpha'\backslash\Psi \mid \alpha' \text{ appears in } \psi\}$. For notational convenience, we write $\text{AllPoints}(\Pi)$ to mean $\bigcup\{\text{AllPoints}(\pi) \mid \pi \in \Pi\}$.

For instance, consider $\alpha_0\backslash\Psi$ where $\Psi = \{\texttt{`A}\ \alpha_1\ \texttt{<:}\ \alpha_0, \texttt{()}\ \texttt{<:}\ \alpha_1\}$. Then $\textsc{AllPoints}(\alpha_0\backslash\Psi) = \{\alpha_0\backslash\Psi, \alpha_1\backslash\Psi\}$: it returns each subpattern of $\alpha_0\backslash\Psi$, each of which starts at a different point in the filter constraint set. This leads us to a simple lemma:

Lemma 5.12. For all acyclic π, $\pi \in \textsc{AllPoints}(\pi)$. Likewise, for all Π containing only acyclic pattern types, $\Pi \subseteq \textsc{AllPoints}(\Pi)$.

Proof. The first statement is immediate from the definition of pattern depth (Definition 5.10), since the root of π necessarily appears within its filter constraint set. The second statement is immediate from the first statement as well as the notation provided in Definition 5.11. ☐

In our proof below, it is also helpful to show some other properties of the all-points function. We first observe that, if a pattern is one of the points of another pattern, then it appears in the points of any set containing that latter pattern. This is a fairly simple property, but stating it here helps to clarify the discussion below.

Lemma 5.13. If $\pi_1 \in \textsc{AllPoints}(\pi_2)$ and $\pi_2 \in \Pi$ then $\pi_1 \in \textsc{AllPoints}(\Pi)$.

Proof. Immediate from Definition 5.11: since π_2 is in Π, $\textsc{AllPoints}(\Pi)$ will include $\textsc{AllPoints}(\pi_2)$ as one of its unioned sets. ☐

We next observe that the all-points function is monotonic on sets, preserving the subset relation.

Lemma 5.14. If $\Pi_1 \subseteq \Pi_2$ then $\textsc{AllPoints}(\Pi_1) \subseteq \textsc{AllPoints}(\Pi_2)$.

Proof. Immediate from Definition 5.11. ☐

We also observe that the all-points function is idempotent on sets.

Lemma 5.15. $\textsc{AllPoints}(\Pi) = \textsc{AllPoints}(\textsc{AllPoints}(\Pi))$

Proof. Immediate by Definition 5.11. By inspection of that definition, $\textsc{AllPoints}(\Pi)$ contains the same filter constraint sets Ψ as Π does; thus, the same type variables appear in those filter constraint sets. Since these are the only properties used in the set comprehension within the definition, the result of the set comprehension will be the same. ☐

5.6.1.3 Bounding the Judgment Places

Using the all-points function, we can define the bounding sets on each place in the relations which could appear in the subproofs of the judgment of discourse.

Definition 5.16 (Compatibility Bounding Sets)**.** We provide the following bounding sets:

- Let $\widehat{T}$ be all types appearing in $\widetilde{C}$ as well as $\widetilde{\tau}$.
- Let $\widehat{A}$ be all type variables appearing in $\widetilde{C}$ unioned with all type variables appearing in $\widetilde{\Pi^{\diamond}} \cup \widetilde{\Pi^{+}} \cup \widetilde{\Pi^{-}}$ as well as $\widetilde{\alpha}$ itself.
- Let $\widehat{\Pi} = \textsc{AllPoints}(\Pi)$ where Π is $\widetilde{\Pi^{\diamond}} \cup \widetilde{\Pi^{+}} \cup \widetilde{\Pi^{-}}$ as well as all patterns appearing in $\widetilde{C}'$.
- Let $\widehat{C}' = \{\tau|^{\Pi^+}_{\Pi^-} <: \alpha \mid \tau \in \widehat{T} \wedge \alpha \in \widehat{A} \wedge \Pi^+ \subseteq \widehat{\Pi} \wedge \Pi^- \subseteq \widehat{\Pi}\}$.

We observe that each of the above sets is finite. We know $\widehat{T}$ and $\widehat{A}$ to be finite because $\widetilde{C}$ and $\widetilde{C}'$ are finite. We know $\widehat{\Pi}$ to be finite because it is a finite set comprehension over the union of finite sets. We know $\widehat{C}'$ to be finite because it is the finite set comprehension of finite sets.

It would be possible to define and use the maximal sets which bound constraint closure (e.g. A_{MAX}) rather than use the bounding sets provided above. The above bounding sets provide a somewhat more precise bound and more clearly demonstrate the relationship to the judgment of discourse. We later demonstrate that these bounding sets are subsets of those used to bound overall constraint closure.

5.6.1.4 Proving Finitely Many Possible Judgments

Using the above bounding sets, we can now demonstrate that any proof of type compatibility over the judgment of discourse uses only premises with places taken from those bounding sets; that is, we can finitely enumerate the compatibility judgments which may appear in any proof of the judgment of discourse. Formally, we state this property as the following lemma. This lemma is left quite general (and then specialized below) as the general form is quite useful later.

Lemma 5.17. Suppose $\alpha \in \widehat{A}$, $\tau \in \widehat{T}$, and $\Pi^{\diamond} \cup \Pi^{+} \cup \Pi^{-} \subseteq \widehat{\Pi}$. In any proof of $\alpha\backslash C \sim \frac{\Pi^{\diamond}/\Pi^{+}}{\Pi^{-}} \between C'$ or $\tau\backslash C \sim \frac{\Pi^{\diamond}/\Pi^{+}}{\Pi^{-}} \between C'$, all subproofs are either (1) immediately decidable; (2) proofs of the form $\alpha'\backslash C \sim \frac{\Pi^{\diamond\prime}/\Pi^{+\prime}}{\Pi^{-\prime}} \between' C''$ such that $\alpha' \in \widehat{A}$, $\Pi^{\diamond\prime} \cup \Pi^{+\prime} \cup \Pi^{-\prime} \subseteq \widehat{\Pi}$, and $C'' \subseteq \widehat{C}'$; or (3) proofs of the form $\tau'\backslash C \sim \frac{\Pi^{\diamond\prime}/\Pi^{+\prime}}{\Pi^{-\prime}} \between' C''$ such that $\tau' \in \widehat{T}$, $\Pi^{\diamond\prime} \cup \Pi^{+\prime} \cup \Pi^{-\prime} \subseteq \widehat{\Pi}$, and $C'' \subseteq \widehat{C}'$. Further, $C' \subseteq \widehat{C}'$.

Proof. By induction on the size of any such proof. We proceed by case analysis on the proof rule used.

Leaf. If the Leaf rule is used, then the rule has no premises and this case is trivial.

Type Selection. If the Type Selection rule is used, then the proven judgment is of the form $\alpha\backslash C \sim \frac{\Pi^{\diamond}/\Pi^{+}}{\Pi^{-}} \between C'$ and we have two premises: $\tau|^{\Pi_1^+}_{\Pi_1^-} <: \alpha \in C$ and $\tau\backslash C \sim$

$\frac{\Pi^{\phi}/\Pi^{+}\cup\Pi_1^{+}}{\Pi^{-}\cup\Pi_1^{-}} \Updownarrow C'$. Because C is finite, the first premise is trivially decidable. It then suffices in this case to show that $\tau \in \widehat{T}$, $\Pi^{\phi} \cup \Pi^{+} \cup \Pi_1^{+} \cup \Pi^{-} \cup \Pi_1^{-} \subseteq \widehat{\Pi}$, and $C' \subseteq \widehat{C}'$.

We have $\tau \in \widehat{T}$ because τ appears in C and $\widehat{T}$ includes all types appearing in C by Definition 5.16. We have by the inductive hypothesis that $C' \subseteq \widehat{C}'$, so that obligation is trivial. It then suffices in this case to show that $\Pi^{\phi} \cup \Pi^{+} \cup \Pi_1^{+} \cup \Pi^{-} \cup \Pi_1^{-} \subseteq \widehat{\Pi}$. We have by premise that $\Pi^{\phi} \cup \Pi^{+} \cup \Pi^{-} \subseteq \widehat{\Pi}$, so it suffices to show that $\Pi_1^{+} \cup \Pi_1^{-} \subseteq \widehat{\Pi}$. We have this because Π_1^{+} and Π_1^{-} were taken from a type appearing in C and, by Definition 5.16, all points of any patterns appearing in C are in $\widehat{\Pi}$. Thus, this case is finished.

Otherwise, the proven judgment is of the form $\tau\backslash C \sim \frac{\Pi^{\phi}/\Pi^{+}}{\Pi^{-}} \Updownarrow C'$. We continue by case analysis on the proof rule used.

Trivial Non-Constructor Patterns. If the Binding, Empty Onion Pattern, or Conjunction Pattern rules are used, the place-bounding property is trivial. In each case, each premise is either immediately decidable (e.g. a finite set comprehension over finite sets) or is a subjudgment where each place in the relation is either equal to or a subset of the corresponding place in the original judgment (as appropriate). Thus, if each place in the judgment is taken from our bounding sets, each place in the subjudgment is as well.

In all of these cases other than Binding, the obligations to show $C' \subseteq \widehat{C}'$ is trivial by the inductive hypothesis. In the case of Binding, it suffices to show that the additional bindings C'' appear within $\widehat{C}'$, which is immediate from Definition 5.16.

Conjunction Weakening. In the Conjunction Weakening rule, the proven judgment is $\tau\backslash C \sim \frac{\Pi_1^{\phi}/\Pi_1^{+}}{\Pi_1^{-}} \Updownarrow C'$. All premises are trivially decidable except for the subjudgment $\tau\backslash C \sim \frac{\Pi_1^{\phi}\cup\Pi_2^{\phi}/\Pi_1^{+}\cup\Pi_2^{+}}{\Pi_1^{-}\cup\Pi_2^{-}} \Updownarrow C'$. By the same reasoning as in the trivial cases, it suffices to show that $\Pi_2^{\phi} \cup \Pi_2^{+} \cup \Pi_2^{-}$ is from the bounding set $\widehat{\Pi}$.

Without loss of generality, let us consider Π_2^{+}: each pattern π_2 appearing within Π_2^{+} is of the form $\alpha\backslash\Psi$ where some $\pi_1 = \alpha_1 * \alpha_2\backslash\Psi$ appears in Π_1^{+} and either $\alpha = \alpha_1$ or $\alpha = \alpha_2$. By Definition 5.11, $\pi_2 \in \textsc{AllPoints}(\pi_1)$ (because α appears in π_1). Recall from premises that $\pi_1 \in \widetilde{\Pi}$; then by Lemma 5.13, $\pi_2 \in \textsc{AllPoints}(\widetilde{\Pi})$. By Definition 5.16, $\widetilde{\Pi}$ is defined by $\widetilde{\Pi} = \textsc{AllPoints}(\Pi)$ for some set Π; thus, $\pi_2 \in \textsc{AllPoints}(\textsc{AllPoints}(\Pi))$ and, by Lemma 5.15, $\pi_2 \in \textsc{AllPoints}(\Pi)$ so $\pi_2 \in \widetilde{\Pi}$. This demonstrates that each pattern in Π_2^{+} is in the bounding set for the judgment of discourse; as this argument proceeded without loss of generality, the same holds for Π_2^{ϕ} and Π_2^{-}. This case is therefore finished.

Trivial Types. If the Empty Onion rule or the Function rule is used, the only premise is trivial. If the Onion rule is used, both of the non-trivial premises are handled in the same fashion as in the Trivial Non-Constructor Patterns case above: each place is either equal to or a subset of (as appropriate) the place in the original judgment, so this lemma's conclusion holds by the premises of the lemma.

Label. If the Label rule is used, the argument proceeds as in the Conjunction Weakening case: each pattern appearing in the non-trivial subjudgment represents a point from a pattern in the original judgment and so the former pattern appears in the all-points set of the latter. Because the all-points function is idempotent and because the bounding set $\widehat{\Pi}$ is defined using the all-points function, the former pattern appears in this bounding set. Since this is true for each pattern appearing in the subjudgment and all other places in this relation are the same (except the binding state, which is not relevant), this case is complete. $\square$

We then specialize the above lemma to the judgment of discourse:

Lemma 5.18. In any proof of $\widetilde{\alpha}\backslash\widetilde{C} \sim \frac{\widetilde{\Pi^{\phi}}/\widetilde{\Pi^{+}}}{\widetilde{\Pi^{-}}} \; \widetilde{\Phi} \; \widetilde{C}'$ or $\widetilde{\tau}\backslash\widetilde{C} \sim \frac{\widetilde{\Pi^{\phi}}/\widetilde{\Pi^{+}}}{\widetilde{\Pi^{-}}} \; \widetilde{\Phi} \; \widetilde{C}'$, all subproofs are either (1) immediately decidable; (2) proofs of the form $\alpha\backslash\widetilde{C} \sim \frac{\Pi^{\phi}/\Pi^{+}}{\Pi^{-}} \; \Phi \; C'$ such that $\alpha \in \widehat{A}$, $\Pi^{\phi} \cup \Pi^{+} \cup \Pi^{-} \subseteq \widehat{\Pi}$, and $C' \subseteq \widehat{C}'$; or (3) proofs of the form $\tau\backslash\widetilde{C} \sim \frac{\Pi^{\phi}/\Pi^{+}}{\Pi^{-}} \; \Phi \; C'$ such that $\tau \in \widehat{T}$, $\Pi^{\phi} \cup \Pi^{+} \cup \Pi^{-} \subseteq \widehat{\Pi}$, and $C' \subseteq \widehat{C}'$

Proof. Immediate from Lemma 5.17. This lemma is applicable because $\widetilde{\alpha} \in \widehat{A}$ (by definition), $\widetilde{\tau} \in \widehat{T}$ (by definition), and $\widetilde{\Pi^{\phi}} \subseteq \widehat{\Pi}$ (and likewise for $\widetilde{\Pi^{+}}$ and $\widetilde{\Pi^{-}}$) by Lemma 5.12. $\square$

5.6.2 Finite Proof Search

The proof that the set of judgments appearing in the proof tree is bounded constitutes most of the work in demonstrating the property we need for compatibility. Next, we demonstrate that the search space for a proof is finite. We do so by showing that proofs of the judgment exists iff at least one of the proofs can be finitely bounded in height. We first define this bound:

Definition 5.19 (Compatibility Proof Height Bound)**.** For the judgment of discourse, let J be the Cartesian product $(\widehat{A} \cup \widehat{T}) \times \mathcal{P}(\widehat{\Pi}) \times \mathcal{P}(\widehat{\Pi}) \times \mathcal{P}(\widehat{\Pi}) \times \{\phi, \Phi\} \times \widehat{C}'$; because $\widetilde{C}$ is constant throughout all proofs of compatibility for the judgment of discourse, J is a finite set which is isomorphic to all judgments which can be constructed from the bounding sets in Definition 5.16. Let $\widetilde{n} = |J|$ be the size of this set.

Using the above definition, we now show that $\widetilde{n}$ does act as a bound on the size of one of the proof trees (if any proof trees exist).

Lemma 5.20. The judgment $\widetilde{\alpha}\backslash\widetilde{C} \sim \frac{\widetilde{\Pi^{\phi}}/\widetilde{\Pi^{+}}}{\widetilde{\Pi^{-}}} \; \widetilde{\Phi} \; \widetilde{C}'$ (resp. $\widetilde{\tau}\backslash\widetilde{C} \sim \frac{\widetilde{\Pi^{\phi}}/\widetilde{\Pi^{+}}}{\widetilde{\Pi^{-}}} \; \widetilde{\Phi} \; \widetilde{C}'$) holds if and only if there exists a proof of $\widetilde{\alpha}\backslash\widetilde{C} \sim \frac{\widetilde{\Pi^{\phi}}/\widetilde{\Pi^{+}}}{\widetilde{\Pi^{-}}} \; \widetilde{\Phi} \; \widetilde{C}'$ (resp. $\widetilde{\tau}\backslash\widetilde{C} \sim \frac{\widetilde{\Pi^{\phi}}/\widetilde{\Pi^{+}}}{\widetilde{\Pi^{-}}} \; \widetilde{\Phi} \; \widetilde{C}'$) whose tree is of a height no larger than $\widetilde{n}$.

Proof. The forward implication is trivial: if such a proof tree exists, then the judgment holds by Definition 3.26. We prove the reverse implication by induction on the number of nodes in the proof tree.

Suppose that a proof of the judgment exists of height $n > \widetilde{n}$. By definition of the height of a tree, there must exist some path in the tree of length n. All nodes in this proof tree must be elements of J so, because $\widetilde{n} < n$, this path must contain at least one judgment at two different nodes. An equivalent proof may therefore be constructed by replacing node closest to the root of the tree with any node in its strict subtree. (This is analogous to the statement that the proof $Q_1 \Leftarrow Q_2 \Leftarrow Q_3 \Leftarrow Q_2 \Leftarrow Q_4$ is equivalent to the proof $Q_1 \Leftarrow Q_2 \Leftarrow Q_4$ – we may take any proof of Q_2 and we elect to take the shorter one.)

In the resulting proof tree, we have two cases: either that tree's height is larger than $\widetilde{n}$ or it is not. In the case that it is not, we are finished. If the new tree's height is still larger than $\widetilde{n}$, we observe that it has strictly fewer nodes than the original proof tree; by these two facts, we may proceed by induction and so we are finished. $\square$

5.6.3 Proof

Using the above, we can state the decidability of compatibility:

Lemma 5.21. The mutual relations $\alpha\backslash C \sim \frac{\Pi^{\Updownarrow}/\Pi^{+}}{\Pi^{-}} \Updownarrow C'$ and $\tau\backslash C \sim \frac{\Pi^{\Updownarrow}/\Pi^{+}}{\Pi^{-}} \Updownarrow C'$ are decidable.

Proof. Without loss of generality, select a particular judgment of discourse $\widetilde{\alpha}\backslash\widetilde{C} \sim \frac{\widetilde{\Pi^{\Updownarrow}}/\widetilde{\Pi^{+}}}{\widetilde{\Pi^{-}}} \widetilde{\Updownarrow}$ $\widetilde{C}'$ (or $\widetilde{\tau}\backslash\widetilde{C} \sim \frac{\widetilde{\Pi^{\Updownarrow}}/\widetilde{\Pi^{+}}}{\widetilde{\Pi^{-}}} \widetilde{\Updownarrow} \widetilde{C}'$). Let S be the set of all trees which have the following properties: (1) a height no greater than $\widetilde{n}$; (2) an average branching factor no greater than four; and (3) each non-leaf node annotated with a judgment from the set J (with leaf nodes unannotated). We aim to show that, if a proof of our judgment of discourse exists, its proof tree is exactly one of the trees in this set.

If the judgment of discourse holds, we know by Lemma 5.20 that at least one proof tree has height no greater than $\widetilde{n}$. By inspection of Figure 3.14, we observe that each proof rule has at most four premises; thus, the proof tree has an average branching factor of less than four. By inspection of the same figure, each premise is either trivially decidable (a leaf node) or a subjudgment of compatibility (a non-leaf node). By Lemma 5.18, each subjudgment is drawn from the finite set J; also, all trees in S are annotated at all non-leaf nodes with a judgment from J.

So it suffices to determine if a tree appearing in the finite set S is a valid proof tree for the judgment of discourse according to the rules in Figure 3.14. By inspection of this rule set, verifying a proof is trivially decidable. We therefore have finitely many decidable operations to perform to determine if the judgment of discourse holds. As the judgment of discourse was selected without loss of generality, the type compatibility relation is decidable. □

5.7 Decidable Matching

Using the above proof, our next objective is to show that the application matching relation is decidable. This proof follows the same strategy as the above but is simpler: in a proof of application matching, few positions change in each of its subproofs than is the case with compatibility. As with the previous section, we select a judgment of discourse $\widetilde{\alpha}_0\ \widetilde{\alpha}_1\ {}^{\widetilde{\Pi}_1}_{\widetilde{C}}{\rightsquigarrow}^{\widetilde{\Pi}_2}_{\widetilde{\odot}}\ \widetilde{\alpha}_2\backslash\widetilde{C}'\ @\ \widetilde{C}''$ for discussion throughout this section.

5.7.1 Bounding the Judgment Places

As in Section 5.6.1, we define finite sets from which we intend to demonstrate that all subproofs draw the places of their judgments. In this case, our bounding sets are somewhat simpler:

Definition 5.22 (Matching Bounding Sets)**.** We provide the following bounding sets:

- Let $\widehat{A}'$ be the set of all type variables appearing in $\widetilde{A}$.
- Let $\widehat{\Pi}'$ be the set of all patterns appearing in $\widetilde{C}$.

Each of the above sets is, of course, finite.

5.7.1.1 Proving Finitely Many Possible Judgments

Next, we give a lemma to demonstrate that there are finitely many judgments of application matching in any proof tree of that relation. In a fashion similar to Section 5.6.1.3, we first give the lemma in a general form:

Lemma 5.23. *Suppose that $\alpha_0 \in \widehat{A}'$ and $\Pi_1 \subseteq \widehat{\Pi}'$. In all proofs of $\alpha_0\ \alpha_1\ {}^{\Pi_1}_{\widetilde{C}}{\rightsquigarrow}^{\Pi_2}_{\odot}\ \alpha_2\backslash C_1'\ @\ C_1''$, all subproofs are either (1) immediately decidable, (2) shown decidable by a previous lemma (and thus leaf nodes), or (3) proofs of the form $\alpha_0'\ \alpha_1\ {}^{\Pi_1'}_{\widetilde{C}}{\rightsquigarrow}^{\Pi_2'}_{\odot'}\ \alpha_2\backslash C_2'\ @\ C_2''$ where $\alpha_0' \in \widehat{A}'$ and $\Pi_1' \cup \Pi_2' \subseteq \widehat{\Pi}'$. Further, $\Pi_2 \subseteq \widehat{\Pi}'$.*

Proof. By induction on the size of the proof. We proceed by induction on the proof rule used.

