DEEP AND SHALLOW TYPES

BEN GREENMAN

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Khoury College of Computer Sciences
Northeastern University
Boston, Mass.

November 2020

Ben Greenman:
Deep and Shallow Types,
Doctor of Philosophy, Northeastern University, Boston, Mass.

Northeastern University
Khoury College of
Computer Sciences

PhD Thesis Approval

Thesis Title: Deep and Shallow Types

Author: [redacted]

This thesis has been duly completed to complete all degree requirements for the PhD in Computer Science

Matthias Felleisen		11 December 2020
	Thesis Advisor	Date
[redacted]		2020/12/17
	Thesis Reader	Date
Amal Ahmed		17 December 2020
	Thesis Reader	Date
[redacted]		2020-12-17
	Thesis Reader	Date
Shriram Krishnamurthi		2020-12-17
	Thesis Reader	Date
Sam Tobin-Hochstadt		2020-12-18

KHOURY COLLEGE APPROVAL

	12/18/20
Associate Dean for Graduate Programs	Date

COPY RECEIVED BY GRADUATE STUDENT SERVICES:

Recipient's Signature	Date

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI Website.

ABSTRACT

The design space of mixed-typed languages is lively but disorganized. On one hand, researchers across academia and industry have contributed language designs that allow typed code to interoperate with untyped code. These design efforts explore a range of goals; some improve the expressiveness of a typed language, and others strengthen untyped code with a tailor-made type system. On the other hand, experience with type-sound designs has revealed major challenges. We do not know how to measure the performance costs of sound interaction. Nor do we have criteria that distinguish "truly sound" mixed-typed languages from others that enforce type obligations locally rather than globally.

In this dissertation, I introduce methods for assessing mixed-typed languages and bring order to the design space. My first contribution is a performance-analysis method that allows language implementors to systematically measure the cost of mixed-typed interaction.

My second contribution is a design-analysis method that allows language designers to understand implications of the type system. The method addresses two central questions: whether typed code can cope with untyped values, and whether untyped code can trust static types. Further distinctions arise by asking whether error outputs can direct a programmer to potentially-faulty interactions.

I apply the methods to several designs and discover limitations that motivate a synthesis of two ideas from the literature: deep types and shallow types. Deep types offer strong guarantees but impose a high interaction cost. Shallow types offer weak guarantees and better worst-case costs. This dissertation proves that deep and shallow types can interoperate and measures the benefits of a three-way mix.

To Sarah and the Little Bears

ACKNOWLEDGMENTS

Thank you Sarah Lee Greenman for eight years of love, support, and civilizing guidance. Without Sarah, I might still be a Ph.D. student (why not?) living the monastic lifestyle. You know the drill: saltine crackers and peanut butter, candlelight, frequent nights at the lab, and austere conditions at home. With Sarah, I have two cats to attend and a baby on the way. It's been more fun, too.

Thank you parents for twenty-nine years of love and all kinds of support. One stormy high-school afternoon, I set out to test my domestic liberty by reading The Communist Manifesto at home. But instead of outrage, this action met calm encouragement. "That's great." I may have finished the book all the same, but what stands out is a lesson in patience and respect for primary sources.

Thank you Matthias Felleisen for teaching me how to be a scientist, and that there is much more to programming languages than dependent type systems and lambda combinators.

Thank you committee members: Jan Vitek, Amal Ahmed, Fritz Henglein, Shriram Krishnamurthi, and Sam Tobin Hochstadt. Suffice it to say, they improved this dissertation and taught rigorous theory, reproducible artifacts, and critical discussion.

Ten years ago I was happily working as a janitor for Pete Migliorini and attending Hudson Valley Community College, getting ready to transfer to the ILR School at Cornell the next year. One night, between cleanings, I got talking to the security guard (Rusty?) about life and college. "Learn a trade," he said, "something they can't take away from you." That was some excellent advice. Next semester I took a programming course from Andrew Hurd, who suggested we students code on paper first instead of throwing ideas at the Java compiler. What an idea. The following Fall at Cornell, my housemates Kevin Dolan, Dan Rothenberg, Dave Viera, and Mark Vigeant told me to minor in computing rather than physics. "Jobs are good in CS; you can earn as much as the majors." Nicole Roy explained the degree requirements, and things snowballed from there. Thorsten Joachims sparked my interest in programming languages by remarking that there is no Moore's Law for humans. Ramin Zabih taught me a first decent language (OCaml 3.12). Nate Foster hired me as a teaching assistant; did he know I was an ILR major? Classmates Ben Carriel and Sam Park, among many others, inspired harder and better work. Ross Tate helped me to succeed at a first research project and to apply broadly for graduate school. Ehsan Hoque insisted I go somewhere new, really anywhere but Cornell. Fabian Muehlboeck recommended the PRL Lab at Northeastern. Thank you all.

CONTENTS

1

WHAT IT'S ALL ABOUT

A language that can mix typed and untyped code must balance three conflicting dimensions:

- *Proofs*: Static types should be accurate predictions about the way a program behaves at run-time. If a type makes a claim about an expression, then other code—typed or untyped—may depend on it.

- *Performance*: Adding types to part of a codebase should not cripple its running time. On the contrary, a smart compiler should use types to generate efficient code.

- *People*: Untyped code must be free to create all sorts of values and typed code must be able to interact with many untyped designs. Programmers should not have to work around tough constraints on the boundary between typed and untyped code.

The ideal mixed-typed language would satisy all three goals, letting programmers add descriptive types to any component in a program and supporting those types with deep guarantees and fast performance. This ideal is not here yet. Friction between the dimensions raises a whole host of problems about how to enforce types at run-time. In particular, *performance* is the driving question. Type guarantees that can (in principle) be enforced against untyped code often bring an overwhelming cost, slowing a program down by several orders of magnitude.

Researchers have addressed the performance question with designs that advertise low costs, but overall progress towards the ideal is marginal because these designs are incomparable. For one, the performance of a new mixed-typed language is intertwined with the implementation of its host language; comparing performance across different languages is hopeless. Second, the new designs typically compromise on proofs or people. Lacking an apples-to-apples comparison, it is impossible to decide whether a language has solved the performance question.

The first half of this dissertation untangles the design space. I present a method to measure performance, a method to measure type guarantees, and basic requirements concerning the expressiveness of such type systems. I apply these methods and conclude that there are two promising designs: deep types via the natural semantics and shallow types via the transient semantics. The impasse leads to my

thesis question, which asks whether a language can effectively combine both techniques. In the second half of this dissertation, I provide affirmative support for the thesis.

1.1 THESIS STATEMENT

Deep and shallow types can coexist in a way that preserves their formal properties Programmers can combine these types to strengthen shallow-type guarantees, avoid unimportant deep-type runtime errors, and lower the running time of typed/untyped interactions.

1.2 DISSERTATION OVERVIEW

Looking ahead, the first order of business is to lay down ground rules for expressiveness. My goal is to combine typed and untyped code in a *migratory typing* system, in which types accommodate the grown idioms of an untyped host language (chapter 2). Languages that fail the expressiveness criteria, however, can still benefit from the results. Chapter 3 presents the first systematic method for evaluating performance and validates this method through an empirical study of two migratory typing systems: Typed Racket and Reticulated Python. Both languages guarantee type soundness, but come with very different performance characteristics; more surpringly, they compute incompatible results for seemingly-equal code. Chapter 4 brings these two languages, and several others, into a common model for a precise comparison of designs. The design-space analysis motivates a compromise between two semantics: natural and transient. Chapter 5 presents the first half of the compromise; namely, a transient semantics for Typed Racket. Chapter 6 formally proves that deep and shallow types can interoperate and reports the performance of a Typed Racket variant that supports both natural and transient behavior. The dissertation ends with a view to future work (chapter 7) and reflections on the wider research context (chapter 8).

Overall, I present four major contributions:

1. the first performance-analysis method to systematically explore the interactions enabled by a mixed-typed language;

2. the first design-analysis method to articulate the meaning of types for both typed and untyped parts of a codebase;

3. a scaled-up transient that handles a rich language of types and employs ahead-of-time optimizations; and

4. the first language that lets programmers migrate untyped code to two type-sound disciplines: deep and shallow types.

1.3 SPECIFICATION, IMPLEMENTATION, AND NAMING

This dissertation is about different ways of mixing typed and untyped code in a programming language. Each "way" starts from a rough idea, comes to life via a formal semantics, and is tested against formal specifications. Different instances of these three concepts need names.

My primary focus is on two rough ideas: deep types and shallow types. Deep types are nearly as good as static types. If types in a statically-typed language provide a certain guarantee, then the deep versions of those types strive for the same guarantee no matter what untyped code throws at them. Shallow types are weaker than deep types, but better than nothing. A shallow type may provide a temporary guarantee, and may permit more behaviors than the corresponding static type.

These two ideas are accompanied by two leading semantics: natural and transient (chapter 4). Natural realizes deep types by carefully monitoring the boundaries between typed and untyped code—either with exhaustive assertions or proxy wrappers. Transient realizes shallow types by rewriting all typed code to check the basic shape of every value that might be from untyped.

The two properties that distinguish these semantics, and thereby provide a formal distinction between deep and shallow and weaker ideas, are complete monitoring and type soundness (chapter 4). Natural satisfies complete monitoring while transient does not. Both natural and transient satisfy a non-trivial type soundness. Weaker mixings are unsound.

I use informal words to talk about different "ways of mixing typed and untyped code," including: methods, strategies, and approaches. There is no hope in trying to be authoritative because the research community is still seeking a best method for a useful combination.

1.3.1 *Names in Prior Work*

Tunnell Wilson et al. [109] introduce the names *deep* and *shallow*, but use them for the natural and transient implementations. Greenman and Felleisen [43] use *higher-order* for the deep idea and *first-order* for the shallow idea.

Natural goes by many names. Vitousek et al. [114] and several others call it *guarded* because the semantics keeps a firm barrier between typed and untyped code. Chung et al. [21] introduce the word *behavioral* for both the semantics and its characteristic wrapper values. Foundational papers simply call it gradual typing [54, 86, 103].

The name *natural* comes from Matthews and Findler [65], who use it to describe a proxy method for transporting untyped functions into a typed context. Earliers works on higher-order contracts [33], remote procedure calls [76], and typed foreign function interfaces [78] em-

ploy a similar method. New et al. [73] present a semantic argument that natural is indeed the only way to enforce the key properties of static types.

2

MIGRATORY TYPING

Migratory typing is a novel approach to an old desire: mixing typed and untyped code. A typed programming language comes with a strict sub-language (of types) that articulates what a program computes. For better or worse, code that does not fit the sub-language may not run. An untyped language runs any program in which the primitive computations stick to legal values. The mixed-typed idea is to somehow combine some good aspects of both. A programmer should have some untyped flexibility and some typed guarantees.

Of course, flexibility and guarantees lie at two ends of a tradeoff. More freedom to run programs means less knowledge about what a new program might do, unless there are run-time checks to catch extreme behaviors. Run-time checks slow down a computation, thus a mixed-typed language needs to balance three desires: expressiveness, guarantees, and performance.

Before a language design can address the central 3-way tradeoff, its creators must decide what kinds of mixing to allow and what goals to strive for. Migratory typing is one such theory. The goal is to add static typing onto an independent untyped language. Programmers create a mixed-typed program by writing types for one chunk of untyped code; that is, by migrating the chunk into the typed half of the language. Both the goal and the method incorporate lessons from earlier mixed-typed efforts (chapter 2.1), along with basic observations about programming (chapter 2.2). The observations, in particular, motivate design choices that characterize migratory typing (chapter 2.3).

2.1 PRE-MT: HITS AND MISSES

In the days before migratory typing, language designers explored several ways to mix typed and untyped code. Some mixtures began with an untyped language and allowed user-supplied type annotations. Others began with a typed language and added untyped flexibility. The following early works helped form the migratory typing ideas behind my research.

2.1.1 *Type Hints*

Early Lisps, including MACLISP [70] and Common Lisp [92], have compilers that accept type hints. In MACLISP, for example, a programmer can hint that a function expects two floating-point numbers

```
(DECLARE (FLONUM (F FLONUM FLONUM)))
(DEFUN (F A B) (PLUS A B))
```

Figure 1: Example type hint in MACLISP [77]. The compiler may rewrite PLUS into code that assumes floating-point inputs.

and returns one to encourage the compiler to specialize the function body (figure 1).

Any speedup due to type hints, however, comes at a risk. There is no static type system to prove that hints are sensible claims. If a hint is nonsense, then the compiled code may behave in unexpected ways. Similarly, there is no dynamic guarantee that compiled code receives valid inputs. If the function F in figure 1 is invoked on two strings, it may compute an invalid result. In other words, type hints come with all the perils of types in a C-like language.

History Note: Other early type systems for Lisp and Scheme go well beyond the spartan type hints that appear in the MACLISP and Common Lisp specifications. Cartwright [17] infers types that a theorem prover can depend on. Wand [119] presents a semantic prototyping system (SPS) that includes a type system for Scheme. Haynes [50] uses row types to handle Scheme idioms, including variable-arity polymorphism [80].

2.1.2 *Soft and Set-Based Inference*

In principle, type inference can bring static types to untyped code. Research on soft typing pursues this goal in an ideal form by constructing types for any untyped program. Soft type systems never raise a type error. Instead, a soft type checker widens types as needed and inserts run-time checks to protect implicit down-casts [27, 121].

The key to the soft typing problem is to adapt inference from equalities to inequalities [107]. In a language such as ML, a type describes exactly how a variable may be used. Any out-of-bounds use is an error by definition. Thus ML inference asks for a solution to a system of equalities between variables and types. Inference for an untyped language must relax the equality assumption to deal with the less-structured design of untyped programs. Here, the natural types describe sets of values with compatible behavior. The inference problem asks for types that over-approximate the behaviors in a set of values.

There are two known methods to solve type inequalities. Soft inference adds slack variables to types, turns the inequalities into equalities, and then uses Hindley-Milner style inference [27]. Set-

based inference solves the inequalities by computing a transitive closure through constructors over the entire program [3, 4, 34, 35, 36]. Both solutions, unfortunately, reveal major challenges for inference. Types can quickly become unreadable as inference computes supersets based on the syntax of a program. Type structure depends on the whole program; small syntactic changes can disrupt the overall typing, and reasoning about flows can lead to compile-time performance issues [67]. These challenge suggest that full inference for untyped code is impractical.

Wright [121] notes that user-provided annotations can help with brittleness and readability, despite friction with the tenets of soft typing. Meunier [66] improves the performance of set-based analysis by leveraging contracts at module boundaries. Their observations anticipate the migratory typing approach to mixed-typed code.

2.1.3 *Inference via Dynamic Typing*

Henglein's dynamic typing uses standard types and general-purpose coercions to compile untyped code to an efficient representation [52]. The method starts with a conventional typed language and adds three related ingredients: a universal sum type, called the dynamic type; coercions that inject any precisely-typed value up to the dynamic type; and (partial) coercions that project a dynamically-typed value down to a non-dynamic type. This augmented core is the basis of a mixed-typed language. Typed code maps directly to the core with no additional coercions. Untyped code may require coercions, but a smart compiler can minimize their use. Henglein [51] compiles Scheme to a monomorphic type system and is able to resolve at least 50% of the coercions in six benchmarks. Henglein and Rehof [53] extend the method to polymorphic types and implement IEEE Scheme using Standard ML.

Conventional types, however, are not always sufficient to capture untyped designs. For example, Henglein and Rehof [53] note that Scheme conditionals end up with extra coercions into the dynamic type. The designs can be expressed, but a tailored type system is needed to maximize run-time efficiency.

2.1.4 *Optional Typing*

An optional, or pluggable, type system adds a static analysis to an untyped language [15]. The approach is related to type hints in that programmers must add annotations to untyped code. Optional types are supported, however, by a full-fledged type checker and a no-op compiler. The type checker is the static analysis; it uses types to find basic logical errors. Compilation erases types to arrive at an untyped program that can safely interoperate with the rest of the program.

Despite their widespread adoption (chapter 4.2), optional types are a bit of a disappointment for the research community because these types are unsound. A programmer cannot use optional types to predict the inputs that a function will receive, and likewise a compiler cannot trust an optional type without inserting a run-time guard or studying the values that can flow to the typed position.

History Note: Optional typing is one valid way to use Lisp type hints. A Lisp compiler need not optimize based on type hints, and it may even ignore types completely [70, 92]. Bracha and Griswold [15] independently developed the optional style, explained why it is a practical mode of use, and identified ways to safely optimize parts of optional programs.

2.1.5 *Type Dynamic*

Statically-typed languages often need to interact with untyped values, perhaps though a database connection, web socket, or interactive prompt. Both Abadi et al. [1] and Leroy and Mauny [62] thus present static type systems with a special dynamic type. Typed code can interact with a untyped value by first testing its structure; the type system records observations.

Quasi-static typing extends the type-dynamic idea with implicit structure tests [101]. Instead of asking the user to write and maintain type-testing code, the quasi-static system generates run-time checks. Consequently, programmers have less incentive to handle the dynamic type at the boundary to untrusted code. The result is a mixed-typed language because entire blocks of code may have the dynamic type throughout. Gradual typing emphasizes the mixed-typed idea in quasi-static typing, contributes major technical improvements and design discipline [86, 87], and has inspired a large body of static-to-dynamic research (`github.com/samth/gradual-typing-bib`).

Implicit coercions to type dynamic, however, weaken type-proofs in a gradual or quasi-static language. Rather than showing that components *do* fit together, a gradually-typed program is something that *can* fit together given good values at each occurrence of the dynamic type. Words such as "plausibility" [101] and "consistency" [86] aptly describe the weakened guarantees; gradual types can only point out implausibilities and inconsistencies among non-dynamic types.

2.2 MT: OBSERVATIONS

Migratory typing stands on three observations: untyped code exists, type annotations improve maintainability, and sound types are a worthwhile ideal. On the surface, these basic opinions simply motivate a typed/untyped mix; between the lines, however, they suggest requirements for an effective mixed-typed language.

2.2.1 *MT-01: untyped code exists*

Untyped code is a fact. Large companies such as Dropbox, Facebook, and Twitter started as untyped projects. Small teams continue to employ untyped languages; indeed, most repositories on GitHub use either JavaScript, Python, PHP, or Ruby (`githut.info`).

Once an untyped codebase is off the ground and the lack of reliable type information becomes a maintenance bottleneck, programmers have two options. The extreme option is to change languages. Twitter, for example, was able to port its Ruby codebase over to Scala [112]. For teams that lack the time and expertise to make such a switch, the alternative is to re-create any necessary benefits of types. An exemplar of the second option is Sweden's pension system, which depends on a contract-laden Perl program [61]. The contracts ensure that components in this huge program behave as intended.

Research on mixed-typed languages can be a great help to teams in this second camp, that cannot afford to rewrite their codebase. General knowledge about how to design a companion type system can reduce the development cost of an in-house solution like Sweden's contracts. And a tailor-made type system, if one exists, provides an immediate solution to maintenance issues.

2.2.2 *MT-02: types communicate*

Type annotations are an important channel of communication. For human readers, they describe the high-level design of code. Even the original author of a function can benefit from reading the types after some time away from the codebase. For a compiler, annotations are hints about what the programmer expects. Any type error messages that can point to part of an annotation have a syntactic link to the programmer who needs to deal with the errors.

2.2.3 *MT-03: sound types catch bugs*

All static types can find typo-level mistakes, but only sound types guarantee type-specified behavior. In a mixed-typed setting, a guarantee can make a world of difference. Picture a large untyped codebase made up of several interacting components, and suppose that one component behaves strangely. Adding unsound types to that one component can reveal a syntactic mistake, but nothing more. Sound types, on the other hand, will halt the program as soon as an incorrect value appears in typed code. If the language can additionally report the source of the untyped value and the rationale for the mismatched type expectation, then the programmer has two clues about where to begin debugging.

Going beyond soundness, a mixed-typed language that satisfies complete monitoring guarantees the run-time behavior of every type. If a value flows across a type-annotated source position, then future users of the value can depend on the type—no matter whether these uses occur in typed or untyped code (chapter 4). Type soundness makes no guarantee about behavior once a value flows out to an untyped context; with complete monitoring, the types are in control of all interactions.

2.3 MT: DESIGN CHOICES

Taken broadly, migratory typing studies how to add types to untyped code. My dissertation builds on a more focused theory that is grounded in the following principles. Typed Racket shares the same theory [102, 107]. Our aim is to maximize the potential benefits of a mixed-typed language, in spite of the risk that some goals may prove unattainable.

2.3.1 *MT-r1: types for untyped code*

Migratory typing begins with an independent untyped language and adds a companion type system. The new types and type system must express common idioms from the untyped world; in other words, a type system that demands a re-organizion of untyped code is not acceptable.

2.3.2 *MT-r2: require annotations, reject programs*

Programmers must write type annotations for top-level and recursive definitions. Extra annotations may guide type inference.

The type checker will reject ill-typed programs instead of creating a run-time cast to bridge unequal types. Programmers must deal with the type errors, either by inserting a cast or re-designing code.

2.3.3 *MT-r3: sound types*

Well-typed code must provide a soundness guarantee in which types constrains the possible results of an evaluation. Both deep and shallow types are acceptable, but nothing less. For example, optional typing satisfies an inadequate soundness guarantee that promises a well-formed value but nothing about how the type of an expression limits the outcomes.

Blame

Sound types catch bugs, but make no claims about actionable error outputs. Blame is an additional property geared to useful errors that tell a programmer where to begin debugging. To this end, a mixed-typed language should try to present meaningful source locations along with every run-time type mismatch error.

2.3.4 *MT-r4: clear boundaries*

Typed and untyped code must be linked at static and clearly visible API boundaries. In order for a typed module to interact with an untyped value, the module must declare a type specification for the value. An untyped module does not need to give specifications because any typed value that it imports comes with a static specification for correct use.

By contrast, this dissertation is not directly concerned with true gradual languages that include a dynamic type and infer boundaries during typechecking [87]. Such languages can still benefit from my results at an intermediate step, after occurrences of the dynamic type have been replaced with precise types and casts. But one pragmatic question remains: a true gradual language must find a way to communicate trouble at an inferred boundary up to the programmer, who may not understand why the language decided to insert the cast that eventually failed.

Requiring boundaries greatly simplifies and strengthens the type system. It is simpler because there is no dynamic type; standard definitions of types, subtyping, and all the rest suffice for the type checker. It is stronger because there is no type precision relation to allow constructions that a standard type system would rule out. In a gradual language, code that looks typed can diverge through clever use of the dynamic type; with boundaries, there is a clear line between the typed and untyped code. Refer to chapter 5.4.4 for an example of how odd behaviors can slip into apparently-typed Reticulated Python code.

Macro, Micro

Prior works make a distinction between *macro*-level and *micro*-level gradual typing systems [99, 100]. These names express the same idea as my boundary requirement, but in terms of granularity and with the term "gradual typing" broadly construed to refer to any sound mixed-typed language. Macro allows interaction between typed and untyped chunks of code [103] whereas micro allows "fine-grained" mixing via a dynamic type [86].

Looking back, I think there were two dimensions at play. First is whether to include a dynamic type. Second is how to mix: whether to

migrate from an untyped host language or to add flexibility to a static type system [42]. Micro/macro is a useful mnemonic for the first dimension, but it is more direct to talk about dynamic/non-dynamic and migratory/non-migratory as two choices in the design of a new mixed-typed language.

2.4 RECENT HISTORY

Tobin-Hochstadt [102] developed migratory typing alongside Typed Racket. The basic ideas arose from work on soft typing [27, 121], higher-order contracts [31], language interoperability [40], and modular set-based analysis [66]. Subsequent work adapted migratory typing to multi-paradigm language features: compositional flow-based reasoning [105], delimited continuations [98], variable-arity polymorphism [93], type-driven optimization [88], first-class classes [96], units (first-class modules) [28], and refinement types [58]. Refer to Tobin-Hochstadt et al. [107] for a ten-year retrospective. My dissertation adds one step to this lineage. I began by studying the most pressing challenge, performance costs, and arrived at the combination of deep and shallow types.

3 PERFORMANCE ANALYSIS METHOD

This chapter is based on joint work with: Matthias Felleisen, Daniel Feltey, Robert Bruce Findler, Zeina Migeed, Max S. New, Asumu Takikawa, Sam Tobin-Hochstadt, and Jan Vitek [44, 46, 100]. The Typed Racket benchmarks presented in this chapter have been improved over the years by: Spenser Bauman, Lukas Lazarek, Cameron Moy, and Sam Sundar.

Sound types come with performance overhead in a mixed-typed language because soundness is a claim about behavior and the only way to control the behavior of untyped code is via run-time checks. These checks impose a cost in proportion to the frequency of mixed-typed interactions, the complexity of the type specifications that govern boundaries, and the strength of the soundness guarantee.

Language designers must measure the performance of a mixed-typed language to judge its overall usefulness in light of its guarantees. Type-sound code that runs too slowly is worthless. At a finer grain, users need an idea of what overhead to expect when they begin experimenting with types. Implementors need a comprehensive performance summary to measure improvements to a language and to compare alternative mixed-typed designs. Despite these realities, early reports on mixed-typed languages typically lack performance evaluation. A few acknowledge performance issues in passing [5, 104, 114]. Others show only the performance of fully-typed code relative to fully-untyped code, skipping the novel configurations in between [79, 115]. But in their defense, the development of a performance method is a challenge in itself.

This chapter presents a systematic and scalable method to assess the performance of a mixed-typed language. The method summarizes performance for the exponentially-many ways that a programmer can mix typed and untyped code by focusing on a binary quality measure. Informally, a mixture is good if it runs within a user-supplied overhead limit. Random sampling can approximate the proportion of good mixtures for programs in which exhaustive evaluation is not practical. Two language evaluations, for Typed Racket and Reticulated Python, validate the method.

3.1 DESIGN CRITERIA

The goal of performance evaluation is to predict the experiences of future users. Intuition suggests that the users of migratory typing will begin with an untyped codebase and add types step-by-step. Expe-

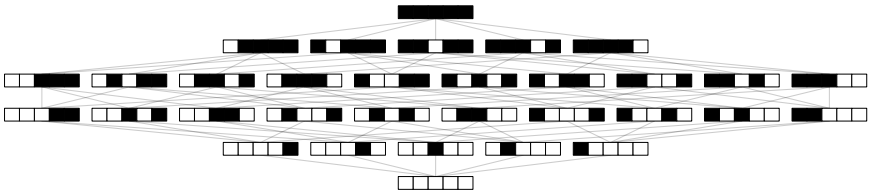


Figure 2: A Racket program with 5 modules supports 32 mixed-typed configurations, including the fully-untyped and fully-typed versions.

rience with Typed Racket supports the intuition. Programmers add types in an incremental fashion and experiment with all sorts of combinations. When typed libraries enter the picture, untyped programmers unknowingly create mixed-typed applications. In a typical evolution, programmers compare the performance of the modified program with the previous version. If the current performance is on par with the previous, then all is well. Otherwise, the easy solutions are: adding more types, and rewinding to a less-typed version. These observations and assumptions about users suggest three basic criteria for an evaluation method.

3.1.1 *Representative Benchmarks*

An evaluation method has to measure programs, and the results of a particular evaluation are limited by the chosen benchmarks. Benchmark programs that stem from realistic code and exercise a variety of features are an important step toward generalizable results.

3.1.2 *Exponential Compression*

A mixed-typed language promises to support exponentially-many combinations of typed and untyped code. In Typed Racket, for example, a programmer can add types to any module of a program. Thus a program with 5 modules leads to 2^5 possible combinations (figure 2). Languages that can mix at a finer granularity support 2^k configurations, where k is the number of potentially-typed blocks.

Without evidence against certain mixtures, an evaluation must collect data for every mixed-typed configuration. These huge datasets call for a way to compress exponentially-many observations into a compact summary.

3.1.3 *Report Overheads*

Because migratory typing starts from an untyped language, programmers can always revert to an untyped version of their codebase if types prove too expensive. The existence of this fully-untyped baseline helps anchor an evaluation. If a programmer can tolerate a certain overhead, say 13x, then a single number can summarize the good parts of the exponentially-large configuation space; namely, the percent of configurations that run fast enough. A second benefit is that overheads make it easy to find points where types improve upon the baseline; look for overheads under 1x.

3.2 EXHAUSTIVE EVALUATION METHOD

An exhaustive evaluation considers all ways that a programmer might toggle type annotations. The method begins with a fully-typed codebase, measures all possible mixed-typed configurations, and introduces a compact visualization to summarize the results.

3.2.1 *By Example*

A Racket program is a collection of modules. Technically, there are two kinds of modules in such a collection: the *migratable* modules that the program's author has direct control over, and the *contextual* modules that come from an external library. A programmer can add types to any migratable module. Thus a program with N migratable modules opens a space of 2^N mixed-typed configurations, and each configuration depends on the same contextual modules.

For example, fsm is a small Racket program that simulates an economy (chapter 3.5.2). The main functionality is split across four modules; with migratory typing, this leads to sixteen mixed-typed configurations. Figure 3 shows all these configurations in a lattice, with the untyped configuration on the bottom and the fully-typed configuration on top. Nodes in the middle mix typed and untyped code; each row groups all configurations with the same number of typed modules. Lines between nodes represent the addition (or removal) of types from one module.

The label below a configuration node reports its overhead relative to the untyped configuration on Racket version 6.4. With these labels, a language implementor can draw several conclusions about performance overhead in fsm. A first observation is that the fully-typed code runs equally fast as the untyped baseline. This 1x overhead is also the overall best point in the lattice. Six other configurations run within a 2x overhead, but the rest suffer from orders-of-magnitude slowdowns. Types in fsm can come at a huge cost.

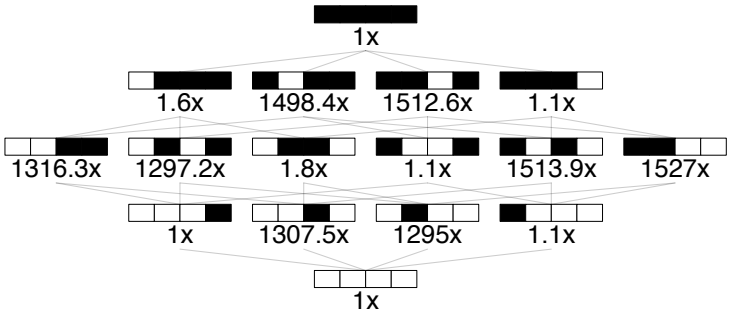


Figure 3: Performance overhead in fsm, on Racket v6.4.

Drawing such conclusions is not easy, however, even for this small program. Manually analyzing a lattice for programs with eight or more modules is clearly infeasible. Figure 4 presents a graphical alternative, an *overhead plot*, that reports configurations' overhead relative to the untyped baseline. Overhead plots are cumulative distribution functions. As the function proceeds left-to-right for numbers D along the x-axis, the curve shows the proportion of all configurations that run at most D times slower than the untyped configuration. For short, these are D-deliverable configurations. On the left, there is always at least one 1-deliverable configuration; namely, the fully-untyped configuration itself. The question is whether other configurations run fast as well. To read such a plot quickly, focus on the area under the curve. A large shaded area implies that a large number of configurations have low overhead.

The second most important aspects of an overhead plot are the two values of D where the curve starts and ends. More precisely, if $h : \mathbb{R}^+ \to \mathbb{N}$ is the CDF that counts the proportion of D-deliverable configurations in a benchmark, the critical points are the smallest overheads d_0, d_1 such that $h(d_0) > 0\%$ and $h(d_1) = 100\%$. An ideal start-value would lie between zero and one; if $d_0 < 1$ then at least one configuration runs faster than the baseline. The end-value d_1 is the overhead of the slowest-running configuration.

Lastly, the slope of a curve corresponds to the likelihood that accepting a small increase in performance overhead increases the number of deliverable configurations. A flat curve (zero slope) suggests that the performance of a group of configurations is dominated by a common set of type annotations. Such observations are no help to programmers facing performance issues, but may help language implementors fix inefficiencies.

Overhead plots scale to arbitrarily large datasets by compressing exponentially-many points into a proportion. Furthermore, plotting two curves on one axis compares relative performance. Figure 5

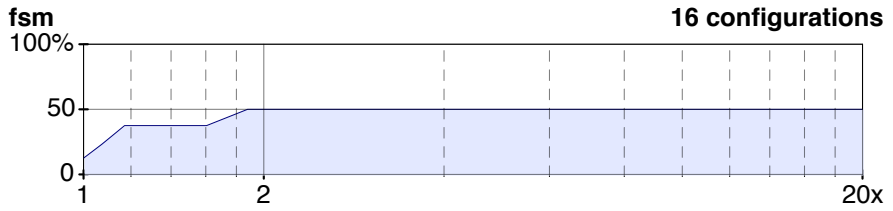


Figure 4: Overhead plot for fsm, on Racket v6.4. The unlabeled vertical ticks mark, from left-to-right: 1.2x, 1.4x, 1.6x, 1.8x, 4x, 6x, 8x, 10x, 12x, 14x, 16x, and 18x.

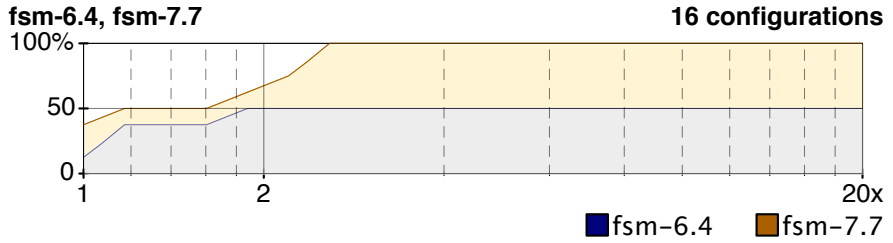


Figure 5: Overhead plots for fsm, on Racket v6.4 and v7.7. The orange curve for v7.7 is higher, showing a relative improvement.

demonstrates two curves for fsm: on Racket v6.4 and v7.7. The latter curve shows a huge improvement thanks to collapsible contracts [29]. Indeed, every fsm configuration is 4-deliverable on Racket v7.7.

3.2.2 *By Definition*

The exhaustive evaluation method applies to other mixed-typed languages as well as Typed Racket. To encourage adaptations, the following definitions highlight key concepts. The prose uses Reticulated Python as a running example.

In Reticulated, every parameter to a function, every function return position, and every class field can be typed or untyped. This is a much finer *granularity* than Typed Racket's, in which entire modules must be typed as a unit. The added flexibility means that an experimenter must choose whether to explore: coarse, module-grained mixes; fine, function-parameter mixes; or something in between.

Definition (*granularity*) The *granularity* of an experiment is the syntactic unit at which it adds/removes type annotations.

For example, Takikawa et al. [100] evaluate Typed Racket at the granularity of modules. Vitousek et al. [115] evaluate Reticulated at the granularity of whole programs (missing all the ways that a programmer can mix types), and Greenman and Migeed [44] evaluate Reticulated at the granularity of whole functions and whole class field sets (chapter 3.6.1).

After choosing a granularity, an experimenter must pick a suite of programs to measure. A potential complication is that programs may depend on external libraries or other modules that lie outside the realistic scope of the evaluation.

Definition (*migratable, contextual*) The *migratable* code in a program defines its configurations. The *contextual* code in a program is common across all configurations.

The granularity and the migratable code define the *configurations* of a fully-typed program.

Definition (*configurations*) Let $C \to C'$ if and only if program C' can be obtained from program C by annotating one migratable syntactic unit. Let $\to^*$ be the reflexive, transitive closure of the $\to$ relation. The *configurations* of a fully-typed program C_τ are all programs C such that $C \to^* C_\tau$. Furthermore, C_τ is a *fully-typed configuration*. An *untyped configuration* C_λ has the property $C_\lambda \to^* C$ for all configurations C.

In terms of prior work, the $\to$ relation includes all possible *type conversion steps* [46, 100]. The $\to^*$ relation corresponds to *term precision* [87] as follows: $e_0 \to^* e_1$ only if $e_1 \sqsubseteq e_0$.

An evaluation must measure overhead relative to a useful baseline. For migratory typing, the correct baseline is the original host-language program.

Definition (*baseline*) The *baseline performance* of a program is its running time in the absence of migratory typing.

In Typed Racket, the baseline is the performance of Racket running the untyped configuration. In Reticulated, the baseline is Python running the untyped configuration. Be advised, Python-running-untyped differs from Reticulated-running-untyped because Reticulated inserts checks in every migratable module that it sees [114].

Definition (*performance ratio*) A *performance ratio* is the running time of a configuration divided by the baseline performance.

An exhaustive performance evaluation measures the performance of every configuration. To summarize the data, choose a notion of "good performance" and count the proportion of "good" configurations. In this spirit, Takikawa et al. [100] ask programmers to consider the performance overhead they could deliver to clients.

Definition (*D-deliverable*) A configuration is *D-deliverable*, for some $D \in \mathbb{R}^+$, if its performance ratio is no greater than D.

3.2.3 *Known Limitations*

Evaluation begins with a fixed set of types, but there are usually many ways to type a piece of code. Consider the application of an identity function to a number:

```
((λ(x) x) 61)
```

In Typed Racket, the parameter `x` can be given infinitely many correct types. The obvious choices are `Integer` and `Number`, but other base types work, including `Real` and `Natural`. Untagged unions bring many options: `(U Real String)`, `(U Real String (-> Boolean))`, and so on. Different choices entail different run-time checks, but the method lacks a systematic way to explore equally-valid typings.

Along the same lines, the definition of granularity does not talk about imprecise types. In Reticulated, the type `Function([Str], Int)` has three less-precise variants that incorporate the dynamic type. The method only looks at one fully-typed variant, but the others may have notable performance implications.

Overhead plots (figure 4) rest on two assumptions. First is that configurations with less than 2x overhead are significantly more practical than configurations with a 10x overhead or more. Hence the plots use a log-scaled x-axis to encourage fine-grained comparison in the 1.2x to 1.6x overhead range and to blur the distinction among larger numbers. Second is that configurations with more than 20x overhead are completely unusable in practice. Pathologies like the 1000x slowdowns in figure 3 represent a challenge for implementors, but if these overheads suddenly dropped to 30x, the configurations would still be useless to developers.

The main limitation of exhaustive evaluation, however, is its exhaustiveness. With 20 migratable units, an experiment requires over 1 million measurements. At a module-level granularity, this limit is somewhat reasonable because each module can be arbitrarily large. But at function-level granularity and finer, the practical limit quickly rules out interesting programs.

3.3 APPROXIMATE EVALUATION METHOD

The proportion of D-deliverable configurations in a program can be approximated using random sampling. First, choose several configurations and measure the proportion of D-deliverable configurations in the sample. Next, repeat the experiment several times. Combining the proportions in a confidence interval provides an estimate for the true proportion.

Definition (95%-r, s-*approximation*) Given r samples each containing s configurations chosen uniformly at random, a simple random

approximation is a 95% confidence interval for the proportion of D-deliverable configurations in each sample.

Intuitively, this method should lead to good results because it randomly samples a stable population. If the true proportion of D-deliverable configurations in a program happens to be 10%, then a random configuration has a 1 in 10 chance of being D-deliverable.

A statistical justification depends on the law of large numbers and the central limit theorem. Let d be a predicate that checks whether a configuration is D-deliverable. Since d is either true or false for every configuration, this predicate defines a Bernoulli random variable X_d with parameter p, where p is the true proportion of D-deliverable configurations. Consequently, the expected value of this random variable is p. The law of large numbers states that the average of *infinitely* many samples of X_d converges to p, the true proportion of deliverable configurations. We cannot draw infinitely many samples, but perhaps this convergence property means that the average of "enough" samples is "close" to p. The central limit theorem implies that any sequence of such averages is normally distributed around the true proportion. A 95% confidence interval generated from sample averages is therefore likely to contain the true proportion.

The statistical argument reveals two weaknesses:

- First, there is no guarantee that every confidence interval based on sampling contains the true proportion of D-deliverable configurations. The results can mislead.

- Second, the confidence intervals could be huge. A wide interval offers little insight, even if it happens to contain the true proportion. In the extreme, a totally useless interval says that 0% to 100% of configurations are D-deliverable.

The argument does say, however, that an interval is very likely to be useful if it is based on a huge number of samples each with a huge number of configurations. The challenge is to find parameters that engineer a compromise between size and precision.

By comparing sample data to the ground-truth from an exhaustive evaluation, I have found that linear sampling gives small and accurate intervals. Figures 6 and 7 demonstrate on a few Typed Racket and Reticulated programs. The blue curve and shaded area on each plot is the exhaustive data. The orange interval is a 95% confidence interval based on $r = 10$ each containing $s = 10 * N$ configurations, where N is the number of typed units in the benchmark program. The sample intervals all tightly cover the true proportion of D-deliverable configurations. Appendix A.1 contains additional empirical data.

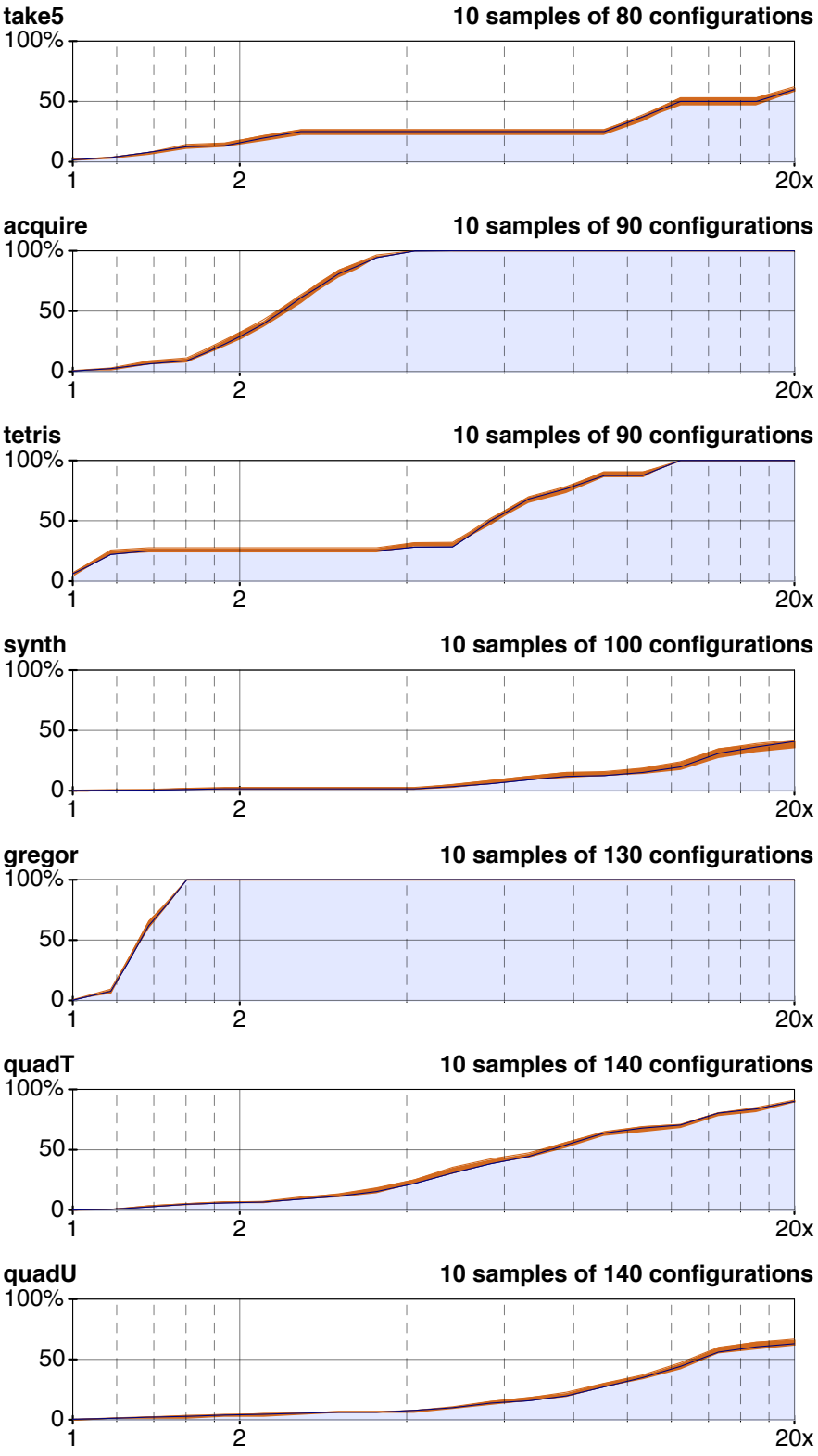


Figure 6: Typed Racket sample validation.

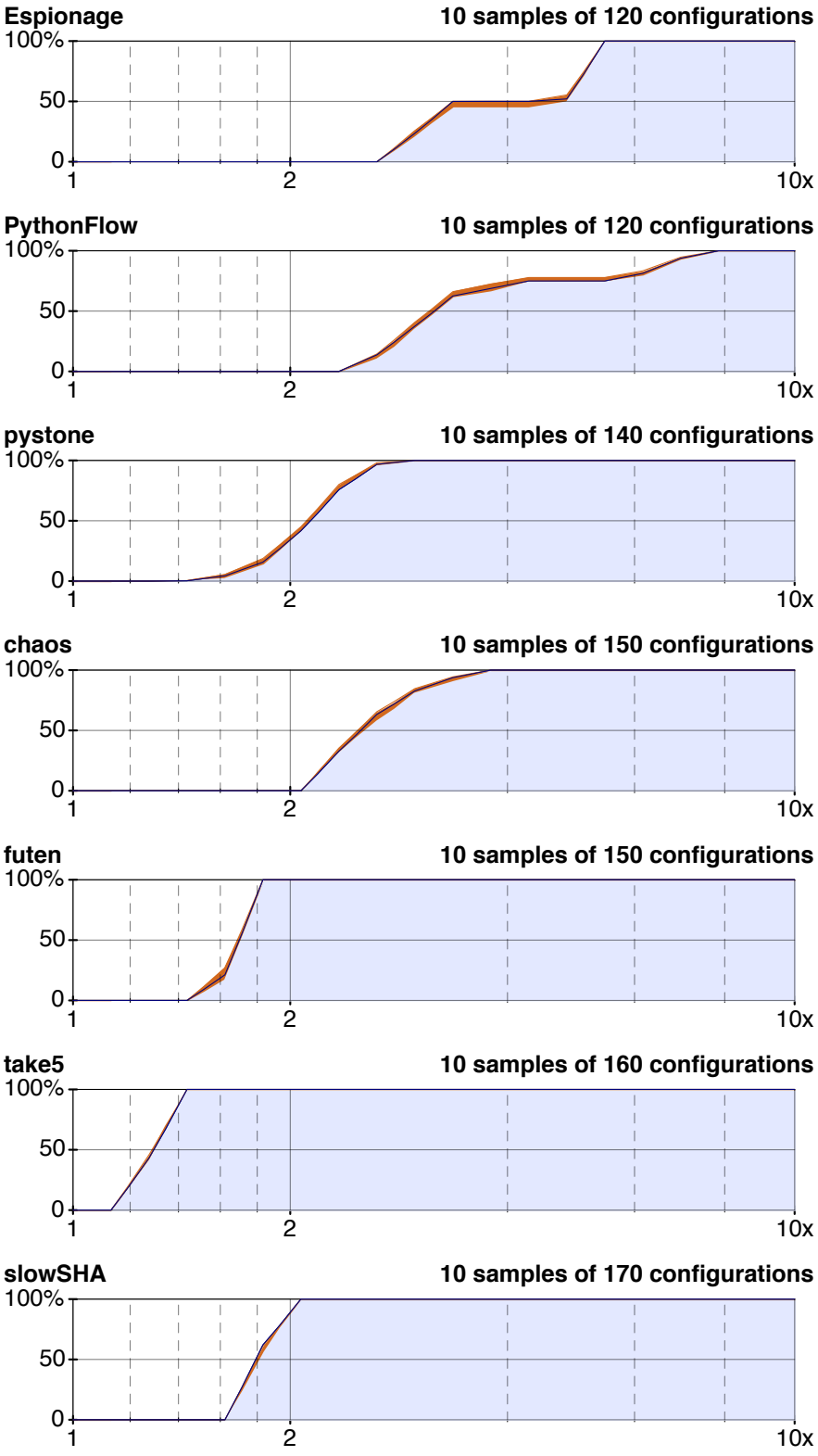


Figure 7: Reticulated sample validation.

3.3.1 *Statistical Protocol*

For readers interested in reproducing the above results, here are additional details about the protocol behind figures 6 and 7.

To generate one random sample, select $10*N$ configurations uniformly at random and compute their overhead. Sampling with replacement gives the same theoretical results as sampling without replacement. The figures employ sampling without replacement in the hope of finding new configurations with exceptional overhead.

To generate a confidence interval for the number of D-deliverable configurations based on a group of samples, calculate the proportion of D-deliverable configurations in each sample and generate a 95% confidence interval from the proportions. This is the simple index method for computing a confidence interval from a sequence of ratios (`arxiv.org/pdf/0710.2024v1.pdf`). A more advanced method may give tighter intervals, if extra precision is needed [30].

3.4 BENCHMARK SELECTION

Representative benchmarks are difficult to come by. My best-effort approach is to seek out programs that serve a realistic purpose. Several implement games, and re-play a game round. Others adapt library code with an example use. All of the forthcoming Typed Racket benchmarks follow this approach (chapter 3.5.2). Many of the Reticulated benchmarks (chapter 3.6.2), however, come from prior work and are smaller scripts in the spirit of the Gabriel benchmarks [38].

3.4.1 *From Programs to Benchmarks*

To convert a program into a benchmark, we:

1. partition the program into migratable and contextual code;
2. build a migratable driver module that runs the program and collects timing information;
3. remove any non-determinism or I/O actions;
4. find types for the migratable code.

The final step, finding types for untyped code, can be difficult. First, the type checker may require casts or refactorings to deal with untyped code. For example, untyped Racket code may assume that the application `(string->number "42")` returns an integer. The assumption is correct, but the type checker cannot follow the reasoning and needs a run-time check. Reticulated does not have union types, and therefore falls back to `Dyn` for many common untyped patterns.

An experimenter must choose whether to rewrite the pattern or accept the trivial typing.

Second, some deep type boundaries may lack run-time support. Typed Racket cannot enforce the type `(U (-> Real) (-> Integer))` at a boundary because its contracts lack unions for higher-order wrappers. The work-around is to rewrite the boundaries or, if possible, simplify the types. For the above, `(-> Real)` is a viable choice.

Third, each static import of a struct type into Typed Racket code generates a unique datatype. Typed modules that share instances of an untyped struct must therefore reference a common definition. Typed Racket benchmarks with this issue include additional contextual modules, called *adaptor modules*, to provide a canonical import.

3.5 APPLICATION 1: TYPED RACKET

This subchapter presents an exhaustive evaluation of Typed Racket v7.7 on a set of twenty-one benchmark programs; namely, the GTP suite v6.0 (`docs.racket-lang.org/gtp-benchmarks/index.html`). The main purpose of this evaluation is to confirm that the exhaustive method provides a useful summary of a mixed-typed language. A secondary result is that it reveals performance challenges that Typed Racket must overcome.

3.5.1 *Protocol*

GRANULARITY The granularity of this evaluation is *modules*, same as the granularity of Typed Racket. One syntactic unit in the experiment is one entire module.

DATA COLLECTION For each configuration in each benchmark, a control script compiled the whole program, ran once ignoring performance, and ran four more times collecting data. These actions used the standard Racket 7.7 BC bytecode compiler, JIT compiler, and runtime settings. The control script ran on a dedicated Linux machine with a i7-4790 processor. The processor has 16GB RAM and four cores, and ran at 3.60GHz.

3.5.2 *Benchmarks*

The GTP benchmark suite consists of twenty-one programs. Below, these benchmarks appear in order of increasing size, as measured by the number of migratable modules. Each comes with a summary and four fields: *Origin* indicates the benchmark's source, *Purpose* describes what it computes, *Author* credits the original author, and *Depends* lists significant contextual libraries.

sieve

Origin : Synthetic
Purpose: Generate prime numbers
Author : Ben Greenman
Depends: None

Demonstrates a scenario where client code is tightly coupled to higher-order library code. The library implements a stream data structure; the client builds a stream of prime numbers.

forth

Origin : Library
Purpose: Forth interpreter
Author : Ben Greenman
Depends: None

Interprets Forth programs. The interpreter represents calculator commands as a list of first-class objects.

fsm, fsmoo

Origin : Economics research
Purpose: Economy simulator
Author : Linh Chi Nguyen
Depends: None

Simulates the interactions of economic agents via finite-state automata [75]. This benchmark comes in two flavors: `fsm` stores the agents in a mutable vector and `fsmoo` uses a first-class object.

mbta

Origin : Educational
Purpose: Interactive map
Author : Matthias Felleisen
Depends: `graph`

Builds a map of Boston's subway system and answers reachability queries. The map interacts with Racket's untyped `graph` library.

morsecode

Origin : Library
Purpose: Morse code trainer
Author : John Clements and Neil Van Dyke
Depends: None

Computes Levenshtein distances [63] and morse code translations for a fixed sequence of pairs of words.

zombie

Origin : Research
Purpose: Educational Game
Author : David Van Horn
Depends: None

Implements a game where players dodge "zombie" tokens. Curried functions over symbols implement game entities and repeatedly cross type boundaries.

dungeon

Origin : Application — *Author* : Vincent St. Amour
Purpose: Maze generator — *Depends*: None

Builds a grid of wall and floor objects by choosing first-class classes from a list of "template" pieces. Originally, the program imported the Racket `math` library for array operations. The benchmark uses Racket's vectors instead of the `math` library's arrays because Typed Racket v6.2 cannot compile the type `(Mutable-Array (Class))` to a contract.

jpeg

Origin : Library — *Author* : Andy Wingo
Purpose: JPEG toolkit — *Depends*: `math/array`, `rnrs/bytevectors-6`

Parses a bytestream of JPEG data to an internal representation, then serializes the result.

zordoz

Origin : Library — *Author* : Ben Greenman
Purpose: Explore bytecode — *Depends*: `compiler-lib`

Traverses Racket bytecode (`.zo` files). The untyped `compiler-lib` library defines the bytecode data structures.

lnm

Origin : Synthetic — *Author* : Ben Greenman
Purpose: Data visualization — *Depends*: `plot`, `math/statistics`

Renders overhead plots. Two modules are tightly-coupled to Typed Racket libraries.

suffixtree

Origin : Library — *Author* : Danny Yoo
Purpose: String tools — *Depends*: None

Implements Ukkonen's suffix tree algorithm [110] and computes longest common subsequences between strings.

kcfa

Origin : Educational — *Author* : Matt Might
Purpose: Explanation of k-CFA — *Depends*: None

Performs 1-CFA on a lambda calculus term that computes $2 * (1 + 3) = 2 * 1 + 2 * 3$ via Church numerals. The (mutable) binding environment flows throughout functions in the benchmark.

snake

Origin : Research — *Author* : David Van Horn
Purpose: Educational Game — *Depends*: None

Implements the Snake game; the benchmark replays a fixed sequence of moves.

take5

Origin : Educational
Purpose: Game
Author : Matthias Felleisen
Depends: None

Manages a card game between AI players.

acquire

Origin : Educational
Purpose: Game
Author : Matthias Felleisen
Depends: None

Simulates a board game via message-passing objects. These objects encapsulate the core data structures and seldom cross module boundaries.

tetris

Origin : Research
Purpose: Educational Game
Author : David Van Horn
Depends: None

Replays a pre-recorded game of Tetris.

synth

Origin : Application
Purpose: Music synthesis DSL
Author : Vincent St. Amour and Neil Toronto
Depends: None

Converts a description of notes and drum beats to `WAV` format. Modules in the benchmark come from two sources, a music library and an array library.

gregor

Origin : Library
Purpose: Date and time library
Author : Jon Zeppieri
Depends: `cldr`, `tzinfo`

Provides tools for manipulating calendar dates. The benchmark builds tens of date values and runs unit tests on these values.

quadT, quadU

Origin : Application
Purpose: Typesetting
Author : Matthew Butterick
Depends: `csp`

Converts S-expression source code to `PDF` format. The two versions of this benchmark came from the original author. First, `quadU` is based on a foundational untyped codebase. Second, `quadT` comes from a migrated, typed codebase with slightly different behavior. Overhead is worse in `quadT`, but the types in `quadU` are far less descriptive.

Figure 8 tabulates the size of the migratable code in the benchmark programs. The column labeled **N** reports the number of migratable modules; the configuration space for each program has 2^N points. The SLOC column reports lines of code in the fully-typed configuration. With type annotations, these benchmarks gain between 10 and 300 lines of code. For details about the graph structure of each benchmark and the specific types on boundaries, refer to the GTP web page: `docs.racket-lang.org/gtp-benchmarks/index.html`.

Benchmark	**N**	SLOC
sieve	2	52
forth	4	300
fsm	4	248
fsmoo	4	279
mbta	4	357
morsecode	4	264
zombie	4	342
dungeon	5	624
jpeg	5	1599
zordoz	5	1624
lnm	6	636
suffixtree	6	671
kcfa	7	296
snake	8	209
take5	8	364
acquire	9	2012
tetris	9	357
synth	10	974
gregor	13	1156
quadT	14	7019
quadU	14	7055

Figure 8: Static characteristics of the migratable code in the GTP benchmarks. **N** = number of components = number of modules. SLOC = source lines of fully-typed code as reported by David A. Wheeler's `sloccount`.

Benchmark	typed/untyped
sieve	0.97
forth	0.65
fsm	0.54
fsmoo	0.88
mbta	1.63
morsecode	0.73
zombie	1.79
dungeon	0.99
jpeg	0.40
zordoz	1.35
lnm	0.64
suffixtree	0.69
kcfa	1.04
snake	0.96
take5	0.97
acquire	1.22
tetris	0.97
synth	0.96
gregor	0.98
quadT	0.99
quadU	0.79

Figure 9: Coarse ratios for the GTP benchmarks v6.0 on Racket v7.7.

3.5.3 *Performance Ratios*

Figure 9 lists the overhead of fully-typed code relative to untyped code. In sieve, for example, the typed configuration runs slightly faster than untyped. In mbta, the typed configuration is over 1.5x slower because of a boundary to an untyped contextual module.

Overall, many benchmarks run significantly faster with types (8 of 21). These programs have few boundaries to untyped contextual modules and benefit from type-directed compilation. The highest ratios stay within a modest 2x overhead.

3.5.4 *Overhead Plots*

Figures 10, 11, and 12 present an exhaustive evaluation in a series of overhead plots. As in figure 4, the plots are cumulative distribution functions for the proportion of *D*-deliverable configurations.

