Safe, Fast and Easy: Towards Scalable Scripting Languages

by

Pottayil Harisanker Menon

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

Baltimore, Maryland

Feb, 2017

Abstract

Scripting languages are immensely popular in many domains. They are characterized by a number of features that make it easy to develop small applications quickly - flexible data structures, simple syntax and intuitive semantics. However they are less attractive at scale: scripting languages are harder to debug, difficult to refactor and suffers performance penalties. Many research projects have tackled the issue of safety and performance for existing scripting languages with mixed results: the considerable flexibility offered by their semantics also makes them significantly harder to analyze and optimize.

Previous research from our lab has led to the design of a typed scripting language built specifically to be flexible without losing static analyzability. In this dissertation, we present a framework to exploit this analyzability, with the aim of producing a more efficient implementation

Our approach centers around the concept of adaptive tags: specialized tags attached to values that represent how it is used in the current program. Our framework abstractly tracks the flow of deep structural types in the program, and thus can

ABSTRACT

efficiently tag them at runtime. Adaptive tags allow us to tackle key issues at the heart of performance problems of scripting languages: the framework is capable of performing efficient dispatch in the presence of flexible structures.

Acknowledgments

At the very outset, I would like to express my gratitude and appreciation to my advisor Prof. Scott Smith. He has been a constant source of ideas, inspiration and encouragement. His patience has been exemplary and he has constantly fostered my intellectual curiosity. I would like to thank him for helping me grow not only as a scientist but as a person as well.

I would also like to thank, Prof. John Peterson and Prof. Yanif Ahmad for serving as my committee members, and providing me with their comments and suggestions. A special thanks towards my lab-mates Zachary Palmer, Alexander Rozenstheyn, Leandro Facchinetti, for all the intellectual discussions and constant support they provided at every stage.

Last but not the least I would like to extend my appreciation towards my family and friends, without whose motivation and support I would not have been able to embark on this long journey. A special thanks to my mother and my wife, whose encouragement, understanding and wishes kept me focused on this path.

Contents

Abstract **ii**

Acknowledgments **iv**

List of Figures **ix**

1 Introduction **1**

1.1 Previous Work 3

1.2 Our Approach 7

1.3 Outline 9

2 Design of Typed Scripting Languages **11**

2.1 Duck Typing 11

2.2 Flexible data types 15

2.3 Pattern Matching 17

2.4 Call-site Polymorphism 20

2.5 TinyBang Core 20

3 Overview of the Tag Framework **22**

3.1 Pattern matching 22

3.2 Adaptive Tags 24

3.3 Porting to TinyBang 30

4 Formalization of TinyBang Core **33**

4.1 Notation 33

4.2 Grammar 34

4.3 Well-formed Expressions 35

4.4 Operational Semantics 36

4.4.1 Compatibility 36

4.4.2 Small-Step Evaluation 39

4.5 Type System 41

4.5.1 Type Grammar and Initial Alignment 42

4.5.2 Type Compatibility 44

4.5.3 Constraint Closure 45

4.6 Properties of the Semantics 48

5 Formalization of Tagged TinyBang Core **52**

5.1 Notation 52

5.2 Grammar 53

5.3 Operational Semantics 54

5.4 Tag Derivation 57

5.4.1 Type Compatibility 58

5.4.2 Constraint Closure 59

5.4.3 Deriving the Parameters 64

5.5 Properties of Tagged TinyBang Core 66

6 The Bisimulation Proof **68**

6.1 Bisimulation of Type Systems 69

6.2 Properties of the Tag Generation System 72

6.3 Properties of the Environment(s) 75

6.4 Tag Propogation 78

6.5 Uniqueness of Dispatch 83

6.6 Proof of Bisimulation 86

7 Implementation **87**

7.1 Interpreters 88

7.2 Performance 91

8 Related Work **94**

8.1 Pattern Matching 94

8.2 Object-Oriented Dispatch 95

8.3 Heterogeneous Cases 97

9 Future Work **98**

9.1 Improving the Tag Framework 98

9.2 Onions and Data Layout 101

10 Conclusions **104**

A Formalizing Tagged TinyBang **106**

A.1 Grammar 106

A.2 Operational Semantics 107

A.3 Tag Derivation 109

A.4 The Bisimulation Property 115

Bibliography **128**

Curriculum Vitae **136**

List of Figures

4.1 TinyBang Core ANF Grammar 35
4.2 Extensions to the TinyBang Core grammar 36
4.3 Extended notation for TinyBang Core compatibility 37
4.4 TinyBang Core compatibility relation 38
4.5 TinyBang Core operational semantics 41
4.6 TinyBang Core stuck states 41
4.7 TinyBang Core type grammar 43
4.8 Initial constraint derivation 44
4.9 Extended notation for TinyBang Core type compatibility 45
4.10 TinyBang Core type compatibility relation 46
4.11 TinyBang Core constraint closure 47
4.12 TinyBang Core inconsistentency 48
4.13 Simulation of expressions and constraints in TinyBang Core 50

5.1 Extensions to the TinyBang Core grammar 53
5.2 Extensions to the Tagged TinyBang Core grammar 54
5.3 Tagged TinyBang Core operational semantics 56
5.4 Type compatibility extensions for Tagged TinyBang Core type grammar . . 58
5.5 Tagged TinyBang Core type compatibility relation 60
5.6 Constraint closure discontinuity example 61
5.7 Constraint Closure extensions for Tagged TinyBang Core type grammar . . 62
5.8 Tagged TinyBang Core constraint closure 63
5.9 Tagged TinyBang Core Tag Closure 64

6.1 Augmenting TinyBang Core with an extra rule 80

A.1 TinyBang ANF Grammar 106
A.2 Extensions to the TinyBang grammar 107
A.3 Extensions to the TinyBang Core grammar 107
A.4 Tagged TinyBang operational semantics 109
A.5 Type compatibility extensions for Tagged TinyBang type grammar 110
A.6 Tagged TinyBang type compatibility relation 111
A.7 Tagged TinyBang type application matching 112
A.8 Constraint Closure extensions for Tagged TinyBang type grammar 113

A.9 Tagged TinyBang constraint closure 113
A.10 Tagged TinyBang Tag Closure 114
A.11 Augmenting TinyBang with an extra rule 119

Chapter 1

Introduction

The term "scripting language" is not formally defined, but the languages are usually characterized by a number of features that make it easy to develop small applications quickly:

- **Minimal redundancy**: These languages tend to be terse: variable and type declarations are usually not necessary (and often not allowed) while common idioms can be expressed with minimal syntax. Most scripting languages also aim for an intuitive semantics; what you see is usually what you get. Interface declarations are not required since the languages usually operate by "duck typing" - a piece of code will work correctly as long as its parameters have the expected "structure".

- **Fluid data**: In comparison to traditional languages, scripting languages tend to allow significant changes to the structure and behavior of their data types at runtime. For example, languages like Python and Javascript permit dynamic addition and modification of fields and methods on objects and many languages support handling

failed method dispatches.

- **Flexible Typing**: Due to the level of flexibility they offer, scripting languages tend to be hard to type; such languages are largely dynamically typed. The languages tend to emphasize unit testing and documentation to ensure correctness in lieu of type systems or complex static analysis.

With increasing code and team sizes, the advantages of scripting languages are reduced considerably. Large programs in these languages tend to be harder to extend and maintain as they offer very limited static guarantees. In languages like Python and Javascript, where there are no explicit interfaces to program to, type errors resulting from improper use of interfaces emerge only at runtime. Isolating bugs can therefore be a tedious process in comparison to large programs written in a statically typed language.

A related issue that arises with the increasing scale is that of performance. Runtime performance of scripting languages is weaker than their statically typed counterparts [1]. Much of the issue stems from two sources - the lack of static, context sensitive type information and the extreme flexibility in the structure of runtime data. The absence of type information forces expensive runtime checks, while the ad-hoc nature of data types makes it hard to establish fixed layouts for runtime objects.

Several attempts have been made to establish static type systems over existing dynamic languages in order to recover some of its benefits [16, 33, 9, 28, 57]. In practice, however, retrofitting a type system on to an existing dynamic language is very difficult; the languages have been designed without a type system in mind and often have features and semantics that are adversarial to type checking.

Some scripting languages use a just-in-time (JIT) compiler to to address performance issues; the JIT runs side-by-side with the program and performs optimizations when it appears possible as per its heuristics. The past decade has seen a tremendous amount of research in JIT technologies for dynamic languages, which in turn, indicates the increasing importance of performance in this domain. Just-in-time compilation has been a success. For example, JITs have been instrumental in making modern Javascript viable in the browser and on the server. However modern JIT compilers are complex beasts: the heuristics are complicated, performance requirements are stringent and the optimizations brittle.

In practice, only a small set of features in scripting languages such as `eval` function, prove to be a fundamental hindrance to static analysis. Indeed most of these features are not strictly necessary and were added to the language before analysis was a concern. Indeed, typed scripting languages already exist [28] demonstrating that static analyzability is not orthogonal to flexibility. A previous dissertation from our lab [39] presented LittleBang, a small, statically typed scripting language designed from scratch and TinyBang a core calculus for scripting. In this dissertation, we present a framework to exploit the analyzability of typed scripting languages, with the aim of producing efficient implementations.

1.1 Previous Work

A considerable amount of effort has been devoted, over the years, to making scripting languages safe and performant. We briefly discuss this research in order to provide some context.

1.1.1 Typechecking Existing Languages

Retrofitting a static type system to a language has the advantage of adding value to an existing language. So it is unsurprising that a number of attempts [16, 33, 28] have been made in this direction in the past with varying degrees of success.

A major issue when dealing with existing languages is that real code has already been written for it and they tend to make significant use of the dynamism of the language: for example, an analysis of Javascript programs [46] found that many rely on polymorphism, addition/removal of object fields, variadic functions and dynamic operations like `eval`. This is a significant constraint on the type system designer and necessitates all manners of compromise to make the system usable.

DRuby [28] is arguably the most successful attempt at typing an existing scripting language at the time of this writing. It offers a complex type system based on subtype constraint theory and local flow sensitivity that captures the behavior of many Ruby programs; however the system is neither sound nor complete; the language admits some unsound programs and rejects some valid ones.

1.1.2 Gradual Typing

Gradual typing [52] is another common approach to add some static guarantees to otherwise dynamic languages. In this model, typed and untyped code are allowed to co-exist with runtime checks enforcing the boundary between the two worlds.

For example, Typed Racket supports both typed and untyped modules. Type signatures are required and enforced at the top level for typed modules. When interacting with

untyped code, dynamic checks ensure the type correctness of data crossing the boundary.

TypeScript [21] is essentially a superset of Javascript with gradual typing support. Unlike Typed Racket, it does not require type annotations. It defines a structural subtyping relation between objects and then infers the types when possible. TypeScript performance appears to be on par with Javascript, but their type system is intentionally unsound in order to support annotations on existing Javascript code. However further work in the area appears to have produced two variants [45, 47] with sound type systems.

Gradual typing is a viable strategy for existing languages. However its primary focus is on providing a reasonably safe conduit between typed and untyped code. This unfortunately restricts us from making a number of optimizations and runtime guarantees regarding gradually typed code, which is both contrary to our safety goals *and* is likely to significantly affect performance. Indeed a recent study [54] seems to indicate that sound gradual typing may have significant and somewhat unpredictable overhead in systems where the typed and untyped modules mix.

1.1.3 Scripting and Performance

Scripting languages are slow. A part of the reason is that they are typically implemented via interpreters: micro-benchmarking[1] shows that interpreted languages run an order of magnitude slower than compiled ones [1]. But there are also other, more inherent, issues which can be broadly classified under two headings - data layout issues and dynamic dispatch issues. For example, it is well known that lookup operations in languages like PHP and Python are slow by default [57]. This is primarily an issue of data layout.

[1] with all the associated caveats

The languages are, typically, incapable of statically ascertaining the structure of runtime data and must perform the lookups at runtime dynamically. Another common source of inefficiency, and an example of a dispatch issue, is polymorphic code; the implementation must usually perform expensive runtime checks before dispatching to real operations on the underlying hardware.

Considerable efforts have been dedicated to speeding up scripting languages [57, 8, 48, 3, 20] . The most common way is to use a Just-in-Time (JIT) compiler. A heroic amount of effort, both industrial and academic, has gone in to optimizing Javascript virtual machines using JIT compilation over the last decade, even though the language was never designed to be JIT-compiled. With Python, at the time of this writing, the most common strategy for performance sensitive code is to write it in C and interact with that via a foreign function interface (FFI). But the language also has number of ongoing and defunct JIT compilers at various stages of completion and performance; e.g. [48]. JIT compilers have made attempts to solve many of the issues we discussed. For example, modern Javascript JITs cache lookup operations in a manner similar to [19], though such optimizations sometimes fail [32]. Trace-based dynamic type specialization [29] can help improve the performance of polymorphic code.

However, JIT compilers are highly constrained both in execution time and memory usage; so even though they have access to runtime program traces, which are very useful for optimization, they are often constrained in how much of the code they can legitimately examine and optimize. Further they are often battling with languages that sport an optimization-hostile semantics. As such the process is heavily driven by heuristics. These

heuristics are often not well-documented, leaving lay-programmers to guess at performance bottlenecks. The performance improvements often come at the cost of a more constrained semantics compared to the original language, forcing programmers to learn that in addition to the original language. Sharp declines in performance can occur when a JIT compiler fails to optimize a critical path [32, 2].

1.2 Our Approach

We take a different approach and focus on designing and building, from the ground up, a statically typed scripting language that yields the safety and performance benefits of a static type system while maintaining the terseness and flexibility of scripting languages.

1.2.1 Typed Scripting

Previous research from our lab [40, 39] has led to the development of TinyBang, a small, but flexible typed scripting language core. The semantics of TinyBang was carefully chosen to be statically analyzable, but be sufficiently expressive to encode common scripting language idioms; for example, TinyBang does *not* permit object mutation, but allows functional extension of objects.

A previous dissertation [39] described the process of designing typed scripting languages and provides a complete formalization of TinyBang including a proof of soundness and decidability for its type system. TinyBang is a core calculus. The dissertation also defines a higher level language, LittleBang, which supports common scripting features and encodes down to TinyBang. This division has obvious benefits for formalization. But it is

also advantageous from the point of view of an implementation: TinyBang is effectively a typed intermediate language on which implementation and low-level optimization efforts can be focused. The GHC Haskell compiler [34], takes a similar approach: the top-level language is first compiled down to Haskell Core, a type annotated intermediate representation, before optimization and further translations to low-level code.

1.2.2 From Safety to Speed

In the case of compilers for languages like C and C++, types provide a set of invariants that can be used to determine a gamut of necessary low-level details such as data representation, alignment and dispatch mechanics. For example, an optimizing compiler can decide to prefer register allocation for values of specific types. At a slightly higher level, in C++, the structure of an object's virtual table (used for method dispatch) is effectively determined by its type.

Languages like TinyBang encode a significant amount of information in its type system to ensure safety. However exploiting, or even extracting, that information in the context of a low-level implementation is non-trivial. Morrisett makes a similar observation in his thesis on type-directed compilation of ML [38]. However the types in typed scripting languages are significantly more complex than ML types due to the nature of the underlying languages. For example, TinyBang's type system is based on subtype constraints and capable of encoding union, intersection and recursive types while also being polymorphic like ML.

Our approach to this problem involves the concept of *adaptive* tags: specialized tags attached to values that represent how it is *used* in the current program. While many

functional language implementations use tagged values, the tag in those cases is usually based directly on the constructor of the value and therefore independent of the program.

This dissertation proposes a framework for implementing typed scripting languages based on adaptive tags. The framework addresses the issues discussed in Section 1.1.3. It is capable of assigning precise types to values, performing efficient type-based dispatch and lends itself to solving the data layout problem. The core part of our framework is a scheme for abstractly tracking the flow of deep structural types during analysis and then "replaying" it at runtime by efficiently encoding it in the form of tags. We expect that the framework is extensible to other issues, related to the implementation of typed scripting languages, that can benefit from this information.

In this dissertation, we focus on the specification of this framework which is composed of three related elements:

- An efficient operational semantics based on adaptive tags
- A type checker to collect tag information across the program
- And, an efficient implementation of the tag semantics

We will discuss adaptive tagging and the general framework in more detail in Chapter 3.

1.3 Outline

The rest of this thesis is organized as follows. Chapter 2 provides an overview of typed scripting languages with a particular focus on design. Chapter 3 discusses adaptive

tags and our tag framework informally.

The semantics of TinyBang is intuitive, but its type system is complex. Thus for clarity, we restrict the initial formalization to a simpler language we call TinyBang Core which omits features largely orthogonal to the adaptive tag semantics. Chapter 4 formalizes the language semantics and type system while Chapter 5 formalizes the adaptive tag framework including an alternate operational semantics for our language as well as a scheme for generating adaptive tags. In Chapter 6 we prove the equivalence between the two semantics. Chapter 7 discusses our proof-of-concept implementation while Chapters 8, 9 and 10 discuss related work, future work and conclusions respectively. In the appendix we present a formalization of the complete TinyBang language.

Chapter 2

Design of Typed Scripting Languages

In this chapter, we provide a brief overview of the design of typed scripting languages focusing on features of TinyBang and TinyBang Core. Naturally, this discussion is heavily influenced by TinyBang [40, 39]. We haven't yet defined the grammar of the two languages; so the examples in this chapter are written in a pseudo-ML dialect.

2.1 Duck Typing

Many scripting languages support a style of typing commonly called "duck typing". The primary idea is that a structure is identified by the behavior it exhibits. Consider the code in Listing 2.1 and Listing 2.2. The Java code requires an explicit declaration of an `Animal` interface and the implementation of each class must explicitly declare its conformance to the interface. The Python code is less verbose and will run correctly as long

```
interface Animal {
    public void walk()
}
class Duck implements Animal{
    public void walk(){
        System.out.println("Waddle");
    }
}
class Dog implements Animal{
    public void walk(){
        System.out.println("Run");
    }
}
class Mouse implements Animal{
    public void walk(){
        System.out.println("Scurry");
    }
}

Animal animal = ...
animal.walk()
```

Listing 2.1: Java Explicit Interface Declaration

```
class Duck:
    def walk(self): print "Waddle"
class Dog:
    def walk(self): print "Run"
class Mouse:
    def walk(self): print "Scurry"

animal = ...
animal.walk()
```

Listing 2.2: Python: Duck Typing

as the `animal` variable contains an object with a `walk` method. The additional paraphernalia required by Java becomes tedious when the interface is so simple.

In fact, Python makes extensive use of such implicitly defined interfaces, usually called *protocols*, in its api, relying on documentation to guide developers into doing the right thing[1]. For example, the notion of an `iterator` is essentially a protocol defined by the standard library.

[1]Though there has been a push for using abstract base classes [7] more recently

Statically typing such behavior requires our language to have the ability to infer the structure of an object from its usage. It also requires some notion of subtyping to support object polymorphism like in the example above.

The problem has been studied before in literature and a common choice to type check such scripting constructs is subtype constraint types [11]. Types in this system are represented as a bag of subtyping constraints. For example, the code in Listing 2.3 produces the initial constraint set $\{\texttt{int} <: \alpha_x, \alpha_x <: \alpha_y\}$ where α_x and α_y are type variables generated for `x` and `y` respectively. The left hand side of a constraint is called a *lower bound* and the right hand side is the *upper bound*. In TinyBang, upper bounds are always type variables and a lower bound type indicates a potential data flow to a program point. For example, the constraint $\texttt{int} <: \alpha_x$ indicates that an integer value can potentially be assigned to `x` at runtime.

Type checking is performed by applying a closure operation on the generated constraint set. If no inconsistencies are found, types can be "read off" the final constraint set; for instance, the type of `y` in the above example has the type $\alpha_y \backslash \{\texttt{int} <: \alpha_x, \alpha_x <: \alpha_y, \texttt{int} <: \alpha_y\}$; i.e. the type of `y` is the type variable α_y under the constraints $\{\texttt{int} <: \alpha_x, \alpha_x <: \alpha_y, \texttt{int} <: \alpha_y\}$.

Subtype constraints offer a great deal of power in terms of their ability to represent the types for complex language constructs. Consider the code in Listing 2.4. The result of the function `choose` is a string or an integer value depending on a runtime condition. Then the type of the variable `r` is union of `int` and `string`. This can be represented with subtype constraints as the set $\{\texttt{int} <: \alpha_r, \texttt{string} <: \alpha_r\}$. Informally we say that α_r is

```
x = 4
y = x
```

Listing 2.3: Constraint examples - 1

```
def choose():
    if randint(0, 1) == 0:
        return "Hello"
    else:
        return 42
r = choose()
```

Listing 2.4: Constraint examples - 2

the type `int` ∪ `string`. TinyBang's support for inferring such *ad-hoc* unions and tracking them across the program in a path sensitive manner is a cornerstone of its ability to model scripting behavior.

Subtype constraints have been studied extensively in the literature [10, 11, 43, 44, 55] and there is a natural inference algorithm that requires no type annotations. Its utility in the domain of typed scripting languages has been validated by previous attempts [28, 33, 16] to add type systems to existing scripting languages.

TinyBang has a number of features that fit well with a subtype constraint type system. In addition to duck typing, the language includes a specialized form of record concatenation and complex pattern matching capabilities, both of which have been shown to be amenable to type checking via a subtype constraint type model [43]. However given the complexity of TinyBang types, these solutions cannot usually be directly applied.

However choosing subtype constraints have downsides as well. In many languages, including object oriented languages like Java and functional languages like Haskell, type declarations serve two purposes: as a constraint on the type of a value and as a form of documentation for programmers. In the example in Listing 2.1, the interface `Animal` not

only provides an interface constraint to the typechecker, but also conveys "out-of-band" information regarding programmer intention.

Subtype constraints tend to produce very precise types; but they are not always what we want: for instance, it may deduce that a function requires as input, an iterable, ordered collection of characters with a replacement operation, when a human programmer would have considered `string` as the type of the input. Simplifying the bag-of-constraints to a human-readable type is a hard problem and attempts to solve it [42, 26] have met with limited success. However, this is less of a concern in the domain of scripting languages; scripting programmers do not work with types in the same way functional programmers do and simply spitting out a type error, even in a simplified form, is unlikely to be well received. We hope to address this issue by never directly showing types, but instead implementing an interactive type debugging approach. But that is beyond the scope of this dissertation.

2.2 Flexible data types

Scripting languages are often characterized by the extreme flexibility they allow in the structure of runtime data. For instance, both Python and Javascript objects allow adding and replacing fields and methods dynamically. This is a common programming and metaprogramming tool in these languages. Indeed the Javascript language itself relies on the ability to add fields dynamically to implement prototype inheritance.

As an example, the code in Listing 2.5 creates an object, calls a method on it, and then replaces the method with another implementation. This degree of mutation is very hard to statically type: the type of the object is *flow-sensitive*; i.e. it differs depending on

```
class Car:
    def __init__(self):
        self.type = "Mustang";
        self.year = 1969;
    def since(self):
        return self.year

def new_since(self):
    return str(self.year + 31)

obj = Car()
obj.since(); # returns number 1969
obj.since = types.MethodType(new_since, obj)
obj.since(); # returns string "2000"
```

Listing 2.5: Object surgery

where you are within the program flow. However, in practice, this functionally is used in a much more disciplined way. Typically, dynamic structure modification is only used during the object construction phase to add functionality to an object like `Car` in the previous example. This feature can be adequately modeled by allowing *functional* extensions of objects.

TinyBang chooses to model this behavior via a novel structure called an "onion". Onions, at their core, are *type-indexed* records [50]. For example the record `{42, "foo"}` contains two elements `42` and `"foo"` indexed by their type: i.e. it is equivalent to the record `{int = 42, string = "foo"}`. Projection is done via the type; `{42, "foo"}.int` evaluates to 42. New types are introduced via labels; `{NewInt = 24}` is a single-element record indexed by the type `NewInt`.

In fact, everything in TinyBang is an onion: `5` is simply a 1-ary onion indexed by the integer type. Further onions support concatenation via the `&` operator: `{42}&{"foo"}` yields `{42, "foo"}`. The `Car` constructor from Listing 2.5 can be represented as: `{} & {type = "Mustang"} & {year = 1969} & {since = fun self -> self.year}`.

The onion operation is asymmetric. When the two onions have overlapping fields, precedence is granted to the left side of the operator: for example, `{foo = 2, bar = 3} & {bar = True, moo = 5}` evaluates to `{foo = 2, bar = 3, moo = 5}`. The type system will correctly determine that the concatenation of two records of type `{foo:Int, bar:Int}` and `{bar:Bool, moo:Int}` in sequence produces a record of type `{foo:Int, bar:Int, moo:Int}`. This asymmetric concatenation behavior serves as the basis for the encoding of a number of object-oriented features like overloading and mixins in languages built on top of TinyBang.

2.3 Pattern Matching

In functional languages, pattern matching is a commonly available feature that allows for concise testing and destructuring of data at runtime. While pattern matching has started to make its way in to mainstream static programming languages [4], it is notably less prevalent in the scripting world, even though the intuitive semantics make it a natural fit for the domain [13, 14].

Consider, for example, Listing 2.6 which is part of the constructor of a fraction object. The code must deal with different input (types) and is organized as a series of tedious if-else clauses. Listing 2.7 rewrites the snippet in pseudo-Python using pattern matching; the latter offers a significant improvement in conciseness and readability.

Aspects of pattern matching such as, destructuring assignments, are part of popular scripting languages like Python and Ruby and have even made it in to Javascript [6]; but pattern matching as a core feature has not. Less popular and research-oriented languages (e.g. Thorn [13] and NewSpeak [31]) have, however, demonstrated the utility of the fea-

```
class Fraction(numbers.Rational):
  def __new__(cls, num=0, denom=None, *, _normalize=True):
    ....
    if type(num) is int:
      self._num = num
      self._denom = 1
      return self
    elif isinstance(num, numbers.Rational):
      self._num = num.num
      self._denom = num.denom
      return self
    elif isinstance(num, (float, Decimal)):
      # Exact conversion
      self._num, self._denom = num.as_integer_ratio()
      return self
    ....
```

Listing 2.6: Simplified Python code from fractions.py

```
class Fraction(numbers.Rational):
  def __new__(cls, num=0, denom=None, *, _normalize=True):
    ....
    self._num, self._denom = case num of
    | int -> (num, 1)
    | Rational(num, denom) -> (num, denom)
    | float | Decimal -> num.as_integer_ratio()
    ....
```

Listing 2.7: Code rewritten with pattern matching

ture in the domain. Thorn, in particular, offer powerful, well-integrated, pattern matching facilities that fully embrace the dynamic nature of the language. For example, the Thorn pattern `[x:string,y,$x]` matches a 3-element palindromic list whose first element is a string.

2.3.1 TinyBang and Compound Functions

Many functional languages allow the definition of functions by *cases*. Consider the Haskell example in Listing 2.8: the `map` function is defined as three cases, one each for each pattern, "glued" together.

```
last          :: [a] -> Maybe a
last  []      = Nothing
last  [h]     = Just h
last  (h:t)   = last t
```

Listing 2.8: Haskell function to find the last element of a list

TinyBang extends this notion and makes it first-class: each function has the form `fun p -> e`, where `p` is a pattern and `e` is the function body, and *compound* functions are created by explicitly combining them with the standard onion operator. The `last` function will be defined as follows: `let last = (fun [] -> Nothing) &(fun [h] -> Just h) &(fun (x:xs) -> last xs)`. This scheme is essentially a generalization of first class case clauses from MLPolyR [15].

When applying a compound function, matching is performed in order starting from the left-most function. TinyBang's type checker ensures that at least one of the cases match; otherwise it results in a compile-time error.

TinyBang is capable of assigning precise types to compound functions which is a major reason for its ability to model scripting language behavior in a type safe way. Not only are different cases allowed to return different types, the type system also tracks the relationship between input and output types.

Consider for example the compound function: `let f = fun (x:int) -> x & fun (y:char) -> y`. Standard type systems unify the input and output, resulting in a loss of "alignment" between the two. In such cases, the type of `f` would be inferred as: $(\texttt{int} \cup \texttt{char}) \texttt{->} (\texttt{int} \cup \texttt{char})$. The TinyBang type system (effectively) infers the *intersection* type $(\texttt{int} \texttt{->} \texttt{int}) \cap (\texttt{char} \texttt{->} \texttt{char})$. The function application `f 'x'` is ascribed the type `char` in TinyBang as opposed to $\texttt{int} \cup \texttt{char}$.

2.4 Call-site Polymorphism

Programming in a scripting language involves a certain amount of context sensitive reasoning: different invocations of the same function are treated differently depending on the context. This corresponds to a notion of parametric polymorphism in the domain of types.

TinyBang uses a *call-site* polymorphism model as opposed to the more common `let`-polymorphism models. With the latter, functions are abstracted when they are bound to variables and freshly instantiated every time the variable is used. Polymorphic instantiations are limited to the scope of the `let` statement, which makes it easier to reason about and compile. However this limits its expressiveness. For example, a function "loses" its polymorphism when it escapes the scope of the `let`-binding, which can be confusing to scripting programmers.

TinyBang's call-site polymorphism model is inspired by flow analysis [51, 55]. Every function is automatically given a polymorphic type. Polymorphic instantiation is delayed until the function is called. This offers a number of advantages: for example, functions retain their polymorphism even when being returned out of their scope. In particular, this allows TinyBang to infer much more precise types for compound functions than it can with a `let`-polymorphism model.

2.5 TinyBang Core

While the semantics of TinyBang is fairly simple, ensuring its safety is less so. Extending the TinyBang type system to support our tag framework and then proving

equivalence with our tag based semantics is non-trivial. Therefore, we restrict our initial formalization to a simpler language, TinyBang Core. Unlike TinyBang, which is powerful enough to serve as a typed intermediate language for scripting, TinyBang Core is much simpler. It was chiefly designed as a vehicle to demonstrate our approach to scripting language efficiency and to serve as a stepping stone to the more complex theory of TinyBang presented in the appendix.

TinyBang Core is a ML-style language with full type inference. It supports duck typing and ad-hoc unions like in TinyBang. However the language does not support onions and concatenation. While very useful for programming, the type-indexed nature of onions do not affect the tag framework significantly and concatenation is essentially a quick way to build up onion subtypes. Instead the language supports regular ML-style records. The language also eschews pattern matching functions and compound functions in the style of TinyBang, opting instead for a standard `case` statement. This simplifies the type and tag analysis considerably, but does not model the extensible matching behavior of onions. However, case branches are still allowed to return different types like in TinyBang and the type system can infer precise types for the case statement. As we will see in the appendix, our approach can be adapted to onion dispatch in a straightforward manner.

Chapter 3

Overview of the Tag Framework

This chapter provides an overview of our adaptive tag framework. We discuss the standard approaches to pattern match compilation and contrast with our adaptive tags model. We also informally discuss implementation strategies to make adaptive tags efficient.

A full formalization of the framework can be found in Chapter 5.

3.1 Pattern matching

Pattern matching is usually implemented with the help of *tags*: runtime data in the language have a few bits of tag information attached which is then inspected at runtime to perform matching. In ML-style languages, the tags typically correspond to the value's constructor. Further since destruction is also specified in terms of constructors, a *shallow* pattern match is simply a matter of comparing the runtime tag to a constructor from the pattern match. Consider the pattern match in Figure 3.1 written in a hypothetical ML-style language. The compiler assigns numeric tag values for the three constructors in the

```
data color = White | Black | Rgb Int Int Int
let c = Black in
case c of
  White -> 1
  Black -> 0
  Rgb _ _ _ -> -1
```

Listing 3.1: Shallow pattern matching in an ML-style language

```
data color = White | Black | Rgb Int Int Int
data option = Some a | None
let copt = Some Black in
case copt of
  Some(White) -> 0
  Some(Black) -> 1
  None -> 2
  _ -> -1
```

Listing 3.2: Deep pattern matching in an ML-style language

snippet. Patterns share the same constructor names and get the same numeric identifiers. To compile the pattern match expression, a table of three entries is built where each entry maps one of the patterns to the entry point of the corresponding code block. For example, the entry for `Black` points to the code block that returns 0. The table is then hard-coded in the generated code. Pattern matching against a data object is implemented at runtime by extracting the tag from the object, looking up in the table and jumping to the correct code block.

Patterns can be deeply nested. While this is a powerful feature, matching then becomes more complex. The example in Listing 3.2 requires multiple tests at runtime in order to ascertain the matching branch. Further, test ordering matters and naive approaches can result in performance penalties. The usual approach is to translate the pattern matching definition in to a "matching automata", either a backtracking automata [35] or more recently an optimized decision tree [36], organized around low-level checks.