Base Cases. If the Function Match rule is used, then $\Pi_2 = \{\pi\}$ for some π appearing in a function type in $\widetilde{C}$; thus, by definition, $\Pi_2 \subseteq \widehat{\Pi}'$. The first two premises are trivially decidable. The next premise is a use of the Sensible$(-,-,-,-)$ predicate, which is defined solely in terms of compatibility; thus, it and the fourth premise (a compatibility condition) are decidable by Lemma 5.21. If the Function Mismatch rule is used, then this argument proceeds the same way. If the Non-Function rule is used, the same argument applies except that $\Pi_2 = \emptyset$ (from which the lemma conclusion follows trivially) and there is one less compatibility premise.

Onion Left. If the Onion Left rule is used, the argument proceeds as in the Function Match case above except in handling the final premise. That premise is addressed by the inductive hypothesis; the inductive premises are satisfied because α_2 appears in a type which occurs in $\widetilde{C}$ and so is by definition in $\widehat{A}'$ and also because Π_1 and Π_2 are the same in this proof and its subproof.

Onion Right. If the Onion Right rule is used, we proceed as in the Onion Left rule. As a result of the inductive conclusion for the first application matching premise, we have that $\Pi_2 \subseteq \widehat{\Pi}'$; this is necessary to show the inductive premise that $\Pi_1 \cup \Pi_2 \subseteq \widehat{\Pi}'$ for the second application matching premise. Likewise, the inductive conclusion that $\Pi_3 \subseteq \widehat{\Pi}'$ is necessary to show this lemma's conclusion that $\Pi_2 \cup \Pi_3 \subseteq \widehat{\Pi}'$. $\square$

5.7.2 Finite Proof Search

As in Section 5.6.2, we now demonstrate that only finitely many trees need to be inspected to determine if a proof of a given application matching judgment exists. We first define the bound on the judgments to be inspected:

Definition 5.24 (Matching Proof Height Bound)**.** For the judgment of discourse, let J' be the Cartesian product $\widehat{A}' \times \mathcal{P}(\widehat{\Pi}') \times \mathcal{P}(\widehat{\Pi}')$; thus, J' is a finite set isomorphic to all judgments which can be constructed from the bounding sets in Definition 5.22. Let $\widetilde{n}' = |J'|$ be the size of this set.

Using this definition, we show (as with compatibility) that $\widetilde{n}'$ is a bound on the size of one of the proof trees (if any proof trees exist):

Lemma 5.25. The judgment $\widetilde{\alpha}_0\ \widetilde{\alpha}_1\ {}^{\widetilde{\Pi}_1}_{\widetilde{C}}\!\!\rightsquigarrow^{\widetilde{\Pi}_2}_{\widetilde{\odot}}\ \widetilde{\alpha}_2\backslash\widetilde{C}'\ @\ \widetilde{C}''$ holds iff there exists a proof of that judgment whose tree is of a height no larger than $\widetilde{n}'$.

Proof. By the same strategy as Lemma 5.20, inducting on the number of nodes in the proof tree and using Lemma 5.23 to demonstrate that any proof tree taller than $\widetilde{n}'$ includes a compressable path. $\square$

5.7.3 Proof

Finally, we show that application matching is decidable in much the same way as we showed for compatibility in Section 5.6.3.

Lemma 5.26. The relation $\alpha_0\ \alpha_1\ {}^{\Pi_1}_{C}\!\!\rightsquigarrow^{\Pi_2}_{\odot}\ \alpha'\backslash C'\ @\ C''$ is decidable.

Proof. By the same strategy as Lemma 5.21, using Lemma 5.21 to show the compatibility premises decidable. In particular, if any judgment of application matching holds, then at least one proof is of bounded height and average branching factor and all non-leaf nodes are identified by one of the judgments in the finite set J'. Thus, it suffices to consider the finitely many trees which have this property to determine if any of them are valid proof trees; as all non-matching premises in the rules are trivially decidable, we are finished. □

5.8 Decidable Constraint Closure

This section demonstrates that the single-step constraint closure relation is decidable. This property is actually trivial to show, especially in comparison to the compatibility and application matching relations:

Lemma 5.27. The relation $C \Longrightarrow C'$ is decidable.

Proof. By inspection of the rules in Figure 3.16. All premises are either trivially decidable or are decidable via Lemma 5.26. □

It is also necessary to show decidability for the inconsistency of a constraint set.

Lemma 5.28. Whether a constraint set is inconsistent (Definition 3.35) is decidable.

Proof. By Definition 3.35, a constraint set C is decidable iff $\alpha_0\ \alpha_1 <: \alpha_2 \in C$ such that $\alpha_0\ \alpha_1\ {}^{\emptyset}_{C}{\leadsto}^{\Pi}_{\bigcirc}\ \alpha'_1\backslash C'_1 \mathbin{@} C''_1$. By Lemma 5.26, the latter condition is decidable; thus, it suffices to test this condition for each constraint of the form $\alpha_0\ \alpha_1 <: \alpha_2$ appearing in C. Because C is finite, inconsistency is decidable. □

5.9 Closure Convergence

In the previous sections, we have demonstrated that a single constraint closure step is decidable. To show that typechecking is decidable, we will also need to demonstrate that only finitely many closure steps are necessary; that is, constraint closure must converge. Our strategy is similar to the above in that we select a finite set and show that it is an upper bound of the constraints which may appear during closure.

5.9.1 Bounding the Constraints

Bounding the constraints is a relatively straightforward process. Constraints are only introduced during closure by Transitivity (which is quite straightforward to bound), function bodies through type variable replacement, and the bindings produced by compatibility. Although we cannot bound constraint closure in general, we can do so for the arbitrary expression e_{INIT} (which, having been selected without loss of generality, allows us to demonstrate later that typechecking is decidable in general).

Definition 5.29 (Closure Bounding Sets)**.** We give the following bounding sets:

- Let $\Pi_{\text{MAX}} = \textsc{AllPoints}(\Pi)$ where Π is the set of all patterns appearing in $[\![e_{\text{INIT}}]\!]_{\text{E}}$ as well as all replacements of variables appearing in those patterns with variables appearing in A_{MAX}.

- $$\begin{aligned} \text{Let } C_{\text{MAX}} = \; & \{\alpha_0 <: \alpha_1 \mid \{\alpha_0, \alpha_1\} \subseteq A_{\text{MAX}}\} \\ \cup \; & \{\alpha_0 \; \alpha_1 <: \alpha_2 \mid \{\alpha_0, \alpha_1, \alpha_2\} \subseteq A_{\text{MAX}}\} \\ \cup \; & \{ \tau|_{\Pi^-}^{\Pi^+} <: \alpha \mid \alpha \in A_{\text{MAX}} \wedge \tau \in T_{\text{MAX}} \wedge (\Pi^+ \cup \Pi^-) \subseteq \Pi_{\text{MAX}}\} \end{aligned}$$

We know Π_{MAX} to be finite by Definition 5.11 and because both $[\![e_{\text{INIT}}]\!]_{\text{E}}$ and A_{MAX} are finite. We know C_{MAX} to be finite because it is the union of finite set comprehensions of finite sets.

5.9.1.1 Properties of the Constraint Bounds

In order to simplify the discussion of the constraint bounds provided above, we prove some simple lemmas.

Lemma 5.30. If τ appears in C_{MAX} then $\tau \in T_{\text{MAX}}$.

Proof. By inspection of Definition 5.29, all types appearing in C_{MAX} are taken directly from T_{MAX}. □

Lemma 5.31. If π appears in C_{MAX} then $\pi \in \Pi_{\text{MAX}}$.

Proof. By Definition 5.29, all patterns appearing in C_{MAX} either (1) appear in Π_{MAX} directly or (2) appear within a function type within C_{MAX}. By Lemma 5.30, all patterns in the latter set appear within a function type in T_{MAX}. By Definition 5.6, all such function types are taken from $[\![e_{\text{INIT}}]\!]_{\text{E}}$ with variables replaced by those in A_{MAX}. By Definition 5.29, Π_{MAX} includes all such patterns. □

Lemma 5.32. If α appears in C_{MAX} then $\alpha \in A_{\text{MAX}}$.

Proof. By inspection of Definition 5.29, all variables appearing in C_{MAX} are (1) taken directly from A_{MAX}, (2) appear in some τ in C_{MAX}, or (3) appear in some π in C_{MAX}. In the first case, we are finished.

In the second case, the variable appears in a type. By Lemma 5.30, that type must appear in T_{MAX}. By Definition 5.6, all types in T_{MAX} contain variables which are either replacements from A_{MAX} (in which case we are finished) or which appear in $[\![e_{\text{INIT}}]\!]_{\text{E}}$. By

Definition 5.2, A_{MAX} contains all variables appearing in $[\![e_{\text{INIT}}]\!]_{\text{E}}$ (because, by Definition 5.1, ϑ includes all variables in the original set). Thus, the second case is complete.

In the third case, the variable appears in a pattern. By Lemma 5.31, that pattern must appear in Π_{MAX}. By Definition 5.29, all patterns in Π_{MAX} contain only variables taken directly from $[\![e_{\text{INIT}}]\!]_{\text{E}}$ or replacement variables from A_{MAX}. The argument proceeds as in the previous case.

Because type variables appearing immediately within C_{MAX}, within types in C_{MAX}, and within patterns in C_{MAX} all appear within A_{MAX}, we are finished. $\square$

5.9.1.2 Bounding Bindings

To show that constraint closure respects the bounds in Definition 5.29, we must first demonstrate that the bindings produced by application matching (and thus compatibility) do so. We begin by showing this property for compatibility, for which the general lemmas and definitions provided in Section 5.6 are quite helpful.

Lemma 5.33. Suppose $\alpha\backslash C \sim \frac{\Pi^{\phi}/\Pi^{+}}{\Pi^{-}} \Downarrow C'$ such that $\alpha \in A_{\text{MAX}}$, $C \subseteq C_{\text{MAX}}$, and $\Pi^{\phi} \cup \Pi^{+} \cup \Pi^{-} \subseteq \Pi_{\text{MAX}}$. Then $C' \subseteq C_{\text{MAX}}$.

Proof. By Lemma 5.17, we have that $C' \subseteq \widehat{C}'$. By Definition 5.16, this is (for this judgment) equivalent to $C' \subseteq \{\, \tau|_{\Pi^{-}}^{\Pi^{+}} <: \alpha \mid \alpha \in \widehat{A} \wedge \tau \in \widehat{T} \wedge (\Pi^{+} \cup \Pi^{-}) \subseteq \widehat{\Pi} \}$. By Definition 5.29, it suffices to show each of $\widehat{A} \subseteq A_{\text{MAX}}$, $\widehat{T} \subseteq T_{\text{MAX}}$, and $\widehat{\Pi} \subseteq \Pi_{\text{MAX}}$.

Recall that $\Pi^{\phi} \cup \Pi^{+} \cup \Pi^{-} \subseteq \Pi_{\text{MAX}}$. By Definition 5.16, we have that $\widehat{\Pi} =$ AllPoints(Π) where Π contains all patters appearing in C as well as Π^{ϕ}, Π^{+}, and Π^{-}. Because all of these patters are contained in Π_{MAX}, we have that $\Pi \subseteq \Pi_{\text{MAX}}$. By Lemma 5.14, we have AllPoints$(\Pi) = \widehat{\Pi} =$ AllPoints(Π_{MAX}); because Π_{MAX} is defined immediately as the result of the all-points function on a set of patterns, Lemma 5.15 gives us that $\widehat{\Pi} \subseteq \Pi_{\text{MAX}}$.

Recall that $\widehat{A}$ contains all variables appearing in C as well as those appearing in Π^{ϕ}, Π^{+}, and Π^{-}. Because $C \subseteq C_{\text{MAX}}$, we know that all variables appearing in C must appear in C_{MAX}; we know the same for the pattern sets and Π_{MAX}. By inspection of Definitions 5.11 and 5.29, all such variables appearing in those sets must appear in A_{MAX}; thus, $\widehat{A} \subseteq A_{\text{MAX}}$.

So it remains to show that $\widehat{T} \subseteq T_{\text{MAX}}$. By Definition 5.16, $\widehat{T}$ contains only those types appearing in C. Since we have by assumption that $C \subseteq C_{\text{MAX}}$, we know that $\widehat{T}$ contains only types appearing in C_{MAX}. By Lemma 5.30, $\widehat{T} \subseteq T_{\text{MAX}}$ and we are finished. $\square$

We next show that application matching preserves this bound on bindings. This is relatively simple, as bindings in application matching are taken from a single compatibility proof; the only work is in showing that the premises of the previous lemma are satisfied. We also show the same for the function body it produces.

Lemma 5.34. Suppose $\alpha_0\ \alpha_1\ {}^{\Pi_1}_{C}\!\!\leadsto^{\Pi_2}_{\odot}\ \alpha'\backslash C'\ @\ C''$ such that $\alpha_1 \in A_{\text{MAX}}$, $C \subseteq C_{\text{MAX}}$, and $\Pi_1 \subseteq \Pi_{\text{MAX}}$. Then $\Pi_2 \subseteq \Pi_{\text{MAX}}$. Further, if $\odot = \bullet$ then $\alpha' \in A_{\text{MAX}}$ and $C' \subseteq C_{\text{MAX}}$, $C'' \subseteq C_{\text{MAX}}$.

Proof. By induction on the size of the proof. We proceed by case analysis on the rule used.

Function Match. If the Function Match rule is used, then we have $\Pi_2 = \{\pi\}$ for $\pi \to \alpha'\backslash C'$ which appears in C. We have by Lemma 5.32 that, because α' appears in C, $\alpha' \in A_{\text{MAX}}$. Because π appears in C, Lemma 5.31, we have that $\pi \in \Pi_{\text{MAX}}$ and so $\Pi_2 \subseteq \Pi_{\text{MAX}}$. This gives us the premises of Lemma 5.33, so we use it to conclude that $C'' \subseteq C_{\text{MAX}}$.

It remains to show that $C' \subseteq C_{\text{MAX}}$. We know by Lemmas 5.30, 5.31, and 5.32 that all types, patterns, and type variables appearing within any constraint in C' appear within T_{MAX}, Π_{MAX}, or A_{MAX} (respectively). By inspection of Definition 5.29, C_{MAX} contains all constraints which can be constructed from anything in those sets; thus, every constraint in C' must appear in C_{MAX} and this case is finished.

Other Cases. In all other cases, the proof is relatively trivial. If the Function Mismatch rule is used, then the argument proceeds as above. Because $\odot = \circ$, we need only show the property on the set Π_2. If the Non-Function rule is used, $\odot = \circ$ and $\Pi_2 = \emptyset$ and so that case is trivial. If the Onion Left rule is used, the proof is trivial by induction: the premises satisfy the inductive premises and the inductive conclusions satisfy the conclusions. Finally, if the Onion Right rule is used, then we have by induction on the first matching subproof that $\Pi_2 \subseteq \Pi_{\text{MAX}}$. By this and induction on the second matching subproof, we satisfy our conclusions. □

5.9.1.3 Bounding Constraint Closure

Using the above, we now demonstrate that constraint closure is closed under the bounding set C_{MAX}. We begin by observing a critical property of type variable replacements generated by our polymorphism model:

Lemma 5.35. Suppose $\alpha \in A_{\text{MAX}}$ and $\sigma = \Phi(\alpha, -)$. If $C \subseteq C_{\text{MAX}}$, then $\sigma\big|_{C'}(C) \subseteq C_{\text{MAX}}$ for any C'. Further, $\textsc{TFresh}(\sigma\big|_{C'}, C) \subseteq C_{\text{MAX}}$.

Proof. For any variable $\alpha \in A_{\text{MAX}}$, we have by Definition 5.2 that $\sigma\big|_{C'}(\alpha) \in A_{\text{MAX}}$ for any C' (since the result is in A_{MAX} whether it is a replacement or not). By Definition 5.29, all constraints appearing in C_{MAX} select their type variables arbitrarily from A_{MAX}; the same is true for type variable selections within the definitions of Π_{MAX} and T_{MAX}. This demonstrates that replacing a variable in A_{MAX} with another in A_{MAX} does not change whether a constraint appears in C_{MAX} and we are finished. □

We then proceed to the main bounding lemma:

Lemma 5.36. If $C \subseteq C_{\text{MAX}}$ and $C \Longrightarrow C'$ then $C \subseteq C_{\text{MAX}}$.

Proof. By case analysis on the rules used.

Transitivity. If the Transitivity rule is used, the only new constraint appearing in C' is $\tau|_{\Pi^-}^{\Pi^+} <: \alpha_2$. By Lemmas 5.30, 5.31, and 5.32, $\tau \in T_{\text{MAX}}$, $\Pi^+ \subseteq \Pi_{\text{MAX}}$, $\Pi^- \subseteq \Pi_{\text{MAX}}$, and $\alpha_2 \in A_{\text{MAX}}$. By Definition 5.29, this new constraint appears in C_{MAX}; thus, this case is finished.

Application. If the Application rule is used, then we begin by observing that, by Lemma 5.32, we have $\alpha_1 \in A_{\text{MAX}}$ and $\alpha_2 \in A_{\text{MAX}}$. Trivially, $\emptyset \subseteq \Pi_{\text{MAX}}$; thus, we use Lemma 5.34 to conclude that $\alpha_1' \in A_{\text{MAX}}$, $C_1' \subseteq A_{\text{MAX}}$, and $C_1'' \subseteq A_{\text{MAX}}$. By Definition 5.2, $\alpha_2' \in A_{\text{MAX}}$; by Lemma 5.35, $C_2' \subseteq C_{\text{MAX}}$ and $C_2'' \subseteq C_{\text{MAX}}$. Because $\alpha_2 \in A_{\text{MAX}}$ and $\alpha_2' \in A_{\text{MAX}}$, we have by Definition 5.29 that $\alpha_2' <: \alpha_2 \in C_{\text{MAX}}$. Therefore, $C' \subseteq C_{\text{MAX}}$ and this case is finished.

Merging. By the same strategy as the Transitivity case. In the Merging case, we also handle the intermediate and application constraints using this strategy (as Lemma 5.30 is sufficient in those cases). □

5.9.2 Proof

Using the above, we now demonstrate that constraint closure always converges. We do so by demonstrating that no closure chain can have more steps than there are constraints in C_{MAX} (excluding trivial steps of the form $C \Longrightarrow C$). We begin by observing a monotonicity property on constraint closure:

Lemma 5.37. If $C \Longrightarrow C'$ then $C \subseteq C'$.

Proof. Trivial by inspection of Figure 3.16. □

We then proceed to demonstrate that closure sequences must be bounded by the number of elements in C_{MAX}:

Lemma 5.38. Suppose $C_0 \subseteq C_{\text{MAX}}$. For any closure sequence $C_0 \Longrightarrow C_1 \Longrightarrow \ldots \Longrightarrow C_n$ such that $C_i \neq C_{i+1}$ for any i, $n \leq |C_{\text{MAX}}|$.

Proof. Because closure is monotonic (Lemma 5.37), we observe that $|C_i| < |C_{i+1}|$ for all i. We have that $|C_0| \geq 0$; thus, by induction on the length of the chain, we have that $|C_n| \geq n$. Also by induction on the length of the chain and using Lemma 5.36, we have that $C_n \subseteq C_{\text{MAX}}$; thus, $|C_n| \leq |C_{\text{MAX}}|$. Thus, $n \leq |C_{\text{MAX}}|$ and we are finished. □

5.10 Proving Decidability

In this section, we use the lemmas provided in the previous sections to prove the decidability theorem originally stated in Section 5.2.

Theorem 2 (Decidability)**.** With a finitely freshening polymorphism model, typechecking is decidable.

Proof. By Definition 3.36, a closed expression e_{INIT} typechecks iff $[\![e_{\text{INIT}}]\!]_{\text{E}} = \alpha\backslash C$ and $C \Longrightarrow^* C'$ implies that C' is consistent. The initial alignment function is trivially decidable by inspection of Figure 3.13; Lemma 5.28 demonstrates that the consistency of any C' is decidable. So it suffices to show that the set of all C' such that $C \Longrightarrow^* C'$ can be computed.

Lemma 5.38 demonstrates that all sequences $C \Longrightarrow^* C'$ have finitely many steps; Lemma 5.36 demonstrates that all such sequences contain only sets which are subsets of C_{MAX}, a finite set. There are finitely many lists of constraint sets of finitely bounded length which are subsets of C_{MAX}; it therefore suffices to consider each of these lists to determine if they represent valid closure sequences. For any such list, Lemma 5.27 demonstrates by induction on the length of the list that it is decidable whether that list represents a valid closure sequence. This is to say that, to compute all C' such that $C \Longrightarrow^* C'$, it suffices to consider finitely many judgments of a decidable relation; thus, the set of all such C' is computable and we are finished. $\square$

Although the above demonstrates that typechecking is decidable, it does not bound the complexity of the operation. We argue that typechecking is feasible by describing our proof-of-concept implementation in Chapter 9.

Chapter 6

Polymorphism in TinyBang

This chapter presents a polymorphism model suitable for TinyBang and, in particular, the use cases given in Chapter 2. The TinyBang type system defined in Section 3.5 is parametric in its polymorphism model, which is defined in terms of two functions: Φ, which polyinstantiates constraint sets; and Υ, which generates replacements from variable sets. Chapter 4 proves the type system system sound regardless of which polymorphism model is being used. To ensure that the type system is decidable, Section 3.5.5.1 outlines a set of necessary properties; Section 5.1 gives formal definitions for those properties.

Section 3.5.5.1 proposes a monomorphic model $\Phi_{\text{Mono}}/\Upsilon_{\text{Mono}}$ for the purposes of illustration. This model trivially satisfies the properties in Section 5.1, but it is clearly unsatisfying. To typecheck the examples in Chapter 2, we must use a considerably more sophisticated approach.

This chapter proceeds by formalizing a polymorphism model $\Phi_{\text{CR}}/\Upsilon_{\text{CR}}$ (which we call CR) that we have used in the TinyBang implementation. After we formalize this model, we prove that it satisfies the formal properties in Section 5.1. The TinyBang type system used with CR typechecks all of the examples provided thus far in this text.

