TOWARDS EFFICIENT GRADUAL TYPING VIA MONOTONIC REFERENCES AND COERCIONS

Deyaaeldeen Almahallawi

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Luddy School of Informatics, Computing, and Engineering
Indiana University
June 2020

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

Doctoral Committee

Jeremy Siek, Ph.D.

Sam Tobin-Hochstadt, Ph.D.

Amr Sabry, Ph.D.

Daniel Leivant, Ph.D.

5/18/2020

Copyright 2020
Deyaaeldeen Almahallawi
All rights reserved

To Enas, my wonderful wife.

Acknowledgements

My work was funded by NSF grants CCF-1518844 and CCF-1763922. First and foremost I would like to thank my advisor, Jeremy Siek, for his encouragement, support, and guidance throughout my graduate career. His door was always open, and his insights immensely valuable. I am indebted to him for all that he has taught me, both technical and not. I thank my thesis committee: Sam Tobin-Hochstadt, Amr Sabry, and Daniel Leivant for their advice and encouragement. A special thanks goes to Andre Kuhlenschmidt for hours of stimulating discussions and arguments concerning gradual typing, compilers, and mechanized proofs. Many people provided support and friendship during my time in graduate school. I would like to especially thank Spenser Bauman for countless hours spent debating various topics in compilers in grad school and in our time at MathWorks. Thanks also to all the other students and postdocs who made my stay at the department more enjoyable, especially (in no particular order) Jaime Guerrero, Timothy Zakian, Ambrose Bonnaire-Sergeant, Ryan Scott, Caleb Kisby, Matthew Heimerdinger, Cameron Swords, Matteo Cimini, David Christiansen, Michael Vitousek, Praveen Narayanan, Chris Wailes, Tori Lewis, Laurel Carter, Andrew Kent, Vikraman Choudhury, Omer Agacan, Wren Romano, Rajan Walia, Caner Derici, Mike Vollmer, Tianyu Chen, Kuang-Chen Lu, Fred Fu, Chaitanya Koparkar, Peter Fogg, and Sarah Spall. A special thanks to my friend Mohammad Alaggan for encouraging me to pursue grad school. Without a doubt, I wouldn't be here today if it weren't for his support and belief in me. Furthermore, I would like to thank my parents and sisters for their endless love and support. Finally, I would like to thank Enas, my soulmate, for her love, encouragement, and support.

Deyaaeldeen Almahallawi

TOWARDS EFFICIENT GRADUAL TYPING VIA MONOTONIC REFERENCES AND COERCIONS

Integrating static and dynamic typing into a single programming language enables programmers to choose which discipline to use in each code region. Different approaches for this integration have been studied and put into use at large scale, e.g. TypeScript for JavaScript and adding the dynamic type to C#. Gradual typing is one approach to this integration that preserves type soundness by performing type-checking at run-time using casts. For higher order values such as functions and mutable references, a cast typically wraps the value in a proxy that performs type-checking when the value is used. This approach suffers from two problems: (1) chains of proxies can grow and consume unbounded space, and (2) statically typed code regions need to check whether values are proxied. Monotonic references solve both problems for mutable references by directly casting the heap cell instead of wrapping the reference in a proxy.

In this dissertation, an integration is proposed of monotonic references with the coercion-based solution to the problem of chains of proxies for other values such as functions. Furthermore, the prior semantics for monotonic references involved storing and evaluating cast expressions (not yet values) in the heap and it is not obvious how to implement this behavior efficiently in a compiler and run-time system. This dissertation proposes novel dynamic semantics where only values are written to the heap, making the semantics straightforward to implement. The approach is implemented in Grift, a compiler for a gradually typed programming language, and a few key optimizations are proposed. Finally, the proposed performance evaluation methodology shows that the proposed approach eliminates all overheads associated with gradually typed references in statically typed code regions without introducing significant average-case overhead.

Contents

Chapter 1. Overview 1

1. Static Typing 1
2. Dynamic Typing 2
3. Combining Static Typing and Dynamic Typing 2
4. Problem Statement 4
5. Thesis Statement 9
6. Methodology 10
7. Outline 11

Chapter 2. Review of Gradual Typing 12

1. Type System 12
2. Operational Semantics 14
3. Performance Challenge 16

Chapter 3. Review of Monotonic References 20

1. Adding References to the Gradually Typed Lambda Calculus 20
2. The Efficiency Problem in Proxied References 21
3. Introduction to Monotonic References 23
4. The Monotonic Machine: Monotonic References for Gradual Typing 25

Chapter 4. λ_c^{ref}: Reduction Semantics for Monotonic References 34

1. Static Semantics 34
2. Dynamic Semantics 37
3. Type Safety 41
4. Summary 45

Chapter 5. Practical And Space-Efficient Monotonic References 47

1. The Problem of Expressions on the Heap 47

2. λ_p^{ref}: Practical Dynamic Semantics For Monotonic References 49
3. λ_p^{ref} Type Safety 53
4. λ_S^{ref}: Practical and Space-Efficient Semantics for Monotonic References 59
5. λ_S^{ref} Type Safety 66
6. Space-Efficiency 68

Chapter 6. Grift: A Compiler for Gradual Typing 70
1. Value Representation 72
2. Implementation of Coercions 72
3. Credits 76

Chapter 7. Implementation of Monotonic References in Grift 77
1. Runtime Representations 77
2. Compiling Referece Operations 79
3. Optimizations 81

Chapter 8. Performance Evaluation 84
1. Experimental Methodology for Evaluating Space-Efficiency 84
2. The Runtime Cost of Space Efficiency 90
3. Gradual Typing Overhead and Comparisons 91
4. Experimental Methodology for Evaluating Monotonic References 94
5. Efficient Statically Typed Code 95
6. Evaluation of Monotonic References on Partially Typed Programs 96
7. Evaluation of Monotonic References on Untyped Programs 99
8. Threats to Validity 100

Chapter 9. Conclusions 101

Bibliography 102

Appendix A. Agda Mechanization 110
1. Agda Formalization 110

Appendix B. Grift Macros 112
1. Values, Macros, and Compose 112

Appendix C. Implementation of Monotonic Vectors in Grift 119

Appendix D. Type Hashconsing 121

Curriculum Vitae

CHAPTER 1

Overview

Programming languages can be classified as *statically* or *dynamically typed*. On one hand, statically typed languages perform checks on the source code looking for inconsistencies according to a defined set of typing rules. On the other hand, dynamically typed languages typically interpret the source code right away, and perform checks on inputs to primitives as they are called. Each approach has its own strengths and they are discussed in the next two sections.

1. Static Typing

A static type system allows programmers to specify properties of functions that are proved before each execution. The specification language also serves as a design language that programmers can use to design a big part of the architecture of their program without even writing any code that implements this specification yet. Furthermore, this specification language is closely tied to the code allowing for smooth refactoring and code changes.

Static type systems are deemed valuable as they are a first line of defense against programming errors. In particular, they can catch type errors early in the development cycle by using the typing information to prove the absence of a particular class of bad program behaviors (e.g. loading a string from the heap and try to add it to an integer).

Furthermore, a compiler can take advantage of the typing information present in the source code and produce efficient target code. For instance, if an operation on integers is guaranteed to only receive integer inputs, there is no need to insert checks on the input that verify this condition at runtime. Moreover, the runtime representation of values could be leaner as they do not have to store what their types are. This space saving can lead to more runtime efficiency.

Moreover, the typing information can serve as a reliable, never out-of-date, documentation of the code. If a code change leads to a typing change, type information has to be updated so that the program could be type-checked in order for it to be executed.

Finally, IDEs use typing information to provide a better experience to programmers as they are interactively writing their programs. For instance, IDEs can perform type-checking continuously to highlight code regions that contain type errors and show the error messages if requested. Also, type information helps in auto-completing by providing live suggestions for what to write next.

2. Dynamic Typing

Because of these reasons, some consider static typing as the world's most successful formal method. However, there are scenarios where the static type system can get in the way. As Leroy and Mauny put it "there are programming situations that seem to require dynamic typing in an essential way" [Leroy and Mauny, 1991]. A dynamically typed language accepts all programs at compile-time and type-checking is deferred to runtime. Type-checking occurs in every primitive operation and if one of the arguments fails the runtime type-check, the program execution raises an exception. All values in dynamically typed languages are tagged, i.e. runtime values store its type, which are checked against the expected type in primitive operations.

Dynamic typing has advantages over static typing. Dynamic typing is more expressive than static typing because a static type system can not decide whether or not an arbitrary program will get stuck or reach a state of undefined behavior. Therefore, the type system rejects programs conservatively, even ones that do not have any errors.

Furthermore, some abstractions can not be type-checked. For instance, the `eval` function takes a string as input, interprets it as an expression of the language down to a value, and returns that value. The type of the returned value cannot be known at compile-time, without knowing the type of the expression, so it is type-checked at runtime instead. For this type-checking to work, values returned by eval must carry some type information at runtime.

Finally, language design is simplied with dynamic typing which enables rapid development, especially for prototyping and testing.

3. Combining Static Typing and Dynamic Typing

Consider the following scenario: a programmer had an idea for a new tool or service and created a prototype in an easy-to-write dynamically typed language, e.g. Python or JavaScript. If the prototype gains traction, it would be put into production use, possibly at large scale. The complexity of

the architecture of the prototype could grow over time as new features are added. This could lead to growth in bugs and performance issues. Without having static typing available in the programming language, developers are forced to port the code base to a statically typed language to unlock better performance based on type information and fix bugs that stem from type inconsistencies. However, the cost of such cross-language migration is prohibitive.

Given that each approach has its unique features and advantages, it can be desirable to have both approaches available to programmers instead of limiting them to one per program. With a language that combines both approaches, the prototype could have been first written with no static types, and developers would add type annotations to it over time as needed without the need to migrate to another different language.

The integration of static and dynamic typing and allowing interoperability between typed and untyped code has attracted a lot of interest, in both academia and industry. Many companies built different languages that mix static and dynamic typing in different ways. In particular, Google built Dart [Team, 2014], Facebook engineered HHVM for Hack [Zhao et al., 2012, Ottoni, 2018], and Microsoft added the type Dynamic to C# [Bierman et al., 2010] and introduced TypeScript [Bierman et al., 2014]. On the academic side, earlier approaches of interoperation were proposed [Thatte, 1990, Findler and Felleisen, 2002b, Anderson and Drossopoulou, 2003, Gray and Flatt, 2004, Bracha, 2004, Flanagan, 2006, Lagorio and Zucca, 2006] before gradual typing was proposed [Anderson and Drossopoulou, 2003, Siek and Taha, 2006, Tobin-Hochstadt and Felleisen, 2006, Gronski et al., 2006, Matthews and Findler, 2007]. In the past decade, considerable progress has been made on the theory of gradual typing, such as understanding interactions with objects [Siek and Taha, 2007], generics [Ina and Igarashi, 2011, Ahmed et al., 2011, 2017, Igarashi et al., 2017], mutable state [Herman et al., 2007, Siek et al., 2015c], recursive and set-theoretic types [Siek and Tobin-Hochstadt, 2016, Castagna and Lanvin, 2017], control operators [Sekiyama et al., 2015], and type inference [Siek and Vachharajani, 2008, Garcia and Cimini, 2015, Campora et al., 2017, Migeed and Palsberg, 2019].

The term *gradual typing* is often used to describe language designs that integrate static and dynamic typing. However, some of these designs do not satisfy the original intent of gradual typing because they do not support the convenient evolution of code between the two typing disciplines. This property was formally defined [Siek et al., 2015a] and named *the gradual guarantee*. Without this property, it would be harder to incrementally migrate untyped code to be typed because some

of the configurations, obtained by adding/removing type annotations in the source program, might exhibit unexpected runtime errors.

Furthermore, there are two properties that make gradual typing appealing: soundness and interoperability. Regarding soundness, programmers, compilers, and IDEs would like to trust that the program execution respects the type annotations in the source code. Regarding interoperability, programmers would like to write code where typed code regions and untyped code regions interoperate seamlessly. To preserve soundness and interoperability, gradual typing inserts casts at the boundaries between typed and untyped code regions which can affect performance negatively.

4. Problem Statement

Beginning with the observations of Herman et al. [2007, 2010], that the standard operational semantics for the Gradually Typed Lambda Calculus can exhibit unbounded space leaks, and continuing with the experiments of Takikawa et al. [2016], which showed that Typed Racket [Tobin-Hochstadt and Felleisen, 2008] exhibits high overheads on real programs, and the observation of Siek and Vitousek [2013] that overheads occur even in statically typed programs, it has become clear that efficiency is a serious concern for gradually typed languages.

Herman et al. [2007] attribute the efficiency problem to the space leak which is due to proxies. Casting a function value wraps it in a proxy [Findler and Felleisen, 2002b] that checks, at application sites, whether the types of the input and output values matches that of the cast. Similarly, casting a reference value wraps the reference in a proxy that checks at read sites if the value on the heap could be cast to the type the context expects, and checks at write sites if the written value could be cast to the type of the heap cell.

Using proxies to cast higher-order values causes two problems. First, each time a value of a higher-order type crosses the boundary between a statically and dynamically code regions, it gets wrapped in a proxy. Multiple boundary crossings accumulate into a chain of proxies. These chains of proxies can grow without bounds, causing space leaks. Furthermore, the chains of proxies can cause runtime slowdowns to the extent of changing the asymptotic complexity of a program. Kuhlenschmidt et al. [2019] demonstrate this problem with a simple quicksort program that is fully statically typed except for one annotation, causing a proxy to be created around the vector being sorted at each recursive call to the sort function. The runtimes collected from running the program on different input sizes suggest that the asymptotic complexity changes from $O(n^2)$ to $O(n^3)$.

The second problem with using proxies is the overhead of runtime dispatch on higher-order values at use sites, especially in statically typed code. At each use site, the implementation must check whether the incoming value is proxied, and if so, perform runtime checks before and/or after processing the underlying value. This runtime dispatch is necessary even in statically typed code regions, because a reference from a dynamically typed region may flow into a static region. This problem prevents the performance of statically-typed code in a gradually-typed programming language to be on par with corresponding code in statically-typed programming languages.

To address this concern, the research community has made some first steps towards answering the important scientific question: *What is the essential overhead of gradual typing?* This is a difficult and complex question to answer. First, there is a large language design space: choices regarding the semantics of a gradually typed language have significant impacts on efficiency. Second, there is the engineering challenge of developing the implementation technology necessary for performance evaluations. Third, there is the scientific challenge of inventing techniques to improve efficiency. I discuss these three aspects in the following paragraphs and describe where the work of this dissertation fits in. While we cannot hope to outright answer this question, this work eliminates some spurious factors and provides a rigorous baseline for further experimentation.

The Language Design Space Siek et al. [2015b] describe five criteria for gradually typed languages, including *type soundness* and the *gradual guarantee.*. The type soundness criteria requires that the value of an expression must have the type that was predicted by the type system. The gradual guarantee states that changing type annotations should not change the semantics of the program, except that incorrect type annotations may induce compile-time or run-time errors.

For expediency, many languages from industry (TypeScript [Hejlsberg, 2012, Bierman et al., 2014], Hack [Verlaguet and Menghrajani, 2014], and Flow [Flo, 2017]) are implemented by retrofitting a type system on an existing dynamically typed languages and the compiler erases types and compiles to that dynamically typed language. This approach does not provide type soundness in the above sense.

Several of the designs from academia (Thorn [Bloom et al., 2009], TS* [Swamy et al., 2014], Safe TypeScript [Rastogi et al., 2015, Richards et al., 2017], Strong Script [Richards et al., 2015], and Typed Racket [1] [Tobin-Hochstadt and Felleisen, 2008]) place restrictions on which implicit casts

[1]Typed Racket is sound and partially supports the gradual guarantee: its type system does not satisfy the static part of the gradual guarantee because it requires what amounts to an explicit downcast to use a

Language	Sound	Gradual Guarantee wrt. Structural Types	Nominal Types	Granularity	Approach	Space-Efficient
Gradualtalk	●	●	●	Fine	Retrofit	○
Guarded Reticulated Python	●	●	●	Fine	Retrofit	○
Nom	●	–	●	Fine	From-Scratch	●
GTLC+	●	●	–	Fine	From-Scratch	●
TypeScript	○	●	●	Fine	Retrofit	●
Safe TypeScript	●	○	○	Fine	Retrofit	●
Typed Racket	●	◐	◐	Coarse	Retrofit	●
Transient Reticulated Python	◐	●	●	Fine	Retrofit	●

FIGURE 1. A comparison of gradually typed programming languages.

are allowed, typically for the sake of efficiency, but at the price of losing the gradual guarantee. The fundamental tension is that providing both the gradual guarantee and type soundness means that an implementation must perform runtime type checking at the boundaries between statically typed and dynamically typed regions of code, and that runtime checking can be time consuming.