It appears possible to extend the matching automata approach to work with languages that have record subtyping (say, by an approach similar to Clojure's map patterns [5]); however, it is unclear whether we can achieve performance similar to the original in the general case. Further this approach is unwieldy in the presence of ad-hoc unions and deep record subtyping and it is hard to extend the scheme to handle extensible cases or onion dispatch. So we consider an alternative approach in the next section.

3.2 Adaptive Tags

Consider the example in Listing 3.3. Let `choose` be a function that arbitrarily picks one of `x1` or `x2` at runtime. If it picked `x1`, the first case branch of the case statement fires and we get `True` as the result; otherwise the second case branch triggers and the result is `False`.

We could achieve fast pattern matching like in the previous section if `x4` had a unique tag corresponding to the pattern it could match, for example one of the following tags: `{c:{a:_},d:_}` or `{c:{b:_},d:_}`. However assigning such tags to `x4` requires non-trivial mechanics:

- Tags must be specialized for each `case` construct; computing the universe of tags statically requires knowledge of what objects flow to which pattern matches.
- Tags of this kind represent an expectation about the deep structure of a value; so tag information must be propagated appropriately to other points in the program.
- The tag assigned to `x4` depends on the tag assigned to `x3` at runtime. Tag assignment

```
1 let x1 = {a = 3} in
2 let x2 = {b = 4} in
3 let x3 = choose x1 x2 in
4 let x4 = {c = x3, d = 1} in
5 case x4 of
6   {c = {a = _}, d = _ } -> True
7   {c = {b = _}, d = _ } -> False
```

Listing 3.3: Pattern matching with subtyping

is more complex since it is no longer a fixed value determined by the constructor, but a function on the tags of the constituents.

In this dissertation we explore an adaptive tagging scheme which addresses these issues. In particular, we will address two questions:

- How do we generate adaptive tags automatically?
- How can we design and implement efficient operational semantics based these tags?

3.2.1 Tag Generation and Semantics

Our tag generation scheme relies on the observation that construction and destruction are dual operations; so by observing the process of destruction, it is possible to "learn" the sequence of construction. To this end, at compile time, we conservatively trace destruction operations in the program, generating potential tags along the way. At runtime, our operational semantics ensure that the tags are assigned as appropriate and pattern matching is a simple lookup similar to our description for Listing 3.1.

Tracing the program requires a static analysis capable of tracking data flows across the program and it must be capable of handling our language features. Fortunately the TinyBang Core type system is one such analysis, which can be further extended to generate

tags.

Once again consider the program in Listing 3.3. The inference assigns a type of `{a:Int}` ∪ `{b:Int}` to the variable `x3` and correspondingly for `x4` it infers the type: `{c:{a:Int},d:Int}` ∪ `{c:{b:Int},d:Int}`. To type check the pattern match, the checker must verify that a match occurs for each "leg" of this union type. This turns out to be the case since the type `{c:{a:Int},d:Int}` matches the first case branch and the type `{c:{b:Int},d:Int}` matches the second case branch. It then generates two potential tags, `{c:{a:_},d:_}` and `{c:{b:_},d:_}` for `x4`. The analysis also assigns `x3` two potential tags, `{a:_}` and `{b:_}` and so on. The system also records internally that for this `case` statement, `{c:{a:_},d:_}` matches the first branch and `{c:{b:_},d:_}` matches the second.

At runtime, tags are assigned to values at construction by consulting the set of potential tags computed at compile time. For example, when `x4` is constructed, we assign it a tag `{c:{a:_},d:_}` or `{c:{b:_},d:_}` depending on whether `x3` had a tag of `{a:_}` or `{b:_}`. Pattern matching is now straightforward: a dispatch table is created and reified based on the data collected during type checking; to match against `x4`, its tag is extracted and used in conjunction with the table to recover the address of a branch's code block and jump to that location.

Observe that the tag on `x4` is conditioned on the tag on `x3`. So it is necessary, at runtime, to inspect the tag on `x3`, in order to decide on the tag to assign to `x4`. This introduces an overhead that does not exist with standard shallow tagging schemes. However, there are some mitigating circumstances:

```
let x1 = {a = 3} in
let x2 = {b = 4} in
let x3 = choose x1 x2 in
let x4 = {c = x3, d = 1} in
let x5 = case x4 of
  {c : {a : _}, d : _ } -> True
  {c : {b : _}, d : _ } -> False
in
let x6 = case x4 of
  {d : int } -> 100
  {d : { } } -> { }
in
...
```

Listing 3.4: Pattern matching with multiple callsites

- In many programs, data *use* dominates data construction. Since tags are primarily assigned during construction, the upfront cost may be amortized over the uses.
- In many cases, only one tag applies to the data being constructed. We can then directly assign the tag to the value without any extra inspections.

But in the general case, this overhead does exist. Reducing its impact is critical to the practicality of our adaptive tag framework. We will discuss a number of approaches that reduce the overhead to manageable levels in Section 3.2.3 of the current chapter as well as in Chapters 7 and 9.

3.2.2 Scaling the Tag System

So far we have only considered programs with a single pattern match on a given data item. But regular programs have many data items and multiple match sites. So conceptually each data value is tagged with a *set of* tags in our model, each corresponding to a different pattern match site.

Consider the example in Listing 3.4 where `x4` is destructed multiple times. For

the second `case` statement, `x4` always matches the first branch. Along with the two tags from the earlier example, the system also generates an extra tag `{c:_,d:int}` for `x4`. It also records (internally) that a tag of the form `{c:_,d:int}` matches the first branch.

At runtime the system assigns *all* viable tags from this potential set of tags; `x4` always gets `{c:_,d:int}` and one of `{c:{a:_},d:_}` or `{c:{b:_},d:_}` (depending on the tags assigned to `x3`). Pattern matching works like before by reifying a dispatch table and consulting it at runtime.

3.2.3 Implementing Tag Sets

Unfortunately a naive implementation based on reifying a set of tags at runtime has a significant memory and performance cost. However, our tag framework has a set of features that we can exploit to design a much better scheme:

- Our analysis is capable of approximating the universe of tags for each variable in a given program: for example, in Listing 3.4 the tags for `x4` are drawn from the set containing `{c:{a:_},d:_}`, `{c:{b:_},d:_}` and `{c:_,d:int}`. At runtime, the set of tags associated with the variable is then an element of the power set of this universe. This makes it possible to represent each potential tag set by a numerical value.

- A tag represents a specific pattern match's expectation of the structure of the value being matched. Tags are assigned to a value during construction only when it meets this expectation. At runtime, a set of tags associated with a value cannot conflict since each tag represents some part of the same underlying structure. For example, the set of

tags associated with `x4` cannot contain both `{c:{a:_},d:_}` and `{c:{b:_},d:_}` simultaneously as they conflict. This makes it feasible to extend many properties of a single tag to sets of non-conflicting tags; specifically this enables extending pre-computed dispatch tables to sets of tags.

Together this allows us to represent tag operations, i.e. tag assignment and tag based dispatch, as mappings over integers. Consider the following numerical assignment for tag sets (corresponding to the code in Listing 3.4):

$$\texttt{x3} = \begin{cases} \texttt{\{a:\_\}} & = 1 \\ \texttt{\{b:\_\}} & = 2 \end{cases} \qquad \texttt{x4} = \begin{cases} \{\texttt{\{c:\{a:\_\}, d:\_\}}, \texttt{\{c:\_, d:int\}}\} & = 3 \\ \{\texttt{\{c:\{b:\_\}, d:\_\}}, \texttt{\{c:\_, d:int\}}\} & = 4 \end{cases}$$

Tag assignment for `x4` is then the following mapping from `x3`: $\{1 \mapsto 3, 2 \mapsto 4\}$ while the dispatch table for the first case is the mapping from tag to branch index: $\{3 \mapsto 0, 4 \mapsto 1\}$. The tables can be reified at runtime and lookups implemented via some standard integer hashing scheme.

However it is often, but not necessarily always, possible to do better for two reasons: we have control over the numbers assigned to tag sets and the tag assignment and dispatch mappings (like those above) are essentially *functions* on integers. It may then be possible to select a tag set numbering scheme such that the function induced by our mappings has an efficient-to-compute closed-form solution. For our example above, the tag assignment function for `x4` is simply $f(x) = x+2$ while the dispatch function is $f(x) = x-3$ both of which are very efficient at runtime and cost no extra memory.

One way to determine the numerical assignments is to use an SMT solver. We introduce variables corresponding to each tag set in the system and encode the mapping

between them as formulae. We also provide a set of known efficient functions (for example, a collection of generic binary formulas like $f(x, n) = x + n$, $f(x, n) = x$ `xor` n etc, where n is also treated as another variable). If the solver finds a set of numerical assignments that can be computed via one of the functions we provided, then we use the data from its generated model to encode tag assignments and dispatch. If not, we fall back to the table based scheme.

3.3 Porting to TinyBang

We have so far focused on the features of TinyBang Core. But we have also fully formalized the adaptive tag semantics for TinyBang, even though we do not have an implementation for it yet. The precise details are presented in the appendix. In this section, we will briefly discuss the process of adapting the tag system to the full TinyBang language.

3.3.1 Onions and Compound Functions

Like records in TinyBang Core, onions are the primary compound data type in TinyBang. To track onion structures, we introduce *onion tags*. An onion tag is a (potentially incomplete) representation of the deep structure of an onion value.

As discussed in Section 2.3.1, TinyBang also supports compound functions with an asymmetric dispatch semantics. Building an efficient dispatch scheme for compound functions is significantly more tricky than for `case` statements in TinyBang Core.

Consider the code in Listing 3.5. Once again let `choose` arbitrarily select one of its inputs. Then, at runtime, the compound function `fn` is an onion with either `f1` or `f2`

on its left hand side and either `f3` or `f4` on the right. The code that executes when `fn` is applied to `v` depends on the choices previously made by `choose` as well as the structure of the argument `v`. In other words, compound function dispatch is, effectively, a dual dispatch scheme where the dispatch depends on the dynamic types of both the receiving compound function as well as the argument.

At a high level, the process of generating adaptive tags and collecting dispatch information is still the same as in TinyBang Core. We conservatively track destructions in the program, emitting tags along the way, for both the compound function and the argument. When recording dispatch decisions, we track tags from both sides of the application.

TinyBang offers a richer pattern semantics than TinyBang Core. For example it supports conjunction patterns and bindings. The former does not affect the adaptive tag semantics significantly, except for adding to the complexity of the type system. However, the latter is somewhat challenging due to the nature of compound function dispatch. We handle bindings in TinyBang by conservatively approximating possible pattern matches in the program and pre-computing "paths" for each one. A path, in this case, is an abstract representation of the location of a particular value inside an onion. The details are presented in the appendix.

3.3.2 Data Layout

Consider TinyBang Core's projection operator. Generating code to implement it requires the knowledge of where in the runtime structure the data corresponding to the label resides. In a language with ad-hoc unions and subtypes like ours, this is not straightforward since all data types that reach that point have the specific label, but they are not necessarily

```
let f1 = fun p1 -> ...
let f2 = fun p2 -> ...
let f3 = fun p3 -> ...
let f4 = fun p4 -> ...

let x1 = choose f1 f2
let x2 = choose f3 f4
let fn = x1 & x2

let v = ...
let res = fn v
```

Listing 3.5: Compound Function Example

in the same spot. Thus the generated code must deal with potentially different layouts: it must first detect what the layout of a value is at runtime and then extract the data from the correct spot.

A key facet of tags is that they are an abstract representation of the structure of a value. Since runtime tag sets cannot conflict, projection can then be implemented by pre-computating a table keyed on tag sets that map to the location of a particular label. Now projection is a simple, shallow operation and can be implemented in many ways; however the key insight here is that tag sets can be used to detect the layout of data at runtime.

Data layout is not in the scope of the current dissertation. Nonetheless it is a critical aspect of performance, especially in scripting languages with fluid runtime data [20]. Within the context of TinyBang, onions bring up a number of data layout questions which we discuss in Section 9.2.

Chapter 4

Formalization of TinyBang Core

In this chapter, we formalize the basic semantics of TinyBang Core. While the language is modeled on TinyBang, we have chosen to trade-off power, for example Onions and Compound Functions, for clarity and simplicity. The language of TinyBang Core is thus more ML-like. In section 4.2 we define its grammar. Section 4.4 defines the operational semantics for the language and Section 4.5 defines the type system for it. We continue our formalization in Chapter 5 where we define an alternate, tag-based semantics for TinyBang Core. Finally, in Chapter 6, we present the proof of eqivalence between the two systems.

4.1 Notation

We make use of a number of standard notational devices in our formalization. In this section, we briefly describe the common ones.

Notation 4.1 (Basic Notation)**.** Using v and k to represent an arbitrary grammar constructs, we define the following:

- The list $[v_1, \ldots, v_n]$ of length n can be shortened as $\overset{n}{\overrightarrow{v}}$. The length is elided when unnecessary: $\overrightarrow{v}$.
- The operation $\overrightarrow{v'} \parallel \overrightarrow{v''}$ indicates the concatenation of the two lists $\overrightarrow{v'}$ and $\overrightarrow{v''}$.
- For convenience, we extend the standard set builder notation to support list comprehensions: e.g. $[v \mid v \in V]$. The ordering of the resultant list is non-deterministic.
- We also extend the membership operator to lists: we write $v \in \overset{n}{\overrightarrow{v}}$ to indicate that a particular value v is present in the list $\overset{n}{\overrightarrow{v}}$.
- In rare cases, we use $_\square$ to indicate index positions: for example $\overrightarrow{v_\square / v'}$ is a shorthand for $[v_0/v', \ldots, v_n/v']$.
- Sets $\{v_1, \ldots, v_n\}$ of cardinality n are shortened as $\overset{n}{\overbrace{v}}$ with n elided if unimportant.
- We usually treat dictionaries $\{k_1 \mapsto v_1, \ldots, k_n \mapsto v_n\}$ as a sets of mappings: $\overset{n}{\overbrace{k_\square \mapsto v_\square}}$. We elide the n and $_\square$ when they are unnecessary.
- We extend the membership operator to dictionaries. Since dictionaries are treated as sets, we use $k \mapsto v \in Q$, where Q is a dictionary, to indicate that a particular mapping exists in the dictionary. We override this notation to test for keys: we write $k \in Q$ to indicate that the key k is present in the dictionary.

4.2 Grammar

Our grammar is defined in an A-normal form for two reasons:

$$
\begin{array}{lcll}
e & ::= & \overrightarrow{s} & \textit{expressions} \\
s & ::= & x \texttt{ = } r & \textit{clauses} \\
r & ::= & v \mid x \mid x\ x \mid x.l \mid \texttt{case}\ x\ \texttt{of}\ \overrightarrow{p \rightsquigarrow f} & \textit{rhs} \\
v & ::= & \mathbb{Z} \mid f \mid \texttt{\{...,}\ l\texttt{=}x\ \texttt{,...\}} & \textit{values} \\
f & ::= & x \texttt{ -> } e & \textit{functions} \\
p & ::= & \texttt{int} \mid \texttt{\{...,}\ l\texttt{:}p\ \texttt{,...\}} & \textit{patterns} \\
x & & & \textit{variables} \\
l & & & \textit{field labels}
\end{array}
$$

Figure 4.1: TinyBang Core ANF Grammar

- TinyBang Core is based on TinyBang which is essentially a typed intermediate language. A-normal form is a good choice for both analysis and code generation at that level.

- This aligns well with our type theory. Constraints in our type system can be interpreted as statements of facts about points in a program. By labeling points explicitly and representing the program as a flat list of clauses, the form of our program corresponds exactly with the form of the constraints in the type system. We will use this correspondence between the two systems to prove the soundness of our type system in Section 4.6

A formal description of the grammar of TinyBang Core is given in Figure 4.1.

4.3 Well-formed Expressions

In order to define our semantics, we need a notion of *well-formed* expressions over the TinyBang Core grammar. This is defined as follows:

Definition 4.2 (Well-formed Expressions)**.** An expression e is considered well-formed if it meets the following criteria:

$$\begin{array}{rcll} E & ::= & \overrightarrow{x = v} & \textit{environment} \\ P^+, P^- & ::= & \overline{p} & \textit{pattern sets} \end{array}$$

Figure 4.2: Extensions to the TinyBang Core grammar

- e is closed.
- Each variable is bound in at most one clause.

For the rest of this document we only consider well-formed expressions unless stated otherwise.

4.4 Operational Semantics

We define the small-step operational semantics of TinyBang Core in this section. Our semantics is defined over a slightly augmented TinyBang Core grammar. The additional grammar non-terminals are defined in Figure 4.2.

It is worth noting that the grammar of the environment E is a subset of that of e. For convenience, we also define a lookup function on well-formed environments:

Definition 4.3 (TinyBang Core Environment Lookup)**.** Let E be a well-formed. Then $E(x) = v$ if and only if $x = v \in E$.

We must first define a number of auxiliary relations before proceeding to the actual semantics.

4.4.1 Compatibility

Pattern matching is at the heart of our operational semantics. The compatibility relation is a formalization of the pattern matching process. The mechanics of pattern

$$l_i \in \textsc{Fields}(\texttt{\{}\ldots\texttt{,}\ l_i\texttt{=}x_i\ \texttt{,}\ldots\texttt{\}})$$
$$l_i \in \textsc{Fields}(\texttt{\{}\ldots\texttt{,}\ l_i\texttt{:}p_i\ \texttt{,}\ldots\texttt{\}})$$

$$\textsc{IsRecord}(v) \text{ iff } v \text{ has the form } \texttt{\{}\ldots\texttt{\}}$$
$$\textsc{IsRecord}(p) \text{ iff } p \text{ has the form } \texttt{\{}\ldots\texttt{\}}$$

$$\texttt{\{}\ldots\texttt{,}\ l\texttt{=}x\ \texttt{,}\ldots\texttt{\}}\texttt{.}l \triangleq x$$
$$\texttt{\{}\ldots\texttt{,}\ l\texttt{:}p\ \texttt{,}\ldots\texttt{\}}\texttt{.}l \triangleq p$$

$$P\texttt{.}l \triangleq \{p\texttt{.}l \mid p \in P\}$$
$$lP \triangleq \{\texttt{\{}\,l\texttt{:}p\,\texttt{\}} \mid p \in P\}$$

Figure 4.3: Extended notation for TinyBang Core compatibility

matching is fairly intuitive. But we follow TinyBang's lead and define the compatibility relation in a somewhat non-standard way: we use the 4-ary predicate, $x\backslash E \sim \frac{P^+}{P^-}$ which is satisfied when the variable x under environment E simultaneously matches all patterns in P^+ and fails to match all patterns in P^-. We do this for two reasons:

- It is beneficial to the soundness proofs for the operational semantics to be strictly aligned with the type system.
- The TinyBang type system must deal with subtle issues related to *union alignment* [25] which can be circumvented by a simultaneous matching approach. While TinyBang Core is not subject to this issue to the same degree in the absence of conjunction patterns, we would like our formal relations to remain adaptable.

We first define some notation, for convenience, in Figure 4.3. Compatibility is then defined as follows:

Definition 4.4. We let $x\backslash E \sim \frac{P^+}{P^-}$ and $v\backslash E \sim \frac{P^+}{P^-}$ to be the mutually defined relations satisfying the rules in Figure 4.4.

$$\textsc{Leaf}\ \frac{}{x \backslash E \sim \frac{\emptyset}{\emptyset}} \qquad \textsc{Value Selection}\ \frac{x = v \in E \qquad v \backslash E \sim \frac{P^+}{P^-}}{x \backslash E \sim \frac{P^+}{P^-}}$$

$$\textsc{Record}\ \frac{v = \{\ldots,\ l_i = x_i\ ,\ldots\} \quad \forall p \in P^+.\ \textsc{Fields}(p) \subseteq \textsc{Fields}(v) \qquad \forall i. x_i \backslash E \sim \frac{P^+.l_i}{P_i^-} \qquad \forall p \in P^+.\ \textsc{IsRecord}(p)}{v \backslash E \sim \frac{P^+}{\bigcup_i l_i P_i^-}}$$

$$\textsc{Record - Absent Field}\ \frac{v \backslash E \sim \frac{P^+}{P^-} \qquad \textsc{IsRecord}(v) \qquad l \notin \textsc{Fields}(v)}{v \backslash E \sim \frac{P^+}{\{l : p\} \cup P^-}}$$

$$\textsc{Record - Extension}\ \frac{v \backslash E \sim \frac{P^+}{p \cup P^-} \quad \textsc{IsRecord}(v) \qquad p = \{\ldots,\ l_i : p_i\ ,\ldots\} \qquad p' = \{\ldots,\ l_i : p_i,\ \ldots,\ l'' : p''\}}{v \backslash E \sim \frac{P^+}{p' \cup P^-}}$$

$$\textsc{Record - Non-Record Pattern}\ \frac{v \backslash E \sim \frac{P^+}{P^-} \qquad \textsc{IsRecord}(v) \qquad \neg\textsc{IsRecord}(p')}{v \backslash E \sim \frac{P^+}{p' \cup P^-}}$$

$$\textsc{Integer}\ \frac{v \in \mathbb{Z} \qquad P^+ = \{\texttt{int}\} \qquad \{\texttt{int} \in P^-\} = \emptyset}{v \backslash E \sim \frac{P^+}{P^-}}$$

Figure 4.4: TinyBang Core compatibility relation

The compatibility rules are relatively straightforward. Integer values match integer patterns as long as they do not appear in the negative pattern set as well. Similarly record values match record patterns when both agree on the fields and recursively on the field's contents.

Handling record patterns in the negative position is slightly more complicated. To show that a subject value fails to match a record pattern, it suffices for one of the following to hold:

- The subject value is not a record.
- The subject value is missing a field compared the the pattern.
- Or finally, the subject value has the same shallow structure as the record pattern, but recursively fails to match on at least one of its fields.

Compatibility has rules to deal with all of these cases: the first two are handled directly by rules; for the final case above, the RECORD EXTENSION rule specifies that as long as a "shorter" record pattern (i.e. one with fields removed, but never empty) can be shown to anti-match the subject value, the longer pattern also fails to match; this allows us to non-deterministically test the pattern fields for a recursive mismatch.

4.4.2 Small-Step Evaluation

We can now define the small step semantics for TinyBang Core: $e \longrightarrow^* e'$. Our operational semantics is neither precisely environment based or substitution based, but has aspects of both: variables are looked up in the environment while function bodies are inlined. We choose this model to align better with our type system.

Since variables are looked up from the environment, we must ensure that the latter remains well-formed throughout evaluation. In particular, we must *freshen* function bodies and case bodies before the clauses are inlined during evaluation. For this purpose, we utilize a deterministic freshening function $\boldsymbol{\alpha}(x, x')$ (where x corresponds to a call-site and x' is a variable to be renamed).

Definition 4.5 (Freshening Function)**.** We define $\boldsymbol{\alpha}(x, x')$ to be a function with three properties: the function is injective, its co-domain is disjoint from the variables in the original

program and there are no cycles in any variable's lineage. We also overload $\boldsymbol{\alpha}(x, v)$ to indicate the freshening of all variables bound in v.

A technical note: the deterministic aspect is not strictly necessary in monomorphic TinyBang Core as opposed to TinyBang where it is instrumental in aligning fresh variables with poly-instantiated type variables. But, as before, we try to ensure that the basic relations have the same "shape" in both systems.

We need another auxiliary definition before the operational semantics. In the small step relation, we sometimes need to examine the last bound variable in an expression, for example, to fetch the result of a function application. We define a small function for the purpose:

Definition 4.6 (Return Variable)**.** We let $\text{RV}(e) = x_n$ if and only if $e = [x_1 = r_1, \ldots, x_n = r_n]$

The small step relation is then defined as follows:

Definition 4.7 (TinyBang Core Small Step Semantics)**.** We let $e \longrightarrow^1 e'$ be the relationship satisfying the rules in Figure 4.5.

A partially evaluated expression, in general, has the form: $E \parallel e$. Small step evaluation proceeds by acting on the "head" of the unevaluated part, producing a new expression $E' \parallel e'$. For example, if e has the form $[x = x'] \parallel e'$, the LOOKUP rule applies, which *replaces* the head clause with a fully evaluated clause; with the rest of the expression remains unchanged.

It is convenient to define the stuck states of this evaluation explicitly:

Definition 4.8. An expression is *stuck* if and only if it satisfies at least one of the rules in Figure 4.6.

$$\textsc{Variable Lookup}\quad \frac{E(x_2) = v}{E \parallel x_1 \texttt{=} x_2 \parallel e \longrightarrow^1 E \parallel x_1 \texttt{=} v \parallel e}$$

$$\textsc{Application}\quad \frac{E(x_2) = x_4' \texttt{->} e_1' \qquad E(x_3) = v \qquad \boldsymbol{\alpha}(x_1, x_4' \texttt{->} e_1') = x_4 \texttt{->} e_1}{E \parallel x_1 \texttt{=} x_2\ x_3 \parallel e_2 \longrightarrow^1 E \parallel x_4 \texttt{=} v \parallel e_1 \parallel x_1 = \text{RV}(e_1) \parallel e_2}$$

$$\textsc{Projection}\quad \frac{E(x_2) = \texttt{\{}\ldots\texttt{,}\ \ l\texttt{=}x_3\ \ \texttt{,}\ldots\texttt{\}} \qquad E(x_3) = v}{E \parallel x_1 \texttt{=} x_2\texttt{.}l \parallel e \longrightarrow^1 E \parallel x_1 \texttt{=} v \parallel e}$$

$$\textsc{Pattern Match}\quad \frac{v = E(x_2) \qquad i \leq n \qquad v \backslash E \sim \frac{\{p_i\}}{\{p_j \mid j<i\}} \qquad \boldsymbol{\alpha}(x_1, f_i) = x_i \texttt{->} e_i}{E \parallel x_1 \texttt{=} \texttt{case}\ x_2\ \texttt{of}\ \overrightarrow{{}^{n} p_\square \rightsquigarrow f_\square} \parallel e \longrightarrow^1 E \parallel x_i \texttt{=} v \parallel e_i \parallel x_1 \texttt{=} \text{RV}(e_i) \parallel e}$$

Figure 4.5: TinyBang Core operational semantics

$$\textsc{Application Failure}\quad \frac{E(x_2) = v \qquad v \text{ is not of the form } x \texttt{->} e'}{E \parallel x_1 \texttt{=} x_2\ x_3 \parallel e_2 \text{ is stuck}}$$

$$\textsc{Non-Record Projection}\quad \frac{E(x_2) = v \qquad \neg\textsc{IsRecord}(v)}{E \parallel x_1 \texttt{=} x_2\texttt{.}l \parallel e \text{ is stuck}}$$

$$\textsc{Projection Failure}\quad \frac{E(x_2) = v \qquad l \notin \textsc{Fields}(v)}{E \parallel x_1 \texttt{=} x_2\texttt{.}l \parallel e \text{ is stuck}}$$

$$\textsc{Pattern Match Failure}\quad \frac{x' \backslash E \sim \frac{\emptyset}{\{p_1 \ldots p_n\}}}{E \parallel x \texttt{=} \texttt{case}\ x'\ \texttt{of}\ \overrightarrow{{}^{n} p_\square \texttt{->} e_\square} \parallel e \text{ is stuck}}$$

Figure 4.6: TinyBang Core stuck states

The process of evaluation is simply a sequence of small step evaluations which we define formally:

Definition 4.9 (Multiple Small Steps)**.** We let $e_0 \longrightarrow^* e_n$ if and only if $e_0 \longrightarrow^1 e_1 \longrightarrow^1 \ldots \longrightarrow^1 e_n$.

4.5 Type System

Our type system is based on subtype constraints. But our formulation is somewhat unusual:

- The standard approach to presenting a type inference system is to give a set of type checking rules and a separate type inference algorithm. Instead we present a simpler formalism: type checking in TinyBang Core is performed by first generating an initial constraint set from the program, computing the fixed point of a constraint closure relation and then checking whether the final closed constraint set is *consistent*.

- Proving soundness by the standard approach of "progress and preservation" is complex for a constraint type system. Instead we exploit the correspondence between TinyBang Core's expression and type systems to prove soundness by *simulation*, an approach commonly seen in the context of abstract interpretation [37, 49].

We begin our formal presentation in the next section by specifying the type grammar and a set of rules to translate a program expression in to the initial set of constraints. We then present type system rules for compatibility and constraint closure. A formal definition of type checking based on a notion of constraint set consistency is presented at the end.

4.5.1 Type Grammar and Initial Alignment

The grammar for types in TinyBang Core is presented in Figure 4.7. It has a close correspondence with the expression grammar: τ corresponds to v, program clauses corresponds to type constraints and so forth. Types in the language are effectively abstractions over values. Patterns have a direct analogue in the type system. A *sequence* of program clauses are represented by an *unordered* set of constraints in the type grammar such that each constraint is an abstraction of a program clause.

$$
\begin{array}{rcll}
C & ::= & \overline{c} & \textit{expressions} \\
c & ::= & b <: \alpha & \textit{constraints} \\
b & ::= & \tau|^{\Pi^+}_{\Pi^-} \mid \alpha \mid \alpha\ \alpha \mid \alpha.l \mid \texttt{case}\ \alpha\ \texttt{of}\ \overrightarrow{\pi \rightsquigarrow \phi} & \textit{lower bounds} \\
t & ::= & \alpha \backslash C & \textit{constrained types} \\
\phi & ::= & \alpha\ \texttt{->}\ t & \textit{function types} \\
\tau & ::= & \texttt{int} \mid \phi \mid \texttt{\{}\ldots\texttt{,}\ l\texttt{:}\alpha\ \texttt{,}\ldots\texttt{\}} & \textit{types} \\
\Pi^+, \Pi^- & ::= & \overline{\pi} & \textit{pattern sets} \\
\pi & ::= & \texttt{int} \mid \texttt{\{}\ldots\texttt{,}\ l\texttt{:}\pi\ \texttt{,}\ldots\texttt{\}} & \textit{patterns} \\
\alpha & & & \textit{type variables} \\
l & & & \textit{field labels}
\end{array}
$$

Figure 4.7: TinyBang Core type grammar

TinyBang Core's type system is *path sensitive*. When type checking a `case` statement, for each relevant branch, the system *refines* the type of the input to conform to the requirements imposed by the preceding pattern match. This allows for a more precise determination of the output type and consequently the whole case statement. TinyBang Core achieves this by utilizing *filtered types*, an idea introduced in the TinyBang type system. A filtered type has the form: $\tau|^{\Pi^+}_{\Pi^-}$ and represents a base type τ restricted to cases where it matches patterns in Π^+ and fails to match those in Π^-. Filtered types are introduced during the type analysis of `case` branches and occur as lower bounds on type variables.

A note on notation: if both pattern sets are empty, the type is considered unfiltered and we often elide Π^+ and Π^-.

We can then define the initial derivation of constraints for a program as follows:

Definition 4.10 (Initial Alignment)**.** Let $[\![x]\!]_{\mathrm{V}}$ be an injective function from variables to type variables. Then the initial set of constraints for an expression e is given by $[\![e]\!]_{\mathrm{E}}$ where $[\![-]\!]_{\mathrm{E}}$ is defined in Listing 4.8.

The unordered nature of constraint sets introduces a minor wrinkle: the RV func-

$$
\begin{array}{lcl}
[\![\overrightarrow[n]{s}]\!]_{\mathrm{E}} & = & \alpha_n \backslash \underbrace{c}^{n} \text{ where } \forall i \leq n. [\![s_i]\!]_{\mathrm{E}} = \alpha_i \backslash c_i \\
\\
[\![x = v]\!]_{\mathrm{E}} & = & [\![x]\!]_{\mathrm{V}} \backslash [\![v]\!]_{\mathrm{E}}|^{\emptyset}_{\emptyset} <: [\![x]\!]_{\mathrm{E}} \\
[\![x_1 = x_2]\!]_{\mathrm{E}} & = & [\![x_1]\!]_{\mathrm{V}} \backslash [\![x_2]\!]_{\mathrm{V}} <: [\![x_1]\!]_{\mathrm{V}} \\
[\![x_1 = x_2\ x_3]\!]_{\mathrm{E}} & = & [\![x_1]\!]_{\mathrm{V}} \backslash [\![x_2]\!]_{\mathrm{V}}\ [\![x_3]\!]_{\mathrm{V}} <: [\![x_1]\!]_{\mathrm{V}} \\
[\![x_1 = x_2\,.\,l]\!]_{\mathrm{E}} & = & [\![x_1]\!]_{\mathrm{V}} \backslash [\![x_2]\!]_{\mathrm{V}}\,.\,l <: [\![x_1]\!]_{\mathrm{V}} \\
[\![x_1 = \texttt{case}\ x_2\ \texttt{of}\ \overrightarrow[n]{p_{\square} \texttt{->} f_{\square}}]\!]_{\mathrm{E}} & = & [\![x_1]\!]_{\mathrm{V}} \backslash \texttt{case}\ [\![x_2]\!]_{\mathrm{V}}\ \texttt{of}\ \underbrace{[\![p_{\square}]\!]_{\mathrm{P}} \texttt{->} [\![f_{\square}]\!]_{\mathrm{E}}}^{n} <: [\![x_1]\!]_{\mathrm{V}} \\
\\
[\![\mathbb{Z}]\!]_{\mathrm{E}} & = & \texttt{int} \\
[\![x \texttt{->} e]\!]_{\mathrm{E}} & = & [\![x]\!]_{\mathrm{V}} \texttt{->} [\![e]\!]_{\mathrm{E}} \\
[\![\{\ldots,\ l\texttt{=}x\ ,\ldots\}]\!]_{\mathrm{E}} & = & \{\ldots,\ l\texttt{:}[\![x]\!]_{\mathrm{V}}\ ,\ldots\} \\
\\
[\![\texttt{int}]\!]_{\mathrm{P}} & = & \texttt{int} \\
[\![\{\ldots,\ l\texttt{:}p\ ,\ldots\}]\!]_{\mathrm{P}} & = & \{\ldots,\ l\texttt{:}[\![p]\!]_{\mathrm{P}}\ ,\ldots\}
\end{array}
$$

Figure 4.8: Initial constraint derivation

tion, used to fetch the last bound variable in an expression, cannot be modeled directly in the type system; instead we opt to represent the type of e by the constrainted type $\alpha \backslash C$ where α is the type-theoretic analogue of $\textsc{rv}(e)$.