6.1 Motivating CR

We design the CR model for a good trade-off between expressiveness and efficiency. More traditional approaches such as `let`-bound polymorphism lack the expressiveness needed for common programming patterns. For example, consider:

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

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

To provide sufficient precision, the TinyBang type system uses call site polymorphism: rather than polyinstantiating variables where they are named, we polyinstantiate functions when the function is invoked. We thus view every function as having a polymorphic type. In order to ensure maximal polymorphism, every variable bound by the body of the function should be polyinstantiated. This approach is inspired by flow analyses [62, 1] and has previously been ported to a type constraint context [74]. A program can have an unbounded number of function call sequences, so some compromise must be made for recursive programs. A standard solution to this problem is to chop off call sequences at some fixed point. While such arbitrary cutoffs may work for program analyses, they work less well for type systems: they make the system hard for users to understand and potentially brittle to small refactorings.

Our approach is to conservatively model the runtime stack (as in [43]). In particular, we abstract call sequences as restricted regular expressions called *contours*[1] and a single type variable associated with a regular expression will represent all runtime variables whose call strings match the regular expression. This model is powerful enough to assign useful types to the above code as well as all of the code in the overview. Like most polymorphic type systems (including ML's), CR is exponential when presented with pathological code examples. It behaves well in practice, however; implementation details are given in Chapter 9.

6.2 The Definition of CR

We now define CR. As described in Section 3.5.5.1, this consists of defining two functions: Φ_{CR} and Υ_{CR}. In this section, we provide some supporting definitions and then give definitions of those functions.

[1]We pick the name CR as an abbreviation of "contour regexes".

6.2.1 Contours

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

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

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

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

Definition 6.1 (Contour)**.** A *contour* $\mathcal{C}$ is a regular expression in which all symbols are type variables. Further, a contour is a concatenation only of literals and Kleene closures over disjunctions of literals. We use ϵ to denote the contour matching only the empty call string. We use $\varnothing$ to denote the contour matching no call strings.

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

Definition 6.2 (Well-Formed Contour)**.** A contour $\mathcal{C}$ is *well-formed* if each type variable appears at most once within it.

For example, the contour $\alpha_0(\alpha_1|\alpha_2)^*$ is well-formed but the contour $\alpha_0\alpha_1\alpha_0$ is not. The bijection only admits descriptors with well-formed contours. In theory, an unboundedly long contour could still consist of unique variables. Throughout our use of the polymorphism model, however, we bound the variables appearing in a contour with the variables appearing during initial alignment.

For the purposes of the definitions below, we define simple notation on contours:

Notation 6.3 (Contours)**.** We provide the following notation for contours:

- We write $\mathcal{C}\,||\,\mathcal{C}'$ to indicate the concatenation on contours in the sense of regular expressions.
- We write $\mathcal{C} \leq \mathcal{C}'$ to denote that the set of call strings matched by $\mathcal{C}$ is a subset of the call strings matched by $\mathcal{C}'$.
- We say that $\mathcal{C}$ *overlaps* $\mathcal{C}'$ (equiv. $\mathcal{C}$ and $\mathcal{C}'$ overlap) if there exists any call string accepted by both contours.
- We may write a type variable's descriptor in place of that type variable. For instance, we use `int` $<: \langle \alpha, \varnothing \rangle$ as shorthand for "`int` $<: \alpha'$ such that $\langle \alpha, \varnothing \rangle \Leftrightarrow \alpha'$".

Note that, while contours are closed under concatenation, well-formed contours are not.

Given the above, we can also provide a formal definition of type variable descriptors:

Definition 6.4 (Type Variable Descriptor)**.** A *type variable descriptor* for a variable α is a pair $\langle \alpha', \mathcal{C} \rangle$. We call α' the *root variable* of α; we call $\mathcal{C}$ the contour of α.

6.2.1.1 Least Well-Formed Contours

As contours are a restricted form of regular expression, we extend the partial ordering of regular expressions to cover them. Formally, we state the following:

Definition 6.5 (Contour Partial Ordering)**.** A contour $\mathcal{C}$ is *less* than a contour $\mathcal{C}'$ if the set of call strings accepted by the former is a strict subset of those call strings accepted by the latter.

As an example of the above, the contour ${\alpha_0}^*\alpha_1$ is less than the contour ${\alpha_0}^*{\alpha_1}^*$, but the contours ${\alpha_0}^*\alpha_1$ and $\alpha_0{\alpha_1}^*$ have no ordering. This ordering becomes important throughout the definition and discussion of $\Phi_{\mathrm{CR}}/\Upsilon_{\mathrm{CR}}$ because it gives rise to the notion of a *least well-formed contour* greater than another. As an example, the contour $\alpha_0\alpha_1\alpha_0$ is ill-formed; the least well-formed contour which subsumes it is $(\alpha_0|\alpha_1)^*$. We demonstrate here that, for any contour, a unique least well-formed contour exists.

Lemma 6.6. For any contour $\mathcal{C}$, there exists a unqiue least well-formed contour $\mathcal{C}'$ which subsumes it.

Proof. If $\mathcal{C}$ is well-formed, then $\mathcal{C} = \mathcal{C}'$ and this lemma is trivial. We thus consider the case in which $\mathcal{C}$ is ill-formed.

If $\mathcal{C}$ is ill-formed, then there exists some type variable α which appears in $\mathcal{C}$ more than once. Contours may only be comprised of terminal symbols and Kleene closures of disjunctions of terminal symbols (Definition 6.1); thus, the only means by which this duplication may be eliminated is to join the duplicate symbols together with all intervening symbols into a single Kleene closure (e.g. $\ldots\alpha_1\alpha_2(\alpha_3|\alpha_4)^*\alpha_1\ldots$ becomes $\ldots(\alpha_1|\alpha_2|\alpha_3|\alpha_4)^*\ldots$). This duplicate-eliminating process is confluent (because the Kleene closures are unordered) and decidable (by induction on the length of the contour): thus, the resulting well-formed contour $\mathcal{C}'$ subsuming $\mathcal{C}$ is unique. $\square$

6.2.2 Polyinstantiation

We begin our management of polymorphism with initial alignment. In particular, we always choose the type variable bijection such that each fresh type variable $[\![x]\!]_{\mathrm{V}}$ chosen by initial alignment has a fixed mapping.

Definition 6.7 (Initial Alignment Bijection)**.** If the variable x is defined within a pattern or the body of a function, then $\langle [\![x]\!]_{\mathrm{V}}, \varnothing\rangle \Leftrightarrow [\![x]\!]_{\mathrm{V}}$. Otherwise, $\langle [\![x]\!]_{\mathrm{V}}, \epsilon\rangle \Leftrightarrow [\![x]\!]_{\mathrm{V}}$ (to reflect the fact that α represents x at top level and in the empty call string).

Other than this initial set, new type variables are only introduced to closure via polyinstantiation or merging. We define the polyinstantiation function Φ_{CR} to instantiate variables appearing in a constraint set as follows:

Definition 6.8 (Polyinstantiation for CR)**.** Without loss of generality, consider $\alpha'_1 = \langle\alpha_1, \mathcal{C}_1\rangle$ and $\alpha'_2 = \langle\alpha_2, \mathcal{C}_2\rangle$. Let $\mathcal{C}'_1$ be the least well-formed contour such that, if some call string s is accepted by $\mathcal{C}_1$, then $s\,||\,\alpha_1$ is accepted by $\mathcal{C}'_1$. If $\mathcal{C}_2 = \varnothing$, then $\Phi_{\mathrm{CR}}(\alpha'_1, \alpha'_2) = \langle\alpha_2, \mathcal{C}'_1\rangle$; otherwise, $\Phi_{\mathrm{CR}}(\alpha'_1, \alpha'_2) = \alpha'_2$.

As an example, consider a call site $\alpha'_1 = \langle\alpha_1, \alpha_3\alpha_4\rangle$. This variable represents the program point α_1 when the call stack has a call from α_3 followed by a call from α_4. Consider also a type variable $\alpha'_2 = \langle\alpha_2, \varnothing\rangle$ representing program point α_2 inside of the function being called. For this polyinstantiation, we replace α'_2 with $\langle\alpha_2, \alpha_3\alpha_4\alpha_1\rangle$: this represents the type of program point α_2 when the call stack has a call from α_3 followed by a call from α_4 and then by a call from α_1.

As another example, consider $\alpha'_1 = \langle\alpha_1, \alpha_3\alpha_1\alpha_4\rangle$ with the same α_2 above. In this case, we have witnessed a call cycle: the call site α_1 reaches the call site α_4 which then returns to the call site α_1. Because $\alpha_3\alpha_1\alpha_4\alpha_1$ is not a well-formed contour, it is replaced by the least well-formed contour which covers this case: $\alpha_3(\alpha_1|\alpha_4)^*$.

We observe that Φ_{CR} does not freshen variables if they have already been polyinstantiated once. Only variables for which the contour is empty are replaced. This ensures that variables captured in the closure of a function do not present any difficulties.

6.2.3 Merging

Although the above polyinstantiation function is sufficient to express the polymorphism we require for our code examples, it is incredibly inefficient. If a set of call sites may be visited in any order before repeating (such as may occur in e.g. the body of an interpreter's evaluation function), then the number of contours created is factorial in the number of call sites. This problem is addressed by defining a merging function to unify variables which exist in overlapping contours; then, once a cycle is discovered, it collapses every variable which might participate in it. While this is a weakening in theory, the only code it affects in practice is code whose type correctness depends on the specific statically-known order in which it recurses; such code is simultaneously unusual and quite poor form, so we embrace this concession.

The key to the merging function is that it merges all visible contours which intersect. We formally define this function as follows:

Definition 6.9 (Merging for CR)**.** Let f be a function of two type variables returning a single type variable replacement function. For any variable α_0 which is not α_1, we define $\sigma = f(\alpha_1, \alpha_2)$ such that $\sigma(\alpha_0) = \alpha_0$. For α_1, we let $\sigma(\alpha_1) = \alpha_2$ if $\mathcal{C}_1$ overlaps $\mathcal{C}_2$ and $\sigma(\alpha_1) = \alpha_1$ otherwise. Given this definition, $\Upsilon_{\mathrm{CR}}(A) = \{f(\alpha_1, \alpha_2) \mid \alpha_1 \in A, \alpha_2 \in A\}$.

As formalized here, the merging function does not reduce the number of constraints during closure; to the contrary, it considerably increases them. Due to the nature of this replacement, however, it becomes possible in an implementation to *replace constraints* rather than adding new constraints with the replaced variables. Consider two constraints $\langle \alpha_1, \mathcal{C}_1 \rangle <: \alpha_2$ and $\langle \alpha_1, \mathcal{C}'_1 \rangle <: \alpha_2$. When $\mathcal{C}_1 \leq \mathcal{C}'_1$, the first constraint becomes redundant (due to the definitions of inconsistency and typechecking). We defer discussion of implementation details to Chapter 9.

6.3 Finite Freshening of CR

As mentioned above, Chapter 4 demonstrates that the type system is sound regardless of which polymorphism model is used. To show that the TinyBang type system is decidable when used with CR, however, we must demonstrate the finite freshening property defined in Section 5.1. We do so in this section.

6.3.1 Freshness Bounding Function

The first step in this process is to define the *freshness bounding function* for the polymorphism model. As described in Section 5.1, this function must be able to identify all of the (finitely many) type variables which will be generated by the polymorphism model based on the type variables gathered from initial alignment. We give such a function below:

Definition 6.10 (Freshness Bounding for CR)**.** For CR, we define the freshness bounding function $\vartheta_{\mathrm{CR}}(e)$ as follows. Let A be the set of type variables appearing in $[\![e]\!]_{\mathrm{E}}$. Let A' be the root variables of the variables appearing in A; that is, $A' = \{\alpha \mid \langle\alpha, \mathcal{C}\rangle \in A\}$. Let $\overset{\sqcup}{\mathcal{C}}$ be the set of all well-formed contours which can be constructed from the variables appearing in A'. Then $\vartheta_{\mathrm{CR}}(e) = \{\langle\alpha, \mathcal{C}\rangle \mid \alpha \in A', \mathcal{C} \in \overset{\sqcup}{\mathcal{C}}\}$.

6.3.2 Properties of Freshness Bounding

In order to demonstrate that the function above is a suitable freshness bounding function, we must demonstrate that it has the properties specified in Definition 5.1. We begin with the assertion that it identifies all variables appearing in the initial alignment of the expression:

Lemma 6.11. $\vartheta_{\mathrm{CR}}(e)$ is a finite superset of the variables appearing in $[\![e]\!]_{\mathrm{E}}$.

Proof. Let A be the set of type variables appearing in $[\![e]\!]_{\mathrm{E}}$. By Definition 6.7, each of these variables is of the form $\langle\alpha, \mathcal{C}\rangle$ where $\mathcal{C} = \epsilon$ or $\mathcal{C} = \varnothing$. By Definition 6.10, $\vartheta_{\mathrm{CR}}(e)$ contains all variables of the form $\langle\alpha', \mathcal{C}'\rangle$ where α' appears as the root of some variable in A and $\mathcal{C}'$ is a well-formed contour constructed of the roots of the variables in A. Because both ϵ and $\varnothing$ are trivially well-formed, any variable $\langle\alpha, \mathcal{C}\rangle \in A$ also appears in $\vartheta_{\mathrm{CR}}(e)$.

It remains to show that $\vartheta_{\mathrm{CR}}(e)$ is finite. We observe that $\vartheta_{\mathrm{CR}}(e)$ is isomorphic to the product of the root variables appearing in $[\![e]\!]_{\mathrm{E}}$ (finite because e is finite) and the set of well-formed contours which can be constructed from a finite set of variables. As variables cannot repeat within well-formed contours, as there are finitely many variables, and as the syntax of a contour does not contain any repeating terms, this set of well-formed contours is finite. Therefore, $\vartheta_{\mathrm{CR}}(e)$ is finite and we are finished. □

We next demonstrate that the polyinstantiation function does not escape the bounds of this set.

Lemma 6.12. For each α_1 and α_2 in $\vartheta_{\mathrm{CR}}(e)$, $\Phi_{\mathrm{CR}}(\alpha_1, \alpha_2) \in \vartheta_{\mathrm{CR}}(e)$.

Proof. Let A be the set of roots of variables appearing in $[\![e]\!]_{\mathrm{E}}$; then we have by Definition 6.10 that $\alpha_1 = \langle\alpha_1', \mathcal{C}_1\rangle$ and $\alpha_2 = \langle\alpha_2', \mathcal{C}_2\rangle$ where $\alpha_1' \in A$, $\alpha_2' \in A$, and both $\mathcal{C}_1$ and $\mathcal{C}_2$

are well-formed contours constructed from variables in A. If $\mathcal{C}_2 \neq \varnothing$ then, by Definition 6.8, $\Phi_{\mathrm{CR}}(\alpha_1, \alpha_2) = \alpha_2$ and, because we already know $\alpha_2 \in \vartheta_{\mathrm{CR}}(e)$, we are finished.

Otherwise, $\mathcal{C}_2 = \varnothing$. Let $\mathcal{C}'_1$ be the least well-formed contour accepting all strings which are accepted by $\mathcal{C}_1 \,||\, \alpha'_1$; this contour is, by definition, both well-formed and constructed from variables in A. In this case, $\Phi_{\mathrm{CR}}[\Phi_{\mathrm{CR}}]\alpha_1\alpha_2 = \langle \alpha'_2, \mathcal{C}'_1 \rangle$, so it suffices to show that $\langle \alpha'_2, \mathcal{C}'_1 \rangle \in \vartheta_{\mathrm{CR}}(e)$. By Definition 6.10, $\vartheta_{\mathrm{CR}}(e)$ contains all variables with a root variable from the set of roots in $[\![e]\!]_{\mathrm{E}}$ (defined here as A) and a well-formed contour constructed from those variables. Because $\alpha'_2 \in A$ and because $\mathcal{C}'_1$ is such a well-formed contour, we are finished. □

Finally, we show that the merging function also remains within the bounds of the set.

Lemma 6.13. For every $\alpha \in \vartheta_{\mathrm{CR}}(e)$ and $A \subseteq \vartheta_{\mathrm{CR}}(e)$ and $\sigma \in \Upsilon_{\mathrm{CR}}(A)$, we have that $\sigma(\alpha) \in \vartheta_{\mathrm{CR}}(e)$.

Proof. Without loss of generality, select some $\alpha_0 \in \vartheta_{\mathrm{CR}}(e)$ and $A \subseteq \vartheta_{\mathrm{CR}}(e)$. We have by Definition 6.9 that each σ appearing in $\Upsilon_{\mathrm{CR}}(A)$ is derived from some pair of variables α_1 and α_2 which appear in A. Select some σ and the corresponding α_1 and α_2 without loss of generality. If $\alpha_0 \neq \alpha_1$ then $\sigma(\alpha_0) = \alpha_0$ and we are finished.

Otherwise, $\alpha_0 = \alpha_1$. We have two cases: either the contours of α_1 and α_2 overlap or they do not. If they do not, then $\sigma(\alpha_0) = \alpha_0$ and we are finished. Otherwise, let $\alpha_1 = \langle \alpha'_1, \mathcal{C}_1 \rangle$ and let $\alpha_2 = \langle \alpha'_2, \mathcal{C}_2 \rangle$. We have that $\sigma(\alpha_0) = \alpha_2$; because α_2 also appears within A, we are finished. □

6.3.3 Statement of Finite Freshening

Finally, we can formally state the finite freshening property of CR:

Theorem 3 (CR is Finitely Freshening)**.** The polymorphism model $\Phi_{\mathrm{CR}}/\Upsilon_{\mathrm{CR}}$ is finitely freshening.

Proof. By Definition 5.1 and Lemmas 6.11, 6.12, and 6.13. □

6.4 Computability of CR

The proof of decidability also assumes that the functions defining the polymorphism model are computable. We demonstrate this relatively trivial property here, beginning with the polyinstantiation function.

Lemma 6.14. Φ_{CR} is computable.

Proof. All evaluation of $\Phi_{\text{CR}}(\alpha_1, \alpha_2)$ is trivially computable except for the determination of the least well-formed contour for an arbitrary contour. That determination is shown to be decidable by Lemma 6.6. We are therefore finished. □

We then give a lemma for the same property on merging:

Lemma 6.15. Υ_{CR} is computable.

Proof. Trivial by inspection of Definition 6.9: the function computes a finite set comprehension over a finite set. □

6.5 The Expressiveness of CR

As stated above, this polymorphism model is expressive enough to typecheck all of the examples presented above. In general, it loses no information except at type variable unions (an inevitable consequence of the type system) and at recursion. We discuss related work in the field of abstract interpretation and then present a concrete example of CR's expressiveness.

As stated in Chapter 3.5, TinyBang's type system is similar to abstract interpretations; it is appropriate, then, to compare its expressiveness to abstract interpretation polyvariance models [33]. As a point of reference, the monomorphic system $\Phi_{\text{Mono}}/\Upsilon_{\text{Mono}}$ is analogous to 0-CFA, the degenerate form of Shivers' seminal work in control-flow analysis [62].

The more general form, k-CFA, bears more resemblance to $\Phi_{\text{CR}}/\Upsilon_{\text{CR}}$ in that it uses a call history of depth k to distinguish between the different ways a single point in a program is reached. Though useful in many program analyses, k-CFA has the undesirable property of producing arbitrary cutoffs – the selection of k is insensitive to the particular needs of a programmer at any given point in the program – which, in a type system, would yield a sense of brittleness as minor refactorings could introduce false type errors [76, 68]. In comparison to models involving arbitrary cutoff, we believe the information loss of CR to be intuitive.

Might and Shivers' ΔCFA [43] make use of call string abstractions in a fashion similar to the CR polymorphism model presented in this chapter. In that work, however, call string abstractions focus on the number of transitions between functions and not the call sites themselves. As a result, ΔCFA and CR are differently expressive; neither subsumes the other.

We call out the above polyvariance models because they are especially comparable to CR. Other common polyvariance models (such as CPA [1], which creates new contours based on the types of function arguments) are not similar enough to allow straightforward comparison. For a deeper discussion of the expressiveness and complexity of other polyvariance models, we refer the reader to [33].

6.5.1 An Example

For an example of how CR loses (and retains) precision, consider the following nested TinyBang code:

```
1 let fixpoint = fun f -> (fun g -> fun x -> g g x) (fun h -> fun y -> f (h h) y) in
2 let map = fun f -> fixpoint (fun self -> fun x ->
3             ( (fun ‘Nil _ -> ‘Nil ())
4             & (fun ‘Hd y * ‘Tl t -> ‘Hd (f y) & ‘Tl (self t)) )) in
5 let fn = (fun ‘A x -> ‘Z x) &
6          (fun ‘B y -> ‘Y y) in
7 map atoz (‘Hd (‘A ()) & ‘Tl (‘Hd (‘B ()) & ‘Tl ‘Nil ()))
```

Informally, this code is mapping the list [‘A (), ‘B ()] using the function `fn`; it results in the list [‘Z (), ‘Y ()]. In the type system, however, the `atoz` function is called only from one place: the `f y` invocation on line 4. As a result, the body of the function only receives a single contour for both invocations of `fn` and the resulting list is given a union type. However, the code

```
1 let fixpoint = fun f -> (fun g -> fun x -> g g x) (fun h -> fun y -> f (h h) y) in
2 let map2 = f -> fixpoint (fun self -> fun x ->
3              ( (fun ‘Nil _ -> ‘Nil ())
4              & (fun ‘Hd y * ‘Tl ‘Nil _ -> ‘Hd (f y) & ‘Tl ‘Nil ())
5              & (fun ‘Hd y * ‘Tl (‘Hd z & ‘Tl t) ->
6                     ‘Hd (f y) & ‘Tl (‘Hd (f z) & ‘Tl (self t))) )) in
7 let fn = (fun ‘A x -> ‘Z x) &
8          (fun ‘B y -> ‘Y y) in
9 map2 atoz (‘Hd (‘A ()) & ‘Tl (‘Hd (‘B ()) & ‘Tl ‘Nil ()))
```

does not exhibit the same behavior. This code is identical in behavior to the previous example, but the `map2` function maps *two* elements at a time. As a result, there are two call sites on line 6 and so two union types are ascribed to the resulting list: one for the even-indexed positions and the other for the odd-indexed positions. In this example, the type given to the list is quite precise. If we extend this example with an additional case,

```
1 let fixpoint = fun f -> (fun g -> fun x -> g g x) (fun h -> fun y -> f (h h) y) in
2 let map3 = f -> fixpoint (fun self -> fun x ->
3              ( (fun ‘Nil _ -> ‘Nil ())
4              & (fun ‘Hd y * ‘Tl ‘Nil _ -> ‘Hd (f y) & ‘Tl ‘Nil ())
5              & (fun ‘Hd y * ‘Tl (‘Hd z & ‘Tl ‘Nil _) ->
 6                     ‘Hd (f y) & ‘Tl (‘Hd (f z) & ‘Tl (‘Nil ())))) in
 7              & (fun ‘Hd y * ‘Tl (‘Hd z & ‘Tl (‘Hd w & ‘Tl t))) ->
 8                     ‘Hd (f y) & ‘Tl (‘Hd (f z) & ‘Tl (‘Hd (f w) & ‘Tl (self t))) ))
 9 let fn = (fun ‘A x -> ‘Z x) &
10          (fun ‘B y -> ‘Y y) in
11 map3 atoz (‘Hd (‘A ()) & ‘Tl (‘Hd (‘B ()) & ‘Tl ‘Nil ()))
```

then the call to `map3` reaches line 6, which doesn’t recurse. This invocation is indistinguishable from calling a non-recursive function and the type of the resulting list is exactly correct: the type system has knowledge that the first element’s type is `‘Z ()`, the second element’s type is `‘Y ()`, and that the list contains exactly two elements. While this is not a scalable way to achieve more type precision on lists in general, these examples demonstrate that TinyBang equipped with CR is capable of inferring precise types for fairly complex data structures.