Another aspect of the language design space that impacts efficiency is whether a gradually typed language includes structural or nominal types. With nominal types, the runtime check for whether a value has a given type is efficient and straightforward to implement. Indeed, Nom [Muehlboeck and Tate, 2017], a nominally typed object-oriented language (without generics or function types), exhibits low overhead on the `sieve` and `snake` benchmarks from the Gradual Typing Performance Benchmarks [GTP, 2018]. On the other hand, with structural types, the runtime check can be much more complex, e.g., for higher-order types it may involve the use of a proxy to mediate between a value and its context.

Finally, gradual typing can be applied at varying granularities. For example, in Typed Racket, a module may be typed or untyped. I refer to this as *coarse-grained gradual typing*. In contrast, in TypeScript [Hejlsberg, 2012, Bierman et al., 2014] and Reticulated Python [Vitousek et al., 2014, 2017], each variable may be typed or untyped, and furthermore, a type annotation can be partial through the use of the unknown type. I refer to this as *fine-grained gradual typing*.

Figure 1 summarizes the discussion up to this point. The top-half of the table lists four languages that meet the criteria for gradual typing whereas the bottom-half includes languages that do not provide type soundness and/or the gradual guarantee. Compared to the other three languages that satisfy the criteria, the GTLC+ language (and Grift compiler) described in this paper is novel in its support for structural types and guaranteed space efficiency.

Racket module from a Typed Racket module. However, the semantics of Typed Racket's runtime checks are compatible with the dynamic part of the gradual guarantee.

This dissertation studies the efficiency of gradually typed languages that include structural types and employ fine-grained gradual typing.

Implementation Technology A popular approach to implementing gradually typed languages is to retrofit a pre-existing language implementation. The benefit of this approach is that it quickly provides support for large number of language features, facilitating performance evaluations on a large number of real programs. The downside is that the pre-existing implementation was not designed for gradual typing and may include choices that interfere with obtaining efficiency on partially typed programs. Many of the gradually typed languages to date have been implemented via compilation to a dynamically typed language, including TypeScript, Typed Racket, Gradualtalk, Reticulated Python, and many more. These implementations incur incidental overhead in the statically typed regions of a program.

The opposite approach is to develop a from-scratch implementation of a gradually typed language. The benefit is that its authors can tailor every aspect of the implementation for efficient gradual typing, but an enormous engineering effort is needed to implement all the language features necessary to run real programs.

For a gradually typed language, one of the most important choices is how to implement runtime type checks. For expediency, Typed Racket uses the Racket contract library [Findler and Felleisen, 2002a]. The contract library is more general than is necessary because it supports arbitrary predicates instead of just type tests. In subsequent years since the performance evaluation of Typed Racket [Takikawa et al., 2016], several performance problems have been fixed [Bauman et al., 2017, Feltey et al., 2018]. It is unclear how much performance is left on the table given the extra layers of abstraction and indirection in the contract library.

To better isolate the essential overheads of gradual typing, this dissertation studies a from-scratch implementation in the context of a simple ahead-of-time compiler with a close-to-the-metal implementation of runtime type checks. The alternative of using just-in-time compilation is a fascinating one [Bauman et al., 2017, Richards et al., 2017], but it is important to also study an implementation whose performance is more predictable, enabling the isolation of causes of overhead. Furthermore, the compiler can be configured to turn off gradual typing all together, i.e. the source language becomes statically typed. This mode enables measuring the overheads in statically typed benchmarks compiled with gradual typing and also verifying the effectiveness of solutions to the

dynamic dispatch problem such as using a closure representation that can act as either a regular closure or as a function proxy but with a uniform calling convention [Siek and Garcia, 2012].

Innovations to Improve Efficiency Perhaps the most challenging obstacle to determining the essential overhead of gradual typing is that creative researchers continue to make innovations that can lower the overhead of gradual typing! To make claims about the essential overhead of gradual typing, these ideas must be implemented and evaluated.

Herman et al. [2007, 2010] observe that the *coercions* of Henglein [1994] (originally designed for the compile-time optimization of dynamically typed languages) can be used to guarantee space efficiency in the proxies needed for higher-order structural types by normalizing coercions. Space efficiency is guaranteed because arbitrarily long sequence of coercions are compressed into an equivalent coercion of length three or less. Thus, the size of a coercion c in normal form is bounded by its height h, $\mathit{size}(c) \leq 5(2^h - 1)$. The height of the coercions generated at compile time and runtime are bounded by the height of the types in the source program. Thus, at any moment during program execution, the amount of space used by the program is $O(n)$, where n is the amount of space ignoring coercions. To date the research on coercions for gradual typing has been of a theoretical nature. The Grift compiler is the first to empirically test the use of coercions to implement runtime casts for gradually typed languages.

For implementations that rely on contracts for runtime checking, space efficiency is also a concern and Greenberg [2014] discovered a way to compress sequences of contracts, making them space efficient. Feltey et al. [2018] implement This technique, *collapsible contracts*, in the Racket contract library and demonstrate that it significantly improves the performance of Typed Racket on some benchmarks. However, contracts cannot be compressed to the same degree as coercions (predicates are more expressive than types), which means that the time overhead has larger constant factors, factors that depend on the total number of contracts in a program.

Another innovation is the notion of *monotonic references* [Siek et al., 2015c], which has the potential to eliminate the overhead of gradual typing in statically typed regions of code, effectively solving the dynamic dispatch issue for references. However, Siek et al. [2015c] left as future work the challenge of identifying a normal form for coercions with monotonic references and of developing a function for composing such coercions. Furthermore, the monotonic heap maps addresses to *expressions* that are reduced using a small-step reduction relation. (Normally only values are stored in the heap.) It is not obvious how to implement this reduction-in-the-heap in a compiled

implementation of a gradually-typed language. In this dissertation, I study monotonic references, design a new dynamic semantics for it that writes values only to the heap and integrate it with the coercion-based solution to the space-efficiency problem. Furthermore, I use the new semantics to reduce overhead in the context of the Grift compiler.

Furthermore, I present evidence that efficiency can be achieved in a fine-grained gradually typed language with structural types. I contributed to the design, implementation, and evaluation of an ahead-of-time compiler, named Grift, that uses carefully chosen runtime representations to implement coercions.

The input language includes a selection of language features that are difficult to implement efficiently in a gradually typed language: first-class functions, mutable arrays, and equirecursive types. The language is an extension of the Gradually Typed Lambda Calculus, abbreviated GTLC+.

5. Thesis Statement

In this dissertation, I study the challenge of achieving efficiency in a gradual typing system for a challenging combination of points in the language design space, implementation technology, and innovations. Regarding the language design space, I study the efficiency problem in the context of a gradual type system that is conjectured to satisfy the gradual guarantee and enjoys soundness. Furthermore, this gradual type system supports structural types and fine-grained gradual typing. Regarding implementation technology, the gradual type system is implemented from-scratch as an ahead-of-time compiler. Finally, regarding research innovations, the studied gradual type system uses coercions to achieve space-efficiency in partially typed code and a novel dynamic semantics for monotonic references to achieve efficiency in statically typed code.

The thesis of this dissertation is that using monotonic references and coercions can bring the cost of fine-grained gradual typing down to a much more reasonable level:

Monotonic references can be combined with coercions in normal form in a manner that guarantees space-efficiency and the new calculus can be implemented in a way that writes values only to the heap and it eliminates all overheads related to gradually typed references from statically typed code regions and improves the performance on average in partially typed code regions.

6. Methodology

In this thesis, I utilize monotonic references and coercion to minimize the cost of runtime type checking in fine-grained gradual typing with structural types. I design a new dynamic semantics for monotonic references that writes values only to the heap. The new semantics is mechanized in the Agda proof assistant. Furthermore, I design a new normal form for coercions that include one for monotonic references along with a composition operator and I prove that their height is bounded to guarantee space-efficiency.

Moreover, I contributed to the design and implementation of an ahead-of-time compiler for a gradually typed language. The support for coercions in the compiler can be turned on and off to enable measuring the cost of space-efficiency. Furthermore, the support for monotonic references can also be turned on and off, to enable measuring the cost of monotonic references. Finally, the compiler can be configured to turn off the support for gradual typing all together, i.e. the source language becomes statically typed, to enable measuring the overhead of gradual typing in statically typed benchmarks.

Finally, I contributed to the design and implementation of various experiments to evaluate the performance of gradual typing and to answer a few key research questions about the efficiency of gradual typing, namely:

(1) **What is the time cost of achieving space efficiency with coercions?**

(2) **What is the overhead of gradual typing?** I subdivide this question into the overheads on programs that are (a) statically typed, (b) untyped, and (c) partially typed.

(3) **Do monotonic references eliminate overheads associated with gradual typing from statically typed code?**

(4) **What is the overhead of using monotonic references in partially typed and untyped code?**

7. Outline

The rest of the dissertation is organized as follows. Chapter 2 provides background on gradual typing including the type-based and coercions approaches. In Chapter 3, I review monotonic references by presenting an earlier work [Siek and Vitousek, 2013], the monotonic abstract machine that is mechanized in Isabelle/HOL. Chapter 4 presents a variant of the reduction semantics for monotonic references [Siek et al., 2015c] that I mechanized in the Agda proof assistant and solves a few minor problems in prior work. In Chapter 5, I present two novel dynamic semantics for monotonic references that are also mechanized in the Agda proof assistant. The first one provides a cast function that returns a value (that can then be written to the heap) and a list of casts on addresses that are put into a queue for subsequent processing, making the semantics straightforward to implement. Furthermore, the second semantics adds space-efficiency to the first by providing a new normal form for coercions along with a composition operator. Chapter 6 presents the design of an ahead-of-time compiler, named Grift, for the GTLC+ and in Chapter 7 I describe the implementation of monotonic references in Grift. Finally, the performance of Grift is evaluated in Chapter 8 to answer the research questions listed earlier and Chapter 9 concludes.

CHAPTER 2

Review of Gradual Typing

In this chapter, I review gradual typing by presenting the reduction semantics for the gradually typed lambda calculus with pairs. From a language design perspective, gradual typing touches both the type system and the operational semantics. The key to the type system is the *consistency* relation on types, which enables implicit casts to and from the dynamic type (a.k.a the unknown type), while still catching static type errors [Anderson and Drossopoulou, 2003, Siek and Taha, 2006, Gronski et al., 2006].

1. Type System

Recall the typing rule for applications in a traditional statically typed language.

$$\frac{\Gamma \vdash M : T_1 \to T_2 \qquad \Gamma \vdash N : T_1}{\Gamma \vdash M\,N : T_2}$$

The rule specifies that the type of the argument N has to exactly match the type expected by the function M. This rigidness is relaxed in a gradual typed language as the type of the argument no longer needs to exactly match the type of function parameter. Instead, they just needs to be consistent, i.e. the two types needs to match except for the unknown parts. More formally, the application typing rule is defined as follows:

$$\frac{\Gamma \vdash M : T_1 \to T_2 \qquad \Gamma \vdash N : T_1' \qquad T_1 \sim T_1'}{\Gamma \vdash M\,N : T_2}$$

The consistency relation is defined in Figure 1 where the dynamic type is denoted by $\star$. The consistency relation is reflexive and symmetric but unlike subtyping it is not transitive.

Furthermore, values of the dynamic type are also allowed to be applied as functions to arguments of arbitrary types. This flexibility resembles that of dynamically typed languages as no checks are carried out by the type system and are deferred instead until runtime. This idea is captured by the following extra application typing rule:

$$\begin{array}{llcl} \text{Base types} & B & ::= & () \mid \mathsf{Bool} \mid \mathsf{Int} \\ \text{Types} & T, S \in \mathcal{T} & ::= & \star \mid B \mid T \to T \mid T \times T \\ \text{Contexts} & \Gamma & ::= & \emptyset \mid \Gamma, x : T \end{array}$$

Consistency $\boxed{T \sim T}$

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

Type precision $\boxed{T \sqsubseteq T}$

$$T \sqsubseteq \star \qquad B \sqsubseteq B \qquad \frac{T_1 \sqsubseteq T_3 \quad T_2 \sqsubseteq T_4}{T_1 \to T_2 \sqsubseteq T_3 \to T_4} \qquad \frac{T_1 \sqsubseteq T_3 \quad T_2 \sqsubseteq T_4}{T_1 \times T_2 \sqsubseteq T_3 \times T_4}$$

FIGURE 1. Types for the gradually typed lambda calculus.

The syntax of the source language

$$\text{Expressions} \quad e, M, N \quad ::= \quad k \mid \lambda x.\, e \mid e\, e \mid \langle e, e \rangle \mid \texttt{fst}\ e \mid \texttt{snd}\ e$$

Typing rules for the source language $\boxed{\Gamma \vdash M : T}$

$$\frac{}{\Gamma \vdash k : \textsc{type}(k)} \qquad \frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash \lambda(x : T_1).\, M : T_1 \to T_2}$$

$$\frac{\Gamma \vdash M : T_1 \to T_2 \quad \Gamma \vdash N : T_1' \quad T_1 \sim T_1'}{\Gamma \vdash M\, N : T_2} \qquad \frac{\Gamma \vdash M : \star \quad \Gamma \vdash N : T}{\Gamma \vdash M\, N : \star}$$

$$\frac{\Gamma \vdash M : T_1 \quad \Gamma \vdash N : T_2}{\Gamma \vdash \langle M, N \rangle : T_1 \times T_2} \qquad \frac{\Gamma \vdash \langle M, N \rangle : T_1 \times T_2}{\Gamma \vdash \texttt{fst}\ M : T_1} \qquad \frac{\Gamma \vdash \langle M, N \rangle : T_1 \times T_2}{\Gamma \vdash \texttt{snd}\ M : T_2}$$

$$\frac{\Gamma \vdash \langle M, N \rangle : \star}{\Gamma \vdash \texttt{fst}\ M : \star} \qquad \frac{\Gamma \vdash \langle M, N \rangle : \star}{\Gamma \vdash \texttt{snd}\ M : \star}$$

FIGURE 2. The grammar and the typing rules for the source language.

$$\frac{\Gamma \vdash M : \star \quad \Gamma \vdash N : T}{\Gamma \vdash M\, N : \star}$$

Figure 1 defines base types to be the unit type (), Booleans Bool, and integers Int. Types are the dynamic type, base types, and structural types (Functions and Pairs). All of the types, except for $\star$, classify unboxed values, e.g. Int is the type of native integers. On the other hand, values of the dynamic type are values of all other types tagged with their runtime type.

The type precision relation $\sqsubseteq$ (a.k.a naive subtyping) is a partial ordering on types where related types are consistent and the type on the left has more information than the type on the right. For example $\mathsf{Int} \to \mathsf{Int} \sqsubseteq \mathsf{Int} \to \star \sqsubseteq \star$ but $\mathsf{Int} \to \star \not\sqsubseteq \star \to \mathsf{Int}$. This relation plays a central role in monotonic references as we shall see in the next chapter.

Typing rules for the target language $\boxed{\Gamma \vdash M : T}$

$$\frac{}{\Gamma \vdash k : \textsc{type}(k)} \qquad \frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash M : T}{\Gamma \vdash M\langle T \Rightarrow T' \rangle : T'} \qquad \frac{}{\Gamma \vdash \texttt{error} : T}$$

$$\frac{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash \lambda(x : T_1).\, M : T_1 \to T_2} \qquad \frac{\Gamma \vdash M : T_1 \to T_2 \quad \Gamma \vdash N : T_1}{\Gamma \vdash M\, N : T_2}$$

$$\frac{\Gamma \vdash M : T_1 \quad \Gamma \vdash N : T_2}{\Gamma \vdash \langle M, N \rangle : T_1 \times T_2} \qquad \frac{\Gamma \vdash \langle M, N \rangle : T_1 \times T_2}{\Gamma \vdash \texttt{fst}\ M : T_1} \qquad \frac{\Gamma \vdash \langle M, N \rangle : T_1 \times T_2}{\Gamma \vdash \texttt{snd}\ M : T_2}$$

Cast insertion $\boxed{\Gamma \vdash M \hookrightarrow N : T}$

$$\frac{(x : T) \in \Gamma}{\Sigma \mid \Gamma \vdash x \hookrightarrow x : T} \qquad \frac{\Sigma \mid x : T_1, \Gamma \vdash M \hookrightarrow N : T_2}{\Sigma \mid \Gamma \vdash \lambda(x : T_1).\, M \hookrightarrow \lambda(x : T_1).\, N : T_1 \to T_2}$$

$$\frac{\Sigma \mid \Gamma \vdash M \hookrightarrow M' : T_1 \to T_2 \quad \Sigma \mid \Gamma \vdash N \hookrightarrow N' : T_1'}{\Sigma \mid \Gamma \vdash M\, N \hookrightarrow M'\, (N'\langle T_1' \Rightarrow T_1 \rangle) : T_2}$$

$$\frac{\Sigma \mid \Gamma \vdash M \hookrightarrow M' : \star \quad \Sigma \mid \Gamma \vdash N \hookrightarrow N' : T}{\Sigma \mid \Gamma \vdash M\, N \hookrightarrow (M'\langle \star \Rightarrow (\star \Rightarrow \star) \rangle)\, (N'\langle T \Rightarrow \star \rangle) : \star}$$

$$\frac{\Sigma \mid \Gamma \vdash M \hookrightarrow M' : T_1 \quad \Sigma \mid \Gamma \vdash N \hookrightarrow N' : T_2}{\Sigma \mid \Gamma \vdash \langle M, N \rangle \hookrightarrow \langle M', N' \rangle : T_1 \times T_2}$$

$$\frac{\Sigma \mid \Gamma \vdash M \hookrightarrow N : T_1 \times T_2}{\Sigma \mid \Gamma \vdash \texttt{fst}\ M \hookrightarrow \texttt{fst}\ N : T_1} \qquad \frac{\Sigma \mid \Gamma \vdash M \hookrightarrow N : T_1 \times T_2}{\Sigma \mid \Gamma \vdash \texttt{snd}\ M \hookrightarrow \texttt{snd}\ N : T_2}$$

$$\frac{\Sigma \mid \Gamma \vdash M \hookrightarrow N : \star}{\Sigma \mid \Gamma \vdash \texttt{fst}\ M \hookrightarrow \texttt{fst}\ (N\langle \star \Rightarrow \star \times \star \rangle) : \star} \qquad \frac{\Sigma \mid \Gamma \vdash M \hookrightarrow N : \star}{\Sigma \mid \Gamma \vdash \texttt{snd}\ M \hookrightarrow \texttt{snd}\ (N\langle \star \Rightarrow \star \times \star \rangle) : \star}$$

Figure 3. Static semantics for the target language.