Many curves are quite flat; they demonstrate that migratory typing introduces large and widespread performance overhead in the corresponding benchmarks. Among benchmarks with fewer than six

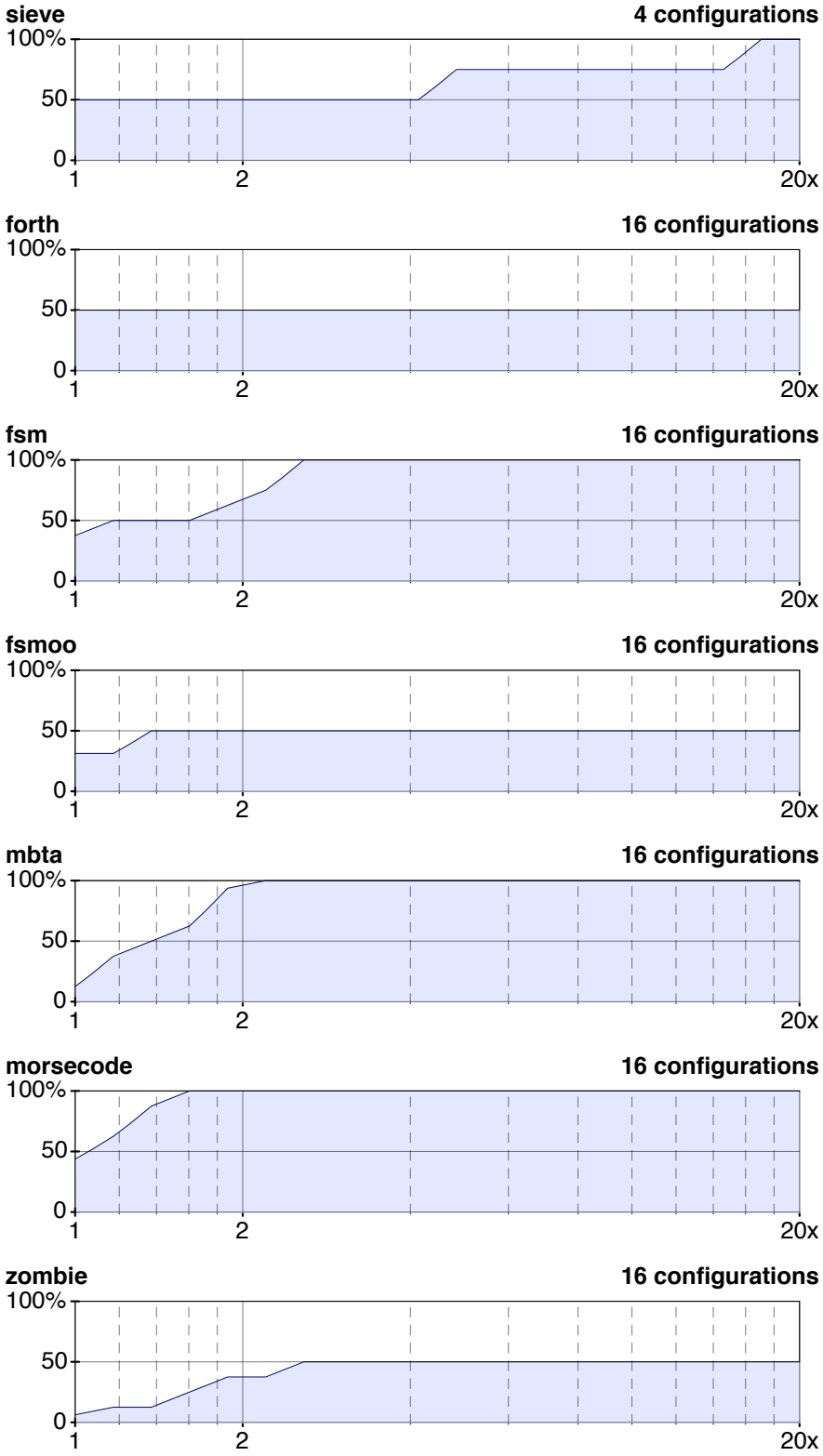


Figure 10: Typed Racket overhead plots (1/3). The x-axis ranges over slowdown factors, the y-axis counts configurations, and a point (x, y) shows the proportion of x-deliverable configurations.

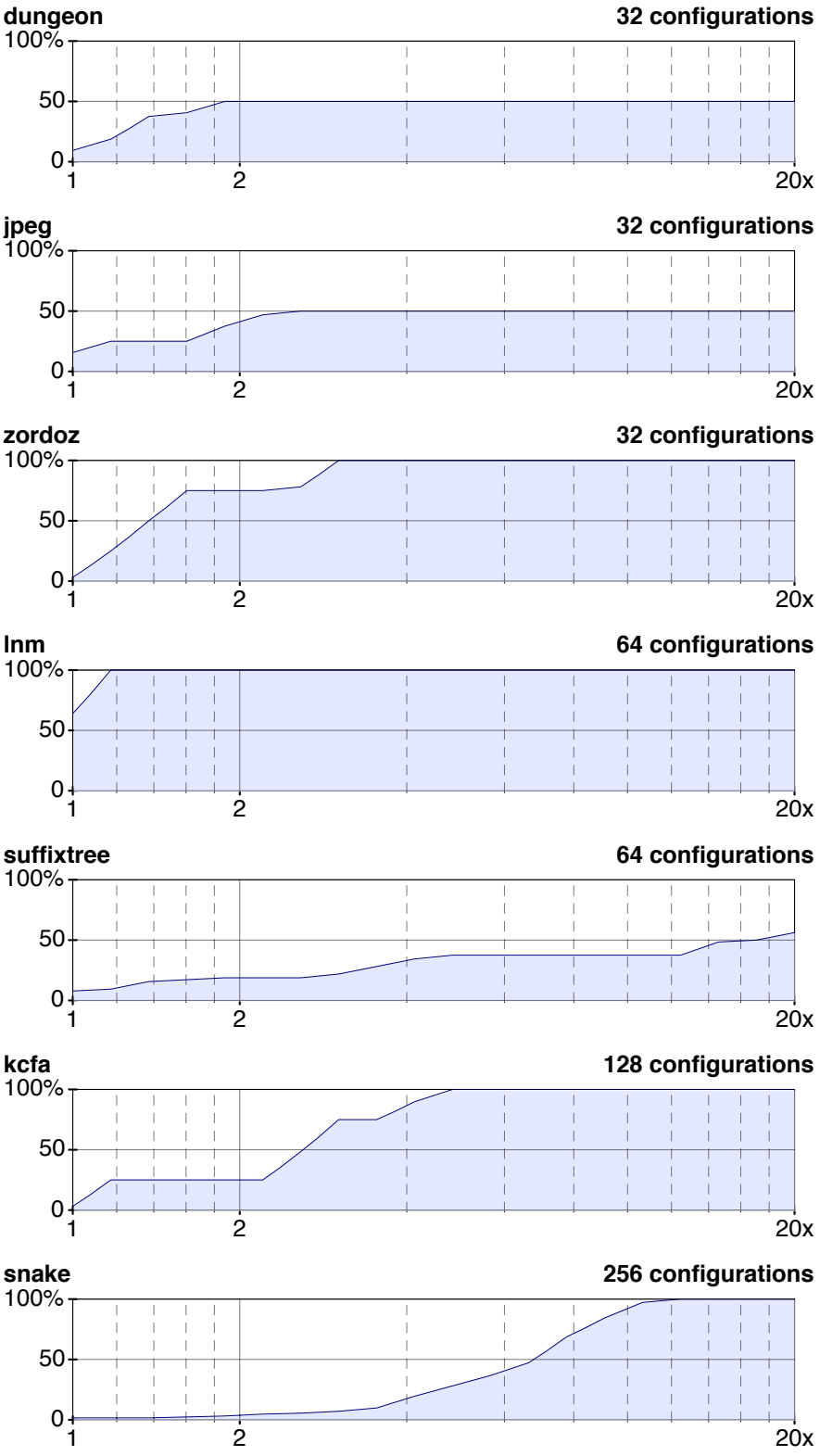


Figure 11: Typed Racket overhead plots (2/3).

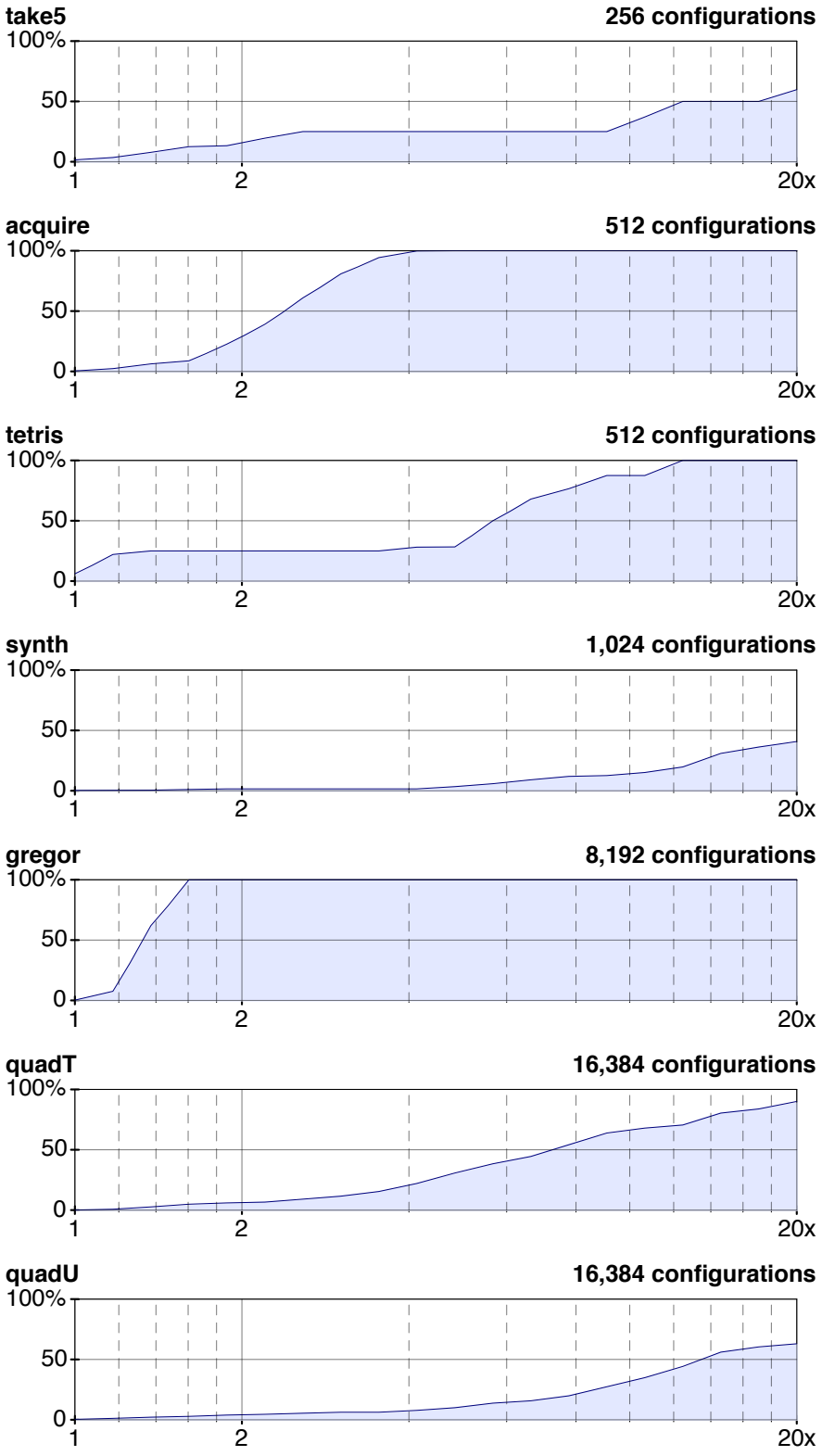


Figure 12: Typed Racket overhead plots (3/3).

modules, the most common shape is a flat line near the 50% mark. Such lines imply that the performance of a group of configurations is dominated by a single type boundary. For instance, there is one type boundary in fsmoo that adds overwhelming slowdown when present; all eight configurations with this boundary have over 20 overhead. Benchmarks with six or more modules generally have smoother slopes, but five such benchmarks have essentially flat curves. The overall message is that for many values of D between 1 and 20, few configurations are D-deliverable.

In 15 benchmarks, no more than half the configurations are 2-deliverable. This is quite bad. The situation is worse for lower (more realistic) overheads, and rarely improve at slightly higher overheads. Even at a generous 10x factor, no more than half the configurations in nine benchmarks are good enough.

The curves' endpoints describe the extremes of migratory typing. The left endpoint gives the percentage of configurations that run at least as quickly as the untyped configuration. With few exceptions, notably lnm, these configurations are a low proportion of the total. The right endpoint shows how many configurations suffer at least 20x overhead. Ten benchmarks have at least one such configuration.

In summary, the application of the evaluation method projects a negative image of Typed Racket's sound migratory typing. Only a small number of configurations in the benchmark suite run with low overhead; a mere 16% of all configurations are 1.4-deliverable on Racket v7.7. Many demonstrate extreme overhead; only 81% of all configurations are 20-deliverable on v7.7.

3.5.5 *Threats to Validity*

The concerns raised in chapter 3.4.1 affect this evaluation. In particular, each benchmark explores one choice of types. Different types may lead to different conclusions, as quadT and quadU demonstrate at a small scale.

The kcfa benchmark is modularized according to comments in the original, single-module program. If the original author had made the split, there could be different overhead.

The suffixtree, synth, and gregor benchmarks each have a single file containing all their data structure definitions, but the original programs defined these structures in the same module as the functions on the structures. Additionally, snake and tetris define their data structures in one file. The boundary to the data is often expensive, and may lead to higher overheads than a typical design that puts tightly-coupled functions next to the data.

The gregor, mbta, quad, and zordoz benchmarks depend on untyped libraries. To enable library use from typed code, each benchmark comes with a typed interface file. The interface solution is standard

for Typed Racket users, but it adds overhead that may skew the picture of costs within the benchmark. If the libraries came with typed interface files, then an experiment could use those contextual modules with no questions. As it stands, our choice of interface types constitutes a threat. Other interface choices may lead to different conclusions. Alternatively, it may be best to remove these costs by typing the libraries or (unsafely) trusting the interface.

3.6 APPLICATION 2: RETICULATED PYTHON

Reticulated Python is the original home of the transient semantics for mixed-typed programs [113]. Transient is a type-sound semantics (chapter 4) that does not rely on higher-order wrappers or full runtime checks. Instead, transient uses light "shape checks" throughout typed code. One would expect fast performance from transient on mixed-typed code. The first-ever evaluation, however, only reports data for untyped and fully-typed programs [115].

This subchapter presents a systematic evaluation of Reticulated without its experimental blame algorithm [44]. The data offers a big-picture view of transient, further validates the approximate method, and identifies bugs in the measured version of Reticulated. Overall, transient checks never exceed a 10x slowdown in the benchmarks.

3.6.1 *Protocol*

GRANULARITY The granularity of this evaluation is *function and class fields*. One syntactic unit in the experiment is either one function, one method, or the collection of all fields for one class. Figure 13 demonstrates this granularity with a simple Reticulated module. The class `Cash` has two fields and one method that requires three arguments; the module also include a function that instantiates a `Cash` object with exactly 5 dollars. Reticulated permits the removal of every type in the figure, giving 128 possible configurations. The granularity for our experiment, however, explores the 8 configurations obtained by removing types from the field declaration, the method, and/or the function each as a complete unit.

DATA COLLECTION The data is exhaustive for benchmarks with at most 2^{17} configurations and approximate for larger benchmarks. The approximations use ten samples each containing $10*(F+C)$ configurations, where F is the number of functions in the benchmark and C is the number of classes.

All data comes from jobs that we ran on the *Karst at Indiana University* computing cluster. Each job:

1. reserved all processors on one node;

```
@fields({"dollars": Int, "cents":Int})
class Cash:
  def __init__(self:Cash, d:Int, c:Int)->Void:
    self.dollars = d
    self.cents = c

def make_five()->Cash:
  return Cash(5, 0)
```

Figure 13: Reticulated code that leads to 2^3 configurations in the experiment, but supports 2^7 total.

2. downloaded fresh copies of Python 3.4.3 and Reticulated commit `e478343`;

3. repeatedly: selected a random configuration from a random benchmark, ran the configuration's main module 40 times, and recorded the result of each run.

Cluster nodes are IBM NeXtScale nx360 M4 servers with two Intel Xeon E5-2650 v2 8-core processors, 32 GB of RAM, and 250 GB of local disk storage. All data collection scripts are online: `github.com/nuprl/retic_performance`

3.6.2 *Benchmarks*

There are twenty-one benchmarks in total. Five originate from case studies by Vitousek et al. [114]. Twelve are from the evaluation by Vitousek et al. [115] on programs from the Python Performance Benchmark Suite. The remaining Four originate from open-source programs.

The following descriptions credit each benchmark's original author, state whether the benchmark depends on any contextual modules, and briefly summarize its purpose.

fannkuch

Origin : `pyperformance` *Author* : Sokolov Yura
Purpose: Test integers, vectors *Depends*: None

Implements a classic LISP microbenchmark [8].

nqueens

Origin : `pyperformance` *Author* : unknown
Purpose: Puzzle *Depends*: None

Solves the 8-queens problem by a brute-force algorithm.

http2

Origin : Library — *Author* : Joe Gregorio
Purpose: HTTP utilities — *Depends*: `urllib`

Converts a collection of Internationalized Resource Identifiers to equivalent ASCII resource identifiers.

nbody

Origin : `pyperformance` — *Author* : Kevin Carson
Purpose: Test float ops — *Depends*: None

Models the orbits of Jupiter, Saturn, Uranus, and Neptune.

pidigits

Origin : `pyperformance` — *Author* : unknown
Purpose: Test big integer ops — *Depends*: None

Microbenchmarks big-integer arithmetic.

spectralnorm

Origin : `pyperformance` — *Author* : Sebastien Loisel
Purpose: Test arithmetic — *Depends*: None

Computes the largest singular value of an infinite matrix.

call_simple

Origin : `pyperformance` — *Author* : unknown
Purpose: Test function calls — *Depends*: None

Same as call_method, using functions rather than methods.

float

Origin : `pyperformance` — *Author* : Factor
Purpose: Test float ops — *Depends*: `math`

Microbenchmarks floating-point operations.

call_method

Origin : `pyperformance` — *Author* : unknown
Purpose: Test method calls — *Depends*: None

Microbenchmarks simple method calls; the calls do not use argument lists, keyword arguments, or tuple unpacking.

go

Origin : `pyperformance` — *Author* : unknown
Purpose: Game — *Depends*: `math`, `random`

Implements the game Go. This benchmark is split across three files: a migratable module that implements the game board, a contextual module that defines constants, and a contextual module that implements an AI and drives the benchmark.

meteor

Origin : `pyperformance` — *Author* : Daniel Nanz
Purpose: Puzzle — *Depends*: None

Solves the Shootout benchmarks meteor puzzle.

Espionage

Origin : Synthetic
Purpose: Graph algorithm
Author : Zeina Migeed
Depends: `operator`

Implements Kruskal's spanning-tree algorithm.

PythonFlow

Origin : Synthetic
Purpose: Flow algorithm
Author : Alfian Ramadhan
Depends: `os`

Implements the Ford-Fulkerson algorithm.

pystone

Origin : `pyperformance`
Purpose: Test integer ops
Author : Chris Arndt
Depends: None

Implements Weicker's Dhrystone benchmark.

chaos

Origin : `pyperformance`
Purpose: Create fractals
Author : Carl Friedrich Bolz
Depends: `math`, `random`

Creates fractals using the chaos game method.

futen

Origin : Library
Purpose: SSH configuration
Author : `momijiame`
Depends: `fnmatch`, `os.path`, `re`, `shlex`, `socket`

Converts an OpenSSH configuration file to an inventory file for the Ansiable automation framework.

take5

Origin : Educational
Purpose: Game
Author : Maha Alkhairy and Zeina Migeed
Depends: `random`, `copy`

Implements a card game and a simple player AI.

slowSHA

Origin : Library
Purpose: Hashing
Author : Stefano Palazzo
Depends: `os`

Applies the SHA-1 and SHA-512 algorithms to English words.

sample_fsm

Origin : Economics research
Purpose: Economy simulator
Author : Zeina Migeed
Depends: `itertools`, `os`, `random`

Adapted from the Typed Racket fsm benchmark.

aespython

Origin : Library
Purpose: Encryption
Author : Adam Newman, Demur Remud
Depends: `os`, `struct`

Implements the Advanced Encryption Standard.

Benchmark	**N**	SLOC	modules	functions	classes	methods
fannkuch	1	41	1	1	-	-
nqueens	2	37	1	2	-	-
http2	4	86	2	1	1	2
nbody	5	101	1	5	-	-
pidigits	5	33	1	5	-	-
spectralnorm	5	31	1	5	-	-
call_simple	6	113	1	6	-	-
float	6	36	1	2	1	3
call_method	7	115	1	1	1	5
go	7	80	1	1	1	5
meteor	8	100	1	8	-	-
Espionage	12	93	2	7	1	4
PythonFlow	12	112	1	-	1	11
pystone	14	177	1	11	1	2
chaos	15	190	1	-	3	12
futen	15	221	3	5	2	8
take5	16	130	3	3	2	11
slowSHA	17	210	4	4	3	10
sample_fsm	19	148	5	8	2	9
aespython	34	403	6	-	5	29
stats	79	1118	13	79	-	-

Figure 14: Static summary of the migratable code in the Reticulated benchmarks. **N** = number of components = functions + classes + methods. SLOC = source lines of code as reported by David A. Wheeler's `sloccount`.

stats

Origin : Library — *Author* : Gary Strangman
Purpose: Statistics — *Depends*: `copy`, `math`

Implements first-order statistics functions; in other words, transformations on either floats or (possibly-nested) lists of floats. The original program consists of two modules. The benchmark is modularized according to comments in the program's source code to reduce the size of each module's configuration space.

Figure 14 tabulates information about the size and structure of the migratable portions of these benchmarks. The six columns report the number of migratable units (**N** = num. functions + methods + classes), lines of code (SLOC), number of modules, number of function definitions, number of classes, and number of method definitions. Most benchmarks are small, with 1–3 modules and fewer than 200 lines of code. The number of mixed-typed configurations in the experiment,

Benchmark	retic/python	typed/retic	typed/python
fannkuch	1.14	1.01	1.15
nqueens	1.25	1.57	1.96
http2	3.07	1.18	3.63
nbody	1.78	1.01	1.80
pidigits	1.02	1.02	1.05
spectralnorm	2.01	3.47	6.98
call_simple	1.00	3.10	3.11
float	2.18	1.52	3.32
call_method	4.48	1.74	7.79
go	3.77	1.97	7.44
meteor	1.56	1.37	2.13
Espionage	2.87	1.79	5.14
PythonFlow	2.38	3.04	7.23
pystone	1.36	2.06	2.79
chaos	2.08	1.77	3.69
futen	1.58	1.06	1.68
take5	1.21	1.14	1.38
slowSHA	1.66	1.18	1.96
sample_fsm	2.80	2.16	6.07
aespython	3.41	1.74	5.93
stats	1.09	1.39	1.52

Figure 15: Performance ratios for three important points in a configuration space: fully-typed code (typed), untyped code run through Reticulated (retic), and untyped code run via Python (python).

however, is prohibitively large. The relatively small sample_fsm describes half a million configurations. For the largest two benchmarks, aespython and stats, exhaustive measurement is out of the question.

3.6.3 *Performance Ratios*

The table in figure 15 lists the endpoints of migratory typing in Reticulated. From left to right, these are: the performance of the untyped configuration relative to the Python baseline (the retic/python ratio), the performance of the fully-typed configuration relative to the untyped configuration (the typed/retic ratio), and the overall delta between fully-typed and Python (the typed/python ratio).

For example, the row for futen reports a retic/python ratio of 1.58. This means that the average time to run the untyped configuration of the futen benchmark using Reticulated is that much slower than the average time of running the same code using Python. The typed/retic

ratio for futen states that the fully-typed configuration is 1.06 times slower than the untyped configuration.

Migrating a benchmark to Reticulated, or from untyped to fully-typed, always adds performance overhead. This overhead is always within one order of magnitude.

Fourteen benchmarks have retic/python ratios that are larger than their typed/retic ratios. One would expect retic/python ratios close to 1 because untyped Reticulated need not differ from Python. But Reticulated duplicates some of Python's run-time checks. For example, Reticulated checks that a method is bound before proceeding with method dispatch.

3.6.4 *Overhead Plots*

Figures 16, 17, and 18 summarize the overhead of migratory typing in the benchmark programs. Each plot reports the percent of D-deliverable configurations (y-axis) for values of D between 1x overhead and 10x overhead (x-axis). The heading above a plot states the benchmark's name and indicates whether the data is exhaustive or approximate. Exhaustive plots show the total number of configurations. Approximate plots show the number of samples and the number of randomly-selected configurations in each sample.

The curves for the approximate data—sample_fsm, aespython, and stats—are intervals rather than fixed-width lines. For instance, the height of an interval at $x = 4$ is the range of the 95%-10, $[10(F + C)]$-approximation for the number of 4-deliverable configurations. These intervals are thin because there is little variance in the proportion of D-deliverable configurations across the ten samples, but the sample_fsm curve is slightly thicker than the aespython curve.

Curves in these figures typically cover a large area and reach the top of the y-axis at a low value of D. This value is always less than 10. In other words, every configuration in the experiment is 10-deliverable. For many benchmarks, the maximum overhead is significantly lower. Indeed, eight benchmarks are nearly 2-deliverable.

None of the configurations in the experiment run faster than the Python baseline. This is to be expected, given the retic/python ratios in figure 15 and the fact that Reticulated translates type annotations into run-time checks.

Fourteen benchmarks have relatively smooth slopes. The plots for the other four benchmarks have wide, flat segments. These flat segments are due to functions that are frequently executed in the benchmarks' traces; all configurations in which one of these functions is typed incur a significant performance overhead.

Eighteen benchmarks are roughly T-deliverable, where T is the typed/python ratio listed in figure 15. In these benchmarks, the fully-typed configuration is one of the slowest configurations. The

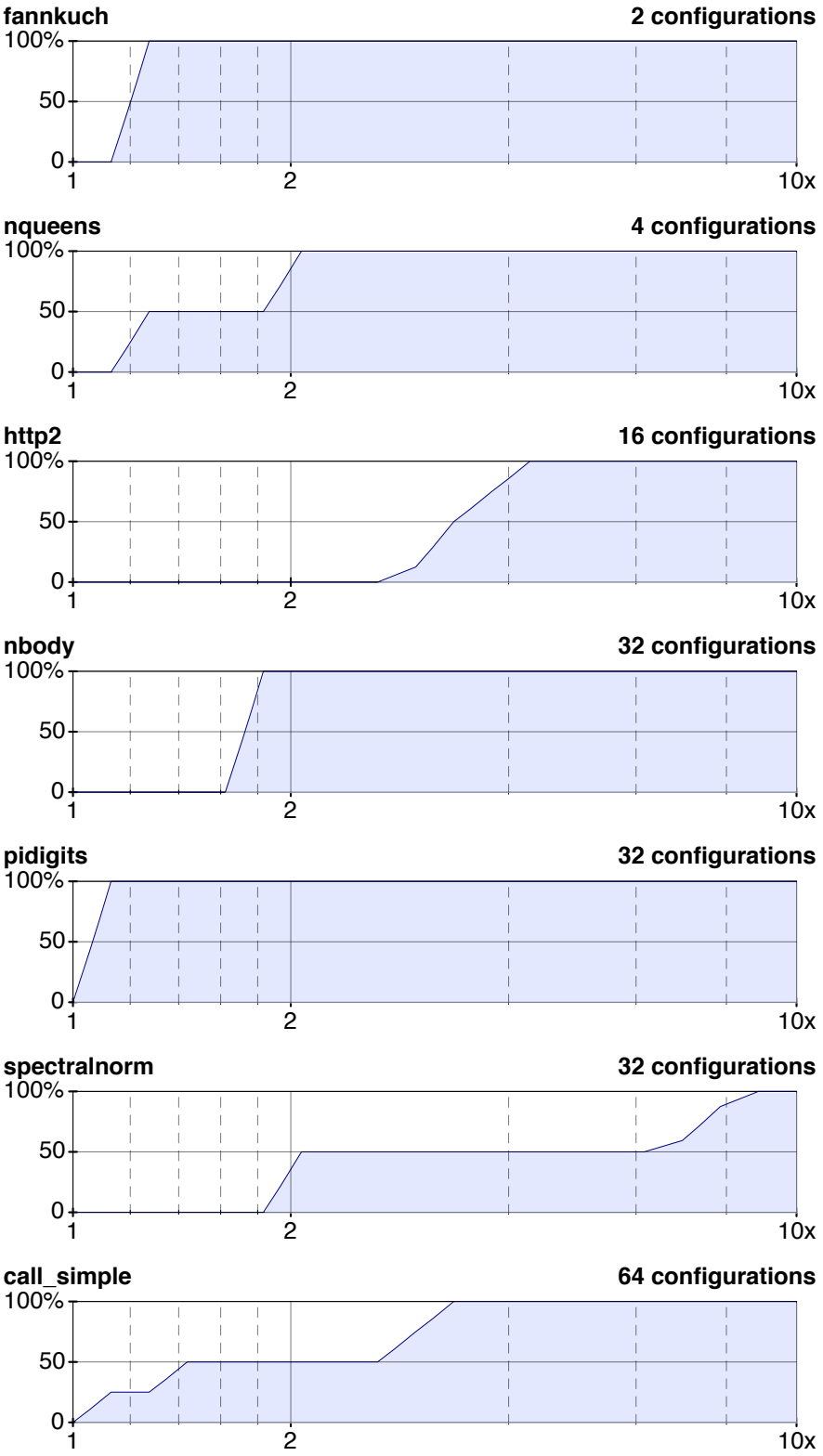


Figure 16: Reticulated overhead plots (1/3). The x-axis ranges over slowdown factors, the y-axis counts configurations, and a point (x, y) shows the proportion of x-deliverable configurations.

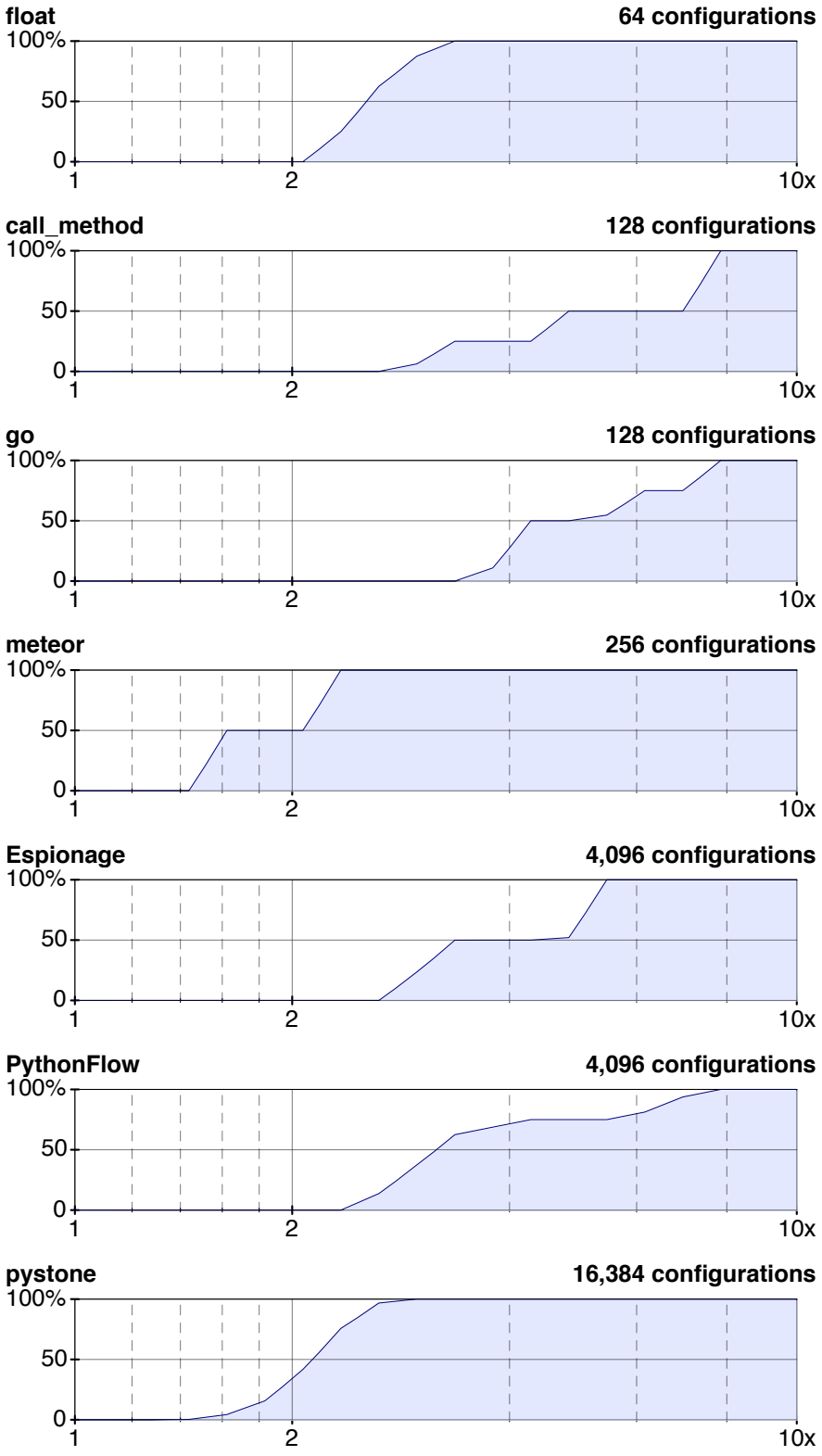


Figure 17: Reticulated overhead plots (2/3).

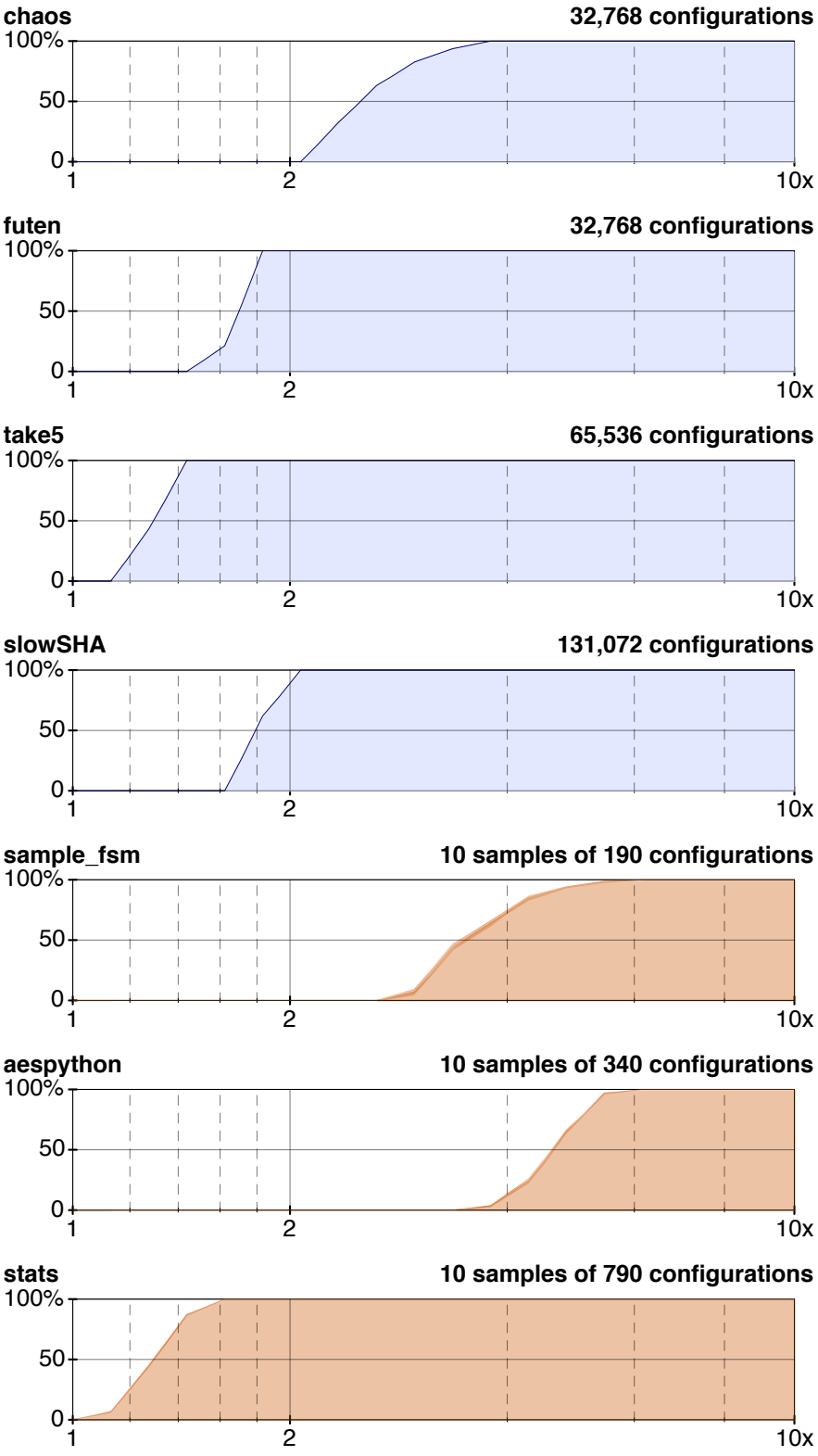


Figure 18: Reticulated overhead plots (3/3).

notable exception is spectralnorm, in which the fully-typed configuration runs faster than 38% of all configurations. Unfortunately, this speedup comes from a soundness bug that we discovered thanks to this performance evaluation. Reticulated at commit `e478343` does not type-check the contents of tuples (`mvitousek/reticulated #36`).

3.6.5 *Threats to Validity*

We have identified five sources of systematic bias. Three have been noted above: the decision to measure one set of type annotations (chapter 3.2.3), the coarse granularity (chapter 3.6.1), and the imprecision of Reticulated types (chapter 3.4.1). Here, we can offer a few details on type-expressiveness. The take5 benchmark contains one function that must stay untyped because it accepts optional arguments (`mvitousek/reticulated #32`). The go benchmark uses dynamic typing because Reticulated cannot validate its use of a recursive class definition. Two other benchmarks, pystone and stats, use dynamic typing to overcome Reticulated's lack of untagged union types.

A third issue is that the experiment uses rather small benchmarks. The PyPI Ranking (`pypi-ranking.info/alltime`, accessed 2018) shows that widely-used Python packages have far more functions and methods. The `simplejson` library contains over 50 functions and methods, the `requests` library contains over 200, and the `Jinja2` library contains over 600.

Fourth and last, the aespython, futen, http2, and slowSHA benchmarks read from a file within their timed computation. Despite the unpredictable running times of system calls, we believe our results are representative.

3.7 ADDITIONAL VISUALIZATIONS

The method presented in this chapter targets our most effective answer to the question of how to evaluate the performance of a mixed-typed language. In particular, the notion of *D*-deliverable configurations is a clear and scalable way to summarize performance. A mixed-typed language has other interesting properties, however, and these call for tailored visualizations.

3.7.1 *Exact Runtime Plots*

The raw data behind an overhead plot is a sequence of running times for every configuration. An overhead plot summarizes the running times into an average, and uses these averages to group configurations into buckets. Unfortunately, this method hides outliers in the data and syntactic relations (think back to the lattice, figure 2) among configurations.

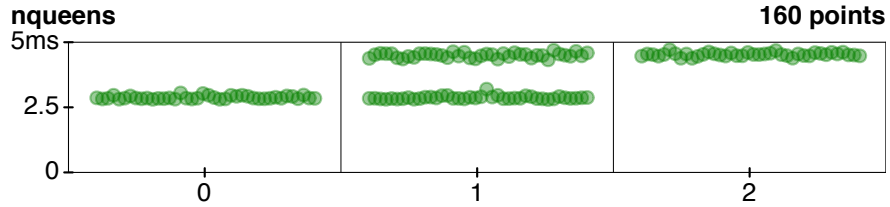


Figure 19: Number of type annotations vs. Running time on the Reticulated nqueens benchmark.The x-axis ranges over the number of active typed units, the y-axis shows exact running times, and a point (x, y) shows one running time for one configuration with x types.

Figure 19 addresses both concerns. Instead of summarizing one configuration with its average runtime, the plot contains one point for every running time in the dataset. These points are spread left-to-right in one of the three columns of the figure. If a plot like this does not consist of distinct, horizontal lines, the underlying dataset may have irregular running times. Each column contains all configurations that have the same number of types. In terms of the configuration lattice (figure 2), the left-most column contains the bottom level and each successive column present a higher levels. At a glance, figure 19 therefore shows the overall effect of adding types.

3.7.2 *Relative Scatterplots*

Collapsible contracts are a new representation for deep run-time type checks [29]. The representation greatly improves some mixed-typed programs, but can slow down others. To assess the implementation of collapsible, we used a scatterplot technique due to Spenser Bauman [10]. Figure 20 shows one representative example from our work. Each point in the scatterplot shows how collapsible affects one configuration. Points above the diagonal line are improved; points below the line get worse with collapsible contracts. More precisely, a point (X, Y) shows the overhead in both systems. The first coordinate, X, is the overhead with collapsible. The Y coordinate is the baseline overhead, without collapsible. If collapsible always led to a lower overhead, then all points would lie above the $X = Y$ line (because $X < Y$ when collapsible wins).

3.7.3 *Best-Path Plots*

The plots in chapter 3.5 paint a bleak picture of Typed Racket. Many benchmarks have many configurations that run far slower than the

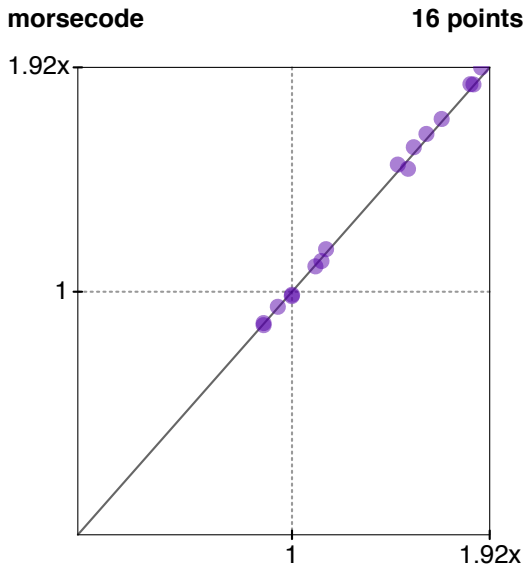


Figure 20: Scatterplot comparing morsecode configurations before and after collapsible contracts. The x-axis ranges over collapsible overhead and the y-axis ranges over baseline overhead. A point (x, y) is a head-to-head comparison; points above the diagonal represent configurations where collapsible is faster than the baseline.

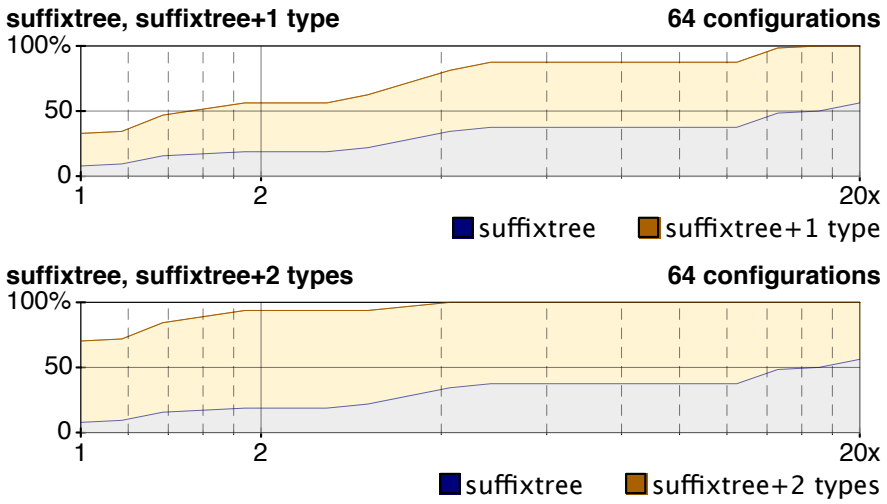


Figure 21: Overhead plots for suffixtree comparing the base slowdown to the best-possible improvement after adding types to 1 (top) and 2 (bottom) modules.

untyped code. A natural question, though, is whether these results are brittle. If a programmer can escape the slow configurations by converting one or two more modules, then the bleak conclusion may be unwarranted.

Figure 21 presents two plots for the suffixtree benchmark that compare the original data against the best-possible performance after converting additional modules. In the top plot, a configuration is X-deliverable in the orange curve if it can reach a X-deliverable configuration from the blue curve after adding types to one module. Similarly, the bottom plot shows the best-possible performance after typing two untyped modules.

The plots show that significant improvements are possible, but not guaranteed even in the best case. Other large benchmarks typically show similar patterns [46].

4

DESIGN ANALYSIS METHOD

This chapter is based on joint work with: Christos Dimoulas and Matthias Felleisen [43, 45, 47].

Over the years, researchers have developed several languages that mix typed and untyped code. Typed Racket and Reticulated are but two implementations in a wide space. To a first approximation, the designs fall into four broad strategies:

- *Optional* typing adds a best-effort static analysis but ignores type annotation at runtime [12, 15].
- *Transient* inserts shape checks in type-checked code to guarantee that operations cannot not "go wrong" due to untyped values [113, 115]. A shape check validates a top-level value constructor with respect to a top-level type constructor, which is just enough for this notion of safety.
- *Natural* enforces types with higher-order checks and thereby ensures the full integrity of types [86, 103].
- *Concrete* requires that every value is tagged with a type and maintains integrity with simple checks [72, 123].

In addition, though, researchers have proposed and implemented hybrid techniques [13, 41, 43, 82, 85]. An outstanding and unusual exemplar of this kind is Pyret, a language targeting the educational realm (`pyret.org`).

Each of these type-enforcement strategies picks a tradeoff among static guarantees, expressiveness, and run-time costs (chapter 4.2). If stringent constraints on untyped code are acceptable, then *concrete* offers strong and inexpensive guarantees. If the goal is to interoperate with an untyped language that does not support wrapper/proxy values, then *transient* may offer the strongest possible guarantees. If performance is not an issue, then *natural* is the perfect choice.

Unfortunately, the literature provides little guidance to programmers and language designers on how to compare different semantics. Standard meta-theoretical tools do not articulate what is gained and lost in each tradeoff (chapter 4.3). The gradual guarantee [87], for example, is trivially satisfied by any optionally-typed language. Simply put, the field lacks an apples-to-apples way of comparing different type-enforcement strategies and considering their implications for programmers.

Table 1: Informal sketch of the design-space analysis.

	N		C		F		T		A		E
type soundness	✓		✓		✓		✓		✓		×
complete monitoring	✓		✓		×		×		×		×
blame soundness	✓		✓		✓		×		✓		✓
blame completeness	✓		✓		×		×		✓		×
error preorder	N	$\lesssim$	C	$\lesssim$	F	$\lesssim$	T	$\eqsim$	A	$\lesssim$	E

This chapter introduces a framework for systematically comparing the behavioral guarantees offered by different mixed-typed semantics. Because each semantics is essentially a method of enforcing static types, the comparison begins with a common mixed-typed syntax. This surface syntax is then assigned multiple semantics, each of which follows a distinct protocol for enforcing type specifications. With this semantic framework, one can directly observe the possible behaviors for a single program.

The chosen models illustrate *natural* (N), *transient* (T), *optional* (also known as *erasure*, E), and three theoretical strategies (*co-natural* C, *forgetful* F, and *amnesic* A) that demonstrate how to fill design gaps. The comparison excludes two classes of prior work: *concrete*, because of the constraints it places on untyped code (chapter 4.2.2), and mixed-typed languages that must analyze untyped code to interoperate with it. Our focus is on strategies that can deal with untyped code as a "dusty deck" without needing to recompile the untyped world each time a new type boundary appears.

Table 1 sketches the results of the evaluation (chapter 4.5). The six letters in the top row correspond to different type-enforcement strategies, and thus different semantics, for the common surface language. As to be expected, Natural (N) accepts the fewest programs without raising a run-time type mismatch, and Erasure (E) accepts the greatest number of programs; the symbols $\lesssim$ and $\eqsim$ indicate these behavioral differences. Lower rows introduce additional properties that underlie our comparison. Type soundness guarantees the validity of types in typed code. Complete monitoring guarantees that the type system moderates all boundaries between typed and untyped code—even boundaries that arise at run-time. Blame soundness ensures that when a run-time check goes wrong, the error message points to boundaries that are relevant to the problem. Blame completeness guarantees that error messages come with *all* relevant information. For both blame soundness and completeness, *relevance* is determined by an independent (axiomatic) specification that tracks values as they cross boundaries between typed and untyped code (chapter 4.4.4).

In sum, the five properties enable a uniform analysis of existing strategies and can guide the search for new strategies. Indeed, the

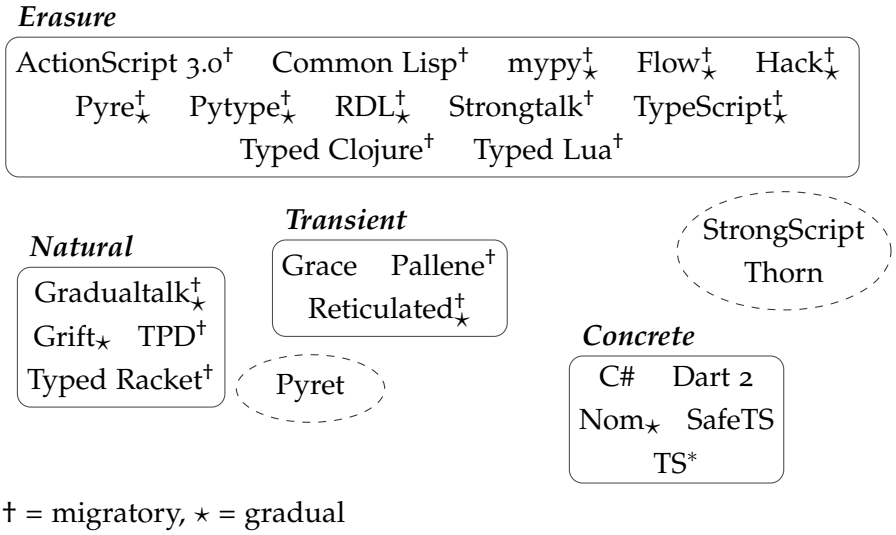


Figure 22: Landscape of mixed-typed languages

synthetic Amnesic semantics (A) demonstrates how a semantics can fail complete monitoring but guarantee sound and complete blame.

4.1 CHAPTER OUTLINE

Chapters 4.2 through 4.4 explain the *what*, *why*, and *how* of our design-space analysis. There is a huge body of work on mixed-typed language that desperately needs organizing principles (chapter 4.2). Past attempts to organize fall short; by contrast, the properties that frame table 1 offer an expressive and scalable basis for comparison (chapter 4.3). These properties guide an apples-to-apples method that begins with a common surface language and studies different semantics (chapter 4.4). In particular, this chapter analyzes six semantics based on six ideas for enforcing static types.

Chapter 4.5 presents the six semantics and the key results. Expert readers may wish to begin there and refer back to chapter 4.4 as needed. An appendix contains a complete formal account of our results.

4.2 ASSORTED BEHAVIORS BY EXAMPLE

There are many mixed-typed languages. Figure 22 arranges a few of their names into a rough picture of the design space. Each language enables some kind of mix between typed and untyped code. Languages marked with a star (⋆) come with a special dynamic type, often styled as ⋆, or ?, that allows partially-defined types [87]. Tech-

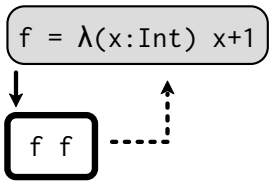
nically, the type system supports implicit down-casts from the dynamic type to any other type—unlike, say, `Object` in Java. Languages marked with a cross (†) add a tailor-made type system to an untyped language, but may require types for an entire module at a time [107]. Other languages satisfy different goals.

For the most part, these mixed-typed languages fit into the broad forms introduced in chapter 4. Erasure is by far the most popular strategy; perhaps because of its uncomplicated semantics and ease of implementation [14, 64, 81, 92]. The Natural languages come from academic teams that are interested in types that offer strong guarantees [5, 10, 104, 120]. Transient is gaining traction as a compromise between types and performance [48, 84, 115], and Concrete has generated interest among industry teams [11, 23] as well as academics [72, 79, 95]. Nevertheless, several languages explore a hybrid approach. StrongScript and Thorn offer a choice of concrete and erased types [82, 123]. Pyret uses Natural-style checks to validate fixed-size data and Transient-style checks for recursive types (e.g. lists) and higher-order types (personal communication with Benjamin Lerner and Shriram Krishnamurthi). The literature presents additional variations. Castagna and Lanvin [18] present a Natural semantics that drops certain wrappers. Siek et al. [85] give a monotonic semantics that associates types with heap positions instead of creating wrappers. There are several implementation of the monotonic idea [7, 79, 83, 95].

Our goal is a systematic comparison of type guarantees across the wide design space. Such a comparison is possible despite the variety because the different guarantees arise from choices about how to enforce types at the boundaries between type-checked code and arbitrary dynamically-typed code. To illustrate, the following three subchapters discuss type boundary examples in the context of four languages: Flow [20], Reticulated [115], Typed Racket [107], and Nom [72]. Flow is a migratory typing system for JavaScript, Reticulated equips Python with gradual types, Typed Racket extends Racket, and Nom is a new gradual-from-the-start language.

4.2.1 *Enforcing a Base Type*

One of the simplest ways that a mixed-typed interaction can go awry is for untyped code to send incorrect input to a typed context that expects a flat value. The first example illustrates one such interaction:



(1)

The typed function on top expects an integer. The untyped context on the bottom imports this function f and applies f to itself; thus the typed function receives a function rather than an integer. The question is whether the program halts or invokes the typed function f on a nonsensical input.

Figure 23 translates the program to four languages. Each white box represents type-checked code and each grey box represents untyped and, ideally, un-analyzed code that is linked in at run-time. Nom is an exception, however, because it cannot interact with truly untyped code (chapter 4.2.2). Despite the differences in syntax and types, each clearly defines a typed function that expects an integer on the top and applies the function to itself in an untyped context on the bottom.

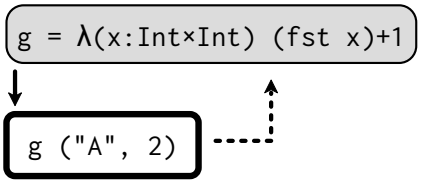
In Flow, the program does not detect a type mismatch. The typed function receives a function from untyped JavaScript and surprisingly computes a string (`ECMA-262` edition 10, § `12.8.3`). In the other three languages, the program halts with a *boundary error* message that alerts the programmer to the mismatch between two chunks of code.

Flow does not detect the run-time type mismatch because it follows the *erasure*, or optional typing, approach to type enforcement. Erasure is hands-off; types have no effect on the behavior of a program. These static-only types help find logical mistakes and enable type-directed IDE tools, but disappear during compilation. Consequently, the author of a typed Erasure function cannot assume that it receives only well-typed input.

The other languages enforce static types with some kind of dynamic check. For base types, the check validates the shape of incoming data. The checks for other types reveal differences among these non-trivial type enforcement strategies.

4.2.2 *Validating an Untyped Data Structure*

The second example is about pairs. It asks what happens when typed code declares a pair type and receives an untyped pair:



(2)

The typed function on top expects a pair of integers and uses the first element of the input pair as a number. The untyped code on the bottom applies this function to a pair that contains a string and an integer.

Figure 24 translates this idea into Reticulated, Typed Racket, and Nom. The encodings in Reticulated and Typed Racket define a pair in untyped code and impose a type in typed code. The encoding

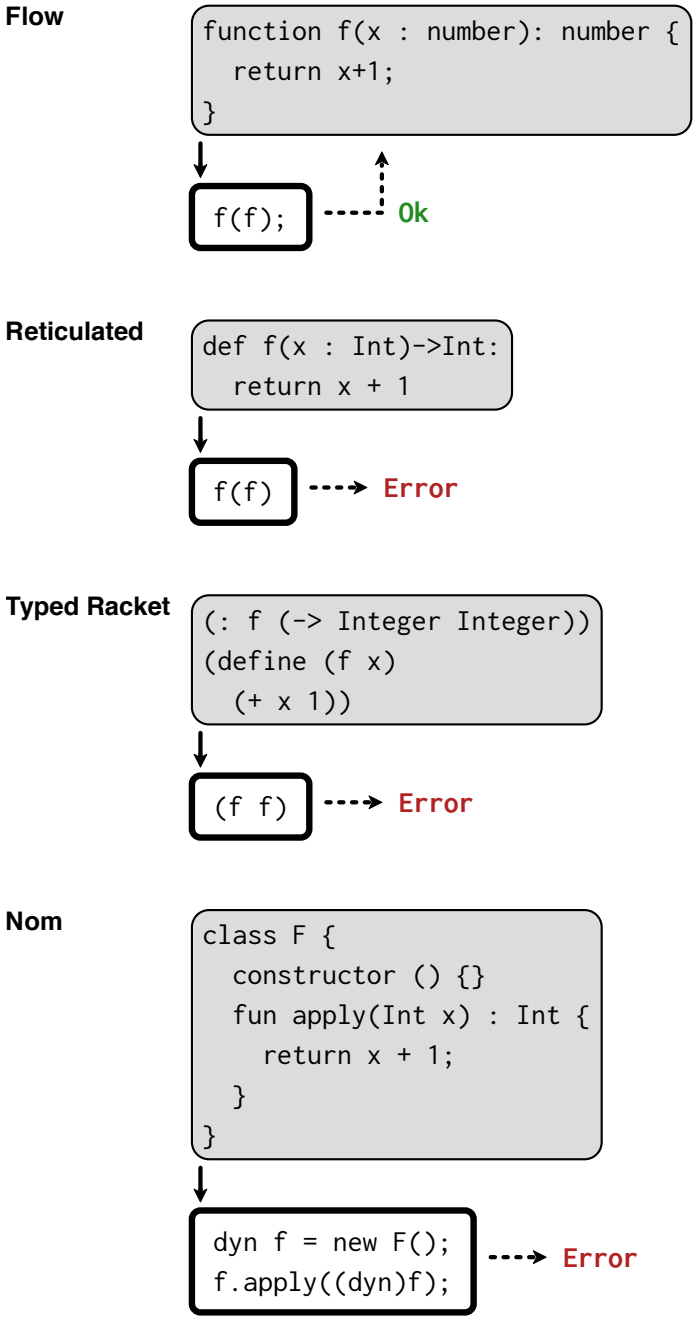


Figure 23: Program (1) translated to four languages

in Nom is different; the typed code expects an instance of one data structure but the untyped code provides something else. This shape mismatch leads to a run-time error.

Nom cannot express program (2) directly because the language does not allow partially-typed values. There is no common pair constructor that: (1) untyped code can use without constraints and (2) typed code can receive at a particular type. All type structure must be specified with the data structure. On one hand, this requirement greatly simplifies run-time validation because the outermost shape of any value determines the shape of its elements. On the other hand, it imposes a significant burden on the programmer. To add refined static type checking at the use-sites of an untyped data structure, a programmer must either add a cast to each use in typed code or edit the untyped code for a new data definition. Because Nom and other concrete languages require this kind of type structure in untyped code [23, 72, 82, 123], the model in chapter 4.5 does not support them.

Both Reticulated and Typed Racket raise an error on program (2), but for substantially different reasons. Typed Racket rejects the untyped pair at the boundary to the typed context because the pair does not fully match the declared type. Reticulated accepts the value at the boundary because it is a pair, but raises an exception at the elimination form `y[0]` because typed code expects an integer result but receives a string. These sample behaviors are indicative of a wider difference; Typed Racket eagerly checks the contents of data structures while Reticulated lazily validates use-sites.

4.2.3 *Uncovering the Source of a Mismatch*

Figures 25 and 26 present excerpts from realistic programs that mix typed and untyped code. These examples follow the same general structure: an untyped client interacts with an untyped library via a thin layer of typed code. Both programs also signal run-time errors, but for different reasons and with different implications for the programmer.

Figure 25 consists of an untyped library, an *incorrect* layer of type annotations, and an untyped client of the types. The module on top, `net/url`, is a snippet from an untyped library that has been part of Racket for two decades (`github.com/racket/net`). The typed module on the middle-right defines types for part of the untyped library. Lastly, the module at the bottom of the figure imports the typed library and calls the library function `call/input-url`.