Chapter 7

Built-Ins

The formalization presented in Chapter 3 gives a restricted version of ANF TinyBang which, for purposes of illustration, elides basic components such as integer types and state. This chapter briefly formalizes a version of ANF TinyBang including those operations. The soundness and decidability arguments for these new components follow largely in the form of those in Chapters 4 and 5.

We introduce these built-in operations via a general framework for n-ary operators. The examples we provide below demonstrate how integers, addition, and state can be added to TinyBang; similar primitives follow the same pattern.

7.1 Environment

To ensure that this framework is sufficiently general to properly handle state (which makes cyclic data references possible), our environment must be able to handle cyclic scope. Conveniently, none of our discussion thus far as relied upon the ordering of any E. To this end, we amend the definition of well-formed expressions: every clause in an expression $E \,||\, e$ is closed in the variable x if *any* clause in E binds x (regardless of its position). We formalize this definition as follows:

Definition 7.1 (Bound and Free Variables (Revised))**.** We modify Definition 3.6 with the following handling of expressions:

Expressions

$$\textsc{bv}(E) = \{x \mid x \mathbin{\texttt{=}} v \in E\} \qquad \textsc{fv}(E) = \bigcup\{\textsc{fv}(v) \mid x \mathbin{\texttt{=}} v \in E\} - \textsc{bv}(E)$$

$$\textsc{bv}(E \,||\, \overset{n}{\overrightarrow{s}}) = \textsc{bv}(E \,||\, \overset{n-1}{\overrightarrow{s}}) \cup \textsc{bv}(s_n) \qquad \textsc{fv}(E \,||\, \overset{n}{\overrightarrow{s}}) = \textsc{fv}(E \,||\, \overset{n-1}{\overrightarrow{s}}) \cup (\textsc{fv}(s_n) - \textsc{bv}(\overset{n-1}{\overrightarrow{s}}))$$

$$\begin{array}{rcll} s & ::= & x \mathrel{=} \boxdot \, \vec{x} \mid \ldots & \textit{clauses} \\ \boxdot & ::= & + \mid \ldots & \textit{built-ins} \end{array}$$

$$\textsc{Builtin}\quad \frac{\boxdot(E, x', \overset{n}{\vec{x}}) = \langle E', x'' \rangle}{E \,||\, [x' \mathrel{=} \boxdot \, \overset{n}{\vec{x}}] \,||\, e \longrightarrow^{1} E' \,||\, [x' \mathrel{=} x''] \,||\, e}$$

Figure 7.1: Built-In Framework Operational Semantics Additions

$$c \quad ::= \quad \boxdot \, \vec{\alpha} <: \alpha \mid \ldots \qquad \textit{constraints}$$

$$\llbracket x' \mathrel{=} \boxdot \; \overset{n}{\overrightarrow{x_{\square}}} \rrbracket_{\mathrm{E}} = \llbracket x' \rrbracket_{\mathrm{V}} \setminus \boxdot \; \overset{n}{\overrightarrow{\llbracket x_{\square} \rrbracket_{\mathrm{V}}}} <: \llbracket x' \rrbracket_{\mathrm{V}}$$

$$\textsc{Builtin}\quad \frac{\boxdot \overset{n}{\vec{\alpha}} <: \alpha' \qquad \boxdot(C, \alpha', \overset{n}{\vec{\alpha}}) = \langle C', \alpha'' \rangle}{C \Longrightarrow^{1} C \cup C' \cup \{\alpha'' <: \alpha'\}}$$

Figure 7.2: Built-In Framework Type System Additions

We modify Definition 3.20 in a corresponding fashion.

7.2 Framework

The built-in framework makes additions to the language grammar and operational semantics; these additions appear in Figure 7.1. Each individual built-in operator is represented by a symbol in the grammar $\boxdot$. We define for each builtin a metatheoretic function $\boxdot(-,-,-)$ that defines the operator's behavior; this function is from an environment, a call site variable, and a list of operand variables onto a pair between a resulting environment and value.

The contents of Figure 7.2 parallel these modifications at the type system level. We also overload our operator function: we define a metatheoretic function $\boxdot(-,-,-)$ from a constraint set, call site type variable, and list of operand type variables onto a pair between a resulting constraint set and a resulting type variable.

To create a built-in operator, it is then sufficient to define the value and type level metatheoretic functions. Note that these functions may be partial; not every built-in can accept every variable. If the value level built-in function is partial, then evaluation becomes

$$\begin{array}{rcll} v & ::= & \mathbb{Z} \mid \ldots & \textit{values} \\ \varphi & ::= & x : \texttt{int} \mid \ldots & \textit{filters} \end{array}$$

Let $\textsc{CtrPat}(x : \texttt{int} \backslash F)$ be true.

$$\textsc{Integer}\quad \frac{\textsc{CtrPat}(P) \qquad P^{\obar} \cup P^{+} = \{x : \texttt{int} \backslash F \in P\} \qquad P^{-} = P - (P^{\obar} \cup P^{+})}{\mathbb{Z}\backslash E \sim \frac{P^{\obar}/P^{+}}{P^{-}} \Uparrow [x = \mathbb{Z}]}$$

Figure 7.3: Built-In Integer Operational Semantics Additions

stuck. Correspondingly, we modify the definition of inconsistency in the following way:

Definition 7.2 (Inconsistency)**.** A constraint set C is *inconsistent* either as indicated in Definition 3.35 or iff there exists some $\boxdot \overset{n}{\overrightarrow{\alpha}} <: \alpha' \in C$ such that $\boxdot(C, \alpha', \overset{n}{\overrightarrow{\alpha}})$ is undefined.

In order to preserve the soundness proof, it is necessary for each pair of functions $\boxdot(E, x', x)$ and $\boxdot(C, \alpha', \alpha)$ to preserve the bisimulation property of Definition 4.1. Although this imposes the burden of showing soundness on an operator-by-operator basis, it permits the operators considerable freedom in the constraints that they generate.

7.3 Integers and Addition

For simplicity, integers were not included in the formalization in Chapter 3. We introduce integer values, patterns, types, and operations here to illustrate built-ins. We begin with the changes to the operational semantics shown in Figure 7.3. This figure introduces literal integer values and an integer pattern filter. It defines integer patterns to be constructor patterns and it gives a compatibility rule which operates on integer patterns and integer types.

The integer pattern filter defined in this figure is of the form $x : \texttt{int}$ rather than simply `int`. Note that a pattern is then of the form $x_0\backslash\{x_1 : \texttt{int} <: x_0\}$. There are two variables associated with the filter: one which is part of the structure of the pattern (x_0) and one which is part of the filter itself (x_1). The intention here is to allow the *strict* binding of the integer. A strict binding excludes all content which does not participate in a pattern match. For this case, consider matching the above pattern with the value `5 & ‘A 0`. Although x_0 would be bound to the entire value, x_1 would be bound to the `5` only.

This strict binding is critical to the definition of an addition function which can correctly handle onion values. We give that definition here:

$$\begin{array}{rcll} \tau & ::= & \texttt{int} \mid \ldots & \textit{types} \\ \phi & ::= & \alpha : \texttt{int} \mid \ldots & \textit{filters} \end{array}$$

Let $\textsc{CtrPat}(\alpha : \texttt{int} \backslash \Psi)$ be true.

$$\textsc{Integer}\quad \frac{\textsc{CtrPat}(\Pi) \qquad \Pi^{\diamond} \cup \Pi^{+} = \{\alpha : \texttt{int} \backslash \Psi \in \Pi\} \qquad \Pi^{-} = \Pi - (\Pi^{\diamond} \cup \Pi^{+})}{\texttt{int} \backslash C \sim \frac{\Pi^{\diamond}/\Pi^{+}}{\Pi^{-}} \diamond \{\texttt{int} <: \alpha\}}$$

Figure 7.4: Built-In Integer Type System Additions

Definition 7.3 (Operational Semantics of Addition)**.** Suppose $p = x_0 : \texttt{int} \backslash F$ for some x_0. We define $\texttt{+}\,(E, x', [x_1, x_2]) = \langle E \,||\, [x_3 \texttt{=} n_3], x_3 \rangle$ when all of the following are true:

- $x_1 \backslash E \sim \frac{\{p\}/\emptyset}{\emptyset} \diamond E'$ such that $x_0 \texttt{=} n_1 \in E'$,
- $x_2 \backslash E \sim \frac{\{p\}/\emptyset}{\emptyset} \diamond E'$ such that $x_0 \texttt{=} n_2 \in E'$,
- n_3 is the sum of n_1 and n_2, and
- $x_3 = \boldsymbol{\alpha}(x', x')$.

This definition uses the strict binding properties of the integer pattern to extract just the integer component from the value. We then use this integer component to determine the addition. In this way, we have a proper type-driven destructor for primitive types in onions. This allows us to evaluate an expression such as `(5 & ‘A 4) + 1` to an appropriate value (such as `6`). The variable x' is used here both as the call site and, in its freshened form, the return variable; this is solely for convenience.

The above extensions define a TinyBang operational semantics which includes addition. To typecheck this addition, we provide similar, nearly parallel extensions to the type system. These modifications can be found in Figure 7.4. These rules are nearly syntactically identical to the operational semantics rules; alignment between them follows exactly as in the soundness proof in Chapter 4. This parallel includes the strict binding variable seen in the evaluation system's pattern filter.

Finally, we define a type system function for closing over addition operations:

Definition 7.4 (Typing of Addition)**.** Suppose $\pi = \alpha_0 : \texttt{int} \backslash \Psi$ for some α_0. We define $\texttt{+}\,(C, \alpha', [\alpha_1, \alpha_2]) = \langle C \cup \{\texttt{int} <: \alpha_3\}, \alpha_3 \rangle$ when all of the following are true:

- $\alpha_1 \backslash C \sim \frac{\{\pi\}/\emptyset}{\emptyset} \diamond C'$ such that $\texttt{int} <: \alpha_0 \in C'$,
- $\alpha_2 \backslash C \sim \frac{\{\pi\}/\emptyset}{\emptyset} \diamond C'$ such that $\texttt{int} <: \alpha_0 \in C'$,
- $\alpha_3 = \Phi(\alpha', \alpha')$.

$$
\begin{array}{rcll}
v & ::= & \ldots \mid \texttt{ref}\, x & \textit{values} \\
\varphi & ::= & \ldots \mid \texttt{ref}\, x & \textit{filters}
\end{array}
$$

Let $\textsc{CtrPat}(\texttt{ref}\, x\backslash F)$ be true.

$$
\textsc{Reference}\quad
\frac{\textsc{CtrPat}(P) \qquad P^{⌽} \cup P^{+} = \{\texttt{ref}\, x\backslash F \in P\} \qquad P^{-} = P - (P^{⌽} \cup P^{+})}{\texttt{ref}\, x'\backslash E \sim \frac{P^{⌽}/P^{+}}{P^{-}} \mathbin{⌽} [x = x']}
$$

Figure 7.5: Built-In State Operational Semantics Additions

7.4 State

The previous section demonstrated how a simple primitive type and its built-in operators can be introduced to TinyBang. This section shows how a somewhat more complex property – state – can be introduced. We introduce state via explicit reference cells. We introduce a primitive reference cell type as well as an operator for assigning to that cell. Retrieval from cells is accomplished by pattern matching; in fact, reference cells are grammatically very similar to labels. We introduce the new grammar rules and other definition changes in Figure 7.5.

The Reference rule of compatibility added here has the property of binding the variable under the reference pattern to the contents of the reference type. This has the effect of extracting the contents of that cell as evaluation proceeds. In order to allow cells to be updated, we define an operator `:=` and define its behavior as follows:

Definition 7.5 (Operational Semantics of Assignment)**.** Suppose $p = \texttt{ref}\, x_0\backslash F$ for some x_0. We define $\texttt{:=}(E, x', [x_1, x_2]) = \langle E' \,||\, [x_3 = \texttt{()}], x_3\rangle$ when all of the following are true:

- $x_1\backslash E \sim \frac{\{p\}/\emptyset}{\emptyset} \mathbin{⌽} E''$ such that $x_0 = x'' \in E''$ (that is, the first argument has a cell with body x''),
- $x_2 = v \in E$ (that is, x_2 has a value),
- E' is exactly E where $x'' = v'$ is replaced by $x'' = v$ (that is, the cell's content variable is assigned to the value), and
- $x_3 = \boldsymbol{\alpha}(x', x')$ (so a return variable can be defined).

Here, the definition of this function uses the bindings from compatibility to identify the content variable of the reference cell so that its clause in the environment may be modified. The assignment operation always returns `()`, which it stores in a fresh variable.

To modify the type system accordingly, we again perform a perfect syntactic translation of the operational semantics which appears in Figure 7.6. We then give a definition

$$\begin{array}{rcll} \tau & ::= & \ldots \mid \texttt{ref}\,\alpha & \textit{types} \\ \phi & ::= & \ldots \mid \texttt{ref}\,\alpha & \textit{filters} \end{array}$$

Let $\textsc{CtrPat}(\texttt{ref}\,\alpha\backslash\Psi)$ be true.

$$\textsc{Reference}\quad \frac{\textsc{CtrPat}(\Pi) \qquad \Pi^{\phi} \cup \Pi^{+} = \{\texttt{ref}\,\alpha\backslash\Psi \in \Pi\} \qquad \Pi^{-} = \Pi - (\Pi^{\phi} \cup \Pi^{+})}{\texttt{ref}\,\alpha'\backslash C \sim \frac{\Pi^{\phi}/\Pi^{+}}{\Pi^{-}} \Uparrow \{\alpha' <: \alpha\}}$$

Figure 7.6: Built-In State Type System Additions

of the type system assignment function as follows:

Definition 7.6 (Typing of Assignment)**.** Suppose $\pi = \texttt{ref}\,\alpha_0\backslash\Psi$ for some α_0. We define $\texttt{:=}\,(C, \alpha', [\alpha_1, \alpha_2]) = \langle C \cup C' \cup \{\,\texttt{()} <: \alpha_3\}, \alpha_3\rangle$ when all of the following are true:

- $\alpha_1\backslash C \sim \frac{\{\pi\}/\emptyset}{\emptyset} \Uparrow C''$ such that $\alpha'' <: \alpha_0 \in C''$,
- $C' = \{\tau|_{\Pi^-}^{\Pi^+} <: \alpha'' \mid \tau|_{\Pi^-}^{\Pi^+} <: \alpha_2 \in C\}$ (that is, all lower bounds of the argument flow into the cell), and
- $\alpha_3 = \Phi(\alpha', \alpha')$.

As in evaluation, this function identifies the content variable of the reference cell. Where evaluation copies the value of the argument into the cell, this function copies the lower-bounding type of the argument into the cell. The set comprehension is necessary because there may be many lower-bounding types. Unlike the evaluation system, however, no replacement is necessary; the constraint system is monotonic in nature, so we simply add the new lower bounds. This produces a flow-insensitive model of state in TinyBang: the type of the contents of a cell is the union of all types it has ever been or will be.

Chapter 8

LittleBang

The previous chapters have defined a formal type system for ANF TinyBang, a mechanism by which built-in operations such as addition can be defined, and an A-translation mechanism to convert nested TinyBang expressions into the ANF language. This chapter defines LittleBang, a language with a variety of common scripting language features, by encoding them into nested TinyBang. The initial exploration of several of these encodings was formalized in [26]; we give many of them again here for the purposes of refinement and completeness. In each case, we present the formal encoding and discuss our choices especially with respect to static typing.

8.1 The Encoding Process

Rather than presenting a formal LittleBang syntax all at once, we begin with a basic LittleBang syntax and incrementally add to its grammar as we define their encodings. This basic LittleBang syntax appears in Figure 8.1. This syntax is nearly identical to the nested TinyBang syntax in Figure 2.1, but the concrete syntax of functions and application have been changed in order to distinguish them from the forms we will give them in LittleBang. Specifically, TinyBang's functions are prefixed with `tbfun` (rather than `fun`) and TinyBang application is written using the explicit infix operator $\diamond$.

The syntax in Figure 8.1 is simply a different concrete form of nested TinyBang; the manner in which it is translated into TinyBang is evident. For each encoding, we give an extension to the syntax in Figure 8.1 and then provide an encoding function which translates this extension into the syntax which has already been defined. As the chapter proceeds,

$$\begin{array}{rcll}
e & ::= & x \mid \texttt{()} \mid \mathbb{Z} \mid \mathbb{S} \mid l\ e \mid e \,\texttt{\&}\, e \mid \texttt{ref}\, e \mid \texttt{!}\, e \mid x \,\texttt{:=}\, e \,\texttt{in}\, e \mid & \textit{expressions} \\
 & & \texttt{let}\, x \,\texttt{=}\, e \,\texttt{in}\, e \mid \texttt{tbfun}\, p \,\texttt{->}\, e \mid e \diamond e \mid e \boxdot e & \\
p & ::= & x \mid \texttt{()} \mid \texttt{int} \mid \texttt{str} \mid l\ p \mid p \,\texttt{*}\, p & \textit{patterns} \\
\boxdot & ::= & \textit{(binary operators: equality, addition, etc.)} & \textit{operators} \\
l & ::= & \text{`}\,\textit{(alphanumeric)} & \textit{labels} \\
x & & & \textit{variables}
\end{array}$$

Figure 8.1: LittleBang Basic Syntax

later sections may translate either to syntax appearing in Figure 8.1 or to extensions which have been defined in previous sections. Thus, a full LittleBang program may be translated to nested TinyBang by applying the encodings in the opposite order in which they appear in this chapter.

To describe an encoding, we use "banana brackets" parameterized with the grammar non-terminal that they desugar; for instance, we may write $(\!|\texttt{foo}|\!)_e = \texttt{bar}$ to indicate that the form `foo` is encoded as the form `bar` in the context of an expression (e). The use of the grammar non-terminal here does not indicate e.g. a particular expression but rather expressions in general.

As encodings are often required to use fresh variable names, we use the notation $\overset{\star}{x}$ to denote a fresh variable. When the same fresh variable symbol appears in the same encoding (with the same subscript, if appropriate), it is the same variable; for instance, $(\!|\texttt{foo}\ x_1|\!)_e = \texttt{bar}\ \overset{\star}{x}_2\ \overset{\star}{x}_2\ \overset{\star}{x}_1\ x$ indicates that the code `foo x` may desugar to e.g. `bar z z y x`. We use $\overset{\star}{l}$ as a similar notation for label names.

We give each of our encodings below. What follows is a laundry list of encodings presented, when possible, in order from simplest to most complex.

8.2 Booleans

We begin our encodings with a simple example: boolean values. Rather than requiring users to write `` `True () ``, we expect to write (as in most languages) `true`. We formally define that behavior as follows:

Encoding 8.1. We extend the grammar of LittleBang below:

$$e \ ::= \ \ldots \mid \texttt{true} \mid \texttt{false}$$

This grammar extension is encoded as follows:

$$(\!|\texttt{true}|\!)_e = \texttt{`True ()}$$

$$(\!|\texttt{false}|\!)_e = \texttt{`False ()}$$

8.2.1 Boolean Operations

For the sake of completeness, we can also define some boolean operations. These do not need to be addressed using the built-ins mechanism described in Chapter 7 because the operations are defined over a finite space determined entirely by value types.

Encoding 8.2. We extend the grammar of LittleBang below:

$$e \quad ::= \quad \ldots \mid e\,\texttt{and}\,e \mid e\,\texttt{or}\,e \mid \texttt{not}\,e$$

This grammar extension is encoded as follows:

```
(|e1 and e2|)e  =  ( (`1 `True ()  * `2 `True ()  -> `True ())
                   & (`1 `False () * `2 `True ()  -> `False ())
                   & (`2 `True ()  * `2 `False () -> `False ())
                   & (`2 `False () * `2 `False () -> `False ())
                   ) (`1 (e1) & `2 (e2))
(|e1 or e2|)e   =  ( (`1 `True ()  * `2 `True ()  -> `True ())
                   & (`1 `False () * `2 `True ()  -> `True ())
                   & (`2 `True ()  * `2 `False () -> `True ())
                   & (`2 `False () * `2 `False () -> `False ())
                   ) (`1 (e1) & `2 (e2))
(|not e|)e      =  ( (`True () -> `False ())
                   & (`False () -> `True ())
                   ) (e)
```

In practice, of course, a compiler would not perform function application for every boolean operation; dispatch tables such as those above can be recognized and transformed into the appropriate machine instructions. While optimization is outside of the scope of this document, the TinyBang type system was designed with eventual optimization in mind. The above is simply presented as a formal encoding such as is used in the proof-of-concept interpreter implementation.