Figure 2 presents the syntax and typing rules for our gradually typed language. The typing rules include the two rules I presented earlier in addition to rules for the rest of the syntax where there is two rules for each elimination form and the extra rule handles the case where the expression being eliminated is typed at $\star$.

2. Operational Semantics

The dynamic semantics of the gradually typed lambda calculus is defined by a type-directed translation to the simply typed lambda calculus with explicit casts (a.k.a the cast calculus) and each use of consistency between types T and T' in the type system becomes an explicit cast from T to T'. The compilation of gradually-typed terms into the cast calculus is otherwise straightforward and is defined in Figure 3.

Runtime structures

$$
\begin{array}{llll}
\text{Values} & v & ::= & k \mid \lambda(x:T).\,e \mid \langle v, v\rangle \mid v\langle T \Rightarrow \star\rangle \mid v\langle T_1 \to T_2 \Rightarrow T_1' \to T_2'\rangle \\
\text{Frames} & \mathrm{F} & ::= & \square\, e \mid v\,\square \mid \langle \square, e\rangle \mid \langle v, \square\rangle \mid \texttt{fst}\ \square \mid \texttt{snd}\ \square \mid \square\langle T \Rightarrow T'\rangle
\end{array}
$$

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

$$
\begin{array}{rcll}
v\langle T \Rightarrow T\rangle & \longrightarrow_c & v & \\
v\langle I \Rightarrow \star\rangle\langle \star \Rightarrow J\rangle & \longrightarrow_c & v\langle I \Rightarrow J\rangle & \\
v\langle I \Rightarrow J\rangle & \longrightarrow_c & \texttt{error} & \text{if } I \nsim J \\
\langle v_1, v_2\rangle\langle T_1 \times T_2 \Rightarrow T_1' \times T_2'\rangle & \longrightarrow_c & \langle v_1\langle T_1 \Rightarrow T_1'\rangle, v_2\langle T_2 \Rightarrow T_2'\rangle\rangle &
\end{array}
$$

Program reduction rules $\boxed{e \longrightarrow e}$

$$
\begin{array}{rcl}
(\lambda(x:T).\,M)\ v & \longrightarrow & [x := v]M \\
(v_1\langle T_1 \to T_2 \Rightarrow T_1' \to T_2'\rangle)\ v_2 & \longrightarrow & (v_1\ (v_2\langle T_1' \Rightarrow T_1\rangle))\langle T_2 \Rightarrow T_2'\rangle
\end{array}
\qquad
\begin{array}{rcl}
\texttt{fst}\ \langle v_1, v_2\rangle & \longrightarrow & v_1 \\
\texttt{snd}\ \langle v_1, v_2\rangle & \longrightarrow & v_2
\end{array}
$$

$$
\textsc{Cast}\ \frac{e \longrightarrow_c e'}{e \longrightarrow e'}
\qquad
\textsc{Cong}\ \frac{M \longrightarrow N}{\mathrm{F}[M] \longrightarrow \mathrm{F}[N]}
\qquad
\textsc{CongErr}\ \frac{}{\mathrm{F}[\texttt{error}] \longrightarrow \texttt{error}}
$$

Figure 4. Simple operational semantics for gradual typing

The dynamic semantics of casts is closely related to the semantics of *contracts* [Findler and Felleisen, 2002a, Gray et al., 2005], *coercions* [Henglein, 1994], and *interlanguage migration* [Tobin-Hochstadt and Felleisen, 2006, Matthews and Findler, 2007]. Because of the shared mechanisms with these other lines of research, much of the ongoing research in those areas benefits the theory of gradual typing, and vice versa [Guha et al., 2007, Matthews and Ahmed, 2008, Greenberg et al., 2010, Dimoulas et al., 2011, Strickland et al., 2012, Chitil, 2012, Dimoulas et al., 2012, Greenberg, 2015].

Before discussing the operational semantics, I define injectable types as types that are allowed to be cast to the dynamic type, which are base types and structural types (Functions and Pairs).

$$I, J ::= B \mid T \to T \mid T \times T$$

Figure 4 presents the definitions of values and frames and lists the reduction rules. In contrast to the simply typed lambda calculus, the gradually typed lambda calculus includes two extra values, the first is $v\langle I \Rightarrow \star\rangle$, a value of the dynamic type which is an arbitrary value v tagged with its type I. Furthermore, the values of the function type now includes normal functions and proxied functions (functions that were cast) $v\langle T_1 \to T_2 \Rightarrow T_1' \to T_2'\rangle$.

Cast expressions are reduced as follows. If the source and target types are the same, the cast expression is reduced to just the value. Furthermore, if there is a cast to a type J on a dynamic value tagged with the type I, the expression reduces to a value with a cast from I to J. Moreover,

if the source and target types of the cast are not consistent, the expression reduces to `error`. Finally, a cast expression on a pair reduces to a pair expression where each projection is cast with the corresponding cast. But what happens when a function value gets cast? A function proxy is created, wrapping the old value and storing the type information of the cast. When such a proxy is applied, the incoming arguments get cast to the type of the underlying function. Furthermore, after the function application finishes, the return value get cast to the proxy return type. We shall see that these proxies pose performance challenges.

3. Performance Challenge

Consider the classic example of Herman et al. [2007] shown in Figure 5. Two mutually recursive functions, `even?` and `odd?`, are written in GTLC+. This example uses continuation passing style to concisely illustrate efficiency challenges in gradual typing. While this example is contrived, the same problems occur in real programs under complex situations [Feltey et al., 2018]. On the left side of the figure we have a partially typed function, named `even?`, that checks if an integer is even. On the right side of the figure we have a fully typed function, named `odd?`, that checks if an integer is odd. With gradual typing, both functions are well typed because implicit casts are allowed to and from `Dyn`. For example, in `even?` the parameter `n` is implicitly cast from `Dyn` to `Int` when it is used as the argument to `=` and `-`. These casts check that the dynamic value is tagged as an integer and perform the conversion needed between the representation of tagged values and integers. Conversely, the value `#t` is cast to `Dyn` because it is used as the argument to a function that expects `Dyn`. This cast tags the value with runtime type information so that later uses can be checked. Likewise, in `odd?` the `Int` passed as the first argument to `even?` is cast to `Dyn`.

The types of the variables named `k`, `(Dyn -> Bool)` and `(Bool -> Bool)` are consistent with each other. As such, when `k` is passed as an argument to `even?` or `odd?`, there is an implicit cast between these two types. This cast is traditionally implemented by wrapping the function with a proxy that checks the argument and return values [Findler and Felleisen, 2002a], but Herman et al. [2007] observe that the value of `k` passes through this cast at each iteration, causing a build up of proxies that changes the space complexity from constant to $\mathcal{O}(n)$.

In this dissertation I consider two approaches to the implementation of runtime casts: traditional casts, which I refer to as *type-based casts*, and *coercions*. Type-based casts provide the most

```
(define even? : (Dyn (Dyn -> Bool) -> Bool)
  (lambda ([n : Dyn] [k : (Dyn -> Bool)])
    (if (= n 0)
         (k #t)
         (odd? (- n 1) k))))

(define odd? : (Int (Bool -> Bool) -> Bool)
  (lambda ([n : Int] [k : (Bool -> Bool)])
    (if (= n 0)
         (k #f)
         (even? (- n 1) k))))
```

FIGURE 5. Gradually typed even? and odd? functions that have been written in continuation passing style.

straightforward implementation, but the proxies they generate can accumulate and consume an unbounded amount of space as discussed above.

Getting back to the `even?` and `odd?` example, I compare the time and space usage of type-based casts versus coercions in Figure 7 (left hand side). The three plots show the runtime, number of casts performed, and length of the longest chain of proxies, as the input parameter `n` is increased. The plot concerning longest proxy chains shows that type-based casts accumulate longer chains of proxies as we increase parameter `n`. On the other hand, coercions use a constant amount of space by compressing these proxies chains into a single proxy of constant size.

The appearance of long proxy chains can also change the time complexity of a program. I refer to such a change as a *catastrophic slowdown*. Figure 6 shows the code for the quicksort algorithm. The program is statically typed except for the the vector parameter of `sort!`. The single `Dyn` annotation in this type causes runtime overhead inside the auxiliary `partition!` and `swap!` functions. Like function types, reference types require proxies that apply casts during read and write operations. Again, a naive implementation of casts allows the proxies to accumulate; each recursive call to `sort!` causes a cast that adds a proxy to the vector being sorted. In quicksort, this changes the worst-case time complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n^3)$ because each read (`vector-ref`) and write (`vector-set!`) traverses a chain of proxies of length $\mathcal{O}(n)$.

Returning to Figure 7, but focusing on the right-hand side plots for quicksort, we observe that, for typed-based casts, the longest proxy chain grows as we increase the size of the array being sorted. On the other hand, coercions do extra work at each step to compress the cast. As a result they pay more overhead for each cast, but when they use the cast value later the overhead is guaranteed

```
(define sort! : ((Vect Int) Int Int -> ())
  (lambda ([v : (Vect Dyn)]
           [lo : Int] [hi : Int])
    (when (< lo hi)
      (let ([pivot : Int (partition! v lo hi)])
        (sort! v lo (- pivot 1))
        (sort! v (+ pivot 1) hi)))))

(define swap! : ((Vect Int) Int Int -> ())
  (lambda ([v : (Vect Int)] [i : Int] [j : Int])
    (let ([tmp : Int (vector-ref v i)])
      (vector-set! v i (vector-ref v j))
      (vector-set! v j tmp1))))

(define partition! : ((Vect Int) Int Int -> Int)
  (lambda ([v : (Vect Int)] [l : Int] [h : Int])
    (let ([p : Int (vector-ref v h)]
          [i : (Ref Int) (box (- h 1))])
      (repeat (j l h)
        (when (<= (vector-ref v j) p)
          (box-set! i (+ (unbox i) 1))
          (swap! v (unbox i) j)))
      (swap! v (+ (unbox i) 1) h)
      (+ (unbox i) 1))))
```

FIGURE 6. The `sort!` function implements the Quicksort algorithm in the GTLC+.

to be constant. This can be seen in the way the runtime grows rapidly for type-based casts, while coercions remain (relatively) low. I confirmed via polynomial regression that the type-based cast implementation's runtime is modeled by a third degree polynomial, i.e. $\mathcal{O}(n^3)$.

Review of Coercions

Coercions are combinators that specify how to convert from one type to another type. The following grammar shows the coercions needed to represent casts between types that include the unknown type `Dyn`, base types (units, integers, and Booleans), function types, and pairs.

$$c, d \quad ::= \quad I? \mid I! \mid \iota \mid \bot \mid c \to d \mid c \times d$$

The coercion $I?$ is a projection that checks whether a tagged value is of type I. If it is, the underlying value is returned. If not, an error is signaled. The coercion $I!$ in an injection that tags a value with its type. The identity coercion ι just returns the input value. The failure coercion $\bot$ signals an error when applied to a value. A function coercion $c \to d$ changes the type of a function

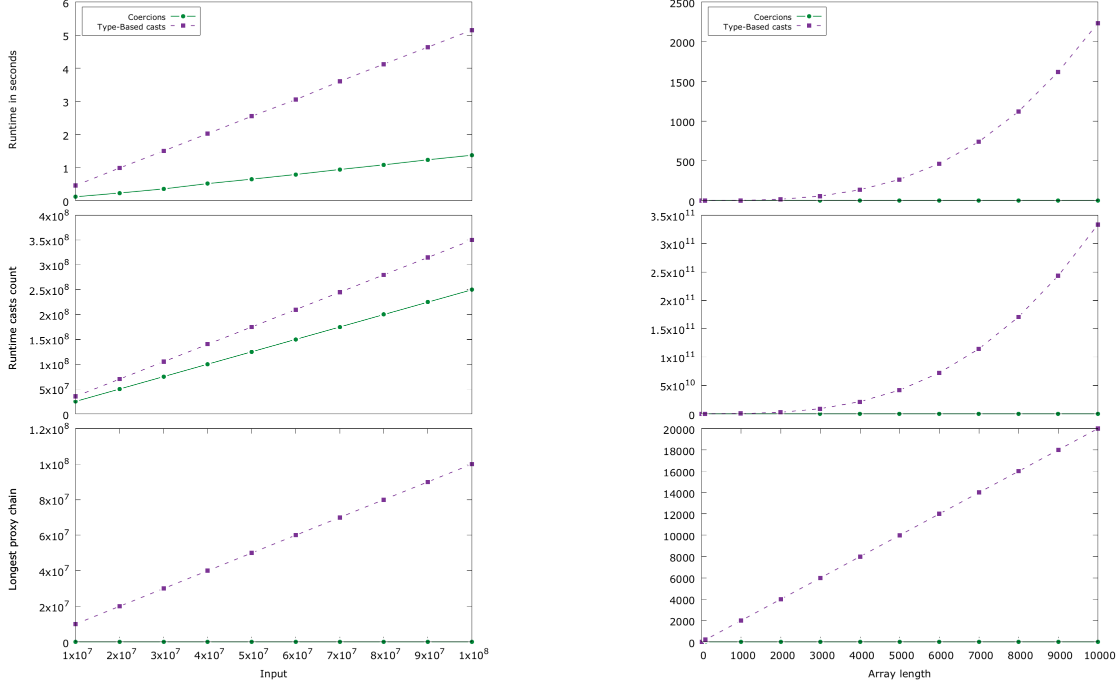


FIGURE 7. The runtime, number of casts, and longest proxy chain (y-axes) as we increase the parameter `n` (x-axis) for even/odd (left) and the array length (x-axis) for quicksort (right). The plots for longest proxy chain show that coercions compress casts and thus operate in constant space. The plot of runtime for quicksort shows that long proxy chains can also increase the asymptotic time complexity of an algorithm.

by applying c to the argument and d to the return value. A pair coercion $c \times d$ changes the type of a pair by applying c to the first projection and d to the second.

When two coercions are applied in sequence, they are composed. However, composing coercions requires that sequences of compositions are treated as equal up to associativity. Siek et al. [2015a] defines a normal form for coercions along with a composition operator, written $c \mathbin{;} d$, that takes two coercions in normal form and directly computes the normal form of sequencing them together. The Grift compiler implements this approach using efficient bit-level representations.

CHAPTER 3

Review of Monotonic References

There are a few different gradual typing systems with mutable references in the literature. For instance, Siek and Taha [2006] present one design for mutable references where reference types are invariant with respect to consistency. In this design, implicit casts are disallowed between different pointed-to types. However, because of this limitation, it has been shown that this design does not satisfy the gradual guarantee [Siek et al., 2015b], a property that ensures proper behavior when changing types in a program. Proxied references [Herman et al., 2007, 2010] fixes this issue by allowing implicit casts between different pointed-to types if those types are consistent. To ensure type safety, cast addresses are wrapped in proxies which perform runtime type-checking at read and write sites. However, this design suffers from overheads in statically typed code regions because of the dynamic dispatch on incoming values. Monotonic references [Siek and Vitousek, 2013, Siek et al., 2015c] solves this efficiency problem by casting the heap cell directly instead of creating a proxy.

In this chapter, earlier work on monotonic references [Siek and Vitousek, 2013] is reviewed. The chapter is organized as follows: Section 1 extends the source language with references. Section 2 gives a formal account of a simple variant of proxied references and explains the efficiency issues with that semantics. Section 3 presents a high level overview of monotonic references and the motivation behind its design. Section 4 presents the first formal definition of monotonic references [Siek and Vitousek, 2013], an abstract machine for monotonic references that is mechanized in Isabelle/HOL, and discusses how this semantics solves the efficiency issue presented in Section 2.

