GRADUAL TYPING FOR PYTHON, UNGUARDED

Michael M. Vitousek

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the School of Informatics, Computing, and Engineering
Indiana University
May 2019

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

Jeremy G. Siek, Ph.D.

Sam Tobin-Hochstadt, Ph.D.

Amr Sabry, Ph.D.

Lawrence S. Moss, Ph.D.

September 5, 2018

For Ani.

Acknowledgements

In some sense I'm acknowledging the people who have made me who I am today—that's the kind of dominant role that grad school has in life, or at least in mine. This dissertation is a story about gradual typing, but it's also an artifact of me growing up: of the transition from young adulthood to... well, whatever it is that comes next. As such, there's a lot of emotions instilled in this document, and producing it took the support—intellectual, social, emotional—of so many other people. Mentioning them all is, of course, impossible. But I'm gonna try and thank a few.

The most important professional mentor I've had is of course my advisor, the inimitable Jeremy G. Siek. Starting from when I emailed him my undergrad senior thesis and asked him if he was looking for students (he replied with thoughtful suggestions on how to improve my thesis and an invitation to come meet with him), to my first couple years of training and instruction at the University of Colorado, and through my more independent research at Indiana, Jeremy has consistently supported me and helped me succeed. His "evil hat" has driven many thoughtful critiques of my work that have vastly strengthened the end result, and his patience and warmth have made his office a comfortable place to express my thoughts and ideas. Maybe most importantly, he's adapted his style for every stage of my career with him: spending a ton of time getting me up to speed when I started out, to working closely together on new ideas as I began to stretch my wings in research, to providing hands-off support as I developed my own ideas and my own program. I've never felt seriously neglected nor have I felt that my freedom to explore was limited, and that balance has been deeply important to my development as a scientist. I'm looking forward to having Jeremy as a colleague and as a friend in the future.

Of course, Jeremy isn't alone in shaping my career. Similarly important have been my outstanding research committee as well as many other faculty members who I've worked with. Sam Tobin-Hochstadt has been a thoughtful, detailed, and enjoyable mentor to have, and his perspective has always improved

my work and clarified my thinking. Beyond gradual typing, Amr Sabry's comments and questions over the years have made me a better researcher, and Larry Moss has helped me shape and target this dissertation to be as effective as I can make it. Back at University of Colorado, Evan Chang played a critical role in my first years in graduate school, and is a lot of why, even as my research has focused on language design, I still approach a lot of problems through the lens of static analysis. Finally, Liz Bradley's *Intro to the PhD Program* class at Colorado played a crucial role in helping me navigate the challenges of graduate school, and I'm grateful for how she helped me grow a foundation of research skills that I've built upon.

A few other mentors in my academic life come to mind, who I can thank (or, perhaps, blame?) for putting me on this path. Willamette University's computer science department was a formative place for me—and how could I not end up as a programming languages researcher after I went to a department with a total of four faculty, two of whom were PL folks? John Lasseter was the first professor with whom I had a real rapport for research, and working with him on undergraduate research solidified my interest in the study of programming languages. He is also one of the kindest people I know, and working with him was always rewarding. Fritz Ruehr further guided me as I approached graduate school and helped me navigate through my senior thesis, graduate school applications, and health complications. My other undergraduate advisor from my history degree also shaped who I am as a researcher: this thesis doesn't often touch on, say, Cold War politics or political transformations in the Eastern Bloc, but Bill Smaldone helped develop my critical thinking and writing skills more than almost anyone else. Finally, I have to mention my first computer science instructor, Gunn High School's Josh Paley. His wit and genuine kindness made him as fantastic of a teacher as I can imagine, and I was probably cursed to be a programming language researcher from the beginning, given that his first classes covered both Scheme and Java and encouraged us to think about the differences between them.

Of course, it isn't just teachers, professors, and advisers that form the groundwork for success in a graduate program. Equally important to me have been my friends and colleagues in my programs, both at Indiana and Colorado, who have both been excellent professional colleagues and have formed a personal support system that I've always been able to rely on. There are, of course, too many folks to mention, but I'll try to call out a few. At Colorado, I owe a lot to friends like Mario (BRADDAH); Daniel and Angela (thanks for turning me on to homebrewing, a hobby that also probably is responsible for me surviving graduate school); Anshul and Mariah (and now Liara!); Hal, Andy, and the rest of the sadly defunct Distinguished Cooking Series crew; and Shashank and Jonathan, my first lab-mates in the program. At Indiana, I want to mention Chris (a fellow CU ex-pat whose clever ideas are even bigger than his dog), Mike and Tori, Andre and Laura, Matthew, Kyle, Jaime, Spenser, Rajan, and many other fantastic drinking buddies; and the outstanding gradual typing crew, equal parts friends and research colleagues, Matteo, Andrew, Sarah, David, Caner, and Ambrose (among others mentioned elsewhere). I especially have to mention though my dude Cameron Swords, who has been as close of a friend and as outstanding of a colleague as anyone could ask for. He and his wife Rebecca—a similarly awesome person—have been some of my closest friends in Bloomington ever since the day Cam cold-emailed me after a homotopy type theory class and demanded we get a drink together sometime. We've been coauthors on a top tier paper, a bunch of delicious fermented beverages, and a bizarre role-playing game or two, and when we work together on anything the result is always over-the-top and awesome.

Naturally, funding support has been essential in completing this work, and there are several institutions and organizations that have helped me significantly along the way. Fellowships from the University of Colorado made my funding situation for my first two years very straightforward, and I'm honored and grateful for the support they granted me. The National Science Foundation funded most of the latter part of my degree, as did funds from Indiana University itself. Google's Lime Scholarship provided me with important travel funding throughout my degree, and a generous grant from Facebook enabled me

to finish this dissertation by funding my last year in the program. I'm grateful for and humbled by the support from each of these institutions.

The last and most important people I want to mention are my family.

The best thing that graduate school did for me, and the reason I'm so grateful that I moved to Indiana with Jeremy, was giving me the opportunity to meet "my person," my fiancée Amanda. She's been the kind of friend and partner that I wonder how I ever did without. Whether we're discovering an awesome new brewery, hiking in Brown County or the California Coast Range, playing video games on a snowy day, or just spending time with each other, she makes me a better person, and it's with her that I first really started to think about the future. I'm so excited for the rest of our lives, and I'm so grateful to have had her love and support as I completed this work.

During graduate school I've watched my sister Liana start *and* finish college and a graduate program of her own, and I've watched her grow into an amazing woman. She's forging a path for herself with deep strength and enthusiasm, and I'm really happy to be a part of it. I'm so proud to be her brother.

Finally, the love and support of my parents, Pam and Peter, has made me who I am, and is behind everything I've done. Thank you, Mom and Dad, and I love you.

Michael M. Vitousek

GRADUAL TYPING FOR PYTHON, UNGUARDED

Gradual typing integrates static and dynamic typing with the guarantee that statically typed regions of a program will never have their types violated even if they interact with dynamically typed regions. Traditionally, this guarantee has been achieved by using proxies to mediate when values flow between statically and dynamically typed code. However, languages such as Python cannot support these proxies, and so a different approach is needed to adapt gradual typing to Python. In my dissertation, I show that the guarded approach is a poor match for Python, but also that sound and efficient gradual typing for Python can be achieved with enforcement strategies that do not use proxies.

I show two approaches for achieving soundness without proxies. The first, the *monotonic* approach for mutable references, uses runtime type information to ensure that data corresponds to all types at which it has been referenced, removing the need for proxies. However, the need to track this information is a serious challenge.

Alternatively, lightweight *transient* checks may be inserted throughout the statically typed regions of the compiled program, which ensure that all values shallowly correspond to the static types they are expected to have. I implemented this strategy in a gradually typed version of Python, and found that the transient strategy is a good match with Python: transient checks can be straightforwardly implemented and are efficient. In order to improve the runtime performance of transient gradual typing, I developed a type-inference-based approach to erase transient checks when they can be statically shown to be redundant. I studied a number of benchmarks in Python with this implementation and found that the performance impact of transient gradual typing is minimal in most cases, especially when used in combination with PyPy, a JIT-based Python runtime.

Contents

CHAPTER 1

Introduction

1. Overview

Static and dynamic typing are well-suited to different programming tasks [59]. Static typing excels at documenting and enforcing constraints, enabling IDE support such as auto-completion, and helping compilers generate more efficient code. Dynamic typing, on the other hand, supports rapid prototyping and is suited to metaprogramming and reflection. Because of these tradeoffs, different parts of a program may be better served by one typing discipline or the other. Further, the same project may be best suited to different type systems at different points in time—a system may start out in an amorphous state with quickly-changing structures and design goals, but then evolve into a larger system with precise specifications that benefit from the automatic verification provided by static type systems.

For this reason, combining static and dynamic typing within a single language and type system has been a popular goal, especially in the last decade. Early approaches include those of Abadi et al. [3], Thatte [91], and Bracha and Griswold [22]. Siek and Taha [75] introduced *gradual typing* as an approach to merging static and dynamic typing using a notion of consistency between (partially dynamic) types, together with higher-order casts. Over the last decade, gradual typing has been of great interest to both the research community [67, 88, 86, 84, 70, 10, 5], and to industry developers, who have introduced several languages with elements of gradual typing, such as TypeScript [61], Flow [31, 25], and Hack [30].

Vitousek and Siek [98] draw a distinction between two approaches to combining static and dynamic typing: *optional typing* and *gradual typing*. These approaches share a common approach to static typechecking: both approaches include a static type system that detects incompatibilities between statically typed regions of code, just as a traditional typechecker would. Both gradually and optionally typed languages include a "dynamic type" (often written as $\star$, $?$, dyn, or any; in this dissertation the symbol $\star$

denotes the dynamic type) that represents that the static typechecker should be optimistic: an expression of any type may be passed into the dynamic type and vice versa without the typechecker raising a static type error. This gives programmers an "escape hatch" when they do not want to, or cannot, precisely specify the types of their programs, while allowing them to gradually transition towards full static typing if desired.

The distinction between optional and gradual typed languages is centered in their dynamic semantics. Programs in an optionally typed language are usually translated to an existing dynamically-typed language by erasing type annotations. For example, TypeScript and Flow translate to JavaScript, Typed Clojure [20] to Clojure, Hack [30] to PHP, and Typed Lua [58] to Lua. The resulting programs are then executed on existing runtimes for their target languages. Because of the existence of the dynamic type in the optionally typed language's static type system, it is possible for these programs to have values flow from dynamically typed to statically typed regions of code, and for those values to not correspond to the static types they are expected to have. In an optionally typed language, nothing at runtime enforces that values correspond to their expected static types, and so no errors result in such situations (although the semantics of the target language may cause such programs to reach an error state, for example if a number is called as if it were a function). Optionally typed languages are therefore *unsound with respect to their type annotations*—they may not be unsafe in the sense of lacking memory safety or eliciting undefined behavior (unless the target language is itself unsafe), but the type specifications given by the programmer are not necessarily respected at runtime.

By contrast, a *gradually typed* language is defined as being *sound* with respect to its type annotations. This is a property that gradual typing shares with traditional statically typed languages. Like optionally typed languages, gradually typed languages usually translate their programs into a target language, but this translation enforces sound interaction between static and dynamic code at runtime by inserting runtime type checks, which inspect values at runtime. Runtime type checks are used to prevent values from inhabiting variables of the wrong type where static types are expected. Soundness with respect to type annotations is a desirable property because it improves the debugging experience for programmers by allowing them to trust their type annotations, but the need to perform runtime type checking degrades performance at runtime.

The traditional approach used for runtime type checks in gradually typed languages is for the translation to the target language to insert casts at the site of implicit conversions between static and dynamic code [75, 48, 103, 93]. Casts on first-order types are constant-time operations that either succeed or fail immediately, but higher-order casts create wrappers or *proxies*. These proxies then mediate further interaction between the proxied value and the code it is embedded in, and will perform its own additional runtime checks to ensure that static types are never invalidated. Proxies can also be used to perform *blame-tracking* [103], which allows runtime type errors to be traced back to the boundary crossing site that eventually led to the error, helping debugging. I refer to this design as the *guarded* semantics for gradual typing.

Unfortunately, guarded has issues with both efficiency and practicality in many circumstances. Not all dynamic languages support robust proxies, and the pervasive use of proxies has been found to degrade performance in existing gradually typed languages. The designer of a gradually or optionally typed language may not be able to alter the target language to fix these problems: the designers of TypeScript, for example, cannot unilaterally alter JavaScript to better suit their designs without losing compatibility with existing JavaScript codebases. These factors interfere with the desirable goal of transitioning unsound optionally typed languages into gradually typed ones that are sound with respect to their annotations. However, new techniques that avoid dependence on proxies may smoothen the path for optionally typed languages to become gradually typed ones, and I will examine this space in this dissertation. I will do so in the context of a gradually typed language that translates to Python, a dynamically typed language that does not support robust proxies. Modifying Python itself is outside the scope of this work in order to maintain compatibility with existing Python programs and lower the barrier to entry for programmers already using Python.

In this dissertation, I will explore the following thesis:

> *The guarded approach is a poor match for a gradually typed variant of Python implemented as a source-to-source translation to Python, but I develop alternate enforcement strategies and achieve sound and efficient gradual typing for Python without relying on proxies.*

I will discuss two alternate enforcement strategies which I have developed, as well as the system that I use to evaluate them in practice. Chapter 3 of this dissertation will discuss a system for efficient gradual typing for mutable references. Chapter 4 will present the Reticulated Python programming language, a gradually-typed version of Python that I use as a testbed. It also informally introduces the approach to sound gradual typing that relies on pervasive lightweight checks: the *transient* semantics for gradual typing. Chapter 5 will dive deeper into the transient semantics, proving several key theorems that show its practicality. Chapter 6 will evaluate the performance of the transient semantics in Reticulated Python. The remainder of this chapter will discuss and summarize the key contributions of this dissertation.

1.1. Reticulated Python: Gradual Typing for Python. I developed the Reticulated Python programming language to explore the challenges of gradual typing. Reticulated Python is a testbed for gradual typing for the Python 3 programming language which typechecks type-annotated Python code using a gradual type system, and then translates this code into standard Python 3. Because Reticulated Python is a gradually typed language, it inserts code during translation that perform runtime checks.[1]

Initially, Reticulated Python used the guarded semantics for these checks, but I encountered significant problems with this approach. In particular, Python's support for proxies is limited: proxies ought to behave indistinguishably from the proxied object except for raising runtime type errors when type guarantees are violated, but Python's proxies are easily distinguishable from their underlying objects in ways that cause programs to fail unexpectedly. For example, the `type()` function (for inspecting the class of runtime values) returns the class of the proxy rather than the proxied value, and a proxied value is not pointer-identical to the proxied value, which is revealed by using Python's `is` operator [100]. Most significantly, proxies are unable to interact with the compiled C++ code that implements the Python runtime itself, so when proxied objects are passed into built-in language functions, unexpected errors arise [101]. I refer to languages like Python, with limited proxy support, as *spartan hosts*, and without modifying the Python language, proxies powerful enough to implement the guarded approach are impossible.

[1] Reticulated Python is available at `github.com/mvitousek/reticulated`.

In addition, even if proxies were possible in Reticulated Python, their use necessarily demands significant performance overhead. For example, when a proxied mutable reference is dereferenced even within statically typed code a cast will have to occur at runtime—the proxied value will first have to be dereferenced, and then the resulting value casted to ensure that it corresponds to its expected static type.

Furthermore, Takikawa et al. [90] showed via practical experiments that Typed Racket, a prominent gradually typed language that uses the guarded design [93, 85], exhibits severe overhead (in some cases, over 100×!) in certain combinations of static and dynamic typing when compared to either wholly typed or wholly untyped versions of the same program.

1.2. Alternative Strategies for Sound Gradual Typing. I initially experimented with a solution to these problems with the *monotonic* semantics for gradual typing, which allows mutable data to be type-safe at runtime without proxying. This approach works by causing mutable values to keep track of their own types at runtime, and as a value passes into contexts where it is expected to have a static type, it becomes monotonically more precise (i.e. less use of the dynamic type). This allows statically typed uses of the value to be efficient, and in theory allows for compiler optimizations to leverage type information. I implemented this approach in Reticulated [100] and it was proved sound in a calculus [83]. However, I found that this approach, while promising in some contexts, was not appropriate for Python: functions still require proxies, Python does not robustly support the mechanisms needed to enforce monotonicity, and requiring that types become monotonically more precise during execution could cause unexpected type errors that would not occur with the guarded semantics. Additionally, the most significant optimization that monotonic supports, removing the need for dispatching at statically-typed dereferences, is unavailable in Reticulated Python, because it is a source-to-source translator that uses the standard Python runtime. As such, while the monotonic semantics remains a subject of interest in gradual typing and may be appropriate for other languages (or a specialized implementation of the Python language itself), I have ceased developing it for Reticulated Python.

Instead, my recent work has been focused on the *transient* semantics. This approach uses no proxies, and instead inserts lightweight runtime checks pervasively throughout the statically typed portion of the

program. These checks ensure that values *shallowly* correspond with their expected types: for example, a function that takes an argument f of type int $\rightarrow$ int will have a check at its entry point to ensure that f is a function. Then if f is called, another check will ensure that the result has type int. This approach avoids many pitfalls of other techniques because no wrappers or reflection are used anywhere—the only language features required is the ability to inspect the shallow runtime type (or type tag) of a value, which Python fully supports, allowing Reticulated Python to use the transient approach without errors [100].

I proved that this approach is sound in a calculus, and further that it obeys the property of *open-world soundness* [101]. This requires that a gradually typed program can be translated to a dynamic target language and can then interact soundly with code native to that target language. This ensures that translated code can serve as a library to be used by target-language clients, and that it can be embedded in code (like the Python runtime) that does not respect its expected types, without any errors arising other than ones raised by transient type checks or from the code native to the target language.

I also showed that this approach avoids the efficiency problems discovered by Takikawa et al. [90] by analyzing Reticulated programs with various combinations of type annotations. To further improve performance, I combined transient with static type inference to discover which runtime checks are unnecessary and remove them [102]. This resulted in an average slowdown of just 6% compared to the performance of Python on an untyped program. When combined with a tracing JIT implementation of Python, which can further remove checks via a dynamic analysis, the overhead of gradual typing was reduced further.

In this dissertation I will discuss my work in gradual typing for Python and my approaches to preserving soundness while avoiding the use of proxies. While my work has been guided by Python and its limitations, and the traditional approaches may work for other languages, by developing the design space of gradual typing outside of proxy-based approaches I hope to help expand the space of its useful applications. In particular, I aim to smoothen the path for optionally typed languages to become gradually typed ones. In Section 2 of this chapter I discuss the Reticulated Python programming language, the platform I use to explore these issues. In Section 3 I describe the *monotonic* semantics for gradual typing, which ensures soundness of mutable data and displays no overhead from heap lookups in statically

typed code, but which I found to not be suitable for Reticulated Python. Section 4 explores the *transient* semantics, which inserts lightweight typechecks pervasively throughout statically typed code. Section 5 concludes this chapter.

2. Designing Gradually Typed Python

With the goal of adapting gradual typing to the Python programming language, I initially worked on the Jython programming language with Jeremy G. Siek, Shashank Bharadwaj, and Jim Baker. Jython is a compiler for Python that produces bytecode for the Java virtual machine [51], and in this context we first studied alternatives to the standard semantics for gradual typing such as the monotonic approach, described below [99]. However, Jython is a very heavyweight piece of software and using it as a testbed for experimentation quickly proved to be impractical.

As an alternative, I developed the Reticulated Python programming language[2] [100], which I will discuss in Chapter 4 (and further develop in Chapters 5 and 6). Zeina Migeed, Benjamin Greenman, and Andrew Kent also contributed to Reticulated Python's development. Rather than a new implementation of Python, Reticulated is a source-to-source translator that accepts syntactically valid Python 3 code with type annotations, typechecks this code, and generates Python 3 code, which it then executes using an off-the-shelf Python runtime. The dynamic semantics of Reticulated programs differ from Python 3 programs in that Reticulated's translation process may insert run-time checks to mediate between static and dynamically typed code, which will raise runtime errors if static types are violated at runtime. The run-time checks are implemented as calls into Python libraries which are distributed with Reticulated, and Reticulated's source-to-source translator is intended to be somewhat agnostic with regards to the details of these libraries. This approach enabled fast experimentation with the design of the runtime libraries without much need for altering the translator (although, as discussed below in Section 4, some approaches require the translation to insert more calls to the runtime library than others).

While I experimented with varying approaches to Reticulated Python's dynamic semantics, all share a common static type system [100]. This type system supports first-class functions, object and class types,

[2] Reticulated Python is named for *Python reticulatus*, the largest species of snake in the python genus. I refer to it as "Reticulated" for short.

and accounts for Python's approach to method binding. Reticulated gives nominal types to objects and classes, but also supports structural types when desired. Also supported are many of the basic data structures provided by Python, such as lists, sets, and dictionaries. Also included is the dynamic type, which can be explicitly written by the programmer using the keyword `Any`, or which is assumed in the absence of an annotation.

Since this type system is suited to Python programs, and since Python itself provides libraries that model much of its own functionality, Reticulated leverages these libraries in implementing a type system for Python programs. For example, the specification of a Python function's parameters can be very complex: parameters can have default values or be keyword-only (meaning they can only be called if the caller explicitly maps an argument to the parameter by name), and both positional and keyword-only rest arguments are supported. Similarly, function call sites can be complex. Therefore, rather than building a type specification for function arguments that accounts for this complexity, Reticulated represents function types using Python's reflective `Signature` object, which contains a full specification of a function's arguments. When typechecking a call site, the `Signature.bind` method, which implements Python's argument-parameter binding algorithm, is used to bind argument types to parameter types, after which they are compared to determine if the call site was well-typed.

As of version 3.5, Python does not provide syntax for giving type annotations to local variables. Instead, Reticulated uses local type inference to determine the static type of locals. This inference is conservative, and results in the dynamic type if a single type cannot be inferred.

3. Gradual Typing with Monotonic References

With Jeremy Siek, I developed the monotonic design for gradual typing by observing a fundamental issue for implementing sound gradual typing with mutable references (and by extension, other forms of mutable data) using guarded. In general, it is impossible to statically determine whether any dereference site is dereferencing a proxied value or a bare reference. This means that a compiler for a gradually typed language with references would need to emit code that dispatches on the form of the dereferenced value, degrading performance for *all* dereferences, including those that do not involve proxies. This essentially what Python and other dynamic languages always do—list lookups, for example, always check

that the value being looked into is definitely a list (or some other value that supports lookup) before allowing the operation to proceed.

The monotonic design attempts to recover performance in these cases. It does not use proxies to mediate between static and dynamic. Instead, the runtime maintains the invariant that the contents of a reference always have a type that is more or equally precise than every static type that the reference inhabits. Here, "precision" is how static a type is: int $\to$ int is more precise than int $\to \star$ and $\star \to$ int, which are both more precise than $\star \to \star$, which is itself more precise than $\star$. The runtime also maintains this most-precise-type as part of the reference itself. When a reference flows through a cast between static and dynamic code (which are inserted at implicit conversion sites during translation to a target language), if the target of the cast is more precise than the reference's internal type, the internal type is updated with this new type and the referenced value is downcasted to it.

Since references are never proxied, there is no need to check whether a dereference is of a proxy or a bare reference—the system can optimize with the assumption that the target of the dereference is always a bare reference. After the dereference happens, however, the result may be of a more precise type than is expected for the context: for example, if a reference with type ref $\star$ is dereferenced, but the reference value has flowed into code where it has type ref int, then the result of the dereference would have type int rather than $\star$, which is relevant if values with type $\star$ are boxed but typed values are not. In such cases, after the dereference occurs, the result needs to be casted from its precise type to the less-precise type that it is expected to have at the dereference site.

However, if the reference has a *fully-static* type, with no appearances of $\star$, then no cast needs to occur because the value must already have this type, since no type is strictly more precise than any fully-static types. The overall result is that fully-static code can be made efficient, at the cost of some additional overhead at cast sites.

With Jeremy Siek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia, I developed a gradually-typed lambda calculus for the monotonic approach. We proved it sound and showed that it obeyed the blame-subtyping theorem, which allows runtime type errors to be traced back to the cast (or casts) that potentially caused the error.

In Chapter 4 I will discuss an experimental implementation of the monotonic approach in Reticulated Python. Several problems arose that prevent monotonic from being an ideal solution for this context. First, it still uses proxies in some situations, such as for functions, and proxies cause significant problems for Python, even beyond performance overhead, as discussed below in Section 4.1. Additionally, the key benefits of the monotonic approach to gradual typing are unavailable in Reticulated Python under its current approach. In order to use type information to optimize dereferences, as monotonic is theoretically capable of doing, the actual runtime or compiler has to be able to efficiently perform dereferences without dispatching on the form of the reference value. Reticulated Python, however, is simply a source-to-source translator—applying monotonic to Python would require significant modification of the Python runtime itself, which is beyond the scope of this work. Other researchers are currently investigating the monotonic approach in a compiler for a gradually typed language, which may bear more fruit [56]-, but for Reticulated Python the monotonic approach does not appear to be ideal at the moment.

4. Transient Gradual Typing

The most successful approach used by Reticulated Python is the *transient* approach. This approach, like monotonic, attempts to solve some of the underlying problems of the guarded semantics, including performance issues and limitations of Python.

4.1. Python Supports Proxies Poorly. As discussed above, the use of proxies incurs serious performance costs, and it does so even beyond the overhead inherent in dynamically-typed languages which don't leverage static types for efficiency (the kinds of overhead that monotonic is designed to minimize). This is due to the need to instantiate proxy values at runtime and the indirection that they cause. Takikawa et al. [90] found that in some combinations of static and dynamically typed code, programs performed as much as $100\times$ worse than either the fully typed or the fully untyped version of the program. While the underlying language for Typed Racket is Racket rather than Python, Python would likely perform *worse* because Typed Racket has proxies as a core language feature [85] while Python proxies are implemented using general-purpose (and slow) reflection.

In addition, the guarded approach failed to cope well with significant real-world Python programs because Python does not have powerful enough support for proxies. To fully implement the guarded approach, a proxy for a value must be indistinguishable from that value, except if it detects a violation of the expected static types and raises a runtime error. In Python, however, this is impossible, as I discuss in Chapter 4: proxies are implemented as instances of a special proxy class, and using the builtin `type()` function on a proxy (which returns the class value that its argument is an instance of) will reveal that the proxy is an instance of the proxy class, not of the proxied value's class. Similarly, a proxied value can be distinguished from its underlying value with the `is` operator, which implements pointer equality. I found that this behavior caused problems in practice in case studies.

While this problem could be avoided by rewriting all uses of `type()`, `is`, and other operations that reveal proxies, in Chapter 5 I will discuss an even more severe problem with proxies in Python: proxies cannot interact with parts of the basic Python runtime without causing errors. This issue is shown in the following program:

```
1 def append42(y:List[int]):
2   list.append(y, 42)
3
4 def pass_to_append(z:*):
5   append42(z)
6
7 x = [1,2,3]
8 pass_to_append(x)
```

Here, the dynamically-typed `pass_to_append` function passes its argument to `append42`, which expects a list of integers. With guarded, a cast will be inserted at this call site to ensure that the dynamically typed value behaves as a list of integers. This cast will install a proxy around `z`, and the proxy is passed into `append42`. Then, inside `append42`, this proxy is passed into `list.append()`, which is the Python class method for lists that appends the second argument onto the first.[3] The expected result of running this program is that the list `x` will now contain `42`. However, in reality (when using the guarded

[3] Typically this code would be written as `y.append(42)`, and if `y` were an unproxied list the result would be identical.

semantics), either an error will occur because `list.append` was passed a proxy value rather than a list, or the program will run to completion but will fail to actually mutate `x` to contain `42`. Which problem occurs depends on choices made in implementing guarded proxies and involve other tradeoffs; regardless both results occur because the `list.append` function is implemented in compiled code that does not respect the mechanisms that I use to implement Python proxies (overriding the `__getattribute__` method). Instead this code will attempt to directly alter the internal structure of the list in memory, and since the value passed into `list.append` is not the original list but is instead a proxy, this mutation fails.

Because Python's proxies are not robust enough to allow for proxied code to interact with foreign functions and because they cannot successfully imitate their underlying values, Python is a *spartan host*. By contrast, the language that most successfully implements the guarded approach (though with performance caveats) is Typed Racket, and its target language Racket is not a spartan host: its chaperones and impersonators are specifically designed to implement proxies and are core language features [85]. However, making it a non-spartan host took dedicated engineering work, as chaperones and proxies were implemented in Racket specifically as an aid to supporting guarded gradual typing; systems where the target language is out of the control of the designer of the gradually typed language face similar challenges.

4.2. Gradual Typing through Transient Checks. Chapter 4 will discuss how I developed the *transient* approach to gradual typing in an effort to solve this problem in Reticulated Python. This approach does not use proxies at all—instead, the translation process inserts lightweight *checks* into programs. These checks inspect runtime values to ensure that they *shallowly* correspond to the static types they are expected to have. For example, if a function `foo` has a parameter `bar` that is expected to have type int $\rightarrow$ int, then at the entry to `foo`, a check will ensure that `bar` is a function. Then, when `bar` is called, another check will ensure that the result of the call is an integer, since `bar` is expected to return values of type int. Each of these checks is "transient" in its effect—it simply examines a runtime value and compares its runtime type information to a type tag [13]. If the comparison holds, then the value is returned without being wrapped or mutated in any way; if it fails the result is an error. This approach is

sound despite the limited nature of transient checks because the checks are inserted pervasively (though see Section 4.6 below). As an example, consider the following program, written in Reticulated Python:

```
1 def f(x:⋆):
2   x[0] = 'hello world'
3
4 def g(y:List[int]):
5   f(y)
6   y[0]
7
8 g([0])
```

In this program, `g` passes its argument `y`, expected to be a list of integers, to `f` and then looks up `y`'s zeroth element. In `f`, the dynamically typed argument is treated as a list and updated to have its zeroth elements be a string. This program contains a runtime type error, because the lookup at line 6 will return a string when it is expected to be an integer. In order to detect this and raise a runtime error, the transient approach inserts a check at the site of the list lookup:

```
1 def f(x):
2   x[0] = 'hello world'
3
4 def g(y):
5   y⇓List
6   f(y)
7   y[0]⇓int
8
9 g([0])
```

The check at line 5 (written `y⇓List`) checks that `y` is a list, which it is, so the program proceeds with an unaltered, unproxied list. Then at line 7, the check inspects the result of the list lookup to ensure that it is an integer. When it sees that the value is a string instead, an error is raised.

4.3. Properties of Transient Gradual Typing. In Chapter 5 I will develop a formal analysis of the transient semantics in a calculus and prove several key properties. Transient gradual typing is designed very differently than guarded. It offloads the responsibility of ensuring soundness to the user of values, rather than the values themselves, and since all checks are inserted statically, all dereferences and call sites in statically typed code will be checked—even if, at runtime, the check cannot fail. However, this pessimism means that transient is sound with respect to type annotations, even in the presence of unmoderated interaction with foreign functions and the "open world:" programs native to the target language that interact with translated code.

In order to study the soundness of transient gradual typing as an approach, I developed the $\lambda^{\star}_{\rightarrow}$ calculus, which is a gradually-typed lambda calculus with mutable references that uses the transient design. It is translated to another calculus, the $\lambda^{\Downarrow}_{\ell}$ calculus, which is dynamically typed and which models Python. I model the embedding of translated Reticulated Python code within standard Python by using *program contexts* $\mathcal{C}$, which are essentially $\lambda^{\star}_{\rightarrow}$ programs with holes in them into which other programs can be embedded. With this system, I proved the following *open-world soundness* property:

Theorem 1.1 (Open-World Soundness). *Suppose t is a $\lambda^{\star}_{\rightarrow}$ expression of type T that translates to the term e in $\lambda^{\Downarrow}_{\ell}$. Then, for any program context $\mathcal{C}$, either:*

- $\mathcal{C}[e]$ *reduces to a value in zero or more steps, or*
- $\mathcal{C}[e]$ *diverges, or*
- $\mathcal{C}[e]$ *reduces to a runtime check failure, or*
- $\mathcal{C}[e]$ *causes an uncaught type error while evaluating in $\mathcal{C}$.*

This definition has the flavor of type soundness (defined for gradually typed languages in Section 1.2.2 of Chapter 2) with one fundamental difference: instead of prohibiting (in isolation) translated $\lambda^{\star}_{\rightarrow}$ programs from misbehaving, it allows interaction with code native to $\lambda^{\Downarrow}_{\ell}$, and guarantees that any incorrect behavior is due to the program context $\mathcal{C}$, which may represent a client program interacting with the translated code or the translated code interacting with an untyped library.

Because $\lambda^{\star}_{\rightarrow}$ admits open-world soundness, a program written in $\lambda^{\star}_{\rightarrow}$ can be used by native $\lambda^{\Downarrow}_{\ell}$ clients. For example, an $\lambda^{\star}_{\rightarrow}$ library may put type annotations on its API boundaries, preventing ill-typed terms

from raising difficult-to-diagnose errors deep within the library—even if the library interacts with code which has no concept of static types.

Furthermore, the $\lambda^{\star}_{\rightarrow}$ code is protected from errors arising due to mutation: while foreign functions are not modeled directly in the $\lambda^{\star}_{\rightarrow}$ and $\lambda^{\Downarrow}_{\ell}$ calculi, the distinction between untranslated target-language programs and foreign, compiled C code is only relevant in guarded because of the presence of proxies. The transient design lacks proxies, and thus this distinction is irrelevant—foreign functions may be modeled as native $\lambda^{\Downarrow}_{\ell}$ code.

Finally, I also proved that the *gradual guarantee* holds for $\lambda^{\star}_{\rightarrow}$. The gradual guarantee is a promise to the programmer that their programs can be moved from static to dynamic and vice versa without being hindered by the semantics of the program changing. This theorem, defined by Siek et al. [82] and discussed in further depth in Section 1.2.4 of Chapter 2 ensures that changing the static type annotations in a program does not alter either the static or dynamic semantics of the program, except by raising a static type error or causing blame at runtime if the type annotations are strengthened [82]. This property allows programmers to be confident in gradually adding types to their program: they know that a program will never produce an entirely different result because of a change to the type annotations. They are also guaranteed that if a program raises a new error after a type annotation is added or strengthened, it is because the new annotation was "wrong": it did not correspond to other types in the program (if the error is static) or to the program's values at runtime (if it is a runtime blame error).

4.4. Blame Tracking with Transient. With the guarded approach, proxies served as the mechanism for tracking and propagating blame information [35, 104, 93]. The runtime system uses this information when a runtime type error is encountered to report the source of the error—not just the location where the error was discovered and the error was raised, but also which boundary-crossing site between static and dynamic was violated. This information helps the programmer debug the issue more efficiently.

Transient, lacking proxies, does not have the basic mechanism used for blame tracking, but in Chapter 5 I will show a solution to this problem by tracking blame information in a global *blame map*, updating the relevant blame information at every implicit conversion. The system tracks when values are passed between different static types by statically inserting casts into the program as in guarded; rather than

serving as a type enforcement mechanism, these casts only update the blame map—checks are still the main mechanism for detecting type errors. When a check fails, this blame map is used in conjunction with the type information at the failure site to construct the full error account to the programmer. Furthermore, this construction process provides a *blame history*, indicating the conflicting assumptions that different pieces of the program made to produce this error.

Through use of the blame map, casts and checks collaborate to reconstruct the chains of responsibility that guarded proxies provide. Even so, the transient blame behavior differs from guarded: the algorithm may blame multiple casts if each of them is reachable in the blame map and *may* be responsible for the check failure occurring, and if a sequence of incompatible casts are applied to a value but the value is never used (e.g., a function that is never applied), then no error occurs.

Despite the difference, I proved that this approach, when implemented in the $\lambda^{\star}_{\rightarrow}/\lambda^{\Downarrow}_{\ell}$ system, preserves the *blame-subtyping theorem* of Wadler and Findler [104]. This property, described in depth in Section 1.2.3 of Chapter 2, guarantees that casts (or other boundary crossings from one type to another) can only be *blamed* for a runtime error if they are unsafe. This result ensures that programs whose implicit type conversions are safe will never be blamed for cast failures.

4.5. Implementation and Efficiency in Reticulated Python. The runtime checks required for sound gradual typing inevitably impose some degree of runtime overhead no matter what approach is used. Since gradual typing is designed to allow programmers to gradually vary their programs between static and dynamic [82], it is also important that adding or removing individual annotations does not dramatically degrade the program's performance. Takikawa et al. [90] examine the performance of Typed Racket's guarded gradual typing with this criterion in mind by studying programs through the lens of a *typing lattice* made up of differently-typed *configurations* of the same program. The top of the lattice is a fully typed configuration of the program and the bottom is unannotated, and incrementally adding types moves up the lattice.

Takikawa et al. show that in Typed Racket, certain configurations result in catastrophic slowdown compared to either the top or bottom configurations. This indicates that the guarded semantics incurs a

substantial cost when interaction between static and dynamic code is frequent. Many of their benchmarks show mean overheads of over 30× and worst cases of over 100×, which "projects an extremely negative image of *sound* gradual typing" [90].

In Chapter 6 I will discuss how I determined whether Reticulated Python and the transient semantics face the same problem. I analyzed the performance of several benchmarks across their typing lattices, and found that the cost of transient gradual typing increases as the number of type annotations grows. As a program evolves from dynamic to static, its performance linearly degrades, with the worst performance in the most static configurations. This is because each static type annotation induces checks to ensure that values correspond to that type. This is, of course, counter to one hypothetical benefit of static typing—ideally, static types should aid performance, or at least not degrade it. On the other hand, the linear degradation of performance to a worst-case of less than 6× overhead means that the catastrophic configurations encountered in Typed Racket never occur and the cost of adding an individual type annotation to a program is predictable.

4.6. Optimizing Transient with Type Inference. The transient approach inserts checks throughout the program, but not all checks are necessary for the program to be sound because some checks may be redundant and always succeed. In Chapter 6 I will discuss how, in order to remove unnecessary checks, I modified Reticulated Python to perform type inference on the program after checks have been inserted. This inference algorithm is based on those of Aiken and Fähndrich [6] and Rastogi et al. [67] and uses subtyping constraints as well as special *check constraints*, generated by transient checks. I proved that this algorithm can soundly remove unnecessary checks in a transient calculus similar to the $\lambda^{\star}_{\rightarrow}/\lambda^{\Downarrow}_{\ell}$ system:

Theorem 1.2 (Optimization Correctness). *Suppose that an expression t evaluates to a value v. If the type inference system finds that t has type T and T corresponds to the type tag S, then a transient check that v has tag S will always succeed.*

Since such checks are proven to always succeed, they can be removed without altering the semantics of the program.

After modifying Reticulated Python with this algorithm, I measured its performance, again sampling at all points of the typing lattices. With redundant checks removed, the linear increase in execution times disappears, resulting in the fully-typed configurations displaying negligible overhead and a 6% average overhead over all sampled configurations. I will discuss this work in Chapter 6 as well.

5. Conclusions

Gradual typing is a promising approach in programming language design, and Python is a language that appears well-suited to it. Python is commonly used in a multitude of different applications, from scientific computing to web backends, and in many cases static typechecking would be highly valuable—I found several potential bugs in Python programs by adding typechecking [100]. Other Python projects, such as MyPy [57], are also interested in adding types to Python (though they are optional, not gradual, type systems) and the Python syntax itself has been extended with function parameter annotations (which are Python expressions that are evaluated at runtime).

Reticulated Python is designed as a testbed for studying gradual typing in Python. With it, I discovered that there are several major obstacles to implementing sound gradual typing in Python. Python is a spartan host for gradual typing—it does not support the kind of proxies that the traditional approach to gradual typing requires. Python proxies can be easily distinguished from their underlying values, causing surprising errors, and they cannot soundly interact with the compiled C code that makes up much of the Python runtime.

I explored these challenges in the context of Python, and developed two designs that depart from the traditional proxy-based semantics. One of these, the monotonic approach, does not appear to be an appropriate solution for Python, but it may turn out to be a powerful technique in other contexts. The transient approach, on the other hand, is well-suited to Reticulated Python—Python supports inspection of runtime type information, which is all that the transient design needs. Furthermore, transient supports open-world soundness, allowing Reticulated Python programs to work soundly with compiled Python runtime functions, standard Python libraries, and even standard Python clients if the translated Reticulated program is distributed as a library.

The transient approach introduces its own set of challenges: blame tracking requires a more complex approach than with guarded, and there is substantial overhead involved in pervasively performing run-time checks. However, I studied and experimentally implemented solutions to both problems: blame tracking is possible by using a global map that tracks the flow of values between static and dynamic, and the performance overhead can be vastly reduced by using type inference to remove unnecessary checks.

As a result, Reticulated Python with the transient approach appears to be a promising way forwards for gradual typing in the Python programming language. Furthermore, the problems that we encountered in this work are not unique to Python, and our solutions are likely to be translatable to other contexts. As such, the transient design may lower the barrier to entry for converting optionally typed languages to gradually typing, allowing programmers to reap the benefits of being able to truly trust their type annotations.

CHAPTER 2

Background

In this chapter I will review the necessary background for the research that will be presented in later chapters, focusing especially on the state of the art in runtime enforcement for gradually typed languages, as well as the background of gradual typing itself.

1. Understanding Gradual Typing

Gradual typing was introduced by Siek and Taha [75] in order to *seamlessly* blend static and dynamic typing within the same language, without requiring the programmer specially handle either paradigm. Gradually typed languages use the dynamic type $\star$ and a *consistency relation* on types to govern how statically typed and untyped code interacts: the consistency relation plays the role that type equality usually does in the type system. Types are consistent if they are equal up to the presence of $\star$. For example, a *statically* typed lambda calculus might include a rule for function application as follows:

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_3 \qquad T_1 = T_3}{\Gamma \vdash e_1\, e_2 : T_2}$$

This rule requires that e_1 is a function type, and that its source type T_1 is equal to the type of the argument e_2. In a gradually typed language, this same rule must hold *if* the types of e_1 and e_2 are static. The presence of the dynamic type, however, must serve as an "opt-out" from the static type system: it must not reject any programs due to the presence of the dynamic type, while still rejecting ill-typed code where the dynamic type is not used. The traditional rules for achieving this are two-fold: first, the above rule is modified to use *consistency* in place of equality:

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_3 \qquad T_1 \sim T_3}{\Gamma \vdash e_1\ e_2 : T_2}$$

Here, the requirement that $T_1 \sim T_3$ (pronounced "T_1 is consistent with T_3") means that T_1 and T_3 are identical *up to* the presence of the dynamic type. Consistency is defined, for a language with the dynamic type $\star$, base types B, and function types $T_1 \to T_2$, as follows:

$\boxed{T \sim T}$

$$\star \sim T \qquad T \sim \star \qquad B \sim B \qquad \frac{T_1 \sim T_3 \qquad T_2 \sim T_4}{T_1 \to T_2 \sim T_3 \to T_4}$$

The definition of the consistency relation, and the way that it is used in the above typing judgment, means that arguments of dynamic type can be passed into any function and that functions whose source type is $\star$ can accept any argument, but it is still illegal to pass a string into a function that accepts integers.

This rule is on its own insufficient for a gradually typed language, however: it still requires that e_1 be statically typable as a function. In order to allow dynamically typed values to be called, the approach of Siek and Taha [75] is to add a second function call judgment to the system:

$$\frac{\Gamma \vdash e_1 : \star \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1\ e_2 : \star}$$

That is, if a callee is dynamically typed, then calling it is well-typed at type $\star$ as long as the argument is well-typed. Later authors such as Rastogi et al. [67] and Cimini and Siek [26] have collapsed these two rules down to one:

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_4 \qquad T_1 \rhd T_2 \to T_3 \qquad T_2 \sim T_4}{\Gamma \vdash e_1\ e_2 : T_3}$$

where the $\rhd$ operator, called "matching," is defined as follows:

$\boxed{T \rhd T}$

$$\frac{}{\star \rhd \star \to \star} \qquad \frac{}{T_1 \to T_2 \rhd T_1 \to T_2} \qquad \ldots$$

1.1. A Gradually Typed Lambda Calculus. I use this approach in the type system of a *gradually typed lambda calculus* (GTLC), the syntax and static type system of which is given in Figure 1. This calculus includes references (with creation `ref` e, dereferencing $!e$, and mutation $e_1 := e_2$), which are treated similarly to functions with respect to consistency, following Herman et al. [48]: a departure from Siek and Taha [75], whose references are invariant with respect to consistency. Most of the interesting rules in this calculus are discussed above; one unusual feature is that functions in this calculus include return type annotations in addition to parameter types: $\lambda(x{:}T_1){\to}T_2.\ e$ is a function with type $T_1 \to T_2$, and its typing judgment requires that its body e has a type consistent with T_2. I make this choice to remain consistent with many of the other approaches described in this dissertation, in particular the key calculi of Chapters 5 and 6, but it is not an essential feature of gradually typed languages.

1.2. Properties of Gradually Typed Languages. I will use this calculus to explore the properties that I use to guide development of gradually typed languages. Several properties were initially proposed by Siek and Taha [75], but Siek et al. [82] developed a more robust program of ideal properties for gradually typed languages, all of which hold for the GTLC (in some cases, depending on its dynamic semantics). Note that these properties were developed by myself and others in the course of the research described in this dissertation: this work helped inform the development of these properties as well as vice versa. The criteria in this section are claims about both the static and dynamic behavior of the GTLC, and while claims about the static behavior can be verified against the type system for the GTLC shown in Figure 1, properties about the GTLC's dynamic semantics will guide its development below in Section 1.3, as well as guide the development of the various languages and calculi discussed in the remainder of this dissertation.

1.2.1. Gradual typing is a superset of static and dynamic. Since gradual typing is intended to allow static and dynamic typing to intermingle seamlessly, a basic requirement is that it can admit static typing and dynamic typing directly. Let $e \Downarrow v$ denote some evaluation function for a GTLC expression

$$
\begin{array}{lrcl}
\text{variables} & x, y & & \\
\text{numbers} & n & \in & \mathbb{Z} \\
\text{expressions} & e & ::= & n \mid x \mid \lambda(x{:}T){\to}T.\, e \mid e\, e \mid \texttt{ref}\ e \mid\ !e \mid e{:=}e \mid e\ +\ e \\
\text{types} & T & ::= & \star \mid \mathsf{int} \mid T \to T \mid \mathsf{ref}\, T \\
\text{type environments} & \Gamma & ::= & \cdot \mid \Gamma, x{:}T
\end{array}
$$

$$\boxed{\Gamma \vdash e : T}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{}{\Gamma \vdash n : \mathsf{int}} \qquad \frac{\Gamma, x{:}T_1 \vdash e : T_2' \qquad T_2 \sim T_2'}{\Gamma \vdash \lambda(x{:}T_1){\to}T_2.\, e : T_1 \to T_2}$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_4 \qquad T_1 \rhd T_2 \to T_3 \qquad T_2 \sim T_4}{\Gamma \vdash e_1\, e_2 : T_3} \qquad \frac{\Gamma \vdash e : T}{\Gamma \vdash \texttt{ref}\ e : \mathsf{ref}\, T}$$

$$\frac{\Gamma \vdash e : T_1 \qquad T_1 \rhd \mathsf{ref}\, T_2}{\Gamma \vdash\ !e : T_2} \qquad \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_3 \qquad T_1 \rhd \mathsf{ref}\, T_2 \qquad T_3 \sim T_2}{\Gamma \vdash e_1{:=}e_2 : T_1}$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad T_1 \rhd \mathsf{int} \qquad T_2 \rhd \mathsf{int}}{\Gamma \vdash e_1\ +\ e_2 : \mathsf{int}}$$

$$\boxed{T \sim T}$$

$$\star \sim T \qquad T \sim \star \qquad \mathsf{int} \sim \mathsf{int} \qquad \frac{T_1 \sim T_3 \qquad T_2 \sim T_4}{T_1 \to T_2 \sim T_3 \to T_4} \qquad \frac{T_1 \sim T_2}{\mathsf{ref}\, T_1 \sim \mathsf{ref}\, T_2}$$

$$\boxed{T \rhd T}$$

$$\frac{}{\star \rhd \star \to \star} \qquad \frac{}{T_1 \to T_2 \rhd T_1 \to T_2} \qquad \frac{}{\star \rhd \mathsf{ref}\, \star} \qquad \frac{}{\mathsf{ref}\, T \rhd \mathsf{ref}\, T} \qquad \frac{}{\star \rhd \mathsf{int}} \qquad \frac{}{\mathsf{int} \rhd \mathsf{int}}$$

Figure 1. Syntax and static type system for the GTLC.

e that produces a value v, and let $e \Downarrow_S v$ denote the same evaluation except without any rules dealing specifically with dynamically typed code or with the interaction of static and dynamic. Let also $\vdash_S$ be

the typing judgment for the statically typed lambda calculus (obtained by replacing $\triangleright$ and $\sim$ with $=$ in Figure 1).

Criterion 2.1 (Gradual typing subsumes static typing). *Let e be a GTLC term whose type annotations do not contain $\star$. Then:*

- $\emptyset \vdash e : T$ *if and only if* $\emptyset \vdash_S e : T$.
- $e \Downarrow v$ *if and only if* $e \Downarrow_S v$.

Likewise, a gradually typed language must admit dynamic typing. This is somewhat more challenging to represent, given that using the rules in Figure 1, some expressions always have non-dynamic types: an integer will always have type int and a function will always have an arrow type. To show that dynamic typing can be represented in gradual typing, I therefore use an embedding function $\lfloor e \rfloor$, which η-expands all expressions in e by wrapping them with a dynamically-typed identity function. For example, n becomes $(\lambda(x{:}\star){\rightarrow}\star\ .\ x)\ n$, a term whose type is $\star$. Let also $e \Downarrow_D v$ be the evaluation function obtained by removing any evaluation rules that deal with specifically statically typed code, or the interaction between static and dynamic.

Criterion 2.2 (Gradual typing subsumes dynamic typing). *Let e be a GTLC term where all annotations are $\star$. Then:*

- $\emptyset \vdash \lfloor e \rfloor : \star$.
- $e \Downarrow v$ *if and only if* $\lfloor e \rfloor \Downarrow_D v$.

Similar criteria are given by Siek and Taha [75], and they are explored in greater depth by Siek et al. [82].

1.2.2. Gradual typing is sound. Gradually typed languages also display a soundness property analogous to that of statically typed languages. Let $e \Uparrow$ denote that e diverges when evaluated.

Criterion 2.3 (Type soundness). *If $\emptyset \vdash e : T$ then either:*

- $e \Downarrow v$ *and* $\emptyset \vdash v : T$*, or*
- $e \Uparrow$*, or*
- e *evaluates to a* runtime enforcement error.

$$\boxed{T <:_b T}$$

$$\text{int} <:_b \text{int} \qquad \star <:_b \star \qquad \text{int} <:_b \star \qquad \frac{T_3 <:_b T_1 \qquad T_2 <:_b T_4}{T_1 \to T_2 <:_b T_3 \to T_4} \qquad \frac{T_1 \to T_2 <:_b \star \to \star}{T_1 \to T_2 <:_b \star}$$

$$\frac{T_1 <:_b T_2 \qquad T_2 <:_b T_1}{\text{ref}\, T_1 <:_b \text{ref}\, T_2} \qquad \frac{\text{ref}\, T <:_b \text{ref}\, \star}{\text{ref}\, T <:_b \star}$$

Figure 2. Blame safety and blame subtyping for the GTLC.

This criterion differs from the typical form of a type soundness theorem [107] in the presence of the third condition. If e evaluates to a runtime enforcement error, then somehow the interaction between static and dynamic code has reached a state where, were further evaluation to occur, an uncaught runtime type error would arise (for example, if a value whose static type is $\star$ but whose value at runtime is a string were to be passed into a function that expects arguments of type int). In such cases, the evaluation function must raise an error and halt the program's evaluation in order to be sound.

This property is subsumed by a further property desirable for some gradually typed languages, the *open-world soundness* property introduced in Section 4.3 of Chapter 1, which I discuss in more detail in Chapter 5.

1.2.3. Runtime enforcement errors only arise from unsafe code. In addition to evaluating to runtime enforcement errors when need be, it is also important that gradually typed languages do not evaluate to runtime enforcement errors when no runtime errors are This is captured by the following theorem, using the *safe subtyping* rules in Figure 2.

Criterion 2.4 (Blame-subtyping property). *Suppose that $\emptyset \vdash e : T$. Suppose also that e contains some expression where a subterm e' is passed from a context where it is expected to have type T_1 to one where it is expected to have T_2, and that $T_1 <:_b T_2$. Then if e evaluates to a runtime enforcement error, the error was* not *caused by this boundary-crossing site.*

Traditionally, this property is expressed using a notion of *blame labels*: every boundary crossing site is given a unique label that, at runtime, is attached to a mechanism of runtime enforcement, which

$$\boxed{T \sqsubseteq T}$$

$$T \sqsubseteq \star \qquad \text{int} \sqsubseteq \text{int} \qquad \frac{T_1 \sqsubseteq T_3 \qquad T_2 \sqsubseteq T_4}{T_1 \to T_2 \sqsubseteq T_3 \to T_4} \qquad \frac{T_1 \sqsubseteq T_2}{\text{ref}\, T_1 \sqsubseteq \text{ref}\, T_2}$$

$$\boxed{e \sqsubseteq e}$$

$$x \sqsubseteq x \qquad n \sqsubseteq n \qquad \frac{T_1 \sqsubseteq T_3 \qquad T_2 \sqsubseteq T_4 \qquad e_1 \sqsubseteq e_2}{\lambda(x{:}T_1){\to}T_2.\, e_1 \sqsubseteq \lambda(x{:}T_3){\to}T_4.\, e_2} \qquad \frac{e_1 \sqsubseteq e_3 \qquad e_2 \sqsubseteq e_4}{e_1\, e_2 \sqsubseteq e_3\, e_4}$$

$$\frac{e_1 \sqsubseteq e_2}{\texttt{ref}\, e_1 \sqsubseteq \texttt{ref}\, e_2} \qquad \frac{e_1 \sqsubseteq e_2}{!e_1 \sqsubseteq !e_2} \qquad \frac{e_1 \sqsubseteq e_3 \qquad e_2 \sqsubseteq e_4}{e_1{:=}e_2 \sqsubseteq e_3{:=}e_4} \qquad \frac{e_1 \sqsubseteq e_3 \qquad e_2 \sqsubseteq e_4}{e_1 \;+\; e_2 \sqsubseteq e_3 \;+\; e_4}$$

Figure 3. Precision on types and expressions.

blames the label if it detects a violation of the expected static types. The blame-subtyping property shows that if the use of the consistency relation in the static semantics that allowed the boundary-crossing site to be well-typed was also a valid use of the safe subtyping relation, then this use of the consistency relation is "innocent" of any runtime errors that may occur later on. In many systems, a corollary of this property is that if *all* boundary crossings are safe, then the program *cannot* evaluate to a runtime enforcement error. Wadler and Findler [104] introduced this property and informally describe it as "well-typed programs can't be blamed."

1.2.4. Programs must be able to evolve between static and dynamic. The ultimate design goal of gradual typing is to enable programmers to mix static and dynamic typing where desirable, and one important reason to do so is to allow programs to start off as dynamic prototypes and then become statically typed as requirements become more concrete. For a gradually typed language to allow this key behavior, it must not be the case that altering the type annotations within the program alters the program's result, unless by a static type error or a runtime enforcement error if the "wrong" type was added.

We can reason about shifting between different typings for the same program by introducing a partial order on types and terms, the *precision* relation shown in Figure 3. This relation was also referred to as

"naive subtyping" by Wadler and Findler [104] and others. The precision relation on types captures how much dynamicity a type contains:

$$\text{int} \to \text{int} \sqsubseteq \star \to \text{int} \sqsubseteq \star \to \star \sqsubseteq \star$$

The precision relation applied to terms simply uses precision on types in the places where types appear in the program's syntax: in the case of the GTLC, exclusively in function type annotations.

With this partial order, it is possible to define a criterion for gradual typing that states that programs must behave the same when type information is added or removed. Siek et al. [82] refers to this property as the *gradual guarantee*; it has also been named *graduality* (by analogy with parametricity) by New and Ahmed [64].

Criterion 2.5. *Suppose* $e \sqsubseteq e'$ *and* $\emptyset \vdash e : T$. *Then:*

- $\emptyset \vdash e' : T'$ *and* $T \sqsubseteq T'$.
- *If* $e \Downarrow v$, *then* $e' \Downarrow v'$ *and* $v \sqsubseteq V'$.
- *If* $e \Uparrow$ *then* $e' \Uparrow$.
- *If* $e' \Downarrow v'$, *then either* $e \Downarrow v$ *and* $v \sqsubseteq v'$, *or* e *evaluates to a runtime enforcement error.*
- *If* $e' \Uparrow$, *then either* $e \Uparrow$ *or* e *evaluates to a runtime enforcement error.*

The gradual guarantee ensures that removing type annotations from a well-typed program results in another well-typed program, and if it ran correctly originally the resulting program will also run correctly. Of course, the more common case is likely to be the reverse: taking a program that runs correctly and *adding* static types to it. In this case, it is not always true that the program will behave the same, because the type added could be the "wrong" one. For example, the program

$$(\lambda(x{:}\star){\to}\star\,.\,x)\;42$$

is well-typed and evaluates to 42, but

$$(\lambda(x{:}\text{str}){\to}\text{str}.\;x)\;42$$

is ill-typed, and

$$(\lambda(x{:}\star){\to}\text{str}.\;x)\;42$$

evaluates to a runtime enforcement error (when using the evaluation rules shown below in Section 1.3.1, because the return type of the function is annotated to be str). However, the programmer still has the guarantee that the *only* ways that the program can behave differently by going up the precision lattice is if it becomes ill-typed or reaches an enforcement error, and if the "correct" type annotation is added (if it exists) the behavior will not change at all (other than possibly evaluating to a value that is more precise than the original result of evaluation).

1.3. Dynamic Semantics and Enforcement Strategies for Gradual Typing. The above criteria make claims about both the static and dynamic behavior of gradually typed languages. The static components are all satisfied by the type system for gradually typed lambda calculus defined in Figure 1, so I now turn to defining a dynamic semantics for the GTLC that satisfies the dynamic requirements.

The dynamic semantics for a gradually typed lambda calculus that satisfies the above criteria must perform some amount of runtime inspection of types or values. A dynamic semantics that, for example, ignored type annotations in the source program and evaluated terms as though they originated in a dynamically typed lambda calculus would fail Criterion 2.3 (type soundness): the statically well-typed program $(\lambda(x{:}\star){\rightarrow}\text{int}.\ x)\ (\texttt{ref}\ 42)$ will evaluate to a reference, which does not have the type int as the term is expected to have.

For this reason, when defining the dynamic semantics of a gradually typed language, care needs to be taken to ensure that the above criteria hold. In this dissertation, I describe dynamic semantics as using *enforcement strategies* to ensure that the above criteria hold; the enforcement strategy is the means or method that the dynamic semantics uses to ensure that the properties are not violated.

1.3.1. The Guarded Enforcement Strategy. When Siek and Taha [75] introduced gradual typing, they used an approach to ensuring soundness that I dubbed the "guarded" enforcement strategy, so-called because higher-order values are wrapped with guards that make sure that, when the values are used or passed around, their behavior conforms to the types they are expected to inhabit. The guarded strategy is by far the most commonly used approach to gradual typing: Herman et al. [48] extended it to references, Wadler and Findler [104] applied blame-tracking to it, and numerous researchers have extended it [76, 5, 52, 88, 106, 43], optimized it [79, 49, 9], and studied variations on it [80, 12]. I discuss this

related work below in Section 2. Notably, this approach is the one taken by Typed Racket, a prominent gradually typed language used in professional practice [92] (see Section 3).

A dynamic semantics that uses the guarded enforcement strategy operates in two phases: first, a compilation phase translates the gradually typed program (with implicit conversions between types at the boundaries between static and dynamic) into a statically typed intermediate language where the conversions are made explicit. These explicit conversions or *casts* are inserted at the boundary of static and dynamic code. Once this compilation has occurred, the resulting program can be evaluated. When cast expressions are evaluated, they can result in *wrappers* being inserted or removed around values at runtime, and when a wrapped value passes through a cast with which it is incompatible, the result is a runtime enforcement error (in this context usually just called a *cast error*).

Figure 4 shows the syntax and static type system for the intermediate language used by the guarded strategy, called the *cast calculus*. Expressions in this calculus are mostly identical to the expressions of the GTLC, except for addresses a (the result of evaluating a reference) and casts $t::T_1 \Rightarrow T_2$, which is the expression representing an expression t being casted from type T_1 (the *source* of the cast) to T_2 (its *target*).[1] This calculus is statically typed: the $\triangleright$ matching operator is unused and the $\sim$ consistency operator is only used in the type judgment for casts, to ensure that "nonsense" casts between incompatible static types such as $42::\text{int} \Rightarrow \text{ref int}$ are rejected. Casts require that the expression being cast has the type of the cast's source, and the type of the cast overall is that of its target. Value forms for the cast calculus are also specified in Figure 4; they include numbers, reference addresses, and functions as usual, but they also include a limited subset of cast expressions: namely those where the casted expression is also a value and the cast is either from a *ground type* to the dynamic type or is between two higher-order types with the same constructor. Values that have been casted to dynamic are *boxed* or *injected* values; in real-world systems this can correspond to tagged values or values with a level of indirection. The source type for the cast on a boxed value is limited to *ground types* G, following Siek et al. [82], which are restricted to types with a non-dynamic constructor but with only $\star$ appearing as arguments to the type constructor. I refer to values casted from one higher-order type to another as

[1] When consulting related work, readers should be aware that the syntax for casts varies significantly from paper to paper.

$$
\begin{array}{lrcl}
\text{addresses} & a & \in & \mathbb{Z} \\
\text{ground types} & G & ::= & \text{int} \mid \star \to \star \mid \text{ref} \star \\
\text{expressions} & t & ::= & n \mid a \mid x \mid \lambda(x{:}T){\to}T.\, t \mid t\, t \mid a \mid \texttt{ref}\ t \mid !t \mid t{:=}t \mid t\ +\ t \mid t{::}T{\Rightarrow}T \\
\text{values} & v & ::= & n \mid a \mid \lambda(x{:}T){\to}T.\, t \mid v{::}G{\Rightarrow}\star \mid v{::}T \to T{\Rightarrow}T \to T \mid v{::}\text{ref}\, T{\Rightarrow}\text{ref}\, T \\
\text{heap type} & \Sigma & ::= & \cdot \mid \Sigma[a \mapsto T]
\end{array}
$$

$$\boxed{\Gamma; \Sigma \vdash t : T}$$

$$\frac{\Gamma(x) = T}{\Gamma; \Sigma \vdash x : T} \qquad \frac{}{\Gamma; \Sigma \vdash n : \text{int}} \qquad \frac{\Sigma(x) = T}{\Gamma; \Sigma \vdash a : \text{ref}\, T}$$

$$\frac{\Gamma, x{:}T_1; \Sigma \vdash t : T_2}{\Gamma; \Sigma \vdash \lambda(x{:}T_1){\to}T_2.\, t : T_1 \to T_2} \qquad \frac{\Gamma; \Sigma \vdash t_1 : T_1 \to T_2 \qquad \Gamma; \Sigma \vdash t_2 : T_1}{\Gamma; \Sigma \vdash t_1\, t_2 : T_2}$$

$$\frac{\Gamma; \Sigma \vdash t : T}{\Gamma; \Sigma \vdash \texttt{ref}\ t : \text{ref}\, T} \qquad \frac{\Gamma; \Sigma \vdash t : \text{ref}\, T}{\Gamma; \Sigma \vdash !t : T} \qquad \frac{\Gamma; \Sigma \vdash t_1 : \text{ref}\, T \qquad \Gamma; \Sigma \vdash t_2 : T}{\Gamma; \Sigma \vdash t_1{:=}t_2 : T_1}$$

$$\frac{\Gamma; \Sigma \vdash t_1 : \text{int} \qquad \Gamma; \Sigma \vdash t_2 : \text{int}}{\Gamma; \Sigma \vdash t_1\ +\ t_2 : \text{int}} \qquad \frac{\Gamma; \Sigma \vdash t : T_1 \qquad T_1 \sim T_2}{\Gamma; \Sigma \vdash t{::}T_1{\Rightarrow}T_2 : T_2}$$

Figure 4. Syntax and type system for the cast calculus.

"proxies:" the cast is a proxy for the underlying value, representing it at a different type than its source but behaving identically other than performing type enforcement.

Figure 5 defines a translation (or *cast insertion*) from the GTLC to the cast calculus. This translation inserts a cast for each use of the consistency or matching relations, resulting in a program in the cast calculus in which the conversions between types that these operations represent are made explicit in the form of casts. Some of these casts are identity casts and therefore redundant; some presentations are defined in such a way that redundant casts are not inserted [75], but their presence does not alter the program's outputs.

$$\boxed{\Gamma \vdash e \rightsquigarrow t : T}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x \rightsquigarrow x : T} \qquad \frac{}{\Gamma \vdash n \rightsquigarrow n : \mathsf{int}}$$

$$\frac{\Gamma, x{:}T_1 \vdash e \rightsquigarrow t : T_2' \qquad T_2 \sim T_2'}{\Gamma \vdash \lambda(x{:}T_1){\to}T_2.\, e \rightsquigarrow \lambda(x{:}T_1){\to}T_2.\, (t{::}T_2'{\Rightarrow}T_2) : T_1 \to T_2}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow t_1 : T_1 \qquad \Gamma \vdash e_2 \rightsquigarrow t_2 : T_4 \qquad T_1 \rhd T_2 \to T_3 \qquad T_2 \sim T_4}{\Gamma \vdash e_1\, e_2 \rightsquigarrow (t_1{::}T_1{\Rightarrow}T_2 \to T_3)\, (t_2{::}T_4{\Rightarrow}T_2) : T_3}$$

$$\frac{\Gamma \vdash e \rightsquigarrow t : T}{\Gamma \vdash \mathtt{ref}\ e \rightsquigarrow \mathtt{ref}\ t : \mathsf{ref}\, T} \qquad \frac{\Gamma \vdash e \rightsquigarrow t : T_1 \qquad T_1 \rhd \mathsf{ref}\, T_2}{\Gamma \vdash !e \rightsquigarrow !(t{::}T_1{\Rightarrow}\mathsf{ref}\, T_2) : T_2}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow t_1 : T_1 \qquad \Gamma \vdash e_2 \rightsquigarrow t_2 : T_3 \qquad T_1 \rhd \mathsf{ref}\, T_2 \qquad T_3 \sim T_2}{\Gamma \vdash e_1{:=}e_2 \rightsquigarrow (t_1{::}T_1{\Rightarrow}\mathsf{ref}\, T_2){:=}(t_2{::}T_3{\Rightarrow}T_2) : T_1}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow t_1 : T_1 \qquad \Gamma \vdash e_2 \rightsquigarrow t_2 : T_2 \qquad T_1 \rhd \mathsf{int} \qquad T_2 \rhd \mathsf{int}}{\Gamma \vdash e_1 \ + \ e_2 \rightsquigarrow (t_1{::}T_1{\Rightarrow}\mathsf{int}) \ + \ (t_2{::}T_2{\Rightarrow}\mathsf{int}) : \mathsf{int}}$$

Figure 5. Compilation from the GTLC to the cast calculus.

$$\begin{array}{rcl} \varsigma & ::= & \langle t, \mu \rangle \mid \mathtt{fail} \\ \mu & ::= & \cdot \mid \mu[a := v] \\ E & ::= & \square \mid E + t \mid v + E \mid (E\, t) \mid (v\, E) \mid \mathtt{ref}\ E \mid !E \mid E{:=}t \mid v{:=}E \mid E{::}T{\Rightarrow}T \end{array}$$

$\boxed{\lfloor T \rfloor = G}$

$$\begin{array}{rcl} \lfloor T_1 \to T_2 \rfloor & = & \star \to \star \\ \lfloor \mathsf{ref}\, T \rfloor & = & \mathsf{ref}\, \star \\ \lfloor \mathsf{int} \rfloor & = & \mathsf{int} \end{array}$$

Figure 6. Runtime syntax and auxiliary functions for the cast calculus.

$\boxed{\langle e, \mu\rangle \longrightarrow \varsigma}$

EIdInt	$\langle v{::}\mathsf{int}{\Rightarrow}\mathsf{int}, \mu\rangle$	$\longrightarrow$	$\langle v, \mu\rangle$	
EIdDyn	$\langle v{::}\star{\Rightarrow}\star, \mu\rangle$	$\longrightarrow$	$\langle v, \mu\rangle$	
ESucceed	$\langle (v{::}G{\Rightarrow}\star){::}\star{\Rightarrow}G, \mu\rangle$	$\longrightarrow$	$\langle v, \mu\rangle$	
EFail	$\langle (v{::}G_1{\Rightarrow}\star){::}\star{\Rightarrow}G_2, \mu\rangle$	$\longrightarrow$	$\texttt{fail}$	if $G_1 \neq G_2$
EInject	$\langle v{::}T{\Rightarrow}\star, \mu\rangle$	$\longrightarrow$	$\langle (v{::}T{\Rightarrow}\lfloor T\rfloor){::}\lfloor T\rfloor{\Rightarrow}\star, \mu\rangle$	if $T \neq \lfloor T\rfloor$, $T \neq \star$
EProject	$\langle v{::}\star{\Rightarrow}T, \mu\rangle$	$\longrightarrow$	$\langle (v{::}\star{\Rightarrow}\lfloor T\rfloor){::}\lfloor T\rfloor{\Rightarrow}T, \mu\rangle$	if $T \neq \lfloor T\rfloor$, $T \neq \star$
ERef	$\langle \texttt{ref}\ v, \mu\rangle$	$\longrightarrow$	$\langle a, \mu[a := v]\rangle$	where a fresh
EDeref	$\langle !a, \mu\rangle$	$\longrightarrow$	$\langle v, \mu\rangle$	where $\mu(a) = v$
EDerefCast	$\langle !(v{::}\mathsf{ref}\,T_1{\Rightarrow}\mathsf{ref}\,T_2), \mu\rangle$	$\longrightarrow$	$\langle (!v){::}T_2{\Rightarrow}T_1, \mu\rangle$	
EUpdt	$\langle a{:=}v, \mu\rangle$	$\longrightarrow$	$\langle a, \mu[a := v]\rangle$	where $\mu(a) = v'$
EUpdtCast	$\langle (v_1{::}\mathsf{ref}\,T_1{\Rightarrow}\mathsf{ref}\,T_2){:=}v_2, \mu\rangle$	$\longrightarrow$	$\langle v_1{:=}v_2{::}T_2{\Rightarrow}T_1, \mu\rangle$	
EApp	$\langle ((\lambda(x{:}T_1){\rightarrow}T_2.\ t)\ v), \mu\rangle$	$\longrightarrow$	$\langle t[x/v], \mu\rangle$	
EAppCast	$\langle ((v_1{::}T_1 \rightarrow T_2{\Rightarrow}T_3 \rightarrow T_4)\ v_2), \mu\rangle$	$\longrightarrow$	$\langle (v_1\ (v_2{::}T_3{\Rightarrow}T_1)){::}T_2{\Rightarrow}T_4, \mu\rangle$	
EAdd	$\langle n_1\ +\ n_2, \mu\rangle$	$\longrightarrow$	$\langle n', \mu\rangle$	where $n_1 + n_2 = n'$

$$\frac{\langle e, \mu\rangle \longrightarrow \langle e', \mu'\rangle}{\langle E[e], \mu\rangle \longmapsto \langle E[e'], \mu'\rangle} \qquad \frac{\langle e, \mu\rangle \longrightarrow \texttt{fail}}{\langle E[e], \mu\rangle \longmapsto \texttt{fail}}$$

Figure 7. Dynamic semantics for the cast calculus.

Once a GTLC program has been translated into the cast calculus, it can then be evaluated using the single-step, substitution-based evaluation rules shown in Figure 7, with syntactic forms and auxiliary functions defined in Figure 6. This reduction system steps from configurations $\langle e, \mu\rangle$, which contain

the expression being evaluated e and a heap μ which maps addresses to values, to either other configurations or to the special state `fail`, which is the result of evaluating a program with a cast error. The EApp rule is standard β-reduction and the EAdd rule is simple addition, and ERef, EDeref, and EUpdt are standard for creating, dereferencing, and mutating a reference on the heap. The rest of the rules fall into two categories: those used for reducing or expanding casts (EIdInt, EIdDyn, ESucceed, EFail, EInject, EProject) and elimination forms for casted values (EDerefCast, EUpdtCast, EAppCast).

Identity casts for the int type and for $\star$ are immediately removed via the EIdInt and EIdDyn rules. Identity casts for other, higher order types do not need to be removed because their elimination forms use other rules, as discussed below.

The EProject and EInject rules handle cases where expressions are casted to and from the dynamic type. Since injections (casts to dynamic) are only values when their sources are ground types G, if a non-ground type is the source of a cast to dynamic the EInject rule decomposes into two casts, first to a ground type corresponding to the original source type (using the partial function $\lfloor T \rfloor$, defined in Figure 6), and from there to $\star$. Likewise, if a value is casted (or "projected") from $\star$ to a static type, the EProject rule casts it to the ground type corresponding to its target type, and from there to the target.

The ESucceed and EFail rules deal with situations where expressions are casted to the dynamic type and then casted again to some other type. For example, the following program in the surface-language GTLC passes the int-typed value 42 into a function that takes dynamic arguments, which in turn passes it into a function that takes int arguments.

$$(\lambda(x{:}\star){\to}\text{int}.\ (\lambda(y{:}\text{int}){\to}\text{int}.\ y)\ x)\ 42$$

After this program is translated to the cast calculus, 42 will be casted to $\star$, and x casted to int:

$$(\lambda(x{:}\star){\to}\text{int}.\ (\lambda(y{:}\text{int}){\to}\text{int}.\ y)\ (x{::}\star{\Rightarrow}\text{int}))\ (42{::}\text{int}{\Rightarrow}\star)$$

By EApp, this expression will reduce to

$$(\lambda(y{:}\text{int}){\to}\text{int}.\ y)\ ((42{::}\text{int}{\Rightarrow}\star){::}\star{\Rightarrow}\text{int})$$

Recall that in the cast calculus, $v{::}\text{int}{\Rightarrow}\star$ is a value but $v{::}\star{\Rightarrow}\text{int}$ is not. The right-hand side of the application in the above expression must instead be evaluated using the ESucceed rule, which removes

both casts altogether, producing the value 42.

$$(\lambda(y{:}\text{int}){\to}\text{int}.\ y)\ 42$$

The ESucceed rule was only able to be applied in this example because the source of the inner cast and the target of the outer cast are the same ground type. This explains the restriction of injected values allowing only ground types as sources: determining whether a value can be unboxed when casted to and then from dynamic requires only a syntactic equality check, corresponding to examining a type tag, and not requiring a deep comparison.

By comparison, the following example changes the target of the outer cast to the type ref $\star$.

$$\begin{aligned}&\langle(\lambda(x{:}\star){\to}\star\ .\ (\lambda(y{:}\text{ref}\,\star){\to}\star\ .\ !y)\ (x{::}\star{\Rightarrow}\text{ref}\,\star))\ (42{::}\text{int}{\Rightarrow}\star), \mu\rangle\\ \longrightarrow\ &\langle(\lambda(y{:}\text{int}){\to}\text{ref}\ \star\ .\ !y)\ ((42{::}\text{int}{\Rightarrow}\star){::}\star{\Rightarrow}\text{ref}\,\star), \mu\rangle\end{aligned}$$

In this case, the EFail rule steps to the `fail` state, indicating that a cast failure has occurred. Indeed, this is the only rule that produces a `fail` state, and this describes how the guarded strategy uses casts to ensure soundness: it detects and reports errors exactly when a value has gone from one ground type to another *through the dynamic type*, and the value's original source and final destination make inconsistent demands about its type. It is also worth noting that the actual form of the value itself is never examined during evaluation—that is, the error was not reached because of runtime typechecking, but simply because two casts with inconsistent types were combined. The fact that 42 is an integer was relevant during the cast insertion process, when the cast $42{::}\text{int}{\Rightarrow}\star$ was inserted, but the evaluation rules themselves only examine types in casts.

Finally, the EAppCast, EDerefCast, and EUpdtCast rules are needed because the callees of function calls and the values being dereferenced or updated in dereference or mutation expressions may actually be proxies. For example, focusing on functions, consider the following program in the GTLC, in which a dynamically-typed function receives a statically typed function of type int $\to$ int as its argument and then calls it on the number 22.

$$(\lambda(x{:}\star){\to}\star\ .\ x\ 22)\ (\lambda(y{:}\text{int}){\to}\text{int}.\ y + 20)$$

Cast insertion will produce the following program in the cast calculus, in which the int $\to$ int function has been casted to $\star$, and the argument of the dynamically typed function has been cast from $\star$ to $\star \to \star$.

$$(\lambda(x{:}\star){\to}\star\,.\,(x{::}\star{\Rightarrow}\star\to\star)\,(22{::}\text{int}{\Rightarrow}\star))\,((\lambda(y{:}\text{int}){\to}\text{int}.\,y+20){::}\text{int}{\to}\text{int}{\Rightarrow}\star)$$

By EInject, the cast from int $\to$ int to $\star$ will be decomposed:

$$(\lambda(x{:}\star){\to}\star\,.\,(x{::}\star{\Rightarrow}\star\to\star)\,(22{::}\text{int}{\Rightarrow}\star))\,(((\lambda(y{:}\text{int}){\to}\text{int}.\,y+20){::}\text{int}{\to}\text{int}{\Rightarrow}\star{\to}\star){::}\star{\to}\star{\Rightarrow}\star)$$

The term $((\lambda(y{:}\text{int}){\to}\text{int}.\,y+20){::}\text{int}{\to}\text{int}{\Rightarrow}\star{\to}\star){::}\star\to\star{\Rightarrow}\star$ is a value, so through EApp, this program partially evaluates to

$$((((\lambda(y{:}\text{int}){\to}\text{int}.\,y+20){::}\text{int}{\to}\text{int}{\Rightarrow}\star{\to}\star){::}\star\to\star{\Rightarrow}\star){::}\star{\Rightarrow}\star\to\star)\,(22{::}\text{int}{\Rightarrow}\star)$$

Through ESucceed, this becomes

$$((\lambda(y{:}\text{int}){\to}\text{int}.\,y+20){::}\text{int}\to\text{int}{\Rightarrow}\star\to\star)\,(22{::}\text{int}{\Rightarrow}\star)$$

The terms $(\lambda(y{:}\text{int}){\to}\text{int}.\,y+20){::}\text{int}\to\text{int}{\Rightarrow}\star\to\star$ and $22{::}\text{int}{\Rightarrow}\star$ are both values in this calculus, and so cannot be further evaluated, and instead the top-level application must be eliminated. However, the EApp rule cannot apply here, since the left-hand side value of the application does not have the form $\lambda(x{:}T){\to}T.\,t$. Instead, the EAppCast rule uses the cast on the function to ensure that the function's argument comports to the type it is expected to have by the function cast's source type, and that the result of the function call comports to the type expected to have by the target type. To do this, the function cast decomposes into two casts: the argument is casted from the domain of the target type to the domain of the source type (reversing the direction of the cast due to contravariance), and then (after this casted argument is passed to the underlying function value) the result of the application is casted from the codomain of the source type to the codomain of the target type, as illustrated below:

$$((\lambda(y{:}\text{int}){\to}\text{int}.\,y+20)\,((22{::}\text{int}{\Rightarrow}\star){::}\star{\Rightarrow}\text{int})){::}\text{int}{\Rightarrow}\star$$

Through ESucceed, EApp, and EAdd, this finally evaluates to the result $42{::}\text{int}{\Rightarrow}\star$.

Siek et al. [82] proved that guarded enforcement strategy satisfies the criteria for gradual typing defined in Section 1.2, although their formulation of the gradually typed lambda calculus is slightly different

from that in Figure 1 and their statements of Criteria 2.2 and 2.4 are specific to the guarded enforcement strategy.

2. Further Research in Gradual Typing

Beyond the gradually typed lambda calculus and its semantics as presented above, research in gradual typing has extended in many directions. In this section I will summarize the current state of research in gradual typing.

2.1. Origins of Gradual Typing. Much work has gone into integrating static and dynamic typing in the same system. Early work on the subject includes the `dynamic` of Abadi et al. [3] and the quasi-static typing of Thatte [91], as well as Strongtalk [22]. Cecil [24] and the Bigloo variant of Scheme [73] allow optional type annotations, but do not encode runtime checks between static and dynamic code. Gray et al. [40] extended Java with contracts to enable interoperability with Scheme.

Siek and Taha [75] introduced gradual typing as an approach to combining static and dynamic typing with the consistency relation. Contemporary work that is similarly foundational to the state of the art in gradual typing includes that of Tobin-Hochstadt and Felleisen [92], which provided a framework for evolving a multi-module program from dynamic to static at a per-module basis, and the hybrid type checking of Gronski et al. [43], which combined static typechecking with dynamic checks and automated theorem proving in order to synthesize dynamic typing and refinement types.

2.2. Exploring the Language Feature Space of Gradual Typing. The original work of Siek and Taha [75] only applied gradual typing to functions. Herman et al. [48] extended gradual typing to mutable references, and Siek and Taha [76] defined gradual typing for objects, combining consistency and subtyping into a unified *consistent-subtyping* relation. Siek and Vachharajani [77] combined gradual typing with unification-based type inference, à la ML, and their work was simplified and clarified by Garcia and Cimini [37]. Ahmed et al. [5] combined gradual typing with parametric polymorphism using runtime sealing to guarantee parametricity, and Ina and Igarashi [52] combine gradual typing with Java-style generics. Takikawa et al. [88] explore an approach to gradual typing with first-class classes with

inheritance, as used in Racket. Wolff et al. [106] develop gradual typestate, which combines static and dynamic checking to provide permission guarantees in imperative languages.

Garcia et al. [38] developed a new foundation for gradual typing based on abstract interpretation called "Abstracting Gradual Typing," and use this technique to apply gradual typing to record subtyping. They show that languages developed with this technique straightforwardly satisfy the gradual guarantee. Similarly, Cimini and Siek [26, 27] show a technique for automatically constructing gradually typed languages with arbitrary language features when given a corresponding statically typed language as an input.

2.3. Blame-Tracking in Gradual Typing. Blame-tracking, as a technique to point programmers towards the causes of runtime violations rather than simply where those violations arise, was developed for contract systems well before the development of gradual typing [35]. Tobin-Hochstadt and Felleisen [92] extended this notion to languages that combine static and dynamic modules, where the region of code that violated the expectations of static code is blamed when the violation is detected. They also showed that statically typed modules cannot be responsible for an error and therefore all errors originate in untyped code. Wadler and Findler [104] refined this property for a gradually typed lambda calculus and defined the blame-subtyping theorem, described in Section 1.2.3. Siek et al. [80] describe two variations on the rules of blame-tracking, describing whether both upcasts and downcasts on the blame-subtyping lattice (Figure 2) can be blamed or only downcasts. Dimoulas et al. [29] introduce *complete monitoring*, a correctness criterion for blame tracking that which uses a notion of ownership, where different components of a program own and take responsibility for their values. Complete monitoring ensures that if an error occurs that blames a party, then a value owned by that party must have crossed a boundary and this boundary-crossing led to the error.

2.4. Semantics of Gradual Typing. Even for simple gradually-typed lambda calculi, the rules defined in Figure 7 are just one possible semantics for gradual typing. Siek et al. [80] and Siek and Garcia [74] explore several variations of the semantics of gradually typed lambda calculi, identifying the rules shown above as the *lazy* semantics, so-called because cast errors are only detected at use-sites, as well as showing an alternative *eager* approach.

Confined gradual typing, due to Allende et al. [12], enhances the language of types in a gradually typed lambda calculus with qualifiers describing whether values inhabiting those types have passed through dynamically typed code in the past, and whether they will flow into dynamically typed code in the future. These qualifiers allow for reasoning about efficiency and the presence of proxies in a system with the guarded enforcement strategy.

As discussed below in Section 3.1.1 and in greater detail in Chapter 4, the use of proxies in gradually typed languages, as required by the guarded enforcement strategy, can degrade performance. Wrigstad et al. [108] address this by introducing a distinction between *like types* and *concrete types*. Concrete types are the usual types of a statically-typed language and incur zero run-time overhead, because they may be inhabited only by non-proxied values. However, because of this restriction, interoperability between dynamic code and concretely typed code is limited: dynamically-typed values cannot flow into concretely typed locations nor vice-versa. Like types, on the other hand, may interact with dynamically typed values using the guarded strategy, but they incur run-time overhead.