The above encoding works together with the type system to perform some amount of partial evaluation on boolean conditions. Recall from Chapter 2 that built-in operations like `==` are expected to return (using Notation 3.27) values of type `` `True () `` $\cup$

`‘False ()`. Thus, the expression `1 == 2` will return a general boolean type. The expression `1 == 2 or true`, however, will *always* yield a value of type `‘True ()`: no recursion appears at this call site and the only arguments passed to the function in the `or` encoding lead to branches which yield a `‘True ()` value. Therefore, the LittleBang expression `if (1 == 2 or true) then 3 else ’z’` is inferred to have type `int`.

8.3 Records

We continue by encoding ML-style records: a data structure example which includes a pattern form. As indicated in Section 1.3, records are trivially encoded in TinyBang using labels and onions. We formally define that behavior here:

Encoding 8.3. We extend the grammar of LittleBang below:

$$\begin{array}{rcll} z & ::= & \textit{(alphanumeric)} & \textit{names} \\ e & ::= & \ldots \mid \texttt{\{}\, z_1 \texttt{=} e_1 \texttt{,} \ldots \texttt{,} z_n \texttt{=} e_n \,\texttt{\}} & \end{array}$$

This grammar extension is encoded as follows:

$$(\!| \texttt{\{}\, z_1 \texttt{=} e_1 \texttt{,} \ldots \texttt{,} z_n \texttt{=} e_n \,\texttt{\}} |\!)_e = (\texttt{‘}\, z_1\ e_1) \,\texttt{\&} \ldots \texttt{\&}\, (\texttt{‘}\, z_n\ e_n)$$

Thus, the expression `{a=5,b=4,c=3}` is encoded as `(‘a 5) & (‘b 4) & (‘c 3)`. This encoding is quite simple, but it serves to illustrate the process we use in defining encodings for LittleBang. Of course, it is convenient to have a pattern form for matching records as well; we define one here:

Encoding 8.4. We extend the grammar of LittleBang below:

$$p \ ::= \ \ldots \mid \texttt{\{}\, z_1 \texttt{=} p_1 \texttt{,} \ldots \texttt{,} z_n \texttt{=} p_n \,\texttt{\}}$$

This grammar extension is encoded as follows:

$$(\!| \texttt{\{}\, z_1 \texttt{=} p_1 \texttt{,} \ldots \texttt{,} z_n \texttt{=} p_n \,\texttt{\}} |\!)_p = (\texttt{‘}\, z_1\ p_1) \,\texttt{*} \ldots \texttt{*}\, (\texttt{‘}\, z_n\ p_n)$$

8.3.1 Projection

For convenience, we also define syntax to project a record component. This syntax is defined in terms of the record patterns shown above:

Encoding 8.5. We extend the grammar of LittleBang below:

$$e \ ::= \ \ldots \mid e \,\texttt{.}\, z$$

This grammar extension is encoded as follows:

$$(\!| e \,\texttt{.}\, z |\!)_e = \texttt{(tbfun \{} z \,\texttt{=}\, \overset{\star}{x} \texttt{\} ->} \ \overset{\star}{x} \,\texttt{)} \diamond e$$

This function simply matches on the 1-ary record with a value under the projected label and returns its contents. This demonstrates the manner in which we define encodings in terms of other encodings.

8.4 Lists

Standard cons-cell lists are trivial to encode in TinyBang. We give that encoding here for completeness and because LittleBang lists have interesting typing properties. We will use the typical syntax: brackets for literal lists and `::` for list construction.

Encoding 8.6. We extend the grammar of LittleBang below:

$$\begin{array}{rcl} e & ::= & \ldots \mid \texttt{[}\, e \,\texttt{,}\, \ldots \,\texttt{,}\, e \,\texttt{]} \mid e \,\texttt{::}\, e \\ p & ::= & \ldots \mid \texttt{[}\, p \,\texttt{,}\, \ldots \,\texttt{,}\, p \,\texttt{]} \mid p \,\texttt{::}\, p \end{array}$$

This grammar extension is encoded as follows:

$$\begin{array}{lcl} (\!| e_1 \,\texttt{::}\, e_2 |\!)_e & = & \texttt{`Hd}\ e_1 \,\texttt{\&}\, \texttt{`Tl}\ e_2 \\ (\!| \texttt{[]} |\!)_e & = & \texttt{`Nil ()} \\ (\!| \texttt{[}\, e_1 \,\texttt{,}\, \ldots \,\texttt{,}\, e_n \,\texttt{]} |\!)_e & = & \texttt{(}\, e_1 \,\texttt{)} \,\texttt{::}\, \texttt{(}\, \texttt{[}\, e_2 \,\texttt{,}\, \ldots \,\texttt{,}\, e_n \,\texttt{]} \,\texttt{)} \\ (\!| p_1 \,\texttt{::}\, p_2 |\!)_p & = & \texttt{`Hd}\ p_1 \,\texttt{\&}\, \texttt{`Tl}\ p_2 \\ (\!| \texttt{[]} |\!)_p & = & \texttt{`Nil ()} \\ (\!| \texttt{[}\, p_1 \,\texttt{,}\, \ldots \,\texttt{,}\, p_n \,\texttt{]} |\!)_p & = & \texttt{(}\, p_1 \,\texttt{)} \,\texttt{::}\, \texttt{(}\, \texttt{[}\, p_2 \,\texttt{,}\, \ldots \,\texttt{,}\, p_n \,\texttt{]} \,\texttt{)} \end{array}$$

8.4.1 Lists Are Tuples

An interesting consequence of the TinyBang type system is that no encoding is necessary for tuples. Recall from Section 3.2 that each subexpression in nested TinyBang is given a unique variable in A-normal form. Recall also that Section 6.2 uniquely polyinstantiates any variables that are not immediately subject to recursion. As a result, literal lists have heterogeneous types (while iteratively-constructed lists have homogeneous types). For instance, consider the following LittleBang code:

```
let f0 = (tbfun self ->
          tbfun `1 (n * int) * `2 v1 * `3 v2 ->
            let hd = (if n > 3 then v1 else v2) in
            if n == 0 then [] else hd::(self (`1 (n-1) & `2 v))
5          ) in
6 let f = f0 f0 in
7 f (`1 8 & `2 'z' & `3 0)
```

The above code creates a function `f` which recursively generates a list of some size; all elements in the list are the value given in the `` `2 `` label excepting the last three, which are the value given in the `` `3 `` label. Here, for instance, the list which is generated is `['z','z','z','z','z',0,0,0]`. Because the function recurses on itself at the call site on line 4, the contours dictating the types of each element in the list are merged (to prevent unbounded growth) and so the list has the type $\alpha\backslash\{\texttt{`Nil}\ \texttt{()} <: \alpha, \texttt{`Hd}\,(\texttt{int}\cup\texttt{char})\ \texttt{\&}\ \texttt{`Tl}\ \alpha <: \alpha\}$ (that is, each element in the list is either a `char` or an `int`). This should be relatively unsurprising and is the type commonly inferred by systems with subtyping.

On the other hand, the CR polymorphism model is quite expressive when recursion is absent. Consider the following LittleBang code, which is a literal unrolling of the above:

```
['z','z','z','z','z',0,0,0]
```

Here, each element of the list is named explicitly at a different point in the program; the A-translation of this code gives a distinct variable name to each value in the list. As a result, the list has type $[\texttt{char}, \texttt{char}, \texttt{char}, \texttt{char}, \texttt{char}, \texttt{int}, \texttt{int}, \texttt{int}]$; that is, the type system knows that the first element in this list is a character while the last element in this list is an integer. This is exactly the precision (and complexity) demonstrated by tuples in other languages and, as a result, we need not provide a tuple encoding in LittleBang; we use literal lists instead.

8.5 Functions and Application

As indicated in Section 2.4, scripting languages typically use C-like parameter list function syntax (as opposed to the curried application form used in functional programming languages). We define functions and their application in LittleBang to conform to the former. Additionally, we give a combined encoding here that can handle both named and default arguments.

We begin by giving the following definition for the invocation of functions. We start with invocations because they are simpler and give an idea of how we expect the arguments to be handled:

Encoding 8.7. We extend the grammar of LittleBang below:

$$\begin{array}{rcll} a & ::= & e \mid z \,\texttt{=}\, e & \textit{arguments} \\ e & ::= & \ldots \mid e\,\texttt{(}\,a\,\texttt{,}\,\ldots\,\texttt{,}\,a\,\texttt{)} & \end{array}$$

This grammar extension is encoded as follows:

$$(\!|e\ \texttt{(}\ a_1\ \texttt{,}\ \ldots\ \texttt{,}\ a_n\ \texttt{)}|\!)_e = e \diamond (\textsc{ArgDec}(1, a_1)\ \texttt{\&} \ldots \texttt{\&}\ \textsc{ArgDec}(n, a_n))$$

$$\begin{array}{rcl} \text{where } \textsc{ArgDec}(i, z\,\texttt{=}\,e) & = & \texttt{`}\,z\ e \\ \textsc{ArgDec}(i, e) & = & \texttt{`}\,i\ e \end{array}$$

Quite simply, application wraps each argument in a label and onions those labeled arguments together. If the argument is named, the label matches its name; otherwise, the label matches the index of its position. For this application to work correctly, our encoding of LittleBang functions must be prepared to accept either the named or unnamed form of each argument; it must also be prepared to handle missing arguments if a default has been defined for the parameter in question. We approach this by defining the function in four stages: a series of argument extraction functions, a function representing the actual body to execute, a function which applies default values, and a function which organizes the arguments into a normalized, nominal form.

Encoding 8.8. We extend the LittleBang grammar as follows:

$$\begin{array}{rcll} q & ::= & x \mid x : p \mid x\,\texttt{=}\,e \mid x : p\,\texttt{=}\,e & \textit{parameters} \\ e & ::= & \ldots \mid \texttt{fun}\ \texttt{(}\ q\ \texttt{,}\ \ldots\ \texttt{,}\ q\ \texttt{)}\ \texttt{->}\ e & \end{array}$$

We restrict the above extension such that, in a function's parameter list, no parameter which includes default expressions (that is, of the form of the last two productions) may appear before a parameter without a default expression.

We then encode this grammar extension as depicted in Figure 8.2.

8.5.1 Examples

Although the mathematical notation for this encoding is somewhat opaque, the intuition is fairly straightforward and we illustrate it by example. Consider a LittleBang function declaration `let f = fun (x,y,z=4) -> x+y-z in ...`. Here, we have three parameters, two of which are required; thus, $n = 3$ and $k = 2$. This desugars to the following (using variables prefixed with `a` as fresh variables):

```
let a1'' = ( (tbfun `3 a0'' -> `z a0'')
           & (tbfun a0'' -> () ) ) in
let a1 = tbfun `x x * `y y * `z z -> x+y-z in
let a2 = tbfun a0'' -> a1 ◇ (a0'' & a1'' ◇ a0'' & `z 4) in
( (a0 * `1 (a1' * ()) * `2 (a2' * ()) -> a2 ◇ (`x a1' & `y a2' & a0') )
& (a0 * `1 (a1' * ()) * `y (a2' * ()) -> a2 ◇ (`x a1' & a0') )
& (a0 * `x (a1' * ()) * `y (a2' * ()) -> a2 ◇ a0' )
```

$$
\begin{array}{lcl}
(\!|x|\!)_q & = & x\,\texttt{:}\,\texttt{()} \\
(\!|x\,\texttt{=}\,e|\!)_q & = & x\,\texttt{:}\,\texttt{()}\,\texttt{=}\,e \\
(\!|\texttt{fun}\,\texttt{(}\,q_1\,\texttt{,}\,\ldots\,\texttt{,}\,q_n\,\texttt{)}\,\texttt{->}\,e|\!)_e & = &
\end{array}
$$

$$
\begin{array}{l}
\qquad\texttt{let}\ \overset{\star}{x}''_{k+1}\,\texttt{=}\ \texttt{(}\ \texttt{(}\ \texttt{tbfun}\ \texttt{`}\,(k+1)\ \overset{\star}{x}''_0\,\texttt{->}\,\texttt{`}\,x_{k+1}\ \overset{\star}{x}''_0\ \texttt{)} \\
\qquad\qquad\qquad\texttt{\&}\ \texttt{(}\ \texttt{tbfun}\ \overset{\star}{x}''_0\,\texttt{->}\,\texttt{()}\ \texttt{)}\ \texttt{)} \\
\qquad\texttt{in} \\
\qquad\vdots \\
\qquad\texttt{let}\ \overset{\star}{x}''_{n}\,\texttt{=}\ \texttt{(}\ \texttt{(}\ \texttt{tbfun}\ \texttt{`}\,(n)\ \overset{\star}{x}''_0\,\texttt{->}\,\texttt{`}\,x_{n}\ \overset{\star}{x}''_0\ \texttt{)} \\
\qquad\qquad\quad\texttt{\&}\ \texttt{(}\ \texttt{tbfun}\ \overset{\star}{x}''_0\,\texttt{->}\,\texttt{()}\ \texttt{)}\ \texttt{)} \\
\qquad\texttt{in} \\
\qquad\texttt{let}\ \overset{\star}{x}_1\ \texttt{=}\ \texttt{tbfun}\ \texttt{`}\,x_1\,(x_1)\,\texttt{*}\ldots\texttt{*}\,\texttt{`}\,x_n\,(x_n)\,\texttt{->}\,e\ \texttt{in} \\
\qquad\texttt{let}\ \overset{\star}{x}_2\ \texttt{=}\ \texttt{tbfun}\ \overset{\star}{x}''_0\,\texttt{->}\,\overset{\star}{x}_1\,\diamond \\
\qquad\qquad\qquad\qquad(\overset{\star}{x}''_0\,\texttt{\&}\,\overset{\star}{x}''_{k+1}\,\diamond\,\overset{\star}{x}''_0\,\texttt{\&}\ldots\texttt{\&}\,\overset{\star}{x}''_{k+1}\,\diamond\,\overset{\star}{x}''_0\,\texttt{\&} \\
\qquad\qquad\qquad\qquad\ \texttt{`}\,x_{k+1}\ e'_{k+1}\,\texttt{\&}\ldots\texttt{\&}\,\texttt{`}\,x_n\ e'_n)\ \texttt{in} \\
\qquad\texttt{(}\ \texttt{(}\ \overset{\star}{x}'_0\,\texttt{*}\,\texttt{`}\,\texttt{1}\ (\overset{\star}{x}'_1\,\texttt{*}\,p_1)\,\texttt{*}\ldots\texttt{*}\,\texttt{`}\,k\ (\overset{\star}{x}'_k\,\texttt{*}\,p_k)\,\texttt{->} \\
\qquad\qquad\overset{\star}{x}_2\,\diamond\,(\texttt{`}\,x_1\ \overset{\star}{x}'_1\,\texttt{\&}\ldots\texttt{\&}\,\texttt{`}\,x_k\ \overset{\star}{x}'_k\,\texttt{\&}\,\overset{\star}{x}'_0)\,\texttt{)} \\
\qquad\texttt{\&}\ \texttt{(}\ \overset{\star}{x}'_0\,\texttt{*}\,\texttt{`}\,\texttt{1}\ (\overset{\star}{x}'_1\,\texttt{*}\,p_1)\,\texttt{*}\ldots\texttt{*}\,\texttt{`}\,(k-1)\ (\overset{\star}{x}'_{k-1}\,\texttt{*}\,p_{k-1})\,\texttt{*}\,\texttt{`}\,x_k\ (\overset{\star}{x}'_k\,\texttt{*}\,p_k)\,\texttt{->} \\
\qquad\qquad\overset{\star}{x}_2\,\diamond\,(\texttt{`}\,x_1\ \overset{\star}{x}'_1\,\texttt{\&}\ldots\texttt{\&}\,\texttt{`}\,x_{k-1}\ \overset{\star}{x}'_{k-1}\,\texttt{\&}\,\overset{\star}{x}'_0)\,\texttt{)} \\
\qquad\texttt{\&}\qquad\qquad\vdots \\
\qquad\texttt{\&}\ \texttt{(}\ \overset{\star}{x}'_0\,\texttt{*}\,\texttt{`}\,x_1\ (\overset{\star}{x}'_1\,\texttt{*}\,p_1)\,\texttt{*}\ldots\texttt{*}\,\texttt{`}\,x_1\ (\overset{\star}{x}'_k\,\texttt{*}\,p_k)\,\texttt{->} \\
\qquad\qquad\overset{\star}{x}_2\,\diamond\,(\overset{\star}{x}'_0)\,\texttt{)} \\
\qquad\texttt{)}
\end{array}
$$

$$
\begin{array}{lcl}
\text{where }\textsc{VarOf}(x:p) & = & x \\
\textsc{VarOf}(x:p\,\texttt{=}\,e) & = & x \\
\textsc{PatOf}(x:p) & = & p \\
\textsc{PatOf}(x:p\,\texttt{=}\,e) & = & p \\
\textsc{DfltOf}(x:p\,\texttt{=}\,e) & = & e \\
x_i & = & \textsc{VarOf}(q_i) \\
p_i & = & \textsc{PatOf}(q_i) \\
e'_i & = & \textsc{DfltOf}(q_i) \\
1 \le k \le n & = & \text{the index of the last non-default parameter}
\end{array}
$$

Figure 8.2: Function Definition Encoding

```
8 )
```

If we invoke this function as `f(2,3,5)`, then the onion `(‘1 2 & ‘2 3 & ‘3 5)` is passed to the argument normalization function at line 5. This argument matches the first simple function, so the contents of the `‘1` and `‘2` labels are copied onto the left of the argument record as `‘x` and `‘y` labels; this organizes the *required* arguments into a normalized nominal form. That record is passed to the function `a2`, which adds two elements to the right of it before passing it to `a1`: the expression `tbapply a1’’ a0’’`, which similarly copies the *optional* argument `‘3 5` into a label `‘z 5` and the defaulting expression `‘z 4` which, since there is a `‘z 5` to the left of it, has no effect. Thus, the record `‘x 2 & ‘y 3 & ‘1 2 & ‘2 3 & ‘3 5 & ‘z 5 & ‘z 4` is passed to the function `a1`, which extracts `2`, `3`, and `5` as `x`, `y`, and `z` (respectively) and correctly returns `0`.

The above illustrates positional arguments and the handling of present optional arguments. This next example shows how positional and nominal arguments are combined and how absent optional arguments are handled: consider the invocation `f(2,y=3)`. The onion `(‘1 2 & ‘y 3)` is passed to the normalization function at line 5. Because there is no `‘2` label, the argument fails to match the first simple function but succeeds in matching the second; this causes `‘x 2 & ‘1 2 & ‘y 3` to be passed to `a2`. Here the same two expressions are added to the right of the record. This time, the expression `tbapply a1’’ a0’’` yields an empty onion `()` because no `‘3` label is present in the argument; nonetheless, `‘z 4` is added to the right, leaving `‘x 2 & ‘1 2 & ‘y 3 & () & ‘z 4` to be passed to `a1`. This way, `a1` binds `x`, `y`, and `z` to `2`, `3`, and `4` (respectively).

8.5.2 Performance

The above pattern-matching process is fairly involved and requires numerous TinyBang function invocations at each LittleBang call site. Although a naïve implementation of the above would suffer greatly in performance, this is not a property fundamental to the encoding. The BigBang research group is, concurrent to this work, developing a layout calculus which reduces the effort involved at the dispatch of each function call to a single table lookup (based on tags attributed to onions as they are constructed, which is similarly cheap); thus, each function call here costs at most a table lookup.

In cases where the static layout of the arguments is known (e.g. a typical call such as `f(2,3,5)` above), we expect that this, combined with generous inlining by the compiler, will result in approximately the same overhead as argument marshalling in traditional languages such as C. The above encoding does open the possibility of fundamentally dynamic dispatches which cannot be statically inlined, such as when the argument record is union typed: e.g.

```
let f = ... in
let b = ... in
let x = `1 2 & (if b then `2 3 else `2 'z') in
f ◇ x
```

Here, we assume `b` to be a runtime-determined boolean and `f` to be a function of two arguments which is overloaded such that the second argument may be either an integer or a character. Because it is not known at compile time the specific overloading to which the call will dispatch, inlining cannot prevent the table lookup in this case. Similarly, inlining is prevented when the function type itself is a higher-order argument. With the optimizations above, we expect this performance overhead to be comparable to current compiled languages. However, the specifics of this optimization process are well outside the scope of this document.

8.5.3 Extra Arguments

The above encoding admits spurious named and positional arguments at call sites. For instance, invoking the example function above as `f(2,3,4,5)` has an identical effect to `(2,3,4)`; in the encoding, the last argument is encoded as `` `4 5`` and, since none of the patterns match that argument, it is ignored. This is undesirable and can easily lead to user error.

Addressing this problem in the presence of positional arguments is quite easy. At line 5, for instance, we can left-onion another simple function of the form `` tbfun `4 () -> () ◇ () `` onto the normalization function. If a fourth argument is present, this will attempt to evaluate the application of the empty onion, which is a type error. Because TinyBang's application is typechecked using a path-sensitive model – function bodies which are not called are never expanded – this will not produce a type error unless an invocation with at least four arguments appears. We can use a similar approach for arguments which are named in duplicate (e.g. `f(2,3,4,z=5)`, where `` `3 4`` and `` `z 5`` refer to the same argument).

Addressing spurious nominal arguments (e.g. `f(2,3,q=0)`) is somewhat more difficult as it is not possible to enumerate all labels which should not appear at a call site. In a sense, this is fundamental to the structurally subtyped nature of onions: pattern matching will always match a *subtype* and the addition of an absent label to a record produces a structural subtype. Spurious arguments are, however, one example of a case in which structural subtyping is not desirable. Although rather involved encodings exist which can generate type errors for spurious nominal arguments, they are relatively inefficient; instead, this problem is best addressed by augmenting the pattern matching process with a form of pattern which fails when labels outside of a given set are present. Because this pattern operates contrary to the subtyping properties of other patterns, it requires special handling when matching onions. Nonetheless, the existence of such a pattern is not unsound.