1. Adding References to the Gradually Typed Lambda Calculus

Figure 1 extends the set of types in the gradually typed lambda calculus with the reference type. The reference type $\mathsf{Ref}\ T$, where T is the type of the pointed-to value, is added to the set of injectable types. The type constructor Ref is covariant, so the consistency and type precision relations are

$$\begin{array}{llcl} \text{Injectable types} & I, J & ::= & \cdots \mid \mathsf{Ref}\ T \\ \text{Heap typing} & \Sigma & ::= & \emptyset \mid \Sigma, a : T \end{array}$$

Consistency $\boxed{T \sim T}$ Type precision $\boxed{T \sqsubseteq T}$

$$\cdots \quad \frac{T_1 \sim T_2}{\mathsf{Ref}\ T_1 \sim \mathsf{Ref}\ T_2} \qquad \cdots \quad \frac{T_1 \sqsubseteq T_2}{\mathsf{Ref}\ T_1 \sqsubseteq \mathsf{Ref}\ T_2}$$

FIGURE 1. Extending types and relations on them (defined in Figure 1 in Chapter 2) with references.

The syntax of the source language

$$\text{Expressions} \quad e, M, N \quad ::= \quad \cdots \mid \texttt{ref}\ e \mid !e \mid e := e$$

Typing rules for the source language $\boxed{\Gamma \vdash M : T}$

$$\vdots$$

$$\frac{\Sigma \mid \Gamma \vdash M : T}{\Sigma \mid \Gamma \vdash \texttt{ref}\ M : \mathsf{Ref}\ T} \qquad \frac{\Gamma \vdash M : \mathsf{Ref}\ T}{\Gamma \vdash !M : T} \qquad \frac{\Gamma \vdash M : \star}{\Gamma \vdash !M : \star}$$

$$\frac{\Gamma \vdash M : \mathsf{Ref}\ T \quad \Gamma \vdash N : T' \quad T \sim T'}{\Gamma \vdash M := N : ()} \qquad \frac{\Gamma \vdash M : \star \quad \Gamma \vdash N : T}{\Gamma \vdash M := N : ()}$$

FIGURE 2. Extending the typing rules for the source language (defined in Figure 2 in Chapter 2) with references.

extended accordingly. Heap typing is an association list of addresses and types. $\Sigma(a) = T$ indicates that the address a has type T in Σ.

Figure 2 presents the typing rules for references. A new reference is allocated by the form $\texttt{ref}\ M$ and the pointed-to value is read by the form $!M$ and is updated by the form $M := N$. The typing rules for those forms in the source language are standard [Herman et al., 2007, 2010] and they allow reading from and writing to either a reference or a dynamic expression.

Target languages with different semantics for references will be presented throughout the thesis.

2. The Efficiency Problem in Proxied References

Figure 3 extends the semantics of the gradually typed lambda calculus, presented in Figures 3 and 4 in Chapter 2, with proxied references. The semantics uses the space-inefficient type-based representation of casts to simplify the presentation. However, the reader interested in the space-efficient semantics can refer to the literature [Herman et al., 2007, 2010]. The typing judgment relation for the target language becomes a quaternary relation, as it now also relates store typings in addition to contexts, expressions, and types. The typing rules for the target language are also

Typing rules for the target language $\boxed{\Sigma \mid \Gamma \vdash M : T}$

$$\ldots \qquad \text{WT-Addr}\frac{\Sigma(a) = T}{\Sigma \mid \Gamma \vdash a : \mathsf{Ref}\ T} \qquad \frac{\Sigma \mid \Gamma \vdash M : T}{\Sigma \mid \Gamma \vdash \mathtt{ref}\ M : \mathsf{Ref}\ T} \qquad \frac{\Sigma \mid \Gamma \vdash M : \mathsf{Ref}\ T}{\Sigma \mid \Gamma \vdash !M : T}$$

$$\frac{\Sigma \mid \Gamma \vdash M : \mathsf{Ref}\ T \qquad \Sigma \mid \Gamma \vdash N : T}{\Sigma \mid \Gamma \vdash M := N : ()}$$

Cast insertion $\boxed{\Gamma \vdash M \hookrightarrow N : T}$

$$\ldots \qquad \frac{\Gamma \vdash M \hookrightarrow N : T}{\Gamma \vdash \mathtt{ref}\ M \hookrightarrow \mathtt{ref}\ N : \mathsf{Ref}\ T}$$

$$\frac{\Gamma \vdash M \hookrightarrow N : \mathsf{Ref}\ T}{\Gamma \vdash !M \hookrightarrow !N : T} \qquad \frac{\Gamma \vdash M \hookrightarrow N : \star}{\Gamma \vdash !M \hookrightarrow !(N\langle \star \Rightarrow \mathsf{Ref}\ \star\rangle) : \star}$$

$$\frac{\Gamma \vdash M \hookrightarrow M' : \mathsf{Ref}\ T \qquad \Gamma \vdash N \hookrightarrow N' : T' \qquad T \sim T'}{\Gamma \vdash M := N \hookrightarrow M' := (N'\langle T' \Rightarrow T\rangle) : ()}$$

$$\frac{\Gamma \vdash M \hookrightarrow M' : \star \qquad \Gamma \vdash N \hookrightarrow N' : T}{\Gamma \vdash M := N \hookrightarrow (M'\langle \star \Rightarrow \mathsf{Ref}\ \star\rangle) := (N'\langle T' \Rightarrow \star\rangle) : ()}$$

Runtime structures

$$\begin{array}{llcl} \text{Values} & v & ::= & \cdots \mid a \mid v\langle \mathsf{Ref}\ T \Rightarrow \mathsf{Ref}\ T'\rangle \\ \text{Heap} & \mu & ::= & \emptyset \mid \mu(a \mapsto v) \\ \text{Frames} & F & ::= & \cdots \mid \mathtt{ref}\ \square \mid !\square \mid v := \square \end{array}$$

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

$$\ldots$$

(Alloc) $\quad \mathtt{ref}\ v, \mu \longrightarrow a, \mu(a \mapsto v) \quad \text{if } a \notin \mathrm{dom}(\mu)$

(Read) $\quad !a, \mu \longrightarrow \mu(a), \mu$

(ProxyRead) $\quad !(v\langle \mathsf{Ref}\ T \Rightarrow \mathsf{Ref}\ T'\rangle), \mu \longrightarrow (!v)\langle T \Rightarrow T'\rangle, \mu$

(Write) $\quad a := v, \mu \longrightarrow \mathrm{Unit}, \mu(a \mapsto v)$

(ProxyWrite) $\quad (v\langle \mathsf{Ref}\ T \Rightarrow \mathsf{Ref}\ T'\rangle) := v', \mu \longrightarrow v := (v'\langle T' \Rightarrow T\rangle), \mu$

Figure 3. Extension of the target language (Figures 3 and 4 in Chapter 2) with proxied references.

standard and the key rule WT-Addr says that the type of any allocated address has to match the type of the pointed-to heap cell. Later on, it will be shown that the corresponding typing rule for monotonic references is slightly different to allow monotonic changes to the types of heap cells. The heap for proxied references is well typed, written $\Sigma \vdash \mu$, if and only if $\mathrm{dom}(\Sigma) = \mathrm{dom}(\mu)$ and $\forall a \in \mathrm{dom}(\Sigma).\ \Sigma \mid \emptyset \vdash \mu(a) : \Sigma(a)$.

Heaps are maps from addresses to values. The program reduction relation $\longrightarrow$ becomes quaternary and relates the old and new heaps in addition to the old and new expressions. Moreover, the set of values is extended with the two valid values of a reference type: an address and a value

wrapped in a proxy that holds a reference cast. The program reduction relation has a rule for each combination of a reference value and operation. The READ and WRITE rules reduce reference operations on addresses, where the former rule reduces to the value mapped to that address and keeps the heap the same and the latter rule reduces to a unit and a new heap where the address maps to the new value. Furthermore, The PROXYREAD and PROXYWRITE rules reduce reference operations on proxied values, where the former rule performs a type check to see whether the read value from the heap respects the cast carried by the proxy. In particular, PROXYREAD reduces to another read expression that is cast from the source pointed-to type to the target pointed-to type. On the other hand, PROXYWRITE performs a type check to see whether the value that is about to be written to the heap respects the cast carried by the proxy. In other words, PROXYWRITE reduces the write expression into another where the written value is cast from the target pointed-to type to the source pointed-to type.

As explained, there are two different kinds of values for proxied references and reduction has to handle each kind for each reference operation. This is implemented in practice by dynamic dispatch on the incoming value. Figure 4 presents the C code for the `ref_read` function that performs a read on a value of the proxied reference type. A version of this function is called in compiled code by Grift (in the type-based casts and proxied references mode) for a read operation. A dynamic dispatch is performed (at line 5) to check whether the incoming value is proxied and act accordingly. This dynamic dispatch can affect performance negatively and is performed even in statically typed code regions. This prevents the performance of statically typed code in a gradually typed language from being on par with the performance of statically typed languages.

3. Introduction to Monotonic References

With gradual typing, programmers can initially prototype their programs quickly as dynamically typed, without paying the mental overhead of ensuring consistent typing up-front. Later, they can gradually add type annotations to their programs until they become statically typed or typed enough to unlock the best performance. However, if that performance is not on par with the performance of existing statically typed programming languages, adaptability of gradual typing could be at risk, especially in domains that require high performance.

Monotonic references is another design for gradually typed mutable references that does not have the performance problem in proxied references discussed in the last section. The motivation

```c
int64_t apply_cast(int64_t value, int64_t source, int64_t target);
bool is_ref_proxied(int64_t value);

int64_t ref_read(int64_t val) {
    if (is_ref_proxied val) {
        int64_t underlying_val = ref_proxy_val(val);
        int64_t source_type = ref_proxy_source(val);
        int64_t target_type = ref_proxy_target(val);
        int64_t read_val = ref_read(underlying_val);
        return apply_cast(read_val, source_type, target_type);
    } else {
        return *((int64_t *) val);
    }
}
```

FIGURE 4. The C code for the `ref_read` function that performs a read on a value of the proxied reference type.

Static types $\boxed{\S\, T}$

$$\S\, B \qquad \frac{\S\, T_1 \quad \S\, T_2}{\S\, T_1 \to T_2} \qquad \frac{\S\, T_1 \quad \S\, T_2}{\S\, T_1 \times T_2} \qquad \frac{\S\, T}{\S\, \mathsf{Ref}\ T}$$

Meet operation (greatest lower bound) $\boxed{T \sqcap T = T}$

$$\star \sqcap T = T \sqcap \star = T \qquad B \sqcap B = B \qquad (T_1 \times T_2) \sqcap (T_3 \times T_4) = (T_1 \sqcap T_3) \times (T_2 \sqcap T_4)$$
$$\mathsf{Ref}\ T_1 \sqcap \mathsf{Ref}\ T_2 = \mathsf{Ref}\ (T_1 \sqcap T_2) \qquad (T_1 \to T_2) \sqcap (T_3 \to T_4) = (T_1 \sqcap T_3) \to (T_2 \sqcap T_4)$$

FIGURE 5. The definition of the static type predicate and the meet operation.

behind the design of monotonic references is to have a performance in statically typed code regions on par with the performance of statically typed programming languages. In particular, monotonic references ensures that read and write expressions can be compiled to just loads and stores, something a typical statically typed programming language does. The key feature of monotonic references is to have addresses only as the value of the reference type so there is no different kind of values to do a dynamic dispatch on when performing a reference operation. This section gives a high level overview of monotonic references.

Figure 5 defines relations and functions on types. The unary relation $\S$ is defined to hold for types that do not contain the dynamic type $\star$ anywhere.

When an address gets cast, the heap cell mapped to that particular address gets cast. In other words, monotonic references does strong updates to the heap. Furthermore, the type of the cell

is ensured to be consistent with the types of all references pointing to it, to maintain type safety via an invariant, named the monotonic invariant. This invariant restricts the type of the heap cell to be always more precise than the type of any reference pointing to it in the program and is maintained as follows: when an address gets cast, the heap cell gets cast with a type that is the result of combining the target type of the cast and the current type of the heap cell and that new type is more precise than both those types. The $\sqcap$ function [Siek and Wadler, 2010] creates that new type by computing the greatest lower bound of the input types.

The monotonic invariant is key to unlock the best performance in statically typed code regions. If an address is statically typed, then it must be the case that the value it points to has the same type. This way, reading and writing expressions can be compiled in the same way they are compiled in an efficient statically typed programming language, to loads and stores.

If the address is not statically typed, then it may be the case that the value on the heap has a type that is different from the type of the address. In this case, a cast between those two types needs to be performed when reading and writing.

The two cases can be distinguished by the cast insertion relation based on whether the type of the reference is fully static.

4. The Monotonic Machine: Monotonic References for Gradual Typing

Monotonic references were first introduced as an abstract machine [Siek and Vitousek, 2013]. This work is presented in this section and is referred to as the *monotonic machine*.

4.1. Type System. Figure 6 present the grammar and the typing rules of the target language. The syntax is in A-normal form, so that complex computations are built from a sequence of variable assignments. This representation makes sure expressions are not built from sub-expressions, which simplifies the presentation of operational semantics and type safety proofs because there is no need for evaluation contexts or congruence rules. An in-depth argument for this technique can be found in Siek's Five Easy Lemmas series [Siek, 2012a,b].

Expressions are constant literals, functions, applications of built-in operators, pairs, and reading from a statically typed reference. Statements are let-binding of complex expressions where the

The syntax of the target language

$$\begin{array}{llcl} \text{Expressions} & e, M, N & ::= & k \mid x \mid \lambda(x:T).\,\mathcal{S} \mid op\, e \mid \langle e, e\rangle \mid\, !e \\ \text{Statements} & \mathcal{S} & ::= & \text{let } x = e \text{ in } \mathcal{S} \mid \text{return } e \mid \text{let } x = e\,e \text{ in } \mathcal{S} \mid \text{return } e\,e \mid \\ & & & \text{let } x = \texttt{ref}\ e@T \text{ in } \mathcal{S} \mid e := e;\ \mathcal{S} \mid e := e@T;\ \mathcal{S} \mid \\ & & & \text{let } x = !e@T \text{ in } \mathcal{S} \mid \text{let } x = e\langle T \Rightarrow T\rangle \text{ in } \mathcal{S} \end{array}$$

Typing rules for the target language $\boxed{\Gamma \vdash M : T}$

$$\frac{}{\Gamma \vdash k : \textsc{type}(k)} \qquad \frac{(x:T) \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\textsc{op-type}(op) = T_1 \to T_2 \quad \Gamma \vdash M : T_1}{\Gamma \vdash op\, M : T_2}$$

$$\frac{\Gamma, x : T_1 \vdash \mathcal{S} : T_2}{\Gamma \vdash \lambda(x:T_1).\,\mathcal{S} : T_1 \to T_2} \qquad \frac{\Gamma \vdash M : T_1 \quad \Gamma \vdash N : T_2}{\Gamma \vdash \langle M, N\rangle : T_1 \times T_2} \qquad \frac{\Gamma \vdash M : \mathsf{Ref}\ T \quad \int T}{\Gamma \vdash !M : T}$$

Statement typing for the target language $\boxed{\Gamma \vdash \mathcal{S} : T}$

$$\frac{\Gamma \vdash M : T_1 \quad \Gamma, x : T_1 \vdash \mathcal{S} : T_2}{\Gamma \vdash \text{let } x = M \text{ in } \mathcal{S} : T_2} \qquad \frac{\Gamma \vdash M : T}{\Gamma \vdash \text{return } M : T}$$

$$\frac{\Gamma \vdash M : T_1 \to T_2 \quad \Gamma \vdash N : T_1 \quad \Gamma, x : T_2 \vdash \mathcal{S} : T_3}{\Gamma \vdash \text{let } x = M\,N \text{ in } \mathcal{S} : T_3} \qquad \frac{\Gamma \vdash M : T_1 \to T_2 \quad \Gamma \vdash N : T_1}{\Gamma \vdash \text{return } (M\,N) : T_2}$$

$$\frac{\Gamma \vdash M : T \quad \Gamma, x : \mathsf{Ref}\ T \vdash \mathcal{S} : T'}{\Gamma \vdash \text{let } x = \texttt{ref}\ M@T \text{ in } \mathcal{S} : T'} \qquad \frac{\Gamma \vdash M : \mathsf{Ref}\ T \quad \Gamma \vdash N : T \quad \int T \quad \Gamma \vdash \mathcal{S} : T'}{\Gamma \vdash M := N;\ \mathcal{S} : T'}$$

$$\frac{\Gamma \vdash M : \mathsf{Ref}\ T \quad \Gamma \vdash N : T \quad \Gamma \vdash \mathcal{S} : T'}{\Gamma \vdash M := N@T;\ \mathcal{S} : T'} \qquad \frac{\Gamma \vdash M : \mathsf{Ref}\ T \quad \Gamma, x : T \vdash \mathcal{S} : T'}{\Gamma \vdash !M@T;\ \mathcal{S} : T'}$$

$$\frac{\Gamma \vdash M : T_1 \quad \Gamma, x : T_2 \vdash \mathcal{S} : T_3}{\Gamma \vdash \text{let } x = M\langle T_1 \Rightarrow T_2\rangle \text{ in } \mathcal{S} : T_3}$$

FIGURE 6. Typing rules for expressions and statements in the monotonic machine.

bound variable is in the scope of another statement. Complex expressions include simple expressions, reference allocation, writing into a statically typed reference, reading from and writing into a reference that is not statically typed, and a cast expression.