4.5.2 Type Compatibility

Type compatibility is analogous to the compatibility definition from Section 4.4.1. For convenience, we define some notation in Figure 4.9. Type compatibility is then defined in a fashion similar to the compatibility definition for the operational semantics:

Definition 4.11. We let $\alpha \backslash C \sim \frac{\Pi^+}{\Pi^-}$ and $\tau \backslash C \sim \frac{\Pi^+}{\Pi^-}$ be the mutually defined relations satisfying the rules in Figure 4.10.

The type compatibility rules are nearly all identical to the compatibility rules in Figure 4.4 with appropriate syntactic substitutions. The only exception is the Type Selection rule (which corresponds to the Value Selection rule in the operational semantics).

$$l_i \in \textsc{Fields}(\{\ldots,\ l_i{=}\alpha_i\ ,\ldots\})$$
$$l_i \in \textsc{Fields}(\{\ldots,\ l_i{:}\pi_i\ ,\ldots\})$$

$\textsc{IsRecord}(\tau)$ iff τ has the form $\{\ldots\}$
$\textsc{IsRecord}(\pi)$ iff π has the form $\{\ldots\}$

$$\{\ldots,\ l{=}\alpha\ ,\ldots\}.l \triangleq \alpha$$
$$\{\ldots,\ l{:}\pi\ ,\ldots\}.l \triangleq \pi$$

$$\Pi.l \triangleq \{\pi.l \mid \pi \in \Pi\}$$
$$l\Pi \triangleq \{\{\, l{:}\pi \,\} \mid \pi \in \Pi\}$$

Figure 4.9: Extended notation for TinyBang Core type compatibility

Consider the compatibility proof for $\alpha\backslash C \sim \frac{\Pi_1^+}{\Pi_1^-}$. Unlike values, types, being conservative approximations, can have unions; in the constraint system, this implies that a type variable can have multiple lower bounds. The above rule selects a particular lower bound and verifies its compatibility. However lower bounds are filtered types of the form: $\tau|_{\Pi^-}^{\Pi^+}$; we must ensure that this restricted type matches Π_1^+ and fails to match Π_1^-. The Type Selection rule solves this by incorporating the patterns in to the induction; i.e. by checking the compatibility of $\alpha\backslash C \sim \frac{\Pi_1^+ \cup \Pi^+}{\Pi_1^- \cup \Pi^-}$.

4.5.3 Constraint Closure

The constraint closure is responsible for the propogation of constraints via a process that abstractly models the execution of the program. The relation itself is analogous to the small step relation from the operational semantics.

Definition 4.12 (TinyBang Core Constraint Closure)**.** We let $C \Longrightarrow^1 C'$ be the relationship satisfying the rules in Figure 4.11.

Each constraint closure rule corresponds to a rule in the small step operational

$$\textsc{Leaf}\quad \frac{}{\alpha \backslash C \sim \frac{\emptyset}{\emptyset}} \qquad \textsc{Type Selection}\quad \frac{\tau|_{\Pi_1^-}^{\Pi_1^+} <: \alpha \in C \qquad \tau \backslash C \sim \frac{\Pi_1^+ \cup \Pi_2^+}{\Pi_1^- \cup \Pi_2^-}}{\alpha \backslash C \sim \frac{\Pi_2^+}{\Pi_2^-}}$$

$$\textsc{Record}\quad \frac{\tau = \{\ldots,\ l_i : \alpha_i\ ,\ldots\} \qquad \forall \pi \in \Pi^+.\ \textsc{Fields}(\pi) \subseteq \textsc{Fields}(\tau) \qquad \forall i.\alpha_i \backslash C \sim \frac{\Pi^+.l_i}{\Pi_i^-} \qquad \forall \pi \in \Pi^+.\ \textsc{IsRecord}(\pi)}{\tau \backslash C \sim \frac{\Pi^+}{\bigcup_i l_i \Pi_i^-}}$$

$$\textsc{Record - Absent Field}\quad \frac{\tau \backslash C \sim \frac{\Pi^+}{\Pi^-} \qquad \textsc{IsRecord}(\tau) \qquad l \notin \textsc{Fields}(\tau)}{\tau \backslash C \sim \frac{\Pi^+}{\{l : \pi\} \cup \Pi^-}}$$

$$\textsc{Record - Extension}\quad \frac{\tau \backslash C \sim \frac{\Pi^+}{\pi \cup \Pi^-} \qquad \textsc{IsRecord}(\tau) \qquad \pi = \{\ldots,\ l_i : \pi_i\ ,\ldots\} \qquad \pi' = \{\ldots,\ l_i : \pi_i,\ \ldots,\ l'' : \pi''\}}{\tau \backslash C \sim \frac{\Pi^+}{\pi' \cup \Pi^-}}$$

$$\textsc{Record - Non-Record Pattern}\quad \frac{\tau \backslash C \sim \frac{\Pi^+}{\Pi^-} \qquad \textsc{IsRecord}(\tau) \qquad \neg \textsc{IsRecord}(\pi')}{\tau \backslash C \sim \frac{\Pi^+}{\pi' \cup \Pi^-}} \qquad \textsc{Integer}\quad \frac{\Pi^+ = \{\texttt{int}\} \qquad \texttt{int} \notin \Pi^-}{\texttt{int} \backslash C \sim \frac{\Pi^+}{\Pi^-}}$$

Figure 4.10: TinyBang Core type compatibility relation

semantics. In fact most are simply renderings of the latter in the type grammar with minor exceptions:

- The closure rules lacks a type analogue to the freshening function $\boldsymbol{\alpha}(x, x')$. TinyBang Core is currently monomorphic; so there is no renaming of type variables.

- The Pattern Match rule in the type system is slightly more complicated than the one for the operational semantics. Since type lower bounds are filtered types, the compatibility checks must account for both the existing filters and the new positive/negative pattern sets similar to the Type Selection rule in 4.5.2. Further since TinyBang Core is path sensitive, the generated binding constraint must account

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

$$\textsc{Application}\quad \frac{\{\alpha_1\ \alpha_2 <: \alpha_3, \tau_1|^{\Pi_1^+}_{\Pi_1^-} <: \alpha_1, \tau_2|^{\Pi_2^+}_{\Pi_2^-} <: \alpha_2\} \subseteq C \qquad \tau_1 = \alpha_4 \texttt{->} \alpha' \backslash C'}{C \Longrightarrow^1 C \cup C' \cup \{\tau_2|^{\Pi_2^+}_{\Pi_2^-} <: \alpha_4, \alpha' <: \alpha_3\}}$$

$$\textsc{Projection}\quad \frac{\{\alpha_1.l <: \alpha_2, \texttt{\{...,}\ l\texttt{:}\alpha_3\ \texttt{,...\}}|^{\Pi_1^+}_{\Pi_1^-} <: \alpha_1, \tau|^{\Pi_2^+}_{\Pi_2^-} <: \alpha_3\} \subseteq C \qquad \tau\backslash C \sim \frac{\Pi_2^+ \cup \Pi_1^+.l}{\Pi_2^-}}{C \Longrightarrow^1 C \cup \{\tau|^{\Pi_2^+ \cup \Pi_1^+.l}_{\Pi_2^-} <: \alpha_2\}}$$

$$\textsc{Pattern Match}\quad \frac{\{\texttt{case}\ \alpha'\ \texttt{of}\ \overrightarrow{{}^{n} \pi_\square \rightsquigarrow \phi_\square} <: \alpha, \tau|^{\Pi^+}_{\Pi^-} <: \alpha'\} \subseteq C \qquad i \leq n \qquad \Pi_1^+ = \Pi^+ \cup \{\pi_i\} \qquad \Pi_1^- = \Pi^- \cup \{\pi_j \mid j < i\} \qquad \tau\backslash C \sim \frac{\Pi_1^+}{\Pi_1^-} \qquad \phi_i = \alpha_i \texttt{->} \alpha_i' \backslash C_i'}{C \Longrightarrow^1 C \cup \{\tau|^{\Pi_1^+}_{\Pi_1^-} <: \alpha_i\} \cup C_i' \cup \{\alpha_i' <: \alpha\}}$$

Figure 4.11: TinyBang Core constraint closure

for the additional filters.

- The Projection rule has a similar issue: at the type level, the "source" of a projection operation is a filtered record type. For precision, this filtering information is propogated to the projected component.

A prerequisite for our type checking process is to run the constraint closure to a fixed point. So it is convenient to define a notion of multiple closure steps:

Definition 4.13 (TinyBang Core Multiple Constraint Closure Steps)**.** We let $C_0 \Longrightarrow^* C_n$ if and only if $C_0 \Longrightarrow^1 C_1 \Longrightarrow^1 \ldots \Longrightarrow^1 C_n$

We now define the notion of an *inconsistent* constraint set. Informally an inconsistent constraint set is one that contains contradictory or otherwise unsatisfiable constraints.

$$\frac{\{\alpha_1\ \alpha_2 <: \alpha_3, \tau_1|^{\Pi_1^+}_{\Pi_1^-} <: \alpha_1\} \subseteq C \qquad \tau_1 \text{ is not a function type}}{C \text{ is inconsistent}}$$

Application Failure

$$\frac{\{\alpha_1 . l <: \alpha_2, \tau|^{\Pi^+}_{\Pi^-} <: \alpha_1\} \subseteq C \qquad \neg\textsc{IsRecord}(\tau)}{C \text{ is inconsistent}}$$

Non-Record Projection

$$\frac{\{\alpha_1 . l <: \alpha_2, \tau|^{\Pi^+}_{\Pi^-} <: \alpha_1\} \subseteq C \qquad l \notin \textsc{Fields}(\tau)}{C \text{ is inconsistent}}$$

Projection Failure

$$\frac{\{\texttt{case}\ \alpha'\ \texttt{of}\ \overrightarrow{\pi_\square \texttt{->} \phi_\square}^{n} <: \alpha\} \subseteq C \qquad \alpha' \backslash C \sim \frac{\emptyset}{\{\pi_1 ... \pi_n\}}}{C \text{ is inconsistent}}$$

Pattern Match Failure

Figure 4.12: TinyBang Core inconsistentency

Formally:

Definition 4.14 (Inconsistent Constraint Set)**.** A constraint set C is inconsistent if and only it satisfies at least one of the rules in Figure 4.12. Otherwise it is consistent.

The rules for inconsistency mirror those of stuck states from the operational semantics. We can finally define a formal notion of type correctness:

Definition 4.15 (Typechecking)**.** A closed expression e typechecks if and only if $[\![e]\!]_{\mathrm{E}} = \alpha \backslash C$ and for all $C \Longrightarrow^* C'$, C' is consistent.

4.6 Properties of the Semantics

Our ultimate goal is to prove the equivalence of TinyBang Core and Tagged TinyBang Core (define in Chapter 5). To this end, we will define in this section, a number of properties of the former, including the notion of type soundness.

The standard approach to proving type soundness for a subtyping system is to show *progress and preservation*, which involves the demonstration of two properties:

- Stuck expressions are never assigned valid types
- Given $e \longrightarrow^1 e'$, we must show that the type assigned to e is a super set of that assigned to e'.

Unfortunately, this approach is somewhat hard to apply to TinyBang Core's type system given the way it was defined. Instead we prove our type soundness by *simulation*, an approach partially inspired by abstract interpretation. We start by defining a simulation relation between TinyBang Core expressions and its type system and then showing that the initial alignment operations establishes this relation. We then demonstrate that the relation is preserved by the operational semantics and the constraint closure relations. Finally we demonstrate that stuck expressions are only simulated by inconsistent constraint sets; therefore if constraint closure only produces consistent sets, the expression is never stuck.

We only provide proof sketches, instead of detailed proofs, for most of the properties defined here. Our choice is influenced by two factors: type soundness is not our *primary* proof goal and our definitions and proofs are essentially simplified versions of TinyBang's. The reader is directed to [39] for a more detailed treatment of soundness proofs in this domain.

4.6.1 The Simulation Relation

We define the simulation relation between TinyBang Core expressions and type constraints in this section. Simulation is a four-place relation: the third place is the current

e	$\preccurlyeq_M$	$\alpha \backslash C$	iff	$\text{RV}(e) \preccurlyeq_M \alpha$ and $e \preccurlyeq_M C$
$\overrightarrow{s}$	$\preccurlyeq_M$	$\overrightarrow{c}$	iff	$\forall 1 \leq i \leq n.\, s_i \preccurlyeq_M c_i$
$x = r$	$\preccurlyeq_M$	$b <: \alpha$	iff	$x \preccurlyeq_M \alpha$ and $r \preccurlyeq_M b$
x	$\preccurlyeq_M$	α	iff	$M(x) = \alpha$
$x = v$	$\preccurlyeq_M$	$\tau\big\vert_{\Pi^-}^{\Pi^+} <: \alpha$	iff	$x \preccurlyeq_M \alpha$ and $v \preccurlyeq_M \tau$ and $P^+ \preccurlyeq_M \Pi^+$ and $P^- \preccurlyeq_M \Pi^-$ and $v \backslash E \sim \frac{P^+}{P^-}$
$x = x_1\ x_2$	$\preccurlyeq_M$	$\alpha_1\ \alpha_2 <: \alpha$	iff	$x \preccurlyeq_M \alpha$ and $x_1 \preccurlyeq_M \alpha_1$ and $x_2 \preccurlyeq_M \alpha_2$
$x = x'.l$	$\preccurlyeq_M$	$\alpha'.l <: \alpha$	iff	$x \preccurlyeq_M \alpha$ and $x' \preccurlyeq_M \alpha'$
$x = \texttt{case}\ x'\ \texttt{of}\ \overrightarrow{p \rightsquigarrow f}^n$	$\preccurlyeq_M$	$\texttt{case}\ \alpha\ \texttt{of}\ \overrightarrow{\pi \rightsquigarrow \phi}^n <: \alpha$	iff	$x \preccurlyeq_M \alpha$ and $x' \preccurlyeq_M \alpha'$ and $\forall 1 \leq i \leq n.\, p_i \preccurlyeq_M \pi_i$ and $f_i \preccurlyeq_M \phi_i$
$\mathbb{Z}$	$\preccurlyeq_M$	$\texttt{int}$		
$x \texttt{->} e$	$\preccurlyeq_M$	$\alpha \texttt{->} t$	iff	$x \preccurlyeq_M \alpha$ and $e \preccurlyeq_M t$
$\texttt{\{...,}\ l_i\texttt{=}x_i\ \texttt{,...\}}$	$\preccurlyeq_M$	$\texttt{\{...,}\ l_i\texttt{:}\alpha_i\ \texttt{,...\}}$	iff	$\forall i.\, x_i \preccurlyeq_M \alpha_i$

Figure 4.13: Simulation of expressions and constraints in TinyBang Core

environment and the fourth is a mapping, M, which maps each variable x to a type variable α. The formal definition is as follows:

Definition 4.16 (Simulation)**.** The simulation relation $\preccurlyeq_M$ is defined as the least relation satisfying the rules in Figure 4.13.

The definition of simulation is mostly straightforward with each expression grammar element aligning with the corresponding type grammar element. The alignment between a value and the corresponding type is slightly different. We must accomodate for the fact that our types can potentially be filtered. Filtered types gather and retain information gathered from analyzing pattern matching; so any value simulated by a filtered type must agree with it on the restrictions to the type. We achieve this by incorporating a compatibility proof obligation in to the simulation result.

Lemma 4.17 (Initial Alignment)**.** If e is well-formed, then $e \preccurlyeq_M [\![e]\!]_{\text{E}}$.

Lemma 4.18 (Compatibility)**.** Suppose $E \ {}_E\preccurlyeq_M \ C$, $P^+ \ {}_E\preccurlyeq_M \ \Pi^+$ and $P^- \ {}_E\preccurlyeq_M \ \Pi^-$, then $x\backslash E \sim \frac{P^+}{P^-}$ and $x \ {}_E\preccurlyeq_M \ \alpha$ together imply $\alpha\backslash C \sim \frac{\Pi^+}{\Pi^-}$. Further $v\backslash E \sim \frac{P^+}{P^-}$ and $v \ {}_E\preccurlyeq_M \ \tau$ together imply $\tau\backslash C \sim \frac{\Pi^+}{\Pi^-}$.

Lemma 4.19 (Combining Compatibility)**.** Suppose $x\backslash E \sim \frac{P_1^+}{P_1^-}$ and $x\backslash E \sim \frac{P_2^+}{P_2^-}$. Then $x\backslash E \sim \frac{P_1^+ \cup P_2^+}{P_1^- \cup P_2^-}$.

Lemma 4.20 (Simulation Preservation)**.** Suppose $E_0 \parallel e_0 \longrightarrow^1 E_1 \parallel e_1$ and $E_0 \parallel e_0 \ {}_{E_0}\preccurlyeq_{M_0} \ C_0$, then there exists C_1 and M_1 such that $C_0 \Longrightarrow^1 C_1$ and $E_1 \parallel e_1 \ {}_{E_1}\preccurlyeq_{M_1} \ C_1$.

Lemma 4.21 (Whole Program Simulation)**.** Given a program e, if $e \longrightarrow^* e'$, then $C \Longrightarrow^* C'$ where $C = [\![e]\!]_{\mathrm{E}}$ and $e' \ {}_E\preccurlyeq_M \ C'$.

Lemma 4.22 (Simulation of Stuck States)**.** If $e \ {}_E\preccurlyeq_M \ C$ and e is stuck then C is inconsistent.

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

Chapter 5

Formalization of Tagged TinyBang Core

In this chapter, we formalize an alternate, but equivalent, semantics for TinyBang Core based on the notion of adaptive tags described in Chapter 3. For distinguishing this system from the standard TinyBang Core system, we will use the term Tagged TinyBang Core to refer to it. Our definition is divided in to two parts: Section 5.3, defines an operational semantics, parameterized by a universe of tags and a dispatch table and Section 5.4 describes a formal scheme to derive this information via an enhanced TinyBang Core type system.

5.1 Notation

This chapter introduces a number of mapping structures where the same key is mapped multiple times. Such "multimaps" can be viewed as a mapping from a key to a *set*

$$
\begin{array}{rcll}
\mathfrak{t} & ::= & \star \mid \texttt{int} \mid \{\ldots, \ l{:}\mathfrak{t} \ , \ldots\} & \textit{tag} \\
\Omega & ::= & x \mapsto \underbrace{\mathfrak{t}} & \textit{var tag map} \\
\Delta & ::= & \underbrace{\mathfrak{d}} & \textit{dispatch table} \\
\mathfrak{d} & ::= & (x, \underbrace{\mathfrak{t}}, i) & \textit{dispatch table entry}
\end{array}
$$

Figure 5.1: Extensions to the TinyBang Core grammar

of values or simply as a set of unique key-value mappings. We use one view or the other based on convenience. Let k and l be arbitrary grammar elements and let Q be a multimap, then:

- We overload the set membership operator to work on such maps; e.g. $k \mapsto l \in Q$
- We define a generic lookup operation on multimaps that return a set of mapped values for a given key: i.e. $Q(k) = \{l \mid k \mapsto l \in Q\}$.
- We use the $\uplus$ to indicate the merging or union of two multimaps.

5.2 Grammar

In order to define our semantics, we need to augment the TinyBang Core grammar from Section 4.2. The additional non-terminals are defined in Figure 5.1.

Tags are an abstract representation of the deep structure of a value. So we model their grammar on types, although in contrast to the TinyBang Core types (Figure 4.1) which are shallow, the tag grammar is deep. Tags need not represent the complete structure of a value; so the tag grammar includes a "leaf" tag ($\star$) that can be used to elide parts of the structure that are irrelevant.

Ω and Δ are both parameters to our operational semantics. The former represents

$$\begin{array}{rcll} \hat{E} & ::= & \overrightarrow{x = \hat{v}} & \textit{environment} \\ \hat{v} & ::= & \langle v, \overbrace{t}^{} \rangle & \textit{tagged values} \end{array}$$

Figure 5.2: Extensions to the Tagged TinyBang Core grammar

the universe of tags, described in Section 3.2, and is a multimap from variables to tags. The operational semantics assigns tags to values exclusively from this collection. The latter is the dispatch table which is a collection of dispatch entries where each entry is a tuple consisting of a program point in the form of a type variable, a tag set and the index of the branch that matches the tag set. The semantics consults this table to perform efficient dispatch for `case` statements.

5.3 Operational Semantics

The small step operational semantics for Tagged TinyBang Core is defined over an environment of tagged values. The enhanced grammar is defined in 5.2. Each tagged value is represented as a pair consisting of the actual value and a set of tags.

Since $\hat{E}$ is not explicitly a subset of e, we define a separate well-formedness criteria for it below in the same vein as Definition 4.2.

Definition 5.1 (Well-formed Tagged Environment)**.** An expression $\hat{E}$ is considered well-formed if it meets the following criteria:

- $\hat{E}$ is closed.
- Each variable is bound in at most one clause.

It is also convenient to define a set of lookup operations on the environment:

Definition 5.2 (Tagged TinyBang Core Environment Lookup)**.** If $\hat{E}$ be a well-formed environment, then:

- $\hat{E}(x) = \hat{v}$ if and only if $x = \hat{v} \in \hat{E}$.
- $\hat{E}^{v}(x) = v$ if and only if $x = \langle v, \overline{\mathfrak{t}} \rangle \in \hat{E}$.
- $\hat{E}^{\mathfrak{t}}(x) = \overline{\mathfrak{t}}$ if and only if $x = \langle v, \overline{\mathfrak{t}} \rangle \in \hat{E}$.

Many of our operational semantics rules need to select the leaf tag from Ω, if one exists. We define a simple function for that purpose:

Definition 5.3 (Leaf Tag Selection)**.** We let $\textsc{Leaf}(\Omega, x) = \{\star \mid \star \in \Omega(x)\}$.

We can now formally define the small step relation for the system as well as the multi-step evaluation process:

Definition 5.4 (Tagged TinyBang Core Small Step Semantics)**.** We let $E \parallel e \underset{\Omega,\Delta}{\longrightarrow}{}^{1}_{\delta} E' \parallel e'$ be the relationship satisfying the rules in Figure 5.3.

Definition 5.5 (Tagged TinyBang Core Multiple Small Steps)**.** We let $\hat{E}_0 \parallel e_0 \longrightarrow^{*}_{\delta} \hat{E}_n \parallel e_n$ if and only if $\hat{E}_0 \parallel e_0 \longrightarrow^{1} \hat{E}_1 \parallel e_1 \longrightarrow^{1} \ldots \longrightarrow^{1} \hat{E}_n \parallel e_n$.

The small-step operational semantics is parameterized over Ω and Δ. We often elide the parameters when it is clear from context. All the rules, except for Pattern Match are mechanically similar to the rules from TinyBang Core (Figure 4.5) except for the assignment of tags.

Given an expression of the form $x = r$, the tags assigned are always a subset of $\Omega(x)$ and will include leaf tags, if present. For example, when assigning tags for an expression of the form $x = \texttt{100}$, the Integer rule deterministically assigns the subset of $\{\texttt{int}, \star\}$ that

$$\textsc{Integer Value}\quad \frac{n \in \mathbb{Z} \qquad \hat{v} = \langle n, \overline{\mathfrak{t}} \rangle \qquad \overline{\mathfrak{t}} = \{\texttt{int} \mid \texttt{int} \in \Omega(x)\} \cup \textsc{Leaf}(\Omega, x)}{\hat{E} \parallel x \texttt{ = } n \parallel e \longrightarrow^1_\delta \hat{E} \parallel x \texttt{ = } \hat{v} \parallel e}$$

$$\textsc{Function Value}\quad \frac{\hat{v} = \langle x_2 \texttt{ -> } e_2, \overline{\mathfrak{t}} \rangle \qquad \overline{\mathfrak{t}} = \{\mathfrak{t} \mid \mathfrak{t} = [\![x_2 \texttt{ -> } e_2]\!]_{\mathrm{E}} \wedge \mathfrak{t} \in \Omega(x_1)\} \cup \textsc{Leaf}(\Omega, x_1)}{\hat{E} \parallel x_1 \texttt{ = } x_2 \texttt{ -> } e_2 \parallel e_3 \longrightarrow^1_\delta \hat{E} \parallel x_1 \texttt{ = } \hat{v} \parallel e_3}$$

$$\textsc{Record Value (TODO: Rework)}\quad \frac{\hat{v} = \langle \{\ldots,\ l_i \texttt{=} x_i\ ,\ldots\}, \overline{\mathfrak{t}} \rangle \qquad \overline{\mathfrak{t}} = \{\{\ldots,\ l_i \texttt{:} \mathfrak{t}_j\ ,\ldots\} \in \Omega(x) \mid \mathfrak{t}_j \in \hat{E}^{\mathfrak{t}}(x_i)\} \cup \textsc{Leaf}(\Omega, x)}{\hat{E} \parallel x \texttt{ = } \{\ldots,\ l_i \texttt{=} x_i\ ,\ldots\} \parallel e \longrightarrow^1_\delta \hat{E} \parallel x \texttt{ = } \hat{v} \parallel e}$$

$$\textsc{Variable Lookup}\quad \frac{v = \hat{E}^v(x_2) \qquad \overline{\mathfrak{t}} = (\hat{E}^{\mathfrak{t}}(x_2) \cap \Omega(x_1)) \cup \textsc{Leaf}(\Omega, x_1)}{\hat{E} \parallel x_1 \texttt{ = } x_2 \parallel e \longrightarrow^1_\delta \hat{E} \parallel x_1 \texttt{ = } \langle v, \overline{\mathfrak{t}} \rangle \parallel e}$$

$$\textsc{Application}\quad \frac{\hat{E}^v(x_2) = x_4' \texttt{ -> } e_1' \qquad \hat{E}^v(x_3) = v \qquad \overline{\mathfrak{t}} = (\hat{E}^{\mathfrak{t}}(x_3) \cap \Omega(x_4)) \cup \textsc{Leaf}(\Omega, x_4) \qquad \boldsymbol{\alpha}(x_1, x_4' \texttt{ -> } e_1') = x_4 \texttt{ -> } e_1}{\hat{E} \parallel x_1 \texttt{ = } x_2\ x_3 \parallel e_2 \longrightarrow^1_\delta \hat{E} \parallel x_4 \texttt{ = } \langle v, \overline{\mathfrak{t}} \rangle \parallel e_1 \parallel x_1 = \textsc{rv}(e_1) \parallel e_2}$$

$$\textsc{Projection}\quad \frac{\hat{E}^v(x_2) = \{\ldots,\ l \texttt{=} x_3\ ,\ldots\} \qquad v = \hat{E}^v(x_3) \qquad \overline{\mathfrak{t}} = (\hat{E}^v(x_3) \cap \Omega(x_1)) \cup \textsc{Leaf}(\Omega, x_1)}{\hat{E} \parallel x_1 \texttt{ = } x_2 \texttt{ . } l \parallel e \longrightarrow^1_\delta \hat{E} \parallel x_1 \texttt{ = } \langle v, \overline{\mathfrak{t}} \rangle \parallel e}$$

$$\textsc{Pattern Match}\quad \frac{\overline{\mathfrak{t}_1} = \hat{E}^{\mathfrak{t}}(x') \qquad (x, \overline{\mathfrak{t}_1}, i) \in \Delta \qquad f_i = x_i \texttt{ -> } e_i \qquad v = \hat{E}^v(x') \qquad \overline{\mathfrak{t}_2} = (\overline{\mathfrak{t}_1} \cap \Omega(x_i)) \cup \textsc{Leaf}(\Omega, x_i)}{\hat{E} \parallel x \texttt{ = case } x' \texttt{ of } \overrightarrow{{}^{n}\, p_\square \rightsquigarrow f_\square} \parallel e \longrightarrow^1_\delta \hat{E} \parallel x_i \texttt{ = } \langle v, \overline{\mathfrak{t}_2} \rangle \parallel e_i \parallel x \texttt{ = } \textsc{rv}(e_i) \parallel e}$$

Figure 5.3: Tagged TinyBang Core operational semantics

appear in $\Omega(x)$. In the case of other expressions like $x_1 = x_2$, tag assignment depends on the current environment: the rule must examine the set of tags assigned to x_2 to determine the set of tags to assign to x_1. The most complex rule of this kind is the RECORD VALUE rule; each field at runtime has a set of tags associated with it; any record tag formed by selecting any one tag for each field is a viable candidate; the rule selects the subset of such tags that appears in Ω.

The semantics handles pattern matches differently from TinyBang Core. The rule does not invoke compatibility, but merely examines the dispatch table Δ to determine the appropriate matching branch. The rest, i.e. binding of arguments, freshening the branch body and inlining, are handled in a manner identical to TinyBang Core. Generating the dispatch table Δ, along with an appropriate Ω, such that dispatch behavior of Tagged TinyBang Core is equivalent to that of TinyBang Core, is a goal of the thesis.

5.4 Tag Derivation

In this section we formally address the question of deriving adaptive tags for a given program. Specifically we discuss the process of computing Ω and Δ, the parameters to the operational semantics from Section 5.3, by extending the TinyBang Core type system from Section 4.5.

The operational semantics from the previous section provides a framework to target. The goal of a tag derivation system is then two fold: it must generate tags and dispatch tables for each `case` statement and it must then distribute tags amongst program points in such a way that, at runtime, when the small step evaluation reaches a `case` statement, the

$$\omega \quad ::= \quad \alpha \mapsto \mathfrak{t} \quad \textit{type tag multimap}$$

Figure 5.4: Type compatibility extensions for Tagged TinyBang Core type grammar

tags present on the subject of the pattern match ensure correct dispatch.

A key observation from our initial discussion of adaptive tags in Section 3.2 is that construction and destruction are dual operations; it is then possible to generate tags by observing the destruction process. The TinyBang Core operational semantics and type systems are closely aligned and the latter simulates the former. Together this allows us to conservatively trace the destruction process and propogate tag information through the program.

The rest of this section is organized as follows: in Section 5.4.1 we define an extended type compatibility relation that generates tag information; then in Section 5.4.2 we define the extended constraint closure relation that collects tag and dispatch information from across the whole program and finally in Section 5.4.3 we define the scheme to generate the tag map Ω and dispatch table Δ given this information.

5.4.1 Type Compatibility

Our first task is to extend type compatibility to include tags. This involves modifying the rules from Figure 4.10 to also "emit" tags. The modification is sometimes straightforward: for example, in the case of the Integer rule, the tag is always `int`. However type compatibility is inductively defined; so in the general case the tag depends on the structure of the proof tree. But otherwise the extensions are fairly intuitive.

We first extend our grammar slightly in Figure 5.4 to include ω, a multimap from

type variables to tags.

Our extended type compatibility relation has the form: $\alpha \backslash C \sim \frac{\Pi^+}{\Pi^-} \triangleright (\mathfrak{t}, \omega)$. Here α, C, Π^+ and Π^- have the same meanings as in the TinyBang Core type compatibility relation from Figure 4.10; $\mathfrak{t}$ is the tag associated with this compatibility invocation while ω accumulates tags in the current proof tree.

These rules are in the same spirit as the type compatibility rules for TinyBang Core in Figure 4.10. They have been extended to trace usage and extract tag information. Each instance of type compatibility represents a pattern match on a particular value with a specific set of union choices and generates a single tag. The constraint closure, which we define in Section 5.4.2, ensures that we collect information from across all possible matches and union choices, which we then use to build up a collection of potential tags for each value.