2.5. Representations of Casts. Casts can be represented in various ways that achieve the same semantics but may display different time and space complexity and characteristics. The original work of Siek and Taha [75] does not treat values casted from one function type to another as values, but instead evaluates such casts to a fresh function that performs the cast:

$$v{::}T_1 \to T_2 \Rightarrow T_3 \to T_4 \longrightarrow \lambda(x{:}T_3) \to T_4.\ ((v\ (x{::}T_3 \Rightarrow T_1)){::}T_2 \Rightarrow T_4) \quad \text{with fresh } x$$

This approach treats function proxies as simply functions, rather than a special proxy value.

Herman et al. [48] introduced the use of the *coercion calculus* of Henglein [47] to represent casts. Coercions are composed of *injections* of the form $T!$, which describes a conversion from T to $\star$; projections $T?$ which convert values from $\star$ to T (and which may fail when evaluated, if the value is not an injected value of type T), conversions from one higher order type to another type with the same constructor (for example, a coercion from one function type to another would have the form $c_1 \to c_2$ where c_1 and c_2 are other coercions, which describe the coercion to be applied to the function's argument and result respectively), and sequences of coercions. Herman et al. [48] prove that any sequence of casts can

be reduced to at most three coercions, allowing for conversions between types to be described space-efficiently. Siek and Wadler [79] represent casts with "threesomes," written $T_1 \overset{T_2}{\Longrightarrow} T_3$, which describe a cast from T_1 to T_3 "through" T_2, and they similarly show that any sequence of casts can be represented with a single threesome.

3. Gradual and Optional Typing in Practice

Interest in gradual typing is not limited to academic research, and a number of languages use gradual typing or related ideas.

3.1. Gradual Typing for Practical Languages. The guarded enforcement strategy has been implemented in a number of real-world languages, most prominently Typed Racket [92], a language which derives from Racket, a Scheme/LISP-based dynamically typed impure functional language.

3.1.1. Typed Racket. In Typed Racket the programmer's choice between static and dynamic typing occurs at the level of modules rather than individual identifiers: a module is either fully statically typed or entirely dynamically typed (i.e. written in standard Racket). However, typed modules may depend on untyped modules and vice versa, so Typed Racket still faces the same problem of ensuring soundness in typed code that more fine-grained gradually typed languages do. Typed Racket's particular implementation of the guarded semantics uses *chaperones*, a feature of the core Racket runtime that implements proxies that can apply checks or install other chaperones at their boundaries [85]. (Racket also provides a more powerful structure in the form of *impersonators*, which are more powerful in what kinds of operations they can perform at their boundaries and which are needed for e.g. parametricity-enforcing proxies on polymorphic functions.) With this feature, typed and untyped Racket code can interoperate soundly, with reasonable blame-tracking behavior and with the ability to evolve between dynamic and static as per the gradual guarantee (with the caveat that entire modules must be transitioned between static and dynamic).

However, Typed Racket also needs to be performant to fulfill its goal of real-world use. In many cases, Typed Racket is able to leverage static type information to improve the efficiency of operations—for example, vector dereferencing (a.k.a. array access) can soundly be performed without bounds checking

if the index is known (by refinement typing) to be within the length of the vector [54], so when Typed Racket is used as an entirely statically typed language (i.e. not using any untyped modules), it achieves this goal. However, Takikawa et al. [90] showed that this is often not the case when statically and dynamically typed code interact. They established this by studying many different *configurations* of the same basic program by varying whether each module in the program is statically or dynamically typed. These configurations form a lattice with a height of the number of modules in the program, where the top configuration is entirely statically typed and the bottom configuration entirely dynamic, a structure referred to as the *typing* or *performance lattice*.

With this lens, Takikawa et al. showed that while configurations at the top and bottom of the lattice are very performant, some combinations of static and dynamic are very slow—to the point of over $100\times$ overhead compared to either the top or bottom. This result threatens the practical usefulness of Typed Racket as a gradually typed language: adding or removing types from part of a program can catastrophically affect the performance of the program in unpredictable ways, rendering impractical the gradual evolution from dynamic to static that gradual typing offers. Takikawa et al. [90] also show that this performance degradation is a direct result of the use of proxies to mediate between static and dynamic code. This is not an unexpected result; it is obvious that the need to create and install proxies (as performed by cast insertion) and the need to apply casts to the input and output of values with higher-order types (corresponding to the EAppCast, EUpdtCast, and EDerefCast rules in Figure 7) requires overhead that scales with the number of proxies in the program—and when interaction between static and dynamic is frequent, many or most values will be proxied.

3.1.2. Other Languages. While Typed Racket is the most prominent gradually typed language in practical use, other work has extended sound gradual typing in various forms to other languages, though without widespread adoption.

Responding to the performance issues of Typed Racket, Muehlboeck and Tate [63] designed a gradually typed language called Nom with nominal object types and without structural types or functions and examined its performance across the lattices of several benchmarks, finding negligible overhead.

Gradualtalk, a gradually-typed variant of the Smalltalk programming language, was developed by Allende et al. [10] with an eye towards ensuring sound interaction with existing Smalltalk programs. Allende et al. [11] develop a cast insertion scheme for Gradualtalk intended to facilitate interaction between typed libraries and untyped clients without requiring client recompilation. Their approach, called *execution semantics*, inserts casts on function arguments at the entry points of the functions, rather than at their call sites. This allows dynamically typed code to be compiled without needing to insert casts used within typed code, and is therefore similar to the transient enforcement strategy for gradual typing, developed below in Chapters 4 and 5. However, Gradualtalk's casts still generate proxies à la the guarded strategy.

C# [16] provides a dynamic type, though implicit conversions to and from it are limited. Rastogi et al. [67] analyzed the performance of ActionScript, which similarly has limited interaction between static and dynamic code, by taking fully-typed benchmarks and removing all type annotations except for those on interfaces. They found that this resulted in significant overhead compared to the original fully-typed versions. They then used a inference-based to reconstruct type annotations; this fully recovered performance in most benchmarks.

3.2. Optional Typing. While the guarded approach to gradual typing has problems in practice, the appeal of combining static and dynamic typing has undeniable practical appeal. In particular, as dynamically typed scripting languages such as Python, Lua, Clojure, and especially JavaScript have become widespread, there has been a practical software engineering desire to have the ability to apply static typing where possible. As a result, optionally typed languages that compile to these existing dynamic languages have become popular. Such languages allow programmers to use static type systems to reason about and document their code, while allowing their code, once compiled, to execute on off-the-shelf runtimes and interoperate with existing codebases native to the target language of compilation.

JavaScript is an especially important backend for optionally typed languages, with several competing approaches taken. TypeScript [17, 61] is a syntactic superset of JavaScript that programmers with type annotations and support for generics and classes, and is intended to support common JavaScript idioms

without the need to refactor code. The TypeScript community also maintain type definitions for standard JavaScript code, allowing typed interoperability. Flow [25, 31], by contrast, places less emphasis on type annotations and a greater emphasis on type inference, using a flow-based intermodular type inference algorithm to detect static type errors. As optionally typed languages, both TypeScript and Flow execute as standard JavaScript without runtime checks, but the completeness and expressivity of their type systems means that, in principle, users should be able to write their programs in an entirely statically typed style should they desire to.

Beyond JavaScript, other dynamic languages are also targeted by optionally typed languages. Python has two robust type systems that target it, the Mypy project [57] and Pyre [32]. The Python language itself, while dynamically typed, syntactically supports annotations and provides recommendations for the syntax of type annotations [96]; both Mypy and Pyre target these annotations. Typed Clojure [20, 19] provides optional typing for Clojure, including Java interoperability and occurrence typing, and Typed Lua [58] does the same for Lua.

3.2.1. Towards gradually typed TypeScript. Existing approaches do exist to applying sound gradual typing to optionally typed languages, especially TypeScript. TS* [86] and Safe Typescript [68] are further variants of TypeScript that are sound and allow some interaction between static and dynamic code. They avoid the guarded enforcement strategy and do not rely on proxies, instead monotonically "locking down" values at static types as they pass from dynamic to static code, in a manner similar to the monotonic enforcement strategy I present in Chapter 3. However, compared to full gradual typing (including the monotonic strategy), these languages are restrictive in what kinds of programs are accepted. In these systems, implicit conversions are only allowed on upcasts (using a subtyping lattice similar to that shown in Figure 2). For example, a function of the type any $\rightarrow$ any cannot be cast to bool $\rightarrow$ bool (where any corresponds to $\star$). The *gradual guarantee* (Criterion 2.5) therefore does not hold.

The approach of *concrete* and *like* types, as discussed above in Section 2.4, has also been applied to TypeScript by Richards et al. [72], again partially avoiding the guarded strategy. They designed a sound variant of TypeScript called StrongScript, which allows for interaction between soundly typed StrongScript code and arbitrary external JavaScript. However, because of the incompatibility of concrete and like

types, it is not straightforward to evolve a program in StrongScript from dynamic to static, as is frequently desirable—similar to TS*. As a result, it does not satisfy the gradual guarantee.

CHAPTER 3

Monotonic References for Efficient Gradual Typing

1. Introduction

Gradual typing enables the seamless integration of static and dynamic typing, but the goal of providing all the benefits of static typing, such as efficiency, in statically-typed regions, is elusive: there are challenges regarding the efficiency of gradual typing due to the traditional *guarded* runtime enforcement semantics. One issue concerns mutable references in statically-typed regions of code. Consider the following statically-typed function f that dereferences its parameter x.

$$\texttt{let } f = \lambda x{:}\text{Ref Int}.\ !x \texttt{ in}$$
$$f(\texttt{ref } 4);$$
$$f(\texttt{ref } (4 \texttt{ as } \star))$$

In the first application of f, a normal reference to an integer flows into f. For the second application, I allocate a reference of type ref $\star$ then implicitly cast it to ref int before applying f. According to the guarded enforcement strategy of Herman et al. [48], this cast wraps the reference in a proxy which performs dynamic checks on reads and writes. Thus code generated for the dereference in the body of f must inspect the reference to find out whether it is a normal reference or a proxied reference, and in the proxied case, apply a coercion.

Before discussing solutions to this problem, recall the *gradual guarantee* of Boyland [21] and Siek et al. [81], an important property of the standard semantics for mutable references, and of gradual typing in general. The gradual guarantee promises that removing type annotations, or changing type annotations to be less precise, does not affect the behavior of a program: it should still type check and the result should be the same modulo proxies. (Adding or making type annotations more precise, on the other hand can sometimes induce static type errors and runtime cast errors.) Consider the statically-typed program on the left that allocates a reference to an integer and then dereferences it from within

a function. In the code on the right, I change the annotation on h from ref int to ref $\star$, but the program still type checks and the result remains 42.

$$
\begin{array}{lcl}
\texttt{let}\ r = \texttt{ref}\ 42\ \texttt{in} & & \texttt{let}\ r = \texttt{ref}\ 42\ \texttt{in} \\
\texttt{let}\ f = \lambda h{:}\text{ref int}.\ !h & \Longrightarrow & \texttt{let}\ f = \lambda h{:}\text{ref}\ \star\ .\ !h \\
\texttt{in}\ \ f(r) & & \texttt{in}\ \ f(r)
\end{array}
$$

As discussed in Chapter 2, Section 2.4, Wrigstad et al. [108] address the efficiency problem by introducing a distinction between *like types* and *concrete types*. The distinction between like types and concrete types achieves the efficiency goals, but the restrictions in their type system means that removing concrete type annotations, as in the above example, can trigger a static type error—violating the gradual guarantee.

In this chapter I investigate this run-time overhead problem in the context of the gradually-typed lambda calculus with mutable references. I propose an enforcement strategy, the *monotonic references* strategy, that enables the compilation of statically-typed regions to machine code that is free of any of the indirection or run-time checking associated with dynamic typing, like boxing or bit tags. Monotonic references allow dynamically-typed values to flow into code with (concrete) static types. When a reference flows through a cast, the cast may coerce its underlying heap cell to become more statically typed. In general, this means that values in the heap may evolve monotonically with respect to the precision relation (Section 2).

Monotonic references preserve a global invariant that a value in the heap is at least as precise as any reference that points to it. Thus, a static reference always points to a value of the same type, so there is no overhead associated with reading or writing through the reference: the reads and writes may be implemented as machine loads and stores. By a *static* reference I mean that there are no occurrences of the dynamic type $\star$ in the pointed-to type of the reference, such as ref int and ref $(\text{int} \times \text{bool})$. Reads and writes to references that are not static, such as ref $\star$ and ref $(\star \times \text{bool})$, still require casts: the dynamic regions of code have to pay their own way. The intermediate representation that I compile to contains different instructions for fast, static loads and stores versus non-static loads and stores that require casts.

Swamy et al. [86] and Rastogi et al. [68] integrate static and dynamic typing in the context of TypeScript with the TS* and Safe TypeScript languages. Both use a notion of monotonicity in the heap, but with respect to subtyping, treating $\star$ as a universal supertype, instead of with respect to the precision relation. Because these languages compile to JavaScript, they inherit the overhead of dynamic typing, whereas with monotonic references, the overhead of dynamic typing occurs only in dynamically-typed code. In the example above, making the type annotation on h less precise causes TS* to halt the program with a cast error at the implicit cast from ref int to ref $\star$. TS* does not allow casts from one mutable reference type to a different one because its references are invariant with respect to subtyping. Thus, TS* does not satisfy the gradual guarantee.

In gradually-typed languages with higher-order features such as first-class functions and objects, blame tracking plays an important role in providing meaningful error messages when casts fail. Blame tracking enables fine-grained guarantees, via a blame theorem, regarding which regions of the code are statically type safe. In this chapter I present blame tracking for monotonic references and prove a blame theorem. This design uses the labeled types of Siek and Wadler [79] as run-time type information (RTTI), together with three new operations on labeled types: a bidirectional cast operator that captures the dual read-/write nature of mutable references, a merge operator that models how casts on separate aliases to the same heap cell interact over time, and an operator that casts heap cells between labeled types.

To summarize, this chapter presents a new semantics for gradually-typed mutable references that delivers guaranteed efficiency for the statically-typed parts of a program, maintains type safety, and provides blame tracking, while continuing to enable fine-grained migration between static and dynamic code. This chapter makes the following technical contributions:

(1) I define the semantics of monotonic references (Sections 3 and 5).
(2) I discuss a proof of type safety, mechanized in Isabelle (Section 4).
(3) I augment monotonic references with blame tracking and prove the blame-subtyping theorem (Section 6).

I review the gradually-typed lambda calculus with references in Section 2 and discuss the run-time overhead associated with mutable references. I address an implementation concern regarding strong updates in Section 7. The chapter concludes in Section 9.

2. Background and Problem Statement

Figures 1 and 2 review the syntax and static semantics of a gradually-typed lambda calculus with references, generally similar to the calculus shown in Section 1.3.1 of the previous chapter, but with a larger space of expressions and with blame-tracking. The *consistency* relation, also defined in Figure 2, is used where the equality relation would be in a statically-typed lambda calculus. The consistency relation enables implicit casts to and from $\star$. (In contrast, an object-oriented language only allows implicit casts to the top `Object` type.) This consistency relation is a congruence, even for reference types [48], which differs from the original treatment of references as invariant [75]. The more flexible treatment of references enables the passing of references between more and less dynamically typed regions of code, but is also the source of the difficulties that I solve in this chapter. The precision relation, which says whether one type is more or less dynamic than another, is also defined in Figure 2, and is closely related to consistency. Two types are consistent when there exists a greatest lower bound with respect to the precision relation. This relation is also known as naïve subtyping [104].

In this calculus, all of the types, except for $\star$, classify unboxed values. So, for example, int is the type for native integers (e.g. 64-bit integers). The auxiliary relations *fun*, *pair*, and *ref*, defined in Figure 2, implement pattern matching on types, enabling a more concise presentation of the typing rules compared to prior presentations of gradual type systems. Labels ℓ represent source code locations that are captured during parsing.

The dynamic semantics of the gradually-typed lambda calculus is defined by a type-directed translation to the coercion calculus [47], using the guarded enforcement strategy due to Herman et al. [48].

Each use of consistency between types T_1 and T_2 in the type system, and each use of one of the auxiliary relations, becomes an explicit cast from T_1 to T_2. The coercion calculus expresses casts in terms of combinators that say how to cast from one type to another. Figure 3 gives the compilation of casts into

$$
\begin{array}{lrcl}
\text{Base types} & B & ::= & \text{int} \mid \text{bool} \\
\text{Types} & T & ::= & B \mid T \to T \mid T \times T \mid \text{ref}\, T \mid \star \\
\text{Labels} & \ell & & \\
\text{Operators} & \mathit{op} & ::= & \texttt{plus} \mid \texttt{minus} \mid \texttt{is} \mid \cdots \\
\text{Expressions} & e & ::= & k \mid \mathit{op}^{\ell}(\vec{e}) \mid x \mid \lambda x{:}T.\, e \mid (e\, e)^{\ell} \mid e\ \texttt{as}^{\ell}\ T \mid \\
 & & & (e, e) \mid \texttt{fst}^{\ell} e \mid \texttt{snd}^{\ell} e \mid \texttt{ref}\ e \mid !^{\ell} e \mid e{:=}^{\ell} e \\
 & & & \lambda x.\, e \equiv \lambda x{:}\star .\, e
\end{array}
$$

Figure 1. Syntax for the gradually typed λ calculus with mutable references.

coercions, written $(T \Rightarrow^{\ell} T) = c$. The compilation of gradually-typed terms into the coercion-based calculus is otherwise straightforward, so I give just the function application rule as an example:

$$
\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : T_1 \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : T_2 \qquad \mathit{fun}(T_1, T_{11}, T_{12}) \qquad T_2 \sim T_{11} \qquad (T_1 \Rightarrow^{\ell} T_{11} \to T_{12}) = c_1 \qquad (T_2 \Rightarrow^{\ell} T_{11}) = c_2}{\Gamma \vdash (e_1\, e_2)^{\ell} \rightsquigarrow e_1'\langle c_1 \rangle\, e_2'\langle c_2 \rangle : T_{12}}
$$

Figures 4, 5, and 6 define the coercion-based calculus. The parts of the definition related to references are highlighted, as they are of particular interest here. I review the coercion calculus in the context of discussing the run-time overhead problem in the next subsection. For an introduction to the coercion calculus, I refer to Henglein [47].

2.1. Run-time overhead in fully-static code. Recall the example in Section 1 in which the dereference of a statically-typed reference must first check whether the reference is proxied or not.

$$
\begin{array}{l}
\texttt{let}\ f = \lambda x{:}\text{Ref Int}.\, !x\ \texttt{in} \\
f(\texttt{ref}\ 4); \\
\texttt{let}\ r = \texttt{ref}\ (4\ \texttt{as}\ \star)\ \texttt{in}\ \ f(r)
\end{array}
$$

The overhead can be seen in the dynamic semantics (Figure 6), where there are two reduction rules for dereferencing: (Deref) and (DerefCast), and two reduction rules for updating references: (Update) and (UpdateCast). Another way to look at this problem is that there are two canonical forms of type ref int, a plain address a and also a value wrapped in a reference coercion, $v\langle \text{ref}\, c_1\ c_2 \rangle$, so operations on values

Consistency $\boxed{T \sim T}$

$$\star \sim T \quad T \sim \star \quad B \sim B \quad \frac{T_1 \sim T_2}{\mathsf{ref}\, T_1 \sim \mathsf{ref}\, T_2}$$

$$\frac{T_1 \sim T_3 \qquad T_2 \sim T_4}{T_1 \to T_2 \sim T_3 \to T_4} \qquad \frac{T_1 \sim T_3 \qquad T_2 \sim T_4}{T_1 \times T_2 \sim T_3 \times T_4}$$

Precision $\boxed{T \sqsubseteq T}$

$$T \sqsubseteq \star \quad B \sqsubseteq B \quad \frac{T_1 \sqsubseteq T_2}{\mathsf{ref}\, T_1 \sqsubseteq \mathsf{ref}\, T_2}$$

$$\frac{T_1 \sqsubseteq T_3 \qquad T_2 \sqsubseteq T_4}{T_1 \to T_2 \sqsubseteq T_3 \to T_4} \qquad \frac{T_1 \sqsubseteq T_3 \qquad T_2 \sqsubseteq T_4}{T_1 \times T_2 \sqsubseteq T_3 \times T_4}$$

Expression typing $\boxed{\Gamma \vdash e : T}$

$$\frac{k : B}{\Gamma \vdash k : B} \qquad \frac{\Gamma \vdash \vec{e} : \vec{T} \qquad \mathit{op} : \vec{B} \to B \qquad \vec{T} \sim \vec{B}}{\Gamma \vdash \mathit{op}^{\ell}(\vec{e}) : B} \qquad \frac{\Gamma \vdash e : T_1 \qquad T_1 \sim T_2}{\Gamma \vdash e\ \mathtt{as}^{\ell}\ T_2 : T_2}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{\Gamma(x \mapsto T_1) \vdash e : T_2}{\Gamma \vdash \lambda x{:}T_1.\, e : T_1 \to T_2} \qquad \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad \mathit{fun}(T_1, T_{11}, T_{12}) \qquad T_2 \sim T_{11}}{\Gamma \vdash (e_1\ e_2)^{\ell} : T_{12}}$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1, e_2) : T_1 \times T_2} \qquad \frac{\Gamma \vdash e : T \qquad \mathit{pair}(T, T_1, T_2)}{\Gamma \vdash \mathtt{fst}^{\ell} e : T_1} \qquad \frac{\Gamma \vdash e : T \qquad \mathit{pair}(T, T_1, T_2)}{\Gamma \vdash \mathtt{snd}^{\ell} e : T_2}$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \mathtt{ref}\, e : \mathsf{ref}\, T} \qquad \frac{\Gamma \vdash e : T \qquad \mathit{ref}(T, T')}{\Gamma \vdash !^{\ell} e : T'} \qquad \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad \mathit{ref}(T_1, T_1') \qquad T_2 \sim T_1'}{\Gamma \vdash e_1 :=^{\ell} e_2 : T_1}$$

Type matching

$$\mathit{fun}(T_{11} \to T_{12}, T_{11}, T_{12}) \quad \mathit{fun}(\star, \star, \star)$$

$$\mathit{pair}(T_{11} \times T_{12}, T_{11}, T_{12}) \quad \mathit{pair}(\star, \star, \star)$$

$$\mathit{ref}(\mathsf{ref}\, T, T) \quad \mathit{ref}(\star, \star)$$

Figure 2. Gradually-typed λ calculus with mutable references

of this type need to dispatch on which form occurs at runtime. To eliminate this overhead a design is needed with only a single canonical form for values of reference type.

The run-time overhead for references affects every read and write to the heap and is particularly detrimental in tight loops over arrays. When adding support for contracts to mutable data structures in

$$\boxed{(T \Rightarrow^{\ell} T) = c}$$

$$\begin{array}{llll} (B \Rightarrow^{\ell} B) & = \iota & (I \Rightarrow^{\ell} \star) & = I! \\ (\star \Rightarrow^{\ell} \star) & = \iota & (\star \Rightarrow^{\ell} I) & = I?^{\ell} \end{array}$$

$$(T_1 \to T_2) \Rightarrow^{\ell} (T_1' \to T_2') = (T_1' \Rightarrow^{\ell} T_1) \to (T_2 \Rightarrow^{\ell} T_2')$$

$$(T_1 \times T_2) \Rightarrow^{\ell} (T_1' \times T_2') = (T_1 \Rightarrow^{\ell} T_1') \times (T_2 \Rightarrow^{\ell} T_2')$$

$$\mathsf{ref}\, T \Rightarrow^{\ell} \mathsf{ref}\, T' = \mathsf{ref}\, (T \Rightarrow^{\ell} T')\, (T' \Rightarrow^{\ell} T)$$

Figure 3. Compile casts to coercions

$$\begin{array}{llll} \text{Expressions} & e & ::= & k \mid \mathit{op}(\vec{e}) \mid x \mid \lambda x.\, e \mid e\, e \mid (e, e) \mid \mathtt{fst}\, e \mid \mathtt{snd}\, e \mid \\ & & & \mathtt{ref}\, e \mid\, !e \mid e{:=}e \mid e\langle c \rangle \mid \mathtt{blame}\, \ell \\ \text{Injectibles} & I & ::= & B \mid T \to T \mid T \times T \mid \mathsf{ref}\, T \\ \text{Coercions} & c & ::= & \iota \mid I?^{\ell} \mid I! \mid c \to c \mid c \times c \mid c\,;c \mid \mathsf{ref}\, c\, c \\ \text{Values} & v & ::= & k \mid \lambda x.\, e \mid (v, v) \mid v\langle I! \rangle \mid a \mid v\langle \mathsf{ref}\, c\, c \rangle \\ \text{Heap} & \mu & ::= & \emptyset \mid \mu(a \mapsto v) \\ \text{Heap Typing} & \Sigma & ::= & \emptyset \mid \Sigma(a \mapsto T) \\ \text{Frames} & F & ::= & \mathit{op}(\vec{v}, \square, \vec{e}) \mid \square\, e \mid v\, \square \mid (\square, e) \mid (v, \square) \mid \mathtt{fst}\, \square \mid \mathtt{snd}\, \square \mid \\ & & & \mathtt{ref}\, \square \mid\, !\square \mid \square{:=}e \mid v{:=}\square \mid \square\langle c \rangle \end{array}$$

Figure 4. Syntax for the coercion-based calculus with mutable references

Racket, Strickland et al. [85, Figure 9] measured this overhead at approximately 25% for fully-typed code on a bubble-sort microbenchmark.

2.2. Non-determinism in multi-threaded code. The guarded enforcement strategy for mutable references produces an error only if type inconsistency is witnessed by some read or write to a particular reference, so in a non-deterministic multi-threaded program, whether a check will fail at run-time is difficult to predict.

Coercion typing $\boxed{c : T \Rightarrow T}$

$$\iota : T \Rightarrow T \quad \frac{c_1 : T_3 \Rightarrow T_1 \qquad c_2 : T_2 \Rightarrow T_4}{c_1 \to c_2 : (T_1 \to T_2) \Rightarrow (T_3 \to T_4)}$$

$$I?^{\ell} : \star \Rightarrow I \quad \frac{c_1 : T_1 \Rightarrow T_3 \qquad c_2 : T_2 \Rightarrow T_4}{c_1 \times c_2 : (T_1 \times T_2) \Rightarrow (T_3 \times T_4)}$$

$$I! : I \Rightarrow \star \quad \frac{c_1 : T_1 \Rightarrow T_2 \qquad c_2 : T_2 \Rightarrow T_3}{c_1 \,;\, c_2 : T_1 \Rightarrow T_3}$$

$$\frac{c_1 : T_1 \Rightarrow T_2 \qquad c_2 : T_2 \Rightarrow T_1}{\mathsf{ref}\, c_1\, c_2 : \mathsf{ref}\, T_1 \Rightarrow \mathsf{ref}\, T_2}$$

Expression typing $\boxed{\Gamma; \Sigma \vdash e : T}$

$$\cdots \quad \frac{\Sigma(a) = T}{\Gamma; \Sigma \vdash a : T} \qquad \frac{\Gamma; \Sigma \vdash e : T_1 \qquad c : T_1 \Rightarrow T_2}{\Gamma; \Sigma \vdash e\langle c \rangle : T_2}$$

Figure 5. Type system for the coercion-based calculus with mutable references

The contract system in Racket implements the guarded semantics [36]. For example, the following program sometimes fails and blames `b1`, sometimes fails and blames `b2`, and sometimes succeeds, as explained below.

```
#lang racket
(define b (box #f))
(define/contract b1 (box/c integer?) b)
(define/contract b2 (box/c string?)  b)

(thread (lambda ()
          (for ([i 2])
            (set-box! b1 5)
```

Reduction rules for functions, primitives, and pairs $\boxed{e \longrightarrow e}$

$$\begin{array}{llll} (\lambda x.\, e)\, v & \longrightarrow [x := v]e & \texttt{fst}\,(v_1, v_2) & \longrightarrow v_1 \\ \mathit{op}(\vec{k}) & \longrightarrow \delta(\mathit{op}, \vec{k}) & \texttt{snd}\,(v_1, v_2) & \longrightarrow v_2 \end{array}$$

Cast reduction rules $\boxed{e \longrightarrow_c e}$

$$\begin{array}{rl} v\langle \iota \rangle & \longrightarrow_c v \\ v\langle I_1! \rangle \langle I_2?^\ell \rangle & \longrightarrow_c v\langle I_1 \Rightarrow^\ell I_2 \rangle \quad \text{if } I_1 \sim I_2 \\ v\langle I_1! \rangle \langle I_2?^\ell \rangle & \longrightarrow_c \texttt{blame}\,\ell \quad \text{if } I_1 \not\sim I_2 \\ v\langle c_1 \to c_2 \rangle & \longrightarrow_c \lambda x.\, v\,(x\langle c_1 \rangle)\langle c_2 \rangle \\ (v_1, v_2)\langle c_1 \times c_2 \rangle & \longrightarrow_c (v_1\langle c_1 \rangle, v_2\langle c_2 \rangle) \\ v\langle c_1 \,;\, c_2 \rangle & \longrightarrow_c v\langle c_1 \rangle \langle c_2 \rangle \end{array}$$

Reference reduction rules $\boxed{e, \mu \longrightarrow_r e, \mu}$

(AllocRef) $\texttt{ref}\, v, \mu \longrightarrow_r a, \mu(a \mapsto v) \qquad \text{if } a \notin \mathit{dom}(\mu)$

(Deref) $!a, \mu \longrightarrow_r \mu(a), \mu$

(DerefCast) $!(v\langle \text{ref}\, c_1\, c_2 \rangle), \mu \longrightarrow_r (!v)\langle c_1 \rangle, \mu$

(Update) $a{:=}v, \mu \longrightarrow_r a, \mu(a \mapsto v)$

(UpdateCast) $v_1\langle \text{ref}\, c_1\, c_2 \rangle{:=}v_2, \mu \longrightarrow_r v_1{:=}v_2\langle c_2 \rangle, \mu$

State reduction rules

$$\frac{e \longrightarrow e'}{e, \mu \longrightarrow e', \mu} \qquad \frac{e \longrightarrow_c e'}{e, \mu \longrightarrow e', \mu} \qquad \frac{e, \mu \longrightarrow_r e', \mu'}{e, \mu \longrightarrow e', \mu'}$$

$$\frac{e, \mu \longrightarrow e', \mu'}{F[e], \mu \longrightarrow F[e'], \mu'} \qquad F[\texttt{blame}\,\ell], \mu \longrightarrow \texttt{blame}\,\ell, \mu$$

Figure 6. Reduction rules for the coercion-based calculus with mutable references

```
(sleep 0.000000001)
(add1 (unbox b1)))))
```

```
(thread (lambda ()
          (for ([i 2])
            (set-box! b2 "hello")
            (sleep 0.000000001)
            (string-append "world" (unbox b2)))))
```

The program creates a single heap cell `b`, and accesses it through two distinct proxies, `b1` and `b2`, each with its own dynamic check. When the two threads do not interleave, the program succeeds, but if the second thread changes `b2` to contain a string between the `set-box!` and `unbox` calls for `b1`, the system halts, blaming one of the parties.

In contrast, if `box/c` implemented monotonic references, then an error would *deterministically* occur when `define/contract` is used for the second time.

3. Monotonic References Without Blame

Figures 7, 8, 9, and 10 define the syntax and semantics of the new coercion calculus with monotonic references, but without blame. Figure 12 defines the compilation of casts to monotonic coercions, also without blame. The addition of blame adds considerable complexity, so I postpone its treatment to Section 5. Typical of gradually-typed languages, there is a value form for values that have been boxed and injected to $\star$, which is $v\langle I!\rangle$. The I plays the role of a tag that records the type of v. The values at all other types are unboxed, as they would be in a statically-typed language.

With monotonic references, only one kind of value has reference type: normal addresses. When a cast is applied to a reference, instead of wrapping the reference with a cast, the systems casts the underlying value on the heap. To make sure that the new type of the value is consistent with all the outstanding references, it requires that a cast only make the type of the value more precise (Figure 2). Otherwise the cast results in a run-time error. Thus, it maintains the heap invariant that the type of each reference in the program is less or equally precise as the type of the value on the heap that it points to, as captured in the typing rule (WTRef).

One might wonder why the heap invariant uses the precision relation instead of subtyping. Could same efficiency goals be obtained using subtyping instead? Consider the following program in which a function of type $\star \to \text{int}$ is referenced from the static type $\text{int} \to \text{int}$. (Recall that $\star \to \text{int} <: \text{int} \to \text{int}$.)

$$\texttt{let}\ r_1 = \texttt{ref}\ (\lambda x : \star.\ x\ \texttt{as}\ \text{int})\ \texttt{in}$$
$$\texttt{let}\ r_2 = (r_1\ \texttt{as}\ \text{ref}\,(\text{int} \to \text{int}))\ \texttt{in}$$
$$!r_2\ 42$$

The dereference of r_2 should not require overhead, but there is a function of type $\star \to \text{int}$ that is to be applied to an integer, and the conversion from int to $\star$ requires boxing in this setting. Thus, the dereference of r_2 is not simply a load instruction, but it must handle the casting from $\star \to \text{int}$ to $\text{int} \to \text{int}$. In general, given a reference of type $\text{ref}\,T_2$, even when T_2 is a static type, there are many types T_1 such that $T_1 <: T_2$ and $T_1 \neq T_2$.

The syntax of the monotonic calculus differs from the guarded calculus in that there are two kinds of dereference and update expressions. Programmers need not worry about choosing which of the two dereference or update expressions to use because this choice is type-directed and therefore is handled during compilation from the source language to the coercion calculus. The forms $!e$ and $e_1{:=}e_2$ are reserved for situations in which the reference type is fully static (See the typing rules in Figure 8). In these situations the value in the heap must have the same type as the reference. Thus, if a reference has a fully static type, such as $\text{ref}\,\text{int}$, the corresponding value on the heap must be an actual integer (and not an injection to $\star$), so only one reduction rule is needed for dereferencing a fully-static reference (DerefM), and one rule is needed for updating a fully-static reference (UpdM).

For expressions of reference type that are not fully-static, we introduce the syntactic forms $!e@T$ and $e_1{:=}e_2@T$ for dereference and update, respectively. The type annotation T records the compile-time type of e, that is, e has type $\text{ref}\,T$. For example, T could be $\star$, $\star \times \star$, or $\star \times \text{int}$. Because the value on the heap might be more precise than T, a cast is needed to mediate between T and the run-time type of the heap cell.

The reduction rule (DynDerefM) casts from the addresses' run-time type, which is stored next to the heap cell, to the compile-time type T. I write $\mu(a)_{\text{rtti}}$ for the run-time type information for reference a and I write $\mu(a)_{\text{val}}$ for the value in the heap cell. The reduction rule (DynUpdM) casts the incoming

$$
\begin{array}{lrcl}
\text{Expressions} & e & ::= & .. \mid \texttt{ref}_T\, e \mid !e@T \mid e{:=}e@T \mid \texttt{error} \\
\text{Coercions} & c & ::= & \iota \mid I? \mid I! \mid c \to c \mid c \times c \mid c\,;c \mid \text{ref}\,T \\
\text{Values} & v & ::= & k \mid \lambda x.\, e \mid (v, v) \mid v\langle I! \rangle \mid a \\
\text{Casted Values} & cv & ::= & v \mid cv\langle c \rangle \mid (cv, cv) \\
\text{Heap} & \mu & ::= & \emptyset \mid \mu(a \mapsto v : T) \\
\text{Evolving Heap} & \nu & ::= & \emptyset \mid \nu(a \mapsto cv : T) \\
\text{Frames} & F & ::= & .. \mid !\Box@T \mid \Box{:=}e@T \mid v{:=}\Box@T
\end{array}
$$

Figure 7. Syntax for monotonic references without blame

value v from T to the address's run-time type, so the new content of the cell is $cv = v\langle T \Rightarrow \mu(a)_{\text{rtti}} \rangle$. This cv is not a value yet, so storing it in the heap is unusual. In earlier versions of the semantics I tried to reduce cv to a value before storing it in the heap, but there are complications that force this design, which is discussed later in this section . To summarize my treatment of dereference and update, I present efficient semantics for the fully-static dereference and update but have slightly increased the overhead for dynamic dereferences and updates. This is a price paid to have dynamic typing "pay its own way".

The crux of the monotonic semantics is in the reduction rules that apply a reference coercion to an address: (CastRef1), (CastRef2), and (CastRef3). In (CastRef1) there is an address that maps to cv of type T_1 and cv is casted so that it is no more dynamic than (i.e. at least as static as) both the target type T_2 and all of the existing references to the cell. To accomplish this, the rule takes the greatest lower bound $T_3 = T_1 \sqcap T_2$ (Figure 11) to be the new type of the cell, so the new contents is $cv' = cv\langle T_1 \Rightarrow T_3 \rangle$. There are two side conditions on (CastRef1): $T_1 \sqcap T_2$ must be defined and $T_3 \neq T_1$. If $T_1 \sqcap T_2$ is undefined, or equivalently, if $T_1 \not\sim T_2$, an error is instead signaled, as handled by (CastRef3). If $T_3 = T_1$, then there is no need to cast cv, which is handled by (CastRef2).

The last coercion reduction rule (PureCast) imports the reduction rules from the guarded semantics (Figure 6) though here I ignore blame, i.e., replace $\texttt{blame}\ \ell$ with $\texttt{error}$, $I_2?^\ell$ with $I_2?$, and $I_1 \Rightarrow^\ell I_2$ with $I_1 \Rightarrow I_2$.

Expression typing $\boxed{\Gamma; \Sigma \vdash e : T}$

$$\frac{\Gamma; \Sigma \vdash e : \text{ref}\, T \qquad \textit{static}\, T}{\Gamma; \Sigma \vdash\, !e : T} \qquad \frac{\Gamma; \Sigma \vdash e_1 : \text{ref}\, T \qquad \Gamma; \Sigma \vdash e_2 : T \qquad \textit{static}\, T}{\Gamma; \Sigma \vdash e_1 := e_2 : \text{ref}\, T} \qquad \frac{\Gamma; \Sigma \vdash e : \text{ref}\, T}{\Gamma; \Sigma \vdash\, !e@T : T}$$

(WTRef)
$$\frac{\Gamma; \Sigma \vdash e_1 : \text{ref}\, T \quad \Gamma; \Sigma \vdash e_2 : T}{\Gamma; \Sigma \vdash e_1 := e_2@T : \text{ref}\, T} \quad \cdots \quad \frac{\Sigma(a) \sqsubseteq T_2}{\Gamma; \Sigma \vdash a : T_2}$$

Figure 8. Type system for monotonic references without blame

$$\boxed{e, \nu \longrightarrow_{cr} e, \nu}$$

(PureCast)
$$\frac{e \longrightarrow_c e'}{e, \nu \longrightarrow_{cr} e', \nu}$$

(CastRef1)
$$\frac{\nu(a) = cv : T_1 \qquad T_3 = T_1 \sqcap T_2 \qquad T_3 \neq T_1 \qquad cv' = cv\langle T_1 \Rightarrow T_3 \rangle}{a\langle \text{ref}\, T_2 \rangle, \nu \longrightarrow_{cr} a, \nu(a \mapsto cv' : T_3)}$$

(CastRef2)
$$\frac{\nu(a) = cv : T_1 \qquad T_1 = T_1 \sqcap T_2}{a\langle \text{ref}\, T_2 \rangle, \nu \longrightarrow_{cr} a, \nu}$$

(CastRef3)
$$\frac{\nu(a) = cv : T_1 \qquad T_1 \not\sim T_2}{a\langle \text{ref}\, T_2 \rangle, \nu \longrightarrow_{cr} \texttt{error}, \nu}$$

Figure 9. Cast reduction rules for monotonic references without blame

The meet function defined in Figure 11 computes the greatest lower bound with respect to the precision relation.

To motivate my organization of the heap, I present two examples that demonstrate why run-time type information and casted values, not just values, are stored on the heap.

Cycles and termination. The first complication is that there can be cycles in the heap and we need to make sure that when a cast is applied to an address in a cycle, the cast terminates. Consider the

Program reduction rules $\boxed{e, \mu \longrightarrow_e e, \nu}$

$$e, \mu \longrightarrow_e e', \mu \qquad \text{if } e \longrightarrow e'$$

$$\texttt{ref}_T\, v, \mu \longrightarrow_e a, \mu(a \mapsto v : T) \qquad \text{if } a \notin \mathit{dom}(\mu)$$

(DerefM) $$!a, \mu \longrightarrow_e \mu(a)_{\text{val}}, \mu$$

(DynDerefM) $$!a@T, \mu \longrightarrow_e \mu(a)_{\text{val}}\langle \mu(a)_{\text{rtti}} \Rightarrow T\rangle, \mu$$

(UpdM) $$a{:=}v, \mu \longrightarrow_e a, \mu(a \mapsto v : \mu(a)_{\text{rtti}})$$

(DynUpdM) $$a{:=}v@T, \mu \longrightarrow_e a, \mu(a \mapsto cv : \mu(a)_{\text{rtti}})$$

$$\text{where } cv = v\langle T \Rightarrow \mu(a)_{\text{rtti}}\rangle$$

$$\frac{e, \nu \longrightarrow_X e', \nu' \qquad X \in \{cr, e\}}{F[e], \nu \longrightarrow_X F[e'], \nu'} \qquad \frac{X \in \{cr, e\}}{F[\texttt{error}], \nu \longrightarrow_X \texttt{error}, \nu}$$

State reduction rules $\boxed{e, \nu \longrightarrow e, \nu}$

(HCast) $$\frac{e, \mu \longrightarrow_X e', \nu \quad X \in \{cr, e\}}{e, \mu \longrightarrow e', \nu} \qquad \frac{\nu(a) = cv : T \qquad cv, \nu \longrightarrow_{cr} cv', \nu' \qquad \nu'(a)_{\text{rtti}} = T}{e, \nu \longrightarrow e, \nu'(a \mapsto cv' : T)}$$

(HDrop)

$$\frac{\nu(a) = cv : T \qquad cv, \nu \longrightarrow_{cr} \texttt{error}, \nu'}{e, \nu \longrightarrow \texttt{error}, \nu'} \qquad \frac{\nu(a) = cv : T \qquad cv, \nu \longrightarrow_{cr} cv', \nu' \qquad \nu'(a)_{\text{rtti}} \neq T}{e, \nu \longrightarrow e, \nu'}$$

Figure 10. Monotonic references without blame

$\boxed{T \sqcap T = T}$

$$\star \sqcap T = T \qquad (T_1 \times T_2) \sqcap (T_3 \times T_4) = (T_1 \sqcap T_3) \times (T_2 \sqcap T_4)$$

$$T \sqcap \star = T \qquad (T_1 \to T_2) \sqcap (T_3 \to T_4) = (T_1 \sqcap T_3) \to (T_2 \sqcap T_4)$$

$$B \sqcap B = B \qquad \text{ref}\, T_1 \sqcap \text{ref}\, T_2 = \text{ref}\, (T_1 \sqcap T_2)$$

Figure 11. The meet function (greatest lower bound)

following example in which a pair is created whose second element is a reference back to itself.

$$\begin{aligned}
&\texttt{let}\ r_1 = \texttt{ref}\ (42, 0\ \texttt{as}\ \star)\ \texttt{in}\\
&r_1 {:=} (42, r_1\ \texttt{as}\ \star);\\
&\texttt{let}\ r_2 = r_1\ \texttt{as}\ \text{ref}\,(\text{int} \times \text{ref}\,\star)\ \texttt{in}\\
&\texttt{fst}\ !r_2
\end{aligned}$$

Once the cycle is established, r_1 is casted from type $\text{ref}\,(\text{int} \times \star)$ to $\text{ref}\,(\text{int} \times \text{ref}\,\star)$. The presence of the nested ref $\star$ in the target type means that the cast on r_1 will trigger another cast on r_1. The correct result of this program is 42 but a naïve dynamic semantics would diverge. My semantics avoids divergence by checking whether the new run-time type is equal to the old run-time type; in such cases the heap cell is left unchanged (see rule (CastRef2)).

Casted values in the heap. Consider the following example in which a triple of type $\star \times \star \times \star$ is created whose third element is a reference back to itself.

$$\begin{aligned}
&\texttt{let}\ r_0 = \texttt{ref}\ (42\ \texttt{as}\ \star, 7\ \texttt{as}\ \star, 0\ \texttt{as}\ \star)\ \texttt{in}\\
&r_0 {:=} (42\ \texttt{as}\ \star, 7\ \texttt{as}\ \star, r_0\ \texttt{as}\ \star);\\
&\texttt{let}\ r_1 = r_0\ \texttt{as}\ \text{ref}\,(\text{int} \times \star \times \text{ref}\,(\text{int} \times \text{int} \times \star))\ \texttt{in}\\
&\texttt{fst}\ (\texttt{fst}\ !r_1)
\end{aligned}$$

Suppose a_0 is the address created in the allocation on the first line. On line three a_0 is casted in such a way that it triggers two casts on a_0. Considering the action of these casts on just the first two elements of the triple, we have:

$$\star \times \star \Rightarrow \text{int} \times \star \Rightarrow \text{int} \times \text{int}$$

The second cast occurs while the first is still in progress. Now, suppose we delayed updating the heap cell until we finished reducing to a value. At the moment when we apply the second cast, we would still have the original value, of type $\star \times \star$, in the heap. This is problematic because our next step would be to apply a cast from $\text{int} \times \star \Rightarrow \text{int} \times \text{int}$ to this value, but the value's type and the source type of the cast don't match! In fact, in this example the result would be incorrect; we would get $42\langle\text{int}!\rangle$ instead of 42.

There are several solutions to this problem, and they all require storing more information on the heap or as a separate map. Here we take the most straightforward approach of immediately updating the heap with casted values, that is, with values that are in the process of being cast.

I walk through the execution of the above example, explaining the rules for reducing casted values in the heap and showing snapshots of the heap. I use the following abbreviations.

$$T_0 = \star \times \star \times \star$$

$$T_1 = \text{int} \times \star \times \text{ref}\, T_2$$

$$T_2 = \text{int} \times \text{int} \times \star$$

$$c = \text{int}? \times \iota \times (\text{ref}\, T_2)?$$

The first line of the program allocates a triple.

$$a_0 \mapsto (42\langle\text{int!}\rangle, 7\langle\text{int!}\rangle, 0\langle\text{int!}\rangle) : T_0$$

The second line sets the third element to be a reference to itself.

$$a_0 \mapsto (42\langle\text{int!}\rangle, 7\langle\text{int!}\rangle, a_0\langle(\text{ref}\, T_0)!\rangle) : T_0$$

The third line casts the reference to ref T_1 via (CastRef1).

$$a_0 \mapsto (42\langle\text{int!}\rangle, 7\langle\text{int!}\rangle, a_0\langle(\text{ref}\, T_0)!\rangle)\langle c\rangle : T_1$$

We have a casted value in the heap that needs to be reduced. We apply (HCast) and (PureCast) to get

$$a_0 \mapsto (42, 7\langle\text{int!}\rangle, a_0\langle\text{ref}\, T_2\rangle) : T_1$$

We cast address a_0 again, this time to $T_1 \sqcap T_2$, via rule (HDrop) and (CastRef1).

$$a_0 \mapsto (42, 7\langle\text{int!}\rangle, a_0\langle\text{ref}\, T_2\rangle)\langle\iota \times \text{int}? \times \text{ref}\, T_2\rangle : \text{int} \times \text{int} \times \text{ref}\, T_2$$

A few reductions via (HCast) and (PureCast) give us

$$a_0 \mapsto (42, 7, a_0\langle\text{ref}\, T_2\rangle) : \text{int} \times \text{int} \times \text{ref}\, T_2$$

The final cast applied to a_0 is a no-op because the run-time type is already more precise than T_2. So we reduce via (HCast) and (CastRef2) to:

$$a_0 \mapsto (42, 7, a_0) : \text{int} \times \text{int} \times \text{ref}\, T_2$$

Even though casted values are allowed on the heap, all such casts must be normalized before returning to the execution of the program. Normal heaps of values, μ, are distinguished from and evolving heaps, ν, that may contain both values and casted values. Normal heaps are a subset of the evolving heaps.

Encoding permissive references. The monotonic discipline and its run-time invariant-enforcement seems to restrict how developers can formulate their programs. It is natural to ask whether monotonic references are compatible with the flexibility that is expected in dynamic languages. In this section I show that the monotonic discipline admits permissive references through a syntactic discipline that can be conveniently provided to programmers.

Consider the following program that uses an allocated reference cell at two incompatible types, int and bool.

$$
\begin{array}{l}
\texttt{let}\ x = \texttt{ref}\ (4\ \texttt{as}\ \star)\ \texttt{in} \\
\texttt{let}\ y = (x\ \texttt{as}\ \text{ref int})\ \texttt{in} \\
\texttt{let}\ z = (x\ \texttt{as}\ \text{ref bool})\ \texttt{in} \\
!y; \\
z{:=}\text{true}; \\
!z
\end{array}
$$

Under the guarded reference semantics, this program runs without incident, but under monotonic references it fails in the cast to ref bool. This flexibility may be regained under monotonic references via a disciplined use of $\star$ typed reference cells. Consider the following rewrite of this program:

$$
\begin{array}{ll}
\texttt{let}\ x = \texttt{ref}\ (4\ \texttt{as}\ \star)\ \texttt{in} & \\
\texttt{let}\ y = x\ \texttt{in} & //\ \text{treat } y \text{ like ref int} \\
\texttt{let}\ z = x\ \texttt{in} & //\ \text{treat } z \text{ like ref bool} \\
(!y)\ \texttt{as}\ \text{int}; & \\
(z{:=}(\text{true})\ \texttt{as}\ \star)\ \texttt{as}\ \text{bool}; & \\
(!z)\ \texttt{as}\ \text{bool} &
\end{array}
$$

In this encoding, all references have type ref $\star$, and typing is enforced only at dereferences and updates, using ascriptions. This program runs successfully under the monotonic semantics, but it would be tedious and error prone to insert these ascriptions by hand.

Luckily there is no need: this permissive reference discipline can be codified by introducing a surface language that makes this convenient. I extend the expressions with *permissive references* $\widetilde{\texttt{ref}}\ e$, and the types with a corresponding type $\widetilde{\text{ref}}\,T$. Consistency is extended so that permissive references have the same consistency properties as monotonic references, but permissive references are not consistent with monotonic references.

Finally I introduce a type-directed transformation $\Gamma| - e : T \rightsquigarrow e$ that translates permissive references to monotonic references. The interesting cases are presented below.

$$\frac{x : \widetilde{\text{ref}}\,T \in \Gamma}{\Gamma| - x : \widetilde{\text{ref}}\,T \rightsquigarrow x} \qquad \frac{\Gamma| - e : T \rightsquigarrow e'}{\Gamma| - \widetilde{\texttt{ref}}\ e : \widetilde{\text{ref}}\,T \rightsquigarrow \texttt{ref}\ (e'\ \texttt{as}\ \star)}$$

$$\frac{\Gamma| - e : \widetilde{\text{ref}}\,T \rightsquigarrow e'}{\Gamma| - !e : T \rightsquigarrow (!e')\ \texttt{as}\ T} \qquad \frac{\Gamma| - e_1 : \widetilde{\text{ref}}\,T_1 \rightsquigarrow e_1' \qquad \Gamma| - e_2 : T_2 \rightsquigarrow e_2' \qquad T_1 \sim T_2}{\Gamma| - e_1 := e_2 : T_1 \rightsquigarrow (e_1' := (e_2'\ \texttt{as}\ \star))\ \texttt{as}\ T_1}$$

Note that the static semantics for permissive references enforces type consistency at assignments, even though the assigned value is ultimately cast to $\star$. Furthermore, reference values translate to themselves, so object identity is preserved. However cast overhead is introduced at each dereference and update, so permissive references pay their own way with respect to performance.

If we revisit the initial example in this section and replace `ref` with $\widetilde{\texttt{ref}}$ and ref with $\widetilde{\text{ref}}$, then this judgment translates the first program above into the second.

Proposition 3.1 (Translation). *If* $\Gamma| - e : T \rightsquigarrow e'$ *then* $|\Gamma|| - e' : |T|$*, Where* $|\cdot|$ *is the compatible extension of the equation* $|\widetilde{\text{ref}}\,T| = \text{ref}\,\star$.

This syntactic extension gives programmers access to both permissive references and monotonic references as desired.

Permissive references are a useful abstraction for the programmer and provide strong guarantees. However, such guarantees are provided only as long as permissive references do not flow into monotonic references. Consider the program above (with permissive references) where the following code comes after the `let` statements.

$$\boxed{(T \Rightarrow T) = c}$$

$$\begin{array}{lll} (B \Rightarrow B) & = \iota & (I \Rightarrow \star) = I! \\ (\star \Rightarrow \star) & = \iota & (\star \Rightarrow I) = I? \end{array}$$

$$(T_1 \to T_2) \Rightarrow (T_1' \to T_2') = (T_1' \Rightarrow T_1) \to (T_2 \Rightarrow T_2')$$

$$(T_1 \times T_2) \Rightarrow (T_1' \times T_2') = (T_1 \Rightarrow T_1') \times (T_2 \Rightarrow T_2')$$

$$\mathsf{ref}\, T \Rightarrow \mathsf{ref}\, T' = \mathsf{ref}\, T'$$

Figure 12. Compile casts to monotonic coercions (without blame)

$$\begin{array}{l} \texttt{let}\ w_1 = (x\ \texttt{as}\ \star)\ \texttt{in} \\ \texttt{let}\ w_2 = (w_1\ \texttt{as}\ \mathsf{ref}\,\mathsf{bool})\ \texttt{in} \\ w_2 \texttt{:=} \texttt{true}; \end{array}$$

The program finds itself in the same situation as the original program that the monotonic semantics could not run without error. This example shows an important syntactic discipline for programmers that want to employ the monotonic paradigm for gradual references: *permissive references should not flow into monotonic references*.

4. Type Safety for Monotonic References

I present the high-points of the type safety proof here. The full proof is mechanized in Isabelle 2013 and available on arxiv [78]. The semantics in the mechanized version differs from the semantics presented here in that it uses an abstract machine instead of a reduction semantics, as the mechanized proof was found to be easier to carry out on an abstract machine while the reduction semantics is more approachable.

I begin by lifting the precision relation to heap typings.

Definition 3.1 (Precision relation on heap typings). $\Sigma' \sqsubseteq \Sigma$ *iff* $\mathit{dom}(\Sigma') = \mathit{dom}(\Sigma)$ *and* $\Sigma(a) = T$ *implies* $\Sigma'(a) = T'$ *where* $T' \sqsubseteq T$.

My first lemma below is important: expression typing is preserved when moving to a more precise heap typing.

Lemma 3.1 (Strengthening wrt. the heap typing). *If* $\Gamma; \Sigma \vdash e : T$ *and* $\Sigma' \sqsubseteq \Sigma$*, then* $\Gamma; \Sigma' \vdash e : T$.

Proof sketch. The interesting case is for addresses. We have

$$\frac{\Sigma(a) \sqsubseteq T}{\Gamma; \Sigma \vdash a : T}$$

From $\Sigma' \sqsubseteq \Sigma$ and transitivity of $\sqsubseteq$, we have $\Sigma'(a) \sqsubseteq T$. Therefore $\Gamma; \Sigma' \vdash a : T$. □

The definition of well-typed heaps is standard.

Definition 3.2 (Well-typed heaps). *A heap* ν *is well-typed with respect to heap typing* Σ*, written* $\Sigma \vdash \nu$*, iff* $\forall a\, T.\ \Sigma(a) = T$ *implies* $\nu(a) = cv : T$ *and* $\emptyset; \Sigma \vdash cv : T$ *for some* cv.

From the strengthening lemma, we have the following corollary.

Corollary 3.1 (Monotonic heap update). *If* $\Sigma \vdash \nu$ *and* $\Sigma(a) = T$ *and* $T' \sqsubseteq T$ *and* $\emptyset; \Sigma \vdash cv : T'$*, then* $\Sigma(a \mapsto T') \vdash \nu(a \mapsto cv : T')$.

sketch. Let $\Sigma' = \Sigma(a \mapsto T')$. From $T' \sqsubseteq T$ we have $\Sigma' \sqsubseteq \Sigma$, so by Lemma 3.1 we have $\emptyset; \Sigma' \vdash cv : T'$ and $\Sigma' \vdash \nu$. Thus, $\Sigma(a \mapsto T') \vdash \nu(a \mapsto cv : T')$. □

Lemma 3.2 (Progress and Preservation). *Suppose* $\emptyset; \Sigma \vdash e : T$ *and* $\Sigma \vdash \nu$*. Exactly one of the following holds:*

(1) (a) e *is a value, or*
    (b) $e =$ `error`*, or*
    (c) $e, \nu \longrightarrow e', \nu'$ *for some* e' *and* ν'.
(2) *for all* e', ν'*, if* $e, \nu \longrightarrow e', \nu'$ *then* $\emptyset; \Sigma' \vdash e' : T$ *and* $\Sigma' \vdash \nu'$ *and* $\Sigma' \sqsubseteq \Sigma$ *for some* Σ'.

Theorem 3.1 (Type Safety). *Suppose* $\emptyset; \Sigma \vdash e : T$ *and* $\Sigma \vdash \nu$*. Exactly one of the following holds:*

(1) $e, \nu \longrightarrow^* v, \nu'$ *and* $\emptyset; \Sigma' \vdash v : T$ *for some* Σ'*, or*

(2) $e, \nu \longrightarrow^* \mathit{error}, \nu'$, *or*

(3) e *diverges.*

Proof. If e diverges we immediately conclude the proof. Otherwise, suppose e does not diverge. Then $e, \nu \longrightarrow^* e', \nu'$ and e' cannot reduce. We proceed by induction on the length $e, \nu \longrightarrow^* e', \nu'$, and use Lemma 3.2 to conclude. □

5. Monotonic References with Blame

I turn to the challenge of designing blame tracking for monotonic references, presenting several examples that motivate and provide intuitions for the design. The later part of this section presents the dynamic semantics of monotonic references with blame tracking.

Consider the following example in which I allocate a reference of dynamic type and then, separately, cast from ref $\star$ to ref int and to ref bool.

$$\begin{array}{l}\texttt{let}\ r_0 = \texttt{ref}\ (42\ \texttt{as}^{\ell_1}\ \star)\ \texttt{in}\\ \texttt{let}\ r_1 = r_0\ \texttt{as}^{\ell_2}\ \text{ref int}\ \texttt{in}\\ \texttt{let}\ r_2 = r_0\ \texttt{as}^{\ell_3}\ \text{ref bool}\ \texttt{in}\\ !r_2\end{array}$$

With monotonic references, the cast at ℓ_3 triggers an error, because int and bool are inconsistent. But what blame labels should the error message include? Is it only the fault of ℓ_3? Not really; because ℓ_3 would not cause an error if it were not for the cast at ℓ_2. The casts at ℓ_2 and ℓ_3 disagree with each other regarding the type of the heap cell, so both are blamed. The result of this program is $\texttt{blame}\ \{\ell_2, \ell_3\}$.

Next consider an example in which a reference is allocated at type ref int, cast it to ref $\star$, and then has a Boolean written to it.

$$\begin{array}{l}\texttt{let}\ r_0 = \texttt{ref}\ 42\ \texttt{in}\\ \texttt{let}\ r_1 = r_0\ \texttt{as}^{\ell_1}\ \text{ref}\ \star\ \texttt{in}\\ r_1 :=^{\ell_3} (\texttt{true}\ \texttt{as}^{\ell_2}\ \star)\end{array}$$

The update on the third line triggers an error, and there are three possible locations to blame: ℓ_1, ℓ_2, and ℓ_3. The cast at ℓ_2 is from bool to $\star$, which is harmless. There is no cast at ℓ_3, just a write of a value of type $\star$ to a reference of type ref $\star$. The real culprit here is ℓ_1, which casts from ref int to ref $\star$,

$$B <: B \qquad T <: \star \qquad \mathsf{ref}\, T <: \mathsf{ref}\, T$$

$$\frac{T_1' <: T_1 \qquad T_2 <: T_2'}{T_1 \to T_2 <: T_1' \to T_2'} \qquad \frac{T_1 <: T_1' \qquad T_2 <: T_2'}{T_1 \times T_2 <: T_1' \times T_2'}$$

Figure 13. Subtyping relation

thereby opening up the potential for the later cast error. Naïvely, this looks like an upcast, but a proper treatment of subtyping for references makes references invariant. So we have $\mathsf{ref}\,\mathsf{int} \not<: \mathsf{ref}\,\star$ and the result of this program is $\texttt{blame}\,\{\ell_1\}$. Figure 13 presents the subtyping relation.

Consider a pair of examples below that differ only on the fourth line. A reference to a pair is allocated at type $\mathsf{ref}\,(\star \times \star)$ then cast to $\mathsf{ref}\,(\mathsf{int} \times \star)$ and to $\mathsf{ref}\,(\star \times \mathsf{int})$. In the first example, an update occurs through the original reference, writing a Boolean and integer, whereas in the second example an integer and a Boolean are written. Here is the first example:

$$\begin{array}{l}
\texttt{let}\, r_0 = \texttt{ref}\, (1\ \texttt{as}^{\ell_1}\ \star, 2\ \texttt{as}^{\ell_2}\ \star)\ \texttt{in} \\
\texttt{let}\, r_1 = r_0\ \texttt{as}^{\ell_3}\ \mathsf{ref}\,(\mathsf{int} \times \star)\ \texttt{in} \\
\texttt{let}\, r_2 = r_0\ \texttt{as}^{\ell_4}\ \mathsf{ref}\,(\star \times \mathsf{int})\ \texttt{in} \\
r_0 := (\texttt{true}\ \texttt{as}^{\ell_5}\ \star, 2\ \texttt{as}^{\ell_6}\ \star); \\
\texttt{fst}\, !r_0
\end{array}$$

and here is the second example, just showing the fourth line:

$$\begin{array}{l}
\ldots \\
r_0 := (1\ \texttt{as}^{\ell_7}\ \star, \texttt{true}\ \texttt{as}^{\ell_8}\ \star); \\
\ldots
\end{array}$$

The first example should produce $\texttt{blame}\,\{\ell_3\}$ while the second example should produce $\texttt{blame}\,\{\ell_4\}$, but the challenge is how to associate multiple blame labels with the same heap cell?

I take inspiration from Siek and Wadler [79] and use *labeled types* for this treatment of run-time type information. With a labeled type, each type constructor within the type can be labeled with a blame label. Figures 14 and 15 give the syntax of labeled types and operations on them, which are explained

later in this section. In the above examples, the run-time type information for the heap cell evolves as follows:

$$(\star \times^{\emptyset} \star) \Rightarrow (\text{int}^{\ell_3} \times^{\emptyset} \star) \Rightarrow (\text{int}^{\ell_3} \times^{\emptyset} \text{int}^{\ell_4})$$

In the first example, when `true` is written into the first element of the pair, the cast to int fails and blames ℓ_3, as desired. In the second example, when `true` is written into the second element, the cast to int fails and blames ℓ_4, as desired.

The next example brings up a somewhat ambiguous situation. A reference is allocated at type ref $\star$, casted to ref int twice, then has a Boolean written to it.

$$\begin{aligned}
&\texttt{let } r_0 = \texttt{ref } (42 \texttt{ as}^{\ell_1} \star) \texttt{ in} \\
&\texttt{let } r_1 = r_0 \texttt{ as}^{\ell_2} \text{ ref int } \texttt{in} \\
&\texttt{let } r_2 = r_0 \texttt{ as}^{\ell_3} \text{ ref int } \texttt{in} \\
&r_0 := (\texttt{true as}^{\ell_4} \star)
\end{aligned}$$

Which should be blamed: ℓ_2 or ℓ_3? In some sense, they are both just as guilty and the ideal would be to blame them both. On the other hand, maintaining potentially large sets of blame labels would induce some space overhead. This design instead blames the first cast with respect to execution order, in this case ℓ_2.

For the final example, the above example is adapted to have a function in the heap cell so that we can consider the behavior to the left of the arrow.

$$\begin{aligned}
&\texttt{let } r_0 = \texttt{ref } (\lambda x{:}\star\,.\,\texttt{true}) \texttt{ in} \\
&\texttt{let } r_1 = r_0 \texttt{ as}^{\ell_1} \text{ ref } (\text{int} \to \text{bool}) \texttt{ in} \\
&\texttt{let } r_2 = r_0 \texttt{ as}^{\ell_2} \text{ ref } (\text{int} \to \text{bool}) \texttt{ in} \\
&r_0 := \lambda x{:}\text{int}.\, \texttt{zero?}(x); \\
&!r_0 \; (\texttt{true as}^{\ell_3} \star)
\end{aligned}$$

The run-time type information for the heap cell evolves in the following way:

$$(\star \to^{\emptyset} \text{bool}^{\emptyset}) \Rightarrow (\text{int}^{\ell_1} \to^{\emptyset} \text{bool}^{\emptyset}) \Rightarrow (\text{int}^{\ell_1} \to^{\emptyset} \text{bool}^{\emptyset})$$

The function application on the last line of the example triggers a cast error, with the blame going to ℓ_1, again because the first cast with respect to execution order should be blamed. However, to obtain this

$$
\begin{array}{lllll}
\text{Optional labels} & p, q & ::= & \emptyset \mid \{\ell\} \\
\text{Label sets} & L & ::= & \emptyset \mid \{\ell\} \mid \{\ell_1, \ell_2\} \\
\text{Labeled types} & P, Q & ::= & B^p \mid P \to^p P \mid P \times^p P \mid \mathsf{ref}^p P \mid \star
\end{array}
$$

Erase labels $\boxed{|P| = T}$

$$|B^p| = B \quad |P \to^p Q| = |P| \to |Q| \quad |P \times^p Q| = |P| \times |Q| \quad |\mathsf{ref}^p P| = \mathsf{ref}\,|P| \quad |\star| = \star$$

Top label $\boxed{\mathit{lab}(P) = L}$

$$\mathit{lab}(B^p) = p \quad \mathit{lab}(P \to^p Q) = p \quad \mathit{lab}(P \times^p Q) = p \quad \mathit{lab}(\mathsf{ref}^p P) = p \quad \mathit{lab}(\star) = \emptyset$$

Merge optional labels $\boxed{p \vartriangle p = p}$

$$\{\ell\} \vartriangle q = \{\ell\} \qquad \emptyset \vartriangle q = q$$

Figure 14. Syntax and utilities for labeled types

semantics some care must be taken. On the second cast, the labeled type for the second cast merges with the current run-time type information:

$$(\mathsf{int}^{\ell_1} \to^{\emptyset} \mathsf{bool}^{\emptyset}) \vartriangle (\mathsf{int}^{\ell_2} \to^{\emptyset} \mathsf{bool}^{\emptyset})$$

If this operation were to use the composition function from Siek and Wadler [79], the result would be $\mathsf{int}^{\ell_2} \to^{\emptyset} \mathsf{bool}^{\emptyset}$ because that composition function is contravariant for function parameters. Here we instead want to be covariant on function parameters, so the result is $\mathsf{int}^{\ell_1} \to^{\emptyset} \mathsf{bool}^{\emptyset}$. I define a new function for merging labeled types, $\vartriangle$, in Figure 15.

5.1. Semantics of monotonic references with blame. Armed with the intuitions from the above examples, I discuss the semantics of monotonic references with blame, defined in Figures 17, 18, and 19. The semantics is largely similar to the semantics without blame except that the run-time type information is represented as labeled types and I replace the functions, such as meet ($\sqcap$) that operate on types, with functions such as merge ($\vartriangle$) that operate on labeled types.

Merge labeled types $\boxed{P \vartriangle P = P \text{ or } \bot^L}$

$$B^p \vartriangle B^q = B^{p \vartriangle q}$$

$$P \vartriangle \star = P \qquad \star \vartriangle Q = Q$$

$$(P \to^p P') \vartriangle (Q \to^q Q') = (P \vartriangle Q) \hat{\to}^{p \vartriangle q} (P' \vartriangle Q')$$

$$(P \times^p P') \vartriangle (Q \times^q Q') = (P \vartriangle Q) \hat{\times}^{p \vartriangle q} (P' \vartriangle Q')$$

$$\text{ref}^p P \vartriangle \text{ref}^q Q = \hat{\text{ref}}^{p \vartriangle q} (P \vartriangle Q)$$

$$P \vartriangle Q = \bot^{lab(P) \cup lab(Q)} \qquad \text{otherwise}$$

Bidirectional cast between labeled types $\boxed{P \Leftrightarrow P = P \text{ or } \bot^L}$

$$B^p \Leftrightarrow B^q = B^{\emptyset}$$

$$P \Leftrightarrow \star = P \qquad \star \Leftrightarrow Q = Q$$

$$(P \to^p P') \Leftrightarrow (Q \to^q Q') = (P \Leftrightarrow Q) \hat{\to}^{\emptyset} (P' \Leftrightarrow Q')$$

$$(P \times^p P') \Leftrightarrow (Q \times^q Q') = (P \Leftrightarrow Q) \hat{\times}^{\emptyset} (P' \Leftrightarrow Q')$$

$$\text{ref}^p P \Leftrightarrow \text{ref}^q Q = \hat{\text{ref}}^{\emptyset} (P \Leftrightarrow Q)$$

$$P \Leftrightarrow Q = \bot^{lab(P) \cup lab(Q)} \qquad \text{otherwise}$$

Cast between labeled types $\boxed{P \Rightarrow P = c \text{ or } \bot^L}$

$$B^p \Rightarrow B^q = \iota \qquad \star \Rightarrow \star = \iota$$

$$P \Rightarrow \star = P! \qquad \star \Rightarrow Q = Q?$$

$$(P \to^p P') \Rightarrow (Q \to^q Q') = (Q \Rightarrow P) \hat{\to}^{\emptyset} (P' \Rightarrow Q')$$

$$(P \times^p P') \Rightarrow (Q \times^q Q') = (P \Rightarrow Q) \hat{\times}^{\emptyset} (P' \Rightarrow Q')$$

$$\text{ref}^p P \Rightarrow \text{ref}^q Q = \hat{\text{ref}}^{\emptyset} (P \Leftrightarrow Q)$$

$$P \Rightarrow Q = \bot^{lab(P) \cup lab(Q)} \quad \text{otherwise}$$

Figure 15. Labeled types and their operations

$\boxed{(T \Rightarrow^\ell T) = c}$

$$\begin{array}{llll} (B \Rightarrow^\ell B) & = \iota & (T \Rightarrow^\ell \star) & = T^{\emptyset}! \\ (\star \Rightarrow^\ell \star) & = \iota & (\star \Rightarrow^\ell T) & = T^{\ell}? \end{array}$$

$$(T_1 \to T_2) \Rightarrow^\ell (T_1' \to T_2') = (T_1' \Rightarrow^\ell T_1) \to (T_2 \Rightarrow^\ell T_2')$$

$$(T_1 \times T_2) \Rightarrow^\ell (T_1' \times T_2') = (T_1 \Rightarrow^\ell T_1') \times (T_2 \Rightarrow^\ell T_2')$$

$$\mathsf{ref}\, T_1 \Rightarrow^\ell \mathsf{ref}\, T_2 = \mathsf{ref}\, (T_1^\ell \Leftrightarrow T_2^\ell)$$

Add labels to a type $\boxed{T^\ell = P}$

$$B^\ell = B^\ell \quad (T_1 \to T_2)^\ell = T_1^\ell \to^\ell T_2^\ell \quad (T_1 \times T_2)^\ell = T_1^\ell \times^\ell T_2^\ell$$

$$(\mathsf{ref}\, T)^\ell = \mathsf{ref}^{\,\ell} T^\ell \quad \star^\ell = \star$$

Figure 16. Compile casts to monotonic coercions (with blame)

Proposition 3.2 (Meet is the erasure of merge).
If $|P_1| \sim |P_2|$*, then* $|P_1 \vartriangle P_2| = |P_1| \sqcap |P_2|$.
If $|P_1| \not\sim |P_2|$*, then* $P_1 \vartriangle P_2 = \bot^L$ *for some* L.

As discussed with the example above, the definition of $P_1 \vartriangle P_2$ takes into account that P_1 is temporally prior to P_2 and should therefore take precedence with respect to blame responsibility. I use the auxiliary function $p \vartriangle q$ to choose between two optional labels, returning the first if it is present and the second otherwise.

When a reference is casted via rule (CastR1B), the heap cell must be updated from labeled type P_1 to P_3. This is accomplished with a new operator $P_1 \Rightarrow P_3$ that produces a coercion. The most interesting line of its definition is for reference types. There a different operator is used, $P \Leftrightarrow Q$, that produces a labeled type and captures the bidirectional read/write nature of mutable references.

The definitions of $\triangle$, $\Rightarrow$, and $\Leftrightarrow$ need to percolate errors, written as $\bot^L$ where L is a set of blame labels. I use "smart" constructors $\hat{\rightarrow}$, $\hat{\times}$, and $\hat{\text{ref}}$ that return $\bot^L$ if either argument is $\bot^L$ (with precedent to the left if both arguments are errors), but otherwise act like the underlying constructor.

In the rule for allocation, the RTTI is initialized to $T^\emptyset$. (Figure 16 defines converting a type to a labeled type.) In the rule for a dynamic dereference, (DynDrfMB), the reference's run-time labeled type is casted to T by promoting T to the labeled type $T^\emptyset$ and then applying the $\Rightarrow$ function to cast between labeled types, so we have $\mu(a)_{\text{rtti}} \Rightarrow T^\emptyset$. Suppose that $\mu(a)_{\text{rtti}}$ is ref int$^\ell$ and T is ref $\star$. Then the coercion we apply during the dereference is int$^\ell$!; so injection coercions contain labeled types. The rule for dynamic update, (DynUpdMB), is dual, performing the cast $T^\emptyset \Rightarrow \mu(a)_{\text{rtti}}$.

Because injection and projection coercions contain labeled types, the (Collapse) rule becomes

$$v\langle P_1!\rangle\langle P_2?\rangle \longrightarrow_c v\langle P_1 \Rightarrow P_2\rangle \qquad \text{if } |P_1| \sim |P_2|$$

Similar changes are needed to the (Conflict) rule.

Figure 16 defines the compilation of casts to monotonic coercions. Compared to the compilation without blame (Figure 12), there are three differences. The first two concern injection and projection coercions: instead of only having a blame label on projections there are labeled types inside both injections and projections, as noted above. In the compilation of a cast labeled ℓ, the rule generates a labeled type for the injection from T by adding the empty label to T, and for the projection to T by adding ℓ to T. The third difference is in the formation of the reference coercion. Instead of simply taking the target type, the bidirectional operator $\Leftrightarrow$ is used. Recall the second example of this section in which execution blamed the cast from ref int to ref $\star$. By using $\Leftrightarrow$, the resulting coercion is ref int$^{\ell_1}$ instead of ref $\star$.

6. The Blame-Subtyping Theorem

The blame-subtyping theorem pin-points the source of cast errors in gradually-typed programs. The blame-subtyping theorem states that if a program results in a cast error, `blame` L, then the blame labels in L identify the location of implicit casts that did not respect subtyping. That is, the blame labels that occur in a safe implicit cast, $T_1 \Rightarrow T_2$ where $T_1 <: T_2$, can never be blamed.

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & \cdots \mid \texttt{blame}\ L \\
\text{Coercions} & c & ::= & \iota \mid P? \mid P! \mid c \to c \mid c \times c \mid c\,;c \mid \mathsf{ref}\, P \\
\text{Values} & v & ::= & k \mid \lambda x.\, e \mid (v, v) \mid v\langle P! \rangle \mid a \\
\text{Casted Values} & cv & ::= & v \mid cv\langle c \rangle \mid (cv, cv) \\
\text{Heap} & \mu & ::= & \emptyset \mid \mu(a \mapsto v : P) \\
\text{Evolving Heap} & \nu & ::= & \emptyset \mid \nu(a \mapsto cv : P)
\end{array}
$$

Figure 17. Syntax for monotonic references with blame

Coercion typing $\boxed{c : T \Rightarrow T}$

$$P? : \star \Rightarrow |P| \quad P! : |P| \Rightarrow \star \quad \cdots$$

Pure cast reduction rules $\boxed{e \longrightarrow_c e}$

(Collapse) $\cdots \quad v\langle P_1! \rangle\langle P_2? \rangle \longrightarrow_c v\langle P_1 \Rightarrow P_2 \rangle \quad \text{if } |P_1| \sim |P_2|$

(Conflict) $v\langle P_1! \rangle\langle P_2? \rangle \longrightarrow_c \texttt{blame}\ L \quad \text{if } P_1 \Rightarrow P_2 = \bot^L$

Cast reduction rules $\boxed{e, \nu \longrightarrow_{cr} e, \nu}$

(PCastB)
$$\frac{e \longrightarrow_c e'}{e, \nu \longrightarrow_{cr} e', \nu}$$

(CastR1B)
$$\frac{\nu(a) = cv : P_1 \qquad P_3 = P_1 \vartriangle P_2 \qquad |P_3| \neq |P_1| \quad cv' = cv\langle P_1 \Rightarrow P_3 \rangle}{a\langle \mathsf{ref}\, P_2 \rangle, \nu \longrightarrow_{cr} a, \nu(a \mapsto cv' : P_3)}$$

(CastR2B)
$$\frac{\nu(a) = cv : P_1 \qquad P_1 = P_1 \vartriangle P_2}{a\langle \mathsf{ref}\, P_2 \rangle, \nu \longrightarrow_{cr} a, \nu}$$

(CastR3B)
$$\frac{\nu(a) = cv : P_1 \qquad P_1 \vartriangle P_2 = \bot^L}{a\langle \mathsf{ref}\, P_2 \rangle, \nu \longrightarrow_{cr} \texttt{blame}\ L, \nu}$$

Figure 18. Type system and cast reduction rules for monotonic references with blame

Program reduction rules $\boxed{e, \mu \longrightarrow_e e, \mu}$

$$\texttt{ref}_T\, v, \mu \longrightarrow_e a, \mu(a \mapsto v : T^{\emptyset}) \qquad \text{if } a \notin \mathit{dom}(\mu)$$

(DerefMB) $$!a, \mu \longrightarrow_e \mu(a)_{\text{val}}, \mu$$

(DynDrfMB) $$!a@T, \mu \longrightarrow_e \mu(a)_{\text{val}}\langle \mu(a)_{\text{rtti}} \Rightarrow T^{\emptyset} \rangle, \mu$$

(UpdMB) $$a{:=}v, \mu \longrightarrow_e a, \mu(a \mapsto v : \mu(a)_{\text{rtti}})$$

(DynUpdMB) $$a{:=}v@T, \mu \longrightarrow_e a, \mu(a \mapsto cv : \mu(a)_{\text{rtti}})$$

$$\text{where } cv = v\langle T^{\emptyset} \Rightarrow \mu(a)_{\text{rtti}} \rangle$$

For $X \in \{cr, e\}$:

$$\frac{e, \nu \longrightarrow_X e', \nu'}{F[e], \nu \longrightarrow_X F[e'], \nu'} \qquad F[\texttt{blame}\, L], \nu \longrightarrow_X \texttt{blame}\, L, \nu$$

State reduction rules $\boxed{e, \nu \longrightarrow e, \nu}$

$$\frac{e, \mu \longrightarrow_X e', \nu \qquad X \in \{cr, e\}}{e, \mu \longrightarrow e', \nu} \qquad \frac{\nu(a) = cv : P \qquad cv, \nu \longrightarrow_{cr} \texttt{blame}\, L, \nu'}{e, \nu \longrightarrow \texttt{blame}\, L, \nu'}$$

$$\frac{\nu(a) = cv : P \qquad cv, \nu \longrightarrow_{cr} cv', \nu' \qquad |\nu'(a)_{\text{rtti}}| = |P|}{e, \nu \longrightarrow e, \nu'(a \mapsto cv' : P)}$$

$$\frac{\nu(a) = cv : P \qquad cv, \nu \longrightarrow_{cr} cv', \nu' \qquad |\nu'(a)_{\text{rtti}}| \neq |P|}{e, \nu \longrightarrow e, \nu'}$$

Figure 19. Monotonic references with blame

I prove the blame-subtyping theorem via a preservation-style proof in which I preserve the e safe ℓ predicate [104]. This proof is conducted on the coercion calculus, so to relate the result back to the gradually-typed λ-calculus, a theorem is needed concerning the relationship between subtyping and coercion blame safety, Theorem 3.2. Recall that subtyping is defined in Figure 13 and compilation to coercions is defined in Figure 16.

Theorem 3.2 (Blame-Subtyping Theorem for coercion calculus). *For all* T_1, T_2, *and* ℓ, *it holds that* $T_1 <: T_2$ *iff* $(T_1 \Rightarrow^{\ell} T_2)$ safe ℓ.

Lemma 3.3 (Preservation of blame safety).
For all e, e', ν, ν', *and* ℓ, *if* e, ν safe ℓ *and* $e, \nu \longrightarrow e', \nu'$ *then* e', ν' safe ℓ.

I now move away from the coercion calculus and prove these important results on the gradually typed λ calculus with references. This latter language is indeed the one that programmers are expected to use. The following definitions will help to recast the results into the setting of the gradually typed language.

Definition 3.3 (Casts for a label in an expression). *Let* e *be an expression and* ℓ *a label, we say that* e contains the cast $T_1 \Rightarrow T_2$ for ℓ *whenever, in the derivation of* $\Gamma \vdash e \rightsquigarrow e' : T$, *there is the creation of a coercion via* $T_1 \Rightarrow^{\ell} T_2$.

Definition 3.4 (Blame safety for gradually-typed expressions). *A gradually-typed expression* e is safe for ℓ *if all the casts contained in* e *labeled* ℓ *respect subtyping.*

I now have all the ingredients to state and prove one of the main contributions of this chapter, i.e. the Blame-Subtyping Theorem for the gradually-typed λ calculus with references.

Lemma 3.4 (Translation preserves blame safety). *If* e safe ℓ *and* $\Gamma \vdash e \rightsquigarrow e' : T$, *then* e' safe ℓ.

Proof. The proof is a straightforward induction on $\Gamma \vdash e \rightsquigarrow e' : T$. □

Theorem 3.3 (Blame-Subtyping Theorem). *For all* e, e', T_1, T_2, ℓ, *if* $\emptyset \vdash e \rightsquigarrow e' : T$, e safe ℓ, *and* $e', \emptyset \longrightarrow \texttt{blame}\, L, \nu$, *then* $\ell \notin L$.

Proof. From the assumptions we have e' safe ℓ by Lemma 3.4. Then we conclude by applying the Blame-Subtyping Theorem for the coercion calculus. □

7. Implementation Concerns for Strong Updates

The monotonic semantics for references performs in-place updates to the heap with values of different type. In languages where values have uniform size, like many functional and object-oriented languages,

this does not pose a problem. However, for languages where values may have different sizes, in-place updates pose a problem. This issue can be addressed using an approach inspired by garbage collection techniques. When the semantics is to update a cell with a larger value than the current one, the implementation allocates a new piece of memory and places a forwarding pointer in the old location. When reading and writing through dynamic references, the implementation must check for and follow the forwarding pointers. However, when reading and writing through fully-static references, the implementation does not need to consider forwarding pointers because fully-static heap cells never move. Then during a garbage collection, the implementation can collapse sequences of forwarding pointers to reduce overhead in subsequent execution.

8. Related Work

Here I mention related work that is not discussed in the introduction or elsewhere in this chapter.

The casts and coercions studied in this chapter bear many similarities with contracts [35]. Racket [36] provides contracts for mutable values in the form of impersonators [85], which, for my purposes, can be viewed as implementing the guarded semantics of Herman et al. [48], as seen in Section 2.

Fähndrich and Leino [33] introduce a technique similar to monotonic references with their monotonic typestate. In this design, objects may flow from less restrictive to more restrictive typestates, but not vice versa. Unlike monotonic references, which require runtime checks due to the existence of dynamically-typed regions of code, their system enforces monotonicity statically.

9. Conclusions

I have presented a new design for gradually-typed mutable references, called monotonic references, the first to incur zero-overhead for reference accesses in statically typed code while maintaining the full expressiveness of a gradual type system. I defined a runtime enforcement strategy for monotonic references and presented a mechanized proof of type safety. Further, I defined blame tracking based on using labeled types in the run-time type information and proved the blame-subtyping theorem.

CHAPTER 4

Design and Evaluation of Gradual Typing for Python

1. Introduction

In order to evaluate the practical advantages and disadvantages of different approaches to runtime enforcement in gradually typed languages, a powerful testbed is required, beyond simply specialized calculi. In this chapter, I present *Reticulated Python*,[1] a framework for developing gradual typing for the Python language. Reticulated provides programmers with a variant of Python 3 that offers optional type annotations, and can then be used to experiment with different approaches as to the enforcement of these type annotations. Reticulated uses a static type system based on the first-order object calculus of Abadi and Cardelli [2], including structural object types. I augment this system with the dynamic type and *open* object types. Reticulated uses Python 3's annotation syntax for type annotations and a dataflow-based type inference system to infer types for local variables. Reticulated is available for download at `https://github.com/mvitousek/reticulated`.