There are two syntactic forms for reading from and writing into a reference. let $x = !e@T$ in $\mathcal{S}$ and $e := e@T; \mathcal{S}$ are used for references that are not statically typed. They record the type of the reference (the type annotation T) to be used to perform the mediating cast. As explained earlier, the mediating cast is performed because the type of the value on the heap does not necessarily have the same type T. On the other hand, the syntactic forms $!e$ and $e := e$ in $\mathcal{S}$ are used for statically typed references as their typing rules require the reference types to satisfy the predicate $\int$. The operational semantics for those forms are efficient and those forms could be compiled to loads and stores. The typing rules for all other forms are standard.

Runtime structures

$$
\begin{array}{llcl}
\text{Values} & v \in \mathcal{V} & ::= & k \mid \langle \lambda(x:T).\, e, \rho \rangle \mid \langle v, v \rangle \mid v\langle T \Rightarrow \star \rangle \mid a \\
\text{Environments} & \rho \in \mathcal{E} & ::= & \emptyset \mid \rho, x : v \\
\text{Delayed casts} & dc & ::= & v \mid v\langle T \Rightarrow T \rangle \\
\text{Heaps} & \mu & ::= & \emptyset \mid \mu(a \mapsto v : T) \\
\text{Evolving heaps} & \nu \in \mathcal{H} & ::= & \mu \mid \nu(a \mapsto dc : T)
\end{array}
$$

Environments typing rules $\boxed{\Sigma \mid \Gamma \vdash \rho}$

$$
\Sigma \mid \emptyset \vdash \emptyset \qquad \frac{\Sigma \vdash v : T \qquad \Sigma \mid \Gamma \vdash \rho}{\Sigma \mid \Gamma, x : T \vdash \rho, x : v}
$$

Values typing rules $\boxed{\Sigma \vdash v : T}$

$$
\frac{}{\Sigma \vdash k : \textsc{type}(k)} \qquad \frac{\Sigma(a) \sqsubseteq T}{\Sigma \vdash a : \mathsf{Ref}\ T} \qquad \frac{\Sigma, x : T_1 \vdash \mathcal{S} : T_2 \qquad \Sigma \mid \Gamma \vdash \rho}{\Sigma \vdash \langle \lambda x : T_1.\, \mathcal{S}, \rho \rangle : T_1 \to T_2}
$$

$$
\frac{\Sigma \vdash v_1 : T_1 \qquad \Sigma \vdash v_2 : T_2}{\Sigma \vdash \langle v_1, v_2 \rangle : T_1 \times T_2} \qquad \frac{\Sigma \vdash v : T}{\Sigma \vdash v\langle T \Rightarrow \star \rangle}
$$

Delayed casts typing rules $\boxed{\Sigma \vdash dc : T}$

$$
\Sigma \vdash v : T \qquad \frac{\Sigma \vdash v : T \qquad T' \sqsubseteq T}{\Sigma \vdash v\langle T \Rightarrow T' \rangle : T'}
$$

FIGURE 7. Typing rules for runtime structures in the monotonic machine.

Figure 7 present the definition of the runtime structures along with typing rules for them. Environments are association lists of variables and values. Furthermore, heaps for monotonic references are unusual in two ways. First, every heap cell is tagged with its type. For convenient access, the type of a heap cell is stored next to the cell in the heap, and is referred to as the run-time type information (RTTI). Second, In addition to values, heaps also could contain cast expressions, called delayed casts. Delayed casts are arbitrary casts on values and are stored on the heap to ensure the correctness of cast reduction in the presence of cyclic values. The need for storing delayed casts on the heap will be explained in detail after presenting the operational semantics. $\nu(a \mapsto dc : T)$ indicates that the address a in the heap ν maps to a heap cell that contains the delayed cast dc and the cell is tagged with the type T. Furthermore, $\nu(a)_{\text{val}}$ and $\nu(a)_{\text{rtti}}$ are functions to read both components of the heap cell mapped to address a in the heap ν.

The typing rule for addresses allow the address to have a type that is different from the type of the heap cell. However, the rule enforces the monotonic invariant by requiring the type of heap cell to be more precise than the type of the address. The typing rules for other values are standard. However for delayed casts, the target type of the cast is required to be more precise than the source type. This makes sure that the heap can be cast only to a more precise type.

$$\begin{array}{llll} \text{Activation lists} & \gamma \in \mathcal{A} & ::= & \emptyset \mid \gamma, a \\ \text{Cast result} & \mathsf{r} \in \mathcal{R} & ::= & (v, \nu, \gamma) \mid \texttt{error} \end{array}$$

Type grounding function $\boxed{\lfloor\ \rfloor : \mathcal{T} \to \mathcal{T}}$

$$\lfloor \star \rfloor = \star \qquad \lfloor B \rfloor = B \qquad \lfloor T_1 \to T_2 \rfloor = \star \to \star \qquad \lfloor T_1 \times T_2 \rfloor = \star \times \star \qquad \lfloor \mathsf{Ref}\ T \rfloor = \mathsf{Ref}\ \star$$

Cast application function $\boxed{\text{apply-cast} : \mathcal{V} \times \mathcal{T} \times \mathcal{T} \times \mathcal{H} \times \mathcal{A} \to \mathcal{R}}$

$$\begin{array}{rcl}
\text{apply-cast}(a, \mathsf{Ref}\ _, \mathsf{Ref}\ T, \nu, \gamma) & = & (a, \nu, \gamma) \text{ if } \nu(a)_{\text{rtti}} = \nu(a)_{\text{rtti}} \sqcap T \\
\text{apply-cast}(a, \mathsf{Ref}\ _, \mathsf{Ref}\ T_1, \nu, \gamma) & = & (a, \nu(a \mapsto v\langle T_2 \Rightarrow T_3\rangle : T_3), (a, \gamma)) \\
& & \text{if} \begin{cases} \nu(a) = v : T_2, \\ T_3 = T_1 \sqcap T_2, \\ T_3 \neq T_2 \end{cases} \\
\text{apply-cast}(a, \mathsf{Ref}\ _, \mathsf{Ref}\ T_1, \nu, \gamma) & = & (a, \nu(a \mapsto v\langle T_2 \Rightarrow T_4\rangle : T_4), (a, \gamma)) \\
& & \text{if} \begin{cases} \nu(a) = v\langle T_2 \Rightarrow T_3\rangle : T_3, \\ T_4 = T_1 \sqcap T_3, \\ T_4 \neq T_3 \end{cases} \\
\text{apply-cast}(\langle v_1, v_2\rangle, T_1 \times T_2, T_3 \times T_4, \nu, \gamma) & = & (\langle v_1', v_2'\rangle, \nu'', \gamma'') \\
& & \text{if} \begin{cases} \text{apply-cast}(v_1, T_1, T_3, \nu, \gamma) & = & (v_1', \nu', \gamma'), \\ \text{apply-cast}(v_2, T_2, T_4, \nu', \gamma') & = & (v_2', \nu'', \gamma'') \end{cases} \\
\text{apply-cast}(v, B, B, \nu, \gamma) & = & (v, \nu, \gamma) \\
\text{apply-cast}(v, \star, \star, \nu, \gamma) & = & (v, \nu, \gamma) \\
\text{apply-cast}(v, I, \star, \nu, \gamma) & = & (v\langle I \Rightarrow \star\rangle, \nu, \gamma) \\
\text{apply-cast}(f, T_1 \to T_2, T_3 \to T_4, \nu, \gamma) & = & (\langle(\lambda(x : T_3).\, \text{let } y = x\langle T_3 \Rightarrow T_1\rangle \text{ in let } z = f\, y \\
& & \ \text{in let } w = z\langle T_2 \Rightarrow T_4\rangle \text{ in return } w), (\emptyset, f : f)\rangle, \\
& & \nu, \gamma) \\
\text{apply-cast}(v\langle T \Rightarrow \star\rangle, \star, I, \nu, \gamma) & = & \text{apply-cast}(v, T, I, \nu, \gamma) \text{ if } \lfloor T \rfloor = \lfloor I \rfloor \\
\text{apply-cast}(v, T, T', \nu, \gamma) & = & \texttt{error}
\end{array}$$

FIGURE 8. Cast dynamic semantics for the monotonic machine.

4.2. Dynamic Semantics of Casts. Figure 8 present the function apply-cast that applies a cast on a value and could update the heap in the case of a cast on a reference. It takes a value to cast, the source type, the target type, the heap, and an activation list. The activation list is a list of addresses that maps to heap cells that possibly contain delayed casts that need to be reduced to normal values. The function apply-cast returns a tuple of the new value, the updated heap, and the updated activation list, and could possibly error.

There are four cases when casting an address. First, if the greatest lower bound of the target type and the RTTI is the same as the RTTI, then there is no need to cast the heap cell, so the heap stays the same. Second, if the greatest lower bound is a different type, then the heap needs to be updated and there is two cases here. If the heap contains a value, a delayed cast is created from that value and the cast from the old RTTI to the new computed type and is written to the

Expression evaluation function $\boxed{[\![\ ]\!] : \mathsf{E} \times \mathcal{E} \times \mathcal{H} \rightarrow \mathcal{V}}$

$$\begin{array}{rcl} [\![x]\!]\rho\ \mu & = & \rho(x) \\ [\![k]\!]\rho\ \mu & = & k \\ [\![op\, e]\!]\rho\ \mu & = & \delta\ op\ ([\![e]\!]\rho\ \mu) \end{array} \qquad \begin{array}{rcl} [\![\langle M, N\rangle]\!]\rho\ \mu & = & \langle([\![M]\!]\rho\ \mu), ([\![N]\!]\rho\ \mu)\rangle \\ [\![\lambda(x:T).\,\mathcal{S}]\!]\rho\ \mu & = & \langle\lambda(x:T).\,\mathcal{S}, \rho\rangle \\ [\![!e]\!]\rho\ \mu & = & \mu([\![e]\!]\rho\ \mu)_{\mathrm{val}} \end{array}$$

FIGURE 9. Expression evaluation in the monotonic machine.

heap. On the other hand, if there is already a delayed cast on the heap, the old cast on the value is discarded and a new cast is created from the source type of the old cast to the new computed type and the new delayed cast is written to the heap as well. Furthermore, in both previous cases, the address being cast is added to the activation list so that the new delayed cast on the heap is signaled to be reduced. Finally, if the greatest lower bound is undefined, apply-cast errors.

In the case of pairs, apply-cast is applied to both projections from left to right and a pair of the results is returned. If the source and target types are matching base types or the dynamic type, the input value is returned as is. If a value is cast to the dynamic type and the value itself is not dynamic, a dynamic value is returned. If a dynamic value gets cast to a type that is not dynamic, the underlying value gets cast to the target type. If the value being cast is a function f, f gets wrapped in another function that casts the argument to f from the new argument type to the type of the argument f expects, and casts the return value from the return type of f to the new one. Note that this semantics is not space-efficient because functions get wrapped in these proxies every time they get cast, causing a space-leak. Furthermore, when such a function gets applied, a possibly long chain of casts gets applied before and after the underlying closure gets applied, causing a significant runtime overhead. Finally, the function apply-cast errors on all other cases.

4.3. Dynamic Semantics of Programs. Figure 9 present the evaluation function for expressions. In the case of reading from a statically typed reference, the statically typed value on the heap is returned. The evaluation of all other expressions is standard.

Figure 10 presents the state transition rules for the abstract machine. Transition rules relate configuration states consisting of five parts, a statement to reduce, an environment, the procedure call stack, the heap, and the activation list. The first four rules are transitions in the case of a non-empty activation list. All delayed casts have to be reduced to normal values so that the heap is in a normal state before resuming program reduction. The FALSEPOSITIVE rule discards addresses from the activation list that map to cells with normal values. NORTTICHANGE reads a delayed cast from the heap, reduces it to a value, and writes the result back to the heap if the RTTI did not

$$
\begin{array}{llcl}
\text{Procedure call stack} & k & ::= & \emptyset \mid (x, \mathcal{S}, \rho), k \\
\text{States} & \mathsf{s} & ::= & \langle \mathcal{S}, \rho, k, \nu, \gamma \rangle \mid \texttt{error}
\end{array}
$$

State transition rules $\boxed{\mathsf{s} \longrightarrow \mathsf{s}}$

$$
\textsc{FalsePositive}\ \frac{\nu(a) = v : T}{\langle \mathcal{S}, \rho, k, \nu, (a, \gamma) \rangle \longrightarrow \langle \mathcal{S}, \rho, k, \nu, \gamma \rangle}
$$

$$
\textsc{NoRTTIChange}\ \frac{\nu(a) = v\langle T_1 \Rightarrow T_2 \rangle : T_2 \quad \text{apply-cast}(v, T_1, T_2, \nu, (a, \gamma)) = (v', \nu', \gamma') \quad \nu'(a)_{\text{rtti}} = T_2}{\langle \mathcal{S}, \rho, k, \nu, (a, \gamma) \rangle \longrightarrow \langle \mathcal{S}, \rho, k, \nu'(a \mapsto v' : T_2), \gamma' \rangle}
$$

$$
\textsc{RTTIChanged}\ \frac{\nu(a) = v\langle T_1 \Rightarrow T_2 \rangle : T_2 \quad \text{apply-cast}(v, T_1, T_2, \nu, (a, \gamma)) = (v', \nu', \gamma') \quad \nu'(a)_{\text{rtti}} \neq T_2}{\langle \mathcal{S}, \rho, k, \nu, (a, \gamma) \rangle \longrightarrow \langle \mathcal{S}, \rho, k, \nu', \gamma' \rangle}
$$

$$
\textsc{Error1}\ \frac{\nu(a) = v\langle T_1 \Rightarrow T_2 \rangle : T_2 \quad \text{apply-cast}(v, T_1, T_2, \nu, (a, \gamma)) = \texttt{error}}{\langle \mathcal{S}, \rho, k, \nu, (a, \gamma) \rangle \longrightarrow \texttt{error}}
$$

$$
\textsc{Alloc}\ \frac{[\![M]\!]\rho\ \mu = v \quad a = |\mu|}{\langle (\text{let } x = \texttt{ref } M@T \text{ in } \mathcal{S}), \rho, k, \mu, \emptyset \rangle \longrightarrow \langle \mathcal{S}, (\rho, x : a), k, (\mu, (a, v : T)), \emptyset \rangle}
$$

$$
\textsc{Write}\ \frac{[\![M]\!]\rho\ \mu = a \quad [\![N]\!]\rho\ \mu = v \quad \mu(a) = v' : T}{\langle (M := N; \mathcal{S}), \rho, k, \mu, \emptyset \rangle \longrightarrow \langle \mathcal{S}, \rho, k, \mu(a \mapsto v : T), \emptyset \rangle}
$$

$$
\textsc{DynRead}\ \frac{[\![M]\!]\rho\ \mu = a \quad \mu(a) = v : T' \quad \text{apply-cast}(v, T', T, \mu, \emptyset) = (v', \nu, \gamma)}{\langle (\text{let } x = !M@T \text{ in } \mathcal{S}), \rho, k, \mu, \emptyset \rangle \longrightarrow \langle \mathcal{S}, (\rho, x : v'), k, \nu, \gamma \rangle}
$$

$$
\textsc{FailedDynRead}\ \frac{[\![M]\!]\rho\ \mu = a \quad \mu(a) = v : T' \quad \text{apply-cast}(v, T', T, \mu, \emptyset) = \texttt{error}}{\langle (\text{let } x = !M@T \text{ in } \mathcal{S}), \rho, k, \mu, \emptyset \rangle \longrightarrow \texttt{error}}
$$

$$
\textsc{DynWrite}\ \frac{[\![M]\!]\rho\ \mu = a \quad [\![N]\!]\rho\ \mu = v \quad \mu(a) = v' : T'}{\langle (M := N@T; \mathcal{S}), \rho, k, \mu, \emptyset \rangle \longrightarrow \langle \mathcal{S}, \rho, k, \mu(a \mapsto v\langle T \Rightarrow T' \rangle : T'), \{a\} \rangle}
$$

$$
\textsc{cast}\ \frac{[\![M]\!]\rho\ \mu = v \quad \text{apply-cast}(v, T, T', \mu, \emptyset) = (v', \nu, \gamma)}{\langle (\text{let } x = M\langle T \Rightarrow T' \rangle \text{ in } \mathcal{S}), \rho, k, \mu, \emptyset \rangle \longrightarrow \langle \mathcal{S}, (\rho, x : v'), k, \nu, \gamma \rangle}
$$

$$
\textsc{FailedCast}\ \frac{[\![M]\!]\rho\ \mu = v \quad \text{apply-cast}(v, T, T', \mu, \emptyset) = \texttt{error}}{\langle (\text{let } x = M\langle T \Rightarrow T' \rangle \text{ in } \mathcal{S}), \rho, k, \mu, \emptyset \rangle \longrightarrow \texttt{error}}
$$

$$
\frac{[\![M]\!]\rho\ \mu = v}{\langle (\text{let } x = M \text{ in } \mathcal{S}), \rho, k, \mu, \emptyset \rangle \longrightarrow \langle \mathcal{S}, (\rho, x : v), k, \mu, \emptyset \rangle}
$$

$$
\frac{[\![M]\!]\rho\ \mu = v \quad [\![N]\!]\rho\ \mu = v' \quad v' = \langle \lambda(y : T).\, \mathcal{S}', \rho' \rangle}{\langle (\text{let } x = M\ N \text{ in } \mathcal{S}), \rho, k, \mu, \emptyset \rangle \longrightarrow \langle \mathcal{S}', (\rho', y : v'), ((x, \mathcal{S}, \rho), k), \mu, \emptyset \rangle}
$$

$$
\frac{[\![M]\!]\rho\ \mu = v \quad [\![N]\!]\rho\ \mu = v' \quad v' = \langle \lambda(x : T).\, \mathcal{S}', \rho' \rangle}{\langle (\text{return } M\ N), \rho, k, \mu, \emptyset \rangle \longrightarrow \langle \mathcal{S}', (\rho', x : v'), k, \mu, \emptyset \rangle}
$$

$$
\frac{[\![M]\!]\rho\ \mu = v}{\langle (\text{return } M), \rho, ((x, \mathcal{S}, \rho'), k), \mu, \emptyset \rangle \longrightarrow \langle \mathcal{S}, (\rho', x : v), k, \mu, \emptyset \rangle}
$$