Our formal definition of type compatibility is similar to Definition 4.11 from TinyBang Core:

Definition 5.6. We let $\alpha \backslash C \sim \frac{\Pi^+}{\Pi^-} \triangleright (\mathfrak{t}, \omega)$ and $\tau \backslash C \sim \frac{\Pi^+}{\Pi^-} \triangleright (\mathfrak{t}, \omega)$ be the mutually defined relations satisfying the rules in Figure 5.5.

5.4.2 Constraint Closure

Our modified type compatibility relation is capable of generating tags from individual pattern matches. We must now modify our constraint closure to utilize it to not only generate and accumulate tags across the whole program, but also to record dispatch data.

We must also handle a minor, but fairly technical, wrinkle. Consider code and

$$\textsc{Leaf}\quad \frac{}{\alpha \backslash C \sim \frac{\emptyset}{\emptyset} \triangleright (\star, \{\alpha \mapsto \star\})}$$

$$\textsc{Type Selection}\quad \frac{\tau|^{\Pi_1^+}_{\Pi_1^-} <: \alpha \in C \qquad \tau \backslash C \sim \frac{\Pi_1^+ \cup \Pi_2^+}{\Pi_1^- \cup \Pi_2^-} \triangleright (\mathfrak{t}, \omega) \qquad \omega' = \omega \uplus \{\alpha \mapsto \mathfrak{t}\}}{\alpha \backslash C \sim \frac{\Pi_2^+}{\Pi_2^-} \triangleright (\mathfrak{t}, \omega')}$$

$$\textsc{Record}\quad \frac{\begin{array}{c}\tau = \{\ldots,\ l_i : \alpha_i\ ,\ldots\} \\ \forall \pi \in \Pi^+.\ \textsc{IsRecord}(\pi) \qquad \forall \pi \in \Pi^+.\ \textsc{Fields}(\pi) \subseteq \textsc{Fields}(\tau) \\ \forall i. \alpha_i \backslash C \sim \frac{\Pi^+.l_i}{\Pi_i^-} \triangleright (\mathfrak{t}_i, \omega_i) \qquad \mathfrak{t} = \{\ldots,\ l_i : \mathfrak{t}_i\ ,\ldots\} \qquad \omega = \biguplus \omega_i\end{array}}{\tau \backslash C \sim \frac{\Pi^+}{\bigcup_i l_i \Pi_i^-} \triangleright (\mathfrak{t}, \omega)}$$

$$\textsc{Record - Absent Field}\quad \frac{\tau \backslash C \sim \frac{\Pi^+}{\Pi^-} \triangleright (\mathfrak{t}, \omega) \qquad \textsc{IsRecord}(\tau) \qquad l \notin \textsc{Fields}(\tau)}{\tau \backslash C \sim \frac{\Pi^+}{\{l : \pi\} \cup \Pi^-} \triangleright (\mathfrak{t}, \omega)}$$

$$\textsc{Record - Extension}\quad \frac{\begin{array}{c}\tau \backslash C \sim \frac{\Pi^+}{\pi \cup \Pi^-} \triangleright (\mathfrak{t}, \omega) \\ \textsc{IsRecord}(\tau) \qquad \pi = \{\ldots,\ l_i : \pi_i\ ,\ldots\} \qquad \pi' = \{\ldots,\ l_i : \pi_i,\ \ldots,\ l'' : \pi''\}\end{array}}{\tau \backslash C \sim \frac{\Pi^+}{\pi' \cup \Pi^-} \triangleright (\mathfrak{t}, \omega)}$$

$$\textsc{Record - Non-Record Pattern}\quad \frac{\tau \backslash C \sim \frac{\Pi^+}{\Pi^-} \triangleright (\mathfrak{t}, \omega) \qquad \textsc{IsRecord}(\tau) \qquad \neg\textsc{IsRecord}(\pi')}{\tau \backslash C \sim \frac{\Pi^+}{\pi' \cup \Pi^-} \triangleright (\mathfrak{t}, \omega)}$$

$$\textsc{Integer}\quad \frac{\Pi^+ = \{\texttt{int}\} \qquad \texttt{int} \notin \Pi^-}{\texttt{int} \backslash C \sim \frac{\Pi^+}{\Pi^-} \triangleright (\texttt{int}, \emptyset)}$$

Figure 5.5: Tagged TinyBang Core type compatibility relation

the corresponding initial constraint set like in Listing 5.6 where the type variable α_i corresponds to program variable x_i. The modified type compatibility relation will analyze the case branch and assign a potential tag of `{a:int}` to α_2. However the variable x_1 (and correspondingly α_1) is never directly involved in any destruction operation given the way the rules of compatibility and small step evaluation are defined; so it is never directly assigned a tag. Recollect that tags at runtime are assigned from a fixed set. If x_1 has no tags associated with it, it will never be assigned any at runtime which causes x_2 to also have no

```
1  x0 = 3
2  x1 = { a = x0 }
3  ...
4  x2 = x1
5  ...
6  x3 = case x2 of
7    | { a:int } -> ...
```

(a) Code listing

$\texttt{int} <: \alpha_0$

$\texttt{\{a:}\alpha_0\texttt{\}} <: \alpha_1$

$\alpha_1 <: \alpha_2$

$\texttt{case } \alpha_2 \texttt{ of [\{a: int\}} \rightsquigarrow \ldots\texttt{]} <: \alpha_3$

$\ldots$

(b) Constraint set

Figure 5.6: Constraint closure discontinuity example

runtime tags and the dispatch fails at x_3 which would be unsound. To solve this issue we must ensure that tags are properly shared between x_2 and x_1.

An important point is that assignment operations in the program correspond to type constraints of the form $\alpha_1 <: \alpha_2$ which are easily detected within the constraint set. Similar points of discontinuity exist for many other operations; for example in the case of pattern matching, there is a similar disconnect between branch variables and the subject variables; these are not easily found by examining the constraint set. So we chose to explicitly track such data "feeds" during constraint closure. For example, the feed corresponding to $\alpha_1 <: \alpha_2$ is $\alpha_1 \twoheadrightarrow \alpha_2$.

The formal grammar is listed in Figure 5.7. Our extended constraint closure tracks four pieces of data: the set of constraints C, the set of tags discovered so far ω, a set of dispatch entries δ and a collection of feeds η. We formally define the relationship below:

Definition 5.7 (Tagged TinyBang Core Constraint Closure)**.** We let $\hat{C} \Longrightarrow^{1}_{\delta} \hat{C}'$ be the relationship satisfying the rules in Figure 5.8.

The extensions to the standard TinyBang Core constraint closure rules (Figure

$$
\begin{array}{rcll}
\eta & ::= & \overbrace{\alpha \rightarrowtail \alpha} & \textit{feeds} \\
\delta & ::= & \overbrace{(\alpha, \mathfrak{t}, i)} & \textit{dispatch data} \\
\hat{C} & ::= & (C, \omega, \delta, \eta) & \textit{augmented constraints}
\end{array}
$$

Figure 5.7: Constraint Closure extensions for Tagged TinyBang Core type grammar

4.11) are fairly straightforward and work on some general principles:

- For each rule, the underlying constraint propogation is *identical* to the corresponding rule for TinyBang Core's constraint closure.

- Whenever a rule requires a compatibility check, we extend the existing map of tags with the new set of tags from the compatibility as well as the top level tag it emits. For example, the Pattern Match rule defines the new tag map ω'' as $\omega \uplus \omega' \cup \{\alpha \mapsto \mathfrak{t}\}$ where ω is the original map, ω' is the set of tags from the type compatibility relation in the premises of the rule and $\mathfrak{t}$ is the corresponding top level tag.

- Whenever the lower bound of a type variable is selected and copied to another type variable, we extend the set of feeds to indicate that the former feeds in to the latter. For example in the Application rule, we introduce a feed $\alpha_2 \rightarrowtail \alpha_4$ between the argument α_2 and the function's formal parameter α_4.

- The Pattern Match rule extends the dispatch data with a new entry $(\alpha, \mathfrak{t}, i)$ where α corresponds to the program point with the `case` statement, $\mathfrak{t}$ is the top-level tag corresponding to the successful type compatibility match and i is the index of the branch that matched.

We also define the notion of multiple constraint closure steps:

$$\textsc{Transitivity}\quad \frac{\{\tau|^{\Pi^+}_{\Pi^-} <: \alpha_1, \alpha_1 <: \alpha_2\} \subseteq C \qquad \eta' = \eta \uplus \{\alpha_1 \rightsquigarrow \alpha_2\}}{(C, \omega, \delta, \eta) \Longrightarrow^1_\delta (C \cup \{\tau|^{\Pi^+}_{\Pi^-} <: \alpha_2\}, \omega, \delta, \eta')}$$

$$\textsc{Application}\quad \frac{\begin{array}{c}\{\alpha_1\ \alpha_2 <: \alpha_3, \tau_1|^{\Pi^+_1}_{\Pi^-_1} <: \alpha_1, \tau_2|^{\Pi^+_2}_{\Pi^-_2} <: \alpha_2\} \subseteq C \\ \tau_1 = \alpha_4 \texttt{->} \alpha' \backslash C' \qquad C'' = C \cup C' \cup \{\tau_2|^{\Pi^+_2}_{\Pi^-_2} <: \alpha_4, \alpha' <: \alpha_3\} \qquad \eta'' = \eta \cup \{\alpha_2 \rightsquigarrow \alpha_4\}\end{array}}{(C, \omega, \delta, \eta) \Longrightarrow^1_\delta (C'', \omega, \delta, \eta'')}$$

$$\textsc{Projection}\quad \frac{\begin{array}{c}\{\alpha_1.l <: \alpha_2, \{\ldots,\ l : \alpha_3\ ,\ldots\}|^{\Pi^+_1}_{\Pi^-_1} <: \alpha_1, \tau|^{\Pi^+_2}_{\Pi^-_2} <: \alpha_3\} \subseteq C \qquad \tau\backslash C \sim \frac{\Pi^+_2 \cup \Pi^+_1.l}{\Pi^-_2} \triangleright (\mathfrak{t}', \omega') \\ C' = C \cup \{\tau'|^{\Pi^+_2 \cup \Pi^+_1.l}_{\Pi^-_2} <: \alpha_2\} \qquad \omega'' = \omega \uplus \omega' \uplus \{\alpha_3 \mapsto \mathfrak{t}'\} \qquad \eta' = \eta \uplus \{\alpha_3 \rightsquigarrow \alpha_2\}\end{array}}{(C, \omega, \delta, \eta) \Longrightarrow^1_\delta (C', \omega'', \delta, \eta')}$$

$$\textsc{Pattern Match}\quad \frac{\begin{array}{c}\{\texttt{case}\ \alpha'\ \texttt{of}\ \overrightarrow{{}^{n}\ \pi_\square \rightsquigarrow \phi_\square} <: \alpha, \tau|^{\Pi^+}_{\Pi^-} <: \alpha'\} \subseteq C \\ i \leq n \qquad \Pi^+_1 = \Pi^+ \cup \{\pi_i\} \qquad \Pi^-_1 = \Pi^- \cup \{\pi_j \mid j < i\} \\ \tau\backslash C \sim \frac{\Pi^+_1}{\Pi^-_1} \triangleright (\mathfrak{t}, \omega') \qquad \phi_i = \alpha_i \texttt{->} \alpha'_i \backslash C'_i \qquad C' = C \cup \{\tau|^{\Pi^+_1}_{\Pi^-_1} <: \alpha_i\} \cup C'_i \cup \{\alpha'_i <: \alpha\} \\ \omega'' = \omega \uplus \omega' \uplus \{\alpha' \mapsto \mathfrak{t}\} \qquad \delta' = \delta \cup \{(\alpha, \mathfrak{t}, i)\} \qquad \eta' = \eta \cup \{\alpha' \rightsquigarrow \alpha_i\}\end{array}}{(C, \omega, \delta, \eta) \Longrightarrow^1_\delta (C', \omega'', \delta', \eta')}$$

Figure 5.8: Tagged TinyBang Core constraint closure

Definition 5.8 (Tagged TinyBang Core Multiple Constraint Closure Steps)**.** We define $\hat{C}_0 \Longrightarrow^*_\delta \hat{C}_n$ if and only if $\hat{C}_0 \Longrightarrow^1_\delta \hat{C}_1 \Longrightarrow^1_\delta \cdots \Longrightarrow^1_\delta \hat{C}_n$

And a notion of a full closure:

Definition 5.9 (Tagged TinyBang Core Complete Constraint Closure)**.** We define $\hat{C}_0 \Longrightarrow^!_\delta \hat{C}_n$ if and only if $\hat{C}_0 \Longrightarrow^*_\delta \hat{C}_n$ and there exists no $\hat{C} \neq \hat{C}_n$ such that $\hat{C}_n \Longrightarrow^1_\delta \hat{C}$.

At the end of constraint closure, the co-domain of ω has all the tags ever required by the whole program. However we still must solve the tag set discontinuity issue from earlier in the section. This is now fairly simple since we have also accumulated all pairs of variables at which the issue appears. To address it, all we need to do is to define a

$$\frac{\alpha' \not\hookrightarrow \alpha \in \eta \qquad \mathfrak{t} \in \omega(\alpha) \qquad \omega' = \omega \uplus \{\alpha' \mapsto \mathfrak{t}\}}{(C, \omega, \delta, \eta) \downarrow (C, \omega', \delta, \eta)}$$

Figure 5.9: Tagged TinyBang Core Tag Closure

tag-closure operation on $\hat{C}$ which pushes tags back to the source of the feed.

Definition 5.10 (Tag Closure)**.** We let $\hat{C} \downarrow \hat{C}'$ be the relation that satisfies the rules in Figure 5.9.

We also define a notion of a complete tag closure:

Definition 5.11 (Complete Tag Closure)**.** We say $\hat{C}_n$ is a *complete* tag closure of $\hat{C}_0$, i.e. $\hat{C}_0 \downarrow^! \hat{C}_n$, if and only if $\hat{C}_0 \downarrow \hat{C}_1 \ldots \downarrow \hat{C}_n$ and there exists no $\hat{C} \neq \hat{C}_n$ such that $\hat{C}_n \downarrow \hat{C}$.

For convenience, we define an extended version of the initial derivation for Tagged TinyBang Core:

Definition 5.12 (Tagged TinyBang Core Initial Derivation)**.** We let $[\![e]\!]_\delta = (C, \emptyset, \emptyset, \emptyset)$ if and only if $\alpha \backslash C = [\![e]\!]_{\mathrm{E}}$.

Definition 5.13 (Tagged TinyBang Core Complete Closure)**.** We say $\hat{C}_n$ is a *complete closure* of $\hat{C}_0$ if and only if there exists $\hat{C}_m$ which is the complete constraint closure of $\hat{C}_0$ as per Definition 5.9 and $\hat{C}_n$ is the complete tag closure of $\hat{C}_m$ as per Definition 5.11.

5.4.3 Deriving the Parameters

We are almost ready to define Ω, but defining Δ requires a little bit more work. All the dispatch data we have collected during constraint closure have the form $(\alpha, \mathfrak{t}, i)$. Runtime data, on the other hand, is associated with a *set of* tags. Dispatch would be inefficient if we have to search through such a set at runtime. So we would like to index

dispatch tables directly on tag sets.

Conceptually the tag set associated with a value at runtime is highly structured; the tags represent parts of the same structure and are therefore closely related. We formalize this using a notion of *non-conflict*:

Definition 5.14 (Non-Conflicting Tags)**.** Two tags $\mathfrak{t}$ and $\mathfrak{t}'$ are non-conflicting ($\mathfrak{t} \nleftrightarrow \mathfrak{t}'$) if any of the following hold:

- $\mathfrak{t} = \mathfrak{t}'$
- $\mathfrak{t} = \star$ or $\mathfrak{t}' = \star$
- $\mathfrak{t} = \{\ldots,\ l_i : \mathfrak{t}_i\ ,\ldots\}$ and $\mathfrak{t}' = \{\ldots,\ l_i : \mathfrak{t}'_i\ ,\ldots\}$ and for all i, $\mathfrak{t}_i \nleftrightarrow \mathfrak{t}'_i$

Tag sets that occur at runtime cannot have conflicting tags in them. We define the notion of non-conflicting tag sets formally:

Definition 5.15 (Non-Conflicting Tag Sets)**.** A tag set $\overline{\mathfrak{t}}$ is non-conflicting if and only if for any $\{\mathfrak{t}_1, \mathfrak{t}_2\} \in \overline{\mathfrak{t}}$, $\mathfrak{t}_1 \nleftrightarrow \mathfrak{t}_2$.

Since the source of our data is the type system, our tables are all currently keyed on type variables. However the operational semantics parameters Ω and Δ are expected to be keyed on program variables. So we need a mapping between the two. Assuming such a mapping, we can finally define the two parameters for the operational semantics:

Definition 5.16 (Extracting Ω and Δ parameters)**.** Given an expression e and a mapping from program variables to type variables, M, let the complete closure of $[\![e]\!]_\delta$ as per Definition 5.13 be $(C, \omega, \delta, \eta)$. Further let C be consistent. Then Ω and Δ are derived as follows:

- $\Omega = \{x \mapsto \overline{\mathfrak{t}} \mid M(x) \mapsto \overline{\mathfrak{t}} \in \omega\}$.

- Let $t(\alpha)$ be the complete set of applicable tags for a `case` statement at program point α, that is: $t(\alpha) = \omega(\alpha')$ for `case` α' `of` $\overrightarrow{\pi \rightsquigarrow \phi}$ $<: \alpha \in C$. Then $\Delta = \{x \mapsto (\alpha, \overline{\mathfrak{t}}, i) \mid \exists M(x) \mapsto (\alpha, \mathfrak{t}, i) \in \delta. \exists \overline{\mathfrak{t}}. \mathfrak{t} \in \overline{\mathfrak{t}} \wedge \overline{\mathfrak{t}} \subseteq t(\alpha) \wedge \overline{\mathfrak{t}}$ is non-conflicting$\}$

We denote the above derivation of parameters Ω and Δ from e and M as the operation $(\Delta, \Omega) \twoheadleftarrow_M e$.

The definition of Ω is trivial given M; it simply translates the variables using M and then defers to ω. The definition of the dispatch table is slightly more involved: for each `case` statement, it collects the set of all tags and then "distributes" the dispatch entries to non-conflicting subsets that includes the tag from the entry. This relies on two properties: tags on any particular value do not conflict among themselves at runtime and that non-conflicting tag sets will not disagree on dispatch. We will prove both properties in Chapter 6.

5.5 Properties of Tagged TinyBang Core

Our ultimate goal is to demonstrate the equivalence between the operational semantics of Tagged TinyBang Core and TinyBang Core. Our strategy is to show that the two systems *bisimulate*: specifically we prove that the two systems execute in lock-step with each other and execute the same computations.

To compare the two systems, we need to define a notion of equivalence between the environments:

Definition 5.17. $E \ {}_M\dot{\approx}_{e_0} \ \hat{E}$ if and only if $e_0 \longrightarrow^* E \parallel e$ for some e and there exists $(\Omega, \Delta) \twoheadleftarrow_M e_0$ such that $e_0 \xrightarrow[\Omega,\Delta]{*}_{\delta} \hat{E} \parallel e$ and the following conditions are met:

- $|E| = |\hat{E}| = n$.
- For all $i \leq n$, $E(x_i) = \hat{E}^v(x_i)$

The equivalence above is somewhat complicated by the fact that we must constrain the set of tags to those legitimately derived from the scheme described in Section 5.4. The definition is, therefore, parameterized over an initial expression e_0 and the mapping, M, from variables to type variables.

Then the theorem below states that if we execute a program using both the TinyBang Core and the Tagged TinyBang Core semantics, each system executes a small step if and only the other does. Further if their environments were equivalent at the start, they remain equivalent after the step. The formal statement is as follows:

Theorem 2 (Bisimulation of the Operational Semantics)**.** Let e_0 be the initial program. Further let $e_0 \longrightarrow^* E_1 \parallel e_1$ and $e_0 \xrightarrow[\Omega,\Delta]{*}_{\delta} \hat{E}_1 \parallel e_1$ such that $(\Omega, \Delta) \twoheadleftarrow_M e_0$ and $E_1 \ {}_M\dot{\approx}_{e_0} \ \hat{E}_1$. Then $E_1 \parallel e_1 \longrightarrow^1 E_2 \parallel e_2$ if and only if there exists $\hat{E}_2$ such that $\hat{E}_1 \parallel e_1 \longrightarrow^1_{\delta} \hat{E}_2 \parallel e_2$ and $E_2 \ {}_M\dot{\approx}_{e_0} \ \hat{E}_2$.

We present a proof of this theorem in Chapter 6.

Chapter 6

The Bisimulation Proof

In this chapter, we prove the equivalence between TinyBang Core and Tagged TinyBang Core via bisimulation.

Most of the operational semantics rules assign tags to values. Then, at pattern matches, these tags are used to make dispatch decisions. From the perspective of an equivalence proof for Tagged TinyBang Core, we must show that, for any `case` statement, the tags present on the subject value can be used to make dispatch decisions that are consistent with those made by TinyBang Core. Recall, from Section 4.4, that the latter performs pattern matching and dispatch by inspecting the value at runtime.

Our overall strategy is to show that the two operational semantics *bisimulate* each other. Specifically, we define an equivalence between the environments defined by TinyBang Core and Tagged TinyBang Core and then show that when either of the operational semantics takes a step, there is a corresponding step in the other that maintains this equivalence. But this involves a number of sub-tasks, which we tackle in the following sections.

6.1 Bisimulation of Type Systems

The type system for Tagged TinyBang Core, defined in Section 5.4 is an extension of the TinyBang Core type system defined in Section 4.5. We formalize this idea by demonstrating a bisimulation relation between the two type systems. In turn, this allows us to "port" many of the properties of the TinyBang Core type system defined in Section 4.6 to the extended type system.

The two constraint closure relations are very similar to each other except that the Tagged TinyBang Core system also "emits" some structural information. This has the advantage of making their bisimulation fairly easy to prove; but first we must ensure that the extra data emitted by the tagged system is properly constrained. We do this by parameterizing the relation over an *initial constraint set* and adjusting our definition accordingly. The bisimulation relation is defined formally below:

Definition 6.1. We let $C \dot{\approx}_{C_0} (C, \omega, \delta, \eta)$ if and only if $(C_0, \emptyset, \emptyset, \emptyset) \Longrightarrow_\delta^* (C, \omega, \delta, \eta)$

We use this relation to define a series of lemmas that relate the two type systems. First we relate the two compatibility relations:

Lemma 6.2. $\alpha \backslash C \sim \frac{\Pi^+}{\Pi^-}$ if and only if there exists fixed values of $\mathfrak{t}$ and ω such that $\alpha \backslash C \sim \frac{\Pi^+}{\Pi^-} \triangleright (\mathfrak{t}, \omega)$. Similarly $\tau \backslash C \sim \frac{\Pi^+}{\Pi^-}$ if and only if $\tau \backslash C \sim \frac{\Pi^+}{\Pi^-} \triangleright (\mathfrak{t}, \omega)$ for some $\mathfrak{t}$ and ω.

Proof. The forward implication follows from induction over the height of the proof tree of $\alpha \backslash C \sim \frac{\Pi^+}{\Pi^-}$, followed by case analysis of the rules from Figure 4.10. Similarly the converse property follows from induction over the height of the proof tree of $\alpha \backslash C \sim \frac{\Pi^+}{\Pi^-} \triangleright (\mathfrak{t}, \omega)$ followed by case analysis of the rules from Figure 5.5. A similar argument can be made for

$\tau \backslash C \sim \frac{\Pi^+}{\Pi^-}$ and $\tau \backslash C \sim \frac{\Pi^+}{\Pi^-} \triangleright (\mathfrak{t}, \omega)$. □

Furthermore, for every TinyBang Core constraint closure, there is a corresponding closure operation in the Tagged TinyBang Core type system and vice-versa. We define this in two steps:

Lemma 6.3. Let $C_1 \dot{\approx}_{C_0} \hat{C}_1$. Then $C_1 \Longrightarrow^1 C_2$ if and only if $\hat{C}_1 \Longrightarrow^1_\delta \hat{C}_2$ such that $C_2 \dot{\approx}_{C_0} \hat{C}_2$.

Proof. Direct by case analysis of the rules in Figure 5.8. □

Lemma 6.4. Given an initial constraint set C_0, $C_0 \Longrightarrow^* C$ if and only if there exists ω, δ and η such that $(C_0, \emptyset, \emptyset, \emptyset) \Longrightarrow^*_\delta (C, \omega, \delta, \eta)$ and $C \dot{\approx}_{C_0} (C, \omega, \delta, \eta)$.

Proof. By induction on the length of $C_0 \Longrightarrow^* C$ and using Lemma 6.3. □

As we discussed in Chapter 4, the TinyBang Core type closure simulates its small step semantics. Intuitively, for each small step, there is a corresponding step in the constraint closure. From the previous lemmas, we have established that the constraint closures of TinyBang Core and Tagged TinyBang Core bisimulate. Therefore, for each small step in TinyBang Core, there is a constraint closure step in Tagged TinyBang Core or in other words, the TinyBang Core small step semantics is simulated by Tagged TinyBang Core constraint closure. Similarly the compatibility relation of TinyBang Core is simulated by the type compatibility relation of Tagged TinyBang Core. We formalize these notion below.

Definition 6.5 (Simulation)**.** We let $e \; {}_E\dot{\preccurlyeq}_{M,C_0} \hat{C}$ if and only if $\hat{C} = (C, \omega, \delta, \eta)$, $e \; {}_E\preccurlyeq_M C$ and $C \dot{\approx}_{C_0} \hat{C}$. The simulation relation $e \; {}_E\preccurlyeq_M C$, is defined as per 4.16.

Lemma 6.6. Suppose $e \ {}_E\dot{\preccurlyeq}_{M,C_0} \ \hat{C}$, $P^+ \ {}_E\preccurlyeq_M \ \Pi^+$ and $P^- \ {}_E\preccurlyeq_M \ \Pi^-$, then:

- $x\backslash E \sim \frac{P^+}{P^-}$ and $x \ {}_E\preccurlyeq_M \ \alpha$ together imply $\alpha\backslash C \sim \frac{\Pi^+}{\Pi^-} \triangleright (\mathfrak{t}, \omega)$ for some fixed $\mathfrak{t}$ and ω.
- $v\backslash E \sim \frac{P^+}{P^-}$ and $v \ {}_E\preccurlyeq_M \ \tau$ together imply $\tau\backslash C \sim \frac{\Pi^+}{\Pi^-} \triangleright (\mathfrak{t}, \omega)$ for some fixed $\mathfrak{t}$ and ω..

Proof. Let $\hat{C} = (C, \omega, \delta, \eta)$. Then from Definition 6.5, we have $e \ {}_E\preccurlyeq_M \ C$. From $x\backslash E \sim \frac{P^+}{P^-}$, this gives us $\alpha\backslash C \sim \frac{\Pi^+}{\Pi^-}$, by Lemma 4.18 where $P^+ \ {}_E\preccurlyeq_M \ \Pi^+$ and $P^- \ {}_E\preccurlyeq_M \ \Pi^-$. Lemma 6.2 then gives us $\alpha\backslash C \sim \frac{\Pi^+}{\Pi^-} \triangleright (\mathfrak{t}, \omega)$ for some $\mathfrak{t}$ and ω. Similarly for $v\backslash E \sim \frac{P^+}{P^-}$, Lemma 4.18 and Lemma 6.2 together gives us $\tau\backslash C \sim \frac{\Pi^+}{\Pi^-} \triangleright (\mathfrak{t}, \omega)$ for some $\mathfrak{t}$ and ω □

Lemma 6.7. If $E_1 \parallel e_1 \longrightarrow^1 E_2 \parallel e_2$ and $E_1 \parallel e_1 \ {}_{E_1}\dot{\preccurlyeq}_{M_1,C_0} \ \hat{C}_1$, then there exists $\hat{C}_2$ and M_2 such that $\hat{C}_1 \Longrightarrow^1_\delta \hat{C}_2$ where $E_2 \parallel e_2 \ {}_{E_2}\dot{\preccurlyeq}_{M_2,C_0} \ \hat{C}_2$.

Proof. Let $\hat{C}_1 = (C_1, \omega_1, \delta_1, \eta_1)$. From $E_1 \parallel e_1 \longrightarrow^1 E_2 \parallel e_2$, the simulation $E_1 \parallel e_1 \ {}_{E_1}\dot{\preccurlyeq}_{M_1,C_0}$ $\hat{C}_1$ and Definition 6.5, we have $E_1 \parallel e_1 \ {}_{E_1}\preccurlyeq_{M_1} \ C_1$ and $(C_0, \emptyset, \emptyset, \emptyset) \Longrightarrow^*_\delta \hat{C}_1$. Then, from Lemma 4.20, we have C_2 and M_2 such that $C_1 \Longrightarrow^1 C_2$ and $E_2 \parallel e_2 \ {}_{E_2}\preccurlyeq_{M_2} \ C_2$. Then by Lemma 6.3, there exists $\hat{C}_2 = (C_2, \omega_2, \delta_2, \eta_2)$ such that $\hat{C}_1 \Longrightarrow^1_\delta \hat{C}_2$ and $C_2 \ \dot{\approx}_{C_0} \ \hat{C}_2$. By definition $(C_0, \emptyset, \emptyset, \emptyset) \Longrightarrow^*_\delta \hat{C}_2$. From Definition 6.5, this gives us $E_2 \parallel e_2 \ {}_{E_2}\dot{\preccurlyeq}_{M_2,C_0} \ \hat{C}_2$. This completes the proof. □

Lemma 6.8. Let e be the initial program and $C_0 = [\![e]\!]_{\mathrm{E}}$. Then if $e \longrightarrow^* e'$, then $\hat{C} \Longrightarrow^*_\delta \hat{C}'$ where $\hat{C} = (C_0, \emptyset, \emptyset, \emptyset)$ and $e' \ {}_E\dot{\preccurlyeq}_{M,C_0} \ \hat{C}'$.

Proof. From $e \longrightarrow^* e'$ and Lemma 4.21 we have $C_0 \Longrightarrow^* C'$ such that $e' \ {}_E\preccurlyeq_M \ C'$. From Lemma 6.4, it follows that there exists ω, δ and η such that $(C_0, \emptyset, \emptyset, \emptyset) \Longrightarrow^*_\delta (C', \omega, \delta, \eta)$ and $C' \ \dot{\approx}_{C_0} \ (C', \omega, \delta, \eta)$. Then from Definition 6.5, we have $e' \ {}_E\dot{\preccurlyeq}_{M,C_0} \ (C', \omega, \delta, \eta)$. □

6.2 Properties of the Tag Generation System

In this section, we define a set of general, and often obvious, properties related to tag generation that are used in the rest of the document. Our first lemma concerns the monotonicity of Tagged TinyBang Core's constraint closure operation.

Lemma 6.9. The constraint closure operation is monotonic on C, ω, δ and η. That is, if $(C_0, \omega_0, \delta_0, \eta_0) \Longrightarrow_{\delta}^{1} (C_1, \omega_1, \delta_1, \eta_1) \Longrightarrow_{\delta}^{*} (C_n, \omega_n, \delta_n, \eta_n)$, then $C_0 \subseteq C_1 \subseteq C_n$, $\omega_0 \subseteq \omega_1 \subseteq \omega_n$, $\delta_0 \subseteq \delta_1 \subseteq \delta_n$ and $\eta_0 \subseteq \eta_1 \subseteq \eta_n$.

Proof. Follows immediately from the definition of constraint closure in Definition 5.7 and multiple constriant closure steps in Definition 5.8. □

In Section 5.4.2, we discussed the necessity of tracking "feeds" between variables explicitly. A tag closure process was created to "push" tags across the feed boundary in order to mitigate the issue. The next two lemmas concern feeds and their effects on the tag system. Our first lemma effectively asserts that, if there is a feed between two variables, then our tag derivation process ensures that all the tags from the destination side have been pushed to the source.

Lemma 6.10. Let e_0 be the initial expression and let x_1, x_2 be two variables in e_0. Further let $[\![e_0]\!]_{\delta} \Longrightarrow_{\delta}^{*} (C, \omega, \delta, \eta)$ and $M(x_1) \looparrowright M(x_2) \in \eta$. Then, $(\Omega, \Delta) \twoheadleftarrow_M e_0$ implies $\Omega(x_2) \subseteq \Omega(x_1)$.

Proof. Let $(C', \omega', \delta', \eta')$ be the complete closure of $[\![e_0]\!]_{\delta}$ as per Definition 5.13. Given $M(x_1) \looparrowright M(x_2) \in \eta$, by Lemma 6.9, $M(x_1) \looparrowright M(x_2) \in \eta'$. Further from the same definition and Definition 5.10, we have $\omega'(M(x_2)) \subseteq \omega'(M(x_1))$. Finally from Definition

5.16, this gives us $\Omega(x_2) \subseteq \Omega(x_1)$. □

The next lemma explicitly connects TinyBang Core's runtime clauses with the extended type system's collected feeds and tags.