Operationally, the library function flows from `net/url` to the typed module and then to the client. When the client calls this function, it sends client data to the untyped library code via the typed module. The client application clearly relies on the type specification from `typed/net/url` because the first argument is a URL structure, the sec-

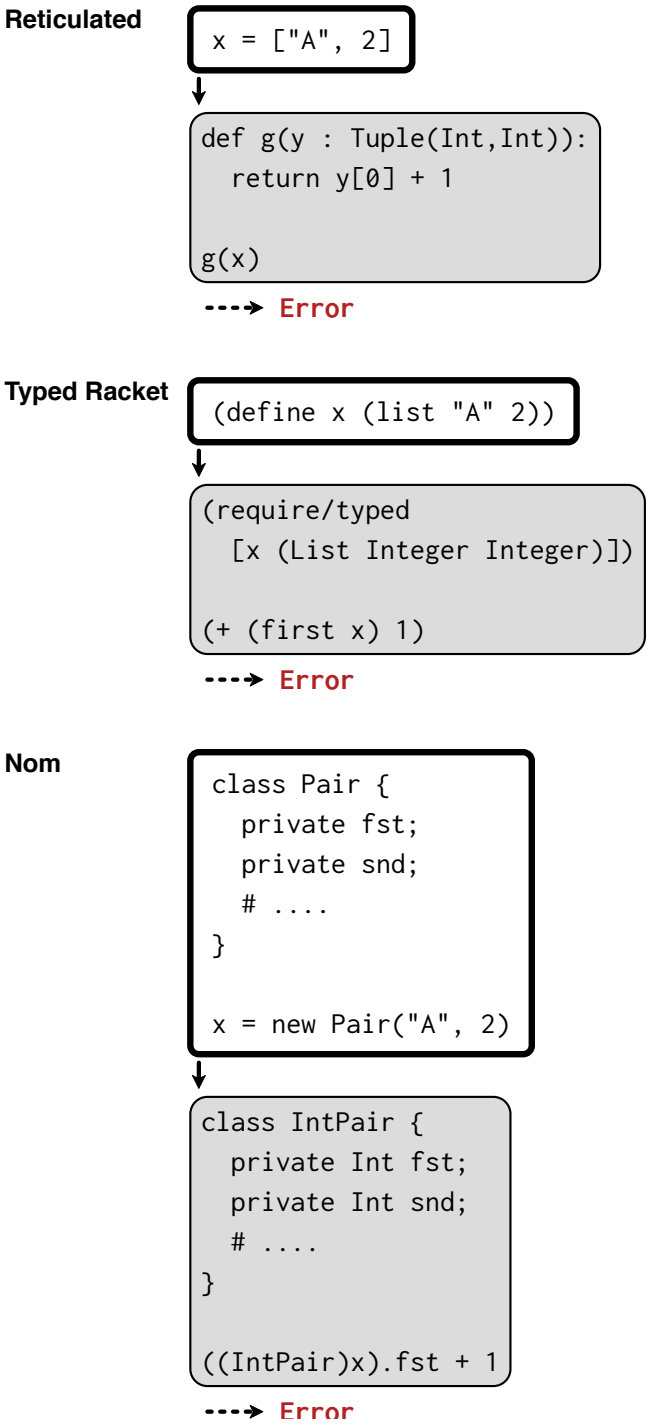


Figure 24: Program (2) translations

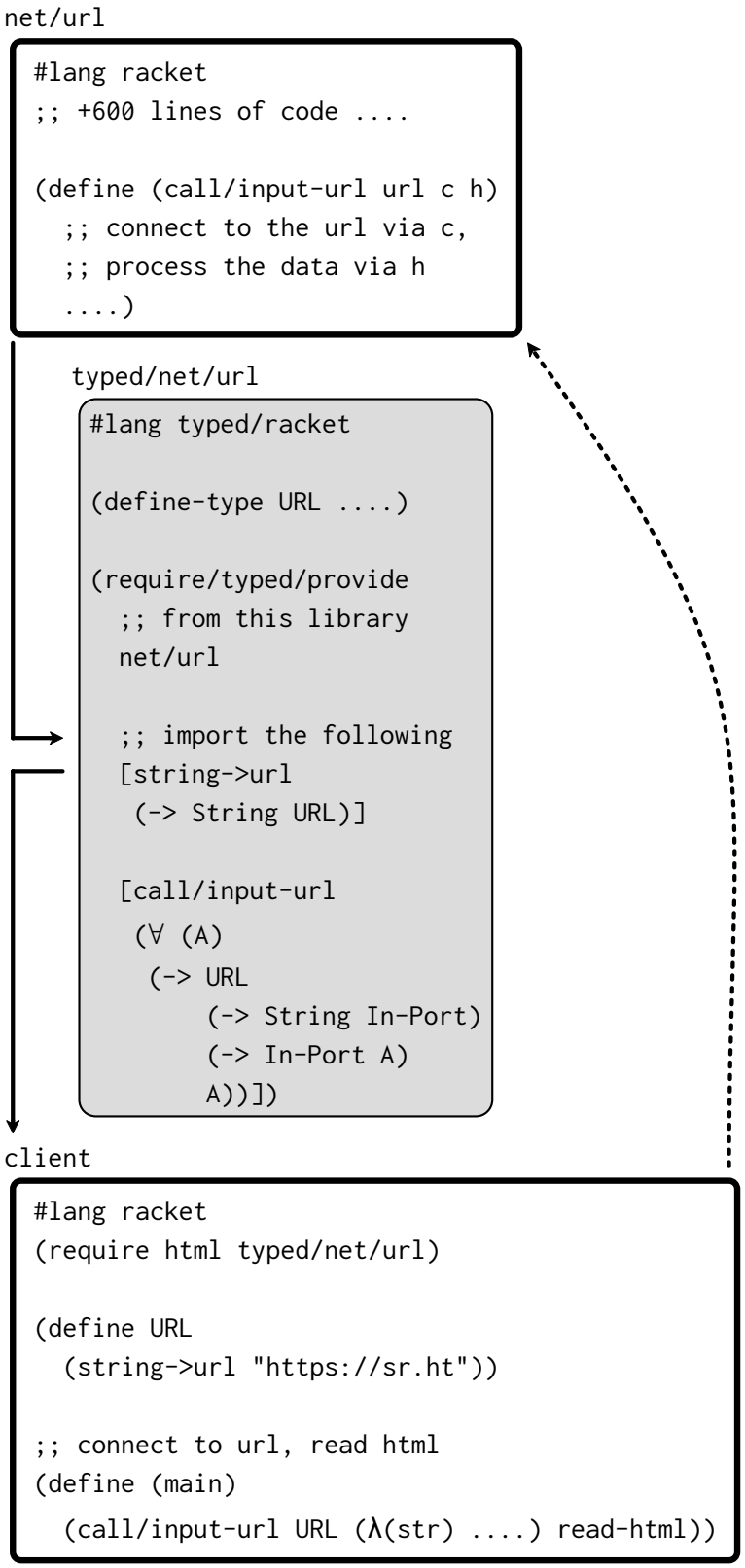


Figure 25: Using Typed Racket to define an API

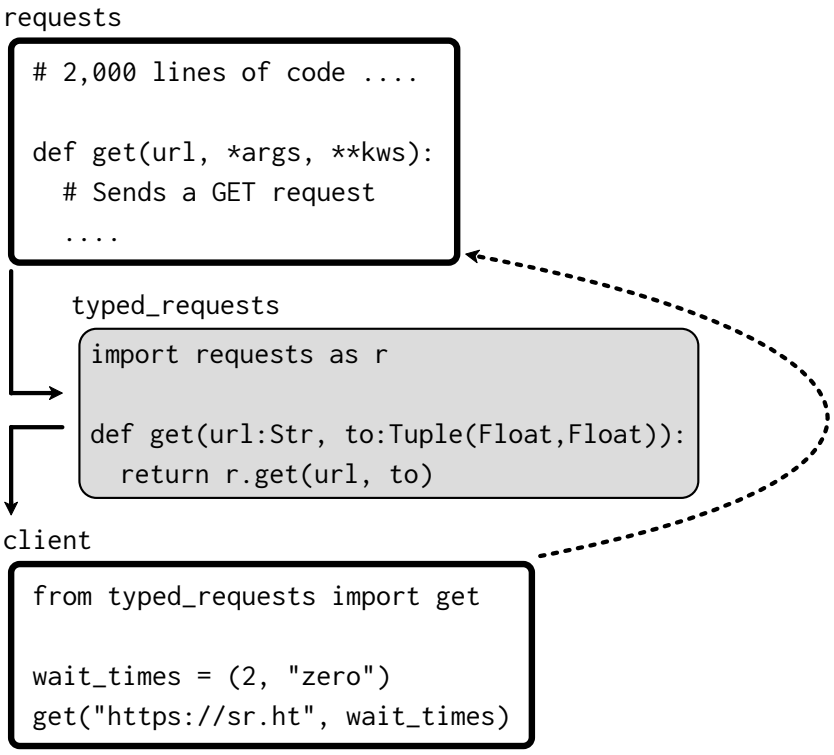


Figure 26: Using Reticulated to define an API

ond is a function that accepts a string, and the third is a function that maps an input port to an HTML representation. Unfortunately for the client, the type declaration in figure 25 is buggy. The library applies the first callback of `call/input-url` to a URL struct, rather than a string as the developer expects.

Fortunately for the developer, Typed Racket compiles types to contracts and thereby catches the mismatch. Here, the compilation of `typed/net/url` generates a contract for `call/input-url`. The generated contract ensures that the untyped client provides three type-matching argument values and that the library applies the callback to a string. When the `net/url` library eventually applies the callback function to a URL structure, the function contract for the callback halts the program. The blame message says that interface for `call/input-url` broke the contract, but warns the developer on the last line with "assuming the contract is correct." A quick look confirms that the contract—that is, the type from which the contract is derived—is wrong.

Figure 26 presents an arrangement of three Transient Reticulated modules, similar to the code in figure 25. The module on the top exports a function that retrieves data from a URL (adapted from the `requests` library, `github.com/psf/requests`). This function accepts several optional and keyword arguments. The typed adaptor module on the right formulates types for one valid use of the function; a client may supply a URL as a string and a timeout as a pair of floats. These types are correct, but the client module on the bottom sends a tuple that contains an integer and a string.

Reticulated's runtime checks ensure that the typed function receives a string and a tuple, but do not validate the tuple's contents. These same arguments then pass to the untyped `get` function in the `requests` module. When the untyped `get` eventually uses the string `"zero"` as a float, Python raises an exception that originates from the `requests` module. A completly untyped version of this program gives the same behavior; the Reticulated types are no help for debugging.

In this example, the programmer is lucky because the call to the typed version of `get` is still visible on the stack trace, providing a hint that this call might be at fault. If Python were to properly implement tail calls, or if the library accessed the pair some time after returning control to the client, this hint would disappear.

In sum, types in Transient Reticulated do not monitor all channels of communication between modules. A value may cross a type boundary without a full check, making it difficult to discover type-value mismatches or pinpoint their source. Reticulated mitigates this problem with a global map from heap addresses to source locations. The analysis in chapter 4.5 demonstrates, however, that this map can result in incorrect blame.

4.3 TOWARDS A FORMAL COMPARISON

The design of a type-enforcement strategy is a multi-faceted problem. A strategy determines many aspects of behavior: whether mismatches between type specifications and value flows are discovered; whether the typed portion of the code is really statically typed, in a conventional sense; what typed APIs mean for untyped client code; and whether an error message can pinpoint which type specification does not match which value. All of these decisions imply consequences for the programmer and the language designer.

The examples in chapter 4.2 show that various languages choose different points in this multi-faceted design space. But, examples can only motivate a systematic analysis; they cannot serve as the basis of such an endeavor. The selection of example programs and their translation across languages require too much insight. Worse, the examples tell us little about the broader implications of each choice; at best they can demonstrate issues.

A systematic analysis needs a suite of formal properties that capture the consequences of design choices for the working developer and language designer. Such properties must apply to a wide (if not the full) spectrum of design options, articulate benefits of type specifications to typed and untyped code alike, and come with proof techniques that scale to complex language features.

The literature on gradual typing suggests few adequate properties. Our analysis therefore brings new properties to the toolbox.

4.3.1 *Comparative Properties in Prior Work*

Type soundness is one formal property that meets the above criteria. A type soundness theorem can be tailored to a range of type systems, such a theorem has meaning for typed and untyped code, and the syntactic proof technique scales to a variety of language features [122]. The use of type soundness in the gradual typing literature, however, does not promote a level comparison. Consider the four example languages from the previous section. Chaudhuri et al. [20] present a model of Flow and prove a conventional type soundness theorem under the assumption that all code is statically-typed. Vitousek et al. [115] prove a type soundness theorem for Reticulated Python; a reader will eventually notice that the theorem talks about the *shape* of values not their types. Muehlboeck and Tate [72] prove a full type soundness theorem for Nom, which implements the concrete approach. Tobin-Hochstadt and Felleisen [103] prove a full type soundness theorem for a prototypical Typed Racket that includes a weak blame property. To summarize, the four advertised type soundness theorems differ in several regards: one focuses on the typed half of the language; a second proves a claim about a loose relationship

between values and types; a third is a truly conventional type soundness theorem; and the last one incorporates a claim about the quality of error messages.

Siek et al. [87] propose the *gradual guarantee* as a test to identify languages that enable smooth transitions between typed and untyped. They and others show that the gradual guarantee holds for relatively simple type languages and syntactic constructs; proving that it generalizes to complex type systems is the subject of active research [55, 74, 108]. The guarantee itself, however, does not tell apart the behaviors in chapter 4.2. Both Reticulated and Nom come with published proofs of the gradual guarantee [72, 115]. Typed Racket states the guarantee as an explicit design goal. Even Flow, thanks to its lack of dynamic checks, satisfies the criteria for a smooth transition.

The KafKa framework is able to distinguish behaviors but lacks a meta-theoretical analysis [21]. The sole theorem in the paper states type soundness for a statically-typed evaluation language. Different behaviors arise, however, from four translations of a mixed-typed surface language into this evaluation language. One can observe the behaviors, but the model does not characterize them.

New et al. [73] distinguish gradual typing systems via *equivalence preservation*. Starting from a set of axioms for typed expressions—for example, β and η equations—they ask whether interactions with untyped code can violate the axioms. Equivalence preservation does define a spectrum; the Natural semantics preserves η for pairs and functions, and a lazy variant (Co-Natural) fails for pairs. But, this spectrum is rather coarse. The Transient and Erasure behaviors are indistinguishable under equivalence preservation because both fail to preserve the axioms. Furthermore, the type-centric nature of the equivalences offers no direct information to the untyped side. Authors of untyped code can at best deduce that the behavior of their programs cannot be affected by certain changes in typed libraries. As a final remark, techniques for proving that the chosen equivalences hold are an active area of research but results so far indicate that they require ingenuity to adapt from one linguistic setting to another.

Another well-studied property is the *blame theorem* [2, 85, 103, 115, 117, 118]. Despite the authoritative name, this property is not the final word on blame. It states that a run-time mismatch may occur only when an untyped value enters a typed, or more-precisely typed, context; a typed value cannot trigger an error by crossing to less-typed code. The property is a useful design principle, but does not distinguish the various semantics in the literature. To its credit, the blame theorem does justify the slogan "well typed programs can't be blamed" for a Natural semantics under the assumption that all boundary types are correct. The slogan does not apply, however, to a semantics such as Transient that lets a value cross a boundary without a complete type check. Nor does it hold for incorrect types that were

retroactively added to an untyped program; refer to figure 25 for one example and Dimoulas et al. [26] for further discussion.

4.3.2 *Our Analysis*

The primary formal property has to be type soundness, because it tells a programmer that evaluation is well-defined in each component of a mixed-typed programs. In addition, the canonical forms lemma that enables a proof of type soundness also enables optimizations by specifying exactly which values can arise in well-typed code.

The second property, *complete monitoring*, asks whether types guard all statically-declared and dynamically-created channels of communication between typed and untyped code. That is, whether every interaction between typed and untyped code is mediated by run-time checks.

When a run-time check discovers a mismatch between a type specification and a flow of values and the run-time system issues an error message, the question arises how informative the message is to a debugging programmer. *Blame soundness* and *blame completeness* ask whether a mixed-typed semantics can identify the responsible parties when a run-time type mismatch occurs. Soundness asks for a subset of the potential culprits; completeness asks for a superset.

Furthermore, the differences among type soundness theorems and the gap between type soundness and complete monitoring suggests the question how many errors an enforcement regime discovers. The answer is an *error preorder* relation, which compares semantics in terms of the run-time mismatches that they discover.

Individually, each property characterizes a particular aspect of a type-enforcement semantics. Together, the properties inform us about the nature of the multi-faceted design space that this semantics problem opens up. And in general, this work should help with the articulation of consequences of design choices for the working developer.

4.4 EVALUATION FRAMEWORK

This section introduces the basic ideas of the evaluation framework; detailed formal definitions are deferred to chapter 4.5. To formulate different type-enforcement strategies on an equal footing, the framework begins with one mixed-typed surface language (chapter 4.4.1) and models strategies as distinct semantics (chapter 4.4.2). The properties listed above support an analysis. Type soundness (chapter 4.4.3) and complete monitoring (chapter 4.4.4) characterize the type mismatches that a semantics detects. Blame soundness and blame completeness (chapter 4.4.5) measure the quality of error messages. The error preorder (chapter 4.4.6) enables direct behavioral comparisons.

4.4.1 *Surface Language*

The surface multi-language combines two independent pieces in the style of Matthews and Findler [65]. Statically-typed expressions constitute one piece; dynamically-typed expressions are the other half. Technically, these expression languages are identified by two judgments: typed expressions e_0 satisfy $\vdash e_0 : \tau_0$ for some type τ_0, and untyped expressions e_1 satisfy $\vdash e_1 : \mathcal{U}$ for the dynamic type. Boundary expressions connect the two languages syntactically and enable run-time interactions.

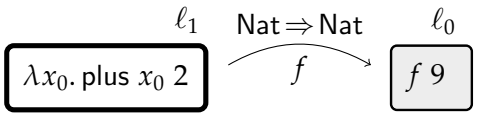
Note that $\mathcal{U}$ is not the flexible dynamic type that is compatible with any static type [86, 101], rather, it is the uni-type that describes all well-formed untyped expressions [65]. Consequently, there is no need for a type precision judgment in the surface language because all mixed-typed interactions occur through boundary expressions. How to add a dynamic type is a separate dimension that is orthogonal to the question of how to enforce types; with or without such a type, our results apply to the language's type-enforcement strategy. Whether the dynamic type is useful is a question for another time [42].

The core statically-typed (v_s) and dynamically-typed (v_d) values are mirror images, and consist of integers, natural numbers, pairs, and functions. This common set of values is the basis for typed-untyped communication. Types τ summarize values:

$$
\begin{aligned}
v_s &= i \mid n \mid \langle v_s, v_s \rangle \mid \lambda(x : \tau).\, e_s \\
v_d &= i \mid n \mid \langle v_d, v_d \rangle \mid \lambda x.\, e_d \\
\tau &= \mathsf{Int} \mid \mathsf{Nat} \mid \tau \Rightarrow \tau \mid \tau \times \tau
\end{aligned}
$$

These value sets are relatively small, but suffice to illustrate the behavior of gradual types for the basic ingredients of a full language. First, the values include atomic data, finite structures, and higher-order values. Second, the natural numbers n are a subset of the integers i to motivate a subtyping judgment for the typed half of the language. Subtyping helps the model distinguish between two type-sound methods of enforcing types (declaration-site vs. use-site) and demonstrates how the model can scale to include true union types, which must be part of any type system for originally-untyped code [18, 105, 107].

Surface expressions include function application, primitive operations, and boundaries. The details of the first two are fairly standard (chapter 4.5.1), but note that function application comes with an explicit app operator (app e_0 e_1). Boundary expressions are the glue that enables mixed-typed programming. A program starts with named chunks of code, called components. Boundary expressions link these chunks together with a static type to describe the types of values that may cross the boundary. Suppose that a typed component named ℓ_0 imports and applies an untyped function from component ℓ_1:

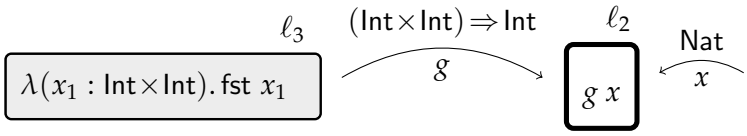


(3)

The surface language can model the composition of these components with a boundary expression that embeds an untyped function in a typed context. The boundary expression is annotated with a *boundary specification* $(\ell_0 \blacktriangleleft \mathsf{Nat} \Rightarrow \mathsf{Nat} \blacktriangleleft \ell_1)$ to explain that component ℓ_0 expects a function from sender ℓ_1:

$$(3) \ = \ \mathsf{app}\ (\mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \Rightarrow \mathsf{Nat} \blacktriangleleft \ell_1)\ (\lambda x_0.\ \mathsf{plus}\ x_0\ 2))\ 9$$

In turn, this two-component expression may be imported into a larger untyped component. The sketch below shows an untyped component in the center that imports two typed components: a new typed function on the left and the expression (3) on the right.



(4)

When linearized to the surface language, this term becomes:

$$\begin{aligned}(4) \ = \ & \mathsf{app}\ (\mathsf{stat}\ (\ell_2 \blacktriangleleft \mathsf{Int} \times \mathsf{Int} \Rightarrow \mathsf{Int} \blacktriangleleft \ell_3)\ (\lambda(x_1 : \mathsf{Int} \times \mathsf{Int}).\ \mathsf{fst}\ x_1)) \\ & (\mathsf{stat}\ (\ell_2 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_0)\ (3))\end{aligned}$$

Technically, a boundary expression combines a boundary specification b and a sender expression. The specification includes the names of the client and sender components, in that order, along with a type to describe values that are intended to cross the boundary. Names, such as ℓ_0, come from some countable set ℓ. The boundary types guide the static type checker, but are mere suggestions unless a semantics decides to enforce them:

$$\begin{aligned} e_s \ &= \ \ldots \mid \mathsf{dyn}\ b\ e_d & \qquad b \ &= \ (\ell \blacktriangleleft \tau \blacktriangleleft \ell) \\ e_d \ &= \ \ldots \mid \mathsf{stat}\ b\ e_s & \qquad \ell \ &= \ \text{countable set of names} \end{aligned}$$

The typing judgments for typed and untyped expressions require a mutual dependence to handle boundary expressions. A well-typed expression may include any well-formed untyped code. Conversely, a well-formed untyped expression may include any typed expression that matches the specified annotation:

$$\frac{\boxed{\Gamma \vdash e : \tau} \qquad \Gamma_0 \vdash e_0 : \mathcal{U}}{\Gamma_0 \vdash \mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ e_0 : \tau_0} \qquad \frac{\boxed{\Gamma \vdash e : \mathcal{U}} \qquad \Gamma_0 \vdash e_0 : \tau_0}{\Gamma_0 \vdash \mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ e_0 : \mathcal{U}}$$

Each surface-language component must have a name, drawn from a set ℓ of labels. These names must be *coherent* according to a judgment that validates an expression relative to a current name and a mapping from variables to names ($\mathcal{L}; \ell \Vdash e$, chapter 4.5.1). All boundary specifications must have a client name that matches the current

name, and variables bound in one component cannot appear free in a different one.

The purpose of the names is to enable a notion of *ownership*, or responsibility. As an expression reduces to a value, ownership determines which components are responsible for the current expression and all subexpressions. Since component names appear in the surface syntax, they can help explain a run-time mismatch in terms of source-code boundaries. Suppose a program halts due to a mismatch between a type and a value. If one component is responsible for the value and the language can find both the client with the type expectation and source of the incompatible value, then a programmer knows exactly where to start debugging.

4.4.2 *Semantic Framework*

The surface language enables the construction of mixed-typed expressions. The next step is to assign behaviors to these programs via formal semantics that differ only in the way they enforce boundary types.

The first ingredient of a semantics is the set of result values v that expressions may reduce to. A result set typically extends the core typed and untyped values mentioned above ($v \supseteq v_s \cup v_d$). Potential reasons for the extended value set include the following:

1. to permit untyped values in typed code, and vice versa;
2. to track the identity of values on a heap;
3. to associate a value with a delayed type-check; and
4. to record the boundaries that a value has previously crossed.

Reasons 3 and 4 introduce two kinds of wrapper value. A guard wrapper, written $\mathbb{G}\, b\ v$, associates a boundary specification with a value to achieve delayed type checks. A trace wrapper, written $\mathbb{T}\, \overline{b}\, v$, attaches a list of boundaries to a value as metadata. Guards are similar to boundary expressions; they separate a context component from a value component. Trace wrappers simply annotate values.

Note that a language with the dynamic type will need a third wrapper for basic values that have been assigned type dynamic. We conjecture that this wrapper is the only change needed to transfer our positive results. Our negative results do not require changes for the dynamic type because such a language can express all our "precisely-typed" counterexample terms.

Second, a semantics must give reduction rules for boundary expressions. These rules initiate a type-enforcement strategy. For example, the Natural semantics (chapter 4.5.5) enforces full types via classic techniques [33, 65]. It admits the following two reductions. Note

a filled triangle (▶) describes a step in untyped code and an open triangle (▷) is for statically-typed code:

$$\text{a} - \quad \mathsf{stat}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ 42 \ \blacktriangleright_{\mathsf{N}}\ 42$$

$$\text{b} - \quad \mathsf{dyn}\ (\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\ (\lambda x_0.\, -8)\ \triangleright_{\mathsf{N}}\ \mathbb{G}\ (\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\ (\lambda x_0.\, -8)$$

The first rule lets a typed number enter an untyped context. The second rule gives typed code access to an untyped function through a newly-created guard wrapper. Guard wrappers are a *higher-order* tool for enforcing higher-order types. As such, wrappers require elimination rules. The Natural semantics includes the following rule to unfold the application of a typed, guarded function into two boundaries:

$$\text{c} - \quad \mathsf{app}\ (\mathbb{G}\ (\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\ (\lambda x_0.\, -8))\ 1\ \triangleright_{\mathsf{N}}\ \mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ (\mathsf{app}\ (\lambda x_0.\, -8)\ (\mathsf{stat}\ (\ell_1 \blacktriangleleft \mathsf{Int} \blacktriangleleft \ell_0)\ 1))$$

Other semantics have different behavior at boundaries and different supporting rules. The Transient semantics (chapter 4.5.8) takes a *first-order* approach to boundaries. Instead of using wrappers, it checks shapes at a boundary and guards elimination forms with shape-check expressions. For example, the following simplified reduction demonstrates a successful check:

$$\text{d} - \quad \mathsf{check}\{(\mathsf{Nat} \times \mathsf{Nat})\}\ \langle -1, -2 \rangle\ \ \blacktriangleright_{\mathsf{T}}\ \ \langle -1, -2 \rangle$$

The triangle is filled gray (▶) because Transient is defined via one notion of reduction that handles both typed and untyped code.

These two points, values and checking rules, are the distinctive aspects of a semantics. Other ingredients can be shared: errors, evaluation contexts, and interpretation of primitive operations. Indeed, chapter 4.5.2 defines three evaluation languages—higher-order, first-order, and erasure—that abstract over the common ingredients.

4.4.3 *Type Soundness*

Type soundness asks whether evaluation is well-defined, and whether a surface-language type predicts aspects of the result. Since there are two kinds of surface expression, soundness has two parts: one for statically-typed code and one for dynamically-typed code.

For typed code, the question is whether code can trust the types of its subexpressions. If an expression with static type τ_0 reduces to a value, the question is what (if anything) the type τ_0 predicts about that value. There are a range of possible answers. At one end, the result value may match the full type τ_0 according to an evaluation-language typing judgment. The other extreme is that the result is a well-formed value of indeterminate shape. In both cases, the programmer knows that typed code cannot reach an undefined state during evaluation.

For untyped code, there is one surface type. If an expression reduces to a value, then uni-type soundness can only guarantee that the result is a well-formed value of indeterminate shape. The practical benefit of such a theorem is that untyped code cannot reach an undefined state through mixed-typed interactions.

Both parts combine into the following rough definition, where the function F and judgment $\vdash_{\mathcal{X}}$ are parameters. The function maps surface types to observations that one can make about a result; varying the choice of F offers a spectrum of soundness for typed code. The judgment $\vdash_{\mathcal{X}}$ matches a value with a description.

DEFINITION SKETCH (f-type soundness)

If e_0 has static type τ_0 ($\vdash e_0 : \tau_0$), then one of the following holds:

- *e_0 reduces to a value v_0 and $\vdash_{\mathcal{X}} v_0 : F(\tau_0)$*
- *e_0 reduces to an allowed error*
- *e_0 reduces endlessly.*

If e_0 is untyped ($\vdash e_0 : \mathcal{U}$), then one of the following holds:

- *e_0 reduces to a value v_0 and $\vdash_{\mathcal{X}} v_0 : \mathcal{U}$*
- *e_0 reduces to an allowed error*
- *e_0 reduces endlessly.*

4.4.4 *Complete Monitoring*

Complete monitoring tests whether a mixed-typed semantics has control over every interaction between typed and untyped code. If the property holds, then a programmer can rely on the language to run checks at the proper points, for example, between the library and client demonstrated in figure 25. Concretely, if a value passes through the type $(\mathsf{Int} \Rightarrow \mathsf{Int})$ then complete monitoring guarantees that the language has control over every input to the function and every result that the function computes, regardless of whether these interactions occur in a typed or untyped context.

Because all such interactions originate at the boundaries between typed and untyped code, a simplistic way to formalize complete monitoring is to ask whether each boundary comes with a full run-time check when possible and an error otherwise. A language that meets this strict requirement certainly has full control. However, other good designs fail. Suppose typed code expects a pair of integers and a semantics initially admits any pair at the boundary but eventually checks that the pair contains integers. Despite the incomplete check at the boundary, this delayed-checking semantics eventually performs all necessary checks and should satisfy a complete monitoring theorem. Higher-order values raise a similar question because a single run-time check cannot prove that a function value always behaves a certain way. Nevertheless, a language that checks every call and return is in full control of the function's interactions.

Our definition of complete monitoring translates these ideas about interactions and control into statements about *ownership labels* [24]. At the start of an evaluation, no interactions have occurred yet and every expression has one owner: the enclosing component. The reduction of a boundary term is the semantics of an interaction in which a value flows from one sender component to a client. At this point, the sender loses full control over the value. If the value fully matches the type expectations of the client, then the loss of control is no problem and the client gains full ownership. Otherwise, the sender and client may have to assume joint ownership of the value, depending on the nature of the reduction relation. If a semantics can create a value with multiple owners, then it admits that a component may lose full control over its interactions with other components.

Technically, an ownership label $^{\ell_0}$ names one source-code component. Expressions and values come with at least one ownership label; for example, $(42)^{\ell_0}$ is an integer with one owner and $(((42)^{\ell_0})^{\ell_1})^{\ell_2}$ is an integer with three owners, written $((42))^{\ell_0\ell_1\ell_2}$ for short. A complete monitoring theorem requires two ingredients that manage these labels. First, a reduction relation $\rightarrow^*_{\mathbf{r}}$ must propagate ownership labels to reflect interactions and checks. Second, a single-ownership judgment $\Vdash$ must test whether every value in an expression has a unique owner. To satisfy complete monitoring, reduction must preserve single-ownership.

The key single-ownership rules deal with labeled expressions and boundary terms:

$\boxed{\mathcal{L}; \ell \Vdash e}$

$$\frac{\mathcal{L}_0; \ell_0 \Vdash e_0}{\mathcal{L}_0; \ell_0 \Vdash (e_0)^{\ell_0}} \qquad \frac{\mathcal{L}_0; \ell_1 \Vdash e_0}{\mathcal{L}_0; \ell_0 \Vdash \mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ e_0}$$

Values such as $((42))^{\ell_0\ell_1}$ represent a communication that slipped past the run-time checking protocol, and therefore fail to satisfy single ownership. **Sneak Preview:** One way that a semantics can transfer a higher-order value without creating a joint-ownership is by providing controlled access through a wrapper. The client owns the wrapper, and the sender retains ownership of the enclosed value.

DEFINITION SKETCH (complete monitoring)
For all $\Vdash e_0$, *any reduction* $e_0 \rightarrow^*_{\mathbf{r}} e_1$ *implies* $\Vdash e_1$.

The definition of complete monitoring is deceptively simple because it assumes a reduction relation that correctly propagates labels. In practice, a language comes with an unlabeled reduction relation, and it is up to a researcher to design a lifted relation that handles labeled terms. Lifting requires insight to correctly transfer labels and to ensure that labels do not change the behavior of programs. If labels do not transfer correctly, then a complete monitoring theorem

becomes meaningless. And if the lifted relation depends on labels to compute a result, then a complete monitoring theorem says nothing about the original reduction relation.

How to lift a reduction relation

The models in chapter 4.5 present six reduction relations for a mixed-typed language. Each relation needs a lifted version to support an attempt at a complete monitoring proof. These lifted reduction relations are deferred to an appendix, but come about semi-automatically through the following informal guidelines, or natural (scientific) laws, for proper labeling.

Each law describes a way that labels may be transferred or dropped during evaluation. To convey the general idea, each law also comes with a brief illustration, namely, an example reduction and a short comment. The example reductions use a hypothetical $\mathbf{r}$ relation over the surface language. Recall that stat and dyn are boundary terms; they link two components, a context and an enclosed expression, via a type. When reading an example, accept the transitions $e \ \mathbf{r} \ e$ as axioms and focus on how the labels change in response.

1. If a base value reaches a boundary with a matching base type, then the value must drop its current labels as it crosses the boundary.

$$(\mathsf{stat}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ ((0))^{\ell_2\ell_1})^{\ell_0} \ \mathbf{r} \ (0)^{\ell_0}$$

The value 0 *fully matches the type* Nat.

2. Any other value that crosses a boundary must acquire the label of the new context.

$$(\mathsf{stat}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ (\langle -2, 1\rangle)^{\ell_1})^{\ell_0} \ \mathbf{r} \ ((\langle -2, 1\rangle))^{\ell_1\ell_0}$$

The pair $\langle -2, 1\rangle$ *does not match the type* Nat.

3. Every value that flows out of a value v_0 acquires the labels of v_0 and the context.

$$(\mathsf{snd}\ ((\langle (1)^{\ell_0}, (2)^{\ell_1}\rangle))^{\ell_2\ell_3})^{\ell_4} \ \mathbf{r} \ ((2))^{\ell_1\ell_2\ell_3\ell_4}$$

The value 2 *flows out of the pair* $\langle 1, 2\rangle$.

4. Every value that flows into a function v_0 acquires the label of the context and the reversed labels of v_0.

$$(\mathsf{app}\ ((\lambda x_0.\, \mathsf{fst}\ x_0))^{\ell_0\ell_1}\ (\langle 8, 6\rangle)^{\ell_2})^{\ell_3} \ \mathbf{r}$$
$$(((\mathsf{fst}\ ((\langle 8, 6\rangle))^{\ell_2\ell_3\ell_1\ell_0}))^{\ell_0\ell_1})^{\ell_3}$$

The argument value $\langle 8, 6\rangle$ *is input to the function.*
The substituted body flows out of the function, and
by law 3 acquires the function's labels.

5. A primitive operation (δ) may remove labels on incoming base values.

$$(\mathsf{plus}\ (2)^{\ell_0}\ (3)^{\ell_1})^{\ell_2}\ \ \mathbf{r}\ \ (5)^{\ell_2}$$
$$\textit{Assuming}\ \delta(\mathsf{plus}, 2, 3) = 5.$$

6. Consecutive equal labels may be dropped.

$$((0))^{\ell_0\ell_0\ell_1\ell_0} = ((0))^{\ell_0\ell_1\ell_0}$$

7. Labels on an error term may be dropped.

$$(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Int} \blacktriangleleft \ell_1)\ (\mathsf{plus}\ 9\ (\mathsf{DivErr})^{\ell_1}))^{\ell_0}\ \ \mathbf{r}\ \ \mathsf{DivErr}$$

Note: law 4 talks about functions, but generalizes to reference cells and other values that accept input.

To show how these laws generate a lifted reduction relation, the following rules lift the examples from chapter 4.4.2. Each rule accepts input with any sequence of labels ($\overline{\ell}$), pattern-matches the important ones, and shuffles via the guidelines. The first rule (a′) demonstrates a base-type boundary (law 1). The second (b′) demonstrates a higher-order boundary (law 2); the new guard on the right-hand side implicitly inherits the context label. The third rule (c′) sends an input (law 4) and creates new application and boundary expressions. The fourth rule (d′) applies law 3 for an output.

a′ – $(\mathsf{stat}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ ((42))^{\overline{\ell}_2})^{\ell_3} \blacktriangleright_{\mathsf{N}} (42)^{\ell_3}$

b′ – $(\mathsf{dyn}\ (\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\ ((\lambda x_0.\ ((-8))^{\overline{\ell}_2}))^{\overline{\ell}_3})^{\ell_4} \rhd_{\mathsf{N}}$
$(\mathbb{G}\ (\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\ ((\lambda x_0.\ ((-8))^{\overline{\ell}_2}))^{\overline{\ell}_3})^{\ell_4}$

c′ – $(\mathsf{app}\ ((\mathbb{G}\ (\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\ (v_0)^{\ell_2}))^{\overline{\ell}_3}\ ((1))^{\overline{\ell}_4})^{\ell_5} \rhd_{\mathsf{N}}$
$(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ (\mathsf{app}\ v_0\ (\mathsf{stat}\ (\ell_1 \blacktriangleleft \mathsf{Int} \blacktriangleleft \ell_0)\ ((1))^{\overline{\ell}_4 \ell_5 rev(\overline{\ell}_3)}))^{\ell_2})^{\ell_5}$

d′ – $(\mathsf{check}\{(\mathsf{Nat} \times \mathsf{Nat})\}\ ((\langle((-1))^{\overline{\ell}_0}, ((-2))^{\overline{\ell}_1}\rangle))^{\overline{\ell}_2})^{\ell_3} \rhd_{\mathsf{T}}$
$((\langle((-1))^{\overline{\ell}_0}, ((-2))^{\overline{\ell}_1}\rangle))^{\overline{\ell}_2 \ell_3}$

Although the design of a lifted reduction relation is a challenge for every language, the laws in this section bring across the intuition behind prior formalizations of complete monitoring [24, 25, 71, 97] and should help guide future work.

4.4.5 *Blame Soundness, Blame Completeness*

Blame soundness and blame completeness ask whether a semantics can identify the responsible parties in the event of a run-time mismatch. A type mismatch occurs when a typed context receives an unexpected value. The value may be the result of a boundary expression

or an elimination form, and the underlying issue may lie with either the value, the current type expectation, or some prior communication. In any event, a programmer needs to know which components previously handled the value to begin debugging. A semantics offers information by blaming a set of boundaries (b^*); the meta-question is whether those boundaries have any connection to the value at hand.

Suppose that a reduction halts on the value v_0 and blames the set b_0^* of boundaries. Ideally, the names in these boundaries should list exactly the components that have handled this value. Ownership labels let us state the question precisely. The lifted variant of the same reduction provides an independent specification of the responsible components; namely, the owners that get attached to v_0 as it crosses boundaries. Relative to this source-of-truth, blame soundness asks whether the names in b_0^* are a subset of the true owners. Blame completeness asks for a superset of the true owners.

A semantics can trivially satisfy blame soundness alone by reporting an empty set of boundaries. Conversely, the trivial way to achieve blame completeness is to blame every boundary for every possible mismatch. The real challenge is to satisfy both or implement a pragmatic tradeoff.

DEFINITION SKETCH (blame soundness)
For all reductions that end in a mismatch for value v_0 blaming boundaries b_0^*, the names in b_0^* are a **subset** *of the labels on v_0.*

DEFINITION SKETCH (blame completeness)
For all reductions that end in a mismatch for value v_0 blaming boundaries b_0^*, the names in b_0^* are a **superset** *of the labels on v_0.*

The propagation laws above (chapter 4.4.4) specify one way to manage ownership labels. But other ground-truth strategies are possible, and may provide insights about semantics that fail to be blame-sound and blame-complete with the standard labeling. As a case in point, the Transient semantics (chapter 4.5.8) uses heap addresses to allow mixed-typed interaction without wrapper expressions. The evaluation of a function, for example, draws a fresh heap address p_0 and stores the function on a value heap ($\mathcal{H}$).

$$(\lambda x_0.\, x_0); \mathcal{H}_0; \mathcal{B}_0 \;\rhd_{\mathsf{T}}\; \mathsf{p}_0; (\{\mathsf{p}_0 \mapsto (\lambda x_0.\, x_0)\} \cup \mathcal{H}_0); (\{\mathsf{p}_0 \mapsto \emptyset\} \cup \mathcal{B}_0)$$
where p_0 fresh in $\mathcal{H}_0$ and $\mathcal{B}_0$

When the pointer p_0 crosses a boundary, the semantics records the crossing on a blame heap ($\mathcal{B}$). The blame heap provides a set of boundaries if a type mismatch occurs, but this set is typically unsound because it conflates different pointers to the same value. Propagating labels onto the heap, however, enables a conjecture that Transient blames only boundaries that are relevant to the address of the incompatible value.

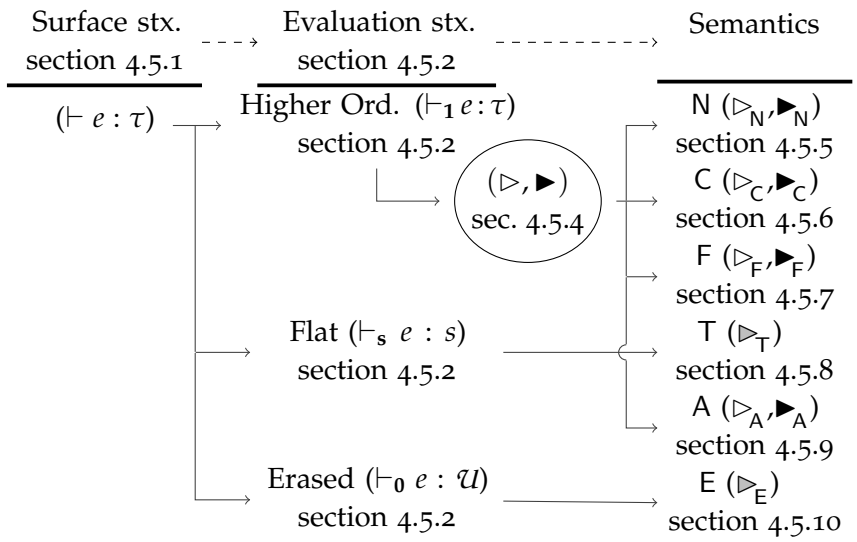


Figure 27: Map of basic definitions in chapter 4.5

4.4.6 *Error Preorder*

Whereas the preceding properties characterize the semantics independently of each other, an *error preorder relation* allows direct comparisons. Strategies that perform many eager run-time checks have a lower position in the order.

One semantics lies below another in this preorder, written $X \lesssim Y$, if it raises errors on at least as many well-formed input expressions. Put another way, $X \lesssim Y$ if and only if the latter reduces at least as many expressions to a result value. Note that semantics do not need to raise the same error when they both agree that a program is faulty. When two semantics agree about which expressions raise run-time errors, the notation $X \approx Y$ shows that they lie below one another.

DEFINITION SKETCH (error preorder $\lesssim$)
$X \lesssim Y$ *iff* $\{e_0 \mid \exists\, v_0.\ e_0 \to^*_{\mathsf{X}} v_0\} \subseteq \{e_1 \mid \exists\, v_1.\ e_1 \to^*_{\mathsf{Y}} v_1\}$.

DEFINITION SKETCH (error equivalence $\approx$)
$X \approx Y$ *iff* $X \lesssim Y$ *and* $Y \lesssim X$.

4.5 TECHNICAL DEVELOPMENT

This section presents the main technical details of our analysis: the model, the six semantics, and the properties that each semantics satisfies. Because this is a long and intricate section, figure 27 gives an outline. The discussion begins with one surface syntax (chap-

ter 4.5.1) and proceeds with three target languages that can run surface programs (chapter 4.5.2). Each target comes with a target type system; type soundness relates surface types to target types. Chapter 4.5.4 presents notions of reduction that are shared among several languages. The final sections state the six base semantics and assess their formal properties.

Several properties depend on a lifted semantics that propagates ownership labels in accordance with the guidelines from chapter 4.4.4. This means that the map in figure 27 is only half of the formal development; each syntax and semantics has a parallel, lifted version. Chapter 4.5.1 presents the lifted surface syntax, but other sections give only the most important details regarding ownership. Full definitions appear in the appendix.

4.5.1 *Surface Syntax, Types, and Ownership*

Figure 28 presents the syntax and typing judgments for the common syntax sketched in chapter 4.4.1. Expressions e include variables, integers, pairs, functions, primitive operations, applications, and boundary expressions. The primitive operations consist of pair projections and arithmetic functions, to model interactions with a runtime system. A dyn boundary expression embeds a dynamically-typed expression into a statically-typed context, and a stat boundary expression embeds a typed expression in an untyped context.

A type specification $\tau/_{\mathcal{U}}$ is either a static type τ or the symbol $\mathcal{U}$ for untyped code. Fine-grained mixtures of τ and $\mathcal{U}$, such as $\mathsf{Int} \times \mathcal{U}$, are not permitted; the model describes two parallel syntaxes that are connected through boundary expressions (chapter 4.4.1). A statically-typed expression e_0 is one where the judgment $\Gamma_0 \vdash e_0 : \tau_0$ holds for some type environment and type. This judgment depends on a standard notion of subtyping ($\leqslant:$) that is based on the relation $\mathsf{Nat} \leqslant: \mathsf{Int}$, is covariant for pairs and function codomains, and is contravariant for function domains. The metafunction Δ determines the output type of a primitive operation. For example the sum of two natural numbers is a natural ($\Delta(\mathsf{plus}, \mathsf{Nat}, \mathsf{Nat}) = \mathsf{Nat}$) but the sum of two integers returns an integer. A dynamically-typed expression e_1 is one for which $\Gamma_1 \vdash e_1 : \mathcal{U}$ holds for some environment.

Every function application and operator application comes with a type specification $\tau/_{\mathcal{U}}$ for the expected result. These annotations serve two purposes: to determine the behavior of the Transient and Amnesic semantics, and to tell apart statically-typed and dynamically-typed redexes. An implementation could easily infer valid annotations from well-typed subexpressions. The model keeps them explicit to easily formulate examples where subtyping affects behavior; for instance, the source-language terms $\mathit{unop}\{\mathsf{Nat}\}\, e_0$ and $\mathit{unop}\{\mathsf{Int}\}\, e_0$ may lead to different run-time checks.

Surface Language

$$
\begin{array}{rcl@{\qquad}rcl}
e & = & x \mid i \mid n \mid \langle e, e\rangle \mid \lambda x.\, e \mid & b & = & (\ell \blacktriangleleft \tau \blacktriangleleft \ell) \\
 & & \lambda(x:\tau).\, e \mid & b^* & = & \mathcal{P}(b) \\
 & & \mathsf{app}\{{}^{\tau}/_{\mathcal{U}}\}\, e\ e \mid & \ell & = & \text{countable set of names} \\
 & & \mathit{unop}\{{}^{\tau}/_{\mathcal{U}}\}\, e \mid & \overline{\ell} & = & \text{sequences of names} \\
 & & \mathit{binop}\{{}^{\tau}/_{\mathcal{U}}\}\, e\, e \mid & \Gamma & = & \cdot \mid (x : {}^{\tau}/_{\mathcal{U}}), \Gamma \\
 & & \mathsf{dyn}\ b\ e \mid \mathsf{stat}\ b\ e & i & = & \mathbb{Z} \\
\tau & = & \mathsf{Int} \mid \mathsf{Nat} \mid \tau \Rightarrow \tau \mid \tau \times \tau & n & = & \mathbb{N} \\
{}^{\tau}/_{\mathcal{U}} & = & \tau \mid \mathcal{U} & \mathit{unop} & = & \mathsf{fst} \mid \mathsf{snd} \\
 & & & \mathit{binop} & = & \mathsf{plus} \mid \mathsf{quotient}
\end{array}
$$

$\boxed{\Gamma \vdash e : \tau}$ (selected rules)

$$\frac{(x_0 : \tau_0) \in \Gamma_0}{\Gamma_0 \vdash x_0 : \tau_0} \qquad \frac{(x_0 : \tau_0), \Gamma_0 \vdash e_0 : \tau_1}{\Gamma_0 \vdash \lambda(x_0 : \tau_0).\, e_0 : \tau_0 \Rightarrow \tau_1}$$

$$\frac{\Gamma_0 \vdash e_0 : \tau_1 \qquad \Delta(\mathit{unop}, \tau_1) \leqslant: \tau_0}{\Gamma_0 \vdash \mathit{unop}\{\tau_0\}\, e_0 : \tau_0} \qquad \frac{\Gamma_0 \vdash e_0 : \tau_1 \Rightarrow \tau_2 \qquad \Gamma_0 \vdash e_1 : \tau_1 \qquad \tau_2 \leqslant: \tau_0}{\Gamma_0 \vdash \mathsf{app}\{\tau_0\}\, e_0\ e_1 : \tau_0}$$

$$\frac{\Gamma_0 \vdash e_0 : \mathcal{U}}{\Gamma_0 \vdash \mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ e_0 : \tau_0}$$

$\boxed{\Gamma \vdash e : \mathcal{U}}$ (selected rules)

$$\frac{(x_0 : \mathcal{U}) \in \Gamma_0}{\Gamma_0 \vdash x_0 : \mathcal{U}} \qquad \frac{(x_0 : \mathcal{U}), \Gamma_0 \vdash e_0 : \mathcal{U}}{\Gamma_0 \vdash \lambda x_0.\, e_0 : \mathcal{U}}$$

$$\frac{\Gamma_0 \vdash e_0 : \tau_0}{\Gamma_0 \vdash \mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ e_0 : \mathcal{U}}$$

Figure 28: Surface syntax and typing rules

Ownership Syntax

$$
\begin{array}{rcl}
e & = & x \mid i \mid n \mid \langle e, e \rangle \mid \lambda x.\, e \mid \lambda(x : \tau).\, e \mid \mathsf{app}\{{}^{\tau}/_{\mathcal{U}}\}\, e\, e \mid \mathit{unop}\{{}^{\tau}/_{\mathcal{U}}\}\, e \mid \\
 & & \mathit{binop}\{{}^{\tau}/_{\mathcal{U}}\}\, e\, e \mid \mathsf{dyn}\ b\ (e)^{\ell} \mid \mathsf{stat}\ b\ (e)^{\ell} \mid (e)^{\ell} \\
\ell & = & \text{countable set} \\
\mathcal{L} & = & \cdot \mid (x : \ell), \mathcal{L}
\end{array}
$$

$e : {}^{\tau}/_{\mathcal{U}}$ **wf**

$(e_0)^{\ell_0} : \tau_0$ **wf**
  if $\ell_0 \Vdash (e_0)^{\ell_0}$ and $\cdot \vdash e_0 : \tau_0$

$(e_0)^{\ell_0} : \mathcal{U}$ **wf**
  if $\ell_0 \Vdash (e_0)^{\ell_0}$ and $\cdot \vdash e_0 : \mathcal{U}$

$\mathcal{L}; \ell \Vdash e$

$$\frac{(x_0 : \ell_0) \in \mathcal{L}_0}{\mathcal{L}_0; \ell_0 \Vdash x_0} \qquad \frac{}{\mathcal{L}_0; \ell_0 \Vdash i_0} \qquad \frac{(x_0 : \ell_0), \mathcal{L}_0; \ell_0 \Vdash e_0}{\mathcal{L}_0; \ell_0 \Vdash \lambda x_0.\, e_0}$$

$$\frac{(x_0 : \ell_0), \mathcal{L}_0; \ell_0 \Vdash e_0}{\mathcal{L}_0; \ell_0 \Vdash \lambda(x_0 : \tau_0).\, e_0} \qquad \frac{\mathcal{L}_0; \ell_0 \Vdash e_0 \qquad \mathcal{L}_0; \ell_0 \Vdash e_1}{\mathcal{L}_0; \ell_0 \Vdash \langle e_0, e_1 \rangle}$$

$$\frac{\mathcal{L}_0; \ell_0 \Vdash e_0}{\mathcal{L}_0; \ell_0 \Vdash \mathit{unop}\{{}^{\tau}/_{\mathcal{U}}\}\, e_0} \qquad \frac{\mathcal{L}_0; \ell_0 \Vdash e_0 \qquad \mathcal{L}_0; \ell_0 \Vdash e_1}{\mathcal{L}_0; \ell_0 \Vdash \mathit{binop}\{{}^{\tau}/_{\mathcal{U}}\}\, e_0\, e_1}$$

$$\frac{\mathcal{L}_0; \ell_0 \Vdash e_0 \qquad \mathcal{L}_0; \ell_0 \Vdash e_1}{\mathcal{L}_0; \ell_0 \Vdash \mathsf{app}\{{}^{\tau}/_{\mathcal{U}}\}\, e_0\, e_1} \qquad \frac{\mathcal{L}_0; \ell_1 \Vdash e_0}{\mathcal{L}_0; \ell_0 \Vdash \mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ e_0}$$

$$\frac{\mathcal{L}_0; \ell_1 \Vdash e_0}{\mathcal{L}_0; \ell_0 \Vdash \mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ e_0} \qquad \frac{\mathcal{L}_0; \ell_0 \Vdash e_0}{\mathcal{L}_0; \ell_0 \Vdash (e_0)^{\ell_0}}$$

Figure 29: Ownership syntax and single-owner consistency

Figure 29 extends the surface syntax with ownership labels and introduces a single-owner ownership consistency relation. These labels record the component from which an expression originates. The extended syntax brings one addition, labeled expressions $(e)^{\ell}$, and a requirement that boundary expressions label their inner component. The single-owner consistency judgment $(\mathcal{L}; \ell \Vdash e)$ ensures that every subterm of an expression has a unique owner. This judgment is parameterized by a mapping from variables to labels ($\mathcal{L}$) and a context label (ℓ). Every variable reference must occur in a context that matches the variable's map entry, every labeled expression must match the context, and every boundary expressions must have a client name that matches the context label. For example, the expression $(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ (x_0)^{\ell_1})^{\ell_0}$ is consistent under a mapping that contains $(x_0 : \ell_1)$ and the ℓ_0 context label. The expression $((42)^{\ell_0})^{\ell_1}$, also written $((42))^{\ell_0\ell_1}$ (figure 31), is inconsistent for any parameters.

Labels correspond to component names but come from a distinct set. Thus the expression $(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ (x_0)^{\ell_1})$ contains two names, ℓ_0 and ℓ_1, and one label $^{\ell_1}$ that matches the inner component name. The distinction separates an implementation from a specification. A semantics, or implementation, manipulates component names to explain errors. Labels serve as a specification to assess whether a semantics uses component names in a sensible way. If the two could mix, then the specification would be a biased measure.

Lastly, a surface expression is well-formed ($e : \tau/_{\mathcal{U}}$ **wf**) if it satisfies a typing judgment—either static or dynamic—and single-owner consistency under some labeling and context label ℓ_0. The theorems below all require well-formed expressions.

4.5.2 *Three Evaluation Syntaxes*

Each semantics requires a unique evaluation syntax, but overlaps among these six languages motivate three common definitions. A *higher-order* evaluation syntax supports type-enforcement strategies that require wrappers. A *flat* syntax, with simple checks rather than wrappers, supports Transient. And an *erased* syntax supports the compilation of typed and untyped code to a common untyped host.

Figure 30 defines common aspects of the evaluation syntaxs. These include errors Err, shapes (or, constructors) s, evaluation contexts, and evaluation metafunctions.

A program evaluation may signal four kinds of errors:

1. A dynamic tag error (TagErr) occurs when an untyped redex applies an elimination form to a mis-shaped input. For example, the first projection of an integer signals a tag error.

2. An invariant error (InvariantErr) occurs when the shape of a typed redex contradicts static typing; a "tag error" in typed

Evaluation Language extends Surface Language

$$
\begin{array}{lcl}
\mathsf{Err} & = & \mathsf{TagErr} \mid \mathsf{InvariantErr} \mid \mathsf{DivErr} \mid \mathsf{BndryErr}\,(b^*, v) \\
e & = & \ldots \mid \mathsf{Err} \\
s & = & \mathsf{Int} \mid \mathsf{Nat} \mid \mathsf{Pair} \mid \mathsf{Fun} \\
E & = & [\,] \mid \mathsf{app}\{\tau/_{\mathcal{U}}\}\,E\,e \mid \mathsf{app}\{\tau/_{\mathcal{U}}\}\,v\,E \mid \langle E, e\rangle \mid \langle v, E\rangle \mid \mathit{unop}\{\tau/_{\mathcal{U}}\}E \mid \\
 & & \mathit{binop}\{\tau/_{\mathcal{U}}\}\,E\,v \mid \mathit{binop}\{\tau/_{\mathcal{U}}\}\,v\,E \mid \mathsf{dyn}\,b\;E \mid \mathsf{stat}\,b\;E
\end{array}
$$

$$
\lfloor \tau_0 \rfloor = \begin{cases}
\mathsf{Nat} & \text{if } \tau_0 = \mathsf{Nat} \\
\mathsf{Int} & \text{if } \tau_0 = \mathsf{Int} \\
\mathsf{Pair} & \text{if } \tau_0 \in \tau \times \tau \\
\mathsf{Fun} & \text{if } \tau_0 \in \tau \Rightarrow \tau
\end{cases}
$$

$$
\mathit{shape\text{-}match}\,(s_0, v_0) = \begin{cases}
\mathsf{True} \\
\quad \text{if } s_0 = \mathsf{Nat} \text{ and } v_0 \in n \\
\quad \text{or } s_0 = \mathsf{Int} \text{ and } v_0 \in i \\
\quad \text{or } s_0 = \mathsf{Pair} \text{ and} \\
\qquad v_0 \in \langle v, v\rangle \cup \\
\qquad\quad (\mathbb{G}\ (\ell \blacktriangleleft (\tau \times \tau) \blacktriangleleft \ell)\ v) \\
\quad \text{or } s_0 = \mathsf{Fun} \text{ and} \\
\qquad v_0 \in (\lambda x.\, e) \cup (\lambda (x : \tau).\, e) \cup \\
\qquad\quad (\mathbb{G}\ (\ell \blacktriangleleft (\tau \Rightarrow \tau) \blacktriangleleft \ell)\ v) \\
\mathit{shape\text{-}match}\,(s_0, v_1) \\
\quad \text{if } v_0 = \mathbb{T}\, b_0^*\, v_1 \\
\mathsf{False} \\
\quad \text{otherwise}
\end{cases}
$$

$$
\delta(\mathit{unop}, \langle v_0, v_1\rangle) = \begin{cases}
v_0 & \text{if } \mathit{unop} = \mathsf{fst}\{\tau/_{\mathcal{U}}\} \\
v_1 & \text{if } \mathit{unop} = \mathsf{snd}\{\tau/_{\mathcal{U}}\}
\end{cases}
$$

$$
\delta(\mathit{binop}, i_0, i_1) = \begin{cases}
i_0 + i_1 \\
\quad \text{if } \mathit{binop} = \mathsf{plus}\{\tau/_{\mathcal{U}}\} \\
\mathsf{DivErr} \\
\quad \text{if } \mathit{binop} = \mathsf{quotient}\{\tau/_{\mathcal{U}}\} \\
\quad \text{and } i_1 = 0 \\
\lfloor i_0 / i_1 \rfloor \\
\quad \text{if } \mathit{binop} = \mathsf{quotient}\{\tau/_{\mathcal{U}}\} \\
\quad \text{and } i_1 \neq 0
\end{cases}
$$

Figure 30: Common evaluation syntax and metafunctions

$$rev\,(b_0^*) = \{(\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_0) \mid (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1) \in b_0^*\}$$

$$senders\,(b_0^*) = \{\ell_1 \mid (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1) \in b_0^*\}$$

$$rev\,(\ell_0 \cdots \ell_n) = \ell_n \cdots \ell_0$$

$$owners\,(v_0) = \begin{cases} \{\ell_0\} \cup owners\,(v_1) & \text{if } v_0 = (v_1)^{\ell_0} \\ owners\,(v_1) & \text{if } v_0 = \mathbb{T}\ b_0^*\ v_1 \\ \{\} & \text{otherwise} \end{cases}$$

$$((e_0))^{\ell_n \cdots \ell_1} = e_1 \quad \Longleftrightarrow \quad e_1 = (\cdots (e_0)^{\ell_n} \cdots)^{\ell_1}$$

Figure 31: Metafunctions for boundaries and labels

code is one way to reach an invariant error. One goal of type soundness is to eliminate such contradictions.

3. A division-by-zero error (DivErr) may be raised by an application of the quotient primitive; a full language will contain many similar primitive errors.

4. A boundary error (BndryErr (b^*, v)) reports a mismatch between two components. One component, the sender, provided the enclosed value. A second component rejected the value. The set of witness boundaries suggests potential sources for the fault; intuitively, this set should include the client–sender boundary. The error BndryErr $(\{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\}, v_0)$, for example, says that a mismatch between value v_0 and type τ_0 prevented the value sent by the ℓ_1 component from entering the ℓ_0 component.

The four shapes, s, correspond both to type constructors and to value constructors. Half of the correpondence is defined by the $\lfloor \cdot \rfloor$ metafunction, which maps a type to a shape. The *shape-match* metafunction is the other half; it checks the top-level shape of a value.