FIGURE 10. The state transition rules for the monotonic machine.

```
1  (: f Dyn)
2  (define f (lambda (x) x))
3  (: r Dyn)
4  (define r (box (cons f '())))
5  (set-box! r (cons f r)) ;; establish a cycle
6  (define (g [x : (Boxof (Pair (Dyn -> Integer) (Boxof (Pair (Integer -> Dyn) Dyn))))])
7    ((car (unbox x)) 42))
8  (g r)
```

FIGURE 11. A program written in a variant of Typed Racket syntax to demonstrate the need for writing delayed casts to the heap.

change. RTTICHANGED discards the result of the cast if the RTTI has changed. The intuition is that if the RTTI has changed, this means that the heap already has a more precise value/delayed cast, so the less precise value in hand can no longer be written to the heap. Finally, ERROR1 reduces to the error state if the reduction of the delayed cast raised an error.

The next five rules specify the operational semantics for reference operations. The ALLOC rule allocates a new heap cell by extending the heap by one element at the end. The RTTI is initialized from the type annotation recorded in the syntactic form. This type is the type of the value of the reference as found by the type-checker. The WRITE rule writes a value to a statically typed reference and it does not perform any casts because the type of the written value matches the type of the heap cell according to the monotonic invariant. The DYNREAD rule reads from a partially typed reference and performs a cast from the RTTI to the type of the reference. FAILEDDYNREAD handles the case when the cast raised an error. Finally, the DYNWRITE specifies how to write a value to a partially typed reference. In particular, a delayed cast is written to the heap that casts from the type of the written value to the RTTI.

The CAST and FAILEDCAST rules are defined in terms of the apply-cast function. The rest of the rules are standard.

To understand why the monotonic machine puts cast expressions on the heap, consider the example in Figure 11 that creates a cycle in the heap, written in a variant of Typed Racket syntax that supports the dynamic type `Dyn` to enable fine-grained gradual typing. (Typed Racket uses the term "box" to mean reference. The `Boxof` type corresponds to the Ref type.). On line 2, a reference is allocated at some address, call it a, and placed in variable r; the heap cell at a is initialized with a pair of values of the dynamic type. On line 3, a cycle is created by assigning to

r another pair whose second element is r. So at this point the heap is in the following state.

$$\{a \mapsto \langle(\lambda(x:\star).\,\text{return } x)\langle\star\to\star\Rightarrow\star\rangle, a\langle\mathsf{Ref}\ (\star\times\star)\Rightarrow\star\rangle\rangle : \star\times\star\}$$

On line 6, function g is applied to r, but g expects a reference to a pair of type

$$(\star\to\mathsf{Int})\times\mathsf{Ref}\ ((\mathsf{Int}\to\star)\times\star)$$

So a cast will be performed before proceeding with the function application. The cast reduces using the CAST tule that performs the call:

$$\text{apply-cast}(a, (\mathsf{Ref}\ \star\times\star), (\mathsf{Ref}\ (\star\to\mathsf{Int})\times\mathsf{Ref}\ ((\mathsf{Int}\to\star)\times\star)), \mu, \emptyset)$$

which updates address a with the following expression, where the value part of the delayed cast is colored purple and the cast is colored magenta:

$$a \mapsto \langle(\lambda(x:\star).\,\text{return } x)\langle\star\to\star\Rightarrow\star\rangle, a\langle\mathsf{Ref}\ (\star\times\star)\Rightarrow\star\rangle\rangle\langle\star\times\star\Rightarrow(\star\to\mathsf{Int})\times\mathsf{Ref}\ ((\mathsf{Int}\to\star)\times\star)\rangle$$
$$: (\star\to\mathsf{Int})\times\mathsf{Ref}\ ((\mathsf{Int}\to\star)\times\star)$$

The call returns the address a, the updated heap ν, and the activation list $(\emptyset, a)$. Because the activation list is not empty, a step is taken via either NoRTTIChange or RTTIChanged, depending on the heap returned from the call

$$\text{apply-cast}(\langle(\lambda(x:\star).\,\text{return } x)\langle\star\to\star\Rightarrow\star\rangle, a\langle\mathsf{Ref}\ (\star\times\star)\Rightarrow\star\rangle\rangle, (\star\times\star),$$
$$((\star\to\mathsf{Int})\times\mathsf{Ref}\ ((\mathsf{Int}\to\star)\times\star)), \nu, (\emptyset, a))$$

apply-cast casts the first projection first, then the second projection and returns the result value. However, the pair has a dynamic address in the second projection causing the following cast application:

$$\text{apply-cast}(a, (\mathsf{Ref}\ (\star\times\star)), (\mathsf{Ref}\ ((\mathsf{Int}\to\star)\times\star)), \nu, (\emptyset, a))$$

The address a is cast to the greatest lower bound of its current type and the target type of the cast, which amounts to:

$$(\mathsf{Int}\to\mathsf{Int})\times\mathsf{Ref}\ ((\mathsf{Int}\to\star)\times\star)$$

This cast is interesting because it causes another cast on the first element of the pair. This cast will update the RTTI and the delayed cast on the heap to:

$$a \mapsto \langle(\lambda(x:\star).\,\text{return }x)\langle\star\to\star\Rightarrow\star\rangle, a\langle\mathsf{Ref}\ (\star\times\star)\Rightarrow\star\rangle\rangle\langle\star\times\star\Rightarrow(\mathsf{Int}\to\mathsf{Int})\times\mathsf{Ref}\ ((\mathsf{Int}\to\star)\times\star)\rangle$$
$$:(\mathsf{Int}\to\mathsf{Int})\times\mathsf{Ref}\ ((\mathsf{Int}\to\star)\times\star)$$

This means the top-level reduction is via the RTTIChanged rule because the RTTI has changed during the apply-cast call, so the result of the call is not written to the heap and discarded.

Proceeding with a reduction step via NoRTTIChange yields the following state.

$$a \mapsto \langle(\lambda(x:\star).\,\text{return }x)\langle\star\to\star\Rightarrow\mathsf{Int}\to\mathsf{Int}\rangle, a\rangle : (\mathsf{Int}\to\mathsf{Int})\times\mathsf{Ref}\ ((\mathsf{Int}\to\star)\times\star)$$

In the above sequence of reductions, it was crucial that the cast on the first element of the pair to $\star\to\mathsf{Int}$ was written to the heap before the first element was cast again to the type $\mathsf{Int}\to\star$, enabling a greatest lower bound of $\mathsf{Int}\to\mathsf{Int}$ that took both types into account.

4.4. Type Safety. The semantics is type safe and the proof is mechanized in Isabelle. The reader interested in the proof details can refer to the paper [Siek and Vitousek, 2013].

CHAPTER 4

$\lambda_c^{\mathbf{ref}}$: Reduction Semantics for Monotonic References

In the previous chapter, monotonic references were introduced via an abstract machine that uses the type-based approach to represent casts. A subsequent work on monotonic references [Siek et al., 2015c], will be referred to as λ^{ref}, presents a different formalism of monotonic references based on reduction semantics. Although it is known how to derive a reduction semantics from an abstract machine [Biernacka and Danvy, 2009], there are still other differences between λ^{ref} and the machine. One major difference is in the representation of casts; the machine uses type-based casts and λ^{ref} uses coercions. Other differences will be discussed during the chapter.

As the first technical contribution of this dissertation, this chapter presents λ_c^{ref}, a variant of λ^{ref}, that is mechanized in the Agda proof assistant and fixes a few problems in λ^{ref}. The chapter is organized as follows: Section 1 presents the type system, Section 2 presents the dynamic semantics, and Section 3 presents the type safety proof and associated lemmas.

1. Static Semantics

Types and relations on them are the same as the ones in Figure 1 and 5 in Chapter 3. Furthermore, a weaker notion of consistency, called shallow consistency [Siek and Wadler, 2010], is defined and it relates the top-level type constructors only and does not require the subparts to be related. More precisely:

$$\star \smile T \qquad T \smile \star \qquad B \smile B \qquad T_1 \to T_2 \smile T_3 \to T_4 \quad T_1 \times T_2 \smile T_3 \times T_4 \qquad \mathsf{Ref}\ T_1 \smile \mathsf{Ref}\ T_2$$

Shallow consistency is needed to implement lazy error detection.

The first major difference between the monotonic machine and $\lambda^{\text{ref}}/\lambda_c^{\text{ref}}$ is that the latter use coercions to represent casts. Coercions are combinators on types, originally designed for compile-time optimization for dynamically typed languages [Henglein, 1994]. They were later used to guarantee space-efficiency at run-time for gradual typing [Herman et al., 2007, 2010]. Although

$$\begin{array}{llll}
\text{Coercions} & c, d \in \mathcal{C} & ::= & \iota \mid I! \mid I? \mid c \to d \mid c \times d \mid \mathsf{Ref}\ T \mid \bot^{I,J} \\
\text{Inert coercions} & c\uparrow & ::= & I! \mid c \to d \\
\text{Active coercions} & c\downarrow & ::= & \iota \mid I? \mid c \times d \mid \mathsf{Ref}\ T \mid \bot^{I,J}
\end{array}$$

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

$$\frac{}{\iota : T \Longrightarrow T} \qquad \frac{}{I! : I \Longrightarrow \star} \qquad \frac{}{I? : \star \Longrightarrow I} \qquad \frac{I \not\sim J}{\bot^{I,J} : I \Longrightarrow J}$$

$$\frac{c : T_1' \Longrightarrow T_1 \quad d : T_2 \Longrightarrow T_2'}{c \to d : T_1 \to T_2 \Longrightarrow T_1' \to T_2'} \qquad \frac{c : T_1 \Longrightarrow T_1' \quad d : T_2 \Longrightarrow T_2'}{c \times d : T_1 \times T_2 \Longrightarrow T_1' \times T_2'}$$

$$\frac{}{\mathsf{Ref}\ T' : \mathsf{Ref}\ T \Longrightarrow \mathsf{Ref}\ T'}$$

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

$$\begin{array}{ll}
(\star \Rightarrow \star) = (B \Rightarrow B) = \iota & (T_1 \to T_2 \Rightarrow T_1' \to T_2') = (T_1' \Rightarrow T_1) \to (T_2 \Rightarrow T_2') \\
(I \Rightarrow \star) = I! \qquad (\star \Rightarrow I) = I? & (T_1 \times T_2 \Rightarrow T_1' \times T_2') = (T_1 \Rightarrow T_1') \times (T_2 \Rightarrow T_2') \\
(I \Rightarrow J) = \bot^{I,J} \qquad \text{if } I \not\sim J & (\mathsf{Ref}\ T \Rightarrow \mathsf{Ref}\ T') = \mathsf{Ref}\ T'
\end{array}$$

FIGURE 1. Simple coercions and their operations.

λ^{ref} uses coercions to represent casts, it does not have the space-efficiency guarantee. λ_c^{ref} follows suit and does not guarantee space-efficiency either.

Figure 1 presents the definition of coercions and a function to create them. There are many different semantics for coercions [Siek et al., 2009]. λ_c^{ref} uses the Lazy-D semantics that allows any type, other than $\star$ itself, to be injected to $\star$. A cast $v\langle c\rangle$ applies a coercion c to the value v. The identity coercion ι acts as the identity function. The coercion $I!$ is an injection into $\star$ that tags v with the source type. The projection coercion $I?$ projects the tagged value v from $\star$ to I by checking if the tag is consistent with I. If this is the case, the underlying value is returned. The failure coercion $\bot^{I,J}$ signals an error when the source type I and the target type J are not consistent. The function coercion $c \to d$ creates a proxy around the function v that applies coercion c to the argument and coercion d to the return value. The pair coercion $c \times d$ applies coercion c to the first projection of the pair, and coercion d to the second projection. Finally, the reference coercion $\mathsf{Ref}\ T$ casts the value in the store to the greatest lower bound of T and the RTTI.

When applied to values, inert coercions, denoted $c\uparrow$, create new values and include the injection and function coercions. All other coercions are active coercions, denoted $c\downarrow$. Lemma 1 asserts that the two sets of coercions are mutually exclusive.

LEMMA 1 (Inert and active coercions). $\forall c : T \Longrightarrow T'$, $c\uparrow$ *or* $c\downarrow$

The syntax of the target language

$$\text{Expressions} \quad e, M, N \quad ::= \quad \cdots \mid e\langle c\rangle \mid a \mid \mathtt{ref}\ e@T \mid !e \mid !e@T \mid e := e \mid e := e@T$$

Typing rules for the target language $\boxed{\Sigma \mid \Gamma \vdash M : T}$

$$\vdots$$

$$\frac{\Gamma \vdash M : T \qquad c : T \Longrightarrow T'}{\Gamma \vdash M\langle c\rangle : T'}$$

$$\frac{\Sigma \mid \Gamma \vdash M : T}{\Sigma \mid \Gamma \vdash \mathtt{ref}\ M@T : \mathsf{Ref}\ T} \qquad \frac{\Sigma \mid \Gamma \vdash M : \mathsf{Ref}\ T \qquad \int T}{\Sigma \mid \Gamma \vdash !M : T}$$

$$\frac{\Sigma \mid \Gamma \vdash M : \mathsf{Ref}\ T \qquad \not\int T}{\Sigma \mid \Gamma \vdash !M@T : T} \qquad \frac{\Sigma \mid \Gamma \vdash M : \mathsf{Ref}\ T \qquad \Sigma \mid \Gamma \vdash N : T \qquad \int T}{\Sigma \mid \Gamma \vdash M := N : ()}$$

$$\frac{\Sigma \mid \Gamma \vdash M : \mathsf{Ref}\ T \qquad \Sigma \mid \Gamma \vdash N : T \qquad \not\int T}{\Sigma \mid \Gamma \vdash M := N@T : ()}$$

$$\textsc{WT-Addr}\ \frac{\Sigma(a) \sqsubseteq T}{\Sigma \mid \Gamma \vdash a : \mathsf{Ref}\ T}$$

Cast insertion $\boxed{\Sigma \mid \Gamma \vdash M \hookrightarrow N : T}$

$$\vdots$$

$$\textsc{CI-Allocation}\ \frac{\Sigma \mid \Gamma \vdash M \hookrightarrow N : T}{\Sigma \mid \Gamma \vdash \mathtt{ref}\ M \hookrightarrow \mathtt{ref}\ N@T : \mathsf{Ref}\ T}$$

$$\textsc{CI-SRead}\ \frac{\Sigma \mid \Gamma \vdash M \hookrightarrow N : \mathsf{Ref}\ T \qquad \int T}{\Sigma \mid \Gamma \vdash !M \hookrightarrow !N : T}$$

$$\textsc{CI-Read}\ \frac{\Sigma \mid \Gamma \vdash M \hookrightarrow N : \mathsf{Ref}\ T \qquad \not\int T}{\Sigma \mid \Gamma \vdash !M \hookrightarrow !N@T : T} \qquad \frac{\Sigma \mid \Gamma \vdash M \hookrightarrow N : \star \qquad c = (\star \Rightarrow \mathsf{Ref}\ \star)}{\Sigma \mid \Gamma \vdash !M \hookrightarrow !(N\langle c\rangle)@\star : \star}$$

$$\textsc{CI-SWrite}\ \frac{\Sigma \mid \Gamma \vdash M \hookrightarrow M' : \mathsf{Ref}\ T \qquad \Sigma \mid \Gamma \vdash N \hookrightarrow N' : T' \qquad \int T \qquad c = (T' \Rightarrow T)}{\Sigma \mid \Gamma \vdash M := N \hookrightarrow M' := (N'\langle c\rangle) : ()}$$

$$\textsc{CI-Write}\ \frac{\Sigma \mid \Gamma \vdash M \hookrightarrow M' : \mathsf{Ref}\ T \qquad \Sigma \mid \Gamma \vdash N \hookrightarrow N' : T' \qquad \not\int T \qquad c = (T' \Rightarrow T)}{\Sigma \mid \Gamma \vdash M := N \hookrightarrow M' := (N'\langle c\rangle)@T' : ()}$$

$$\frac{\Sigma \mid \Gamma \vdash M \hookrightarrow M' : \star \qquad \Sigma \mid \Gamma \vdash N \hookrightarrow N' : T \qquad c = (\star \Rightarrow \mathsf{Ref}\ \star) \qquad d = (T \Rightarrow \star)}{\Sigma \mid \Gamma \vdash M := N \hookrightarrow (M'\langle c\rangle) := (N'\langle d\rangle)@\star : ()}$$

FIGURE 2. λ_c^{ref} syntax and type system.