Lemma 6.11. Let e_0 be the initial expression and let $e_0 \longrightarrow^* E_1 \parallel e_1$ and $(\Omega, \Delta) \twoheadleftarrow_M e_0$. Then:

- If the first clause of e_1 has the form: $x = x'$, then $\Omega(x) \subseteq \Omega(x')$.
- If the first clause of e_1 has the form: $x_3 = x_1\ x_2$ and $x_1 = x_4 \texttt{ -> } e_1 \in E_1$, then $\Omega(x_4) \subseteq \Omega(x_2)$.
- If the first clause of e_1 has the form: $x_2 = x_1.l$ and $x_1 = \texttt{\{} \ldots \texttt{,}\ l\texttt{=}x_3\ \texttt{,} \ldots \texttt{\}} \in E_1$, then $\Omega(x_2) \subseteq \Omega(x_3)$.
- If the first clause of e_1 has the form: $x = \texttt{case}\ x'\ \texttt{of}\ \overset{n}{\overrightarrow{p_\square \rightsquigarrow f_\square}}$ and we have $x' = v \in E_1$ and $v \backslash E_1 \sim \frac{\{p_i\}}{\{p_j \mid j<i\}}$ for some $i \leq n$, then $\Omega(x_i) \subseteq \Omega(x')$ given $f_i = x_i \texttt{ -> } e_i$.

Proof. Given $e_0 \longrightarrow^* E_1 \parallel e_1$, from Lemma 6.8, we have $(C_1, \omega_1, \delta_1, \eta_1)$ such that it simulates $E_1 \parallel e_1$. This also gives us $E_1 \parallel e_1\ {}_{E_1}\preccurlyeq_M C_1$.

Suppose the next small step operation exists and has the form: $E_1 \parallel e_1 \longrightarrow^1 E_2 \parallel e_2$; then from Lemma 6.7, we have $(C_2, \omega_2, \delta_2, \eta_2)$ such that $E_2 \parallel e_2\ {}_{E_2}\dot{\preccurlyeq}_{M,C_0}\ (C_2, \omega_2, \delta_2, \eta_2)$. Our next step is to show that for each of the statements in the lemma the small step operation and the corresponding constraint closure step exists and that the specified variable pairs will be included in η_2. This allows us to utilize Lemma 6.10 to demonstrate the subset property of tags.

- Given that the form of the first clause is $x = x'$, the VARIABLE LOOKUP small step rule applies; then from Definition 6.8, we have $\alpha' <: \alpha \in C$ where $\alpha = M_1(x)$ and $\alpha' = M_1(x')$. In this case, the TRANSITIVITY rule from the closure rules in Figure 5.8 applies, giving us $\alpha' \looparrowright \alpha \in \eta_2$. Then, from Lemma 6.10, we have $\Omega(x) \subseteq \Omega(x')$.

- For the second statement, the APPLICATION rule applies. From Definition 6.8, we have $\alpha_1\ \alpha_2 <: \alpha_3 \in C_1$, $\tau|_{\Pi^-}^{\Pi^+} <: \alpha_1$ and $\tau = \alpha_4 \texttt{->} t_1$, such that $\alpha_i = M(x_i)$ for $i \in \{1,2,3,4\}$ and $e_1\ {}_{E_1}\dot{\preccurlyeq}_{M,C_0}\ t_1$. In this case, the APPLICATION rule from the closure rules in Figure 5.8 applies, giving us $\alpha_2 \looparrowright \alpha_4 \in \eta_2$. Then, from Lemma 6.10, we have $\Omega(x_4) \subseteq \Omega(x_2)$.

- The first clause has the form: $x_2 = x_1.l$ and we are given $x_1 = \texttt{\{}\ldots\texttt{,}\ l\texttt{=}x_3\ \texttt{,}\ldots\texttt{\}} \in E_1$. Here the PROJECTION rule applies. From Definition 6.8, we have $\alpha_1.l <: \alpha_2 \in C_1$, $\texttt{\{}\ldots\texttt{,}\ l\texttt{:}\alpha_3\ \texttt{,}\ldots\texttt{\}} <: \alpha_1 \in C_1$ such that $\alpha_i = M(x_i)$ for $i \in \{1,2,3\}$. The PROJECTION closure rule from Figure 5.8 then applies which gives us $\alpha_3 \looparrowright \alpha_2 \in \eta_2$. Then, from Lemma 6.10, we have $\Omega(x_2) \subseteq \Omega(x_3)$.

- For the final statement, the first clause has the form: $x = \texttt{case}\ x'\ \texttt{of}\ \overrightarrow{{}^{n}p_{\square} \rightsquigarrow f_{\square}}$. We are also given $x' = v \in E_1$ and $v\backslash E \sim \frac{\{p_i\}}{\{p_j | j<i\}}$ for some $i \leq n$. Therefore the PATTERN MATCH operational semantics rule applies. Then from Lemma 6.7, we have $\texttt{case}\ \alpha'\ \texttt{of}\ \overrightarrow{{}^{n}\pi_{\square} \rightsquigarrow \phi_{\square}} <: \alpha \in C_1$ and $\tau|_{\Pi^-}^{\Pi^+} <: \alpha \in C_1$ where $\alpha = M(x)$, $\alpha' = M(x')$ and $v\ {}_{E_1}\dot{\preccurlyeq}_{M,C_0}\ \tau|_{\Pi^-}^{\Pi^+}$. Further, for all $i \leq n$, $p_i\ {}_{E_1}\preccurlyeq_M\ \pi_i$ and $f_i\ {}_{E_1}\preccurlyeq_M\ \phi_i$.

  Given the simulation, $v\backslash E_1 \sim \frac{\{p_i\}}{\{p_j | j<i\}}$ and Lemma 4.18, we have $\tau\backslash C_1 \sim \frac{\Pi_1^+}{\Pi_1^-}$ for some fixed Π_1^+ and Π_1^-. By Lemma 6.2, we have $\tau\backslash C_1 \sim \frac{\Pi_1^+}{\Pi_1^-} \triangleright (\mathfrak{t}, \omega)$, for some $\mathfrak{t}$ and ω. Let $\phi_i = \alpha_i \texttt{->} t_i$. Since $f_i\ {}_{E_1}\preccurlyeq_M\ \phi_i$, $\alpha_i = M(x_i)$. The PATTERN MATCH rule from

the closure rules in Figure 5.8 then applies, which gives us $\alpha' \twoheadrightarrow \alpha_i \in \eta_2$. Then, from Lemma 6.10, we have $\Omega(x_i) \subseteq \Omega(x')$.

□

6.3 Properties of the Environment(s)

We have already defined a notion of equivalence between the environments of TinyBang Core and Tagged TinyBang Core in the previous chapter (Definition 5.17). We use this equivalence to define a few properties that relate the data flow in TinyBang Core to tag flow in Tagged TinyBang Core.

Many of the lemmas in this and the next section are conditioned on simultaneously executing both TinyBang Core and Tagged TinyBang Core on the same initial program for a fixed number of steps such that the resulting environments are equivalent (as per Definition 5.17). Informally, we term such pairs of executions, "equivalent" executions.

The next three lemmas are partially statements that the operational semantics rules do not *arbitrarily* throw away tags. Each rule selects all tags that meet a certain criteria and does not "lose" tags thereafter. For example, the next lemma states that, given a pair of equivalent executions, for a record value in the environment, and any record tag formed out of a combination of runtime tags from its fields, if the resultant tag is present in Ω, then it will also be included in the set of runtime tags attached to the record value, irrespective of its genesis.

Lemma 6.12. Let e_0 be the initial program. Let $e_0 \longrightarrow^* E \parallel e$ and $e_0 \xrightarrow[\Omega,\Delta]{}{}^*_\delta \hat{E} \parallel e$ such that $(\Omega, \Delta) \twoheadleftarrow_M e_0$ and $E \ {}_M\dot{\approx}_{e_0} \hat{E}$. Then given $x = \{\ldots,\ l_i = x_i\ ,\ldots\} \in E$ and a list of tags, $\vec{t}$,

where each $\mathfrak{t}_i \in \hat{E}^{\mathfrak{t}}(x_i)$, if $\{\ldots,\ l_i{:}\mathfrak{t}_i\ ,\ldots\} \in \Omega(x)$ then $\{\ldots,\ l_i{:}\mathfrak{t}_i\ ,\ldots\} \in \hat{E}^{\mathfrak{t}}(x)$.

Proof. By induction on the number of steps in the computation of E and then by case analysis on the operational semantics rules.

Suppose the lemma holds at the end of n steps of computation. Further suppose the preconditions all hold. We will show by case analysis that the lemma holds after $n+1$ steps. First observe that any step that *does not* introduce $x = \{\ldots,\ l_i{=}x_i\ ,\ldots\}$ in to the environment trivially satisfies the lemma. Thus we only have to consider the following cases:

- The first clause of e has the form $x = \{\ldots,\ l_i{=}x_i\ ,\ldots\}$. From $E\ {}_M\dot{\approx}_{e_0}\ \hat{E}$, we have $x = \langle\{\ldots,\ l_i{=}x_i\ ,\ldots\}, \overline{\mathfrak{t}}\rangle$. By the tagged operational semantics rules for record values in Figure 5.3, it follows that if $\{\ldots,\ l_i{:}\mathfrak{t}_i\ ,\ldots\} \in \Omega(x)$ then $\{\ldots,\ l_i{:}\mathfrak{t}_i\ ,\ldots\} \in \underline{\overline{\mathfrak{t}}}$.

- The initial clause of e is an assignment clause: i.e. a clause of the form $x = x'$ where $x' = \{\ldots,\ l_i{=}x_i\ ,\ldots\} \in E$. From Lemma 6.11, we have $\Omega(x) \subseteq \Omega(x')$. Then, if $\{\ldots,\ l_i{:}\mathfrak{t}_i\ ,\ldots\} \in \Omega(x)$, it must be the case that it is also in $\Omega(x')$. Further by induction, since $\{\ldots,\ l_i{:}\mathfrak{t}_i\ ,\ldots\} \in \Omega(x')$, we have $\{\ldots,\ l_i{:}\mathfrak{t}_i\ ,\ldots\} \in \hat{E}^{\mathfrak{t}}(x')$. From the tagged operational semantics rules for VARIABLE LOOKUP in Figure 5.3, $\hat{E}^{\mathfrak{t}}(x) = \Omega(x) \cap \hat{E}^{\mathfrak{t}}(x')$. Thus $\{\ldots,\ l_i{:}\mathfrak{t}_i\ ,\ldots\} \in \hat{E}^{\mathfrak{t}}(x)$.

  Similar arguments apply for cases where the initial clause of e is a projection, application or a pattern match.

□

The next lemma is similar to the previous, but about integer tags.

Lemma 6.13. Let e_0 be the initial program. Let e_0 be the initial program. Let $e_0 \longrightarrow^* E \parallel e$ and $e_0 \xrightarrow[\Omega,\Delta]{*}_{\delta} \hat{E} \parallel e$ such that $(\Omega, \Delta) \twoheadleftarrow_M e_0$ and $E \ {}_M\dot{\approx}_{e_0} \ \hat{E}$. Then, given $x = v \in E$ such that $v \in \mathbb{Z}$, if $\texttt{int} \in \Omega(x)$ then $\texttt{int} \in \hat{E}^{\langle\rangle}(x)$.

Proof. By induction on the number of steps in the computation of E and then by case analysis on the operational semantics rules.

Suppose the lemma holds at the end of n steps of computation. Further suppose the preconditions all hold. We will show by case analysis that the lemma holds after $n+1$ steps. First observe that any step that *does not* introduce $x = v$ in to the environment trivially satisfies the lemma. Thus we only have to consider the following cases:

- The first clause of e has the form $x = v$ such that $v \in \mathbb{Z}$. From $E \ {}_M\dot{\approx}_{e_0} \ \hat{E}$, we have $x = \langle v, \overline{\langle\rangle} \rangle \in \hat{E}$. By the tagged operational semantics rules for record values in Figure 5.3, it follows that if $\texttt{int} \in \Omega(x)$ then $\texttt{int} \in \overline{\langle\rangle}$.

- The initial clause of e is an assignment clause: i.e. a clause of the form $x = x'$ where $x' = \texttt{int} \in E$. From Lemma 6.11, we have $\Omega(x) \subseteq \Omega(x')$. Then if $\texttt{int} \in \Omega(x)$, it must be the case that it is also in $\Omega(x')$. Further by induction, since $\texttt{int} \in \Omega(x')$, we have $\texttt{int} \in \hat{E}^{\langle\rangle}(x')$. From the tagged operational semantics rules for VARIABLE LOOKUP in Figure 5.3, $\hat{E}^{\langle\rangle}(x) = \Omega(x) \cap \hat{E}^{\langle\rangle}(x')$. Thus $\texttt{int} \in \hat{E}^{\langle\rangle}(x)$.

  Similar arguments apply for cases where the initial clause of e is a projection, application or a pattern match.

□

The next lemma states that for any value in the environment, if a leaf tag ($\star$) is

present in Ω then it is always selected to be part of the value's runtime tag set.

Lemma 6.14. Let e_0 be the initial program. Let $e_0 \xrightarrow[\Omega,\Delta]{}{}^{*}_{\delta} \hat{E} \parallel e$ such that $(\Omega, \Delta) \twoheadleftarrow_M e_0$. Then, given $x = \langle v, \overline{\mathfrak{t}} \rangle \in \hat{E}$, if $\star \in \Omega(x)$ then $\star \in \overline{\mathfrak{t}}$.

Proof. By induction on the number of steps in the computation of $\hat{E}$ and then by case analysis on the operational semantics rules.

Suppose the lemma holds at the end of n steps of computation. Further suppose the preconditions all hold. We will show by case analysis that the lemma holds after $n+1$ steps. First observe that any step that *does not* introduce $x = \langle v, \overline{\mathfrak{t}} \rangle$ in to the environment trivially satisfies the lemma. For any case where the particular clause gets introduced, the leaf tags are in $\overline{\mathfrak{t}}$ since $\star \in \Omega(x)$ by assumption and all operational semantics rules for Tagged TinyBang Core picks leaf tags. □

6.4 Tag Propogation

In this section we discuss a number of properties of the tag flow entailed by the structure of our tagged operational semantics. Our first lemma shows that a compatibility check in TinyBang Core induces a particular tag that, given an equivalent run of Tagged TinyBang Core, appears in the tag set of the subject value.

Lemma 6.15. Let e_0 be the initial program. Let $e_0 \longrightarrow^* E \parallel e$ and $e_0 \xrightarrow[\Omega,\Delta]{}{}^{*}_{\delta} \hat{E} \parallel e$ such that $(\Omega, \Delta) \twoheadleftarrow_M e_0$ and $E \ {}_M\dot{\approx}_{e_0} \hat{E}$. Further let $E \parallel e \ {}_E\dot{\preccurlyeq}_{M,C_0} \hat{C}$ where $C_0 = [\![e_0]\!]_{\mathrm{E}}$. Then:

- If $x \ {}_E\preccurlyeq_M \alpha$ and $x \backslash E \sim \frac{P^+}{P^-}$ then $\alpha \backslash C \sim \frac{\Pi^+}{\Pi^-} \triangleright (\mathfrak{t}, \omega)$ such that $P^+ \ {}_E\preccurlyeq_M \Pi^+$ and $P^- \ {}_E\preccurlyeq_M \Pi^-$. Further if $\mathfrak{t} \in \Omega(x)$ then $\mathfrak{t} \in \hat{E}^{\mathfrak{t}}(x)$.

- If $x = v \in E$, $v \ {}_{E}\preccurlyeq_{M} \tau$ and $v\backslash E \sim \frac{P^+}{P^-}$ then $\tau\backslash C \sim \frac{\Pi^+}{\Pi^-} \triangleright (\mathfrak{t}, \omega)$ such that $P^+ \ {}_{E}\preccurlyeq_{M} \Pi^+$ and $P^- \ {}_{E}\preccurlyeq_{M} \Pi^-$. Further if $\mathfrak{t} \in \Omega(x)$ then $\mathfrak{t} \in \hat{E}^{\mathfrak{t}}(x)$.

Proof. We proceed by mutual induction on the height of the compatibility proof trees and further by case analysis on the rule used at the root.

Leaf. In this case we have $x\backslash E \sim \frac{\emptyset}{\emptyset}$. From the assumptions we have $x \ {}_{E}\preccurlyeq_{M} \alpha$. From Lemma 6.6, we have $\alpha\backslash C \sim \frac{\emptyset}{\emptyset} \triangleright (\star, \{\alpha \mapsto \star\})$ where $P^+ \ {}_{E}\preccurlyeq_{M} \Pi^+$ and $P^- \ {}_{E}\preccurlyeq_{M} \Pi^-$. From the assumptions of the lemma, we have $\star \in \Omega(x)$. From $E \ {}_{M}\dot{\approx}_{e_0} \hat{E}$, we have $x \in \hat{E}$. By Lemma 6.14, $\star \in \hat{E}^{\mathfrak{t}}(x)$.

Value Selection. From the premises of the rule, given $x\backslash E \sim \frac{P^+}{P^-}$, we have $x = v \in E$ and $v\backslash E \sim \frac{P^+}{P^-}$. From Definition 4.16, there exists $\tau|^{\Pi_1^+}_{\Pi_1^-} <: \alpha \in C$ such that $v \ {}_{E}\preccurlyeq_{M} \tau$. Then, given the strengthening clause, by induction, we have $\tau\backslash C \sim \frac{\Pi^+}{\Pi^-} \triangleright (\mathfrak{t}, \omega)$ such that $P^+ \ {}_{E}\preccurlyeq_{M} \Pi^+$ and $P^- \ {}_{E}\preccurlyeq_{M} \Pi^-$ and if $\mathfrak{t} \in \Omega(x)$ then $\mathfrak{t} \in \hat{E}^{\mathfrak{t}}(x)$.

Record. In this case we have $v\backslash E \sim \frac{P^+}{\bigcup_i {}^{l_i} P_i^-}$ where v has the form $\{\ldots,\ l_i{=}x_i\ ,\ldots\}$. By the assumptions of the lemma, we have $v \ {}_{E}\preccurlyeq_{M} \tau$ where $\tau = \{\ldots,\ l_i : \alpha_i\ ,\ldots\}$ and for all i, $x_i \ {}_{E}\preccurlyeq_{M} \alpha_i$. Then, by Lemma 6.6, we have $\tau\backslash C \sim \frac{\Pi^+}{\bigcup_i {}^{l_i}\Pi_i^-} \triangleright (\mathfrak{t}, \omega)$ where $P^+ \ {}_{E}\preccurlyeq_{M} \Pi^+$ and for all i, $P_i^- \ {}_{E}\preccurlyeq_{M} \Pi_i^-$. Observe that $\mathfrak{t}$ has the form $\{\ldots,\ l_i : \mathfrak{t}_i\ ,\ldots\}$ by inspection of the applicable type compatibility rule.

From the premises of the RECORD rule from Figure 4.4, for all i, $x_i\backslash E \sim \frac{P^+.l_i}{P_i^-}$. Then, by induction, we have for all i, $\alpha_i\backslash C \sim \frac{\Pi^+.l_i}{\bigcup_i {}^{l_i}\Pi_i^-} \triangleright (\mathfrak{t}_i, \omega_i)$. Further, if $\mathfrak{t}_i \in \Omega(x_i)$ then $\mathfrak{t}_i \in \hat{E}^{\mathfrak{t}}(x_i)$.

Finally, from the assumptions of the lemma, we have $\{\ldots,\ l_i : \mathfrak{t}_i\ ,\ldots\} \in \Omega(x)$ and $x = v \in E$. Together, this gives us $\{\ldots,\ l_i : \mathfrak{t}_i\ ,\ldots\} \in \hat{E}^{\mathfrak{t}}(x)$, by Lemma 6.12.

$$\textsc{Slow Step}\quad \frac{E' = E \parallel x = v}{E \parallel x = v \parallel e' \longrightarrow^{1} E' \parallel e'}$$

Figure 6.1: Augmenting TinyBang Core with an extra rule

Integer. In this case $v \backslash E \sim \frac{P^+}{P^-}$ where $v \in \mathbb{Z}$, $P^+ = \{\texttt{int}\}$ and $\texttt{int} \notin P^-$. By the assumptions of the lemma, we have $v \ {}_{E}\preccurlyeq_{M} \ \texttt{int}$. Then by Lemma 6.6, we have $\texttt{int} \backslash C \sim \frac{\Pi^+}{\Pi^-} \triangleright$ where $P^+ \ {}_{E}\preccurlyeq_{M} \ \Pi^+$ and $P^- \ {}_{E}\preccurlyeq_{M} \ \Pi^-$. From the assumptions we have $\texttt{int} \in \Omega(x)$. Further, since $x = v \in E$, from the assumptions, it follows by Lemma 6.13 that $\texttt{int} \in \hat{E}^{\mathfrak{t}}(x)$.

The Rest. The remaining cases are straightforward via induction.

□

When attempting to prove the bisimulation between the two operational semantics, we must be able to compare their steps. There is a minor wrinkle here: there are fewer operational semantics rules in TinyBang Core compared to Tagged TinyBang Core. This is fueled by the fact that the grammar of E is a subset of e. So, for example, there is no rule for clauses of the form $x = 42$ since it is already part of the E grammar. We will augment TinyBang Core with the extra rule in Figure 6.1. The rule is a no-op as far the current semantics of TinyBang Core goes, but it helps align the two operational semantics.

Our next lemma is a formal statement of a property of the tag framework which we have discussed many times in the text: tag sets are non-conflicting at runtime. This is fairly intuitive since, at each step, the rules only select tags that has the same shallow structure. Formally:

Lemma 6.16 (Tags at runtime are non-Conflicting)**.** If $e_0 \longrightarrow_{\delta}^{*} \hat{E} \parallel e_1$ then for all $x =$

$\langle v, \overline{\mathfrak{t}} \rangle \in \hat{E}$, $\overline{\mathfrak{t}}$ is a non-conflicting tag set as per Definition 5.15.

Proof. By induction on the number of steps in computation, followed by case analysis of the operational semantics rules. □

We are finally ready to state the first simulation lemma between the operational semantics.

Lemma 6.17. Let e_0 be the initial program. Further let $e_0 \longrightarrow^* E_1 \parallel e_1$ and $e_0 \xrightarrow[\Omega,\Delta]{}{}^*_\delta \hat{E}_1 \parallel e_1$ such that $(\Omega, \Delta) \twoheadleftarrow_M e_0$ and $E_1 \ {}_M\dot{\approx}_{e_0} \hat{E}_1$. If $E_1 \parallel e_1 \longrightarrow^1 E_2 \parallel e_2$, then there exists $\hat{E}_2$ such that $\hat{E}_1 \parallel e_1 \longrightarrow^1_\delta \hat{E}_2 \parallel e_2$ and $E_2 \ {}_M\dot{\approx}_{e_0} \hat{E}_2$.

Proof. By induction on the number of steps and then by case analysis of the operational semantics rules from Figure 4.5.

Value. Consider the case when e has the form: $x = v \parallel e'$ where $v \in \mathbb{Z}$. Then $E_1 \parallel e \longrightarrow^1 E_2 \parallel e'$ where $E_2 = E_1 \parallel x = v$. Similarly from the INTEGER VALUE rule in Figure 5.3, we have $\hat{E}_1 \parallel e \longrightarrow^1_\delta \hat{E}_2 \parallel e'$ such that $\hat{E}_2 = \hat{E}_1 \parallel x = \langle v, \overline{\mathfrak{t}} \rangle$ for a fixed value of $\overline{\mathfrak{t}}$. From Definition 5.17, $E_2 \ {}_M\dot{\approx}_{e_0} \hat{E}_2$, which completes this case. Similar arguments can be made for other values; i.e. functions and records.

Variable Lookup. When the e has the form: $x_1 = x_2 \parallel e'$, the VARIABLE LOOKUP rule applies. Then $E_1 \parallel e \longrightarrow^1 E_2 \parallel e'$ where $E_1(x_2) = v$ and $E_2 = E_1 \parallel x_1 = v$. Similarly for the Tagged TinyBang Core semantics, we have $\hat{E}_1 \parallel e \longrightarrow^1_\delta \hat{E}_2 \parallel e'$ where $E_1(x_2) = \langle v, \overline{\mathfrak{t}} \rangle$ and $E_2 = E_1 \parallel x_1 = \langle v, \overline{\mathfrak{t}} \cap \Omega(x_1) \rangle$. From Definition 5.17, $E_2 \ {}_M\dot{\approx}_{e_0} \hat{E}_2$, which completes this case.

Observe that the Tagged TinyBang Core rule is very similar to the TinyBang Core

rule in the way it operates on values. The differences only pertain to the propagation of tags which does not affect the equivalence between the environments. Other rules of this ilk are Application and Projection; their cases are similar to the Variable Lookup case. The main difference is in the Pattern Match rule which we consider next.

Pattern Match. Consider the case when e has the form: $x_1 = \texttt{case}\ x_2\ \texttt{of}\ \overset{n}{\overrightarrow{p_\square \rightsquigarrow f_\square}} \parallel e'$. From the premises of the Pattern Match, we have $x_2 = v \in E_1$ and some $i \leq n$, such that $v \backslash E_1 \sim \frac{\{p_i\}}{\{p_j \mid j<i\}}$. Then $E_1 \parallel e_1 \longrightarrow^1 E_2 \parallel e_2$ where we set $E_2 = E_1 \parallel x_i = v$, $e_2 = e_i \parallel x_1 = \mathrm{RV}(e_i) \parallel e'$ and $\boldsymbol{\alpha}(x_1, f_i) = x_i \texttt{ -> } e_i$.

We also have, from Lemma 6.8, $\hat{C}_1$ such that $E_1 \parallel e_1 \ \dot{\preccurlyeq}_{M,C_0} \hat{C}_1$. Further there exists α_2, τ, Π^+ and Π^- such that $x_2 = v \ \preccurlyeq_M \tau\big|^{\Pi^+}_{\Pi^-} <: \alpha_2$. From Definition 4.16, we have $x_2 \preccurlyeq_M \alpha_2$, $v \preccurlyeq_M \tau$, $P^+ \preccurlyeq_M \Pi^+$ and $P^- \preccurlyeq_M \Pi^-$ such that $v \backslash E_1 \sim \frac{P^+}{P^-}$. Lemma 4.19 allows us to combine the two compatibilities in to one: $v \backslash E_1 \sim \frac{P_1^+}{P_1^-}$ where $P_1^+ = P^+ \cup \{p_i\}$ and $P_1^- = P^- \cup \{p_j \mid j < i\}$.

Given $v \preccurlyeq_M \tau$, from Lemma 6.15, this gives us $\alpha \backslash C \sim \frac{\Pi_1^+}{\Pi_1^-} \triangleright (\mathfrak{t}, \omega)$ for some $\mathfrak{t}$ and ω such that $P_1^+ \preccurlyeq_M \Pi_1^+$ and $P_1^- \preccurlyeq_M \Pi_1^-$ which completes the first part of this case.

We have also satisfied all the pre-conditions for the Pattern Match constraint closure rule from Figure 5.8 to apply. Therefore there exists $\hat{C}_2 = (C_2, \omega_2, \delta_2, \eta_2)$ such that $\hat{C}_1 \Longrightarrow^1_\delta \hat{C}_2$. From the rule, $\alpha_2 \mapsto \mathfrak{t} \in \omega_2$ and $(\alpha_1, \mathfrak{t}, i) \in \delta$. Given the monotonicity of closure (Lemma 6.9) and Definition 5.16, $\mathfrak{t} \in \Omega(x_2)$. Then from Lemma 6.15, we can conclude that $\mathfrak{t} \in \hat{E}_1^{\mathfrak{t}}(x_2)$.

Lemma 6.9 and Definition 5.16 also allow us to conclude that $(x_1, \overset{\smile}{\mathfrak{t}}, i) \in \Delta$ *for all* $\overset{\smile}{\mathfrak{t}}$ such that $\overset{\smile}{\mathfrak{t}}$ is a non-conflicting subset of $\Omega(x_2)$ *and* it contains $\mathfrak{t}$. From Lemma

6.16, $\hat{E}_1^{\overline{\tau}}(x_2)$ is non-conflicting and we have already shown that $\overline{\tau}$ is a member of this set. It then follows that, there exists at least one $(x_1, \overline{\overline{\tau}}, i) \in \Delta$, which match the runtime tag set attached to x_2.

The PATTERN MATCH rule for Tagged TinyBang Core, from Figure 5.3, now applies and it gives us $\hat{E}_1 \parallel e_1 \longrightarrow_\delta^1 \hat{E}_2 \parallel e_2$ where $\hat{E}_2 = \hat{E}_1 \parallel x_i = \langle v, \overline{\overline{\tau}} \cap \Omega(x_i) \rangle$, e_2 and $x_i \texttt{->} e_i$ are the same as defined above. From the assumptions, $E_2 \ {}_M\dot{\approx}_{e_0} \ \hat{E}_2$, which concludes this case and the proof. □

6.5 Uniqueness of Dispatch

We have shown, in the previous section, that when TinyBang Core takes a step, so does Tagged TinyBang Core. The section is dedicated to demonstrating the converse. Given that the operational semantics parameter derivation in Definition 5.16 requires a consistent set of constraints, it is clear that TinyBang Core will not be stuck. So we are primarily trying to show the uniqueness of the step.

One of the primary concerns, therefore, is to ensure that there are no conflicting dispatch entries: i.e. for a point x, there does not exist $(x, \overline{\overline{\tau}}, i)$ and $(x, \overline{\overline{\tau}}, j)$ in Δ where $i \neq j$. We are going to chip away at this problem by demonstrating that in certain common situations, the tags generated conflict with each other, which prevents them from appearing together at runtime. First we need a definition of *shallow non-conflict* of types: essentially types that have the same "shape" in a shallow sense.

Definition 6.18 (Type Shallow Non-Conflict)**.** Two types τ and τ' are in shallow non-conflict (the relation $\asymp$) in the following cases:

- $\tau = \texttt{int}$ and $\tau' = \texttt{int}$
- $\tau = \phi$ and $\tau' = \phi$
- τ and τ' are record types and $\textsc{Fields}(\tau) = \textsc{Fields}(\tau')$

In all other cases the types shallow conflict: i.e. $\tau \not\asymp \tau'$.

Our next lemma states that if two types shallow conflict, the corresponding compatibility proofs produce conflicting tags.

Lemma 6.19. Suppose we have $\tau_1 \backslash C \sim \frac{\Pi_1^+}{\Pi_1^-} \triangleright (\mathfrak{t}_1, \omega_1)$ and $\tau_2 \backslash C \sim \frac{\Pi_2^+}{\Pi_2^-} \triangleright (\mathfrak{t}_2, \omega_2)$ and $\tau_1 \not\asymp \tau_2$, then $\mathfrak{t}_1 \not\leftrightsquigarrow \mathfrak{t}_2$

Proof Sketch. By strong induction on the sum of the heights of the two compatibility proof trees and case analysis on the roots of the proof trees. □

Next we argue that two compatibility proof trees, where the set of positive patterns in one overlaps with the set of negative patterns in the other, would produce conflicting tags. The intuition here is that the two proof trees must diverge at some point as one tries to match a pattern and the other attempts to prove a match failure; specifically such a situation can only be resolved by choosing different lower bounds somewhere deep in the proof trees, which, in turn, leads to the situation described in Lemma 6.19. Formally we state the lemma as follows:

Lemma 6.20. Let at least one of the following be true: $\Pi_1^+ \cap \Pi_2^- \neq \emptyset$ or $\Pi_2^+ \cap \Pi_1^- \neq \emptyset$. Then, given $\alpha_1 \backslash C \sim \frac{\Pi_1^+}{\Pi_1^-} \triangleright (\mathfrak{t}_1, \omega_1)$ and $\alpha_2 \backslash C \sim \frac{\Pi_2^+}{\Pi_2^-} \triangleright (\mathfrak{t}_2, \omega_2)$, it must be the case that $\mathfrak{t}_1 \not\leftrightsquigarrow \mathfrak{t}_2$. Further if $\tau_1 \backslash C \sim \frac{\Pi_1^+}{\Pi_1^-} \triangleright (\mathfrak{t}_1, \omega_1)$ and $\tau_2 \backslash C \sim \frac{\Pi_2^+}{\Pi_2^-} \triangleright (\mathfrak{t}_2, \omega_2)$, then $\mathfrak{t}_1 \not\leftrightsquigarrow \mathfrak{t}_2$.