Both metafunctions use an $\cdot \in \cdot$ judgment, which holds if a value is a member of a set. The claim $v_0 \in n$, for example, holds when the value v_0 is a member of the set of natural numbers. By convention, a variable without a subscript typically refers to a set and a term containing a set describes a comprehension. The term $(\lambda x.\, v)$, for instance, describes the set $\{(\lambda x_i.\, v_j) \mid x_i \in x \wedge v_j \in v\}$ of all functions that return some value.

The *shape-match* metafunction also makes reference to two value constructors unique to the higher-order evaluation syntax: guard $(\mathbb{G}\ b\ v)$ and trace $(\mathbb{T}\ b^*\ v)$ wrappers. A guard has a shape determined by the type in its boundary. A trace is metadata, so *shape-match* looks

$\boxed{\text{Higher-Order Evaluation Syntax}}$ extends Evaluation Language

$e \;=\; \ldots \mid \mathsf{trace}\, b^*\, e$

$v \;=\; i \mid n \mid \langle v, v\rangle \mid \lambda x.\, e \mid \lambda(x:\tau).\, e \mid \mathbb{G}\, b\; v \mid \mathbb{T}\, b^*\, v$

$\boxed{\Gamma \vdash_{\mathbf{1}} e : \tau}$ (selected rules), extends $\Gamma \vdash e : \tau$

$$\frac{\Gamma_0 \vdash_{\mathbf{1}} v_0 : \mathcal{U}}{\Gamma_0 \vdash_{\mathbf{1}} \mathbb{G}\,(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 : \tau_0} \qquad \frac{}{\Gamma_0 \vdash_{\mathbf{1}} \mathsf{Err} : \tau_0}$$

$\boxed{\Gamma \vdash_{\mathbf{1}} e : \mathcal{U}}$ (selected rules), extends $\Gamma \vdash e : \mathcal{U}$

$$\frac{\Gamma_0 \vdash_{\mathbf{1}} v_0 : \tau_0}{\Gamma_0 \vdash_{\mathbf{1}} \mathbb{G}\,(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 : \mathcal{U}} \qquad \frac{\Gamma_0 \vdash_{\mathbf{1}} v_0 : \mathcal{U}}{\Gamma_0 \vdash_{\mathbf{1}} \mathbb{T}\, b_0^*\, v_0 : \mathcal{U}} \qquad \frac{}{\Gamma_0 \vdash_{\mathbf{1}} \mathsf{Err} : \mathcal{U}}$$

$\boxed{\mathcal{L}; \ell \Vdash e}$ (selected rules), extends $\mathcal{L}; \ell \Vdash e$

$$\frac{\mathcal{L}_0; \ell_1 \Vdash v_0}{\mathcal{L}_0; \ell_0 \Vdash \mathbb{G}\,(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0} \qquad \frac{\mathcal{L}_0; \ell_0 \Vdash v_0}{\mathcal{L}_0; \ell_0 \Vdash \mathbb{T}\, b_0^*\, v_0}$$

Figure 32: Higher-Order syntax, typing rules, and ownership consistency

past it. Chapter 4.4.2 informally justifies the design, and figure 32 formally introduces these wrapper values.

The final components of figure 30 are the δ metafunctions. These provide a standard, partial specification of the primitive operations.

Figure 31 lists extra metafunctions for boundaries and ownership labels. For boundaries, *rev* flips every client and sender name in a set of specifications. Both Transient and Amnesic reverse boundaries at function calls. If a function \$f\$ crosses one set b_0^* of boundaries, then an input to \$f\$ crosses the reversed boundaries $\mathit{rev}\,(b_0^*)$ before entering the function. The *senders* metafunction extracts the sender names from the right-hand side of every boundary specification in a set. For labels, *rev* reverses a sequence. The *owners* metafunction collects the labels around an unlabeled value stripped of any trace-wrapper metadata. Guard wrappers are not stripped because they represent boundaries. Lastly, the abbreviation $(\!(\cdot)\!)^{\cdot}$ captures a list of boundaries. The term $(\!(4)\!)^{\ell_0 \ell_1}$ is short for $(\!(4)^{\ell_0})^{\ell_1}$ and $(\!(5)\!)^{\overline{\ell}_0}$ matches 5 with $\overline{\ell}_0$ bound to the empty list.

Higher-Order Syntax, Path-Based Ownership Consistency

The higher-order evaluation syntax (figure 32) introduces the two wrapper values described in chapter 4.4.2. A guard $(\mathbb{G}\,(\ell \blacktriangleleft \tau \blacktriangleleft \ell)\; v)$

represents a boundary between two components. A trace wrapper $(\mathbb{T}\, b^*\, v)$ attaches metadata to a value.

Type-enforcement strategies typically use guard wrappers to constrain the behavior of a value. For example, the upcoming Co-Natural semantics wraps any pair that crosses a boundary with a guard; this wrapper validates the elements of the pair upon future projections. Trace wrappers do not constrain behavior. A traced value simply comes with extra information; namely, a collection of the boundaries that the value has previously crossed.

The higher-order typing judgments, $\Gamma \vdash_{\mathbf{1}} e : {}^{\tau}/_{\mathcal{U}}$, extend the surface typing judgments with rules for wrappers and errors. Guard wrappers may appear in both typed and untyped code; the rules in each case mirror those for boundary expressions. Trace wrappers may only appear in untyped code; this restriction simplifies the Amnesic semantics (figure 41). A traced expression is well-formed iff the enclosed value is well-formed. An error term is well-typed in any context.

Figure 32 also extends the single-owner consistency judgment to handle wrapped values. For a guard wrapper, the outer client name must match the context and the enclosed value must be single-owner consistent with the inner sender name. For a trace wrapper, the inner value must be single-owner consistent relative to the context label.

Flat Syntax

The flat syntax (figure 33) supports wrapper-free gradual typing. A new expression form, $(\mathsf{check}\{{}^{\tau}/_{\mathcal{U}}\}\, e\, \mathsf{p})$, represents a shape check. The intended meaning is that the given type must match the value of the enclosed expression. If not, then the location p may be the source of the fault. Locations are names for the pairs and functions in a program. These names map to pre-values in a heap ($\mathcal{H}$) and, more importantly, to sets of boundaries in a blame map ($\mathcal{B}$). Pairs and functions are second-class pre-values (w) that must be allocated before they may be used.

Three meta-functions define heap operations: $\cdot(\cdot)$, $\cdot[\cdot \mapsto \cdot]$, and $\cdot[\cdot \cup \cdot]$. The first gets an item from a finite map, the second replaces a blame heap entry, and the third extends a blame heap entry. Because maps are sets, set union suffices to add new entries.

The flat typing judgments check the top-level shape (s) of an expression and the well-formedness of any subexpressions. These judgments rely on a store typing ($\mathcal{T}$) to describe heap-allocated values. These types must be consistent with the actual values on the heap, a standard technical device that is spelled out in the appendix. Untyped functions may appear in a typed context and vice-versa; there are no wrappers to enforce a separation. Shape-check expressions are valid in typed and untyped contexts.

First-Order Evaluation Syntax extends Evaluation Language

$$
\begin{array}{lll}
e & = & \ldots \mid \mathsf{p} \mid \mathsf{check}\{\tau/_{\mathcal{U}}\}\, e\, \mathsf{p} \\
v & = & i \mid n \mid \mathsf{p} \\
\mathsf{w} & = & \lambda x.\, e \mid \lambda(x:\tau).\, e \mid \langle v, v\rangle \\
\mathsf{p} & = & \text{countable set of heap locations}
\end{array}
\qquad
\begin{array}{lll}
\mathcal{H} & = & \mathcal{P}((\mathsf{p} \mapsto \mathsf{w})) \\
\mathcal{B} & = & \mathcal{P}((\mathsf{p} \mapsto b^*)) \\
\mathcal{T} & = & \cdot \mid (\mathsf{p} : s), \mathcal{T}
\end{array}
$$

$$
\mathcal{H}_0(v_0) = \begin{cases} \mathsf{w}_0 & \text{if } v_0 \in \mathsf{p} \text{ and } (v_0 \mapsto \mathsf{w}_0) \in \mathcal{H}_0 \\ v_0 & \text{if } v_0 \notin \mathsf{p} \end{cases}
$$

$$
\mathcal{B}_0(v_0) = \begin{cases} b_0^* & \text{if } v_0 \in \mathsf{p} \text{ and } (v_0 \mapsto b_0^*) \in \mathcal{B}_0 \\ \emptyset & \text{otherwise} \end{cases}
$$

$$
\mathcal{B}_0[v_0 \mapsto b_0^*] = \begin{cases} \{v_0 \mapsto b_0^*\} \cup (\mathcal{B}_0 \setminus (v_0 \mapsto b_1^*)) \\ \qquad \text{if } v_0 \in \mathsf{p} \text{ and } (v_0 \mapsto b_1^*) \in \mathcal{B}_0 \\ \mathcal{B}_0 \quad \text{otherwise} \end{cases}
$$

$$
\mathcal{B}_0[v_0 \cup b_0^*] = \mathcal{B}_0[v_0 \mapsto b_0^* \cup \mathcal{B}_0(v_0)]
$$

$\boxed{\mathcal{T}; \Gamma \vdash_{\mathbf{s}} e : s}$ (selected rules)

$$
\frac{(\mathsf{p}_0 : s_0) \in \mathcal{T}_0}{\mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} \mathsf{p}_0 : s_0}
\qquad
\frac{(x_0 : \tau_0) \in \Gamma_0}{\mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} x_0 : \lfloor \tau_0 \rfloor}
\qquad
\frac{\mathcal{T}_0; (x_0 : \mathcal{U}), \Gamma_0 \vdash_{\mathbf{s}} e_0 : \mathcal{U}}{\mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} \lambda x_0.\, e_0 : \mathsf{Fun}}
$$

$$
\frac{\mathcal{T}_0; (x_0 : \tau_0), \Gamma_0 \vdash_{\mathbf{s}} e_0 : s_0}{\mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} \lambda(x_0 : \tau_0).\, e_0 : \mathsf{Fun}}
\qquad
\frac{\mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} e_0 : \mathsf{Fun} \qquad \mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} e_1 : s_0}{\mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} \mathsf{app}\{\tau_0\}\, e_0\, e_1 : \lfloor \tau_0 \rfloor}
$$

$$
\frac{\mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} e_0 : \mathsf{Pair}}{\mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} \mathit{unop}\{\tau_0\}\, e_0 : \lfloor \tau_0 \rfloor}
\qquad
\frac{\mathcal{T}_0; \Gamma_0 \vdash e_0 : \mathcal{U}}{\mathcal{T}_0; \Gamma_0 \vdash \mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ e_0 : \lfloor \tau_0 \rfloor}
$$

$$
\frac{\mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} e_0 : s_0}{\mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} \mathsf{check}\{\tau_0\}\, e_0\, \mathsf{p}_0 : \lfloor \tau_0 \rfloor}
\qquad
\frac{\mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} e_0 : \mathcal{U}}{\mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} \mathsf{check}\{\tau_0\}\, e_0\, \mathsf{p}_0 : \lfloor \tau_0 \rfloor}
$$

$\boxed{\mathcal{T}; \Gamma \vdash_{\mathbf{s}} e : \mathcal{U}}$ (selected rules)

$$
\frac{(\mathsf{p}_0 : s_0) \in \mathcal{T}_0}{\mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} \mathsf{p}_0 : \mathcal{U}}
\qquad
\frac{(x_0 : \mathcal{U}) \in \Gamma_0}{\mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} x_0 : \mathcal{U}}
$$

$$
\frac{\mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} e_0 : \lfloor \tau_0 \rfloor}{\mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} \mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ e_0 : \mathcal{U}}
\qquad
\frac{\mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} e_0 : \mathcal{U}}{\mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} \mathsf{check}\{\mathcal{U}\}\, e_0\, \mathsf{p}_0 : \mathcal{U}}
$$

$$
\frac{\mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} e_0 : s_0}{\mathcal{T}_0; \Gamma_0 \vdash_{\mathbf{s}} \mathsf{check}\{\mathcal{U}\}\, e_0\, \mathsf{p}_0 : \mathcal{U}}
$$

Figure 33: Flat syntax, typing rules, and ownership consistency

$\boxed{\text{Erasure Evaluation Syntax}}$ extends Evaluation Language

$v \;=\; i \mid n \mid \langle v, v\rangle \mid \lambda x.\, e \mid \lambda(x : \tau).\, e$

$\boxed{\Gamma \vdash_{\mathbf{0}} e : \mathcal{U}}$ (selected rules)

$$\frac{(x_0 : {}^{\tau}/_{\mathcal{U}}) \in \Gamma_0}{\Gamma_0 \vdash_{\mathbf{0}} x_0 : \mathcal{U}} \qquad \frac{(x_0 : \mathcal{U}), \Gamma_0 \vdash_{\mathbf{0}} e_0 : \mathcal{U}}{\Gamma_0 \vdash_{\mathbf{0}} \lambda x_0.\, e_0 : \mathcal{U}} \qquad \frac{(x_0 : \tau_0), \Gamma_0 \vdash_{\mathbf{0}} e_0 : \mathcal{U}}{\Gamma_0 \vdash_{\mathbf{0}} \lambda(x_0 : \tau_0).\, e_0 : \mathcal{U}}$$

$$\frac{\Gamma_0 \vdash_{\mathbf{0}} e_0 : \mathcal{U}}{\Gamma_0 \vdash_{\mathbf{0}} \mathsf{dyn}\ b_0\ e_0 : \mathcal{U}} \qquad \frac{\Gamma_0 \vdash_{\mathbf{0}} e_0 : \mathcal{U}}{\Gamma_0 \vdash_{\mathbf{0}} \mathsf{stat}\ b_0\ e_0 : \mathcal{U}}$$

Figure 34: Erased evaluation syntax and typing

Erased Syntax

Figure 34 defines an evaluation syntax for type-erased programs. Expressions include error terms; the typing judgment holds for any expression without free variables. Aside from the type annotations left over from the surface syntax, which could be removed with a translation step, the result is a conventional dynamically-typed language.

4.5.3 *Properties of Interest*

Type soundness guarantees that the evaluation of a well-formed expression (1) cannot end in an invariant error and (2) preserves an evaluation-language image of the surface type. Note that an invariant error captures the classic idea of going wrong [69].

Definition 4.5.1 (*F*-type soundness)**.** *Let F map surface types to evaluation types. A semantics X satisfies* $\mathbf{TS}(F)$ *if for all* $e_0 : {}^{\tau}/_{\mathcal{U}}$ **wf** *one of the following holds:*

- $e_0 \to_{\mathsf{X}}^{*} v_0$ *and* $\vdash_X v_0 : F({}^{\tau}/_{\mathcal{U}})$
- $e_0 \to_{\mathsf{X}}^{*} \{\mathsf{TagErr}, \mathsf{DivErr}\} \cup \mathsf{BndryErr}\,(b^*, v)$
- $e_0 \to_{\mathsf{X}}^{*}$ *diverges.*

Three surface-to-evaluation maps (F) suffice for the evaluation languages: an identity map **1**, a type-shape map **s** that extends the type-to-shape metafunction from figure 30, and a constant map **0**:

$$\mathbf{1}({}^{\tau}/_{\mathcal{U}}) = {}^{\tau}/_{\mathcal{U}} \quad \mathbf{s}({}^{\tau}/_{\mathcal{U}}) = \begin{cases} \mathcal{U} & \text{if } {}^{\tau}/_{\mathcal{U}} = \mathcal{U} \\ \lfloor \tau_0 \rfloor & \text{if } {}^{\tau}/_{\mathcal{U}} = \tau_0 \end{cases} \qquad \mathbf{0}({}^{\tau}/_{\mathcal{U}}) = \mathcal{U}$$

Complete monitoring guarantees that the type on each component boundary monitors all interactions between client and server components. The definition of "all interactions" comes from the path-based

ownership propagation laws (chapter 4.4.4); the labels on a value enumerate all partially-responsible components. Relative to this specification, a reduction that preserves single-owner consistency (figure 29) ensures that a value cannot enter a new component without a full type check.

Definition 4.5.2 (complete monitoring)**.** *A semantics X satisfies* **CM** *if for all $(e_0)^{\ell_0} : \tau/_{\mathcal{U}}$ **wf** and all e_1 such that $e_0 \to^*_{\mathsf{X}} e_1$, the contractum is single-owner consistent: $\ell_0 \Vdash e_1$.*

Blame soundness and *blame completeness* measure the quality of error messages relative to a specification of the components that handled a value during an evaluation. A blame-sound semantics guarantees a subset of the true senders, though it may miss some or even all. A blame-complete semantics guarantees all the true senders, though it may include irrelevant information. A sound and complete semantics reports exactly the components that sent the value across a partially-checked boundary.

The standard definitions for blame soundness and blame completeness rely on the path-based ownership propagation laws from chapter 4.4.4. Relative to these laws, the definitions relate the sender names in a set of boundaries (figure 31) to the true owners of the mismatched value.

Definition 4.5.3 (path-based blame soundness and blame completeness)**.** *For all well-formed e_0 such that $e_0 \to^*_{\mathsf{X}} \mathsf{BndryErr}\,(b_0^*, v_0)$:*

- *X satisfies* **BS** *iff senders $(b_0^*) \subseteq$ owners (v_0)*
- *X satisfies* **BC** *iff senders $(b_0^*) \supseteq$ owners (v_0).*

A second useful specification extends the propagation laws to push the owners for each location (p) onto the value heap ($\mathcal{H}$). Chapter 4.5.8 develops this idea to characterize the blame guarantees of the Transient semantics.

Lastly, the error preorder relation allows direct behavioral comparisons. If X and Y represent two strategies for type enforcement, then $X \lesssim Y$ states that the Y semantics reduces at least as many expressions to a value as the X semantics.

Definition 4.5.4 (error preorder)**.** *$X \lesssim Y$ iff $e_0 \to^*_{\mathsf{Y}} \mathsf{Err}_0$ implies $e_0 \to^*_{\mathsf{X}} \mathsf{Err}_1$ for all well-formed expressions e_0.*

If two semantics lie below one another on the error preorder, then they report type mismatches on the same well-formed expressions.

Definition 4.5.5 (error equivalence)**.** *$X \eqsim Y$ iff $X \lesssim Y$ and $Y \lesssim X$.*

$\boxed{e \rhd e}$

$unop\{\tau_0\}\ v_0 \quad \rhd \mathsf{InvariantErr}$
if $v_0 \notin (\mathbb{G}\ (\ell \blacktriangleleft (\tau \times \tau) \blacktriangleleft \ell)\ v)$
and $\delta(unop, v_0)$ is undefined

$unop\{\tau_0\}\ v_0 \quad \rhd \delta(unop, v_0)$
if $\delta(unop, v_0)$ is defined

$binop\{\tau_0\}\ v_0\ v_1 \rhd \mathsf{InvariantErr}$
if $\delta(binop, v_0, v_1)$ is undefined

$binop\{\tau_0\}\ v_0\ v_1 \rhd \delta(binop, v_0, v_1)$
if $\delta(binop, v_0, v_1)$ is defined

$\mathsf{app}\{\tau_0\}\ v_0\ v_1 \quad \rhd \mathsf{InvariantErr}$
if $v_0 \notin (\lambda(x : \tau).\, e) \cup$
$(\mathbb{G}\ (\ell \blacktriangleleft (\tau \Rightarrow \tau) \blacktriangleleft \ell)\ v)$

$\mathsf{app}\{\tau_0\}\ v_0\ v_1 \quad \rhd e_0[x_0 \leftarrow v_1]$
if $v_0 = (\lambda(x_0 : \tau_1).\, e_0)$

$\boxed{e \blacktriangleright e}$

$unop\{\mathcal{U}\}\ v_0 \quad \blacktriangleright \mathsf{TagErr}$
if $v_0 \notin (\mathbb{G}\ (\ell \blacktriangleleft (\tau \times \tau) \blacktriangleleft \ell)\ v)$
and $\delta(unop, v_0)$ is undefined

$unop\{\mathcal{U}\}\ v_0 \quad \blacktriangleright \delta(unop, v_0)$
if $\delta(unop, v_0)$ is defined

$binop\{\mathcal{U}\}\ v_0\ v_1 \blacktriangleright \mathsf{TagErr}$
if $\delta(binop, v_0, v_1)$ is undefined

$binop\{\mathcal{U}\}\ v_0\ v_1 \blacktriangleright \delta(binop, v_0, v_1)$
if $\delta(binop, v_0, v_1)$ is defined

$\mathsf{app}\{\mathcal{U}\}\ v_0\ v_1 \quad \blacktriangleright \mathsf{TagErr}$
if $v_0 \notin (\lambda x.\, e) \cup$
$(\mathbb{G}\ (\ell \blacktriangleleft (\tau \Rightarrow \tau) \blacktriangleleft \ell)\ v)$

$\mathsf{app}\{\mathcal{U}\}\ v_0\ v_1 \quad \blacktriangleright e_0[x_0 \leftarrow v_1]$
if $v_0 = (\lambda x_0.\, e_0)$

Figure 35: Common notions of reduction for Natural, Co-Natural, Forgetful, and Amnesic

4.5.4 *Common Higher-Order Notions of Reduction*

Four of the semantics build on the higher-order evaluation syntax. In redexes that do not mix typed and untyped values, these semantics share the common behavior specified in figure 35. The rules for typed code ($\rhd$) handle basic elimination forms and raise an invariant error (InvariantErr) for invalid input. Type soundness ensures that such errors do not occur. The rules for untyped code ($\blacktriangleright$) raise a tag error for a malformed redex. Later definitions, for example figure 36, combine relations via set union to build one large relation to accomodate all redexes. The full reduction relation is the reflexive-transitive closure of such a set.

4.5.5 *Natural and its Properties*

Figure 36 presents the values and key reduction rules for the Natural semantics. Conventional reductions handle primitives and unwrapped functions ($\blacktriangleright$ and $\rhd$, figure 35).

A successful Natural reduction yields either an unwrapped value or a guard-wrapped function. Guards arise when a function value reaches a function-type boundary. Thus, the possible wrapped values are drawn from the following two sets:

$$\begin{array}{rcl} v_s & = & \mathbb{G}\ (\ell \blacktriangleleft (\tau \Rightarrow \tau) \blacktriangleleft \ell)\ (\lambda x.\, e) \\ & | & \mathbb{G}\ (\ell \blacktriangleleft (\tau \Rightarrow \tau) \blacktriangleleft \ell)\ v_d \end{array}$$

Natural Syntax extends Higher-Order Evaluation Syntax

$v \;=\; i \mid n \mid \langle v, v\rangle \mid \lambda x.\, e \mid \lambda(x:\tau).\, e \mid \mathbb{G}\,(\ell \blacktriangleleft \tau \Rightarrow \tau \blacktriangleleft \ell)\; v$

$e \rhd_{\mathsf{N}} e$

$\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \Rightarrow \tau_1 \blacktriangleleft \ell_1)\; v_0 \qquad \rhd_{\mathsf{N}} \mathbb{G}\,(\ell_0 \blacktriangleleft \tau_0 \Rightarrow \tau_1 \blacktriangleleft \ell_1)\; v_0$
if *shape-match* $(\lfloor \tau_0 \Rightarrow \tau_1 \rfloor, v_0)$

$\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\; \langle v_0, v_1\rangle \qquad \rhd_{\mathsf{N}} \langle \mathsf{dyn}\; b_0\; v_0, \mathsf{dyn}\; b_1\; v_1\rangle$
if *shape-match* $(\lfloor \tau_0 \times \tau_1 \rfloor, \langle v_0, v_1\rangle)$
where $b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)$ and $b_1 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)$

$\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; i_0 \qquad \rhd_{\mathsf{N}} i_0$
if *shape-match* $(\lfloor \tau_0 \rfloor, i_0)$

$\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \qquad \rhd_{\mathsf{N}} \mathsf{BndryErr}\;(\{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\}, v_0)$
if $\neg$*shape-match* $(\lfloor \tau_0 \rfloor, v_0)$

$\mathsf{app}\{\tau_0\}\;(\mathbb{G}\,(\ell_0 \blacktriangleleft \tau_1 \Rightarrow \tau_2 \blacktriangleleft \ell_1)\; v_0)\; v_1 \rhd_{\mathsf{N}} \mathsf{dyn}\; b_0\; (\mathsf{app}\{\mathcal{U}\}\; v_0\; (\mathsf{stat}\; b_1\; v_1))$
where $b_0 = (\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1)$ and $b_1 = (\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0)$

$e \blacktriangleright_{\mathsf{N}} e$

$\mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \Rightarrow \tau_1 \blacktriangleleft \ell_1)\; v_0 \qquad \blacktriangleright_{\mathsf{N}} \mathbb{G}\,(\ell_0 \blacktriangleleft \tau_0 \Rightarrow \tau_1 \blacktriangleleft \ell_1)\; v_0$
if *shape-match* $(\lfloor \tau_0 \rfloor, v_0)$

$\mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\; \langle v_0, v_1\rangle \qquad \blacktriangleright_{\mathsf{N}} \langle \mathsf{stat}\; b_0\; v_0, \mathsf{stat}\; b_1\; v_1\rangle$
if *shape-match* $(\lfloor \tau_0 \times \tau_1 \rfloor, \langle v_0, v_1\rangle)$
where $b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)$ and $b_1 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)$

$\mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; i_0 \qquad \blacktriangleright_{\mathsf{N}} i_0$
if *shape-match* $(\lfloor \tau_0 \rfloor, i_0)$

$\mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \qquad \blacktriangleright_{\mathsf{N}} \mathsf{InvariantErr}$
if $\neg$*shape-match* $(\lfloor \tau_0 \rfloor, v_0)$

$\mathsf{app}\{\mathcal{U}\}\;(\mathbb{G}\,(\ell_0 \blacktriangleleft \tau_0 \Rightarrow \tau_1 \blacktriangleleft \ell_1)\; v_0)\; v_1 \blacktriangleright_{\mathsf{N}} \mathsf{stat}\; b_0\; (\mathsf{app}\{\tau_1\}\; v_0\; (\mathsf{dyn}\; b_1\; v_1))$
where $b_0 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)$ and $b_1 = (\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_0)$

$e \rightarrow^*_{\mathsf{N}} e \;=\; \rightarrow^*_{\cup\{\rhd_{\mathsf{N}}, \blacktriangleright_{\mathsf{N}}, \blacktriangleright, \rhd\}}$

Figure 36: Natural notions of reduction

$$
\begin{array}{rcl}
v_d & = & \mathbb{G}\ (\ell_{\blacktriangleleft}\ (\tau \Rightarrow \tau)_{\blacktriangleleft}\ \ell)\ (\lambda(x:\tau).e) \\
& | & \mathbb{G}\ (\ell_{\blacktriangleleft}\ (\tau \Rightarrow \tau)_{\blacktriangleleft}\ \ell)\ v_s
\end{array}
$$

The presented reduction rules are those relevant to the Natural strategy for enforcing static types. When a dynamically-typed value reaches a typed context (dyn), Natural checks the shape of the value against the type. If the type and value match, Natural wraps functions and recursively checks the elements of a pair. Otherwise, Natural raises an error at the current boundary. When a wrapped function receives an argument, Natural creates two new boundaries: one to protect the input to the inner, untyped function and one to validate the result.

Reduction in dynamically-typed code ($\blacktriangleright_{\mathsf{N}}$) follows a dual strategy. The rules for stat boundaries wrap functions and recursively protect the contents of pairs. The application of a wrapped function creates boundaries to validate the input to a typed function and to protect the result.

Unsurprisingly, this checking protocol ensures the validity of types in typed code and the well-formedness of expressions in untyped code. The Natural approach furthemore enforces boundary types throughout the execution.

Theorem 4.5.6. *Natural satisfies* **TS(1)**.

Proof Idea. By progress and preservation lemmas for the higher-order typing judgment ($\vdash_{\mathbf{1}}$). For example, if an untyped pair reaches a boundary then a typed step ($\rhd_{\mathsf{N}}$) makes progress to either a new pair or an error. In the former case, the new pair contains two boundary expressions:

$$
\begin{array}{l}
\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\ \langle v_0, v_1 \rangle\ \rhd_{\mathsf{N}} \\
\langle \mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0, \mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)\ v_1 \rangle
\end{array}
$$

The typing rules for pairs and for dyn boundaries validate the type of the result. □

Theorem 4.5.7. *Natural satisfies* **CM**.

Proof Idea. By showing that a lifted variant of the $\rightarrow^*_{\mathsf{N}}$ relation preserves single-owner consistency ($\Vdash$). Full lifted rules for Natural appear in an appendix, but one can derive the rules by applying the guidelines from section 4.4.4. For example, consider the $\blacktriangleright_{\mathsf{N}}$ rule that wraps a function. The lifted version accepts a term with arbitrary ownership labels and propagates these labels to the result:

$$
\begin{array}{l}
(\mathsf{stat}\ (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\ ((v_0))^{\overline{\ell}_2})^{\ell_3}\ \blacktriangleright_{\overline{\mathsf{N}}} \\
\qquad\qquad (\mathbb{G}\ (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\ ((v_0))^{\overline{\ell}_2})^{\ell_3} \\
\quad \text{if } \mathit{shape\text{-}match}\ (\lfloor \tau_0 \Rightarrow \tau_1 \rfloor, v_0)
\end{array}
$$

If the redex satisfies single-owner consistency, then the context label matches the client name ($\ell_3 = \ell_0$) and the labels inside the boundary

match the sender name ($\overline{\ell_2} = \ell_1 \cdots \ell_1$). Under these premises, the result also satisfies single-owner consistency. □

Complete monitoring implies that the Natural semantics detects every mismatch between two components—either immediately, or as soon as a function computes an incorrect result. Consequently, every mismatch is due to a single boundary. Blame soundness and completeness ask whether Natural identifies the culprit.

Lemma 4.5.8. *If e_0 is well-formed and $e_0 \rightarrow^*_{\mathsf{N}} \mathsf{BndryErr}\,(b_0^*, v_0)$, then $\mathit{senders}\,(b_0^*) = \mathit{owners}\,(v_0)$ and furthermore b_0^* contains exactly one boundary specification.*

Proof. The sole Natural rule that detects a mismatch blames a single boundary:

$$\begin{aligned}(e_0)^{\ell_0} &\rightarrow^*_{\mathsf{N}} E[\mathsf{dyn}\ (\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_2)\ v_0] \\ &\rightarrow^*_{\mathsf{N}} \mathsf{BndryErr}\,(\{(\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_2)\}, v_0)\end{aligned}$$

Thus $b_0^* = \{(\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_2)\}$ and $\mathit{senders}\,(b_0^*) = \{\ell_2\}$. This boundary is the correct one to blame only if it matches the true owner of the value; that is, $\mathit{owners}\,(v_0) = \{\ell_2\}$. Complete monitoring guarantees a match via $\ell_0 \Vdash E[\mathsf{dyn}\ (\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_2)\ v_0]$. □

Corollary 4.5.9. *Natural satisfies* **BS** *and* **BC**.

4.5.6 *Co-Natural and its Properties*

Figure 37 presents the Co-Natural strategy. Co-Natural is a lazy variant of the Natural approach. Instead of eagerly validating pairs at a boundary, Co-Natural creates a wrapper to delay element-checks until they are needed.

Relative to Natural, there are two changes in the notions of reduction. First, the rules for a pair value at a pair-type boundary create guards. Second, new projection rules handle guarded pairs; these rules make a new boundary to validate the projected element.

Co-Natural still satisfies both a strong type soundness theorem and complete monitoring. Blame soundness and blame completeness follow from complete monitoring. Nevertheless, Co-Natural and Natural can behave differently.

Theorem 4.5.10. *Co-Natural satisfies* **TS(1)**.

Proof Idea. By progress and preservation lemmas for the higher-order typing judgment ($\vdash_1$). For example, consider the rule that applies a wrapped function in a statically-typed context:

$$\begin{aligned}&\mathsf{app}\{\tau_0\}\ (\mathbb{G}\ (\ell_0 \blacktriangleleft (\tau_1 \Rightarrow \tau_2) \blacktriangleleft \ell_1)\ v_0)\ v_1 \ \rhd_{\mathsf{C}} \\ &\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1)\ (\mathsf{app}\{\mathcal{U}\}\ v_0\ (\mathsf{stat}\ (\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_2)\ v_1))\end{aligned}$$

If the redex is well-typed, then v_1 has type τ_1 and the inner stat boundary is well-typed. Similar reasoning for v_0 shows that the untyped

Co-Natural Syntax extends Higher-Order Evaluation Syntax

$$v \;=\; i \mid n \mid \langle v, v\rangle \mid \lambda x.\, e \mid \lambda(x:\tau).\, e \mid \\ \quad \mathbb{G}\,(\ell \blacktriangleleft \tau \Rightarrow \tau \blacktriangleleft \ell)\; v \mid \mathbb{G}\,(\ell \blacktriangleleft \tau \times \tau \blacktriangleleft \ell)\; v$$

$e \rhd_{\mathsf{C}} e$

$$\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \qquad \rhd_{\mathsf{C}}\; \mathbb{G}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0$$
$$\quad \text{if } \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, v_0) \text{ and } v_0 \in \langle v, v\rangle \cup (\lambda x.\, e) \cup (\mathbb{G}\; b\; v)$$

$$\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; i_0 \qquad \rhd_{\mathsf{C}}\; i_0$$
$$\quad \text{if } \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, i_0)$$

$$\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \qquad \rhd_{\mathsf{C}}\; \mathsf{BndryErr}\;(\{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\}, v_0)$$
$$\quad \text{if } \neg \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, v_0)$$

$$\mathsf{fst}\{\tau_0\}\;(\mathbb{G}\;(\ell_0 \blacktriangleleft \tau_1 \times \tau_2 \blacktriangleleft \ell_1)\; v_0) \qquad \rhd_{\mathsf{C}}\; \mathsf{dyn}\; b_0\; (\mathsf{fst}\{\mathcal{U}\}\; v_0)$$
$$\quad \text{where } b_0 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)$$

$$\mathsf{snd}\{\tau_0\}\;(\mathbb{G}\;(\ell_0 \blacktriangleleft \tau_1 \times \tau_2 \blacktriangleleft \ell_1)\; v_0) \qquad \rhd_{\mathsf{C}}\; \mathsf{dyn}\; b_0\; (\mathsf{snd}\{\mathcal{U}\}\; v_0)$$
$$\quad \text{where } b_0 = (\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1)$$

$$\mathsf{app}\{\tau_0\}\;(\mathbb{G}\;(\ell_0 \blacktriangleleft \tau_1 \Rightarrow \tau_2 \blacktriangleleft \ell_1)\; v_0)\; v_1 \rhd_{\mathsf{C}}\; \mathsf{dyn}\; b_0\; (\mathsf{app}\{\mathcal{U}\}\; v_0\; (\mathsf{stat}\; b_1\; v_1))$$
$$\quad \text{where } b_0 = (\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1) \text{ and } b_1 = (\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0)$$

$e \blacktriangleright_{\mathsf{C}} e$

$$\mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \qquad \blacktriangleright_{\mathsf{C}}\; \mathbb{G}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0$$
$$\quad \text{if } \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, v_0) \text{ and } v_0 \in \langle v, v\rangle \cup (\lambda(x:\tau).\, e) \cup (\mathbb{G}\; b\; v)$$

$$\mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; i_0 \qquad \blacktriangleright_{\mathsf{C}}\; i_0$$
$$\quad \text{if } \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, i_0)$$

$$\mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \qquad \blacktriangleright_{\mathsf{C}}\; \mathsf{InvariantErr}$$
$$\quad \text{if } \neg \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, v_0)$$

$$\mathsf{fst}\{\mathcal{U}\}\;(\mathbb{G}\;(\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\; v_0) \qquad \blacktriangleright_{\mathsf{C}}\; \mathsf{stat}\; b_0\; (\mathsf{fst}\{\tau_0\}\; v_0)$$
$$\quad \text{where } b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)$$

$$\mathsf{snd}\{\mathcal{U}\}\;(\mathbb{G}\;(\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\; v_0) \qquad \blacktriangleright_{\mathsf{C}}\; \mathsf{stat}\; b_0\; (\mathsf{snd}\{\tau_1\}\; v_0)$$
$$\quad \text{where } b_0 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)$$

$$\mathsf{app}\{\mathcal{U}\}\;(\mathbb{G}\;(\ell_0 \blacktriangleleft \tau_0 \Rightarrow \tau_1 \blacktriangleleft \ell_1)\; v_0)\; v_1 \blacktriangleright_{\mathsf{C}}\; \mathsf{stat}\; b_0\; (\mathsf{app}\{\tau_1\}\; v_0\; (\mathsf{dyn}\; b_1\; v_1))$$
$$\quad \text{where } b_0 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1) \text{ and } b_1 = (\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_0)$$

$$e \rightarrow^*_{\mathsf{C}} e \;=\; \rightarrow^*_{\cup\{\rhd_{\mathsf{C}}, \blacktriangleright_{\mathsf{C}}, \blacktriangleright, \rhd\}}$$

Figure 37: Co-Natural notions of reduction

application in the result is well-typed. Thus the dyn boundary has type τ_2 which, by inversion on the redex, is a subtype of τ_0. □

Theorem 4.5.11. *Co-Natural satisfies* **CM**.

Proof Idea. By preservation of single-owner consistency for the lifted $\rightarrow_{\mathsf{C}}^{*}$ relation. Consider the lifted rule that applies a wrapped function:

$$(\mathsf{app}\{\tau_0\}\ ((\mathbb{G}\ (\ell_0 \blacktriangleleft (\tau_1 \Rightarrow \tau_2) \blacktriangleleft \ell_1)\ (v_0)^{\ell_2}))^{\overline{\ell}_3}\ v_1)^{\ell_4} \rhd_{\overline{\mathsf{C}}}$$
$$(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1)\ (\mathsf{app}\{\mathcal{U}\}\ v_0\ (\mathsf{stat}\ (\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0)\ (v_1)^{\ell_4 rev\,(\overline{\ell}_3)}))^{\ell_2})^{\overline{\ell}_3 \ell_4}$$

If the redex satisfies single-owner consistency, then $\ell_0 = \overline{\ell}_3 = \ell_4$ and $\ell_1 = \ell_2$. Hence both sequences of labels in the result contain nothing but the context label ℓ_4. □

Theorem 4.5.12. *Co-Natural satisfies* **BS** *and* **BC**.

Proof Idea. By the same line of reasoning that supports Natural; refer to lemma 4.5.8. □

Theorem 4.5.13. $N \lesssim C$.

Proof Idea. By a stuttering simulation. Natural takes extra steps when a pair reaches a boundary because it immediately checks the contents; Co-Natural creates a guard wrapper. Co-Natural takes additional steps when eliminating a wrapped pair. The appendix defines the simulation relation.

The pair wrappers in Co-Natural imply $C \not\lesssim N$. Consider a typed expression that imports an untyped pair with an ill-typed first element.

$$\mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \times \mathsf{Nat} \blacktriangleleft \ell_1)\ \langle -2, 2 \rangle$$

Natural detects the mismatch at the boundary, but Co-Natural will only raise an error if the first element is accessed. □

4.5.7 *Forgetful and its Properties*

The Forgetful semantics (figure 38) creates wrappers to enforce pair and function types, but strictly limits the number of wrappers on any one value. An untyped value acquires at most one wrapper. A typed value acquires at most two wrappers: one to protect itself from inputs, and a second to reflect the expectations of its current client:

$$\begin{array}{rcl@{\qquad}rcl} v_s & = & \mathbb{G}\ b\ \langle v, v \rangle & v_d & = & \mathbb{G}\ b\ \langle v, v \rangle \\ & | & \mathbb{G}\ b\ \lambda x.e & & | & \mathbb{G}\ b\ \lambda(x:\tau).e \\ & | & \mathbb{G}\ b\ (\mathbb{G}\ b\ \langle v, v \rangle) & & & \\ & | & \mathbb{G}\ b\ (\mathbb{G}\ b\ \lambda(x:\tau).e) & & & \end{array}$$

Forgetful enforces this two-wrapper limit by removing the outer wrapper of any guarded value that exits typed code. Re-entering

$\boxed{\text{Forgetful Syntax}}$ extends Higher-Order Evaluation Syntax

$$v \;=\; i \mid n \mid \langle v, v\rangle \mid \lambda x.\, e \mid \lambda(x:\tau).\, e \mid$$
$$\quad \mathbb{G}\,(\ell_{\blacktriangleleft}\,\tau \Rightarrow \tau_{\blacktriangleleft}\,\ell)\; v \mid \mathbb{G}\,(\ell_{\blacktriangleleft}\,\tau \times \tau_{\blacktriangleleft}\,\ell)\; v$$

$\boxed{e \rhd_{\mathsf{F}} e}$

$\mathsf{dyn}\;(\ell_0{\blacktriangleleft}\,\tau_0{\blacktriangleleft}\,\ell_1)\; v_0 \qquad \rhd_{\mathsf{F}} \mathbb{G}\,(\ell_0{\blacktriangleleft}\,\tau_0{\blacktriangleleft}\,\ell_1)\; v_0$
if *shape-match* $(\lfloor\tau_0\rfloor, v_0)$ and $v_0 \in \langle v, v\rangle \cup (\lambda x.\, e) \cup (\mathbb{G}\; b\; v)$

$\mathsf{dyn}\;(\ell_0{\blacktriangleleft}\,\tau_0{\blacktriangleleft}\,\ell_1)\; v_0 \qquad \rhd_{\mathsf{F}} i_0$
if *shape-match* $(\lfloor\tau_0\rfloor, v_0)$

$\mathsf{dyn}\;(\ell_0{\blacktriangleleft}\,\tau_0{\blacktriangleleft}\,\ell_1)\; v_0 \qquad \rhd_{\mathsf{F}} \mathsf{BndryErr}\;(\{(\ell_0{\blacktriangleleft}\,\tau_0{\blacktriangleleft}\,\ell_1)\}, v_0)$
if $\neg$*shape-match* $(\lfloor\tau_0\rfloor, v_0)$

$\mathsf{fst}\{\tau_0\}\;(\mathbb{G}\;(\ell_0{\blacktriangleleft}\,\tau_1 \times \tau_2{\blacktriangleleft}\,\ell_1)\; v_0) \qquad \rhd_{\mathsf{F}} \mathsf{dyn}\; b_0\; (\mathsf{fst}\{\mathcal{U}\}\; v_0)$
where $b_0 = (\ell_0{\blacktriangleleft}\,\tau_1{\blacktriangleleft}\,\ell_1)$

$\mathsf{snd}\{\tau_0\}\;(\mathbb{G}\;(\ell_0{\blacktriangleleft}\,\tau_1 \times \tau_2{\blacktriangleleft}\,\ell_1)\; v_0) \qquad \rhd_{\mathsf{F}} \mathsf{dyn}\; b_0\; (\mathsf{snd}\{\mathcal{U}\}\; v_0)$
where $b_0 = (\ell_0{\blacktriangleleft}\,\tau_2{\blacktriangleleft}\,\ell_1)$

$\mathsf{app}\{\tau_0\}\;(\mathbb{G}\;(\ell_0{\blacktriangleleft}\,\tau_1 \times \tau_2{\blacktriangleleft}\,\ell_1)\; v_0)\; v_1 \rhd_{\mathsf{F}} \mathsf{dyn}\; b_0\; (\mathsf{app}\{\mathcal{U}\}\; v_0\; (\mathsf{stat}\; b_1\; v_1))$
where $b_0 = (\ell_0{\blacktriangleleft}\,\tau_2{\blacktriangleleft}\,\ell_1)$ and $b_1 = (\ell_1{\blacktriangleleft}\,\tau_1{\blacktriangleleft}\,\ell_0)$

$\boxed{e \blacktriangleright_{\mathsf{F}} e}$

$\mathsf{stat}\;(\ell_0{\blacktriangleleft}\,\tau_0{\blacktriangleleft}\,\ell_1)\; v_0 \qquad \blacktriangleright_{\mathsf{F}} \mathbb{G}\;(\ell_0{\blacktriangleleft}\,\tau_0{\blacktriangleleft}\,\ell_1)\; v_0$
if *shape-match* $(\lfloor\tau_0\rfloor, v_0)$ and $v_0 \in \langle v, v\rangle \cup (\lambda(x:\tau).\, e)$

$\mathsf{stat}\;(\ell_0{\blacktriangleleft}\,\tau_0{\blacktriangleleft}\,\ell_1)\; (\mathbb{G}\; b_1\; v_0) \qquad \blacktriangleright_{\mathsf{F}} v_0$
if *shape-match* $(\lfloor\tau_0\rfloor, v_0)$
and $v_0 \in \langle v, v\rangle \cup (\lambda x.\, e) \cup (\mathbb{G}\; b\; \langle v, v\rangle) \cup (\mathbb{G}\; b\; (\lambda(x:\tau).\, e))$

$\mathsf{stat}\;(\ell_0{\blacktriangleleft}\,\tau_0{\blacktriangleleft}\,\ell_1)\; i_0 \qquad \blacktriangleright_{\mathsf{F}} i_0$
if *shape-match* $(\lfloor\tau_0\rfloor, i_0)$

$\mathsf{stat}\;(\ell_0{\blacktriangleleft}\,\tau_0{\blacktriangleleft}\,\ell_1)\; v_0 \qquad \blacktriangleright_{\mathsf{F}} \mathsf{InvariantErr}$
if $\neg$*shape-match* $(\lfloor\tau_0\rfloor, v_0)$

$\mathsf{fst}\{\mathcal{U}\}\;(\mathbb{G}\;(\ell_0{\blacktriangleleft}\,\tau_0 \times \tau_1{\blacktriangleleft}\,\ell_1)\; v_0) \qquad \blacktriangleright_{\mathsf{F}} \mathsf{stat}\; b_0\; (\mathsf{fst}\{\tau_0\}\; v_0)$
where $b_0 = (\ell_0{\blacktriangleleft}\,\tau_0{\blacktriangleleft}\,\ell_1)$

$\mathsf{snd}\{\mathcal{U}\}\;(\mathbb{G}\;(\ell_0{\blacktriangleleft}\,\tau_0 \times \tau_1{\blacktriangleleft}\,\ell_1)\; v_0) \qquad \blacktriangleright_{\mathsf{F}} \mathsf{stat}\; b_0\; (\mathsf{snd}\{\tau_1\}\; v_0)$
where $b_0 = (\ell_0{\blacktriangleleft}\,\tau_1{\blacktriangleleft}\,\ell_1)$

$\mathsf{app}\{\mathcal{U}\}\;(\mathbb{G}\;(\ell_0{\blacktriangleleft}\,\tau_0 \Rightarrow \tau_1{\blacktriangleleft}\,\ell_1)\; v_0)\; v_1 \blacktriangleright_{\mathsf{F}} \mathsf{stat}\; b_0\; (\mathsf{app}\{\tau_1\}\; v_0\; (\mathsf{dyn}\; b_1\; v_1))$
where $b_0 = (\ell_0{\blacktriangleleft}\,\tau_1{\blacktriangleleft}\,\ell_1)$ and $b_1 = (\ell_1{\blacktriangleleft}\,\tau_0{\blacktriangleleft}\,\ell_0)$

$\boxed{e \rightarrow^*_{\mathsf{F}} e} \;=\; \rightarrow^*_{\cup\{\rhd_{\mathsf{F}}, \blacktriangleright_{\mathsf{F}}, \blacktriangleright, \rhd\}}$

Figure 38: Forgetful notions of reduction

typed code makes a new wrapper, but these wrappers do not accumulate because a value cannot enter typed code twice in a row; it must first exit typed code and lose one wrapper.

Removing outer wrappers does not affect the type soundness of untyped code; all well-formed values match $\mathcal{U}$, with or without wrappers. Type soundness for typed code is guaranteed by the temporary outer wrappers. Complete monitoring is lost, however, because the removal of a wrapper creates a joint-ownership situation. Similarly, Forgetful lies above Co-Natural and Natural in the error preorder.

When a type mismatch occurs, Forgetful blames one boundary. Though sound, this one boundary is generally not enough information to find the source of the problem. So, Forgetful fails to satisfy blame completeness.

Theorem 4.5.14. *Forgetful satisfies* **TS(1)**.

Proof Idea. By progress and preservation lemmas for the higher-order typing judgment ($\vdash_1$). The most interesting proof case shows that dropping a guard wrapper does not break type soundness. Suppose that a pair v_0 with static type $\mathsf{Int} \times \mathsf{Int}$ crosses two boundaries and re-enters typed code at a different type.

$$\mathsf{dyn}\ (\ell_0 \blacktriangleleft (\mathsf{Nat} \times \mathsf{Nat}) \blacktriangleleft \ell_1)\ (\mathsf{stat}\ (\ell_1 \blacktriangleleft \mathsf{Int} \times \mathsf{Int} \blacktriangleleft \ell_2)\ v_0)\ \rightarrow^*_{\mathsf{F}}$$
$$\mathbb{G}\ (\ell_0 \blacktriangleleft (\mathsf{Nat} \times \mathsf{Nat}) \blacktriangleleft \ell_1)\ (\mathbb{G}\ (\ell_1 \blacktriangleleft \mathsf{Int} \times \mathsf{Int} \blacktriangleleft \ell_2)\ v_0)$$

No matter what value v_0 is, the result is well-typed because the context trusts the outer wrapper. If this double-wrapped value—call it v_2—crosses another boundary, Forgetful drops the outer wrapper. Nevertheless, the result is a sound dynamically-typed value:

$$\mathsf{stat}\ (\ell_3 \blacktriangleleft (\mathsf{Nat} \times \mathsf{Nat}) \blacktriangleleft \ell_0)\ v_2\ \rightarrow^*_{\mathsf{F}}$$
$$\mathbb{G}\ (\ell_1 \blacktriangleleft \mathsf{Int} \times \mathsf{Int} \blacktriangleleft \ell_2)\ v_0$$

When this single-wrapped wrapped pair reenters a typed context, it again gains a wrapper to document the context's expectation:

$$\mathsf{dyn}\ (\ell_4 \blacktriangleleft (\tau_1 \times \tau_2) \blacktriangleleft \ell_3)\ (\mathbb{G}\ (\ell_1 \blacktriangleleft \mathsf{Int} \times \mathsf{Int} \blacktriangleleft \ell_2)\ v_0)\ \rightarrow^*_{\mathsf{F}}$$
$$\mathbb{G}\ (\ell_4 \blacktriangleleft (\tau_1 \times \tau_2) \blacktriangleleft \ell_3)\ (\mathbb{G}\ (\ell_1 \blacktriangleleft \mathsf{Int} \times \mathsf{Int} \blacktriangleleft \ell_2)\ v_0)$$

The new wrapper preserves soundness. □

Theorem 4.5.15. *Forgetful does not satisfy* **CM**.

Proof. Consider the lifted variant of the stat rule that removes an outer guard wrapper:

$$(\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ ((\mathbb{G}\ b_1\ v_0))^{\overline{\ell}_2})^{\ell_3} \blacktriangleright_{\overline{\mathsf{F}}}\ ((v_0))^{\overline{\ell}_2 \ell_3}$$
$$\text{if } \mathit{shape\text{-}match}\ (\lfloor \tau_0 \rfloor, (\mathbb{G}\ b_1\ v_0))$$

Suppose $\ell_0 \neq \ell_1$. If the redex satisfies single-owner consistency, then $\overline{\ell}_2$ contains ℓ_1 and $\ell_3 = \ell_0$. Thus the rule creates a contractum with two distinct labels. □

Theorem 4.5.16. *Forgetful satisfies* **BS**.

Proof. By a preservation lemma for a weakened version of the $\Vdash$ judgment, which is defined in the appendix. The judgment asks whether the owners on a value contain at least the name of the current component. Forgetful easily satisfies this invariant because the ownership guidelines (section 4.4.4) never drop an un-checked label. Thus, when a boundary error occurs:

$$\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 \ \rhd_{\mathsf{F}}\ \mathsf{BndryErr}\ (\{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\}, v_0) \quad \text{if } \neg \textit{shape-match}\ (\lfloor \tau_0 \rfloor, v_0)$$

the sender name ℓ_1 matches one of the ownership labels on v_0. □

Theorem 4.5.17. *Forgetful does not satisfy* **BC**.

Proof. The proof of theorem 4.5.15 shows how a pair value can acquire two labels. A function can gain owners in a similar fashion, and reach an incompatible boundary:

$$\mathsf{dyn}\ (\ell_2 \blacktriangleleft \mathsf{Int} \blacktriangleleft \ell_1)\ ((\lambda x_0.\, x_0))^{\ell_0 \ell_1} \ \rhd_{\mathsf{F}}$$
$$\mathsf{BndryErr}\ (\{(\ell_2 \blacktriangleleft \mathsf{Int} \blacktriangleleft \ell_1)\}, ((\lambda x_0.\, x_0))^{\ell_0 \ell_1})$$

In this example, the error does not point to component ℓ_0. □

Theorem 4.5.18. $C \lesssim F$.

Proof Idea. By a stuttering simulation. Co-Natural can take extra steps at an elimination form to unwrap an arbitrary number of wrappers; Forgetful has at most two to unwrap.

In the other direction, $F \not\lesssim C$ because Forgetful drops checks. Let:

$$e_0 = \mathsf{stat}\ b_0\ (\mathsf{dyn}\ (\ell_0 \blacktriangleleft (\mathsf{Nat} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\ (\lambda x_0.\, x_0))$$
$$e_1 = \mathsf{app}\{\mathcal{U}\}\ e_0\ \langle 2, 8 \rangle$$

Then $e_1 \rightarrow_{\mathsf{F}}^{*} \langle 2, 8 \rangle$ and Co-Natural raises a boundary error. □

4.5.8 *Transient and its Properties*

The Transient semantics in figure 39 builds on the flat evaluation syntax (figure 33); it stores pairs and functions on a heap as indicated by the syntax of figure 33, and aims to enforce type constructors (s, or $\lfloor \tau \rfloor$) through shape checks. For every pre-value w stored on a heap $\mathcal{H}$, there is a corresponding entry in a blame map $\mathcal{B}$ that points to a set of boundaries. The blame map provides information if a mismatch occurs, following Reticulated Python [113, 115].

Unlike for the higher-order-checking semantics, there is significant overlap between the Transient rules for typed and untyped redexes. Thus figure 39 presents one notion of reduction. The first group of rules in figure 39 handle boundary expressions and check expressions. When a value reaches a boundary, Transient matches its shape against the expected type. If successful, the value crosses the boundary and its blame map records the fact; otherwise, the program halts. For a dyn boundary, the result is a boundary error. For a stat boundary, the

$\boxed{\text{Transient Syntax}}$ extends First-Order Evaluation Syntax

$v \;=\; i \mid n \mid \mathsf{p}$

$\boxed{e; \mathcal{H}; \mathcal{B} \rhd_{\mathsf{T}} e; \mathcal{H}; \mathcal{B}}$ (selected rules)

$(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0); \mathcal{H}_0; \mathcal{B}_0 \;\rhd_{\mathsf{T}}\; v_0; \mathcal{H}_0; (\mathcal{B}_0[v_0 \cup \{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\}])$
if *shape-match* $(\lfloor \tau_0 \rfloor, \mathcal{H}_0(v_0))$

$(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0); \mathcal{H}_0; \mathcal{B}_0 \;\rhd_{\mathsf{T}}\; \mathsf{BndryErr}\ (\{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\}, v_0); \mathcal{H}_0; \mathcal{B}_0$
if $\neg$*shape-match* $(\lfloor \tau_0 \rfloor, \mathcal{H}_0(v_0))$

$(\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0); \mathcal{H}_0; \mathcal{B}_0 \;\rhd_{\mathsf{T}}\; v_0; \mathcal{H}_0; (\mathcal{B}_0[v_0 \cup \{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\}])$
if *shape-match* $(\lfloor \tau_0 \rfloor, \mathcal{H}_0(v_0))$

$(\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0); \mathcal{H}_0; \mathcal{B}_0 \;\rhd_{\mathsf{T}}\; \mathsf{InvariantErr}; \mathcal{H}_0; \mathcal{B}_0$
if $\neg$*shape-match* $(\lfloor \tau_0 \rfloor, \mathcal{H}_0(v_0))$

$(\mathsf{check}\{\mathcal{U}\}\ v_0\ \mathsf{p}_0); \mathcal{H}_0; \mathcal{B}_0 \;\rhd_{\mathsf{T}}\; v_0; \mathcal{H}_0; \mathcal{B}_0$

$(\mathsf{check}\{\tau_0\}\ v_0\ \mathsf{p}_0); \mathcal{H}_0; \mathcal{B}_0 \;\rhd_{\mathsf{T}}\; v_0; \mathcal{H}_0; (\mathcal{B}_0[v_0 \cup \mathcal{B}_0(\mathsf{p}_0)])$
if *shape-match* $(\lfloor \tau_0 \rfloor, \mathcal{H}_0(v_0))$

$(\mathsf{check}\{\tau_0\}\ v_0\ \mathsf{p}_0); \mathcal{H}_0; \mathcal{B}_0 \;\rhd_{\mathsf{T}}\; \mathsf{BndryErr}\ (\mathcal{B}_0(v_0) \cup \mathcal{B}_0(\mathsf{p}_0), v_0); \mathcal{H}_0; \mathcal{B}_0$
if $\neg$*shape-match* $(\lfloor \tau_0 \rfloor, \mathcal{H}_0(v_0))$

$(unop\{^{\tau}/_{\mathcal{U}}\}\ \mathsf{p}_0); \mathcal{H}_0; \mathcal{B}_0 \;\rhd_{\mathsf{T}}\; (\mathsf{check}\{^{\tau}/_{\mathcal{U}}\}\ \delta(unop, \mathcal{H}_0(\mathsf{p}_0))\ \mathsf{p}_0); \mathcal{H}_0; \mathcal{B}_0$
if $\delta(unop, \mathcal{H}_0(\mathsf{p}_0))$ is defined

$(binop\{^{\tau}/_{\mathcal{U}}\}\ i_0\ i_1); \mathcal{H}_0; \mathcal{B}_0 \;\rhd_{\mathsf{T}}\; \delta(binop, i_0, i_1); \mathcal{H}_0; \mathcal{B}_0$
if $\delta(binop, i_0, i_1)$ is defined

$(\mathsf{app}\{\tau_0\}\ \mathsf{p}_0\ v_0); \mathcal{H}_0; \mathcal{B}_0 \;\rhd_{\mathsf{T}}\; (\mathsf{check}\{\tau_0\}\ e_0[x_0 \leftarrow v_0]\ \mathsf{p}_0); \mathcal{H}_0; \mathcal{B}_1$
if $\mathcal{H}_0(\mathsf{p}_0) = \lambda x_0.\, e_0$
and $\mathcal{B}_1 = \mathcal{B}_0[v_0 \cup rev\,(\mathcal{B}_0(\mathsf{p}_0))]$

$(\mathsf{app}\{\mathcal{U}\}\ \mathsf{p}_0\ v_0); \mathcal{H}_0; \mathcal{B}_0 \;\rhd_{\mathsf{T}}\; (e_0[x_0 \leftarrow v_0]); \mathcal{H}_0; \mathcal{B}_0$
if $\mathcal{H}_0(\mathsf{p}_0) = \lambda x_0.\, e_0$

$(\mathsf{app}\{^{\tau}/_{\mathcal{U}}\}\ \mathsf{p}_0\ v_0); \mathcal{H}_0; \mathcal{B}_0 \;\rhd_{\mathsf{T}}\; (\mathsf{check}\{^{\tau}/_{\mathcal{U}}\}\ e_0[x_0 \leftarrow v_0]\ \mathsf{p}_0); \mathcal{H}_0; \mathcal{B}_1$
if $\mathcal{H}_0(\mathsf{p}_0) = \lambda(x_0 : \tau_0).\, e_0$ and *shape-match* $(\lfloor \tau_0 \rfloor, \mathcal{H}_0(v_0))$
and $\mathcal{B}_1 = \mathcal{B}_0[v_0 \cup rev\,(\mathcal{B}_0(\mathsf{p}_0))]$

$(\mathsf{app}\{^{\tau}/_{\mathcal{U}}\}\ \mathsf{p}_0\ v_0); \mathcal{H}_0; \mathcal{B}_0 \;\rhd_{\mathsf{T}}\; \mathsf{BndryErr}\ (rev\,(\mathcal{B}_0(\mathsf{p}_0)), v_0); \mathcal{H}_0; \mathcal{B}_1$
if $\mathcal{H}_0(\mathsf{p}_0) = \lambda(x_0 : \tau_0).\, e_0$ and $\neg$*shape-match* $(\lfloor \tau_0 \rfloor, \mathcal{H}_0(v_0))$
where $\mathcal{B}_1 = \mathcal{B}_0[v_0 \cup rev\,(\mathcal{B}_0(\mathsf{p}_0))]$

$\mathsf{w}_0; \mathcal{H}_0; \mathcal{B}_0 \;\rhd_{\mathsf{T}}\; \mathsf{p}_0; (\{\mathsf{p}_0 \mapsto \mathsf{w}_0\} \cup \mathcal{H}_0); (\{\mathsf{p}_0 \mapsto \emptyset\} \cup \mathcal{B}_0)$
where p_0 fresh in $\mathcal{H}_0$ and $\mathcal{B}_0$

$\boxed{e; \mathcal{H}; \mathcal{B} \rightarrow^*_{\mathsf{T}} e; \mathcal{H}; \mathcal{B}} \;=\; \rightarrow^*_T$
where $E[e_0]; \mathcal{H}_0; \mathcal{B}_0 \;\; T \;\; E[e_1]; \mathcal{H}_1; \mathcal{B}_1$ if $e_0; \mathcal{H}_0; \mathcal{B}_0 \rhd_{\mathsf{T}} e_1; \mathcal{H}_1; \mathcal{B}_1$

Figure 39: Transient notions of reduction

mismatch reflects an invariant error in typed code. Check expressions similarly match a value against a type-shape. On success, the blame map gains the boundaries associated with the location p_0 from which the value originated. On failure, these same boundaries may help the programmer diagnose the fault.