Reticulated Python is implemented as a source-to-source translator that accepts syntactically valid Python 3 code, typechecks this code, and generates Python 3 code, which it then executes. The dynamic semantics of Reticulated differs from Python 3 in that run-time checks occur where implicit casts are needed to mediate between static and dynamically typed code. The run-time checks are implemented as calls into a Python libraries. In this way, I achieve a system of gradual typing for Python that is portable across different Python implementations and which may be applied to existing Python projects. I also made use of Python's tools for altering the module import process to insure that all imported modules are typechecked and translated at load time.

[1] Named for *Python reticulatus*, the largest species of snake in the python genus. "Reticulated" for short.

In addition to serving as a practical implementation of a gradually typed language, Reticulated has served as a test bed for experimenting with design choices for runtime enforcement strategies to mediate between static and dynamic code. While later work with Reticulated (in Chapters 5 and 6) explores one specific enforcement strategy in depth, the version of Reticulated presented in this chapter—*Reticulated v1*—is designed to be used for developing multiple new enforcement strategies and is agnostic to their particulars. With Reticulated v1, I implemented and evaluated three distinct enforcement strategies for mutable objects: 1) the traditional guarded strategy of Siek and Taha [76] and Herman et al. [48], but optimized with threesomes [79], 2) the *transient* strategy, an approach that does not use proxies but involves ubiquitous lightweight checks, and 3) the *monotonic* strategy, as described previously in Chapter 3. The guarded system is relatively complicated to implement and does not preserve object identity, which was found to be a problem in practice (see Section 3.2.5). The transient approach is straightforward to implement and preserves object identity, but when runtime checks discover that type constraints have been violated, it is unable to report the location of the original cast that caused the error to occur, a process known as blame tracking. It therefore is less helpful when debugging cast errors. The monotonic strategy preserves object identity and enables zero-overhead access of statically-typed object fields but requires locking down object values to conform to the static types they have been cast to.

To evaluate the usability of these designs, I performed case studies in which I applied Reticulated to several third-party codebases. I annotated them with types and then ran them using Reticulated. The type system design fared well, e.g. enabling statically-checked versions of statistics and cryptographic hashing libraries, while requiring only light code modifications. Further, these experiments detected several bugs extant in these projects. The experiments also revealed tensions between backwards compatibility and the ability to statically type portions of code. This tension is particularly revealed by the choice of whether to give static types to literals. These experiments also confirmed results from other languages [94]: type systems for Python should provide some form of occurrence typing to handle design patterns that rely heavily on runtime dispatch. Regarding the evaluation of the three enforcement strategies, the results indicate that the proxies of the guarded design can be problematic in practice due to the presence of identity tests in Python code. The transient and monotonic strategies both fared

well, although the principal efficiency benefits of the monotonic strategy, as described in Chapter 3, are unavailable in this setting.

Gradual typing for Python. Regardless of the choice of enforcement strategy, gradual type systems allow programmers to control of which portions of their programs are statically or dynamically typed. In Reticulated, this choice is made in function definitions, where parameter and return types can be specified, and in class definitions, where Python decorators are used to specify the types of object fields. When no type annotation is given for a parameter or object field, Reticulated gives it the dynamic (aka. unknown) type named `Dyn`. The Reticulated type system allows implicit casts to and from `Dyn`, as specified by the *consistency* relation [75]. In the locations of these implicit casts, Reticulated inserts casts to ensure, at runtime, that the value can be coerced to the target type of the cast.

One of the primary benefits of gradual typing over dynamic typing is that it helps to detect and localize errors. For example, suppose a programmer misunderstands what is expected regarding the arguments of a library function, such as the `moment` function of the `stat.py` module, and passes in a list of strings instead of a list of numbers. In the following, assume `read_input_list` is a dynamically typed function and the value it produces is a list of strings.

```
1 lst = read_input_list()
2 moment(lst, m)
```

In a purely dynamically typed language or an optionally typed language, a runtime type error will occur deep inside the library, perhaps not even in the `moment` function itself but inside some other function it calls, such as `mean`. It is then challenging for the library's client to fix the problem, since they may not know the precise cause nor even if it is in fact their fault, rather than a bug in the library. On the other hand, if library authors make use of gradual typing to annotate the parameter types of their functions, then the error can be localized and caught before the call to `moment`, resulting in easier debugging. The following shows the `moment` function with annotated parameter and return types.

```
def moment(inlist: List(Float), m: Int)→Float:
  ...
```

With this change, the runtime error points to the call to `moment` on line 2, where an implicit cast from `Dyn` to `List(Float)` occurred. A programmer using a library with gradual types need not use static types themselves — they gain the benefit of early localization and detection of errors even if they continue to write their own code in a dynamically typed style.

Casts on base values like integers and booleans are straightforward — either the value is of the expected type, in which case the cast succeeds and the value is returned, or the value is not, in which case it fails. However, casts on mutable values, such as lists, or higher-order values, such as functions and objects, are more complex. For mutable values, it is not enough to verify that the value meets the expected type at the site of the implicit cast because the value can later be mutated to violate the cast's target type. I discuss this issue in detail in Section 2.2.

Contributions.

- I develop Reticulated Python, a source-to-source translator that implements gradual typing on top of Python 3.
- In Section 2, I discuss Reticulated's type system and use Reticulated v1 to discuss three dynamic semantics for mutable objects, one of which is based on proxies and two new approaches: one based on use-site checks and one in which objects become monotonically more precisely typed.
- In Section 3, I carry out several case studies of applying gradual typing to third-party Python programs.
- In Section 4, I evaluate the source-to-source translation approach and consider lessons for other implementers of gradually-typed languages.

2. The Reticulated Python Designs

I present three language designs for Reticulated Python: guarded, transient, and monotonic. The three designs share the same static type system (Section 2.1) but have different dynamic semantics due to their different enforcement strategies (Sections 2.2.1, 2.2.2, and 2.2.3).

labels	ℓ		type variables X
base types	B	$::=$	`Int` | `String` | `Float` | `Bool` | `Bytes`
types	T	$::=$	`Dyn` | B | X | `List`(T) | `Dict`(T,T) | `Tuple`$(\overline{T})$ | `Set`(T) | `Object`$(X)\{\overline{\ell{:}T}\}$ | `Class`$(X)\{\overline{\ell{:}T}\}$`Function`$(P,T)$
parameters	P	$::=$	`Arb` | `Pos`$(\overline{T})$ | `Named`$(\overline{\ell{:}T})$

Figure 1. Static types for Reticulated programs

2.1. Static semantics. From a programmer's perspective, the main way to use Reticulated is to annotate programs with static types. Source programs are Python 3 code with type annotations on function parameters and return types. For example, the definition of a `distance` function could be annotated to require integer parameters and return an integer.

```
def distance(x: Int, y: Int)→ Int:
  return abs(x - y)
```

In Python 3, annotations are arbitrary Python expressions that are evaluated at the function definition site but otherwise ignored. In Reticulated, I restrict the expressions that can appear in annotations to the type expressions shown in Figure 1 and to aliases for object and class types. The absence of an annotation implies that the parameter or return type is the dynamic type `Dyn`.

The type system is primarily based on the first-order object calculus of Abadi and Cardelli [2] with several important differences discussed in this section.

2.1.1. Function types. Reticulated's function parameter types have a number of forms, reflecting the ways in which a function can be called. Python function calls can be made using keywords instead of positional arguments; for example, the `distance` function can be called by explicitly setting `x` and `y` to desired values like `distance(y=42, x=25)`. To typecheck calls like this, parameter names may be included in function types using the `Named` parameter specification, so in this case, the type of `f` is `Function(Named(x:Dyn, y:Dyn), Dyn)`. On the other hand, higher-order functions

$$\Gamma \vdash \texttt{class}\ \ X\colon \overline{\ell_k{:}T_k = e_k} : \texttt{Class}(X)\{\overline{\ell_k{:}T_k}\}$$

$$\frac{\Gamma \vdash e : \texttt{Class}(X)\{\overline{\ell_k{:}T_k}\} \qquad \nexists k.\ \texttt{\_\_init\_\_} = \ell_k}{\Gamma \vdash e() : \texttt{Object}(X)\{\overline{\ell_k{:}\mathit{bind}(T_k)}\}}$$

$$\frac{\Gamma \vdash e : T \qquad T = \texttt{Object}(X)\{\ell{:}T_\ell, \ldots\}}{\Gamma \vdash e.\ell : T_\ell[X/T]}$$

Figure 2. Class and object type rules.

most commonly call their function parameters using positional arguments, so for such cases Reticulated provides the `Pos` annotation. For example, the `map` function would take a parameter of type `Function(Pos(Dyn), Dyn)`; any function that takes a single parameter may then be passed in to `map`, because a `Named` parameters type is a subtype of a `Pos` when their lengths and element types correspond. Function types with `Arb` (for arbitrary) parameters can be called on any form of argument.

2.1.2. Class and object types. Reticulated includes types for both objects and classes, because classes are also runtime values in Python. Both of these types are structural, mapping attribute names to types, and the type of an instance of a class may be derived from the type of the class itself.

As an example of class and object typing, consider the following example:

```
1 class 1DPoint:
2   def move(self:1DPoint, x:Int)→1DPoint:
3     self.x += x
4     return self
5 p = 1DPoint()
```

Here the variable `1DPoint` has the type

```
1 Class(1DPoint){ move : Function(Named(self:1DPoint, x:Int),1DPoint) }
```

The occurrence of `1DPoint` inside of the class's structural type is a type variable bound to the type of an *instance* of `1DPoint`. Figure 2 shows Reticulated's typing rule for class definitions in the simple case where the class being defined has no superclass; classes that do have superclasses with static types also include the superclasses' members in their own type, and check that their instances' type is a subtype of that of their superclasses.

Values with class type may be invoked as though they were functions, as shown in the second rule of Figure 2. In the above example, `p` has type

```
Object(1DPoint) {move:Function(Named(x:Int),1DPoint)}
```

This type is derived from the class type of `1DPoint` by removing the self parameter of all the functions in the class' type. The type parameter `1DPoint` represents the self type. The *bind* meta-function converts function definitions from unbound form — with an explicit self-reference as their first parameter — to a form with this parameter already bound and thus invisible. If the self-referential type parameter `X` in an object type is not used in the types of any of its members I write `Object{…}` instead of `Object(X){…}`.

The type system also includes a rule to handle the situation when the class defines an `__init__` method, which in Python serves as the class' constructor, in which case Reticulated checks that the arguments of the construction call site match the parameters of `__init__`.

In Python, programmers can dynamically add properties to objects at will. In recognition of this, Reticulated's object types are *open* with respect to consistency — two object types are consistent if one type has members that the other does not and their common members are consistent; in other words, implicit downcasts on width subtyping are allowed and checked at runtime. This can be seen in line 3 of the above example: `x` is not part of the type of a `1DPoint`, so when `x` is accessed, an implicit downcast on `self` occurs to check that `x` exists. In this sense, Reticulated's object types are similar to the bounded dynamic types of Ina and Igarashi [52], except that their approach is appropriate for nominal type systems while open objects are appropriate for structural typing.

Programmers specify that object instances should have statically typed fields by using the `@fields()` decorator and supplying the expected field types. For example,

```
1 @fields({'x':Int, 'y':Int})
2 class 2DPoint:
3   def __init__(self:2DPoint):
4     self.x = 42
5     self.y = 21
6 2DPoint().x
```

This class definition requires that an instance of `2DPoint` have `Int`-typed fields named `x` and `y`; this information is included in the type of an instance of `2DPoint`, so the field access at line 6 has type `Int`. If `2DPoint`'s constructor fails to produce an object that meets this type, a runtime cast error is raised.

Lists, tuples, sets, and dictionaries have special, builtin types but they are also given object types that are used when they are the receiver of a method call.

2.1.3. Recursive type aliases. Structural object types are an appropriate match for Python's duck typing, but structural types can be rather verbose. To ameliorate this problem, class names are aliases for the types of their instances, as inferred from the class definition. Such aliases are straightforward when the aliased type only contains function and base types; however, obtaining the correct type for a given alias becomes more interesting in mutually recursive classes such as the following.

```
1 class A:
2   def foo(self, a:A, b:B):
3     pass
4 class B:
5   def bar(self, a:A, b:B):
6     pass
```

In the above code, `A` and `B` are type aliases when they appear in the annotations of `foo` and `bar`. For the remainder of this example, I use $\tilde{\texttt{A}}$ and `B` to represent type aliases and the unadorned `A` and `B` as bound type variables. The task here is to determine the appropriate types to substitute for $\tilde{\texttt{A}}$ and `B` in the type

$$
\begin{array}{rcl}
\texttt{Obj}(Y)\{\overline{\ell_i{:}T_i}\}[\hat{X}/T] & \longrightarrow & \texttt{Obj}(Y)\{\overline{\ell_i{:}T_i[\hat{X}/T']}\} \\
& \text{where} & T' = T[\hat{Y}/Y] \\
\hat{X}[\hat{X}/T] & \longrightarrow & T \\
\texttt{List}(T_1)[\hat{X}/T_2] & \longrightarrow & \texttt{List}(T_1[\hat{X}/T_2]) \\
& \ldots &
\end{array}
$$

Figure 3. Alias substitution

annotations of `foo` and `bar`. To arrive at these types, I start with the mapping

$$
\begin{array}{l}
[\hat{\texttt{A}} \mapsto \texttt{Object(A)\{foo:Function}([\hat{\texttt{A}}, \hat{\texttt{B}}], \texttt{Dyn})\}, \\
\hat{\texttt{B}} \mapsto \texttt{Object(B)\{bar:Function}([\hat{\texttt{A}}, \hat{\texttt{B}}], \texttt{Dyn})\}]
\end{array}
$$

The right-hand side of each pair in this mapping then has all the other pairs substituted into it using the substitution algorithm in Figure 3. This substitution repeats until it reaches a fixpoint, at which point all type aliases will have been replaced by object types or type variables. In the case of this example, the final mapping is

$$
\left[\begin{array}{l}
\hat{\texttt{A}} \mapsto \texttt{Object(A)\{foo:Function}([\texttt{A}, \\
\qquad \texttt{Object(B)\{bar:Function}([\texttt{A}, \texttt{B}], \texttt{Dyn})\}], \\
\quad \texttt{Dyn})\}
\end{array}\right]
$$

`B` receives a similar mapping.

2.1.4. Types of Literals. One of the objectives of Reticulated is to achieve at least the option of full backwards-compatibility with untyped Python code — that is, if a normal Python program is run through Reticulated, one would like the result to be observationally identical to what it would be if it were run directly.[2] This goal leads to certain surprising design choices, however. For example, it is natural to expect that an integer literal have type `Int`. However, that would to statically rejecting a program like `42 + 'hello world'`. This is a valid Python program in that it produces a result when evaluated (an exception), and the programmer has not "opted in" to static typing by using type annotations. So, to

[2]There are some places where this is violated: for example, if a Python function has pre-existing annotations that are syntactically identical to Reticulated's type annotations.

maintain maximal backwards compatibility with Python, even ridiculous programs like this cannot be rejected statically! Therefore I offer a flag in the Reticulated system to switch between giving integer literals the type `Dyn` or `Int`, and similarly for other kinds of literals. In Section 3 I discuss the practical effect of this choice.

2.1.5. Load-time typechecking. Reticulated's type system is static in the sense that typechecking is a syntactic operation, performed on a module's AST. However, when a program first begins to run, it is not always possible to know which modules will be imported and executed. Reticulated's typechecking, therefore, happens at the load time of individual modules. This can mean that a static type error is reported after other modules of a program have already run.

Reticulated does attempt to perform static typechecking ahead of time: when a module is loaded, it tries to locate the corresponding source files for any further modules that need to be loaded, typecheck them, and import the resulting type information into the type environment of the importing module. This is not always possible, however — modules may be imported at runtime from locations not visible to Reticulated statically. In general, programmers cannot count on static type errors to be reported when a program starts to execute, only when the module with the error is loaded.

2.1.6. Dataflow-based type inference. Python 3 does not provide syntax for annotating the types of local variables. Reticulated could use function decorators or comments for this purpose, but I instead choose to infer types for local variables. I perform a simple intraprocedural dataflow analysis [55] in which each variable is given the type that results from joining the types of all the values assigned to it, a process which is repeated until a fixpoint is reached. For example, consider the function

```
1 def h(i:Int):
2   x = i; y = x
3   if random():
4     z = x
5   else: z = 'hello world'
```

The system infers that `x` and `y` have type `Int`, because the expressions on the right-hand sides have that type, and that `z` has type `Dyn`, which is the join of `Int` and `Str`. This join operation is over the

subtyping lattice with Dyn as top from Siek et al. [80], and always results in a type that can be safely casted to.

2.2. Dynamic semantics. With the enforcement-strategy-agnostic approach of Reticulated v1, I explore three different dynamic semantics for Reticulated Python, varying by which enforcement strategy is used. Consider the following example in which a strong (type-changing) update occurs to an object to which there is an outstanding statically-typed reference.

```
1 class Foo:
2   bar = 42
3 def g(x):
4   x.bar = 'hello world'
5 def f(x:Object({bar:Int}))→Int:
6   g(x)
7   return x.bar
8 f(Foo())
```

Function f passes its statically-typed parameter to the dynamically-typed g, which updates the bar field to a string. Function f then accesses the bar field, expecting an Int.

In this section, I discuss how each enforcement strategy detects this error at runtime.

2.2.1. The guarded enforcement strategy. The guarded enforcement strategy is implemented in Reticulated following the standard design of Herman et al. [48], as described in Chapter 2. To briefly review this strategy and to show its use in Reticulated: when using the guarded strategy, Reticulated inserts casts into programs where implicit coercions occur, such as at function call sites (like line 6 above) and field writes. The inserted cast — which is a call to a Python function which performs the cast — is passed the value being casted as well as type tags for the source and target of the cast, plus an error message and line number that will be reported if and when the cast fails (I elide this error information in the examples). For example, the above program will have casts inserted as follows:

```
3 def g(x):
4   cast(x, Dyn, Object({bar:Dyn})).bar = 'hello world'
```

```
5 def f(x:Object({bar:Int}))→Int:
6   g(cast(x, Object({bar:Int}), Dyn))
7   return x.bar
8 f(cast(Foo(), Object({bar:Dyn}), Object({bar:Int})))
```

Casts between Dyn and base types will simply return the value itself (if the cast does not fail — if it does, it will produce a `CastError` exception), but casts between function or object types produce a proxy. This proxy contains the error message and line number information provided by the static system for this cast, which serves as blame information if the cast's constraints are violated later on. Guarded proxies do not just implement individual casts — they represent compressed sequences of casts using the threesomes of Siek and Wadler [79]. In this way, proxies do not build up on themselves, layer after layer — a proxy is always only one step away from the actual, underlying Python value.

Method calls or field accesses on proxies redirect to a lookup on the underlying object, the result of which is casted from the part of the source type that describes the member's type to the same part of the meet type, and then from the meet type to the final target type. This process occurs in reverse for member writes. Proxied functions, when invoked, cast the parameters from target to meet to source, and then cast results from source to meet to target. In the above example, when the field write occurs at line 4, the proxy will attempt to cast `'hello world'` to `Int`, which is the most precise type that the object being written to has had for `bar` in its sequence of casts. This cast fails, and a cast error is reported to the programmer.

One important consequence of the guarded approach is that *casts do not preserve object identity* — that is, if the expression

```
x is cast(x, Dyn, Object({bar:Int}))
```

were added to function g in the example above (where `is` is the Python operator for pointer equality), it would return `False`. Similarly, the Python runtime type is not preserved: the type of a proxy is `Proxy`, not the type of the underlying object, and this information is observable to Python programs that invoke the builtin `type` function on proxies. However, the proxy class is a subclass of the runtime Python class

of the underlying value,[3] instance checks generally return the same result with the bare value as they do with the proxied value. In the case studies in Section 3, I evaluate the consequences of this issue in real-world code.

2.2.2. The transient dynamic semantics. In the transient enforcement strategy, a runtime operation checks that the value has a type consistent with the type it is expected to have, but does not wrap the value in a proxy. Returning to the running example, just as in the guarded semantics, runtime enforcement code is inserted around the argument in the call to function `f`, but instead of being a cast, here the enforcement is performed by a *check*:

```
8 f(check(Foo(), Object({bar:Int})))
```

In the transient strategy, this operation checks that `Foo()` is an object, that it has a member named `bar`, and that `bar` is an `Int`. It then returns the object. The check's effect is therefore *transient*; nothing prevents the field update at line 4. To prevent `f` from returning a string value, an additional check is instead inserted at the point where `f` reads from `bar`:

```
5 def f(x:Object({bar:Int}))→Int:
6   g(check(x, Object({bar:Int}), Dyn))
7   return check(x.bar, Int)
```

This check attempts to verify that `x.bar` has the expected type, `Int`. Because the call to `g` mutates `x.bar` to contain a string, this check fails, preventing an uncaught type error from occurring.

In addition, it is possible that `f` could be called from a context in which its type is not visible, if it was passed into type `Dyn` for example. In this case, the call site cannot check that the argument being passed to `f` is a `List(Int)`, and unlike the guarded system, there is no proxy around `f` to check that the argument has the correct type either. Therefore, `f` needs to check that its parameters have the expected type in its own context. Thus, the final version of the function becomes

```
5 def f(x:Object({bar:Int}))→Int:
6   check(x, Object({bar:Int}))
```

[3] Unless the underlying value's class is non-subclassable, such as `bool` or `function`.

```
7   g(check(x, Object({bar:Int}), Dyn))
8   return check(x.bar, Int)
```

In general, checks are inserted at the use-sites of variables with non-base types and at the entry to function bodies and for-loops. Checks are used in these circumstances to verify that values have the type that they are expected to have in a given context before operations occur that may rely on that being the case. Because of these additional checks, it is sound to remove checks inserted on arguments at call sites: instead of `f(check(Foo(), Object({bar:Int})))`, one could simply write `f(Foo())`, and trust that `f` will check at its own entry point that its argument has type `Object({bar:Int})`. Both strategies have been taken by Reticulated during points in its development; for the remainder of this chapter I will assume that this check is included.

Figure 4 shows an excerpt of the `has_type` function used within transient checks. Some values and types cannot be eagerly checked by `has_type` function, however, such as functions — all that Reticulated can do at the check site is verify that a value is callable, not that it takes or returns values of a

```
1 def has_type(val, ty):
2   if isinstance(ty, Dyn):
3     return True
4   elif isinstance(ty, Int):
5     return isinstance(val, int)
6   elif isinstance(ty, Object):
7     return all(hasattr(val, member) and has_type(getattr(val, member),
          ty.member_type(member)) for member in ty.members)
8   elif isinstance(ty, Function):
9     return callable(val)
10  elif ...
```

Figure 4. Definition for the `has_type` function.

particular type. Function types need to be enforced by checks at call sites. Moreover, even when eager checking is possible, the transient design only determines that values have their expected types at time of the check site, and does not detect or prevent type-altering mutations from occurring later — instead, such mutations are prevented from being observable by use-site checks. Chapter 5 shows a formal calculus for the transient enforcement strategy.

Transient and Dart's checked mode. The transient semantics for Reticulated is reminiscent of Dart's *checked mode* [39], in which values are checked against type annotations (which are otherwise ignored at runtime). However, Dart's checks are performed under different circumstances than Reticulated's, and these choices cause Dart's type system to be unsound even when checked mode is enabled. Consider the following Dart program:

```
1 void g(⋆amic foo) {
2   foo[0] = "hello world";
3 }
4 void f(List⟨int⟩ foo) {
5   g(foo);
6   print(foo[0]);
7 }
8 main() {
9   List⟨⋆amic⟩ foo = new List⟨⋆amic⟩();
10  foo.add(42);
11  f(foo);
12 }
```

Dart does not check the value that results from the index at line 6, unlike transient Reticulated. Therefore "hello world" will be printed despite the presence of a string in a list of integers.

Additionally, in Dart, object updates are checked against the annotated field types of the underlying object, rather than the type of the reference to the object in the scope of the update, causing the following program to fail.

```
1 class Foo {
2   int bar = 42;
3 }
4 void f(⋆amic a) {
5   a.bar = "hello world";
6 }
7 main() {
8   f(new Foo());
9 }
```

In transient Reticulated, the equivalent program would run without error until the `bar` field of the object is read in a context where it is expected to be an `int`. Dart's object update checks therefore behave more like guarded than transient.

2.2.3. The monotonic dynamic semantics. The monotonic enforcement strategy for references, described previously in Chapter 3, is also implemented in Reticulated v1 for its mutable values. Like the transient strategy, the monotonic strategy avoids using proxies, but rather than using transient checks, the monotonic approach preserves type safety by restricting the types of objects themselves. In this approach, when an object flows through a cast from a less precise (closer to `Dyn`) type to a more precise one, the cast has the effect of locking the type of the object at this more precise type. Objects store a type for each of their fields; this type is always *equally or more precise* than any of the types at which the field has been viewed. For example, if an object has had references to it with types `{'foo':Tuple(Int, Dyn)}` and `{'foo':Tuple(Dyn, Int)}`, the object will record that `'foo'` must be of type `Tuple(Int, Int)`, because a value of this type is consistent with both of the references to it.

When field or method updates occur, the newly written value is cast to this precise type. Back to the ongoing example:

```
3 def g(x):
4   cast(x, Dyn, Object({bar:Dyn})).bar = 'hello world'
```

```
5 def f(x:Object({bar:Int}))→Int:
6   g(cast(x, Object({bar:Int}), Dyn))
7   return x.bar
8 f(cast(Foo(), Object({bar:Dyn}), Object({bar:Int})))
```

Under the monotonic semantics, casts are inserted at the same places as they are in guarded, but their effects are different. When the newly created object goes through the cast at line 8, the object is permanently set to only accept values of type `Int` in its `bar` field. When g attempts to write a string to the object at line 4, the string is cast to `Int`, which fails. This cast is performed by the object itself, not by a proxy — the values of `x` in `f` and g are the same even though they appear at different types.

The monotonic system's approach of permanently, monotonically locking down object types results is one clear difference from guarded and transient — it is not possible to pass an object from untyped code into typed code, process it, and then treat it as though it is once again dynamically typed. The object's sojourn into statically-typed code has forever rendered it incapable of containing values whose types conflict with those that the typed code expected. On the other hand, monotonic's key guarantee is that the type of the actual runtime value of an object is at least as precise as any reference to it. Because of this, when a reference is of fully static type, the value of the object has the same type as the reference, and no casts or checks are needed when a field is accessed. Even if the reference is of a partially dynamic type, only an upcast needs to occur. The requirement of monotonicity is enforced by tracking the most precise type a value has been casted to; the type is stored as part of the value.

3. Case Studies and Evaluation

To evaluate the design of Reticulated's type system and runtime systems, I performed case studies on existing, third-party Python programs. I annotated these programs with Reticulated types and ran them under each enforcement strategy. I categorized the code that could not be typed statically and identified several additional features that would let more code be typed. I discovered several situations in which I had to modify the existing code to interact well with the system, and I also discovered several bugs

in these programs in the process. The annotated case studies used in this experiment are available at `http://bit.ly/1rqSvQM`.

3.1. Case study targets. Python is a very popular language and it is used for many applications, from web backends to scientific computation. To represent this wide range of uses, I chose several different code bases to use as case studies for Reticulated.

3.1.1. Statistics library. I chose the Harvard neuroscientist Gary Strangman's statistics library[4] as a case study as an example of the kind of focused numerical programs that are common in scientific Python code. The `stats.py` module contains a wide variety of statistics functions and the auxiliary `pstat.py` module provides list manipulation functions.

To prepare them for use with Reticulated, I removed the libraries' dependence on the external `Numeric` array library. This had the effect of reducing the amount of "Pythonic" dynamicity that exists in the libraries — prior to my modification, two versions of most functions existed, one for `Numeric` arrays and one for native Python data structures, and a dispatch function would redirect any call to the version suited to its parameter. Although I removed this dynamic dispatch from these modules, this kind of behavior is considered in the next case study. I then simply added types to the libraries' functions based on their operations and the provided documentation, and replaced calls into the Python `math` library with calls into statically typed math functions.

3.1.2. CherryPy. CherryPy[5] is a lightweight web application framework written for Python 3. It is object-oriented and makes heavy use of callback functions and dynamic dispatch on values. My intent in studying CherryPy was to realistically simulate how library developers might use gradual typing to protect against bad user input and report useful error messages. To accomplish this, I annotated several functions in the CherryPy system with types. I tried to annotate client-facing API functions with types, but in many cases it was not possible to give specific static types to API functions, for reasons I discuss in Section 3.2.6. In these situations, I annotated the private functions that are called by the API functions instead.

[4] `http://www.nmr.mgh.harvard.edu/Neural_Systems_Group/gary /python.html`

[5] `http://www.cherrypy.org/`

3.1.3. SlowSHA. I added types to Stefano Palazzo's implementation of SHA1/SHA2.[6] This is a straightforward 400 LOC program that provides several SHA hash algorithms implemented as Python classes.

3.2. Results. By running these case studies in Reticulated, I made a number of discoveries about both the system and the targets of the studies themselves. I discovered two classes of bugs that exist in the target programs which were revealed by the use of Reticulated. I also found several deficiencies in the Reticulated type system which need to be addressed, and some challenges that the guarded system in particular faces due to its use of proxies: the CherryPy case study relies on checks against object identity, and the inability of the guarded semantics to preserve object identity under casts causes the program to crash.

3.2.1. Reticulated reveals bugs. I discovered two potential bugs in the target programs by running them with Reticulated.

Missing return values. Below is one of the annotated functions from `stats.py`:

```
1 def βcf(a:Float,b:Float,x:Float)→Float:
2   ITMAX = 200
3   EPS = 3.0e-7
4 # calculations elided...
5   for i in range(ITMAX+1):
6 #   more calculations elided...
7     if (abs(az-aold)⟨(EPS⋆abs(az))):
8       return az
9   print('a or b too big, or ITMAX too small in Betacf.')
```

This function only conditionally returns a value; if the inputs are such that the for loop executes more than `ITMAX+1` times, the function "falls off" the end of its definition. In Python this has the effect of returning the unitary `None` value. This falling-off behavior poses a problem when `betacf` is called by other functions in the library, which do not check if the result is `None`. None of the test cases supplied by the developer trigger this bug, but it could be triggered by client code. This behavior could cause

[6] `http://code.google.com/p/slowsha/`

a confusing error on the caller's side, or even worse, it could continue to execute, likely producing an incorrect result. The printed error message could easily be missed by the programmer, especially since `stats.py` is a library, and it may be used by clients unfamiliar with the design of its functions. By annotating `betacf` with the static return type `Float`, the function is forced to always either return a float value, or to raise an exception. In this case, the appropriate fix is to replace the final line with something like

```
9   raise Exception('a or b too big, or ITMAX too small in Betacf.')
```

which Reticulated's type system will accept.

Parameter name mismatch. Reticulated's type system is designed to guarantee that when one class is a subclass of another, the type of an instance of the subclass is a subtype of the type of an instance of the superclass. Additionally, as I discuss in Section 2.1.1, function types can include the names of their parameters so that calls with keyword arguments may be typed.

These properties of Reticulated cause it to report a static type error when a subclass overrides a superclass's method *without* using the same parameter names. I regard this situation as a latent bug in a program that exhibits it, as illustrated below:

```
1 class Node:
2   def appendChild(self, node):
3 #   ...
4 class Entity(Node): # subclass of Node
5   def appendChild(self, newChild):
6 #   ...
```

If the programmer expects that any instance `nd` of `Node` or its subclasses can be used in the same fashion, then they expect that

```
7 nd.appendChild(node=Node())
```

will always succeed. However, if `node` is actually an instance of `Entity`, this call will result in a Python `TypeError`. This is an easy mistake to make, and this pattern occurs in multiple places — even within

the official Python Standard Library itself, which is where this example arises.[7] I have not encountered situations where this potential bug actually results in a runtime error; it would be unusual for `node/newChild` to be used as a keyword argument.

3.2.2. The Reticulated type system in practice. The choice of whether or not to include typed literals, as discussed in Section 2.1.4, greatly affects the behavior of math-heavy code. Enabling typed literals requires some code to be slightly changed, but lets substantially more code be entirely statically typed.

Invariant mutable types and typed literals mix poorly. The statistics libraries I studied frequently intermingle integers and floating point values, including within lists, as in this example:

```
1 def var (inlist:List(Float))→Float:
2   ...
3   mn = mean(inlist)
4   deviations = [0]*len(inlist)
5   for i in range(len(inlist)):
6     deviations[i] = inlist[i] - mn
7   return ss(deviations)/float(n-1)
```

In this function, the `deviations` variable is initialized to be a list of integer zeros. Reticulated's type inferencer only reasons about assignments to variables, such as that at line 4, not assignments to attributes or elements, such as that at line 6. Therefore, when number literals have static types, the type inference algorithm will determine that the type of `deviations` is `List(Int)`. However, float values are written into it at line 6, and at line 7 it is passed into the function `ss`, which has been given the type `Function([List(Float)],Float)`. The Reticulated type system detects this call as a static type error because list element types are invariant under subtyping (though not under consistency); Reticulated's subtyping rules contain the only the rule

$$\Gamma \vdash \texttt{List}(T_1) <: \texttt{List}(T_1)$$

[7] In the Python 3.2 standard library, in `xml/minidom.py`.

for lists. Even though `Int` is a subtype of `Float`, neither the type `List(Int)` nor `List(Float)` is appropriate for `deviations` as written. In such situations, the code can be rewritten to only use float values; in this case I changed line 4 to

```
4   deviations = [0.0]*len(inlist)
```

Typed literals and type inference allow math to be typed. When typed literals are enabled, and any necessary edits to the source are made as above, Reticulated's approach to type inference allows many of the calculation-heavy statistics functions that I studied to become almost entirely statically typed, with few or no casts to or from `Dyn`. For example, the sum-of-squares function behaves extremely well:

```
1 def ss(inlist: List(float)) →float:
2   _ss = 0
3   for item in inlist:
4     _ss = (_ss + (item *item))
5   return _ss
```

This code above actually shows this function *after* cast insertion — no casts have been inserted at all,[8] and the computations here occur entirely in typed code. Overall, when typed literals are enabled, 48% of the binary operations in `stats.py` occur in typed code, compared to only 30% when literals are assigned the dynamic type. Reticulated, as a test-bed for gradual typing in Python, is not currently able to make use of this knowledge, but a system that does perform type optimizations would be able to execute the mathematical operations in this function entirely on the CPU, without the runtime checks that Python typically must perform.

3.2.3. Cast insertion with open structural object types. In general, structural objects and object aliases worked well for the test cases. However, I discovered one issue that arose in each runtime system because the rules for cast insertion on object accesses made an assumption that clashed with accepted

[8] Using the guarded semantics. Transient checks would be inserted to check that `inlist` is a list of floats at the entry to the function, and that `item` is a `Float` at the beginning of each iteration of the loop.

Python patterns. One rule for cast insertion in the guarded and monotonic enforcement strategies is

$$\frac{\Gamma \vdash e \rightsquigarrow e' : T \qquad T = \texttt{Object}(X)\{\overline{\ell_i{:}T_i}\} \qquad \forall i.\ \ell_i \neq x}{\Gamma \vdash e.x \rightsquigarrow \texttt{cast}(e', T, \texttt{Object}(X)\{x : \texttt{Dyn}\}).x : \texttt{Dyn}}$$

That is, when a program tries to read an object member that does not appear in the object's type, the object is cast to a type that contains that member at type Dyn. A similar check would be inserted by the transient strategy. This design clashes with the common Python pattern below:

```
try:
  value.field
  # do something
except AttributeError:
  # do something else
```

If the static type of `value` does not contain the member `field`, Reticulated verifies at runtime that the field exists. This implicit downcast, allowed because of the *open* design for object types, causes this program to typecheck statically even if the static type of `value` does not contain `field` — without open object types this program would not even be accepted statically. However, even with this design, this program still has a problem: if and when that cast fails, a cast error is triggered, which would not let the program continue down the `except` branch as the programmer intended. In order to support the expected behavior of this program, cast errors are designed to be caught by the try/except block. Cast errors are implemented as Python exceptions, so by letting any cast errors generated by this particular kind of object cast actually be instances of a subclass of `AttributeError`, the enforcement code anticipates the exception that would have been thrown without the cast. This specialized cast failure is then caught by the program itself if it was designed to catch `AttributeErrors`. Otherwise, the error is still reported to the programmer as a cast error with whatever blame information is supplied. A similar solution is implemented for function casts, since programs may call values if they were functions and then catch resulting exceptions if they are not.

3.2.4. Monotonic and inherited fields. The basic principle behind the monotonic approach is that when objects are cast to a more precise type, any future values that may inhabit the object's fields must

meet this new precise type as well. However, it is not always clear *where* this "locking down" should happen. Field accesses on Python objects can return values that are not defined in the object's local dictionary but rather in the object's class or superclasses. Therefore, when a monotonic object goes through a cast that affects the type of a member that is defined in the object's class hierarchy, there are two choices: either that member can be downcasted and monotonically locked down in its original location, or it can be copied to the local object and locked down there. Both designs have problems: the former will cause all instances of the class to behave as though they had gone through the cast, while the latter causes class attributes to be eagerly copied into objects, damaging space efficiency and blinding instances to mutation of class attributes, significantly altering the program's behavior.

Initially I chose the former behavior, monotonically locking down fields and methods in their original locations. However, applying this behavior to lists in the statistics library revealed an additional problem: it is impossible to monotonically downcast and lock members of builtin classes, such as lists. Even if it had been possible to do so, however, this approach would have made it so that there could only ever be one type of list in any Reticulated program. Instead, the monotonic system copies class fields to the dictionary of the casted object.

3.2.5. Challenges for guarded. The guarded enforcement strategy causes several problems that do not occur with the transient strategy and are less severe with monotonic, because proxies do not preserve object or type identity. The loss of object identity was a serious problem that prevented the CherryPy case study from even running successfully, and the loss of type identity meant that I had to modify the statistics library case study for it to run.

Object identity is a serious issue.... I was aware from the outset that the design for the guarded strategy does not preserve object identity. However, it was initially unclear how significant of a problem this is in practice. Object identity was not relevant in the statistics or hashing libraries, but identity checks are used in CherryPy in a number of locations. In particular, there are places where the wrong outcome from an identity check causes an object to be incorrectly initialized, as in the following:

```
1 class _localbase(object):
2   def __new__(cls, *args, **kw):
3     ...
```

```
4     if args or kw and (cls.__init__ is object.__init__):
5       raise TypeError("Initialization arguments are not supported")
6     ...
```

If the parameter `cls` is proxied, then `cls.__init__` will be as well. In that case, the identity check at line 4 will return `False` if the underlying value is `object.__init__`. The `TypeError` exception will not be raised, and confusing errors may occur later.

In many places where identity testing is used simply replacing `is` with $==$ (which is equivalent to calling the `__eq__` method, which is defined per-class) would have the same effect and be compatible with the use of proxies. However, identity testing is an extremely fast operation, and some implementations of `__eq__` may be highly expensive or cause side effects. Furthermore, object identity issues also arose from calls into the Python Standard Library, causing unpredictable, incorrect behavior. Python's pickling library is unable to correctly serialize proxied objects, and file reads on proxied file objects occasionally and unpredictably fail for unclear reasons. The end result of this was that the CherryPy webserver was unable to run without error under the guarded semantics. Modifying it to work with guarded would be a substantial undertaking, requiring modification to the Python Standard Library itself in order to work.

...and type identity sometimes is. Although the statistics library never checks object identity, it includes 28 static code locations where the `type` function is used to get a value's Python runtime type. At these sites, unexpected behavior can arise because proxies' Python types are not the same as the Python types of their underlying objects. Fortunately, these situations only required minor edits to resolve. One problematic location was the following code from `pstat.py`:

```
1 def abut (source, *args)→List(Dyn):
2   if type(source) not in [list, tuple]:
3     source = [source]
4   ...
```

If the value of `source` is a proxy, then the call to `type` on line 2 will return the `Proxy` class, resulting in `source` being wrapped in a list even if the underlying value already is a list. This can be solved by

providing a substitute `type` function that is aware of proxies, or by rewriting this line of code to use `isinstance`:

```
2   if not any(isinstance(source, t) for t in [list,tuple]):
```

Reticulated arranges for the class of a proxy to be a subclass of the class of the underlying value, and with this modification, I was able to run the statistics library under the guarded strategy.

3.2.6. Classifying statically untypable code. In many circumstances I found functions that could not be given a fully static type. This is to be expected — Python is a dynamic language and many Python programs are written in idioms that depends on dynamic typing. Wisely-chosen extensions of Reticulated's static type system would let certain classes of these currently-untypable functions be statically checked, but sometimes the appropriate thing to do is just use `Dyn`. I classify these situations, discuss how frequently they arise, and consider which features, if any, would allow them to be assigned static types.

Dynamically typed parameters may act like generics. One deficiency of Reticulated's type system is its lack of support for generics. Dynamic typing lends itself well to a generic style of programming, and thus it is no surprise that many of the functions and portions of code that could be given static types to would by much more typable if the type system provided generics.

Dispatch occurs on Python runtime types. The below code snippet — the definition of the `update` method invoked in the previous example — shows a pattern used extensively in CherryPy which is difficult to type precisely:

```
1 def update(self, config):
2   if isinstance(config, string):
3     config = Parser().read(config).as_dict()
4   elif hasattr(config, 'read'):
5     config = Parser().readfp(config).as_dict()
6   else:
7     config = config.copy()
8   self._apply(config)
```

(This snippet actually combines together two CherryPy functions for clarity.) The `update` method may take any of three kinds of values: a string, a value with the attribute "`read`" (contextually, a file), or something else, but whatever it initially is, it is eventually converted into a value which is passed into the `_apply` method.

It is clear that `config` cannot be annotated with a single precise type — the logic of the program depends on the fact that `config` may be any of several disparate types. Static typing could be achieved by introducing sum types into the language, or by using the occurrence typing of Tobin-Hochstadt and Felleisen [95]. In the absence of such techniques, static typing can still be helpful if the `_apply` function is annotated instead, as shown below:

```
9 def _apply(self, config:Dict(Str, Dyn))→Void:
10   ...
```

By declaring that this function accepts only `config` values that are dictionaries with string keys, run-time enforcement code will be inserted to ensure that, no matter what the value passed in to `update` was, the one handed off to `_apply` will have the correct type.

Data structures can be heterogeneous. Dynamic typing enables the easy use of heterogeneous data structures, which naturally cannot be assigned static types. In these case studies, I did not find significant use of heterogeneous lists, other than ones that contain both `Int`s and `Float`s as discussed in Section 3.2.2. The same is *not* true of dictionaries, whose values frequently display significant heterogeneity in CherryPy, as seen in this call into CherryPy's API from a client program:

```
1 cherrypy.config.update({
2   'tools.encode.on': True,
3   'tools.encode.encoding': 'utf-8',
4   'tools.staticdir.root': os.path.abspath(os.path.dirname(__file__)),
5 })
```

This dictionary, which contains only strings as keys but both strings and booleans as values, is representative of many similar configuration dictionaries used in CherryPy and other Python libraries. Reticulated can represent such heterogeneous types — this dictionary could be given the type `Dict(Str,`

Dyn). This example could be given a more precise type if sum types were introduced into the Reticulated type system.

`eval` and other villains. Finally there are cases where fundamentally untypable code is used, such as the `eval` function. The effect of eval and its close relative `exec` is, of course, unpredictable at compile time. In some cases, `eval` is used in rather astonishing fashions reminiscent of those described by Richards et al. [71]:

```
1 def dm (listoflists,criterion):
2   function = 'filter(lambda x: '+criterion+',listoflists)'
3   lines = eval(function)
4   return lines
```

This `pstat.py` function is evidently written for the use of novice programmers who do not understand how to use lambdas but who still wish to use higher order functions. Examples like this, and other bizarre operations such as mutation of the active stack can throw a wrench in Reticulated's type system.

Miscellaneous. In addition to these sources of dynamism, values can also be forced to be typed `Dyn` due to more mundane limitations of the Reticulated type system. Functions with variable arity and those that take arbitrary keyword arguments have their input annotations ignored and their parameter type set to `Arb`; designing a type specification for functions that captures all of such behavior is an engineering challenge and is important for practical use of Reticulated. Reticulated also does not yet typecheck metaclasses or function decorators; values that use them are simply typed `Dyn`.

3.2.7. Efficiency. None of these approaches make any attempt to speed up typed code. The mechanisms that they use to perform runtime checks slow it down, and because Reticulated works by generating standard Python 3 code, when a Python interpreter executes this code it will perform its standard runtime checks even when Reticulated's type system has proven them unnecessary. As a result, Reticulated programs perform far worse than their unchecked Python implementations — the slowSHA test suite, for example, has on the order of a 10x slowdown under transient compared to normal Python. The version of Reticulated Python studied in this chapter does not include much optimization effort and this could be much improved, but never beyond baseline Python. However, the enforcement strategies can

be compared to each other meaningfully. The test suite included with the statistics library took 2.7 seconds to run under the guarded semantics and 5.5 under monotonic, but only 1.6 with transient. The slowSHA library test suite took 10.4 seconds with guarded and 13.6 with monotonic, but only 5.1 with transient. These figures are from an 8-core Intel i7 at 2.8 GHz, and they exclude time spent in the type-checker and cast inserter. Due to issues with object identity, CherryPy was unable to run without error at all when using the guarded semantics, as discussed in Section 3.2.5, so I do not show a timing figure that program.

This result indicates that the simple, local checks made by the transient semantics are, taken together, more efficient than the proxy system used by guarded and (for functions only) monotonic, and that the special getters and setters used by monotonic are expensive, even if they do not cause casts to occur. This may simply be because these features rely on Python's slow reflection and introspection features, as discussed in Section 4; in any case, this enhances the advantages of the transient design.

4. Implementation

Reticulated is implemented as a source-to-source translator, and thus the enforcement code for each strategy is itself Python code. In this section, I discuss the implementation of the enforcement operations for each of design studied in this chapter.

4.1. Guarded. Developing the guarded system's proxies was the largest engineering challenge of Reticulated's implementation. These objects need to accurately imitate the behavior of their underlying values. Fundamental limitations of Python prevent them from fully doing so, as I discuss in Section 3.2.5, however, proxies that behave correctly in many cases are implementable.

The proxies of the guarded system are implemented as instances of a `Proxy` class which is defined at the cast site where the proxy is installed. The `__getattribute__` property of the `Proxy` class, which controls attribute access on Python objects, is overrided to instead retrieve attributes from the casted value and then cast them. The `__setattr__` and `__delattr__` methods, which control attribute update and deletion respectively, are similarly overridden. This is sufficient for most proxy use cases.

However, classes themselves can be casted, and therefore must be proxied in the guarded system. Moreover, when a class' constructor is called via a proxy, the result should be an object value that obeys the types of its class, and which therefore itself needs to be a proxy — even though there was no "original," unproxied object in the first place.

Python is a rich enough language to allow us to accomplish this. In Python, classes are themselves instances of metaclasses, and so a class proxy is a class which is an instance of a `Proxy` metaclass. When an object is constructed from a class proxy, the following code is executed:

```
1  underlying = object.__new__(cls)
2  proxy = retic_make_proxy(underlying, src.instance(), trg.instance())
3  proxy.__init__()
```

In this code, `cls` is the underlying class being proxied. The call to `object.__new__` creates an "empty" instance of the underlying class, *without* the class's initialization method being invoked. Then an object proxy is installed on this empty instance; the source and target types of the cast that this proxy implements are the types of instances of the class proxy's source and target types. Finally, the initialization method is invoked on the object proxy.

Another source of complexity in this system comes from the `eval` function. In Python, `eval` is dynamically scoped; if a string being `eval`ed contains a variable, `eval` will look for it in the scope of its caller. This poses a problem when `eval` is coerced to a static type and is wrapped by a function proxy, because then `eval` only has access to the variables in the *proxy's* scope, not the variables in the proxy's caller's scope. To handle this edge case, proxies check at their call site if the function they are proxying is the `eval` function. If it is, the proxy retrieves its caller's local variables using stack introspection, and runs `eval` in that context.

Another challenge comes from the fact that some values in Python are not Python code. In the CPython implementation (initially the only major Python implementation to support Python 3, and therefore the main target of Reticulated's implementation), many core features are implemented in C, not self-hosted by Python itself. However, C code does not respect the indirection that proxies use, and so when it tries to access members from the proxy, it might find nothing. I developed a partial solution for this by

```
1 def retic_check(val, trg, msg)
2   assert has_type(val, trg, msg)
3   return val
```

Figure 5. Casts and checks in the transient system.

removing proxies before values are passed into C whenever possible. The C code is then free to modify the value without respecting the cast's type guarantees, but if a read occurs later, the reinstalled proxy will detect any incorrectly-typed values and report an error. This approach does not, however, work if the value is not already proxied, for example if it is native to typed code.

4.2. Transient. While the guarded strategy is challenging to implement in Python, the implementation of the transient strategy is very straightforward. Figure 5 shows the essence of the transient runtime system; the actual implementation is a bit more complicated for the sake of efficiency, in order to provide more informative runtime errors than simple assertion failures, and to perform the specialized object casts described in Section 3.2.3. The `has_type` function, used by transient's implementation, is shown in Figure 4. The guarded and monotonic approaches depend on the reflection capabilities of the host language. The transient design, on the other hand, is almost embarrassingly simple, and depends on only the ability to check Python types at runtime and check the existence of object fields; it could likely be ported to nearly any language that supports these operations.

4.3. Monotonic. My implementation of the monotonic semantics uses techniques similar to those used by the guarded design, in that it modifies the `__getattribute__` and `__setattr__` methods of objects. In this case, however, these methods are overwritten on the class of the casted object itself, not a proxy. The implementation of the monotonic strategy also installs specialized getters and setters for when the result of the read is statically known to be of a precise static type, and for when it needs to be upcast to some specific imprecise type. For example, an access of the form `o.x` will be rewritten by the cast inserter to `o.__staticgetattr__('x')` if the type of `o` provides `x` with a fully-static type. This is a call to the *original* getter for `o`, which performs no casts or checks. On the other hand, if `o.x` instead has

a fully or partially dynamic type T, the typechecker will rewrite it as `o.__getattr_attype__('x', T)`, which reads the field and then casts the result to T.

Monotonic is not totally free from guarded-style proxies. Although this approach does not use object proxies, it does use proxies to implement function casts. I did not encounter any issues in the case studies that I traced to monotonic function proxies.

4.4. Discussion. When retrofitting gradual typing to an existing language, one's concerns are somewhat different than they might be if gradual typing was integral to the design of the underlying language. Type safety, for example, is not of the same paramount importance, because Python is already safe — an error in the guarded cast implementation may cause unexpected behavior, but it will not lead to a segmentation fault any more than it would have if the program was run without types. On the other hand, by developing the system in this manner I cannot constrain the behavior of Python programs except through runtime enforcement of static types. Behavior that poses a challenge to the system cannot simply be outlawed when that behavior occurs in untyped code, even if it interacts with typed portions of a program.

Previous work involved implementing gradual typing directly in the Jython implementation of Python. We were able to achieve some success in increasing efficiency of typed programs by compiling them into typed Java bytecode. However, the amount of effort it took to make significant modifications to the system made it difficult to use as a platform for prototyping different designs for casts. By taking the source-to-source translation approach with Reticulated, I can rapidly prototype enforcement strategies by simply writing modules that define `cast` and `check` functions, and I can use Python's rich reflection and introspection libraries rather than needing to write lower-level code to achieve the same goals. Reticulated does not depend on the details of any particular implementation of Python, and can be used with both CPython and the alternative runtime PyPy.

5. Conclusions

In this chapter I presented Reticulated Python, a lightweight framework for designing and experimenting with gradually typed dialects of Python. With this system I developed three designs for runtime

enforcement of types. The guarded system implements the design of Siek and Taha [76], while the novel transient enforcement strategy does not require proxies but instead uses additional use-site checking to preserve static type guarantees, and the monotonic approach causes object values to become monotonically more precise such that they are more or equally statically typed than all references to them.

I evaluated these systems by adapting several third party Python programs to use them. By annotating and running these programs using Reticulated, I determined that, while Reticulated's type system is mostly sufficient, much more Python code would be able to be typed statically were the type system to support generics. I also discovered that with the right design choices, including supporting static types for literals and using local, dataflow-based type inference, significant portions of real-world Python programs can be entirely statically typed, and therefore suitable for compiler optimization. I made several alterations to Reticulated's type system based on how it interacted with existing Python patterns, such as modifying casts that check the existence of object members to be catchable by source-program try/except blocks. I also encountered other situations where I had to modify the original program to interact well with the type system, including by replacing object and type identity checks with other operations and preventing lists from varying between different element types. I discovered several potential bugs in the target programs by running them with Reticulated and found that the proxies used by the guarded system cause serious problems because they do not preserve object identity.

CHAPTER 5

"Big Types in Little Runtime:" Open-World Soundness and Collaborative Blame

1. Introduction

Most gradually typed languages operate by translating a surface language program into an underlying target language, which is then executed. In Reticulated Python and similar real-world languages, the target language is a dynamically typed programming language, and gradually typed programs are expected to seamlessly interact with legacy code in the dynamic language that has not been translated. In this chapter, I present a formal treatment of this property, which I refer to as open-world soundness.

In optionally typed languages like TypeScript and Hack, which do not enforce types at runtime when statically typed and dynamically typed code interact, the translation can simply erase the types and no responsibility is placed on target-language programs to supply the translated code with values of the correct type at runtime. This design allows translated programs to straightforwardly interact with target-language programs. However, interoperability is more challenging to achieve for sound gradually typed languages, where runtime enforcement code is inserted to ensure that values flowing into statically typed regions of code satisfy their expected static types. Soundness in these languages is usually shown in a closed world, where the only programs considered are ones that originate in the surface language and are translated and then executed. Open-world soundness extends soundness to translated programs that are embedded in arbitrary target-language code.

In this chapter, I discuss open-world soundness in the context of different runtime enforcement strategy, demonstrating that the traditional guarded strategy fails open-world soundness when the target language is a spartan host (i.e., a language which lacks native runtime support for gradual typing) and that the transient design, described informally in the Chapter 4 and implemented in Reticulated Python,

satisfies it. Having established that the transient strategy supports open-world soundness in a broad range of contexts, I also show an approach to *blame-tracking* for the transient enforcement strategy, an important feature of gradually typed languages that is supported by the guarded and monotonic enforcement strategies. I add this blame-tracking approach to Reticulated's implementation of the transient enforcement strategy.

Moreover, I explore open-world soundness in the context of the *gradual guarantee*, which states that weakening or removing type annotations from a program does not introduce new errors. Languages which support the gradual guarantee allow programmers to evolve their code from untyped scripts to precisely typed programs, and languages that combine open-world soundness and the gradual guarantee let programmers transition projects from dynamic to static at any granularity without needing to change the untyped sections of their code.

In this chapter, I identify the open-world soundness property and prove, for the first time, that it holds for a calculus (using the transient enforcement strategy). However, I hypothesize that open-world soundness holds for some existing gradually-typed languages, including Typed Racket [93, 88], TS* [86] and StrongScript [72]. These designs support some degree of open interaction with dynamically typed code in the target language without error. Unfortunately, many of these systems achieve this interaction by significantly limiting which implicit type conversions are allowed, violating the gradual guarantee.

Typed Racket is aided by Racket's built-in contract support. When functions are exported from Typed Racket into untyped code, they are wrapped with a contract monitor that ensures that interactions between typed and untyped code are sound. Racket is therefore not a spartan host; its features enable Typed Racket to have open-world soundness with the guarded strategy.

Contributions. In this chapter, I explore the design of open-world gradual typing with the gradual guarantee as a guiding principle. Its contributions are:

- I identify and formalize open-world soundness as an important property for integrating gradual typing into existing languages (Section 2.1).

- I demonstrate the difficulties of supporting open-world soundness with the guarded strategy when translating to a language with limited support for proxies, and show how the transient approach recovers it (Section 2.2).
- I discuss the challenge of defining blame tracking for the transient strategy and present a solution (Section 3).
- I define the first formal semantics for the transient strategy as a variant of the gradually typed lambda calculus $\lambda^{\star}_{\rightarrow}$, including the new blame tracking strategy (Section 4).
- I prove that $\lambda^{\star}_{\rightarrow}$ satisfies the Blame Theorem (Section 5.1), Open-World Soundness (Section 5.2), and the Gradual Guarantee (Section 5.3).
- I present experimental results showing that the performance overhead for the transient strategy is "usable" à la Takikawa et al. [90] and avoids the worst-case slowdowns found in Typed Racket (Section 6).

2. Open-World Soundness and Gradual Typing

In this section, we introduce *open-world soundness* for gradual typing, presenting the general idea and then its formalization and proof technique, and discuss how it benefits users. We then contrast approaches to runtime verification for gradual typing in the context of open-world soundness, and show that the guarded design does not support open-world soundness when the target language of the translation is a spartan host.

2.1. Open-World Soundness in a Nutshell. Many gradually typed languages [19, 61, 95, 100] translate programs written in the *source language* into dynamically typed programs in the *target language*. This translation process does not necessarily produce programs that may safely interoperate with untranslated code in the target language without losing the usual type soundness guarantees of gradual typing. This lack of soundness inhibits the implementation and distribution of typed-and-translated libraries, prevents gradually typed programs from using existing, target-language libraries, and misses out on much of the utility of type annotations.

In this work, we propose an additional property, *open-world soundness*: if a program is well-typed and translated from a gradually-typed surface language into an untyped target, it may interoperate with arbitrary untyped code without producing uncaught type errors. If a translation process fulfills open-world soundness, then programmers are free to write their programs in the gradually-typed source language, using untyped third-party libraries and distributing their own code to untyped clients. Open-world soundness is therefore a related property to *full abstraction* for compilers: attackers at the level of the target language cannot violate the guarantees provided by the source language, even if the target language provides significantly more capabilities than the source. Open-world soundness differs, however, from full abstraction in that it only makes this guarantee at the level of the languages' types (and a mapping between the source and target languages' types), rather than their full semantics.

2.1.1. Formalizing Open-World Soundness. To formalize open-world soundness, we must differentiate between terms based on their *origin*: terms e that originate from translated portions of the program are marked $\lozenge$, while program contexts $\mathcal{C}$, which represent target-language code, are marked ⧫. With this distinction in place, we define open-world soundness:

Definition (Open World Soundness). *Suppose e_s is a source expression of type T that translates to term e of the target language. The system fulfills **open-world soundness** if, for any program context $\mathcal{C}$ in the target language, either:*

- *$\mathcal{C}[e]$ reduces to v in zero or more steps for some value v, or*
- *$\mathcal{C}[e]$ diverges, or*
- *$\mathcal{C}[e]$ reduces to a runtime cast error, or*
- *$\mathcal{C}[e]$ reduces to a type error while evaluating a term marked ⧫.*

This definition has the flavor of type soundness with one fundamental difference: instead of prohibiting programs $\mathcal{C}[e]$ in the target language from misbehaving, we ensure that any incorrect behavior is due to the program context $\mathcal{C}$, which may represent a client program interacting with e or the source program e interacting with an untyped library.

2.1.2. Open-World Soundness Helps Programmers. Open-world soundness goes beyond traditional type safety: programmers using gradually typed languages that satisfy open-world soundness can safely

write programs that interact with arbitrary programs in the underlying dynamic language. Moreover, programmers may distribute their translated programs as libraries to users of the underlying language, who will benefit from the improved error detection without having to use—or even know about—the gradually typed source language.

Likewise, open-world soundness benefits users who write their own code with static types to ensure correctness, but nevertheless wish to take advantage of external libraries written in the target language. Open-world soundness guarantees that using such libraries will not cause the typed code to misbehave or produce errors, even in the presence of two-way communication between translated code and the target-language library (e.g. via callbacks).

2.2. Gradual Typing Strategies and Open-World Soundness. We now inspect two runtime enforcement strategies for gradual typing and discuss each in the context of open-world soundness.

2.2.1. The Guarded Strategy Inhibits Open-World Soundness. In the guarded semantics, the traditional approach to gradual typing [75, 48], the programmer annotates part of the program with types and the compiler translates the entire program into a target language. During translation, the types are erased and casts are inserted at every implicit conversion site. At runtime, these casts ensure that values adhere to the specified types, raising runtime errors when they detect type violations. Consider the following example and the translation of its invocation:

```
1 fun filter(f:int→bool, l:list int)→list int.
2   if f(head l)
3   then cons (head l) (filter f (tail l))
4   else filter f (tail l)
5 filter (fun _ (x:⋆)→⋆ . x % 2 = 0) [1,2,3,4]
```

In the guarded strategy, the compiler inserts a cast on the anonymous procedure argument to `filter` from $\star \to \star$ to int $\to$ bool during translation.

```
# Guarded Call Translation
filter ((fun _ (x:⋆)→⋆. x%2 = 0) :: ⋆→⋆⇒^ℓ0(int→bool))
       [1,2,3,4]
# ⟶* [2,4]
```

While casts on first-order values are checked immediately when evaluated at runtime, casts on functions and objects in the guarded strategy install *proxies*: in the example above, the cast from $\star \to \star$ to int $\to$ bool installs a wrapper on the original procedure that, when called, casts the input from int to $\star$, calls the underlying procedure, and then casts the result from $\star$ to bool [35].

Unfortunately, these proxies interfere with open-world soundness on spartan hosts by inhibiting sound module interactions and sound foreign-function calls.

The guarded strategy inhibits sound module interaction. Without modifying the target-language runtime to reason about proxied values, translated programs cannot soundly interact with programs written in the target language. The guarded strategy dictates that *callers*, not *callees*, are responsible for type safety when interacting with typed code. For example, consider the following function definition, which defines `isEven` as a function (also named `isEven`) which takes an integer `n`, and returns whether `n` is even (by checking whether the remainder of $n/2$ is zero).

```
isEven ≜ fun isEven (n:int)→bool. (n % 2) = 0
```

After guarded translation, this program contains no casts; the translation process proved that the output will always be a `bool`, and so the program is provided without additional annotation:

```
## Guarded Translation
isEven ≜ fun isEven n. (n % 2) = 0
```

Unfortunately, while the caller is responsible for invoking `isEven` with an integer, nothing enforces this responsibility. As a result, a program in the target language may incorrectly invoke `isEven`:

```
# Plain Code (Target Language)
isEven("Hi")
# ⟶* Error in (n % 2) = 0:
```

```
4 #         n is a string, expected an integer
```

Under the guarded strategy, the callee fails its responsibility and thus the program yields an error from the internals of the translated program. This error would occur in an untyped version of this program, but is now more difficult to debug: the programmer may incorrectly think that they can trust their type annotations.

The guarded strategy inhibits foreign function calls. Programs translated with the guarded semantics may also misbehave when using foreign functions, including built-in functions provided by the language runtime. For example, consider a program that uses built-in `sum` and `length` functions from the spartan target language, which both expect a list (and are assumed to be type $\star \to \star$ by the translation process) and which do not respect or accept proxies.

```
1 avg      ≜ fun avg (l: list float) →float. (sum l) / (length l)
2 readFile ≜ fun readFile (name : ⋆) →⋆. openAndParse name
3
4 # main
5 avg(readFile "arr")
```

After guarded translation, the program is:

```
1 # Guarded Translation
2 avg      ≜ fun avg l.
3              ((sum (l : list float ⇒^ℓ1 ⋆)) /
4               (length (l : list float ⇒^ℓ2 ⋆))) ::⋆⇒^ℓ3 float
5 readFile ≜ fun readFile name. openAndParse name
6
7 # main
8 avg ((readFile "arr") ::⋆⇒^ℓ0 list float)
9 # ⟶* Error in sum: l is not a list
```

This program yields an error complaining that `l` is not a list: when the argument `l` in `avg` is passed to `sum`, the argument is cast from `list float` to $\star$ (because `sum` is a built-in function assumed to be type

$\star \to \star$.), wrapping `l` in a proxy. The built-in `sum`, however, expects an unproxied list. When it receives a (structurally different) proxied value at runtime, it produces an error. Worse, the programmer may be misled during debugging: `avg` is typed, so they may assume `readFile` (and not the invocation of `sum` itself) is to blame.

This problem can be further exacerbated by objects and classes. For example, in Reticulated Python (which compiles to Python, a spartan host), a list proxy is a subtype of a list (to maintain the behavior of the commonly-used `isinstance` function), causing further misbehavior. The proxied list is a subtype of `list`, which means it contains a private, internal list structure, and when the CPython built-in functions operate over a proxied list, they directly manipulate this internal value (instead of the proxied list, as intended). The result is that calls appear to have no effect, and programmers may be at a loss to explain this behavior.

2.2.2. Transient Semantics Support Open-World Soundness. The transient enforcement strategy eschews proxies in favor of shallow runtime *type checks*—lightweight queries that inspect values' "type tags" [13]—rather than proxy-building casts. During translation, the transient design inserts these checks into function bodies (to defensively ensure its inputs have correct type) and at function call sites (to ensure their results have the expected type).

This type-and-check approach allows the transient strategy to elide proxies, which solves the pointer-identity problems with the guarded semantics as described in Chapter 4 and recovers open-world soundness.

The transient strategy supports sound module interaction. Under the transient strategy, functions are responsible for checking the types of their arguments, function calls are responsible for checking the type of return values, and dereference sites are responsible for checking the type of dereferenced values. As a result, plain programs written in the target language (i.e., Python for Reticulated Python, JavaScript for TypeScript, and Clojure for Typed Clojure) are not responsible for performing any checks themselves, and may invoke their typed-and-translated counterparts and rely on checks within the typed modules to detect and directly report type mismatches to the user. Consider the transient translation of `isEven`, which contains a type check (written $\texttt{n}\Downarrow\langle\ \texttt{int},\ \texttt{isEven},\ \textsc{Arg}\rangle$) to ensure that its input `n` is an integer before executing the function's body:

```
## Transient Translation
isEven ≜ fun isEven n. n⇓⟨int, isEven, ARG⟩; (n % 2) = 0
```

In the case that `n` is not an `int`, the check contains the information to report the error to the user (in this case, `n` was an argument in `isEven`, as indicated by Arg):

```
# Plain Code (Target Language)
isEven("Hi")
# ⟶* Error in isEven:
#         isEven was called with "Hi", which is not an int
```

Under transient semantics, the error correctly indicates that the argument to `isEven` was ill-typed; the user is now guaranteed that once they've added type annotations to a piece of code, it will not misbehave.

The transient strategy supports foreign function calls. Eschewing proxies in the transient strategy enables safe interaction with foreign function calls. Consider the transient translation of the `avg-readFile` example:

```
# Transient Translation
avg      ≜ fun avg l. l⇓⟨list, avg, ARG⟩;
                      (sum (l : list float⇒^ℓ1 ⋆)) /
                      (length (l : list float⇒^ℓ2 ⋆))

readFile ≜ fun readFile name . openAndParse name

(avg ((readFile "arr") ::⋆⇒^ℓ0 list float))⇓⟨float, avg, RES⟩
# ⟶* 47.6
```

Where the guarded strategy installed a proxy around the list `l` before passing it to `sum`, the transient cast at the same place are *only* used to inform blame. As a result, the unproxied list `l` is passed to `sum`, and it behaves correctly. In lieu of the proxy, the *type checks* at the beginning of `avg` and the call site (written `l⇓⟨list,avg,ARG⟩` and `...⇓⟨float,avg,RES⟩`) inspect the top-level type (or type tag) of

each value, ensuring safe interaction and providing the expected type safety to programmers. Moreover, if a built-in or foreign function mutates its input in an ill-typed way, these checks will detect and report the ill-typed value when used.

3. Collaborative Blame

While the transient approach recovers pointer identity and open-world soundness, it does so by removing proxies. In previous work, proxies served as the mechanism for tracking and propagating blame information [35, 104, 93]. The runtime system uses this information when a runtime type error is encountered to report the source of the error—not just the location where the error was discovered and the error was raised, but also which implicit conversion site was violated. This information helps the programmer debug the issue more efficiently.

Traditionally, blame information is propagated through programs at runtime by being included in proxies so that when cast errors occur, the information can be included in errors. Consider the following example, in which *isEven* is cast to $\star \to \star$ and then invoked on a string (all within a gradually typed module):

```
1 ## Guarded Translation
2 isEven ≜ fun isEven n. (n % 2) = 0
3
4 let dyFunc = (isEven ::int→bool ⇒^ℓ0 ⋆→⋆)
5 in dyFunc ("Hi" ::str ⇒^ℓ1 ⋆)
```

When the cast on `isEven` is applied at runtime, the result is a proxy around `isEven` which includes the information that the proxy was created by a cast with the label ℓ_0. When this proxy is applied to the string `"Hi"` (which has been casted to $\star$), it casts the argument to `int`. This cast fails and the resulting error message blames ℓ_0, indicating that the cast `int→bool` $\Rightarrow^{\ell_0} \star \to \star$ is at fault.

Because the transient strategy lacks proxies, it is initially unclear that implementing blame tracking is possible in our system. However, without blame information, the programmer may not have enough

details to properly diagnose errors and, as such, we develop an alternative mechanism to maintain and propagate this information and report blame errors.

3.1. Runtime Blame Management. We solve this problem by tracking blame information in a global *blame map*, updating the relevant blame information at every implicit conversion. We track when values are passed between different static types by statically inserting casts into the program as in the guarded strategy; rather than serving as a type enforcement mechanism, these casts only update the blame map—checks are still the main mechanism for detecting type errors. When a check fails, this blame map is used in conjunction with the type information at the failure site to construct the full error account to the programmer. Furthermore, this construction process provides a *blame history*, indicating the conflicting assumptions that different pieces of the program made to produce this error.

Consider the previous example with `isEven`, now using the transient translation:

```
1 ## Transient Translation
2 isEven ≜ fun isEven n. n⇓⟨int, isEven, ARG⟩; (n % 2) = 0
3
4 let dyFunc = (isEven ::int→bool ⇒^ℓ0 ⋆→⋆)
5 in dyFunc ("Hi" ::str ⇒^ℓ1 ⋆)
```

When the cast ℓ_0 is applied at runtime to `isEven`, the blame map records the cast. Then, when the check at the beginning of `isEven`'s function body detects that `n` is not an integer, the runtime attempts to determine which cast (if any) was responsible by looking up the address of the `isEven` function in the blame map, where it will find the cast `int→bool` $\Rightarrow^{\ell_0} \star\to\star$. Next, the runtime determines if this cast was potentially responsible for the error: the Arg *context tag* at the failed check indicates that it was checking a function argument, and that the cast `int→bool` $\Rightarrow^{\ell_0} \star\to\star$ is unsafe in its argument positions (due to contravariance). Finally, the runtime finds that the actual argument to the function call is `str`, which conflicts with the domain of the function type `int`, and therefore indicates that the `int→bool` $\Rightarrow^{\ell_0} \star\to\star$ cast is at fault.

It is not always possible, however, to go directly from a check failure to an incompatible cast. In the following program, `makeEqChecker` takes a string and returns a function that checks its argument against

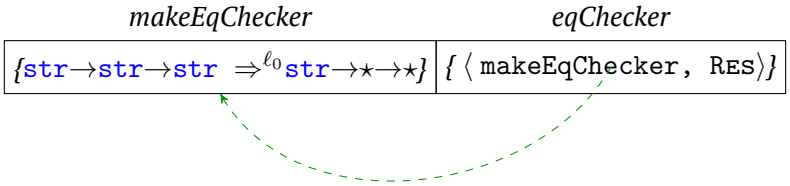


Figure 1. The blame map for `eqChecker` and `makeEqChecker`.

the string. The `makeEqChecker` function is then cast to str $\to \star \to \star$, applied to a string, and then the resulting function is applied to an integer.

```
1 # Transient translation
2 fun makeEqChecker v.
3   v⇓ ⟨ str, makeEqChecker, Arg⟩;
4   fun eqChecker w.
5     w⇓ ⟨ str, eqChecker, Arg⟩;
6     v = w
7
8 let castFunc=(makeEqChecker ::str→str→bool ⇒^{ℓ0} str→⋆→⋆)
9 in ((castFunc "Hi")⇓⟨→, castFunc, Res⟩) (42 ::int⇒^{ℓ1} ⋆)
```

At runtime, a check inside `eqChecker` will detect that 42 is not a string. At this point, the runtime will look up `eqChecker` in the blame map in an attempt to find the responsible cast, but `eqChecker` never passed through a cast; it was implicitly cast as a result of the cast on `makeEqChecker`.

However, there is enough information in the inserted casts and checks to tie the check failure with the cast on `makeEqChecker`: when `makeEqChecker` is applied, a check ensures that the result corresponds with the type tag $\to$. This check updates the blame map before returning the result, adding an internal pointer from the result of the function call (the address of this particular instance of `eqChecker`) to the value that returned it (here `makeEqChecker`). This blame map appears in Figure 1.

When the check fails, the runtime must construct blame information. To do so, it traverses the pointer within the blame map from `eqChecker` to `makeEqChecker`, including the context tag Res that indicates

`eqChecker` is the result of a call to `makeEqChecker`. The runtime uses this data in collaboration with the cast on `makeEqChecker` to discern that the cast is potentially responsible for the check failure, and ultimately blame it.

3.2. Transient Blame is not Guarded Blame. Through use of the blame map, casts and checks collaborate to reconstruct the chains of responsibility that proxies provide with the guarded strategy. Even so, the transient blame behavior differs from that of the guarded strategy: the algorithm may blame multiple casts if each of them is reachable in the blame map and *may* be responsible for the check failure occurring (similar to the behavior of the monotonic approach discussed in Chapter 3). For example, consider the following variation on the `isEven` program above, in which `isEven` is cast *twice* to $\star \to \star$.

```
1 ## Transient Translation
2 isEven ≜ fun isEven n. n⇓⟨int, isEven, ARG⟩; (n % 2) = 0
3
4 let dyFunc1 = (isEven ::int→bool ⇒^ℓ0 ⋆→⋆)
5 in let dyFunc2 = (isEven ::int→bool ⇒^ℓ1 ⋆→⋆)
6 in dyFunc1 ("Hi" ::str ⇒^ℓ2 ⋆)
```

At runtime, the casts on lines 4 and 5 both are recorded in the blame map. When the check in `isEven` detects that an error has occurred, it will find that both casts are potentially at fault. Since the casts both simply return `isEven`, at runtime `dyFunc1` and `dyFunc2` are the same identical value, and the blame tracking system cannot distinguish between them. Therefore, since both casts are unsafe, and both casts could have been responsible for this error, both ℓ_0 and ℓ_1 are blamed. By contrast, in a system using the guarded strategy as defined by Wadler and Findler [104], `dyFunc1` and `dyFunc2` evaluate to separate and distinct proxies, and so when the call at line 6 is evaluated, only ℓ_0 is blamed.

Similarly, the transient strategy only raises errors if ill-typed values are used, and so transient and guarded can blame entirely different casts if multiple errors are present in a program. For example, the following program casts `isEven` to $\star \to \star$ and names the result `dyFunc`, as above. It then casts `dyFunc` to int $\to$ int and names it `badFunc`, then calls `dyFunc` on a string as before, and finally calls `badFunc` on a number.

```
1 ## Transient Translation
2 isEven ≜ fun isEven n. n⇓⟨int, isEven, ARG⟩; (n % 2) = 0
3
4 let dyFunc = (isEven ::int→bool ⇒^ℓ0 ⋆→⋆)
5 in let badFunc = (dyFunc ::⋆→⋆⇒^ℓ1 int→int)
6 in dyFunc ("Hi" ::str ⇒^ℓ2 ⋆);
7 (badFunc 42)⇓⟨bool, badFunc, RES⟩
```

In the transient system, the casts on lines 4 and 5 updated the blame map and return `isEven`. Then the call on line 6 results in a check failure within `isEven`, blaming ℓ_0 as above. With the guarded semantics using the eager strategy of Siek et al. [80], however, the cast on line 5 would fail because it is inconsistent with the previous cast on `dyFunc`, represented at runtime by a proxy. This cast failure would blame ℓ_1—an entirely different result than that produced by the transient strategy.

Despite these differences, the transient blame tracking strategy does result in a system that satisfies the blame-subtyping theorem [104] (as shown in Section 5.1).

4. The Transient Gradual Lambda Calculus $\lambda^{\star}_{\rightarrow}$

In this section, we present the first formal semantics for the transient strategy, including a source-to-target translation, runtime semantics, and a blame system.

We begin with the source language $\lambda^{\star}_{\rightarrow}$ with expressions e_s in Figure 2, which includes variables, recursive functions, mutable references, numbers, and addition. $\lambda^{\star}_{\rightarrow}$ also has types T which range over function types $T_1 \rightarrow T_2$, reference types ref T, integer types int, and the dynamic type $\star$.

Following previous approaches to gradual typing [75, 80, 91, 93], the semantics of $\lambda^{\star}_{\rightarrow}$ are defined by translation into a target language $\lambda^{\Downarrow}_{\ell}$ (as expressions e in Figure 2) which contains type checks as well as the usual type casts. We then define a single-step reduction relation over the $\lambda^{\Downarrow}_{\ell}$ language. Unlike the targets of cast insertion from previous work [75, 48, 104], $\lambda^{\Downarrow}_{\ell}$ is a dynamically typed calculus. Types appear syntactically in the casts of $\lambda^{\Downarrow}_{\ell}$, and shallow type tags S [13] are used in its checks.

$\lambda^\star_\to$ exprs	e_s	::=	$x \mid \texttt{fun}\ f\ (x{:}T) \to T.\ e_s \mid e_s\ e_s \mid \texttt{ref}\ e_s \mid\ !e_s \mid e_s{:=}e_s \mid n \mid e_s\ +\ e_s$
types	T	::=	$\text{int} \mid T \to T \mid \star \mid \text{ref}\ T$
$\lambda^\Downarrow_\ell$ exprs	e	::=	$x \mid v \mid \texttt{fun}\ f\ x.\ e \mid e\ e \mid e\ +\ e \mid \texttt{ref}\ e \mid\ !e \mid e{:=}e \mid e{::}T \Rightarrow^\ell T \mid e \Downarrow \langle S; e; r\rangle$
tags	r	::=	Res | Arg | Deref
type tags	S	::=	$\text{int} \mid \to \mid \text{ref} \mid \star$
addresses	a	$\in$	addresses
labels	ℓ	$\in$	blame labels

Figure 2. Syntax of $\lambda^\star_\to$ and $\lambda^\Downarrow_\ell$.

This section proceeds as follows: we present the transient translation process (§4.1); the runtime semantics for the target language, including blame machinery for the transient strategy (§4.2); and finally present the formal transient blame assignment algorithm (§4.3).

4.1. Translating $\lambda^\star_\to$ to $\lambda^\Downarrow_\ell$. The translation relation, given in Figure 3 with ancillary relations defined in Figure 4, converts a type environment Γ and source term e_s into a target term e at type T, inserting type casts $e{::}T \Rightarrow^\ell T'$ and type checks $e_1 \Downarrow \langle S; e_2; r\rangle$.

The translation proceeds as follows:

- Numbers n and variables x are translated to themselves; numbers have type int and the types of variables are given by Γ.
- Addition translates its operands e_{s1} and e_{s2} into e_1 and e_2 and constructs an output expression that casts e_1 and e_2 to integers with new blame labels ℓ_1 and ℓ_2 before performing addition.
- The translation of a function $\texttt{fun}\ f\ (x{:}T) \to T.\ e_s$ is a $\lambda^\Downarrow_\ell$ function without type annotations. To ensure that the function argument corresponds to the static type T_1 of the original function input, the translation inserts the type check $x \Downarrow \langle T_1; f; \text{Arg}\rangle$ before the translated function body. If this check fails, f indicates the function being eliminated by the call and Arg indicates that the failure was the result of an ill-typed argument.
- Application translates its arguments e_{s1} and e_{s2} into e_1 and e_2, ensures that e_{s1} has type T, and uses the $\triangleright$ relation ("matching") to ensure that T is either dynamic or a function type, which allows

$$\boxed{\Gamma \vdash e_s \rightsquigarrow e : T}$$

$$\frac{}{\Gamma \vdash n \rightsquigarrow n : \mathsf{int}} \qquad \frac{\Gamma(x) = T}{\Gamma \vdash x \rightsquigarrow x : T}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_{s1} \rightsquigarrow e_1 : T_1 \quad T_1 \sim \mathsf{int} \quad \mathsf{fresh}(\ell_1) \\ \Gamma \vdash e_{s2} \rightsquigarrow e_2 : T_2 \quad T_2 \sim \mathsf{int} \quad \mathsf{fresh}(\ell_2)\end{array}}{\Gamma \vdash e_{s1} \ + \ e_{s2} \rightsquigarrow (e_1 :: T_1 \Rightarrow^{\ell_1} \mathsf{int}) + (e_2 :: T_2 \Rightarrow^{\ell_2} \mathsf{int}) : \mathsf{int}}$$

$$\frac{\Gamma, f : T_1 \to T_2, x : T_1 \vdash e_s \rightsquigarrow e' : T_2' \quad T_2 \sim T_2'}{\begin{array}{c}\Gamma \vdash \mathtt{fun}\ f\ (x{:}T_1) \to T_2.\ e_s \rightsquigarrow \\ \mathtt{fun}\ f\ x.\ (\mathtt{let}\ x = x{\Downarrow}\langle \lfloor T_1 \rfloor; f; \mathsf{Arg}\rangle\ \mathtt{in}\ e') : T_1 \to T_2\end{array}}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_{s1} \rightsquigarrow e_1 : T \quad T \rhd T_1 \to T_2 \quad \mathsf{fresh}(f) \\ \Gamma \vdash e_{s2} \rightsquigarrow e_2 : T_1' \quad T_1 \sim T_1' \qquad \mathsf{fresh}(\ell)\end{array}}{\begin{array}{c}\Gamma \vdash e_{s1}\ e_{s2} \rightsquigarrow \mathtt{let}\ f = e_1 :: T \Rightarrow^{\ell} T_1 \to T_2\ \mathtt{in} \\ (f\ (e_2 :: T_1' \Rightarrow^{\ell} T_1)){\Downarrow}\langle \lfloor T_2 \rfloor; f; \mathsf{Res}\rangle : T_2\end{array}}$$

$$\frac{\Gamma \vdash e_s \rightsquigarrow e : T}{\Gamma \vdash \mathtt{ref}\ e_s \rightsquigarrow \mathtt{ref}\ e : \mathsf{ref}\, T}$$

$$\frac{\Gamma \vdash e_s \rightsquigarrow e : T \qquad T \rhd \mathsf{ref}\, T_1 \qquad \mathsf{fresh}(x) \qquad \mathsf{fresh}(\ell)}{\Gamma \vdash !e_s \rightsquigarrow \mathtt{let}\ x = e :: T \Rightarrow^{\ell} \mathsf{ref}\, T_1\ \mathtt{in}\ !x{\Downarrow}\langle \lfloor T_1 \rfloor; x; \mathsf{Deref}\rangle : T_1}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_{s1} \rightsquigarrow e_1 : T \quad T \rhd \mathsf{ref}\, T_1 \quad \mathsf{fresh}(\ell_1) \\ \Gamma \vdash e_{s2} \rightsquigarrow e_2 : T_1' \quad T_1 \sim T_1' \qquad \mathsf{fresh}(\ell_2)\end{array}}{\Gamma \vdash e_{s1} {:=} e_{s2} \rightsquigarrow (e_1 :: T \Rightarrow^{\ell_1} \mathsf{ref}\, T_1) {:=} (e_2 :: T_1' \Rightarrow^{\ell_2} T_1) : \mathsf{int}}$$

Figure 3. Translation from $\lambda^{\star}_{\to}$ to $\lambda^{\Downarrow}_{\ell}$.

$$\boxed{\lfloor T \rfloor = S}$$

$$\lfloor \star \rfloor = \star \qquad \lfloor \text{int} \rfloor = \text{int}$$

$$\lfloor T_1 \to T_2 \rfloor = \to \qquad \lfloor \text{ref}\, T \rfloor = \text{ref}$$

$$\boxed{T \rhd T}$$

$$\text{ref}\, T \rhd \text{ref}\, T \qquad \star \rhd \text{ref}\, \star$$

$$T_1 \to T_2 \rhd T_1 \to T_2 \qquad \star \rhd \star \to \star$$

$$\boxed{T \sim T}$$

$$\text{int} \sim \text{int} \qquad \star \sim T \qquad T \sim \star$$

$$\frac{T_1 \sim T_2}{\text{ref}\, T_1 \sim \text{ref}\, T_2} \qquad \frac{T_1 \sim T_3 \qquad T_2 \sim T_4}{T_1 \to T_2 \sim T_3 \to T_4}$$

Figure 4. Additional relations used in translating from $\lambda^{\star}_{\to}$ to $\lambda^{\Downarrow}_{\ell}$.

it to be broken down into a source type T_1 and a target type T_2. It also ensures that e_{s2} has type T_1' consistent with T_1, and it casts e_1 to type $T_1 \to T_2$ and e_2 to type T_1. The cast of e_1 to $T_1 \to T_2$ does not ensure that the result of the application has type T_2, however, and thus the translation process wraps the application in a check to ensure that the result has the appropriate type. This check includes the information that e_1 (which we let-bind to f to prevent duplicate evaluation) was the function that was applied and that the result Res of the application is being checked.