The syntax and the typing rules of the source language are the same as the ones in Figure 2 in Chapter 3. Figure 2 presents the syntax and the typing rules for the target language. The syntactic forms are similar to the ones for the abstract machine presented in Chapter 3, but they are not in

Runtime structures

Values	v	$::=$	$\cdots \mid v\langle c\uparrow\rangle \mid a$
Delayed Casts	dc	$::=$	$v \mid dc\langle c\rangle \mid \langle dc, dc\rangle$
Heap	μ	$::=$	$\emptyset \mid \mu(a \mapsto v : T)$
Evolving Heap	ν	$::=$	$\mu \mid \nu(a \mapsto dc : T)$
Frames	F	$::=$	$\cdots \mid \square\langle c\rangle \mid \texttt{ref}\ \square@T \mid !\square@T \mid !\square \mid \square := e@T \mid v := \square@T$
Evaluation Contexts	$\mathcal{E}$	$::=$	$\emptyset \mid \mathrm{F}, \mathcal{E}$

Typing rules for delayed casts $\boxed{\Sigma \mid \Gamma \vdash dc : T}$

$$\Gamma \vdash v : T \qquad \frac{\Gamma \vdash dc : T \quad c : T \Longrightarrow T'}{\Gamma \vdash dc\langle c\rangle : T'} \qquad \frac{\Gamma \vdash dc_1 : T_1 \quad \Gamma \vdash dc_2 : T_2}{\Gamma \vdash \langle dc_1, dc_2\rangle : T_1 \times T_2}$$

FIGURE 3. λ_c^{ref} runtime structures.

A-normal form. Furthermore, cast expressions, shaded in light gray, has changed to use coercions to represent casts. The `ref` $e@T$ form allocates a heap cell and initializes the RTTI field to the type T. The syntactic forms $!e$ and $e := e$ read from and write into a statically typed reference respectively. The typing rules for those forms require the type of the reference to satisfy the static type predicate $\S$. Dually, the syntactic forms $!e@T$ and $e := e@T$ are also for reading and writing but on partially typed references and their typing rules require the type of the reference to not satisfy the static type predicate, designated by the symbol $\not\S$. This requirement was not present in λ^{ref} nor in the monotonic machine.

Asking the programmers to use both forms when appropriate would be inconvenient. Fortunately, the compiler can automatically select which form to emit. The compilation from the source language to the target language is given by the type-directed cast insertion relation defined in Figure 2. It compiles to the right form of a reference operation based on the type of the reference. In particular, the CI-SRead and CI-SWrite rules check if the reference type is static to compile to the corresponding efficient form. Dually, the CI-Read and CI-Write rules check if the type of the reference is not static and compile to the forms that perform a mediating cast. The CI-Allocation rule compiles the allocation form to one that records the reference type.

2. Dynamic Semantics

Figure 3 presents the definition of the runtime structures. The definition of values is adjusted so that proxied functions and injected values now use inert coercions to represent casts on them. Delayed casts are defined to be either values, delayed casts with casts on them, or pairs of delayed

casts. This definition is different from the one for the monotonic machine in that casts are no longer restricted such that the target type is more precise than the source type (The restriction can be found in the typing rules for delayed casts in Figure 7 in Chapter 3). This restriction is lifted because the operational semantics for reducing delayed casts for λ_c^{ref} is defined in terms of a small-step reduction relation where intermediate redexes are also written to the heap and there are situations where cast expressions that violate this restriction will be written to the heap. Consider the following heap:

$$a \mapsto (\langle 0, 0\rangle\langle(\mathsf{Int} \times \mathsf{Int})!\rangle)\langle(\star \times \star)?\rangle$$

The address a is mapped to a cell that contains a delayed cast. The delayed cast is a projection on an injected value and it will reduce to $\langle 0, 0\rangle\langle \mathsf{Int}! \times \mathsf{Int}!\rangle$. The pair of zeros is being cast from a pair of integers to a pair of dynamic values and $\star \times \star \not\sqsubseteq \mathsf{Int} \times \mathsf{Int}$. To allow this expression to be written to the heap, the target type of the cast is not required to be more precise than the source type. This was not an issue for the monotonic machine because such an intermediate redex was fully reduced to a value first by the cast function and then only the result value was written to the heap.

Another difference is that, unlike the monotonic machine, λ_c^{ref} allows delayed casts to be nested. When casting a delayed cast, the monotonic machine throws away the old cast and writes the new cast to the heap. This is facilitated by using the type-based representation for casts, a convenient representation for inspecting and modifying the types involved in the cast. On the other hand, λ_c^{ref} uses coercions and it does not know, yet, how to compose adjacent coercions. However, the grammar for delayed casts in λ^{ref} does not allow nesting even though the cast reduction relation assumes that it is allowed. This is believed to be a typo in the grammar rule in λ^{ref}. Finally, λ_c^{ref} reduces casts on pairs to pairs of cast expressions that need to be written to the heap, so the grammar for delayed casts permits pairs of delayed casts. This is not needed by the monotonic machine, as explained before, because the machine reduces casts on pairs to values before writing the result value to the heap.

Figure 4 presents the reduction relations for casts. The relation $\longrightarrow_c$ applies coercions to values. Applying the identity coercion ι to a value reduces to the same value and applying the failure coercion reduces to `error`. Furthermore, reducing a sequence of injection and projection coercions reduces to a new cast expression where the new coercion is created from the types held by the injection and the projection. Note that this is different from the semantics of λ^{ref} as the latter

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

$$\begin{array}{rclrcl} v\langle\iota\rangle & \longrightarrow_c & v & v\langle\bot^{I,J}\rangle & \longrightarrow_c & \texttt{error} \\ v\langle I!\rangle\langle J?\rangle & \longrightarrow_c & v\langle I \Rightarrow J\rangle & \langle v_1, v_2\rangle\langle c \times d\rangle & \longrightarrow_c & \langle v_1\langle c\rangle, v_2\langle d\rangle\rangle \end{array}$$

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

$$\textsc{PureCast}\ \frac{M \longrightarrow_c N}{M, \nu \longrightarrow_c N, \nu}$$

$$\textsc{CastRef1}\ \frac{\nu(a) = dc : T_1 \quad T_1 \sim T_2 \quad T_1 \sqcap T_2 \neq T_1}{a\langle\texttt{Ref}\ T_2\rangle, \nu \longrightarrow_c a, \nu(a \mapsto dc\langle T_1 \Rightarrow (T_1 \sqcap T_2)\rangle : T_1 \sqcap T_2)}$$

$$\textsc{CastRef2}\ \frac{\nu(a)_{\text{rtti}} \sim T \quad \nu(a)_{\text{rtti}} \sqcap T = \nu(a)_{\text{rtti}}}{a\langle\texttt{Ref}\ T\rangle, \nu \longrightarrow_c a, \nu} \qquad \textsc{CastRef3}\ \frac{\nu(a)_{\text{rtti}} \not\sim T}{a\langle\texttt{Ref}\ T\rangle, \nu \longrightarrow_c \texttt{error}, \nu}$$

$$\textsc{Cong}_c\ \frac{M, \nu \longrightarrow_c N, \nu'}{\mathrm{F}[M], \nu \longrightarrow_c \mathrm{F}[N], \nu'} \qquad \textsc{CongErr}_c\ \frac{}{\mathcal{E}[\texttt{error}], \nu \longrightarrow_c \texttt{error}, \nu}$$

FIGURE 4. λ_c^{ref} casts dynamic semantics.

checks for consistency of the types and reduces to an error right away if they are not consistent. Finally, applying a pair coercion reduces to a pair of cast expressions.

When casting a reference, the heap cell pointed to by the reference is cast instead and the heap could change. The cast reduction relation $\longrightarrow_c$ relates cast expressions and heaps to new expressions and possibly updated heaps. When applying a reference cast, the greatest lower bound of the RTTI and the type carried by the coercion is computed using the meet function $\sqcap$. If the result is undefined, this means this cast wants to change the RTTI to a type that is not necessarily consistent with all references. A conservative approach is taken and the CASTREF3 rule reduces to `error`. Moreover, if the greatest lower bound type is defined but is the same as the RTTI, then there is no need to do anything and the CASTREF2 rule reduces to the same heap. Finally, if the greatest lower bound type is different than the RTTI, this means it is more precise than the RTTI, so the RTTI is updated to that new type and a delayed cast is written to the heap to that new type as well.

Figure 5 presents the program and state reduction rules. The pure program reduction relation $\longrightarrow_p$ is standard. The dynamic semantics for reference operations is specified by the reduction relation $\longrightarrow_r$. The ALLOC rule allocates a reference and initialize the RTTI field to the type annotation T. The semantics of the read and write syntactic forms on statically typed references is efficient and corresponds to that of loads and stores. The DYNREAD rule performs a mediating cast from the RTTI to the type of the reference T when reading from a partially typed one. Dually,

Pure reduction rules $\boxed{e \longrightarrow_p e}$

$$\begin{array}{lcl} \texttt{fst}\ \langle v_1, v_2\rangle & \longrightarrow_p & v_1 \\ \texttt{snd}\ \langle v_1, v_2\rangle & \longrightarrow_p & v_2 \end{array} \qquad \begin{array}{lcl} (\lambda(x:T).\,M)\ v & \longrightarrow_p & [x := v]M \\ (v_1\langle c \to d\rangle)\ v_2 & \longrightarrow_p & (v_1\ (v_2\langle c\rangle))\langle d\rangle \end{array}$$

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

(Alloc) $\quad \texttt{ref}\ v@T, \mu \longrightarrow_r a, \mu(a \mapsto v : T) \quad \text{if } a \notin \text{dom}(\mu)$

(Read) $\quad !a, \mu \longrightarrow_r \mu(a)_{\text{val}}, \mu$

(DynRead) $\quad !a@T, \mu \longrightarrow_r \mu(a)_{\text{val}}\langle \mu(a)_{\text{rtti}} \Rightarrow T\rangle, \mu$

(Write) $\quad a := v, \mu \longrightarrow_r \text{Unit}, \mu(a \mapsto v : \mu(a)_{\text{rtti}})$

(DynWrite) $\quad a := v@T, \mu \longrightarrow_r \text{Unit}, \mu(a \mapsto v\langle T \Rightarrow \mu(a)_{\text{rtti}}\rangle : \mu(a)_{\text{rtti}})$

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

$$\text{Pure}\ \frac{M \longrightarrow_p N}{M, \mu \longrightarrow N, \mu} \qquad \text{Mono}\ \frac{M, \mu \longrightarrow_r N, \nu}{M, \mu \longrightarrow N, \nu}$$

$$\text{Cong}\ \frac{M, \nu \longrightarrow N, \nu'}{\text{F}[M], \mu \longrightarrow \text{F}[N], \nu} \qquad \text{CongErr}\ \frac{}{\mathcal{E}[\texttt{error}], \mu \longrightarrow \texttt{error}, \nu}$$

$$\text{Cast}\ \frac{e, \mu \longrightarrow_c e', \nu}{e, \mu \longrightarrow e', \nu} \qquad \text{Error}\ \frac{\nu(a) = dc : T \quad dc, \nu \longrightarrow_c \mathcal{E}[\texttt{error}], \nu'}{e, \nu \longrightarrow \texttt{error}, \nu'}$$

$$\text{NoRTTIChange}\ \frac{\nu(a) = dc : T \quad dc, \nu \longrightarrow_c dc', \nu' \quad \nu'(a)_{\text{rtti}} = T}{e, \nu \longrightarrow e, \nu'(a \mapsto dc' : T)}$$

$$\text{RTTIChanged}\ \frac{\nu(a) = dc : T \quad dc, \nu \longrightarrow_c dc', \nu' \quad \nu'(a)_{\text{rtti}} \neq T}{e, \nu \longrightarrow e, \nu'}$$

Figure 5. λ_c^{ref} dynamic semantics.

The DynWrite rule writes into a partially typed reference a delayed cast casting the written value from the type of the reference to the RTTI.

In addition to program reduction relations, Figure 5 presents state reduction rules that reduce delayed casts on the heap until they become values before continuing reducing the program. Delayed casts are reduced using the cast reduction relation $\longrightarrow_c$. A delayed cast could reduce to an expression that has an inner `error` using a congruence rule. Such expression is not a delayed cast (check the grammar) and can not be written to the heap. If this is the case, the Error rule reduces to `error`. Note that the state reduction relation in λ^{ref} has a rule to handle reduction to an error expression but does not have one to handle reduction to expressions that have an inner `error`. This is believed to be a design error in λ^{ref}.

If the delayed cast reduces to another delayed cast dc', dc' will be written to the heap if the RTTI did not change, via the NoRTTIChange rule. If the RTTI has changed, this means another delayed cast has already been written to the heap and dc' is dated. In this case, dc' is discarded via the RTTIChanged rule.

3. Type Safety

λ^{ref} did not have a full type safety proof, and the paper [Siek et al., 2015c] present high points only and referred to the mechanized type safety proof of the machine [Siek and Vitousek, 2013]. In this section, a full type safety proof that is mechanized in Agda is presented.

3.1. Definitions and Lemmas Regarding The Static Semantics. In this section, lemmas are presented regarding expression and heap typing. These lemmas hold even for other semantics presented later in the dissertation.

The heap typing can move along two orthogonal dimensions, it could become either more precise (Definition 1) or its domain can grow (Definition 2)[1]. Combining both relations results in one, $\sqsubseteq_{p/e}$, that fully captures how the heap typing changes after each step, called Heap Typing Progress (Definition 3).

Definition 1 (Precision relation on heap typings). *$\Sigma' \sqsubseteq_p \Sigma$ iff $dom(\Sigma') = dom(\Sigma)$ and $\Sigma(a) = T$ implies $\Sigma'(a) = T'$ where $T' \sqsubseteq T$.*

Definition 2 (Extension relation on heap typings). *$\Sigma \sqsubseteq_e \Sigma'$ iff $dom(\Sigma) \subseteq dom(\Sigma')$ and $\Sigma(a) = \Sigma'(a)$.*

Definition 3 (Progress relation for heap typings). *$\Sigma' \sqsubseteq_{p/e} \Sigma$ if $\Sigma' \sqsubseteq_p \Sigma$ or $\Sigma \sqsubseteq_e \Sigma'$.*

Lemma 2 asserts that expression typing is preserved when moving to a more precise heap typing. Similarly, Lemma 3 asserts that expression typing is also preserved when moving to a larger heap typing. Corollary 3.1 combines both lemmas and asserts that expression typing is preserved when the heap typing changes with respect to the $\sqsubseteq_{p/e}$ relation.

Lemma 2 (Strengthening wrt. the heap typing precision). *If $\Sigma \mid \Gamma \vdash e : T$ and $\Sigma' \sqsubseteq_p \Sigma$, then $\Sigma' \mid \Gamma \vdash e : T$.*

[1]Garbage collection is ignored and heaps can not become smaller

Proof. For the case of addresses: From $\Sigma' \sqsubseteq_p \Sigma$ and the WT-addr rule and transitivity of $\sqsubseteq$, we have $\Sigma'(a) \sqsubseteq T$. Therefore $\Sigma' \mid \Gamma \vdash a : T$. □

Lemma 3 (Weakening wrt. the heap typing extension). *If* $\Sigma \mid \Gamma \vdash e : T$ *and* $\Sigma \sqsubseteq_e \Sigma'$, *then* $\Sigma' \mid \Gamma \vdash e : T$.