The second group of rules handle primitives and application. Pair projections and function applications must be followed by a check in typed contexts to enforce the type annotation at the elimination form. In untyped contexts, a check for the dynamic type embeds a possibly-typed subexpression. The binary operations are not elimination forms, so they are not followed by a check. Applications of typed functions additionally check the input value against the function's domain type. If successful, the blame map records the check. Otherwise, Transient reports the boundaries associated with the function [115]. Note that untyped functions may appear in typed contexts, and vice-versa.

Applications of untyped functions in untyped code do not update the blame map. This allows an implementation to insert all checks by rewriting typed code at compile-time, leaving untyped code as is. Protected typed code can then interact with any untyped libraries.

Not shown in figure 39 are rules for elimination forms that halt the program. When δ is undefined or when a non-function is applied, the result is either an invariant error or a tag error depending on the context—analogous to the higher-order semantics.

Transient shape checks do not guarantee full type soundness, complete monitoring, or the standard blame soundness and completeness. They do, however, preserve the top-level shape of all values in typed code. Furthermore, Transient satisfies a heap-based notion of blame soundness. Blame completeness fails because Transient does not update the blame map when an untyped function is applied in an untyped context.

Theorem 4.5.19. *Transient does not satisfy* **TS(1)**.

Proof Idea. Let $e_0 = \mathsf{dyn}\ (\ell_0 \blacktriangleleft (\mathsf{Nat} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\ (\lambda x_0.\, -4)$.

- $\vdash e_0 : \mathsf{Nat} \Rightarrow \mathsf{Nat}$ in the surface syntax, but
- $e_0; \emptyset; \emptyset \rightarrow^*_{\mathsf{T}} \mathsf{p}_0; \mathcal{H}_0; \mathcal{B}_0$, where $\mathcal{H}_0(\mathsf{p}_0) = (\lambda x_0.\, -4)$

and $\not\vdash_{\mathbf{1}} (\lambda x_0.\, -4) : \mathsf{Nat} \Rightarrow \mathsf{Nat}$. □

Theorem 4.5.20. *Transient satisfies* **TS(s)**.

Proof Idea. Recall that **s** maps types to type shapes and the unitype to itself. The proof depends on progress and preservation lemmas for the flat typing judgment ($\vdash_{\mathbf{s}}$). Although Transient lets any well-shaped value cross a boundary, the check expressions that appear after elimination forms preserve soundness. Suppose that an untyped

function crosses a boundary and eventually computes an ill-typed result:

$(\mathsf{app}\{\mathsf{Int}\}\ \mathsf{p}_0\ 4); \mathcal{H}_0; \mathcal{B}_0 \ \triangleright_{\mathsf{T}}\ (\mathsf{check}\{\mathsf{Int}\}\ \langle 4, \mathsf{plus}\{\mathcal{U}\}\ 4\ 1\rangle\ \mathsf{p}_0); \mathcal{H}_0; \mathcal{B}_1$
if $\mathcal{H}_0(\mathsf{p}_0) = \lambda x_0.\, \langle x_0, \mathsf{plus}\{\mathcal{U}\}\ x_0\ 1\rangle$
and $\mathcal{B}_1 = \mathcal{B}_0[v_0 \cup \mathit{rev}\,(\mathcal{B}_0(\mathsf{p}_0))]$

The check expression guards the context. □

Theorem 4.5.21. *Transient does not satisfy* **CM**.

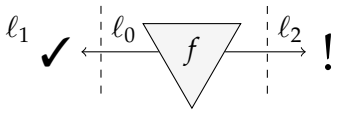
Proof. A structured value can cross any boundary with a matching shape, regardless of the deeper type structure. For example, the following annotated rule adds a new label to a pair:

$(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\ ((\mathsf{p}_0))^{\overline{\ell}_2})^{\ell_3}; \mathcal{H}_0; \mathcal{B}_0 \ \triangleright_{\overline{\mathsf{T}}}\ ((\mathsf{p}_0))^{\overline{\ell}_2 \ell_3}; \mathcal{H}_0; \mathcal{B}_1$
where $\mathcal{H}_0(\mathsf{p}_0) \in \langle v, v\rangle$

□

Theorem 4.5.22. *Transient does not satisfy* **BS**.

Proof. Let component ℓ_0 define a function f_0 and export it to components ℓ_1 and ℓ_2. If component ℓ_2 triggers a type mismatch, as sketched below, then Transient blames both component ℓ_2 and the irrelevant ℓ_1:



The following term expresses this scenario using a let-expression to abbreviate untyped function application:

$(\mathsf{let}\ f_0 = (\lambda x_0.\, \langle x_0, x_0\rangle)\ \mathsf{in}$
$\mathsf{let}\ f_1 = (\mathsf{stat}\ (\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Int}) \blacktriangleleft \ell_1)\ (\mathsf{dyn}\ (\ell_1 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Int}) \blacktriangleleft \ell_0)\ (f_0)^{\ell_0})^{\ell_1})\ \mathsf{in}$
$\mathsf{stat}\ (\ell_0 \blacktriangleleft \mathsf{Int} \blacktriangleleft \ell_2)\ (\mathsf{app}\{\mathsf{Int}\}\ (\mathsf{dyn}\ (\ell_2 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Int}) \blacktriangleleft \ell_0)\ (f_0)^{\ell_0})\ 5)^{\ell_2})^{\ell_0}; \emptyset; \emptyset$

Reduction ends in a boundary error that blames three components. □

Theorem 4.5.23. *Transient does not satisfy* **BC**.

Proof. An untyped function application in untyped code does not update the blame map:

$(\mathsf{app}\{\mathcal{U}\}\ \mathsf{p}_0\ v_0); \mathcal{H}_0; \mathcal{B}_0 \ \triangleright_{\mathsf{T}}\ (e_0[x_0 \leftarrow v_0]); \mathcal{H}_0; \mathcal{B}_0$
if $\mathcal{H}_0(\mathsf{p}_0) = \lambda x_0.\, e_0$

Such applications lead to incomplete blame when the function has previously crossed a type boundary. To illustrate, the term below uses an untyped identity function f_1 to coerce the type of another function (f_0). After the coercion, an application leads to type mismatch.

$(\mathsf{let}\ f_0 = \mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ (\mathsf{dyn}\ (\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_2)\ (\lambda x_0.\, x_0))\ \mathsf{in}$
$\mathsf{let}\ f_1 = \mathsf{stat}\ (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_3)\ (\mathsf{dyn}\ (\ell_3 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_4)\ (\lambda x_1.\, x_1))\ \mathsf{in}$
$\mathsf{stat}\ (\ell_0 \blacktriangleleft (\mathsf{Int} \times \mathsf{Int}) \blacktriangleleft \ell_5)$
$\quad (\mathsf{app}\{\mathsf{Int} \times \mathsf{Int}\}\ (\mathsf{dyn}\ (\ell_5 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0)\ (\mathsf{app}\{\mathcal{U}\}\ f_1\ f_0)^{\ell_0})\ 42)^{\ell_5})^{\ell_0}; \emptyset; \emptyset$

Reduction ends in a boundary error that does not report the crucial labels ℓ_3 and ℓ_4. □

The results so far paint a negative picture of the wrapper-free Transient semantics. It fails **CM** and **BC** because it has no interposition mechanism to keep track of type implications for untyped code. Additionally, its heap-based approach to blame fails **BS** because the blame heap conflates different paths in a program. If several clients use the same library function and one client encounters a type mismatch, everyone gets blamed.

We have struggled to find a positive characterization of Transient's blame behavior. Complete monitoring and blame completeness appear unattainable, even in a weakened form, because Transient has no control over untyped code. Blame soundness, however, is possible under a relaxed specification of ownership that adds one guideline to the "natural laws" from chapter 4.4.4:

8. If an address gains a label, then so does the associated pre-value on the heap.

$$\mathsf{fst}\ (\mathsf{p}_0)^{\ell_0}; \mathcal{H}_0; \mathcal{B}_0; O_0 \to (\mathsf{p}_1)^{\ell_0}; \mathcal{H}_0; \mathcal{B}_0; O_0[\mathsf{p}_1 \cup \{\ell_0\}]$$
$$\textit{where } \mathcal{H}_0(\mathsf{p}_0) = \langle \mathsf{p}_1, \mathsf{p}_2 \rangle$$

Law 3 propagates the outer label, which goes up to the ownership heap (O).

Intuitively, the new specification pushes all ownership labels onto the heap. Rather than push to the value heap ($\mathcal{H}$) directly, though, the extended model of Transient in the appendix introduces a parallel store (O) analogous to the blame heap.

Merging ownership labels on the heap is a non-compositional behavior. A programmer cannot reason about a local expression without thinking about how the rest of the codebase may introduce new owners. Because of this whole-program action, it is unclear whether the weakened notions of blame that follow are useful guarantees to strive for. Nevertheless, they help characterize Transient.

Definition 4.5.24 (heap-based blame soundness and blame completeness)**.** *For all well-formed e_0 such that $e_0 \to^*_{\mathsf{X}} \mathsf{BndryErr}\ (b_0^*, v_0); \mathcal{H}_0; \mathcal{B}_0; O_0$:*

- *X satisfies* **BS-h** *iff senders $(b_0^*) \subseteq$ owners $(v_0) \cup O_0(v_0)$*
- *X satisfies* **BC-h** *iff senders $(b_0^*) \supseteq$ owners $(v_0) \cup O_0(v_0)$.*

Theorem 4.5.25. *Transient satisfies* **BS-h**.

Proof Idea. By a preservation lemma for the $\Vdash_h$ judgment sketched in figure 40 and fully-defined in the appendix. The judgment ensures

$\boxed{\mathcal{O}; \mathcal{L}; \ell \Vdash_h e; \mathcal{B}}$ (selected rules)

$$\frac{\mathit{senders}\,(\mathcal{B}_0(\mathsf{p}_0)) \subseteq \mathcal{O}_0(\mathsf{p}_0)}{\mathcal{O}_0; \mathcal{L}_0; \ell_0 \Vdash_h \mathsf{p}_0; \mathcal{B}_0}$$

$$\frac{\mathcal{O}_0; \mathcal{L}_0; \ell_0 \Vdash_h e_0; \mathcal{B}_0 \qquad \mathit{senders}\,(\mathcal{B}_0(\mathsf{p}_0)) \subseteq \mathcal{O}_0(\mathsf{p}_0)}{\mathcal{O}_0; \mathcal{L}_0; \ell_0 \Vdash_h \mathsf{check}\{\tau_0\}\, e_0\, \mathsf{p}_0; \mathcal{B}_0}$$

Figure 40: Heap-based ownership consistency for Transient

that the blame map records a subset of the true owners on each heap-allocated value. One subtle case of the proof concerns function application, because the unlabeled rule appears to blame a typed function (at address p_0) for an unrelated incompatible value:

$$(\mathsf{app}\{{}^{\tau}/_{U}\}\ \mathsf{p}_0\ v_0); \mathcal{H}_0; \mathcal{B}_0 \ \rhd_{\mathsf{T}}\ \mathsf{BndryErr}\,(\mathit{rev}\,(\mathcal{B}_0(\mathsf{p}_0)), v_0); \mathcal{H}_0; \mathcal{B}_1$$
$$\text{if } \mathcal{H}_0(\mathsf{p}_0) = \lambda(x_0 : \tau_0).\, e_0 \text{ and } \neg \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, \mathcal{H}_0(v_0))$$
$$\text{where } \mathcal{B}_1 = \mathcal{B}_0[v_0 \cup \mathit{rev}\,(\mathcal{B}_0(\mathsf{p}_0))]$$

But, the value is not unrelated because the shape check happens when this value meets the function's type annotation; that is, after the function receives the input value. By law 4, the correct labeling matches the following outline:

$$(\mathsf{app}\{{}^{\tau}/_{U}\}\ ((\mathsf{p}_0))^{\overline{\ell}_0}\ ((v_0))^{\overline{\ell}_1})^{\ell_0}; \ldots \ \rhd_{\overline{\mathsf{T}}}$$
$$(\mathsf{BndryErr}\,(\ldots, ((v_0))^{\overline{\ell}_1 \ell_0 \mathit{rev}\,(\overline{\ell}_0)}))^{\ell_0}; \ldots$$

Additionally blaming $\mathcal{B}_0(v_0)$ seems like a useful change to the original Transient semantics because it offers more information. Thanks to heap-based ownership, the technical justification is that adding these boundaries preserves the **BS-h** property. □

Theorem 4.5.26. *Transient does not satisfy* **BC-h**.

Proof. Blame completeness fails because Transient does not update the blame map during an untyped function application. Refer to the proof of theorem 4.5.23 for an example. □

Theorem 4.5.27. $F \lesssim T$.

Proof Idea. Indirectly, via $T \approx A$ (theorem 4.5.31) and $F \lesssim A$ (theorem 4.5.32). □

4.5.9 *Amnesic and its Properties*

The Amnesic semantics employs the same dynamic checks as Transient but offers path-based blame information. Whereas Transient

$\boxed{\text{Amnesic Syntax}}$ extends Higher-Order Evaluation Syntax

$v \;=\; i \mid n \mid \langle v, v\rangle \mid \lambda x.\, e \mid \lambda(x:\tau).\, e \mid$
$\quad \mathbb{G}\,(\ell \blacktriangleleft \tau \Rightarrow \tau \blacktriangleleft \ell)\; v \mid \mathbb{G}\,(\ell \blacktriangleleft \tau \times \tau \blacktriangleleft \ell)\; v \mid \mathbb{T}\; b^*\; v$

$\boxed{e \rhd_{\mathsf{A}} e}$ (selected rules)

$\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \quad \rhd_{\mathsf{A}} \quad \mathbb{G}\,(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0$
if $\mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, v_0)$
and $\mathit{rem\text{-}trace}\,(v_0) \in \langle v, v\rangle \cup (\lambda(x:\tau).\, e) \cup (\mathbb{G}\; b\; v)$

$\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \quad \rhd_{\mathsf{A}} \quad v_0$
if $\mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, v_0)$ and $\mathit{rem\text{-}trace}\,(v_0) \in i$

$\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \quad \rhd_{\mathsf{A}} \quad \mathsf{BndryErr}\,(\{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\} \cup b_0^*, v_0)$
if $\neg \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, v_0)$ and $b_0^* = \mathit{get\text{-}trace}\,(v_0)$

$\mathsf{fst}\{\tau_0\}\;(\mathbb{G}\,(\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)\; v_0) \quad \rhd_{\mathsf{A}} \quad \mathsf{dyn}\; b_0\; (\mathsf{fst}\{\mathcal{U}\}\; v_0)$
where $b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)$

$\mathsf{snd}\{\tau_0\}\;(\mathbb{G}\,(\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)\; v_0) \quad \rhd_{\mathsf{A}} \quad \mathsf{dyn}\; b_0\; (\mathsf{snd}\{\mathcal{U}\}\; v_0)$
where $b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)$

$\mathsf{app}\{\tau_0\}\;(\mathbb{G}\,(\ell_0 \blacktriangleleft \tau_1 \Rightarrow \tau_2 \blacktriangleleft \ell_1)\; v_0)\; v_1 \quad \rhd_{\mathsf{A}} \quad \mathsf{dyn}\; b_0\; (\mathsf{app}\{\mathcal{U}\}\; v_0\; (\mathsf{stat}\; b_1\; v_1))$
where $b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)$ and $b_1 = (\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0)$

Figure 41: Amnesic notions of reduction (1/2)

$\boxed{e \blacktriangleright_{\mathsf{A}} e}$ (selected rules)

$\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 \quad \blacktriangleright_{\mathsf{A}} \quad \mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0$
if $\mathit{shape\text{-}match}\ (\lfloor \tau_0 \rfloor, v_0)$ and $v_0 \in \langle v, v \rangle \cup (\lambda(x : \tau).\, e)$

$\mathsf{stat}\ b_0\ (\mathbb{G}\ b_1\ (\mathbb{T}^?\ b_0^*\ v_0)) \quad \blacktriangleright_{\mathsf{A}} \quad \mathsf{trace}\ (\{b_0, b_1\} \cup b_0^*)\ v_0$
if $b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)$ and $\mathit{shape\text{-}match}\ (\lfloor \tau_0 \rfloor, v_0)$
and $v_0 \in \langle v, v \rangle \cup (\lambda x.\, e) \cup (\mathbb{G}\ b\ (\lambda(x : \tau).\, e)) \cup (\mathbb{G}\ b\ \langle v, v \rangle)$

$\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ i_0 \quad \blacktriangleright_{\mathsf{A}} \quad i_0$
if $\mathit{shape\text{-}match}\ (\lfloor \tau_0 \rfloor, i_0)$

$\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 \quad \blacktriangleright_{\mathsf{A}} \quad \mathsf{InvariantErr}$
if $\neg \mathit{shape\text{-}match}\ (\lfloor \tau_0 \rfloor, v_0)$

$\mathsf{fst}\{\mathcal{U}\}\ (\mathbb{T}^?\ b_0^*\ (\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\ v_0)) \quad \blacktriangleright_{\mathsf{A}} \quad \mathsf{trace}\ b_0^*\ (\mathsf{stat}\ b_0\ (\mathsf{fst}\{\tau_0\}\ v_0))$
where $b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)$

$\mathsf{snd}\{\mathcal{U}\}\ (\mathbb{T}^?\ b_0^*\ (\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\ v_0)) \quad \blacktriangleright_{\mathsf{A}} \quad \mathsf{trace}\ b_0^*\ (\mathsf{stat}\ b_0\ (\mathsf{snd}\{\tau_1\}\ v_0))$
where $b_0 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)$

$\mathsf{app}\{\mathcal{U}\}\ (\mathbb{T}^?\ b_0^*\ (\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0))\ v_1 \quad \blacktriangleright_{\mathsf{A}} \quad \mathsf{trace}\ b_0^*\ (\mathsf{stat}\ b_0\ (\mathsf{app}\{\tau_2\}\ v_0\ e_0))$
where $\tau_0 = \tau_1 \Rightarrow \tau_2$ and $b_0 = (\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1)$ and $b_1 = (\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0)$
and $e_0 = (\mathsf{dyn}\ b_1\ (\mathit{add\text{-}trace}\ (\mathit{rev}\ (b_0^*), v_1)))$

$\mathsf{trace}\ b_0^*\ v_0 \quad \blacktriangleright_{\mathsf{A}} \quad v_1$
where $v_1 = \mathit{add\text{-}trace}\ (b_0^*, v_0)$

$\boxed{e \to_{\mathsf{A}}^* e} \ = \ \to^*_{\cup\{\triangleright_{\mathsf{A}}, \blacktriangleright_{\mathsf{A}}, \blacktriangleright, \triangleright\}}$

Figure 42: Amnesic notions of reduction (2/2)

$$\textit{add-trace}\,(b_0^*, v_0) = \begin{cases} v_0 & \text{if } b_0^* = \emptyset \\ \mathbb{T}\,(b_0^* \cup b_1^*)\; v_1 & \text{if } v_0 = \mathbb{T}\,b_1^*\,v_1 \\ \mathbb{T}\,b_0^*\,v_0 & \text{if } v_0 \notin \mathbb{T}\,b^*\,v \text{ and } b_0^* \neq \emptyset \end{cases}$$

$$\textit{get-trace}\,(v_0) = \begin{cases} b_0^* & \text{if } v_0 = \mathbb{T}\,b_0^*\,v_1 \\ \emptyset & \text{if } v_0 \notin \mathbb{T}\,b^*\,v \end{cases}$$

$$\textit{rem-trace}\,(v_0) = \begin{cases} v_1 & \text{if } v_0 = \mathbb{T}\,b_0^*\,v_1 \\ v_0 & \text{if } v_0 \notin \mathbb{T}\,b^*\,v \end{cases}$$

$$(\mathbb{T}^?\,b_0^*\,v_0) = v_1 \iff \textit{rem-trace}\,(v_1) = v_0 \text{ and } \textit{get-trace}\,(v_1) = b_0^*$$

Figure 43: Metafunctions for Amnesic

indirectly tracks blame through heap addresses, Amnesic uses trace wrappers to keep boundaries alongside the value at hand.

Amnesic bears a strong resemblance to the Forgetful semantics. Both use guard wrappers in the same way, keeping a sticky "inner" wrapper around typed values and a temporary "outer" wrapper in typed contexts. There are two crucial differences:

- When Amnesic removes a guard wrapper, it saves the boundary specification in a trace wrapper. The number of boundaries in a trace can grow without bound, but the number of wrappers around a value is limited to three.
- At elimination forms, Amnesic checks only the context's type annotation. Suppose an untyped function enters typed code at one type and is used at a supertype:
  $\mathsf{app}\{\mathsf{Int}\}\;(\mathbb{G}\;(\ell_0 \blacktriangleleft (\mathsf{Nat} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\;\lambda x_0.\,-7)\;2$
  Amnesic runs this application without error but Forgetful raises a boundary error.

Thus, the following wrapped values can occur at run-time. Note that $(\mathbb{T}^?\,b^*\,e)$ is short for an expression that may or may not have a trace wrapper (figure 43).

$$\begin{array}{rcl} v_s & = & \mathbb{G}\,b\,(\mathbb{T}^?\,b^*\,\langle v, v\rangle) \\ & | & \mathbb{G}\,b\,(\mathbb{T}^?\,b^*\,\lambda x.\,e) \\ & | & \mathbb{G}\,b\,(\mathbb{T}^?\,b^*\,(\mathbb{G}\,b\,\langle v, v\rangle)) \\ & | & \mathbb{G}\,b\,(\mathbb{T}^?\,b^*\,(\mathbb{G}\,b\,\lambda(x:\tau).\,e)) \end{array} \qquad \begin{array}{rcl} v_d & = & \mathbb{T}\,b^*\,i \\ & | & \mathbb{T}\,b^*\,\langle v, v\rangle \\ & | & \mathbb{T}\,b^*\,\lambda x.\,e \\ & | & \mathbb{T}^?\,b^*\,(\mathbb{G}\,b\,\langle v, v\rangle) \\ & | & \mathbb{T}^?\,b^*\,(\mathbb{G}\,b\,\lambda(x:\tau).\,e) \end{array}$$

The elimination rules for guarded pairs show the clearest difference between checks in Amnesic and Forgetful. Amnesic ignores the type in the guard. Forgetful ignores the type on the primitive operation.

Figure 41 defines three metafunctions and one abbreviation for trace wrappers. The metafunctions extend, retrieve, and remove the

$\boxed{\mathcal{L};\ell \Vdash_p e}$ (selected rules), extends $\mathcal{L};\ell \Vdash e$

$$\frac{b_0^* = \{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1) \cdots (\ell_{n-1} \blacktriangleleft \tau_{n-1} \blacktriangleleft \ell_n)\} \qquad \mathcal{L}_0;\ell_n \Vdash_p v_0}{\mathcal{L}_0;\ell_0 \Vdash_p \left(\mathbb{T}\ b_0^*\ ((v_0))^{\ell_n \cdots \ell_1}\right)^{\ell_0}}$$

Figure 44: Path-based ownership consistency for trace wrappers

boundaries associated with a value. The abbreviation lets reduction rules accept optionally-traced values.

Amnesic satisfies full type soundness thanks to guard wrappers and fails complete monitoring because it drops wrappers. This is no surprise, since Amnesic creates and removes guard wrappers in the same manner as Forgetful. Unlike the Forgetful semantics, Amnesic uses trace wrappers to remember the boundaries that a value has crossed. This information leads to sound and complete blame error outputs.

Theorem 4.5.28. *Amnesic satisfies* **TS(1)**.

Proof Idea. By progress and preservation lemmas for the higher-order typing judgment ($\vdash_1$). Amnesic creates and drops wrappers in the same manner as Forgetful (theorem 4.5.14), so the only interesting proof cases concern elimination forms. For example, when Amnesic extracts an element from a guarded pair it ignores the type in the guard ($\tau_1 \times \tau_2$):

$$\mathsf{fst}\{\tau_0\}\ (\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_1 \times \tau_2 \blacktriangleleft \ell_1)\ v_0)\ \rhd_{\mathsf{A}}\ \mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ (\mathsf{fst}\{\mathcal{U}\}\ v_0)$$

The new boundary enforces the context's assumption (τ_0) instead, but we know that τ_0 is a supertype of τ_1 by the definition of the higher-order typing judgment. □

Theorem 4.5.29. *Amnesic does not satisfy* **CM**.

Proof Idea. Removing a wrapper creates a value with more than one label:

$$(\mathsf{stat}\ (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\ ((\mathbb{G}\ b_1\ ((\mathbb{T}\ b_0^*\ ((\lambda x_0.\ x_0))^{\overline{\ell}_2}))^{\overline{\ell}_3}))^{\overline{\ell}_4})^{\ell_5} \quad \blacktriangleright_{\overline{\mathsf{A}}}$$
$$((\mathsf{trace}\ (\{(\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1), b_1\} \cup b_0^*)\ ((\lambda x_0.\ x_0))^{\overline{\ell}_2}))^{\overline{\ell}_3 \overline{\ell}_4 \ell_5}$$

□

Theorem 4.5.30. *Amnesic satisfies* **BS** *and* **BC**.

Proof Idea. By progress and preservation lemmas for a path-based consistency judgment, $\Vdash_p$, that weakens single-owner consistency to

allow multiple labels around a trace-wrapped value. Unlike the heap-based consistency for Transient, which requires an entirely new judgment, path-based consistency only replaces the rules for trace wrappers (shown in figure 44) and trace expressions. Now consider the guard-dropping rule:

$$(\mathsf{stat}\ (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\ ((\mathbb{G}\ b_1\ ((\mathbb{T}\ b_0^*\ ((\lambda x_0.\ x_0))^{\overline{\ell}_2}))^{\overline{\ell}_3}))^{\overline{\ell}_4\ \ell_5}) \blacktriangleright_{\overline{\mathsf{A}}}$$
$$((\mathsf{trace}\ (\{(\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1), b_1\} \cup b_0^*)\ ((\lambda x_0.\ x_0))^{\overline{\ell}_2}))^{\overline{\ell}_3\overline{\ell}_4\ell_5}$$

Path-consistency for the redex implies that $\overline{\ell}_3$ and $\overline{\ell}_4$ match the component names on the boundary b_1, and that the client side of b_1 matches the outer sender ℓ_1. Thus the new labels on the result match the sender names on the two new boundaries in the trace. □

Theorem 4.5.31. *$T \approx A$.*

Proof Idea. By a stuttering simulation between Transient and Amnesic. Amnesic may take extra steps at an elimination form and to combine traces into one wrapper. Transient takes extra steps to place pre-values on the heap and to conservatively check the result of elimination forms. In fact, Amnesic and Transient behave exactly the same aside from bookkeeping to create wrappers and track blame. □

Theorem 4.5.32. *$F \lesssim A$.*

Proof Idea. By a lock-step bisimulation. The only difference between Forgetful and Amnesic comes from subtyping. Forgetful uses wrappers to enforce the type on a boundary. Amnesic uses boundary types only for an initial shape check, and instead uses the static types in typed code to guide checks at elimination forms. In the following $A \not\lesssim F$ example, a boundary declares one type and an elimination form requires a weaker type:

$$\mathsf{fst}\{\mathsf{Int}\}\ (\mathsf{dyn}\ (\ell_0 \blacktriangleleft (\mathsf{Nat} \times \mathsf{Nat}) \blacktriangleleft \ell_1)\ \langle -4, 4 \rangle)$$

Since -4 is an integer, Amnesic reduces to a value. Forgetful detects an error. □

4.5.10 *Erasure and its Properties*

Figure 45 presents the values and notions of reduction for the Erasure semantics. Erasure ignores all types at runtime. As the first two reduction rules show, any value may cross any boundary. When an incompatible value reaches an elimination form, the result depends on the context. In untyped code, the redex steps to a standard tag error. In typed code, however, the malformed redex indicates that an ill-typed value crossed a boundary. Thus Erasure ends with a boundary error at the last possible moment; these errors come with no information because there is no record of the relevant boundary.

Erasure Syntax extends Erasure Evaluation Syntax

$v = i \mid n \mid \langle v, v \rangle \mid \lambda x.\, e \mid \lambda(x : \tau).\, e$

$e \rhd_{\mathsf{E}} e$

$\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 \rhd_{\mathsf{E}} v_0$

$\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 \rhd_{\mathsf{E}} v_0$

$unop\{\tau_0\}\ v_0 \rhd_{\mathsf{E}} \mathsf{BndryErr}\ (\emptyset, v_0)$
if $\delta(unop, v_0)$ is undefined

$unop\{\mathcal{U}\}\ v_0 \rhd_{\mathsf{E}} \mathsf{TagErr}$
if $\delta(unop, v_0)$ is undefined

$unop\{{}^{\tau}/_{\mathcal{U}}\}\ v_0 \rhd_{\mathsf{E}} \delta(unop, v_0)$
if $\delta(unop, v_0)$ is defined

$binop\{\tau_0\}\ v_0\ v_1 \rhd_{\mathsf{E}} \mathsf{BndryErr}\ (\emptyset, v_i)$
if $\delta(binop, v_0, v_1)$ is undefined and $v_i \notin \mathsf{Int}$

$binop\{\mathcal{U}\}\ v_0\ v_1 \rhd_{\mathsf{E}} \mathsf{TagErr}$
if $\delta(binop, v_0, v_1)$ is undefined

$binop\{{}^{\tau}/_{\mathcal{U}}\}\ v_0\ v_1 \rhd_{\mathsf{E}} \delta(binop, v_0, v_1)$
if $\delta(binop, v_0, v_1)$ is defined

$\mathsf{app}\{\tau_0\}\ v_0\ v_1 \rhd_{\mathsf{E}} \mathsf{BndryErr}\ (\emptyset, v_0)$
if $v_0 \notin (\lambda x.\, e) \cup (\lambda(x : \tau).\, e)$

$\mathsf{app}\{\mathcal{U}\}\ v_0\ v_1 \rhd_{\mathsf{E}} \mathsf{TagErr}$
if $v_0 \notin (\lambda x.\, e) \cup (\lambda(x : \tau).\, e)$

$\mathsf{app}\{{}^{\tau}/_{\mathcal{U}}\}\ (\lambda(x_0 : \tau_0).\, e_0)\ v_0 \rhd_{\mathsf{E}} e_0[x_0 \leftarrow v_0]$

$\mathsf{app}\{{}^{\tau}/_{\mathcal{U}}\}\ (\lambda x_0.\, e_0)\ v_0 \rhd_{\mathsf{E}} e_0[x_0 \leftarrow v_0]$

Figure 45: Erasure notions of reduction

Theorem 4.5.33. *Erasure satisfies neither* **TS(1)** *nor* **TS(s)**.

Proof. Dynamic-to-static boundaries are unsound. An untyped function, for example, can enter a typed context that expects an integer:

$$\mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Int} \blacktriangleleft \ell_1)\ (\lambda x_0.\, 42)\ \rhd_{\mathsf{E}}\ (\lambda x_0.\, 42)$$

□

Theorem 4.5.34. *Erasure satisfies* **TS(0)**.

Proof Idea. By progress and preservation lemmas for the erased, or dynamic-typing, judgment ($\vdash_{\mathbf{0}}$). Given well-formed input, every $\rhd_{\mathsf{E}}$ rule yields a dynamically-typed result. □

Theorem 4.5.35. *Erasure does not satisfy* **CM**.

Proof Idea. A static-to-dynamic boundary can create a value with multiple labels:

$$(\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ (v_0)^{\ell_2})^{\ell_3}\ \rhd_{\overline{\mathsf{E}}}\ ((v_0))^{\ell_2 \ell_3}$$

□

Theorem 4.5.36.

- *Erasure satisfies* **BS**.
- *Erasure does not satisfy* **BC**.

Proof Idea. An Erasure boundary error blames an empty set, for example:

$$\mathsf{fst}\{\mathsf{Int}\}\ (\lambda x_0.\, x_0)\ \rhd_{\mathsf{E}}\ \mathsf{BndryErr}\ (\emptyset, (\lambda x_0.\, x_0))$$

The empty set is trivially sound and incomplete. □

Theorem 4.5.37. $A \lesssim E$.

Proof Idea. By a stuttering simulation. Amnesic takes extra steps at elimination forms, to enforce types, and to create trace wrappers.

As a countexample showing $A \not\lesssim E$, the following applies an untyped function:

$$\mathsf{app}\{\mathsf{Nat}\}\ (\mathsf{dyn}\ (\ell_0 \blacktriangleleft (\mathsf{Nat} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\ (\lambda x_0.\, -9))\ 4$$

Amnesic checks for a natural-number result and errors, but Erasure checks nothing. □

4.6 DISCUSSION

One central design issue of a mixed-typed language is the semantics of types and specifically how their integrity is enforced at the boundaries between typed and untyped code. Among other things, the choice determines whether typed code can trust the static types and the quality of assistance that a programmer receives when a mixed-typed interaction goes wrong. Without an interaction story,

mixed-typed programs are no better than dynamically-typed programs when it comes to run-time errors. Properties that hold for the typed half of the language are only valid under a closed-world assumption [12, 20, 79]; such properties are an important starting point, but make no contribution to the overall goal.

As the analysis of this chapter demonstrates, the limitations of the host language determine the invariants that a language designer can hope to enforce. First, higher-order wrappers enable strong guarantees, but need support from the host runtime system. For example, Typed Racket is a mature language but lacks wrappers for certain higher-order values. A language without wrappers of any sort can provide weaker guarantees by rewriting typed code and maintaining a global map of blame metadata. If this metadata can be attached directly to a value, then stronger blame guarantees are in reach.

More precisely, this chapter analyzes six distinct semantics via four properties (table 2) and establishes an error preorder relation:

- Type soundness is a relatively weak property for mixed-typed programs; it determines whether typed code can trust its own types. Except for the Erasure semantics, which does nothing to enforce types, type soundness does not clearly distinguish the various strategies.

- Complete monitoring is a stronger property, adapted from the literature on higher-order contracts [25]. It holds when *untyped* code can trust type specifications and vice-versa; see chapter 4.2.3 for examples.

The last two properties tell a developer what aid to expect if a type mismatch occurs.

- Blame soundness states that every boundary in a blame message is potentially responsible. Four strategies satisfy blame soundness relative to a standard, path-based notion of responsibility. Transient satisfies blame soundness only if the notion of responsibility is weakened to merge distinct references to the same heap-allocated value. Erasure is trivially blame-sound because it gives the programmer zero type-related information.

- Blame completeness states that every blame error comes with an overapproximation of the responsible parties. Three of the four blame-sound semantics also satisfy blame completeness. Forgetful can be modified to satisfy this property. The Erasure strategy trivially fails blame completeness. And the Transient strategy fails because it has no way to supervise untyped values that flow through a typed context.

Table 2 points out, however, that the weakest strategies are the only ones that do not require wrapper values. Wrappers impose space

Table 2: Technical contributions

	N	$\lesssim$	C	$\lesssim$	F	$\lesssim$	T	$\simeq$	A	$\lesssim$	E
type soundness	**1**		**1**		**1**		**s**†		**1**		**0**
complete monitoring	✓		✓		×		×		×		×
blame soundness	✓		✓		✓		*heap*		✓		∅
blame completeness	✓		✓		×‡		×		✓		×
no wrappers	×		×		×		✓		×		✓

† note that T is bisimilar to A (theorem 4.5.31)
‡ satisfiable by adding A-style trace wrappers, see appendix

costs, time costs, and object identity issues [57, 94, 114], but seem essential for strong mixed-typed guarantees. Perhaps future work can find a way to satisfy additional properties without using wrappers.

5 SHALLOW RACKET

The high costs of deep types and the weak guarantees of shallow types motivate a compromise. In a language that supports both, programmers can mix deep and shallow types to find an optimal tradeoff. This chapter presents the first half of the compromise, namely, a shallow semantics for Typed Racket. By default, Typed Racket provides deep types via the natural semantics. My work brings the transient semantics to the Typed Racket surface syntax.

Henceforth, Deep Racket refers to the original, natural implementation of Typed Racket and Shallow Racket refers to its transient implementation. Typed Racket refers to the common parts: the surface language and the type system.

Transient is a promising companion to Natural because it pursues an opposite kind of implementation. Whereas natural eagerly enforces full types with guard wrappers, transient lazily checks only top-level shapes. The lazy strategy means that transient does not need wrappers, which removes the main source of natural overhead. Transient can also run without blame, removing another form of runtime cost. By contrast, simply removing blame from natural changes little because blame information tags along with the guard wrappers. To fully benefit from removing blame, Natural needs a new strategy for allocating wrappers in the first place.

Adapting the theory of transient [115] to Typed Racket required several generalizations and insights (chapter 5.1). In the course of this work, I adapted the transient heap-based blame algorithm but identified several challenges that make it impractical (chapter 5.2). The final implementation takes care to reuse large parts of Typed Racket (chapter 5.3).

The performance of Shallow Racket is typically an improvement over Deep Racket, but both semantics have distinct strengths (chapter 5.4). Transient always adds overhead relative to untyped Racket, but is the safer bet for mixed-typed programs. Natural has better performance in programs with large chunks of typed code, and surpasses untyped Racket in many cases. Whether Shallow Racket can ever run faster than untyped code is an open question.

5.1 THEORY

Vitousek et al. [115] present a first transient semantics. This semantics communicates the key ideas behind transient and the behavior of Reticulated Python, but four characteristics make it unsuitable for a

transient implementation in Racket. It deals with a simpler language of static types; it includes a dynamic type; it does not include a subtyping relation; and its type checker is intertwined with the *completion* pass that rewrites typed code. This section outlines the design of a suitable model and its properties.

5.1.1 *More-Expressive Static Types*

The main design choices for Shallow Racket concern the run-time checks that enforce types. In terms of the model, there is a rich language τ of static types and the problem is to define a type-shape interpretation $\lfloor \tau \rfloor$ for each. A shape must be decidable; for example, $\lfloor \mathsf{Int} \Rightarrow \mathsf{Int} \rfloor = \mathsf{Int} \Rightarrow \mathsf{Int}$ is unacceptable without a predicate that can decide whether an untyped function always returns an integer when applied to an integer. Beyond decidability, type-shapes should be fast to test and imply useful properties. Shape soundness should be a meaningful property that helps a programmer debug and enables shape-directed optimizations.

The original transient model suggests that type-shapes must be decidable in constant time [115]. This model contains type constructors for only reference cells and functions, both of which are easily recognized in a dynamically-typed language. Reticulated, however, does not follow the constant-time suggestion in order to express object types. The type-shape for an object with N fields/methods checks for the presence of each member. Thus, the cost is linear in the size of an object type.

Shallow Racket includes additional linear-time shapes to support Typed Racket's expressive types with meaningful run-time checks. Some are linear in the size of a type; others are linear in the size of incoming values. In general, the goal is to enforce full constructors. The type-shape for a function checks arity; for example, the types $(\mathsf{Int} \Rightarrow \mathsf{Nat})$ and $(\mathsf{Int}\,\mathsf{Int} \Rightarrow \mathsf{Nat})$ have different shapes. The shape for a vector with a fixed number of elements checks length. And the shape for a list checks for a null-terminated sequence of pairs. Not all types correspond to value constructors, though. These amorphous type *connectives* [18, 19] call for recursive interpretations. For example, $\lfloor \tau_0 \cup \tau_1 \rfloor = \lfloor \tau_0 \rfloor \cup \lfloor \tau_1 \rfloor$ and $\lfloor \forall\, \alpha_0.\ \tau_0 \rfloor = \lfloor \tau_0 \rfloor$ provided $\lfloor \tau_0 \rfloor$ does not depend on the bound variable. Type variables have trivial shapes in other contexts, $\lfloor \alpha_0 \rfloor = \top$. Chapter 5.3.1 goes into more detail about the implementation.

5.1.2 *Removing Type Dynamic*

Reticulated Python provides a dynamic type in the micro gradual typing tradition. Consequently, the type system approves any elimination form on a dyn-typed expression and depends on a run-time

With Dyn

$$\frac{\Gamma_0 \vdash e_0 : \tau_0 \rightsquigarrow e_0' \qquad \Gamma_0 \vdash e_1 : \tau_1 \rightsquigarrow e_1' \qquad \tau_0 \rhd \tau_2 \Rightarrow \tau_3}{\Gamma_0 \vdash e_0\ e_1 : \tau_3 \rightsquigarrow \mathsf{check}\{\tau_3\}\ ((\mathsf{cast}\{(\tau_2 \Rightarrow \tau_3)\}\ e_0')\ (\mathsf{cast}\{\tau_2\}\ e_1'))}$$

Without Dyn

$$\frac{\Gamma_0 \vdash e_0 : \tau_2 \Rightarrow \tau_3 \rightsquigarrow e_0' \qquad \Gamma_0 \vdash e_1 : \tau_2 \rightsquigarrow e_1'}{\Gamma_0 \vdash e_0\ e_1 : \tau_3 \rightsquigarrow \mathsf{check}\{\tau_3\}\ (e_0'\ e_1')}$$

Figure 46: Transient completion rules for an application with a dynamic type (top) and without (bottom). Both rules insert run-time shape checks. The micro rule depends on a type coercion ($\rhd$) meta-function to allow down-casts from the dynamic type [115].

check to ensure that down-casts from the dynamic type work out. Typed Racket does not have a dynamic type; instead it adds run-time tools so that a non-dynamic type system can make assumptions about untyped input. Using this macro approach, only a handful of typing rules need to deal with untyped values.

The differences between the dynamic (micro) and non-dynamic (macro) typing rules have implications for transient run-time checks. In the original model, the evaluation of any expression could bring a dynamically-typed value into a typed context. For example, the application $(f\ 42)$ type checks when f has the dynamic type; a run-time check must test whether the value of f is a function. In a non-dyn model, only boundaries and elimination forms can introduce an untyped value. An application $(f\ 42)$ can only type check when f is a function. Figure 46 illustrates the difference by contrasting the transient checks needed for a function application. On the top, the dynamic approach requires three run-time checks in the worst case: two checks in case the function and argument are dynamically-typed, and one to validate the shape of the result. On the bottom, only one check is needed because the function and argument are certain to have a correct, non-dyn shape.

Note: Adding blame to a non-dynamic language adds the need for an additional blame-map operation in figure 46, but no additional checks. The blame map potentially needs an update because the argument flows in to the function. There is no need for a check because the argument has a non-dynamic type.

Other rules can be simplified in a similar fashion. The benefits are two-fold: non-dyn programs have fewer run-time checks to slow them down, and programmers have fewer places to search if a program manifests a boundary error.

```
;; (-> Real Real (U 'undef Real))
(define (divide n0 n1)
  (if (zero? n1)
    'undef
    (/ n0 n1)))
```

Figure 47: Untyped division function with two kinds of output. A typical gradual language without subtyping can only over-approximate the result with type dynamic.

5.1.3 *Adding Subtyping*

A type system for untyped code must either include a subtyping judgment or force programmers to rewrite their data definitions. Rewriting takes time and invites mistakes, therefore the migratory typing perspective demands a subtyping judgment.

The dynamic type is not a replacement for subtyping because it is an imprecise catch-all. For example, the untyped `divide` function in figure 47 either divides two numbers or returns the symbol `'undef` if the divisor is zero. Typed Racket lets a programmer express this union of two base types and subtyping justifies the code. By contrast, the dynamic type can type the code but asks callers to be ready for any possible return value.

Adapting transient to include subtyping is therefore an essential task for Shallow Racket. The addition is straightforward, but reveals a surprising distinction between boundary types and use-site types. Transient with subtyping may miss certain type mistakes declared at a boundary because its run-time checks are based on elimination forms. If a value goes through an upcast, then transient checks may not test the full type. Figure 48 illustrates the issue using list types. The typed function `nat-product` expects a list of non-negative integers and sends the list through an upcast to an `int-product` function that multiplies the list elements. Because of the upcast, there is a run-time check that `(first is)` returns an integer. There is no check that list elements are natural numbers, and so when untyped code sends an even-length list of negative numbers across the boundary, transient does not detect a mistake.

5.1.4 *From Elaboration to Completion*

Vitousek et al. [115] intertwine typing and transient checks in a type-elaboration judgment. The combination is a good fit for an imple-

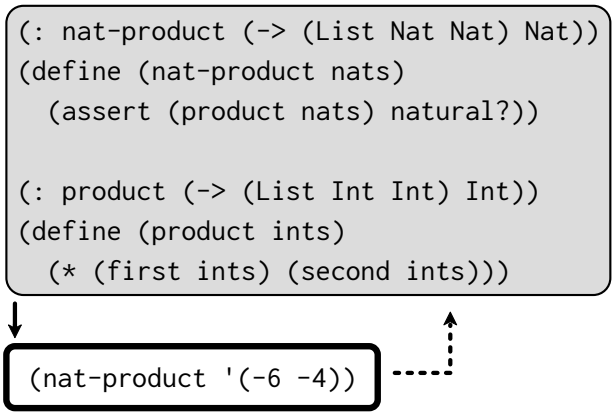


Figure 48: An upcast hides a type boundary mismatch from transient.

mentation because check-insertion depends on static types, and one pass over the program is more efficient than two. For the theory, however, it is better to keep surface typing separate from a second *completion* [52] pass that inserts transient checks

In the model of Shallow Racket, completion is a judgment ($\rightsquigarrow$) that transforms a well-typed surface term to a term with transient checks. The goal is to insert enough checks to create a target-language term with a similar type.

THEOREM SKETCH (completion correctness)
If $\Gamma \vdash e_0 : \tau_0$ *then* $\Gamma \vdash e_0 : \tau_0 \rightsquigarrow e_1$ *and* $\Gamma \vdash_{\mathbf{s}} e_1 : \lfloor \tau_0 \rfloor$.
Lemma 6.1.4 adapts the theorem sketch to a model.

The first benefit of this theorem is that it rules out nonsensical completions. The surface typing judgment ($\vdash$) establishes basic properties that a sensible completion must preserve. By contrast, a type elaboration that converts all surface terms to the integer 42 satisfies every theorem used to validate the original transient because elaboration is the only method for analyzing the surface syntax [115]. (This meta-theoretic lapse made it difficult to adapt transient to a new language of types.)

Second, the clear requirement makes it easier to adapt the idea of transient to a new language. If the language has its own surface-level typing and type-to-shape metafunction ($\lfloor \cdot \rfloor$), then completion correctness theorem guides the next steps.

Third, the separation encourages research on better completions and target-level typings. The challenge is to use as few checks as possible to build the target term. For example, suppose the variable `xy` points to a pair of numbers and consider the expression `(+ (car xy) (car xy))`. The completion for Shallow Racket produces the following term:

```
(+ (check Num (car xy)) (check Num (car xy)))
```

Racket guarantees left-to-right evaluation, however, so the second check can never fail. An improved completion would eliminate this check, other flow-dominated checks, and potentially many others.

5.2 WORK-IN-PROGRESS: BLAME

Blame is an important part of a migratory typing system because it strengthens the key weakness of migratory types. Static types guarantee that certain errors cannot occur at run-time. Migratory types are weak because they cannot offer the same promise. Errors can occur just about anywhere in typed code. With blame, however, type-mismatch errors come with an action plan for debugging. A programmer can follow the blame information to attempt a fix.

The usefulness of such an action plan depends on the blame strategy. The current-best algorithm for transient, from Vitousek et al. [115], blames a set of boundaries. The set is unsound and incomplete in the technical sense of chapter 4, but one would expect that it is more useful than no information.

Early experience with blame in Shallow Racket, however, has identified two significant challenges. First, scaling the original blame algorithm to Typed Racket raises questions about its accuracy. Second, transient blame has a tremendous performance cost. This section explains the challenges; performance concerns are deferred to chapter 5.4.4. Overall, I do not recommend the current blame algorithm for use in future implementations of transient.

5.2.1 *Basics of Transient Blame*

The transient blame algorithm uses a global *blame map* to connect run-time values to source-code boundaries. This blame map uses heap addresses as keys. Every non-primitive value in a program has a heap address and potentially a blame map entry. The values in a blame map are collections of entries. There are two kinds of entry in such a collection:

1. A *boundary entry* combines a type with a source location. Whenever a value crosses one of the static boundaries between typed and untyped code, the blame map gains a boundary entry. For example, if the function `f` flows out of typed code:

   ```
   (define (f (n : Natural)) : String
     ....)
   (provide f)
   ```

   then the blame map gains an entry for `f` that points to the type `(-> Natural String)` and a source location.

2. A *link entry* combines a parent pointer and an action. The parent refers to another blame map key. The action describes the relation between the current value and its parent. Suppose the function `f` from above gets applied to an untyped value `v`. As the value enters the function, the blame map gains a link entry for `v` that points to `f` with the action `'dom`, to remember that the current value is an input to the parent.

If a transient run-time check fails, the blame map can supply a set of boundaries by following parent pointers up from the failed value. Each parent pointer is partially responsible for the mismatched value. Each boundary at the root of all parent paths contains possibly-unchecked type assumptions. The programmer can begin debugging by reviewing these type assumptions.

Vitousek et al. [115] suggest a further refinement to this basic idea. They filter the set of typed boundaries using the failed value and the action path that led to the boundary. The action path gives a list of selectors to apply to the boundary type, ending with a smaller type. Checking this type against the bad value helps rule out irrelevant boundaries. For example, if the bad value is an integer and one of the boundary types expects an integer, then that boundary is not worth reporting.

In summary, the success of the blame map rests on three principles:

- every type boundary in the program adds one boundary entry in the map for each value that crosses the boundary at runtime;
- every elimination form adds a link entry with a correct parent and action; and
- there is a run-time way to test whether a value matches part of a boundary type.

These principles are relatively easy to satisfy in a model language, but pose surprising challenges for a full language.

5.2.2 *Trusted Libraries Obstruct Blame*

A migratory typing system must be able to re-use host language libraries. Racket, for example, comes with a list library that provides a `map` function. Both Deep Racket and Shallow Racket can re-use this function by declaring a static type:

```
map : (All (A B) (-> (-> A B) (Listof A) (Listof B)))
```

Furthermore, both can import `map` at no run-time cost. Deep can trust that the implementation completely follows the type and Shallow—without blame—can trust that `map` always returns a list.

For Shallow with blame, however, trusted re-use leads to imprecise blame. Figure 49 illustrates this phenomenon with a tiny example.

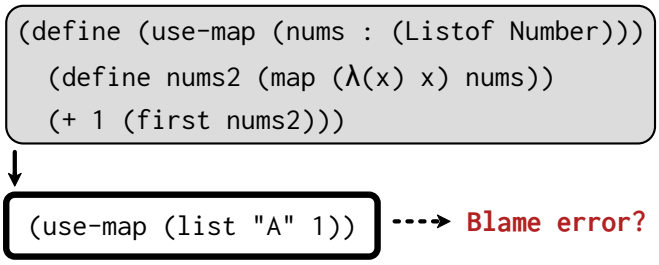


Figure 49: Unless `map` updates the blame map, transient cannot point to any boundaries when `(first nums2)` fails to return a number.

The typed function at the top of this figure expects a list of numbers, applies a trivial `map` to the list, and lastly finds a bad element in the mapped list. Transient blame should point back to the boundary between the typed function and the untyped list but cannot if `map` does not update the blame map. The solution is to register `map` in the blame map with a boundary entry and add link entries before and after every call. Unfortunately, the cost of this extra bookkeeping can add up.

5.2.3 *Complex Flows, Tailored Specifications*

Getting blame right for the `map` function requires careful bookkeeping. The result list must have a link entry that points to the input list. Additionally, the input function should point to this input list in case it receives a bad result.

Blame in Shallow Racket depends on literal syntax to decide when complex reasoning is needed. The original algorithm uses the same method; primitive operations have tailored bookkeeping and other function applications create link entries in a standard way [115].

Obviously, the syntactic approach is brittle. Renaming `map` leads to misleading blame errors. The same goes for applications of an expression instead of a literal identifier. Improving precision is an open challenge.

5.2.4 *Multi-Parent Paths*

A link entry points to one parent. Several functions, however, create data with multiple parents. One basic example is an `append` function on lists:

```
(append xs ys)
```

The result list contains the elements of both inputs. At a minimum, there should be two parents to blame if something goes wrong.

Action Template	Interpretation
`(dom n)`	n-th argument to a function
`(cod n)`	n-th result from a function
`(case-dom (k n))`	n-th argument (of k total) to an overloaded function
`(object-method (m n))`	n-th argument to method m of an object
`list-elem`	Element of a homogeneous list
`list-rest`	Tail of a list
`(list-elem n)`	Element of a heterogeneous list, e.g. `(List Boolean Number String)`
`hash-key`	Key of a hashtable
`hash-value`	Value of a hashtable
`(struct-field n)`	n-th field of a structure
`(object-field f)`	Field f of an object type
`noop`	No action; direct link to parent

Figure 50: Sample blame actions in Shallow Racket.

A second, more complicated example is a `hash-ref` function that may return a default value:

```
(hash-ref h k d)
```

If the table `h` has a binding for the key `k`, then the result comes from the table. Otherwise, the result is computed by the default thunk.

A blame map clearly needs conditional and multi-parent paths to give precise error outputs. But the cost of building and traversing the additional link entries may be high. Thus we leave such paths to future work.

5.2.5 *Expressive Link-Entry Actions*

A check entry in the blame map has two parts: a parent pointer and an action. The action informs the type-based filtering. Given a type for the parent, the action says what part of the type is relevant to the current value.

The model for Reticulated comes with three actions: `Res`, `Arg`, and `Deref`. These help traverse simple function types ($\tau \Rightarrow \tau$) and reference cells; for example, starting from the parent type `ref Int` and applying the `Deref` action focuses on the element type `Int`. The implementation of Reticulated adds one action, `Attr`, and generalizes `Arg` with an index. Starting from the following Reticulated type:

```
Function([int, str], float)
```

the action `[Arg, 1]` focuses on the type `str` of the second positional argument. Similarly, the action `[Attr, "foo"]` focuses on the member `"foo"` of an object type.

Despite the extensions, the action language in Reticulated suffers from imprecision in two ways. First, it has no way to refer to certain parts of a type. If a function uses optional or keyword arguments, then Reticulated has no way to test whether the type is irrelevant; such types cannot be filtered from the blame output. Second, it may conflate types. The action `Deref` seems to apply to any data structure. If a nested list value crosses the boundaries `List(List(int))` and `List(Dict(str, str))`, and then an elimination returns a string where an `int` was expected, a plain `Deref` incorrectly filters out the `Dict` type. The developer needs to see both types because neither matches the actual nested list value.

Shallow Racket thus comes with an extensive action language to prevent imprecision. Figure 50 presents a representative sample of actions and a brief description of each. Function actions must handle multiple arguments, multiple results, methods, and overloadings. Data structures have tailored actions. Lists, for example, require three kinds of actions: `list-elem` to dereference a simple list, `list-rest` to move to the tail of a list keeping the same type, and an indexed element action for fixed-sized lists with distinct types in each position. Finally, the `noop` action adds a direct link to track a copy from one data structure to another (`vector-copy!`) or a wrapper.

5.2.6 *Types at Runtime*

Transient blame needs types at runtime, or a close substitute, to filter irrelevant boundaries. These runtime types must have selectors for each possible action and an interpretation function that checks the shape of a value against the shape of a type.

Shallow Racket's runtime types are a revived version of its static types. During compilation, static types get serialized into a chain of type constructor calls. After a runtime error occurs, Shallow Racket re-evaluates the constructor definitions and uses these constructors to revive types. This revival approach re-uses at least 4,000 lines of Typed Racket to good effect: roughly 3,000 lines of constructor and selector definitions, and 1,000 lines that turn a type into a transient check. It also handles type aliases nicely. The static environment knows all relevant aliases and can serialize them along with the type.

Revival unfortunately fails for generative structure types. The runtime type and the static type are two different entities, and so Shallow Racket is unable to parse the serialized types at run-time. If parsing were to succeed, finding the correct predicate for a generative type is a separate challenge. At compile-time, it suffices to generate a correct

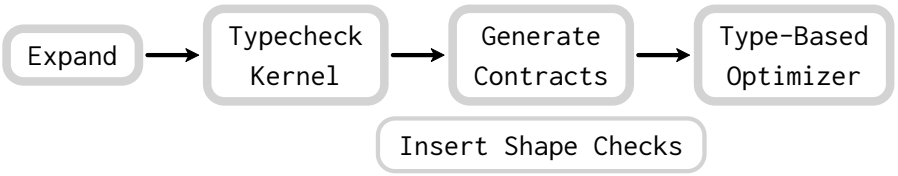


Figure 51: Stages in the Deep Racket compiler. Shallow can re-use the expander and type checker in full, and parts of the optimizer, but must replace contract generation with a rewriting pass.

identifier. At run-time, transient needs to evaluate a correct identifier in the right run-time context to find the predicate.

A different approach may be able to solve the generative-types problem. A related question, though, is whether transient is better off with a different method of filtering.

5.3 IMPLEMENTATION

Shallow Racket is an extension of the (Deep) Typed Racket codebase. The goal is not to create a fork, but rather to adapt the existing compiler and provide a uniform experience to programmers.

Figure 51 presents a high-level organization diagram of the Typed Racket compiler [106]. Source code goes through a macro-expansion step at the start. The type checker operates on expanded code; it validates the program and attaches type annotations as metadata for later passes. Third, the compiler turns boundary types into higher-order contracts. The last major step is the type-driven optimizer, which uses type annotations to remove unnecessary runtime checks.

Shallow Racket can re-use the expander and type checker as-is. The "contract" and "optimize" steps require changes. Contract generation must create transient checks rather than deep higher-order contracts (chapter 5.3.1). Additionally, the contract pass must rewrite all typed code with transient checks (chapter 5.3.2). The "optimize" pass must be restricted because it cannot rely on full types (chapter 5.3.3).

The implementation effort brought a few surprises. Challenges with universal types and occurrence types cause the current Shallow Racket to reject some well-typed code (chapter 5.3.2). Deep Racket rejects the same programs. The experience led to several improvements to Typed Racket, Racket, and libraries (chapter 5.3.4).

5.3.1 *Types to Shapes*

Shallow Racket compiles static types to *type shape* checks. Each check enforces first-order properties of a type constructor. In general, a

successful check means that all well-typed operations should succeed at run-time. For example, the type `(Pairof String String)` uses the `Pairof` type constructor; its shape check, `pair?`, accepts any kind of pair. A successful check `(pair? v)` means that the operations `(car v)` and `(cdr v)` are well-defined, and nothing more. Because these two operations are the only elimination forms for the `Pairof` constructor, the shape meets its goal.

Types that support many first-order properties have more complex shape checks. For example, an object type comes with field and method names. The shape check must ensure that incoming objects have the correct members.