- References `ref` e_s translate e_s to e and yield `ref` e.
- Reference mutation translates both sides of the assignment. Similar to function application, casts are inserted to ensure that the left hand side is a reference and that the right hand side matches the reference type.
- Dereferences $!e_s$ are translated by translating e_s to e, using a cast to ensure that its type is consistent with type ref T_1, and, finally, placing a check around the dereference which ensures that the reference being eliminated is at type T_1 (where the eliminated value is the reference itself and the context tag Deref indicates dereference).

$$
\begin{array}{lrcl}
\text{evaluation contexts} & E & ::= & \Box \mid E\,e \mid v\,E \mid E\ +\ e \mid v\ +\ E \mid \texttt{ref}\ E \mid !E \mid E{:=}e \mid E{::}T{\Rightarrow^{\ell}}T \\
& & \mid & E{\Downarrow}\langle S; e; r\rangle \mid v{\Downarrow}\langle S; E; r\rangle \\
\\
\text{machine states} & \varsigma & ::= & \langle e, \sigma, \mathcal{B}\rangle \mid \text{Blame}(\mathcal{L}) \\
\\
\text{values} & v & ::= & a \mid n \\
\text{heaps} & \sigma & ::= & \cdot \mid a \mapsto h; \sigma \\
\text{heap values} & h & ::= & (\lambda x.e) \mid v \\
\text{blame sets} & \mathcal{B} & ::= & \cdot \mid a \mapsto \overline{b}; \mathcal{B} \\
\text{blame elems.} & b & ::= & \langle a, r\rangle \mid L \\
\text{labeled types} & L & ::= & \text{int}^{q} \mid L \rightarrow^{q} L \mid \text{ref}^{q}\ L \mid \star \mid \bot^{\ell} \\
\\
\text{label sets} & \mathcal{L} & \subset & \text{blame labels} \\
\text{optional labels} & q & ::= & \ell \mid \epsilon
\end{array}
$$

Figure 5. Syntax for evaluating $\lambda^{\Downarrow}_{\ell}$ machine configurations.

4.2. Reduction Semantics for $\lambda^{\Downarrow}_{\ell}$. The $\lambda^{\Downarrow}_{\ell}$ reduction relation, defined in Figure 6, is a single-step reduction that works over configurations defined in Figure 5. Configurations have the form

$$\langle e, \sigma, \mathcal{B}\rangle$$

where e is an expression, σ is a heap, and $\mathcal{B}$ is the runtime *blame map*, which associates heap addresses with cast information. We update this blame map $\mathcal{B}$ at cast and check sites (using the utility definition ϱ), associating new blame information with heap addresses a. When a cast or check fails, we use this blame map to assign blame.

The reduction relation $\longrightarrow$ has a number of unusual features:

$$\boxed{\langle e, \sigma, \mathcal{B}\rangle \longrightarrow \varsigma}$$

$$\begin{array}{rll}
\langle \texttt{fun}\ f\ x.\ e, \sigma, \mathcal{B}\rangle \longrightarrow \langle a, \sigma[a \mapsto (\lambda x.e[a/f])], \mathcal{B}\rangle & \text{where fresh}(a) \\
\langle a\ v, \sigma, \mathcal{B}\rangle \longrightarrow \langle e[v/x], \sigma, \mathcal{B}\rangle & \text{where } \sigma(a) = (\lambda x.e) \\
\\
\langle \texttt{ref}\ v, \sigma, \mathcal{B}\rangle \longrightarrow \langle a, \sigma[a \mapsto v], \mathcal{B}\rangle & \text{where fresh}(a) \\
\langle !a, \sigma, \mathcal{B}\rangle \longrightarrow \langle v, \sigma, \mathcal{B}\rangle & \text{where } \sigma(a) = v \\
\langle a{:=}v, \sigma, \mathcal{B}\rangle \longrightarrow \langle 0, \sigma[a \mapsto v], \mathcal{B}\rangle & \text{where } \sigma(a) = v' \\
\\
\langle n_1 + n_2, \sigma, \mathcal{B}\rangle \longrightarrow \langle n', \sigma, \mathcal{B}\rangle & \text{where } n' = n_1 + n_2 \\
\\
\langle v{::}T_1{\Rightarrow^{\ell}}T_2, \sigma, \mathcal{B}\rangle \longrightarrow \langle v, \sigma, \varrho(\mathcal{B}, a, [\![T_1 \Rightarrow^{\ell} T_2]\!])\rangle & \text{where } \textit{hastype}(\sigma, v, \lfloor T_2 \rfloor), v = a \\
\langle v{::}T_1{\Rightarrow^{\ell}}T_2, \sigma, \mathcal{B}\rangle \longrightarrow \langle v, \sigma, \mathcal{B}\rangle & \text{where } \textit{hastype}(\sigma, v, \lfloor T_2 \rfloor), v \neq a \\
\langle v{::}T_1{\Rightarrow^{\ell}}T_2, \sigma, \mathcal{B}\rangle \longrightarrow \text{Blame}(\{\ell\}) & \text{where } \neg(\textit{hastype}(\sigma, v, \lfloor T_2 \rfloor)) \\
\\
\langle v{\Downarrow}\langle S; a; r\rangle, \sigma, \mathcal{B}\rangle \longrightarrow \langle v, \sigma, \varrho(\mathcal{B}, a', \langle a, r\rangle)\rangle & \text{where } \textit{hastype}(\sigma, v, S), v = a' \\
\langle v{\Downarrow}\langle S; a; r\rangle, \sigma, \mathcal{B}\rangle \longrightarrow \langle v, \sigma, \mathcal{B}\rangle & \text{where } \textit{hastype}(\sigma, v, S), v \neq a' \\
\langle v{\Downarrow}\langle S; a; r\rangle, \sigma, \mathcal{B}\rangle \longrightarrow \textit{blame}(\sigma, v, a, r, \mathcal{B}) & \text{where } \neg(\textit{hastype}(\sigma, v, S))
\end{array}$$

$$\boxed{\varrho(\mathcal{B}, a, b) = \mathcal{B}} \qquad \varrho(\mathcal{B}, a, b) = \mathcal{B}[a \mapsto \mathcal{B}(a) \cup \{b\}]$$

$$\boxed{\langle e, \sigma, \mathcal{B}\rangle \longmapsto \varsigma}$$

$$\frac{\langle e, \sigma, \mathcal{B}\rangle \longrightarrow \langle e', \sigma', \mathcal{B}'\rangle}{\langle E[e], \sigma, \mathcal{B}\rangle \longmapsto \langle E[e'], \sigma', \mathcal{B}'\rangle} \qquad \frac{\langle e, \sigma, \mathcal{B}\rangle \longrightarrow \text{Blame}(\mathcal{L})}{\langle E[e], \sigma, \mathcal{B}\rangle \longmapsto \text{Blame}(\mathcal{L})}$$

$$\boxed{\textit{hastype}(\sigma, v, S)}$$

$$\frac{}{\textit{hastype}(\sigma, n, \text{int})} \qquad \frac{}{\textit{hastype}(\sigma, v, \star)} \qquad \frac{\sigma(a) = (\lambda x.e)}{\textit{hastype}(\sigma, a, \rightarrow)} \qquad \frac{\sigma(a) = v}{\textit{hastype}(\sigma, a, \text{ref})}$$

Figure 6. Reduction definitions and semantics for the machine configuration for e.

$$\boxed{[\![T \Rightarrow^{\ell} T]\!] = L}$$

$$\begin{array}{rcl}
[\![\star \Rightarrow^{\ell} \star]\!] & = & \star \\
[\![\mathsf{int} \Rightarrow^{\ell} \mathsf{int}]\!] & = & \mathsf{int}^{\epsilon} \\
[\![\mathsf{int} \Rightarrow^{\ell} \star]\!] & = & \mathsf{int}^{\epsilon} \\
[\![\star \Rightarrow^{\ell} \mathsf{int}]\!] & = & \mathsf{int}^{\ell} \\
[\![T_1 \to T_2 \Rightarrow^{\ell} T_3 \to T_4]\!] & = & [\![T_3 \Rightarrow^{\ell} T_1]\!] \to^{\epsilon} [\![T_2 \Rightarrow^{\ell} T_4]\!] \\
[\![T_1 \to T_2 \Rightarrow^{\ell} \star]\!] & = & [\![\star \Rightarrow^{\ell} T_1]\!] \to^{\epsilon} [\![T_2 \Rightarrow^{\ell} \star]\!] \\
[\![\star \Rightarrow^{\ell} T_1 \to T_2]\!] & = & [\![T_1 \Rightarrow^{\ell} \star]\!] \to^{\ell} [\![\star \Rightarrow^{\ell} T_2]\!] \\
[\![\mathsf{ref}\, T_1 \Rightarrow^{\ell} \mathsf{ref}\, T_2]\!] & = & \mathsf{ref}^{\epsilon}\, [\![T_2 \Leftrightarrow^{\ell} T_1]\!] \\
[\![\mathsf{ref}\, T_1 \Rightarrow^{\ell} \star]\!] & = & \mathsf{ref}^{\epsilon}\, [\![\star \Leftrightarrow^{\ell} T_1]\!] \\
[\![\star \Rightarrow^{\ell} \mathsf{ref}\, T_1]\!] & = & \mathsf{ref}^{\ell}\, [\![T_1 \Leftrightarrow^{\ell} \star]\!] \\
[\![T_1 \Rightarrow^{\ell} T_2]\!] & = & \bot^{\ell} \text{ otherwise}
\end{array}$$

$$\boxed{[\![T \Leftrightarrow^{\ell} T]\!] = L}$$

$$\begin{array}{rcl}
[\![\star \Leftrightarrow^{\ell} \star]\!] & = & \star \\
[\![\mathsf{int} \Leftrightarrow^{\ell} \mathsf{int}]\!] & = & \mathsf{int}^{\epsilon} \\
[\![\mathsf{int} \Leftrightarrow^{\ell} \star]\!] & = & \mathsf{int}^{\ell} \\
[\![\star \Leftrightarrow^{\ell} \mathsf{int}]\!] & = & \mathsf{int}^{\ell} \\
[\![T_1 \to T_2 \Leftrightarrow^{\ell} T_3 \to T_4]\!] & = & [\![T_3 \Leftrightarrow^{\ell} T_1]\!] \to^{\epsilon} [\![T_2 \Leftrightarrow^{\ell} T_4]\!] \\
[\![T_1 \to T_2 \Leftrightarrow^{\ell} \star]\!] & = & [\![\star \Leftrightarrow^{\ell} T_1]\!] \to^{\ell} [\![T_2 \Leftrightarrow^{\ell} \star]\!] \\
[\![\star \Leftrightarrow^{\ell} T_1 \to T_2]\!] & = & [\![T_1 \Leftrightarrow^{\ell} \star]\!] \to^{\ell} [\![\star \Leftrightarrow^{\ell} T_2]\!] \\
[\![\mathsf{ref}\, T_1 \Leftrightarrow^{\ell} \mathsf{ref}\, T_2]\!] & = & \mathsf{ref}^{\epsilon}\, [\![T_2 \Leftrightarrow^{\ell} T_1]\!] \\
[\![\mathsf{ref}\, T_1 \Leftrightarrow^{\ell} \star]\!] & = & \mathsf{ref}^{\ell}\, [\![\star \Leftrightarrow^{\ell} T_1]\!] \\
[\![\star \Leftrightarrow^{\ell} \mathsf{ref}\, T_1]\!] & = & \mathsf{ref}^{\ell}\, [\![T_1 \Leftrightarrow^{\ell} \star]\!] \\
[\![T_1 \Leftrightarrow^{\ell} T_2]\!] & = & \bot^{\ell} \text{ otherwise}
\end{array}$$

Figure 7. Compilation of casts to labeled types

- Functions are not values in this calculus. Evaluating a function yields a fresh heap address pointing to the function's code (with self-references substituted away). Functions are stored in the heap so that every function has a unique address, which is used in blame tracking.
- Function applications look up the callee's code from the heap and then perform β-reduction as usual.
- Cast expressions $v::T_1 \Rightarrow^{\ell} T_2$ check if the value v corresponds to the tag $\lfloor T_2 \rfloor$ (which is the shallow tag corresponding to the type T_2, as defined in Figure 4) using the *hastype* relation (bottom of Figure 6). Evaluation proceeds as follows, based on *hastype*'s result and v's shape:
  - If *hastype*$(\sigma, v, \lfloor T_2 \rfloor)$ and v is not a heap address, v is returned immediately.
  - If *hastype*$(\sigma, v, \lfloor T_2 \rfloor)$ and v *is* a heap address, the blame map $\mathcal{B}$ is updated to record the cast, and v is returned.
  - If the value and the tag do not match, then the result is an error blaming ℓ.

  Casts alter $\mathcal{B}$ by extending the blame information associated with a particular address with a new *labeled type* L (compiled from the casts as shown in Figure 7) that annotates each element in the type structure with an optional label indicating whether or not the associated cast is responsible for introducing that portion of the type [79].
- Check expressions $v \Downarrow \langle S; a; r \rangle$ use the *hastype* relation to compare v and S. If the comparison succeeds, the checked value v is returned unmodified, and if v is higher-order then $\mathcal{B}$ is updated to record the check. Otherwise, the *blame* algorithm (§4.3) is invoked to assign blame. Successful checks on higher order values add blame pointers $\langle a', r \rangle$ to a blame map entry $\mathcal{B}(a)$, where a' is v, the checked value. Cast insertion ensures that a is the value responsible for this check—if the check is on a function call or argument, a is the address of the function, and if it is a check on a dereference, a is the reference. The blame pointer indicates that any casts applied to a' (or any value reachable from a' in $\mathcal{B}$), can potentially affect any future check on a, and may need to be considered when assigning blame. These pointer chains within the blame map allow the runtime system to blame specific casts when a check signals an error.

4.3. Blame Assignment. As previously stated, if the runtime system detects a type violation, it determines which boundary crossing caused the violation to occur and blames it. If a cast fails immediately,

we blame the cast itself. If, however, we detect the violation through a check, determining the guilty cast is more complicated: the runtime system must determine which cast(s) are responsible for the failure, and blame each cast that could potentially be responsible for the error that occurred. For example, consider the following code, in which a string reference is casted and passed into two dynamically typed functions which both update it with an integer. The original $\lambda^{\star}_{\rightarrow}$ source is on the left and the $\lambda^{\Downarrow}_{\ell}$ translation is on the right.

$\boxed{\lambda^{\star}_{\rightarrow}}$

```
1 g ≜ fun g (x : ⋆) . x := 42
2 h ≜ fun h (y : ⋆) . y := 21
3 rf ≜ ref "hello"
4 g rf;
5 h rf;
6 !rf
```

$\boxed{\lambda^{\Downarrow}_{\ell}}$

```
1 g ≜ fun g x . x := 42
2 h ≜ fun h y . y := 21
3 rf ≜ ref "hello"
4 g (rf::ref str ⇒^ℓ0 ⋆);
5 h (rf::ref str ⇒^ℓ1 ⋆);
6 (!rf)⇓⟨ str, rf, DEREF⟩
```

When the check on the dereference on the last line of the $\lambda^{\Downarrow}_{\ell}$ code occurs, the result is 21, and thus an error is raised. However, the blame assignment algorithm will report that both casts could potentially have led to the error.

Figure 8 shows the *blame* function (used to allocate blame when a check fails), which takes the following arguments:

- the current heap σ,
- the value v which triggered the check failure,
- the heap address a representing the eliminated value of the triggering check,
- a context tag r indicating what operation triggered the check,
- and the current blame set $\mathcal{B}$.

Given these inputs, the algorithm computes a set of labels ℓ which are collectively responsible for the failure, using helper functions *collectblame* and *resolve*.

The *collectblame* function takes a blame element b—either a labeled type L or an internal pointer—and a list of context tags $\overline{r}$, and proceeds based on the shape of b, collecting blame from the blame set $\mathcal{B}$ as follows:

- If the blame element is a pointer $\langle a, r \rangle$, then the pointer's context tag r is prepended to $\overline{r}$, and *collectblame* is recursively invoked on all the blame elements in $\mathcal{B}(a)$.
- If the blame element is a labeled type L, then the *extract* metafunction uses $\overline{r}$ as a "path" through L to extract some L', a subterm of L. For example,

$$\mathit{extract}(\text{Arg:Res:Deref}, (\text{int}^{\epsilon} \rightarrow^{\ell_2} \text{ref}^{\epsilon}\ \text{int}^{\ell_3}) \rightarrow^{\ell_1} \text{int}^{\epsilon}) = \text{int}^{\ell_3}$$

  according to the cast labeled ℓ_3. If L' does not have a label, it cannot be blame candidate: the cast that introduced the new type did not change this portion of the type. If a label is attached, however, then the cast could have introduced this error and thus L' is included in the "potential blame set" $\overline{L}$.

Once *collectblame* has constructed a set $\overline{L}$ of blame candidates, the *resolve* metafunction compares v—the actual value that triggered the error—with each $L \in \overline{L}$. If L is $\bot^{\ell}$ or if, after L is converted to type tag S as $\lfloor L \rfloor = S$, S is not related to v by *hastype*, the top-level label ℓ attached to L is one of the labels blamed.

For example, in the following program, the constant function `cnst42` is passed into `fn1` and `fn2`.

$\boxed{\lambda^{\star}_{\rightarrow}}$

```
1 fn1 ≜ fun g (x:int→int)→str . x
2 fn2 ≜ fun h (y:str→str)→str . y 21
3 cnst42 ≜ fun cnst42 (n : ⋆) →⋆. 42
4 fn1 cnst42;
5 fn2 cnst42
```

$\boxed{\lambda^{\Downarrow}_{\ell}}$

```
1 fn1 ≜ fun g x . x
2 fn2 ≜ fun h y . (y 21)⇓⟨str, y, Res⟩
3 cnst42 ≜ fun cnst42 n . 42
4 fn1 (cnst42 ::⋆→⋆⇒^ℓ0 int→int);
5 fn2 (cnst42 ::⋆→⋆⇒^ℓ1 str→str)
```

Casts are applied to `cnst42` when it is passed into both `fn`s. Each cast adds an entry in $\mathcal{B}$ to the casts that have been applied to the address of `cnst42`: $\text{int}^{\epsilon} \rightarrow^{\ell_0} \text{int}^{\ell_0}$ for the cast labeled ℓ_0, and $\text{str}^{\epsilon} \rightarrow^{\ell_1} \text{str}^{\ell_1}$ for the cast labeled ℓ_1. Then, when `fn2` applies `cnst42` and expects to receive a `str` as the result, an error is raised because 42 is returned instead. The *collectblame* metafunction looks into the casts that have been applied to `cnst42` in $\mathcal{B}$, and since the check that detected the error was marked with Res, extracts the target type from each of the labeled function types. The result is $\{\text{int}^{\ell_0}, \text{str}^{\ell_1}\}$. The value that actually triggered the error, 42 is related to int by *hastype*, so ℓ_0 cannot be blamed. However, it is not related to str, so the result blames ℓ_1.

5. Blame, Soundness, and the Gradual Guarantee

With our languages, translation, and runtime systems in place, we now present theoretical results for the $\lambda^{\star}_{\rightarrow}/\lambda^{\Downarrow}_{\ell}$ formalism, including the blame theorem (§5.1), open-world soundness (§5.2), and the gradual guarantee (§5.3).

5.1. Blame Theorem. The algorithm that the transient strategy uses for blame tracking and assignment is dramatically different from the techniques in guarded systems. Nonetheless, it still obeys the *blame-subtyping* theorem [104]: programs whose implicit type conversions are safe will never be blamed for cast failures. Specifically, conversions which are upcasts with respect to the blame subtyping relation (Figure 9) will never be blamed.

In $\lambda^{\Downarrow}_{\ell}$ it is insufficient to reason solely about terms because blame information also appears in the blame map $\mathcal{B}$. Therefore we extend the standard safe relation [104, 84, 5, 79] to use $\mathcal{B}$:

$\boxed{\mathit{extract}(\overline{r}, L) = L}$

$$
\begin{array}{rcl}
\mathit{extract}(\cdot, L) & = & L \\
\mathit{extract}((\text{Res} : \overline{r}), L_1 \to^q L_2) & = & \mathit{extract}(\overline{r}, L_2) \\
\mathit{extract}((\text{Arg} : \overline{r}), L_1 \to^q L_2) & = & \mathit{extract}(\overline{r}, L_1) \\
\mathit{extract}((\text{Deref} : \overline{r}), \text{ref}^q\ L) & = & \mathit{extract}(\overline{r}, L) \\
\mathit{extract}((r : \overline{r}), \star) & = & \star
\end{array}
$$

$\boxed{\mathit{label}(L) = q}$

$$
\begin{array}{rcl}
\mathit{label}(\star) & = & \epsilon \\
\mathit{label}(\text{int}^q) & = & q \\
\mathit{label}(L_1 \to^q L_2) & = & q \\
\mathit{label}(\text{ref}^q\ L) & = & q \\
\mathit{label}(\bot^\ell) & = & \ell
\end{array}
$$

$\boxed{\lfloor L \rfloor = T}$

$$
\lfloor \star \rfloor = \star \qquad \frac{}{\lfloor \text{int}^q \rfloor = \text{int}} \qquad \frac{\lfloor L_1 \rfloor = T_1 \qquad \lfloor L_2 \rfloor = T_2}{\lfloor L_1 \to^q L_2 \rfloor = T_1 \to T_2} \qquad \frac{\lfloor L \rfloor = T}{\lfloor \text{ref}^q\ L \rfloor = \text{ref}\ T}
$$

$\boxed{\mathit{collectblame}(\overline{r}, \mathcal{B}, b) = \overline{L}}$

$$
\frac{\mathit{extract}(\overline{r}, L) = L' \quad \mathit{label}(L') = \ell}{\mathit{collectblame}(\overline{r}, \mathcal{B}, L) = \{L'\}} \qquad \frac{\mathit{extract}(\overline{r}, L) = L' \quad \mathit{label}(L') = \epsilon}{\mathit{collectblame}(\overline{r}, \mathcal{B}, L) = \emptyset}
$$

$$
\frac{}{\mathit{collectblame}(\overline{r}, \mathcal{B}, \langle a, r \rangle) = \cup_{b \in \mathcal{B}(a)}\ \mathit{collectblame}((r; \overline{r}), \mathcal{B}, b)}
$$

$\boxed{\mathit{resolve}(\sigma, v, \overline{L}) = \mathcal{L}}$

$$
\mathit{resolve}(\sigma, v, (\bot^\ell; \overline{L})) = \ell; \mathit{resolve}(\sigma, v, \overline{L})
$$

$$
\mathit{resolve}(\sigma, v, (L; \overline{L})) = \begin{cases} \mathit{label}(L); \mathit{resolve}(\sigma, v, \overline{L}) & \text{if } \neg \mathit{hastype}(\sigma, v, \lfloor \lfloor L \rfloor \rfloor) \\ \mathit{resolve}(\sigma, v, \overline{L}) & \text{otherwise} \end{cases}
$$

$$
\mathit{resolve}(\sigma, v, \cdot) = \cdot
$$

$\boxed{\mathit{blame}(\sigma, v, a, r, \mathcal{B}) = \varsigma}$

$$
\frac{\overline{L} = \cup_{b \in \mathcal{B}(a)}\ \mathit{collectblame}(r, \mathcal{B}, b) \qquad \mathcal{L} = \mathit{resolve}(\sigma, v, \overline{L})}{\mathit{blame}(\sigma, v, a, r, \mathcal{B}) = \text{Blame}(\mathcal{L})}
$$

Figure 8. The transient blame-assignment algorithm

$$\boxed{T <:_b T}$$

$$\frac{}{\text{int} <:_b \star} \qquad \frac{T_1 \to T_2 <:_b \star \to \star}{T_1 \to T_2 <:_b \star} \qquad \frac{\text{ref}\, T <:_b \text{ref}\, \star}{\text{ref}\, T <:_b \star}$$

$$\frac{}{\text{int} <:_b \text{int}} \qquad \frac{T_3 <:_b T_1 \qquad T_2 <:_b T_4}{T_1 \to T_2 <:_b T_3 \to T_4} \qquad \frac{}{\text{ref}\, T <:_b \text{ref}\, T}$$

Figure 9. Subtyping with respect to blame

Definition 5.1 (Blame Safety for Terms). *A term e is safe with respect to label ℓ under a blame map $\mathcal{B}$, written $\mathcal{B} \vdash e$ safe ℓ, if there are no unsafe-for-ℓ casts are present in e* and *that no reachable unsafe casts have occurred and been stored in $\mathcal{B}$.*

We define this predicate in Figure 6 of Appendix 1, along with similar predicates over L-types, blame elements b, and heaps σ.

To prove the blame theorem, we must also show that the *blame* algorithm presented above does not blame any cast that the program was safe for. We show this using variants of the standard progress and preservation lemmas, showing that safety is preserved by evaluation and that terms that are safe ℓ cannot blame ℓ when evaluated.

Lemma 5.1 (Blame safety progress). *If $\mathcal{B} \vdash e$ safe ℓ and $\mathcal{B} \vdash \sigma$ safe ℓ and $\langle e, \sigma, \mathcal{B}\rangle \longrightarrow \varsigma$, then $\varsigma \neq \text{Blame}(\mathcal{L})$ with $\ell \in \mathcal{L}$.*

Lemma 5.2 (Blame safety preservation). *If $\mathcal{B} \vdash e$ safe ℓ and $\mathcal{B} \vdash \sigma$ safe ℓ and $\langle e, \sigma, \mathcal{B}\rangle \longrightarrow \langle e', \sigma', \mathcal{B}'\rangle$, then $\mathcal{B}' \vdash e'$ safe ℓ and $\mathcal{B}' \vdash \sigma'$ safe ℓ.*

Complete proofs of these lemmas are given in Appendix 2.2. Finally, we state the blame theorem:

Theorem 5.1 (The Blame Theorem). *For any e_s and T, if*

- $\emptyset \vdash e_s \rightsquigarrow e : T$,
- *e contains a subterm e'::$T_1 \Rightarrow^{\ell} T_2$ containing the only occurrence of ℓ in e, and*

$$
\begin{array}{rcl}
e & ::= & x \mid v \mid \texttt{fun}\ f\ x.\ e \mid (e\ e)^p \mid e\ +^p\ e \mid \texttt{ref}\ e \mid\ !e^p \mid e{:=}^p e \mid e{::}T{\Rightarrow}^\ell T \mid e{\Downarrow}\langle T; e; r\rangle \\
p & ::= & \Diamond \mid \blacklozenge \\
\mathcal{C} & ::= & \Box \mid \texttt{fun}\ f\ x.\ \mathcal{C} \mid \mathcal{C}\ e^{\blacklozenge} \mid v\ \mathcal{C}^{\blacklozenge} \mid \ldots
\end{array}
$$

Figure 10. Syntax of $\lambda_\ell^{\Downarrow}$ with origin tracking.

- $T_1 <:_b T_2$,

then $\langle e, \emptyset, \emptyset\rangle \not\longrightarrow^*$ Blame$(\mathcal{L})$ *with* $\ell \in \mathcal{L}$.

The proof of this theorem is given in Appendix 2.2 of Appendix 1.

5.2. Open-World Soundness. Next, we focus on *open-world soundness*, which states that a well-typed term in $\lambda_{\rightarrow}^{\star}$, translated into $\lambda_\ell^{\Downarrow}$, may safely interact with arbitrary $\lambda_\ell^{\Downarrow}$ code.

To prove this property we proceed as follows: first, we introduce *origin tracking* for terms, indicating if a term e is a translated $\lambda_{\rightarrow}^{\star}$ term or a native $\lambda_\ell^{\Downarrow}$ term; then we introduce a type system for $\lambda_\ell^{\Downarrow}$ that uses this origin tracking to ensure that translated terms include the appropriate casts and checks; next, we introduce expression contexts for embedding translated expressions into untyped code; and finally we state the open-world soundness theorem and discuss its proof. The entire proof is in Appendix 2.

Origin tracking for $\lambda_\ell^{\Downarrow}$. To prove that only untranslated code can reach a stuck configuration, we must distinguish between translated and untranslated code. We achieve this by introducing *origin tracking* for $\lambda_\ell^{\Downarrow}$ (similar to the ownership annotations of Dimoulas et al. [28]) and marking the elimination forms of the language (application, addition, dereference, and mutation) with owners. Figure 10 defines this revised version of $\lambda_\ell^{\Downarrow}$ with origin markers p, which ranges over $\Diamond$, for code translated from well-typed $\lambda_{\rightarrow}^{\star}$ terms, and $\blacklozenge$, for code that originates in $\lambda_\ell^{\Downarrow}$ and lacks static types.

With this modified target language, the translation shown in Figure 3 is modified to attach $\Diamond$ markers to translated terms (Figure 1 of Appendix 1. The reduction rules of Figure 6 are unchanged modulo the addition of markers.

$$\boxed{\langle e, \sigma, \mathcal{B}\rangle \text{ stuck } p}$$

$$\frac{}{\langle n\ v^p, \sigma, \mathcal{B}\rangle \text{ stuck } p} \qquad \frac{\sigma(a) = v'}{\langle a\ v^p, \sigma, \mathcal{B}\rangle \text{ stuck } p} \qquad \frac{}{\langle a\ +^p\ v, \sigma, \mathcal{B}\rangle \text{ stuck } p} \qquad \frac{}{\langle n\ +^p\ a, \sigma, \mathcal{B}\rangle \text{ stuck } p}$$

$$\frac{}{\langle !n^p, \sigma, \mathcal{B}\rangle \text{ stuck } p} \qquad \frac{\sigma(a) = (\lambda x.e)}{\langle !a^p, \sigma, \mathcal{B}\rangle \text{ stuck } p}$$

$$\frac{}{\langle n{:=}^p v, \sigma, \mathcal{B}\rangle \text{ stuck } p} \qquad \frac{\sigma(a) = (\lambda x.e)}{\langle a{:=}^p v, \sigma, \mathcal{B}\rangle \text{ stuck } p} \qquad \frac{\langle e, \sigma, \mathcal{B}\rangle \text{ stuck } p}{\langle E[e], \sigma, \mathcal{B}\rangle \text{ stuck } p}$$

Figure 11. Stuck configurations

Next, we define a stuck relation over machine configurations (Figure 11). Configurations are stuck if they cannot be reduced by any evaluation rule, similar to the *faulty expressions* of Wright and Felleisen [107]. The stuck relation also indicates whether a stuck state was caused by a $\Diamond$-marked term or a $\blacklozenge$-marked term. (We will prove below that no $\Diamond$ term is ever stuck.)

Origin-sensitive typing for $\lambda^{\Downarrow}_{\ell}$. In addition to indicating whether a dynamic type error occurred in typed or untyped code, origin markers also let us define a type system for $\lambda^{\Downarrow}_{\ell}$ that places restrictions on translated code. We use this type system to state and prove open-world soundness. This type system relates expressions to type tags S, as defined in Figure 2, which are repurposed as types.

The typing rules are shown in Figure 12, with supplementary rules and relations shown in Figure 13. There are two rules for each marked expression, one for $\Diamond$ and one for $\blacklozenge$.

Each $\blacklozenge$ rule requires that all subexpressions be typed at $\star$ (which may subsume any other type via TSubsump), and thus no programs may be ill-typed unless they contain $\Diamond$-marked expressions. The $\Diamond$ rules are more restrictive and allow us to prove that a $\Diamond$-marked expression is well-typed when it cannot become stuck, and thus $\Diamond$ rules require that subexpressions have the appropriate types to ensure $\longrightarrow$ reducibility.

Finally, TApp and TDeref are judged to have type $\star$. If the result of such an expression is expected to have a more specific type like $\rightarrow$ or ref, then a check has to be inserted around the expression. Checks

$$\boxed{\Gamma;\Sigma \vdash e : S}$$

(TVar) $$\frac{\Gamma(x) = S}{\Gamma;\Sigma \vdash x : S}$$ (TAddr) $$\frac{\Sigma(a) = S}{\Gamma;\Sigma \vdash a : S}$$ (TInt) $$\frac{}{\Gamma;\Sigma \vdash n : \text{int}}$$ (TSubsump) $$\frac{\Gamma;\Sigma \vdash e : S}{\Gamma;\Sigma \vdash e : \star}$$

(TCheck) $$\frac{\Gamma;\Sigma \vdash e_1 : \star \quad \Gamma;\Sigma \vdash e_2 : \mathit{tagtype}(r)}{\Gamma;\Sigma \vdash e_1 \Downarrow \langle S; e_2; r\rangle : S}$$ (TCast) $$\frac{\Gamma;\Sigma \vdash e : \lfloor T_1 \rfloor \quad T_1 \sim T_2}{\Gamma;\Sigma \vdash e :: T_1 \Rightarrow^{\ell} T_2 : \lfloor T_2 \rfloor}$$

(TFun) $$\frac{\Gamma, x : \star, f :\rightarrow;\Sigma \vdash e : \star}{\Gamma;\Sigma \vdash \texttt{fun}\ f\ x.\ e :\rightarrow}$$ (TLet) $$\frac{\Gamma;\Sigma \vdash e_1 : S_1 \quad \Gamma, x : S;\Sigma \vdash e_2 : S_2}{\Gamma;\Sigma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : S_2}$$

(TApp) $$\frac{\Gamma;\Sigma \vdash e_1 :\rightarrow \quad \Gamma;\Sigma \vdash e_2 : \star}{\Gamma;\Sigma \vdash e_1\ e_2^{\Diamond} : \star}$$ (TApp-$\star$) $$\frac{\Gamma;\Sigma \vdash e_1 : \star \quad \Gamma;\Sigma \vdash e_2 : \star}{\Gamma;\Sigma \vdash e_1\ e_2^{\blacklozenge} : \star}$$

(TRef) $$\frac{\Gamma;\Sigma \vdash e : \star}{\Gamma;\Sigma \vdash \texttt{ref}\ e : \text{ref}}$$ (TDeref) $$\frac{\Gamma;\Sigma \vdash e : \text{ref}}{\Gamma;\Sigma \vdash !e^{\Diamond} : \star}$$ (TDeref-$\star$) $$\frac{\Gamma;\Sigma \vdash e : \star}{\Gamma;\Sigma \vdash !e^{\blacklozenge} : \star}$$

(TUpdtRef) $$\frac{\Gamma;\Sigma \vdash e_1 : \text{ref} \quad \Gamma;\Sigma \vdash e_2 : \star}{\Gamma;\Sigma \vdash e_1 :=^{\Diamond} e_2 : \text{int}}$$ (TUpdtRef-$\star$) $$\frac{\Gamma;\Sigma \vdash e_1 : \star \quad \Gamma;\Sigma \vdash e_2 : \star}{\Gamma;\Sigma \vdash e_1 :=^{\blacklozenge} e_2 : \text{int}}$$

Figure 12. Typing rules for $\lambda_{\ell}^{\Downarrow}$ (excluding addition).

accept expressions of type $\star$ and return the type that the expression is checked against. This design ensures that any $\Diamond$-marked expression uses casts and checks in a defensive way.

Expression contexts for explicit embedding. To reason about code interactions, we use program contexts $\mathcal{C}$ [45], defined in Figure 10. These contexts indicate *how* embedding occurs for translated $\Diamond$ code into $\blacklozenge$ programs, and thus these contexts are all marked with $\blacklozenge$ origin. These contexts are typed in the usual way [45]:

$$\mathcal{C} : \Gamma;S_1 \Rightarrow \Gamma';S_2$$

$$\boxed{\mathit{tagtype}(r) = S}$$

$$\mathit{tagtype}(\text{Arg}) =\!\!\rightarrow \qquad \mathit{tagtype}(\text{Res}) =\!\!\rightarrow \qquad \mathit{tagtype}(\text{Deref}) = \text{ref}$$

$$\boxed{\Sigma \vdash \sigma}$$

$$\text{(THeap)}\ \frac{\forall a \in \mathit{dom}(\Sigma),\ \Sigma \vdash \sigma(a) : \Sigma(a)}{\Sigma \vdash \sigma}$$

$$\boxed{\Sigma \vdash h : S}$$

$$\text{(THRef)}\ \frac{\emptyset; \Sigma \vdash v : \star}{\Sigma \vdash v : \text{ref}} \qquad \text{(THFun)}\ \frac{\emptyset, x{:}\star; \Sigma \vdash e : \star}{\Sigma \vdash (\lambda x.e) :\!\!\rightarrow}$$

$$\boxed{\Sigma \sqsubseteq \Sigma}$$

$$\frac{\forall a \in \mathit{dom}(\Sigma_2),\ \Sigma_1(a) = \Sigma_2(a)}{\Sigma_1 \sqsubseteq \Sigma_2}$$

Figure 13. Additional rules for typing $\lambda^{\Downarrow}_{\ell}$.

The context typing rules are in Figure 5 of Appendix 1.

Open-world soundness. Equipped with origin markers and a type system for $\lambda^{\Downarrow}_{\ell}$, we now state open-world soundness (where $\lfloor \Gamma \rfloor$ is the result of applying $\lfloor T \rfloor$ from Figure 4 to all the types in Γ).

Theorem 5.2 (Open-world soundness). *If* $\Gamma \vdash e_s \rightsquigarrow e : T$ *and* $\vdash \mathcal{C} : \lfloor \Gamma \rfloor; \lfloor T \rfloor \Rightarrow \emptyset; S$*, then* $\emptyset; \emptyset \vdash \mathcal{C}[e] : S$ *and either:*

- $\langle \mathcal{C}[e], \emptyset, \emptyset \rangle \longrightarrow^* \langle v, \sigma, \mathcal{B} \rangle$ *and* $\emptyset; \Sigma \vdash v : S$ *and* $\Sigma \vdash \sigma$*, or*
- $\langle \mathcal{C}[e], \emptyset, \emptyset \rangle \longrightarrow^* \text{Blame}(\mathcal{L})$*, or*
- $\langle \mathcal{C}[e], \emptyset, \emptyset \rangle \longrightarrow^* \langle e', \sigma, \mathcal{B} \rangle$ *and* $\langle e', \sigma, \mathcal{B} \rangle$ *stuck* ⧫*, or*
- *for all* ς *such that* $\langle \mathcal{C}[e], \emptyset, \emptyset \rangle \longrightarrow^* \varsigma$*, have that* $\varsigma = \langle e', \sigma, \mathcal{B} \rangle$ *and exists* ς' *such that* $\langle e', \sigma, \mathcal{B} \rangle \longrightarrow \varsigma'$.

The proof is given in Appendix 2.1. The proof combines a progress and preservation type soundness proof (for $\lambda^{\Downarrow}_{\ell}$) with proofs that the translation relation is type preserving and that the composition of

terms and contexts is well-typed. The progress lemma requires that a configuration with a well-typed term either steps to a new ς or is stuck via a $\blacklozenge$-marked term.

This theorem states that stuck configurations can never arise from evaluating a $\lozenge$-marked term. If $\lozenge$-marked terms could become stuck, it would indicate an uncaught type error in translated $\lambda^{\star}_{\rightarrow}$ code. Type soundness is an immediate corollary of this theorem:

Corollary 5.1 (Type soundness). *If* $\emptyset \vdash e_s \rightsquigarrow e : T$ *then* $\emptyset; \emptyset \vdash e : \lfloor T \rfloor$ *and either:*

- $\langle e, \emptyset, \emptyset \rangle \longrightarrow^* \langle v, \sigma, \mathcal{B} \rangle$ *and* $\emptyset; \Sigma \vdash v : \lfloor T \rfloor$ *and* $\Sigma \vdash \sigma$*, or*
- $\langle e, \emptyset, \emptyset \rangle \longrightarrow^* \text{Blame}(\mathcal{L})$*, or*
- *for all* ς *such that* $\langle e, \emptyset, \emptyset \rangle \longrightarrow^* \varsigma$*, have that* $\varsigma = \langle e', \sigma, \mathcal{B} \rangle$ *and exists* ς' *such that* $\langle e', \sigma, \mathcal{B} \rangle \longrightarrow \varsigma'$.

Proof. By Theorem 5.2, taking $\mathcal{C}$ to be the empty context. □

Ramifications of open-world soundness. Because $\lambda^{\star}_{\rightarrow}$ admits open-world soundness, a program written in $\lambda^{\star}_{\rightarrow}$ can be used by native $\lambda^{\Downarrow}_{\ell}$ clients. For example, an $\lambda^{\star}_{\rightarrow}$ library may put type annotations on its API boundaries, preventing ill-typed terms from raising difficult-to-diagnose errors deep within the library—even if the library interacts with code which has no concept of static types.

Furthermore, the $\lambda^{\star}_{\rightarrow}$ code is protected from errors arising due to mutation: while foreign functions are not modeled directly in the $\lambda^{\star}_{\rightarrow}$ and $\lambda^{\Downarrow}_{\ell}$ calculi, the distinction between untranslated target-language programs and foreign, compiled C code is only relevant with the guarded strategy because of the presence of proxies. The transient design lacks proxies, and thus this distinction is irrelevant—foreign functions may be modeled as native $\lambda^{\Downarrow}_{\ell}$ code.

5.3. The Gradual Guarantee. Finally, we prove that the *gradual guarantee* holds for $\lambda^{\star}_{\rightarrow}$. The gradual guarantee ensures that changing the static type annotations in a program does not alter either the static or dynamic semantics of the program, except by raising a static type error or causing blame at runtime if the type annotations are strengthened [82]. This property allows programmers to be confident in gradually adding types to their program: they know that a program will never produce an entirely different result because of a change to the type annotations. They are also guaranteed that if a program

$$\boxed{T \sqsubseteq T}$$

$$T \sqsubseteq \star \quad \text{int} \sqsubseteq \text{int} \quad \frac{T_1 \sqsubseteq T_2}{\text{ref}\, T_1 \sqsubseteq \text{ref}\, T_2} \quad \frac{T_{11} \sqsubseteq T_{21} \qquad T_{12} \sqsubseteq T_{22}}{T_{11} \to T_{12} \sqsubseteq T_{21} \to T_{22}}$$

Figure 14. Type precision

raises a new error after a type annotation is added or strengthened, it is because the new annotation was "wrong": it did not correspond to other types in the program (if the error is static) or to the program's values at runtime (if it is a runtime blame error).

To prove the gradual guarantee, we use a precision relation for types, also referred to as *naïve subtyping* [79], which is defined in Figure 14. A type T_1 is said to be *more precise* than T_2, written $T_1 \sqsubseteq T_2$, if T_2 contains $\star$ in places where T_1 does not.

We also extend precision to expressions in $\lambda^{\star}_{\to}$. For any two expressions e_{s1}, e_{s2}, we have that $e_{s1} \sqsubseteq e_{s2}$ if the expressions are identical *up to* their type annotations, and if every type annotation in e_{s1} is more precise than the same type annotation in e_{s2}. The specification of expression precision is given in Figure 3 of Appendix 1.

To state the gradual guarantee, we also extend precision to heaps and values. These extensions are straightforward and can be found in Figure 4. With them, we prove the gradual guarantee for $\lambda^{\star}_{\to}$.

Theorem 5.3 (The gradual guarantee). *If $e_s \sqsubseteq e'_s$ and $\emptyset \vdash e_s \rightsquigarrow e : T$, then*

(1) *$\emptyset \vdash e'_s \rightsquigarrow e' : T'$, with $T \sqsubseteq T'$, and*

(2) *if $\langle e, \emptyset, \emptyset\rangle \longrightarrow^* \langle v, \sigma, \mathcal{B}\rangle$, then $\langle e', \emptyset, \emptyset\rangle \longrightarrow^* \langle v', \sigma', \mathcal{B}'\rangle$ with $v \sqsubseteq v'$ and $\sigma \sqsubseteq \sigma'$, and*

(3) *if $\langle e, \emptyset, \emptyset\rangle$ diverges, then $\langle e', \emptyset, \emptyset\rangle$ diverges, and*

(4) *if $\langle e', \emptyset, \emptyset\rangle \longrightarrow^* \langle v', \sigma', \mathcal{B}'\rangle$, then either $\langle e, \emptyset, \emptyset\rangle \longrightarrow^* \langle v, \sigma, \mathcal{B}\rangle$ with $v \sqsubseteq v'$ and $\sigma \sqsubseteq \sigma'$, or $\langle e, \emptyset, \emptyset\rangle \longrightarrow^*$ Blame$(\mathcal{L})$, and*

(5) *if $\langle e', \emptyset, \emptyset\rangle$ diverges, then either $\langle e, \emptyset, \emptyset\rangle$ diverges or $\langle e, \emptyset, \emptyset\rangle \longrightarrow^*$ Blame$(\mathcal{L})$.*

The proof is given in Appendix 2.3. Part 1 is proved by induction on $e_s \sqsubseteq e'_s$. Parts 2 and 3 are proved by first showing that $e \sqsubseteq e'$, and then proving a simulation between the evaluation of e and e'. Parts 4 and 5 are corollaries of parts 2 and 3.

Part 1 of this theorem indicates that a less precise expression will always typecheck successfully and be translated into a $\lambda^{\Downarrow}_{\ell}$ expression if a more precise one does: removing or weakening type annotations will never cause a program to behave worse. Parts 2 and 3 show that the weakening of type annotations from a program can never cause the program to behave differently: if the stronger program diverges, so will the weaker one, and if it returns a result, the weaker one will return a result that is weaker than it. Finally, parts 4 and 5 show that strengthening a program's type annotations can only cause it to behave differently than a weaker one by producing a blame error—if the weaker program diverges, then the stronger one will either go to blame or also diverge, and if the weaker program returns a result, then the stronger one will either go to blame or return a stronger result. Adding type annotations can never, for example, cause a program to diverge if it didn't before.

6. Implementation and Evaluation

In this section, we discuss our implementation of the transient system with blame in Reticulated Python. We discuss the importance of open-world soundness in this setting and then show experimental performance results.

Reticulated Python provides a static typechecker, a source-to-source translator from type-annotated Python-like programs to Python 3 programs with the appropriate casts and checks, and runtime libraries that implement these casts and checks. While Reticulated supports several gradual typing designs, we focus on extending the transient semantics with blame tracking as presented in the previous sections. The implementation of Reticulated v1, as described in Chapter 4, was extended with optional support for blame tracking. During evaluation when using blame tracking, casts are recorded and associated with the casted value in a global map and checks add internal pointers within the map and use the algorithm described in Section 4.3. When an error occurs, the runtime uses this map to identify the responsible parties.

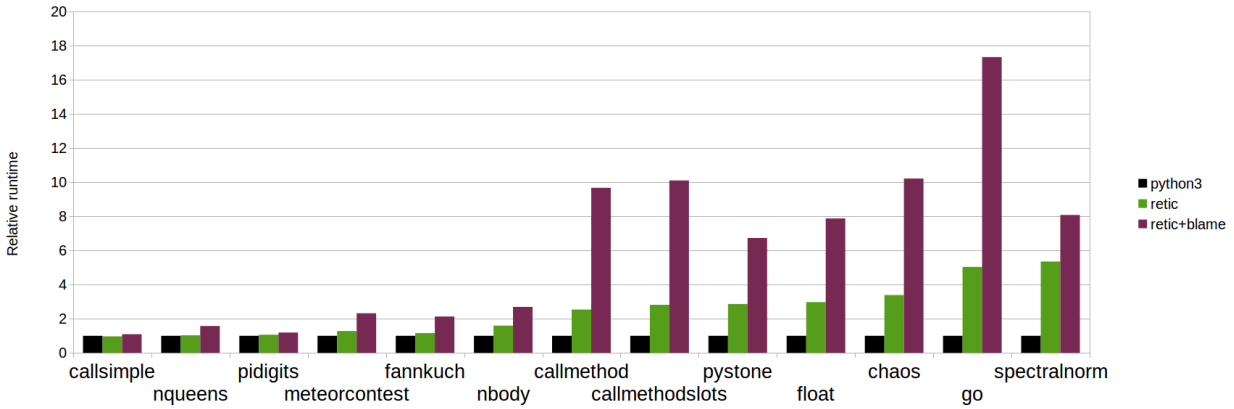


Figure 15. Runtime comparison of Reticulated Python to standard Python 3.4. Experiments were performed on an Ubuntu 14.04 laptop with a 2.8GHz Intel i7-3840QM CPU and 16GB memory.

6.1. Open-World Soundness and Reticulated Python. We conjecture that Reticulated Python is also open-world sound when using the transient semantics; it is certainly closer to open-world soundness than its guarded semantics, and in separate work we proved open-world soundness for Anthill Python, a calculus based on Reticulated and supporting many of Python's features [97]. In Chapter 4 I was able to execute each case study without issue under the transient strategy, but the guarded counterparts required substantial modification to avoid proxy identity problems and, in the case of the CherryPy web framework, the program was unable to run because proxied values could not correctly interact with Python's pickling library. Likewise, the benchmarks that we test with our modified version of Reticulated Python are able to interact with pure Python libraries without errors or incorrect results.

6.2. Performance of Transient Reticulated Python. The pervasive checks that the transient design uses to ensure open-world soundness come at a runtime cost, but our results indicate that the cost is relatively small, especially when blame-tracking features are disabled.

To analyze the runtime performance of the transient strategy, we applied the transient implementation of Reticulated Python to several benchmarks from the official Python benchmark suite.[1] We selected 13 benchmark programs that were compatible with Python 3 and which did not make extensive use of

[1] `https://hg.python.org/benchmarks/`

external libraries. (While Reticulated Python is designed to be open-world sound and therefore allow interaction with external libraries, doing so here would shed little light on the cost of runtime soundness since the bulk of execution time would occur in untranslated code.)

We modified each of these benchmarks, inserting static type annotations wherever possible. In some cases, there were parameters or object fields that were inherently dynamic, or which Reticulated Python's type system is unable to provide static types for. In these cases, we defaulted to the dynamic type. Additionally, in several examples we made minor changes to function bodies to avoid trivial dynamicity—for example, when it couldn't change the semantics of the program, changing a list that was initialized with `None` objects and then filled with `int`s (which could only be typed as `List(⋆)`), to a list initialized with numbers (soundly typed at `List(int)`). We then used Reticulated Python to translate the programs using transient semantics and executed the translated program with standard Python 3.4.

Our experiments consider two versions of Reticulated v1's transient implementation: one using the blame tracking technique described in Section 3 and the non-blame-tracking version described in Chapter 4.

Figure 15 compares the runtime efficiency of the transient translations of the benchmarks with the original untyped code. The green bars show the relative performance of the typed-and-translated programs *without* blame compared to standard Python (the black bars, normalized to 1), while the purple bars show the relative performance of those same programs with blame tracking.

In our tests, the transient system without blame performs at best equally as fast as regular Python, and at worst 5.4x slower. The average slowdown was 2.5x. The test cases that use classes and lists heavily (and thus require checks when members are read from objects or elements read from lists) performed worse than those which primarily used functions and mathematical operators.

The performance degradation exhibited by transient Reticulated Python is significantly less than has been observed in Typed Racket, where in many cases slowdown of over 100x occurs [90]. Transient is usable in contexts where a slowdown of up to $6\times$ is acceptable compared to the performance of the regular Python version.

While this slowdown is not acceptable for all classes of Python programs (where it may be necessary to disable runtime checks for distribution), many other applications are tolerant of this degree of overhead, and in such cases it would be practical to deploy applications with the transient strategy's runtime system in place.

Unsurprisingly, Reticulated Python is less efficient when performing blame tracking. As Figure 15 shows, however, this additional cost is never more than 4x compared to the blame-free version, and never more than 18x compared to the original, untyped code. While substantial, this overhead is not necessarily prohibitive: programmers may still use it for developmental debugging, and in most cases it displays overhead less than $6\times$, and never incurs as much slowdown as Typed Racket does in its worst cases. We envision that a common approach, when blame tracking is too expensive, would be to run programs with blame disabled but then re-enable it when a check failure is detected.

7. Related Work

Open-world soundness. The *execution semantics* of Allende et al. [10], discussed in Chapter 2, Section 1.3.1, is similar to transient in that runtime enforcement code is inserted defensively—functions "protect themselves" by ensuring at their entry points that values being passed into them have the correct type. However, the enforcement strategy actually used at runtime is the guarded strategy, and so is vulnerable to problems of object identity and interactions with foreign functions.

Typed Racket [93, 88] includes first-class classes and strives for open-world interaction between Typed Racket modules and untyped Racket, utilizing Racket's software contract system. While Racket provides robust capabilities for module exports and proxied values, the Typed Racket implementation also faces some of the same problems we address in this paper. First, Racket's native proxies inhibit pointer-based equality checking via `eq?` (though Racket's deep equality operator `equal?` looks through proxies). Second, Racket's runtime system accounts for potential proxies when using built-in operations. As stated previously, this style of support would require modifying the Python runtime, impacting Reticulated Python's portability.

Dimoulas et al. [29] introduce *complete monitoring*, a correctness criterion for contract systems that ensures that each contract violation blaming party k is the result of evaluating a module boundary crossing, where the value crossing the boundary is owned by k. *Open-world soundness* similarly uses origin tracking (applied to reducible expressions, rather than values) to ensure that any stuck state is reached by evaluating untranslated code. Open-world soundness also guarantees that the system will detect any errors original to the gradually-typed program.

The compiler correctness property of *full abstraction* [1, 4, 65] is similar to open-world soundness in that it requires that the behavior of a program in a target language of compilation corresponds to its behavior in the source language. In the case of full abstraction, the guarantee provided is that the behavior of a program component that is observable in the target language is no greater than what is observable in the source [65]. The guarantee provided by open-world soundness is in the same spirit, but only makes a guarantee about the types of observable results, not specific values–open-world soundness does not guarantee semantic preservation, but it does guarantee type soundness.

Alternatives to transient semantics. Other approaches also tackle the problem of object identity: Keil and Thiemann [53] present a solution based on the idea of making proxies transparent with respect to identity and type tests. Optionally typed languages, not having runtime enforcement at all, neatly sidestep this issue.

Contracts. Eiffel [60] first popularized software contracts and the idea of writing programs with pervasive contract checking, and it has inspired a large body of research [35, 28, 5, 53, 29, 34]. This work typically relies on a hybridized guarded/transient approach to verification: functions and objects are wrapped in proxies as they move through contracts [85], but contracts are "defensive": callees are *always* responsible for ensuring their inputs and outputs pass their contracts, and callers are absolved of all *programming* responsibility. The transient approach is more defensive: each callee ensures that its inputs have the correct type, and every translated caller ensures that its outputs have the correct type.

Blame tracking was originally invented for software contracts and has been widely studied in that context [35, 28, 5, 53]. While our approach mirrors the overall blame approaches proposed in these works, gradual typing varies due to its need to enforce global type invariants. Moreover, our values do not carry blame information, and thus we introduce a side-channel, global communication model similar

to approach described by Swords et al. [87], communicating cast expectations to the blame map during execution.

8. Conclusions

We have discussed an important problem in the implementation of sound gradually typed languages: ensuring soundness of typed programs in the presence of unmoderated interaction with untyped, untranslated code. We refer to this problem as *open-world soundness*. We showed that the traditional guarded design for gradual typing, when embedded in a spartan host, inhibits open-world soundness, but that it holds for the transient enforcement strategy. We developed a novel blame tracking technique that does not rely on proxies and is therefore compatible with the transient design, and we showed that the transient design obeys the gradual guarantee, allowing programmers to freely evolve their code between static and dynamic. By evaluating Reticulated Python's transient design and extending it with blame, we showed that the use of the transient design does not sacrifice usable efficiency. We provided the first formal treatment of the transient strategy with the $\lambda^{\star}_{\rightarrow}$ calculus (and its translation target, $\lambda^{\Downarrow}_{\ell}$), and proved open-world soundness, the blame theorem, and the gradual guarantee.

CHAPTER 6

Optimizing and Evaluating Transient Gradual Typing

1. Introduction

In Chapters 4 and 5 I introduced the transient enforcement strategy for gradual typing as a response to challenges faced by the traditional guarded strategy. I also performed an initial performance analysis of the transient strategy as applied to Reticulated Python. In this chapter, I perform a more detailed performance analysis of the transient strategy in Reticulated Python. I show that, when running Reticulated Python and the transient approach on CPython, performance decreases as programs evolve from dynamic to static types, up to a $6\times$ slowdown compared to equivalent Python programs.

To reduce this overhead, I design a static analysis and optimization that removes redundant runtime checks. The optimization employs a static type inference algorithm that solves traditional subtyping constraints and also a new kind of *check constraint*. I implement this optimization on a version of Reticulated Python that focuses exclusively on the transient enforcement strategy: *Reticulated v2*. I evaluate the resulting performance and find that for many programs, the efficiency of partially typed programs is close to their untyped counterparts, removing most of the slowdown of transient checks. Finally, I measure the efficiency of Reticulated Python programs when running on PyPy, a tracing JIT. I find that combining PyPy with this type inference algorithm reduces the overall average overhead to zero.

1.1. Reviewing the Transient Enforcement Strategy. To explain my approach to optimizing the transient enforcement strategy, I will start with a review of an example of the transient strategy as described in Chapter 5. Consider the program in Figure 1a, written in a gradually typed language. Here, the `makeEq` function is a curried equality function on integers with type int $\rightarrow$ int $\rightarrow$ bool. It is called to produce the `eqFive` function on line 12, which is then called at line 14 on the result of calling the `idDyn` function on a string. Because the result of `idDyn` has static type $\star$ (representing the dynamic

```
1 # Gradual surface program
2
3 def idDyn(a:⋆)→⋆: return a
4
5 def makeEq(n:int)→int→bool:
6   def internal(m:int)→bool:
7     return n == m
8   return internal
9
10
11
12 eqFive = makeEq(5)
13 eqFive(20)
14 eqFive(idDyn('hello world'))
```

(a)

```
1 # Program with transient checks
2
3 def idDyn(a): return a
4
5 def makeEq(n):
6   n⇓int
7   def internal(m):
8     m⇓int
9     return n == m
10  return internal
11
12 eqFive = makeEq(5)⇓→
13 eqFive(20)⇓bool
14 eqFive(idDyn('hello world'))⇓bool
```

(b)

Figure 1. Translation to target language using the transient gradual typing approach.

type), this program should pass static typechecking, but a runtime check is needed to detect that at runtime the value being passed into `eqFive` is actually a string and raise an error.

No matter what strategy is used, in a sound gradually typed language the result should be a runtime error. The transient strategy achieves this goal by inserting checks (such as `n⇓int`, which checks that `n` is an integer) as shown in Figure 1b. Type annotations have been erased, and the bodies of `makeEq` and its internal function now contain checks at lines 6 and 8 to ensure that, whatever arguments they are passed are definitely ints. Similarly, the call to `makeEq` at line 12 also contains a check, to ensure that the result of the function call is a function. Note that this check does not ensure that the call returns a value of type int $\rightarrow$ bool—such a check cannot be performed by immediate inspection of the runtime value, but it can verify that the result is a function. It is then up to the function itself to check that

it is only passed ints (as it does with the argument checks discussed above), and additional checks are inserted when `eqFive` is called on lines 13 and 14 to ensure that the result of that call is a bool.

When this program executes, an error will be raised by the check at line 8, because the call to `eqFive` at line 14 passed in a string. This result is expected and correct; if the error had not arisen, there would be an uncaught type error in the body of `makeEq`'s inner function, as a string would inhabit the int-typed variable `m`. This error could then (depending on the semantics of equality testing) lead to a confusing, difficult to debug error. As it is, the programmer is simply informed that a type mismatch occurred and where.

In Chapter 5 I showed that this approach supports the *open-world soundness* property, which states that programs written in a gradually typed language, translated into a dynamic target language, and then embedded in arbitrary code native to that dynamic language, will only "go wrong" due to errors in the native code. The translated, gradually typed program will not be the source of any errors (other than errors caught by transient checks) even in the presence of unmoderated interaction with the "open world."

1.2. Performance of Transient Gradual Typing. Performance is also of critical concern for gradually typed languages. The runtime checks required for sound gradual typing inevitably impose some degree of runtime overhead, but ideally this overhead would be minimized or made up for by type-based compiler optimizations. Since gradual typing is designed to allow programmers to gradually vary their programs between static and dynamic [82], it is also important that adding or removing individual annotations does not dramatically degrade the program's performance. Takikawa et al. [90] examine the performance of Typed Racket with this criterion in mind by studying programs through the lens of a *typing lattice* [89] made up of differently-typed *configurations* of the same program. The top of the lattice is a fully typed configuration of the program and the bottom is unannotated, and incrementally adding types moves up the lattice.

Takikawa et al. show that in Typed Racket, certain configurations result in catastrophic slowdown compared to either the top or bottom configurations. This indicates that the guarded semantics incurs a

substantial cost when interaction between static and dynamic code is frequent. Many of their benchmarks show mean overheads of over 30× and worst cases of over 100×, which "projects an extremely negative image of *sound* gradual typing" [90].

In this work, I aim to establish whether the transient strategy faces the same problem. In Chapter 5 I performed an initial performance evaluation of Reticulated Python benchmarks and found that overheads (compared to an untyped, standard Python version of the same program) ranged from negligible to over a 5× slowdown. However, this analysis was limited to examining a single configuration, the configuration closest to being fully typed. As shown by Takikawa et al. [90], this is insufficient to make a strong claim about the overall performance of Reticulated Python. Additionally, for this work, Reticulated underwent significant engineering changes in order to focus exclusively on the transient enforcement strategy, altered its type system and the meaning of its types as discussed below in Section 2.2. In addition, the implementation of transient checks was made more efficient. The resulting, transient-focused system is referred to as Reticulated v2.

To obtain a clear picture of Reticulated v2's performance, in this chapter I analyze the performance of ten benchmarks across their typing lattices. Since Reticulated Python uses *fine-grained* gradual typing (where the choice to use static types exists on the level of individual identifiers) rather than *coarse-grained* (where the choice is per-module) as is Typed Racket [90], the size of the typing lattice is too large to generate and test every possible configuration. Instead, I generate samples from the lattice by randomly removing type annotations from a fully-typed version of the benchmark, replacing them with the dynamic type and taking care to ensure that each level in the lattice is equally sampled. Each sample is then translated to standard Python 3 using Reticulated Python and executed with CPython, the reference Python runtime.

With this approach, I found that the cost of transient gradual typing increases as the number of type annotations grows. As a program evolves from dynamic to static, its performance linearly degrades, with the worst performance in the most static configurations. This is because each static type annotation induces checks to ensure that values correspond to that type. This is, of course, counter to one hypothetical benefit of static typing—ideally, static types should aid performance, or at least not degrade it. On the other hand, the linear degradation of performance to a worst case 6× overhead means

that the catastrophic configurations encountered in Typed Racket never occur and the cost of adding an individual type annotation to a program is predictable.

1.3. Reducing the Burden of Pervasive Checks. The transient approach inserts checks throughout the program, but not all checks are necessary for the program to be sound because some checks may be redundant and always succeed. To remove unnecessary checks, I perform type inference on the program after checks have been inserted. Our inference algorithm is based on those of Aiken and Fähndrich [6] and Rastogi et al. [67] and uses subtyping constraints as well as new *check constraints*, generated by transient checks. We prove that our algorithm can soundly remove unnecessary checks in a transient calculus similar to that presented in Chapter 5.

I implemented Reticulated v2 to support this optimization and measured its performance, again sampling from the typing lattices at all levels. With redundant checks removed, the linear increase in execution times disappears, resulting in the fully-typed configurations displaying negligible overhead and a 6% average overhead over all sampled configurations.

1.4. Transient Gradual Typing on a Tracing JIT. While this analysis removes many checks statically, the nature of transient checks suggests that they could also be dynamically optimized away by a JIT. Fortunately, there is a tracing JIT for Python 3, PyPy [18]. Reticulated Python compiles to standard Python 3, so it is suitable to use with PyPy. Existing work also suggests that tracing JITs are capable of dramatically improving the performance of gradually typed languages by optimizing away much of the runtime enforcement. Bauman et al. [15] showed that Pycket, a version of Typed Racket that executes on a tracing JIT (which itself uses the same technology as PyPy), worst-case overheads were vastly improved from the results found by Takikawa et al. [90]. Pycket shows a worst-case overhead of no worse than $10.5\times$ in any configuration tested by Bauman et al. [15]; in this chapter I will examine if the same improvements can be derived in Reticulated Python by using PyPy's tracing JIT.

I found that gradually typed programs running on PyPy displayed much less overhead than the same configurations running on CPython—the average overhead over all configurations was 3% with PyPy compared to $2.21\times$ with CPython, suggesting that PyPy is able to optimize away most of the overhead of transient checks. Some benchmarks still incurred a linear increase in time as types were added, but

to a lesser degree than with CPython (with a worst case overhead of 2.61×). By combining PyPy with our type inference optimization, the average overhead was reduced to zero.

1.5. Contributions. In this chapter, I measure the performance of transient gradual typing in Reticulated Python and design techniques to improve it. The contributions of this chapter are:

- I analyze the performance of Reticulated Python programs across their typing lattices, finding an average overhead of 2.21× and much better worst-case performance than Typed Racket (Section 2).
- I develop a type inference optimization for reducing the number of checks needed by the transient approach, and prove it correct (Section 3).
- I implement this optimization in Reticulated Python and show that it reduces the average overhead to just 6% across all typing lattices (Section 4).
- I analyze both the unoptimized and optimized versions of Reticulated when running under a tracing JIT, and find that it performs very well, especially in combination with our optimization (Section 5).

Section 7 discusses related work, and Section 8 concludes.

2. Performance of Transient Gradual Typing

Before investigating approaches to improve the performance of transient gradual typing, we first establish the performance characteristics of Reticulated Python across the typing lattice [90]. A similar performance evaluation of Reticulated Python was previously conducted by Greenman and Migeed [42], wherein they found that "the cost of soundness in Reticulated Python is at most one order of magnitude, and increases linearly with the number of type annotations." In this section, I perform a similar analysis and come to a similar conclusion, which will serve as a starting point for the remainder of this chapter. Conducting the evaluation with the blame tracking system presented in Chapter 5 is important future work.

2.1. Experimental Setup. We selected ten Python 3 programs and translated them to Reticulated Python by inserting type annotations. These benchmarks are mostly drawn from the official Python benchmark suite,[1] with several drawn from the analysis of Takikawa et al. [90] and translated from Racket to Python. In most cases, the resulting Reticulated programs are fully annotated with static types. However, even with a fully annotated program, the Reticulated type checker can assign expressions the type `Any` (the Reticulated Python name for the dynamic type), such as an if-then-else expression where the branches have different types; we did not attempt to guarantee that such cases do not arise.

To examine the typing lattice for a benchmark, we first count the number of type constructors that appear in the program's annotations. We call this the *type weight* of a program. For example, the presence of the type `List[int]` in a program's annotations adds 2 to its weight, and `Callable[[int], bool]` (which is the Reticulated representation of the type int $\rightarrow$ bool) adds 3. We then divide the type weight into a maximum of 100 intervals: a program with a type weight of 300 would have intervals $[0, 3), [3, 6), \ldots, [297, 300)$. Programs with a total type weight of less than 100 naturally have fewer than 100 intervals. For each interval, we randomly erase type annotations and replace them with `Any`, until the program's type weight falls within the interval. This process can "dynamize" types underneath type constructors; both `Any` and `List[Any]` are possible types that could be generated from an original type annotation `List[int]`. Each partially-dynamized program is a configuration from the typing lattice at the level corresponding to its type weight. Ten configurations are generated per interval, plus a single fully-typed configuration consisting of the original program, for a maximum of 1001 configurations. This choice of ten configurations is arbitrary.

Each configuration was executed on an Intel Core i3-4130 CPU with 8GB of RAM running Ubuntu Server 14.04. Configurations were executed repeatedly on CPython 3.4.3 and average runtimes were recorded.

This sampling methodology differs from that used by Greenman and Migeed [42] in that their evaluation is exhaustive for programs with up to 2^{21} total possible configurations, and after that the number of sampled configurations scales in proportion to the number of function and class definitions. Further,

[1] `https://github.com/python/performance`

rather than generating configurations at a per-type-constructor granularity, they generate configurations at the level of individual functions, methods, or sets of class fields. In my evaluation, the number of evaluated configurations scales by type weight, a measure that corresponds to the per-constructor level of granularity, but has a much lower maximum number of configurations as a result of using consumer hardware rather than a computing cluster as a testbed. However, both overall evaluations arrive at similar conclusions, as discussed below.

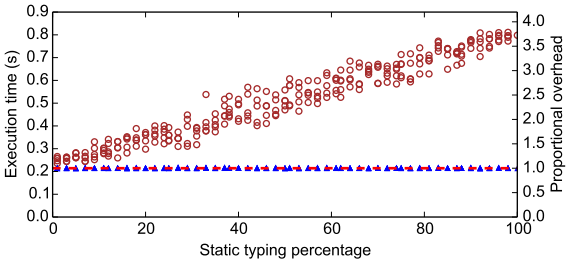
2.2. Results. Figures 2 and 3 show the execution times for configurations across the typing lattice for each benchmark. Each graph corresponds to one benchmark, and each red circle in the graph represents the average execution time of one configuration. The dashed line marks the execution time of the untyped version of the benchmark in standard Python 3. (The blue triangles show the performance of optimized configurations, discussed below in Section 4). Moving from left to right moves up the lattice from untyped to typed, execution time is shown on the left y axis, and relative overhead compared to the untyped program is shown on the right y axis—higher y coordinates indicates slower performance. These results are in concordance with those found by Greenman and Migeed [42]; see especially their Figure 7, which presents the results of a similar analysis. Some differences arise between their evaluation and mine, for example in the `spectralnorm` benchmark; these are likely due to fixed bugs in Reticulated Python, which Greenman and Migeed speculate are the cause of a surprising performance pattern in their evaluation.

Over the entire typing lattices of all benchmarks, Reticulated Python incurs an average overhead of 2.21× compared to the untyped Python versions of the benchmarks. Typically the slowest configurations are the ones with the highest type weight: fully typed configurations have an average overhead of 3.63×. The slowest configuration is from the `nbody` benchmark at 5.95×.

Performance degrades as types are added because changing an annotation from `Any` to a static type results in checks being inserted. The graphs of `pystone`, `snake`, and others display a linear degradation because each check is executed approximately the same number of times when the configuration runs. Graphs with greater variance, such as that of `spectral_norm`, arise when some checks are executed more often than others, so different configurations at the same point in the lattice perform differently

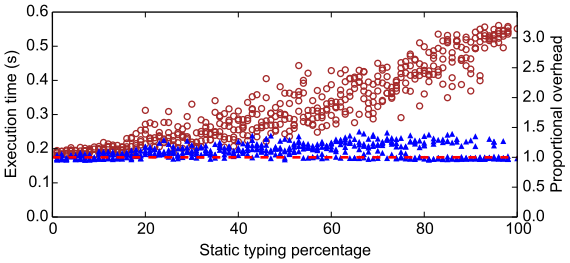
Benchmark: `pystone`

(206 SLoC, 532 configurations)



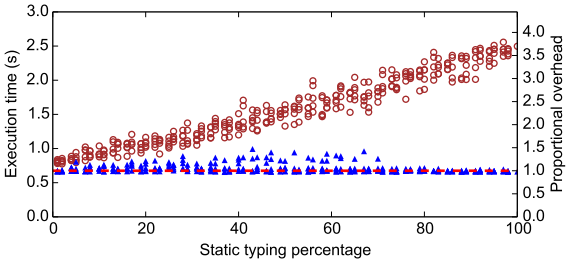
Benchmark: `chaos`

(184 SLoC, 982 configurations)



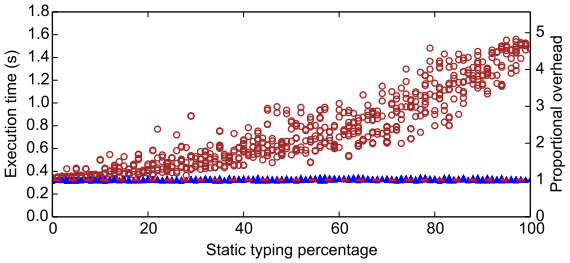
Benchmark: `snake`

(112 SLoC, 662 configurations)



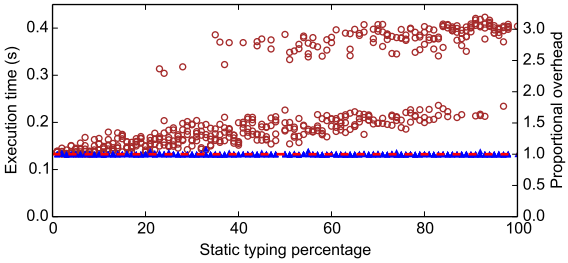
Benchmark: `go`

(394 SLoC, 1001 configurations)



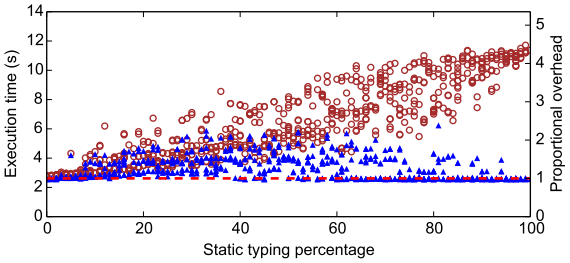
Benchmark: `meteor_contest`

(106 SLoC, 972 configurations)



Benchmark: `suffixtree`

(338 SLoC, 1001 configurations)



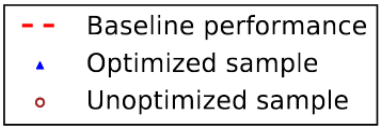


Figure 2. Typing lattices for Reticulated Python benchmarks under CPython (1 of 2).

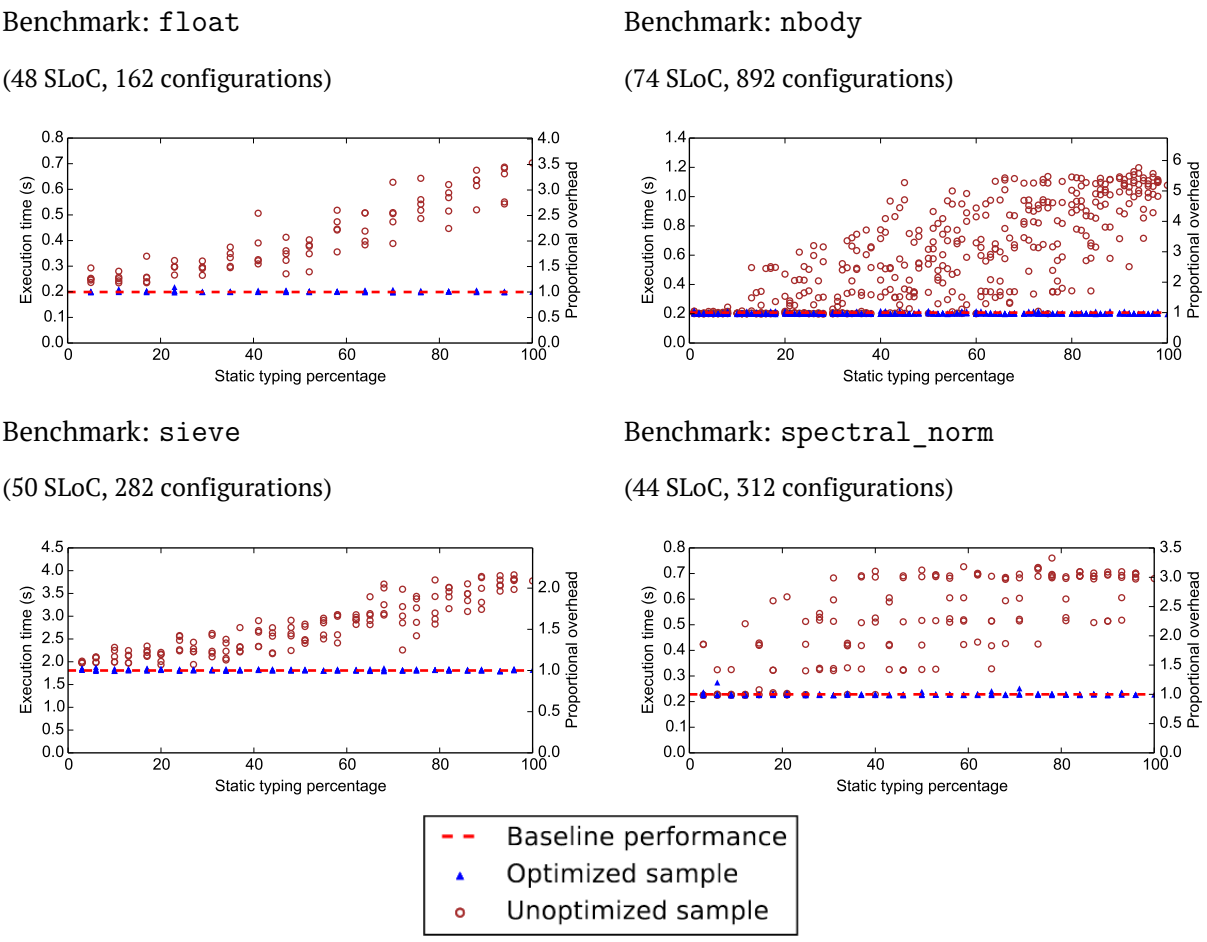


Figure 3. Typing lattices for Reticulated Python benchmarks under CPython (2 of 2).

depending on which annotations are dynamized. The configurations from `meteor_contest` form two large clusters: the parameter `fps` of the `solve` function has type `List[List[List[Set[int]]]]`; `solve` loops over the second and third dimensions of this value and every iteration of a loop includes a check if the loop's target is typed, and so replacing `fps`'s annotation with `Any` or `List[Any]` dramatically reduces the amount of time spent performing transient checks.

In Chapter 5, I examined some of these benchmarks and compared the performance of untyped configurations to configurations that were close to fully-typed. With the analysis presented in this chapter, I find different results in some of these cases: for example, here I show an overhead of 5.19× for the `nbody` benchmark in the fully-typed configuration, while previously it displayed an overhead of less

than 2×. This is because previous work studied Reticulated v1, which differs in important ways from Reticulated v2: Reticulated v2 has a more expressive type system and can handle a fully static version of `nbody` (for example, by allowing type annotations to be placed on functions with default arguments). On the other hand, some of our changes to Reticulated Python's semantics resulted in better performance. For example, Reticulated v2 checks whether an object is an instance of a class rather than checking that it supports all the methods of the class. As a result, the `spectral_norm` benchmark's overhead was reduced from over 5× to 2.98× in fully-typed configurations.

Overall, the performance cost of transient gradual typing in Reticulated Python is significant, but unlike Typed Racket [90], the cost is generally predictable across the lattice and it never approaches worst cases of over 100×. Takikawa et al. [90] suggest that in some contexts an overhead of less than 3× is a cutoff for real-world releasability (with the notation *3-deliverable*) while an overhead in the range 3× to 10× is usable for development purposes (written *3/10-usable*). While they note that such values are "rather liberal" and are unacceptable in many applications, they provide a minimal criterion to evaluate the acceptability of overheads. With Reticulated Python and the baseline transient semantics, the average overhead over every sampled configuration falls within the 3-deliverable range, and all configurations are at least 3/10-usable. This result is in concordance with the analysis of Greenman and Migeed [42], who found an overall performance cost of no worse than one order of magnitude in Reticulated Python programs and also found that the cost of gradual typing increased as programs became more statically-typed, and also corresponds to the experience in Typed Racket reported by Greenman and Felleisen [41].

2.3. Module-Based Configurations. Our sampling methodology is different from the methodology used by Takikawa et al. [90], in which all possible configurations of the tested Typed Racket programs were tested, because the space of possible configurations is much larger in Reticulated Python since Reticulated Python uses fine-grained gradual typing and Typed Racket is coarse-grained. I expect that the fact that dynamic and static are much more intermingled in most of our configurations than they are in Typed Racket would result in our configurations showing even more overhead due to run-time type enforcement. However, to ensure that our sampling did not miss pathological cases that are drawn out by module-level gradual typing, I generated typing lattices equivalent to those that would be generated

`sieve`	(2 modules, 4 config.)	`snake`	(8 modules, 256 config.)
max overhead	2.20×	max overhead	3.81×
mean overhead	1.60×	mean overhead	2.28×

Figure 4. Performance of module-based lattices of `sieve` and `snake`.

by Typed Racket for two of our test cases (cases that Takikawa et al. [90] also analyzed). For these cases, the `sieve` and `snake` benchmarks, we recreated configurations equivalent to those possible in Typed Racket. While the difference in languages precludes a direct comparison, this analysis ensures that if there was some specific interaction between static and dynamic displayed in the Typed Racket configurations that led to a mean overhead of 102.49× (for `sieve`) or 32.30× (for `snake`), I would also encounter it.

Figure 4 shows the performance of the typing lattices of `sieve` and `snake` when generated on a per-module basis *à la* Typed Racket. The performance of these configurations are in line with the overall performance of the benchmarks using our sampling methodology—compare the graphs for `snake` in Figure 2 and `sieve` in Figure 3. In these graphs, the configurations (red circles) that perform worst show approximately the same proportional overhead (the scale shown on the right y-axis) as the worst case configurations of the module-based lattices. Similarly, the mean overheads of the module-based lattices would, if plotted on the graphs in Figures 2 and 3, be close to average. This suggests that the configurations tested by Takikawa et al. [90] are not exceptional cases.

2.4. Comparison to Guarded Gradual Typing. While our performance results compare favorably to those reported by Takikawa et al. [90] in Typed Racket, this is an imprecise comparison because of the different underlying languages involved. As a better comparison between the transient approach and the traditional proxy-based "guarded" approach, I took one configuration of the `sieve` as a case study. Specifically, this was a configuration whose equivalent Typed Racket program showed approximately 100× overhead. For this configuration, I manually created a cast-inserted, proxy-based version of the program. As discussed in Chapters 4 and 5, in general the proxy-based or guarded approach is incompatible with Python, but those incompatibilities do not arise in this limited example.

I found that the performance of the guarded version of the program had an overhead of 11.73× over the standard Python version, compared to an overhead of 1.43× for the transient version. This overhead is significantly lower than that of Typed Racket in this situation, which may be due to the overall performance of the implementations of these languages. Through varying the parameters of the benchmark, I observed that the execution times for guarded `sieve` show the same computational complexity as untyped and transient `sieve`, indicating that the difference in performance is not a result of the use of proxies increasing the complexity of the program. I instrumented the guarded version of the program to verify that there were no chains of proxies—in no case was there a value more than two "layers" deep (i.e. a proxy of a proxy of a value). The high overhead of the guarded approach compared to the transient approach, therefore, is attributable to a large constant factor. Through profiling, I found that the largest contributor to this overhead was the indirection performed by proxies at their use sites, followed by casting, proxy instantiation, and calls from translated user code into the casting code. By contrast, the transient version's reduced overhead comes almost entirely from the runtime inspection of values performed by the code implementing transient checks, and from the calls to these checks. Python performs these inspections very efficiently, and so the overall overhead is relatively low, despite the frequency of the checks.

3. Optimizing Transient Gradual Typing

To improve Reticulated Python's performance, we aim to reduce the number of checks while preserving those required for soundness. The basic idea of transient gradual typing is to use pervasive runtime checks to verify that values correspond to their expected static types. With the transient approach, type annotations are *untrusted*: they do not provide information to be relied on, but rather are claims that must be verified. Therefore, to reduce the runtime burden of transient gradual typing, we move this verification from runtime to compile time wherever possible. We do so by using type inference to determine when types can be *trusted* and do not need runtime verification.

Our inference process is based on the approaches of Aiken and Wimmers [7] and Rastogi et al. [67], using subtyping constraints and also a new form of constraint, the *check constraint*. To determine which checks are redundant, our inference algorithm occurs after transient checks have already been inserted,

because the existence of a check in one part of a program can allow checks elsewhere to be removed. Check constraints let the system reason conditionally about checks, and they express the idea of transient checks: the type of the check expression $e{\Downarrow}S$ and the type of the expression being checked e are constrained to be equal *if*, when solved for, the type of e corresponds to the type tag S (for example, if the type of e is solved to be $\alpha \to \beta$ and S is $\to$). If that is not the case, for example if S is $\to$ and the type of e is solved to be $\star$, then the type of the overall check $e{\Downarrow}S$ is constrained to be the most general type that corresponds with S (in this example, $\star \to \star$).