8.6 Objects

A central motivator in much of the development of TinyBang was a successful encoding of objects. Object-orientation is featured prominently in modern scripting languages and we desire a typed encoding which can handle the flexible object operations they use. Our encoding of objects is relatively simple (in comparison to the previous encoding, for example) and was covered in large part in Section 2.6. We present here a formal encoding that utilizes the function dispatch mechanisms from above:

Encoding 8.9. We extend the grammar of LittleBang below:

$$\begin{array}{lcll} d & ::= & z \,\texttt{=}\, e \mid z\,\texttt{(}\,q\,\texttt{,}\,\ldots\,\texttt{,}\,q\,\texttt{)}\,\texttt{=}\,e & \textit{member declarations} \\ e & ::= & \ldots \mid \texttt{object}\,\texttt{\{}\,d\ldots d\,\texttt{\}} \mid e\,\texttt{.}\,z\,\texttt{(}\,a\,\texttt{,}\,\ldots\,\texttt{,}\,a\,\texttt{)} & \end{array}$$

This grammar extension is encoded as follows:

$$\begin{array}{lcl} (\!| z \,\texttt{=}\, e |\!)_d & = & \texttt{`}\,z\;e \\ (\!| z\,\texttt{(}\,q_1\,\texttt{,}\,\ldots\,\texttt{,}\,q_n\,\texttt{)}\,\texttt{=}\,e |\!)_d & = & \texttt{fun (message : `}\,z\;\texttt{() , self ,}\;q_1\,\texttt{,}\,\ldots\,\texttt{,}\,q_n\,\texttt{) ->}\,e \end{array}$$

$(\!| \texttt{object}\,\texttt{\{}\,d_1\ldots d_n\,\texttt{\}} |\!)_e \;=$

```
let fixpoint =
      tbfun f -> (tbfun g -> tbfun x -> g ◇ g ◇ x)
                 (tbfun h -> tbfun y ->
                     f ◇ (h ◇ h) ◇ y) in
let seal = fixpoint ◇ (tbfun seal -> tbfun obj ->
        (tbfun msg -> obj ◇ (msg & `self (seal ◇ obj)))
        & obj) in
```

$\texttt{seal} \diamond ((\!| d_1 |\!)_d \,\texttt{\&}\ldots\texttt{\&}\, (\!| d_n |\!)_d)$

$$(\!| e\,\texttt{.}\,z\,\texttt{(}\,a_1\,\texttt{,}\,\ldots\,\texttt{,}\,a_n\,\texttt{)} |\!)_e \;=\; \texttt{(}\,e\,\texttt{)}\;\texttt{(`}\,z\;\texttt{() ,}\;a_1\,\texttt{,}\,\ldots\,\texttt{,}\,a_n\,\texttt{)}$$

The above presentation of the `seal` function is the same as in Section 2.6.2; it has been rewritten in the LittleBang grammar given in this section. We can freely intersperse the uses of TinyBang and LittleBang function definitions and application here because the latter desugars to the former. Recall that the purpose of `seal` is to capture the object in a positive position and then add that (recursively sealed) object to every subsequent method invocation as an implicit argument `` `self``. Due to the argument encoding we have selected, this is the same as adding a `self=`... argument in a LittleBang function invocation. As a result, the encoding of methods may require a `self` argument that the encoding of method invocations does not provide (so long as the object is sealed before that method is called, as it is by the `object` encoding).

The above outlines the behavior of `seal`. We discuss the typing of this function in Section 8.6.2 with the assistance of the mixins we present next.

8.6.1 Mixins

As discussed in Section 2.6.4, mixins are trivial to define in this model. Because an object is merely an onion, a mixin is simply an onion which can be concatenated with an object before it is sealed. We formalize an encoding of mixins here. The `mixin` expression defines a mixin; the `mixing` member declaration form incorporates a mixin into an object as it is defined.

Encoding 8.10. We extend the grammar of LittleBang below:

$$\begin{array}{rcl} d & ::= & \ldots \mid \texttt{mixing}\, e \\ e & ::= & \ldots \mid \texttt{mixin}\,\texttt{\{}\, d \ldots d\, \texttt{\}} \end{array}$$

This grammar extension is encoded as follows:

$$\begin{array}{lcl} (\!|\texttt{mixing}\, e|\!)_d & = & e \diamond \texttt{()} \\ (\!|\texttt{mixin}\,\texttt{\{}\, d_1 \ldots d_n\, \texttt{\}}|\!)_e & = & \texttt{tbfun () ->}\ (\!|d_1|\!)_d\, \texttt{\&} \ldots \texttt{\&}\, (\!|d_n|\!)_d \end{array}$$

Note that this encoding is exceedingly simple. The encoding of the `mixin` expression is the same as an object expression except that (1) it is not sealed and (2) it is placed behind a lazy evaluation barrier. The `mixing` member declaration is even simpler: it just results in the mixin expression itself (pulling on the function to yield the members). In a sense, the mixin serves as an alias for a collection of object members. The use of laziness here ensures that, if a mixin declares fields, those fields are created anew for each use of the mixin.

8.6.2 Typing Seal

The typing of the above encoding is somewhat subtle. From Chapter 4, we can glean that this encoding is sound, but how are we to be sure that it generates usable object types? The `fixpoint` function from the above is simply the Y-combinator, the type for which can be inferred in common subtype constraint systems. The typing of `seal` is somewhat more delicate. As mentioned in Section 2, the key is that we capture the type of `‘self` in a closure rather than passing it as an argument at method invocations. For `seal` to be effective, it must capture a different type for each object that is sealed. This is where the TinyBang polymorphism model defined in Chapter 6 comes into play. Consider the following LittleBang code:

```
let obj1 = object {
             x = ref 5
             f() = self.x := !self.x + 1
           } in
let obj2 = object {
             x = `A ()
             f() = self.x
           } in
`First (obj1.f()) & `Second (obj2.f())
```

The above code will typecheck and yield a value of type `` `First int & `Second `A () ``. The `` `self ``s for these two objects are captured separately: there is one invocation of `seal` for `obj1` and another for `obj2`. TinyBang assigns the contents of each of these `` `self `` labels a distinct type variable, so there is no confusion between them. Even if the encoding of `object` were to reuse the definition of `seal` (which it would do in practice), the invocations of `seal` are distinct syntax points in the desugared program and the CR polymorphism model ensures that they are polyinstantiated separately.

This also illustrates the point at which the type of `seal` becomes confused. Consider instead the following TinyBang code:

```
let mixins = [ mixin { f() = self.x }
             , mixin { g() = self.x }
             , mixin { h() = self.x }
             ] in
let fixpoint = ... in
let loop = fixpoint (tbfun self ->
              (tbfun [] -> [])
            & (tbfun (m:ms) ->
                let obj =
                  object {
                    x = 5
                    mixing m
                  } in
                obj:(self ms)
              )) in
loop mixins
```

We assume that `fixpoint` above is the Y-combinator as in the object encoding. The `loop` function above encodes a simple map operation over a list of mixins, using each in the definition of an object. The expression `loop mixins` produces a list of (two) objects. In this case, the TinyBang type system *cannot* determine that the three objects have an `f`, `g`, and `h` method (respectively). This is because the recursion on line 14 causes contour folding: each application handled at that call site uses the same type variable. As a result, it is impossible to tell the difference between e.g. the `g`-method object and the `h`-method object at the type level. In general, type precision is lost whenever the type of the object being created varies based on loops or other recursive structures.

Note that this does not prevent the creation of objects in loops; it only means that, when two objects are created at the same point in different loop iterations, the type ascribed to them is at least the least upper bound of the two. Note also that this weakness only applies when the object construction site is the same. Using Python as an example, TinyBang would be unable to typecheck the equivalent of

```python
class Foo:
  def __init__(self, x): self.y = x
class Bar:
  def __init__(self, x): self.z = x
for b in [True,True,False]:
  obj = Foo(1) if b else Bar(2)
  if b: obj.y
  else: obj.z
```

The above would not typecheck because the type of `obj` hinges upon the value of `b`; TinyBang's type system does not include such general dependent types. However, the equivalent of the following Python code would typecheck correctly:

```python
class Foo:
  def __init__(self, x): self.y = x
  def set(self, z): self.y = z
class Bar:
  def __init__(self, x): self.y = x
  def inc(self): self.y += 1
for b in [True,True,False]:
  obj = Foo(1) if b else Bar(2)
  obj.y
```

Note in this example that the field of `Bar` is now `y`; thus, `obj.y` makes use of an interface common to both types of objects which appear here. It would also be possible to dispatch on the presence of a field in the object to determine how to react to it. In any case, we expect that this degree of type precision is sufficient to typecheck well-designed scripts;

rechecking a condition to determine the type of a value as in the previous example above is, generally speaking, poor form.

8.7 Classes

To develop LittleBang's encoding of classes, we take the perspective that they are merely objects which act as factories for other objects. We provide a notion of static members – members of the class object itself rather than the object it creates – and define construction in terms of the special name `new`. We give that encoding here:

Encoding 8.11. We extend the grammar of LittleBang below:

$$\begin{array}{rcll} \tilde{d} & ::= & \texttt{\char`~}\, d & \textit{static member declarations} \\ e & ::= & \ldots \mid \texttt{class}\,\texttt{(}\, q\,\texttt{,}\,\ldots\,\texttt{,}\, q\,\texttt{)}\ \texttt{\{}\, d \ldots d\ \tilde{d} \ldots \tilde{d}\,\texttt{\}} & \end{array}$$

This grammar extension is encoded as follows:

$$\begin{array}{l} (\!|\texttt{class}\,\texttt{(}\, q_1\,\texttt{,}\,\ldots\,\texttt{,}\, q_n\,\texttt{)}\ \texttt{\{}\, d_1 \ldots d_m\ \texttt{\char`~}\, d'_1 \ldots \texttt{\char`~}\, d'_k\,\texttt{\}}|\!)_e \quad = \\ \quad\quad \texttt{object}\,\texttt{\{} \\ \quad\quad\quad d'_1 \ldots d'_k \\ \quad\quad\quad \texttt{new}\,\texttt{(}\, q_1\,\texttt{,}\,\ldots\,\texttt{,}\, q_n\,\texttt{)} \texttt{ = let thisclass = self in} \\ \quad\quad\quad\quad\quad\quad\quad\quad \texttt{object}\,\texttt{\{}\, d_1 \ldots d_m \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \texttt{getclass}\,\texttt{()}\,\texttt{=}\,\texttt{thisclass}\,\texttt{\}} \\ \quad\quad \texttt{\}} \end{array}$$

The class decodes into an object whose members are the static members declared on the class. The object also has a static method, `new`, which creates an object containing all non-static members as well as a function `getclass` which retrieves the object representing the encoded class. Of course, the above grammatic restriction that static members appear after non-static members is merely for notational convenience; in practice, no specific declaration order is necessary.

8.7.1 Subclasses

A subclass is simply a class with a superset of the members of another class; this is particularly so because we are using a structural subtyping model, so the identity of the class is irrelevant. As a result, encoding subclasses is quite similar to encoding mixins: we merely need to directly incorporate the properties of the parent class. We simply add a syntactic position to the class declaration which identifies the parent class (as an expression) and then make use of that parent class in our encoding.

Encoding 8.12. We extend the grammar of LittleBang below:

$$e \ ::= \ \ldots \mid \texttt{class}\,\texttt{(}\,q\,\texttt{,}\,\ldots\,\texttt{,}\,q\,\texttt{)}\ \texttt{<}e\ \texttt{\{}\,d\ldots d\ \tilde{d}\ldots\tilde{d}\,\texttt{\}}$$

This grammar extension is encoded as follows:

$$\llparenthesis\texttt{class}\,\texttt{(}\,q_1\,\texttt{,}\,\ldots\,\texttt{,}\,q_n\,\texttt{)}\ \texttt{<}\ e\ \texttt{\{}\,d_1\ldots d_m\ \texttt{~}d'_1\ldots\texttt{~}d'_k\,\texttt{\}}\rrparenthesis_e \ =$$

```
let thissuperclass = $e$ in
object {
  $d'_1 \ldots d'_k$
  mixing thissuperclass
  new ( $q_1$ , ... , $q_n$ ) = let thisclass = self in
                     object { $d_1 \ldots d_m$
                          mixing ( thissuperclass . new (
                                 $\textsc{VarOf}(q_1)$ , ... , $\textsc{VarOf}(q_n)$ ) )
                          getclass ( ) = thisclass
                          getsuperclass ( ) = thissuperclass }
}
```

In the object representing the subclass, we add to our previous encoding a `mixin` declaration. This declaration adds the superclass object to the onion which we seal to produce the subclass. This is taking advantage of the object resealing property discussed in Section 2.6.2.1: as a result, the self-aware class will have access to all members of both the superclass (through the mixin) as well as the subclass. The same process occurs when an instance of the class is created: the superclass is instantiated and then treated as a mixin to the subclass, so the object will include the members of both when it is sealed.

Chapter 9

Implementation

This section briefly discusses the current proof-of-concept implementation of LittleBang, which includes a source translator, an interpreter, and a type checker and inference algorithm. The source translator is simple: it merely follows the encodings in the previous chapter to transform a LittleBang AST into a form which can be injected into nested TinyBang. The interpreter for TinyBang is similarly straightforward as an implementation of the deterministic operational semantics in Section 3.4. This chapter therefore focuses on the typechecker. The current TinyBang typechecker uses the type system formalized in Section 3.5 with extensions from Chapter 7 and the polymorphism model described in Chapter 6.

We presented the TinyBang type system from a formal mathematical perspective, but naïve implementation of the strategies described – especially those in Chapter 5, where decidability is proven – is completely infeasible. In Section 3.5.5, for instance, inconsistency is defined in terms of all sets which can be reached via constraint closure; inspecting all such sets is obviously impractical. Here, we discuss properties of the type system which can be used to implement a TinyBang typechecker with reasonable performance; we also discuss the current proof-of-concept typechecker, describe our reasoning for the optimizations we applied, and outline optimizations which have not yet been applied but which should yield considerable improvements over the current system.

9.1 Complete Closure is Sufficient

Typechecking as given in Definition 3.36 indicates that *all* constraint sets reachable by closure must be consistent. Testing these sets individually is, of course, impractical; like other constraint closure models, we aim to show that it is sufficient to complete the deductive closure and then to check the result. The argument for this property is not complex, but we articulate it below for completeness.

For a single, complete closure to be sufficient, we require two properties: that the constraint closure is confluent (all closure sequences converge to the same result) and that inconsistency is monotonic (adding constraints to a set never removes inconsistency). To begin this discussion, we present notation for describing the complete closure of a constraint set.

Notation 9.1 (Complete Closure)**.** We write $C \Longrightarrow^! C'$ to indicate that $C \Longrightarrow^* C'$ and that there exists no $C'' \neq C'$ such that $C' \Longrightarrow C''$.

9.1.1 Confluence of Closure

We begin by demonstrating that constraint closure is confluent. This is not true for all polymorphism models; strictly speaking, the merging function Υ may be defined to return any set of replacements for any set of variables. If we restrict the merging function to be monotonic in the set of variables provided, however, then the addition of constraints only increases the number of available replacements; that is, any merge that can be performed at a given point in closure can always be performed later. We define this monotonicity property as follows:

Definition 9.2 (Monotonic Merging)**.** A merging function Υ is *monotonic* iff $A_1 \subseteq A_2$ implies that $\Upsilon(A_1) \subseteq \Upsilon(A_2)$.

The monomorphic model's merging function is trivially monotonic. The merging function of the polymorphism model presented in Chapter 6, Υ_{CR}, is monotonic because the definition produces at most one replacement function per pair of type variables in its argument (and the replacements it creates considers no other type variables).

Assuming Υ to be monotonic, we can demonstrate the confluence of constraint closure. While this can be a complex property to demonstrate in some systems, it is quite simple in TinyBang:

Lemma 9.3. Constraint closure is confluent; that is, if $C_0 \Longrightarrow^! C_1$ and $C_0 \Longrightarrow^! C_2$ then $C_1 = C_2$.

Proof. By Newman's Lemma [47], it is sufficient to show that constraint closure converges (Lemma 5.38) and that it is locally confluent. By inspection, all constraint closure rules (and the relations upon which they rely) test only for the presence (and not the absence) of constraints. Further, constraint closure is monotonic: no constraint closure rule removes constraints. As a result, any two independent constraint closure steps can be interchanged and constraint closure is locally confluent. We are therefore finished. □

9.1.2 Monotonicity of Inconsistency

We likewise note that inconsistency is monotonic:

Lemma 9.4. Inconsistency of constraint sets is monotonic; that is, if C_1 is inconsistent and $C_2 \supseteq C_1$ then C_2 is inconsistent.

Proof. Again by observing that application matching and compatibility are monotonic in the constraint set. Each of these relations holds by the presence (and not the absence) of constraints; thus, any proof of inconsistency for C_1 holds (by induction on the height of the proof) for C_2. □

9.1.3 Proof of Sufficience of Complete Closure

We can therefore prove that, in practice, it suffices to complete constraint closure in any order and then check for inconsistency:

Lemma 9.5. An expression e typechecks iff $[\![e]\!]_\mathrm{E} = \alpha\backslash C$ and $C \Longrightarrow^! C'$ and C' is consistent.

Proof. We begin by considering the forward implication. If e typechecks, then by Definition 3.36 we have that $C \Longrightarrow^* C''$ implies that C'' is consistent. Because $C \Longrightarrow^! C'$ requires by definition that $C \Longrightarrow^* C'$, we know that C' is consistent.

The reverse implication is also straightforward: assuming C' is consistent, we must show that all $C \Longrightarrow^* C''$ imply that C'' is consistent. Suppose for contradiction that C'' is inconsistent. By Lemma 9.3, we have $C'' \Longrightarrow^* C'$. Because constraint closure is monotonic, we have that $C'' \subseteq C'$. By Lemma 9.4, we have that C' is inconsistent, which is a contradiction. We therefore have that there exists no $C \Longrightarrow^* C''$ such that C'' is inconsistent and we are finished. □

The above lemma demonstrates that, to determine if an expression typechecks, it is sufficient to compute the largest set from constraint closure and then determine the consistency of that set.

9.2 Variable Replacement

The TinyBang polymorphism interface described in Section 3.5.5.1 includes a merging function Υ, the purpose of which is to merge or unify type variables. The formal definition of this function actually produces a set of type variable replacements which may then be applied during closure to any constraint in the constraint set; thus, formally, this merging function has the effect of *increasing* the size of the constraint set rather than decreasing it. In practice, however, these replacements can be used to unify the type variables. We illustrate this by example using the polymorphism model CR (and the notation from Chapter 6).

For the purposes of this example, suppose the existence of two contours $\mathcal{C}_1$ and $\mathcal{C}_2$ which overlap. Let the least well-formed contour which covers both of them be $\mathcal{C}_3$. Intuitively, one can imagine $\mathcal{C}_1$ describing the case in which we recurse within a function f and then call a function g where we do not recurse. One can also imagine $\mathcal{C}_2$ describing the case in which we do not recurse within f but then call g and recurse there. Then $\mathcal{C}_3$ would describe any situation in which we do or do not recurse in those functions in that order.

Let $\alpha'_i = \langle \alpha_i, \mathcal{C}_1 \rangle$, let $\alpha''_i = \langle \alpha_i, \mathcal{C}_2 \rangle$, and let $\alpha'''_i = \langle \alpha_i, \mathcal{C}_3 \rangle$ for $1 \leq i \leq 3$ (presuming the existence of the three root type variables representing program positions). With this in mind, consider the constraint set $C = \{\texttt{()} <: \alpha'_1, \alpha''_1 <: \alpha''_2\}$ (recalling Notation 3.27 for legibility). This constraint set describes a value appearing at α_1 in the first contour; it also describes the value in α_1 moving to α_2 in the second contour. Since there is a case (e.g. we call f and then g but won't recurse either time) in which we could be in both contours, we'd expect to see the empty onion type reach the program point described by α_2 at least in that case.

Constraint closure begins by using the merging function. Given these variables, it would allow e.g. replacing α'_1 with α'''_1. Using that, we can learn $\texttt{()} <: \alpha'''_1$, but this does not provide additional options. Merging at this point also allows replacing α''_1 with α'''_1, which allows us to learn $\alpha'''_1 <: \alpha''_2$. Using these two previous facts, we can conclude $\texttt{()} <: \alpha''_2$. Further, merging allows us to replace α''_2 with α'''_2. This allows us to learn $\texttt{()} <: \alpha'''_2$. We note that this is a weaker statement than we might desire – it overapproximates the call stacks under which the empty onion can reach that point in the program – but to reach perfect precision here is a problem with necessarily exponential complexity.

Following the above reasoning, the complete transitive closure of $C = \{\texttt{()} <: \alpha'_1, \alpha''_1 <: \alpha''_2\}$ is:

$$\left\{\begin{array}{ll} () <: \alpha_1', & \alpha_1'' <: \alpha_2'', \\ () <: \alpha_1''', & \alpha_1''' <: \alpha_2'', \\ () <: \alpha_2'', & \alpha_1'' <: \alpha_2''', \\ () <: \alpha_2''', & \alpha_1''' <: \alpha_2''' \end{array}\right\}$$

We observe, however, that it would have been sufficient to *mutate* instances of e.g. α_1' with α_1''' rather than simply extending the constraint set with the new replacements. This is true for a few reasons. First, the overlap of regular expressions is monotonically increasing with union; for instance, if any contour $\mathcal{C}'$ overlaps with $\mathcal{C}_1$, it will also overlap with $\mathcal{C}_3$. Second, merging and commutes with other constraint closure sequences: there will be numerous situations in which we can learn a constraint either by performing replacement on other constraints and then closing on them or by closing on the original constraints and then performing replacement on the result. Because of these properties, any inconsistency which can be discovered on the pre-replacement variables (e.g. α_1') can be discovered on the post-replacement variables (α_1''') after replacement occurs. If we replace variables rather than extending the set, our completed closure above becomes $\{() <: \alpha_1''', \alpha_1''' <: \alpha_2''', () <: \alpha_2'''\}$; this is considerably more manageable.