Below are several more example types, chosen to illustrate the variety and challenges of extending transient. Each type comes with a shape that illustrates the implementation and a brief discussion. Actual shapes in the implementation do not use contract combinators such as `and/c` for performance.

- τ = `(Listof Real)`
  $\lfloor\tau\rfloor$ = `list?`

  The type represents lists of real numbers. The shape accepts any proper list, but not improper lists such as `(cons 1 (cons 2 3))`. The run-time cost depends on the size of input values in the worst case, but pairs are immutable and the predicate `list?` caches its results. The optimizer uses the shape to rewrite getters.

- τ = `(List Real Real)`
  $\lfloor\tau\rfloor$ = `(and/c list? (λ(v) (= 2 (length v))))`

  Represents a list with exactly two numbers. The shape checks lengths. Doing so lets the optimizer change `(list-ref v 1)` to `(unsafe-list-ref v 1)`.

- τ = `(Rec Chain (U Null (Pairof Chain Real)))`
  $\lfloor\tau\rfloor$ = `(or/c null? pair?)`

  The recursive type is isomorphic to `(Listof Real)`, but enforced with a more primitive check. In general, built-in lists have the only shape whose cost depends on the size of input values.

- τ = `(Vector Real)`
  $\lfloor\tau\rfloor$ = `(and/c vector? (λ(v) (= 1 (vector-length v))))`

  Represents a vector that contains one number. The shape checks length; the optimizer can use this fact.

- τ = `(Mutable-Vectorof Real)`
  $\lfloor\tau\rfloor$ = `(and/c vector? (not/c immutable?))`

  Represents a mutable vector with any number of elements. Vectors can also be immutable; the parent type `Vector` covers both.

The optimizer does not look at mutability, but the type checker does to raise static type errors.

- τ = `(Weak-HashTable Symbol Any)`
  $\lfloor\tau\rfloor$ = `(and/c hash? hash-weak?)`

  Represents a mutable hash table whose keys do not inhibit the garbage collector.

- τ = `(U Real String)`
  $\lfloor\tau\rfloor$ = `(or/c real? string?)`

  Untagged union. The shape accepts either a real number or a string; these predicates are elimination forms for the union because of occurrence typing [102, 105]. Wider unions, with N types inside, have shapes with N components.

- τ = `(Syntaxof String)`
  $\lfloor\tau\rfloor$ = `syntax?`

  Represents a syntax object that contains a string. The shape checks for a syntax object.

- τ = `Integer`
  $\lfloor\tau\rfloor$ = `exact-integer?`

  Represents a mathematical integer. The shape checks for exactness; an inexact integer such as `4.1` is not allowed.

  Other numeric types require larger checks for additional properties, for example `Negative-Integer` looks for an exact integer that is less than zero.

  To the type system, numeric types are wide unions. Shape enforcement flattens these unions wherever possible.

- τ = `(Refine [n : Integer] (= n 42))`
  $\lfloor\tau\rfloor$ = `(and/c exact-integer? (=/c 42))`

  Represents an integer that is equal to `42`.

  Refinement types attach a predicate to a static type. Predicates are limited to a linear arithmetic. The shape check uses the whole predicate.

- τ = `(Class (field [a Natural]) (get-a (-> Natural)))`
  $\lfloor\tau\rfloor$ = `(contract-first-order (class/c (field a) get-a))`

  Represents a class with one field and one method. The shape depends on the `racket/contract` library to check simple properties of class shape. Object types have similar checks, using `object/c` instead.

- τ = `(-> Real String)`
  $\lfloor\tau\rfloor$ = `(arity-includes/c 1)`

  Represents a function with one required argument. The shape checks arity.

- τ = `(-> Real * Real)`
  $\lfloor\tau\rfloor$ = `(arity-includes/c 0)`

  Represents a function that accepts any number of positional arguments. The shape looks for functions that can accept zero arguments, but it does not check whether they accept more.

- τ = `(case-> (-> Real Real) (-> String Real String))`
  $\lfloor\tau\rfloor$ = `(and/c (arity-includes/c 1) (arity-includes/c 2))`

  Represents an overloaded function. The shape checks both arities.

  Functions can also have optional, keyword, and optional keyword arguments. The shapes for such functions check that the keywords are accepted.

- τ = `(All (A) (Box A))`
  $\lfloor\tau\rfloor$ = `box?`

  Represents a polymorphic mutable cell. The shape checks for a cell. If typed code wants to extract a value from the cell, it must instatiate the polymorphic type. The instantiation provides a shape to check the box contents; to ensure soundness, such checks appear at every location where typed code reads from the box.

- τ = `(All (A) A)`
  $\lfloor\tau\rfloor$ = `none/c`

  Represents a value that can be instantiated to any type. The shape rejects all values.

  This type could be allowed with the trivial shape `any/c` in an implementation that checks the result of type instantiation, along the lines of New et al. [74]. Shallow Racket does nothing at instantiation, and therefore rejects the type to prevent unsoundness (figure 54).

5.3.2 *Inserting Shape Checks*

Shallow Racket rewrites typed code to include transient shape checks. Checks guard the positions where an untyped value might appear (chapter 4.5.8); in particular:

- at the source-code boundaries to untyped code;
- around elimination forms;
- and at the entry of every function.

Boundaries clearly need protection. If typed code expects a number and imports a value from untyped code, the value could have any shape and therefore needs a check.

```
(define (sum-list (nums : (Listof Real))) : Real
  (check! list? nums)
  (for/fold ([acc 0])
            ([n (in-list nums)])
    (check! real? n)
    (+ acc n)))
```

Figure 52: A shallow-typed function defended with transient checks.

```
(: array-append (-> (Listof Array) Array))
(define (array-append arrs [k 0])
  ;; append arrays along axis k
  ....)
```

Figure 53: Type prevents callers from sending an optional argument, but the function body can use the default value.

Elimination forms need protection for the same reason, but are an over-approximation. Figure 52 provides a concrete example with a for loop that sums up a list of numbers. Every step of the loop first checks the current list element. If the list came from untyped code, then the checks are clearly needed. The list might come from typed code, though, in which case the checks can never fail.

Figure 52 also contains a function check. The inputs to every typed function are checked to validate the type assumptions in the function body. These checks might be unnecessary if the function never escapes to untyped code, but escapes are hard to detect because a typed function can escape as an argument to a combinator (`map sum-list nss`) or via a macro-introduced reference.

Current Limitations

The current implementation attaches transient checks at two kinds of syntax: boundaries and run-time elimination forms. This approach does not suffice to protect all types, thus some well-typed programs are currently rejected to ensure soundness. Deep Typed Racket currently rejects these programs for the same reason.

Unrestricted universal types are one problem. If the shape $\lfloor \tau \rfloor$ of a universally-quantified type $\forall \alpha.\ \tau$ depends on the bound variable, then Shallow Racket rejects the program (figure 54). The trouble is

```
(require/typed racket/base
  ;; import `cdr`, assume type is correct,
  ;; depend on run-time check to catch nonsense
  (cdr (All (A) A)))

(define fake-str : String
  (inst cdr String))

(string-length fake-str)
```

Figure 54: If the shape of a universal type depends on the bound variable, then transient must either reject the program or treat type instantiation as an elimination form.

```
(require/typed racket/base
 (values (-> Any Any : String)))

(define x : Any 0)

(define fake-str : String
  (if (values x)
    (ann x String)
    (error 'unreachable)))

(string-length fake-str)
```

Figure 55: Occurrence types may change the type environment in each branch of an `if` statement. Transient must either check the changes or disallow occurrence types on untyped functions.

Topic	Shape-Safe?	
apply	✓	Deforest map-reduce exprs.
box	✓	Speed up box access.
dead-code	×	Remove if and case-lambda branches.
extflonum	✓	Rewrite math for extended floats.
fixnum	✓	Rewrite math for fixnums.
float-complex	✓	Unbox & rewrite complex float ops.
float	✓	Rewrite math for normal floats.
list	✓	Speed up list access and length.
number	✓	Rewrite basic numeric operations.
pair	×	Speed up (nested) pair access.
sequence	✓	Insert type hints for the runtime.
string	✓	Speed up string operations.
struct	✓	Speed up struct access.
vector	✓	Speed up vector access.

Figure 56: TR optimizations and whether Shallow can re-use them.

that type instantiation can change the shape of such types, but type instantiation is not currently a run-time elimination form.

Occurrence types at a boundary are a second problem. A program cannot assign an occurrence type to an untyped value, as in figure 55. This code uses `require/typed` to import an untyped function with a nonsensical occurrence type; it passes the type checker, but the compiler raises an error during contract generation because it cannot enforce the occurrence type. Proper enforcement requires rewriting both branches of the conditional to include casts based on the occurrence type. In this case, a check must ensure that `x` is a string.

5.3.3 *Optimizer*

Typed Racket uses static types to compile efficient code [89, 90, 91]. To give a basic example, a dynamically-typed sum `(+ n0 n1)` can be rewritten to add its inputs without first confirming that they are numbers. In principle, such optimizations may rely on full types. These "deep" optimizations are not safe for Shallow Racket because it only guarantees the top type constructor.

Figure 56 lists all optimization topics and shows, suprisingly, that only two are unsafe for shallow types. The `dead-code` pass remove type-inaccessible branches of an overloaded function. With deep types, run-time contracts make these branches inaccessible. Shallow types allow raw functions to flow to untyped code, and therefore the branches are not sealed off by a wrapper. The `pair` pass depends on

	kind	merged?	pull request
1	bugfix	✓	`racket/htdp #98`
2	bugfix	✓	`racket/pict #60`
3	bugfix	✓	`racket/racket #3182`
4	bugfix		`racket/typed-racket #926`
5	bugfix	✓	`racket/typed-racket #919`
6	bugfix	✓	`racket/typed-racket #916`
7	bugfix	✓	`racket/typed-racket #914`
8	bugfix	✓	`racket/typed-racket #912`
9	bugfix	✓	`racket/typed-racket #923`
10	bugfix	✓	`racket/typed-racket #921`
11	bugfix	✓	`racket/typed-racket #918`
12	bugfix	✓	`racket/typed-racket #913`
13	bugfix	✓	`racket/typed-racket #884`
14	bugfix	✓	`racket/typed-racket #855`
15	bugfix	✓	`racket/typed-racket #612`
16	bugfix	✓	`racket/typed-racket #600`
17	enhancement	✓	`racket/typed-racket #927`
18	enhancement	✓	`racket/typed-racket #925`
19	enhancement	✓	`racket/typed-racket #911`
20	enhancement	✓	`racket/typed-racket #907`
21	enhancement		`racket/typed-racket #917`

Figure 57: Pull requests inspired by work on Shallow Racket.

full types to rewrite nested accessors, such as `cdar`, to versions that assume a deep pair structure.

Other passes are re-used in Shallow Racket. The benefit of these optimizations is sometimes enough to outweigh the cost of transient checks (chapter 5.4). Certain re-used passes, though, force design decisions. The `apply` pass directly applies a typed function to the elements of a list. This transformation is shape sound because all shallow-typed functions check their inputs, whether or not they escape to untyped code. The `list` and `sequence` passes depend on the $O(n)$ shape check for list types. The unboxing in the `float-complex` pass is only safe by virtue of a conservative escape analysis.

5.3.4 *Bonus Fixes and Enhancements*

The development of Shallow Racket led to several improvements in other Racket libraries. Debugging sessions occasionally revealed bugs in existing code, and the integration of Shallow and Deep Racket suggested enhancements for the latter. Figure 57 tabulates these fixes

and enhancements. The third column shows that all but two requests are merged. The final column contains links with more details.

Most improvements came about through transient run-time checks. During compilation, transient relies on types embedded in an intermediate representation to generate checks. Missing types and imprecise types caused problems at this completion step; on occasion, the problems were due to Typed Racket bugs. At run-time, transient sometimes found incorrect types with its checks. The HTDP fix offers a simple example (`racket/htdp #98`). A library-provided function promised to return a unit value and actually returned a boolean. Transient caught the unsoundness.

The fix to Racket is especially interesting (`racket/racket #3182`). It came about because some shallow-typed programs failed with a strange error message:

```
Expected a real number, got #<unsafe-undefined>
```

These programs were fully-typed, but somehow a run-time value contradicted the type checker without causing trouble in the Deep semantics. Worse, this sentinel undefined value did not appear in the source code. The problem was due to a disagreement between core Racket and Typed Racket about how to encode a method with optional arguments as a function with a fixed-length argument list. Racket used an extra run-time check; Typed Racket thought the check was redundant. The fix was indeed to change Racket, which means that pre-fix versions of Typed Racket are a hair's breadth from a dangerous unsoundness. Their saving grace is that the type optimizer does not transform methods; if it did, then user code would receive unsafe-undefined values because of the incorrect type assumption.

5.4 PERFORMANCE

Shallow Racket sacrifices static guarantees for a wrapper-free implementation. The loss of wrappers implies a loss of full type soundness, complete monitoring, and correct blame. As compensation, shallow needs to demonstrate improved performance.

This section applies the method from chapter 3 to evaluate Shallow Racket on the GTP benchmarks (version 6.0). The granularity of the experiment is module-level, same as our Deep Racket experiment from chapter 3.5. All data is from a dedicated Linux box with 4 physical i7-4790 3.60GHz cores and 16GB RAM. The experiment uses Racket v7.8.0.5 (`7c90387`) and Shallow Racket extends Typed Racket v1.12 (`c074c93`).

Benchmark	deep/untyped	shallow/untyped
sieve	0.97	4.36
forth	0.65	5.21
fsm	0.54	2.38
fsmoo	0.88	4.28
mbta	1.63	1.69
morsecode	0.73	2.72
zombie	1.79	31.07
dungeon	0.99	4.97
jpeg	0.40	1.66
zordoz	1.35	2.73
lnm	0.64	1.06
suffixtree	0.69	5.51
kcfa	1.04	1.16
snake	0.96	7.67
take5	0.97	2.97
acquire	1.22	1.40
tetris	0.97	8.28
synth	0.96	4.07
gregor	0.98	1.53
quadT	0.99	7.22
quadU	0.79	7.14

Figure 58: Performance ratios (fully-typed vs untyped) for deep and shallow types on the GTP benchmarks.

5.4.1 *Performance Ratios*

Figure 58 presents typed/untyped ratios for the benchmarks. The middle column lists the overhead of fully-typed deep code relative to the untyped configuration. The right column shows the overhead of fully-typed shallow types.

Because these shallow types are implemented with the transient semantics, one would expect them to run slower than deep types because the latter has no overhead in completely typed programs. Indeed, deep runs faster in every row and has a worst-case overhead under 2x. Shallow typically does well, with overhead under 5x, but a few benchmarks have larger slowdowns due to transient checks. The worst is zombie, which suffers a 30x overhead due to the many elimination forms that appear in one module. This module simulates objects with functions and therefore contains several layers of indirection that all slow down with transient checks. A better completion pass may be able to reduce this high cost; that said, the real-time cost of this overhead in zombie is close to one second. The best-case benchmark for transient is lnm, which runs only slightly slower than the fully-untyped configuration.

5.4.2 *Overhead Plots*

Figures 59, 60, and 61 plot the overhead of Deep and Shallow Racket. As before, these plots show the proportion of D-deliverable configurations for values of D between 1x and 20x.

Shallow types lead to a huge improvement, from over 20x down to 8x or lower, in nine benchmarks. With deep types, these benchmarks suffer high overhead due to eager and wrapped checks. The wrapper-free transient semantics removes the issue. Shallow improves on a few other benchmarks and it does equally-well on almost all the rest. The one exception is morsecode, which fares better with deep types. Three characteristics account for the discrepancy: morsecode boundaries create few wrappers; the transient laziness does not end up saving many checks; and the overhead of transient checks ends up slowing down large chunks of typed code. Overall, Shallow Racket lives up to its promise of better mixed-typed performance.

5.4.3 *Exact Runtime Plots*

Figures 62, 63, and 64 offer a different perspective on deep and shallow types. These exact runtime plots show how performance changes as the number of type annotations in a benchmark increases. The left-most column of each plot has one dot for each fully-untyped running time. The right-most columns plot the fully-typed running times, and

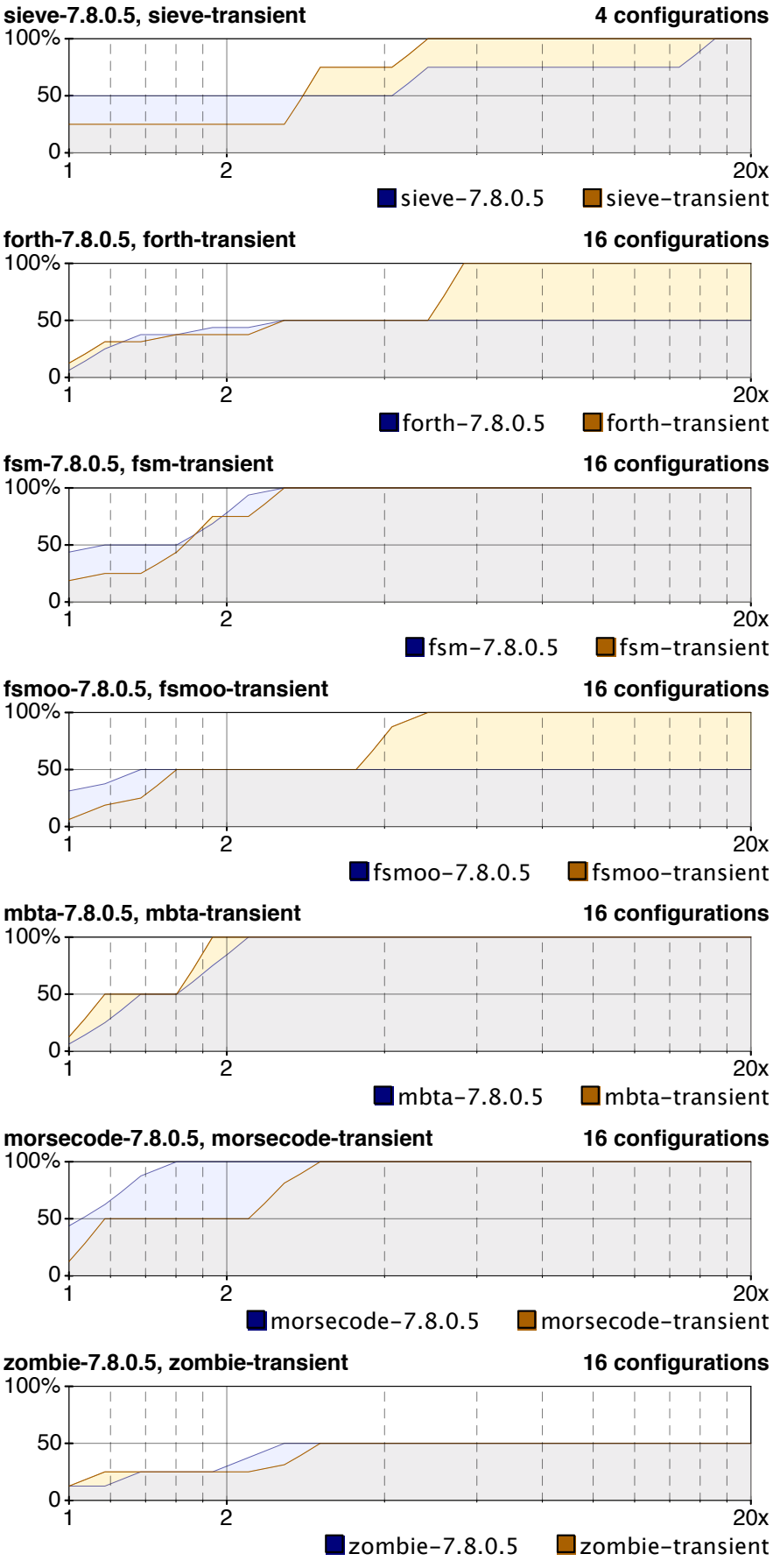


Figure 59: Deep vs. Shallow (1/3).

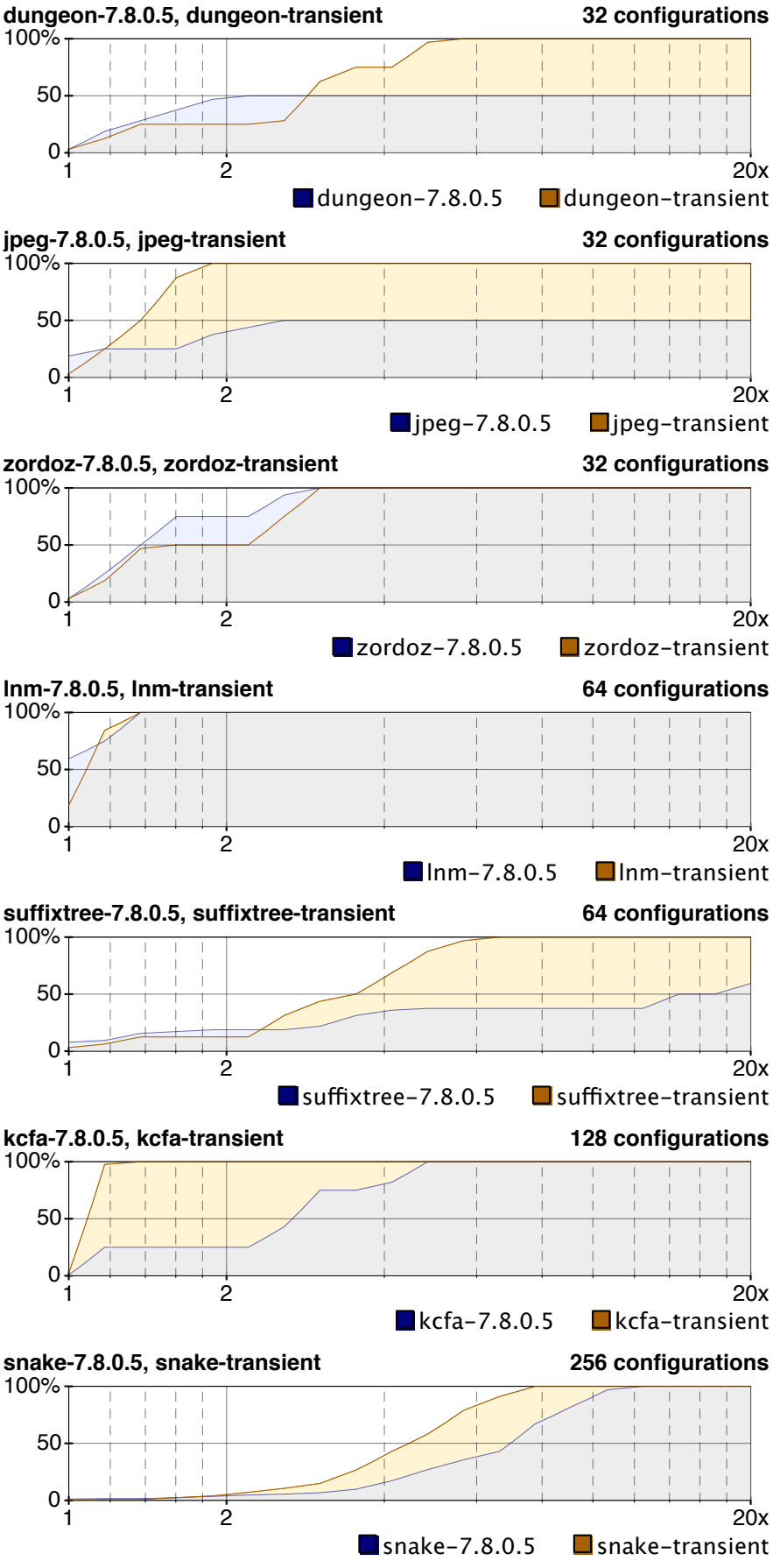


Figure 60: Deep vs. Shallow (2/3).

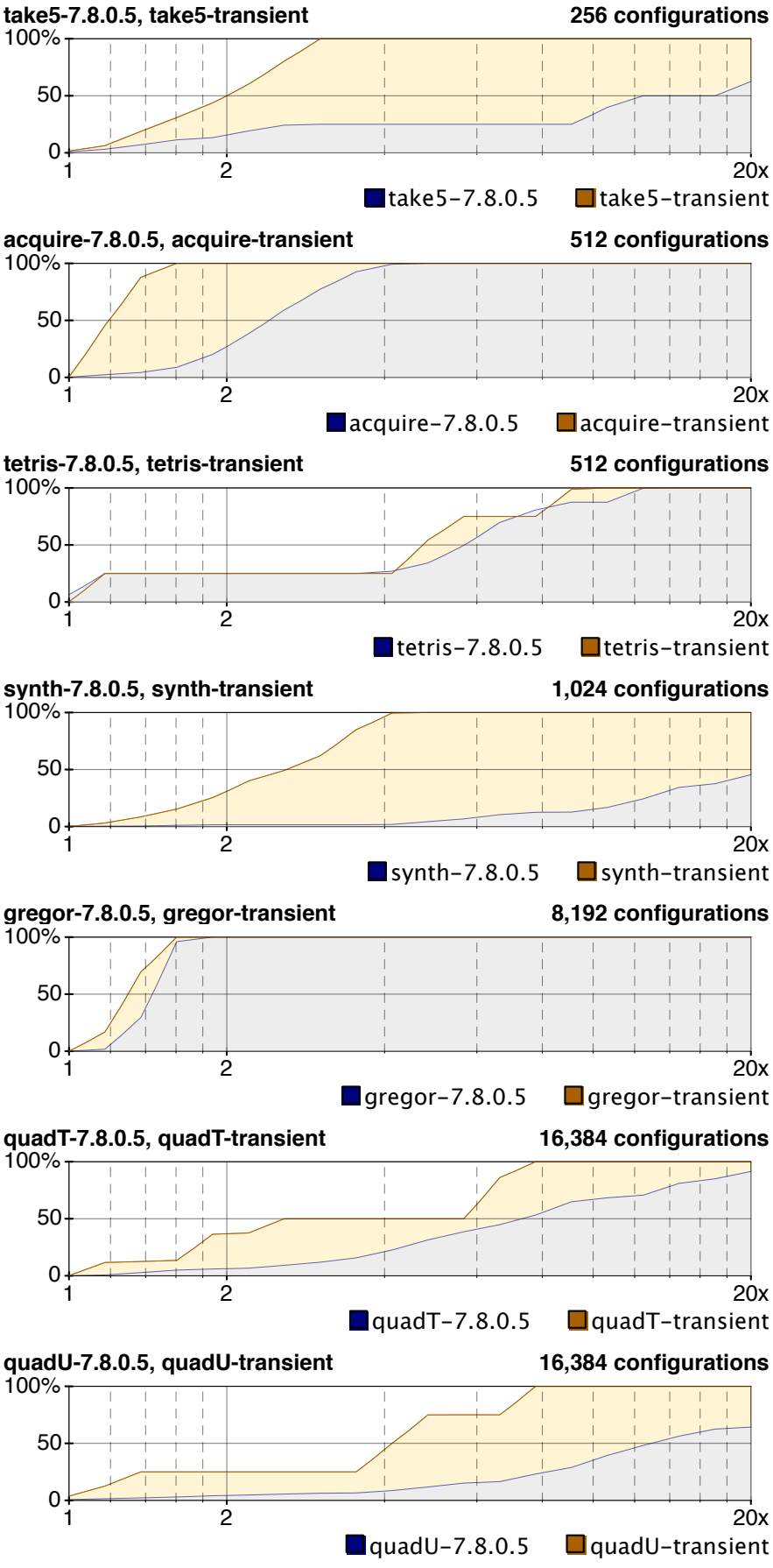


Figure 61: Deep vs. Shallow (3/3).

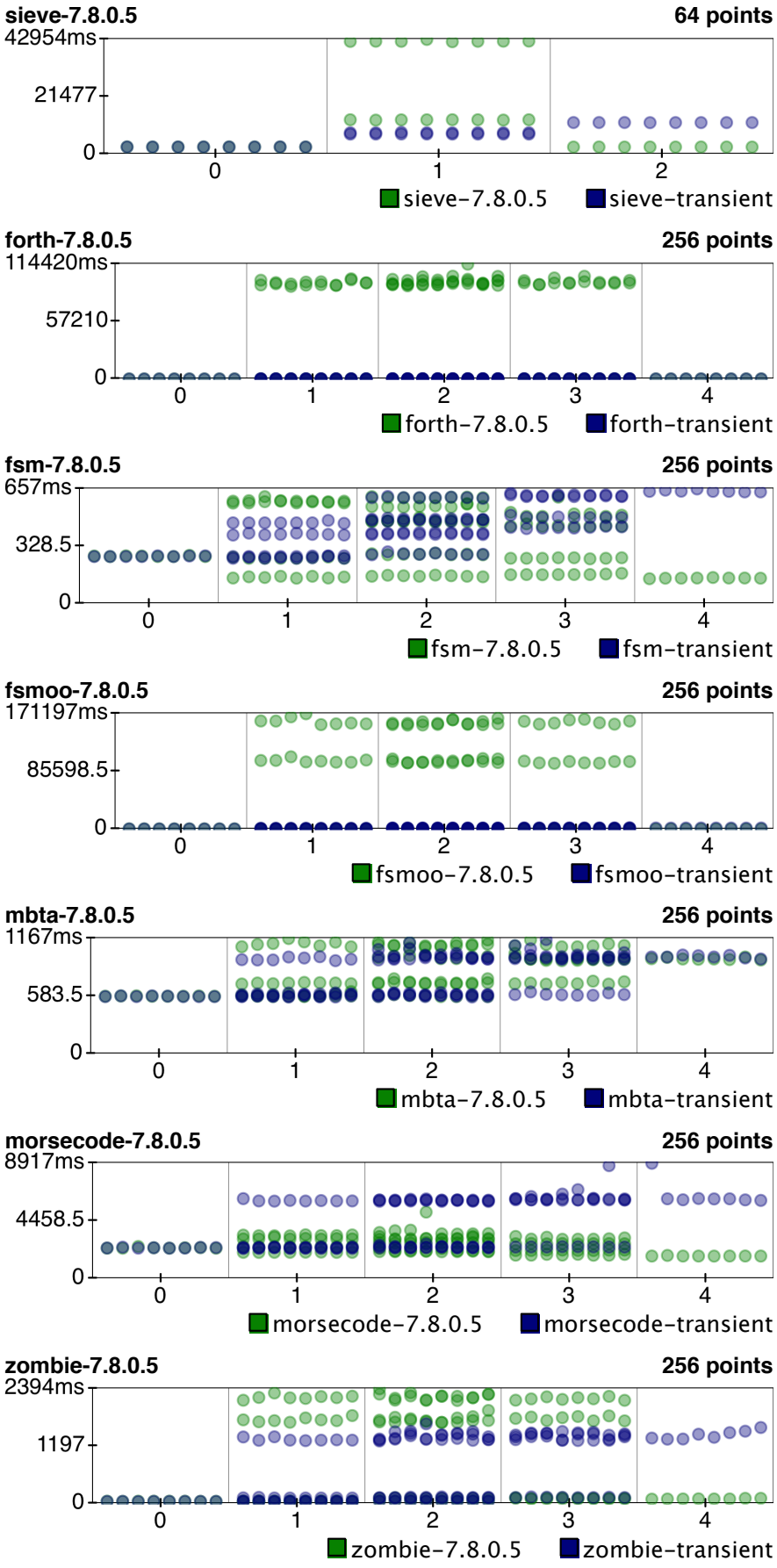


Figure 62: Exact Deep vs Shallow (1/3).

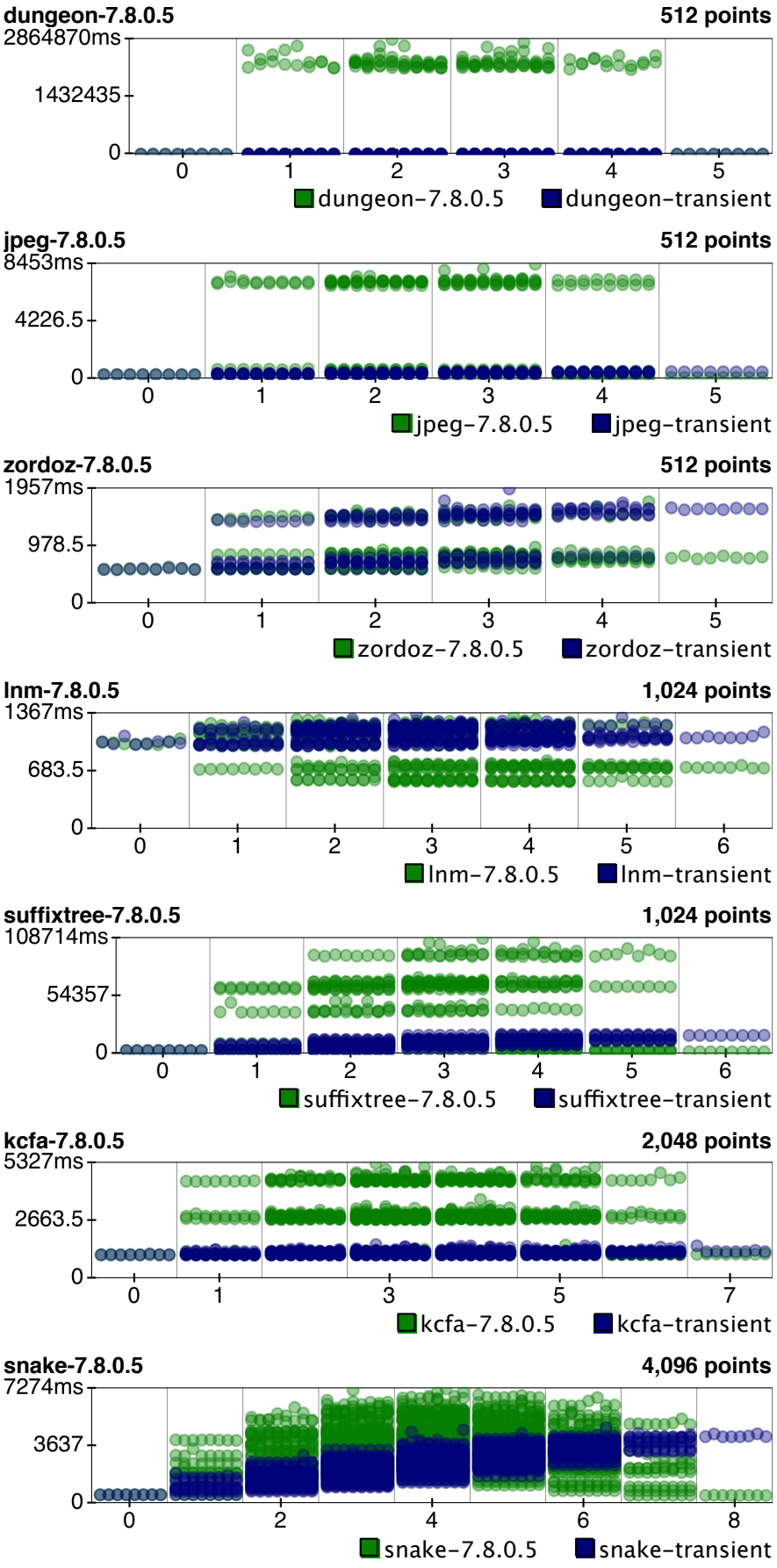


Figure 63: Exact Deep vs Shallow (2/3).

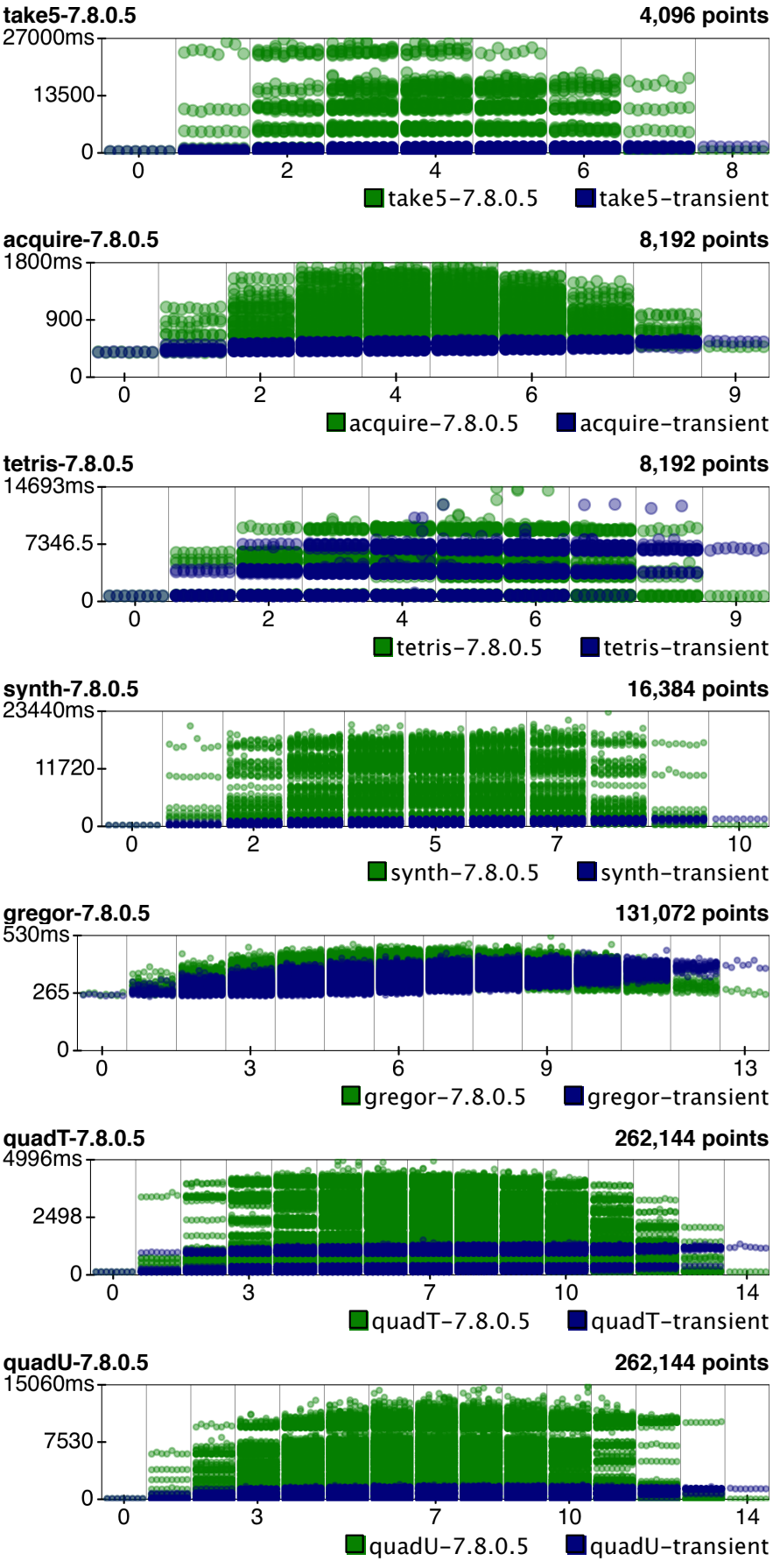


Figure 64: Exact Deep vs Shallow (3/3).

Benchmark	s.blame/untyped	shallow/untyped	deep worst-case
sieve	out of memory	4.36	15.67
forth	41.54	5.21	4010.71
fsm	timeout (>2210)	2.38	2.37
fsmoo	294.24	4.28	451.07
mbta	40.71	1.69	1.92
morsecode	timeout (>251)	2.72	1.47
zombie	563.53	31.07	54.62
dungeon	84.68	4.97	14573.66
jpeg	45.92	1.66	23.16
zordoz	192.54	2.73	2.72
lnm	29.19	1.06	1.11
suffixtree	timeout (>189)	5.51	31.17
kcfa	27.76	1.16	4.43
snake	timeout (>1073)	7.67	11.84
take5	50.77	2.97	32.17
acquire	44.79	1.40	4.15
tetris	timeout (>723)	8.28	11.71
synth	timeout (>1436)	4.07	49.12
gregor	30.71	1.53	1.63
quadT	108.50	7.22	27.10
quadU	561.73	7.14	59.66

Figure 65: Performance ratios for Shallow Racket with blame and Shallow without blame, and also the worst-case of Deep types. The Shallow columns are for the fully-typed configuration; the Deep column uses the slowest configuration. The blame experiments ran on a dedicated Linux machine with 16GB RAM for at most 10 minutes.

columns in between have data for every point at the same level of the performance lattice.

In Deep Racket, mixing typed and untyped code can lead to significant overhead. Points in the middle columns are for mixed configurations, and can have high cost; zombie in particular slows down in the middle. Points on the right columns, however, do not suffer. After critical boundaries are typed, performance is often excellent.

In Shallow Racket, the trend is simple: adding types slows code down. There is a linear, upward trend in every benchmark. As the overhead plots anticipate, the linear cost is typically much lower than the extremes of deep types.

5.4.4 *Blame Performance*

Figure 65 evaluates the overhead of Shallow Racket with blame enabled. The second column of this table measures the overhead of blame on the fully-typed configuration. For comparison, the third column lists the overhead of the same configuration without blame, and the fourth column lists the absolute worst-case of deep types. This table reports only the fully-typed configuration for shallow because this configuration contains the greatest number blame-map updates. Configurations with fewer typed modules have syntactically fewer locations that must touch the global map.

The data shows that blame adds tremendous overhead to Shallow Racket. Six benchmarks fail to terminate within a generous 10-minute limit. One benchmark, sieve, ends with an OS-level memory error after consuming a huge chunk of a 16GB RAM pool. The rest run far slower than shallow without blame.

Surprisingly, the fourth column shows that shallow blame costs more than the worst case of Deep types in 18 benchmarks. Shallow blame slows down every operation by a small factor and allocates a small amount of memory for every value. These small costs add up, even in our relatively short-running benchmarks. Deep is slowest only in benchmarks that frequently send higher-order values across boundaries; collapsible contracts may resolve these issues [29].

Our blame results are far less optimistic than the early report in Vitousek et al. [115], which found an average slowdown on 2.5x and worst-case slowdown of 5.4x on fully-typed configurations. For Shallow Racket benchmarks that terminate, the average slowdown from blame is 32.04x and the worst-case is 78.67x. These different statistics are due to two factors that let Reticulated insert fewer checks: the chosen benchmarks and gradual type inference.

Regarding benchmarks, Vitousek et al. [115] use small programs from the `pyperformance` suite. Three of the twelve benchmarks focus on numeric computations; since the blame map does not track primitive values, adding blame adds little overhead. Four others have since been retired from the Python suite because they are too small, unrealistic, and unstable (`pyperformance.readthedocs.io/changelog.html`). Among the remaining benchmarks, the overhead of blame appears to increase with the size of the program. Larger Reticulated benchmarks should run on par with Shallow Racket. For example, a Reticulated variant of the sieve benchmark runs in about 40 seconds without blame and times out after 10 minutes with blame enabled.

The type inference issue is subtle. Reticulated frequently infers the dynamic type for local variables. Doing so is type-sound and lets Reticulated skip many runtime checks and blame-map updates; however, the programmer gets less precise type and blame information.

For example, the following Python snippet creates a list of numbers, mutates the list, and reads the first element.

```
def set(xs):
    xs[0] = "X"
    return

def main():
    nums = list(range(3))
    set(nums)
    return nums[0] < 1

main()
# TypeError: unorderable types: str() < int()
```

Reticulated infers the dynamic type for the list `nums` and does not check that `nums[0]` returns a number. Running this program leads to a Python exception about comparing strings and integers. For the benchmarks, the lack of checks leads to a faster running time, especially in programs that incrementally update local variables in a loop. If updates lead to the dynamic type, then run-time operations are free of shape checks.

6 DEEP AND SHALLOW, COMBINED

This chapter validates the central point of my thesis: that deep and shallow types can be combined in a companion language for Racket, and the combination is an improvement over either one alone. First, I prove that deep types via natural and shallow types via transient can coexist in a formal model. The two semantics can interoperate without changing the formal properties of either one (chapter 6.1). Second, I report challenges that arose combining Deep Racket and Shallow Racket in a single implementation (chapter 6.2). Overall, the combined implementation has clear benefits (chapter 6.3). Programmers are better off with a choice of deep guarantees and transient performance. Combining the two semantics in one program can further improve performance. And, surprisingly, the addition of shallow types can express programs that Deep Racket currently cannot.

A downside of the combination is that natural and transient cannot easily share the results of their type checks. The reason is simple: transient as-is lacks a way of learning from past checks. Chapter 7.3 explains the synergy challenge in terms of the model and outlines implementation techniques that may get around the issue.

6.1 MODEL AND PROPERTIES

The model combines deep-typed code, shallow-typed code, and untyped code in one surface language. Each of these three disciplines is recognized by a surface-typing judgment and comes with a complier. The three compilers translate well-typed code to a common evaluation syntax that has one untyped semantics.

Although the three varieties of surface code give rise to six kinds of interactions, the model keeps these interactions under control with only three kinds of run-time boundaries (figure 66). A *wrap* boundary inserts a higher-order check to support deep types. A *scan* boundary validates a top-level shape for shallow code. Lastly, a *noop* boundary does nothing. Chapter 6.1.8 proves that these checks are strong enough to realize shallow types that satisfy shape-soundness and deep types that satisfy complete monitoring.

6.1.1 *Syntax*

The surface syntax begins with simple expressions and adds module boundaries to enable a three-way interpretation. The simple expressions are function application ($\mathsf{app}\, s\, s$), primitive operation appli-

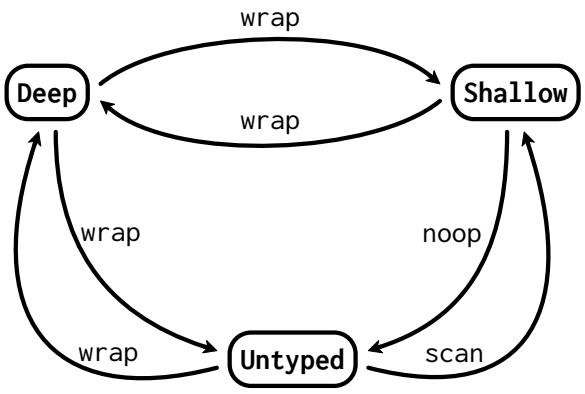


Figure 66: Deep, Shallow, and untyped interactions.

$$
\begin{array}{rcl}
s & = & x \mid i \mid \langle s, s \rangle \mid \\
& & \lambda x.\, s \mid \lambda(x : \tau).\, s \mid \lambda(x : \lfloor \tau \rfloor).\, s \mid \\
& & \mathit{unop}\, s \mid \mathit{binop}\, s\, s \mid \mathsf{app}\, s\, s \mid \\
& & \mathsf{module}\, L\, s \\
\tau & = & \mathsf{Nat} \mid \mathsf{Int} \mid \tau \times \tau \mid \tau \Rightarrow \tau \\
L & = & \mathcal{D} \mid \mathcal{S} \mid \mathcal{U} \\
\mathcal{X} & = & \tau \mid \lfloor \tau \rfloor \mid \mathcal{U}
\end{array}
$$

Figure 67: Surface syntax

cation (*unop* s, *binop* $s\,s$), variables (x), integers (i), pairs ($\langle s, s \rangle$), and functions. Functions come in three flavors: an untyped function has no type annotation ($\lambda x.\, s$), a deep-typed function has a type annotation ($\lambda(x : \tau).\, s$), and a shallow-typed function has an underlined type annotation ($\lambda(x : \lfloor \tau \rfloor).\, s$). The underline mark simplifies proofs, and serves as a hint to readers that only the top-level shape of this type is guaranteed at run-time. It is *not* a meta-function. Types (τ) express natural numbers (Nat), integers (Int), pairs ($\tau \times \tau$), and functions ($\tau \Rightarrow \tau$).

Module-boundary expressions declare the intent of the code within them. For example, the term module $\mathcal{D}\ s_0$ asks for deep types in expression s_0 by default. If another boundary appears within s_0, then its language flag ($\mathcal{D}$, $\mathcal{S}$, or $\mathcal{U}$) sets a new default.

Note that module boundaries are similar to the boundary expressions from chapter 4. Instead of adding a third boundary term to split the old stat boundaries into deep and shallow versions, the present model uses one parameterized term. Both the old and new boundary terms correspond to module boundaries in a realistic language.

6.1.2 *Surface Typing*

In principle, the surface language comes with three typing judgments to recognize deep, shallow, and untyped code. These judgments are mutually recursive at module-boundary terms. To keep things simple, however, figure 68 presents one judgment ($\Gamma \vdash_s e : \mathcal{X}$) that supports three possible conclusions. A conclusion ($\mathcal{X}$) is one of: the unitype $\mathcal{U}$ of untyped code, a type τ for deep-typed code, or a decorated type $\lfloor \tau \rfloor$ for shallow code. The notation is again a hint. A decorated type is equal to a normal type during static type checking, but makes a weaker statement about program behavior.

The typing rules are relatively simple, but declarative. The rules for modules, for example, give no hint about how to find a type conclusion that fits the rest of the program. A second notably aspect is that one module may contain another with the same language flag.

Figure 69 defines a subtyping judgment ($<:$) and a type-assignment for primitive operations (Δ). These are both standard.

6.1.3 *Evaluation Syntax*

The evaluation syntax removes the declarative parts of the surface syntax and adds tools for enforcing types. First to go are the module boundary expressions, which express a desire for a style of type enforcement. Instead, the evaluation syntax has three kinds of run-time check expression: a wrap boundary fully enforces a type, perhaps with a guard wrapper ($\mathbb{G}\ \tau\ v$); a scan boundary checks a type-shape (σ), and a noop boundary checks nothing. Second, the shallow-typed

$$\frac{(x_0 : \mathcal{U}) \in \Gamma}{\Gamma \vdash_s x_0 : \mathcal{U}} \qquad \frac{(x_0 : \tau_0) \in \Gamma}{\Gamma \vdash_s x_0 : \tau_0} \qquad \frac{(x_0 : \lfloor \tau_0 \rfloor) \in \Gamma}{\Gamma \vdash_s x_0 : \lfloor \tau_0 \rfloor} \qquad \frac{}{\Gamma \vdash_s i_0 : \mathcal{U}}$$

$$\frac{}{\Gamma \vdash_s n_0 : \mathsf{Nat}} \qquad \frac{}{\Gamma \vdash_s n_0 : \lfloor \mathsf{Nat} \rfloor} \qquad \frac{}{\Gamma \vdash_s i_0 : \mathsf{Int}} \qquad \frac{}{\Gamma \vdash_s i_0 : \lfloor \mathsf{Int} \rfloor}$$

$$\frac{\Gamma \vdash_s s_0 : \mathcal{U} \qquad \Gamma \vdash_s s_1 : \mathcal{U}}{\Gamma \vdash_s \langle s_0, s_1 \rangle : \mathcal{U}} \qquad \frac{(x_0 : \mathcal{U}), \Gamma \vdash_s e_0 : \mathcal{U}}{\Gamma \vdash_s \lambda x_0.\, e_0 : \mathcal{U}}$$

$$\frac{(x_0 : \tau_0), \Gamma \vdash_s e_0 : \tau_1}{\Gamma \vdash_s \lambda (x_0 : \tau_0).\, e_0 : \tau_0 \Rightarrow \tau_1} \qquad \frac{(x_0 : \lfloor \tau_0 \rfloor), \Gamma \vdash_s e_0 : \lfloor \tau_1 \rfloor}{\Gamma \vdash_s \lambda (x_0 : \tau_0).\, e_0 : \lfloor \tau_0 \Rightarrow \tau_1 \rfloor}$$

$$\frac{\Gamma \vdash_s e_0 : \tau_0 \qquad \Delta(\mathit{unop}, \tau_0) = \tau_1}{\Gamma \vdash_s \mathit{unop}\, e_0 : \tau_1} \qquad \frac{\Gamma \vdash_s e_0 : \lfloor \tau_0 \rfloor \qquad \Gamma \vdash_s e_1 : \lfloor \tau_1 \rfloor \qquad \Delta(\mathit{binop}, \tau_0, \tau_1) = \tau_2}{\Gamma \vdash_s \mathit{binop}\, e_0\, e_1 : \lfloor \tau_2 \rfloor}$$

$$\frac{\Gamma \vdash_s e_0 : \lfloor \tau_0 \Rightarrow \tau_1 \rfloor \qquad \Gamma \vdash_s e_1 : \lfloor \tau_0 \rfloor}{\Gamma \vdash_s \mathsf{app}\{e_0\}\, e_1 \ : \lfloor \tau_1 \rfloor} \qquad \frac{\Gamma \vdash_s e_0 : \lfloor \tau_0 \rfloor \qquad \tau_0 <: \tau_1}{\Gamma \vdash_s e_0 : \lfloor \tau_1 \rfloor}$$

$$\frac{\Gamma \vdash_s e_0 : \mathcal{U}}{\Gamma \vdash_s \mathsf{module}\, \mathcal{U}\, e_0 : \mathcal{U}} \qquad \frac{\Gamma \vdash_s e_0 : \tau_0}{\Gamma \vdash_s \mathsf{module}\, \mathcal{D}\, e_0 : \mathcal{U}}$$

$$\frac{\Gamma \vdash_s e_0 : \lfloor \tau_0 \rfloor}{\Gamma \vdash_s \mathsf{module}\, \mathcal{S}\, e_0 : \mathcal{U}} \qquad \frac{\Gamma \vdash_s e_0 : \mathcal{U}}{\Gamma \vdash_s \mathsf{module}\, \mathcal{U}\, e_0 : \tau_0}$$

$$\frac{\Gamma \vdash_s e_0 : \tau_0}{\Gamma \vdash_s \mathsf{module}\, \mathcal{D}\, e_0 : \tau_0} \qquad \frac{\Gamma \vdash_s e_0 : \lfloor \tau_0 \rfloor}{\Gamma \vdash_s \mathsf{module}\, \mathcal{S}\, e_0 : \tau_0}$$

$$\frac{\Gamma \vdash_s e_0 : \mathcal{U}}{\Gamma \vdash_s \mathsf{module}\, \mathcal{U}\, e_0 : \lfloor \tau_0 \rfloor} \qquad \frac{\Gamma \vdash_s e_0 : \tau_0}{\Gamma \vdash_s \mathsf{module}\, \mathcal{D}\, e_0 : \lfloor \tau_0 \rfloor}$$

$$\frac{\Gamma \vdash_s e_0 : \lfloor \tau_0 \rfloor}{\Gamma \vdash_s \mathsf{module}\, \mathcal{S}\, e_0 : \lfloor \tau_0 \rfloor}$$

Figure 68: Surface typing judgment (selected rules, others in A.3)

$\boxed{\tau <: \tau}$

$$\frac{}{\mathsf{Nat} <: \mathsf{Int}} \qquad \frac{\tau_0 <: \tau_2 \qquad \tau_1 <: \tau_3}{\tau_0 \times \tau_1 <: \tau_2 \times \tau_3} \qquad \frac{\tau_2 <: \tau_0 \qquad \tau_1 <: \tau_3}{\tau_0 \Rightarrow \tau_1 <: \tau_2 \Rightarrow \tau_3}$$

$\boxed{\Delta : \mathit{unop} \times \tau \longrightarrow \tau}$

$$\begin{array}{lcl} \Delta(\mathsf{fst}, \tau_0 \times \tau_1) & = & \tau_0 \\ \Delta(\mathsf{snd}, \tau_0 \times \tau_1) & = & \tau_1 \end{array}$$

$\boxed{\Delta : \mathit{binop} \times \tau \times \tau \longrightarrow \tau}$

$$\begin{array}{lcl} \Delta(\mathsf{plus}, \mathsf{Nat}, \mathsf{Nat}) & = & \mathsf{Nat} \\ \Delta(\mathsf{plus}, \mathsf{Int}, \mathsf{Int}) & = & \mathsf{Int} \\ \Delta(\mathsf{quotient}, \mathsf{Nat}, \mathsf{Nat}) & = & \mathsf{Nat} \\ \Delta(\mathsf{quotient}, \mathsf{Int}, \mathsf{Int}) & = & \mathsf{Int} \end{array}$$

Figure 69: Subtyping and primitive op. typing

$$\begin{array}{lcl} e & = & x \mid v \mid \langle e, e \rangle \mid \mathit{unop}\, e \mid \mathit{binop}\, e\, e \mid \mathsf{app}\, e\, e \mid \mathsf{Err} \mid \\ & & \mathsf{wrap}\, \tau\ e \mid \mathsf{scan}\, \sigma\ e \mid \mathsf{noop}\ \ e \\ v & = & i \mid \langle v, v \rangle \mid \lambda x.\, e \mid \lambda (x : \tau).\, e \mid \lambda (x : \sigma).\, e \mid \mathbb{G}\, \tau\ v \\ \sigma & = & \mathsf{Nat} \mid \mathsf{Int} \mid \mathsf{Pair} \mid \mathsf{Fun} \mid \mathsf{Any} \\ \mathsf{Err} & = & \mathsf{WrapErr} \mid \mathsf{ScanErr} \mid \mathsf{DivZeroErr} \mid \mathsf{TagErr} \\ E & = & \bullet \mid \mathit{unop}\, E \mid \mathit{binop}\, E\, e \mid \mathit{binop}\, v\, E \mid \mathsf{app}\, E\, e \mid \mathsf{app}\, v\, E \mid \\ & & \mathsf{noop}\ \ E \mid \mathsf{scan}\, \sigma\ E \mid \mathsf{wrap}\, \tau\ E \end{array}$$

Figure 70: Evaluation Syntax

functions from the surface syntax ($\lambda (x : \lfloor \tau \rfloor).\, s$) are replaced with shape-annotated functions ($\lambda (x : \sigma).\, e$). Type-shapes ($\sigma$) express the outermost constructor of a type; the weakened function annotation reflects a weakened run-time guarantee.

The evaluation syntax also includes values (v), errors (Err), and evaluation contexts (E). Running a program may produce a value or an error. For the most part, the different errors come from boundaries. A WrapErr arises when a wrap boundary receives invalid input, a ScanErr comes from a failed scan, and a DivZeroErr occurs when a primitive operation rejects its input. The final error, TagErr, is the result of a malformed term that cannot reduce further. Such errors can easily occur in untyped code without any boundaries; for instance, the application of a number (app 2 4) signals a tag error. Reduction in typed code, whether deep or shallow, should never raise a tag error.

6.1.4 *Evaluation Typing*

The evaluation syntax comes with three typing judgments that describe the run-time invariants of deep, shallow, and untyped code. The deep typing judgment ($\vdash_{\mathcal{D}}$) validates full types. The shallow

$$\frac{(x_0 : \tau_0) \in \Gamma}{\Gamma \vdash_{\mathcal{D}} x_0 : \tau_0} \qquad \frac{}{\Gamma \vdash_{\mathcal{D}} n_0 : \mathsf{Nat}} \qquad \frac{}{\Gamma \vdash_{\mathcal{D}} i_0 : \mathsf{Int}}$$

$$\frac{\Gamma \vdash_{\mathcal{D}} e_0 : \tau_0 \qquad \Gamma \vdash_{\mathcal{D}} e_1 : \tau_1}{\Gamma \vdash_{\mathcal{D}} \langle e_0, e_1 \rangle : \tau_0 \times \tau_1} \qquad \frac{(x_0 : \tau_0), \Gamma \vdash_{\mathcal{D}} e_0 : \tau_1}{\Gamma \vdash_{\mathcal{D}} \lambda(x_0 : \tau_0).\, e_0 : \tau_0 \Rightarrow \tau_1}$$

$$\frac{\Gamma \vdash_{\mathcal{U}} v_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{D}} \mathbb{G}\, \tau_0\ v_0 : \tau_0} \qquad \frac{\Gamma \vdash_{\mathcal{S}} v_0 : \sigma_0}{\Gamma \vdash_{\mathcal{D}} \mathbb{G}\, \tau_0\ v_0 : \tau_0} \qquad \frac{\Gamma \vdash_{\mathcal{D}} e_0 : \tau_0 \quad \Delta(\mathit{unop}, \tau_0) = \tau_1}{\Gamma \vdash_{\mathcal{D}} \mathit{unop}\, e_0 : \tau_1}$$

$$\frac{\Gamma \vdash_{\mathcal{D}} e_0 : \tau_0 \qquad \Gamma \vdash_{\mathcal{D}} e_1 : \tau_1 \quad \Delta(\mathit{binop}, \tau_0, \tau_1) = \tau_2}{\Gamma \vdash_{\mathcal{D}} \mathit{binop}\, e_0\, e_1 : \tau_2} \qquad \frac{\Gamma \vdash_{\mathcal{D}} e_0 : \tau_0 \Rightarrow \tau_1 \qquad \Gamma \vdash_{\mathcal{D}} e_1 : \tau_0}{\Gamma \vdash_{\mathcal{D}} \mathsf{app}\, e_0\, e_1 : \tau_1}$$

$$\frac{\Gamma \vdash_{\mathcal{D}} e_0 : \tau_0}{\Gamma \vdash_{\mathcal{D}} \mathsf{noop}\ e_0 : \tau_0} \qquad \frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{D}} \mathsf{wrap}\, \tau_0\, e_0 : \tau_0} \qquad \frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_0}{\Gamma \vdash_{\mathcal{D}} \mathsf{wrap}\, \tau_0\, e_0 : \tau_0}$$

$$\frac{\Gamma \vdash_{\mathcal{D}} e_0 : \tau_0 \qquad \tau_0 <: \tau_1}{\Gamma \vdash_{\mathcal{D}} e_0 : \tau_1} \qquad \frac{}{\Gamma \vdash_{\mathcal{D}} \mathsf{Err} : \tau_0}$$

Figure 71: Deep typing judgment

judgment ($\vdash_{\mathcal{S}}$) checks top-level shapes. In this judgment, elimination forms have a catch-all shape (Any) because they can produce any value at run-time; these must appear within a scan expression to guarantee a non-trivial shape. Lastly, the untyped judgment ($\vdash_{\mathcal{U}}$) guarantees no free variables.