3.1. Overall approach. We generate sets of check constraints and subtype constraints from programs and find a solution that maps type variables to types, and then remove redundant checks. Our approach is as follows:

- Perform transient check insertion as normal.
- Assign a unique type variable to every function argument, return type, and reference in the program.
- Perform a syntax-directed constraint generation pass.
- Solve the constraint system to obtain a mapping from type variables to types.
- Using this mapping, perform a syntax-directed translation to the final target language. For each check in the program, if the inferred type of the term being checked and the tag it is checked against agree, remove the check, otherwise retain it.

As an example, we return to the program shown in Figure 1a, which shows a curried equality function written in a gradually typed language and which should pass static typechecking but fail due to a transient check at runtime. Figure 5a shows the result of this program after the first phase of our optimizing translation. In this phase, transient checks have been inserted exactly as in Figure 1b, but instead of the programmer's type annotations being erased, they have been replaced by type variables $\alpha, \beta, \gamma, \delta, \epsilon, \zeta$.

Our system infers types (which may be entirely different from the programmer's annotations) for these type variables by generating subtyping constraints and special check constraints. Check constraints are generated by transient checks, and serve to connect the type of the checked expression with the type it is used at after the check. For example, at line 12, the result of `makeEq(5)` has type δ, and is then

```
1 # Program with transient checks and
      vars
2
3 def idDyn(a:α)→β: return a
4
5 def makeEq(n:γ)→δ:
6   n⇓int
7   def internal(m:ϵ)→ζ:
8     m⇓int
9     return n == m
10  return internal
11
12 eqFive = makeEq(5)⇓→
13 eqFive(20)⇓bool
14 eqFive(idDyn('hello world'))⇓bool
```

(a)

```
1 # Final optimized program
2
3 def idDyn(a): return a
4
5 def makeEq(n):
6   def internal(m):
7     m⇓int
8     return n == m
9   return internal
10
11
12 eqFive = makeEq(5)
13 eqFive(20)
14 eqFive(idDyn('hello world'))
```

(b)

Figure 5. Stages of optimized transient compilation for the program shown in Figure 1.

checked to ensure that it is a function ($\Downarrow\!\!\rightarrow$). The type of the result of this check, and therefore the type of `eqFive`, is $\eta \rightarrow \theta$, where η, θ are fresh type variables. This type is linked to δ by a check constraint $(\delta{:}\!\rightarrow) = \eta \rightarrow \theta$, which can be read as "if δ is solved to be a function, then its solution is equal to $\eta \rightarrow \theta$." We use check constraints rather than equality constraints [50, 62] because the same variable can be checked against many different types at different points in the program. Check constraints are only generated by transient checks where the checked type tag corresponds to a non-base type, because constraints of the form $(\gamma{:}\text{int}) = \text{int}$ (as would be generated on line 6) add no new information to the system: the type on the right will be int whether γ is solved to be int or not.

Subtyping constraints are also generated from the program. For example, because the call to `makeEq` on line 12 has an integer argument, it generates the constraint $\text{int} <: \gamma$, meaning that γ is constrained

to be a supertype of int. The full set of constraints for this example is:

$$\{\alpha <: \beta, \epsilon \to \zeta <: \delta, \text{bool} <: \zeta, \text{int} <: \gamma, (\delta{:}\to) = \eta \to \theta, \text{int} <: \eta, \beta <: \eta\}$$

We then solve this constraint set to obtain a mapping from each variable to a single non-variable type. The only subtyping constraint on δ is that $\epsilon \to \zeta <: \delta$, so we determine that δ must be a function and that $\epsilon \to \zeta = \eta \to \theta$. This, combined with the fact that both int and (transitively) str must be subtypes of η due to the calls on lines 13 and 14, means that the only solution for η and ϵ is $\star$ (the dynamic type). In all, the solution we find for this constraint set is

$$\alpha{=}\beta{=}\text{str},\ \gamma{=}\text{int},\ \delta{=}\star{\to}\text{bool},\ \epsilon{=}\eta{=}\star,\ \zeta{=}\theta{=}\text{bool}$$

Some of the transient checks in Figure 5a verify information that the constraint solution has already confirmed. For example, the check at line 6 verifies that `n` is an integer—but `n`'s type γ was statically inferred to be integer, and so this check is not needed. However, the check at line 8, which verifies that `m` is an integer, is not redundant: the type $\star$ was inferred for `m`'s variable ϵ. In fact, this check is needed for soundness because it will fail with a string on line 14. The final program, with redundant checks removed and all annotations erased, is listed in Figure 5b.

For the purposes of this example we do not include constraints based on potential interaction with the open world. If this program were to be visible to the open world and potentially used by untyped Python clients, we would need to generate the additional constraints $\star <: \alpha$ and $\star <: \gamma$ in order to maintain open-world soundness, because the open world could pass arbitrary values into these functions.

3.2. Constraint Generation with Check Constraints. In this section we describe our approach to generating type constraints for programs in a transient calculus $\lambda_d^{\Downarrow}$. Programs in $\lambda_d^{\Downarrow}$ are not the surface programs written by the programmer, and $\lambda_d^{\Downarrow}$ is not a gradually typed language. Instead, $\lambda_d^{\Downarrow}$ programs are the result of translating programs in a gradually typed surface language $\lambda_s^{\star}$ into $\lambda_d^{\Downarrow}$. This translation, which along with the syntax for $\lambda_s^{\star}$ is shown in Figure 6, inserts transient checks throughout translated programs in order to enforce the type annotations present in the surface program, exactly as described in Chapter 5. The $\lambda_d^{\Downarrow}$ calculus is analogous to the cast calculi in guarded gradual typing [75, 104, 48] and the transient translation from $\lambda_s^{\star}$ to $\lambda_d^{\Downarrow}$ is nearly identical to the translation from the surface $\lambda_{\to}^{\star}$

$$
\begin{array}{lrcl}
\text{variables} & x, y & & \text{numbers } n \in \mathbb{Z} \\
\lambda^{\star}_{s} \text{ expressions} & s & ::= & x \mid n \mid s + s \mid \lambda(x{:}U){\to}U.\ s \mid s\ s \mid \mathtt{ref}_U\ s \mid\ !s \mid s{:=}s \\
\lambda^{\Downarrow}_{d} \text{ expressions} & d & ::= & x \mid n \mid d + d \mid \lambda(x{:}\alpha){\to}\alpha.\ d \mid d\ d \mid \mathtt{let}\ x = d\ \mathtt{in}\ d \mid \mathtt{ref}_\alpha\ d \mid \\
 & & & !d \mid d{:=}d \mid d{\Downarrow}S \\
\text{types} & U & ::= & \star \mid \mathsf{int} \mid \mathsf{ref}\,U \mid U \to U \\
\text{type tags} & S & ::= & \star \mid \mathsf{int} \mid\ \to\ \mid \mathsf{ref}
\end{array}
$$

$\boxed{U \sim U}$

$$
\frac{}{\star \sim U} \qquad \frac{}{U \sim \star} \qquad \frac{}{\mathsf{int} \sim \mathsf{int}} \qquad \frac{U_1 \sim U_3 \qquad U_2 \sim U_4}{U_1 \to U_2 \sim U_3 \to U_4} \qquad \frac{U_1 \sim U_2}{\mathsf{ref}\,U_1 \sim \mathsf{ref}\,U_2}
$$

$\boxed{U \rhd U}$

$$
\begin{array}{rcl}
U_1 \to U_2 & \rhd & U_1 \to U_2 \\
\star & \rhd & \star \to \star \\
\mathsf{ref}\,U & \rhd & \mathsf{ref}\,U \\
\star & \rhd & \mathsf{ref}\,\star
\end{array}
$$

$\boxed{\lfloor U \rfloor = S}$

$$
\begin{array}{rcl}
\lfloor U_1 \to U_2 \rfloor & = & \to \\
\lfloor \mathsf{ref}\,U \rfloor & = & \mathsf{ref} \\
\lfloor \mathsf{int} \rfloor & = & \mathsf{int} \\
\lfloor \star \rfloor & = & \star
\end{array}
$$

$\boxed{\Gamma \vdash s \rightsquigarrow d : U}$

$$
\frac{\Gamma, x{:}U_1 \vdash s \rightsquigarrow d : U_2' \qquad U_2' \sim U_2 \qquad \alpha, \beta \text{ fresh}}{\Gamma \vdash \lambda(x{:}U_1){\to}U_2.\ s \rightsquigarrow \lambda(x{:}\alpha){\to}\beta.\ \mathtt{let}\ x = x{\Downarrow}\lfloor U_1 \rfloor\ \mathtt{in}\ d : U_1 \to U_2}
$$

$$
\frac{\Gamma \vdash s_1 \rightsquigarrow d_1 : U \qquad U \rhd U_1 \to U_2 \qquad \Gamma \vdash s_2 \rightsquigarrow d_2 : U_1' \qquad U_1' \sim U_1}{\Gamma \vdash s_1\ s_2 \rightsquigarrow ((d_1{\Downarrow}{\to})\ d_2){\Downarrow}\lfloor U_2 \rfloor : U_2} \qquad \frac{\Gamma(x) = U}{\Gamma \vdash x \rightsquigarrow x : U} \qquad \frac{}{\Gamma \vdash n \rightsquigarrow n : \mathsf{int}}
$$

$$
\frac{\Gamma \vdash s \rightsquigarrow d : U_2 \qquad U_2 \sim U_1 \qquad \alpha \text{ fresh}}{\Gamma \vdash \mathtt{ref}_{U_1}\ s \rightsquigarrow \mathtt{ref}_\alpha\ d : \mathsf{ref}\,U_1} \qquad \frac{\Gamma \vdash s \rightsquigarrow d : U \qquad U \rhd \mathsf{ref}\,U'}{\Gamma \vdash\ !s \rightsquigarrow\ !(d{\Downarrow}\mathsf{ref}){\Downarrow}\lfloor U' \rfloor : U'}
$$

$$
\frac{\Gamma \vdash s_1 \rightsquigarrow d_1 : U \qquad U \rhd \mathsf{ref}\,U' \qquad \Gamma \vdash s_2 \rightsquigarrow d_2 : U'' \qquad U'' \sim U'}{\Gamma \vdash s_1{:=}s_2 \rightsquigarrow (d_1{\Downarrow}\mathsf{ref}){:=}d_2 : \mathsf{int}} \qquad \frac{\Gamma \vdash s_1 \rightsquigarrow d_1 : U_1 \qquad U_1 \sim \mathsf{int} \qquad \Gamma \vdash s_2 \rightsquigarrow d_2 : U_2 \qquad U_2 \sim \mathsf{int}}{\Gamma \vdash s_1\ +\ s_2 \rightsquigarrow d_1{\Downarrow}\mathsf{int}\ +\ d_2{\Downarrow}\mathsf{int} : \mathsf{int}}
$$

Figure 6. The $\lambda^{\star}_{s}$ calculus and translation from $\lambda^{\star}_{s}$ to $\lambda^{\Downarrow}_{d}$.

calculus to the target $\lambda^{\Downarrow}_{\ell}$ calculus presented in Chapter 5 (except that the calculi presented here elide features needed for blame tracking). The $\lambda^{\star}_{s}$ calculus includes functions and mutable references (with syntax $\texttt{ref}_U\ s$ for introducing a reference with type U, $!s$ for dereferencing, and $s{:=}s$ for mutation).

Figure 6 shows the syntax for $\lambda^{\Downarrow}_{d}$, which supports all the features of $\lambda^{\star}_{s}$ as well as transient checks, written $d{\Downarrow}S$. The meta-variable d ranges over expressions of $\lambda^{\Downarrow}_{d}$. In the dynamic semantics for $\lambda^{\Downarrow}_{d}$ (defined by translation into a third calculus $\lambda^{\rightarrow}_{e}$ in Figure 13 with evaluation rules given in Figure 16) such checks examine the value of an expression d to determine if it corresponds to the *type tag* S, and fails if not. Type tags, shown in Figure 6, correspond to type constructors. Functions and references are annotated with type variables.

To infer types for these type variables such that the overall program is well-typed, we first generate constraints using the syntax-directed rules defined in Figure 7, in the style of Aiken and Fähndrich [6]. These rules generate sets Ω of constraints C over types A, also defined in Figure 7. Types A are not inductively defined—function and reference types can only contain type variables or $\star$, not arbitrary types.

The rules in Figure 7 generate constraints. Subtyping constraints are generated from function and reference introduction and elimination sites, to ensure that any solution found for these variables is well-typed.

The rule for transient checks differs from the others. First, there is the question of what constraint type to give the result of a check. Consider the program $\lambda(x{:}\alpha){\rightarrow}\beta.\ (x{\Downarrow}{\rightarrow})$. In the body of the function x has type α, but the type of the check expression $x{\Downarrow}{\rightarrow}$ ought to be a function, because the check will fail at runtime if x is *not* a function. The check cannot, however, specify argument and return types. Therefore the type of $x{\Downarrow}{\rightarrow}$ is a function whose argument and return types are fresh type variables. This type is obtained using the $\rhd_S$ relation in Figure 7, where S is the type tag checked against (in this case $\rightarrow$). If the type on the left already corresponds to S, the type on the right is the same, but if it is a variable the right-hand side is a new type that corresponds to S but is otherwise inhabited by fresh variables.

$$
\begin{array}{rrcl}
\text{type variables} & \alpha, \beta, \gamma & & \\
\text{leaf types} & V & ::= & \alpha \mid \star \\
\text{constraint types} & A & ::= & V \mid \mathsf{int} \mid \mathsf{ref}\, V \mid V \to V \\
\text{constraints} & C & ::= & A <: A \mid (A{:}S) = A \\
\text{constraint sets} & \Omega & \in & \mathcal{P}(C)
\end{array}
$$

$\boxed{A \rhd_S A}$

$$
\begin{array}{rcl}
V_1 \to V_2 & \rhd_{\to} & V_1 \to V_2 \\
\alpha & \rhd_{\to} & \beta \to \gamma \\
 & & \text{with } \beta, \gamma \text{ fresh} \\
\mathsf{ref}\, V & \rhd_{\mathsf{ref}} & \mathsf{ref}\, V \\
\alpha & \rhd_{\mathsf{ref}} & \mathsf{ref}\, \beta \\
 & & \text{with } \beta \text{ fresh} \\
\mathsf{int} & \rhd_{\mathsf{int}} & \mathsf{int} \\
V & \rhd_{\mathsf{int}} & \mathsf{int} \\
A & \rhd_{\star} & A
\end{array}
$$

$\boxed{\Gamma \vdash d : A; \Omega}$

$$\frac{\Gamma \vdash d : A_1; \Omega \qquad A_1 \rhd_S A_2}{\Gamma \vdash d{\Downarrow}S : A_2; \Omega \cup \{(A_1{:}S) = A_2\}} \qquad \frac{\Gamma, x{:}\alpha \vdash d : A; \Omega}{\Gamma \vdash \lambda(x{:}\alpha){\to}\beta.\, d : \alpha \to \beta; \Omega \cup \{A <: \beta\}}$$

$$\frac{\Gamma \vdash d_1 : V_1 \to V_2; \Omega_1 \qquad \Gamma \vdash d_2 : A; \Omega_2}{\Gamma \vdash d_1\, d_2 : V_2; \Omega_1 \cup \Omega_2 \cup \{A <: V_1\}} \qquad \frac{\Gamma \vdash d : A; \Omega}{\Gamma \vdash \mathtt{ref}_{\alpha}\, d : \mathsf{ref}\, \alpha; \Omega \cup \{A <: \alpha\}}$$

$$\frac{\Gamma \vdash d_1 : \mathsf{ref}\, V; \Omega_1 \qquad \Gamma \vdash d_2 : A; \Omega_2}{\Gamma \vdash d_1{:=}d_2 : \mathsf{int}; \Omega_1 \cup \Omega_2 \cup \{A <: V\}} \qquad \frac{\Gamma \vdash d_1 : \mathsf{int}; \Omega_1 \qquad \Gamma \vdash d_2 : \mathsf{int}; \Omega_2}{\Gamma \vdash d_1 \;+\; d_2 : \mathsf{int}; \Omega_1 \cup \Omega_2}$$

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A; \emptyset} \qquad \frac{}{\Gamma \vdash n : \mathsf{int}; \emptyset} \qquad \frac{\Gamma \vdash d : \mathsf{ref}\, V; \Omega}{\Gamma \vdash !d : V; \Omega}$$

Figure 7. Syntax and constraint generation for $\lambda_d^{\Downarrow}$

Checks do not generate subtype constraints: checking that something with type α is a function should not introduce the constraint $\alpha <: \beta \to \gamma$, because the solution to α might not actually be a function. For example, if all values that flow into a variable with type α are integers, then at runtime this check will fail, which is an acceptable behavior for gradually typed programs. However, if only functions inhabit

x, then the argument and return types of those functions must be equal to β and γ respectively. Check expressions instead generate check constraints, written $(A_1{:}S) = A_2$, which constrain the solution for A_2 so that, if the solution of A_1 corresponds to the type tag S, then $A_1 = A_2$.

Note that these type-directed constraint generation rules show that $\lambda_d^{\Downarrow}$ is *not* a gradually typed language: for example, the application rule requires that the expression in function position has a function type, not that it be consistent with a function type. This is because the translation process in Figure 6 has inserted check expressions throughout the program already; any well-typed, closed $\lambda_s^{\star}$ terms can be translated to a well-typed $\lambda_d^{\Downarrow}$ term.

Theorem 6.1. *If* $\emptyset \vdash s \rightsquigarrow d : U$*, then* $\emptyset \vdash d : A; \Omega$.

This lemma is an immediate corollary of Lemma B.4, given in Appendix 2.

3.2.1. Constraints from the open world. The constraint generation system in Figure 7 is sufficient to find a solution if the program does not interact with external code (a *closed* world), but not if the program can interact with code that may not know about or respect the types it expects. Previously, I showed that transient check insertion ensures safety in open world contexts, but optimizing transient programs based only on *internal* information could result in the deletion of checks critical to preserving open-world safety.

Fortunately, Rastogi et al. [67] observed that the overall type of a program will encode the information flows that it exchanges with the open world. Figure 8 shows additional constraints generated from the overall type of a program to protect it from the open world. Constraints such as $\star <: V$ constrain V to be dynamic, while constraints like $V <: \star$ allow V to be more specific types, but guarantee that any type variables that flow through V in a contravariant position will be constrained to $\star$. For example, given the constraints

$$V_1 <: \star,\ V_2 \to V_3 <: \star$$

the rules shown below in Section 3.3 guarantee that V_2 will be constrained to $\star$; this is essential because V_2 is in a contravariant position, and if the a function with type $V_2 \to V_3$ flows into the open world, the open world can pass whatever it wishes into V_2. This also allows our analysis to be modular: individual

$\boxed{\vdash A : \Omega}$

$$\frac{}{\vdash V : \{V <: \star\}} \qquad \frac{}{\vdash \mathsf{int} : \emptyset} \qquad \frac{}{\vdash \mathsf{ref}\, V : \{V <: \star, \star <: V\}} \qquad \frac{}{\vdash V_1 \to V_2 : \{\star <: V_1, V_2 <: \star\}}$$

Figure 8. Open world constraint generation.

$$\begin{array}{rrcl}
\text{types} & T & ::= & T \to T \mid \mathsf{ref}\, T \mid \mathsf{int} \mid \star \mid \alpha \\
\text{constraints} & C & ::= & A <: A \mid (A{:}S) = A \mid A = A \mid \alpha : S \mid \alpha \triangleq T \\
\text{maps} & \sigma & ::= & \alpha \mapsto T, \ldots, \alpha \mapsto T
\end{array}$$

$\boxed{\mathit{parts}(A) = \{\alpha, \ldots, \alpha\}}$

$$\begin{array}{rcl}
\mathit{parts}(\star) & = & \emptyset \\
\mathit{parts}(\mathsf{int}) & = & \emptyset \\
\mathit{parts}(\alpha) & = & \emptyset \\
\mathit{parts}(\alpha_1 \to \alpha_2) & = & \{\alpha_1, \alpha_2\} \\
\mathit{parts}(\mathsf{ref}\, \alpha) & = & \{\alpha\}
\end{array}$$

$\boxed{\lceil S \rceil = A}$

$$\begin{array}{rcl}
\lceil \mathsf{int} \rceil & = & \mathsf{int} \\
\lceil \mathsf{ref} \rceil & = & \mathsf{ref}\, \star \\
\lceil \to \rceil & = & \star \to \star \\
\lceil \star \rceil & = & \star
\end{array}$$

Figure 9. Syntax and utility functions for simplifying constraint sets.

modules can be optimized in isolation by making pessimistic assumptions about what kinds of values flow from one module to another.

3.3. Computing Constraint Solutions. We compute solutions σ for constraint sets Ω with an approach based on the algorithm of Aiken and collaborators [6, 8, 7], but with the addition of check constraints, requiring constraint sets to be solved incrementally.

Figure 10 shows rules for simplifying constraint sets, with syntax and utility functions shown in Figure 9. These rules introduce three additional forms of constraints: *equality constraints* $A_1 = A_2$ indicate equality between A_1 and A_2 [50, 62], *tag constraints* $\alpha : S$ indicate that the solution to the variable α must have the type constructor corresponding to S, and *definition constraints* $\alpha \triangleq T$ constrain the solution of variables α to be exactly T, which is a "full" inductive type as shown in Figure 9.

$\boxed{\Omega \longrightarrow \Omega}$

$$
\begin{array}{rl}
\Omega \cup \{V_1 \to V_2 <: \star\} & \longrightarrow \Omega \cup \{\star <: V_1,\ V_2 <: \star\} \\
\Omega \cup \{V_1 \to V_2 <: V_3 \to V_4\} & \longrightarrow \Omega \cup \{V_3 <: V_1,\ V_2 <: V_4\} \\
\Omega \cup \{\mathsf{ref}\, V <: \star\} & \longrightarrow \Omega \cup \{V = \star\} \\
\Omega \cup \{\mathsf{ref}\, V_1 <: \mathsf{ref}\, V_2\} & \longrightarrow \Omega \cup \{V_1 = V_2\} \\
\Omega \cup \{V <: V\} & \longrightarrow \Omega \\
\Omega \cup \{\mathsf{int} <: \star\} & \longrightarrow \Omega \\
\Omega \cup \{V_1 \to V_2 = V_3 \to V_4\} & \longrightarrow \Omega \cup \{V_1 = V_3, V_2 = V_4\} \\
\Omega \cup \{\mathsf{ref}\, V_1 = \mathsf{ref}\, V_2\} & \longrightarrow \Omega \cup \{V_1 = V_2\} \\
\Omega \cup \{A = A\} & \longrightarrow \Omega \\
\Omega \cup \{A = \alpha\} & \longrightarrow \Omega \cup \{\alpha = A\} \\
 & \quad \text{where} \quad A \neq \alpha' \\
\Omega \cup \{\alpha = A\} & \longrightarrow \Omega[\alpha/A] \cup \{\alpha \triangleq A\} \\
 & \quad \text{where} \quad \alpha \notin \mathit{vars}(A) \text{ and } (\alpha \triangleq T) \notin \Omega \\
\Omega \cup \{(A{:}S) = A\} & \longrightarrow \Omega \\
\Omega \cup \{\alpha : S, (\alpha{:}S) = A\} & \longrightarrow \Omega \cup \{\alpha = A\} \\
 & \quad \text{where} \quad ((\alpha{:}S') = A') \notin \Omega \\
 & \qquad\quad A \neq \alpha \\
\Omega \cup \{\alpha : S_1, (\alpha{:}S_2) = A\} & \longrightarrow \Omega \cup \{\alpha : S_1\} \cup \{\alpha' = \star \mid \forall \alpha' \in \mathit{parts}(A)\} \\
 & \quad \text{where} \quad S_1 \neq S_2 \\
\Omega \cup \{(\alpha{:}S) = A_1, (\alpha{:}S) = A_2\} & \longrightarrow \Omega \cup \{(\alpha{:}S) = A_1, A_2 = A_1\} \\
\Omega \cup \{\alpha : S\} & \longrightarrow \Omega \cup \{\alpha = A\} \\
 & \quad \text{where} \quad ((\alpha{:}S') = A') \notin \Omega \text{ and } (\alpha \triangleq T) \notin \Omega \text{ and} \\
 & \qquad\quad (\alpha = A') \notin \Omega \text{ and } \alpha \rhd_S A
\end{array}
$$

Figure 10. Simplification of constraint sets.

The first six rules decompose subtyping constraints, and are followed by rules for decomposing equalities. Equality constraints on variables $\alpha = A$ immediately become definition constraints $\alpha \triangleq A$

$\boxed{\lfloor A \rfloor = S}$

$$\begin{array}{rcl} \lfloor \text{int} \rfloor & = & \text{int} \\ \lfloor \text{ref}\, V \rfloor & = & \text{ref} \\ \lfloor V_1 \to V_2 \rfloor & = & \to \\ \lfloor \star \rfloor & = & \star \end{array}$$

$\boxed{S \sqcup S = S}$

$$\begin{array}{rcll} S \sqcup S & = & S & \\ S_1 \sqcup S_2 & = & \star & \text{if } S_1 \neq S_2 \end{array}$$

$\boxed{\Omega \Downarrow \sigma}$

$$\frac{\Omega = \Omega' \cup \{A_1 <: \alpha, \ldots, A_n <: \alpha\} \quad \alpha \in \mathit{vars}(\Omega') \vee n > 0 \quad \forall i \leq n,\ A_i \neq \alpha' \wedge \alpha \notin \mathit{vars}(A_i) \quad S = \sqcup_{i \leq n} \lfloor A_i \rfloor \quad \Omega' \text{ solvable } \alpha \quad \Omega \text{ normal} \quad \Omega \cup \{\alpha : S\} \Downarrow \sigma}{\Omega \Downarrow \sigma}$$

$$\frac{\Omega \longrightarrow^* \Omega' \quad \Omega' \Downarrow \sigma}{\Omega \Downarrow \sigma} \qquad \frac{\Omega = \{\alpha_1 \triangleq T_1, \ldots, \alpha_n \triangleq T_n\} \quad \forall \alpha, i \leq n,\ \alpha \notin \mathit{vars}(T_i)}{\Omega \Downarrow \alpha_1 \mapsto T_1, \ldots, \alpha_n \mapsto T_n}$$

$\boxed{\Omega \text{ solvable } \alpha}$

$$\frac{\Omega = \Omega_1 \cup \Omega_2 \cup \Omega_3 \cup \Omega_4 \quad \Omega_2 = \{\alpha_1 \triangleq T_1, \ldots, \alpha_n \triangleq T_n\} \quad \Omega_3 = \{(\alpha{:}S_1) = A_1, \ldots, (\alpha{:}S_m) = A_m\} \quad \Omega_4 = \{\alpha <: V_1, \ldots, \alpha <: V_p\} \quad \alpha \notin \mathit{vars}(\Omega_1) \quad \forall i \leq n,\ \alpha_i \neq \alpha \quad \forall j \leq m,\ \alpha \notin \mathit{vars}(A_j)}{\Omega \text{ solvable } \alpha}$$

Figure 11. Solving constraint sets.

(exploiting the fact that all shallow types A are syntactically also full types T) and result in α being substituted by A in the rest of the constraint set. Tag constraints $\alpha : S_1$ and check constraints $(\alpha{:}S_2) = A$ combine to generate equality constraints for α: if the tag constraint's tag and the check constraint's tag are equal ($S_1 = S_2$) then the constraint $\alpha = A$ is added. Otherwise, the leaves (or *parts*) of A are constrained to be $\star$, indicating that the system was unable to prove that the check that generated the check constraint will succeed. The next rule causes multiple check constraints on the same variable and tag to be combined, and finally, if the system contains a tag constraint but no check constraint on its variable,

$$
\begin{array}{rcl}
a & \in & \text{addresses} \\
e & ::= & a \mid x \mid n \mid e + e \mid \lambda x.\, e \mid (e\, e) \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{ref}\ e \mid\ !e \mid e{:=}e \mid e{\Downarrow}S \mid \texttt{fail} \\
\Sigma & ::= & \cdot \mid \Sigma, a{:}T
\end{array}
$$

Figure 12. Syntax for $\lambda_e^{\rightarrow}$.

the variable is constrained to be a type with a constructor corresponding to S but fresh variables in its leaves.

Figure 11 shows the solution algorithm $\Omega \Downarrow \sigma$. This algorithm applies the simplification rules until exhaustion and then selects an unsolved variable α and determines its tag. The variable to be solved can only appear at the top level of subtyping constraints (e.g., $\alpha \to \beta <: \star \notin \Omega$). Further, α must have only other variables or $\star$ as upper bounds—but this requirement is satisfied by all constraints generated from the rules in Figure 7. These conditions are specified by the Ω solvable α relation in Figure 11. If these conditions hold, α's tag is the join of all the lower bounds of α (using the $\sqcup$ operator in Figure 11). The resulting tag constraint is then added to Ω and the result is simplified. This process terminates once Ω is only inhabited by definition constraints, and results in a substitution σ generated from these constraints.

3.4. Check Removal. When a constraint set Ω is solved by a substitution σ, the types in σ can be relied on because they were inferred directly from the program. They may be more or less precise, or entirely different, from the programmer's annotated types. Figure 13 shows the rules for translating the program to final target language, $\lambda_e^{\rightarrow}$, which is defined in Figure 12. This translation uses σ to decide which checks can be deleted, as shown in rules DCheckRemove, DCheckKeep, and DCheckFail: if the inferred type for the checked expression d is found to be more or equally precise (with the $\preceq$ operator) than the tag it is checked against, then the check can be removed by DCheckRemove. If the type of d is less precise (which is only the case when d's type is $\star$), the check remains by DCheckKeep, and if S and d's type are unrelated, then the check will *always* fail at runtime, so by DCheckFail the check is replaced by `fail`, an expression which errors when evaluated. Since this expression always fails, it would be

$\boxed{\Gamma; \sigma \vdash d \rightsquigarrow e : T}$

DAbs

$$\frac{\Gamma, x{:}\alpha; \sigma \vdash d \rightsquigarrow e : T \qquad \vdash T <: \sigma\beta}{\Gamma; \sigma \vdash \lambda(x{:}\alpha){\to}\beta.\, d \rightsquigarrow \lambda x.\, e : \sigma\alpha \to \sigma\beta}$$

DLet

$$\frac{\Gamma; \sigma \vdash d_1 \rightsquigarrow e_1 : T_1 \qquad \Gamma, x{:}T_1; \sigma \vdash d_2 \rightsquigarrow e_2 : T_2}{\Gamma; \sigma \vdash \texttt{let}\ x = d_1\ \texttt{in}\ d_2 \rightsquigarrow \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : T_2}$$

DApp

$$\frac{\Gamma; \sigma \vdash d_1 \rightsquigarrow e_1 : T_1 \to T_2 \qquad \Gamma; \sigma \vdash d_2 \rightsquigarrow e_2 : T_1' \qquad \vdash T_1' <: T_1}{\Gamma; \sigma \vdash d_1\ d_2 \rightsquigarrow (e_1\ e_2) : T_2}$$

DRef

$$\frac{\Gamma; \sigma \vdash d \rightsquigarrow e : T \qquad \vdash T <: \sigma\alpha}{\Gamma; \sigma \vdash \texttt{ref}_\alpha\ d \rightsquigarrow \texttt{ref}\ e : \mathsf{ref}\,\sigma\alpha}$$

DUpdt

DDeref

$$\frac{\Gamma; \sigma \vdash d \rightsquigarrow e : \mathsf{ref}\,T}{\Gamma; \sigma \vdash !d \rightsquigarrow !e : T} \qquad \frac{\Gamma; \sigma \vdash d_1 \rightsquigarrow e_1 : \mathsf{ref}\,T \qquad \Gamma; \sigma \vdash d_2 \rightsquigarrow e_2 : T' \qquad \vdash T' <: T}{\Gamma; \sigma \vdash d_1{:=}d_2 \rightsquigarrow e_1{:=}e_2 : \mathsf{int}}$$

DVar

$$\frac{\Gamma(x) = A}{\Gamma; \sigma \vdash x \rightsquigarrow x : \sigma A}$$

DInt

$$\frac{}{\Gamma; \sigma \vdash n \rightsquigarrow n : \mathsf{int}}$$

DAdd

$$\frac{\Gamma; \sigma \vdash d_1 \rightsquigarrow e_1 : \mathsf{int} \qquad \Gamma; \sigma \vdash d_2 \rightsquigarrow e_2 : \mathsf{int}}{\Gamma; \sigma \vdash d_1\ +\ d_2 \rightsquigarrow e_1\ +\ e_2 : \mathsf{int}}$$

DCheckRemove

$$\frac{\Gamma; \sigma \vdash d \rightsquigarrow e : T \qquad \lfloor T \rfloor \preceq S}{\Gamma; \sigma \vdash d{\Downarrow}S \rightsquigarrow e : T}$$

DCheckKeep

$$\frac{\Gamma; \sigma \vdash d \rightsquigarrow e : \star \qquad S \neq \star}{\Gamma; \sigma \vdash d{\Downarrow}S \rightsquigarrow e{\Downarrow}S : \lceil S \rceil}$$

DCheckFail

$$\frac{\Gamma; \sigma \vdash d \rightsquigarrow e : T \qquad T \neq \star \qquad \lfloor T \rfloor \not\preceq S}{\Gamma; \sigma \vdash d{\Downarrow}S \rightsquigarrow \texttt{fail} : T'}$$

Figure 13. Translation from $\lambda_d^{\Downarrow}$ to $\lambda_e^{\to}$.

reasonable to additionally warn the programmer of the error at compile-time, and Reticulated Python's implementation of this algorithm does so. However, the DCheckFail rule does not cause the program

$\boxed{\lceil S \rceil = T}$

$$\begin{array}{rcl} \lceil \to \rceil & = & \star \to \star \\ \lceil \text{ref} \rceil & = & \text{ref}\,\star \\ \lceil \text{int} \rceil & = & \text{int} \\ \lceil \star \rceil & = & \star \end{array}$$

$\boxed{\lfloor T \rfloor = S}$

$$\begin{array}{rcl} \lfloor T_1 \to T_2 \rfloor & = & \to \\ \lfloor \text{ref}\,T \rfloor & = & \text{ref} \\ \lfloor \text{int} \rfloor & = & \text{int} \\ \lfloor \star \rfloor & = & \star \end{array}$$

$\boxed{S \preceq S}$

$$\frac{}{\to \preceq \to} \qquad \frac{}{\text{ref} \preceq \text{ref}}$$

$$\frac{}{\text{int} \preceq \text{int}} \qquad \frac{}{S \preceq \star}$$

$\boxed{\vdash T <: T}$

$$\frac{}{\vdash \text{int} <: \text{int}} \qquad \frac{\vdash T_3 <: T_1 \qquad \vdash T_2 <: T_4}{\vdash T_1 \to T_2 <: T_3 \to T_4} \qquad \frac{\vdash T_1 <: T_2 \qquad \vdash T_2 <: T_1}{\vdash \text{ref}\,T_1 <: \text{ref}\,T_2}$$

$$\frac{}{\vdash \star <: \star} \qquad \frac{}{\vdash \text{int} <: \star} \qquad \frac{\vdash T_1 \to T_2 <: \star \to \star}{\vdash T_1 \to T_2 <: \star} \qquad \frac{\vdash \text{ref}\,T <: \text{ref}\,\star}{\vdash \text{ref}\,T <: \star}$$

Figure 14. Full types and subtyping.

to be statically rejected, because this analysis is a runtime optimization that maintains the semantics of the gradually typed language, where a check would detect a runtime error and fail.

3.4.1. Soundness of constraint solving. To prove that the solution algorithm shown in Figure 11 generates valid solutions to constraint sets, we must first define what a *solution* to a constraint set Ω must consist of. A constraint set is solved by a mapping σ if all the constraints in Ω are satisfied, and a subtyping constraint $A_1 <: A_2$ is satisfied if $\vdash \sigma A_1 <: \sigma A_2$, where σA is the substitution of all variables in A with their definitions in σ, resulting in a type T, and using the subtyping relation on T defined in Figure 14. Note that the subtyping rules for types T do not admit $\star$ as a universal supertype (i.e. $\top$): function types are only subtypes of $\star$ if the function is a subtype of $\star \to \star$, which in turn requires the function's source type to be $\star$ itself. This is required to ensure that function types that are passed into $\star$ cannot make any assumptions about what arguments are passed into them. Similar reasoning applies to why references are only subtypes of $\star$ if they are subtypes of ref $\star$.

A check constraint $(A_1{:}S) = A_2$ can be satisfied in one of two ways: first, if σA_1 corresponds to the tag S, then the check constraint is simply an equality constraint, and it is satisfied if σA_1 is syntactically

equal to σA_2, written $\vdash \sigma A_1 = \sigma A_2$. On the other hand, if σA_1 does *not* correspond to S, then all that can be known about σA_2 is that it *does* correspond to S, since the transient check that generated the constraint will fail if values flowing through it do not. If S is $\rightarrow$, then $\lfloor \sigma A_2 \rfloor = \ \rightarrow$ but the argument and return types of σA_2 must be dynamic—the values flowing through the check may be from the open world, beyond the reach of the analysis. Likewise, reference types must be references to $\star$ if they are on the right of a check constraint with a mismatch between the checked type on the left and the tag.

Therefore, the definition of a solution σ to Ω is as follows:

Definition 6.1. *A mapping σ is a* solution *to Ω if:*

(1) *For all $A_1 <: A_2 \in \Omega$, $\vdash \sigma A_1 <: \sigma A_2$.*

(2) *For all $(A_1{:}S) = A_2 \in \Omega$:*

  (a) *If $\lfloor \sigma A_1 \rfloor = S$, then $\vdash \sigma A_1 = \sigma A_2$.*

  (b) *Otherwise, for all $\alpha \in$ parts(A_2), $\vdash \sigma\alpha = \star$.*

(3) *For all $A_1 = A_2 \in \Omega$, $\vdash \sigma A_1 = \sigma A_2$.*

(4) *For all $\alpha \triangleq T \in \Omega$, $\vdash \sigma\alpha = T$.*

(5) *For all $\alpha : S \in \Omega$, $\lfloor \sigma\alpha \rfloor = S$.*

We can now prove that the solution algorithm in Figure 11 generates valid solutions.

Theorem 6.2. *If $\Omega \Downarrow \sigma$, then σ is a solution to Ω.*

This proof is shown in Appendix 2.5 and is by induction on $\Omega \Downarrow \sigma$. It relies on a lemma showing that a solution to any Ω is also a solution for Ω' if $\Omega' \longrightarrow \Omega$.

3.4.2. Correctness of check removal. To show that our our approach preserves the semantics of the program, we prove that any check that *would* be removed by the translation process (as given in Figure 13) cannot fail if it remained in the program. Given some $\lambda_d^{\Downarrow}$ program d, suppose that the constraint generation process from Figure 7 gives d the type A (which may be a variable) and generates the constraints Ω, and assume that Ω is solved by some σ. We can translate d directly into $\lambda_e^{\rightarrow}$ by simply erasing its type annotations, *without* using σ to perform the syntax-directed optimization process from Figure 13. This erased program $|d|$, generated using the erasure rules defined in Figure 15, contains all the

$$\boxed{|d| = e}$$

$$\begin{array}{rcl} |x| & = & x \\ |n| & = & n \\ |\lambda(x{:}X){\to}Y.\, d| & = & \lambda x.\, |d| \\ |d_1\ d_2| & = & |d_1|\ |d_2| \\ |\mathtt{ref}_X\ d| & = & \mathtt{ref}\ |d| \\ |!d| & = & !|d| \\ |d_1{:=}d_2| & = & |d_1|{:=}|d_2| \\ |d_1\ +\ d_2| & = & |d_1|\ +\ |d_2| \\ |d{\Downarrow}S| & = & |d|{\Downarrow}S \end{array}$$

$$\rho \ ::= \ \cdot \mid \rho, x = v$$

$$\boxed{\Sigma \vdash \rho : \Gamma}$$

$$\frac{}{\Sigma \vdash \cdot : \emptyset} \qquad \frac{\emptyset; \Sigma \vdash v : T \qquad \Sigma \vdash \rho : \Gamma}{\Sigma \vdash \rho, x = v : \Gamma, x : T}$$

Figure 15. Rules for environments and for erasing to $\lambda_d^{\Downarrow}$ to $\lambda_e^{\to}$.

checks originally present in d. Suppose that $|d|$ evaluates to a value using the small-step dynamic semantics for $\lambda_e^{\to}$ defined in Figure 16, and is then checked against the type tag of its solution, $\lfloor \sigma A \rfloor$. We prove that this check will always succeed at runtime. This is the same criterion used to actually remove checks in Figure 13, so by showing that the checks that would be removed are redundant, we verify that they can be removed.

In this theorem, d may be an open term—$\rho(|d|)$ substitutes values from an environment ρ (defined and given a typing judgment in Figure 15) into $|d|$.

Theorem 6.3. *Suppose* $\Gamma \vdash d : A; \Omega$ *and* σ *is a solution to* Ω *and* $\Sigma \vdash \rho : \sigma\Gamma$ *and* $\Sigma \vdash \mu$ *and* $\langle \rho(|d|), \mu \rangle \longrightarrow^* \langle v, \mu' \rangle$. *If* $\lfloor \sigma A \rfloor \preceq S$, *then* $\langle v{\Downarrow}S, \mu' \rangle \not\longrightarrow \mathtt{fail}$.

This proof is shown in Appendix 2.4. It relies on a preservation lemma and a lemma showing that, from the canonical forms lemma on $\lambda_e^{\to}$, any value v with type σA will necessarily correspond to that type.

$$
\begin{array}{rcl}
\varsigma & ::= & \langle e, \mu \rangle \mid \texttt{fail} \\
v & ::= & a \mid n \mid \lambda x.\, e \\
\mu & ::= & \cdot \mid \mu[a := v] \\
E & ::= & \square \mid E + e \mid v + E \mid (E\ e) \mid (v\ E) \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{ref}\ E \mid\ !E \mid E{:=}e \mid v{:=}E \mid E{\Downarrow}S
\end{array}
$$

$\boxed{\langle e, \mu \rangle \longrightarrow \varsigma}$

ECheck	$\langle v{\Downarrow}S, \mu \rangle$	$\longrightarrow$	$\langle v, \mu \rangle$	if *hastype*(v, S)
ECheckFail	$\langle v{\Downarrow}S, \mu \rangle$	$\longrightarrow$	$\texttt{fail}$	if $\neg$*hastype*(v, S)
EFail	$\langle \texttt{fail}, \mu \rangle$	$\longrightarrow$	$\texttt{fail}$	
ERef	$\langle \texttt{ref}\ v, \mu \rangle$	$\longrightarrow$	$\langle a, \mu[a := v] \rangle$	where a fresh
EDeref	$\langle !a, \mu \rangle$	$\longrightarrow$	$\langle v, \mu \rangle$	where $\mu(a) = v$
EUpdt	$\langle a{:=}v, \mu \rangle$	$\longrightarrow$	$\langle 0, \mu[a := v] \rangle$	where $\mu(a) = v'$
EApp	$\langle ((\lambda x.\, e)\ v), \mu \rangle$	$\longrightarrow$	$\langle e[x/v], \mu \rangle$	
ELet	$\langle \texttt{let}\ x = v\ \texttt{in}\ e, \mu \rangle$	$\longrightarrow$	$\langle e[x/v], \mu \rangle$	
EAdd	$\langle n_1\ +\ n_2, \mu \rangle$	$\longrightarrow$	$\langle n', \mu \rangle$	where $n_1 + n_2 = n'$

$$
\frac{\langle e, \mu \rangle \longrightarrow \langle e', \mu' \rangle}{\langle E[e], \mu \rangle \longmapsto \langle E[e'], \mu' \rangle}
\qquad
\frac{\langle e, \mu \rangle \longrightarrow \texttt{fail}}{\langle E[e], \mu \rangle \longmapsto \texttt{fail}}
$$

$\boxed{\textit{hastype}(v, S)}$

$$
\frac{}{\textit{hastype}(v, \star)}
\qquad
\frac{}{\textit{hastype}(n, \text{int})}
\qquad
\frac{}{\textit{hastype}(\lambda x.\, e, \rightarrow)}
\qquad
\frac{}{\textit{hastype}(a, \text{ref})}
$$

Figure 16. Dynamic semantics for $\lambda_e^{\rightarrow}$.

4. Performance of Optimized Transient Gradual Typing

In this section we apply the above approach to optimize Reticulated Python, and summarize Reticulated's performance characteristics when running on CPython. This required expanding the type system and constraint generation of Section 3 to handle Python features such as objects and classes, data

structures such as lists and dictionaries, bound and unbound methods, and variadic functions. In addition, the constraint generation includes polymorphic functions and intersection types, although they only occur in the pre-loaded type definitions for Python libraries and builtin functions.

Figures 2 and 3, as previously discussed, show the execution time and overheads for optimized configurations from the typing lattice of each benchmark when executed using CPython. Optimized configurations are shown as blue triangles. Performance is dramatically improved compared to the unoptimized (red circle) configurations. In several benchmarks, the overhead is entirely eliminated because the optimization is able to delete nearly every check in every configuration from the typing lattice. For these results, we make the closed world assumption for the benchmarks.

The `suffixtree` benchmark, which Takikawa et al. [90] tested in Typed Racket and found overheads of up to 105×, also performs worse than other benchmarks in Reticulated Python after optimization: although it has negligible overhead in configurations with high type weight, some configurations with intermediate type weight still have an overhead of over 2×. This is because `suffixtree`, unlike the other benchmarks, cannot be fully statically typed using Reticulated Python's type system: the version with the highest type weight, used to generate the other samples, still includes a function, `node_follow`, which is both annotated as returning type `Any`, and which our inference process is unable to generate any type more specific than `Any`, because it is able to return either a function or a boolean, and neither Reticulated Python's surface gradual type system, nor the type system used in inference, support union types.

Because `node_follow`'s return values flow into statically typed code, checks will be needed. In most configurations, checks occur when the result of `node_follow` passes into a statically typed function, and checks can be removed from the rest of the program. However, some configurations allow the dynamic values to flow further into the program before encountering a check; the dynamicity has "infected" more of the program and degraded performance.

Figure 17 summarizes the performance results for both optimized and unoptimized Reticulated Python under CPython. This table shows the mean overhead, maximum overhead, and overhead for the fully typed (or nearest to fully typed) configurations for each benchmark with the original unoptimized approach and with our optimization. Without optimization, the average of all configurations from each

	Unopt. overheads			Opt. overheads		
Benchmark	Mean	Max	Static	Mean	Max	Static
`pystone`	2.39×	3.82×	3.72×	1.01×	1.03×	1.00×
`chaos`	1.84×	3.22×	3.15×	1.10×	1.46×	0.98×
`snake`	2.31×	3.79×	3.70×	1.04×	1.49×	0.98×
`go`	2.32×	4.87×	4.56×	1.02×	1.14×	1.02×
`meteor_contest`	1.82×	3.20×	3.05×	1.00×	1.10×	1.00×
`suffixtree`	2.49×	4.48×	4.34×	1.27×	2.38×	0.98×
`float`	2.04×	3.53×	3.53×	1.01×	1.12×	1.00×
`nbody`	2.70×	5.95×	5.19×	0.98×	1.27×	0.95×
`sieve`	1.52×	2.17×	2.09×	1.01×	1.06×	1.01×
`spectral_norm`	2.19×	3.33×	2.98×	1.00×	1.20×	0.99×
Average	2.21×	5.95×	3.63×	1.06×	2.38×	0.99×

Figure 17. Performance details for Reticulated Python benchmarks under CPython. Red text indicates *worse than* 3-deliverability and blue highlighting indicates 1.25-deliverability.

benchmark meets the cutoff of 3-deliverability, meaning that their overheads were 3× or less, but most static cases in the benchmarks are not 3-deliverable. Our optimization dramatically improves the results: not only are all configurations 3-deliverable, but all fully typed configurations pass the stricter cutoff of *1.25-deliverability*, with slowdowns of 25% or less over the original untyped program. In all, we found that with the optimization, the average overhead across the typing lattices of all benchmarks was only 6%.

5. Performance on PyPy, a Tracing JIT

The analysis described in Section 3 and evaluated in Section 4 removed checks when they can be statically guaranteed to never fail, which suggests that a dynamic analysis could accomplish the same task. We examined this question using a tracing JIT implementation of Python 3, PyPy [18]. As a tracing JIT,

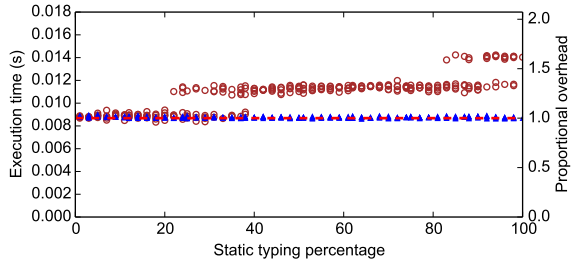
PyPy's implementation of Python 3 is dramatically more efficient in general than CPython, which is implemented as an interpreter; as a result, PyPy performs on average 4.3× better than CPython on the untyped version of the benchmarks when considering absolute execution time. As a result, in order to isolate the specific cost of runtime enforcement of gradual typing, in this section all comparisons are between one PyPy execution and another.

Figures 18 and 19 show the typing lattices for our benchmarks when run on PyPy, both with the standard and optimized transient approaches. Figure 20 summarizes these results. Without static check removal, PyPy's performance varies but is almost always better than CPython's relative to the baseline performance of the untyped benchmarks. In some benchmarks (`float`, `nbody`, `sieve`, `spectral_norm`) configurations across the typing lattice perform almost as well as the untyped version of the program without any static optimizations. In other cases (`pystone`, `chaos`, `snake`) performance still degrades as types are added, although to a lesser degree than with CPython. In one case, `go`, transient gradual typing significantly *improves* performance on average, suggesting that transient checks may cause the JIT compiler to activate earlier or with better traces than it would otherwise. All configurations were 3-deliverable and most were 1.25-deliverable even without optimization, with an overall mean slowdown of 3%, though some configurations have significant overhead (up to 2.61×). In all cases, the parameters of the experiment were tuned in order to allow the JIT to have sufficient time to warm up, and without this additional time the performance overhead of Reticulated Python is relatively larger. For example, in the `pystone` experiment shorter executions led to the fully-typed version of the program displaying a $3\times$ overhead compared to the untyped program, similar to the overhead observed with CPython. However, when increasing the number of iterations of the main loop in this example by $100\times$, the overall overhead (including warmup) drops to $1.7\times$. As Barrett et al. [14] show, JIT warmup behavior is unpredictable, and future work will require further study of the performance of PyPy with Reticulated Python at different stages of warmup.

The best results were obtained by combining our optimization and the PyPy JIT. When run with optimized transient gradual typing, the average overhead was 0% over the baseline, and every configuration fell within the 1.25x-deliverable range. The result is an approach that appears practical for real-world

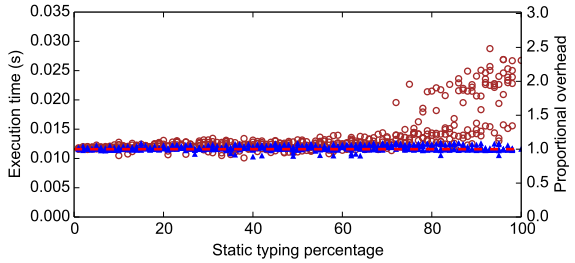
Benchmark: `pystone`

(206 SLoC, 532 configurations)



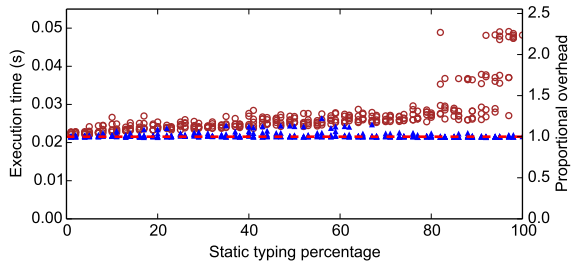
Benchmark: `chaos`

(184 SLoC, 982 configurations)



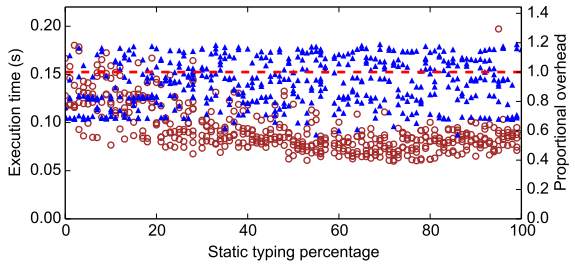
Benchmark: `snake`

(112 SLoC, 662 configurations)



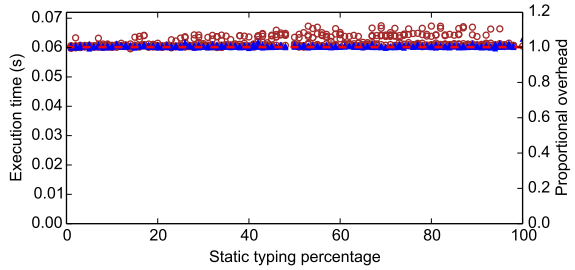
Benchmark: `go`

(394 SLoC, 1001 configurations)



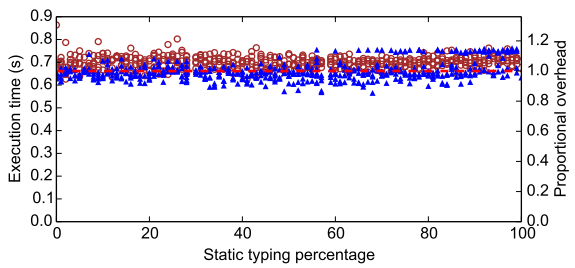
Benchmark: `meteor_contest`

(106 SLoC, 972 configurations)



Benchmark: `suffixtree`

(338 SLoC, 1001 configurations)



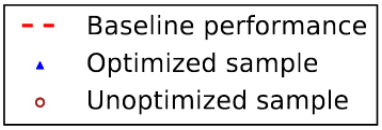


Figure 18. Typing lattices for Reticulated Python benchmarks under PyPy (1 of 2).

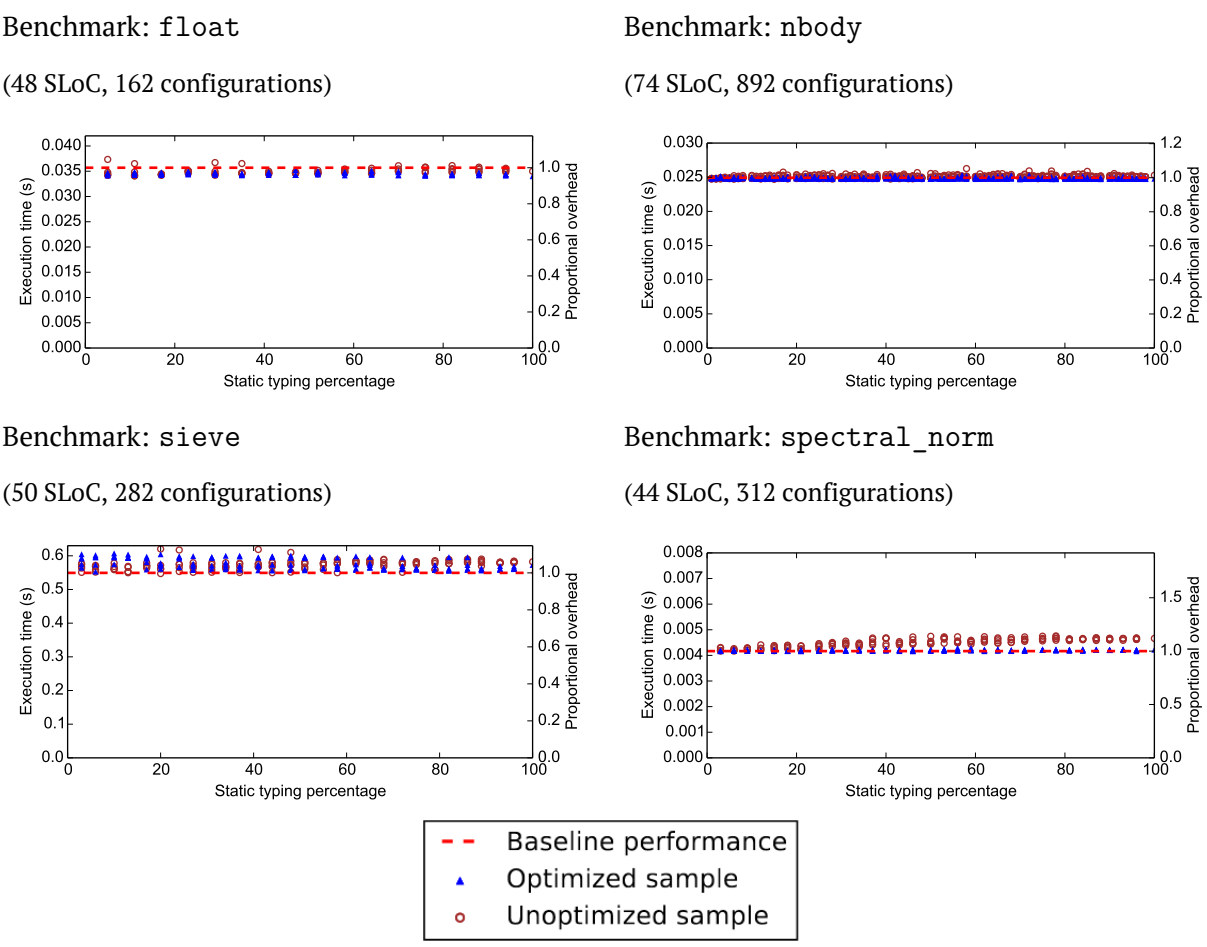


Figure 19. Typing lattices for Reticulated Python benchmarks under PyPy (2 of 2).

applications. In future work, we will further examine the interactions between JITs and the transient approach, for example, to better understand the speedups seen in `go`.

Figure 21 shows the relative performance of each approach, normalized to the performance of standard, unoptimized Reticulated Python running on CPython. Every configuration of every experiment is represented in this graph, normalized against the overhead of CPython without optimization. This representation is owed to Bauman et al. [15]. Since CPython without optimization is normalized against itself, it forms a line $x = y$, while the farther below that line every other configuration is, the better its relative performance. As always, PyPy configurations are normalized against the baseline performance of the untyped experiment under PyPy, and CPython overheads are normalized against CPython

	Unopt. overheads			Opt. overheads		
Benchmark	Mean	Max	Static	Mean	Max	Static
`pystone`	1.24×	1.65×	1.62×	1.01×	1.04×	1.00×
`chaos`	1.18×	2.61×	2.31×	1.02×	1.12×	1.00×
`snake`	1.25×	2.28×	2.24×	1.02×	1.23×	1.00×
`go`	0.61×	1.29×	0.57×	0.92×	1.19×	1.16×
`meteor_contest`	1.02×	1.12×	1.01×	1.01×	1.05×	1.05×
`suffixtree`	1.06×	1.31×	1.06×	1.00×	1.16×	0.96×
`float`	0.98×	1.05×	0.98×	0.97×	1.00×	0.95×
`nbody`	1.00×	1.05×	1.01×	1.00×	1.02×	0.99×
`sieve`	1.05×	1.18×	1.06×	1.05×	1.11×	1.04×
`spectral_norm`	1.08×	1.15×	1.12×	1.01×	1.03×	1.01×
Average	1.03×	2.61×	1.30×	1.00×	1.23×	1.02×

Figure 20. Performance details for Reticulated Python benchmarks under PyPy.

executions. This representation shows that each alternate approach studied in this chapter—the static optimization, running on PyPy, and both—result in a dramatic improvement in performance compared to the standard design.

Reticulated Python is not the only approach to gradual typing that can use a tracing JIT—Bauman et al. [15] showed that Pycket, a language based on Typed Racket but implemented in RPython (a language for automatically generating tracing JITs that PyPy itself is implemented in) [18], performed better than standard Typed Racket but still displayed worst-case overheads of up to 10.5×. Pycket uses the guarded strategy, and while the tracing JIT was successful at reducing the overhead of that approach, our results suggest that the transient enforcement strategy is especially suited to use with tracing JITs.

6. Discussion and Future Work

In this section I will discuss some of the remaining open problems introduced by this work, and suggest directions for future research.

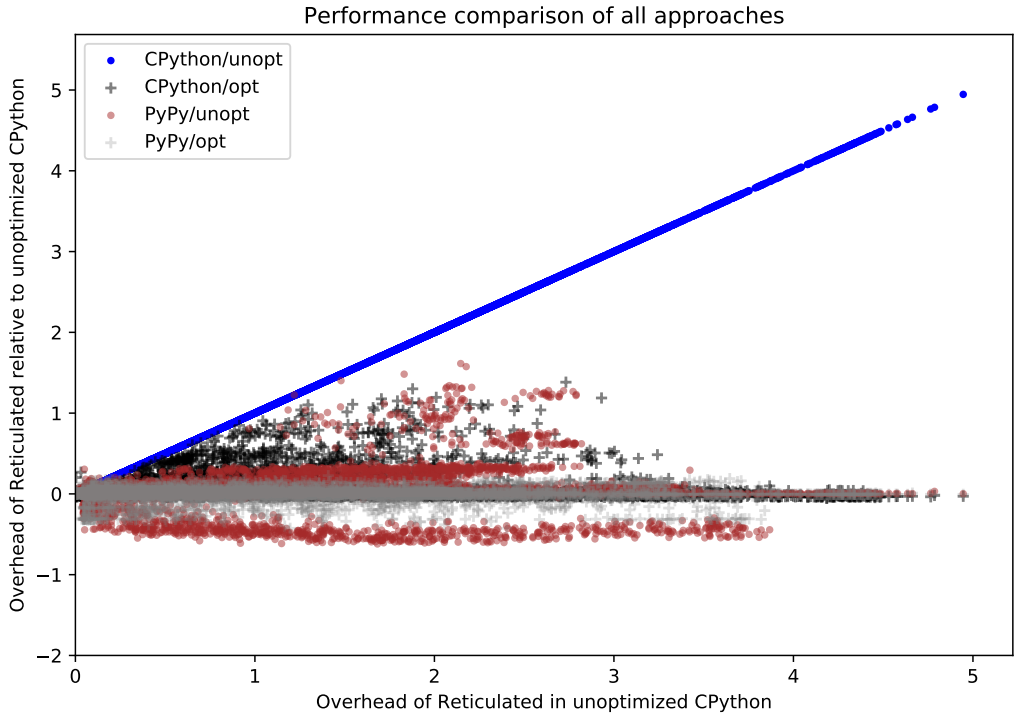


Figure 21. Performance comparison of CPython and PyPy with and without optimization, relative to the performance of Reticulated under CPython without optimization.

6.1. Further Optimizations. The inference technique discussed in this chapter and implemented in Reticulated Python has so far only been applied to remove the checks inserted during the transient translation phase. However, in principle it could be use to perform additional optimizations. For instance, Python programmers often manually write checks that are equivalent to transient enforcement: for example, the following program includes a `isinstance` check which returns whether a variable is an integer or not.

```
1 def f(x):
2   if isinstance(x, int):
3     return x + x
4   else:
5     return -1
6
7 f(42)
```

In this case, the type inference phase could determine that `x` is always an integer (assuming a closed world), and so the `isinstance` check is known to always return true. As a result, the runtime overhead of the check could be removed, and the entire else-branch of the conditional could be deleted as dead code.

Yet more potential optimizations may become available in other contexts. For example, many dynamic languages like Python perform runtime tag checks to ensure that, for example, callees are values that may be called. If a surface language is translated to a target language that explicitly represents these checks, our inference approach may be able to remove unnecessary checks, allowing for the program to become more performant overall; this approach would be similar to the tag-check-removal inference techniques proposed for Scheme by Henglein [46].

6.2. Blame. Blame tracking and the blame theorem are key features of gradually typed languages (cf. Chapter 5), and while the optimization technique and implementation discussed in this chapter does not focus on blame tracking, there are important connections and crucial future work which I will discuss in this section.

6.2.1. Collaborative blame tracking *à la* Chapter 5. In Chapter 5 I described an approach to blame tracking for transient gradual typing that uses a runtime blame map to indicate, when a check fails, a set of possibly responsible boundary-crossing sites. Combining this approach to blame tracking with the optimization presented here is future work made more challenging by the fact that, in the blame tracking scheme, runtime checks do more than just verify that values correspond to type constructors—they also update the blame map when the check passes, in order to link the blame history of the result of

the check with the value that triggered it (for example, if a check is validating that the result of calling a function with type int $\to$ int $\to$ int is a function and the check passes, the check then also connects the resultant returned function's blame information with that of the original function that produced it, in order to blame the source of the original function if the resultant function turns out not to actually return an int). As a consequence, checks cannot be so cavalierly deleted in that system as they are here, because of their dual role.

It is clearly possible and would be a straightforward extension to replace any check which has been proven to be redundant with a command that unconditionally updates the blame map, without needing to examine the value's runtime type. This would remove some of the overhead of transient blame-tracking, though much of it would remain. It is possible that this analysis could be extended to prove whether or not segments of code have to be blame-tracked at all, by proving that not only do they not evaluate to `fail`, but that no blame-tracking path can traverse values originating in that segment of code either. In such a case, all blame-tracking code could be removed altogether; it is not immediately apparent that the inference technique presented here (focused as it is on type *constructors*) extends to the analysis required for that optimization, however.

6.2.2. Static blame computation. The inference technique used here does, however, suggest a different technique for providing blame information to programmers when checks fail, by augmenting those checks that are not deleted with *why* they were not able to be deleted. Recall that the key technique used in Section 3 is to determine the type constructors of all lower bounds of type variables, and delete checks of values whose types are variables if the variable's lower bounds all correspond to the constructor being checked against. In the cases when a check is *not* deleted, it is because one of these lower bounds has a constructor that does not correspond to the check.

In future work, I aim to extend this approach so that constraints include blame labels corresponding to their location in the source, and when a check is not proven to be redundant, the check will be augmented with these blame labels. Then, if the check fails at runtime, it will report the possible locations that could have led to the error occurring.

This approach is distinct from traditional blame tracking because it is entirely static—values themselves do not track blame information and it is not calculated during execution; instead, types and constraints

track blame information, and it is calculated statically and is therefore less precise than traditional blame tracking. Nonetheless, unlike runtime blame tracking, the fact that it can be computed statically means that it is likely to display little or no runtime overhead. I aim to implement this design and explore this hypothesis in future work.

6.2.3. Correctness and the Blame Theorem. Finally, I observe that while *blame-tracking* as a practical development aid to programmers is not a part of this work, properties similar to those required by the *blame theorem* are. Recall that the blame theorem, informally described by Wadler and Findler [104] as "well-typed programs can't be blamed," requires that any conversion between two types (in the context of the guarded strategy, a cast) is "safe," then if a program containing such a conversion results in a runtime enforcement failure, that particular conversion is not blamed. A corollary of this is that if *all* conversions in a program are safe, and assuming that runtime enforcement failures are only generated from conversion sites (as is the case in the guarded strategy), then the program will not fail—again, "well-typed programs can't be blamed."

A similar property also holds for the transient design presented here, as a corollary of Theorem 6.3.

Corollary 6.1. *Suppose* $\emptyset \vdash d : A; \Omega$ *and* σ *is a solution to* Ω*.*

Suppose also that for all terms $e \Downarrow S$ *that appear in the derivation of* $\emptyset \vdash d : A; \Omega$*, there exists a* $\Gamma, \rho, A', \Omega', \Sigma, \mu, \mu'$ *such that* $\Gamma \vdash d : A'; \Omega'$*,* σ *is a solution to* Ω'*,* $\Sigma \vdash \rho : \sigma\Gamma'$*,* $\langle \rho(e), \mu \rangle \longrightarrow^* \langle v, \mu' \rangle$*, and* $\lfloor \sigma A' \rfloor \preceq S$*.*

Then $\langle v \Downarrow S, \mu' \rangle \not\longrightarrow \texttt{fail}$*.*

In other words, if all of the checks that appear in a program are safe, then the program itself will not fail (whether or not those checks are removed). This proof relies on the fact that all checks that could ever be evaluated in a program exist in its source (no new checks are generated), and that the only way for a program to step to `fail` is via a failed check—both assumptions that clearly hold from inspecting the evaluation rules in Figure 12.

One remaining question is what the relationship between the "safety" criterion presented here and that of the blame theorem in guarded systems is. Both are claims about the runtime behavior of a program,

and in both cases the antecedents of the theorem trivially hold for all programs in statically-typed subsets of the languages that the theorem holds in. However, there may be significant differences in what other programs can be considered "safe." As discussed in Chapter 5, the sets of gradually typed programs that evaluate to runtime enforcement failures under guarded and under transient are overlapping, but distinct, sets, which suggests that the safety criteria between them will vary as well. An important step for understanding this may involve the development of criteria that describe safe *evaluations* rather than safe *programs*; I conjecture that such a perspective may reveal behavior relationships between the guarded and transient designs which are not apparent from simply observing the source locations where runtime enforcement code has been inserted, a perspective from which guarded and transient appear very different. In future work, I hope to develop such a perspective and use it to characterize what makes programs safe or unsafe in both settings.

7. Related Work

Gradual type systems. In our work, checks are removed when we infer that the types they verify can be trusted. This is suggestive of the *strict confined gradual typing* of Allende et al. [12], which allows programmers to restrict types such that their inhabitants must never have passed through dynamic code (indicated by $\downarrow T$) or will never pass through dynamic code in the future (indicated by $\uparrow T$). If a term has type $\downarrow T$, the type system verifies that value originated from an introduction form for values of that type. We suspect that this information could be used to remove transient checks or perform other optimizations.

Our approach is also related to *concrete types* [108, 72], as discussed in Chapter 2, Section 2.4. The formulation of concrete vs. like types given by Wrigstad et al. [108] is appropriate for the guarded semantics, but splitting types into those which can be statically relied on and those which need runtime verification is similar to our approach.

In addition to the Reticulated Python performance study of Greenman and Migeed [42], the transient approach was also studied and evaluated by Greenman and Felleisen [41]. This work compared the transient and guarded approaches (there, referred to as "first-order embedding" and "higher order embedding") in the context of Racket. The use of a single host language allows these approaches to be directly

compared, and they found that the transient approach resulted in much better worst-case performance (in their experiments, $20.36\times$ compared to $1072.80\times$) at the cost of losing the guarded approach's efficiency benefits when code is fully-typed.

Type inference. Aiken and Wimmers [7] generalize equational constraints used in standard Hindley-Milner type inference [50, 62] to subset constraints $T \subseteq S$. In this work types are interpreted as subsets of a semantic domain of values and the subset relation is equivalent to subtyping. Their type system includes union and intersection types and generates systems of constraints that may be simplified by rewriting, similar to the rules for constraint simplification shown in Figure 9. Aiken and Fähndrich [6] specialize this approach to determine where coercions are needed in dynamically typed programs with tagging and untagging in order to optimize Scheme programs. This approach is similar to ours, except that while our type system is much simpler and does not include untagged types and values, transient checks must generate check constraints. Check constraints are similar to the *conditional constraints* of Pottier [66], but rather than allowing arbitrary implications, check constraints reason exclusively about type tags.

Soft type systems [23, 8] use a similar approach to integrating static and dynamic typing with type inference, but with a different goal: to allow programs written in dynamically typed languages to leverage the benefits of static typing. Soft type systems use type inference to determine where runtime type checks must be inserted into dynamically typed programs so they are well-typed in some static type system. Cartwright and Fagan [23] reconstruct types for their language and determine where checks are needed using circular unification on equality constraints [105] over an encoding of the supertypes of each type in the program which factors out subtyping [69]. Aiken et al. [8] adapt subset constraint generation to soft typing with similar results. The runtime checks, or *narrowers*, used in soft typing are similar to transient checks: a narrower checks that a value corresponds to a specific type constructor. The value is returned unmodified if so, and an error is raised if not. Narrowers serve a different purpose than checks, however: narrowers let programs written in dynamically typed languages pass static typechecking for optimization, while checks enforce the programmer's claims about the types of terms.

The constraint system used by Flow's type inference system, described for the Flowcore calculus by Chaudhuri et al. [25], reasons about tags and tagchecks in order to find sound typings for JavaScript

programs, though it relies on and trusts type annotations at module boundaries. Likewise Guha et al. [44] relate type tags and types similarly to our approach and their tagchecks are equivalent to transient checks, but their goal is to insert the tagchecks needed to type function bodies with respect to their (trusted) annotations, rather than to detect violations of those annotations by untrusted callers.

Rastogi et al. [67] present an approach to optimizing gradually typed programs by inferring more precise types for program annotations, while preserving the program's semantics. Our approach to ensuring interoperability is based on theirs: visible variables in the overall type of a program and their co- or contravariant positions encodes escape analysis, and solutions that can soundly interoperate with arbitrary code can be generated by adding constraints on these variables. Our constraints, however, are based on subtyping rather than on consistency, because our constraints arise from checks and from elimination forms rather than from casts, which are appropriate for guarded gradual typing rather than transient.

8. Conclusions

Gradual typing allows programmers to combine static and dynamic typing in the same language, but allowing interaction between static and dynamic code while ensuring soundness incurs a runtime cost. Takikawa et al. [90] found that this cost can be serious obstacle to the practical use of Typed Racket, a popular gradually typed language which uses the guarded approach to gradual typing. In this paper, we perform a detailed performance analysis of the transient approach in Reticulated Python by analyzing configurations from the typing lattices of ten benchmarks. We show that, in combination with the standard Python interpreter, performance under the transient design degrades as programs evolve from dynamic to static types, to a maximum of $6\times$ slowdown compared to equivalent untyped Python programs. To reduce this overhead, we use an static type inference algorithm based on subtyping and check constraints to optimize programs after transient checks are inserted. This allows many redundant checks to be removed. We evaluated the performance of this approach with an implementation in Reticulated Python and found that performance across the typing lattices of our benchmarks improved to nearly the efficiency of the original programs—a very promising result for the practicality of gradual

typing. Finally, we re-analyzed our Reticulated Python benchmarks using PyPy, a tracing JIT, as a back-end, and found that it produced good performance even without type inference, and that it displayed a no overhead when used in combination with our static optimization.

Bibliography

[1] Martín Abadi. Protection in programming-language translations. In *ICALP*, 1998.

[2] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[3] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. Dynamic typing in a statically-typed language. In *POPL*, pages 213–227, 1989.

[4] Amal Ahmed and Matthias Blume. An equivalence-preserving cps translation via multi-language semantics. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 431–444, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0865-6. doi: 10.1145/2034773.2034830. URL `http://doi.acm.org/10.1145/2034773.2034830`.

[5] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *POPL*, 2011.

[6] Alexander Aiken and Manuel Fähndrich. Dynamic typing and subtype inference. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 182–191, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-719-7.

[7] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 31–41, New York, NY, USA, 1993. ACM. ISBN 0-89791-595-X. doi: 10.1145/165180.165188. URL `http://doi.acm.org/10.1145/165180.165188`.

[8] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-636-0.

[9] Esteban Allende and Johan Fabry. Application optimization when using gradual typing. In *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '11, pages 3:1–3:6, New York, NY, USA, 2011.

ACM. ISBN 978-1-4503-0894-6. doi: http://doi.acm.org/10.1145/2069172.2069175. URL http://doi.acm.org/10.1145/2069172.2069175.

[10] Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Markus Denker. Gradual typing for smalltalk. *Science of Computer Programming*, August 2013.

[11] Esteban Allende, Johan Fabry, and Éric Tanter. Cast insertion strategies for gradually-typed objects. In *DLS*, 2013.

[12] Esteban Allende, Johan Fabry, Ronald Garcia, and Éric Tanter. Confined gradual typing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 251–270, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660222. URL http://doi.acm.org/10.1145/2660193.2660222.

[13] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 2007.

[14] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual machine warmup blows hot and cold. *Proc. ACM Program. Lang.*, 1(OOPSLA):52:1–52:27, October 2017. ISSN 2475-1421. doi: 10.1145/3133876. URL http://doi.acm.org/10.1145/3133876.

[15] Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. Sound gradual typing: Only mostly dead. *Proc. ACM Program. Lang.*, 1(OOPSLA):54:1–54:24, October 2017. ISSN 2475-1421. doi: 10.1145/3133878. URL http://doi.acm.org/10.1145/3133878.

[16] Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to C#. In *ECOOP*. Springer-Verlag, 2010.

[17] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 257–281. Springer Berlin Heidelberg, 2014. ISBN 978-3-662-44201-2. doi: 10.1007/978-3-662-44202-9_11. URL http://dx.doi.org/10.1007/978-3-662-44202-9_11.

[18] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation,*

Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-541-3. doi: http://doi.acm.org/10.1145/1565824.1565827. URL http://doi.acm.org/10.1145/1565824.1565827.

[19] Ambrose Bonnaire-Sergeant. Typed clojure, 2014. URL http://www.typedclojure.org/.

[20] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical optional types for Clojure. In *ESOP*, 2016.

[21] John Tang Boyland. The problem of structural type tests in a gradual-typed language. In *Foundations of Object-Oriented Langauges*, FOOL, 2014.

[22] Gilad Bracha and David Griswold. Strongtalk: typechecking Smalltalk in a production environment. In *OOPSLA*. ACM Press, 1993. ISBN 0-89791-587-9.

[23] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, New York, NY, USA, 1991. ACM Press. ISBN 0-89791-428-7.

[24] Craig Chambers and the Cecil Group. The Cecil language: Specification and rationale. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, 2004.

[25] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and precise type checking for javascript. *Proc. ACM Program. Lang.*, 1(OOPSLA):48:1–48:30, October 2017. ISSN 2475-1421. doi: 10.1145/3133872. URL http://doi.acm.org/10.1145/3133872.

[26] Matteo Cimini and Jeremy G. Siek. The gradualizer: A methodology and algorithm for generating gradual type systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 443–455, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837632. URL http://doi.acm.org/10.1145/2837614.2837632.

[27] Matteo Cimini and Jeremy G. Siek. Automatically generating the dynamic semantics of gradually typed languages. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 789–803, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009863. URL http://doi.acm.org/10.1145/3009837.3009863.

[28] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: no more scapegoating. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 215–226, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: http://doi.acm.org/10.1145/1926385.1926410. URL `http://doi.acm.org/10.1145/1926385.1926410`.

[29] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In *ESOP*, 2012.

[30] Facebook. Hack, 2013. URL `http://hacklang.org`.

[31] Facebook. Flow: A static type checker for javascript, 2014. URL `http://flow.org`.

[32] Facebook. Pyre: A performant type checker for Python 3, 2018. URL `https://pyre-check.org/`.

[33] Manuel Fähndrich and K. Rustan M. Leino. Heap monotonic typestate. In *International Workshop on Alias Confinement and Ownership*, 2003.

[34] Robert Bruce Findler and Matthias Blume. Contracts as pairs of projections. In *International Symposium on Functional and Logic Programming (FLOPS)*, pages 226–241, 2006.

[35] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP*, 2002.

[36] Matthew Flatt and PLT. The Racket reference 6.0. Technical report, PLT Inc., 2014. `http://docs.racket-lang.org/reference/index.html`.

[37] Ronald Garcia and Matteo Cimini. Principal type schemes for gradual programs. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 303–315, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2676992. URL `http://doi.acm.org/10.1145/2676726.2676992`.

[38] Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 429–442, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837670. URL `http://doi.acm.org/10.1145/2837614.2837670`.

[39] Google. Dart: structured web apps, 2011. URL `http://dartlang.org`.

[40] Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA*. ACM Press, 2005.

[41] Ben Greenman and Matthias Felleisen. A spectrum of type soundness and performance. *Proc. ACM Program. Lang.*, 2(ICFP):71:1–71:32, July 2018. ISSN 2475-1421. doi: 10.1145/3236766. URL http://doi.acm.org/10.1145/3236766.

[42] Ben Greenman and Zeina Migeed. On the cost of type-tag soundness. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '18, pages 30–39, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5587-2. doi: 10.1145/3162066. URL http://doi.acm.org/10.1145/3162066.

[43] Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

[44] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing local control and state using flow analysis. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11, pages 256–275, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19717-8. URL http://dl.acm.org/citation.cfm?id=1987211.1987225.

[45] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012.

[46] Fritz Henglein. Global tagging optimization by type inference. In *LFP '92: Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 205–215, New York, NY, USA, 1992. ACM Press. ISBN 0-89791-481-3.

[47] Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22 (3):197–230, June 1994.

[48] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *Trends in Functional Programming*, 2007.

[49] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23(2):167–189, 2010.

[50] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans AMS*, 146:29–60, 1969.

[51] Jim Hugunin, Barry Warsaw, Samuele Pedroni, Brian Zimmer, Frank Wierzbicki, and Ted Leung. The Jython project. http://www.jython.org/Project/, 1997.

[52] Lintaro Ina and Atsushi Igarashi. Gradual typing for generics. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, 2011.

[53] Matthias Keil and Peter Thiemann. Transparent object proxies in JavaScript. In *ECOOP*, 2015.

[54] Andrew Kent. Refinement types for typed racket. http://blog.racket-lang.org/2017/11/adding-refinement-types.html, 2017.

[55] Gary A. Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM Press, 1973.

[56] Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. An efficient compiler for the gradually typed lambda calculus. To appear at Scheme and Functional Programming Workshop '18, 2018.

[57] Jukka Lehtosalo. Mypy — optional static typing for Python, 2012. URL `http://mypy-lang.org/`.

[58] André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. Typed Lua: An optional type system for Lua. In *Workshop on Dynamic Languages and Applications*, 2014.

[59] Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[60] Bertrand Meyer. Eiffel, 1986. URL `http://www.eiffel.org/`.

[61] Microsoft. Typescript, 2012. URL `http://www.typescriptlang.org/`.

[62] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[63] Fabian Muehlboeck and Ross Tate. Sound gradual typing is nominally alive and well. *Proc. ACM Program. Lang.*, 1(OOPSLA):56:1–56:30, October 2017. ISSN 2475-1421. doi: 10.1145/3133880. URL `http://doi.acm.org/10.1145/3133880`.

[64] Max S. New and Amal Ahmed. Graduality from embedding-projection pairs. *Proc. ACM Program. Lang.*, 2(ICFP):73:1–73:30, July 2018. ISSN 2475-1421. doi: 10.1145/3236768. URL http://doi.acm.org/10.1145/3236768.

[65] Max S. New, William J. Bowman, and Amal Ahmed. Fully abstract compilation via universal embedding. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 103–116, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951941. URL http://doi.acm.org/10.1145/2951913.2951941.

[66] Francois Pottier. A versatile constraint-based type inference system. *Nordic J. of Computing*, 7(4): 312–347, 2000. ISSN 1236-6064.

[67] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The ins and outs of gradual type inference. In *POPL*, 2012.

[68] Aseem Rastogi, Nikhil Swamy, Cedric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for TypeScript. Technical Report MSR-TR-2014-99, Microsoft Research, 2014.

[69] Didier Rémy. Type checking records and variants in a natural extension of ml. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 77–88, New York, NY, USA, 1989. ACM.

[70] Brianna M. Ren, John Toman, T. Stephen Strickland, and Jeffrey S. Foster. The ruby type checker. In *Symposium on Applied Computing*, 2013.

[71] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do: A large-scale study of the use of eval in javascript applications. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 52–78, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22654-0. URL http://dl.acm.org/citation.cfm?id=2032497.2032503.

[72] Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete Types for TypeScript. In *ECOOP*, 2015.

[73] Manuel Serrano. *Bigloo: a practical Scheme compiler*. Inria-Rocquencourt, April 2002.

[74] Jeremy G. Siek and Ronald Garcia. Interpretations of the gradually-typed lambda calculus. In *Scheme and Functional Programming Workshop*, 2012.

[75] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.

[76] Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, volume 4609 of *LCNS*, pages 2–27, August 2007.

[77] Jeremy G. Siek and Manish Vachharajani. Gradual typing with unification-based inference. Technical Report CU-CS-1039-08, University of Colorado at Boulder, January 2008.

[78] Jeremy G. Siek and Michael M. Vitousek. Monotonic references for gradual typing. *CoRR*, abs/1312.0694, 2013.

[79] Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *POPL*, 2010.

[80] Jeremy G. Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. In *ESOP*, 2009.

[81] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *SNAPL: Summit on Advances in Programming Languages*, LIPIcs: Leibniz International Proceedings in Informatics, May 2015.

[82] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *SNAPL '15*, 2015.

[83] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic references for efficient gradual typing. In *European Symposium on Programming*, ESOP, April 2015.

[84] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic references for efficient gradual typing. In *ESOP*, 2015.

[85] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: run-time support for reasonable interposition. In *Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, 2012.

[86] Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual typing embedded securely in javascript. In *POPL*, 2014.