Proof Sketch. By strong induction on the sum of the heights of the two compatibility proof trees and case analysis on the roots of the proof trees. □

Finally we can state our second simulation lemma:

Lemma 6.21. Let e_0 be the initial program. Further let $e_0 \longrightarrow^* E_1 \parallel e_1$ and $e_0 \xrightarrow[\Omega,\Delta]{}{}^*_\delta \hat{E}_1 \parallel e_1$ such that $(\Omega, \Delta) \twoheadleftarrow_M e_0$ and $E_1 \ {}_M\dot{\approx}_{e_0} \hat{E}_1$. If $\hat{E}_1 \parallel e_1 \longrightarrow^1_\delta \hat{E}_2 \parallel e_2$, then there exists E_2 such that $E_1 \parallel e_1 \longrightarrow^1_\delta E_2 \parallel e_2$ and $E_2 \ {}_M\dot{\approx}_{e_0} \hat{E}_2$.

Proof. By induction on the number of steps and then by case analysis of the operational semantics rules from Figure 5.3.

The only interesting case is Pattern Match. All other cases are similar to Lemma 6.17.

Pattern Match. Consider the case when e_1 has the form: $x_1 = \texttt{case}\ x_2\ \texttt{of}\ \overrightarrow{{}^n p_\square \rightsquigarrow f_\square} \parallel e'$. From the premises of the Pattern Match rule in Figure 5.3, we have $x_2 = \langle v, \overline{\mathfrak{t}} \rangle \in \hat{E}_1$ and $(x_1, \overline{\mathfrak{t}}, \iota)$ for some $\iota \leq n$. Then $\hat{E}_1 \parallel e_1 \longrightarrow^1_\delta \hat{E}_2 \parallel e_2$ where we set $\hat{E}_2 = \hat{E}_1 \parallel x_\iota = \langle v, \overline{\mathfrak{t}} \cap \Omega(x_\iota) \rangle$, $e_2 = e_\iota \parallel x_1 = \mathrm{RV}(e_\iota) \parallel e'$ and $\boldsymbol{\alpha}(x_1, f_\iota) = x_\iota \texttt{ -> } e_\iota$.

From Definition 5.16, we must have some $(\alpha_1, \mathfrak{t}, \iota) \in \delta$ where δ is the set of dispatch entries after complete closure.

Given that the program type checks (from Definition 5.16), we know that TinyBang Core must take a step. From $E_1 \ {}_M\dot{\approx}_{e_0} \hat{E}_1$, we have $x_2 = v \in E_1$. From the premises of the corresponding rule, there exists $i \leq n$, such that $v \backslash E_1 \sim \frac{\{p_i\}}{\{p_j \mid j < i\}}$. Then $E_1 \parallel e_1 \longrightarrow^1 E_2 \parallel e_2$ where we set $E_2 = E_1 \parallel x_i = v$, $e_2 = e_i \parallel x_1 = \mathrm{RV}(e_i) \parallel e'$ and $\boldsymbol{\alpha}(x_1, f_i) = x_i \texttt{ -> } e_i$.

We have, from Lemma 6.8, $\hat{C}_1$ such that $E_1 \parallel e_1 \ {}_{E_1}\dot{\preccurlyeq}_{M,C_0} \hat{C}_1$. Leveraging the same argument as in Lemma 6.17, we have $\mathfrak{t}'$ such that $\mathfrak{t}' \in \hat{E}_1^{\mathfrak{t}}(x_2)$, $(\alpha_1, \mathfrak{t}', i) \in \delta$ and

$(x_1, \overline{\mathfrak{t}}, i) \in \Delta$ for some $x_1 \ {}_{E_1}\preccurlyeq_M \alpha_1$.

By Lemma 6.16, $\overline{\mathfrak{t}}$ is a non-conflicting set. It suffices to show that $\iota = i$ given $\mathfrak{t} \leftrightsquigarrow \mathfrak{t}'$. Each dispatch entry in δ is the result of a PATTERN MATCH rule. Therefore we have two proof trees: $\tau \backslash C \sim \frac{\Pi_1^+}{\Pi_1^-} \triangleright (\mathfrak{t}, \omega)$ with $\pi_\iota \in \Pi_1^+$ and $\{\pi_j \mid j < \iota\} \subseteq \Pi_1^-$ and $\tau' \backslash C \sim \frac{\Pi_2^+}{\Pi_2^-} \triangleright (\mathfrak{t}', \omega')$ with $\pi_i \in \Pi_2^+$ and $\{\pi_j \mid j < i\} \subseteq \Pi_2^-$. From Lemma 6.19 and given the non-conflict of tags, $\tau = \tau'$. Further, if $\iota < i$ then $\Pi_1^+ \cap \Pi_2^- \neq \emptyset$ and if not $\Pi_2^+ \cap \Pi_1^- \neq \emptyset$; but by Lemma 6.20 either of these cases would result in conflicting tags. Therefore $i = \iota$, which gives us $E_2 \ {}_M\dot{\approx}_{e_0} \hat{E}_2$, which completes this case and the proof. □

6.6 Proof of Bisimulation

The above lemmas are sufficient to prove the bisimulation between TinyBang Core and Tagged TinyBang Core. We restate the theorem and give its proof here:

Theorem 2 (Bisimulation of the Operational Semantics)**.** Let e_0 be the initial program. Further let $e_0 \longrightarrow^* E_1 \parallel e_1$ and $e_0 \xrightarrow[\Omega,\Delta]{}{}^*_\delta \hat{E}_1 \parallel e_1$ such that $(\Omega, \Delta) \twoheadleftarrow_M e_0$ and $E_1 \ {}_M\dot{\approx}_{e_0} \hat{E}_1$. Then $E_1 \parallel e_1 \longrightarrow^1 E_2 \parallel e_2$ if and only if there exists $\hat{E}_2$ such that $\hat{E}_1 \parallel e_1 \longrightarrow^1_\delta \hat{E}_2 \parallel e_2$ and $E_2 \ {}_M\dot{\approx}_{e_0} \hat{E}_2$.

Proof. Direct from Lemma 6.17 and Lemma 6.21. □

Chapter 7

Implementation

This section describes our proof-of-concept implementation of TinyBang Core. The implementation includes an interpreter for the basic TinyBang Core semantics from Section 4.4 as well as the corresponding type checker from Section 4.5. The Tagged TinyBang Core implementation has the extended typechecker from Section 5.4 as well as a series of interpreters for the tagged semantics, each progressively "closer to the metal" than the previous.

Unfortunately the proof-of-concept does not include a complete compiler. However our more advanced interpreters implement the semantics in a manner closer to the compiler, modulo necessary standard transformations like closure conversion and hoisting. Our focus has been on validating the correctness of our tag framework as opposed to raw performance; so the interpreter implementations are not particularly optimized.

7.1 Interpreters

The adaptive tag framework is complex, even for the simplified semantics of TinyBang Core. So we have chosen to implement a series of interpreters, rather than jumping directly to code generation. The interpreters all implement the same essential semantics from Section 4.4 or the equivalent one from Section 5.3, albeit with different intermediate data structures and tag representations. The primary advantage of this approach is that errors are easier to localize: each interpreter makes only incremental changes to the previous one and since the same tests can run on all of them, it is easier to isolate and debug bugs and regressions. We describe three of the most important interpreters below along with the corresponding type checker implementations.

7.1.1 Standard Interpreter and Typechecker

This is is an implementation of the original operational semantics and type checker for TinyBang Core from Sections 4.4 and 4.5 respectively. This interpreter is the "gold standard" against which we verify all others.

The interpreter directly implements the rules for TinyBang Core with one exception: the implementation of value compatibility is simplified to match against a single pattern; i.e. $x \sim p$ as opposed to the more complex definition from 4.4.1. When evaluating a `case` statement, the implementation of the pattern match rule invokes this rule, once for each pattern, in the sequence dictated by case branches. This simplification is reasonable at runtime since values are union-free and there are no union alignment issues.

The type checker is also a straightforward implementation of the rules. Type

checking proceeds by computing the initial alignment of an expression, performing constraint closure and then verifying consistency.

7.1.2 Naive Interpreter for Tagged TinyBang Core

This simple interpreter for Tagged TinyBang Core is based on the semantics from Section 5.3. Tags are reified individually and values have a set of tags attached to them. Type checking is implemented as per the extended definitions of type compatibility and constraint closure in Section 5.4. The tag universe Ω is derived exactly as described in the section, but the computation of the dispatch table Δ is simplified: the implementation simply uses the collection of $(\alpha, \mathfrak{t}, i)$ obtained from constraint closure, forgoing the collation to tag sets described in Section 5.4.3. Dispatch is then implemented by matching the tag set of the subject value against this set of dispatch entries and jumping to the corresponding branch. The rest of the operational semantics is implemented exactly as described in Section 5.3.

This modification still yields *correct* results given the absence of conflicting tags at runtime and the fact that non-conflicting tags agree on dispatch decisions. The simplified system has the benefit of being straightforward to implement and debug with no complicated pre-computations; so it is a good stepping stone between the standard interpreter and the more complicated table-based interpreters.

7.1.3 Table-based Interpreter for Tagged TinyBang Core

The tag assignment operations in the Tagged TinyBang Core semantics are simply filters on Ω with some set of runtime tags as input; i.e. they are operations from tag sets

to tag sets. Further we are operating on a fixed universe of tags. So many operations can be sped up with some pre-computation. Our advanced interpreters are thus table-based; pre-computed tables attached to each program point and tag assignment and dispatch are implemented as lookups on these tables.

A key feature of our tag based semantics is that runtime tags are always selected from a fixed set; i.e. the set of tags associated with a variable x in the environment is always some subset of $\Omega(x)$. So given a clause like $x' = x$, we can approximate the runtime tag assignment operation by considering each subset of $\Omega(x)$ and applying the tag filtering operation specified by the Variable Lookup rule; in this case that amounts to an intersection operation with $\Omega(x')$. By recording the results in a table, tag assignment at runtime reduces to a simple lookup. The size of this table can be further reduced by observing that tag sets with conflicting tags cannot occur at runtime; we filter out such subsets of $\Omega(x)$ during table creation.

Since the maximal set of tags associated with a program point is finite, so are the subsets. This makes it possible to "index" them; i.e. associate a numeric value with each one. Many viable indexing schemes exist with various costs and benefits. For example, a simple approach is to associate a unique number with a subset based on its contents. With tag set indexing, the pre-computed tables can be rendered as integer-based lookup tables at runtime which has memory and performance benefits.

We implement an "indexed" interpreter for Tagged TinyBang Core based on this idea. Type checking and tag derivation are implemented as described in Section 5.4. The system then computes a set of tables, at least one for each program point, that encode

the tag operation. The operational semantics is implemented to utilize these tables for tag assignment and dispatch.

7.1.4 Validating the Interpreters

To validate the interpreters, we crafted a set of about 15 small programs in the language of TinyBang Core, each ranging between 10 to 120 clauses, that tests most of the features of the language and the tag framework. The suite of tests run on all of our interpreters; it type checks or fails as appropriate and produces the expected runtime values when executed.

7.2 Performance

The long term goal of the current project is to build the core of an efficient scripting language. To this end, we discuss a number of optimizations to the tag framework in Section 9.1. However, the current set of interpreters were built primarily to validate the correctness of the tag framework and not its performance. Nontheless investigating the interpreter performance provides some insight in to the performance characteristics of the tag schema presented in this thesis, even though we do not expect to see particularly great results from micro-benchmarks.

Directly comparing the standard interpreter with, say, the indexed interpreter from the previous section is fraught with issues. The two have very different overheads that are hard to normalize. Instead we ported the standard interpreter to the tag based framework. This interpreter tags values based on shallow tags similar to standard functional

tb

Table 7.1: Comparing the performance of interpreters

Length	Invocations	Iterations	Shallow	Indexed
10	25	100	15.26	15.08
10	50	100	32.37	31.93
10	100	100	67.12	66.70
10	200	100	143.08	141.95
20	25	100	45.78	45.83
20	50	100	96.32	95.99
20	100	100	207.04	206.014
20	200	100	444.14	443.57

language implementations. The indexed interpreter implementation has no optimizations implemented except for one tweak: we have chosen to synchronize tags across feeds; i.e. we ensure that for any feed, $\alpha \nrightarrow \alpha'$, both sides have the same set of tags. The theory only requires that the left hand side be a super set of the right. But synchronizing them has the advantage of eliminating unnecessary "re-tagging" between feeds.

To benchmark the interpreters, we crafted a small series of programs in TinyBang Core, each of which constructs a list and then finds the last element of a list[1], but varying in the length of the list and the number of times the function to find the last element was invoked. We used a micro-benchmarking suite for Ocaml to wrap each interpreter's evaluation function and configured it to run for 100 iterations to ensure that the measured times were significant. The tests were run on a standard desktop machine with an AMD Phenom II processor and 8GB of RAM.

Our results appear in Figure 7.1. The first two columns show the length of the list and the number of invocations of the function in the program respectively. The primary observation is that the indexed interpreter has a small, but mostly consistent performance advantage over the shallow tagged interpreter. The advantage increases slightly as the length

[1]The lists were encoded using records

of the list and the number of invocations increases. This is consistent with theoretical expectations: when the number of destruction operations in a program increases for a fixed number of constructor operations, the adaptive tag based approach tends to perform better. At smaller numbers, the advantage offered by fast dispatch is somewhat mitigated by the overheads introduced by tag assignment at construction. We expect the performance differential to increase in the case of large programs with more complex pattern matching. It is worth noting that the interpreter itself has a very high overhead compared to the programs being run. So a small advantage is likely to translate to larger numbers in a compiled environment.

Chapter 8

Related Work

This chapter summarizes some of the work related to the semantics we presented in the previous chapters. Our primary focus is on features of TinyBang Core though we will touch upon topics relevant to TinyBang as well. For a comprehensive discussion of work related to the latter, see [39].

8.1 Pattern Matching

Pattern matching in programming languages have been around for a long time [17] and shallow tags, in some form, have been used to implement pattern matching almost from the time the feature appeared in functional languages [18, 12]. A significant amount of work has gone in to optimizing pattern matches by building matching automata with carefully chosen heuristics [18, 12, 35, 36]. However there does not appear to have been significant efforts towards extending these schemes towards languages with subtyping.

Clojure's implementation of pattern matching [5] is perhaps the closest: it follows

the broad scheme dictated by [36], but has been expanded to cover more data structures. In particular, it is capable of handling maps with arbitrary keys. During *specialization* (*a la* Maranget) on map patterns, each map is effectively extended to the same size in order to keep the number of columns in the specialized pattern matrix the same. Such an approach can potentially be applied to pattern matching on record subtypes. But it is somewhat tedious in the presence of ad-hoc unions and deep record subtypes.

MLPolyR [15] supports both extensible matching and record extensions, but appears to support pattern matching only among variants and not arbitrary types. Pattern compilation is performed by mapping cases to records of functions and variants to functions that take (converted) cases as input. It is unclear whether this approach can be extended to support arbitrary types. Garrigue explores the implementation of polymorphic variants in [30]. Tags for variants are created by hashing the variant's label; a type error being emitted if there are hash collisions. This scheme is not easily extensible to ad-hoc unions of arbitrary types like in TinyBang Core. Neither system appears to handle deep and wide pattern matches with any degree of efficiency.

8.2 Object-Oriented Dispatch

Dynamic dispatch in object-oriented languages is a well-studied problem. To achieve late binding and dispatch, the runtime system builds and maintains a *dispatching data structure* such that a query can efficiently find the appropriate implementation for a message, based on the dynamic type of the receiver. In a language like C++, virtual function tables (VFT) serve this purpose, supporting both efficient creation and query [53].

This is more challenging in the case of scripting languages, most of which support features like duck typing and are dynamically typed. The simple VFT model is no longer possible. Attribute and method lookup may require a hashing scheme, though some optimization is possible using caches.

If global analysis is possible, as is the case with languages like Smalltalk more sophisticated techniques become applicable. For example, class hierarchy analysis [22], enables the monomorphization of methods which have no overloads. Dispatch, in many cases, can be made more efficient with the use of tables or decision trees. The former involves constructing a large dispatch matrix, with objects on one axis, methods on the other and code pointers as the content and dispatching based on that. Tables are usually sparse and techniques exist to compress them to be more manageable [23]. Decision tree based approaches, on the other hand, encode the potential set of receivers in to an easily searchable binary tree. Dispatch is implemented by walking down this decision tree [56].

TinyBang supports a variant-based object model with the ability to functionally extend objects [40]. OO dispatch, in this model, relies on the underlying pattern matching implementation of TinyBang and can benefit from any optimizations introduced by our tag system. The dispatch semantics of Tagged TinyBang Core is akin to the table based approach above. However our tables are compact and can potentially be elided. Our semantics also incorporates complex pattern matching in to the dispatch system; the feature does not usually apply to traditional OO languages. On the other hand, constructing and extending objects in our system introduces a small overhead for adaptive tag assignment.

8.3 Heterogeneous Cases

Functions in TinyBang can have patterns attached to them. The onioning operator allow such functions to be concatenated in to *compound functions* with support for asymmetric dispatch. This can be considered a generalization of first class cases from MLPolyR [15] where we have eliminated both the requirement of writing branches in CPS as well as the phase distinction between case construction and use. In TinyBang Core, for simplicity, we have chosen to eliminate compound functions and use `case` statements instead. Our `case` statement can be considered a simplification of compound functions where the set of component functions are *fixed*. Unlike traditional ML-style languages, we allow our branches to return differing types and our system is capable of inferring precise types in the vein of conditional constraints [11, 44].

Interestingly the TinyBang Core language does not appear to suffer from the union elimination problem [25] in the absence of conjunction patterns. However our definition is mostly future proof: it uses TinyBang's notion of filtered types to perform union elimination and induce path sensitivity; further the compatibility relation is structured in such a way as to avoid eliminating a given union more than once. Filtered types have some similarities to refinement types [27], but have a somewhat different semantics. The predicates for filtered types are *patterns* which have limited expressiveness compared to the more general predicate expressions common with refinement types. The latter is typically used to augment an existing type system and prove properties about programs. In contrast, filtered types are a fundamental part of our type system and serves to drive our type analysis.

Chapter 9

Future Work

Our long term goal is to improve the safety and performance of scripting languages, especially typed scripting languages. The adaptive tag framework we presented in the previous chapters is a first step towards this goal. In this section we discuss some potential improvements to our theory and discuss how to adapt them to solve other problems in the domain.

9.1 Improving the Tag Framework

A next step for the TinyBang Core project is to implement a code generator for the tagged semantics and investigate its performance on a larger and more diverse set of programs. The semantics implemented by the indexed interpreter is suitable for code generation; but there are a number of potential enhancements that can both simplify the porting and make it more efficient.

9.1.1 Tag Tables to Tag Functions

A key part of the overhead induced by the tag framework is the cost of tag assignment. An observation we made back in Section 3.2.3 was that such operations can potentially be encoded as simple functions which are significantly more efficient than lookup tables. With the indexed interpreter, we have already reached part-way to this scenario: all tag assignments are effectively mappings from integers to integers. Given that we have the ability to chose an indexing scheme to suit our needs, it may be possible to come up with one that has a mapping to simple and efficient functions.

We propose to solve this by encoding it as a satisfiability problem and using an SMT solver. We give a brief overview of this process. Consider a clause like $x_1 = x_2$. The indexed interpreter already has a process to generate a table that maps the tag sets from x_2 to those from x_1. The same process can also be used to generate *symbolic* equations that represent the mapping: i.e. equations of the form $f(\alpha_1) = \beta_1 \ldots f(\alpha_m) = \beta_n$ where f is some, as yet undefined, function and $\alpha_1 \ldots \alpha_m$ represent the tag sets of x_2 and $\beta_1 \ldots \beta_n$ represent the tag sets associated with x_1. We can now set up additional constraints to ensure that $\{\ldots, \alpha_i, \ldots, \beta_i, \ldots\}$ are integers and f is chosen from a set of well-known, efficient, functions. We set up such equations and constraints for all clauses in the program and feed it to an SMT solver. If the solver finds a solution, we can use its generated model to obtain a viable tag assignment. We can fall back to a table-based approach if the solver fails to find a solution.

9.1.2 Tag and Tag Set Elision

In building our lookup tables (and the equations above), we consider all subsets of tags associated with a given variable excluding the conflicting subsets. Even this is something of a conservative approximation of the sets of tags that can appear at runtime. For example, given $\Omega(x) = \{\texttt{int}, \star\}$, the subset $\{\texttt{int}\}$ appears to be a viable subset; but in practice such a tag set does not occur at runtime since any operational semantics rule that assigns the `int` tag will also assign the $\star$ tag. This is a fairly obvious issue that can be easily patched up as a special case. However, in general, a more precise tracking of tags can help reduce the subsets we have to consider.

An advantage of reducing this count is that lookup tables become much more compact. The reduction in the number of tag sets also translates to fewer constraints on the SMT solver and should make tag function inference (as described in the previous section) more likely to succeed. Finally if the cardinality reduces to one for any variable, we can completely elide the lookup table (and tag function) and directly assign the tag set to the value at runtime. Reducing the number of tag sets offer similar advantages with regards to the dispatch table as well. If the cardinality reduces to one, the test can be fully elided and the code inlined.

A first step here is to implement a call-site polymorphism scheme in our type system. Polymorphism "pries apart" a number of type variables that appear conjoined in a monomorphic system. Correspondingly this splits the tag sets associated with each variable which in turn reduces the number of tag sets we have to consider for each point and offers better opportunities for tag elision. It is also worth considering a more *flow*

```
let xtnd r = fun q * (‘A _ * ‘B x) -> q & ‘D x &
             fun r * (‘B _ * ‘C y) -> r & ‘D y
in
let pick = choose { ‘A 3 & ‘B ‘C 0 } { ‘A 2 & ‘B 1 & ‘C 0}
                  { ‘B 5 & ‘C 4 } { ‘B 2 & ‘D 7 & ‘C 4}
in
xtnd pick
```

Listing 9.1: Layout Example

sensitive tracking of data, similar to [41], for extra precision.

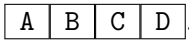
9.2 Onions and Data Layout

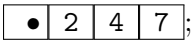
One of our long term goals is to produce an efficient implementation of the TinyBang language. In the appendix we have presented the basic tag framework for TinyBang. Most of augmentations we proposed in the previous section apply directly to that as well. But the flexible nature of onions in TinyBang pose some interesting challenges to optimization.

Consider the code in Listing 9.1. Let `choose` be a four-argument function that arbitrarily selects a record from its input and returns. This program typechecks since for any possible value of `pick`, there is a case branch that matches it. Any layout scheme that we choose must be able to perform these operations in a reasonably fast and memory efficient manner.

The canonical view of onions is as a binary tree where each interior node is an onion. Data and functions reside at the leaves. Reifying onions in this form is easy to implement and supports efficient construction, but has performance penalties if data accesses dominate construction. In such cases it is much more preferable to have a “flat” layout such that data

can be easily accessed via offsets. However this is not straightforward in the presence of unions; at runtime, each "node" in the onion, can be one of a number of different types.

A naive solution would be to represent each record as a pointer to a block of the form 

. Each slot stores the contents of the corresponding label and there are as many slots as labels in the program. With this approach the last parameter to `choose` above will look like this: 

; Such a layout provides efficient access, but is a significant waste of memory when the number of labels are large and records sparse.

A better solution is to tune the layout to each onion. However since an onion can have totally different internal structures at runtime, it is necessary to detect its current "form" before operations. As we discussed in Section 3.3.2, adaptive tags can be used to detect the deep structure of values at runtime. By pre-computing layouts corresponding to each viable runtime tag set, it is straightforward to implement data access for onions.

This approach has the advantage that it does not preclude non-flat layouts. This additional flexibility is warranted: while flat layouts are extremely efficient in terms of data access, in a language which encourages flexible structures and easy concatenation like TinyBang, the copy penalty they incur can be prohibitive.

In general, the choice of data layout, given the nature of data structures that abound in a scripting language, is significantly more complex than in a traditional statically typed language [24, 20]. We propose to pose general layout as an optimization problem. Starting from an initial, possibly sub-optimal, boxed, binary tree layout, we incrementally improve it via a series of well-defined layout transformation that maintain the soundness of the overall system. For example, the initial representation can be flattened partially or

completely to improve access efficiency. Even when flattening is not a choice, layouts can still be reordered to reduce the cost of frequently accessing specific fields. The choice and sequence of operations can be determined by heuristics, by program analysis or by gathering runtime profile information.

Chapter 10

Conclusions

Scripting languages have found widespread acceptance in both industry and academia due to their flexibility and ease of use. However they offer few safety guarantees: most such languages are dynamically typed, or at best gradually typed, and rely primarily on documentation and ambient knowledge to help programmers write correct code. The lack of static type information is often a significant handicap on the scalability of these languages: in addition to making them hard to debug, their performance also tends to suffer at scale.

Previous work from our lab has led to the design of TinyBang, a language with expressiveness similar to scripting languages, but statically typed. In this dissertation we presented a framework, based on the notion of *adaptive tags*, to exploit the analyzability of typed scripting languages like TinyBang, with the goal of producing efficient implementations. Our framework addresses common data layout and dispatch related issues that hamper the performance of scripting languages: it is capable of precisely tracking structural types across scripting programs and dispatching efficiently based on them; we have

also outlined a scheme for utilizing the framework to compute data layouts.

We formalized the framework based on a simplified calculus, TinyBang Core. We first defined the operational semantics and type system for the language. We then introduced our framework in three steps: we defined an efficient operational semantics based on the notion of adaptive tags; we then presented an extended type checker capable of generating tag information for a program and finally we discussed how to efficiently implement the tagged semantics. We then showed that the alternative semantics is equivalent to the original TinyBang Core semantics by demonstrating a *bisimulation* between the two.

We have presented our proof-of-concept implementation and discussed our validation strategy as well as results from some rudimentary benchmarking. While a significant amount of optimization work, both standard and specialized to TinyBang Core, is required to make the language truly efficient, preliminary results are promising. The implementation based on the adaptive tag framework performs slightly, but consistently better than our reference implementation.

Appendix A

Formalizing Tagged TinyBang

In this appendix, we formalize a tagged variant of the TinyBang language, which we call Tagged TinyBang.

A.1 Grammar

We present the original TinyBang grammar in Figure A.1. This is same as Figure 3.1 of [39]. Our extensions to the grammar are in Figure A.2.

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

Figure A.1: TinyBang ANF Grammar

$$
\begin{array}{rcll}
\mathfrak{t} & ::= & () \mid \pi \to t \mid \star \mid l\ \mathfrak{t} \mid \mathfrak{t}\ \&\ \mathfrak{t} & \textit{tag} \\
\Omega & ::= & x \mapsto \underbrace{\mathfrak{t}} & \textit{var tag map} \\
\beta & ::= & \underbrace{x \mapsto \wp} & \textit{binding} \\
\wp & ::= & \overrightarrow{\rho} & \textit{binding path} \\
\rho & ::= & \diamond \mid \leftarrow \mid \rightarrow \mid \downarrow & \textit{path element} \\
\Delta & ::= & \underbrace{\mathfrak{d}} & \textit{dispatch table} \\
\mathfrak{d} & ::= & (x, \underbrace{\mathfrak{t}}, \underbrace{\mathfrak{t}}, \wp, \beta) & \textit{dispatch table entry}
\end{array}
$$

Figure A.2: Extensions to the TinyBang grammar

$$
\begin{array}{rcll}
\hat{E} & ::= & \overrightarrow{x = \hat{v}} & \textit{environments} \\
\hat{v} & ::= & \langle v, \underbrace{\mathfrak{t}} \rangle & \textit{tagged values}
\end{array}
$$

Figure A.3: Extensions to the TinyBang Core grammar

A.2 Operational Semantics

Defining the operational semantics requires a slightly extended grammar, which we define in Figure A.3

We define a well-formedness criteria for $\hat{E}$ below in the same vein as Definition 5.1.

Definition A.1 (Well-formed Tagged Environment)**.** An expression $\hat{E}$ is considered well-formed if it meets the following criteria:

- $\hat{E}$ is closed.
- Each variable is bound in at most one clause.

It is convenient to define a set of lookup operations on the environment:

Definition A.2 (Tagged TinyBang Environment Lookup)**.** If $\hat{E}$ be a well-formed environment, then:

- $\hat{E}(x) = \hat{v}$ if and only if $x = \hat{v} \in \hat{E}$.

- $\hat{E}^{v}(x) = v$ if and only if $x = \langle v, \overline{\mathfrak{t}} \rangle \in \hat{E}$.
- $\hat{E}^{\mathfrak{t}}(x) = \overline{\mathfrak{t}}$ if and only if $x = \langle v, \overline{\mathfrak{t}} \rangle \in \hat{E}$.

Many of our operational semantics rules need to select leaf tags from Ω. Further many of the semantics rules "trim" the tag set down to what is available. We define two simple functions for that purpose:

Definition A.3 (Leaf Tag Selection)**.** We let $\textsc{Leaf}(\Omega, x) = \{\star \mid \star \in \Omega(x)\}$.

Definition A.4 (Tag Trimming)**.** We let $\textsc{Trim}(\Omega, x, \overline{\mathfrak{t}}) = (\overline{\mathfrak{t}} \cap \Omega(x)) \cup \textsc{Leaf}(x)$

We now define a lookup operation, that takes as input a variable and a path, and returns the value obtained by traversing the path with respect to the variable.

Definition A.5 (Tagged TinyBang lookup operation)**.** We define $\textsc{Lookup}(\hat{E}, x, \wp) = \hat{v}$ as follows:

$$\textsc{Lookup}(\hat{E}, x, \diamond) = \hat{v} \quad \text{iff} \quad x = \hat{v} \in \hat{E}$$

$$\textsc{Lookup}(\hat{E}, x, \downarrow \parallel \wp) = \hat{v} \quad \text{iff} \quad x = \langle l\ x', \overline{\mathfrak{t}} \rangle \in \hat{E} \wedge \textsc{Lookup}(\hat{E}, x', \wp) = \hat{v}$$

$$\textsc{Lookup}(\hat{E}, x, \leftarrow \parallel \wp) = \hat{v} \quad \text{iff} \quad x = \langle x' \,\&\, x'', \overline{\mathfrak{t}} \rangle \in \hat{E} \wedge \textsc{Lookup}(\hat{E}, x', \wp) = \hat{v}$$

$$\textsc{Lookup}(\hat{E}, x, \rightarrow \parallel \wp) = \hat{v} \quad \text{iff} \quad x = \langle x' \,\&\, x'', \overline{\mathfrak{t}} \rangle \in \hat{E} \wedge \textsc{Lookup}(\hat{E}, x'', \wp) = \hat{v}$$

We can then define an operation that utilizes this lookup to generate bindings corresponding to pattern matches:

Definition A.6 (Tagged TinyBang bind operation)**.** We define $\textsc{Bind}(\Omega, \hat{E}, x_1, \beta)$ to be equal to $[x = \hat{v} \mid x \mapsto \wp \in \beta \wedge \langle v, \overline{\mathfrak{t}} \rangle = \textsc{Lookup}(\hat{E}, x_1, \wp) \wedge \hat{v} = \langle v, \textsc{Trim}(\Omega, x, \overline{\mathfrak{t}}) \rangle]$

We can now formally define the small step relation for the system as well as the multi-step evaluation process:

$$\textsc{Function Value}\quad \frac{\hat{v} = \langle p \texttt{ -> } e, \overline{\mathfrak{t}} \rangle \qquad \overline{\mathfrak{t}} = \{\mathfrak{t} \mid \mathfrak{t} = [\![p \texttt{ -> } e]\!]_{\mathrm{E}} \wedge \mathfrak{t} \in \Omega(x)\} \cup \textsc{Leaf}(\Omega, x)}{\hat{E} \parallel [x = p \texttt{ -> } e] \parallel e \longrightarrow^1_\delta \hat{E} \parallel [x = \hat{v}] \parallel e}$$

$$\textsc{Empty Onion}\quad \frac{\hat{v} = \langle \texttt{()}, \overline{\mathfrak{t}} \rangle \qquad \overline{\mathfrak{t}} = \{\texttt{()} \mid \texttt{()} \in \Omega(x)\} \cup \textsc{Leaf}(\Omega, x)}{\hat{E} \parallel [x = \texttt{()}] \parallel e \longrightarrow^1_\delta \hat{E} \parallel [x = \hat{v}] \parallel e}$$

$$\textsc{Label}\quad \frac{\hat{v} = \langle l\ x_0, \overline{\mathfrak{t}} \rangle \qquad \overline{\mathfrak{t}} = \{l\ \mathfrak{t}' \in \Omega(x_1) \mid \mathfrak{t}' \in \hat{E}^{\mathfrak{t}}(x_0)\} \cup \textsc{Leaf}(\Omega, x_1)}{\hat{E} \parallel [x_1 = l\ x_0] \parallel e \longrightarrow^1_\delta \hat{E} \parallel [x_1 = \hat{v}] \parallel e}$$

$$\textsc{Onion}\quad \frac{\hat{v} = \langle x_0 \texttt{ \& } x_1, \overline{\mathfrak{t}} \rangle \qquad \overline{\mathfrak{t}} = \{\mathfrak{t}' \texttt{ \& } \mathfrak{t}'' \in \Omega(x_2) \mid \mathfrak{t}' \in \hat{E}^{\mathfrak{t}}(x_0) \wedge \mathfrak{t}'' \in \hat{E}^{\mathfrak{t}}(x_1)\} \cup \textsc{Leaf}(x_2)}{\hat{E} \parallel [x_2 = x_0 \texttt{ \& } x_1] \parallel e \longrightarrow^1_\delta \hat{E} \parallel [x_2 = \hat{v}] \parallel e}$$

$$\textsc{Assignment}\quad \frac{x_0 = \langle v, \overline{\mathfrak{t}} \rangle \in \hat{E} \qquad \overline{\mathfrak{t}'} = \textsc{Trim}(\Omega, x_1, \overline{\mathfrak{t}}) \qquad \hat{v} = \langle v, \overline{\mathfrak{t}'} \rangle}{\hat{E} \parallel [x_1 = x_0] \parallel e \longrightarrow^1_\delta \hat{E} \parallel [x_1 = \hat{v}] \parallel e}$$

$$\textsc{Application}\quad \frac{\begin{array}{c} \mathfrak{t}_f \in \hat{E}^{\mathfrak{t}}(x_0) \qquad \mathfrak{t}_p \in \hat{E}^{\mathfrak{t}}(x_1) \\ (x_2, \mathfrak{t}_f, \mathfrak{t}_p, \wp, \beta) \in \Delta \qquad \langle p \texttt{ -> } e', \_ \rangle = \textsc{Lookup}(\hat{E}, x_0, \wp) \qquad \hat{E}' = \textsc{Bind}(\Omega, \hat{E}, x_1, \beta) \\ \varsigma = \boldsymbol{\alpha}(x_2, -) \qquad \varsigma' = \varsigma\big|_{\hat{E}' \parallel e'} \qquad e'' = \varsigma'(e') \qquad \hat{E}'' = \textsc{BFresh}(\varsigma', \hat{E}') \end{array}}{\hat{E} \parallel [x_2 = x_0\ x_1] \parallel e \longrightarrow^1_\delta \hat{E} \parallel \hat{E}'' \parallel e'' \parallel [x_2 = \textsc{RV}(e'')] \parallel e}$$

Figure A.4: Tagged TinyBang operational semantics

Definition A.7 (Tagged TinyBang Small Step Semantics)**.** We let $\hat{E} \parallel e \xrightarrow[\Omega,\Delta]{}{}^1_\delta \hat{E}' \parallel e'$ be the relationship satisfying the rules in Figure A.4.