6.1.5 *Compilation*

A compilation pass links the surface and evaluation syntaxes. Since the goal of compilation is to insert enough run-time checks to give a well-typed result, the compiler is effectively a *completion* pass that fills in details missing from the surface term [52]. The basic goal is to translate module boundaries to appropriate run-time checks, but other terms may require checks as well. Formally, the goal is to map all well-typed surface expressions to well-typed evaluation expressions (lemma 6.1.4). The completion rules shown in figures 74 and 75 meet this goal via different strategies for each kind of code:

$$\frac{(x_0 : \sigma_0) \in \Gamma}{\Gamma \vdash_{\mathcal{S}} x_0 : \sigma_0} \qquad \frac{}{\Gamma \vdash_{\mathcal{S}} n_0 : \mathsf{Nat}} \qquad \frac{}{\Gamma \vdash_{\mathcal{S}} i_0 : \mathsf{Int}}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_0 \qquad \Gamma \vdash_{\mathcal{S}} e_1 : \sigma_1}{\Gamma \vdash_{\mathcal{S}} \langle e_0, e_1 \rangle : \mathsf{Pair}} \qquad \frac{(x_0 : \mathcal{U}), \Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{S}} \lambda x_0.\, e_0 : \mathsf{Fun}}$$

$$\frac{(x_0 : \sigma_0), \Gamma \vdash_{\mathcal{S}} e_0 : \sigma_1}{\Gamma \vdash_{\mathcal{S}} \lambda (x_0 : \sigma_0).\, e_0 : \mathsf{Fun}} \qquad \frac{\Gamma \vdash_{\mathcal{D}} v_0 : \tau_0}{\Gamma \vdash_{\mathcal{S}} \mathbb{G}\, \tau_0\, v_0 : \mathsf{Fun}} \qquad \frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_0}{\Gamma \vdash_{\mathcal{S}} \mathit{unop}\, e_0 : \mathsf{Any}}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_0 \qquad \Gamma \vdash_{\mathcal{S}} e_1 : \sigma_1}{\Gamma \vdash_{\mathcal{S}} \mathit{binop}\, e_0\, e_1 : \mathsf{Any}} \qquad \frac{\Gamma \vdash_{\mathcal{S}} e_0 : \mathsf{Fun} \qquad \Gamma \vdash_{\mathcal{S}} e_1 : \sigma_0}{\Gamma \vdash_{\mathcal{S}} \mathsf{app}\, e_0\, e_1 : \mathsf{Any}}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_0}{\Gamma \vdash_{\mathcal{S}} \mathsf{noop}\ e_0 : \sigma_0} \qquad \frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{S}} \mathsf{noop}\ e_0 : \mathsf{Any}} \qquad \frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{S}} \mathsf{scan}\, \sigma_0\, e_0 : \sigma_0}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_1}{\Gamma \vdash_{\mathcal{S}} \mathsf{scan}\, \sigma_0\, e_0 : \sigma_0} \qquad \frac{\Gamma \vdash_{\mathcal{D}} e_0 : \tau_0 \qquad \mathit{shape}\,(\tau_0) = \sigma_0}{\Gamma \vdash_{\mathcal{S}} \mathsf{wrap}\, \tau_0\, e_0 : \sigma_0}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_0 \qquad \sigma_0 <: \sigma_1}{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_1} \qquad \frac{}{\Gamma \vdash_{\mathcal{S}} \mathsf{Err} : \sigma_0}$$

$\boxed{\sigma <: \sigma}$

$$\frac{}{\mathsf{Nat} <: \mathsf{Int}} \qquad \frac{}{\sigma_0 <: \mathsf{Any}}$$

$\boxed{\mathit{shape} : \tau \longrightarrow \sigma}$

$$\begin{aligned}
\mathit{shape}\,(\mathsf{Nat}) &= \mathsf{Nat} \\
\mathit{shape}\,(\mathsf{Int}) &= \mathsf{Int} \\
\mathit{shape}\,(\tau_0 \times \tau_1) &= \mathsf{Pair} \\
\mathit{shape}\,(\tau_0 \Rightarrow \tau_1) &= \mathsf{Fun}
\end{aligned}$$

Figure 72: Shallow typing judgment, subtyping, and shape map

$$\frac{(x_0 : \mathcal{U}) \in \Gamma}{\Gamma \vdash_{\mathcal{U}} x_0 : \mathcal{U}} \qquad \frac{}{\Gamma \vdash_{\mathcal{U}} i_0 : \mathcal{U}} \qquad \frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U} \qquad \Gamma \vdash_{\mathcal{U}} e_1 : \mathcal{U}}{\Gamma \vdash_{\mathcal{U}} \langle e_0, e_1 \rangle : \mathcal{U}}$$

$$\frac{(x_0 : \mathcal{U}), \Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{U}} \lambda x_0 . e_0 : \mathcal{U}} \qquad \frac{(x_0 : \sigma_0), \Gamma \vdash_{\mathcal{S}} e_0 : \sigma_1}{\Gamma \vdash_{\mathcal{U}} \lambda (x_0 : \sigma_0) . e_0 : \mathcal{U}}$$

$$\frac{\Gamma \vdash_{\mathcal{D}} v_0 : \tau_0}{\Gamma \vdash_{\mathcal{U}} \mathbb{G}\, \tau_0\, v_0 : \mathcal{U}} \qquad \frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{U}} \mathit{unop}\, e_0 : \mathcal{U}}$$

$$\frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U} \qquad \Gamma \vdash_{\mathcal{U}} e_1 : \mathcal{U}}{\Gamma \vdash_{\mathcal{U}} \mathit{binop}\, e_0\, e_1 : \mathcal{U}} \qquad \frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U} \qquad \Gamma \vdash_{\mathcal{U}} e_1 : \mathcal{U}}{\Gamma \vdash_{\mathcal{U}} \mathsf{app}\, e_0\, e_1 : \mathcal{U}}$$

$$\frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{U}} \mathsf{noop}\ \ e_0 : \mathcal{U}} \qquad \frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_0}{\Gamma \vdash_{\mathcal{U}} \mathsf{noop}\ \ e_0 : \mathcal{U}} \qquad \frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{U}} \mathsf{scan}\, \sigma_0\, e_0 : \mathcal{U}}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_1}{\Gamma \vdash_{\mathcal{U}} \mathsf{scan}\, \sigma_0\, e_0 : \mathcal{U}} \qquad \frac{\Gamma \vdash_{\mathcal{D}} e_0 : \tau_0}{\Gamma \vdash_{\mathcal{U}} \mathsf{wrap}\, \tau_0\, e_0 : \mathcal{U}} \qquad \frac{}{\Gamma \vdash_{\mathcal{U}} \mathsf{Err} : \mathcal{U}}$$

Figure 73: Untyped typing judgment (dynamic typing)

- In deep-typed code, completion inserts wrap expressions at the module boundaries to less-typed code. Other deep expressions have no checks.
- In shallow code, completion scans incoming untyped code and the result of every elimination form.
- In untyped code, completion adds no run-time checks. At the boundaries to deep and shallow code, however, the above strategies call for a wrap or scan check.

Figure 74 in particular shows how surface functions translate to evaluation syntax functions and how applications translate. For deep and untyped code, the completion of an application is simply the completion of its subexpressions. For shallow code, this elimination form requires a scan check to validate the result. Other elimination forms have similar completions.

The completion of a shallow function is deceptively simple. In a realistic language, such functions would translate to an un-annotated function that first scans the shape of its input and then proceeds with the body expression. This model, however, gets an implicit domain check thanks to cooperation from the upcoming semantics. The application of a shallow-typed function always scans the argument before

$$\frac{}{\Gamma \vdash_s x_0 : \mathcal{U} \rightsquigarrow x_0} \qquad \frac{}{\Gamma \vdash_s x_0 : \tau_0 \rightsquigarrow x_0} \qquad \frac{}{\Gamma \vdash_s x_0 : \lfloor \tau_0 \rfloor \rightsquigarrow x_0}$$

$$\frac{(x_0 : \mathcal{U}), \Gamma \vdash_s e_0 : \mathcal{U} \rightsquigarrow e_1}{\Gamma \vdash_s \lambda x_0.\, e_0 : \mathcal{U} \rightsquigarrow \lambda x_0.\, e_1}$$

$$\frac{(x_0 : \tau_0), \Gamma \vdash_s e_0 : \tau_1 \rightsquigarrow e_1}{\Gamma \vdash_s \lambda(x_0 : \tau_0).\, e_0 : \tau_0 \Rightarrow \tau_1 \rightsquigarrow \lambda(x_0 : \tau_0).\, e_1}$$

$$\frac{(x_0 : \lfloor \tau_0 \rfloor), \Gamma \vdash_s e_0 : \lfloor \tau_1 \rfloor \rightsquigarrow e_1 \qquad \mathit{shape}\,(\tau_0) = \sigma_0}{\Gamma \vdash_s \lambda(x_0 : \lfloor \tau_0 \rfloor).\, e_0 : \lfloor \tau_0 \Rightarrow \tau_1 \rfloor \rightsquigarrow \lambda(x_0 : \sigma_0).\, e_1}$$

$$\frac{\Gamma \vdash_s e_0 : \mathcal{U} \rightsquigarrow e_2 \quad \Gamma \vdash_s e_1 : \mathcal{U} \rightsquigarrow e_3}{\Gamma \vdash_s \mathsf{app}\, e_0\, e_1 : \mathcal{U} \rightsquigarrow \mathsf{app}\, e_2\, e_3} \qquad \frac{\Gamma \vdash_s e_0 : \tau_1 \Rightarrow \tau_0 \rightsquigarrow e_2 \quad \Gamma \vdash_s e_1 : \tau_1 \rightsquigarrow e_3}{\Gamma \vdash_s \mathsf{app}\, e_0\, e_1 : \tau_0 \rightsquigarrow \mathsf{app}\, e_2\, e_3}$$

$$\frac{\Gamma \vdash_s e_0 : \lfloor \tau_1 \Rightarrow \tau_0 \rfloor \rightsquigarrow e_2 \quad \Gamma \vdash_s e_1 : \lfloor \tau_1 \rfloor \rightsquigarrow e_3 \qquad \mathit{shape}\,(\tau_0) = \sigma_0}{\Gamma \vdash_s \mathsf{app}\, e_0\, e_1 : \lfloor \tau_0 \rfloor \rightsquigarrow \mathsf{scan}\, \sigma_0\ (\mathsf{app}\, e_2\, e_3)}$$

Figure 74: Surface-to-evaluation completion (selected rules, others in A.3)

substituting into the function body (chapter 6.1.6). This design simplifies the model and proof details regarding substitution, but the lack of an explicit domain check means that the model cannot support a pass that eliminates redundant checks. Fixing this limitation is a top priority for future extensions of the model.

Figure 75 presents the completion rules for module boundaries. Aside from the self-boundaries, the picture in figure 66 is an accurate summary of these rules. Each module represents a channel of communication between a context and the inside of the module. The module declares its type discipline and the context's style is clear from the conclusion of the surface typing judgment. To protect against mis-communications, the side with the stronger type requirements determines the check that a module boundary completes to. Deep always directs, shallow wins over untyped, and the others—with one exception—are clear noops. The exception is for shallow values that exit to untyped code; for integers there is nothing to protect, but functions would seem to need some kind of wrapper to protect their body against untyped input. In fact, these boundaries are safe noops because shallow pre-emptively protects functions as noted above.

$$\frac{\Gamma \vdash_s e_0 : \mathcal{U} \rightsquigarrow e_1}{\Gamma \vdash_s \mathsf{module}\ \mathcal{U}\ e_0 : \mathcal{U} \rightsquigarrow \mathsf{noop}\ e_1}$$

$$\frac{\Gamma \vdash_s e_0 : \tau_0 \rightsquigarrow e_1}{\Gamma \vdash_s \mathsf{module}\ \mathcal{D}\ e_0 : \mathcal{U} \rightsquigarrow \mathsf{wrap}\ \tau_0\ e_1}$$

$$\frac{\Gamma \vdash_s e_0 : \lfloor \tau_0 \rfloor \rightsquigarrow e_1}{\Gamma \vdash_s \mathsf{module}\ \mathcal{S}\ e_0 : \mathcal{U} \rightsquigarrow \mathsf{noop}\ e_1}$$

$$\frac{\Gamma \vdash_s e_0 : \mathcal{U} \rightsquigarrow e_1}{\Gamma \vdash_s \mathsf{module}\ \mathcal{U}\ e_0 : \tau_0 \rightsquigarrow \mathsf{wrap}\ \tau_0\ e_1}$$

$$\frac{\Gamma \vdash_s e_0 : \tau_0 \rightsquigarrow e_1}{\Gamma \vdash_s \mathsf{module}\ \mathcal{D}\ e_0 : \tau_0 \rightsquigarrow \mathsf{noop}\ e_1}$$

$$\frac{\Gamma \vdash_s e_0 : \lfloor \tau_0 \rfloor \rightsquigarrow e_1}{\Gamma \vdash_s \mathsf{module}\ \mathcal{S}\ e_0 : \tau_0 \rightsquigarrow \mathsf{wrap}\ \tau_0\ e_1}$$

$$\frac{\Gamma \vdash_s e_0 : \mathcal{U} \rightsquigarrow e_1}{\Gamma \vdash_s \mathsf{module}\ \mathcal{U}\ e_0 : \lfloor \tau_0 \rfloor \rightsquigarrow \mathsf{scan}\ \sigma_0\ e_1}$$

$$\frac{\Gamma \vdash_s e_0 : \tau_0 \rightsquigarrow e_1}{\Gamma \vdash_s \mathsf{module}\ \mathcal{D}\ e_0 : \lfloor \tau_0 \rfloor \rightsquigarrow \mathsf{wrap}\ \tau_0\ e_1}$$

$$\frac{\Gamma \vdash_s e_0 : \lfloor \tau_0 \rfloor \rightsquigarrow e_1}{\Gamma \vdash_s \mathsf{module}\ \mathcal{S}\ e_0 : \lfloor \tau_0 \rfloor \rightsquigarrow \mathsf{noop}\ e_1}$$

Figure 75: Completion for module boundaries

$$
\begin{array}{ll}
\mathit{unop}\, v_0 & \rhd\ \mathsf{TagErr} \\
\quad \text{if } \delta(\mathit{unop}, v_0) \text{ is undefined} & \\
\mathit{unop}\, v_0 & \rhd\ \delta(\mathit{unop}, v_0) \\
\quad \text{if } \delta(\mathit{unop}, v_0) \text{ is defined} & \\
\mathit{binop}\, v_0\, v_1 & \rhd\ \mathsf{TagErr} \\
\quad \text{if } \delta(\mathit{binop}, v_0, v_1) \text{ is undefined} & \\
\mathit{binop}\, v_0\, v_1 & \rhd\ \delta(\mathit{binop}, v_0, v_1) \\
\quad \text{if } \delta(\mathit{binop}, v_0, v_1) \text{ is defined} & \\
\mathsf{app}\, v_0\, v_1 & \rhd\ \mathsf{TagErr} \\
\quad \text{if } v_0 \notin \lambda x.e \cup \lambda(x:\tau).e \cup \lambda(x:\sigma).e \cup \mathbb{G}\ \tau\ v & \\
\mathsf{app}\, (\lambda x_0.e_0)\, v_0 & \rhd\ e_0[x_0 \leftarrow v_0] \\
\mathsf{app}\, (\lambda(x_0:\tau_0).e_0)\, v_0 & \rhd\ e_0[x_0 \leftarrow v_0] \\
\mathsf{app}\, (\lambda(x_0:\sigma_0).e_0)\, v_0 & \rhd\ \mathsf{ScanErr} \\
\quad \text{if } \neg \mathit{shape\text{-}match}\, (\sigma_0, v_0) & \\
\mathsf{app}\, (\lambda(x_0:\sigma_0).e_0)\, v_0 & \rhd\ e_0[x_0 \leftarrow v_0] \\
\quad \text{if } \mathit{shape\text{-}match}\, (\sigma_0, v_0) & \\
\mathsf{app}\, (\mathbb{G}\ \tau_0 \Rightarrow \tau_1\ v_0)\, v_1 & \rhd\ \mathsf{wrap}\, \tau_1\ (\mathsf{app}\, v_0\, (\mathsf{wrap}\, \tau_0\ v_1)) \\
\mathsf{noop}\ v_0 & \rhd\ v_0 \\
\mathsf{scan}\, \sigma_0\ v_0 & \rhd\ \mathsf{ScanErr} \\
\quad \text{if } \neg \mathit{shape\text{-}match}\, (\sigma_0, v_0) & \\
\mathsf{scan}\, \sigma_0\ v_0 & \rhd\ v_0 \\
\quad \text{if } \mathit{shape\text{-}match}\, (\sigma_0, v_0) & \\
\mathsf{wrap}\, \tau_0\ v_0 & \rhd\ \mathsf{WrapErr} \\
\quad \text{if } \mathit{shape\text{-}match}\, (\mathit{shape}\, (\sigma_0), v_0) & \\
\mathsf{wrap}\, \tau_0 \Rightarrow \tau_1\ v_0 & \rhd\ \mathbb{G}\ \tau_0 \Rightarrow \tau_1\ v_0 \\
\quad \text{if } \mathit{shape\text{-}match}\, (\mathsf{Fun}, v_0) & \\
\mathsf{wrap}\, \tau_0 \times \tau_1\ \langle v_0, v_1 \rangle & \rhd\ \langle \mathsf{wrap}\, \tau_0\ v_0, \mathsf{wrap}\, \tau_1\ v_1 \rangle \\
\mathsf{wrap}\, \tau_0\ v_0 & \rhd\ v_0 \\
\quad \text{if } \tau_0 \in \mathsf{Int} \cup \mathsf{Nat} \text{ and } \mathit{shape\text{-}match}\, (\tau_0, v_0) &
\end{array}
$$

$\boxed{e \rightarrow^* e}$ $\stackrel{def}{=}$ reflexive, transitive, compatible (w.r.t. E) closure of $\rhd$

Figure 76: Semantics for the evaluation syntax

$$(\mathit{unop}\ ((v_0))^{\overline{\ell}_0})^{\ell_1} \quad \triangleright \quad (\mathsf{TagErr})^{\ell_1}$$
if $v_0 \notin (v)^{\ell}$ and $\delta(\mathit{unop}, v_0)$ is undefined

$$(\mathit{unop}\ ((v_0))^{\overline{\ell}_0})^{\ell_1} \quad \triangleright \quad ((\delta(\mathit{unop}, v_0)))^{\overline{\ell}_0 \ell_1}$$
if $\delta(\mathit{unop}, v_0)$ is defined

$$(\mathit{binop}\ ((v_0))^{\overline{\ell}_0}\ ((v_1))^{\overline{\ell}_1})^{\ell_2} \quad \triangleright \quad (\mathsf{TagErr})^{\ell_2}$$
if $v_i \notin (v)^{\ell}$ and $\delta(\mathit{binop}, v_0, v_1)$ is undefined

$$(\mathit{binop}\ ((v_0))^{\overline{\ell}_0}\ ((v_1))^{\overline{\ell}_1})^{\ell_2} \quad \triangleright \quad (\delta(\mathit{binop}, v_0, v_1))^{\ell_2}$$
if $\delta(\mathit{binop}, v_0, v_1)$ is defined

$$(\mathsf{app}\ ((v_0))^{\overline{\ell}_0}\ v_1)^{\ell_1} \quad \triangleright \quad (\mathsf{TagErr})^{\ell_1}$$
if $v_0 \notin (v)^{\ell} \cup \lambda x.\, e \cup \lambda(x : \tau).\, e \cup \lambda(x : \sigma).\, e \cup \mathbb{G}\ \tau\ v$

$$(\mathsf{app}\ ((\lambda x_0.\, e_0))^{\overline{\ell}_0}\ v_0)^{\ell_1} \quad \triangleright \quad ((e_0[x_0 \leftarrow ((v_0))^{\ell_1 \mathit{rev}(\overline{\ell}_0)}]))^{\overline{\ell}_0 \ell_1}$$

$$(\mathsf{app}\ ((\lambda(x_0 : \tau_0).\, e_0))^{\overline{\ell}_0}\ v_0)^{\ell_1} \quad \triangleright \quad ((e_0[x_0 \leftarrow ((v_0))^{\ell_1 \mathit{rev}(\overline{\ell}_0)}]))^{\overline{\ell}_0 \ell_1}$$

$$(\mathsf{app}\ ((\lambda(x_0 : \sigma_0).\, e_0))^{\overline{\ell}_0}\ v_0)^{\ell_1} \quad \triangleright \quad (\mathsf{ScanErr})^{\ell_1}$$
if $\neg \mathit{shape\text{-}match}\ (\sigma_0, v_0)$

$$(\mathsf{app}\ ((\lambda(x_0 : \sigma_0).\, e_0))^{\overline{\ell}_0}\ v_0)^{\ell_1} \quad \triangleright \quad ((e_0[x_0 \leftarrow ((v_0))^{\ell_1 \mathit{rev}(\overline{\ell}_0)}]))^{\overline{\ell}_0 \ell_1}$$
if $\mathit{shape\text{-}match}\ (\sigma_0, v_0)$

$$(\mathsf{app}\ ((\mathbb{G}\ \tau_0 \Rightarrow \tau_1\ (v_0)^{\ell_0}))^{\overline{\ell}_1}\ v_1)^{\ell_2} \quad \triangleright$$
$$((\mathsf{wrap}\ \tau_1\ (\mathsf{app}\ v_0\ (\mathsf{wrap}\ \tau_0\ ((v_1))^{\ell_2 \mathit{rev}(\overline{\ell}_1)}))^{\ell_0}))^{\overline{\ell}_1 \ell_2}$$

$$(\mathsf{noop}\ ((v_0))^{\overline{\ell}_0})^{\ell_1} \quad \triangleright \quad ((v_0))^{\overline{\ell}_0 \ell_1}$$

$$(\mathsf{scan}\ \sigma_0\ ((v_0))^{\overline{\ell}_0})^{\ell_1} \quad \triangleright \quad (\mathsf{ScanErr})^{\ell_1}$$
if $\neg \mathit{shape\text{-}match}\ (\sigma_0, v_0)$

$$(\mathsf{scan}\ \sigma_0\ ((v_0))^{\overline{\ell}_0})^{\ell_1} \quad \triangleright \quad ((v_0))^{\overline{\ell}_0 \ell_1}$$
if $\mathit{shape\text{-}match}\ (\sigma_0, v_0)$

$$(\mathsf{wrap}\ \tau_0\ ((v_0))^{\overline{\ell}_0})^{\ell_1} \quad \triangleright \quad (\mathsf{WrapErr})^{\ell_1}$$
if $\mathit{shape\text{-}match}\ (\mathit{shape}\ (\sigma_0), v_0)$

$$(\mathsf{wrap}\ \tau_0 \Rightarrow \tau_1\ ((v_0))^{\overline{\ell}_0})^{\ell_1} \quad \triangleright \quad (\mathbb{G}\ \tau_0 \Rightarrow \tau_1\ ((v_0))^{\overline{\ell}_0})^{\ell_1}$$
if $\mathit{shape\text{-}match}\ (\mathsf{Fun}, v_0)$

$$(\mathsf{wrap}\ \tau_0 \times \tau_1\ ((\langle v_0, v_1 \rangle))^{\overline{\ell}_0})^{\ell_1} \quad \triangleright$$
$$(\langle \mathsf{wrap}\ \tau_0\ ((v_0))^{\overline{\ell}_0}, \mathsf{wrap}\ \tau_1\ ((v_1))^{\overline{\ell}_0} \rangle)^{\ell_1}$$

$$(\mathsf{wrap}\ \tau_0\ ((v_0))^{\overline{\ell}_0})^{\ell_1} \quad \triangleright \quad (v_0)^{\ell_1}$$
if $\tau_0 \in \mathsf{Int} \cup \mathsf{Nat}$ and $\mathit{shape\text{-}match}\ (\tau_0, v_0)$

Figure 77: Labeled semantics for the evaluation language, derived from figure 76 and the guidelines in chapter 4.4.4.

$\boxed{\delta : \mathit{unop} \times v \longrightarrow v}$

$\delta(\mathsf{fst}, \langle v_0, v_1 \rangle) = v_0$

$\delta(\mathsf{snd}, \langle v_0, v_1 \rangle) = v_1$

$\boxed{\delta : \mathit{binop} \times v \times v \longrightarrow v}$

$\delta(\mathsf{plus}, i_0, i_1) = i_0 + i_1$

$\delta(\mathsf{quotient}, i_0, 0) = \mathsf{DivErr}$

$\delta(\mathsf{quotient}, i_0, i_1) = \lfloor i_0 / i_1 \rfloor$

$\boxed{\mathit{shape\text{-}match} : \sigma \times v \longrightarrow \mathcal{B}}$

$\mathit{shape\text{-}match}\,(\mathsf{Fun}, v_0) = \mathsf{True}$
if $v_0 \in \lambda x.\, e \cup \lambda(x : \tau).\, e \cup \lambda(x : \sigma).\, e \cup \mathbb{G}\ \tau\ v$

$\mathit{shape\text{-}match}\,(\mathsf{Pair}, \langle v_0, v_1 \rangle) = \mathsf{True}$

$\mathit{shape\text{-}match}\,(\mathsf{Int}, i_0) = \mathsf{True}$

$\mathit{shape\text{-}match}\,(\mathsf{Nat}, n_0) = \mathsf{True}$

$\mathit{shape\text{-}match}\,(\mathsf{Any}, v_0) = \mathsf{True}$

$\mathit{shape\text{-}match}\,(\sigma_0, v_0) = \mathsf{False}$
otherwise

Figure 78: Semantic metafunctions

6.1.6 *Reduction Relation*

The semantics of the evaluation syntax is based on one notion of reduction (figure 76). Aside from the domain checks for shallow-typed functions, reduction proceeds in a standard, untyped fashion. Unary and binary operations proceed according to the δ metafunction (figure 78). Basic function application substitutes an argument value into a function body. Wrapped function application decomposes into two wrap boundaries: one for the input and another for the result. Lastly, boundary terms optionally perform a run-time check. A noop boundary performs no check and lets any value cross. A scan boundary checks the top-level shape of a value against the expected type. And a wrap boundary checks top-level shapes and either installs a wrapper around a higher-order value or recursively checks a data structure.

Figure 78 defines evaluation metafunctions. The δ function gives semantics to primitives. The *shape-match* function matches a type shape against the outer structure of a value.

6.1.7 *Single-Owner Consistency*

Deep types are characterized by complete monitoring (chapter 4). To state a complete monitoring theorem, the model needs a labeled syntax, a single-owner consistency judgment, and a reduction relation that propagates labels.

$$
\begin{array}{rcl}
e & = & x \mid v \mid \langle e, e \rangle \mid \mathit{unop}\, e \mid \mathit{binop}\, e\, e \mid \mathsf{app}\, e\, e \mid \mathsf{Err} \mid \\
 & & \mathsf{wrap}\, \tau\ (e)^{\ell} \mid \mathsf{scan}\, \sigma\ (e)^{\ell} \mid \mathsf{noop}\ \ (e)^{\ell} \mid (e)^{\ell} \\
v & = & i \mid \langle v, v \rangle \mid \lambda x.\, e \mid \lambda (x : \tau).\, e \mid \lambda (x : \sigma).\, e \mid \mathbb{G}\, \tau\ (v)^{\ell} \mid (v)^{\ell} \\
E & = & \ldots \mid (E)^{\ell} \\
\ell & = & \mathcal{D}_0 \mid \mathcal{D}_1 \mid \ldots \mid \mathcal{S}_0 \mid \mathcal{S}_1 \mid \ldots \mid \mathcal{U}_0 \mid \mathcal{U}_1 \mid \ldots \\
\overline{\ell} & = & \text{sequence of ownership labels } (\ell) \\
\mathcal{L} & = & \cdot \mid (x : \ell), \mathcal{L}
\end{array}
$$

Figure 79: Ownership syntax

The labeled syntax permits an ownership label around any expression (figure 79). For example, the terms $(4)^{\ell_0}$ and $(\mathsf{app}\{x_0\}\, x_1\ )^{\ell_1}$ illustrate one labeled value and one labeled expression. Most terms may have zero or more labels. Boundary terms are an exception; a wrap, scan, or noop boundary must have at least one label around its subexpression. The notation $((e_0))^{\overline{\ell}_0}$ matches an expression with a sequence of labels ($\overline{\ell}_0$).

An ownership label ℓ_0 carries two pieces of information. First is a typing discipline: $\mathcal{D}$ for deep, $\mathcal{S}$ for shallow, and $\mathcal{U}$ for untyped. Second is a natural number index to distinguish different labels. Initially, in a well-formed expression, these labels state the original owner and typing of a subterm. As expressions reduce to values and flow across boundaries, labels accumulate to show which components are partly responsible for these values.

Ultimately, the goal of our complete monitoring proof effort is to show that only deep-typed code is responsible for deep-typed expressions. Both shallow and untyped may recklessly share values. The single-owner consistency judgment in figure 80 formalizes the target invariant by stating when an expression is consistent for label ℓ_0 and label environment $\mathcal{L}_0$. A variable must be bound to ℓ_0 in the label environment. Non-boundary terms must have consistent subterms. Boundary terms and guard wrappers are ownership switch points; a boundary is consistent if its subterm is consistent with respect to the label inside the boundary. Finally, the rules for explicitly-labeled expressions impose a discipline on labels. A deep-labeled expression may have other deep labels, but nothing weaker. Shallow and untyped-labeled expressions, by contrast, can mix together.

Reduction of a labeled expression begins with the rules for the evaluation language (figure 76) and propagates labels according to the laws stated in chapter 4.4.4. Figure 77 presents the rules in full. In short, labels always accumulate unless a simple value meets a boundary with a matching type shape. Even noop boundaries add a label; this is why ownership consistency allows sequences of deep labels.

$$\frac{(x_0 : \ell_0) \in \mathcal{L}_0}{\ell_0; \mathcal{L}_0 \Vdash x_0} \qquad \frac{}{\ell_0; \mathcal{L}_0 \Vdash i_0} \qquad \frac{\ell_0; \mathcal{L}_0 \Vdash e_0 \qquad \ell_0; \mathcal{L}_0 \Vdash e_1}{\ell_0; \mathcal{L}_0 \Vdash \langle e_0, e_1 \rangle}$$

$$\frac{\ell_0; (x_0 : \ell_0), \mathcal{L}_0 \Vdash e_0}{\ell_0; \mathcal{L}_0 \Vdash \lambda x_0.\, e_0} \qquad \frac{\ell_0; (x_0 : \ell_0), \mathcal{L}_0 \Vdash e_0}{\ell_0; \mathcal{L}_0 \Vdash \lambda (x_0 : \sigma_0).\, e_0}$$

$$\frac{\ell_0; (x_0 : \ell_0), \mathcal{L}_0 \Vdash e_0}{\ell_0; \mathcal{L}_0 \Vdash \lambda (x_0 : \tau_0).\, e_0} \qquad \frac{\ell_0; \mathcal{L}_0 \Vdash e_0}{\ell_0; \mathcal{L}_0 \Vdash \mathit{unop}\, e_0}$$

$$\frac{\ell_0; \mathcal{L}_0 \Vdash e_0 \qquad \ell_0; \mathcal{L}_0 \Vdash e_1}{\ell_0; \mathcal{L}_0 \Vdash \mathit{binop}\, e_0\, e_1} \qquad \frac{\ell_0; \mathcal{L}_0 \Vdash e_0 \qquad \ell_0; \mathcal{L}_0 \Vdash e_1}{\ell_0; \mathcal{L}_0 \Vdash \mathsf{app}\, e_0\, e_1}$$

$$\frac{}{\ell_0; \mathcal{L}_0 \Vdash \mathsf{Err}} \qquad \frac{\ell_1; \mathcal{L}_0 \Vdash e_0}{\ell_0; \mathcal{L}_0 \Vdash \mathsf{noop}\ \ (e_0)^{\ell_1}} \qquad \frac{\ell_1; \mathcal{L}_0 \Vdash e_0}{\ell_0; \mathcal{L}_0 \Vdash \mathsf{scan}\, \sigma_0\ (e_0)^{\ell_1}}$$

$$\frac{\ell_1; \mathcal{L}_0 \Vdash e_0}{\ell_0; \mathcal{L}_0 \Vdash \mathsf{wrap}\, \tau_0\ (e_0)^{\ell_1}} \qquad \frac{\ell_1; \mathcal{L}_0 \Vdash v_0}{\ell_0; \mathcal{L}_0 \Vdash \mathbb{G}\ \tau_0\ (v_0)^{\ell_1}} \qquad \frac{\mathcal{D}_1; \mathcal{L}_0 \Vdash e_0}{\mathcal{D}_0; \mathcal{L}_0 \Vdash (e_0)^{\mathcal{D}_1}}$$

$$\frac{\mathcal{S}_1; \mathcal{L}_0 \Vdash e_0}{\mathcal{S}_0; \mathcal{L}_0 \Vdash (e_0)^{\mathcal{S}_1}} \qquad \frac{\mathcal{U}_0; \mathcal{L}_0 \Vdash e_0}{\mathcal{S}_0; \mathcal{L}_0 \Vdash (e_0)^{\mathcal{U}_0}} \qquad \frac{\mathcal{U}_1; \mathcal{L}_0 \Vdash e_0}{\mathcal{U}_0; \mathcal{L}_0 \Vdash (e_0)^{\mathcal{U}_1}}$$

$$\frac{\mathcal{S}_0; \mathcal{L}_0 \Vdash e_0}{\mathcal{U}_0; \mathcal{L}_0 \Vdash (e_0)^{\mathcal{S}_0}}$$

Figure 80: Single-owner consistency

6.1.8 *Properties*

The primary meta-theoretic results are about type soundness and complete monitoring. Type soundness predicts the possible outcomes of a well-typed expression. Naturally, these outcomes depend on the "strength" of the static types; for example, untyped code has weaker guarantees than shallow code. Complete monitoring asks whether single-owner consistency is an invariant; if so, then programmers can trust deep types as behavioral guarantees.

The statement of type soundness relies on one new notation and a family of metafunctions. The notation $s_0 \to^* e_0$ defines evaluation for surface expressions; the meaning is that s_0 is well-typed somehow ($\exists \mathcal{X}.\ \vdash_s s_0 : \mathcal{X}$), compiles to an evaluation expression ($\vdash_s s_0 : \mathcal{X} \rightsquigarrow e_1$), and then the compiled expression steps to the result ($e_1 \to^* e_0$). The metafunctions—**0**, **shape**, and **1**—map surface-language types to evaluation types. One function, **shape**, extends the similarly-name function from figure 72 to map the unitype $\mathcal{U}$ to itself. The others are simple: **0** maps all types to $\mathcal{U}$ and **1** is the identity. These tools enable a concise, parameterized statement of type soundness.

Note that type soundness does not rule out any particular errors. Two extensions could enable a finer statement: (1) split the one notion of reduction into three and introduce new errors for invariant failues; (2) introduce three kinds of evaluation context and show that steps inside typed code do not raise tag errors. Chapter 4 demonstrates the first method. Greenman and Felleisen [43] demonstrate the second.

Definition 6.1.1 (TS(F))**.** *Language L satisfies* $\mathbf{TS}\,(\mathsf{F})$ *if for all s_0 such that $\vdash_s s_0 : \mathcal{X}$ holds, one of the following holds:*

- $s_0 \to^* v_0$ *and* $\vdash_L v_0 : \mathsf{F}\,(\mathcal{X})$
- $s_0 \to^* \mathsf{Err}$
- $s_0 \to^*$ *diverges*

Theorem 6.1.2 (type soundness)**.**

- *Language $\mathcal{U}$ satisfies* $\mathbf{TS}\,(\mathbf{0})$
- *Language $\mathcal{S}$ satisfies* $\mathbf{TS}\,(\mathbf{shape})$
- *Language $\mathcal{D}$ satisfies* $\mathbf{TS}\,(\mathbf{1})$

Proof. Lemma 6.1.4 guarantees that the compiled form of the surface expression is well-typed. The rest follows from straightforward progress and preservation lemmas for the evaluation typing judgments. Lemma 6.1.5 is essential to preservation for primitive operations. Lemmas 6.1.6 and 6.1.7 are key aspects of preservation for boundary terms. □

Complete monitoring is technically a statement about labeled expressions and a label-propagating reduction relation. But, because the propagating reduction is derived from the basic reduction relation in a straightforward manner, our theorem statement uses the basic symbol ($\rightarrow^*$). Likewise, both e_0 and e_1 refer to a labeled variant of an evaluation-language expression. If no such labeling exist for a term, then the theorem holds vacuously.

Theorem 6.1.3 (complete monitoring)**.** *If* $\vdash_s s_0 : \mathcal{X}$ *and* $\vdash_s s_0 : \mathcal{X} \rightsquigarrow e_0$ *and* $\ell_0; \cdot \Vdash e_0$ *and* $e_0 \rightarrow^* e_1$ *then* $\ell_0; \cdot \Vdash e_1$.

Proof. By a preservation argument. The proofs for each basic reduction step are sketched below. These depend on two metafunctions: *rev* reverses a sequence of labels and *last* extracts the last (outermost) element of such a sequence.

CASE: $(unop\ ((v_0))^{\overline{\ell}_0})^{\ell_1} \rhd (\mathsf{TagErr})^{\ell_1}$

by the definition, $\ell_1; \cdot \Vdash (\mathsf{TagErr})^{\ell_1}$.

CASE: $(unop\ ((v_0))^{\overline{\ell}_0})^{\ell_1} \rhd ((\delta(unop, v_0)))^{\overline{\ell}_0 \ell_1}$

1. $\overline{\ell}_0$ is either all deep labels or a mix of shallow and untyped, by single-owner consistency of the redex.
2. similarly, ℓ_1 must match $\overline{\ell}_0$
3. v_0 is a pair, because δ is defined on it.
4. both components of v_0 are well-labeled, again by single-owner consistency on the redex.
5. by the definition of δ.

CASE: $(binop\ ((v_0))^{\overline{\ell}_0}\ ((v_1))^{\overline{\ell}_1})^{\ell_2} \rhd (\mathsf{TagErr})^{\ell_2}$

by the definition of $\Vdash$.

CASE: $(binop\ ((v_0))^{\overline{\ell}_0}\ ((v_1))^{\overline{\ell}_1})^{\ell_2} \rhd (\delta(binop, v_0, v_1))^{\ell_2}$

by the definition of $\Vdash$ and δ; note that the binary operators are not elimination forms.

CASE: $(\mathsf{app}\ ((v_0))^{\overline{\ell}_0}\ v_1)^{\ell_1} \rhd (\mathsf{TagErr})^{\ell_1}$

by the definition of $\Vdash$.

CASE: $(\mathsf{app}\ ((\lambda x_0.\, e_0))^{\overline{\ell}_0}\ v_0)^{\ell_1} \rhd ((e_0[x_0 \leftarrow ((v_0))^{\ell_1 rev\,(\overline{\ell}_0)}]))^{\overline{\ell}_0 \ell_1}$

1. $\overline{\ell}_0$ is all deep or a mix of shallow and untyped, by single-owner consistency of the redex.

2. $\ell_2; \cdot \Vdash v_0$, also by single-owner consistency of the redex.
3. $\mathit{last}\,(\overline{\ell}_0); \cdot \Vdash ((v_0))^{\ell_1 \mathit{rev}\,(\overline{\ell}_0)}$, by steps 1 and 2.
4. $\mathit{last}\,(\overline{\ell}_0); \cdot \Vdash x_0$ for each occurrence of x_0 in e_0, by single-owner consistency of the redex.
5. by a substitution lemma.

CASE: $(\mathsf{app}\,((\lambda(x_0 : \tau_0).\, e_0))^{\overline{\ell}_0}\, v_0)^{\ell_1} \rhd ((e_0[x_0 \leftarrow ((v_0))^{\ell_1 \mathit{rev}\,(\overline{\ell}_0)}]))^{\overline{\ell}_0 \ell_1}$

similar to the previous case.

CASE: $(\mathsf{app}\,((\lambda(x_0 : \sigma_0).\, e_0))^{\overline{\ell}_0}\, v_0)^{\ell_1} \rhd (\mathsf{ScanErr})^{\ell_1}$

by the definition of $\Vdash$.

CASE: $(\mathsf{app}\,((\lambda(x_0 : \sigma_0).\, e_0))^{\overline{\ell}_0}\, v_0)^{\ell_1} \rhd ((e_0[x_0 \leftarrow ((v_0))^{\ell_1 \mathit{rev}\,(\overline{\ell}_0)}]))^{\overline{\ell}_0 \ell_1}$

similar to the other substitution cases.

CASE: $(\mathsf{app}\,((\mathbb{G}\ \tau_0 \Rightarrow \tau_1\ (v_0)^{\ell_0}))^{\overline{\ell}_1}\, v_1)^{\ell_2} \rhd$
$((\mathsf{wrap}\,\tau_1\ (\mathsf{app}\,v_0\,(\mathsf{wrap}\,\tau_0\ ((v_1))^{\ell_2 \mathit{rev}\,(\overline{\ell}_1)}))^{\ell_0}))^{\overline{\ell}_1 \ell_2}$

1. $\ell_0; \cdot \Vdash v_0$, by single-owner consistency of the redex.
2. $\ell_2; \cdot \Vdash v_1$, again by the redex.
3. $\overline{\ell}_1$ is either all deep or a mix of shallow and untyped, again by the redex.
4. by the definition of $\Vdash$.

CASE: $(\mathsf{noop}\ ((v_0))^{\overline{\ell}_0})^{\ell_1} \rhd ((v_0))^{\overline{\ell}_0 \ell_1}$

by the definition of $\rightsquigarrow$, because a noop boundary connects either: two deep components, two shallow components, two untyped components, or one shallow and one untyped component.

CASE: $(\mathsf{scan}\,\sigma_0\ ((v_0))^{\overline{\ell}_0})^{\ell_1} \rhd (\mathsf{ScanErr})^{\ell_1}$

by the definition of $\Vdash$.

CASE: $(\mathsf{scan}\,\sigma_0\ ((v_0))^{\overline{\ell}_0})^{\ell_1} \rhd ((v_0))^{\overline{\ell}_0 \ell_1}$

by the definition of $\rightsquigarrow$, because a scan boundary only links an untyped component to a shallow component.

CASE: $(\mathsf{wrap}\,\tau_0\ ((v_0))^{\overline{\ell}_0})^{\ell_1} \rhd (\mathsf{WrapErr})^{\ell_1}$

by the definition of $\Vdash$.

CASE: $(\mathsf{wrap}\,\tau_0 \Rightarrow \tau_1\ ((v_0))^{\overline{\ell}_0})^{\ell_1} \rhd (\mathbb{G}\ \tau_0 \Rightarrow \tau_1\ ((v_0))^{\overline{\ell}_0})^{\ell_1}$

by the definition of $\Vdash$.

CASE: $\left(\mathsf{wrap}\,\tau_0 \times \tau_1\ \left(\left(\langle v_0, v_1\rangle\right)\right)^{\overline{\ell}_0}\right)^{\ell_1} \rhd$
$\left(\langle \mathsf{wrap}\,\tau_0\ \left(\left(v_0\right)\right)^{\overline{\ell}_0}, \mathsf{wrap}\,\tau_1\ \left(\left(v_1\right)\right)^{\overline{\ell}_0}\rangle\right)^{\ell_1}$

by the definition of $\Vdash$. Note that the rule moves the elements of the pair in the redex into a new pair in the contractum.

CASE: $\left(\mathsf{wrap}\,\tau_0\ \left(\left(v_0\right)\right)^{\overline{\ell}_0}\right)^{\ell_1} \rhd \left(v_0\right)^{\ell_1}$
where $\tau_0 \in \mathsf{Int} \cup \mathsf{Nat}$ and *shape-match* (τ_0, v_0)

by the definition of $\Vdash$.

□

Lemma 6.1.4 (completion)**.** *If* $\vdash_s s_0 : \mathcal{X}$ *then* $\vdash_s s_0 : \mathcal{X} \rightsquigarrow e_0$ *and either:*

- $\mathcal{X} \in \tau$ *and* $\vdash_{\mathcal{D}} e_0 : \mathcal{X}$
- $\mathcal{X} \in \lfloor \tau \rfloor$ *and* $\vdash_{\mathcal{S}} e_0 : \mathbf{shape}\,(\mathcal{X})$
- $\mathcal{X} \in \mathcal{U}$ *and* $\vdash_{\mathcal{U}} e_0 : \mathcal{U}$

Lemma 6.1.5 (δ, Δ agreement)**.**

- *If* $\Delta(unop, \mathcal{U}) = \mathcal{U}$ *and* $\vdash_{\mathcal{U}} v_0 : \mathcal{U}$
  and $\delta(unop, v_0)$ *is defined then* $\vdash_{\mathcal{U}} \delta(unop, v_0) : \mathcal{U}$
- *If* $\Delta(unop, \sigma_0) = \sigma_1$ *and* $\vdash_{\mathcal{S}} v_0 : \sigma_0$
  and $\delta(unop, v_0)$ *is defined then* $\vdash_{\mathcal{S}} \delta(unop, v_0) : \sigma_1$
- *If* $\Delta(unop, \tau_0) = \tau_1$ *and* $\vdash_{\mathcal{D}} v_0 : \tau_0$
  and $\delta(unop, v_0)$ *is defined then* $\vdash_{\mathcal{D}} \delta(unop, v_0) : \tau_1$
- *If* $\Delta(binop, \mathcal{U}, \mathcal{U}) = \mathcal{U}$ *and* $\vdash_{\mathcal{U}} v_0 : \mathcal{U}$ *and* $\vdash_{\mathcal{U}} v_1 : \mathcal{U}$
  and $\delta(binop, v_0, v_1)$ *is defined then* $\vdash_{\mathcal{U}} \delta(binop, v_0, v_1) : \mathcal{U}$
- *If* $\Delta(binop, \sigma_0, \sigma_1) = \sigma_2$ *and* $\vdash_{\mathcal{S}} v_0 : \sigma_0$ *and* $\vdash_{\mathcal{S}} v_1 : \sigma_1$
  and $\delta(binop, v_0, v_1)$ *is defined then* $\vdash_{\mathcal{S}} \delta(binop, v_0, v_1) : \sigma_2$
- *If* $\Delta(binop, \tau_0, \tau_1) = \tau_2$ *and* $\vdash_{\mathcal{D}} v_0 : \tau_0$ *and* $\vdash_{\mathcal{D}} v_1 : \tau_1$
  and $\delta(binop, v_0, v_1)$ *is defined then* $\vdash_{\mathcal{D}} \delta(binop, v_0, v_1) : \tau_2$

Lemma 6.1.6. *If* $\vdash_{\mathcal{S}} e_0 : \sigma_0$ *then* $\vdash_{\mathcal{U}} e_0 : \mathcal{U}$

Proof. By definition. The key rules are for shape-annotated functions.

□

Lemma 6.1.7 (boundary-crossing)**.**

- *If* $\vdash_{\mathcal{L}} v_0 : \mathcal{X}$ *and shape-match* (σ_0, v_0) *then* $\vdash_{\mathcal{S}} v_0 : \sigma_0$
- *If* $\vdash_{\mathcal{S}} v_0 : \sigma_0$ *then* $\vdash_{\mathcal{U}} v_0 : \mathcal{U}$
- *If* $\vdash_{\mathcal{D}} v_0 : \tau_0$ *and* $\mathsf{wrap}\,\tau_0\ v_0 \rhd v_1$ *then* $\vdash_{\mathcal{S}} v_1 : \mathbf{shape}\,(\tau_0)$ *and* $\vdash_{\mathcal{U}} v_1 : \mathcal{U}$

6.2 IMPLEMENTATION

The implementation of Shallow Racket begins with two new `#lang` languages to communicate the options available to programmers.

- Modules that start with `#lang typed/racket` continue to use deep types, same as earlier versions of Typed Racket;
- `#lang typed/racket/deep` is a new way to opt-in to deep types;
- and `#lang typed/racket/shallow` provides shallow types.

All three languages invoke the same type checker. At steps where deep and shallow disagree, the compiler queries the current language to proceed. For example, the type-directed optimizer checks that it has deep types before rewriting code based on the deep soundness guarantee.

Many parts of the modified compiler use a similar, one-or-the-other strategy to handle deep and shallow types. This section deals with the more challenging aspects. Sharing variables between deep and shallow required changes to type-lookup and wrapper generation (chapter 6.2.1). Sharing macros requires further changes; currently, deep-typed syntax can only be re-used through unsafe mechanisms (chapter 6.2.1). Lastly, Typed Racket has a small API that gives programmers control over the deep type enforcement strategy. This API needed generalizations to handle shallow types (chapter 6.2.3).

6.2.1 *Deep and Shallow Interaction*

Racket supports both separate compilation and hygienic macros [37]. Each module in a program gets compiled to a core language individually, and other modules can re-use the output. Typed Racket cooperates with the separate compilation protocol by serializing the results of type checking [22, 106]. A well-typed module compiles to untyped code (with appropriate contracts) and a local type environment. When one deep module imports from another, it can find the type of the imported identifier in the type environment.

At first glance, it appears that shallow code can use the same protocol to find the type of deep imports. The protocol fails, however, because wrappers get in the way. When deep wants to provide an

identifier, it really provides a piece of syntax called a rename transformer. These transformers expand to one of two identifiers depending on where they appear: deep-typed code gets the original identifier and can easily look up its type, but untyped and shallow code gets a wrapped version. The wrapper causes a direct type lookup to fail.

For deep-to-shallow exports, the solution is to modify type lookup to pass through wrappers. Fortunately, the change was easy to make because the Racket contract library provides enough metadata. At compile time (and only then), a wrapped identifier is associated with a structure that links back to the original. The shallow type checker looks out for these wrappers and uncovers the originals as needed.

Shallow-to-deep exports use a dual method. Like deep, a shallow module provides only rename transformers. These expand to the original identifier in other shallow and untyped code; the original is associated with type information. For deep clients, the transformers expand to a wrapped identifier. Consequently, the deep type checker watches for "untyped" wrappers and tests whether there is an available type. Such types allow static type checks to succeed, and at run-time the wrapper keeps deep code safe.

A surprising consequence of the final protocol is that a shallow module must be prepared to create wrappers for its exports. The wrapper-making code is generated during compilation, at the end of type checking, but it does not run until needed by a deep client. In this way, only programs that depend on deep code suffer from the expressiveness limits of wrappers.

6.2.2 *Syntax Re-Use*

Shallow code cannot use deep macros. Re-use is desirable to avoid copying code, but it requires a static analysis to enforce soundness. This section explains the problem and criteria for a solution.

To appreciate the problem, consider the following simple macro. This macro applies a typed function `f` to an input, and is consequently unsafe:

```
(define-syntax-rule (call-f x) (f x))
```

If this macro could appear in shallow code, then any shallow value `x` could sneak into the deep function. Unless `f` makes no assumptions about its input, such values can break the deep soundness guarantee and lead to dangerous results in optimized code.

One possible fix is to put a contract around every deep identifier that appears in a macro. Doing so would require an analysis to find out which contracts are needed, and a second analysis to install contracts wisely; each identifier requires a contract, but repeated occurrences of one identifier should not lead to repeated contract checks. It

should also be possible to avoid the contracts if the macro goes only to deep clients. These are major changes.

Another possibility is to statically check whether a macro is safe to export. Safe macros appear, for example, in the typed compatibility layer for the RackUnit testing library. RackUnit is an untyped library that exports some functions and some macros. The typed layer provides types for the functions and type-annotated copies of the macros (about 300 lines in total). These macros are safe because they do not expose any deep-typed identifiers. For example, the following macro combines a sequence of expressions into a named RackUnit test case:

```
(define-syntax (test-case stx)
  (syntax-parse stx
    [(_ name expr ...)
     (quasisyntax/loc stx
       (parameterize ([test-name (ensure-str name)])
         (test-begin expr ...)))]))
```

This macro is safe for shallow code, but for complicated reasons. First, `ensure-str` is a typed function that accepts any input. Second, `test-begin` is a macro from the same file that is also safe. Third, `parameterize` comes from untyped Racket.

Currently, the author of a deep library can enable syntax re-use by disabling the optimizer and unsafely providing macros. This workaround requires a manual inspection, but it is more appealing than forking the RackUnit library and asking programmers to choose the correct version.

6.2.3 *Deep–Untyped Utilities*

Typed Racket has a small API by Neil Toronto to let programmers control boundaries between deep and untyped code. The API arose over time, as programmers (including Neil) discovered challenges. Two forms in this API can lead to surprising results due to the existence of shallow code.

The first problem concerns `require/untyped-contract`. This form lets untyped code import a typed identifier whose precise type cannot be expressed with a deep contract. Users supply a supertype of the precise type and Deep Racket uses this weaker type to generate a contract.

For example, the `jpeg` benchmark depends on a library for multi-dimensional arrays (`math/array`). This library accepts two kinds of data for array indices: either a vector of natural numbers or a vector of integers. Helper functions assert that values with the integer type do not actually contain negative numbers using a run-time checking function:

```
(: check-array-shape
   (-> (U (Vectorof Natural) (Vectorof Integer))
       (Vectorof Natural)))
```

Deep contracts cannot express the type for the checking function because they lack support for true unions. The work around is to impose a supertype on untyped clients:

```
(require/untyped-contract
  [check-array-shape
   (-> (Vectorof Integer) (Vectorof Natural))])
```

This form comes with a surprising design choice. If an untyped-contract identifier flows back into typed code, the type checker uses the original type rather than the supertype. For deep code, the choice is convenient because more programs can type-check using the supertype. For shallow, though, the convenience disappears. A shallow client must receive the wrapped version of the identifier, which means shallow code must behave in accordance with the supertype; hence, the shallow type checker uses the supertype as well. Consequently, some well-typed deep programs raise type errors upon switching to shallow types.

The second problematic form is `define-typed/untyped-identifier`, which creates a new identifier from two old ones. The following example defines `f` from two other names:

```
(define-typed/untyped-identifier f
  typed-f
  untyped-f)
```

The meaning of the new `f` depends on the context in which it appears. In typed code, `f` expands to `typed-f`. In untyped code, an `f` is a synonym for `untyped-f`.

The `typed-f` is intended for deep-typed code. It cannot be safely used in a shallow module because it may assume type invariants. Consequently, shallow code gets the untyped id. This means, unfortunately, that changing a deep module to shallow can raise a type checking error because occurrences of `f` that expand to `untyped-f` are plain, untyped identifiers. There is no way to uncover the type that a `typed-f` would have, and anyway there is no guarantee that `typed-f` and `untyped-f` have the same behavior.

For now, such type errors call for programmer-supplied annotations in the shallow client code. In the future, this form would benefit from a third argument that specifies behavior in shallow contexts.

6.3 EVALUATION

The integration of Shallow Racket and Deep Racket has implications for expressiveness (chapter 6.3.1) and performance (chapter 6.3.2). Switching between these two type-enforcement strategies can help programmers express new designs and avoid huge performance costs.

6.3.1 *Expressiveness*

Conversations with Typed Racket users have shown that deep types can lead to unexpected outcomes. In some programs, type enforcement appears overly strict. In others, type enforcement is impossible because the implementation of Deep Racket lacks wrappers for certain kinds of values. Worst of all, the wrappers that Deep inserts can change hehavior. Shallow Racket avoids all of these issues because of its weak, wrapper-free method of enforcing types.

Less-strict Any Type

Inspired by 2 messages to the Racket-Users mailing list:

- *error : Attempted to use a higher-order value passed as 'Any' in untyped code*, sent by Denis Michiels on 2018-04-16.
  `groups.google.com/g/racket-users/c/cCQ6dRNybDg/m/CKXgX1PyBgAJ`
- *Typed Racket: 'Unable to protect opaque value passed as 'Any'' with interesting behavior*, sent by Marc Kaufmann on 2019-12-11.
  `groups.google.com/g/racket-users/c/jtmVDFCGL28/m/jwl4hsjtBQAJ`

The deep type named `Any` is a normal "top" type at compile-time, but it is surprisingly strict at run-time. For compile-time type checking, `Any` is a supertype of every other type. A function that expects an `Any` input must ask occurrence-typing questions before it can do anything to it. At run-time, the `Any` type is enforced with an opaque wrapper. Refer to Findler and Blume [32] for further discussion of why the opaque wrapper is necessary.

The wrapper is a surprise for developers who expect programs such as figure 81 to run without error. This program defines a mutable box in typed code, assigns the `Any` type to the box, and sends it to untyped code. The untyped module attempts to set the box. Deep Racket raises an exception when untyped code tries to modify the box. Unfortunately for the programmer, this error is essential for soundness. If untyped code put an integer in the box, then typed uses of the box would give a result that is inconsistent with its type.

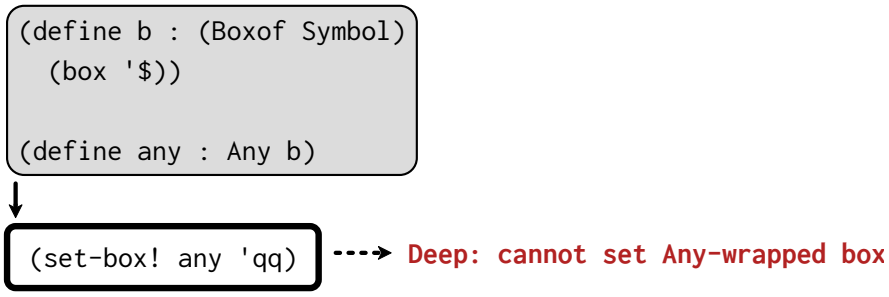


Figure 81: Deep seals mutable values of type `Any` in a wrapper. Shallow lets untyped code modify the box.

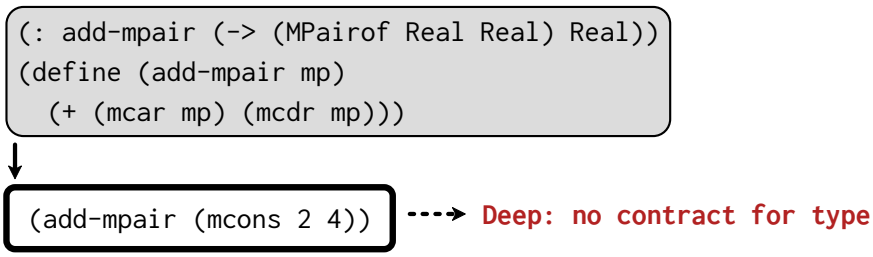


Figure 82: Deep lacks wrappers for mutable pairs and a few other datatypes. Shallow does not need wrappers, and can express mixed-typed programs that share such values with untyped code.

Shallow Racket runs the program without error because of its delayed checking strategy. If shallow-typed code tries to read a symbol from the box, then that access will raise an error. Until then, the program runs.