[87] Cameron Swords, Amr Sabry, and Sam Tobin-Hochstadt. Expressing contract monitors as patterns of communication. In *ICFP*, 2015.

[88] Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *OOPSLA*, 2012.

[89] Asumu Takikawa, Daniel Feltey, Earl Dean, Flatt Matthew, Robert Bruce Findler, Sam Tobin-Hochstadt, and Matthias Felleisen. Towards practical gradual typing. In *ECOOP*, 2015.

[90] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *POPL*, 2016.

[91] Satish Thatte. Quasi-static typing. In *POPL*, 1990.

[92] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *DLS*, 2006.

[93] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *POPL*, January 2008.

[94] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *POPL*, 2008.

[95] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *ICFP*, 2010.

[96] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. PEP 484: Type hints, 2014. URL `https://www.python.org/dev/peps/pep-0484/`.

[97] Michael M. Vitousek and Jeremy G. Siek. Gradual typing in an open world. Technical Report TR729, Indiana University, 2016.

[98] Michael M. Vitousek and Jeremy G. Siek. From optional to gradual typing via transient checks. In *Script to Program Evolution Workshop (STOP)*, 2016.

[99] Michael M. Vitousek, Shashank Bharadwaj, and Jeremy G. Siek. Towards gradual typing in Jython. In *Scripts to Programs Workshop (STOP)*, 2012.

[100] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for Python. In *DLS*, 2014.

[101] 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*, 2017.

[102] Michael M Vitousek, Jeremy G. Siek, and Avik Chaudhuri. Optimizing and evaluating transient gradual typing. Submitted to POPL '19, 2019.

[103] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Workshop on Scheme and Functional Programming*, pages 15–26, 2007.

[104] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *ESOP*, 2009.

[105] Pierre Weis. *The CAML Reference Manual*. INRIA, 1987.

[106] Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *European Conference on Object-Oriented Programming*, ECOOP'11. Springer-Verlag, 2011.

[107] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. ISSN 0890-5401.

[108] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *POPL*, 2010.

APPENDIX A

Appendices to Chapter 5

1. Appendix: Additional semantics

Figure 1 shows the translation from $\lambda^{\star}_{\rightarrow}$ to $\lambda^{\Downarrow}_{\ell}$ including the insertion of origin markers, and is otherwise identical to Figure 3. Similarly,Figure 2 shows the single-step reduction rules for $\lambda^{\Downarrow}_{\ell}$ with origin markers p included; otherwise it is identical to the rules shown in Figure 6.

Figure 3 shows the precision relations for $\lambda^{\Downarrow}_{\ell}$ and $\lambda^{\star}_{\rightarrow}$ used in proving the gradual guarantee. Figure 4 shows related auxiliary precision relations.

Figure 5 shows the typing rules for $\lambda^{\Downarrow}_{\ell}$ expression contexts.

Figure 6 shows blame safety predicates.

2. Appendix: Proofs

2.1. Open world soundness.

Lemma A.1. *If* $\Gamma(x) = T$*, then* $\lfloor\Gamma\rfloor(x) = \lfloor T\rfloor$.

Proof. Straightforward induction on Γ. □

Lemma A.2. *If* $\Gamma; \Sigma \vdash e : S$ *and for all* $x \in$ *dom*(Γ)*,* $\Gamma(x) = \Gamma'(x)$ *then* $\Gamma'; \Sigma \vdash e : S$.

Proof. Induction on $\Gamma; \Sigma \vdash e : S$. □

Lemma A.3 (Translation preserves types). *For all* Γ, e_s, e, T*, if* $\Gamma \vdash e_s \rightsquigarrow e : T$*, then* $\lfloor\Gamma\rfloor; \emptyset \vdash e : \lfloor T\rfloor$.

Proof. By induction on $\Gamma \vdash e_s \rightsquigarrow e : T$.

$$\boxed{\Gamma \vdash e_s \rightsquigarrow e : T}$$

$$\text{(CInt)}\frac{}{\Gamma \vdash n \rightsquigarrow n : \text{int}} \qquad \text{(CVar)}\frac{\Gamma(x) = T}{\Gamma \vdash x \rightsquigarrow x : T}$$

(CAdd)

$$\frac{\begin{array}{c}\Gamma \vdash e_{s1} \rightsquigarrow e_1 : T_1 \quad T_1 \sim \text{int} \quad \text{fresh}(\ell_1) \\ \Gamma \vdash e_{s2} \rightsquigarrow e_2 : T_2 \quad T_2 \sim \text{int} \quad \text{fresh}(\ell_2)\end{array}}{\Gamma \vdash e_{s1} \ + \ e_{s2} \rightsquigarrow (e_1 :: T_1 \Rightarrow^{\ell_1} \text{int}) +^{\Diamond} (e_2 :: T_2 \Rightarrow^{\ell_2} \text{int}) : \text{int}}$$

(CFun)

$$\frac{\Gamma, f : T_1 \to T_2, x : T_1 \vdash e_s \rightsquigarrow e' : T_2' \quad T_2 \sim T_2'}{\begin{array}{c}\Gamma \vdash \texttt{fun}\ f\ (x{:}T_1) \to T_2.\ e_s \rightsquigarrow \\ \texttt{fun}\ f\ x.\ (\texttt{let}\ x = x{\Downarrow}\langle \lfloor T_1 \rfloor; f; \text{Arg} \rangle\ \texttt{in}\ e') : T_1 \to T_2\end{array}}$$

(CApp)

$$\frac{\begin{array}{lll}\Gamma \vdash e_{s1} \rightsquigarrow e_1 : T & T \rhd T_1 \to T_2 & \text{fresh}(f) \\ \Gamma \vdash e_{s2} \rightsquigarrow e_2 : T_1' & T_1 \sim T_1' & \text{fresh}(\ell)\end{array}}{\begin{array}{l}\Gamma \vdash e_{s1}\ e_{s2} \rightsquigarrow \texttt{let}\ f = e_1 :: T \Rightarrow^{\ell} T_1 \to T_2\ \texttt{in} \\ \quad (f\ (e_2 :: T_1' \Rightarrow^{\ell} T_1)^{\Diamond}) {\Downarrow} \langle \lfloor T_2 \rfloor; f; \text{Res} \rangle : T_2\end{array}}$$

$$\text{(CRef)}\frac{\Gamma \vdash e_s \rightsquigarrow e : T}{\Gamma \vdash \texttt{ref}\ e_s \rightsquigarrow \texttt{ref}\ e : \text{ref}\, T}$$

(CDeref)

$$\frac{\Gamma \vdash e_s \rightsquigarrow e : T \qquad T \rhd \text{ref}\, T_1 \qquad \text{fresh}(x) \qquad \text{fresh}(\ell)}{\Gamma \vdash\ !e_s \rightsquigarrow \texttt{let}\ x = e :: T \Rightarrow^{\ell} \text{ref}\, T_1\ \texttt{in}\ !x^{\Diamond} {\Downarrow} \langle \lfloor T_1 \rfloor; x; \text{Deref} \rangle : T_1}$$

(CUpdtRef)

$$\frac{\begin{array}{lll}\Gamma \vdash e_{s1} \rightsquigarrow e_1 : T & T \rhd \text{ref}\, T_1 & \text{fresh}(\ell_1) \\ \Gamma \vdash e_{s2} \rightsquigarrow e_2 : T_1' & T_1 \sim T_1' & \text{fresh}(\ell_2)\end{array}}{\Gamma \vdash e_{s1} {:=} e_{s2} \rightsquigarrow (e_1 :: T \Rightarrow^{\ell_1} \text{ref}\, T_1) {:=}^{\Diamond} (e_2 :: T_1' \Rightarrow^{\ell_2} T_1) : \text{int}}$$

Figure 1. Compilation from $\lambda^{\star}_{\to}$ to $\lambda^{\Downarrow}_{\ell}$ with origin markers.

Case CInt:

$$\frac{}{\Gamma \vdash n \rightsquigarrow n : \text{int}}$$

By TInt, $\lfloor \Gamma \rfloor; \emptyset \vdash n : \text{int}$.

$$\boxed{\langle e, \sigma, \mathcal{B}\rangle \longrightarrow \varsigma}$$

(EFun)	$\langle \texttt{fun}\ f\ x.\ e, \sigma, \mathcal{B}\rangle \longrightarrow \langle a, \sigma[a \mapsto (\lambda x.e[a/f])], \mathcal{B}\rangle$	where fresh(a)
(EApp)	$\langle a\ v^p, \sigma, \mathcal{B}\rangle \longrightarrow \langle e[v/x], \sigma, \mathcal{B}\rangle$	where $\sigma(a) = (\lambda x.e)$
(ERef)	$\langle \texttt{ref}\ v, \sigma, \mathcal{B}\rangle \longrightarrow \langle a, \sigma[a \mapsto v], \mathcal{B}\rangle$	where fresh(a)
(EDeref)	$\langle !a^p, \sigma, \mathcal{B}\rangle \longrightarrow \langle v, \sigma, \mathcal{B}\rangle$	where $\sigma(a) = v$
(EUpdtRef)	$\langle a{:=}^p v, \sigma, \mathcal{B}\rangle \longrightarrow \langle 0, \sigma[a \mapsto v], \mathcal{B}\rangle$	where $\sigma(a) = v'$
(EAdd)	$\langle n_1 +^p n_2, \sigma, \mathcal{B}\rangle \longrightarrow \langle n', \sigma, \mathcal{B}\rangle$	where $n' = n_1 + n_2$
(ECheckHO)	$\langle v{\Downarrow}\langle S; a; r\rangle, \sigma, \mathcal{B}\rangle \longrightarrow \langle v, \sigma, \varrho(\mathcal{B}, a', \langle a, r\rangle)\rangle$	where *hastype*$(\sigma, v, S), v = a'$
(ECheckFirst)	$\langle v{\Downarrow}\langle S; a; r\rangle, \sigma, \mathcal{B}\rangle \longrightarrow \langle v, \sigma, \mathcal{B}\rangle$	where *hastype*$(\sigma, v, S), v \neq a'$
(ECheckFail)	$\langle v{\Downarrow}\langle S; a; r\rangle, \sigma, \mathcal{B}\rangle \longrightarrow$ *blame*$(\sigma, v, a, r, \mathcal{B})$	where $\neg$(*hastype*(σ, v, S))
(ECastHO)	$\langle v{::}T_1{\Rightarrow}^\ell T_2, \sigma, \mathcal{B}\rangle \longrightarrow \langle v, \sigma, \varrho(\mathcal{B}, a, [\![T_1 \Rightarrow^\ell T_2]\!])\rangle$	where *hastype*$(\sigma, v, \lfloor T_2 \rfloor), v = a$
(ECastFirst)	$\langle v{::}T_1{\Rightarrow}^\ell T_2, \sigma, \mathcal{B}\rangle \longrightarrow \langle v, \sigma, \mathcal{B}\rangle$	where *hastype*$(\sigma, v, \lfloor T_2 \rfloor), v \neq a$
(ECastFail)	$\langle v{::}T_1{\Rightarrow}^\ell T_2, \sigma, \mathcal{B}\rangle \longrightarrow$ Blame$(\{\ell\})$	where $\neg$(*hastype*$(\sigma, v, \lfloor T_2 \rfloor)$)

$$\varrho(\mathcal{B}, a, b) \quad = \quad \mathcal{B}[a \mapsto \mathcal{B}(a) \cup \{b\}]$$

$$\boxed{\langle e, \sigma, \mathcal{B}\rangle \longmapsto \varsigma}$$

$$\frac{\langle e, \sigma, \mathcal{B}\rangle \longrightarrow \langle e', \sigma', \mathcal{B}'\rangle}{\langle E[e], \sigma, \mathcal{B}\rangle \longmapsto \langle E[e'], \sigma', \mathcal{B}'\rangle} \qquad \frac{\langle e, \sigma, \mathcal{B}\rangle \longrightarrow \text{Blame}(\mathcal{L})}{\langle E[e], \sigma, \mathcal{B}\rangle \longmapsto \text{Blame}(\mathcal{L})}$$

Figure 2. Single step reduction with markers attached

Case CVar:

$$\frac{\Gamma(x) = T}{\Gamma \vdash x \rightsquigarrow x : T}$$

By Lemma A.1, $\lfloor\Gamma\rfloor(x) = \lfloor T\rfloor$.

By TVar, $\lfloor\Gamma\rfloor; \emptyset \vdash x : \lfloor T\rfloor$.

$$\boxed{e_s \sqsubseteq e_s}$$

(PEVar) (PEInt)

$$x \sqsubseteq x \quad n \sqsubseteq n$$

(PEFun)

$$\frac{T_{11} \sqsubseteq T_{21} \qquad T_{12} \sqsubseteq T_{22} \qquad e_{s1} \sqsubseteq e_{s2}}{\texttt{fun}\ f\ (x{:}T_{11}) \to T_{12}.\ e_{s1} \sqsubseteq \texttt{fun}\ f\ (x{:}T_{21}) \to T_{22}.\ e_{s2}}$$

(PEApp)

$$\frac{e_{s11} \sqsubseteq e_{s21} \qquad e_{s12} \sqsubseteq e_{s22}}{e_{s11}\ e_{s12} \sqsubseteq e_{s21}\ e_{s22}}$$

(PEAdd)

$$\frac{e_{s11} \sqsubseteq e_{s21} \qquad e_{s12} \sqsubseteq e_{s22}}{e_{s11}\ +\ e_{s12} \sqsubseteq e_{s21}\ +\ e_{s22}}$$

(PERef)

$$\frac{e_{s1} \sqsubseteq e_{s2}}{\texttt{ref}\ e_{s1} \sqsubseteq \texttt{ref}\ e_{s2}}$$

(PEDeref)

$$\frac{e_{s1} \sqsubseteq e_{s2}}{!e_{s1} \sqsubseteq !e_{s2}}$$

(PESet)

$$\frac{e_{s11} \sqsubseteq e_{s21} \qquad e_{s12} \sqsubseteq e_{s22}}{e_{s11}{:=}e_{s12} \sqsubseteq e_{s21}{:=}e_{s22}}$$

$$\boxed{e \sqsubseteq e}$$

(PVar) (PInt) (PAddr)

$$x \sqsubseteq x \quad n \sqsubseteq n \quad a \sqsubseteq a$$

(PFun)

$$\frac{e_1 \sqsubseteq e_2}{\texttt{fun}\ f\ x.\ e_1 \sqsubseteq \texttt{fun}\ f\ x.\ e_2}$$

(PApp)

$$\frac{e_{11} \sqsubseteq e_{21} \qquad e_{12} \sqsubseteq e_{22}}{e_{11}\ e_{12} \sqsubseteq e_{21}\ e_{22}}$$

(PAdd)

$$\frac{e_{11} \sqsubseteq e_{21} \qquad e_{12} \sqsubseteq e_{22}}{e_{11}\ +\ e_{12} \sqsubseteq e_{21}\ +\ e_{22}}$$

(PRef)

$$\frac{e_1 \sqsubseteq e_2}{\texttt{ref}\ e_1 \sqsubseteq \texttt{ref}\ e_2}$$

(PDeref)

$$\frac{e_1 \sqsubseteq e_2}{!e_1 \sqsubseteq !e_2}$$

(PSet)

$$\frac{e_{11} \sqsubseteq e_{21} \qquad e_{12} \sqsubseteq e_{22}}{e_{11}{:=}e_{12} \sqsubseteq e_{21}{:=}e_{22}}$$

(PCheck)

$$\frac{e_{11} \sqsubseteq e_{21} \qquad e_{12} \sqsubseteq e_{22} \qquad S_1 \sqsubseteq S_2}{e_{11}{\Downarrow}\langle S_1; e_{12}; r\rangle \sqsubseteq e_{21}{\Downarrow}\langle S_2; e_{22}; r\rangle}$$

(PCast)

$$\frac{e_1 \sqsubseteq e_2 \qquad T_{11} \sqsubseteq T_{21} \qquad T_{12} \sqsubseteq T_{22}}{e_1{::}T_{11}{\Rightarrow^{\ell}}T_{12} \sqsubseteq e_2{::}T_{21}{\Rightarrow^{\ell}}T_{22}}$$

(PLet)

$$\frac{e_{11} \sqsubseteq e_{21} \qquad e_{12} \sqsubseteq e_{22}}{\texttt{let}\ x = e_{11}\ \texttt{in}\ e_{12} \sqsubseteq \texttt{let}\ x = e_{21}\ \texttt{in}\ e_{22}}$$

Figure 3. Expression precision

$$\boxed{S \sqsubseteq S}$$

$$S \sqsubseteq \star \quad \text{int} \sqsubseteq \text{int} \quad \text{ref} \sqsubseteq \text{ref} \quad \to\sqsubseteq\to$$

$$\boxed{\Gamma \sqsubseteq \Gamma}$$

$$\frac{\forall x \in \mathit{dom}(\Gamma),\ \Gamma(x) \sqsubseteq \Gamma'(x)}{\Gamma \sqsubseteq \Gamma'}$$

$$\boxed{\sigma \sqsubseteq \sigma}$$

$$\frac{\forall a \in \mathit{dom}(\sigma),\ \sigma(x) \sqsubseteq_h \sigma'(x)}{\sigma \sqsubseteq \sigma'}$$

$$\boxed{h \sqsubseteq_h h}$$

$$\frac{v \sqsubseteq v'}{v \sqsubseteq_h v'} \qquad \frac{e \sqsubseteq e'}{(\lambda x.e) \sqsubseteq_h (\lambda x.e')}$$

Figure 4. Auxiliary precision relations

$$\boxed{\vdash \mathcal{C} : \Gamma;S \Rightarrow \Gamma;S}$$

(CxHole)
$$\frac{}{\vdash \Box : \Gamma;S \Rightarrow \Gamma;S}$$

(CxSubsump)
$$\frac{\vdash \mathcal{C} : \Gamma;S_1 \Rightarrow \Gamma';S_2}{\vdash \mathcal{C} : \Gamma;S_1 \Rightarrow \Gamma';\star}$$

(CxAddL)
$$\frac{\vdash \mathcal{C} : \Gamma;S \Rightarrow \Gamma';\star \qquad \Gamma';\emptyset \vdash e : \star}{\vdash \mathcal{C} \ +^{\blacklozenge}\ e : \Gamma;S \Rightarrow \Gamma';\text{int}}$$

(CxAddR)
$$\frac{\vdash \mathcal{C} : \Gamma;S \Rightarrow \Gamma';\star \qquad \Gamma';\emptyset \vdash e : \star}{\vdash e \ +^{\blacklozenge}\ \mathcal{C} : \Gamma;S \Rightarrow \Gamma';\text{int}}$$

(CxFun)
$$\frac{\vdash \mathcal{C} : \Gamma;S \Rightarrow \Gamma', f\colon \to, x\colon \star\,;\star}{\vdash \texttt{fun}\ f\ x.\ \mathcal{C} : \Gamma;S \Rightarrow \Gamma';\to}$$

(CxAppL)
$$\frac{\vdash \mathcal{C} : \Gamma;S \Rightarrow \Gamma';\star \qquad \Gamma';\emptyset \vdash e : \star}{\vdash \mathcal{C}\ e^{\blacklozenge} : \Gamma;S \Rightarrow \Gamma';\star}$$

(CxAppR)
$$\frac{\vdash \mathcal{C} : \Gamma;S \Rightarrow \Gamma';\star \qquad [1ex]\Gamma';\emptyset \vdash e : \star}{\vdash e\ \mathcal{C}^{\blacklozenge} : \Gamma;S \Rightarrow \Gamma';\star}$$

(CxRef)
$$\frac{\vdash \mathcal{C} : \Gamma;S \Rightarrow \Gamma';\star}{\vdash \texttt{ref}\ \mathcal{C} : \Gamma;S \Rightarrow \Gamma';\text{ref}}$$

(CxDerefL)
$$\frac{\vdash \mathcal{C} : \Gamma;S \Rightarrow \Gamma';\star \qquad \Gamma';\emptyset \vdash e : \star}{\vdash \mathcal{C} :=^{\blacklozenge} e : \Gamma;S \Rightarrow \Gamma';\text{int}}$$

(CxDerefR)
$$\frac{\vdash \mathcal{C} : \Gamma;S \Rightarrow \Gamma';\star \qquad \Gamma';\emptyset \vdash e : \star}{\vdash e :=^{\blacklozenge} \mathcal{C} : \Gamma;S \Rightarrow \Gamma';\text{int}}$$

Figure 5. Context typing

$\boxed{L \text{ safe } \ell}$

$$\text{(LInt)}\ \frac{q \neq \ell}{\text{int}^q \text{ safe } \ell} \qquad \text{(LBot)}\ \frac{\ell_1 \neq \ell_2}{\bot^{\ell_1} \text{ safe } \ell_2} \qquad \text{(LDyn)}\ \frac{}{\star \text{ safe } \ell}$$

$$\text{(LFun)}\ \frac{q \neq \ell \qquad L_1 \text{ safe } \ell \qquad L_2 \text{ safe } \ell}{L_1 \to^q L_2 \text{ safe } \ell} \qquad \text{(LRef)}\ \frac{q \neq \ell \qquad L \text{ safe } \ell}{\text{ref}^q\ L \text{ safe } \ell}$$

$\boxed{\mathcal{B} \vdash b \text{ safe } \ell}$

$$\text{(SfLType)}\ \frac{L \text{ safe } \ell}{\mathcal{B} \vdash L \text{ safe } \ell} \qquad \text{(SfRef)}\ \frac{\forall b \in \mathcal{B}(a) \setminus \{\langle a, r\rangle\},\ \mathcal{B} \vdash b \text{ safe } \ell}{\mathcal{B} \vdash \langle a, r\rangle \text{ safe } \ell}$$

$\boxed{\mathcal{B} \vdash \sigma \text{ safe } \ell}$

$$\text{(SfEmpHeap)}\ \frac{}{\mathcal{B} \vdash \cdot \text{ safe } \ell} \qquad \text{(SfHeapCell)}\ \frac{\mathcal{B} \vdash v \text{ safe } \ell \qquad \mathcal{B} \vdash \sigma \text{ safe } \ell}{\mathcal{B} \vdash \sigma[a \mapsto v] \text{ safe } \ell}$$

$$\text{(SfHeapClosure)}\ \frac{\mathcal{B} \vdash e \text{ safe } \ell \qquad \mathcal{B} \vdash \sigma \text{ safe } \ell}{\mathcal{B} \vdash \sigma[a \mapsto (\lambda x.e)] \text{ safe } \ell}$$

$\boxed{\mathcal{B} \vdash e \text{ safe } \ell}$

$$\text{(SfCast)}\ \frac{\mathcal{B} \vdash e \text{ safe } \ell_2 \qquad [\![T_1 \Rightarrow^{\ell_1} T_2]\!] \text{ safe } \ell_2}{\mathcal{B} \vdash e :: T_1 \Rightarrow^{\ell_1} T_2 \text{ safe } \ell_2}$$

$$\text{(SfCheck)}\ \frac{\mathcal{B} \vdash e_1 \text{ safe } \ell \qquad \mathcal{B} \vdash e_2 \text{ safe } \ell}{\mathcal{B} \vdash e_1 \Downarrow \langle S; e_2; r\rangle \text{ safe } \ell} \qquad \text{(SfAddr)}\ \frac{\forall b \in \mathcal{B}(a),\ \mathcal{B} \vdash b \text{ safe } \ell}{\mathcal{B} \vdash a \text{ safe } \ell}$$

$$\text{(SfVar)}\ \frac{}{\mathcal{B} \vdash x \text{ safe } \ell} \qquad \text{(SfApp)}\ \frac{\mathcal{B} \vdash e_1 \text{ safe } \ell \qquad \mathcal{B} \vdash e_2 \text{ safe } \ell}{\mathcal{B} \vdash e_1\ e_2 \text{ safe } \ell} \qquad \cdots$$

Figure 6. Blame safety predicates

Case CAdd:

$$\frac{\Gamma \vdash e_{s1} \rightsquigarrow e_1 : T_1 \quad T_1 \sim \text{int} \quad \text{fresh}(\ell_1) \qquad \Gamma \vdash e_{s2} \rightsquigarrow e_2 : T_2 \quad T_2 \sim \text{int} \quad \text{fresh}(\ell_2)}{\Gamma \vdash e_{s1} \ + \ e_{s2} \rightsquigarrow (e_1 :: T_1 \Rightarrow^{\ell_1} \text{int}) +^{\Diamond} (e_2 :: T_2 \Rightarrow^{\ell_2} \text{int}) : \text{int}}$$

By the IH, $\lfloor \Gamma \rfloor ; \emptyset \vdash e_1 : \text{int}$.

By the IH, $\lfloor \Gamma \rfloor ; \emptyset \vdash e_2 : \text{int}$.

By TPlus, $\lfloor \Gamma \rfloor ; \emptyset \vdash e_1 +^{\Diamond} e_2 : \text{int}$.

Case CFun:

$$\frac{\Gamma, f : T_1 \to T_2, x : T_1 \vdash e_s \rightsquigarrow e' : T_2' \quad T_2 \sim T_2'}{\Gamma \vdash \texttt{fun}\ f\ (x{:}T_1) \to T_2.\ e_s \rightsquigarrow \texttt{fun}\ f\ x.\ (\texttt{let}\ x = x{\Downarrow}\langle \lfloor T_1 \rfloor ; f ; \text{Arg} \rangle\ \texttt{in}\ e') : T_1 \to T_2}$$

By TVar, $\lfloor \Gamma \rfloor, f{:}\to, x{:}\star ; \emptyset \vdash x : \star$.

Let us assume that $x \neq f$.

Then by TVar $\lfloor \Gamma \rfloor, f{:}\to, x{:}\star ; \emptyset \vdash f :\to$.

By TCheck,

$\lfloor \Gamma \rfloor, f{:}\to, x{:}\lfloor T_1 \rfloor ; \emptyset \vdash x{\Downarrow}\langle \lfloor T_1 \rfloor ; f ; \text{Arg} \rangle : \lfloor T_1 \rfloor$.

By the IH, $\lfloor \Gamma \rfloor, f{:}\to, x{:}\lfloor T_1 \rfloor ; \emptyset \vdash e' : \lfloor T_2' \rfloor$.

By Lemma A.2, $\lfloor \Gamma \rfloor, f{:}\to, x{:}\star, x{:}\lfloor T_1 \rfloor ; \emptyset \vdash e' : \lfloor T_2' \rfloor$.

By TLet,

$\lfloor \Gamma \rfloor, f{:}\to, x{:}\star ; \emptyset \vdash \texttt{let}\ x = x{\Downarrow}\langle \lfloor T_1 \rfloor ; f ; \text{Arg} \rangle\ \texttt{in}\ e' : \lfloor T_2' \rfloor$.

By TSubsump,

$\lfloor \Gamma \rfloor, f{:}\to, x{:}\star ; \emptyset \vdash \texttt{let}\ x = x{\Downarrow}\langle \lfloor T_1 \rfloor ; f ; \text{Arg} \rangle\ \texttt{in}\ e' : \star$.

By TFun, $\lfloor \Gamma \rfloor ; \emptyset \vdash \texttt{fun}\ f\ x.\ (\texttt{let}\ x = x{\Downarrow}\langle \lfloor T_1 \rfloor ; f ; \text{Arg} \rangle\ \texttt{in}\ e') :\to$.

Case CCall:

$$\frac{\Gamma \vdash e_{s1} \rightsquigarrow e_1 : T \quad T \rhd T_1 \to T_2 \quad \text{fresh}(f) \qquad \Gamma \vdash e_{s2} \rightsquigarrow e_2 : T_1' \quad T_1 \sim T_1' \quad \text{fresh}(\ell)}{\Gamma \vdash e_{s1}\ e_{s2} \rightsquigarrow \texttt{let}\ f = e_1 :: T \Rightarrow^{\ell} T_1 \to T_2\ \texttt{in}\ (f\ (e_2 :: T_1' \Rightarrow^{\ell} T_1)^{\Diamond}){\Downarrow}\langle \lfloor T_2 \rfloor ; f ; \text{Res} \rangle : T_2}$$

By the IH, $\lfloor \Gamma \rfloor ; \Sigma \vdash e_1 : \lfloor T \rfloor$.

By the IH, $\lfloor \Gamma \rfloor ; \Sigma \vdash e_2 : \lfloor T_1' \rfloor$.

By TCast, $\lfloor\Gamma\rfloor; \Sigma \vdash e_2 :: T_1' \Rightarrow^{\ell} T_1 : \lfloor T_1 \rfloor$.

By TSubsump, $\lfloor\Gamma\rfloor; \Sigma \vdash e_2 :: T_1' \Rightarrow^{\ell} T_1 : \star$.

By Lemma A.2, $\lfloor\Gamma\rfloor, f\colon \to; \Sigma \vdash e_2 :: T_1' \Rightarrow^{\ell} T_1 : \star$.

By TCast, $\lfloor\Gamma\rfloor; \Sigma \vdash e_1 :: T \Rightarrow^{\ell} T_1 \to T_2 : \to$.

By TVar, $\lfloor\Gamma\rfloor, f\colon \to; \Sigma \vdash f : \to$.

By TApp, $\lfloor\Gamma\rfloor, f\colon \to; \Sigma \vdash f\ (e_2 :: T_1' \Rightarrow^{\ell} T_1)^{\Diamond} : \star$.

By TCheck,

$\lfloor\Gamma\rfloor, f\colon \to; \Sigma \vdash (f\ (e_2 :: T_1' \Rightarrow^{\ell} T_1)^{\Diamond}) \Downarrow \langle \lfloor T_2 \rfloor; f; \mathsf{Res} \rangle : \lfloor T_2 \rfloor$.

By TLet, $\lfloor\Gamma\rfloor, f\colon \to; \Sigma \vdash \texttt{let}\ f = e_1 :: T \Rightarrow^{\ell} T_1 \to T_2\ \texttt{in}\ (f\ (e_2 :: T_1' \Rightarrow^{\ell} T_1)^{\Diamond}) \Downarrow \langle \lfloor T_2 \rfloor; f; \mathsf{Res} \rangle : \lfloor T_2 \rfloor$.

Case CRef:

$$\frac{\Gamma \vdash e_s \rightsquigarrow e : T}{\Gamma \vdash \texttt{ref}\ e_s \rightsquigarrow \texttt{ref}\ e : \mathsf{ref}\ T}$$

By the IH, $\lfloor\Gamma\rfloor; \Sigma \vdash e : \lfloor T \rfloor$.

By TSubsump, $\lfloor\Gamma\rfloor; \Sigma \vdash e : \star$.

By TRef, $\lfloor\Gamma\rfloor; \Sigma \vdash \texttt{ref}\ e : \mathsf{ref}$.

Case CDeref:

$$\frac{\Gamma \vdash e_s \rightsquigarrow e : T \qquad T \rhd \mathsf{ref}\ T_1 \qquad \mathsf{fresh}(x) \qquad \mathsf{fresh}(\ell)}{\Gamma \vdash\ !e_s \rightsquigarrow \texttt{let}\ x = e :: T \Rightarrow^{\ell} \mathsf{ref}\ T_1\ \texttt{in}\ !x \Downarrow \langle \lfloor T_1 \rfloor; x; \mathsf{Deref} \rangle : T_1}$$

By the IH, $\lfloor\Gamma\rfloor; \Sigma \vdash e : \lfloor T \rfloor$.

By TCast, $\lfloor\Gamma\rfloor; \Sigma \vdash e_1 :: T \Rightarrow^{\ell} \mathsf{ref}\ T_1 : \mathsf{ref}$.

By TVar, $\lfloor\Gamma\rfloor, x\colon\mathsf{ref}; \Sigma \vdash x : \mathsf{ref}$.

By TDeref, $\lfloor\Gamma\rfloor, x\colon\mathsf{ref}; \Sigma \vdash !x^{\Diamond} : \star$.

By TCheck, $\lfloor\Gamma\rfloor, x\colon\mathsf{ref}; \Sigma \vdash !x^{\Diamond} \Downarrow \langle \lfloor T_1 \rfloor; x; \mathsf{Deref} \rangle : \lfloor T_1 \rfloor$

By TLet, $\lfloor\Gamma\rfloor; \Sigma \vdash \texttt{let}\ x = e_1 :: T \Rightarrow^{\ell} \mathsf{ref}\ T_1\ \texttt{in}\ !x^{\Diamond} \Downarrow \langle \lfloor T_1 \rfloor; x; \mathsf{Deref} \rangle : \lfloor T_1 \rfloor$.

Case CUpdt:

$$\frac{\begin{array}{ccc}\Gamma \vdash e_{s1} \rightsquigarrow e_1 : T & T \rhd \mathsf{ref}\ T_1 & \mathsf{fresh}(\ell_1) \\ \Gamma \vdash e_{s2} \rightsquigarrow e_2 : T_1' & T_1 \sim T_1' & \mathsf{fresh}(\ell_2)\end{array}}{\Gamma \vdash e_{s1} := e_{s2} \rightsquigarrow (e_1 :: T \Rightarrow^{\ell_1} \mathsf{ref}\ T_1) :=^{\Diamond} (e_2 :: T_1' \Rightarrow^{\ell_2} T_1) : \mathsf{int}}$$

By the IH, $\lfloor\Gamma\rfloor; \Sigma \vdash e_1 : \lfloor T \rfloor$.

By TCast, $\lfloor\Gamma\rfloor; \Sigma \vdash e_1 :: T \Rightarrow^{\ell_1} \mathsf{ref}\, T_1 : \mathsf{ref}$.

By the IH, $\lfloor\Gamma\rfloor; \Sigma \vdash e_2 : \lfloor T_1' \rfloor$.

By TCast, $\lfloor\Gamma\rfloor; \Sigma \vdash e_2 :: T_1' \Rightarrow^{\ell_2} T_1 : \lfloor T_1 \rfloor$.

By TSubsump, $\lfloor\Gamma\rfloor; \Sigma \vdash e_2 :: T_1' \Rightarrow^{\ell_2} T_1 : \star$.

By TUpdt, $\lfloor\Gamma\rfloor; \Sigma \vdash (e_1 :: T \Rightarrow^{\ell_1} \mathsf{ref}\, T_1) :=^{\Diamond} (e_2 :: T_1' \Rightarrow^{\ell_2} T_1) : \mathsf{int}$

□

Lemma A.4 (Inversion). *Suppose* $\Gamma; \Sigma \vdash e : S$. *Then*

- *If* $e = \mathtt{fun}\, f\ x.\ e'$, *then* $\Gamma, f{:}\to, x{:}\star; \Sigma \vdash e' : \star$ *and* $S \in \{\star, \to\}$.
- *If* $e = e_1\ e_2^{\Diamond}$, *then* $\Gamma; \Sigma \vdash e_1 :\to$ *and* $\Gamma; \Sigma \vdash e_2 : \star$ *and* $S = \star$.
- *If* $e = e_1\ e_2^{\blacklozenge}$, *then* $\Gamma; \Sigma \vdash e_1 : \star$ *and* $\Gamma; \Sigma \vdash e_2 : \star$ *and* $S = \star$.
- *If* $e = \mathtt{ref}\, e'$, *then* $\Gamma; \Sigma \vdash e' : \star$ *and* $S \in \{\star, \mathit{ref}\}$.
- *If* $e = !e'^{\Diamond}$, *then* $\Gamma; \Sigma \vdash e' :$ *ref and* $S = \star$.
- *If* $e = !e'^{\blacklozenge}$, *then* $\Gamma; \Sigma \vdash e' : \star$ *and* $S = \star$.
- *If* $e = e_1 :=^{\Diamond} e_2$, *then* $\Gamma; \Sigma \vdash e_1 :$ *ref and* $\Gamma; \Sigma \vdash e_2 : \star$ *and* $S \in \{\star, \mathit{int}\}$.
- *If* $e = e_1 :=^{\blacklozenge} e_2$, *then* $\Gamma; \Sigma \vdash e_1 : \star$ *and* $\Gamma; \Sigma \vdash e_2 : \star$ *and* $S \in \{\star, \mathit{int}\}$.
- *If* $e = e_1\ +^{\Diamond}\ e_2$, *then* $\Gamma; \Sigma \vdash e_1 :$ *int and* $\Gamma; \Sigma \vdash e_2 :$ *int and* $S \in \{\star, \mathit{int}\}$.
- *If* $e = e_1\ +^{\blacklozenge}\ e_2$, *then* $\Gamma; \Sigma \vdash e_1 : \star$ *and* $\Gamma; \Sigma \vdash e_2 : \star$ *and* $S \in \{\star, \mathit{int}\}$.
- *If* $e = e' :: T_1 \Rightarrow^{\ell} T_2$, *then* $\Gamma; \Sigma \vdash e' : \lfloor T_1 \rfloor$ *and* $T_1 \sim T_2$ *and* $S \in \{\star, \lfloor T_2 \rfloor\}$.
- *If* $e = e_1 \Downarrow \langle S'; e_2; r \rangle$, *then* $\Gamma; \Sigma \vdash e_1 : \star$ *and* $\Gamma; \Sigma \vdash e_2 : \star$ *and* $S \in \{\star, S'\}$.
- *If* $e = a$, *then* $\Sigma(a) = S'$ *and* $S \in \{\star, S'\}$.

Proof. Induction on $\Gamma; \Sigma \vdash e : S$. □

Lemma A.5 (Heap weakening). *If* $\Gamma; \Sigma \vdash e : S$ *and* $\Sigma' \sqsubseteq \Sigma$, *then* $\Gamma; \Sigma' \vdash e : S$.

Proof. By induction on $\Gamma; \Sigma \vdash e : S$. Only interesting case:

Case TAddr: Since $a \in \mathit{dom}(\Sigma)$, and $\Sigma' \sqsubseteq \Sigma$, $\Sigma'(a) = \Sigma(a)$. Therefore $\Gamma; \mathit{heapenv}' \vdash e : \Sigma(a)$.

□

Lemma A.6 (Substitution). *If* $\Gamma, x : S; \Sigma \vdash e : S'$ *and* $\Gamma; \Sigma \vdash v : S$*, then* $\Gamma; \Sigma \vdash e[v/x] : S'$.

Proof. By induction on $\Gamma, x : S; \Sigma \vdash e : S'$. □

Lemma A.7 (Runtime types are sound). *If* $\emptyset; \Sigma \vdash v : \star$ *and* $\Sigma \vdash \sigma$ *and hastype*(σ, v, S)*, then* $\emptyset; \Sigma \vdash v : S$.

Proof. By inversion on *hastype*(σ, v, S).

Case : $\dfrac{}{\textit{hastype}(\sigma, n, \text{int})}$

By TInt, $\emptyset; \Sigma \vdash n :$ int

Case : $\dfrac{}{\textit{hastype}(\sigma, v, \star)}$

Immediate.

Case : $\dfrac{\sigma(a) = (\lambda x.e, \rho)}{\textit{hastype}(\sigma, a, \rightarrow)}$

Since $\Sigma \vdash \sigma$, $\Sigma \vdash (\lambda x.e) : \Sigma(a)$.

By inversion on $\Sigma \vdash (\lambda x.e) : \Sigma(a)$, $\Sigma(a) = \rightarrow$.

By TAddr, $\emptyset; \Sigma \vdash a : \rightarrow$.

Case : $\dfrac{\sigma(a) = v}{\textit{hastype}(\sigma, a, \text{ref})}$

Since $\Sigma \vdash \sigma$, $\Sigma \vdash v : \Sigma(a)$.

By inversion on $\Sigma \vdash v : \Sigma(a)$, $\Sigma(a) =$ ref.

By TAddr, $\emptyset; \Sigma \vdash a :$ ref.

□

Lemma A.8 (Heap extension). *If* $\Sigma \vdash \sigma$ *and* $\Sigma[a \mapsto S] \vdash h : S$ *and* $a \notin$ *dom*(Σ)*, then* $\Sigma[a \mapsto S] \vdash \sigma[a \mapsto h]$.

Proof. Suppose $a' \in \Sigma[a \mapsto S]$. If $a = a'$, then immediately $\Sigma[a \mapsto S] \vdash \sigma(a') : \Sigma[a \mapsto S](a')$.
If $a \neq a'$, then $\Sigma \vdash \sigma(a') : \Sigma(a')$. Cases on $\sigma(a')$.

Case : $\sigma(a') = v, \Sigma(a') =$ ref. Have that $\emptyset; \Sigma \vdash \sigma(a') : \star$. By Lemma A.5, $\emptyset; \Sigma[a \mapsto S] \vdash \sigma(a') : \star$. Thus $\Sigma[a \mapsto S] \vdash \sigma(a') : \Sigma(a')$. Since $a \neq a'$, $\Sigma[a \mapsto S] \vdash \sigma(a') : \Sigma[a \mapsto S](a')$.

Case : $\sigma(a') = (\lambda a.e), \Sigma(a') = \rightarrow$. Have that $\emptyset, x{:}\star; \Sigma \vdash e : \star$. By Lemma A.5, $\emptyset, x{:}\star; \Sigma[a \mapsto S] \vdash e : \star$. Thus $\Sigma[a \mapsto S] \vdash \sigma(a') : \Sigma(a')$. Since $a \neq a'$, $\Sigma[a \mapsto S] \vdash \sigma(a') : \Sigma[a \mapsto S](a')$.

Thus for all $a' \in \Sigma[a \mapsto S]$, have $\Sigma[a \mapsto S] \vdash \sigma(a') : \Sigma[a \mapsto S](a')$, so by THeap, $\Sigma[a \mapsto S] \vdash \sigma[a \mapsto h]$. □

Lemma A.9 (Preservation). *If* $\emptyset; \Sigma \vdash e : S$ *and* $\Sigma \vdash \sigma$ *and* $\langle e, \sigma, \mathcal{B} \rangle \longrightarrow \langle e', \sigma', \mathcal{B}' \rangle$*, then* $\emptyset; \Sigma' \vdash e' : S$ *and* $\Sigma' \vdash \sigma'$ *and* $\Sigma' \sqsubseteq \Sigma$.

Proof. By induction on $\langle e, \sigma, \mathcal{B} \rangle \longrightarrow \langle e', \sigma', \mathcal{B}' \rangle$.

Case EFun: With fresh(a),

$$\langle \texttt{fun}\ f\ x.\ e, \sigma, \mathcal{B} \rangle \longrightarrow \langle a, \sigma[a \mapsto (\lambda x.e[a/f])], \mathcal{B} \rangle$$

By Lemma A.4, $\emptyset, f{:}\rightarrow, x{:}\star; \Sigma \vdash e' : \star$ and $S \in \{\star, \rightarrow\}$.

Since a fresh, $\Sigma[a \mapsto \rightarrow] \sqsubseteq \Sigma$.

By Lemma A.5, $\emptyset, f{:}\rightarrow, x{:}\star; \Sigma[a \mapsto \rightarrow] \vdash e' : \star$

By TAddr, $\emptyset, x{:}\star; \Sigma[a \mapsto \rightarrow] \vdash a : \rightarrow$

By Lemma A.6, $\emptyset, x{:}\star; \Sigma[a \mapsto \rightarrow] \vdash e[a/f] : \star$

By THClosure, $\Sigma[a \mapsto \rightarrow] \vdash (\lambda x.e[a/f]) : \rightarrow$.

By Lemma A.8, $\Sigma[a \mapsto \rightarrow] \vdash \sigma[a \mapsto (\lambda x.e[a/f])]$.

By TAddr, $\emptyset; \Sigma[a \mapsto \rightarrow] \vdash a : \rightarrow$.

By TSubsump, $\emptyset; \Sigma[a \mapsto \rightarrow] \vdash a : \star$.

Since $S \in \{\star, \rightarrow\}$, theorem satisfied.

Case EApp: Where $\sigma(a) = (\lambda x.e)$,

$$\langle a\ v^p, \sigma, \mathcal{B} \rangle \longrightarrow \langle e[v/x], \sigma, \mathcal{B} \rangle$$

By Lemma A.4, $\emptyset; \Sigma \vdash v : \star$ and $S = \star$.

By inversion on $\Sigma \vdash \sigma$, $\Sigma \vdash (\lambda x.e) : S'$ for some S'.

By further inversion, $S' = \rightarrow$ and $\emptyset, x{:}\star; \Sigma \vdash e : \star$.

By Lemma A.6, $\emptyset; \Sigma \vdash e[a/x] : \star$, which satisfies the theorem.

Case ERef: With fresh(a),

$$\langle \texttt{ref}\ v, \sigma, \mathcal{B} \rangle \longrightarrow \langle a, \sigma[a \mapsto v], \mathcal{B} \rangle$$

By Lemma A.4, $\emptyset; \Sigma \vdash v : \star$ and $S \in \{\star, \text{ref}\}$.

Since a fresh, $\Sigma[a \mapsto \text{ref}] \sqsubseteq \Sigma$.

By Lemma A.5, $\emptyset; \Sigma[a \mapsto \text{ref}] \vdash v : \star$

By THRef, $\Sigma[a \mapsto \rightarrow] \vdash v : \rightarrow$.

By Lemma A.8, $\Sigma[a \mapsto \text{ref}] \vdash \sigma[a \mapsto v]$.

By TAddr, $\emptyset; \Sigma[a \mapsto \text{ref}] \vdash a : \text{ref}$.

By TSubsump, $\emptyset; \Sigma[a \mapsto \text{ref}] \vdash a : \star$.

Since $S \in \{\star, \text{ref}\}$, theorem satisfied.

Case EDeref: With $\sigma(a) = v$,

$$\langle !a^p, \sigma, \mathcal{B} \rangle \longrightarrow \langle v, \sigma, \mathcal{B} \rangle$$

By Lemma A.4, $S = \star$.

By inversion on $\Sigma \vdash \sigma$, $\Sigma \vdash v : S'$ for some S'.

By further inversion, $S' = \text{ref}$ and $\emptyset; \Sigma \vdash v : \star$, which satisfies the theorem.

Case EUpdtRef: With $\sigma(a) = v'$,

$$\langle a :=^p v, \sigma, \mathcal{B} \rangle \longrightarrow \langle 0, \sigma[a \mapsto v], \mathcal{B} \rangle$$

By Lemma A.4, $\emptyset; \Sigma \vdash v : \star$ and $S \in \{\star, \text{int}\}$.

By THRef, $\Sigma \vdash v : \text{ref}$. By inversion on $\Sigma \vdash \sigma$, $\Sigma \vdash v' : S'$ for some S'.

By further inversion, $S' = \text{ref}$, and thus $\Sigma(a) = \text{ref}$.

By THeap, $\Sigma \vdash \sigma[a \mapsto v]$.

By TInt, $\emptyset; \Sigma \vdash 0 : \text{int}$.

By TSubsump, $\emptyset; \Sigma \vdash 0 : \star$.

Since $S \in \{\star, \text{int}\}$, theorem satisfied.

Case EAdd: With $n' = n_1 + n_2$,

$$\langle n_1 \ +^p \ n_2, \sigma, \mathcal{B}\rangle \longrightarrow \langle n', \sigma, \mathcal{B}\rangle$$

By Lemma A.4, $S \in \{\star, \text{int}\}$.

By TInt, $\emptyset; \Sigma \vdash n' : \text{int}$.

By TSubsump, $\emptyset; \Sigma \vdash n' : \star$.

Since $S \in \{\star, \text{int}\}$, theorem satisfied.

Cases ECheckFirst and ECheckHO: With *hastype*(σ, v, S), and for some $\mathcal{B}'$,

$$\langle v{\Downarrow}\langle S'; a; r\rangle, \sigma, \mathcal{B}\rangle \longrightarrow \langle v, \sigma, \mathcal{B}'\rangle$$

By Lemma A.4, $\emptyset; \Sigma \vdash v : \star$ and $\emptyset; \Sigma \vdash a : \star$ and $S \in \{\star, S'\}$.

By Lemma A.7, $\emptyset; \Sigma \vdash v : S'$. By TSubsump, $\emptyset; \Sigma \vdash v : \star$.

Since $S \in \{\star, S'\}$, theorem satisfied.

Case ECheckFail: is vacuous.

Cases ECastFirst and ECastHO: With *hastype*$(\sigma, v, \lfloor T_2 \rfloor)$, and for some $\mathcal{B}'$,

$$\langle v{::}T_1{\Rightarrow}^{\ell}T_2, \sigma, \mathcal{B}\rangle \longrightarrow \langle v, \sigma, \mathcal{B}'\rangle$$

By Lemma A.4, $\emptyset; \Sigma \vdash v : \lfloor T_1 \rfloor$ and $S \in \{\star, \lfloor T_2 \rfloor\}$.

By TSubsump, $\emptyset; \Sigma \vdash v : \star$.

By Lemma A.7, $\emptyset; \Sigma \vdash v : \lfloor T_2 \rfloor$. By TSubsump, $\emptyset; \Sigma \vdash v : \star$.

Since $S \in \{\star, \lfloor T_2 \rfloor\}$, theorem satisfied.

Case ECastFail: is vacuous.

□

Lemma A.10 (Canonical forms). *If* $\emptyset; \Sigma \vdash v : S$ *and* $\Sigma \vdash \sigma$*, then*

- *If* $S = \text{int}$*, then* $v = n$.
- *If* $S = {\rightarrow}$*, then* $v = a$ *and* $\sigma(a) = (\lambda x.e)$.
- *If* $S = \text{ref}$*, then* $v = a$ *and* $\sigma(a) = v'$.
- *If* $S = \star$*, then* $\exists S'$*,* $S \neq \star$*, such that* $\emptyset; \Sigma \vdash v : S'$.

Proof. By induction on $\emptyset; \Sigma \vdash v : S$. Most cases vacuous.

Case TSubsump:

$$\frac{\Gamma; \Sigma \vdash v : S'}{\Gamma; \Sigma \vdash v : \star}$$

If $S \neq \star$, case is vacuous. If $S' = \star$, then apply the IH with $\Gamma; \Sigma \vdash v : S'$.

Otherwise, $S' \neq \star$, and theorem satisfied.

Case TInt:

$$\Gamma; \Sigma \vdash n : \text{int}$$

If $S \neq$ int, case is vacuous. If $S =$ int, then $v = n$.

Case TAddr:

$$\frac{\Sigma(a) = S'}{\Gamma; \Sigma \vdash a : S'}$$

Since $\Sigma \vdash \sigma$, $\Sigma \vdash \sigma(a) : S'$.

Subcases on $\sigma(a)$.

Subcase : $\sigma(a) = (\lambda x.e)$

Then $S' = \rightarrow$. If $S \neq \rightarrow$, case is vacuous. Otherwise, theorem satisfied.

Subcase : $\sigma(a) = v$

Then $S' =$ ref. If $S \neq$ ref, case is vacuous. Otherwise, theorem satisfied.

□

Lemma A.11 (Progress). *If* $\emptyset; \Sigma \vdash e : S$ *and* $\Sigma \vdash \sigma$*, then either:*

- $\langle e, \sigma, \mathcal{B} \rangle \longmapsto \langle e', \sigma', \mathcal{B} \rangle$*, or*
- $\langle e, \sigma, \mathcal{B} \rangle \longmapsto \text{Blame}(\mathcal{L})$*, or*
- $e = v$*, or*
- $\langle e, \sigma, \mathcal{B} \rangle$ *stuck* ⧫.

Proof. By induction on $\emptyset; \Sigma \vdash e : S$.

Case TVar: is vacuous.

Cases TAddr and TInt: have $e = v$.

Case TSubsump:

$$\frac{\Gamma; \Sigma \vdash e : S}{\Gamma; \Sigma \vdash e : \star}$$

Immediate from the IH.

Case TAdd:

$$\frac{\Gamma; \Sigma \vdash e_1 : \text{int} \quad \Gamma; \Sigma \vdash e_2 : \text{int}}{\Gamma; \Sigma \vdash e_1 \ +^{\Diamond} \ e_2 : \text{int}}$$

By the IH, for both e_1 and e_2, either they are a value, they step (to either blame or another expression), or they are stuck (blaming ⧫). If either steps to another expression, then e steps. If either steps to blame, then e steps to the same blame. If either is stuck by ⧫, then e is stuck by ⧫. Otherwise, e_1 and e_2 are values.

If e_1 is a value, then by Lemma A.10, $e_1 = n_1$.

If e_2 is a value, then by Lemma A.10, $e_2 = n_2$.

Then by EAdd, $\langle e_1 \ +^{\Diamond} \ e_2, \sigma, \mathcal{B} \rangle \longrightarrow \langle n, \sigma, \mathcal{B} \rangle$, where $n = n_1 + n_2$.

Case TAdd-Dyn:

$$\frac{\Gamma; \Sigma \vdash e_1 : \star \quad \Gamma; \Sigma \vdash e_2 : \star}{\Gamma; \Sigma \vdash e_1 \ +^{\blacklozenge} \ e_2 : \text{int}}$$

By the IH, for both e_1 and e_2, either they are a value, they step (to either blame or another expression), or they are stuck (blaming ⧫). If either steps to another expression, then e steps. If either steps to blame, then e steps to the same blame. If either is stuck by ⧫, then e is stuck by ⧫. Otherwise, e_1 and e_2 are values.

If e_1 is a value, then either $e_1 = n$ or $e_1 = a$. If $e_1 = a$, then $\langle e_1 +^{\blacklozenge} e_2, \sigma, \mathcal{B} \rangle$ stuck ⧫. Suppose that $e_1 = n_1$. If e_2 is a value, then either $e_2 = n$ or $e_2 = a$. If $e_2 = a$, then $\langle e_1 \ +^{\blacklozenge} \ e_2, \sigma, \mathcal{B} \rangle$ stuck ⧫. Suppose that $e_2 = n_2$. Then by EAdd, $\langle e_1 \ +^{\blacklozenge} \ e_2, \sigma, \mathcal{B} \rangle \longrightarrow \langle n, \sigma, \mathcal{B} \rangle$, where $n = n_1 + n_2$.

Case TFun:

$$\frac{\Gamma, x : \star, f :\!\rightarrow; \Sigma \vdash e : \star}{\Gamma; \Sigma \vdash \texttt{fun}\ f\ x.\ e :\!\rightarrow}$$

Immediately have by EFun that $\langle \texttt{fun}\ f\ x.\ e, \sigma, \mathcal{B}\rangle \longrightarrow \langle a, \sigma[a \mapsto (\lambda x.e[a/f])], \mathcal{B}\rangle$ for fresh a.

Case TApp:

$$\frac{\Gamma; \Sigma \vdash e_1 :\rightarrow \quad \Gamma; \Sigma \vdash e_2 : \star}{\Gamma; \Sigma \vdash e_1\ e_2^{\Diamond} : \star}$$

By the IH, for both e_1 and e_2, either they are a value, they step (to either blame or another expression), or they are stuck (blaming ⧫). If either steps to another expression, then e steps. If either steps to blame, then e steps to the same blame. If either is stuck by ⧫, then e is stuck by ⧫. Otherwise, e_1 and e_2 are values.

If e_1 is a value, then by Lemma A.10, $v = a$ and $\sigma(a) = (\lambda x.e')$.

Then, with e_2 a value, by EApp $\langle e_1\ e_2^{\Diamond}, \sigma, \mathcal{B}\rangle \longrightarrow \langle e'[e_2/x], \sigma, \mathcal{B}\rangle$.

Case TApp-Dyn:

$$\frac{\Gamma; \Sigma \vdash e_1 : \star \quad \Gamma; \Sigma \vdash e_2 : \star}{\Gamma; \Sigma \vdash e_1\ e_2^{⧫} : \star}$$

By the IH, for both e_1 and e_2, either they are a value, they step (to either blame or another expression), or they are stuck (blaming ⧫). If either steps to another expression, then e steps. If either steps to blame, then e steps to the same blame. If either is stuck by ⧫, then e is stuck by ⧫. Otherwise, e_1 and e_2 are values. Suppose that e_2 is a value.

If e_1 is a value, then either $e_1 = n$ or $e_1 = a$. If $e_1 = n$, then $\langle e_1\ e_2^{⧫}, \sigma, \mathcal{B}\rangle$ stuck ⧫. Suppose that $e_1 = a$.

By Lemma A.10, $\exists S', S' \neq \star$, such that $\Gamma; \Sigma \vdash e_1 : S'$.

By Lemma A.4, $\Sigma(a) = S'$.

Since $\Sigma \vdash \sigma$, $\Sigma \vdash \sigma(a) : \Sigma(a)$.

Cases on $\sigma(a)$: if $\sigma(a) = v$, then $\langle e_1\ e_2^{⧫}, \sigma, \mathcal{B}\rangle$ stuck ⧫.

If $\sigma(a) = (\lambda x.e')$, then by EApp, $\langle e_1\ e_2^{⧫}, \sigma, \mathcal{B}\rangle \longrightarrow \langle e'[e_2/x], \sigma, \mathcal{B}\rangle$.

Case TRef:

$$\frac{\Gamma; \Sigma \vdash e' : \star}{\Gamma; \Sigma \vdash \texttt{ref}\ e' : \text{ref}}$$

By the IH, either e' is a value, it steps (to either blame or another expression), or it is stuck (blaming ⧫). If it to another expression, then e steps. If it steps to blame, then e steps to the same blame. If

it is stuck by ⧫, then e is stuck by ⧫. Otherwise, e' is a value. Suppose that e' is a value.

Immediately have by by ERef that $\langle \texttt{ref}\ e', \sigma, \mathcal{B}\rangle \longrightarrow \langle a, \sigma[a \mapsto e], \mathcal{B}\rangle$ for fresh a.

Case TDeref:

$$\frac{\Gamma; \Sigma \vdash e' : \text{ref}}{\Gamma; \Sigma \vdash !e'^{\Diamond} : \star}$$

By the IH, either e' is a value, it steps (to either blame or another expression), or it is stuck (blaming ⧫). If it to another expression, then e steps. If it steps to blame, then e steps to the same blame. If it is stuck by ⧫, then e is stuck by ⧫. Otherwise, e' is a value. Suppose that e' is a value.

By Lemma A.10, $v = a$ and $\sigma(a) = v'$.

By EDeref, $\langle !e'^{\Diamond}, \sigma, \mathcal{B}\rangle \longrightarrow \langle v', \sigma, \mathcal{B}\rangle$.

Case TDeref-Dyn:

$$\frac{\Gamma; \Sigma \vdash e' : \star}{\Gamma; \Sigma \vdash !e'^{\blacklozenge} : \star}$$

By the IH, either e' is a value, it steps (to either blame or another expression), or it is stuck (blaming ⧫). If it to another expression, then e steps. If it steps to blame, then e steps to the same blame. If it is stuck by ⧫, then e is stuck by ⧫. Otherwise, e' is a value. Suppose that e' is a value.

If e' is a value, then either $e' = n$ or $e' = a$. If $e' = n$, then $\langle !e'^{\blacklozenge}, \sigma, \mathcal{B}\rangle$ stuck ⧫. Suppose that $e' = a$.

By Lemma A.10, $\exists S', S' \neq \star$, such that $\Gamma; \Sigma \vdash e' : S'$.

By Lemma A.4, $\Sigma(a) = S'$.

Since $\Sigma \vdash \sigma$, $\Sigma \vdash \sigma(a) : \Sigma(a)$.

Cases on $\sigma(a)$: if $\sigma(a) = (\lambda x.e')$, then $\langle !e'^{\blacklozenge}, \sigma, \mathcal{B}\rangle$ stuck ⧫.

If $\sigma(a) = v$, then by EDeref, $\langle !e'^{\blacklozenge}, \sigma, \mathcal{B}\rangle \longrightarrow \langle v, \sigma, \mathcal{B}\rangle$.

Case TUpdtRef:

$$\frac{\Gamma; \Sigma \vdash e_1 : \text{ref} \quad \Gamma; \Sigma \vdash e_2 : \star}{\Gamma; \Sigma \vdash e_1 :=^{\Diamond} e_2 : \text{int}}$$

By the IH, for both e_1 and e_2, either they are a value, they step (to either blame or another expression), or they are stuck (blaming ⧫). If either steps to another expression, then e steps. If either

steps to blame, then e steps to the same blame. If either is stuck by $\blacklozenge$, then e is stuck by $\blacklozenge$. Otherwise, e_1 and e_2 are values.

If e_1 is a value, then by Lemma A.10, $v = a$ and $\sigma(a) = v$.

Then, with e_2 a value, by EUpdtRef $\langle e_1 :=^{\Diamond} e_2, \sigma, \mathcal{B}\rangle \longrightarrow \langle 0, \sigma[a \mapsto e_2], \mathcal{B}\rangle$.

Case TUpdtRef-Dyn:

$$\frac{\Gamma; \Sigma \vdash e_1 : \star \quad \Gamma; \Sigma \vdash e_2 : \star}{\Gamma; \Sigma \vdash e_1 :=^{\blacklozenge} e_2 : \text{int}}$$

By the IH, for both e_1 and e_2, either they are a value, they step (to either blame or another expression), or they are stuck (blaming $\blacklozenge$). If either steps to another expression, then e steps. If either steps to blame, then e steps to the same blame. If either is stuck by $\blacklozenge$, then e is stuck by $\blacklozenge$. Otherwise, e_1 and e_2 are values. Suppose that e_2 is a value.

If e_1 is a value, then either $e_1 = n$ or $e_1 = a$. If $e_1 = n$, then $\langle e_1 :=^{\blacklozenge} e_2, \sigma, \mathcal{B}\rangle$ stuck $\blacklozenge$. Suppose that $e_1 = a$.

By Lemma A.10, $\exists S', S' \neq \star$, such that $\Gamma; \Sigma \vdash e_1 : S'$.

By Lemma A.4, $\Sigma(a) = S'$.

Since $\Sigma \vdash \sigma$, $\Sigma \vdash \sigma(a) : \Sigma(a)$.

Cases on $\sigma(a)$: if $\sigma(a) = (\lambda x.e')$, then $\langle e_1 :=^{\blacklozenge} e_2, \sigma, \mathcal{B}\rangle$ stuck $\blacklozenge$.

If $\sigma(a) = v$, then by EUpdtRef, $\langle e_1 :=^{\blacklozenge} e_2, \sigma, \mathcal{B}\rangle \longrightarrow \langle 0, \sigma[a \mapsto e_2], \mathcal{B}\rangle$.

Case TCheck:

$$\frac{\Gamma; \Sigma \vdash e_1 : \star \quad \Gamma; \Sigma \vdash e_2 : \mathit{tagtype}(r)}{\Gamma; \Sigma \vdash e_1 \Downarrow \langle S; e_2; r\rangle : S}$$

By the IH, for both e_1 and e_2, either they are a value, they step (to either blame or another expression), or they are stuck (blaming $\blacklozenge$). If either steps to another expression, then e steps. If either steps to blame, then e steps to the same blame. If either is stuck by $\blacklozenge$, then e is stuck by $\blacklozenge$. Otherwise, e_1 and e_2 are values.

By definition of *tagtype*, either $\Gamma; \Sigma \vdash e_2 :\rightarrow$ or $\Gamma; \Sigma \vdash e_2 :$ ref. In either case, by Lemma A.10 have that $e_2 = a$.

Let $\mathcal{B}'$ be defined as $\mathcal{B}$ if $e_1 = n$, and $\varrho(\mathcal{B}, a', \langle a, r\rangle)$ if $e_1 = a'$.

We proceed by subcases on $\mathit{hastype}(\sigma, e_1, S)$.

Subcase : *hastype*(σ, v, S).

Then we have $\langle e_1 \Downarrow \langle S; e_2; r\rangle, \sigma, \mathcal{B}\rangle \longrightarrow \langle e_1, \sigma, \mathcal{B}'\rangle$ by ECheckFirst (if $e_1 = n$) or ECheckHO (if $e_1 = a'$).

Subcase : $\neg($*hastype*$(\sigma, v, S))$.

Then by ECheckBlame,

$\langle e_1 \Downarrow \langle S; e_2; r\rangle, \sigma, \mathcal{B}\rangle \longrightarrow$ *blame*$(\sigma, v, a, \mathcal{B},)$.

Case TCast:

$$\frac{\Gamma; \Sigma \vdash e' : \lfloor T_1 \rfloor \quad T_1 \sim T_2}{\Gamma; \Sigma \vdash e' :: T_1 \Rightarrow^{\ell} T_2 : \lfloor T_2 \rfloor}$$

By the IH, either e' is a value, it steps (to either blame or another expression), or it is stuck (blaming ⧫). If it to another expression, then e steps. If it steps to blame, then e steps to the same blame. If it is stuck by ⧫, then e is stuck by ⧫. Otherwise, e' is a value. Suppose that e' is a value.

Let $\mathcal{B}'$ be defined as $\mathcal{B}$ if $e_1 = n$, and $\varrho(\mathcal{B}, a', [\![T_1 \Rightarrow^{\ell} T_2]\!])$ if $e_1 = a$.

We proceed by subcases on *hastype*$(\sigma, e', \lfloor T_2 \rfloor)$.

Subcase : *hastype*$(\sigma, e_1, \lfloor T_2 \rfloor)$. Then we have $\langle e' :: T_1 \Rightarrow^{\ell} T_2, \sigma, \mathcal{B}\rangle \longrightarrow \langle e', \sigma, \mathcal{B}'\rangle$ by ECastFirst (if $e_1 = n$) or ECastHO (if $e_1 = a$).

Subcase : $\neg($*hastype*$(\sigma, v, S))$. Then by ECheckBlame,

$\langle e_1 \Downarrow \langle S; e_2; r\rangle, \sigma, \mathcal{B}\rangle \longrightarrow$ Blame$(\{\ell\})$.

□

Lemma A.12 (Composition). *If* $\vdash \mathcal{C} : \Gamma; S \Rightarrow \Gamma'; S'$ *and* $\Gamma; \emptyset \vdash e : S$*, then* $\Gamma; \emptyset \vdash C[e] : S'$.

Proof. By induction on $\vdash \mathcal{C} : \Gamma; S \Rightarrow \Gamma'; S'$.

Case CxHole:

$$\frac{}{\vdash \Box : \Gamma; S \Rightarrow \Gamma; S}$$

Since $\Box[e] = e$, have that $\Gamma; \emptyset \vdash e : S$.

Case CxSubsump:

$$\frac{\vdash \mathcal{C} : \Gamma; S_1 \Rightarrow \Gamma'; S_2}{\vdash \mathcal{C} : \Gamma; S_1 \Rightarrow \Gamma'; \star}$$

By the IH, $\Gamma; \emptyset \vdash \mathcal{C}[e] : S_2$. By TSubsump, $\Gamma; \emptyset \vdash \mathcal{C}[e] : \star$

Case CxAddL:

$$\frac{\vdash \mathcal{C} : \Gamma; S \Rightarrow \Gamma'; \star \qquad \Gamma'; \emptyset \vdash e' : \star}{\vdash \mathcal{C} \ +^{\blacklozenge} \ e' : \Gamma; S \Rightarrow \Gamma'; \text{int}}$$

By the IH, $\Gamma; \emptyset \vdash \mathcal{C}[e] : \star$. By TAdd-Dyn, $\Gamma; \emptyset \vdash \mathcal{C}[e] \ +^{\blacklozenge} \ e' : \text{int}$.

Remaining cases are similar. □

Lemma A.13 (Open world soundness). *If* $\Gamma \vdash e_s \rightsquigarrow e : T$ *and* $\vdash \mathcal{C} : \lfloor \Gamma \rfloor; \lfloor T \rfloor \Rightarrow \emptyset; S$*, then* $\emptyset; \emptyset \vdash \mathcal{C}[e] : S$ *and either:*

- $\langle \mathcal{C}[e], \emptyset, \emptyset \rangle \longrightarrow^* \langle v, \sigma, \mathcal{B} \rangle$ *and* $\emptyset; \Sigma \vdash v : S$ *and* $\Sigma \vdash \sigma$*, or*
- $\langle \mathcal{C}[e], \emptyset, \emptyset \rangle \longrightarrow^* \text{Blame}(\mathcal{L})$*, or*
- $\langle \mathcal{C}[e], \emptyset, \emptyset \rangle \longrightarrow^* \langle e', \sigma, \mathcal{B} \rangle$ *and* $\langle e', \sigma, \mathcal{B} \rangle$ *stuck* ⧫*, or*
- *for all* ς *such that* $\langle \mathcal{C}[e], \emptyset, \emptyset \rangle \longrightarrow^* \varsigma$*, have that* $\varsigma = \langle e', \sigma, \mathcal{B} \rangle$ *and exists* ς' *such that* $\langle e', \sigma, \mathcal{B} \rangle \longrightarrow \varsigma'$.

Proof. By Lemma A.3, $\lfloor \Gamma \rfloor; \emptyset \vdash e : \lfloor T \rfloor$. By Lemma A.12, $\emptyset; \emptyset \vdash \mathcal{C}[e] : S$.
Suppose that for all ς such that $\langle \mathcal{C}[e], \emptyset, \emptyset \rangle \longrightarrow^* \varsigma$, have that $\varsigma = \langle e', \sigma, \mathcal{B} \rangle$ and exists ς' such that $\langle e', \sigma, \mathcal{B} \rangle \longrightarrow \varsigma'$. Then the theorem is satisfied.
Otherwise, there exists some ς such that $\varsigma \neq \langle e', \sigma, \mathcal{B} \rangle$ or there is no ς' such that $\langle e', \sigma, \mathcal{B} \rangle \longrightarrow \varsigma'$.
If $\varsigma \neq \langle e', \sigma, \mathcal{B} \rangle$, then $\varsigma = \text{Blame}(\mathcal{L})$.
Otherwise, $\langle \mathcal{C}[e], \emptyset, \emptyset \rangle \longrightarrow^* \langle e', \sigma, \mathcal{B} \rangle$ and $\langle e', \sigma, \mathcal{B} \rangle \not\longrightarrow \varsigma'$. By repeating Lemma A.9, $\emptyset; \Sigma \vdash e' : S$ and $\Sigma \vdash \sigma$. By Lemma A.11, either $e' = v$ or $\langle e', \sigma, \mathcal{B} \rangle$ stuck ⧫. □

2.2. Blame.

Lemma A.14. *For all* T_1, T_2, ℓ*,* $[\![T_1 \Leftrightarrow^{\ell} T_2]\!]$ *safe iff* $T_1 = T_2$.

Proof. First we prove that if $T_1 = T_2$, then $[\![T_1 \Leftrightarrow^{\ell} T_2]\!]$ safe ℓ by trivial induction on $T_1 <:_b T_2$.

Example Case: $T_{11} \to T_{12} = T_{21} \to T_{22}$

Have $[\![T_{11} \to T_{12} \Leftrightarrow^{\ell} T_{21} \to T_{22}]\!] = [\![T_{21} \Leftrightarrow^{\ell} T_{11}]\!] \to^{\epsilon} [\![T_{12} \Leftrightarrow^{\ell} T_{22}]\!]$.

By IH, $[\![T_{21} \Leftrightarrow^{\ell} T_{11}]\!]$ safe ℓ since $T_{21} = T_{11}$.

By IH, $[\![T_{12} \Leftrightarrow^{\ell} T_{22}]\!]$ safe ℓ since $T_{12} = T_{22}$.

Hence $[\![T_{21} \Leftrightarrow^{\ell} T_{11}]\!] \to^{\epsilon} [\![T_{12} \Leftrightarrow^{\ell} T_{22}]\!]$ safe ℓ.

Next we prove that if $[\![T_1 \Leftrightarrow^{\ell} T_2]\!]$ safe , then $T_1 = T_2$ by induction on $[\![T_1 \Leftrightarrow^{\ell} T_2]\!]$ safe

Case LInt:

$$\frac{q \neq \ell}{\text{int}^q \text{ safe } \ell}$$

There are three ℓ-labeled casts that compile to int^q:

Subcase : $[\![\text{int} \Leftrightarrow^{\ell} \text{int}]\!]$:

Have that int $=$ int.

Subcases : $[\![\text{int} \Leftrightarrow^{\ell} \star]\!]$ and $[\![\star \Leftrightarrow^{\ell} \text{int}]\!]$:

Then $q = \ell$, which is contradictory.

Case LBot: is vacuous.

Case LDyn:

$$\frac{}{\star \text{ safe } \ell}$$

The only ℓ-labeled cast that compiles to $\star$ is $\star \Leftrightarrow^{\ell} \star$. Have that $\star = \star$.

Case LFunc:

$$\frac{q \neq \ell \qquad L_1 \text{ safe } \ell \qquad L_2 \text{ safe } \ell}{L_1 \to^q L_2 \text{ safe } \ell}$$

There are three ℓ-labeled casts that compile to $L_1 \to^q L_2$:

Subcase : $[\![T_1 \to T_2 \Leftrightarrow^{\ell} T_3 \to T_4]\!]$:

Then $L_1 = [\![T_3 \Leftrightarrow^{\ell} T_1]\!]$ and $L_2 = [\![T_2 \Leftrightarrow^{\ell} T_4]\!]$.

By the IH, $T_3 = T_1$.

By the IH, $T_2 = T_4$.

Thus $T_1 \to T_2 = T_3 \to T_4$.

Subcases : $[\![T_1 \to T_2 \Leftrightarrow^\ell \star]\!]$ and $[\![\star \Leftrightarrow^\ell T_1 \to T_2]\!]$:

Then $q = \ell$, which is contradictory.

Case LRef:

$$\frac{q \neq \ell \qquad L \text{ safe } \ell}{\text{ref}^q\ L \text{ safe } \ell}$$

There are three ℓ-labeled casts that compile to $\text{ref}^q\ L$:

Subcase : $[\![\text{ref}\ T_1 \Leftrightarrow^\ell \text{ref}\ T_2]\!]$:

Then $L = [\![T_2 \Leftrightarrow^\ell T_1]\!]$.

By the IH, $T_1 = T_2$.

Thus $\text{ref}\ T_1 = \text{ref}\ T_2$.

Subcases : $[\![\text{ref}\ T \Leftrightarrow^\ell \star]\!]$ and $[\![\star \Leftrightarrow^\ell T_1 \to T_2]\!]$:

Then $q = \ell$, which is contradictory.

□

Lemma A.15. *For all* T_1, T_2, ℓ, $T_1 <:_b T_2$ *iff* $[\![T_1 \Rightarrow^\ell T_2]\!]$ *safe* ℓ.

Proof. First we prove that if $T_1 <:_b T_2$, then $[\![T_1 \Rightarrow^\ell T_2]\!]$ safe ℓ by induction on $T_1 <:_b T_2$.

Case SIntDyn:

$$\frac{}{\text{int} <:_b \star}$$

$[\![\text{int} \Rightarrow^\ell \star]\!] = \text{int}^\epsilon$, and int^ϵ safe ℓ.

Case SFuncDyn:

$$\frac{T_1 \to T_2 <:_b \star \to \star}{T_1 \to T_2 <:_b \star}$$

Have $[\![T_1 \to T_2 \Rightarrow^\ell \star]\!] = [\![\star \Rightarrow^\ell T_1]\!] \to^\epsilon [\![T_2 \Rightarrow^\ell T_1]\!]$.

Also, $[\![T_1 \to T_2 \Rightarrow^\ell \star \to \star]\!] = [\![\star \Rightarrow^\ell T_1]\!] \to^\epsilon [\![T_2 \Rightarrow^\ell T_1]\!]$.

By the IH, $[\![T_1 \to T_2 \Rightarrow^\ell \star \to \star]\!]$ safe ℓ.

Case SRefDyn:

$$\frac{\text{ref}\ T <:_b \text{ref}\ \star}{\text{ref}\ T <:_b \star}$$

Have $[\![\text{ref}\, T \Rightarrow^{\ell} \star]\!] = \text{ref}^{\epsilon}\, [\![\star \Rightarrow^{\ell} T]\!]$.

Also, $[\![\text{ref}\, T \Rightarrow^{\ell} \text{ref}\, \star]\!] = \text{ref}^{\epsilon}\, [\![\star \Rightarrow^{\ell} T]\!]$.

By the IH, $[\![\text{ref}\, T \Rightarrow^{\ell} \text{ref}\, \star]\!]$ safe ℓ.

Case SIntInt:

$$\frac{}{\text{int} <:_b \text{int}}$$

$[\![\text{int} \Rightarrow^{\ell} \text{int}]\!] = \text{int}^{\epsilon}$, and int^{ϵ} safe ℓ.

Case SFuncFunc:

$$\frac{T_3 <:_b T_1 \qquad T_2 <:_b T_4}{T_1 \to T_2 <:_b T_3 \to T_4}$$

Have $[\![T_1 \to T_2 \Rightarrow^{\ell} T_3 \to T_4]\!] = [\![T_3 \Rightarrow^{\ell} T_1]\!] \to^{\epsilon} [\![T_2 \Rightarrow^{\ell} T_4]\!]$.

By the IH, $[\![T_3 \Rightarrow^{\ell} T_1]\!]$ safe ℓ.

By the IH, $[\![T_2 \Rightarrow^{\ell} T_4]\!]$ safe ℓ.

Therefore $[\![T_3 \Rightarrow^{\ell} T_1]\!] \to^{\epsilon} [\![T_2 \Rightarrow^{\ell} T_1]\!]$ safe ℓ.

Case SRefRef:

$$\frac{}{\text{ref}\, T <:_b \text{ref}\, T}$$

Have $[\![\text{ref}\, T \Rightarrow^{\ell} \text{ref}\, T]\!] = \text{ref}^{\epsilon}\, [\![T \Leftrightarrow^{\ell} T]\!]$.

By Lemma A.14, $[\![T \Leftrightarrow^{\ell} T]\!]$ safe ℓ.

Therefore $\text{ref}^{\epsilon}\, [\![T \Leftrightarrow^{\ell} T]\!]$ safe ℓ.

We now prove that if $[\![T_1 \Rightarrow^{\ell} T_2]\!]$ safe ℓ, then $T_1 <:_b T_2$ by induction on $[\![T_1 \Rightarrow^{\ell} T_2]\!]$ safe ℓ.

Case LInt:

$$\frac{q \neq \ell}{\text{int}^{q} \text{ safe } \ell}$$

There are three ℓ-labeled casts that compile to int^{q}:

Subcase : $[\![\text{int} \Rightarrow^{\ell} \text{int}]\!]$:

Have that $\text{int} <:_b \text{int}$.

Subcase : $[\![\text{int} \Rightarrow^{\ell} \star]\!]$:

Have that $\text{int} <:_b \star$.

Subcase : $[\![\star \Rightarrow^{\ell} \text{int}]\!]$:

Then $q = \ell$, which is contradictory.

Case LBot: is vacuous.

Case LDyn:

$$\frac{}{\star \text{ safe } \ell}$$

The only ℓ-labeled cast that compiles to $\star$ is $\star \Rightarrow^{\ell} \star$. Have that $\star <:_b \star$.

Case LFunc:

$$\frac{q \neq \ell \qquad L_1 \text{ safe } \ell \qquad L_2 \text{ safe } \ell}{L_1 \to^q L_2 \text{ safe } \ell}$$

There are three ℓ-labeled casts that compile to $L_1 \to^q L_2$:

Subcase : $[\![T_1 \to T_2 \Rightarrow^{\ell} T_3 \to T_4]\!]$:

Then $L_1 = [\![T_3 \Rightarrow^{\ell} T_1]\!]$ and $L_2 = [\![T_2 \Rightarrow^{\ell} T_4]\!]$.

By the IH, $T_3 <:_b T_1$.

By the IH, $T_2 <:_b T_4$.

Thus $T_1 \to T_2 <:_b T_3 \to T_4$.

Subcase : $[\![T_1 \to T_2 \Rightarrow^{\ell} \star]\!]$:

Then $L_1 = [\![\star \Rightarrow^{\ell} T_1]\!]$ and $L_2 = [\![T_2 \Rightarrow^{\ell} \star]\!]$.

By the IH, $\star <:_b T_1$.

By the IH, $T_2 <:_b \star$.

Hence $T_1 \to T_2 <:_b \star \to \star$.

Thus $T_1 \to T_2 <:_b \star$.

Subcase : $[\![\star \Rightarrow^{\ell} T_1 \to T_2]\!]$:

Then $q = \ell$, which is contradictory.

Case LRef:

$$\frac{q \neq \ell \qquad L \text{ safe } \ell}{\text{ref}^q\ L \text{ safe } \ell}$$

There are three ℓ-labeled casts that compile to $\text{ref}^q\ L$:

Subcase : $[\![\text{ref}\, T_1 \Rightarrow^{\ell} \text{ref}\, T_2]\!]$:

Then $L = [\![T_2 \Leftrightarrow^{\ell} T_1]\!]$.

By the IH, $T_1 = T_2$.

Thus ref $T_1 <:_b$ ref T_2.

Subcase : $[\![\text{ref } T \Rightarrow^\ell \star]\!]$:

Then $L = [\![\star \Leftrightarrow^\ell T]\!]$.

By the IH, $\star = T$.

Hence ref $T <:_b$ ref $\star$.

Thus ref $T <:_b \star$.

Subcase : $[\![\star \Rightarrow^\ell T_1 \to T_2]\!]$:

Then $q = \ell$, which is contradictory.

□

Lemma A.16. *If $\mathcal{B} \vdash b$ safe ℓ, then $\varrho(\mathcal{B}, a, b) \vdash b$ safe ℓ.*

Proof. By induction on $\mathcal{B} \vdash b$ safe ℓ.

Case SfLType:

$$\frac{L \text{ safe } \ell}{\mathcal{B} \vdash L \text{ safe } \ell}$$

Since $\mathcal{B}$ not used and L safe ℓ, $\varrho(\mathcal{B}, a, L) \vdash L$ safe ℓ.

Case SfPtr:

$$\frac{\forall b' \in \mathcal{B}(a') \setminus \{\langle a', r\rangle\},\ \mathcal{B} \vdash b' \text{ safe } \ell}{\mathcal{B} \vdash \langle a', r\rangle \text{ safe } \ell}$$

Subcases on $a = a'$.

Subcase : $a = a'$

Then $\forall b' \in \mathcal{B}(a) \setminus \{\langle a, r\rangle\},\ \mathcal{B} \vdash b'$ safe ℓ.

Have that $b = \langle a', r\rangle$.

$$\begin{aligned}\varrho(\mathcal{B}, a, \langle a, r\rangle)(a) \setminus \{\langle a, r\rangle\} &= (\mathcal{B}(a) \cup \{\langle a, r\rangle\}) \setminus \{\langle a, r\rangle\} \\ &= \mathcal{B}(a) \setminus \{\langle a, r\rangle\}\end{aligned}$$

By the IH, $\forall b' \in \mathcal{B}(a) \setminus \{\langle a, r\rangle\},\ \varrho(\mathcal{B}, a, b) \vdash b'$ safe ℓ.

Therefore, $\varrho(\mathcal{B}, a, b) \vdash b$ safe ℓ.

Subcase : $a \neq a'$

Then $\mathcal{B}(a') = \varrho(\mathcal{B}, a, b)(a')$.

By the IH, $\forall b' \in \mathcal{B}(a') \setminus \{\langle a', r\rangle\},\ \varrho(\mathcal{B}, a, b) \vdash b'$ safe ℓ.

Therefore, $\varrho(\mathcal{B}, a, b) \vdash b$ safe ℓ.

□

Lemma A.17. *If $\mathcal{B} \vdash b$ safe ℓ and $\mathcal{B} \vdash b'$ safe ℓ, then $\varrho(\mathcal{B}, a, b') \vdash b$ safe ℓ.*

Proof. By induction on $\mathcal{B} \vdash b$ safe ℓ.

Case SfLType:

$$\frac{L \text{ safe } \ell}{\mathcal{B} \vdash L \text{ safe } \ell}$$

Since $\mathcal{B}$ not used and L safe ℓ, $\varrho(\mathcal{B}, a, b') \vdash L$ safe ℓ.

Case SfPtr:

$$\frac{\forall b'' \in \mathcal{B}(a') \setminus \{\langle a', r\rangle\},\ \mathcal{B} \vdash b'' \text{ safe } \ell}{\mathcal{B} \vdash \langle a', r\rangle \text{ safe } \ell}$$

Subcases on $a = a'$.

Subcase : $a = a'$

Then $\forall b'' \in \mathcal{B}(a) \setminus \{\langle a, r\rangle\},\ \mathcal{B} \vdash b''$ safe ℓ.

By the IH, $\forall b'' \in \mathcal{B}(a) \setminus \{\langle a, r\rangle\},\ \varrho(\mathcal{B}, a, b') \vdash b''$ safe ℓ

By Lemma A.16, $\varrho(\mathcal{B}, a, b') \vdash b'$ safe ℓ.

Therefore, $\varrho(\mathcal{B}, a, b') \vdash b$ safe ℓ.

Subcase : $a \neq a'$

Then $\mathcal{B}(a') = \varrho(\mathcal{B}, a, b)(a')$.

By the IH, $\forall b'' \in \mathcal{B}(a') \setminus \{\langle a', r\rangle\},\ \varrho(\mathcal{B}, a, b') \vdash b''$ safe ℓ

Therefore, $\varrho(\mathcal{B}, a, b') \vdash b$ safe ℓ.