One way of viewing this property is to use a non-standard model of the constraint set. We note that, in CR, each type variable is a pair between a program point (which is isomorphic to some unique variable x in the program) and a regular expression over such program points. Consider a meaning function $[\![c]\!]$ which produces the set of pairs between program points and *call strings* represented by a given constraint. For instance, $[\![\langle\alpha_0, \alpha_0'\rangle]\!]$ would quite simply be $\{\langle\alpha_0, \alpha_0'\rangle\}$ whereas $[\![\langle\alpha_0, \alpha_0'^*\rangle]\!]$ would be the infinite set $\{\langle\alpha_0, \epsilon\rangle, \langle\alpha_0, \alpha_0'\rangle, \langle\alpha_0, \alpha_0'\alpha_0'\rangle, \ldots\}$. In this non-standard model, the infinite set $[\![() <: \alpha_1']\!]$ is a subset of the infinite set $[\![() <: \alpha_1''']\!]$. As a result, $[\![\{() <: \alpha_1', () <: \alpha_1'''\}]\!] = [\![\{() <: \alpha_1'''\}]\!]$: the meaning of the set is the same with or without the constraint on α_1'. This meaning function aligns quite closely with the simulation relation in Chapter 4 in that it essentially describes the programs which are simulated by a given constraint, so we are intuitively claiming that any program simulated by the larger set can be simulated by the set without the unnecessary constraints.

The current implementation of the TinyBang type checker uses this reasoning to justify the replacement of type variables during closure. We discuss the role this replacement plays in the next section.

9.3 Current and Future Design

As of the time of this writing, the proof-of-concept TinyBang type checker uses the complete closure property as well as several standard design techniques (e.g. caching, indexed sets) to achieve reasonable performance. The implementation is written in Haskell and requires about 2,100 lines of code (including a custom NFA library used for contours but excluding the output of some simple compile-time metaprogramming facilities). Currently, performance leaves much to be desired: a collection of 30 unit tests on expressions of between 10 and 50 clauses takes approximately nine seconds to run on a typical developer's workstation. That said, the proof-of-concept type checker demonstrates the power of the type system. All of the examples from Chapter 2 typecheck; further, the typechecker successfully infers types for implementations of basic data structures [39] such as linked lists and hash tables, which are of size and complexity similar to that of scripting languages such as Python.

Presently, typechecking proceeds by computing the initial alignment of an expression, performing constraint closure until no more constraints can be learned, and then testing the set for consistency. The closure is computed naïvely by iterating a series of rules over the set, adding the conclusions of each until a full cycle is completed without finding additional constraints. This has the effect of re-executing numerous computations and is a prime candidate for improvement. Caching or a custom forward- and backward-chaining closure algorithm would considerably accelerate this process.

Presently, the proof-of-concept TinyBang type checker does not perform any merging; that is, rather than using $\Phi_{\text{CR}}/\Upsilon_{\text{CR}}$, it uses $\Phi_{\text{CR}}/\Upsilon_{\text{Mono}}$. (As merging was not required by the soundness proof, this is sound, but it leads to exponential growth of the constraint set in some cases.) Previous drafts of the typechecker have applied merging, but to little effect due to how merging operations interfere with naïve indexing structures and caches. This is not a theoretically challenging problem; it is merely a complex engineering task for which we have not had the time.

Chapter 10

Related Work

This chapter summarizes some of the work related to TinyBang and LittleBang. We begin with language features and properties and move on to more abstract discussion of related research topics.

10.1 Onions

TinyBang's record structure, which we call the *onion*, is unusual in that it supports asymmetric record concatenation. The bulk of work on typing record extension addresses symmetric concatenation only [57, 36]. From a theory perspective, symmetric concatenation initially sounds more elegant and appealing, but enforcing that concatenations are symmetric requires assertions on the *absence* of labels; this is contravariant to structural record subtyping and complicates the theory. Approaches using conditional constraints and row types [54] are similarly rather involved. Wand typed an asymmetric concatenation operation [73], but this work is in the context of row types and unification and so requires absence typing as well.

Also, as shown in Chapter 8, asymmetry is a valuable asset: some language features are fundamentally asymmetric and so rely on a notion of semantic asymmetry for succinct encodings. Asymmetry is not strictly necessary for such encodings; previous encodings [12] avoid this problem by reconstructing records rather than existing them, but this requires a fixed, static view of the record type and yields larger, more complex encodings. Although we have found onions to be naturally typable, we have found no subtype constraint-based typing of asymmetric concatenation in the literature. This is likely due in part to the difficulty of modularizing such types; although onions are easily typed in a whole-program context, it is more difficult to accurately type such operations when the full type of the onion is not known.

Onions are also unusual in that they are *type-indexed*; that is, values are labeled by their types rather than by an explicit name. This variation on typical record types first appeared in the λ^{TIR} calculus [61]. Like much of the record concatenation literature, that work also uses rows; TinyBang instead uses type constraints and a shallow binary concatenation operator.

10.2 Dispatch

In addition to concatenating simple data, TinyBang onions can also be used to create *compound functions* which support asymmetric dispatch. This dispatch is sensitive to the types of each of the arguments, much as in a `match` or `case` expression in the ML family of languages. The type of a compound function application is more precise than a `match` expression, however, as only the types of the case branches which can be reached are incorporated into the overall type of the application site; these are the heterogeneous case types mentioned in Section 2.6.1.2. This is consequence of structural subtyping and falls naturally out of our use of subtype constraints. Compound functions generalize the expressiveness of conditional constraints [4, 54] and are related to support for typed first-class messages *a la* [48, 54] (as first-class messages are just labeled data in our object encoding).

To statically type heterogeneous case types in TinyBang, we developed a novel solution to the well-known *union alignment problem*. The root of this problem is that there may not be a consistent view of which branch of a union type was taken in different pattern match clauses. All standard union type systems must make a compromise union elimination rule; [22] contains a review of existing approaches. Our solution is to lazily *filter* the union type with the pattern it matches; this union elimination technique loses none of the context the type system has accumulated and permits refined bindings to be available within the body of the match operation. This filtering approach can be viewed as a (locally) complete notion of union elimination.

Filtered types initially appear similar to *refinement types* [30], which often permit reducing a type from an existing system based on some conditional expression; for instance, the type `x:int | x>0` may express the type of all positive integers. Filtered types in TinyBang differ from refinement types in two significant ways. First, filtered types' predicates are always expressed as patterns (instead of general expressions), considerably limiting the expressiveness and complexity of the condition. Second, refinement types are designed to overlay onto existing, well-typed programs to prove properties about them, in essence forming a second layer of typing. TinyBang's filtered types are a fundamental part of the system; they interact with and influence types of the unfiltered form.

To solve the union alignment problem [22], we have structured the compatibility relation of TinyBang (Section 3.5.3) in such a way that each union appearing at a given point in a type is eliminated at most once. This is accomplished by using sets of patterns during compatibility (rather than checking each pattern for compatibility individually). An alternative (and equally viable) approach involves using selectors [65] to track the union elimination in a separate position of the relation. We choose to use sets of patterns because they interact well with filtered types.

Instantiation constraints [37, 28, 32] in previous work also solve the union alignment problem by limiting multiple polyinstantiations of a type to match each other in certain contexts. Instantiation constraints operate by mapping type variables to the concrete types at which they are instantiated. In TinyBang, such a mapping is insufficient given the extremely contextual precision we require for our domain: a recursive type in TinyBang may be matched against a finite unrolling of that type as a pattern and this information must be preserved. For instance, given the list type [`int` $\cup$ `char`], a match against the pattern x `*` [`int` , `char` , ...] must restrict x such that the first element is an `int` and the second element is a `char`. Because both points in the list type are represented by the same type variable, a type-variable-to-type mapping is insufficient for our purposes.

Typed multimethods [45] perform a dispatch similar to TinyBang's compound functions: in that system, the dispatched multimethod is dependent upon not only the dynamic type of the object but also the dynamic type of its arguments.[1] Typed multimethods and compound functions are differently expressive: typed multimethods only support nominal dispatch but are otherwise generally unconstrained whereas TinyBang compound functions use structural subtyping but only permit dispatch on first-order data (due to the limited expressiveness of the pattern grammar). A system with *function patterns* – patterns which structurally match functions which have a given type – is currently under development and, if successful, would subsume typed multimethods; that work, however, is an extension to TinyBang and beyond the scope of this document.

First-class cases [8] allow composition of case branches much in the fashion of onions of functions. In that work, however, general case concatenation requires case branches to be written in CPS and requires a phase distinction between case construction and case application. TinyBang compound functions may be applied or extended in any order and without any special handling by client code.

[1]This is in contrast to e.g. Java where, even with overloading, the dispatch of a method is based on the dynamic type of the object and the *static* type of the arguments.

10.3 Polymorphism

Polymorphism in TinyBang uses a call-site model (as opposed to e.g. the more traditional `let`-bound models of ML family languages): function types are polyinstantiated at their invocation sites rather than at the site where they are named. Although this distinction is unimportant in many cases, this distinction becomes relevant when functions are passed as arguments. For instance, consider the following LittleBang code:

```
1 let id = fun (x) -> x in
2 let f = fun (g,a,b) -> ‘Left (g a) & ‘Right (g b) in
3 f(id,4,’z’)
```

In a `let`-bound model, the function `id` is polyinstantiated on line 3; thus, `g` on line 2 has a monomorphic type. As a result, the expressions `g a` and `g b` must have the same type – in this case, $\texttt{int} \cup \texttt{char}$. Although this is not incorrect, it is imprecise; we prefer the type of the result should be `‘Left int & ‘Right char`. In general, `let`-bound polymorphism is too imprecise for scripting languages because it forces functions bound outside of the current lexical scope to be handled monomorphically; this would prevent e.g. our object encoding from having polymorphic methods.

Our approach to parametric polymorphism in Chapter 6 is based on flow analysis [62, 74, 33]. In the aforecited, polymorphic contours are permanently distinct once instantiated. In TinyBang, contours are optimistically generated and then merged when a call cycle is detected; this allows more expressive polymorphism since call cycles don’t need to be conservatively approximated up front. Contours are merged by injection into a restricted space of regular expressions over call strings, an approach that follows program analyses [43].

10.4 Object-Orientation

Asymmetric concatenation, case-sensitive dispatch with refinement on bindings, and call-site-based polymorphism are all language features of TinyBang critical to the object encoding presented in Section 8.6. TinyBang’s treatment of objects is unique in the current literature due to its ability to encode object extension and resealing without restricting fundamental subtyping properties.

TinyBang’s object resealing is inspired by the Bono-Fisher object calculus [9], in which mixins and other higher-order object transformations are written as functions. Objects in this calculus must be “sealed” before they are messaged; unlike our resealing, sealed objects cannot be extended. Some related works relax this restriction but add others. In [59], for instance, the power to extend a messaged object comes at the cost of depth subtyping; that is, the equivalent of the Java statement `Set<? extends Number> s = new HashSet<Integer>();` will not typecheck in that work. In [7], depth subtyping is admitted but the type system is nominal rather than structural; this is very similar to the decorator design pattern, in which the decorated object has the interface properties of the decorator and the decoratee type but not those of the concrete type which was decorated.

Unlike the above works, TinyBang's objects are encoded rather than being first-class members of the calculus. Many different approaches have been developed to accurately encode and type objects [12], but these approaches have been based on storing fields and methods as labeled members of a recursively-typed record [18, 34]. Although subtype constraint systems considerably improve the types of these recursive record encodings [24, 25, 23], the aforecited do not provide post-construction extensibility. TinyBang's object encoding makes use of a hybrid object encoding, tracking object fields in the form of a traditional labeled record but using a variant-based approach to dispatch messages. This form lends itself to extension as the underlying language supports primitive operations for extending and invoking onions.

10.5 Static Analysis Techniques

One technical development in this work is the strategy used to prove soundness in Chapter 4 (which this group first published in [49]). Proofs of type soundness typically proceed by "progress and preservation". [75]. Such an approach may be feasible for TinyBang in theory, but it proves cumbersome at best; given $e \longrightarrow^1 e'$, it is quite difficult (and may be impossible [38]) to demonstrate that $[\![e']\!]_{\mathrm{E}}$ is a subtype of $[\![e]\!]_{\mathrm{E}}$. There is also no reasonable regular tree or other form of denotational type model [5, 70, 15] of TinyBang given the flexibility of pattern matching. For example, polymorphism in TinyBang cannot be convex in the sense of the CDuce literature [15] since our pattern matching syntax is expressive enough to distinguish all top-level forms.

Instead, we design the operational semantics and type inference algorithm in tandem, intentionally keeping them quite similar in form (as in Section 3.4.1). By doing so, we prove soundness directly through simulation: we establish a simulation relation, maintain it throughout all program executions, and demonstrate that constraint closure reveals unsound programs through that relation (as in Chapter 4). This approach – establishing a direct relationship between the concrete operational semantics and an abstract representation of it – is quite similar to abstract interpretation; as such, TinyBang's type system can be viewed as a hybrid between a flow analysis [62] or abstract interpretation [20] and a traditional type system.

In comparison to abstract interpretation, however, our subtype constraint system is monotonic in nature. While abstract interpretation simulates the transition of abstracted program states, those transitions may be destructive on some data. This causes the number of states in naïve abstract interpretations to expand rapidly and much of the work in applying abstract interpretations is in collapsing those program states. Traditional type systems, by contrast, expand states by instantiating polymorphic types. This causes a conservative reconsideration of the code represented by those types, which leads to a rougher, less precise, and less expensive analysis.

Again, TinyBang is a hybrid between these two techniques. We accumulate facts monotonically as in traditional type systems; while abstract interpretations are forced to justify the merging of individual states, monotonicity gives confluence to the TinyBang type system, making such justifications trivial. The polymorphism model defined in Chapter 6, inspired in part by literature in static analysis [43, 44, 42], eagerly polyinstantiates contours and then relies upon a monotonic, confluent merging function to eliminate unnecessary flows when possible. Although eager contour creation is more similar to abstract interpretation's eager program state generation, we do not require any particular justification for the merging of polymorphic contours because the soundness proof is independent of the polymorphism model used.

10.6 Dynamic Analysis

Rubydust [6] is a tool which infers static properties for Ruby programs by tracing actual program executions using transparent data wrappers. In this way, Rubydust collects information about how data flows through the program from creation sites to use sites *by example* rather than by conservative static approximation. Method parameters are wrapped as they enter the body of the method under analysis and metadata is collected as uses arise. This has the advantage of avoiding false positives that arise as a result of conservative approximation or lack of alignment.

The techniques used by Rubydust are interesting in their own right, but they do not serve as a replacement for static type systems. Rubydust is sound only for those paths which were taken by one of the program executions in the provided trace suite; as a result, it only produces generally sound types if every possible trace is represented. While the paper notes that it is possible to alert the user in the event that paths exist which are not represented by the trace suite, that path analysis is subject to the same conservative approximation flaws that Rubydust originally avoided by relying on execution traces. In general, dynamic type inference shows promise and has already been demonstrated to yield a useful and interesting tool, but it does not satisfy the objectives of TinyBang because effort beyond that of writing the original program (akin to type signatures) is required to obtain the benefits of static analysis.

10.7 CDuce

In terms of features and expressiveness goals, TinyBang is very similar to CDuce [14]: both aim to be flexible languages built around constraint subtyping. Rather than typed scripting, CDuce's primary motivation is operating in a type safe way on complex heterogeneous data types (such as XSD-specified XML documents). These two goals have considerable intersection and CDuce exhibits much of the type expressiveness captured by TinyBang: it (locally) infers types for conjoined functions and performs path-sensitive typing on the dispatch of those functions.

In terms of formalization, however, TinyBang and CDuce are quite different. TinyBang's function conjunction expressiveness is driven by the type-indexed onion, which is fundamentally asymmetric, while CDuce uses intersection and negation types on functions. The asymmetry of onions is not amenable to denotational type models; because CDuce is formalized using symmetric set operations, it quite naturally fits a denotational model. This asymmetry is fundamental to the object encoding presented in Section 8.6 and, as CDuce's types require explicit orchestration for asymmetric dispatch, our object encoding is not possible in the CDuce language.[2]

10.7.1 Polymorphism

CDuce's type inference algorithm is limited to local inference only. This is fundamentally related to its polymorphism model. To remain modular, CDuce does not assign polymorphic types to function *values*; it uses a `let`-bound model. Although a traditional `let`-bound model would prevent TinyBang from expressing enough polymorphism for e.g. our object encoding, CDuce introduces a novel twist: the polyinstantiated type given to a function value can have multiple type substitutions, any of which can be used when necessary. In this way, the polymorphism of a function can escape the scope in which the function itself was named *so long as* the polyinstantiation routine can anticipate the manner in which the function will eventually be used. This motivates the local limitation to inference: type signatures on functions give the polyinstantiation process a target to use when creating these numerous substitutions.

[2]Strictly speaking, a first-order form of it would be possible. But the object encoding in Section 8.6 allows unusual higher-order expressiveness, such as the ability to create a subclass of a class whose identity is not statically known. Because CDuce does not have first class asymmetry, it is unable to express that behavior.

In a sense, CDuce is creating every polyinstantiation of a given function *a priori*. TinyBang, on the other hand, gives each function value a polymorphic type and polyinstantiates it on demand. This presents difficulties in modularity, as the type of a function may not be possible to fully analyze until the type of its higher-order functional arguments are known. As an example, consider the following LittleBang code:

```
let combine = fun (a,b,f) -> f(a,b) + f(b,a) in ...
```

The TinyBang type system does not attempt to determine precisely the type of `f` which is necessary to make the above function typecheck; it instead waits until an invocation occurs and simply determines type soundness *for that case*. In general, TinyBang makes several difficult problems easier by finitely enumerating the cases in which universal quantifiers are used rather than attempting to prove their universal properties. This has the advantage of allowing quite complex functions and their uses to typecheck successfully but the disadvantage that the full types of some functions are not completely understood by the type system. In other words, it is not sufficient to compile and assign a type to `combine`; the body of `combine` must be reanalyzed at each place that it is invoked.[3]

At the time of this writing, the decidability of the CDuce type inference algorithm remains open. Chapter 5 demonstrates that the TinyBang type inference algorithm is decidable. Although our current proof-of-concept implementation requires considerable tuning and improvement, it appears to be fast enough to be used in small to medium programs in practice.

10.7.2 Convexity

Key to the model of CDuce is a notion of the *convexity* of a set theoretic model. According to [77], for every finite set of types in a convex model, "if every assignment [of types to type variables] makes some of these types empty, then it is so because there exists one particular type that is empty for all possible assignments." In other words, convexity states that type polyinstantiation and type union are commutative. CDuce uses a convex set theoretic model; this ensures that empty sets in its model are treated uniformally and is, as stated by the aforecited, a cornerstone of the CDuce approach.

TinyBang's type system is not convex and this is key to its expressiveness.[4] The

[3]Our research group is currently pursuing an extension to the pattern matching system of TinyBang which would allow modular compilation by introducing function pattern matches on module boundaries, but that work is outside of the scope of this document. See Section 11.1.1 for more information.

[4]Although TinyBang's constraint grammar does not have type upper bounds or negation types (e.g. $\alpha \leq \neg\tau$ from [77]), such forms can be encoded in TinyBang's type system using application and compound function types. In particular, the TinyBang expression `((fun int -> () ()) & (fun () -> 0)) x` produces a type error whenever `x` is an `int` and yields `0` otherwise; this is, in effect, upper-bounding the type of `x` by $\neg$ `int`.

CDuce type system uses convexity to preserve the opaqueness of parametric polymorphism; by opaqueness, we mean e.g. that only the identity function has type $\forall\alpha.\,\alpha \rightarrow \alpha$ (since α cannot be inspected by the body of the function). TinyBang specifically avoids this property[5]: in keeping with the semantics of scripting languages, any value may be type analyzed at any point in the runtime. CDuce uses convexity to enforce a "uniformity of behavior", but the behavior of a scripting function on its argument is often non-uniform by case analysis. This is a consequence of the difference of domains: CDuce aims to provide an ML-like type theory for heterogeneously-typed, structured documents while TinyBang aims to provide a type inference algorithm for a scripting language.

[5]We do not assign misleading types to functions; in fact, in TinyBang, only the identity function has type $\alpha_1\backslash\{\texttt{()} <: \alpha_1\} \rightarrow \alpha_2\backslash\{\alpha_1 <: \alpha_2\}$. But other identity function types exist with distinct flow properties and not all functions which are equivalent to the identity function can be identified by their type.

Chapter 11

Future Work

Our strategy for developing a typed scripting language relies upon developing a core calculus onto which the operations of a scripting language can be encoded. Although TinyBang as presented in Chapter 3 serves as a foundation for several encodings (as seen in Chapter 8), extensions to or reformulations of the core calculus will make other encodings available and lead to a more complete and featureful surface language. Here, we discuss some potential adjustments and augmentations to the theory presented above and describe the language features which may be possible as a result.

11.1 Pattern Expressiveness

Patterns in TinyBang are fairly simple: with the exception of conjunction patterns, each pattern precisely matches some type constructor. Conjunction patterns are a fairly simple addition which essentially just require multiple subproofs of pattern matching. In a language with types as precise as TinyBang's, however, much more sophisticated patterns are possible.

First and foremost, other logical combinations of patterns are trivial to add. Disjunction patterns, for instance, are straightforward to introduce. The naïve implementation of disjunction patterns involves exponential work in the number of disjunctions, but this work should be reasonable in practice as patterns tend not to be terribly large or complex in real code. Generalized negation patterns introduce similar complexity problems, but (due to the constructive nature of pattern matching proofs in TinyBang) *top-level* negation patterns are trivial to introduce. Using these patterns, it becomes possible to encode more efficient and manageable versions of e.g. the function encodings from Section 8.5.