No Missing Wrappers

Every kind of mutable value that can appear in deep code needs a kind of wrapper to protect it against untyped contexts. Wrappers do not exist for some values, causing Deep to reject code that sends such a value across a boundary.

Figure 82 demonstrates the issue with a mutable pair (`MPairof`) type. Deep raises a run-time error when untyped code tries to call the `add-mpair` function. In total, there are twelve types that suffer from this issue. Implementing wrappers for these types is a challenge. For example, syntax objects can contain mutable data and therefore need wrappers. But syntax wrappers would require changes to many parts of the Racket compiler, including the macro expander.

```
(require/typed racket/list
  [index-of
   (All (T)
    (-> (Listof T) T
        (U #f Natural)))])

(index-of '(a b) 'a)
```

```
#f
```

Figure 83: The deep contract for an `All` type can change the behavior of untyped code.

Shallow Racket avoids the question of how to implement complex wrappers thanks to the transient semantics. Consequently, programmers gain the ability to send new types across boundaries and explore new mixed-typed designs.

Uniform Behavior

> *Inspired by 2 messages to the Racket-Users mailing list:*
>
> - *Typed code from untyped code*, sent by Bertrand on 2020-02-17.
>   `groups.google.com/g/racket-users/c/UD20HadJ9Ec/m/Lmuw0U8mBwAJ`
> - *index-of + TR ... parametricity problem?*, sent by John B. Clements on 2019-12-15.
>   `groups.google.com/g/racket-users/c/ZbYRQCy93dY/m/kF_Ek0VvAQAJ`

Although the purpose of Deep Racket wrappers is to reject certain operations without changing anything else about a program, wrappers can cause some programs to run differently. One obvious case is code that explicitly looks for wrappers; the answers to low-level observations such as `has-contract?` may depend on the type boundaries in a deep program. Figure 83 presents a second, more subtle case. This typed module imports an untyped function, `index-of`, with a precise polymorphic type. The wrapper that enforces this type creates a new wrapper for every input to the function—to enforce parametric polymorphism [49]. Unfortunately, these input wrappers change the behavior of `index-of`; it ends up searching the list for a wrapped version of the symbol `'a` and returns a "not found" result (`#f`) instead of the correct position.

Shallow Racket avoids all such changes in behavior, including the well-know object identity issues [56, 94, 111, 114], because the transient semantics does not use wrappers to enforce types.

6.3.2 *Performance*

With the Shallow Racket implementation, the tradeoffs of chapter 5 disappear. For all our benchmarks, the choice improves the worst-case overhead of type boundaries. By implication, Typed Racket can offer a new migration story (appendix A.4):

> *use shallow types when converting an untyped application and switch to deep types after the boundaries stabilize.*

Mixing deep and shallow types in one program offers new ways of improving performance.

GTP Benchmarks, Worst-Case

Now that Racket programmers can easily switch between deep and shallow types, worst-case overheads improve by orders of magnitude. Before, the cost of deep types overwhelmed many configurations. After, the costs can be avoided by changing the first line (the language specification) of the typed modules.

Figure 84 quantifies the improvements in the Typed Racket benchmarks. The first data column reports the old worst-case overheads. The second columns reports the new worst-case, now that programmers can pick the best of deep and shallow types. The final column is the quotient between the first two. In short, the "after" case is always better and can be an arbitrarily large improvement.

Case Studies: Deep and Shallow

Early experience with Shallow Racket shows that the combination of deep and shallow types can be better that either alone. Here are three motivating case studies. Appendix A.4 contains additional data.

SYNTH The synth benchmark is derived from an untyped program that interacts with part of a typed math library. When the library code uses deep types, the original client runs with high overhead—14x slower that a deep-typed client.

Changing the library to use shallow types improves the gap between an untyped and deep-typed client to 5x. This fast untyped configuration is about 2x slower than the fast deep-deep configuration, but the worst-case is 1.39x faster (3 seconds) than before. Overall, the shallow library is a better tradeoff for synth.

Benchmark	worst before	worst after	improvement
sieve	15.67x	4.36x	3x
forth	4010.71x	5.51x	727x
fsm	2.38x	2.37x	<2x
fsmoo	451.07x	4.28x	105x
mbta	1.92x	1.74x	<2x
morsecode	2.77x	1.47x	<2x
zombie	54.62x	31.42x	<2x
dungeon	14573.66x	4.97x	2930x
jpeg	23.16x	1.66x	13x
zordoz	2.75x	2.72x	<2x
lnm	1.21x	1.11x	<2x
suffixtree	31.17x	5.80x	5x
kcfa	4.43x	1.24x	3x
snake	11.84x	7.67x	<2x
take5	32.17x	2.99x	10x
acquire	4.15x	1.42x	2x
tetris	11.71x	9.93x	<2x
synth	49.12x	4.20x	11x
gregor	1.63x	1.59x	<2x
quadT	27.10x	7.39x	3x
quadU	59.66x	7.57x	7x

Figure 84: Worst-case overhead before (deep types) and after (either deep or shallow) the integration of Deep and Shallow Racket.

MSGPACK MessagePack is a serialization format. MsgPack is a Typed Racket library that maps Racket values to binary data according to the format. The author of this library reported a performance hit after narrowing some types from `Any` to a more-precise union type for serializable inputs. Tests that formerly passed on the package server timed out after the change.

I cloned MsgPack commit `64a6098` and found that running all unit tests took 320 seconds. Changing one file to shallow types brought the time down to 204 seconds—a huge improvement for a one-line switch. Moving the rest of the library from deep to shallow types adds only a slight improvement (down to 202 seconds), which suggests that a mix of deep and shallow is best.

EXTERNAL DATA Typed code that deals with data from an external source is often better off with shallow types because they lazily validate data as it is accessed. By contrast, Typed Racket's implementation of deep types eagerly traverses a data structure as soon as it reaches a type boundary. If the boundary types allow mutable values, then the traversal is even more expensive because it creates wrappers as it copies the dataset.

To illustrate the pitfall, I wrote a typed script that reads a large dataset of apartment data using on off-the-shelf JSON parser and accesses one field from each object in the dataset. Deep types make the script run over 10x slower than shallow types.

In principle, deep code can avoid the slowdown with a custom parser that validates data as it reads it. Indeed, Phil Nguyen has written a library for JSON that mitigates the overhead of deep types. Such libraries are ideal, but until we have them for the next data exchange format (SQL, XML, YAML, ...) shallow types get the job done with the parsers that are available today.

Release Information

Shallow Typed Racket is publicly available in a pull request to Typed Racket: `racket/typed-racket #948`. The patch adds support for shallow types, giving Typed Racket programmers a choice between shallow and deep type guarantees. I expect to merge the pull request early in 2021. After the release, I look forward to studying programmers' experience with the multi-faceted system.

7

FUTURE WORK

Now that we have a language that provides deep types via the natural semantics and shallow types via the transient one, two lines of crucial future work are apparent: improving blame and improving the performance of transient.

7.1 TRANSIENT WITH BLAME, NATURAL WITHOUT BLAME

The most surprising result of my research is the huge cost of transient blame (chapter 5.4.4). Because so many benchmarks run slower with blame than in the worst case of deep types, Shallow Racket does not even attempt to track blame.

This result demands a two-step investigation. The first step is to assess the usefulness of the original transient blame algorithm. Both user studies and automated analyses [60] can help. Preliminary investigations suggest that transient can effectively ignore all but the first boundary in a blame set. Once the community knows more about what makes blame valuable, then the second step is the development of efficient algorithms that are tailored to developers' needs.

As for deep types, the natural semantics is designed with blame in mind. If blame is not needed, then an alternative semantics could enforce the same type guarantees using fewer wrappers. Feltey et al. [29] show that removing some wrappers while preserving blame behavior leads to better performance. An implementation could remove many more wrappers if it ignores blame.

7.1.1 *Transient Blame Filtering*

My implementation of blame for Shallow Racket makes an effort to filter irrelevant boundaries as suggested by Vitousek et al. [115]. Filtering, however, is expensive and fails on boundaries that use generative struct types (chapter 5.2.6). The failure warrants further investigation. But regardless of whether run-time filtering can cover more types, we need to measure its usefulness. If filtering is unlikely to help programmers diagnose issues, then removing it can save a tremendous amount of bookkeeping. The blame map can drop all types and actions.

Despite the large improvement relative to natural, the cost of transient types is still high. The fully-typed configurations of the benchmarks make this problem apparent (figure 58); in the worst case, transient is 30x slower than untyped even with type-directed optimizations. Transient needs a way to reduce the cost of shape checks.

Vitousek et al. [116] have demonstrated that a static analysis and a tracing JIT compiler can greatly improve performance in Reticulated Python under the assumption that the program does not interact with un-analyzed "open world" code. The work sets a high bar; every benchmark runs within 1.25x overhead. What remains to be seen is how well an analysis can do without the closed-world assumption, and whether ahead-of-time techniques can replicate the speedups enabled by the JIT.

Earlier versions of Shallow Racket ran much slower due to redundant checks and the overhead of contract library combinators. Perhaps further analysis and ahead-of-time optimization can close the gap between fully-typed shallow and deep. Starting points for such an analysis include occurrence typing [105], modular set-based analysis [68], and Henglein's tagging optimization [51]. My investigations in chapter 5 suggest two additional starting points, based on the observation that Shallow Racket checks the result of almost every function call that occurs in typed code:

- The only function calls that are not protected with a shape check have the form `(f x ....)` where the identifier `f` appears in a trusted environment. For example, Shallow Racket trusts that calls to `map` return proper lists. This approach has major limitations. Checking identifiers is brittle; an alias to `map` defeats the optimization. Furthermore, the current approach cannot trust deeper properties of a type. A call to `filter`, for example, guarantees the shape of the result *and* the shape of every element in the list. There should be some way to encode this shape knowledge in a type environment, rather than a flat identifier environment.

- Some user-defined functions do not need transient result checks. If a transient module defines a function `f = (λ (x) ....)` then there is no need for the current module to check its results because static typing guarantees a shape-correct output. Other functions that are defined indirectly, for example by reading a function from an untyped list (`f = (car f*)`), cannot be trusted.

To experiment with similar improvements, the model from chapter 6.1 must gain syntax for the function-domain checks that are currently baked in to the semantics. Refer to chapter 6.1.5 for a discussion.

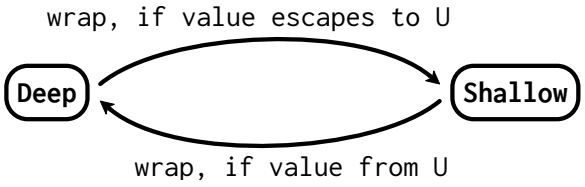


Figure 85: With an escape analysis, the deep–shallow boundaries could be weakened.

The Pycket compiler adds a JIT to Deep Typed Racket and significantly reduces the overhead of type boundaries [9, 10]. Adapting this backend to Shallow Racket may reduce costs immediately, without the need for an analysis. In the context of a simpler type system, Roberts et al. [84] report that a tracing JIT eliminates the cost of transient-inspired checks in Grace.

7.3 IMPROVING DEEP–TRANSIENT INTERACTION

The model in chapter 6 is safe, but makes deep types expensive. Every boundary to deep code gets protected with a wrap check (figure 66). For boundaries between deep and untyped this is no surprise, because the untyped code is unconstrained. For shallow code, though, static typing provides some checked claims; one would hope to get away with a less expensive check at the boundary. After all, closed programs that use only deep and shallow code need no checks in principle because every line of code is validated by the strong surface-language type checker.

One possible way to optimize is to weaken the boundary between deep and shallow. Deep can avoid wrapping an export if the value never interacts with untyped code going forward. Likewise, deep can trust an import if the value was never handled or influenced by untyped code. Figure 85 sketches the boundaries that could change via this strategy; the deep–untyped and shallow–untyped boundaries are unaffected. Note, however, that determining whether a value interacts with untyped code requires a careful analysis. Developing a correct analysis that runs quickly is a research challenge in itself.

A second possibility is to make the deep–shallow boundary a noop by delaying wrappers until a deep value reaches untyped code. Ideally, this strategy can work with an escape analysis to avoid wrapping untyped values that never reach deep code (figure 86). The challenge here is to design an escape analysis and to add wrapper-making code to shallow without losing the expressiveness that transient gains by avoiding wrappers altogether. For first-order interactions, Shallow can be careful about the identifiers that it sends to untyped code.

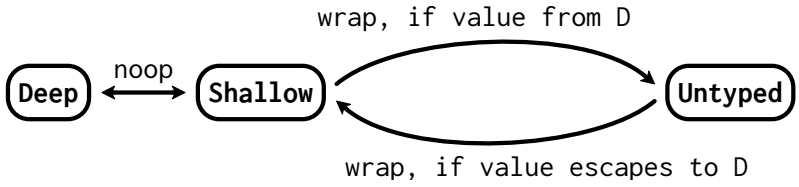


Figure 86: With an escape analysis and the ability to create wrappers in shallow code, all runtime type checks could be pushed to the boundaries with untyped code.

Higher-order communication is the real source of difficulties. For example, if shallow imports an untyped map function, then shallow must be prepared to wrap every function that it sends to map just in case one of the functions is deep-typed.

If a language can create wrappers in shallow code, however, then the Forgetful semantics (chapter 4) may be a better fit than Transient. Shallow types via Forgetful do not require shape checks throughout typed code, and the 1-level wrappers can dynamically cooperate with deep-wrapped values; that is, the interactions do not require a static analysis because the wrappers carry information.

A different approach is to adapt the idea of confined types [6]. If the type system can prove that a value originates in typed code and never escapes to untyped, then deep and shallow can freely share the value. In particular, a shallow function with a confined-type domain may not require any shape checks.

7.4 EVALUATE ALTERNATIVE SHAPE DESIGNS

The shape checks in Shallow Racket enforce full type constructors (chapter 5.1.1). Other designs are possible, though, and may lead to a better tradeoff between type guarantees and performance.

One direction is to strenghten the run-time checks to go beyond the outermost type constructor. Some designs may benefit from two or three levels of constructor checks. In the limit, a transient could enforce all first-order properties.

A second alternative is to weaken run-time checks for maximal performance. The current shapes check too much, in the sense that the Typed Racket optimizer cannot use all the information. For example, the Shallow check `procedure?` does not help any optimizations. If performance is the only concern, then an implementation can let the dynamically-typed runtime system handle function applications.

7.5 OTHER CHALLENGES

- In a performance lattice, an inspection of the configurations with exactly one typed unit can reveal the lack of fast paths through the lattice. Namely, if any of these bottom-level configurations suffer high overhead then a one-by-one conversion path is going to suffer similar overhead at some step. Perhaps there are other properties that can be predicted without exploring a full lattice. Gariano et al. [39], for example, suggest that transient slowdowns can be diagnosed by studying each typed unit individually.

- The performance evaluation method begins by toggling types at a certain granularity. The definition of granularity in chapter 3.2.2 does not allow for imprecise types such as `List(Dyn)` and `Function([Dyn], Str)`. Adapting the definition to such types would improve our understanding of prior work that randomly generates imprecise types [59, 116].

- Design a semantics, X, that eagerly checks pairs like the Natural semantics and wraps/unwraps functions like Forgetful. Prove that X does not satisfy complete monitoring, but can satisfy blame soundness and completeness. There may be an undiscovered variant of complete monitoring that distinguishes this X semantics from the basic Forgetful semantics, which may omit checks on the elements of a pair.

- Rephrase complete monitoring in semantic terms, using types and observable behaviors instead of syntax.

- The error preorder ($\lesssim$) looks like the term precision relation ($\sqsubseteq$) from the gradual typing literature [73, 87]. To investigate whether there is a deeper connection, use the Natural and Forgetful semantics to design two compilers into a core language that satisfies graduality. KafKa may be a good starting point [21]. Prove that the Forgetful compiler always gives less-precise expressions according to the term precision relation. Test whether core-language term precision can be used to indirectly prove the surface-language error preorder.

- Implement transient blame with multiple parents per link entry. For operations such as `hash-ref`, dynamically choose which parent to follow. A language of blame types may be necessary to guide choices. Measure the quality of errors and the performance cost that results from the extra bookkeeping.

- Build a method to help programmers find the best mixture of deep and shallow types in a codebase. Static analysis may suffice because the goal is to predict relative performance, not the

absolute cost [16]. Running the untyped configuration can provide data about the number of boundary-crossings that occur. Running the configurations with exactly one typed module may help predict the cost of interactions, especially for transient [39].

- The cost of a deep boundary depends heavily on its type, and an existential type is often cheaper than types that exposes internal details. Design a refactoring that converts a transparent typed API to use opaque existentials.

- Add erased types to the mix; determine what is needed for deep types, shallow types, and optional types to interact. The prior work on *like types*, which combines optional and concrete types, may be a useful guide [82, 123].

8 CONCLUSION

Deep and shallow types can interoperate, both in theory and in a practical implementation, and the synthesis brings measurable benefits. The benefits improve all three main dimensions of a mixed-typed programming:

- *Proofs*: Switching from shallow to deep types strengthens the formal guarantees for a block of code. In Typed Racket, a one-line change thus improves types from local spot-checks to claims that hold throughout the program, including in untyped modules.
- *Performance*: Flipping between deep and shallow can improve performance. In fully-typed programs, deep types have zero cost—and often run faster due to type-directed optimizations. In mixed programs, shallow avoids the tremendous overheads of deep type boundaries.
- *People*: Shallow types can express new combinations of typed and untyped code because they enforce weaker guarantees. Programmers can choose between this flexibility and the stability of deep types as they see fit, for each part of a codebase.

Integrating deep and shallow within one codebase—as opposed to picking one or the other—improves several concrete examples (chapter 6.3.2). These examples all use shallow types for code that is tightly coupled to an untyped boundary and deep types everywhere else. More experience is likely to reveal other patterns and best practices. For now, I recommend shallow types when initially converting an untyped program. Once the types are in place and the boundaries are clear, then moving from shallow to deep may assist with debugging tasks and may improve performance.

The foundations of this work are the methods that I developed to systematically measure mixed-typed languages.

1. The performance evaluation methods from (chapter 3) offer a comprehensive and scalable picture of run-time costs. An exhaustive method summarizes the complete dataset when feasible, and an approximate method gives an empirically-justified weakening otherwise.
2. The design evaluation method (chapter 4) rigorously assess the strengths and weaknesses of static types. Our application of this method leads to the most precise characterization of designs in the literature.

Overall, my dissertation brings us closer to useful mixed-typed languages. The step from *untyped-or-typed* to *mixed-typed* has presented a serious challenge to the conventional wisdom about static types. Standard techniques that realize strong guarantees and fast performance in a fully-typed setting yield weaker guarantees and slower running times in mixed programs. In the words of one anonymous Racket survey respondent, mixed languages "seemed to combine the best of both worlds .... but in practice seem to combine mainly the downsides" because of friction between static and dynamic typing. Methods and measurements have improved our understanding of the design space and articulated the benefits of mixing deep and shallow types to soften the edges. With both styles available, programmers can avoid severe performance and expressiveness issues. Yet much remains to be done, especially to see how programmers comprehend the new types and leverage the new choices.

A

APPENDIX

A.1 SAMPLE VALIDATION

The approximate evaluation method in chapter 3.3 uses simple random sampling to guess the proportion of D-deliverable configurations in a benchmark. Random sampling is statistically likely to yield an accurate and precise guess, and indeed figures 6 and 6 present correct and thin intervals. But sampling can go poorly. An unlucky guess based on $r = 10$ samples that each contain $s = 10 * N$ of the absolute-fastest configurations is overly optimistic. The figures in this section double-check our earlier results with additional samples and suggest that precise intervals are indeed the norm.

These validation results use new, randomly-generated random samples. In this section, one *sampling experiment* for a benchmark with N modules is a collection of $r = 10$ samples that each contain $s = 10 * N$ configurations chosen uniformly at random without replacement. The question is whether the sampling experiments, as a whole, are typically accurate. One test of accuracy is to measure the average distance from sample conclusions to the truth. More precisely, for 200 evenly-spaced values of D between 1 and 20, the first test compares the true proportion of D-deliverable configurations to average distance across all upper and lower-bound guesses generated from each sample experiment. A second test is to validate each upper and lower bound individually by taking its distance and negating the number if the guess is in the wrong direction.

Figure 87 counts the average distance from an approximation to the truth across 800 sampling experiments. If the samples are accurate, these distances should be close to zero. And indeed, the worst average distance is 4% away from the true proportion of D-deliverable configurations.

Figure 88 presents data on both the accuracy and correctness of 800 different sampling experiments. For each individual guess, the data records both its distance from the truth and whether the direction is correct. Ideally, every guess should end up with a small positive number. The figure is slightly worse, but still good. Most guesses fall within 0% and 5% from the truth. The few bad guesses end up with a worst case of 9% off in the correct direction and -4% off in the misleading one.

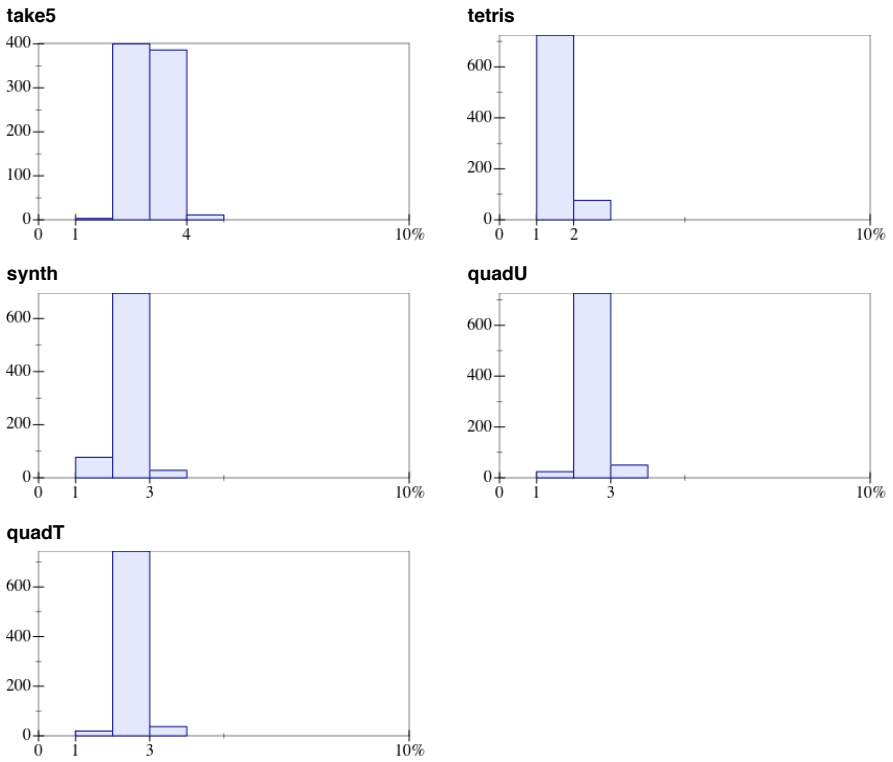


Figure 87: Average distance from the true proportion of D-deliverable configurations across 800 approximate intervals.

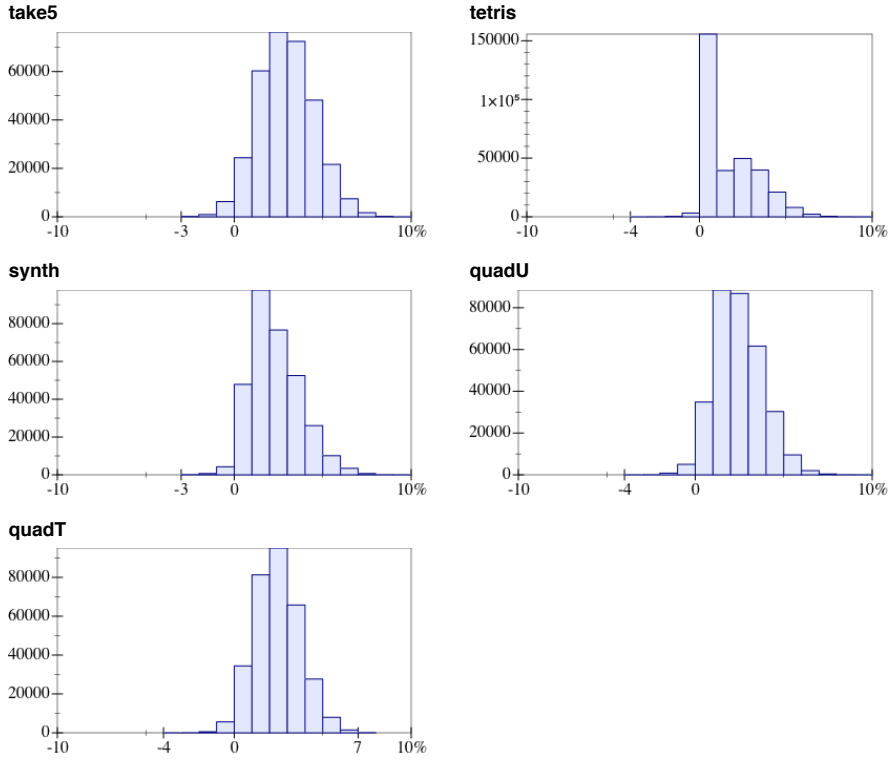


Figure 88: Exact difference between the true proportion of D-deliverable configurations and the bounds at each point along 800 approximate intervals. A negative number reflects a misleading upper or lower bound.

A.2 DEEP VS. SHALLOW OVERHEAD

Figures 89, 90, and 91 compare deep and shallow types to the extreme. Whereas the plots in chapter 5.4.2 stop at 20x overhead, these go out to the worst-case overhead.

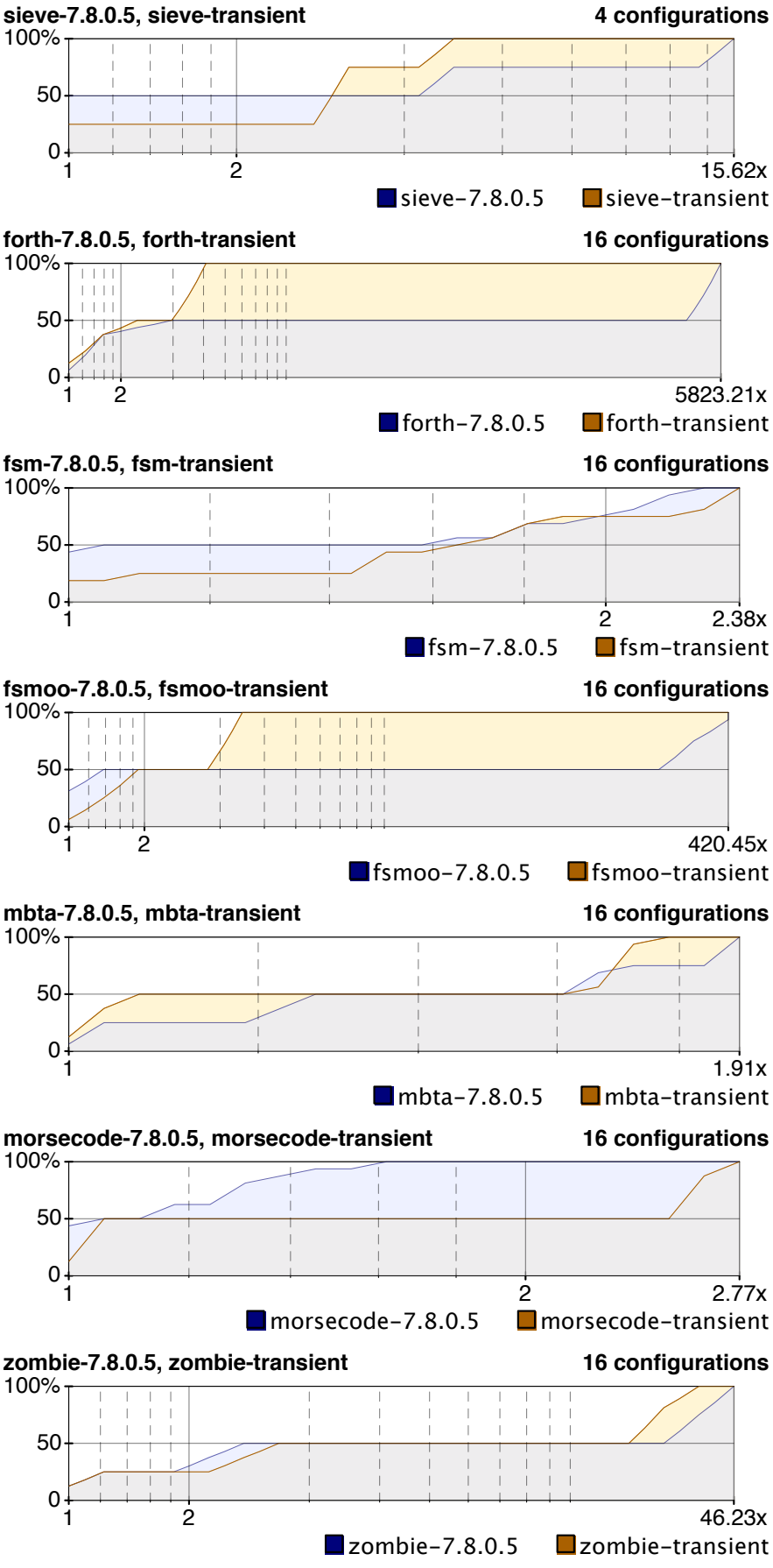


Figure 89: Deep vs. Shallow (1/3).

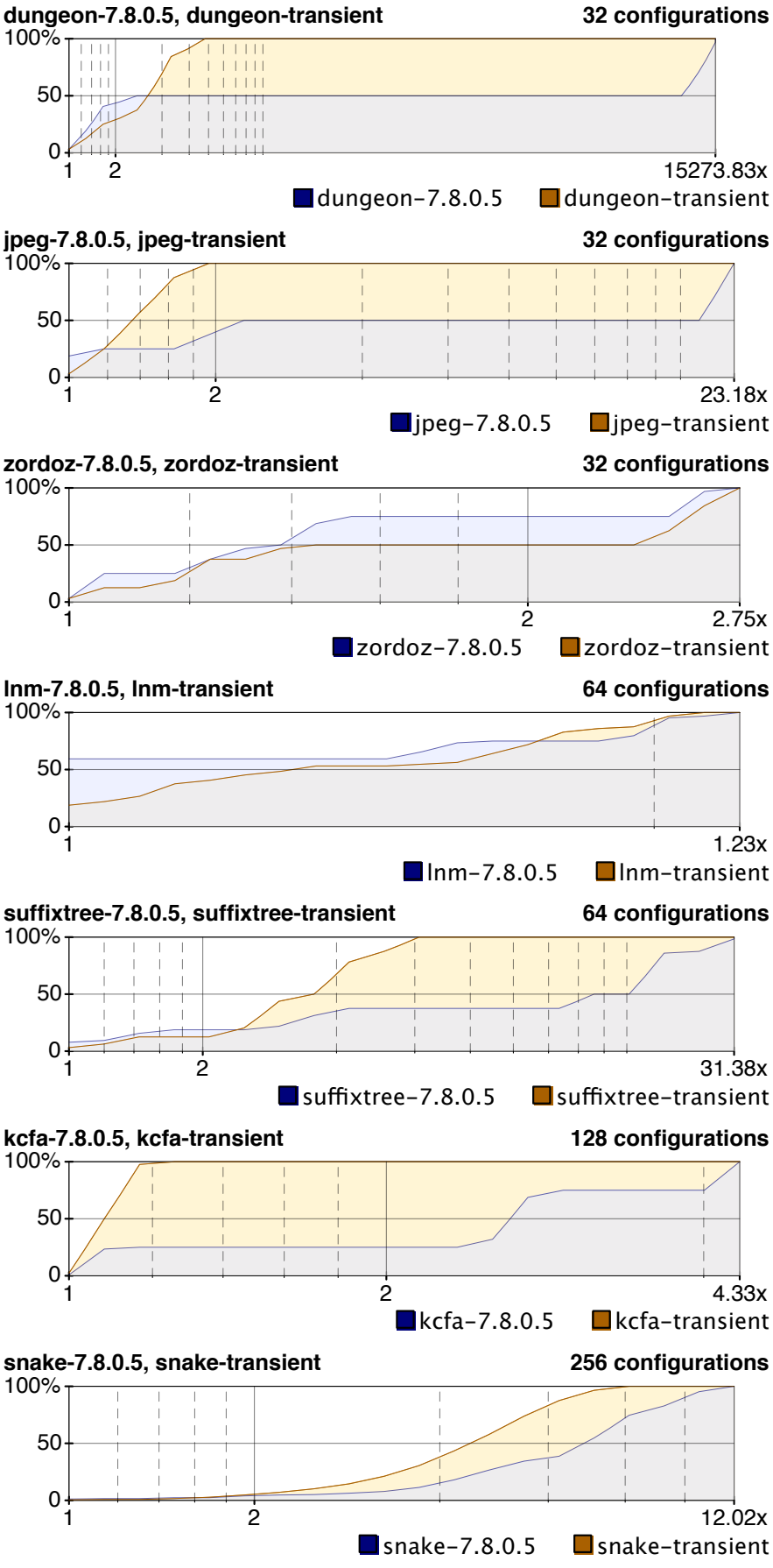


Figure 90: Deep vs. Shallow (2/3).

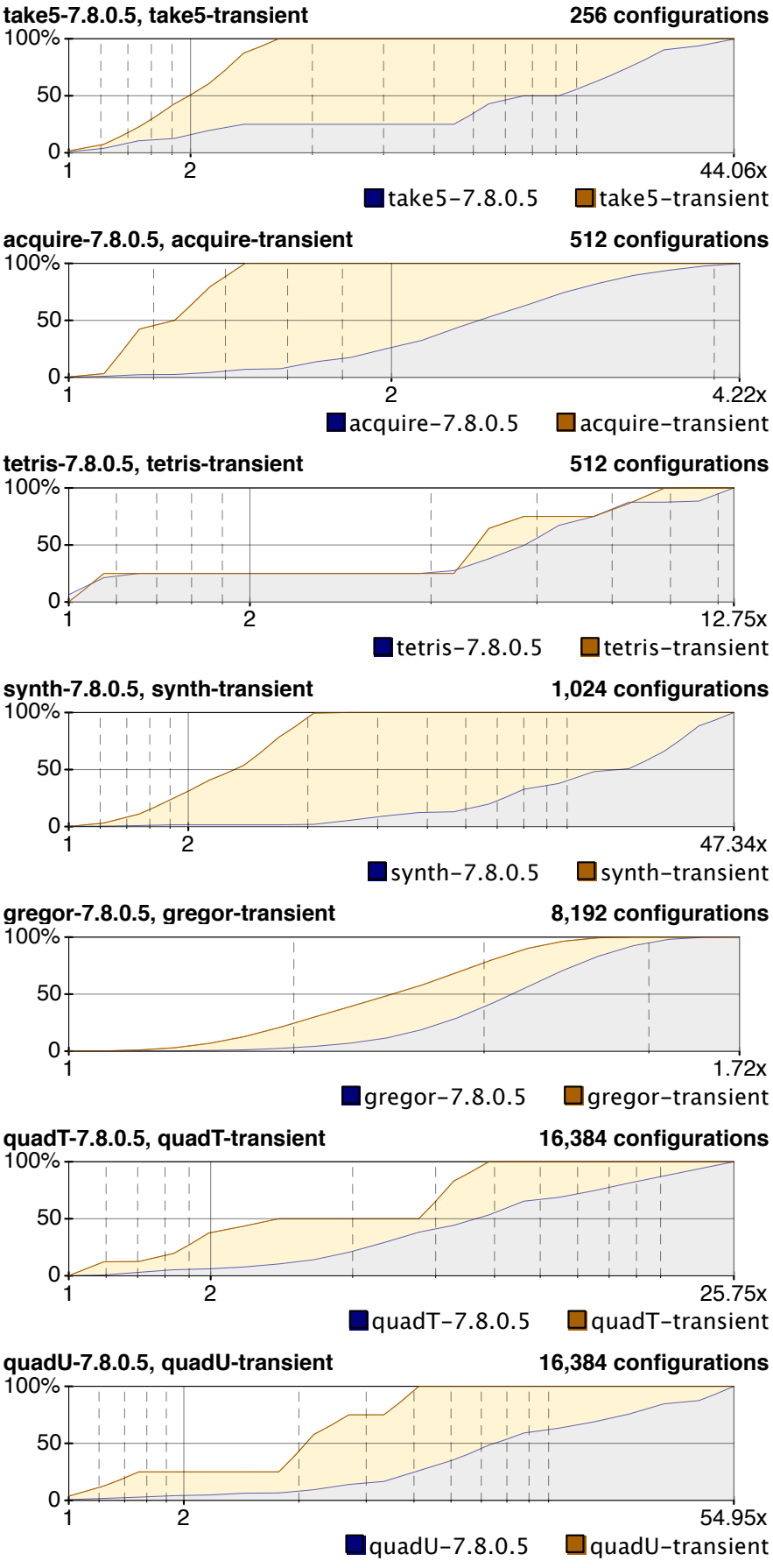


Figure 91: Deep vs. Shallow (3/3).

$$\frac{\Gamma \vdash_s s_0 : \tau_0 \qquad \Gamma \vdash_s s_1 : \tau_1}{\Gamma \vdash_s \langle s_0, s_1 \rangle : \tau_0 \times \tau_1} \qquad \frac{\Gamma \vdash_s s_0 : \lfloor \tau_0 \rfloor \qquad \Gamma \vdash_s s_1 : \lfloor \tau_1 \rfloor}{\Gamma \vdash_s \langle s_0, s_1 \rangle : \lfloor \tau_0 \times \tau_1 \rfloor}$$

$$\frac{\Gamma \vdash_s s_0 : \mathcal{U}}{\Gamma \vdash_s \mathit{unop}\, s_0 : \mathcal{U}} \qquad \frac{\Gamma \vdash_s e_0 : \lfloor \tau_0 \rfloor \qquad \Delta(\mathit{unop}, \tau_0) = \tau_1}{\Gamma \vdash_s \mathit{unop}\, e_0 : \lfloor \tau_1 \rfloor}$$

$$\frac{\Gamma \vdash_s s_0 : \mathcal{U} \qquad \Gamma \vdash_s s_1 : \mathcal{U}}{\Gamma \vdash_s \mathit{binop}\, s_0\, s_1 : \mathcal{U}} \qquad \frac{\Gamma \vdash_s e_0 : \tau_0 \qquad \Gamma \vdash_s e_1 : \tau_1 \qquad \Delta(\mathit{binop}, \tau_0, \tau_1) = \tau_2}{\Gamma \vdash_s \mathit{binop}\, e_0\, e_1 : \tau_2}$$

$$\frac{\Gamma \vdash_s s_0 : \mathcal{U} \qquad \Gamma \vdash_s s_1 : \mathcal{U}}{\Gamma \vdash_s \mathsf{app}\{s_0\}\, s_1 \ : \mathcal{U}} \qquad \frac{\Gamma \vdash_s e_0 : \tau_0 \Rightarrow \tau_1 \qquad \Gamma \vdash_s e_1 : \tau_0}{\Gamma \vdash_s \mathsf{app}\{e_0\}\, e_1 \ : \tau_1}$$

$$\frac{\Gamma \vdash_s e_0 : \tau_0 \qquad \tau_0 <: \tau_1}{\Gamma \vdash_s e_0 : \tau_1}$$

Figure 92: Additional surface typing rules

A.3 MISSING RULES

Figure 92 and figure 93 present unremarkable typing and completion rules that are omitted from chapter 6.1. Note that the completion rules depend on the full typing derivation of a term to uncover, for example, the types of the arguments to a binary operation. The model in chapter 4 avoids this technicality by asking for annotations on elimination forms.

$$\frac{}{\Gamma \vdash_s i_0 : \mathcal{U} \rightsquigarrow i_0} \qquad \frac{}{\Gamma \vdash_s i_0 : \tau_0 \rightsquigarrow i_0} \qquad \frac{}{\Gamma \vdash_s i_0 : \lfloor \tau_0 \rfloor \rightsquigarrow i_0}$$

$$\frac{\Gamma \vdash_s e_0 : \mathcal{U} \rightsquigarrow e_2 \quad \Gamma \vdash_s e_1 : \mathcal{U} \rightsquigarrow e_3}{\Gamma \vdash_s \langle e_0, e_1 \rangle : \mathcal{U} \rightsquigarrow \langle e_2, e_3 \rangle} \qquad \frac{\Gamma \vdash_s e_0 : \tau_0 \rightsquigarrow e_2 \quad \Gamma \vdash_s e_1 : \tau_1 \rightsquigarrow e_3}{\Gamma \vdash_s \langle e_0, e_1 \rangle : \tau_0 \times \tau_1 \rightsquigarrow \langle e_2, e_3 \rangle}$$

$$\frac{\Gamma \vdash_s e_0 : \lfloor \tau_0 \rfloor \rightsquigarrow e_2 \quad \Gamma \vdash_s e_1 : \lfloor \tau_1 \rfloor \rightsquigarrow e_3}{\Gamma \vdash_s \langle e_0, e_1 \rangle : \lfloor \tau_0 \times \tau_1 \rfloor \rightsquigarrow \langle e_2, e_3 \rangle} \qquad \frac{\Gamma \vdash_s e_0 : \mathcal{U} \rightsquigarrow e_1}{\Gamma \vdash_s \mathit{unop}\, e_0 : \mathcal{U} \rightsquigarrow \mathit{unop}\, e_1}$$

$$\frac{\Gamma \vdash_s e_0 : \tau_0 \times \tau_1 \rightsquigarrow e_1}{\Gamma \vdash_s \mathit{unop}\, e_0 : \tau_0 \rightsquigarrow \mathit{unop}\, e_1} \qquad \frac{\Gamma \vdash_s e_0 : \lfloor \tau_0 \times \tau_1 \rfloor \rightsquigarrow e_1 \quad \mathit{shape}\,(\tau_0) = \sigma_0}{\Gamma \vdash_s \mathit{unop}\, e_0 : \lfloor \tau_0 \rfloor \rightsquigarrow \mathsf{scan}\, \sigma_0\, \mathsf{fst}\, e_1}$$

$$\frac{\Gamma \vdash_s e_0 : \mathcal{U} \rightsquigarrow e_2 \quad \Gamma \vdash_s e_1 : \mathcal{U} \rightsquigarrow e_3}{\Gamma \vdash_s \mathit{binop}\, e_0\, e_1 : \mathcal{U} \rightsquigarrow \mathit{binop}\, e_2\, e_3} \qquad \frac{\Gamma \vdash_s e_0 : \tau_0 \rightsquigarrow e_2 \quad \Gamma \vdash_s e_1 : \tau_1 \rightsquigarrow e_3}{\Gamma \vdash_s \mathit{binop}\, e_0\, e_1 : \tau_2 \rightsquigarrow \mathit{binop}\, e_2\, e_3}$$

$$\frac{\Gamma \vdash_s e_0 : \lfloor \tau_0 \rfloor \rightsquigarrow e_2 \quad \Gamma \vdash_s e_1 : \lfloor \tau_1 \rfloor \rightsquigarrow e_3}{\Gamma \vdash_s \mathit{binop}\, e_0\, e_1 : \lfloor \tau_2 \rfloor \rightsquigarrow \mathit{binop}\, e_2\, e_3}$$

Figure 93: Additional surface-to-evaluation completion rules

Benchmark	Deep %	Shallow %	D. or S. %
sieve	0	0	100
forth	0	0	50
fsm	100	100	100
fsmoo	0	0	50
mbta	100	100	100
morsecode	100	100	100
zombie	0	0	50
dungeon	0	0	67
jpeg	0	100	100
zordoz	100	100	100
lnm	100	100	100
suffixtree	0	0	12
kcfa	33	100	100
snake	0	0	0
take5	0	100	100

Figure 94: Percent of 3-deliverable paths in three lattices: the deep-typed lattice, the shallow-typed lattice, and a hybrid that chooses the best of deep or shallow types at each point.

A.4 MORE EVIDENCE FOR DEEP AND SHALLOW

A.4.1 *Migration Paths*

Shallow types make step-by-step migration more practical in Typed Racket. Originally, with deep types, a programmer who adds types one module at a time is likely to hit a performance wall; that is, a few configurations along the migration path are likely to suffer a large overhead. Adding more deep types is a sure way to reduce the overhead, especially if the programmer adds the best-possible types (figure 21), but these multi-step pitfalls contradict the promise of migratory typing. High overhead makes it hard to tell whether the new types are compatible with the rest of the codebase.

By choosing deep or shallow types at each point along a path, the worst-case overhead along migration paths goes down. Figure 94 quantifies the improvement by showing the percent of all paths that are 3-deliverable at each step. With deep types alone, all paths in nine benchmarks hit a point that exceeds the 3x limit. With shallow types alone, all paths in seven benchmarks exceed the limit as well. With the mix, however, only one benchmark (snake) has zero 3-deliverable paths. Fine-grained combinations of deep and shallow types can further improve the number of viable migration paths. In fsm, for exam-

ple, every path is 1.1-deliverable if the programmer picks the fastest-running mix of deep and shallow types for each configuration.

A.4.2 *Case Study: GTP Benchmarks*

For six small benchmarks, I measured the full space of 3^N configurations that can arise by combining deep and shallow types. Each configuration ran successfully, affirming that deep and shallow can interoperate. Furthermore, a surprising percent of all 2^N mixed-typed configurations in each benchmark ran fastest using a mixture of deep and shallow types:

- 37.50% of fsm configurations;
- 25.00% of morsecode configurations;
- 37.50% of jpeg configurations;
- 55.47% of kcfa configurations;
- 6.25% of zombie configurations; and
- 46.88% of zordoz configurations.

In fsm, for example, there are sixteen mixed-typed configurations. Five of these cannot mix deep and shallow because they contain at most one typed module. Of the remaining 11 configurations, over half run fastest with a combination of deep and shallow types.

BIBLIOGRAPHY

[1] Martin Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. Dynamic typing in a statically typed language. *TOPLAS*, 13(2):237–268, 1991.

[2] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *POPL*, pages 201–214, 2011.

[3] Alexander Aiken and Brian R. Murphy. Static type inference in a dynamically typed language. In *POPL*, pages 279–290, 1991.

[4] Alexander Aiken, Edward L. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *POPL*, pages 163–173, 1994.

[5] Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, 96(1):52–69, 2013.

[6] Esteban Allende, Johan Fabry, Ronald Garcia, and Éric Tanter. Confined gradual typing. In *OOPSLA*, pages 251–270, 2014.

[7] Deyaaeldeen Almahallawi. *Towards Efficient Gradual Typing via Monotonic References and Coercions*. PhD thesis, Indiana University, 2020.

[8] Kenneth R. Anderson and Duane Rettig. Performing Lisp analysis of the FANNKUCH benchmark. *ACM SIGPLAN Lisp Pointers*, 7(4):2–12, 1994.

[9] Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfield, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. Pycket: A tracing JIT for a functional language. In *ICFP*, pages 22–34, 2015.

[10] Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. Sound gradual typing: only mostly dead. *PACMPL*, 1(OOPSLA):54:1–54:24, 2017.

[11] Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to C#. In *ECOOP*, pages 76–100, 2010.

[12] Gavin Bierman, Martin Abadi, and Mads Torgersen. Understanding TypeScript. In *ECOOP*, pages 257–281, 2014.

[13] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad.

Thorn: Robust, concurrent, extensible scripting on the JVM. In *OOPSLA*, pages 117–136, 2009.

[14] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical optional types for Clojure. In *ESOP*, pages 68–94, 2016.

[15] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *OOPSLA*, pages 215–230, 1993.

[16] John Peter Campora, Sheng Chen, and Eric Walkingshaw. Casts and costs: Harmonizing safety and performance in gradual typing. *PACMPL*, 2(ICFP):98:1–98:30, 2018.

[17] Robert Cartwright. User-defined types as an aid to verifying LISP programs. In *ICALP*, pages 228–256, 1976.

[18] Giuseppe Castagna and Victor Lanvin. Gradual typing with union and intersection types. *PACMPL*, 1(ICFP):41:1–41:28, 2017.

[19] Guiseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. Gradual typing: A new perspective. *PACMPL*, 3(POPL):16:1–16:32, 2019.

[20] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levy. Fast and precise type checking for JavaScript. *PACMPL*, 1(OOPSLA):56:1–56:30, 2017.

[21] Benjamin W. Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek. KafKa: Gradual typing for objects. In *ECOOP*, pages 12:1–12:23, 2018.

[22] Ryan Culpepper, Sam Tobin-Hochstadt, and Matthew Flatt. Advanced macrology and the implementation of Typed Scheme. In *SFP. Université Laval, DIUL-RT-0701*, pages 1–14, 2007.

[23] Dart. The Dart type system, 2020. URL `https://dart.dev/guides/language/type-system`. Accessed 2020-09-04.

[24] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: No more scapegoating. In *POPL*, pages 215–226, 2011.

[25] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In *ESOP*, pages 214–233, 2012.

[26] Christos Dimoulas, Max S. New, Robert Bruce Findler, and Matthias Felleisen. Oh lord, please don't let contracts be misunderstood (functional pearl). In *ICFP*, pages 117–131, 2016.

[27] Mike Fagan. *Soft Typing: An Approach to Type Checking for Dynamically Typed Languages*. PhD thesis, Rice University, 1992.

[28] Daniel Feltey. Gradual typing for first-class modules. Master's thesis, Northeastern University, 2015.

[29] Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. Collapsible contracts: Fixing a pathology of gradual typing. *PACMPL*, 2 (OOPSLA):133:1–133:27, 2018.

[30] E.C. Fieller. Some problems in interval estimation. *Journal of the Royal Statistical Society*, 16(2):175–185, 1957.

[31] Robert Bruce Findler. *Behavioral Software Contracts*. PhD thesis, Rice University, 2002.

[32] Robert Bruce Findler and Matthias Blume. Contracts as pairs of projections. Technical Report TR-2006-01, University of Chicago, 2006.

[33] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP*, pages 48–59, 2002.

[34] Cormac Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, 1997.

[35] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *PLDI*, pages 235–248, 1997.

[36] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *PLDI*, pages 23–32, 1996.

[37] Matthew Flatt. Compilable and composable macros: You want it *when?* In *ICFP*, pages 72–83, 2002.

[38] Richard P. Gabriel. *Performance and evaluation of LISP systems*. MIT Press, 1985.

[39] Isaac Oscar Gariano, Richard Roberts, Stefan Marr, Michael Homer, and James Noble. Which of my transient type checks are not (almost) free? In *VMIL*, pages 58–66, 2019.

[40] Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA*, pages 231–245, 2005.

[41] Michael Greenberg. Space-efficient manifest contracts. In *POPL*, pages 181–194, 2015.

[42] Michael Greenberg. The dynamic practice and static theory of gradual typing. In *SNAPL*, pages 6:1–6:20, 2019.

[43] Ben Greenman and Matthias Felleisen. A spectrum of type soundness and performance. *PACMPL*, 2(ICFP):71:1–71:32, 2018.

[44] Ben Greenman and Zeina Migeed. On the cost of type-tag soundness. In *PEPM*, pages 30–39, 2018.

[45] Ben Greenman, Matthias Felleisen, and Christos Dimoulas. Complete monitors for gradual types. *PACMPL*, 3(OOPSLA): 122:1–122:29, 2019.

[46] Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. How to evaluate the performance of gradual type systems. *JFP*, 29(e4): 1–45, 2019.

[47] Ben Greenman, Christos Dimoulas, and Matthias Felleisen. How to evaluate the semantics of gradual types. *Submitted for publication*, 2020.

[48] Hugo Musso Gualandi and Roberto Ierusalimschy. Pallene: a companion language for Lua. *Science of Computer Programming*, 189(102393):1–15, 2020.

[49] Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric polymorphic contracts. In *DLS*, pages 29–40, 2007.

[50] Christopher T. Haynes. Infer: A statically-typed dialect of Scheme. Technical Report Technical Report 367, Indiana University, 1995.

[51] Fritz Henglein. Global tagging optimization by type inference. In *LFP*, pages 205–215, 1992.

[52] Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, 1994.

[53] Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for a dynamically typed language: Translating Scheme to ML. In *FPCA*, pages 192–203, 1995.

[54] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *HOSC*, 23(2):167–189, 2010.

[55] Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. On polymorphic gradual typing. *PACMPL*, 1(ICFP):40:1–40:29, 2017.

[56] Matthias Keil and Peter Theimann. Blame assignment for higher-order contracts with intersection and union. In *ICFP*, pages 375–386, 2015.

[57] Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann. Transparent object proxies in JavaScript. In *ECOOP*, pages 149–173, 2015.

[58] Andrew M. Kent. *Advanced Logical Type Systems for Untyped Languages*. PhD thesis, Indiana University, 2019.

[59] Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. Toward efficient gradual typing for structural types via coercions. In *PLDI*, pages 517–532, 2019.

[60] Lukas Lazarek, Alexis King, Samanvitha Sundar, Robert Bruce Findler, and Christos Dimoulas. Does blame shifting work? *PACMPL*, 4(POPL):65:1–65:29, 2020.

[61] Erwan Lemonnier. Pluto: or how to make Perl juggle with billions, 2006. URL `http://erwan.lemonnier.se/talks/pluto.html`. Accessed 2020-08-25.

[62] Xavier Leroy and Michael Mauny. Dynamics in ML. In *FPCA*, pages 406–426, 1991.

[63] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710, 1966.

[64] Andre Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. A formalization of Typed Lua. In *DLS*, pages 13–25, 2015.

[65] Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *TOPLAS*, 31(3):1–44, 2009.

[66] Philippe Meunier. *Modular Set-Based Analysis from Contracts*. PhD thesis, Northeastern University, 2006.

[67] Philippe Meunier, Robert Bruce Findler, Paul Steckler, and Mitchell Wand. Selectors make set-based analysis too hard. *HOSC*, 18:245–269, 2005.

[68] Philippe Meunier, Robert Bruce Findler, and Matthias Felleisen. Modular set-based analysis from contracts. In *POPL*, pages 218–231, 2006.

[69] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[70] David A. Moon. MACLISP reference manual, Revision 0. Technical report, MIT Project MAC, 1974.

[71] Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong. Extensible access control with authorization contracts. In *OOPSLA*, pages 214–233, 2016.

[72] Fabian Muehlboeck and Ross Tate. Sound gradual typing is nominally alive and well. *PACMPL*, 1(OOPSLA):56:1–56:30, 2017.

[73] Max S. New, Daniel R. Licata, and Amal Ahmed. Gradual type theory. *PACMPL*, 3(POPL):15:1–15:31, 2019.

[74] Max S. New, Dustin Jamner, and Amal Ahmed. Graduality and parametricity: together again for the first time. *PACMPL*, 4 (POPL):46:1–46:32, 2020.

[75] Linh Chi Nguyen and Luciano Andreozzi. Tough behavior in the repeated bargaining game. A computer simulation study. *EAI Endorsed Trans. Serious Games*, 3(8):e5, 2016.

[76] Atsushi Ohori and Kazuhiko Kato. Semantics for communication primitives in a polymorphic language. In *POPL*, pages 99–112, 1993.

[77] Kent M. Pittman. The revised MACLISP manual. Technical Report MIT/LCS/TR-295, MIT Laboratory for Computer Science, 1983.

[78] Norman Ramsey. Embedding an interpreted language using higher-order functions and types. *JFP*, 21(6):585–615, 2008.

[79] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for TypeScript. In *POPL*, pages 167–180, 2015.

[80] Type reconstruction for variable-arity procedures. Dzeng, hsianlin and haynes, christopher t. In *LFP*, pages 239–249, 1994.

[81] Brianna M. Ren, John Toman, T. Stephen Strickland, and Jeffrey S. Foster. The Ruby type checker. In *SAC*, pages 1565–1572, 2013.

[82] Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete types for TypeScript. In *ECOOP*, pages 76–100, 2015.

[83] Gregor Richards, Ellen Arteca, and Alexi Turcotte. The vm already knew that: Leveraging compile-time knowledge to optimize gradual typing. *PACMPL*, 1(OOPSLA):55:1–55:27, 2017.

[84] Richard Roberts, Stefan Marr, Michael Homer, and James Noble. Transient typechecks are (almost) free. In *ECOOP*, pages 15:1–15:29, 2019.

[85] Jeremy Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic references for efficient gradual typing. In *ESOP*, pages 432–456, 2015.

[86] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *SFP. University of Chicago, TR-2006-06*, pages 81–92, 2006.

[87] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *SNAPL*, pages 274–293, 2015.

[88] Vincent St-Amour. *How to Generate Actionable Advice About Performance Problems*. PhD thesis, Northeastern University, 2015.

[89] Vincent St-Amour and Neil Toronto. Experience report: Applying random testing to a base type environment. In *ICFP*, pages 351–356, 2013.

[90] Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. Optimization coaching: Optimizers learn to communicate with programmers. In *OOPSLA*, pages 163–178, 2012.

[91] Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, and Matthias Felleisen. Typing the numeric tower. In *PADL*, pages 289–303, 2012.

[92] Guy L. Steele, Jr. *Common Lisp*. Digital Press, 2nd edition, 1990.

[93] T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. Practical variable-arity polymorphism. In *ESOP*, pages 32–46, 2009.

[94] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: Runtime support for reasonable interposition. In *OOPSLA*, pages 943–962, 2012.

[95] Nikhil Swamy, Cédric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual typing embedded securely in JavaScript. In *POPL*, pages 425–437, 2014.

[96] Asumu Takikawa. *The Design, Implementation, and Evaluation of a Gradual Type System for Dynamic Class Composition*. PhD thesis, Northeastern University, 2016.

[97] Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *OOPSLA*, pages 793–810, 2012.

[98] Asumu Takikawa, T. Stephen Strickland, and Sam Tobin-Hochstadt. Constraining delimited control with contracts. In *ESOP*, pages 229–248, 2013.

[99] Asumu Takikawa, Daniel Feltey, Earl Dean, Robert Bruce Findler, Matthew Flatt, Sam Tobin-Hochstadt, and Matthias Felleisen. Towards practical gradual typing. In *ECOOP*, pages 4–27, 2015.

[100] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *POPL*, pages 456–468, 2016.

[101] Satish Thatte. Quasi-static typing. In *POPL*, pages 367–381, 1990.

[102] Sam Tobin-Hochstadt. *Typed Scheme: From Scripts to Programs*. PhD thesis, Northeastern University, 2010.

[103] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *DLS*, pages 964–974, 2006.

[104] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *POPL*, pages 395–406, 2008.

[105] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *ICFP*, pages 117–128, 2010.

[106] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *PLDI*, pages 132–141, 2011.

[107] Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. Migratory typing: Ten years later. In *SNAPL*, pages 17:1–17:17, 2017.

[108] Matías Toro, Elizabeth Labrada, and Éric Tanter. Gradual parametricity, revisited. *PACMPL*, 3(POPL):17:1–17:30, 2019.

[109] Preston Tunnell Wilson, Ben Greenman, Justin Pombrio, and Shriram Krishnamurthi. The behavior of gradual types: a user study. In *DLS*, pages 1–12, 2018.

[110] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[111] Tom Van Cutsem and Mark S Miller. Trustworthy proxies. In *ECOOP*, pages 154–178, 2013.

[112] Bill Venners. Twitter on Scala, 2009. URL `https://www.artima.com/scalazine/articles/twitter_on_scala.html`. Accessed 2020-08-25.

[113] Michael M. Vitousek. *Gradual Typing for Python, Unguarded*. PhD thesis, Indiana University, 2019.

[114] Michael M. Vitousek, Andrew Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for Python. In *DLS*, pages 45–56, 2014.

[115] Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big types in little runtime: Open-world soundness and collaborative blame for gradual type systems. In *POPL*, pages 762–774, 2017.

[116] Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. Optimizing and evaluating transient gradual typing. In *DLS*, pages 28–41, 2019.

[117] Philip Wadler. A complement to blame. In *SNAPL*, pages 309–320, 2015.

[118] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *ESOP*, pages 1–15, 2009.

[119] Mitchell Wand. A semantic prototyping system. In *CC*, pages 213–221, 1984.

[120] Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski. Mixed messages: Measuring conformance and non-interference in TypeScript. In *ECOOP*, pages 28:1–28:29, 2017.

[121] Andrew K. Wright. *Practical Soft Typing*. PhD thesis, Rice University, 1994.

[122] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, pages 38–94, 1994. First appeared as Technical Report TR160, Rice University, 1991.

[123] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *POPL*, pages 377–388, 2010.