□

Lemma A.18. *If $\mathcal{B} \vdash e$ safe ℓ and $\mathcal{B} \vdash b$ safe ℓ, then $\varrho(\mathcal{B}, a, b) \vdash e$ safe ℓ.*

Proof. By straightforward induction on $\mathcal{B} \vdash e$ safe ℓ. Only interesting case:

Case SfAddr:

$$\frac{\forall b' \in \mathcal{B}(a'),\ \mathcal{B} \vdash b' \text{ safe } \ell}{\mathcal{B} \vdash a' \text{ safe } \ell}$$

By Lemma A.17, $\forall b' \in \varrho(\mathcal{B}, a, b)(a'),\ \varrho(\mathcal{B}, a, b) \vdash b'$ safe ℓ.

Therefore $\varrho(\mathcal{B}, a, b) \vdash a'$ safe ℓ

□

Lemma A.19. *If $\mathcal{B} \vdash \sigma$ safe ℓ and $\mathcal{B} \vdash b$ safe ℓ, then $\varrho(\mathcal{B}, a, b) \vdash \sigma$ safe ℓ.*

Proof. By straightforward induction on $\mathcal{B} \vdash \sigma$ safe ℓ, using Lemma A.18. □

Lemma A.20. *If $\mathcal{B} \vdash e_1$ safe ℓ and $\mathcal{B} \vdash e_2$ safe ℓ, then $\mathcal{B} \vdash e_1[e_2/x]$ safe .*

Proof. By straightforward induction on $\mathcal{B} \vdash e_1$ safe ℓ. □

Lemma A.21. *If $\mathcal{B} \vdash e$ safe ℓ and $\mathcal{B} \vdash \sigma$ safe ℓ and $\langle e, \sigma, \mathcal{B}\rangle \longrightarrow \langle e', \sigma', \mathcal{B}'\rangle$, then $\mathcal{B}' \vdash e'$ safe ℓ and $\mathcal{B}' \vdash \sigma'$ safe ℓ.*

Proof. By induction on $\langle e, \sigma, \mathcal{B}\rangle \longrightarrow \langle e', \sigma', \mathcal{B}'\rangle$.

Case EFun: With fresh(a),

$$\langle \texttt{fun}\ f\ x.\ e, \sigma, \mathcal{B}\rangle \longrightarrow \langle a, \sigma[a \mapsto (\lambda x.e[a/f])], \mathcal{B}\rangle$$

Since $\mathcal{B} \vdash \texttt{fun}\ f\ x.\ e$ safe ℓ, have $\mathcal{B} \vdash e$ safe ℓ.

Since a fresh, $\mathcal{B}(a) = \emptyset$. Thus by SfAddr, $\mathcal{B} \vdash a$ safe ℓ.

By Lemma A.20, $\mathcal{B} \vdash e[a/f]$ safe ℓ.

By SfHeapClosure, $\mathcal{B} \vdash \sigma[a \mapsto (\lambda x.e[a/f])]$ safe ℓ.

Case EApp: Where $\sigma(a) = (\lambda x.e)$,

$$\langle a\ v^p, \sigma, \mathcal{B}\rangle \longrightarrow \langle e[v/x], \sigma, \mathcal{B}\rangle$$

Since $\mathcal{B} \vdash \sigma$ safe ℓ, $\mathcal{B} \vdash e$ safe ℓ.

By inversion, $\mathcal{B} \vdash v$ safe ℓ.

By Lemma A.20, $\mathcal{B} \vdash e[v/x]$ safe $\mathcal{B}$.

Case ERef: With fresh(a),

$$\langle \texttt{ref}\ v, \sigma, \mathcal{B}\rangle \longrightarrow \langle a, \sigma[a \mapsto v], \mathcal{B}\rangle$$

Since $\mathcal{B} \vdash \texttt{ref}\ v$ safe ℓ, have $\mathcal{B} \vdash v$ safe ℓ.

Since a fresh, $\mathcal{B}(a) = \emptyset$. Thus by SfAddr, $\mathcal{B} \vdash a$ safe ℓ.

By SfHeapCell, $\mathcal{B} \vdash \sigma[a \mapsto v]$ safe ℓ.

Case EDeref: With $\sigma(a) = v$,

$$\langle !a^p, \sigma, \mathcal{B}\rangle \longrightarrow \langle v, \sigma, \mathcal{B}\rangle$$

Since $\mathcal{B} \vdash \sigma$ safe ℓ, $\mathcal{B} \vdash v$ safe ℓ.

Case EUpdtRef: With $\sigma(a) = v'$,

$$\langle a :=^p v, \sigma, \mathcal{B}\rangle \longrightarrow \langle 0, \sigma[a \mapsto v], \mathcal{B}\rangle$$

Since $\mathcal{B} \vdash a :=^p v$ safe ℓ, have $\mathcal{B} \vdash v$ safe ℓ.

By SfHeapCell, $\mathcal{B} \vdash \sigma[a \mapsto v]$ safe ℓ.

By SfInt, $\mathcal{B} \vdash 0$ safe ℓ.

Case EAdd: With $n' = n_1 + n_2$,

$$\langle n_1\ +^p\ n_2, \sigma, \mathcal{B}\rangle \longrightarrow \langle n', \sigma, \mathcal{B}\rangle$$

By SfInt, $\mathcal{B} \vdash n'$ safe ℓ.

Case ECheckFirst: With *hastype*(σ, v, S) and $v \neq a$,

$$\langle v \Downarrow \langle S'; a; r\rangle, \sigma, \mathcal{B}\rangle \longrightarrow \langle v, \sigma, \mathcal{B}\rangle$$

By inversion, $\mathcal{B} \vdash v$ safe ℓ.

Case ECheckHO: With *hastype*(σ, a', S),

$$\langle a' \Downarrow \langle S'; a; r\rangle, \sigma, \mathcal{B}\rangle \longrightarrow \langle a', \sigma, \varrho(\mathcal{B}, a', \langle a, r\rangle)\rangle$$

By inversion $\mathcal{B} \vdash a$ safe ℓ and $\mathcal{B} \vdash a'$ safe ℓ.

Therefore, for all $b \in \mathcal{B}(a)$, $\mathcal{B} \vdash b$ safe ℓ.

Hence $\mathcal{B} \vdash \langle a, r \rangle$ safe ℓ.

By Lemma A.18, $\varrho(\mathcal{B}, a', \langle a, r \rangle) \vdash a'$ safe ℓ.

By Lemma A.19, $\varrho(\mathcal{B}, a', \langle a, r \rangle) \vdash \sigma$ safe ℓ.

Case ECheckFail: is vacuous.

Case ECastFirst: With *hastype*$(\sigma, v, \lfloor T_2 \rfloor)$ and $v \neq a$,

$$\langle v :: T_1 \Rightarrow^{\ell'} T_2, \sigma, \mathcal{B} \rangle \longrightarrow \langle v, \sigma, \mathcal{B} \rangle$$

By inversion $\mathcal{B} \vdash v$ safe ℓ.

Case ECastHO: With *hastype*$(\sigma, a, \lfloor T_2 \rfloor)$,

$$\langle a :: T_1 \Rightarrow^{\ell'} T_2, \sigma, \mathcal{B} \rangle \longrightarrow \langle a, \sigma, \varrho(\mathcal{B}, a, [\![T_1 \Rightarrow^{\ell'} T_2]\!]) \rangle$$

Immediately, $[\![T_1 \Rightarrow^{\ell'} T_2]\!]$ safe ℓ.

Therefore $\mathcal{B} \vdash [\![T_1 \Rightarrow^{\ell'} T_2]\!]$ safe ℓ.

By inversion $\mathcal{B} \vdash a$ safe ℓ.

By Lemma A.18, $\varrho(\mathcal{B}, a, [\![T_1 \Rightarrow^{\ell'} T_2]\!]) \vdash a$ safe ℓ.

By Lemma A.19, $\varrho(\mathcal{B}, a, [\![T_1 \Rightarrow^{\ell'} T_2]\!]) \vdash \sigma$ safe ℓ.

Case ECastFail: is vacuous.

□

Lemma A.22. *If L safe ℓ, then extract$(\overline{r}, L)$ safe ℓ.*

Proof. By straightforward induction on *extract*$(\overline{r}, L)$, using inversion on the safe relation. □

Lemma A.23. *If $\mathcal{B} \vdash b$ safe ℓ and collectblame$(\overline{r}, \mathcal{B}, b) = \overline{L}$, then $\ell \notin \{$label$(L) \mid L \in \overline{L}\}$.*

Proof. By induction on *collectblame*$(\overline{r}, \mathcal{B}, b) = \overline{L}$.

Case:

$$\frac{\textit{extract}(\overline{r}, L) = L' \qquad \textit{label}(L') = \ell'}{\textit{collectblame}(\overline{r}, \mathcal{B}, L) = \{L'\}}$$

Since $b = L$ and $\mathcal{B} \vdash L$ safe ℓ, L safe ℓ. By Lemma A.22, L' safe ℓ. Therefore *label*$(L') \neq \ell$.

Case:

$$\frac{\mathit{extract}(\overline{r}, L) = L' \qquad \mathit{label}(L') = \epsilon}{\mathit{collectblame}(\overline{r}, \mathcal{B}, L) = \emptyset}$$

Trivially, $\ell \notin \{\mathit{label}(L) \mid L \in \emptyset\}$.

Case:

$$\frac{}{\mathit{collectblame}(\overline{r}, \mathcal{B}, \langle a, r\rangle) = \cup_{b' \in \mathcal{B}(a)}\, \mathit{collectblame}((r;\overline{r}), \mathcal{B}, b')}$$

Since $b = \langle a, r\rangle$ and $\mathcal{B} \vdash \langle a, r\rangle$ safe ℓ, $\forall b' \in \mathcal{B}(a)$, $\mathcal{B} \vdash b'$ safe ℓ.

By the IH, for each a', $\ell \notin \{\mathit{label}(L) \mid L \in \mathit{collectblame}((r;\overline{r}), \mathcal{B}, b')\}$.

Hence $\ell \notin \{\mathit{label}(L) \mid L \in \cup_{b' \in \mathcal{B}(a)}\, \mathit{collectblame}((r;\overline{r}), \mathcal{B}, b')\}$.

□

Lemma A.24. *Have that* $\mathit{resolve}(\sigma, v, \overline{L}) \subseteq \{\mathit{label}(L) \mid L \in \overline{L}\}$.

Proof. Straightforward induction on $\mathit{resolve}(\sigma, v, \overline{L})$. □

Lemma A.25. *If* $\mathcal{B} \vdash a$ *safe* ℓ *and* $\mathit{blame}(\sigma, v, a, r, \mathcal{B}) = \text{Blame}(\mathcal{L})$*, then* $\ell \notin \mathcal{L}$.

Proof. Have

$$\frac{\overline{L} = \cup_{b \in \mathcal{B}(a)}\, \mathit{collectblame}(r, \mathcal{B}, b) \qquad \mathcal{L} = \mathit{resolve}(\sigma, v, \overline{L})}{\mathit{blame}(\sigma, v, a, r, \mathcal{B}) = \text{Blame}(\mathcal{L})}$$

Since $\mathcal{B} \vdash a$ safe ℓ, have that for all $b \in \mathcal{B}(a)$, $\mathcal{B} \vdash b$ safe ℓ.

By Lemma A.23, $\ell \notin \{\mathit{label}(L) \mid L \in \overline{L}\}$.

By Lemma A.24, $\mathcal{L} \subseteq \{\mathit{label}(L) \mid L \in \overline{L}\}$

Therefore $\ell \notin \mathcal{L}$. □

Lemma A.26. *If* $\emptyset; \Sigma \vdash v : \lfloor T_1 \rfloor$ *and* $\Sigma \vdash \sigma$ *and* $[\![T_1 \Rightarrow^{\ell} T_2]\!]$ *safe* ℓ*, then* $\mathit{hastype}(\sigma, v, \lfloor T_2 \rfloor)$.

Proof. By induction on $\emptyset; \Sigma \vdash v : \lfloor T_1 \rfloor$. Most cases vacuous.

Case TInt:

Then $T_1 = \text{int}$ and $v = n$. For it to hold that $[\![\text{int} \Rightarrow^{\ell} T_2]\!]$ safe ℓ, then either $T_2 = \text{int}$ or $T_2 = \star$.

In either case, $\mathit{hastype}(\sigma, n, \lfloor T_2 \rfloor)$.

Case TAddr:

Then $v = a$ and $\lfloor T_1 \rfloor = \Sigma(a)$.

Subcase: $\lfloor T_1 \rfloor =$ ref:

Then $\sigma(a) = v$.

For it to hold that $[\![T_1 \Rightarrow^{\ell} T_2]\!]$ safe ℓ with $\lfloor T_1 \rfloor =$ ref, either $\lfloor T_2 \rfloor = \star$ or $\lfloor T_2 \rfloor =$ ref. (This is necessary but not sufficient.) In either case, *hastype*$(\sigma, a, \lfloor T_2 \rfloor)$.

Subcase: $\lfloor T_1 \rfloor = \rightarrow$:

Then $\sigma(a) = (\lambda x.e)$.

For it to hold that $[\![T_1 \Rightarrow^{\ell} T_2]\!]$ safe ℓ with $\lfloor T_1 \rfloor = \rightarrow$, either $\lfloor T_2 \rfloor = \star$ or $\lfloor T_2 \rfloor = \rightarrow$. (This is necessary but not sufficient.) In either case, *hastype*$(\sigma, a, \lfloor T_2 \rfloor)$.

Case TSubsump:

Then $T_1 = \star$. For it to hold that $[\![\star \Rightarrow^{\ell} T_2]\!]$ safe ℓ, we must have $T_2 = \star$. Then we have that *hastype*$(\sigma, v, \star)$.

□

Lemma A.27. *If* $\emptyset; \Sigma \vdash e : S$ *and* $\Sigma \vdash \sigma$ *and* $\mathcal{B} \vdash e$ *safe* ℓ *and* $\mathcal{B} \vdash \sigma$ *safe* ℓ *and* $\langle e, \sigma, \mathcal{B} \rangle \longrightarrow \varsigma$*, then* $\varsigma \neq \text{Blame}(\mathcal{L})$ *with* $\ell \in \mathcal{L}$.

Proof. By induction on $\langle e, \sigma, \mathcal{B} \rangle \longrightarrow \varsigma$. Most cases vacuous.

Case ECheckFail: With $\neg($*hastype*$(\sigma, v, S))$,

$$\langle v \Downarrow \langle S; a; r \rangle, \sigma, \mathcal{B} \rangle \longrightarrow \textit{blame}(\sigma, v, a, r, \mathcal{B})$$

By inversion, $\mathcal{B} \vdash a$ safe ℓ.

Suppose *blame*$(\sigma, v, a, r, \mathcal{B}) = \text{Blame}(\mathcal{L})$.

Then by Lemma A.25, $\ell \notin \mathcal{L}$.

Case ECastFail: With $\neg($*hastype*$(\sigma, v, \lfloor T_2 \rfloor))$,

$$\langle v :: T_1 \Rightarrow^{\ell'} T_2, \sigma, \mathcal{B} \rangle \longrightarrow \text{Blame}(\{\ell'\})$$

Subcase : $\ell \neq \ell'$:

Have immediately that $\ell \notin \{\ell'\}$.

Subcase : $\ell = \ell'$:

Since $\mathcal{B} \vdash v{::}T_1 \Rightarrow^{\ell} T_2$ safe ℓ, $[\![T_1 \Rightarrow^{\ell} T_2]\!]$ safe ℓ.

By Lemma A.4, $\emptyset; \Sigma \vdash v : \lfloor T_1 \rfloor$.

By Lemma A.26, *hastype*$(\sigma, v, \lfloor T_2 \rfloor)$. But this contradicts $\neg($*hastype*$(\sigma, v, \lfloor T_2 \rfloor))$.

□

Lemma A.28. *Suppose that* $\emptyset; \emptyset \vdash e : S$ *and that* e *contains a subterm* $e'{::}T_1 \Rightarrow^{\ell} T_2$ *containing the only occurrence of* ℓ *in* e. *Then if* $T_1 <:_b T_2$, $\langle e, \emptyset, \emptyset \rangle \not\longrightarrow$ Blame$(\mathcal{L})$ *with* $\ell \in \mathcal{L}$.

Proof. By Lemma A.15, $[\![T_1 \Rightarrow^{\ell} T_2]\!]$ safe ℓ. Since ℓ does not otherwise occur in e, $\emptyset \vdash e$ safe ℓ. Then by applying Lemmas A.9, A.27, and A.21, we have that $\langle e, \emptyset, \emptyset \rangle \not\longrightarrow^*$ Blame$(\mathcal{L})$ with $\ell \in \mathcal{L}$. □

2.3. The gradual guarantee.

Lemma A.29. *If* $T_1 \sim T_2$ *and* $T_1 \sqsubseteq T_1'$ *and* $T_2 \sqsubseteq T_2'$, *then* $T_1' \sim T_2'$.

Proof. By induction on $T_1 \sim T_2$, and then cases on T_1' and T_2'.

Case : $T_{11} \to T_{12} \sim T_{21} \to T_{22}$

$$\frac{T_{11} \sim T_{21} \qquad T_{12} \sim T_{22}}{T_{11} \to T_{12} \sim T_{21} \to T_{22}}$$

Have that T_1' must be either $\star$ or $T_{11}' \to T_{12}'$.

In the former case, theorem holds immediately.

Otherwise, have that $T_{11} \sqsubseteq T_{11}'$ and $T_{12} \sqsubseteq T_{12}'$.

Then have that T_2' must be either $\star$ or $T_{21}' \to T_{22}'$.

In the former case, theorem holds immediately.

Otherwise, have that $T_{21} \sqsubseteq T_{21}'$ and $T_{22} \sqsubseteq T_{22}'$.

Then by the IH, $T_{11}' \sim T_{21}'$ and $T_{12}' \sim T_{22}'$.

Therefore $T_{11}' \to T_{12}' \sim T_{21}' \to T_{22}'$.

Remaining cases are similar. □

Lemma A.30. *If* $T_1 \sqsubseteq T_2$ *and* $T_1 \rhd T_{11} \to T_{12}$ *then* $T_2 \rhd T_{21} \to T_{22}$ *and* $T_{11} \sqsubseteq T_{21}$ *and* $T_{12} \sqsubseteq T_{22}$.

Proof. By cases on $T_1 \sqsubseteq T_2$.

Case : $T \sqsubseteq \star$

Then $T_{21} = T_{22} = \star$.

Proceed by subcases on T

Subcase : $T = \star$ Then $T_{11} = T_{12} = \star$.

Have $\star \sqsubseteq \star$.

Subcase : $T = T'_{11} \to T'_{12}$ Then $T_{11} = T'_{11}$ and $T_{12} = T'_{12}$.

Have $T_{11} \sqsubseteq \star$ and $T_{12} \sqsubseteq \star$.

Other subcases vacuous.

Case : $T'_{11} \to T'_{12} \sqsubseteq T'_{21} \to T'_{22}$

Have $T'_{11} \sqsubseteq T'_{21}$ and $T'_{12} \sqsubseteq T'_{22}$

Have $T_{11} = T'_{11}$ and $T_{12} = T'_{12}$ and $T_{21} = T'_{21}$ and $T_{22} = T'_{22}$.

Other subcases vacuous. □

Lemma A.31. *If* $T_1 \sqsubseteq T_2$ *and* $T_1 \rhd$ *ref* T'_1 *then* $T_2 \rhd$ *ref* T'_2 *and* $T'_1 \sqsubseteq T'_2$.

Proof. By cases on $T_1 \sqsubseteq T_2$.

Case : $T \sqsubseteq \star$

Then $T'_2 = \star$.

Proceed by subcases on T

Subcase : $T = \star$ Then $T'_1 = \star$.

Have $\star \sqsubseteq \star$.

Subcase : $T =$ ref T''_1 Then $T'_1 = T''_1$.

Have $T'_1 \sqsubseteq \star$.

Other subcases vacuous.

Case : ref $T_1'' \sqsubseteq$ ref T_2''

Have $T_1'' \sqsubseteq T_2''$.

Have $T_1'' = T_1'$ and $T_2'' = T_2'$.

Other subcases vacuous. □

Lemma A.32 (Weakening preserves cast insertion). *If* $e_{s1} \sqsubseteq e_{s2}$ *and* $\Gamma_1 \sqsubseteq \Gamma_2$ *and* $\Gamma_1 \vdash e_{s1} \rightsquigarrow e_1 : T_1$*, then* $\Gamma_2 \vdash e_{s2} \rightsquigarrow e_2 : T_2$ *and* $T_1 \sqsubseteq T_2$.

Proof. By induction on $e_{s1} \sqsubseteq e_{s2}$.

Case PEVar:

Since $\Gamma_1 \sqsubseteq \Gamma_2$ and $\Gamma_1(x) = T_1$, have that $\Gamma_2(x) = T_2$ and $T_1 \sqsubseteq T_2$. Therefore $\Gamma_2 \vdash x \rightsquigarrow x : T_2$.

Case PEInt: is immediate.

Case PEFun:

$$\frac{T_{11} \sqsubseteq T_{21} \qquad T_{12} \sqsubseteq T_{22} \qquad e_{s1} \sqsubseteq e_{s2}}{\texttt{fun}\ f\ (x{:}T_{11}) \to T_{12}.\ e_{s1} \sqsubseteq \texttt{fun}\ f\ (x{:}T_{21}) \to T_{22}.\ e_{s2}}$$

Immediately, $T_{11} \to T_{12} \sqsubseteq T_{21} \to T_{22}$.

Since $\Gamma_1 \sqsubseteq \Gamma_2$ and $T_{11} \to T_{12} \sqsubseteq T_{21} \to T_{22}$ and $T_{11} \sqsubseteq T_{12}$, have that $\Gamma_1, f : T_{11} \to T_{12}, x : T_{11} \sqsubseteq \Gamma_2, f : T_{21} \to T_{22}, x : T_{21}$.

By inversion, $\Gamma_1, f : T_{11} \to T_{12}, x : T_{11} \vdash e_{s1} \rightsquigarrow e_1 : T_{12}'$ and $T_{12}' \sim T_{12}$.

By the IH, $\Gamma_2, f : T_{21} \to T_{22}, x : T_{21} \vdash e_{s2} \rightsquigarrow e_2 : T_{22}'$ and $T_{12'} \sqsubseteq T_{22}'$.

By Lemma A.29, $T_{22}' \sim T_{22}$.

Therefore by CFun $\Gamma' \vdash \texttt{fun}\ f\ (x{:}T_{21}) \to T_{22}.\ e_{s2} \rightsquigarrow e : T_{21} \to T_{22}$.

Case PEApp:

$$\frac{e_{s11} \sqsubseteq e_{s21} \qquad e_{s12} \sqsubseteq e_{s22}}{e_{s11}\ e_{s12} \sqsubseteq e_{s21}\ e_{s22}}$$

By inversion, $\Gamma_1 \vdash e_{s11} \rightsquigarrow e_{11} : T_1$.

By the IH, $\Gamma_2 \vdash e_{s21} \rightsquigarrow e_{21} : T_2$ and $T_1 \sqsubseteq T_2$.

By inversion, $T_1 \rhd T_{11} \to T_{12}$.

By Lemma A.30, $T_2 \rhd T_{21} \to T_{22}$ and $T_{11} \sqsubseteq T_{21}$ and $T_{12} \sqsubseteq T_{22}$. By inversion, $\Gamma_1 \vdash e_{s12} \rightsquigarrow e_{12} :$

T'_{11}.

By the IH, $\Gamma_2 \vdash e_{s22} \rightsquigarrow e_{22} : T'_{21}$ and $T'_{11} \sqsubseteq T'_{21}$.

By Lemma A.29, $T'_{21} \sim T_{21}$.

By CApp, $\Gamma_2 \vdash e_{s21}\ e_{s22} \rightsquigarrow e : T_{22}$.

Case PERef:

$$\frac{e_{s1} \sqsubseteq e_{s2}}{\texttt{ref}\ e_{s1} \sqsubseteq \texttt{ref}\ e_{s2}}$$

By inversion, $\Gamma_1 \vdash e_{s1} \rightsquigarrow e_1 : T_1$.

By the IH, $\Gamma_2 \vdash e_{s2} \rightsquigarrow e_2 : T_2$ and $T_1 \sqsubseteq T_2$.

Thus ref $T_1 \sqsubseteq$ ref T_2.

By CRef, $\Gamma_2 \vdash \texttt{ref}\ e_{s2} \rightsquigarrow e :$ ref T_2.

Case PEDeref:

$$\frac{e_{s1} \sqsubseteq e_{s2}}{!e_{s1} \sqsubseteq !e_{s2}}$$

By inversion, $\Gamma_1 \vdash e_{s1} \rightsquigarrow e_1 : T_1$.

By the IH, $\Gamma_2 \vdash e_{s2} \rightsquigarrow e_2 : T_2$ and $T_1 \sqsubseteq T_2$.

By inversion, $T_1 \rhd$ ref T'_1.

By Lemma A.31, $T_2 \rhd$ ref T'_2 and $T'_1 \sqsubseteq T'_2$.

By CDeref, $\Gamma_2 \vdash !e_{s2} \rightsquigarrow e : T'_2$.

Case PESet:

$$\frac{e_{s11} \sqsubseteq e_{s21} \qquad e_{s12} \sqsubseteq e_{s22}}{e_{s11} := e_{s12} \sqsubseteq e_{s21} := e_{s22}}$$

By inversion, $\Gamma_1 \vdash e_{s11} \rightsquigarrow e_{11} : T_1$.

By the IH, $\Gamma_2 \vdash e_{s21} \rightsquigarrow e_{21} : T_2$ and $T_1 \sqsubseteq T_2$.

By inversion, $T_1 \rhd$ ref T'_1.

By Lemma A.31, $T_2 \rhd$ ref T'_2 and $T'_1 \sqsubseteq T'_2$.

By inversion, $\Gamma_1 \vdash e_{s12} \rightsquigarrow e_{12} : T'_1$.

By the IH, $\Gamma_2 \vdash e_{s22} \rightsquigarrow e_{22} : T'_2$ and $T'_1 \sqsubseteq T'_2$.

By inversion, $T_1 \sim T'_1$.

By Lemma A.29, $T_2 \sim T_2'$.

By CUpdtRef, $\Gamma_2 \vdash e_{s21} := e_{s22} \rightsquigarrow e$: int.

Case PEAdd:

$$\frac{e_{s11} \sqsubseteq e_{s21} \qquad e_{s12} \sqsubseteq e_{s22}}{e_{s11} \ + \ e_{s12} \sqsubseteq e_{s21} \ + \ e_{s22}}$$

By inversion, $\Gamma_1 \vdash e_{s11} \rightsquigarrow e_{11} : T_{11}$.

By the IH, $\Gamma_2 \vdash e_{s21} \rightsquigarrow e_{21} : T_{21}$ and $T_{11} \sqsubseteq T_{21}$.

By inversion, $\Gamma_1 \vdash e_{s12} \rightsquigarrow e_{12} : T_{12}$.

By the IH, $\Gamma_2 \vdash e_{s22} \rightsquigarrow e_{22} : T_{22}$ and $T_{12} \sqsubseteq T_{22}$.

By inversion $T_{11} \sim$ int and $T_{12} \sim$ int.

By Lemma A.29, $T_{21} \sim$ int and $T_{22} \sim$ int.

By CAdd, $\Gamma_2 \vdash e_{s21} \ + \ e_{s22} \rightsquigarrow e$: int.

□

Lemma A.33. *If* $T_1 \sqsubseteq T_2$ *then* $\lfloor T_1 \rfloor \sqsubseteq \lfloor T_2 \rfloor$.

Proof. Immediately by cases on $T_1 \sqsubseteq T_2$. □

Lemma A.34 (Cast insertion preserves precision). *If* $\Gamma \vdash e_s \rightsquigarrow e : T$ *and* $\Gamma' \vdash e_s' \rightsquigarrow e' : T'$ *and* $e_s \sqsubseteq e_s'$ *and* $\Gamma \sqsubseteq \Gamma'$*, then* $e \sqsubseteq e'$ *and* $T \sqsubseteq T'$.

Proof. By induction on $e_s \sqsubseteq e_s'$.

Case PEVar:

$$x \sqsubseteq x$$

Since $\Gamma(x) = T$ and $\Gamma \sqsubseteq \Gamma'$, $T \sqsubseteq T'$.

By PVar, $x \sqsubseteq x$.

Case PEInt:

$$n \sqsubseteq n$$

Have $T = T' =$ int.

By PInt, $n \sqsubseteq n$.

Case PEFun:

$$\frac{T_{11} \sqsubseteq T_{21} \qquad T_{12} \sqsubseteq T_{22} \qquad e_{s1} \sqsubseteq e_{s2}}{\texttt{fun}\ f\ (x{:}T_{11}) \to T_{12}.\ e_{s1} \sqsubseteq \texttt{fun}\ f\ (x{:}T_{21}) \to T_{22}.\ e_{s2}}$$

Have $\Gamma \vdash \texttt{fun}\ f\ (x{:}T_{11}) \to T_{12}.\ e_{s1} \rightsquigarrow \texttt{fun}\ f\ x.\ (\texttt{let}\ x = x{\Downarrow}\langle \lfloor T_{11} \rfloor; f; \mathsf{Arg} \rangle\ \texttt{in}\ e_1) : T_{11} \to T_{12}$.

Have $\Gamma' \vdash \texttt{fun}\ f\ (x{:}T_{21}) \to T_{22}.\ e_{s1} \rightsquigarrow \texttt{fun}\ f\ x.\ (\texttt{let}\ x = x{\Downarrow}\langle \lfloor T_{21} \rfloor; f; \mathsf{Arg} \rangle\ \texttt{in}\ e_2) : T_{21} \to T_{22}$.

Immediately have $T_{11} \to T_{12} \sqsubseteq T_{21} \to T_{22}$.

By inversion, $\Gamma, f : T_{11} \to T_{12}, x : T_{11} \vdash e_{s1} \rightsquigarrow e_1 : T'_{12}$.

By inversion, $\Gamma', f : T_{21} \to T_{22}, x : T_{21} \vdash e_{s2} \rightsquigarrow e_2 : T'_{22}$.

Have that $\Gamma, f : T_{11} \to T_{12}, x : T_{11} \sqsubseteq \Gamma', f : T_{21} \to T_{22}, x : T_{21}$.

By the IH, $e_1 \sqsubseteq e_2$.

By PVar, $x \sqsubseteq x$ and $f \sqsubseteq f$.

By Lemma A.33, $\lfloor T_{11} \rfloor \sqsubseteq \lfloor T_{21} \rfloor$.

Hence by PCheck, $x{\Downarrow}\langle \lfloor T_{11} \rfloor; f; \mathsf{Arg} \rangle \sqsubseteq x{\Downarrow}\langle \lfloor T_{21} \rfloor; f; \mathsf{Arg} \rangle$.

Hence by PLet and PFun, $e \sqsubseteq e'$.

Case PEApp:

$$\frac{e_{s11} \sqsubseteq e_{s21} \qquad e_{s12} \sqsubseteq e_{s22}}{e_{s11}\ e_{s12} \sqsubseteq e_{s21}\ e_{s22}}$$

Have $\Gamma \vdash e_{s11}\ e_{s12} \rightsquigarrow \texttt{let}\ f = e_{11}{::}T_1 {\Rightarrow^{\ell}} T_{11} \to T_{12}\ \texttt{in}\ (f\ (e_{12}{::}T'_{11} {\Rightarrow^{\ell}} T_{11})){\Downarrow}\langle \lfloor T_{12} \rfloor; f; \mathsf{Res} \rangle : T_{12}$.

Have that ℓ, f fresh in the cast inserion of $e_{s11}\ e_{s12}$.

Assume without loss of generality that ℓ, f fresh in the cast insertion of $e_{s21}\ e_{s22}$, and select them for use in its translation.

Have $\Gamma' \vdash e_{s21}\ e_{s22} \rightsquigarrow \texttt{let}\ f = e_{21}{::}T_2 {\Rightarrow^{\ell}} T_{21} \to T_{22}\ \texttt{in}\ (f\ (e_{22}{::}T'_{21} {\Rightarrow^{\ell}} T_{11})){\Downarrow}\langle \lfloor T_{22} \rfloor; f; \mathsf{Res} \rangle : T_{22}$.

By inversion, $\Gamma \vdash e_{s11} \rightsquigarrow e_{11} : T_1$ and $T_1 \rhd T_{11} \to T_{12}$.

By inversion, $\Gamma' \vdash e_{s21} \rightsquigarrow e_{21} : T_2$ and $T_2 \rhd T_{21} \to T_{22}$.

By the IH, $e_{11} \sqsubseteq e_{21}$ and $T_1 \sqsubseteq T_2$.

By Lemma A.30, $T_{11} \sqsubseteq T_{21}$ and $T_{21} \sqsubseteq T_{22}$, and therefore $T_{11} \to T_{12} \sqsubseteq T_{21} \to T_{22}$.

By inversion, $\Gamma \vdash e_{s12} \rightsquigarrow e_{12} : T'_{11}$.

By inversion, $\Gamma' \vdash e_{s22} \rightsquigarrow e_{22} : T'_{21}$.

By the IH, $e_{12} \sqsubseteq e_{22}$ and $T'_{11} \sqsubseteq T'_{21}$.

By Lemma A.33, $\lfloor T_{11} \rfloor \sqsubseteq \lfloor T_{21} \rfloor$.

By PCast, PVar, PApp, PCheck, and PLet, $e \sqsubseteq e'$.

Case PERef:

$$\frac{e_{s1} \sqsubseteq e_{s2}}{\texttt{ref}\ e_{s1} \sqsubseteq \texttt{ref}\ e_{s2}}$$

Have $\Gamma \vdash \texttt{ref}\ e_{s1} \rightsquigarrow \texttt{ref}\ e_1 : \text{ref}\, T_1$.

Have $\Gamma' \vdash \texttt{ref}\ e_{s2} \rightsquigarrow \texttt{ref}\ e_2 : \text{ref}\, T_2$.

By inversion, $\Gamma \vdash e_{s1} \rightsquigarrow e_1 : T_1$.

By inversion, $\Gamma' \vdash e_{s2} \rightsquigarrow e_2 : T_2$.

By the IH, $e_1 \sqsubseteq e_2$ and $T_1 \sqsubseteq T_2$.

Therefore $\texttt{ref}\ e_1 \sqsubseteq \texttt{ref}\ e_2$ (by PRef) and $\text{ref}\, T_1 \sqsubseteq \text{ref}\, T_2$.

Case PEDeref:

$$\frac{e_{s1} \sqsubseteq e_{s2}}{!e_{s1} \sqsubseteq !e_{s2}}$$

Have $\Gamma \vdash !e_{s1} \rightsquigarrow \texttt{let}\ x = e_1 :: T_1 \Rightarrow^{\ell} \text{ref}\, T'_1\ \texttt{in}\ !x \Downarrow \langle \lfloor T'_1 \rfloor; x; \text{Deref} \rangle : T'_1$.

Have that ℓ, x fresh in the cast inserion of $!e_{s1}$.

Assume without loss of generality that ℓ, x fresh in the cast insertion of $!e_{s2}$, and select them for use in its translation.

Have $\Gamma' \vdash !e_{s2} \rightsquigarrow \texttt{let}\ x = e_2 :: T_2 \Rightarrow^{\ell} \text{ref}\, T'_2\ \texttt{in}\ !x \Downarrow \langle \lfloor T'_2 \rfloor; x; \text{Deref} \rangle : T'_2$.

By inversion, $\Gamma \vdash e_{s1} \rightsquigarrow e_1 : T_1$ and $T_1 \rhd \text{ref}\, T'_1$.

By inversion, $\Gamma' \vdash e_{s2} \rightsquigarrow e_2 : T_2$ and $T_2 \rhd \text{ref}\, T'_2$.

By the IH, $e_1 \sqsubseteq e_2$ and $T_1 \sqsubseteq T_2$.

By Lemma A.31, $T'_1 \sqsubseteq T'_2$, and therefore $\text{ref}\, T'_1 \sqsubseteq \text{ref}\, T'_2$.

By Lemma A.33, $\lfloor T'_1 \rfloor \sqsubseteq \lfloor T'_2 \rfloor$. By PCast, PVar, PDeref, PCheck, and PLet, $e \sqsubseteq e'$.

Case PESet:

$$\frac{e_{s11} \sqsubseteq e_{s21} \qquad e_{s12} \sqsubseteq e_{s22}}{e_{s11} := e_{s12} \sqsubseteq e_{s21} := e_{s22}}$$

Have $\Gamma \vdash e_{s11} := e_{s12} \rightsquigarrow (e_{11} :: T_1 \Rightarrow^{\ell_1} \text{ref}\, T_1' := e_{12} :: T_1'' \Rightarrow^{\ell_2} T_1') : \text{int}$.

Have that ℓ_1, ℓ_2 fresh in the cast inserion of $e_{s11} := e_{s12}$.

Assume without loss of generality that ℓ_1, ℓ_2 fresh in the cast insertion of $e_{s21} := e_{s22}$, and select them for use in its translation.

Have $\Gamma' \vdash e_{s21} := e_{s22} \rightsquigarrow (e_{21} :: T_2 \Rightarrow^{\ell_1} \text{ref}\, T_2' := e_{22} :: T_2'' \Rightarrow^{\ell_2} T_2') : \text{int}$.

By inversion, $\Gamma \vdash e_{s11} \rightsquigarrow e_{11} : T_1$ and $T_1 \rhd \text{ref}\, T_1'$.

By inversion, $\Gamma' \vdash e_{s21} \rightsquigarrow e_{21} : T_2$ and $T_2 \rhd \text{ref}\, T_2'$.

By the IH, $e_{11} \sqsubseteq e_{21}$ and $T_1 \sqsubseteq T_2$.

By Lemma A.31, $T_1' \sqsubseteq T_2'$, and therefore $\text{ref}\, T_1' \sqsubseteq \text{ref}\, T_2'$.

By inversion, $\Gamma \vdash e_{s12} \rightsquigarrow e_{12} : T_1''$.

By inversion, $\Gamma' \vdash e_{s22} \rightsquigarrow e_{22} : T_2''$.

By the IH, $e_{12} \sqsubseteq e_{22}$ and $T_1'' \sqsubseteq T_2''$.

By PCast and PSet, $e \sqsubseteq e'$.

Case PEAdd:

$$\frac{e_{s11} \sqsubseteq e_{s21} \qquad e_{s12} \sqsubseteq e_{s22}}{e_{s11} \ + \ e_{s12} \sqsubseteq e_{s21} \ + \ e_{s22}}$$

Have $\Gamma \vdash e_{s11} \ + \ e_{s12} \rightsquigarrow e_{11} :: T_{11} \Rightarrow^{\ell_1} \text{int} \ + \ e_{12} :: T_{12} \Rightarrow^{\ell_2} \text{int} : \text{int}$.

Have that ℓ_1, ℓ_2 fresh in the cast inserion of $e_{s11} \ + \ e_{s12}$.

Assume without loss of generality that ℓ_1, ℓ_2 fresh in the cast insertion of $e_{s21} \ + \ e_{s22}$, and select them for use in its translation.

Have $\Gamma' \vdash e_{s21} \ + \ e_{s22} \rightsquigarrow e_{21} :: T_{21} \Rightarrow^{\ell_1} \text{int} \ + \ e_{22} :: T_{22} \Rightarrow^{\ell_2} \text{int} : \text{int}$.

By inversion, $\Gamma \vdash esrcn_{11} \rightsquigarrow e_{11} : T_{11}$.

By inversion, $\Gamma \vdash esrcn_{21} \rightsquigarrow e_{21} : T_{21}$.

By the IH, $e_{11} \sqsubseteq e_{21}$ and $T_{11} \sqsubseteq T_{21}$.

By inversion, $\Gamma \vdash esrcn_{11} \rightsquigarrow e_{11} : T_{11}$.

By inversion, $\Gamma \vdash esrcn_{21} \rightsquigarrow e_{21} : T_{21}$.

By the IH, $e_{11} \sqsubseteq e_{21}$ and $T_{11} \sqsubseteq T_{21}$.

By PCast and PAdd, $e \sqsubseteq e'$.

□

Lemma A.35. *If $v \sqsubseteq e$, then e is a value.*

Proof. By cases on $v \sqsubseteq e$. The only non-vacuous cases are PAddr and PInt. In both cases, e is a value.

□

Lemma A.36. *If $e \sqsubseteq v$, then e is a value.*

Proof. By cases on $e \sqsubseteq v$. The only non-vacuous cases are PAddr and PInt. In both cases, e is a value.

□

Lemma A.37 (Preservation of simulation under substitution). *Suppose $e_1 \sqsubseteq e_1'$ and $e_2 \sqsubseteq e_2'$. Then $e_1[e_2/x] \sqsubseteq e_1'[e_2'/x]$.*

Proof. By induction on $e_1 \sqsubseteq e_1'$.

Case PVar:

$$y \sqsubseteq y$$

If $x = y$, then $y[e_2/x] = e_2$ and $y[e_2'/x] = e_2'$, and theorem proved by assumption.

Otherwise, $y[e_2/x] = y$ and $y[e_2'/x] = y$, and we prove by applying PVar.

Cases PInt and PAddr: are trivial.

Case PFun:

$$\frac{e_1 \sqsubseteq e_1'}{\texttt{fun}\ f\ y.\ e_1 \sqsubseteq \texttt{fun}\ f\ y.\ e_1'}$$

If $y = x$, then $\texttt{fun}\ f\ y.\ e_1[e_2/x] = \texttt{fun}\ f\ y.\ e_1$ and $\texttt{fun}\ f\ y.\ e_1'[e_2'/x] = \texttt{fun}\ f\ y.\ e_1'$, and the theorem holds by assumption.

Similar if $f = x$.

Otherwise, $\texttt{fun}\ f\ y.\ e_1[e_2/x] = \texttt{fun}\ f\ y.\ e_1[e_2/x]$ and $\texttt{fun}\ f\ y.\ e_1'[e_2'/x] = \texttt{fun}\ f\ y.\ e_1'[e_2'/x]$.

By the IH, $e_1[e_2/x] \sqsubseteq e_1'[e_2'/x]$.

By PFun $\texttt{fun}\ f\ y.\ e_1[e_2/x] \sqsubseteq \texttt{fun}\ f\ y.\ e_1'[e_2'/x]$.

Remaining cases are similar. □

Lemma A.38. *Suppose* $v_1 \sqsubseteq v_2$ *and* $\sigma_1 \sqsubseteq \sigma_2$ *and* $S_1 \sqsubseteq S_2$. *If hastype*(σ_1, v_1, S_1)*, then hastype*(σ_2, v_2, S_2).

Proof. By cases on *hastype*(σ_1, v_1, S_1).

Case : *hastype*$(\sigma_1, n_1, \text{int})$

Since $n_1 \sqsubseteq v_2$, $v_2 = n_2$.

Since int $\sqsubseteq S_2$, either $S_2 =$ int or $S_2 = \star$.

In either case, *hastype*(σ_2, n_2, S_2).

Case : *hastype*$(\sigma_1, v_1, \star)$

Since $\star \sqsubseteq S_2$, $S_2 = \star$.

For any σ_2, v_2, *hastype*$(\sigma_2, v_2, \star)$.

Case : *hastype*$(\sigma_1, a, \rightarrow)$

Since $\rightarrow \sqsubseteq S_2$, either $S_2 = \rightarrow$ or $S_2 = \star$.

Suppose that $S_2 = \rightarrow$.

Have that $\sigma_1(a) = (\lambda x.e_1)$.

Since $a \sqsubseteq v_2$, $v_2 = a$.

Since $\sigma_1 \sqsubseteq \sigma_2$, $\sigma_2(a) = (\lambda x.e_2)$.

Therefore *hastype*$(\sigma_2, a, \rightarrow)$. Now suppose $S_2 = \star$.

For any σ_2, v_2, *hastype*$(\sigma_2, v_2, \star)$.

Case : *hastype*$(\sigma_1, a, \text{ref})$

Since ref $\sqsubseteq S_2$, either $S_2 =$ ref or $S_2 = \star$.

Suppose that $S_2 =$ ref.

Have that $\sigma_1(a) = v_1'$.

Since $a \sqsubseteq v_2$, $v_2 = a$.

Since $\sigma_1 \sqsubseteq \sigma_2$, $\sigma_2(a) = v_2'$.

Therefore *hastype*$(\sigma_2, a, \text{ref})$. Now suppose $S_2 = \star$.

For any σ_2, v_2, *hastype*$(\sigma_2, v_2, \star)$.

□

Lemma A.39 (Simulation of more precise programs). *Suppose* $e_1 \sqsubseteq e_2$ *and* $\sigma_1 \sqsubseteq \sigma_2$. *If* $\langle e_1, \sigma_1, \mathcal{B}_1\rangle \longrightarrow \langle e_1', \sigma_1', \mathcal{B}_1'\rangle$, *then* $\langle e_2, \sigma_2, \mathcal{B}_2\rangle \longrightarrow \langle e_2', \sigma_2', \mathcal{B}_2'\rangle$ *and* $e_1' \sqsubseteq e_2'$ *and* $\sigma_1' \sqsubseteq \sigma_2'$.

Proof. By cases on $e_1 \sqsubseteq e_2$.

Cases PVar, PInt, PAddr: are vacuous.

Case PApp:

$$\frac{e_{11} \sqsubseteq e_{21} \qquad e_{12} \sqsubseteq e_{22}}{e_{11}\ e_{12} \sqsubseteq e_{21}\ e_{22}}$$

Since $\langle e_{11}\ e_{12}, \sigma_1, \mathcal{B}_1\rangle \longrightarrow \langle e_1', \sigma_1', \mathcal{B}_1'\rangle$, have that $e_{11} = a$, $e_{12} = v_1$, $\sigma_1(a) = (\lambda x.e_{1h})$, $e_1' = e_{1h}[v_1/x]$, and $\sigma_1' = \sigma_1$.

Since $a \sqsubseteq e_{21}$, $e_{21} = a$.

Since $v_1 \sqsubseteq e_{22}$, by Lemma A.35 $e_{22} = v_2$.

Since $\sigma_1 \sqsubseteq \sigma_2$, $(\lambda x.e_{1h}) \sqsubseteq_h \sigma_2(a)$.

Therefore $\sigma_2(a) = (\lambda x.e_{2h})$ and $e_{1h} \sqsubseteq e_{2h}$.

Hence by EApp, $\langle e_{21}\ e_{22}, \sigma_2, \mathcal{B}_2\rangle \longrightarrow \langle e_{2h}[v_2/x], \sigma_2, \mathcal{B}_2\rangle$.

By Lemma A.37, $e_{1h}[v_1/x] \sqsubseteq e_{2h}[v_2/x]$.

Case PFun:

$$\frac{e_1 \sqsubseteq e_2}{\texttt{fun}\ f\ x.\ e_1 \sqsubseteq \texttt{fun}\ f\ x.\ e_2}$$

Since $\langle \texttt{fun}\ f\ x.\ e_1, \sigma_1, \mathcal{B}_1\rangle \longrightarrow \langle e_1', \sigma_1', \mathcal{B}_1'\rangle$, have that $e_1' = a$ for fresh a and $\sigma_1' = \sigma_1[a \mapsto (\lambda x.e_1[a/f])]$.

Suppose without loss of generality that a is fresh for the evaluation of both e_1 and e_2.

Then $\langle \texttt{fun}\ f\ x.\ e_2, \sigma_2, \mathcal{B}_2\rangle \longrightarrow \langle a, \sigma_2[a \mapsto (\lambda x.e_2[a/f])], \mathcal{B}_2\rangle$.

Have immediately that $a \sqsubseteq a$.

By Lemma A.37, $e_1[a/f] \sqsubseteq e_2[a/f]$.

Therefore $(\lambda x.e_1[a/f]) \sqsubseteq_h (\lambda x.e_2[a/f])$.

Therefore $\sigma_1[a \mapsto (\lambda x.e_1[a/f])] \sqsubseteq \sigma_2[a \mapsto (\lambda x.e_2[a/f])]$.

Case PRef:

$$\frac{e_1 \sqsubseteq e_2}{\texttt{ref}\ e_1 \sqsubseteq \texttt{ref}\ e_2}$$

Since $\langle \texttt{ref}\ e_1, \sigma_1, \mathcal{B}_1 \rangle \longrightarrow \langle e_1', \sigma_1', \mathcal{B}_1' \rangle$, have that $e_1 = v_1$, $e_1' = a$ for fresh a and $\sigma_1' = \sigma_1[a \mapsto v_1]$.

Suppose without loss of generality that a is fresh for the evaluation of both e_1 and e_2.

By Lemma A.35 $e_2 = v_2$.

Then $\langle \texttt{ref}\ v_2, \sigma_2, \mathcal{B}_2 \rangle \longrightarrow \langle a, \sigma_2[a \mapsto v_2], \mathcal{B}_2 \rangle$.

Have immediately that $a \sqsubseteq a$.

Have that $v_1 \sqsubseteq_h v_2$.

Therefore $\sigma_1[a \mapsto v_1] \sqsubseteq \sigma_2[a \mapsto v_2]$.

Case PDeref:

$$\frac{e_1 \sqsubseteq e_2}{!e_1 \sqsubseteq !e_2}$$

Since $\langle !e_1, \sigma_1, \mathcal{B}_1 \rangle \longrightarrow \langle e_1', \sigma_1', \mathcal{B}_1' \rangle$, have $e_1 = a$, $\sigma_1(a) = v_1$, and $\sigma_1' = \sigma_1$.

Since $a \sqsubseteq e_2$, $e_2 = a$.

Since $\sigma_1 \sqsubseteq \sigma_2$, $\sigma_2(a) = h$ and $v_1 \sqsubseteq_h h$.

Therefore $h = v_2$ and $v_1 \sqsubseteq v_2$.

Thus $\langle !e_2, \sigma_2, \mathcal{B}_2 \rangle \longrightarrow \langle v_2, \sigma_2, \mathcal{B}_2 \rangle$.

Case PSet:

$$\frac{e_{11} \sqsubseteq e_{21} \qquad e_{12} \sqsubseteq e_{22}}{e_{11} := e_{12} \sqsubseteq e_{21} := e_{22}}$$

Since $\langle e_{11} := e_{12}, \sigma_1, \mathcal{B}_1 \rangle \longrightarrow \langle e_1', \sigma_1', \mathcal{B}_1' \rangle$, have $e_{11} = a$, $e_{12} = v_1$, $\sigma_1(a) = v_1'$, $e_1' = 0$, and $\sigma_1' = \sigma_1[a \mapsto v_1]$.

Since $a \sqsubseteq e_{21}$, $e_{21} = a$.

By Lemma A.35 $e_{22} = v_2$.

Since $\sigma_1 \sqsubseteq \sigma_2$, $\sigma_2(a) = v_2'$.

Therefore $\langle e_{21} := e_{22}, \sigma_2, \mathcal{B}_2 \rangle \longrightarrow \langle 0, \sigma_2[a \mapsto v_2], \mathcal{B}_2 \rangle$.

Immediately have $0 \sqsubseteq 0$.

Since $v_1 \sqsubseteq v_2$, $v_1 \sqsubseteq_h v_2$. Thus, since $\sigma_1 \sqsubseteq \sigma_2$, $\sigma_1[a \mapsto v_1] \sqsubseteq \sigma_2[a \mapsto v_2]$.

Case PCheck:

$$\frac{e_{11} \sqsubseteq e_{21} \qquad e_{12} \sqsubseteq e_{22} \qquad S_1 \sqsubseteq S_2}{e_{11} \Downarrow \langle S_1; e_{12}; r \rangle \sqsubseteq e_{21} \Downarrow \langle S_2; e_{22}; r \rangle}$$

Since $\langle e_{11} \Downarrow \langle S_1; e_{12}; r \rangle, \sigma_1, \mathcal{B}_1 \rangle \longrightarrow \langle e_1', \sigma_1', \mathcal{B}_1' \rangle$, have that $e_{11} = v_1$, $e_{12} = a$, *hastype*(σ_1, v_1, S_1), $e_1' = v_1$, and $\sigma_1' = \sigma_1$.

By Lemma A.35 $e_{21} = v_2$.

By Lemma A.38, *hastype*(σ_2, v_2, S_2).

Since $a \sqsubseteq e_{22}$, $e_{22} = a$.

Therefore, $\langle v_2 \Downarrow \langle S_2; a; r \rangle, \sigma_2, \mathcal{B}_2 \rangle \longrightarrow \langle v_2, \sigma_2, \mathcal{B}_2' \rangle$ for some $\mathcal{B}_2'$.

Case PCast:

$$\frac{e_1 \sqsubseteq e_2 \qquad T_{11} \sqsubseteq T_{21} \qquad T_{12} \sqsubseteq T_{22}}{e_1 :: T_{11} \Rightarrow^{\ell} T_{12} \sqsubseteq e_2 :: T_{21} \Rightarrow^{\ell} T_{22}}$$

Since $\langle e_1 :: T_{11} \Rightarrow^{\ell} T_{12}, \sigma_1, \mathcal{B}_1 \rangle \longrightarrow \langle e_1', \sigma_1', \mathcal{B}_1' \rangle$, have that $e_1 = v_1$, *hastype*$(\sigma_1, v_1, \lfloor T_{12} \rfloor)$, $e_1' = v_1$, and $\sigma_1' = \sigma_1$.

By Lemma A.35 $e_2 = v_2$.

By Lemma A.33 $\lfloor T_{12} \rfloor \sqsubseteq \lfloor T_{22} \rfloor$.

By Lemma A.38, *hastype*$(\sigma_2, v_2, \lfloor T_{22} \rfloor)$.

Therefore, $\langle v_2 :: T_{21} \Rightarrow^{\ell} T_{22}, \sigma_2, \mathcal{B}_2 \rangle \longrightarrow \langle v_2, \sigma_2, \mathcal{B}_2' \rangle$ for some $\mathcal{B}_2'$.

Case PAdd:

$$\frac{e_{11} \sqsubseteq e_{21} \qquad e_{12} \sqsubseteq e_{22}}{e_{11} \ + \ e_{12} \sqsubseteq e_{21} \ + \ e_{22}}$$

Since $\langle e_{11} \ + \ e_{12}, \sigma_1, \mathcal{B}_1 \rangle \longrightarrow \langle e_1', \sigma_1', \mathcal{B}_1' \rangle$, have that $e_{11} = n_1$, $e_{12} = n_2$, $e_1' = n'$ where $n' = n_1 + n_2$, and $\sigma_1' = \sigma_1$.

Since $n_1 \sqsubseteq e_{21}$, $e_{21} = n_1$.

Since $n_2 \sqsubseteq e_{22}$, $e_{22} = n_2$.

Therefore, $\langle e_{21} \ + \ e_{22}, \sigma_2, \mathcal{B}_2 \rangle \longrightarrow \langle n', \sigma_2, \mathcal{B}_2 \rangle$.

□

Lemma A.40. *Suppose* $e_1 \sqsubseteq e_2$ *and* $\sigma_1 \sqsubseteq \sigma_2$. *For any* n, *if* $\langle e_1, \sigma_1, \mathcal{B}_1\rangle \longrightarrow^n \langle e_1', \sigma_1', \mathcal{B}_1'\rangle$, *then* $\langle e_2, \sigma_2, \mathcal{B}_2\rangle \longrightarrow^n \langle e_2', \sigma_2', \mathcal{B}_2'\rangle$ *and* $e_1' \sqsubseteq e_2'$ *and* $\sigma_1' \sqsubseteq \sigma_2'$.

Proof. By induction on n.

Case : $n = 0$.

Then $e_1' = e_1$ and $\sigma_1' = \sigma_1$. Have that $\langle e_2, \sigma_2, \mathcal{B}_2\rangle \longrightarrow^0 \langle e_2, \sigma_2, \mathcal{B}_2\rangle$, so $e_2' = e_2$ and $\sigma_2' = \sigma_2$. Proof completed by assumptions.

Case : $n = n' + 1$.

For some $e_1'', \sigma_1'', \mathcal{B}_1''$, $\langle e_1, \sigma_1, \mathcal{B}_1\rangle \longrightarrow^{n'} \langle e_1'', \sigma_1'', \mathcal{B}_1''\rangle$ and $\langle e_1'', \sigma_1'', \mathcal{B}_1''\rangle \longrightarrow \langle e_1', \sigma_1', \mathcal{B}_1'\rangle$. By the IH, $\langle e_2, \sigma_2, \mathcal{B}_2\rangle \longrightarrow^{n'} \langle e_2'', \sigma_2'', \mathcal{B}_2''\rangle$ and $e_1'' \sqsubseteq e_2''$ and $\sigma_1'' \sqsubseteq \sigma_2''$. Then by Lemma A.39, $\langle e_2'', \sigma_2'', \mathcal{B}_2''\rangle \longrightarrow \langle e_2', \sigma_2', \mathcal{B}_2'\rangle$ and $e_1' \sqsubseteq e_2'$ and $\sigma_1' \sqsubseteq \sigma_2'$. Finally, have that $\langle e_2, \sigma_2, \mathcal{B}_2\rangle \longrightarrow^n \langle e_2', \sigma_2', \mathcal{B}_2'\rangle$.

□

Definition A.1 (Divergence). *A* $\lambda^\star_\rightarrow$ *term* e diverges, *written* $e \Uparrow$, *if for all* $e', \sigma, \mathcal{B}$ *such that* $\langle e, \emptyset, \emptyset\rangle \longrightarrow^* \langle e', \sigma, \mathcal{B}\rangle$, *there exists some* $e'', \sigma', \mathcal{B}'$ *such that* $\langle e', \sigma, \mathcal{B}\rangle \longrightarrow \langle e'', \sigma', \mathcal{B}'\rangle$.

Lemma A.41 (The gradual guarantee). *If* $e_s \sqsubseteq e_s'$ *and* $\emptyset \vdash e_s \rightsquigarrow e : T$, *then*

(1) $\emptyset \vdash e_s' \rightsquigarrow e' : T'$, *with* $T \sqsubseteq T'$, *and*

(2) *if* $\langle e, \emptyset, \emptyset\rangle \longrightarrow^* \langle v, \sigma, \mathcal{B}\rangle$, *then* $\langle e', \emptyset, \emptyset\rangle \longrightarrow^* \langle v', \sigma', \mathcal{B}'\rangle$ *with* $v \sqsubseteq v'$ *and* $\sigma \sqsubseteq \sigma'$, *and*

(3) *if* $e \Uparrow$, *then* $e' \Uparrow$, *and*

(4) *if* $\langle e', \emptyset, \emptyset\rangle \longrightarrow^* \langle v', \sigma', \mathcal{B}'\rangle$, *then either* $\langle e, \emptyset, \emptyset\rangle \longrightarrow^* \langle v, \sigma, \mathcal{B}\rangle$ *with* $v \sqsubseteq v'$ *and* $\sigma \sqsubseteq \sigma'$, *or* $\langle e, \emptyset, \emptyset\rangle \longrightarrow^* \mathrm{Blame}(\mathcal{L})$, *and*

(5) *if* $e' \Uparrow$, *then either* $e \Uparrow$ *or* $\langle e, \emptyset, \emptyset\rangle \longrightarrow^* \mathrm{Blame}(\mathcal{L})$.

Proof. We prove part 1 by applying Lemma A.32. From Lemma A.34 we have that $e \sqsubseteq e'$.

Suppose that $\langle e, \emptyset, \emptyset\rangle \longrightarrow^* \langle v, \sigma, \mathcal{B}\rangle$. Then by Lemma A.40, $\langle e', \emptyset, \emptyset\rangle \longrightarrow^* \langle v', \sigma', \mathcal{B}'\rangle$ with $v \sqsubseteq v'$ and $\sigma \sqsubseteq \sigma'$, proving part 2.

Now suppose that for all e_1 such that $\langle e, \emptyset, \emptyset\rangle \longrightarrow^* \langle e_1, \sigma_1, \mathcal{B}_1\rangle$, there exists some e_2 such that $\langle e_1, \sigma_1, \mathcal{B}_1\rangle \longrightarrow \langle e_2, \sigma_2, \mathcal{B}_2\rangle$. By Lemma A.40, $\langle e', \emptyset, \emptyset\rangle \longrightarrow^* \langle e'_1, \sigma'_1, \mathcal{B}'_1\rangle$ with $e_1 \sqsubseteq e'_1$ and $\sigma_1 \sqsubseteq \sigma'_2$. Then by Lemma A.39, there exists some e'_2 such that $\langle e'_1, \sigma'_1, \mathcal{B}'_1\rangle \longrightarrow \langle e'_2, \sigma'_2, \mathcal{B}'_2\rangle$. Therefore $\langle e', \emptyset, \emptyset\rangle \Uparrow$, proving part 3.

Suppose that $\langle e', \emptyset, \emptyset\rangle \longrightarrow^n \langle v', \sigma, \mathcal{B}\rangle$ for some n. By Lemma A.13, either

(1) $\langle e, \emptyset, \emptyset\rangle \longrightarrow^* \langle v, \sigma, \mathcal{B}\rangle$ or
(2) $\langle e, \emptyset, \emptyset\rangle \longrightarrow^* \text{Blame}(\mathcal{L})$ or
(3) $\langle e, \emptyset, \emptyset\rangle \longrightarrow^* \langle e, \sigma, \mathcal{B}\rangle$ and $\langle e, \sigma, \mathcal{B}\rangle$ stuck $\blacklozenge$ or
(4) $\langle e, \emptyset, \emptyset\rangle \Uparrow$.

Case 2 satisfies the theorem. Case 3 is impossible because e does not contain any $\blacklozenge$-marked terms. Case 4 is impossible because $\langle e, \emptyset, \emptyset\rangle \longrightarrow^n \langle e_n, \sigma_n, \mathcal{B}_n\rangle$ for some e_n, and by Lemma A.40, $e_n \sqsubseteq v'$. By Lemma A.36, e_n is a value, and therefore $\langle e_n, \sigma_n, \mathcal{B}_n\rangle \not\longrightarrow \langle e'_n, \sigma'_n, \mathcal{B}'_n\rangle$. Finally, have that $\langle e, \emptyset, \emptyset\rangle \longrightarrow^n \langle v, \sigma, \mathcal{B}\rangle$, because if this evaluation took m steps with $m < n$, then e_m would not be a value and $e_m \sqsubseteq v'$, which is ruled out by Lemma A.36, and if $m > n$, then e_n would not be a value and $e_n \sqsubseteq v'$, also ruled out by Lemma A.36. This proves part 4.

Finally, suppose that $\langle e', \emptyset, \emptyset\rangle \Uparrow$. For some n, by Lemma A.13, either

(1) $\langle e, \emptyset, \emptyset\rangle \longrightarrow^n \langle v, \sigma, \mathcal{B}\rangle$ or
(2) $\langle e, \emptyset, \emptyset\rangle \longrightarrow^n \text{Blame}(\mathcal{L})$ or
(3) $\langle e, \emptyset, \emptyset\rangle \longrightarrow^n \langle e, \sigma, \mathcal{B}\rangle$ and $\langle e, \sigma, \mathcal{B}\rangle$ stuck $\blacklozenge$ or
(4) $\langle e, \emptyset, \emptyset\rangle \Uparrow$.

Case 2 satisfies the theorem. Case 3 is impossible because e does not contain any $\blacklozenge$-marked terms. Case 1 is impossible because $\langle e', \emptyset, \emptyset\rangle \longrightarrow^n \langle e'_n, \sigma'_n, \mathcal{B}'_n\rangle$ for some e'_n which is not a value, and by Lemma A.40, $v \sqsubseteq e'_n$. By Lemma A.35, e'_n is a value, resulting in a contradiction. Case 4 satisfies the theorem. This proves part 5. $\square$

APPENDIX B

Appendices to Chapter 6

1. Appendix: Semantics

Figure 1 shows the static type system for $\lambda_s^\star$. Figure 2 shows the shallow static type system for $\lambda_d^\Downarrow$. Figure 3 relates surface types U with constraint types A.

Figure 4 shows the syntax for $\lambda_e^\rightarrow$, the final target language of translation. Figure 5 defines the Curry-style type system for $\lambda_e^\rightarrow$.

Figure 6 shows the dynamic semantics of $\lambda_e^\rightarrow$, while utility relations are shown in Figure 7. Figure 8 relates weaker heap types with stronger ones.

Figure 9 shows rules for translating $\lambda_d^\Downarrow$ to $\lambda_e^\rightarrow$ directly by removing type annotations (without removing checks). It also shows definition and typing rules for value environments.

2. Appendix: Proofs

2.1. Soundness of $\lambda_s^\star$.

Lemma B.1. *If* $A_1 \rhd_{\lfloor U \rfloor} A_2$*, then* $U \approx A_2$.

Proof. By cases on $A_1 \rhd_{\lfloor U \rfloor} A_2$. □

Lemma B.2. *Suppose* $U \approx A$.

(1) *If* $U \rhd U_1 \to U_2$*, then* $A \rhd_\to V_1 \to V_2$.

(2) *If* $U \rhd \mathit{ref}\, U'$*, then* $A \rhd_{\mathit{ref}} \mathit{ref}\, V$.

(3) *If* $U \sim \mathit{int}$*, then* $A \rhd_{\mathit{int}} \mathit{int}$.

$\boxed{\Gamma \vdash s : U}$

$$\textsc{SAbs}\ \frac{\Gamma, x{:}U_1 \vdash s : U_2' \qquad U_2' \sim U_2}{\Gamma \vdash \lambda(x{:}U_1){\to}U_2.\ s : U_1 \to U_2} \qquad \textsc{SApp}\ \frac{\Gamma \vdash s_1 : U \qquad U \rhd U_1 \to U_2 \qquad \Gamma \vdash s_2 : U_1' \qquad U_1' \sim U_1}{\Gamma \vdash s_1\ s_2 : U_2}$$

$$\textsc{SRef}\ \frac{\Gamma \vdash s : U_2 \qquad U_2 \sim U_1}{\Gamma \vdash \mathtt{ref}_{U_1}\ s : \mathsf{ref}\, U_1} \qquad \textsc{SDeref}\ \frac{\Gamma \vdash s : U \qquad U \rhd \mathsf{ref}\, U'}{\Gamma \vdash !s : U'}$$

$$\textsc{SUpdt}\ \frac{\Gamma \vdash s_1 : U \qquad U \rhd \mathsf{ref}\, U' \qquad \Gamma \vdash s_2 : U'' \qquad U'' \sim U'}{\Gamma \vdash s_1{:=}s_2 : \mathsf{int}}$$

$$\textsc{SAdd}\ \frac{\Gamma \vdash s_1 : U_1 \qquad U_1 \sim \mathsf{int} \qquad \Gamma \vdash s_2 : U_2 \qquad U_2 \sim \mathsf{int}}{\Gamma \vdash s_1\ +\ s_2 : \mathsf{int}} \qquad \textsc{SVar}\ \frac{\Gamma(x) = U}{\Gamma \vdash x : U} \qquad \textsc{SInt}\ \frac{}{\Gamma \vdash n : \mathsf{int}}$$

Figure 1. Type system for $\lambda_s^\star$.

Proof. We prove part 1 by cases on U. If $U = \star$, then $A = \alpha$, and $\alpha \rhd_\to \beta \to \gamma$. If $U = U_1 \to U_2$, then either $A = \alpha$ and the theorem holds as above, or $A = \mathsf{ref}\, V_1 V_2$, and $A \rhd_\to V_1 \to V_2$.

The proofs of parts 2 and 3 are similar. □

Lemma B.3. *If* $\Gamma \vdash d : A; \Omega$ *and for all* $x \in$ *dom*(Γ), $\Gamma'(x) = \Gamma(x)$, *then* $\Gamma' \vdash d : A; \Omega$.

Proof. Induction on $\Gamma \vdash d : A; \Omega$. □

Lemma B.4. *If* $\Gamma \vdash s \rightsquigarrow d : U$ *and for all* $x \in$ *dom*(Γ), $\Gamma(x) \approx \Gamma'(x)$, *then* $\Gamma' \vdash d : A; \Omega$ *and* $U \approx A$.

Proof. By induction on $\Gamma \vdash s \rightsquigarrow d : U$.

$\boxed{\Gamma \vdash d : S}$

$$\textsc{PAbs}\ \frac{\Gamma, x{:}\star \vdash d : \star}{\Gamma \vdash \lambda(x{:}X){\to}Y.\, d : \to} \qquad \textsc{PApp}\ \frac{\Gamma \vdash d_1 : \to \qquad \Gamma \vdash d_2 : \star}{\Gamma \vdash d_1\ d_2 : \star}$$

$$\textsc{PRef}\ \frac{\Gamma \vdash d : \star}{\Gamma \vdash \mathtt{ref}_X\ d : \mathsf{ref}} \qquad \textsc{PDeref}\ \frac{\Gamma \vdash d : \mathsf{ref}}{\Gamma \vdash\ !d : \star}$$

$$\textsc{PUpdt}\ \frac{\Gamma \vdash d_1 : \mathsf{ref} \qquad \Gamma \vdash d_2 : \star}{\Gamma \vdash d_1{:=}d_2 : \mathsf{int}}$$

$$\textsc{PAdd}\ \frac{\Gamma \vdash d_1 : \mathsf{int} \qquad \Gamma \vdash d_2 : \mathsf{int}}{\Gamma \vdash d_1\ +\ d_2 : \mathsf{int}} \qquad \textsc{PVar}\ \frac{\Gamma(x) = S}{\Gamma \vdash x : S} \qquad \textsc{PInt}\ \frac{}{\Gamma \vdash n : \mathsf{int}}$$

Figure 2. Simple type system for $\lambda_d^{\Downarrow}$.

$\boxed{U \approx A}$

$$\frac{}{\star \approx \alpha} \qquad \frac{}{U_1 \to U_2 \approx V_1 \to V_2}$$

$$\frac{}{\mathsf{ref}\, U \approx \mathsf{ref}\, V} \qquad \frac{}{\mathsf{int} \approx \mathsf{int}}$$

Figure 3. Relating U and A.

Case: UAbs:

$$\frac{\Gamma, x{:}U_1 \vdash s \rightsquigarrow d : U_2' \qquad U_2' \sim U_2 \qquad X, Y \text{ fresh}}{\Gamma \vdash \lambda(x{:}U_1){\to}U_2.\, s \rightsquigarrow \qquad \lambda(x{:}X){\to}Y.\, \mathtt{let}\ x = x{\Downarrow}\lfloor U_1 \rfloor\ \mathtt{in}\ d : U_1 \to U_2}$$

By IVar, $\Gamma', x : X \vdash x : X; \emptyset$.

Have that $X \rhd_{\lfloor U_1 \rfloor} A_1$.

By ICheck, $\Gamma', x : X \vdash x{\Downarrow}\lfloor U_1 \rfloor : A_1; \emptyset$.

$$
\begin{array}{rcl}
a & \in & \text{addresses} \\
e & ::= & a \mid x \mid n \mid e +^{w} e \mid \lambda x.\, e \mid (e\, e)^{w} \mid \mathtt{let}\ x = e\ \mathtt{in}\ e \mid \mathtt{ref}\ e \mid\ !e^{w} \mid e{:=}^{w} e \mid e{\Downarrow}S \mid \mathtt{fail} \\
w & ::= & \Diamond \mid \blacklozenge \\
\Sigma & ::= & \cdot \mid \Sigma, a{:}T
\end{array}
$$

Figure 4. Syntax for $\lambda_e^{\rightarrow}$.

By Lemma B.1, $U_1 \approx A_1$.

By the IH, $\Gamma', x{:}A_1 \vdash d : A_2; \Omega$ and $U_2' \approx A_2$.

By Lemma B.3, $\Gamma', x{:}X, x{:}A_1 \vdash d : A_2; \Omega$.

By ILet, $\Gamma', x{:}X \vdash \mathtt{let}\ x = x{\Downarrow}\lfloor U_1 \rfloor\ \mathtt{in}\ d : A_2; \Omega$.

By IAbs, $\Gamma' \vdash \lambda(x{:}X){\rightarrow}Y.\ \mathtt{let}\ x = x{\Downarrow}\lfloor U_1 \rfloor\ \mathtt{in}\ d : X \rightarrow Y; \Omega, A_2 <: Y$.

Have that $U_1 \rightarrow U_2 \approx X \rightarrow Y$.

Case: UApp:

$$\frac{\Gamma \vdash s_1 \rightsquigarrow d_1 : U \qquad U \rhd U_1 \rightarrow U_2 \qquad \Gamma \vdash s_2 \rightsquigarrow d_2 : U_1' \qquad U_1' \sim U_1}{\Gamma \vdash s_1\ s_2 \rightsquigarrow (d_1{\Downarrow}{\rightarrow})\ d_2{\Downarrow}\lfloor U_2 \rfloor : U_2}$$

By the IH, $\Gamma' \vdash d_1 : A_1; \Omega_1$ and $U \approx A_1$.

By Lemma B.2, $A_1 \rhd_{\rightarrow} V_1 \rightarrow V_2$.

By ICheck, $\Gamma' \vdash d_1{\Downarrow}{\rightarrow} : V_1 \rightarrow V_2$.

By the IH, $\Gamma' \vdash d_2 : A_2; \Omega_2$ and $U_1' \approx A_2$.

By IApp, $\Gamma' \vdash (d_1{\Downarrow}{\rightarrow})\ d_2 : V_2; \Omega_1, \Omega_2, A_2 <: V_1$.

Have that $V_2 \rhd_{\lfloor U_2 \rfloor} A_3$.

By ICheck, $\Gamma' \vdash ((d_1{\Downarrow}{\rightarrow})\ d_2){\Downarrow}\lfloor U_2 \rfloor : A_3; \Omega_1, \Omega_2, A_2 <: V_1$.

By Lemma B.1, $U_2 \approx A_3$.

Case: URef:

$$\frac{\Gamma \vdash s \rightsquigarrow d : U_2 \qquad U_2 \sim U_1 \qquad X \text{ fresh}}{\Gamma \vdash \mathtt{ref}_{U_1}\ s \rightsquigarrow \mathtt{ref}_X\ d : \mathsf{ref}\, U_1}$$

$\boxed{\Gamma;\Sigma \vdash e : T}$

TSubsump
$$\frac{\Gamma;\Sigma \vdash e : T_1 \qquad \vdash T_1 <: T_2}{\Gamma;\Sigma \vdash e : T_2}$$

TAbs
$$\frac{\Gamma, x{:}T_1;\Sigma \vdash e : T_2}{\Gamma \vdash \lambda x.\, e : T_1 \to T_2}$$

TLet
$$\frac{\Gamma;\Sigma \vdash e_1 : T_1 \qquad \Gamma, x{:}T_1;\Sigma \vdash e_2 : T_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : T_2}$$

TApp
$$\frac{\Gamma;\Sigma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash (e_1\ e_2)^{\Diamond} : T_2}$$

TAppOW
$$\frac{\Gamma;\Sigma \vdash e_1 : \star \qquad \Gamma \vdash e_2 : \star}{\Gamma \vdash (e_1\ e_2)^{\blacklozenge} : \star}$$

TRef
$$\frac{\Gamma;\Sigma \vdash e : T}{\Gamma;\Sigma \vdash \texttt{ref}\ e : \mathsf{ref}\, T}$$

TDeref
$$\frac{\Gamma;\Sigma \vdash e : \mathsf{ref}\, T}{\Gamma;\Sigma \vdash !e^{\Diamond} : T}$$

TDerefOW
$$\frac{\Gamma;\Sigma \vdash e : \star}{\Gamma;\Sigma \vdash !e^{\blacklozenge} : \star}$$

TUpdt
$$\frac{\Gamma;\Sigma \vdash e_1 : \mathsf{ref}\, T \qquad \Gamma;\Sigma \vdash e_2 : T}{\Gamma;\Sigma \vdash e_1 {:=}^{\Diamond} e_2 : \mathsf{int}}$$

TUpdtOW
$$\frac{\Gamma;\Sigma \vdash e_1 : \star \qquad \Gamma;\Sigma \vdash e_2 : \star}{\Gamma;\Sigma \vdash e_1 {:=}^{\blacklozenge} e_2 : \mathsf{int}}$$

TVar
$$\frac{\Gamma(x) = T}{\Gamma;\Sigma \vdash x : T}$$

TAddr
$$\frac{\Sigma(a) = T}{\Gamma;\Sigma \vdash a : \mathsf{ref}\, T}$$

TInt
$$\frac{}{\Gamma;\Sigma \vdash n : \mathsf{int}}$$

TCheckRedundant
$$\frac{\Gamma;\Sigma \vdash e : T \qquad \lfloor T \rfloor \preceq S}{\Gamma;\Sigma \vdash e \Downarrow S : T}$$

TAdd
$$\frac{\Gamma;\Sigma \vdash e_1 : \mathsf{int} \qquad \Gamma;\Sigma \vdash e_2 : \mathsf{int}}{\Gamma;\Sigma \vdash e_1\ +^{\Diamond}\ e_2 : \mathsf{int}}$$

TAddOW
$$\frac{\Gamma;\Sigma \vdash e_1 : \star \qquad \Gamma;\Sigma \vdash e_2 : \star}{\Gamma;\Sigma \vdash e_1\ +^{\blacklozenge}\ e_2 : \mathsf{int}}$$

TCheck
$$\frac{\Gamma;\Sigma \vdash e : \star}{\Gamma;\Sigma \vdash e \Downarrow S : \lceil S \rceil}$$

TFail
$$\frac{}{\Gamma;\Sigma \vdash \texttt{fail} : T}$$

TCheckFail
$$\frac{\Gamma;\Sigma \vdash e : T \qquad T \neq \star \qquad \lfloor T \rfloor \not\preceq S}{\Gamma;\Sigma \vdash e \Downarrow S : T'}$$

Figure 5. Type system for $\lambda_e^{\rightarrow}$.

By the IH, $\Gamma' \vdash d : A; \Omega$ and $U_2 \approx A$.

By IRef, $\Gamma' \vdash \texttt{ref}_X\ d : \mathsf{ref}\, X; \Omega, A <: X$.

Have that $\mathsf{ref}\, U \approx \mathsf{ref}\, X$.

$$
\begin{array}{rcl}
\varsigma & ::= & \langle e, \mu\rangle \mid \texttt{fail} \\
v & ::= & a \mid n \mid \lambda x.\, e \\
\mu & ::= & \cdot \mid \mu[a := v] \\
E & ::= & \square \mid E +^w e \mid v +^w E \mid (E\ e)^w \mid (v\ E)^w \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{ref}\ E \mid !E^w \mid E :=^w e \mid \\
 & & v :=^w E \mid E \Downarrow S
\end{array}
$$

$\boxed{\langle e, \mu\rangle \longrightarrow \varsigma}$

ECheck	$\langle v \Downarrow S, \mu\rangle$	$\longrightarrow$	$\langle v, \mu\rangle$	if *hastype*(v, S)
ECheckFail	$\langle v \Downarrow S, \mu\rangle$	$\longrightarrow$	$\texttt{fail}$	if $\neg$*hastype*(v, S)
EFail	$\langle \texttt{fail}, \mu\rangle$	$\longrightarrow$	$\texttt{fail}$	
ERef	$\langle \texttt{ref}\ v, \mu\rangle$	$\longrightarrow$	$\langle a, \mu[a := v]\rangle$	where a fresh
EDeref	$\langle !a^w, \mu\rangle$	$\longrightarrow$	$\langle v, \mu\rangle$	where $\mu(a) = v$
EUpdt	$\langle a :=^w v, \mu\rangle$	$\longrightarrow$	$\langle 0, \mu[a := v]\rangle$	where $\mu(a) = v'$
EApp	$\langle ((\lambda x.\, e)\ v)^w, \mu\rangle$	$\longrightarrow$	$\langle e[x/v], \mu\rangle$	
ELet	$\langle \texttt{let}\ x = v\ \texttt{in}\ e, \mu\rangle$	$\longrightarrow$	$\langle e[x/v], \mu\rangle$	
EAdd	$\langle n_1\ +^w\ n_2, \mu\rangle$	$\longrightarrow$	$\langle n', \mu\rangle$	where $n_1 + n_2 = n'$

$$
\frac{\langle e, \mu\rangle \longrightarrow \langle e', \mu'\rangle}{\langle E[e], \mu\rangle \longmapsto \langle E[e'], \mu'\rangle} \qquad \frac{\langle e, \mu\rangle \longrightarrow \texttt{fail}}{\langle E[e], \mu\rangle \longmapsto \texttt{fail}}
$$

$\boxed{\textit{hastype}(v, S)}$

$$
\frac{}{\textit{hastype}(v, \star)} \qquad \frac{}{\textit{hastype}(n, \text{int})} \qquad \frac{}{\textit{hastype}(\lambda x.\, e, \to)} \qquad \frac{}{\textit{hastype}(a, \text{ref})}
$$

Figure 6. Dynamic semantics for $\lambda_e^{\to}$.

Case: UDeref:

$$
\frac{\Gamma \vdash s \rightsquigarrow d : U \qquad U \rhd \text{ref}\ U'}{\Gamma \vdash !s \rightsquigarrow !(d \Downarrow \text{ref}) \Downarrow \lfloor U' \rfloor : U'}
$$

By the IH, $\Gamma' \vdash d : A_1; \Omega$ and $U \approx A_1$.

By Lemma B.2, $A_1 \rhd_{\text{ref}} \text{ref}\ V$.

$\boxed{\langle e, \mu\rangle \text{ stuck } w}$

$$\frac{}{\langle (n\ v)^w, \mu\rangle \text{ stuck } w} \qquad \frac{}{\langle (a\ v)^w, \mu\rangle \text{ stuck } w}$$

$$\frac{}{\langle a +^w v, \mu\rangle \text{ stuck } w} \qquad \frac{}{\langle (\lambda x.\ e) +^w v, \mu\rangle \text{ stuck } w}$$

$$\frac{}{\langle n +^w a, \mu\rangle \text{ stuck } w} \qquad \frac{}{\langle n +^w (\lambda x.\ e), \mu\rangle \text{ stuck } w} \qquad \frac{a \notin \mathit{dom}(\mu)}{\langle !a^w, \mu\rangle \text{ stuck } w} \qquad \frac{}{\langle !n^w, \mu\rangle \text{ stuck } w}$$

$$\frac{}{\langle !(\lambda x.\ e)^w, \mu\rangle \text{ stuck } w} \qquad \frac{a \notin \mathit{dom}(\mu)}{\langle a :=^w v, \mu\rangle \text{ stuck } w} \qquad \frac{}{\langle n :=^w v, \mu\rangle \text{ stuck } w}$$

$$\frac{}{\langle (\lambda x.\ e) :=^w v, \mu\rangle \text{ stuck } w} \qquad \frac{\langle e, \mu\rangle \text{ stuck } w}{\langle E[e], \mu\rangle \text{ stuck } w}$$

$\boxed{\Sigma \vdash \mu}$

$$\frac{\mathit{dom}(\Sigma) = \mathit{dom}(\mu) \qquad \forall a \in \mathit{dom}(\Sigma),\ \emptyset; \Sigma \vdash \mu(a) : \Sigma(a)}{\Sigma \vdash \mu}$$

Figure 7. Additional semantics for $\lambda_e^{\rightarrow}$.

$\boxed{\Sigma \sqsubseteq \Sigma}$

$$\frac{\forall a \in \mathit{dom}(\Sigma_2),\ \Sigma_1(a) = \Sigma_2(a)}{\Sigma_1 \sqsubseteq \Sigma_2}$$

Figure 8. Weaker and stronger heap types.

By ICheck, $\Gamma' \vdash d\Downarrow\text{ref} : \text{ref}\, V; \Omega$.

By IDeref, $\Gamma' \vdash !(d\Downarrow\text{ref}) : V; \Omega$.

Have that $V \rhd_{\lfloor U' \rfloor} A_2$.

By ICheck, $\Gamma' \vdash !(d\Downarrow\text{ref})\Downarrow\lfloor U' \rfloor : A_2; \Omega$.

By Lemma B.1, $U' \approx A_2$.

$$\boxed{|d| = e}$$

$$\begin{array}{rcl} |x| & = & x \\ |n| & = & n \\ |\lambda(x{:}X){\to}Y.\, d| & = & \lambda x.\, |d| \\ |d_1\ d_2| & = & (|d_1|\ |d_2|)^{\Diamond} \\ |\mathtt{ref}_X\ d| & = & \mathtt{ref}\ |d| \\ |!d| & = & !|d|^{\Diamond} \\ |d_1{:=}d_2| & = & |d_1|{:=}^{\Diamond}|d_2| \\ |d_1\ +\ d_2| & = & |d_1|\ +^{\Diamond}\ |d_2| \\ |d{\Downarrow}S| & = & |d|{\Downarrow}S \end{array}$$

$$\rho \ ::= \ \cdot \mid \rho, x = v$$

$$\boxed{\Sigma \vdash \rho : \Gamma}$$

$$\frac{}{\Sigma \vdash \cdot : \emptyset} \qquad \frac{\emptyset; \Sigma \vdash v : T \qquad \Sigma \vdash \rho : \Gamma}{\Sigma \vdash \rho, x = v : \Gamma, x : T}$$

Figure 9. Rules for environments and for erasing to $\lambda_d^{\Downarrow}$ to $\lambda_e^{\rightarrow}$.