More interestingly, the subtype constraint form of TinyBang's patterns naturally supports *recursive patterns*. Recursive patterns are not novel; existing work [27] typechecks recursive patterns in the context of an ML-style type system where type declarations are available. TinyBang may be extended to include recursive patterns simply by relaxing the acyclicity restriction appearing in Section 3.3.2. In doing so, however, the contractiveness of patterns must be preserved. Much like contractive types [40], contractive patterns are those which may only recurse after having made progress (e.g. by passing through a meaningful constructor); thus, the pattern $\alpha\backslash\{\texttt{`A}\,\alpha <: \alpha\}$ is contractive but the pattern $\alpha\backslash\{\alpha * \alpha <: \alpha\}$ is not. Further, the formalization of pattern matching in the type system must then include an occurrence check to ensure the existence of finite proofs of recursive patterns matching recursive data; the soundness proof must make use of a pumping lemma to argue that this occurrence check is safe. In a structurally subtyped language such as TinyBang, recursive patterns make it possible to e.g. verify the recursive structure of a list by pattern matching. This allows more sophisticated interface specification where it is desired.

11.1.1 Function Patterns

As is true in almost any language with pattern matching, patterns in TinyBang only match first-order data. We believe that considerable expressiveness may be gained by including *function patterns*: patterns which match the behavior of functions. For instance, one may imagine a function pattern `int ~> char` matching only those functions which, when given a value containing an `int`, will return a `char`; any other arguments will fail to match that pattern. Our previous explorations into this problem have revealed it to be subtle and complex: with such patterns, for instance, it is possible to create paradoxical pattern matches (e.g. "this function is the identity function only if it is not") which must be identified and reported by the type system. Our most recent work in that area [41] demonstrates that these corner cases can be identified and rejected and, in general, that function patterns are a viable and untapped form of expressiveness in the programming languages literature.

Function patterns make an entire range of sophisticated encodings possible. Type signatures in the form of function patterns become possible: for instance, the pseudocode expression `(f * (int ~> int) -> f)` describes a restricted form of the identity function which only accepts `int` $\rightarrow$ `int` functions. To verify that a function `g` has that type, one merely needs to pass it through that restricted identity function when it is defined. If `g` does not have that type, it fails to match the function pattern and, as there are no alternative patterns, a type error would result.

Function patterns also suggest the possibility of a structural multimethod dispatch. As mentioned in Section 10.2, existing work on multimethods only covers nominal types. By conditionally matching on the dispatches of a compound function, one can analyze a variant-encoded object to determine the types of its methods without invoking them. In essence, this would allow a form of `instanceof` (to use Java terminology) on structurally subtyped objects.

11.2 Separate Compilation

The TinyBang type system, as presently defined, produces extremely precise types that capture much of the flow of the program. The functions `fun x -> x + 1` and `fun x -> 1 + x`, for instance, have distinct types (although they are operationally equivalent in practice). Further, the types of functions are open in a sense: it is not immediately evident from the TinyBang type of `fun f -> f 3` that its argument must be a function accepting an integer. As a result, small changes to some code may dramatically influence the types of code elsewhere in the program. This means both that quite subtle bugs are admitted by the language and that typechecking is a whole-program effort. We view this as an effect of accurately typing a scripting language: a common criticism of dynamic languages is that they admit subtle bugs with far reaching effects and TinyBang's type system is simply discovering this bug statically (as a type system is meant to do). We still, however, wish to improve upon this result.

Typechecking in the system presented here is whole-program for the same reason that inference is only local in CDuce: both systems must know all of the ways in which a function may be used in order to polyinstantiate it correctly. In a sense, there is a dimension of opaqueness which is missing from the polymorphism of TinyBang: since the caller's behavior may always type analyze its argument, TinyBang (as presented in Chapter 3) has no way of ensuring that a different argument type (even a subtype of one already being used) would not produce a different output type altogether. In a sense, this problem is fundamental to the domain. The TinyBang type system is designed to infer subtle, unstated invariants about the behavior of a script and, in theory, there is no boundary at which we can say for any program that the information is no longer relevant. In Python, for instance, one may create an object with only a `write` method and pass it to the constructor of a `GzipFile`. This is sound as long as one only uses the `GzipFile` in a writing fashion; calling the `read` method of `GzipFile` would be an error. Inferring the exact boundary at which this invariant is no longer relevant is, generally speaking, impossible because the `GzipFile` may be stored on the heap and be subject to aliasing.

With the addition of top-level type signatures, however, separate compilation may be recovered. Although our intent is never to *require* top-level type signatures in our typed scripting language, providing them for the purpose of improving compilation performance and isolating bugs is not contrary to that objective. Of course, the constraint types which are used in TinyBang are both far too complex to be used in signatures (users would be at a loss to write the correct constraint set for a function) and far too fragile (tiny changes to the code make the type signature incorrect). That said, the function patterns mentioned above may provide ML-like type signatures which can be checked by the pattern matching logic. Once a function is confirmed to conform to a given function pattern, a type equivalent to that function pattern can be used in lieu of the function's actual constraint type. This allows total separation of the implementation from the client as long as the function pattern remains the same; thus, decorating each exposed member of a module with an appropriate pattern would allow that module to be typechecked separately from the code that uses it.

11.3 Performance

One challenge to producing a scalable typechecker for TinyBang is creating an efficient constraint closure algorithm. A considerable body of work has been developed around constraint simplification ([55, 23] among numerous others) and some of that work may be applied to this theory. The nature of polymorphism in TinyBang, however, presents additional challenges.

In TinyBang's type system, we eagerly create new polymorphic contours when applications occur; we later merge those contours which we deem unnecessarily precise. Between instantiation and merging, however, a considerable amount of work may be done and, unfortunately, we may discover at merge time that this work has been redundant. To prevent this, an algorithm must endeavor to discover contour merges as quickly as possible. How this may be done is an open question, although the quality of a strategy is clearly dependent upon the polymorphism model in question (which defines the merging function Υ). In CR, for instance, merging happens during recursion. By pursuing constraint closure over variables with longer contours first, we can improve our chances of discovering recursion before replicating work. In a sense, we suggest that a sort of "depth-first" closure using CR may yield the best results. Performance may also be improved by e.g. tracking which functions in the source have participated in recursive calls and prioritizing their expansion.

In terms of the constraint closure algorithm itself, some questions remain regarding efficient implementation. Typically, a constraint closure algorithm is improved by adding a number of indexing data structures to the constraint set, making it easier to find e.g. lower bounds for a given type variable. Due to the effects of contour merging (as discussed in Section 9.2), TinyBang does not lend itself as readily to such optimizations: when a merge is performed, the indexes of the data structure must be updated to ensure that they reflect the new equivalence between type variables. We suspect that some benefits may be realized through a union-find-like indexing structure, but the performance benefits of this versus other approaches is not yet clear.

11.4 Type Errors

Subtype constraint systems produce incredibly precise types. As a result, type errors are often bafflingly complex: even the simplest programs are expressed using hundreds of constraints, most of which are involved in a given error. Constraint simplification (as mentioned above) has the advantage of mitigating this problem slightly, but even simplified constraint sets are unreadable in practice. Tools have been developed to display the program flow more directly on the code [29], but this approach met with limited success due to the sheer amount of information each type error conveys. For TinyBang's type system to be usable in practice, we must develop a presentation mechanism which relays type errors in a usable fashion.

When a type error arises in the TinyBang type system, the type checker has already constructed a proof of inconsistency. Simply providing the proof of the error (as compilers do today) is ungainly: the programmer must then read through the information (most of which he or she already knows) in order to understand the problem. We believe that an interactive type error debugging tool is necessary; such a tool would assist the programmer in navigating the erroneous flow. As an example, consider the following LittleBang code:

```
1 let repeat = fun(f,x,n) ->
2       if n == 0 then x
3       else let y = f(x) in repeat(f,y,n-1)
4 in
5 let foo = (x:char) -> 4 in
6 repeat(foo,'a',5)
```

Here, the `repeat` function applies a function to an argument some number of times. In this use of that function, type error occurs on line 3: after one invocation of `foo`, the argument becomes an `int` (rather than a `char`) and the function can no longer be called with the argument. There are several possible causes of this problem:

- The output on line 5 may be incorrect; perhaps `foo` should return a `char`.
- The input on line 6 and the pattern on line 5 may be incorrect; perhaps `foo` should receive an `int`.
- The logic on line 3 may be incorrect; perhaps the programmer did not intend `repeat` to do what it currently appears to do.

- Perhaps none of the above are incorrect but (for some reason) the programmer wanted to use `repeat` for a simple application; in that case, the argument `4` on line 6 should instead be a `1`. (TinyBang's type system is powerful enough to infer a correct type for this.)

Determining which of the above (if any) are the actual problem requires us to understand the programmer's intentions. We can start with the inconsistency on line 3 where an `int` is passed to `foo` and ask the programmer what about the current situation seems wrong. Supposing for a moment that the first problem above is the issue at hand – `foo` should return a `char` – the programmer can indicate that the argument at this point should never have been an `int`. Using the proof of inconsistency, the tool can then demonstrate how that `int` reached that point in the program: as the argument to `repeat`, which was received as the output of `f` having the type of `foo` (which then leads the programmer to the output of `foo` on line 5). If the second problem above is the issue, the programmer instead indicates that the argument is fine but the function is in error; in that case, the tool can indicate the manner in which the function `f` arrived at that point in the program, tracing back to the call on line 6.

While we believe an interactive tool such as the above would make type errors manageable in a subtype constraint system such as TinyBang's, there are further problems to solve. The strategy described above may lose precision in some cases of recursion. Additional tooling is necessary to ensure that *LittleBang* type errors are properly represented by the tool. To properly develop a broader strategy for addressing these errors, it may also help to classify the kinds of mistakes which appear in LittleBang programs and determine if blaming individual flows is sufficient to diagnose problems.

11.5 Flow Sensitivity

As stated in Section 10.5, there are strong connections between the TinyBang type system and static analysis literature. In that literature, analyses are described to be "sensitive" to certain runtime properties. An analysis which is aware of conditional branching is said to be *path-sensitive*. An analysis that reconsiders code when it is invoked at different points is said to be *context-sensitive*. An analysis which is aware of the chronological ability of data to reach certain program points (e.g. is aware of the passage of time) is said to be *flow-sensitive*.

Considered as an abstract interpretation, the TinyBang type system is path-sensitive (due to compound function dispatch) and context-sensitive (due to call-site polymorphism) but not flow-sensitive. If TinyBang is extended with reference cells as in Section 7.4, for instance, the type of the reference cell is merely a union of all types which could appear within it. As a result, the following LittleBang code would produce a false positive type error:

```
let r = ref 'z' in
(r := 4; !r + 1)
```

In the above, the contents of `r` are given the type `int` ∪ `char`, so the addition on the second line produces a type error even though, at runtime, there is no way for `r` to contain a `char` at that point.

Previous flow-sensitive type inference systems exist; the FT calculus [52], for instance, incorporates program flow and acts as a foundation for the Whiley programming language, a hybrid object-oriented and functional language with an emphasis on sophisticated static contract checking. That system is neither context- nor path-sensitive, however, and a naïve application of the concepts of FT to the TinyBang calculus is not possible due to fundamental differences in the theories.

Nonetheless, we believe it to be possible to extend TinyBang to include flow-sensitive reasoning by adding *chronological constraints*. In such a system, the transitivity rule is replaced by a lazy lower-bound lookup; that is, when attempting to determine the type of `!r` in the above example, we move backward chronologically until we find either the definition of `r` or (in this case) an assignment to `r`. This chronological model cannot be perfect, of course: as with the polymorphism model, we expect to lose information on recursion. Extending TinyBang with flow-sensitivity would more precisely model the behavior of some scripting language phenomena and, contrary to intuition, may actually improve the performance of the constraint closure algorithm [74] by eliminating some spurious constraints and their respective conclusions.

Chapter 12

Conclusions

Scripting languages have gained widespread acceptance in recent decades because they allow programmers to work faster and more efficiently. By and large, scripting languages provide that efficiency by eliminating any static notion of types (or significantly weakening it as in the case of more recent languages such as TypeScript). Without the benefits of static typing, scripting languages are less scalable than traditional languages: their runtime performance suffers and debugging is more difficult.

We have presented here a strategy for designing a language which has much of the expressiveness and terseness of scripting languages but which is supported by a static type system and type inference engine. Our strategy is founded in subtype constraint theory but is deeply related to abstract interpretation: we observe that, in A-normalized forms of both the evaluation system and the type system, the process of typechecking becomes quite similar to abstractly executing the program. We have shown how to combine this typechecking strategy with an expressive data conjoiner called the *onion*, an asymmetrically-concatenated, type-indexed record, to capture common scripting language functionality such as default arguments, operator overloading, and extensible object orientation. This expressiveness is supported by a novel call-site (as opposed to `let`-bound) polymorphism model which aggressively polyinstantiates contexts and then unifies them later if their separation is judged to be unnecessary. We also used the connection between our type system and abstract interpretation to prove the system's soundness by simulation, a strategy common in abstract interpretation literature but novel in the realm of type theory.

In order to keep the formalization simple, we developed an operational semantics and type system for the minimal core language TinyBang; we then demonstrated the encoding of LittleBang, a scripting language which includes common and non-trivial scripting language features not found in TinyBang. In particular, the encoding of LittleBang's object system constitutes the first fully type-inferred variant-based object model. This object model also has extensibility properties beyond that of extensible typed object encodings found in the literature.

The expressiveness of LittleBang suggests that it (or another language with similar core operations) may be usable in practice in the domain of scripting. Considerable engineering effort is needed to build an efficient and useable toolchain for the language, but the theory presented here has shown the type system to be sound and decidable. The language development strategy presented in this text has therefore shown it possible to develop a typed scripting language: a programming language with the key flexibility of modern scripting languages but with the benefits of static typing.

Bibliography

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

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

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

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

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

[6] J.-h. D. An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic inference of static types for Ruby. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 459–472, New York, NY, USA, 2011. ACM.

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

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

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

[10] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, pages 303–311, New York, NY, USA, 1990. ACM.

[11] G. Bracha and D. Griswold. Strongtalk: typechecking Smalltalk in a production environment. In *OOPSLA*. ACM, 1993.

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

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

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

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

[16] M. M. T. Chakravarty, G. Keller, S. P. Jones, and S. Marlow. Associated types with class. POPL '05. ACM.

[17] R. Chugh, P. M. Rondon, and R. Jhala. Nested refinements: A logic for duck typing. In *POPL*, 2012.

[18] W. R. Cook, W. Hill, and P. S. Canning. Inheritance is not subtyping. In *POPL*, 1990.

[19] M. Corporation. *TypeScript Language Specification v1.4*, October 2014.

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

[21] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

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

[23] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *OOPSLA Conference Proceedings*, 1995.

[24] J. Eifrig, S. F. 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.

[25] J. Eifrig, S. F. Smith, V. Trifonov, and A. Zwarico. Application of OOP type theory: State, decidability, integration. In *OOPSLA Conference Proceedings*, pages 16–30, 1994.

[26] J. Estep. Encoding classes and modules in TinyBang. Master's thesis, The Johns Hopkins University, 2014.

[27] M. Fähndrich and J. Boyland. Statically checkable pattern abstractions. In *ICFP*, pages 75–84. ACM Press, 1997.

[28] M. Fähndrich, J. Rehof, and M. Das. From polymorphic subtyping to CFL reachability: Context-sensitive flow analysis using instantiation constraints. Technical report, Microsoft Research, March 2000.

[29] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching bugs in the web of program invariants. In *PLDI*, 1996.

[30] T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, 1991.

[31] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks. Static type inference for Ruby. In *SAC*, 2009.

[32] M. Furr and J. S. Foster. Polymorphic type inference for the JNI. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP'06, pages 309–324, Berlin, Heidelberg, 2006. Springer-Verlag.

[33] T. Gilray and M. Might. A survey of polyvariance in abstract interpretations. In *Trends in Functional Programming*. 2014.

[34] N. Glew. An efficient class and object encoding. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA, pages 311–324, New York, NY, USA, 2000. ACM.

[35] J. O. Graver and R. E. Johnson. A type system for Smalltalk. In *In Seventeenth Symposium on Principles of Programming Languages*, pages 136–150. ACM Press, 1990.

[36] R. Harper and B. Pierce. A record calculus based on symmetric concatenation. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 131–142, Orlando, Florida, Jan. 1991. ACM Press.

[37] F. Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, Apr. 1993.

[38] F. Henglein and J. Rehof. Constraint automata and the complexity of recursive subtype entailment. In *ICALP*, volume 1443 of *Lecture Notes in Computer Science*, 1998.

[39] N. Jimenez. LittleBang data structure encodings, 2014.

[40] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. In *POPL*, 1984.

[41] P. H. Menon, Z. Palmer, A. Rozenshteyn, and S. F. Smith. What is your function?: Static pattern matching on function behavior. Submitted (ICFP 2015).

[42] M. Might and P. Manolios. A posteriori soundness for non-deterministic abstract interpretations. In *In VMCAI '09: Proceedings of the 10th International Conference*

on Verification, Model Checking, and Abstract Interpretation, pages 260–274. Springer-Verlag.

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

[44] M. Might and O. Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *ICFP*, Portland, Oregon, 2006.

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

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

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

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

[49] Z. Palmer, P. Menon, A. Rozenshteyn, and S. Smith. Types for flexible objects. In *Programming Languages and Systems*. 2014.

[50] J. Palsberg and S. F. Smith. Constrained types and their expressiveness. *TOPLAS*, 18(5):519–527, September 1996.

[51] J. Palsberg and T. Zhao. Type inference for record concatenation and subtyping. *Information and Computation*, 189(1):54–86, 2004.

[52] D. J. Pearce. A calculus for constraint-based flow typing. In *Proceedings of the 15th*

Workshop on Formal Techniques for Java-like Programs, FTfJP '13, pages 7:1–7:7, New York, NY, USA, 2013. ACM.

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

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

[55] F. Pottier. Simplifying subtyping constraints: a theory. *Inf. Comput.*, 170:153–183, November 2001.

[56] A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe & efficient gradual typing for TypeScript. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 167–180, New York, NY, USA, 2015. ACM.

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

[58] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI*, 2010.

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

[60] Ruby Users Group. *About Ruby*, February 2015.

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

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

[63] shootout.debian.org. The computer language benchmarks game. `http://shootout.alioth.debian.org/`.

[64] J. Siek and W. Taha. Gradual typing for objects. In E. Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 2–27. Springer Berlin / Heidelberg, 2007.

[65] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP, pages 117–128, New York, NY, USA, 2010. ACM.

[66] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI, pages 132–141, New York, NY, USA, 2011. ACM.

[67] V. Trifonov and S. F. Smith. Subtyping constrained types. In *Proceedings of the Third International Static Analysis Symposium*, volume 1145 of *LNCS*, pages 349–365, Sept. 1996.

[68] D. Van Horn and H. G. Mairson. Deciding kCFA is complete for EXPTIME. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 275–282, New York, NY, USA, 2008. ACM.

[69] M. M. Vitousek, A. M. Kent, J. G. Siek, and J. Baker. Design and evaluation of gradual typing for Python. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, pages 45–56, New York, NY, USA, 2014. ACM.

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

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

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

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

[74] T. Wang and S. F. Smith. Precise constraint-based type inference for Java. In *ECOOP*, pages 99–117, 2001.

[75] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1992.

[76] A. K. Wright and S. Jagannathan. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems*, 20:166–207, 1998.

[77] Z. Xu. *Parametric Polymorphism for XML Processing Languages.* PhD thesis, Université Paris-Diderot-Paris VII, 2013.

Cirriculum Vitae

EDUCATION

The Johns Hopkins University	Baltimore, MD
Doctor of Philosophy	August 2015
The Johns Hopkins University	Baltimore, MD
Master's of Computer Science	May 2010
Indiana University of Pennsylvania	Indiana, PA
Bachelor's of Computer Science, Summa Cum Laude	May 2004

TEACHING EXPERIENCE

Lecturer	*Object-Oriented Software Engineering*	Fall 2014
	Department of Computer Science	The Johns Hopkins University
Lecturer	*Introduction to Programming in Java*	Summers 2009–2011, 2014
	Department of Computer Science	The Johns Hopkins University
Teaching Assistant	*Object-Oriented Software Engineering*	Falls 2008-2013
	Department of Computer Science	The Johns Hopkins University
Teaching Assistant	*Programming Languages*	Springs 2010, 2011, 2013
	Department of Computer Science	The Johns Hopkins University
Teaching Assistant	*Artificial Intelligence*	Spring 2009
	Department of Computer Science	The Johns Hopkins University

TEACHING AWARDS

Professor Joel Dean Excellence in Teaching Award	May 2015
Whiting School of Engineering Outstanding Teaching Award	May 2010

PUBLICATIONS AND WRITINGS

Peer-Reviewed Research

H. Menon, **Z. Palmer**, A. Rozenshteyn, S. Smith. What is Your Function?. *20th ACM SIGPLAN Conference on Functional Programming (ICFP)* (2015 *submitted*).

H. Menon, **Z. Palmer**, A. Rozenshteyn, S. Smith. Types for Flexible Objects. *12th Asian Symposium on Programming Languages and Systems (APLAS)* (2014).

H. Menon, **Z. Palmer**, A. Rozenshteyn, S. Smith. A Practical, Typed Variant Object Model. *International Workshop on Foundations of Object-Oriented Programming Languages (FOOL)* (2012).

P. Shyamshankar, **Z. Palmer**, Y. Ahmad. K3: Language Design for Building Multi-Platform, Domain-Specific Runtimes. *International Workshop on Cross-Model Language Design and Implementation (XLDI)* (2012).

H. Menon, **Z. Palmer**, A. Rozenshteyn, S. Smith. BigBang: Designing a Statically-Typed Scripting Language. *International Workshop on Scripts to Programs (STOP)* (2012).

Z. Palmer, S. Smith.. Backstage Java: Making a Difference in Metaprogramming. *26th ACM SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)* (2011).

Educational Writing

M. Grant, **Z. Palmer**, S. Smith. *Principles of Programming Languages.* (2009)

INDUSTRY EXPERIENCE

Software Engineering Consultant Amentra, Inc. / Red Hat, Inc.	July 2007 – July 2008
Independent Developer	August 2006 – July 2007
Student Intern Didera, Inc.	Summers 2001 – 2003