A.3 Tag Derivation

A.3.1 Type Compatibility

We first extend our grammar slightly in Figure A.5 to include ω, a multi-map from type variables to tags and another multi-map from type variables to type variables to track

$$\begin{array}{rcll} \omega & ::= & \underbrace{\alpha \mapsto \text{ϯ}}_{\alpha \twoheadrightarrow \alpha} & \textit{type tag multi-map} \\ \eta & ::= & & \textit{feeds} \end{array}$$

Figure A.5: Type compatibility extensions for Tagged TinyBang type grammar

the "feeds" as discussed in Section 5.4.2.

We also define an operator for manipulating prepending to a collections of paths:

Definition A.8. We let $\rho \boxplus \beta = \{\alpha \mapsto (\rho \parallel \wp) \mid \alpha \mapsto \wp \in \beta\}$

Our formal definition of type compatibility is similar to Definition 5.6 from Tagged TinyBang Core:

Definition A.9. We let $\alpha \backslash C \sim \frac{\Pi^{\Phi}/\Pi^{+}}{\Pi^{-}} \Mapsto C' \triangleright (\text{ϯ}, \omega, \beta, \eta)$ and $\tau \backslash C \sim_{\alpha} \frac{\Pi^{\Phi}/\Pi^{+}}{\Pi^{-}} \Mapsto C' \triangleright (\text{ϯ}, \omega, \beta, \eta)$ be the mutually defined relations satisfying the rules in Figure A.6.

The compatibility rules are an extension of the TinyBang type compatibility rules from Figure 3.14 in [39]. The rules follow the same principles as in Figure 5.5; but TinyBang also supports bindings which must be tracked. Further bindings introduce discontinuities; so feeds must be tracked as well.

A.3.2 Pattern Matching

In TinyBang, the left hand side of an application is a compound function. The semantics inspects the compound function, in a left-to-right order, and dispatches to the first function that matches the function. The TinyBang type system has a corresponding set of Type Application Matching rules (Chapter 3, Figure 3.15). We extend these rules to also track tag information. Our formal definition of type application pattern matching is defined as follows:

$$\textsc{Leaf}\quad \alpha\backslash C \sim \tfrac{\emptyset/\emptyset}{\emptyset} \Mapsto \emptyset \triangleright (\star, \{\alpha \mapsto \star\}, \emptyset, \emptyset)$$

$$\textsc{Type Selection}\quad \frac{\tau\big|_{\Pi_1^-}^{\Pi_1^+} <: \alpha \in C \qquad \tau\backslash C \sim_\alpha \frac{\Pi^\Phi/\Pi_1^+\cup\Pi_2^+}{\Pi_1^-\cup\Pi_2^-} \Mapsto C' \triangleright (\mathfrak{t}, \omega, \beta, \eta) \qquad \omega' = \omega \uplus \{\alpha \mapsto \mathfrak{t}\}}{\alpha\backslash C \sim \frac{\Pi^\Phi/\Pi_2^+}{\Pi_2^-} \Mapsto C' \triangleright (\mathfrak{t}, \omega', \beta, \eta)}$$

$$\textsc{Binding}\quad \frac{\begin{array}{c}\tau\backslash C \sim_{\mathring{\alpha}} \frac{\Pi^\Phi/\Pi^+}{\Pi^-} \Mapsto C' \triangleright (\mathfrak{t}, \omega, \beta, \eta) \qquad C'' = \{\tau\big|_{\Pi^-}^{\Pi^\Phi\cup\Pi^+} <: \alpha \mid \alpha\backslash\Psi \in \Pi^\Phi\} \\ \beta' = \{\alpha \mapsto [\Diamond] \mid \alpha\backslash\Psi \in \Pi^\Phi\} \qquad \eta' = \{\mathring{\alpha} \curvearrowright \alpha \mid \alpha\backslash\Psi \in \Pi^\Phi\}\end{array}}{\tau\backslash C \sim_{\mathring{\alpha}} \frac{\Pi^\Phi/\Pi^+}{\Pi^-} \Downarrow C' \cup C'' \triangleright (\mathfrak{t}, \omega, \beta \cup \beta', \eta \cup \eta')}$$

$$\textsc{Empty Onion Pattern}\quad \frac{\tau\backslash C \sim_{\mathring{\alpha}} \frac{\Pi_1^\Phi/\Pi_1^+}{\Pi^-} \Downarrow C' \triangleright (\mathfrak{t}, \omega, \beta, \eta) \qquad \Pi_2^\Phi \cup \Pi_2^+ = \{() \backslash \Psi\}}{\tau\backslash C \sim_{\mathring{\alpha}} \frac{\Pi_1^\Phi\cup\Pi_2^\Phi/\Pi_1^+\cup\Pi_2^+}{\Pi^-} \Mapsto C' \triangleright (\mathfrak{t}, \omega, \beta, \eta)}$$

$$\textsc{Conjunction Pattern}\quad \frac{\begin{array}{c}\tau\backslash C \sim_{\mathring{\alpha}} \frac{\Pi_1^\Phi/\Pi_1^+}{\Pi_1^-} \Downarrow C' \triangleright (\mathfrak{t}, \omega, \beta, \eta) \qquad \Pi_2^\Phi \subseteq \{\alpha_1' * \alpha_2'\backslash\Psi \mid \alpha_1'\backslash\Psi \in \Pi_1^\Phi \wedge \alpha_2'\backslash\Psi \in \Pi_1^\Phi\} \\ \Pi_2^+ \subseteq \{\alpha_1' * \alpha_2'\backslash\Psi \mid \alpha_1'\backslash\Psi \in \Pi_1^+ \wedge \alpha_2'\backslash\Psi \in \Pi_1^+\} \\ \Pi_2^- \subseteq \{\alpha_1' * \alpha_2'\backslash\Psi \mid \alpha_1'\backslash\Psi \in \Pi_1^- \vee \alpha_2'\backslash\Psi \in \Pi_1^-\}\end{array}}{\tau\backslash C \sim_{\mathring{\alpha}} \frac{\Pi_1^\Phi\cup\Pi_2^\Phi/\Pi_1^+\cup\Pi_2^+}{\Pi_1^-\cup\Pi_2^-} \Mapsto C' \triangleright (\mathfrak{t}, \omega, \beta, \eta)}$$

$$\textsc{Conjunction Weakening}\quad \frac{\begin{array}{c}\tau\backslash C \sim_{\mathring{\alpha}} \frac{\Pi_1^\Phi\cup\Pi_2^\Phi/\Pi_1^+\cup\Pi_2^+}{\Pi_1^-\cup\Pi_2^-} \Downarrow C' \triangleright (\mathfrak{t}, \omega, \beta, \eta) \qquad \Pi_2^\Phi \subseteq \{\alpha_i\backslash\Psi \mid i \in \{1,2\} \wedge \alpha_1 * \alpha_2\backslash\Psi \in \Pi_1^\Phi\} \\ \Pi_2^+ \subseteq \{\alpha_i\backslash\Psi \mid i \in \{1,2\} \wedge \alpha_1 * \alpha_2\backslash\Psi \in \Pi_1^+\} \\ \Pi_2^- \subseteq \{\alpha_i\backslash\Psi \mid i \in \{1,2\} \wedge \alpha_1 * \alpha_2\backslash\Psi \in \Pi_1^-\}\end{array}}{\tau\backslash C \sim_{\mathring{\alpha}} \frac{\Pi_1^\Phi/\Pi_1^+}{\Pi_1^-} \Downarrow C' \triangleright (\mathfrak{t}, \omega, \beta, \eta)}$$

$$\textsc{Empty Onion}\quad \frac{\textsc{CtrPat}(\Pi^-)}{() \backslash C \sim \frac{\emptyset/\emptyset}{\Pi^-} \Mapsto \emptyset \triangleright (() , \emptyset, \emptyset, \emptyset)}$$

$$\textsc{Function}\quad \frac{\textsc{CtrPat}(\Pi^-)}{(\pi \to t)\backslash C \sim \frac{\emptyset/\emptyset}{\Pi^-} \Mapsto \emptyset \triangleright (\pi \to t, \emptyset, \emptyset, \emptyset)}$$

$$\textsc{Onion}\quad \frac{\begin{array}{c}\alpha_1\backslash C \sim \frac{\Pi_1^\Phi/\Pi_1^+}{\Pi^-\cup\Pi_2^\Phi\cup\Pi_2^+} \Mapsto C_1' \triangleright (\mathfrak{t}_1, \omega_1, \beta_1, \eta_1) \qquad \alpha_2\backslash C \sim \frac{\Pi_2^\Phi/\Pi_2^+}{\Pi^-} \Mapsto C_2' \triangleright (\mathfrak{t}_2, \omega_2, \beta_2, \eta_2) \\ \textsc{CtrPat}(\Pi_1^\Phi \cup \Pi_2^\Phi \cup \Pi_1^+ \cup \Pi_2^+ \cup \Pi^-) \qquad \beta = (\leftarrow \boxplus \beta_1) \cup (\rightarrow \boxplus \beta_2)\end{array}}{\alpha_1 \,\&\, \alpha_2\backslash C \sim_{\mathring{\alpha}} \frac{\Pi_1^\Phi\cup\Pi_2^\Phi/\Pi_1^+\cup\Pi_2^+}{\Pi^-} \Mapsto C_1' \cup C_2' \triangleright (\mathfrak{t}_1 \,\&\, \mathfrak{t}_2, \omega_1 \uplus \omega_2, \beta, \eta_1 \cup \eta_2)}$$

$$\textsc{Label}\quad \frac{\alpha\backslash C \sim \frac{\Pi^\Phi/\Pi^+}{\Pi_1^-} \Downarrow C' \triangleright (\mathfrak{t}, \omega, \beta, \eta) \qquad \textsc{CtrPat}(\Pi_2^-) \qquad \{l\ \alpha'\backslash\Psi \in \Pi_2^-\} = \emptyset}{l\ \alpha\backslash C \sim_{\mathring{\alpha}} \frac{l\,\Pi^\Phi/l\,\Pi^+}{(l\,\Pi_1^-)\cup\Pi_2^-} \Mapsto C' \triangleright (l\ \mathfrak{t}, \omega, \downarrow \boxplus \beta, \eta)}$$

Figure A.6: Tagged TinyBang type compatibility relation

FUNCTION MATCH

$$\frac{\tau|^{\Pi^+}_{\Pi^-} <: \alpha_0 \in C \qquad \tau = \pi \to \alpha' \backslash C' \qquad \textsc{Sensible}(\tau, \Pi^+, \Pi^-, C) \qquad \alpha_1 \backslash C \sim \frac{\{\pi\}/\emptyset}{\Pi} \Downarrow C'' \triangleright (\mathfrak{t}_p, \omega_p, \beta, \eta) \qquad \mathfrak{t} = \tau \qquad \omega = \{\alpha_0 \mapsto \mathfrak{t}\} \cup \omega_p \qquad \wp = [\diamond]}{\alpha_0\ \alpha_1\ {}^{\Pi}_{C}\!\!\rightsquigarrow^{\{\pi\}}_{\bullet}\ \alpha' \backslash C' \mathbin{@} C'' \triangleright (\mathfrak{t}, \mathfrak{t}_p, \wp, \omega, \eta, \beta)}$$

FUNCTION MISMATCH

$$\frac{\tau|^{\Pi^+}_{\Pi^-} <: \alpha_0 \in C \qquad \tau = \pi \to \alpha' \backslash C' \qquad \textsc{Sensible}(\tau, \Pi^+, \Pi^-, C) \qquad \alpha_1 \backslash C \sim \frac{\emptyset/\emptyset}{\Pi \cup \{\pi\}} \Downarrow \emptyset \triangleright (\mathfrak{t}_p, \emptyset, \emptyset, \emptyset) \qquad \mathfrak{t} = \tau \qquad \omega = \{\alpha_0 \mapsto \mathfrak{t}\}}{\alpha_0\ \alpha_1\ {}^{\Pi}_{C}\!\!\rightsquigarrow^{\{\pi\}}_{\circ}\ \alpha' \backslash \emptyset \mathbin{@} \emptyset \triangleright (\mathfrak{t}, \mathfrak{t}_p, \emptyset, \omega, \emptyset, \emptyset)}$$

NON-FUNCTION

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

ONION LEFT

$$\frac{\begin{array}{c}\tau|^{\Pi^+}_{\Pi^-} <: \alpha_0 \in C \\ \tau = \alpha_2 \,\&\, \alpha_3 \qquad \textsc{Sensible}(\tau, \Pi^+, \Pi^-, C) \qquad \alpha_2\ \alpha_1\ {}^{\Pi_1}_{C}\!\!\rightsquigarrow^{\Pi_2}_{\bullet}\ \alpha' \backslash C' \mathbin{@} C'' \triangleright (\mathfrak{t}, \mathfrak{t}_p, \wp, \omega, \eta, \beta) \\ \mathfrak{t}' = \mathfrak{t} \,\&\, \star \qquad \omega' = \omega \cup \{\alpha_0 \mapsto \mathfrak{t}'\} \qquad \wp' = (\leftarrow \parallel \wp)\end{array}}{\alpha_0\ \alpha_1\ {}^{\Pi_1}_{C}\!\!\rightsquigarrow^{\Pi_2}_{\bullet}\ \alpha' \backslash C' \mathbin{@} C'' \triangleright (\mathfrak{t}', \mathfrak{t}_p, \wp', \omega', \eta, \beta)}$$

ONION RIGHT

$$\frac{\begin{array}{c}\tau|^{\Pi^+}_{\Pi^-} <: \alpha_0 \in C \\ \tau = \alpha_2 \,\&\, \alpha_3 \qquad \textsc{Sensible}(\tau, \Pi^+, \Pi^-, C) \qquad \alpha_2\ \alpha_1\ {}^{\Pi_1}_{C}\!\!\rightsquigarrow^{\Pi_2}_{\circ}\ \alpha'_2 \backslash C'_2 \mathbin{@} C''_2 \triangleright (\mathfrak{t}, \_, \emptyset, \omega_1, \emptyset, \emptyset) \\ \alpha_3\ \alpha_1\ {}^{\Pi_1 \cup \Pi_2}_{C}\!\!\rightsquigarrow^{\Pi_3}_{\odot}\ \alpha'_3 \backslash C'_3 \mathbin{@} C''_3 \triangleright (\mathfrak{t}', \mathfrak{t}_p, \wp', \omega_2, \eta, \beta) \\ \mathfrak{t}'' = \mathfrak{t} \,\&\, \mathfrak{t}' \qquad \omega' = \omega_1 \cup \omega_2 \cup \{\alpha_0 \mapsto \mathfrak{t}''\} \qquad \wp'' = (\rightarrow \parallel \wp')\end{array}}{\alpha_0\ \alpha_1\ {}^{\Pi_1}_{C}\!\!\rightsquigarrow^{\Pi_2 \cup \Pi_3}_{\odot}\ \alpha'_3 \backslash C'_3 \mathbin{@} C''_3 \triangleright (\mathfrak{t}'', \mathfrak{t}_p, \wp'', \omega', \eta, \beta)}$$

Figure A.7: Tagged TinyBang type application matching

Definition A.10. We let $\alpha_0\ \alpha_1\ {}^{\Pi^+}_{C}\!\!\rightsquigarrow^{\Pi^-}_{\odot}\ \alpha' \backslash C' \mathbin{@} C'' \triangleright (\mathfrak{t}, \mathfrak{t}_p, \wp, \omega, \eta, \beta)$ be the relation satisfying the rules in Figure A.7.

A.3.3 Constraint Closure

For defining the constraint closure, we need to extend the grammar slightly. The extra non-terminals are listed in Figure A.8. As in the case of Tagged TinyBang Core, the extended constraint closure tracks four pieces of data: the set of constraints, the set of tags

$$\begin{array}{rcll} \delta & ::= & \overbrace{(\alpha, \lightning, \lightning, \wp, \beta)} & \textit{dispatch data} \\ \hat{C} & ::= & (C, \omega, \delta, \eta) & \textit{augmented constraints} \end{array}$$

Figure A.8: Constraint Closure extensions for Tagged TinyBang type grammar

$$\textsc{Transitivity}\quad \frac{\{\tau|^{\Pi^+}_{\Pi^-} <: \alpha_1, \alpha_1 <: \alpha_2\} \subseteq C \qquad \eta' = \eta \uplus \{\alpha_1 \rightsquigarrow \alpha_2\}}{(C, \omega, \delta, \eta) \Longrightarrow^*_\delta (C \cup \{\tau|^{\Pi^+}_{\Pi^-} <: \alpha_2\}, \omega, \delta, \eta')}$$

$$\textsc{Application}\quad \frac{\begin{array}{c} \alpha_0\ \alpha_1 <: \alpha_2 \in C \qquad \alpha_0\ \alpha_1 \overset{\emptyset}{\underset{C}{\rightsquigarrow}}{}^{\Pi}_{\odot}\ \alpha_1'\backslash C_1'\ @\ C_1''\ \triangleright\ (\lightning_f, \lightning_p, \wp, \omega', \eta', \beta) \\ \sigma = \Phi(\alpha_2, -) \qquad \sigma' = \sigma|_{C_1' \cup C_1''} \qquad \alpha_2'\backslash C_2' = \sigma'(\alpha_1'\backslash C_1') \\ C_2'' = \textsc{TFresh}(\sigma', C_1'') \qquad \delta' = \delta \cup \{(\alpha_2, \lightning_f, \lightning_p, \wp, \beta)\} \qquad \omega'' = \omega \cup \omega' \qquad \eta'' = \eta \cup \eta' \end{array}}{(C, \omega, \delta, \eta) \Longrightarrow^*_\delta (C \cup C_2' \cup C_2'' \cup \{\alpha_2' <: \alpha_2\}, \omega'', \delta', \eta'')}$$

Figure A.9: Tagged TinyBang constraint closure

discovered so far, a set of dispatch entries and a collection of feeds. We formally define the relation below:

Definition A.11 (Tagged TinyBang Core Constraint Closure)**.** We let $\hat{C} \Longrightarrow^1_\delta \hat{C}'$ be the relationship satisfying the rules in Figure A.9.

We also define the notion of multiple constraint closure steps:

Definition A.12 (Tagged TinyBang Multiple Constraint Closure Steps)**.** We define $\hat{C}_0 \Longrightarrow^*_\delta \hat{C}_n$ if and only if $\hat{C}_0 \Longrightarrow^1_\delta \hat{C}_1 \Longrightarrow^1_\delta \cdots \Longrightarrow^1_\delta \hat{C}_n$

We define a tag-closure operation on $\hat{C}$ which pushes tags back to the source of the feed.

Definition A.13 (Tag Closure)**.** We let $\hat{C} \downarrow \hat{C}'$ be the relation that satisfies the rules in Figure A.10.

For convenience, we define an extended version of the initial derivation for Tagged

$$\frac{\alpha' \looparrowright \alpha \in \eta \qquad \mathfrak{t} \in \omega(\alpha) \qquad \omega' = \omega \uplus \{\alpha' \mapsto \mathfrak{t}\}}{(C, \omega, \delta, \eta) \downarrow (C, \omega', \delta, \eta)}$$

Figure A.10: Tagged TinyBang Tag Closure

TinyBang:

Definition A.14 (Tagged TinyBang Initial Derivation)**.** We let $[\![e]\!]_\delta = (C, \emptyset, \emptyset, \emptyset)$ if and only if $\alpha \backslash C = [\![e]\!]_{\mathbb{E}}$.

Definition A.15 (Tagged TinyBang Complete Closure)**.** We let $\hat{C}_n$ be a complete closure of $\hat{C}_0$ if and only if there exists $\hat{C}_m$ which is the closure of $\hat{C}_0$ with respect to $\Longrightarrow^1_\delta$ and $\hat{C}_n$ is the closure of $\hat{C}_m$ with respect to $\downarrow$.

A.3.4 Deriving the Parameters

Once again we first define the notion of non-conflicting tags:

Definition A.16 (Non-Conflicting Tags)**.** Two tags $\mathfrak{t}$ and $\mathfrak{t}'$ are non-conflicting ($\mathfrak{t} \leftrightsquigarrow \mathfrak{t}'$) if any of the following hold:

- $\mathfrak{t} = \mathfrak{t}'$
- $\mathfrak{t} = \star$ or $\mathfrak{t}' = \star$
- $\mathfrak{t} = l\mathfrak{t}''$ and $\mathfrak{t}' = l\mathfrak{t}'''$ and $\mathfrak{t}'' \leftrightsquigarrow \mathfrak{t}'''$
- $\mathfrak{t} = \mathfrak{t}_1 \,\&\, \mathfrak{t}_2$ and $\mathfrak{t}' = \mathfrak{t}'_1 \,\&\, \mathfrak{t}'_2$ and $\mathfrak{t}_1 \leftrightsquigarrow \mathfrak{t}'_1$ and $\mathfrak{t}_2 \leftrightsquigarrow \mathfrak{t}'_2$

Tag sets that occur at runtime cannot have conflicting tags in them. We define the notion of non-conflicting tag sets formally:

Definition A.17 (Non-Conflicting Tag Sets)**.** A tag set $\overline{\mathfrak{t}}$ is non-conflicting if and only if for any $\{\mathfrak{t}_1, \mathfrak{t}_2\} \in \overline{\mathfrak{t}}$, $\mathfrak{t}_1 \nleftrightsquigarrow \mathfrak{t}_2$.

Since the source of our data is the type system, our tables are all currently keyed on type variables. However the operational semantics parameters Ω and Δ are expected to be keyed on program variables. So we need a mapping between the two. Assuming such a mapping, we can finally define the two parameters for the operational semantics:

Definition A.18 (Parameters for Tagged TinyBang Operational Semantics)**.** Given an expression e and a mapping from program variables to type variables, M, let the complete closure of $[\![e]\!]_\delta$ as per Definition A.15 be $(C, \omega, \delta, \eta)$. Further let C be consistent. Then the parameters are derived as follows:

- $\Omega = \{x \mapsto \overline{\mathfrak{t}} \mid M(x) \mapsto \overline{\mathfrak{t}} \in \omega\}$.
- Let α be a program point with an application lower bound. Let $t_f(\alpha)$ be the applicable tags on the left hand side of this application and let $t_p(\alpha)$ be the applicable tags for the right hand side; that is: $t_f(\alpha) \triangleq \omega(\alpha_1)$ and $t_p(\alpha) \triangleq \omega(\alpha_2)$ if and only if $\alpha_1\ \alpha_2 <: \alpha \in C$. Then $\Delta = \{x \mapsto (\alpha, \overline{\mathfrak{t}_f}, \overline{\mathfrak{t}_p}, \wp, \beta) \mid \exists M(x) \mapsto (\alpha, \overline{\mathfrak{t}_f}, \overline{\mathfrak{t}_p}, \wp, \beta) \in \delta.\, \exists \overline{\mathfrak{t}_f}.\, \exists \overline{\mathfrak{t}_p}.\, \mathfrak{t}_f \in \overline{\mathfrak{t}_f} \wedge \mathfrak{t}_p \in \overline{\mathfrak{t}_p} \wedge \overline{\mathfrak{t}_f} \subseteq t_f(\alpha) \wedge \overline{\mathfrak{t}_p} \subseteq t_p(\alpha) \wedge \overline{\mathfrak{t}_f}$ and $\overline{\mathfrak{t}_p}$ are both non-conflicting$\}$

We denote the derivation of parameters Ω and Δ from e as the operation $(\Delta, \Omega) \twoheadleftarrow_M e$.

A.4 The Bisimulation Property

As in the case of Tagged TinyBang Core, our ultimate goal is to show that Tagged TinyBang bisimulates TinyBang: i.e. the two systems operate in lock-step and performs

all the same computations. In this section, we present a theorem to this effect as well as a series of supporting lemmas. Our structure largely follows Chapter 6.

To compare the two systems, we first need to define a notion of equivalence between the environments similar to what we did for Tagged TinyBang Core:

Definition A.19. $E \ {}_M\dot{\approx}_{e_0} \ \hat{E}$ if and only if $e_0 \longrightarrow^* E \parallel e$ for some e and there exists $(\Omega, \Delta) \twoheadleftarrow_M e_0$ such that $e_0 \xrightarrow[\Omega,\Delta]{*}_{\delta} \hat{E} \parallel e$ and the following conditions are met:

- $|E| = |\hat{E}| = n$.
- For all $i \leq n$, $E(x_i) = \hat{E}^v(x_i)$

Then the theorem below states that if we execute a program using both the TinyBang and the Tagged TinyBang semantics, each system executes a small step if and only the other does. Further if their environments were equivalent at the start, they remain equivalent after the step. The formal statement is as follows:

Theorem 3 (Bisimulation of the TinyBang Operational Semantics)**.** Let e_0 be the initial program. Further let $e_0 \longrightarrow^* E_1 \parallel e_1$ and $e_0 \xrightarrow[\Omega,\Delta]{*}_{\delta} \hat{E}_1 \parallel e_1$ such that $(\Omega, \Delta) \twoheadleftarrow_M e_0$ and $E_1 \ {}_M\dot{\approx}_{e_0} \ \hat{E}_1$. Then $E_1 \parallel e_1 \longrightarrow^1 E_2 \parallel e_2$ if and only if there exists $\hat{E}_2$ such that $\hat{E}_1 \parallel e_1 \longrightarrow^1_{\delta} \hat{E}_2 \parallel e_2$ and $E_2 \ {}_M\dot{\approx}_{e_0} \ \hat{E}_2$.

We present the lemmas in the following sections.

A.4.1 Bisimulation of Type Systems

The constraint closure relation of the extended type system is very similar to that of TinyBang. We now define a bisimulation relation between the two:

Definition A.20. We let $C \dot{\approx}_{C_0} (C, \omega, \delta, \eta)$ if and only if $(C_0, \emptyset, \emptyset, \emptyset) \Longrightarrow^*_{\delta} (C, \omega, \delta, \eta)$

We use this relation to define a few lemmas that relate the two type systems. We start with the bisimulation of the two compatibility relations:

Lemma A.21. $\alpha\backslash C \sim \frac{\Pi^{\Phi}/\Pi^{+}}{\Pi^{-}} \Mapsto C'$ if and only if there exists fixed values of $\mathfrak{t}$, ω, β and η such that $\alpha\backslash C \sim \frac{\Pi^{\Phi}/\Pi^{+}}{\Pi^{-}} \Mapsto C' \triangleright (\mathfrak{t}, \omega, \beta, \eta)$

The relation $\alpha\backslash C \sim \frac{\Pi^{\Phi}/\Pi^{+}}{\Pi^{-}} \Mapsto C'$ is defined in Chapter 3 of [39] as Definition 3.26.

We define a similar relation for application matching as well.

Lemma A.22. $\alpha_0\ \alpha_1\ {}^{\Pi_1}_{C}\!\rightsquigarrow^{\Pi_2}_{\odot}\ \alpha'\backslash C' \mathbin{@} C''$ if and only if there exists fixed values of $\mathfrak{t}_f$, $\mathfrak{t}_p$, ω, β and η such that $\alpha_0\ \alpha_1\ {}^{\Pi_1}_{C}\!\rightsquigarrow^{\Pi_2}_{\odot}\ \alpha'\backslash C' \mathbin{@} C'' \triangleright (\mathfrak{t}_f, \mathfrak{t}_p, \wp, \omega, \eta, \beta)$.

The relation $\alpha_0\ \alpha_1\ {}^{\Pi_1}_{C}\!\rightsquigarrow^{\Pi_2}_{\odot}\ \alpha'\backslash C' \mathbin{@} C''$ is specified by Definition 3.29 of [39].

For every TinyBang constraint closure, there is a corresponding closure in Tagged TinyBang. We define two lemmas to this effect.

Lemma A.23. Let $C_1 \mathrel{\dot{\approx}}_{C_0} \hat{C}_1$. Then $C_1 \Longrightarrow^1 C_2$ if and only if $\hat{C}_1 \Longrightarrow^1_{\delta} \hat{C}_2$ such that $C_2 \mathrel{\dot{\approx}}_{C_0} \hat{C}_2$.

Lemma A.24. Given an initial constraint set C_0, $C_0 \Longrightarrow^* C$ if and only if there exists ω, δ and η such that $(C_0, \emptyset, \emptyset, \emptyset) \Longrightarrow^*_{\delta} (C, \omega, \delta, \eta)$ and $C \mathrel{\dot{\approx}}_{C_0} (C, \omega, \delta, \eta)$.

The constraint closure operation $\Longrightarrow^1$ and $\Longrightarrow^*$ are given by Definitions 3.33 and 3.34 of [39].

This bisimulation allows us to define a simulation relation between the small step semantics of TinyBang and the Tagged TinyBang type closure relation and also define a lemma asserting the existence of the simulation.

Definition A.25 (Simulation)**.** We let $e\ {}_{E}\dot{\preccurlyeq}_{M,C_0}\ \hat{C}$ if and only if $\hat{C} = (C, \omega, \delta, \eta)$, $e\ {}_{E}\preccurlyeq_{M}\ C$ and $C \mathrel{\dot{\approx}}_{C_0} \hat{C}$. The simulation relation $e\ {}_{E}\preccurlyeq_{M}\ C$, is defined as per Definition 4.2 in [39].

Lemma A.26. If $E_1 \parallel e_1 \longrightarrow^1 E_2 \parallel e_2$ and $E_1 \parallel e_1 \;{}_{E_1}\dot{\preccurlyeq}_{M_1,C_0} \hat{C}_1$, then there exists $\hat{C}_2$ and M_2 such that $\hat{C}_1 \Longrightarrow^1_\delta \hat{C}_2$ where $E_2 \parallel e_2 \;{}_{E_2}\dot{\preccurlyeq}_{M_2,C_0} \hat{C}_2$.

A.4.2 Properties of the Tag Generation System

In this section we define general properties related to tag generation. Our first lemma concerns the monotonicity of Tagged TinyBang Core's constraint closure operation.