Case: UUpdt:

$$\frac{\Gamma \vdash s_1 \rightsquigarrow d_1 : U \qquad U \rhd \mathsf{ref}\, U' \qquad \Gamma \vdash s_2 \rightsquigarrow d_2 : U'' \qquad U'' \sim U'}{\Gamma \vdash s_1{:=}s_2 \rightsquigarrow d_1{\Downarrow}\mathsf{ref}{:=}d_2 : \mathsf{int}}$$

By the IH, $\Gamma' \vdash d_1 : A_1; \Omega_1$ and $U \approx A_1$.

By Lemma B.2, $A_1 \rhd_{\mathsf{ref}} \mathsf{ref}\, V$.

By ICheck, $\Gamma' \vdash d_1{\Downarrow}\mathsf{ref} : \mathsf{ref}\, V; \Omega_1$.

By the IH, $\Gamma' \vdash d_2 : A_2; \Omega_2$ and $U'' \approx A_2$.

By IUpdt, $\Gamma' \vdash d_1{\Downarrow}\mathsf{ref}{:=}d_2 : \mathsf{int}; \Omega_1, \Omega_2, A_2 <: V$.

Case: UAdd:

$$\frac{\Gamma \vdash s_1 \rightsquigarrow d_1 : U_1 \qquad U_1 \sim \mathsf{int} \qquad \Gamma \vdash s_2 \rightsquigarrow d_2 : U_2 \qquad U_2 \sim \mathsf{int}}{\Gamma \vdash s_1\ +\ s_2 \rightsquigarrow d_1{\Downarrow}\mathsf{int}\ +\ d_2{\Downarrow}\mathsf{int} : \mathsf{int}}$$

By the IH, $\Gamma' \vdash d_1 : A_1; \Omega_1$ and $U_1 \approx A_1$.

By Lemma B.2, $A_1 \rhd_{\mathsf{int}} \mathsf{int}$.

By ICheck, $\Gamma' \vdash d_1{\Downarrow}\mathsf{int} : \mathsf{int}; \Omega_1$.

By the IH, $\Gamma' \vdash d_2 : A_2; \Omega_2$ and $U_2 \approx A_2$.

By Lemma B.2, $A_2 \rhd_{\mathsf{int}} \mathsf{int}$.

By ICheck, $\Gamma' \vdash d_2 \Downarrow \text{int} : \text{int}; \Omega_2$.

By IAdd, $\Gamma' \vdash d_1 \Downarrow \text{int} \ + \ d_2 \Downarrow \text{int} : \text{int}; \Omega_1, \Omega_2$.

Case: UVar:

$$\frac{\Gamma(x) = U}{\Gamma \vdash x \rightsquigarrow x : U}$$

Have that $\Gamma'(x) = A$ and $U \approx A$.

By IVar, $\Gamma' \vdash x : A; \emptyset$.

Case: UInt: Immediate.

□

2.2. Soundness of $\lambda_d^{\Downarrow}$.

Lemma B.5. *If σ is a solution to $\Omega_1 \cup \Omega_2$, then σ is a solution to Ω_1 and σ is a solution to Ω_2.*

Proof. Since σ is a solution for every constraint in Ω, and for all $C \in \Omega_1$, $C \in \Omega$, so σ is a solution for every constraint in Ω_1, so it is a solution to Ω_1. Likewise for Ω_1. □

Lemma B.6. *If $\vdash T_1 <: T_2$ and $\vdash T_2 <: T_3$, then $\vdash T_1 <: T_3$.*

Proof. Straightforward induction. □

Lemma B.7. *If $\lfloor T_1 \rfloor \preceq S$ and $\vdash T_2 <: T_1$, then $\lfloor T_2 \rfloor \preceq S$.*

Proof. Cases on T_1. □

Lemma B.8. *If $\Gamma \vdash d : A; \Omega$ and σ is a solution for Ω, then $\Gamma; \sigma \vdash d \rightsquigarrow e : T$ and $\vdash T <: \sigma A$.*

Proof. By induction on $\Gamma \vdash d : A; \Omega$

Case: IVar:

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A; \emptyset}$$

Have that $\Gamma(x) = A$.

By DVar, $\Gamma; \sigma \vdash x \rightsquigarrow x : \sigma A$.

Case: IAbs:

$$\frac{\Gamma, x{:}\alpha \vdash d : A; \Omega}{\Gamma \vdash \lambda(x{:}\alpha){\to}\beta.\ d : \alpha \to \beta; \Omega, A <: \beta}$$

Since σ is a solution for $\Omega \cup \{A <: \beta\}$, by Lemma B.5, σ is a solution for Ω.

By the IH, $\Gamma, x{:}\alpha; \sigma \vdash d \rightsquigarrow e : T$ and $\vdash T' <: \sigma A$.

Since σ is a solution to $\Omega, A <: \beta$, $\vdash \sigma A <: \sigma\beta$.

By Lemma B.6, $\vdash T' <: \sigma\beta$.

By DAbs, $\Gamma; \sigma \vdash \lambda(x{:}\alpha){\to}\beta.\ d \rightsquigarrow \lambda x.\ e : \sigma\alpha \to \sigma\beta$.

Case: IApp:

$$\frac{\Gamma \vdash d_1 : V_1 \to V_2; \Omega_1 \qquad \Gamma \vdash d_2 : A; \Omega_2}{\Gamma \vdash d_1\ d_2 : V_2; \Omega_1, \Omega_2, A <: V_1}$$

Since σ is a solution for $\Omega_1, \Omega_2, A_1 <: V_1, A <: V_1 \to V_2$, by Lemma B.5, σ is a solution for Ω_1 and σ is a solution for Ω_2.

By the IH, $\Gamma; \sigma \vdash d_1 \rightsquigarrow e_1 : T_1$ and $\vdash T_1 <: \sigma V_1 \to \sigma V_2$.

Therefore $T_1 = T_{11} \to T_{12}$ and $\vdash \sigma V_1 <: T_{11}$ and $\vdash T_{12} <: \sigma V_2$.

By the IH, $\Gamma; \sigma \vdash d_2 \rightsquigarrow e_2 : T_2$ and $\vdash T_2 <: \sigma A$.

Since σ is a solution to $\Omega, A <: V_1$, $\vdash \sigma A <: \sigma V_1$.

By Lemma B.6, $\vdash T_2 <: T_{11}$.

By DApp, $\Gamma; \sigma \vdash d_1\ d_2 \rightsquigarrow (e_1\ e_2)^{\Diamond} : T_{12}$.

Case: ICheck:

$$\frac{\Gamma \vdash d : A_1; \Omega \qquad A_1 \rhd_S A_2}{\Gamma \vdash d{\Downarrow}S : A_2; \Omega; (A_1{:}S) = A_2}$$

By the IH, $\Gamma; \sigma \vdash d \rightsquigarrow e : T$ and $\vdash T <: \sigma A_1$.

If $S = \star$, then $A_2 = A_1$, and by DCheckRemove, $\Gamma; \sigma \vdash d{\Downarrow}S \rightsquigarrow e : T$.

We proceed by cases on A_1 and $\lfloor \sigma A_1 \rfloor$.

Case: $A_1 \neq \alpha$ and $S = \lfloor \sigma A_1 \rfloor$:

Then $S = \lfloor A_1 \rfloor$.

Then $A_2 = A_1$.

By Lemma B.7, $S = \lfloor T \rfloor$.

By DCheckRemove, $\Gamma; \sigma \vdash d{\Downarrow}S \rightsquigarrow e : T$.

Case: $A_1 \neq \alpha$ and S is not the constructor of σA_1: Vacuous, since $S \neq \star$.

Case: $A_1 = \alpha$ and S is the constructor of σA_1:

Then $\sigma A_1 = \sigma A_2$.

Therefore $\vdash T <: \sigma A_2$.

By Lemma B.7, $\lfloor T \rfloor \preceq S$.

By DCheckRemove, $\Gamma; \sigma \vdash d{\Downarrow}S \rightsquigarrow e : T$.

Case: $A_1 = \alpha$ and S is not the constructor of σA_1:

Since σ is a solution for $\Omega \cup \{(A_1{:}S) = A_2\}$, for all $\beta \in \mathit{parts}(A_2)$, $\sigma\beta = \star$.

By the definition of $\rhd_S$, since $S \neq \star$, $\lfloor A_2 \rfloor = S$.

Therefore $\sigma A_2 = \lceil S \rceil$.

If $T = \star$, then by DCheckKeep, $\Gamma; \sigma \vdash d{\Downarrow}S \rightsquigarrow e{\Downarrow}S : \lceil S \rceil$.

Otherwise, by DCheckFail, $\Gamma; \sigma \vdash d{\Downarrow}S \rightsquigarrow \texttt{fail} : \lceil S \rceil$.

Case: IRef:

$$\frac{\Gamma \vdash d : A; \Omega}{\Gamma \vdash \texttt{ref}_\alpha\, d : \mathsf{ref}\,\alpha; \Omega, A <: \alpha}$$

By the IH, $\Gamma; \sigma \vdash d \rightsquigarrow e : T'$ and $\vdash T' <: \sigma A$.

Since σ is a solution to $\Omega \cup \{A <: \alpha\}$, $\vdash \sigma A <: \sigma\alpha$.

By Lemma B.6, $\vdash T' <: \sigma\alpha$.

By DRef, $\Gamma; \sigma \vdash \texttt{ref}_\alpha\, d \rightsquigarrow \texttt{ref}\ e : \mathsf{ref}\,\sigma\alpha$.

Case: IDeref:

$$\frac{\Gamma \vdash d : \mathsf{ref}\,V; \Omega}{\Gamma \vdash !d : V; \Omega}$$

By the IH, $\Gamma; \sigma \vdash d \rightsquigarrow e : T$ and $\vdash T <: \mathsf{ref}\,\sigma V$.

Therefore $T = \mathsf{ref}\,T'$ and $\vdash T' <: \sigma V$.

By DDeref, $\Gamma; \sigma \vdash !d \rightsquigarrow !e^{\Diamond} : T'$.

Case: IUpdt:

$$\frac{\Gamma \vdash d_1 : \text{ref}\, V; \Omega_1 \qquad \Gamma \vdash d_2 : A; \Omega_2}{\Gamma \vdash d_1{:=}d_2 : \text{int}; \Omega_1, \Omega_2, A <: V}$$

Since σ is a solution to $\Omega_1 \cup \Omega_2 \cup \{A <: V\}$, by Lemma B.5, σ is a solution to Ω_1 and σ is a solution to Ω_2 and σ is a solution to $\{A <: V\}$.

By the IH, $\Gamma; \sigma \vdash d_1 \rightsquigarrow e_1 : T_1$ and $\vdash T_1 <: \text{ref}\, \sigma V$.

Therefore $T_1 = \text{ref}\, T_1'$ and $\vdash \sigma V <: T_1'$

By the IH, $\Gamma; \sigma \vdash d_2 \rightsquigarrow e_2 : T_2$ and $\vdash T_2 <: \sigma A$.

Since σ is a solution to $\{A <: V\}$, $\vdash \sigma A <: \sigma V$.

By Lemma B.6, $\vdash T_2 <: T_1'$.

By DDeref, $\Gamma; \sigma \vdash d_1{:=}d_2 \rightsquigarrow e_1{:=}^{\Diamond}e_2 : \text{int}$.

Case: IAdd:

$$\frac{\Gamma \vdash d_1 : \text{int}; \Omega_1 \qquad \Gamma \vdash d_2 : \text{int}; \Omega_2}{\Gamma \vdash d_1 \ + \ d_2 : \text{int}; \Omega_1, \Omega_2}$$

Since σ is a solution to $\Omega_1 \cup \Omega_2$, by Lemma B.5, σ is a solution to Ω_1 and σ is a solution to Ω_2.

By the IH, $\Gamma; \sigma \vdash d_1 \rightsquigarrow e_1 : T_1$ and $\vdash T_1 <: \text{int}$.

By the IH, $\Gamma; \sigma \vdash d_2 \rightsquigarrow e_2 : T_2$ and $\vdash T_2 <: \text{int}$.

Therefore $T_1 = T_2 = \text{int}$.

Therefore by DAdd, $\Gamma; \sigma \vdash d_1 \ + \ d_2 \rightsquigarrow (e_1 \ + \ e_2)^{\Diamond} : \text{int}$.

Case : IInt: Immediate by DInt.

□

Lemma B.9. *If* $\Gamma \vdash d : A; \Omega_1$ *and* $\vdash A : \Omega_2$ *and* σ *is a solution for* $\Omega_1 \cup \Omega_2$*, then* $\Gamma; \sigma \vdash d \rightsquigarrow e : T$ *and* $\vdash T <: \sigma A$.

Proof. By Lemma B.5, σ is a solution for Ω_1.

By Lemma B.8, $\Gamma; \sigma \vdash d \rightsquigarrow e : T$ and $\vdash T <: \sigma A$. □

Lemma B.10. *If* $\Gamma; \sigma \vdash d \rightsquigarrow e : T$*, then* $\sigma\Gamma; \emptyset \vdash e : T$.

Proof. By induction on $\Gamma; \sigma \vdash d \rightsquigarrow e : T$.

Case : OAbs:

$$\frac{\Gamma, x{:}\alpha; \sigma \vdash d \rightsquigarrow e : T \qquad \vdash T <: \sigma\beta}{\Gamma; \sigma \vdash \lambda(x{:}\alpha){\to}\beta.\, d \rightsquigarrow \lambda x.\, e : \sigma\alpha \to \sigma\beta}$$

By the IH, $\sigma\Gamma, x{:}\sigma\alpha;\emptyset \vdash e : T$.

By TSubsump, $\sigma\Gamma, x{:}T_1;\emptyset \vdash e : \sigma\beta$.

By TAbs, $\sigma\Gamma;\emptyset \vdash \lambda x.\, e : \sigma\alpha \to \sigma\beta$.

Case : OLet:

$$\frac{\Gamma; \sigma \vdash d_1 \rightsquigarrow e_1 : T_1 \qquad \Gamma, x{:}T_1; \sigma \vdash d_2 \rightsquigarrow e_2 : T_2}{\Gamma; \sigma \vdash \texttt{let}\, x = d_1\, \texttt{in}\, d_2 \rightsquigarrow \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : T_2}$$

By the IH, $\sigma\Gamma;\emptyset \vdash e_1 : T_1$.

By the IH, $\sigma\Gamma, x{:}T_1;\emptyset \vdash e_2 : T_2$.

By TLet, $\sigma\Gamma;\emptyset \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : T_2$.

Case : OApp:

$$\frac{\Gamma; \sigma \vdash d_1 \rightsquigarrow e_1 : T_1 \to T_2 \qquad \Gamma; \sigma \vdash d_2 \rightsquigarrow e_2 : T_1' \qquad \vdash T_1' <: T_1}{\Gamma; \sigma \vdash d_1\, d_2 \rightsquigarrow (e_1\, e_2)^{\Diamond} : T_2}$$

By the IH, $\sigma\Gamma;\emptyset \vdash e_1 : T_1 \to T_2$.

By the IH, $\sigma\Gamma;\emptyset \vdash e_2 : T_1'$.

By TSubsump, $\sigma\Gamma;\emptyset \vdash e_2 : T_2$.

By TApp, $\sigma\Gamma;\emptyset \vdash (e_1\, e_2)^{\Diamond} : T_2$.

Case : ORef:

$$\frac{\Gamma; \sigma \vdash d \rightsquigarrow e : T \qquad \vdash T <: \sigma\alpha}{\Gamma; \sigma \vdash \texttt{ref}_{\alpha}\, d \rightsquigarrow \texttt{ref}\, e : \mathsf{ref}\, T}$$

By the IH, $\sigma\Gamma;\emptyset \vdash e : T$.

By TSubsump, $\sigma\Gamma;\emptyset \vdash e : \sigma\alpha$.

By TRef, $\sigma\Gamma;\emptyset \vdash \texttt{ref}\, e : \mathsf{ref}\, \sigma\alpha$.

Case : ODeref:

$$\frac{\Gamma; \sigma \vdash d \rightsquigarrow e : \text{ref}\, T}{\Gamma; \sigma \vdash !d \rightsquigarrow !e^{\Diamond} : T}$$

By the IH, $\sigma\Gamma;\emptyset \vdash e : \text{ref}\, T$.

By TDeref, $\sigma\Gamma;\emptyset \vdash !e^{\Diamond} : T$.

Case : OUpdt:

$$\frac{\Gamma; \sigma \vdash d_1 \rightsquigarrow e_1 : \text{ref}\, T \qquad \Gamma; \sigma \vdash d_2 \rightsquigarrow e_2 : T' \qquad \vdash T' <: T}{\Gamma; \sigma \vdash d_1 {:=} d_2 \rightsquigarrow e_1 {:=}^{\Diamond} e_2 : \text{int}}$$

By the IH, $\sigma\Gamma;\emptyset \vdash e_1 : \text{ref}\, T$.

By the IH, $\sigma\Gamma;\emptyset \vdash e_2 : T'$.

By TSubsump, $\sigma\Gamma;\emptyset \vdash e_2 : T$.

By TDeref, $\sigma\Gamma;\emptyset \vdash e_1 {:=}^{\Diamond} e_2 : \text{int}$.

Case : OVar:

$$\frac{\Gamma(x) = A}{\Gamma; \sigma \vdash x \rightsquigarrow x : \sigma A}$$

Have that $\sigma\Gamma(x) = \sigma A$.

By TVar, $\sigma\Gamma;\emptyset \vdash x : \sigma A$.

Case : OInt: Immediate from TInt.

Case : OAdd:

$$\frac{\Gamma; \sigma \vdash d_1 \rightsquigarrow e_1 : \text{int} \qquad \Gamma; \sigma \vdash d_2 \rightsquigarrow e_2 : \text{int}}{\Gamma; \sigma \vdash d_1 \ + \ d_2 \rightsquigarrow e_1 \ +^{\Diamond} \ e_2 : \text{int}}$$

By the IH, $\sigma\Gamma;\emptyset \vdash e_1 : \text{int}$.

By the IH, $\sigma\Gamma;\emptyset \vdash e_2 : \text{int}$.

By TAdd, $\sigma\Gamma;\emptyset \vdash e_1 \ +^{\Diamond} \ e_2 : \text{int}$.

Case : OCheckRemove:

$$\frac{\Gamma; \sigma \vdash d \rightsquigarrow e : T \qquad \lfloor T \rfloor \preceq S}{\Gamma; \sigma \vdash d \Downarrow S \rightsquigarrow e : T}$$

Immediate from the IH.

Case : OCheckKeep:

$$\frac{\Gamma;\sigma \vdash d \rightsquigarrow e : \star}{\Gamma;\sigma \vdash d{\Downarrow}S \rightsquigarrow e{\Downarrow}S : \lceil S \rceil}$$

By the IH, $\sigma\Gamma;\emptyset \vdash e : \star$.

By TCheck, $\sigma\Gamma;\emptyset \vdash e{\Downarrow}S : \lceil S \rceil$.

Case : OCheckFail:

$$\frac{\Gamma;\sigma \vdash d \rightsquigarrow e : T \qquad T \neq \star \qquad \lfloor T \rfloor \not\preceq S}{\Gamma;\sigma \vdash d{\Downarrow}S \rightsquigarrow \texttt{fail} : T'}$$

Immediate from TFail.

□

2.3. Soundness of $\lambda_e^{\rightarrow}$.

Lemma B.11 (Inversion). *Suppose* $\Gamma;\Sigma \vdash e : T$. *Then:*

- *If* $e = \lambda x.\, e$, *then there exists* T_1 *such that* $\Gamma, x{:}T_1;\Sigma \vdash e : T_2$ *and* $\vdash T_1 \rightarrow T_2 <: T$.
- *If* $e = a$, *then* $\Sigma(a) = T'$ *and* $\vdash \mathit{ref}\, T' <: T$.
- *If* $e = n$, *then* $\vdash \mathit{int} <: T$.
- *If* $e = e{\Downarrow}S$, *then either:*
  (1) $\Gamma;\Sigma \vdash e : \star$ *and* $\vdash \lceil S \rceil <: T$, *or*
  (2) $\Gamma;\Sigma \vdash e : T'$ *and* $\lfloor T' \rfloor \preceq S$ *and* $\vdash T' <: T$, *or*
  (3) $\Gamma;\Sigma \vdash e : T''$ *and* $\lfloor T' \rfloor \not\preceq S$ *and* $T'' \neq \star$.
- *If* $e = x$, *then* $\Gamma(x) = T'$ *and* $\vdash T' <: T$.
- *If* $e = (e_1\ e_2)^{\Diamond}$, *then* $\Gamma;\Sigma \vdash e_1 : T_1 \rightarrow T_2$ *and* $\Gamma;\Sigma \vdash e_2 : T_1$ *and* $\vdash T_2 <: T$.
- *If* $e = (e_1\ e_2)^{\blacklozenge}$, *then* $\Gamma;\Sigma \vdash e_1 : \star$ *and* $\Gamma;\Sigma \vdash e_2 : \star$ *and* $\vdash \star <: T$.
- *If* $e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$, *then* $\Gamma;\Sigma \vdash e_1 : T_1$ *and* $\Gamma, x : T_1;\Sigma \vdash e_2 : T_2$ *and* $\vdash T_2 <: T$.
- *If* $e = \texttt{ref}\, e$, *then* $\Gamma;\Sigma \vdash e : T'$ *and* $\vdash \mathit{ref}\, T' <: T$.
- *If* $e = !e^{\Diamond}$, *then* $\Gamma;\Sigma \vdash e : \mathit{ref}\, T'$ *and* $\vdash T' <: T$.
- *If* $e = !e^{\blacklozenge}$, *then* $\Gamma;\Sigma \vdash e : \star$ *and* $\vdash \star <: T$.

- *If* $e = e_1 :=^{\Diamond} e_2$*, then* $\Gamma;\Sigma \vdash e_1 : \mathit{ref}\, T'$ *and* $\Gamma;\Sigma \vdash e_2 : T'$ *and* $\vdash \mathit{int} <: T$.
- *If* $e = e_1 :=^{\blacklozenge} e_2$*, then* $\Gamma;\Sigma \vdash e_1 : \star$ *and* $\Gamma;\Sigma \vdash e_2 : \star$ *and* $\vdash \mathit{int} <: T$.
- *If* $e = e_1 \ +^{\Diamond}\ e_2$*, then* $\Gamma;\Sigma \vdash e_1 : \mathit{int}$ *and* $\Gamma;\Sigma \vdash e_2 : \mathit{int}$ *and* $\vdash \mathit{int} <: T$.
- *If* $e = e_1 \ +^{\blacklozenge}\ e_2$*, then* $\Gamma;\Sigma \vdash e_1 : \star$ *and* $\Gamma;\Sigma \vdash e_2 : \star$ *and* $\vdash \mathit{int} <: T$.

Proof. Induction on $\Gamma;\Sigma \vdash e : R$. □

Lemma B.12 (Canonical forms). *Suppose* $\emptyset;\Sigma \vdash v : T$ *and* $\Sigma \vdash \mu$.

(1) *If* $T = \mathit{int}$*, then* $v = n$.

(2) *If* $T = T_1 \to T_2$*, then* $v = \lambda x.\, e$.

(3) *If* $T = \mathit{ref}\, T'$*, then* $v = a$ *and* $\mu(a) = v'$.

(4) *If* $T = \star$*, then* $\exists T'$, $T' \neq \star$*, such that* $\emptyset;\Sigma \vdash v : T'$.

Proof. We prove each part separately.

(1) We prove by induction on $\emptyset;\Sigma \vdash v :$ int. Most cases vacuous.

Case : TInt

$$\frac{}{\emptyset;\Sigma \vdash n : \text{int}}$$

Immediate.

Case : TSubsump

$$\frac{\emptyset;\Sigma \vdash v : T_1 \qquad \vdash T_1 <: \text{int}}{\emptyset;\Sigma \vdash v : \text{int}}$$

Since $\vdash T_1 <:$ int, $T_1 =$ int.

By the IH, $v = n$.

(2) We prove by induction on $\emptyset;\Sigma \vdash v : T_1 \to T_2$. Most cases vacuous.

Case : TAbs

$$\frac{x{:}T_1;\Sigma \vdash e : T_2}{\emptyset;\Sigma \vdash \lambda x.\, e : T_1 \to T_2}$$

Immediate.

Case : TSubsump

$$\frac{\emptyset;\Sigma \vdash v : T' \qquad \vdash T' <: T_1 \to T_2}{\emptyset;\Sigma \vdash v : T_1 \to T_2}$$

Since $\vdash T' <: T_1 \to T_2$, $T' = T_1' \to T_2'$.

By the IH, $v = \lambda x.\, e$.

(3) We prove by induction on $\emptyset;\Sigma \vdash v : \text{ref}\, T'$. Most cases vacuous.

Case : TAddr

$$\frac{\Sigma(a) = T'}{\emptyset;\Sigma \vdash a : \text{ref}\, T'}$$

Immediately have $v = a$. Since $\Sigma(a) = T'$, exists v' such that $\mu(a) = v'$.

Case : TSubsump

$$\frac{\emptyset;\Sigma \vdash v : T'' \qquad \vdash T'' <: \text{ref}\, T'}{\emptyset;\Sigma \vdash v : \text{ref}\, T'}$$

Since $\vdash T'' <: \text{ref}\, T'$, $T'' = \text{ref}\, T'''$.

By the IH, $v = a$ and $\mu(a) = v'$.

(4) We prove by induction on $\emptyset;\Sigma \vdash v : \star$. Most cases vacuous.

Case : TSubsump

$$\frac{\emptyset;\Sigma \vdash v : T' \qquad \vdash T' <: \star}{\emptyset;\Sigma \vdash v : \star}$$

If $T' = \star$, then we apply the IH to find that $\exists T''$, $T'' \neq \star$, such that $\emptyset;\Sigma \vdash v : T''$.

If $T' \neq \star$, then immediate.

□

Lemma B.13 (Heap weakening). *If* $\Gamma;\Sigma \vdash e : T$ *and* $\Sigma' \sqsubseteq \Sigma$*, then* $\Gamma;\Sigma' \vdash e : T$.

Proof. By induction on $\Gamma;\Sigma \vdash e : T$. Only interesting case:

Case : TAddr:

$$\frac{\Sigma(a) = T}{\Gamma;\Sigma \vdash a : \text{ref}\, T}$$

Because $\Sigma' \sqsubseteq \Sigma$, $\Sigma'(a) = T$.

By TAddr, $\Gamma;\Sigma \vdash a : \text{ref}\, T$.

□

Lemma B.14 (Heap extension). *If* $\Sigma \vdash \mu$ *and* $\emptyset;\Sigma,a{:}T \vdash v : T$ *and* $a \notin$ *dom*(Σ)*, then* $\Sigma,a{:}T \vdash \mu[a := v]$.

Proof. Since $a \notin$ *dom*(Σ), for all $a' \in$ *dom*(Σ), $\Sigma(a') = (\Sigma,a{:}T)(a')$.

Therefore $\Sigma,a{:}T \sqsubseteq \Sigma$.

Suppose $a' \in$ *dom*$(\Sigma,a{:}T)$. If $a = a'$, then immediately $\emptyset;\Sigma,a{:}T \vdash \mu[a := v](a) : (\Sigma,a{:}T)(a)$.

If $a \neq a'$, then $\emptyset;\Sigma \vdash \mu(a') : \Sigma(a')$.

Immediately have $\mu(a') = \mu[a := v](a')$ and $\Sigma(a') = (\Sigma,a{:}T)(a')$.

By Lemma B.13, $\emptyset;\Sigma,a{:}T \vdash \mu[a := v](a') : (\Sigma,a{:}T)(a')$.

Therefore, $\Sigma,a{:}T \vdash \mu[a := v]$. □

Lemma B.15 (Substitution). *If* $\Gamma,x{:}T_1;\Sigma \vdash e : T_2$ *and* $\Gamma;\Sigma \vdash v : T_1$*, then* $\Gamma;\Sigma \vdash e[x/v] : T_2$.

Proof. By induction on $\Gamma,x{:}T_1;\Sigma \vdash e : T_2$. Only interesting cases:

Case: TVar:

$$\frac{(\Gamma,x{:}T_1)(y) = T_2}{\Gamma,x{:}T_1;\Sigma \vdash y : T_2}$$

If $y \neq x$, then $\Gamma(x) = T_2$. Then by TVar, $\Gamma;\Sigma \vdash y : T_2$.

If $y = x$, then $y[x/v] = v$ and $(\Gamma,x{:}T_1)(x) = T_1$, so $T_1 = T_2$. Have immediately that $\Gamma;\Sigma \vdash v : T_2$.

Case: TAbs:

$$\frac{\Gamma,x{:}T_1,y{:}T_1';\Sigma \vdash e : T_2'}{\Gamma,x{:}T_1;\Sigma \vdash \lambda y.\, e : T_1' \to T_2'}$$

If $y = x$, then $(\lambda y.\, e)[x/v] = \lambda y.\, e$ and $\Gamma,x{:}T_1,y{:}T_1' \equiv \Gamma,y{:}T_1'$.

Therefore $\Gamma,y{:}T_1';\Sigma \vdash e : T_2'$.

By TAbs, $\Gamma;\Sigma \vdash \lambda y.\, e : T_1' \to T_2'$.

If $y \neq x$, then Γ,x:T_1,y:$T_1' \equiv \Gamma$,y:T_1',x:T_1.

Therefore Γ,y:T_1',x:T_1;$\Sigma \vdash e : T_2'$.

By the IH, Γ,y:$T_1'\Sigma \vdash e[x/v] : T_2'$.

By TAbs, $\Gamma;\Sigma \vdash \lambda y.\, e[x/v] : T_1' \to T_2'$.

Case: TLet:

$$\frac{\Gamma,x{:}T_1;\Sigma \vdash e_1 : T' \qquad \Gamma,x{:}T_1,y{:}T';\Sigma \vdash e_2 : T_2}{\Gamma,x{:}T_1;\Sigma \vdash \texttt{let}\ y = e_1\ \texttt{in}\ e_2 : T_2}$$

By the IH, $\Gamma;\Sigma \vdash e_1[x/v] : T'$.

If $y = x$, then $(\texttt{let}\ y = e_1\ \texttt{in}\ e_2)[x/v] = \texttt{let}\ y = e_1[x/v]\ \texttt{in}\ e_2$ and Γ,x:T_1,y:$T' \equiv \Gamma$,y:T'.

Therefore Γ,y:T';$\Sigma \vdash e_2 : T_2$.

By TLet, $\Gamma;\Sigma \vdash \texttt{let}\ y = e_1[x/v]\ \texttt{in}\ e_2 : T_2$.

If $y \neq x$, then Γ,x:T_1,y:$T' \equiv \Gamma$,y:T',x:T_1.

Therefore Γ,y:T',x:T_1;$\Sigma \vdash e_2 : T_2$.

By the IH, Γ,y:$T'\Sigma \vdash e_2[x/v] : T_2$.

By TLet, $\Gamma;\Sigma \vdash \texttt{let}\ y = e_1[x/v]\ \texttt{in}\ e_2[x/v] : T_2$.

□

Lemma B.16. *If* $\emptyset;\Sigma \vdash v : T$ *and* $\lfloor T \rfloor \not\preceq S$ *and* $S \not\preceq \lfloor T \rfloor$*, then* $\neq$ *hastype*(v, S).

Proof. By cases on T.

Case: $T =$ int: By Lemma B.12, $v = n$. Have that $S \notin \{\text{int}, \star\}$, so $\neg$*hastype*(n, S).

Case: $T = T_1 \to T_2$: By Lemma B.12, $v = \lambda x.\, e$. Have that $S \notin \{\to, \star\}$, so $\neg$*hastype*$(\lambda x.\, e, S)$.

Case: $T = \text{ref}\, T'$: By Lemma B.12, $v = a$. Have that $S \notin \{\text{ref}, \star\}$, so $\neg$*hastype*(a, S).

Case: $T = \star$: Vacuous.

□

Lemma B.17 (Preservation). *Suppose* $\emptyset;\Sigma \vdash e : T$ *and* $\Sigma \vdash \mu$*. If* $\langle e, \mu\rangle \longrightarrow \langle e', \mu'\rangle$*, then* $\emptyset;\Sigma' \vdash e' : T$ *and* $\Sigma' \vdash \mu'$ *and* $\Sigma' \sqsubseteq \Sigma$.

Proof. By induction on $\langle e, \mu\rangle \longrightarrow \langle e', \mu'\rangle$.

Case : ECheck:

$$\langle v{\Downarrow}S, \mu\rangle \longrightarrow \langle v, \mu\rangle \qquad \text{if } \mathit{hastype}(v, S)$$

By Lemma B.11, we have three cases:

Case: $\emptyset; \Sigma \vdash v : T'$ and $\lfloor T' \rfloor \preceq S$ and $\vdash T' <: T$:

Immediate by subsumption.

Case: $\emptyset; \Sigma \vdash v : T''$ and $\lfloor T' \rfloor \not\preceq S$ and $T'' \neq \star$:

Vacuous, by Lemma B.16.

Case: $\emptyset;\Sigma \vdash v{\Downarrow}S : \lceil S \rceil$ and $\emptyset;\Sigma \vdash v : \star$.

We proceed by cases on S:

Subcase : $S = \star$: Then $\lceil S \rceil = \star$, and the theorem holds.

Subcase : $S = \text{int}$: Then $\lceil S \rceil = \text{int}$. Because $\mathit{hastype}(v, \text{int})$, $v = n$. Thus by TInt, $\emptyset;\Sigma \vdash v : \text{int}$.

Subcase : $S = \rightarrow$: Then $\lceil S \rceil = \star \rightarrow \star$.

Because $\mathit{hastype}(v, \rightarrow)$, $v = \lambda x.\, e$.

By Lemma B.11, for some T_1, T_2 have $\emptyset,x{:}T_1;\Sigma \vdash e : T_2$ and $\vdash T_1 \rightarrow T_2 <: \star$.

Then have that $\vdash T_2 <: \star$ and $\vdash \star <: T_1$, and hence $T_1 = \star$.

By TSubsump, $\emptyset,x{:}\star;\Sigma \vdash e : \star$.

Therefore $\emptyset;\Sigma \vdash \lambda x.\, e : \star \rightarrow \star$.

Subcase : $S = \text{ref}$: Then $\lceil S \rceil = \text{ref}\,\star$.

Because $\mathit{hastype}(v, \rightarrow)$, $v = a$.

By Lemma B.11, for some T' have $\Sigma(a) = T'$ and $\vdash \text{ref}\, T' <: \star$.

Then have that $\vdash T' <: \star$ and $\vdash \star <: T'$, and hence $T' = \star$.

Therefore by TAddr $\emptyset;\Sigma \vdash a : \text{ref}\,\star$.

Case : ECheckFail: Vacuous.

Case : EFail: Vacuous.

Case : ERef:

$$\langle \texttt{ref}\ v, \mu\rangle \longrightarrow \langle a, \mu[a := v]\rangle \qquad \text{where } a \text{ fresh}$$

By Lemma B.11, for some T' have $\emptyset;\Sigma \vdash v : T'$ and $\vdash \mathsf{ref}\, T' <: T$.

Let $\Sigma' = \Sigma, a{:}T'$.

Then by TAddr, $\emptyset;\Sigma' \vdash a : \mathsf{ref}\, T'$.

By TSubsump, $\emptyset;\Sigma' \vdash a : T$.

Since a is fresh, $\Sigma' \sqsubseteq \Sigma$.

By Lemma B.13, $\emptyset;\Sigma' \vdash v : T'$.

By Lemma B.14, $\Sigma' \vdash \mu[x := v]$.

Case : EApp:

$$\langle ((\lambda x.\, e)\; v)^w, \mu\rangle \longrightarrow \langle e[x/v], \mu\rangle$$

We proceed by cases on w.

Subcase : $w = \Diamond$:

By Lemma B.11, for some T_1, T_2 have $\emptyset;\Sigma \vdash \lambda x.\, e : T_1 \to T_2$ and $\emptyset;\Sigma \vdash v : T_1$ and $\vdash T_2 <: T$.

By Lemma B.11, for some T_1', T_2' have $\emptyset, x{:}T_1';\Sigma \vdash e : T_2'$ and $\vdash T_1' \to T_2' <: T_1 \to T_2$.

Hence $\vdash T_1 <: T_1'$ and $\vdash T_2' <: T_2$.

By TSubsump, $\emptyset;\Sigma \vdash v : T_1'$.

By Lemma B.15, $\emptyset;\Sigma \vdash e[x/v] : T_2'$.

By TSubsump, $\emptyset;\Sigma \vdash e[x/v] : T_2$, and by TSubsump again $\emptyset;\Sigma \vdash e[x/v] : T$.

Subcase : $w = \blacklozenge$:

By Lemma B.11, have $\emptyset;\Sigma \vdash \lambda x.\, e : \star$ and $\emptyset;\Sigma \vdash v : \star$ and $\vdash \star <: T$.

Therefore $\star = T$.

By Lemma B.11, for some T_1, T_2 have $\emptyset, x{:}T_1;\Sigma \vdash e : T_2$ and $\vdash T_1 \to T_2 <: \star$.

Hence $\vdash \star <: T_1$, and therefore $T_1 = \star$, and $\vdash T_2 <: \star$.

By Lemma B.15, $\emptyset;\Sigma \vdash e[x/v] : T_2$.

By TSubsump, $\emptyset;\Sigma \vdash e[x/v] : \star$.

Case : ELet:

$$\langle \mathtt{let}\; x = v\; \mathtt{in}\; e, \mu\rangle \longrightarrow \langle e[x/v], \mu\rangle$$

By Lemma B.11, for some T_1, T_2 have $\emptyset;\Sigma \vdash v : T_1$ and $\emptyset,x{:}T_1;\Sigma \vdash e : T_2$ and $\vdash T_2 <: T$.

By Lemma B.15, $\emptyset;\Sigma \vdash e[x/v] : T_2$.

By TSubsump, $\emptyset;\Sigma \vdash e[x/v] : T$.

Case : EDeref:

$$\langle !a^w, \mu\rangle \longrightarrow \langle \mu(a), \mu\rangle$$

We proceed by cases on w.

Subcase : $w = \Diamond$:

By Lemma B.11, for some T' have $\emptyset;\Sigma \vdash a : \text{ref } T'$ and $\vdash T' <: T$.

By Lemma B.11, $\Sigma(a) = T''$ and $\vdash \text{ref } T'' <: \text{ref } T'$.

Hence $\vdash T' <: T''$ and $\vdash T'' <: T'$.

Since $\Sigma \vdash \mu$, $\emptyset;\Sigma \vdash \mu(a) : T''$. By TSubsump, $\emptyset;\Sigma \vdash \mu(a) : T'$, and by TSubsump again $\emptyset;\Sigma \vdash \mu(a) : T$.

Subcase : $w = \blacklozenge$:

By Lemma B.11, have $\emptyset;\Sigma \vdash a : \star$ and $\vdash \star <: T$.

Therefore $\star = T$.

By Lemma B.11, $\Sigma(a) = T'$ and $\vdash \text{ref } T' <: \star$.

Hence $\vdash T' <: \star$ and $\vdash \star <: T'$, so $T' = \star$.

Since $\Sigma \vdash \mu$, $\emptyset;\Sigma \vdash \mu(a) : \star$.

Case : EUpdt:

$$\langle a{:=}^w v, \mu\rangle \longrightarrow \langle 0, \mu[a := v]\rangle$$

By TInt, $\emptyset;\Sigma \vdash 0 : \text{int}$.

We continue by cases on w.

Subcase : $w = \Diamond$:

By Lemma B.11, for some T' have $\emptyset;\Sigma \vdash a : \text{ref } T'$ and $\emptyset;\Sigma \vdash v : T'$ and $\vdash \text{int} <: T$.

By TSubsump, $\emptyset;\Sigma \vdash 0 : T$. By Lemma B.11, $\Sigma(a) = T''$ and $\vdash \text{ref } T'' <: \text{ref } T'$.

Hence $\vdash T' <: T''$ and $\vdash T'' <: T'$.

By TSubsump, $\emptyset;\Sigma \vdash v : T''$.

Therefore $\Sigma \vdash \mu[a := v]$.

Subcase : $w = \blacklozenge$:

By Lemma B.11, have $\emptyset;\Sigma \vdash a : \star$ and $\emptyset;\Sigma \vdash v : \star$ and $\vdash \text{int} <: T$.

By TSubsump, $\emptyset;\Sigma \vdash 0 : T$. By Lemma B.11, $\Sigma(a) = T'$ and $\vdash \text{ref}\, T' <: \star$.

Hence $\vdash T' <: \star$ and $\vdash \star <: T'$, so $T' = \star$.

Therefore $\Sigma \vdash \mu[a := v]$.

Case : EAdd:

$$\langle n_1 \ +^w \ n_2, \mu\rangle \longrightarrow \langle n', \mu\rangle \qquad \text{where } n_1 + n_2 = n'$$

By Lemma B.11 (applying once for each case on w), $\vdash \text{int} <: T$.

Immediately have $\emptyset;\Sigma \vdash n' : \text{int}$.

By TSubsump, $\emptyset;\Sigma \vdash n' : T$.

□

Lemma B.18 (Multi-step preservation). *Suppose* $\emptyset;\Sigma \vdash e : T$ *and* $\Sigma \vdash \mu$. *If* $\langle e, \mu\rangle \longrightarrow *\langle e', \mu'\rangle$, *then* $\emptyset;\Sigma' \vdash e' : T$ *and* $\Sigma' \vdash \mu'$ *and* $\Sigma' \sqsubseteq \Sigma$.

Proof. By induction on $\langle e, \mu\rangle \longrightarrow *\langle e', \mu'\rangle$.

Cases: where $\langle e, \mu\rangle \longrightarrow^* \texttt{fail}$ are vacuous.

Case: $\langle e, \mu\rangle \longrightarrow^* \langle e, \mu\rangle$: Immediate.

Case:

$$\frac{\langle e, \mu\rangle \longrightarrow \langle e', \mu'\rangle \qquad \langle e', \mu'\rangle \longrightarrow^* \langle e'', \mu''\rangle}{\langle e, \mu\rangle \longrightarrow^* \langle e'', \mu''\rangle}$$

By Lemma B.17, $\emptyset;\Sigma' \vdash e' : T$ and $\Sigma' \vdash \mu'$ and $\Sigma' \sqsubseteq \Sigma$.

By the IH, $\emptyset;\Sigma'' \vdash e'' : T$ and $\Sigma'' \vdash \mu''$ and $\Sigma'' \sqsubseteq \Sigma'$.

By transitivity of equality on types, $\Sigma'' \sqsubseteq \Sigma$.

□

Lemma B.19 (Progress). *Suppose* $\emptyset;\Sigma \vdash e : T$ *and* $\Sigma \vdash \mu$. *Then either*

- $\langle e, \mu\rangle \longrightarrow \langle e', \mu'\rangle$,
- $\langle e, \mu\rangle \longrightarrow \texttt{fail}$,

- e *is a value, or*
- $\langle e, \mu \rangle$ *stuck* ⧫.

Proof. By induction on $\emptyset;\Sigma \vdash e : T$.

For each case, if there exists some E, e' with e' not a value such that $e = E[e']$, then by the IH, e' is either a value, it steps to `fail` or another expression, or it is an error blaming ⧫. If it steps to an expression, then e steps to an expression by EStep. If it steps to `fail`, then e steps to `fail` by EFail. If $\langle e', \mu \rangle$ stuck ⧫, then $\langle e, \mu \rangle$ stuck ⧫.

We now proceed to each case, assuming that no such E, e' exists.

Cases : TAbs, TAddr, and TInt: Immediately have $e = v$.

Case : TVar: Vacuous.

Case : TSubsump: with $T = T_2$,

$$\frac{\emptyset;\Sigma \vdash e : T_1 \qquad \vdash T_1 <: T_2}{\emptyset;\Sigma \vdash e : T_2}$$

Immediate from the IH.

Case : TCheck: with $T = \lceil S \rceil$,

$$\frac{\emptyset;\Sigma \vdash e' : \star}{\emptyset;\Sigma \vdash e' \Downarrow S : \lceil S \rceil}$$

Assume $e' = v$. Then either *hastype*(v, S) or ¬*hastype*(v, S).

In the former case, by ECheck, $\langle v \Downarrow S, \mu \rangle \longrightarrow \langle v, \mu \rangle$.

Otherwise, by ECheckFail, $\langle v \Downarrow S, \mu \rangle \longrightarrow$ `fail`.

Case : TRedundantCheck:

$$\frac{\Gamma \vdash e' : T \qquad \lfloor T \rfloor \preceq S}{\Gamma \vdash e' \Downarrow S : T}$$

Assume $e' = v$. Then either *hastype*(v, S) or ¬*hastype*(v, S).

In the former case, by ECheck, $\langle v \Downarrow S, \mu \rangle \longrightarrow \langle v, \mu \rangle$.

Otherwise, by ECheckFail, $\langle v \Downarrow S, \mu \rangle \longrightarrow$ `fail`.

Case : TFailCheck:

$$\frac{\Gamma \vdash e' : T \qquad T \neq \star \qquad \lfloor T \rfloor \not\preceq S}{\Gamma \vdash e' \Downarrow S : T'}$$

Assume $e' = v$. Then either *hastype*(v, S) or $\neg$*hastype*(v, S).

In the former case, by ECheck, $\langle v \Downarrow S, \mu \rangle \longrightarrow \langle v, \mu \rangle$.

Otherwise, by ECheckFail, $\langle v \Downarrow S, \mu \rangle \longrightarrow \texttt{fail}$.

Case : TFail:

$$\frac{}{\Gamma \vdash \texttt{fail} : T}$$

Immediately by EFail, $\langle \texttt{fail}, \mu \rangle \longrightarrow \texttt{fail}$.

Case : TRef: with $T = \text{ref}\, T'$,

$$\frac{\Gamma;\Sigma \vdash e' : T'}{\Gamma;\Sigma \vdash \texttt{ref}\ e' : \text{ref}\, T'}$$

Assume $e' = v$.

By ERef, with a fresh, $\langle \texttt{ref}\ v, \mu \rangle \longrightarrow \langle a, \mu[a := v] \rangle$.

Case : TApp: with $T = T_2$,

$$\frac{\emptyset;\Sigma \vdash e_1 : T_1 \to T_2 \qquad \emptyset;\Sigma \vdash e_2 : T_1}{\emptyset;\Sigma \vdash (e_1\ e_2)^{\Diamond} : T_2}$$

Assume $e_1 = v_1$ and $e_2 = v_2$.

By Lemma B.12, $v_1 = \lambda x.\ e_1'$.

By EApp, $\langle ((\lambda x.\ e_1')\ v_2)^{\Diamond}, \mu \rangle \longrightarrow \langle e_1'[x/v_2], \mu \rangle$.

Case : TAppOW: with $T = \star$,

$$\frac{\emptyset;\Sigma \vdash e_1 : \star \qquad \emptyset;\Sigma \vdash e_2 : \star}{\emptyset;\Sigma \vdash (e_1\ e_2)^{\blacklozenge} : \star}$$

Assume $e_1 = v_1$ and $e_2 = v_2$.

Have that $v_1 \in \{a, n, \lambda x.\ e_1'\}$.

If $v_1 = a$ or $v_1 = n$, $\langle (v_1\ v_2)^{\blacklozenge}, \mu \rangle$ stuck $\blacklozenge$.

Otherwise, $v_1 = \lambda x.\ e_1'$ and $\langle ((\lambda x.\ e_1')\ v_2)^{\blacklozenge}, \mu \rangle \longrightarrow \langle e_1'[x/v_2], \mu \rangle$.

Case : TLet: with $T = T_2$,

$$\frac{\Gamma;\Sigma \vdash e_1 : T_1 \qquad \Gamma, x{:}T_1;\Sigma \vdash e_2 : T_2}{\Gamma;\Sigma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : T_2}$$

Assume $e_1 = v$.

By ELet, $\langle \texttt{let}\ x = v\ \texttt{in}\ e_2, \mu\rangle \longrightarrow \langle e_2[x/v], \mu\rangle$.

Case : TAdd: with $T = \text{int}$,

$$\frac{\Gamma;\Sigma \vdash e_1 : \text{int} \qquad \Gamma;\Sigma \vdash e_2 : \text{int}}{\Gamma;\Sigma \vdash e_1\ +^{\Diamond}\ e_2 : \text{int}}$$

Assume $e_1 = v_1$ and $e_2 = v_2$.

By Lemma B.12, $v_1 = n_1$ and $v_2 = n_2$.

By EAdd, with $n' = n_1 + n_2$, $\langle n_1\ +^{\Diamond}\ n_2, \mu\rangle \longrightarrow \langle n', \mu\rangle$.

Case : TAddOW: with $T = \text{int}$,

$$\frac{\Gamma;\Sigma \vdash e_1 : \star \qquad \Gamma;\Sigma \vdash e_2 : \star}{\Gamma;\Sigma \vdash e_1\ +^{\blacklozenge}\ e_2 : \text{int}}$$

Assume $e_1 = v_1$ and $e_2 = v_2$.

Have that $v_1 \in \{a, n_1, \lambda x.\, e_1'\}$.

If $v_1 = a$ or $v_1 = \lambda x.\, e$, $\langle e_1\ +^{\blacklozenge}\ e_2, \mu\rangle$ stuck $\blacklozenge$.

Otherwise $v_1 = n_1$.

By the same reasoning, either $\langle e_1\ +^{\blacklozenge}\ e_2, \mu\rangle$ stuck $\blacklozenge$ or $v_2 = n_2$.

By EAdd, with $n' = n_1 + n_2$, $\langle n_1\ +^{\Diamond}\ n_2, \mu\rangle \longrightarrow \langle n', \mu\rangle$.

Case : TDeref:

$$\frac{\Gamma;\Sigma \vdash e' : \text{ref}\ T}{\Gamma;\Sigma \vdash !e'^{\Diamond} : T}$$

Assume $e' = v$.

By Lemma B.12, $v = a$ and $\mu(a) = v'$.

By EDeref, $\langle !a^{\Diamond}, \mu\rangle \longrightarrow \langle \mu(a), \mu\rangle$.

Case : TDerefOW: with $T = \star$,

$$\frac{\Gamma;\Sigma \vdash e' : \star}{\Gamma;\Sigma \vdash !e'^{\blacklozenge} : \star}$$

Assume $e' = v$.

Have that $v \in \{a, n, \lambda x.\, e_1'\}$.

If $v = n$ or $v = \lambda x.\, e$, $\langle !e'^{\blacklozenge}, \mu\rangle$ stuck $\blacklozenge$.

Otherwise $v = a$.

If $a \notin \mathit{dom}(\mu)$, then $\langle !a^{\blacklozenge}, \mu\rangle$ stuck $\blacklozenge$.

Otherwise, by EDeref, $\langle !a^{\Diamond}, \mu\rangle \longrightarrow \langle \mu(a), \mu\rangle$.

Case : TUpdt: with $T = \text{int}$,

$$\frac{\Gamma;\Sigma \vdash e_1 : \text{ref}\, T \qquad \Gamma;\Sigma \vdash e_2 : T}{\Gamma;\Sigma \vdash e_1 :=^{\Diamond} e_2 : \text{int}}$$

Assume $e_1 = v_1$ and $e_2 = v_2$.

By Lemma B.12, $v_1 = a$.

By EUpdt, $\langle a :=^{\Diamond} v_2, \mu\rangle \longrightarrow \langle 0, \mu[a := v_2]\rangle$.

Case : TDerefOW: with $T = \text{int}$,

$$\frac{\Gamma;\Sigma \vdash e_1 : \star \qquad \Gamma;\Sigma \vdash e_2 : \star}{\Gamma;\Sigma \vdash e_1 :=^{\blacklozenge} e_2 : \text{int}}$$

Assume $e_1 = v_1$ and $e_2 = v_2$.

Have that $v_1 \in \{a, n, \lambda x.\, e_1'\}$.

If $v_1 = n$ or $v_1 = \lambda x.\, e$, $\langle v_1 :=^{\blacklozenge} v_2, \mu\rangle$ stuck $\blacklozenge$.

Otherwise $v_1 = a$.

If $a \notin \mathit{dom}(\mu)$, then $\langle a :=^{\blacklozenge} v_2, \mu\rangle$ stuck $\blacklozenge$.

Otherwise, by EDeref, $\langle a :=^{\blacklozenge} v_2, \mu\rangle \longrightarrow \langle 0, \mu[a := v_2]\rangle$.

□

Corollary B.1. *If $\emptyset \vdash d : A; \Omega$ and σ is a solution for Ω, then:*

- $\emptyset \vdash \sigma d \rightsquigarrow e : T$*, and*

- $\emptyset; \emptyset \vdash e : T$, *and*
- *if* $\langle e, \emptyset \rangle \longrightarrow^* \langle v, \mu \rangle$, *then* $\Sigma \vdash \mu$ *and* $\emptyset; \Sigma \vdash v : T$ *and* $\vdash T <: \sigma A$.

Proof. By Lemma B.9, $\emptyset \vdash \sigma d : T$ and $\vdash T <: \sigma A$.

By Lemma B.10, $\emptyset \vdash \sigma d \rightsquigarrow e : T$.

If for all ς such that $\langle e, \emptyset \rangle \longrightarrow \varsigma$, there exists some ς' such that $\varsigma \longrightarrow \varsigma'$, then the theorem is satisfied.

Otherwise, there exists some ς such that $\varsigma \not\longrightarrow \varsigma'$.

If $\varsigma = \texttt{fail}$, then the theorem is satisfied.

Otherwise, $\varsigma = \langle e', \mu \rangle$.

By repeating Lemma B.17, $\emptyset; \Sigma \vdash e' : T$ and $\Sigma \vdash \mu$.

By Lemma B.19, either $e' = v$ or $\langle e, \mu \rangle$ stuck ⧫. □

2.4. Analysis correctness.

Lemma B.20. *If* $\Gamma \vdash d : A; \Omega$ *and* σ *is a solution for* Ω, *then* $\sigma\Gamma; \emptyset \vdash |d| : \sigma A$.

Proof. By induction on $\Gamma \vdash d : A; \Omega$.

Case: IVar:

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A; \emptyset}$$

Have that $\sigma\Gamma(x) = \sigma A$.

By TVar, $\sigma\Gamma; \emptyset \vdash x : \sigma A$.

Case: IAbs:

$$\frac{\Gamma, x{:}\alpha \vdash d : A; \Omega}{\Gamma \vdash \lambda(x{:}\alpha){\to}\beta.\ d : \alpha \to \beta; \Omega, A <: \beta}$$

Since σ is a solution for $\Omega \cup \{A <: \beta\}$, by Lemma B.5, σ is a solution for Ω.

By the IH, $\sigma\Gamma, x{:}\sigma\alpha; \emptyset \vdash |d| : \sigma A$.

Since σ is a solution to $\Omega \cup \{A <: Y\}$, $\vdash \sigma A <: \sigma\beta$.

By TSubsump, $\sigma\Gamma, x{:}\sigma\alpha; \emptyset \vdash |d| : \sigma\beta$.

By TAbs, $\sigma\Gamma; \emptyset \vdash \lambda x.\ |d| : \sigma\alpha \to \sigma\beta$.

Case: IApp:

$$\frac{\Gamma \vdash d_1 : V_1 \to V_2; \Omega_1 \qquad \Gamma \vdash d_2 : A; \Omega_2}{\Gamma \vdash d_1\ d_2 : V_2; \Omega_1, \Omega_2, A <: V_1}$$

Since σ is a solution for $\Omega_1 \cup \Omega_2 \cup \{A <: V_1\}$, by Lemma B.5, σ is a solution for Ω_1 and σ is a solution for Ω_2 and σ is a solution for $A <: V_1$.

By the IH, $\sigma\Gamma; \emptyset \vdash |d_1| : \sigma V_1 \to \sigma V_2$.

By the IH, $\sigma\Gamma; \emptyset \vdash |d_2| : \sigma A$.

Since σ is a solution to $\{A <: V_1\}, \vdash \sigma A <: \sigma V_1$.

By TSubsump, $\sigma\Gamma; \emptyset \vdash |d_2| : \sigma V_1$.

By TApp, $\sigma\Gamma; \emptyset \vdash (|d_1|\ |d_2|)^\Diamond : \sigma V_2$.

Case: ICheck:

$$\frac{\Gamma \vdash d : A_1; \Omega \qquad A_1 \rhd_S A_2}{\Gamma \vdash d \Downarrow S : A_2; \Omega}$$

By the IH, $\sigma\Gamma; \emptyset \vdash |d| : \sigma A_1$.

If $S = \star$ then $A_2 = A_1$, and by TRedundantCheck, $\sigma\Gamma; \emptyset \vdash |d| \Downarrow S : \sigma A_2$.

Otherwise, we proceed by cases on A_1 and σA_1.

Case: $A_1 \neq V$ and S is the constructor of σA_1:

Then S is the constructor of A_1.

Then $A_2 = A_1$.

By TRedundantCheck, $\sigma\Gamma; \emptyset \vdash |d| \Downarrow S : \sigma A_2$.

Case: $A_1 \neq V$ and S is not the constructor of σA_1: Vacuous, since $S \neq \star$.

Case: $A_1 = V$ and S is the constructor of σA_1:

Then $\sigma A_1 = \sigma A_2$.

By TRedundantCheck, $\sigma\Gamma; \emptyset \vdash |d| \Downarrow S : \sigma A_2$.

Case: $A_1 = V$ and S is not the constructor of σA_1:

Since σ is a solution to Ω, for all $\alpha \in \mathit{parts}(A_2)$ of A_2, $\sigma\alpha = \star$.

Therefore $\lceil S \rceil = \sigma A_2$.

Cases on $\vdash \sigma A_1 <: \star$:

Subcase: $\vdash \sigma A_1 <: \star$:

By TSubsump, $\sigma\Gamma; \emptyset \vdash |d| : \star$.

By TCheck, $\sigma\Gamma; \emptyset \vdash |d|\Downarrow S : \lceil S \rceil$.

Subcase: $\vdash \sigma A_1 \not<: \star$:

Since S is not the constructor of σA_1 and $S \neq \star$, $\lfloor \sigma A_1 \rfloor \not\preceq S$.

By TFailCheck, $\sigma\Gamma; \emptyset \vdash |d|\Downarrow S : \lceil S \rceil$.

Case: IRef:

$$\frac{\Gamma \vdash d : A; \Omega}{\Gamma \vdash \texttt{ref}_X\ d : \text{ref}\, X; \Omega, A <: X}$$

By the IH, $\sigma\Gamma; \emptyset \vdash |d| : \sigma A$.

Since σ is a solution to $\Omega, A <: X, \vdash \sigma A <: \sigma X$.

By TSubsump, $\sigma\Gamma; \emptyset \vdash |d| : \sigma X$.

By TRef, $\sigma\Gamma; \emptyset \vdash \texttt{ref}\ |d| : \text{ref}\, \sigma X$.

Case: IDeref:

$$\frac{\Gamma \vdash d : \text{ref}\, V; \Omega}{\Gamma \vdash !d : V; \Omega}$$

By the IH, $\sigma\Gamma; \emptyset \vdash |d| : \text{ref}\, \sigma V$.

By TDeref, $\sigma\Gamma; \emptyset \vdash !|d|^{\Diamond} : \sigma V$.

Case: IUpdt:

$$\frac{\Gamma \vdash d_1 : \text{ref}\, V; \Omega_1 \qquad \Gamma \vdash d_2 : A; \Omega_2}{\Gamma \vdash d_1 := d_2 : \text{int}; \Omega_1, \Omega_2, A <: V}$$

Since σ is a solution to $\Omega_1 \cup \Omega_2 \cup \{A <: V\}$, by Lemma B.5, σ is a solution to Ω_1 and σ is a solution to Ω_2 and σ is a solution to $\{A <: V\}$.

By the IH, $\sigma\Gamma; \emptyset \vdash |d_1| : \text{ref}\, \sigma V$.

By the IH, $\sigma\Gamma; \emptyset \vdash |d_2| : \sigma A$.

Since σ is a solution to $\{A <: V\}, \vdash \sigma A <: \sigma V$.

By TSubsump, $\sigma\Gamma; \emptyset \vdash |d_2| : \sigma V$.

By TUpdt, $\sigma\Gamma; \emptyset \vdash |d_1| :=^{\Diamond} |d_2| : \text{int}$.

Case: IAdd:

$$\frac{\Gamma \vdash d_1 : \text{int}; \Omega_1 \qquad \Gamma \vdash d_2 : \text{int}; \Omega_2}{\Gamma \vdash d_1 \ + \ d_2 : \text{int}; \Omega_1, \Omega_2}$$

Since σ is a solution to $\Omega_1 \cup \Omega_2$, by Lemma B.5, σ is a solution to Ω_1 and σ is a solution to Ω_2.

By the IH, $\sigma\Gamma; \emptyset \vdash |d_1| : \text{int}$.

By the IH, $\sigma\Gamma; \emptyset \vdash |d_2| : \text{int}$.

Therefore by TAdd, $\sigma\Gamma; \emptyset \vdash |d_1| \ + \ |d_2| : \text{int}$.

Case : IInt: Immediate by TInt.

□

Lemma B.21. *If* $\Gamma; \Sigma \vdash e : T$ *and for all* $x \in$ *dom*(Γ), $\Gamma'(x) = \Gamma(x)$, *then* $\Gamma'; \Sigma \vdash e : T$.

Proof. Induction on $\Gamma; \Sigma \vdash e : T$. The only interesting cases are variables, where the correspondence between Γ and Γ' ensure that the result is the same, and functions, where we can immediately show that for all $x \in$ *dom*$(\Gamma, y : T)$, $(\Gamma', y : T)(x) = (\Gamma, y : T)(x)$. □

Lemma B.22. *If* $\Gamma; \Sigma \vdash e : T$ *and* $\Sigma \vdash \rho : \Gamma$, *then* $\emptyset; \Sigma \vdash \rho(e) : T$.

Proof. By induction on $\Sigma \vdash \rho : \Gamma$.

Case :

$$\frac{}{\Sigma \vdash \cdot : \emptyset}$$

Have that $\rho(e) = e$.

Immediately $\emptyset; \Sigma \vdash e : T$.

Case :

$$\frac{\emptyset; \Sigma \vdash v : T' \qquad \Sigma \vdash \rho : \Gamma}{\Sigma \vdash \rho, x = v : \Gamma, x : T'}$$

Have $\Gamma, x : T; \Sigma \vdash e : T$ and $\emptyset, \Sigma \vdash v : T'$.

By Lemma B.21, $\Gamma; \Sigma \vdash v : T'$.

By Lemma B.15, $\Gamma; \Sigma \vdash e[x/v] : T$.

By the IH, $\emptyset; \Sigma \vdash \rho(e[x/v]) : T$.

Have that $(\rho, x = v)(e) = \rho(e[x/v])$.

□

Lemma B.23. *If* $\Gamma; \Sigma \vdash v : T$ *and* $\Sigma \vdash \mu$*, then hastype*$(v, \lfloor T \rfloor)$.

Proof. By cases on T.

Case: $T = \text{int}$: By Lemma B.12, $v = n$. Therefore *hastype*(n, int).

Case: $T = T_1 \to T_2$: By Lemma B.12, $v = \lambda x.\ e$. Therefore *hastype*$(\lambda x.\ e, \to)$.

Case: $T = \text{ref}\ T'$: By Lemma B.12, $v = a$. Therefore *hastype*(a, ref).

Case: $T = \star$: Immediate.

□

Theorem 6.3. *Suppose* $\Gamma \vdash d : A; \Omega$ *and* σ *is a solution to* Ω *and* $\Sigma \vdash \rho : \sigma\Gamma$ *and* $\Sigma \vdash \mu$ *and* $\langle \rho(|d|), \mu \rangle \longrightarrow^* \langle v, \mu' \rangle$*. If* $\lfloor \sigma A \rfloor \preceq S$*, then* $\langle v \Downarrow S, \mu' \rangle \not\longrightarrow \texttt{fail}$.

Proof. By Lemma B.20, $\sigma\Gamma; \emptyset \vdash |d| : \sigma A$.

By Lemma B.13, $\sigma\Gamma; \Sigma \vdash |d| : \sigma$.

By Lemma B.22, $\emptyset; \Sigma \vdash \rho(|d|) : \sigma A$.

By Lemma B.18, $\emptyset; \Sigma' \vdash v : \sigma A$ and $\Sigma' \vdash \mu'$.

By Lemma B.23, *hastype*$(v, \lfloor \sigma A \rfloor)$.

Since $\lfloor \sigma A \rfloor \preceq S$, either $S = \star$ or $S = \lfloor \sigma A \rfloor$. In either case, *hastype*$(v, \lfloor \sigma A \rfloor)$.

Therefore ECheckFail does not apply, and no other step can be taken from $\langle v \Downarrow S, \mu' \rangle$ to `fail`. □

2.5. Soundness of constraint solving. Figure 1 restates the definition for constraint set simplification from Figure 10.

Lemma B.24. *If* $T_1 = T_2$ *then* $\vdash T_1 <: T_2$ *and* $\vdash T_2 <: T_1$.

Proof. Straightforward induction on T_1. □

Lemma B.25. *If* σ *is a solution to* Ω_1 *and* σ *is a solution to* Ω_2*, then* σ *is a solution to* $\Omega_1 \cup \Omega_2$.

$\boxed{\Omega \longrightarrow \Omega}$

(1) $\Omega \cup \{V_1 \to V_2 <: \star\} \longrightarrow \Omega \cup \{\star <: V_1,\ V_2 <: \star\}$

(2) $\Omega \cup \{V_1 \to V_2 <: V_3 \to V_4\} \longrightarrow \Omega \cup \{V_3 <: V_1,\ V_2 <: V_4\}$

(3) $\Omega \cup \{\mathsf{ref}\, V <: \star\} \longrightarrow \Omega \cup \{V = \star\}$

(4) $\Omega \cup \{\mathsf{ref}\, V_1 <: \mathsf{ref}\, V_2\} \longrightarrow \Omega \cup \{V_1 = V_2\}$

(5) $\Omega \cup \{V <: V\} \longrightarrow \Omega$

(6) $\Omega \cup \{\mathsf{int} <: \star\} \longrightarrow \Omega$

(7) $\Omega \cup \{V_1 \to V_2 = V_3 \to V_4\} \longrightarrow \Omega \cup \{V_1 = V_3, V_2 = V_4\}$

(8) $\Omega \cup \{\mathsf{ref}\, V_1 = \mathsf{ref}\, V_2\} \longrightarrow \Omega \cup \{V_1 = V_2\}$

(9) $\Omega \cup \{A = A\} \longrightarrow \Omega$

(10) $\Omega \cup \{A = \alpha\} \longrightarrow \Omega \cup \{\alpha = A\}$

where $A \neq \alpha'$

(11) $\Omega \cup \{\alpha = A\} \longrightarrow \Omega[\alpha/A] \cup \{\alpha \triangleq A\}$

where $\alpha \notin \mathit{vars}(A)$
$(\alpha \triangleq B) \notin \Omega$

(12) $\Omega \cup \{(A{:}S) = A\} \longrightarrow \Omega$

(13) $\Omega \cup \{\alpha : S, (\alpha{:}S) = A\} \longrightarrow \Omega \cup \{\alpha = A\}$

where $((\alpha{:}S') = A') \notin \Omega$
$A \neq \alpha$

(14) $\Omega \cup \{\alpha : S_1, (\alpha{:}S_2) = A\} \longrightarrow \Omega \cup \{\alpha : S_1\} \cup \{\alpha' = \star \mid \forall \alpha' \in \mathit{parts}(A)\}$

where $S_1 \neq S_2$

(15) $\Omega \cup \{(\alpha{:}S) = A_1, (\alpha{:}S) = A_2\} \longrightarrow \Omega \cup \{(\alpha{:}S) = A_1, A_2 = A_1\}$

(16) $\Omega \cup \{\alpha : S\} \longrightarrow \Omega \cup \{\alpha = A\}$

where $((\alpha{:}S') = A') \notin \Omega$
$(\alpha \triangleq T) \notin \Omega$
$(\alpha = A') \notin \Omega$
$\alpha \rhd_S A$

Figure 1. Simplification of constraint sets (restated from Figure 10).

Proof. Since σ is a solution for every constraint in Ω_1 and Ω_2, and for all $C \in \Omega_1 \cup \Omega_2$, $C \in \Omega_1$ or $C \in \Omega_2$, so σ is a solution for every constraint in $\Omega_1 \cup \Omega_2$ so it is a solution to $\Omega_1 \cup \Omega_2$.

□

Lemma B.26. *If $\Omega \longrightarrow \Omega'$ and σ is a solution to Ω', then σ is a solution to Ω.*

Proof. By cases on $\Omega \longrightarrow \Omega'$. Many cases are immediate using Lemmas B.5 and B.25.

Case: 1 $\Omega \cup \{V_1 \to V_2 <: \star\} \longrightarrow \Omega \cup \{\star <: V_1,\ V_2 <: \star\}$:

Immediate from subtyping definitions.

Case: 2 $\Omega \cup \{V_1 \to V_2 <: V_3 \to V_4\} \longrightarrow \Omega \cup \{V_3 <: V_1,\ V_2 <: V_4\}$:

Immediate from subtyping definitions.

Case: 3 $\Omega \cup \{\text{ref}\, V <: \star\} \longrightarrow \Omega \cup \{V = \star\}$:

Immediate from subtyping definitions and from Lemma B.24.

Case: 4 $\Omega \cup \{\text{ref}\, V_1 <: \text{ref}\, V_2\} \longrightarrow \Omega \cup \{V_1 = V_2\}$:

Immediate from subtyping definitions and from Lemma B.24.

Case: 5 $\Omega \cup \{V <: V\} \longrightarrow \Omega$:

Immediate since subtyping is reflexive.

Case: 6 $\Omega \cup \{\text{int} <: \star\} \longrightarrow \Omega$:

Immediate from subtyping definitions.

Case: 7 $\Omega \cup \{V_1 \to V_2 = V_3 \to V_4\} \longrightarrow \Omega \cup \{V_1 = V_3, V_2 = V_4\}$:

Immediate.

Case: 8 $\Omega \cup \{\text{ref}\, V_1 = \text{ref}\, V_2\} \longrightarrow \Omega \cup \{V_1 = V_2\}$:

Immediate.

Case: 9 $\Omega \cup \{A = A\} \longrightarrow \Omega$:

Immediate.

Case: 10 $\Omega \cup \{A = \alpha\} \longrightarrow \Omega \cup \{\alpha = A\}$:

Immediate.

Case: 11 $\Omega \cup \{\alpha = A\} \longrightarrow \Omega[\alpha/A] \cup \{\alpha \triangleq A\}$:

Since σ is a solution to $\{\alpha \triangleq A\}$, have that $\sigma\alpha = \sigma A$.

Therefore for any type A' with α as a component, $\sigma A'[\alpha/A] = \sigma A'$.

Therefore σ is a solution to Ω, and thus a solution to $\Omega \cup \{\alpha = A\}$.

Case: 12 $\Omega \cup \{(A{:}S) = A\} \longrightarrow \Omega$:

Immediate.

Case: 13 $\Omega \cup \{\alpha : S, (\alpha{:}S) = A\} \longrightarrow \Omega \cup \{\alpha = A\}$:

Have that $\sigma\alpha = \sigma A$ and $\lfloor \sigma\alpha \rfloor = S$.

Therefore $(\alpha{:}S) = A$ is satisfied.

Rest is immediate.

Case: 14 $\Omega \cup \{\alpha : S_1, (\alpha{:}S_2) = A\} \longrightarrow \Omega \cup \{\alpha : S_1\} \cup \{\alpha' = \star \mid \forall \alpha' \in \mathit{parts}(A)\}$:

Have that $S_1 \neq S_2$.

Have that $\lfloor \sigma\alpha \rfloor = S_1$.

Therefore $\lfloor \sigma\alpha \rfloor \neq S_2$.

Have that $\sigma\alpha' = \star$ for all $\alpha' \in \mathit{parts}(A)$.

Therefore $(\alpha{:}S_1) = A$ is satisfied.

Rest is immediate.

Case: 15 $\Omega \cup \{(\alpha{:}S) = A_1, (\alpha{:}S) = A_2\} \longrightarrow \Omega \cup \{(\alpha{:}S) = A_1, A_1 = A_2\}$:

First, suppose that $\sigma\alpha = \sigma A_1$ and $\lfloor \sigma\alpha \rfloor = S$.

Since $\sigma A_1 = \sigma A_2$, $(\alpha{:}S) = A_2$ is satisfied.

Now suppose that $\lfloor \sigma\alpha \rfloor \neq S$.

Then for all parts α_1 of A_1, $\sigma\alpha_1 = \star$.

Since $\sigma\alpha_1 = \sigma\alpha_2$, for all parts α_2 of A_2, $\sigma\alpha_2 = \star$.

Therefore $(\alpha{:}S) = A_2$ is satisfied.

Rest is immediate.

Case: 16 $\Omega \cup \{\alpha : S\} \longrightarrow \Omega \cup \{\alpha = A\}$:

Since $\sigma\alpha = \sigma A$ and $\lfloor A \rfloor = S$, $\lfloor \sigma\alpha \rfloor = S$. Rest is immediate.

□

Lemma B.27. *If $\Omega = \{\alpha_1 \triangleq T_1, \ldots, \alpha_n \triangleq T_n\}$, and for all $i, j \leq n$, $\alpha_i \notin T_j$, then $\sigma = \alpha_1 \mapsto T_1, \ldots, \alpha_n \mapsto T_n$ is a solution to Ω.*

Proof. Since $\alpha_i \notin \mathit{vars}(T_j)$ for any i, j, $\sigma\Omega = \{T_1 \triangleq T_1, \ldots, T_n \triangleq T_n\}$. Therefore σ is a solution to Ω. □

Theorem 6.2. *If $\Omega \Downarrow \sigma$, then σ is a solution to Ω.*

Proof. By induction on $\Omega \Downarrow \sigma$.

Base case.: By Lemma B.27, σ is a solution to Ω.

Simplification case.: By the IH, σ is a solution to Ω'', σ is a solution to Ω'.

By repeating Lemma B.26, σ is a solution to Ω.

Solving case.: By the IH, σ is a solution to $\Omega \cup \{\alpha : S\}$.

By Lemma B.5, σ is a solution to Ω.

□

Michael M. Vitousek

Contact Information

mmvitousek@gmail.com
mvitousek@fb.com
http://homes.soic.indiana.edu/mvitouse
http://github.com/mvitousek
http://linkedin.com/michael-vitousek-7b88b82b

Research Interests

I am interested in designing programming languages and techniques that support users throughout the development cycle, from prototyping to production, and that reduce barriers to entry while guaranteeing properties programmers rely on. My research so far has focused on soundly combining static and dynamic typing within the same programming language, also known as *gradual typing*, and I'm especially interested in techniques for applying gradual typing to existing programming languages practically, soundly, and efficiently.

Education

Indiana University

Ph.D., Computer Science, August 2013–May 2019

*Formerly of **University of Colorado Boulder**, August 2011–August 2013*

- Advisor: Jeremy G. Siek
- Research committee: Jeremy G. Siek, Sam Tobin-Hochstadt, Amr Sabry, Lawrence S. Moss
- Dissertation title: *Gradual Typing for Python, Unguarded*

Willamette University

B.A., Computer Science and History (minor in Mathematics), *magna cum laude*, December 2010

Employment History

Facebook, Inc.

Research scientist, October 2018–present

Indiana University

Graduate research assistant, June 2016–May 2018
Associate instructor, January 2016–May 2016
Graduate research assistant, August 2013–December 2015

Mozilla

Research intern, May 2013–August 2013

Adobe Systems

Research intern, May 2012–August 2012

University of Colorado Boulder

Graduate research assistant, September 2012–May 2013
Graduate research assistant, June 2011–April 2012

Honors and Awards

2015–16	Nominated, associate instructor of the year, IU School of Informatics and Computing
2013	Honorable mention, NSF Graduate Research Fellowship
2013	Google Lime Scholarship
2012	Third place, PLDI Student Research Competition
2011–13	Chancellor's Fellowship, University of Colorado Boulder
2011–12	University Fellowship, University of Colorado Boulder
2010	Phi Beta Kappa, Willamette University
2009	Meritorious Winner, Mathematical Competition in Modeling
2008	Kenneth G. Batchelder Memorial Scholarship, Willamette University

Conference Publications

M.M. Vitousek, C. Swords, J.G. Siek. *Big types in little runtime: open world soundness and collaborative blame for gradual type systems*. POPL '17: Symposium on Principles of Programming Languages (January 2017).

J.G. Siek, M.M. Vitousek, M. Cimini, J.T. Boyland. *Refined criteria for gradual typing*. SNAPL '15: Summit on Advances in Programming Languages (May 2015).

J.G. Siek, M.M. Vitousek, M. Cimini, S. Tobin-Hochstadt, R. Garcia. *Monotonic references for efficient gradual typing*. ESOP '15: European Symposium on Programming (April 2015).

M.M. Vitousek, A.M. Kent, J.G. Siek, J. Baker. *Design and evaluation of gradual typing for Python*. DLS '14: Symposium on Dynamic Languages (October 2014).

Workshop Appearances

M.M. Vitousek, J.G. Siek. *From optional to gradual typing via transient checks*. STOP '16: Script To Program Evolution Workshop (July 2016).

J.G. Siek, M.M. Vitousek, J.D. Turner. *Effects for funargs*. HOPE '12: Workshop on Higher-Order Programming with Effects (September 2012).

M.M. Vitousek, S. Bharadwaj, J.G. Siek. *Towards gradual typing in Jython*. STOP '12: Script To Program Evolution Workshop (June 2012).

Technical Reports and Work in Progress

M.M. Vitousek, J.G. Siek, A. Chaudhuri. *Optimizing and evaluating transient gradual typing*. Draft (April 2018).

M.M. Vitousek, J.G. Siek. *Gradual typing in an open world*. Indiana University Technical Report TR729 (October 2016).

J.G. Siek, M.M. Vitousek, J.D. Turner. *Effects for funargs*. Draft (December 2011).

Selected Talks

Gradual typing for Python, unguarded. Final dissertation defense (September 2018).

Gradual typing for Python, unguarded. Galois (May 2018); GrammaTech (June 2018).

Optimizing and evaluating transient gradual typing with trusted type inference. Facebook, Inc. (December 2017).

Big types in little runtime: open world soundness and collaborative blame for gradual type systems. POPL '17: Symposium on Principles of Programming Languages (January 2017).

From optional to gradual typing via transient checks. STOP '16: Script To Program Evolution Workshop (July 2016).

Lightweight gradual typing with transient checks. PI Meeting, NSF Award 1518844, SHF Large: Gradual Typing Across the Spectrum (May 2016).

Design and evaluation of gradual typing for Python. DLS '14: Symposium on Dynamic Languages (October 2014).

Empirical analysis of megamorphic property accesses. Mozilla (August 2013).

Gradual typing with efficient object casts. PLDI Student Research Competition, final round (June 2012).

Towards gradual typing in Jython. STOP '12: Script To Program Evolution Workshop (June 2012).

Research Experience

Runtime techniques for gradual typing

For interaction between statically and dynamically typed code to be safe with respect to the expected static types from the static code, runtime checks need to be performed. For higher-order types like functions and mutable objects, these checks cannot be performed eagerly (when values cross from static to dynamic or vice versa) but need to ensure that when the values are used, the result is of the expected type. Historically this has been performed by installing proxies (also called wrappers) on higher-order values, but this leads to problems in performance and compatibility. I am interested in alternative designs to runtime enforcement for gradual typing. I developed and formalized the *transient* approach to runtime checks (**DLS '14**, **POPL '17**), which uses pervasive shallow checks rather than proxies. This approach is promising for transitioning from optional (unsafe) typing to safe gradual typing (**STOP '16**), and it supports blame tracking using a side-channel strategy (**POPL '17**). I identified the *open-world soundness* property, supported by the transient approach, which allows gradually typed programs to safely be embedded in dynamic, open-world contexts (**POPL '17**). In upcoming work I show that, in combination with type inference, the runtime overhead of transient can be minimized.

With colleagues I also developed the *monotonic* approach, which "locks down" mutable values at their most specific static type for compiler optimizations (**STOP '12**, **ESOP '15**).

Gradual typing for Python (and other languages)

The Python programming language is dynamically typed, but many of its users desire the option to use static types where desired. I have worked on implementing gradual typing in the Python programming language, first by modifying the Jython compiler to support gradual typing (**STOP '12**), and then by developing a source-to-source translator, Reticulated Python, which typechecks gradually typed source programs and then translates them to valid Python 3 code with safety-preserving runtime checks inserted (**DLS '14**). I used Reticulated Python as an experimental platform in developing the transient and monotonic approaches described above. I also took the same approach in developing CheckScript, a version of TypeScript that uses the transient design to ensure runtime safety (**STOP '16**). I also assisted in the development of PEP 484, the Python standard for type annotations.

Gradual typing criteria

I worked with colleagues to introduce and develop the *gradual guarantee*, a crucial formal property for gradually typed languages that ensures that programs in such languages can be gradually evolved from dynamic to static as long as there exists a corresponding statically-typed version of the program (**SNAPL '15**). I also introduced *open-world soundness*, which allows gradually typed programs to safely be embedded in dynamic, open-world contexts (**POPL '17**).

Empirical analysis of JavaScript megamorphism

(at **Mozilla**, May–August 2013, with Luke Wagner.)

I worked at Mozilla as a research intern on the JavaScript engine team. I instrumented the Firefox

JavaScript JIT, IonMonkey, to determine the overhead of fallback inline-caches at megamorphic property access sites, and I used this to determine the most important design patterns and use cases that result in this overhead (mixins and monolithic functions).

Joint dispatch for binary methods
(at **Adobe Systems**, May–August 2012, with Avik Chaudhuri.)
I worked at Adobe Systems as a research intern on the ActionScript team. I developed a new approach to the binary method problem, allowing for binary methods to be implemented in Java-like languages without requiring F-bounded polymorphism or exact This-types. I also worked with Avik Chaudhuri to design a system of bounded generics defined in terms of type intervals, and developed techniques to efficiently implement such a design.

Upwards funargs and effects
(at **University of Colorado Boulder**, June–November 2011, with Jeremy G. Siek and Jonathan D. Turner.)
I worked with colleagues to create a type-and-effect system which detects possible dangling references in stack-allocating languages with first-class functions (**HOPE '12**). This, combined with a kind of function-local storage, serves as a solution to the classic upwards funarg problem, and increases the viability of first-class functions in stack-allocating languages like Chapel.

Software Projects

Flow. Contributed to October 2018–present.
Flow is a typechecker for JavaScript code with a focus on soundness and type inference.
https://github.com/facebook/flow

Reticulated Python: Gradual Typing for Python. December 2012–September 2018.
Reticulated Python is a source-to-source translator that takes type-annotated Python code, typechecks it, and produces standard Python code with runtime enforcement code inserted.
https://github.com/mvitousek/reticulated

CheckScript. April 2016–September 2018.
CheckScript is a modification to the TypeScript programming language that adds runtime enforcement of type annotations. CheckScript is still in an experimental stage but it supports runtime enforcement of the entire TypeScript type system.
https://github.com/mvitousek/checkscript

Service

Reviewing for ESOP '16: European Symposium on Programming (April 2016).

Reviewing for OOPSLA '15: Conference on Object-Oriented Programming Systems, Languages, and Applications (October 2015).

Reviewing for POPL '14: Symposium on Principles of Programming Languages (January 2014).

Teaching Experience

Summer	2019	Intern manager, Facebook, PhD student research intern
Fall	2016	Mentor, visiting Northeastern University undergraduate researcher
Spring	2016	Associate instructor, programming language implementation
Summer	2014	REU mentor, dynamic analysis of gradually typed programs
Spring	2013	Lecturer, readings in gradual typing
Spring	2008	Teaching assistant, intro to programming
Fall	2007	Teaching assistant, intro to programming

Skills

Languages:	Python, OCaml, Racket, JavaScript, TypeScript, Java, C#, C/C++, Haskell, Prolog, Arduino, SQL
Proof assistants:	Coq, Isabelle, Agda
Typesetting:	LaTeX, Markdown

Selected Graduate Coursework

- ❑ Programming Language Theory
- ❑ Homotopy Type Theory
- ❑ Program Analysis
- ❑ Software Engineering
- ❑ Theorem-proving using Isabelle
- ❑ Compilers
- ❑ Cyberphysical Systems
- ❑ Theory of Computation
- ❑ Analysis of Algorithms
- ❑ Operating Systems

Other Experience

Summers, 2008–10	Programming Languages Summer School, **University of Oregon**
Summers, 2006–08	Biogeochemistry lab and field assistant, **Stanford University**