Lemma A.27. The constraint closure operation is monotonic on C, ω, δ and η. That is, if $(C_0, \omega_0, \delta_0, \eta_0) \Longrightarrow^1_\delta (C_1, \omega_1, \delta_1, \eta_1) \Longrightarrow^*_\delta (C_n, \omega_n, \delta_n, \eta_n)$, then $C_0 \subseteq C_1 \subseteq C_n$, $\omega_0 \subseteq \omega_1 \subseteq \omega_n$, $\delta_0 \subseteq \delta_1 \subseteq \delta_n$ and $\eta_0 \subseteq \eta_1 \subseteq \eta_n$.

A.4.3 Properties of the Environments

As in Chapter 6, we define a series of lemmas that state that the operational semantics rules do not throw away tags. They are conceptually similar to the lemmas from Section 6.3, but are for TinyBang specific entities like labeled values and onions.

Lemma A.28. Let e_0 be the initial program. Let $e_0 \longrightarrow^* E \parallel e$ and $e_0 \xrightarrow[\Omega,\Delta]{}{}^*_\delta \hat{E} \parallel e$ such that $(\Omega, \Delta) \twoheadleftarrow_M e_0$ and $E \;{}_M\dot{\approx}_{e_0} \hat{E}$. Then given $x = l\ x' \in E$ and $\mathfrak{t} \in \hat{E}^{\cdot\mathfrak{t}}(x')$, then $l\ \mathfrak{t} \in \Omega(x)$ implies $l\ \mathfrak{t} \in \hat{E}^{\cdot\mathfrak{t}}(x)$.

Lemma A.29. Let e_0 be the initial program. Let $e_0 \longrightarrow^* E \parallel e$ and $e_0 \xrightarrow[\Omega,\Delta]{}{}^*_\delta \hat{E} \parallel e$ such that $(\Omega, \Delta) \twoheadleftarrow_M e_0$ and $E \;{}_M\dot{\approx}_{e_0} \hat{E}$. Then given $x = x_1 \,\&\, x_2 \in E$ and $\mathfrak{t}_1 \in \hat{E}^{\cdot\mathfrak{t}}(x_1)$ and $\mathfrak{t}_2 \in \hat{E}^{\cdot\mathfrak{t}}(x_2)$, then $\mathfrak{t}_1 \,\&\, \mathfrak{t}_2 \in \Omega(x)$ implies $\mathfrak{t}_1 \,\&\, \mathfrak{t}_2 \in \hat{E}^{\cdot\mathfrak{t}}(x)$.

Our final lemma in this section states that if a leaf tag ($\star$) is present in Ω then it

$$\textsc{Slow Step}\quad \frac{E' = E \parallel x = v}{E \parallel x = v \parallel e' \longrightarrow^1 E' \parallel e'}$$

Figure A.11: Augmenting TinyBang with an extra rule

is always selected to be part of the value's runtime tag set.

Lemma A.30. Let e_0 be the initial program. Let $e_0 \xrightarrow[\Omega,\Delta]{}{}^*_\delta \hat{E} \parallel e$ such that $(\Omega, \Delta) \twoheadleftarrow_M e_0$. Then, given $x = \langle v, \overline{\mathfrak{t}} \rangle \in \hat{E}$, if $\star \in \Omega(x)$ then $\star \in \overline{\mathfrak{t}}$.

A.4.4 Tag Propagation

The next lemma shows that a compatibility check in TinyBang induces a tag that, given an equivalent run of Tagged TinyBang, appears in the tag set of the subject value.

Lemma A.31. Let e_0 be the initial program. Let $e_0 \longrightarrow^* E \parallel e$ and $e_0 \xrightarrow[\Omega,\Delta]{}{}^*_\delta \hat{E} \parallel e$ such that $(\Omega, \Delta) \twoheadleftarrow_M e_0$ and $E \ {}_M\!\approx_{e_0} \hat{E}$. Further let $E \parallel e \ {}_E\!\dot{\preccurlyeq}_{M,C_0} \hat{C}$ where $C_0 = [\![e_0]\!]_{\mathrm{E}}$. Then:

- If $x \ {}_E\!\preccurlyeq_M \alpha$ and $x \backslash E \sim \frac{P^\Phi / P^+}{P^-} \Phi E'$ then $\alpha \backslash C \sim \frac{\Pi^\Phi / \Pi^+}{\Pi^-} \Phi C' \triangleright (\mathfrak{t}, \omega, \beta, \eta)$ such that $P^\Phi \ {}_E\!\preccurlyeq_M \Pi^\Phi$, $P^+ \ {}_E\!\preccurlyeq_M \Pi^+$ and $P^- \ {}_E\!\preccurlyeq_M \Pi^-$. Further if $\mathfrak{t} \in \Omega(x)$ then $\mathfrak{t} \in \hat{E}^{\mathfrak{t}}(x)$.
- If $x = v \in E$, $v \ {}_E\!\preccurlyeq_M \tau$ and $v \backslash E \sim \frac{P^\Phi / P^+}{P^-} \Phi E'$ then $\alpha \backslash C \sim \frac{\Pi^\Phi / \Pi^+}{\Pi^-} \Phi C' \triangleright (\mathfrak{t}, \omega, \beta, \eta)$ such that $P^\Phi \ {}_E\!\preccurlyeq_M \Pi^\Phi$, $P^+ \ {}_E\!\preccurlyeq_M \Pi^+$ and $P^- \ {}_E\!\preccurlyeq_M \Pi^-$. Further if $\mathfrak{t} \in \Omega(x)$ then $\mathfrak{t} \in \hat{E}^{\mathfrak{t}}(x)$.

Comparing TinyBang and TinyBang Core operational semantics suffers from the same issue as comparing TinyBang Core and Tagged TinyBang Core in Chapter 6: the former has fewer rules than the latter. We introduce a no-op semantics rule to align the two in Figure A.11.

The next lemma states that tags are non-conflicting at runtime.

Lemma A.32 (Tags at runtime are non-Conflicting)**.** If $e_0 \longrightarrow_\delta^* \hat{E} \parallel e_1$ then for all $x = \langle v, \overline{\mathfrak{t}} \rangle \in \hat{E}$, $\overline{\mathfrak{t}}$ is a non-conflicting tag set as per Definition A.17.

And finally we can state the first simulation lemma:

Lemma A.33. Let e_0 be the initial program. Further let $e_0 \longrightarrow^* E_1 \parallel e_1$ and $e_0 \xrightarrow[\Omega,\Delta]{}{}_\delta^* \hat{E}_1 \parallel e_1$ such that $(\Omega, \Delta) \twoheadleftarrow_M e_0$ and $E_1 \; {}_M\dot{\approx}_{e_0} \; \hat{E}_1$. If $E_1 \parallel e_1 \longrightarrow^1 E_2 \parallel e_2$, then there exists $\hat{E}_2$ such that $\hat{E}_1 \parallel e_1 \longrightarrow_\delta^1 \hat{E}_2 \parallel e_2$ and $E_2 \; {}_M\dot{\approx}_{e_0} \; \hat{E}_2$.

A.4.5 Uniqueness of Dispatch

This section develops a series of lemmas aimed at showing that for every step taken by Tagged TinyBang there is a corresponding step in TinyBang. Given a consistent set of constraints, TinyBang will not be stuck. So we are effectively trying to show the uniqueness of the step. Demonstrating this for Tagged TinyBang is *significantly* harder than for Tagged TinyBang Core due to the high degree of non-determinism in the type compatibility and application matching rules. We divide the problem in to more manageable chunks:

- Our first set of lemmas attempt to reduce the effect of this non-determinism by demonstrating that these proof trees still have some well-defined sub-structures.
- The next set of lemmas state that compatibility proof trees on the same data, producing non-conflicting tags, will match the same function and produce identical set of bindings. We state a similar property for application matching as well.
- Finally we state the second simulation lemma - that for every step taken by Tagged TinyBang there is a corresponding step in TinyBang.

Our first lemma states that there are no compatibility proof trees where the negative and positive pattern sets overlap.

Lemma A.34 (No Compatibility with Conflicting Patterns)**.** If $(\Pi^{\Phi} \cup \Pi^{+}) \cap \Pi^{-} \neq \emptyset$, then there are no type compatibility proof trees of the form $\alpha \backslash C \sim \frac{\Pi^{\Phi}/\Pi^{+}}{\Pi^{-}} \Varid C' \triangleright (\mathfrak{t}, \omega, \beta, \eta)$.

Before we move on to the next lemma, we need to introduce a few definitions. The first one is that of *well-formed* pattern sets. Palmer [39] defined the notion of well-formed patterns: roughly patterns that do not have free variables and have a unique lower bound per variable. We use this definition as a basis to define well-formed pattern sets.

Definition A.35. A pattern set Π is *well-formed* if and only if for all $\{\pi, \pi'\} \subseteq \Pi$ both π and π' are well-formed patterns and any type variable α that appears as an upper bound to a filter constraint in π does not appear as an upper bound to a filter constraint in π'.

For example, a pattern set: $\{\alpha_1 \backslash \Psi, \alpha_2 \backslash \Psi\}$ is well-formed, however the pattern $\{\alpha_1 * \alpha_2 \backslash \Psi, \alpha_1 \backslash \Psi, \alpha_2 \backslash \Psi\}$ is not.

Well-formedness of pattern sets is a strong property which we cannot maintain throughout a compatibility check: during the conjunction elimination process, the pattern sets are not necessarily well-formed. However a weaker form is sufficient to prove some initial properties. But this needs an additional definition - that of BindVars. This is essentially the set of pattern variables that are expected to bind at the current position.

Definition A.36 (Binding Variables)**.** We let BindVars be defined by the rules below:

$$\textsc{BindVars}(\alpha \backslash \Psi) = \{\alpha\} \cup \textsc{BindVars}(\alpha_1 \backslash \Psi) \cup \textsc{BindVars}(\alpha_2 \backslash \Psi) \quad \text{if} \quad \alpha_1 * \alpha_2 <: \alpha \in \Psi$$

$$\textsc{BindVars}(\alpha \backslash \Psi) = \{\alpha\} \quad \text{if} \quad \alpha_1 * \alpha_2 \not<: \alpha \in \Psi$$

Definition A.37 (Binding Variables on Pattern Sets)**.** We extend BindVars to operate on pattern sets: i.e. $\textsc{BindVars}(\Pi) = \bigcup\{\textsc{BindVars}(\alpha\backslash\Psi) \mid \alpha\backslash\Psi \in \Pi\}$

This allows us to give a definition of *weakly well-formed* pattern sets:

Definition A.38 (Weakly Well-Formed Pattern Sets)**.** Given a pattern set Π, let $\Pi' = \{\alpha\backslash\Psi \mid \alpha'\backslash\Psi \in \Pi \wedge l\alpha <: \alpha' \in \Psi\}$. Then Π is weakly well-formed if and only if all of the following conditions are met:

- Each pattern in Π is well-formed
- No element of $\textsc{BindVars}(\Pi)$ appears as an an upper bound in Π'
- Π' is weakly well-formed

One of the primary sources of non-determinism in type compatibility is conjunction elimination: not only are there multiple rules that can apply at each point, it is also possible to repeatedly apply a series of rules without "making progress". This is problematic when attempting to prove properties between compatibility proofs (say, their equivalence under some conditions). So our first task is to work around this by showing that for any compatibility proof tree, there is a related proof tree where all conjunction patterns and superflous empty onion patterns have been eliminated. Note that this proof tree is not guaranteed to be fully deterministic; it can still contain repeated applications of certain rules; however in the absence of conjunction patterns they are much easier to reason about.

First a note on notation: we write $\alpha_1 * \alpha_2 \not<: \alpha$ to indicate $\nexists \alpha_1, \alpha_2.\, \alpha_1 * \alpha_2 <: \alpha$ in the set under consideration.

We now define the Crunch operation that deeply destructs a conjunction pattern

and eliminates empty onion patterns:

Definition A.39 (Crunching Patterns)**.** We let CRUNCH be defined by following rules:

$$\begin{aligned}
\textsc{Crunch}(() \backslash \Psi) &= \emptyset \\
\textsc{Crunch}(\alpha_1 * \alpha_2 \backslash \Psi) &= \textsc{Crunch}(\alpha_1 \backslash \Psi) \cup \textsc{Crunch}(\alpha_2 \backslash \Psi) \\
\textsc{Crunch}(\alpha \backslash \Psi) &= \{\alpha \backslash \Psi\} \text{ (for all other cases)}
\end{aligned}$$

We overload the above definition of CRUNCH to sets of patterns as follows:

Definition A.40 (Crunching Pattern Sets)**.** We extend CRUNCH such that it works over pattern sets: i.e. $\textsc{Crunch}(\Pi) = \bigcup\{\textsc{Crunch}(\alpha \backslash \Psi) \mid \alpha \backslash \Psi \in \Pi\}$

Conjunction patterns in the negative space introduce a more subtle challenge. It is sufficient to show anti-match against *some* sub-component(s) and not all of them. We will capture this using a slightly different relation SHRED. First a definition of SHRED for patterns:

Definition A.41 (Shredding Patterns)**.** We let SHRED be defined by the rules below:

$$\begin{aligned}
\textsc{Shred}(\alpha_1 * \alpha_2 \backslash \Psi) &= \textsc{Shred}(\alpha_1 \backslash \Psi) \cup \textsc{Shred}(\alpha_2 \backslash \Psi) \\
\textsc{Shred}(\alpha \backslash \Psi) &= \{\alpha \backslash \Psi\} \text{ (for all other cases)}
\end{aligned}$$

We now extend SHRED to *sets* of patterns. Notice that $\textsc{Shred}(\Pi)$ is a relation and not a function.

Definition A.42 (Shredding Pattern Sets)**.** We extend SHRED such that it works over pattern sets:

$$\textsc{Shred}(\{\pi_1 \cdots \pi_n\}, \Pi') \text{ iff } \exists \Pi_1 \cdots \Pi_n. \forall 1 \leq i \leq n. \Pi_i \subseteq \textsc{Shred}(\pi_i) \wedge \Pi_i \neq \emptyset \wedge \Pi' = \bigcup_n \Pi_i$$

We now define predicates that indicate whether further CRUNCH is needed:

Definition A.43 (Crunchable Pattern Sets)**.** We define CanCrunch by the rules below:

$$\textsc{CanCrunch}(\Pi) \quad \text{iff} \quad \begin{cases} \exists \alpha \backslash \Psi \in \Pi.\ \texttt{()} <: \alpha \in \Psi \text{ or} \\ \\ \exists \alpha \backslash \Psi \in \Pi.\ \alpha_1 * \alpha_2 <: \alpha \in \Psi \text{ for some } \alpha_1, \alpha_2 \end{cases}$$

And similarly for Shred:

Definition A.44 (Shreddable Pattern Sets)**.** We define CanShred as follows: $\textsc{CanShred}(\Pi)$ iff $\exists \alpha \backslash \Psi \in \Pi.\ \alpha_1 * \alpha_2 <: \alpha \in \Psi$ for some α_1, α_2

Our next major lemma states that for any type compatibility proof there is a corresponding proof where the match-patterns have been completely "crunched" and the anti-match patterns have been fully "shredded".

Lemma A.45 (Conjunction Elimination)**.** For any compatibility proof, the following statements hold:

- If $\tau \backslash C \sim_{\mathring{\alpha}} \frac{\Pi_1^{\Phi} / \Pi_1^{+}}{\Pi_1^{-}} \Phi\!\!\downarrow\ C' \triangleright (\mathfrak{t}, \omega_1, \beta_1, \eta_1)$ and Π_1^{Φ} is weakly well-formed, then there exists $\tau \backslash C \sim_{\mathring{\alpha}} \frac{\Pi_2^{\Phi} / \Pi_2^{+}}{\Pi_2^{-}} \Phi\!\!\uparrow\ C'' \triangleright (\mathfrak{t}, \omega_2, \beta_2, \eta_2)$ such that $\Pi_2^{\Phi} = \textsc{Crunch}(\Pi_1^{\Phi})$, $\Pi_2^{+} = \textsc{Crunch}(\Pi_1^{+})$, $\textsc{Shred}(\Pi_1^{-}, \Pi_2^{-})$ and $\beta_1 = \beta_2 \cup \beta'$ where $\beta' = \{\alpha \mapsto [\diamond] \mid \alpha \in \textsc{BindVars}(\Pi_1^{\Phi})\}$.

- If $\tau \backslash C \sim_{\mathring{\alpha}} \frac{\Pi_1^{\Phi} / \Pi_1^{+}}{\Pi_1^{-}} \Phi\!\!\uparrow\ C' \triangleright (\mathfrak{t}, \omega_1, \beta_1, \eta_1)$, Π_1^{Φ} is weakly well-formed and at least one of $\textsc{CanCrunch}(\Pi_1^{\Phi})$, $\textsc{CanCrunch}(\Pi_1^{+})$ and $\textsc{CanShred}(\Pi_1^{-})$ is true, then there exists $\tau \backslash C \sim_{\mathring{\alpha}} \frac{\Pi_2^{\Phi} / \Pi_2^{+}}{\Pi_2^{-}} \Phi\!\!\uparrow\ C'' \triangleright (\mathfrak{t}, \omega_2, \beta_2, \eta_2)$ such that $\Pi_2^{\Phi} = \textsc{Crunch}(\Pi_1^{\Phi})$, $\Pi_2^{\Phi} = \textsc{Crunch}(\Pi_1^{+})$, $\textsc{Shred}(\Pi_1^{-}, \Pi_2^{-})$ and $\beta_1 = \beta_2 \cup \beta'$ where $\beta' = \{\alpha \mapsto [\diamond] \mid \alpha \in \textsc{BindVars}(\Pi_1^{\Phi} - \Pi_3^{\Phi})\}$ for some $\Pi_3^{\Phi} \subseteq \textsc{ImmElim}(\Pi_1^{\Phi})$.

Our next set of lemmas focus on the uniqueness of type compatibility results given

that they produce non-conflicting tags. Comparing two proof trees is non-trivial. But we are going to chip away at this problem by demonstrating that in certain common situations, the tags generated conflict with each other and thus need not be considered. First we need a definition of *shallow non-conflict* of types: essentially types that have the same "shape" in a shallow sense.

Definition A.46 (Shallow Non-Conflict)**.** Two types τ and τ' are in shallow non-conflict (the relation $\asymp$) in the following cases:

- $\tau = \texttt{()}$ and $\tau' = \texttt{()}$
- $\tau = \texttt{int}$ and $\tau' = \texttt{int}$
- $\tau = l\ \alpha$ and $\tau' = l\ \alpha'$ for any α and α'
- $\tau = \alpha_1 \,\&\, \alpha_2$ and $\tau' = \alpha_1' \,\&\, \alpha_2'$ for any α_1, α_2, α_1' and α_2'.
- $\tau = \pi \to t$ and $\tau' = \pi \to t$

In all other cases the types shallow conflict; i.e. $\tau \not\asymp \tau'$.

If two type compatibility proof trees are based on shallowly-conflicting subject types, we argue that the two proof trees produce conflicting tags.

Lemma A.47 (Conflicting tags on conflicting types)**.** Suppose we have $\tau \backslash C \sim_{\mathring{a}} \frac{\Pi_1^{\Phi}/\Pi_1^{+}}{\Pi_1^{-}}\ \Phi\ C'' \triangleright (\mathfrak{t}, \omega, \beta, \eta)$ and $\tau' \backslash C \sim_{\mathring{a}} \frac{\Pi_2^{\Phi}/\Pi_2^{+}}{\Pi_2^{-}}\ \Phi\ C''' \triangleright (\mathfrak{t}', \omega', \beta', \eta')$ and $\tau \not\asymp \tau'$, then $\mathfrak{t} \not\leftrightsquigarrow \mathfrak{t}'$.

Our next lemma states that given two type compatibility proof trees such that there is an overlap between the positive patterns of one and the negative patterns of the other, then the two proof trees have conflicting tags.

Lemma A.48 (Conflicting tags with conflicting patterns)**.** If $\alpha\backslash C \sim \frac{\Pi_1^\Phi/\Pi_1^+}{\Pi_1^-} \Phi\, C' \triangleright (\mathfrak{t}, \omega, \beta, \eta)$, $\alpha'\backslash C \sim \frac{\Pi_2^\Phi/\Pi_2^+}{\Pi_2^-} \Phi\, C'' \triangleright (\mathfrak{t}', \omega', \beta', \eta')$ and at least one of the following holds: $\Pi_1^\Phi \cap \Pi_2^- \neq \emptyset$ or $\Pi_2^\Phi \cap \Pi_1^- \neq \emptyset$, then $\mathfrak{t} \not\leftrightsquigarrow \mathfrak{t}'$.

The next main lemma states that two compatibility proofs with non-conflicting tags will have the same bindings.

Lemma A.49 (Unique compatibility)**.** Suppose $\alpha\backslash C \sim \frac{\Pi^\Phi/\Pi_1^+}{\Pi_1^-} \Phi\, C' \triangleright (\mathfrak{t}, \omega, \beta, \eta)$, $\alpha'\backslash C \sim \frac{\Pi^\Phi/\Pi_2^+}{\Pi_2^-} \Phi\, C'' \triangleright (\mathfrak{t}', \omega', \beta', \eta')$ where Π^Φ is well-formed and $\mathfrak{t} \leftrightsquigarrow \mathfrak{t}'$ then $\beta = \beta'$.

We also state that two application matching proofs with non-conflicting tags will match the same function and have the same bindings.

Lemma A.50 (Unique Pattern Matching)**.** Suppose $\alpha_0\ \alpha_1 \overset{\Pi}{C}\!\!\rightsquigarrow^{\Pi_1}_{\bullet}\ \alpha_1'\backslash C_1'\ @\ C_1'' \triangleright (\mathfrak{t}_f, \mathfrak{t}_p, \wp, \omega, \eta, \beta)$ and $\alpha_2\ \alpha_3 \overset{\Pi}{C}\!\!\rightsquigarrow^{\Pi_2}_{\bullet}\ \alpha_3'\backslash C_3'\ @\ C_3'' \triangleright (\mathfrak{t}_f', \mathfrak{t}_p', \wp', \omega', \eta', \beta')$ and further suppose $\mathfrak{t}_f \leftrightsquigarrow \mathfrak{t}_f'$ and $\mathfrak{t}_p \leftrightsquigarrow \mathfrak{t}_p'$, then $\wp = \wp'$ and $\beta = \beta'$.

Finally we can state our second simulation lemma:

Lemma A.51. Let e_0 be the initial program. Further let $e_0 \longrightarrow^* E_1 \parallel e_1$ and $e_0 \xrightarrow[\Omega,\Delta]{}{}^*_\delta \hat{E}_1 \parallel e_1$ such that $(\Omega, \Delta) \twoheadleftarrow_M e_0$ and $E_1\ {}_M\dot{\approx}_{e_0}\ \hat{E}_1$. If $\hat{E}_1 \parallel e_1 \longrightarrow^1_\delta \hat{E}_2 \parallel e_2$, then there exists E_2 such that $E_1 \parallel e_1 \longrightarrow^1_\delta E_2 \parallel e_2$ and $E_2\ {}_M\dot{\approx}_{e_0}\ \hat{E}_2$.

The above lemmas are sufficient to prove the bisimulation between TinyBang and Tagged TinyBang. We restate the theorem:

Theorem 3 (Bisimulation of the TinyBang Operational Semantics)**.** Let e_0 be the initial program. Further let $e_0 \longrightarrow^* E_1 \parallel e_1$ and $e_0 \xrightarrow[\Omega,\Delta]{}{}^*_\delta \hat{E}_1 \parallel e_1$ such that $(\Omega, \Delta) \twoheadleftarrow_M e_0$ and $E_1\ {}_M\dot{\approx}_{e_0}\ \hat{E}_1$. Then $E_1 \parallel e_1 \longrightarrow^1 E_2 \parallel e_2$ if and only if there exists $\hat{E}_2$ such that

$\hat{E}_1 \parallel e_1 \longrightarrow^1_\delta \hat{E}_2 \parallel e_2$ and $E_2 \ {}_M\dot{\approx}_{e_0} \hat{E}_2$.

Bibliography

[1] The computer language benchmarks game. http://shootout.alioth.debian.org/.

[2] Performance tips for JavaScript in v8. https://web.archive.org/web/20170115004538/https://www.html5rocks.com/en/tutorials/speed/v8/.

[3] Pyston - technical overview. https://web.archive.org/web/20170206154532/https://github.com/dropbox/pyston/wiki/Technical-overview, 2016.

[4] What's new in C# 7.0. https://web.archive.org/web/20160825071856/https://blogs.msdn.microsoft.com/dotnet/2016/08/24/whats-new-in-csharp-7-0/, 2016.

[5] Clojure pattern matching algorithm. https://web.archive.org/web/20170102084605/https://github.com/clojure/core.match/wiki/Understanding-the-algorithm, 2017.

[6] ECMAScript 2015 language specification. https://web.archive.org/web/20170117052144/http://www.ecma-international.org/ecma-262/6.0/, 2017.

[7] PEP 3119 – introducing abstract base classes. `https://web.archive.org/web/20170101000000*/https://www.python.org/dev/peps/pep-3119/`, 2017.

[8] K. Adams, J. Evans, B. Maher, G. Ottoni, A. Paroski, B. Simmers, E. Smith, and O. Yamauchi. The hiphop virtual machine. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 777–790, New York, NY, USA, 2014. ACM.

[9] O. Agesen. The cartesian product algorithm. In *European Conference on Object-Oriented Programming*, pages 2–26. Springer, 1995.

[10] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 31–41, New York, NY, USA, 1993. ACM.

[11] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 163–173, New York, NY, USA, 1994. ACM.

[12] L. Augustsson. Compiling pattern matching. In *Functional Programming Languages and Computer Architecture*, pages 368–381. Springer, 1985.

[13] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strniša, J. Vitek, and T. Wrigstad. Thorn: Robust, concurrent, extensible scripting on the JVM. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 117–136, New York, NY, USA, 2009. ACM.

[14] B. Bloom and M. J. Hirzel. Robust scripting via patterns. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 29–40, New York, NY, USA, 2012. ACM.

[15] M. Blume, U. A. Acar, and W. Chae. Extensible programming with first-class cases. In *ACM SIGPLAN Notices*, volume 41, pages 239–250. ACM, 2006.

[16] G. Bracha and D. Griswold. Strongtalk: Typechecking smalltalk in a production environment. In *ACM SIGPLAN Notices*, volume 28, pages 215–230. ACM, 1993.

[17] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. HOPE: An experimental applicative language. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, pages 136–143. ACM, 1980.

[18] L. Cardelli. Compiling a functional language. In *LISP and Functional Programming*, pages 208–217, January 1984.

[19] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language. *SIGPLAN Not.*, 24(7):146–160, June 1989.

[20] W. Choi, S. Chandra, G. Necula, and K. Sen. SJS: A type system for javascript with fixed object layout. In *Static Analysis*, pages 181–198. Springer, 2015.

[21] M. Corporation. *TypeScript Language Specification v2.1*, January 2016.

[22] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-*

Oriented Programming, ECOOP '95, pages 77–101, London, UK, UK, 1995. Springer-Verlag.

[23] K. Driesen and U. Hölzle. Minimizing row displacement dispatch tables. In *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '95, pages 141–155, New York, NY, USA, 1995. ACM.

[24] R. Ducournau. Implementing statically typed object-oriented programming languages. *ACM Computing Surveys (CSUR)*, 43(3):18, 2011.

[25] J. Dunfield. Elaborating intersection and union types. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 17–28, New York, NY, USA, 2012. ACM.

[26] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *ACM SIGPLAN Notices*, volume 30, pages 169–184. ACM, 1995.

[27] T. Freeman and F. Pfenning. *Refinement types for ML*, volume 26. ACM, 1991.

[28] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks. Static type inference for ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1859–1866, New York, NY, USA, 2009. ACM.

[29] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not.*, 44(6):465–478, June 2009.

[30] J. Garrigue. Programming with polymorphic variants. In *ML Workshop*, volume 13. Baltimore, 1998.

[31] F. Geller, R. Hirschfeld, and G. Bracha. Pattern matching for an object-oriented dynamically typed programming language. Technical report, Hasso Plattner Institute, University of Potsdam, 2010.

[32] L. Gong, M. Pradel, and K. Sen. JITProf: Pinpointing JIT-unfriendly javascript code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 357–368, New York, NY, USA, 2015. ACM.

[33] J. O. Graver and R. E. Johnson. A type system for smalltalk. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 136–150. ACM, 1989.

[34] K. Hammond, W. Partain, P. Wadler, C. Hall, and S. Peyton Jones. The glasgow haskell compiler: a technical overview. In *Proceedings of Joint Framework for Information Technology Technical Conference, Keele*, pages 249–257. DTI/SERC, March 1993.

[35] F. Le Fessant and L. Maranget. Optimizing pattern matching. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 26–37, New York, NY, USA, 2001. ACM.

[36] L. Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 35–46. ACM, 2008.

[37] M. Might and P. Manolios. A posteriori soundness for non-deterministic abstract inter-

pretations. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 260–274. Springer, 2009.

[38] G. Morrisett. *Compiling with Types*. PhD thesis, School of Computer Science, Carnegie Mellon University, December 1995.

[39] Z. Palmer. *Designing a Typed Scripting Language*. PhD thesis, The Johns Hopkins University, 2015.

[40] Z. Palmer, P. H. Menon, A. Rozenshteyn, and S. Smith. *Types for Flexible Objects*, pages 99–119. Springer International Publishing, Cham, 2014.

[41] Z. Palmer and S. F. Smith. Higher-Order Demand-Driven Program Analysis. In S. Krishnamurthi and B. S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:25, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[42] F. Pottier. A framework for type inference with subtyping. In *ACM SIGPLAN Notices*, volume 34, pages 228–238. ACM, 1998.

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

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

[45] 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.

[46] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM.

[47] G. Richards, F. Z. Nardelli, and J. Vitek. Concrete Types for TypeScript. In J. T. Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 76–100, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[48] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 944–953, New York, NY, USA, 2006. ACM.

[49] D. Schmidt and B. Steffen. Program analysis as model checking of abstract interpretations. In *International Static Analysis Symposium*, pages 351–380. Springer, 1998.

[50] M. Shields and E. Meijer. Type-indexed rows. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 261–275, New York, NY, USA, 2001. ACM.

[51] O. G. Shivers. *Control-flow Analysis of Higher-order Languages of Taming Lambda.* PhD thesis, Pittsburgh, PA, USA, 1991. UMI Order No. GAX91-26964.

[52] J. Siek and W. Taha. Gradual typing for objects. In *Proceedings of the 21st European Conference on ECOOP 2007: Object-Oriented Programming*, ECOOP '07, pages 2–27, Berlin, Heidelberg, 2007. Springer-Verlag.

[53] B. Stroustrup. *The Design and Evolution of C++.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[54] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, and M. Felleisen. Is sound gradual typing dead? In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 456–468, New York, NY, USA, 2016. ACM.

[55] T. Wang and S. F. Smith. Precise constraint-based type inference for java. In *European Conference on Object-Oriented Programming*, pages 99–117. Springer, 2001.

[56] O. Zendra, D. Colnet, and S. Collin. Efficient dynamic dispatch without virtual function tables: The smalleiffel compiler. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 125–141, New York, NY, USA, 1997. ACM.

[57] H. Zhao, I. Proctor, M. Yang, X. Qi, M. Williams, Q. Gao, G. Ottoni, A. Paroski, S. MacVicar, J. Evans, and S. Tu. The hiphop compiler for PHP. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 575–586, New York, NY, USA, 2012. ACM.

Curriculum Vitae

EDUCATION

The Johns Hopkins University — Baltimore, MD
Doctor of Philosophy — February 2017

The Johns Hopkins University — Baltimore, MD
Master of Science in Engineering, Computer Science — Dec 2007

University of Calicut — Calicut, India
Bachelor of Technology — May 1999

TEACHING EXPERIENCE

Teaching Assistant	*Object-Oriented Software Engineering* Department of Computer Science	Falls 2012-2015 The Johns Hopkins University
Teaching Assistant	*Programming Languages* Department of Computer Science	Springs 2012-2016 The Johns Hopkins University
Teaching Assistant	*Data Structures* Department of Computer Science	Spring 2011 The Johns Hopkins University

TEACHING AWARDS

Whiting School of Engineering Outstanding Teaching Award — May 2011

BIBLIOGRAPHY

PUBLICATIONS AND WRITINGS

Peer-Reviewed Research

P. H. Menon, Z. Palmer, A. Rozenshteyn, S. Smith. Types for Flexible Objects. *12th Asian Symposium on Programming Languages and Systems (APLAS)* (2014).

P. 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. H. Menon, Z. Palmer, A. Rozenshteyn, S. Smith. BigBang: Designing a Statically-Typed Scripting Language. *International Workshop on Scripts to Programs (STOP)* (2012).

INDUSTRY EXPERIENCE

Engineer Feb 2008 – Aug 2009
NVIDIA

Technical Architect Nov 1999 – Aug 2006
SIEMENS