Proof. For the case of addresses: we have $\Sigma'(a) \sqsubseteq T$ from $\Sigma \sqsubseteq_e \Sigma$ and the WT-addr rule. Therefore $\Sigma' \mid \Gamma \vdash a : T$. □

Corollary 3.1 (Weakening wrt. the heap typing progress). *If* $\Sigma \mid \Gamma \vdash e : T$ *and* $\Sigma' \sqsubseteq_{p/e} \Sigma$, *then* $\Sigma' \mid \Gamma \vdash e : T$.

Proof. By applying Lemmas 2 and 3 in the cases of $\sqsubseteq_p$ and $\sqsubseteq_e$ respectively. □

3.2. Lemmas Regarding The Dynamic Semantics. Heaps are well-typed when the delayed casts inside respect heap typing.

Definition 4 (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* $\exists\ dc$ *s.t.* $\Sigma \mid \emptyset \vdash dc : T$ *and* $\nu(a) = dc : T$.

The following corollary follows from Lemma 2.

Corollary 3.2 (Heap cast). *If* $\Sigma \vdash \nu$ *and* $\nu(a) = dc : T$ *and* $T' \sqsubseteq T$, *then* a *can be cast to* $\mathsf{Ref}\ T'$ *such that* $\Sigma(a \mapsto T') \vdash \nu(a \mapsto (dc\langle T \Rightarrow T' \rangle) : T')$.

Proof. Let $\Sigma' = \Sigma(a \mapsto T')$ and $dc' = dc\langle T \Rightarrow T' \rangle$. From $T' \sqsubseteq T$ and $\Sigma \vdash \nu$ we have $T' \sqsubseteq \Sigma(a)$ which implies $\Sigma' \sqsubseteq_p \Sigma$. Next, Lemma 2 is applied to all delayed casts in the heap and to dc', so we get $\Sigma' \mid \emptyset \vdash dc' : T'$, thus $\Sigma' \vdash \nu(a \mapsto dc' : T')$. □

Similarly, the following corollary follows from Lemma 3.

Corollary 3.3 (Heap extension). *If* $\Sigma \vdash \mu$ *and* $\Sigma \mid \emptyset \vdash v : T$ *and* $a \notin dom(\Sigma)$, *then* v *can be added at* a *such that* $\Sigma(a \mapsto T) \vdash \mu(a \mapsto v : T)$.

Proof. Let $\Sigma' = \Sigma(a \mapsto T)$. We have $\Sigma \sqsubseteq_e \Sigma'$ from $a \notin \text{dom}(\Sigma)$. Next, Lemma 3 is applied to all values in the heap and to v to get $\Sigma' \mid \emptyset \vdash v : T$. We add v to the heap at address a, thus $\Sigma' \vdash \mu(a \mapsto v : T)$. □

Lemma 1 shows that coercions are either inert or active. Active coercions do not create values, so it better be the case that they can take a step. Lemma 4 asserts this fact:

LEMMA 4 (Active cast progress). *If* $\Sigma \vdash \nu$ *and* $\Sigma \mid \emptyset \vdash v : T$ *and* $c : T \Longrightarrow T'$ *and* $c\downarrow$*, then* $\exists\ e, \nu'$ *s.t.* $v\langle c\rangle, \nu \longrightarrow_c e, \nu'$

PROOF. By case analysis on $c\downarrow$. In the case of monotonic reference coercion, v must be some address a, and $v\langle \texttt{Ref}\ T'\rangle$ takes a step via $\longrightarrow_c$. In particular:

CastRef1: if $T' \sim \nu(a)_{\mathrm{rtti}}$ and $T' \sqcap \nu(a)_{\mathrm{rtti}} \neq \nu(a)_{\mathrm{rtti}}$.

CastRef2: if $T' \sim \nu(a)_{\mathrm{rtti}}$ and $T' \sqcap \nu(a)_{\mathrm{rtti}} = \nu(a)_{\mathrm{rtti}}$.

CastRef3: if $T' \nsim \nu(a)_{\mathrm{rtti}}$.

□

The following lemma proves that a delayed cast can take step if it is not a value.

LEMMA 5 (Reducible delayed cast progress). *If* $\Sigma \vdash \nu$ *and* $\Sigma \mid \emptyset \vdash dc : T$ *and* dc *is not a* value*, then* $\exists\ e, \nu'$ *s.t.* $dc, \nu \longrightarrow_c e, \nu'$

PROOF. By case analysis on dc.

Case v**:** Contradiction by the premise that dc is not a value.

Case $dc'\langle c\rangle$**:** By cases on whether dc' is a value

Case *yes***:** We know that dc is not a value, so it must be the case that c is neither an injection or a function coercion, so Lemma 4 is applied.

Case *no***:** The induction hypothesis is applied to dc and a step is taken via $\textsc{Cong}_c$.

Case $\langle dc_1, dc_2\rangle$**:** It must be the case that either or both projections are not values, so the induction hypothesis is applied to first projection that is not a value and a step is taken via $\textsc{Cong}_c$.

□

However this is not true for arbitrary cast expressions because inert coercions form values.

COROLLARY 5.1 (Cast progress). *If* $\Sigma \vdash \nu$ *and* $\Sigma \mid \emptyset \vdash v : T$ *and* $c : T \Longrightarrow T'$*, then either*

(1) $v\langle c\rangle$ *is a value, or*

(2) $\exists\ e,\ \nu'$ *s.t.* $v\langle c\rangle, \nu \longrightarrow_c e, \nu'$

PROOF. By Lemma 1, c is either $c\uparrow$ or $c\downarrow$. In the case of the former, we conclude that $v\langle c\rangle$ is a value. Otherwise, Lemma 4 is applied. □

If a delayed cast took a reduction step, the result expression should be another delayed cast or an erroneous expression.

LEMMA 6 (Reducible delayed cast preservation). *If* $\Sigma \vdash \nu$ *and* $\Sigma \mid \emptyset \vdash dc : T$ *and* $\Sigma' \mid \emptyset \vdash e : T$ *and* $\Sigma' \vdash \nu'$ *and* $dc, \nu \longrightarrow_c e, \nu'$*, then either*

(1) e *is a delayed cast, or*

(2) $e = \mathcal{E}[\texttt{error}]$

Progress holds when the heap contains at least one delayed cast that is not a value, as follows:

LEMMA 7 (Progress with evolving heap). *If* $\Sigma \vdash \nu$ *and* ν *is not a normal heap and* $\Sigma \mid \emptyset \vdash M : T$*, then* $\exists\ \nu'$ *s.t.* $M, \nu \longrightarrow M, \nu'$

PROOF. Because ν is not normal, then $\exists\ a, dc\ s.t.\ \nu(a)_{\mathrm{val}} = dc$ and dc is not a *value*. By Lemma 5, dc takes a step to an expression e. By Lemma 6, there are two cases:

Case $e = \mathcal{E}[\texttt{error}]$**:** : A step is taken via ERROR.

Case e **is a delayed cast:** : By cases on whether $\nu(a)_{\mathrm{rtti}} = \nu'(a)_{\mathrm{rtti}}$

Case $\nu(a)_{\mathbf{rtti}} = \nu'(a)_{\mathbf{rtti}}$**:** A step is taken via NoRTTIChange

Case $\nu(a)_{\mathbf{rtti}} \neq \nu'(a)_{\mathbf{rtti}}$**:** A step is taken via RTTIChanged

□

Progress in the case of a normal heap is standard.

LEMMA 8 (Progress with normal heap). *If* $\Sigma \vdash \mu$ *and* $\Sigma \mid \emptyset \vdash M : T$*, then either*

(1) M *is a value, or*

(2) $M = \texttt{error}$*, or*

(3) $\exists\ N, \nu$ *s.t.* $M, \mu \longrightarrow N, \nu$

PROOF. By Lemma 5.1 and standard induction on the typing derivation. □

From Lemmas 7 and 8, the full proof of progress is assembled by cases on whether the heap is in a normal state:

COROLLARY 8.1 (Progress). *If* $\Sigma \vdash \nu$ *and* $\Sigma \mid \emptyset \vdash M : T$*, then either*

(1) M *is a value, or*

(2) $M = \texttt{error}$*, or*

(3) $\exists\ N, \nu'$ *s.t.* $M, \nu \longrightarrow N, \nu'$

Lemma 9 (Type preservation). *If* $\Sigma \vdash \nu$ *and* $\Sigma \mid \emptyset \vdash M : T$ *and* $M, \nu \longrightarrow N, \nu'$*, then* $\exists\ \Sigma'$ *s.t.* $\Sigma' \mid \emptyset \vdash N : T$ *and* $\Sigma' \vdash \nu'$ *and* $\Sigma' \sqsubseteq_{p/e} \Sigma$

Type safety follows from Corollary 8 and Lemma 9.

Theorem 10 (Type safety). *If* $\Sigma \vdash \nu$ *and* $\Sigma \mid \emptyset \vdash e : T$*, then either*

(1) e *diverges, or*

(2) $e = \texttt{error}$*, or*

(3) $\exists\ v, \Sigma', \nu'$ *s.t.* $e, \nu \longrightarrow^* v, \nu'$ *and* $\Sigma' \mid \emptyset \vdash v : T$ *and* $\Sigma' \vdash \nu'$

4. Summary

λ_c^{ref} is a reduction semantics for monotonic references that uses coercions, and is a variant of λ^{ref} [Siek et al., 2015c]. λ_c^{ref} fixes the following minor problems in λ^{ref}:

- The grammar for casted values[2] does not allow casts on other casted values. However, the rule CastRef1 in the cast reduction relation reads a casted value from the heap and then wraps it in another cast expression that is written to the heap. λ_c^{ref} fixes this issue by changing the grammar of delayed casts to allow casts on delayed casts (Figure 3).
- A casted value could reduce to an expression that has an inner error using a congruence rule. Such expression is not a casted value and can not be written to the heap. The state reduction relation has a rule to handle reduction to an error expression but does not have one to handle reduction to expressions that have an inner error. λ_c^{ref} fixes this issue by updating the Error rule to make sure the result delayed cast could be decomposed into a context and an error expression.
- The type preservation lemma, the second item in Lemma 2, relates the old heap typing and the new heap typing with the precision relation on heap typing (Definition 1). However, this does not account for the fact that the heap could grow by creating new references. λ_c^{ref} fixes this issue by recognizing that the heap typing could move along two orthogonal dimensions, it could become either more precise (Definition 1) or larger (Definition 2). The type preservation lemma is updated accordingly (Lemma 9).

[2] λ_c^{ref} uses the term *delayed casts* instead to avoid confusion with values that have casts on them

Furthermore, a full type-safety proof mechanized in Agda was presented.

CHAPTER 5

Practical And Space-Efficient Monotonic References

Delayed casts play an important role in the operational semantics of both the monotonic machine and λ_c^{ref}, namely to ensure the correct reduction of casts on cyclic values in the heap. However, delayed casts are essentially expressions that are written to the heap. In λ_c^{ref}, those expressions are reduced using a small-step reduction relation. It is not obvious how to implement this reduction-in-the-heap in a compiled implementation of a gradually typed language. A practical semantics should write values only to the heap.

Furthermore, although λ_c^{ref} uses coercions to represent casts, it does not guarantee space-efficiency. Space-efficiency is an important property to have in gradually typed programming languages to ensure $O(1)$ access to proxied values. Although monotonic references does not create proxied addresses, other higher-order features in the language create proxies such as functions.

This chapter presents a new reduction semantics for monotonic references that writes values only to the heap. In this design, a cast function is provided to cast values on the heap and it returns values and a list of casts on addresses that are put into a queue for subsequent processing, making the semantics straightforward to implement. Furthermore, this design is refined further to add space-efficiency to functions.

The chapter is organized as follows: Section 1 illustrates why λ_c^{ref} writes cast expressions to the heap. Sections 2 and 3 present our approach and prove type safety. Finally, Section 4 presents the space-efficient version and Section 6 proves bounds on space consumption.

1. The Problem of Expressions on the Heap

It was discussed in detail in Section 4.3 in Chapter 3 why the monotonic machines writes delayed casts to the heap. This section reviews how λ_c^{ref} puts expressions on the heap that are reduced-in-the-heap using a small-step reduction relation and discusses why this process is not convenient to implement in a runtime system. Consider the example listed in Figure 11 in Chapter 3 that creates a cycle in the heap.

Recall that, on line 6, function g is applied to r, but g expects a reference to a pair of different types.

A cast will be performed and the address a will be updated to

$$a \mapsto \left\langle \begin{array}{l} (\lambda(x:\star).\, x)\langle \star \to \star \Rightarrow \star \rangle\langle \star \Rightarrow (\star \to \mathsf{Int})\rangle, \\ a\langle \mathsf{Ref}\ (\star \times \star) \Rightarrow \star \rangle\langle \star \Rightarrow \mathsf{Ref}\ ((\mathsf{Int} \to \star) \times \star)\rangle \end{array} \right\rangle$$

The cast on the first element of the pair proceeds without further ado.

$$a \mapsto \left\langle \begin{array}{l} (\lambda(x:\star).\, x)\langle \star \to \star \Rightarrow \star \to \mathsf{Int}\rangle, \\ a\langle \mathsf{Ref}\ (\star \times \star) \Rightarrow \star \rangle\langle \star \Rightarrow \mathsf{Ref}\ ((\mathsf{Int} \to \star) \times \star)\rangle \end{array} \right\rangle$$

The cast on the second element of the pair is a cast to a reference type, the address a will be cast to the greatest lower bound of the target type of the cast and the current RTTI, which amounts to:

$$(\mathsf{Int} \to \mathsf{Int}) \times (\mathsf{Ref}\ ((\mathsf{Int} \to \star) \times \star))$$

This cast causes another cast on the first element of the pair. Proceeding with a reduction step yields the following state.

$$a \mapsto \left\langle \begin{array}{l} (\lambda(x:\star).\, x)\langle \star \to \star \Rightarrow \star \to \mathsf{Int}\rangle\langle \star \to \mathsf{Int} \Rightarrow \mathsf{Int} \to \mathsf{Int}\rangle, \\ a \end{array} \right\rangle$$

In the above sequence of reductions, it was crucial that the cast on the first element of the pair to $\star \to \mathsf{Int}$ was written to the heap before the first element was cast again to the type $\mathsf{Int} \to \star$, enabling a greatest lower bound of $\mathsf{Int} \to \mathsf{Int}$ that took both types into account.

Unfortunately, it is not obvious how to implement this process efficiently in a compiler and runtime system. Literally placing expressions in the heap would require a runtime representation of abstract syntax trees and an interpreter, which would be costly to implement and rather slow to execute.

Alternatively, one might try to cast the value in a heap cell in one big step before writing it back to the heap, but the straightforward version of this idea is problematic because of race conditions. In the above example, the second cast on the first element, to $\mathsf{Int} \to \star$, would see the current type as still $\star \to \star$, so the greatest lower bound would be $\mathsf{Int} \to \star$. This would break the monotonicity property (and therefore type safety), because $\mathsf{Int} \to \star$ is not less precise than $\star \to \mathsf{Int}$. In the rest of the chapter a refinement of the big-step approach is presented. The key insight is that in the process of casting the value in a heap cell, in the case of an address, instead of immediately casting

it, the address is left as-is and instead the address and the target type of the cast are put in a queue. Once the entire value has been processed, the result value is written back to the heap and then the addresses in the queue gets cast.

2. λ_p^{ref}: Practical Dynamic Semantics For Monotonic References

In this section, a new dynamic semantics for monotonic references is presented, named λ_p^{ref}. Instead of storing expressions on the heap, λ_p^{ref} uses a simple recursive function to cast values in the heap down to values. To maintain the correctness of reference casts in the presence of cycles, the function does not apply any inner reference casts. Instead it returns the address and puts the inner reference cast in a queue. Reduction proceeds by examining the queue, so a suspended cast is picked from the queue and gets applied. Applying a cast can return more suspended casts, so those get appended to the queue. This process continues until the queue is empty.

The main insight behind this design is that the correct ordering of cast reduction is a top-down breadth first traversal, as discussed in Section 1. With this semantics, the example in Section 1 can be reduced correctly without writing expressions to the heap. Recall that the value at address a is allocated at type $\star \times \star$ and is cast to

$$(\star \to \mathsf{Int}) \times (\mathsf{Ref}\ ((\mathsf{Int} \to \star) \times \star))$$

Because the top-level cast is a reference cast, it will be pushed to the queue. Subsequently, the cast will be read from the queue and applied to the pair in the heap. The cast on the first projection will be $\star \Rightarrow \star \to \mathsf{Int}$ and will fully reduce to a value. The cast on the second projection will be suspended by reducing to the address a and the cast itself will be pushed to the queue. So this value will be written to the heap as follows:

$$\{a \mapsto \langle(\lambda(x:\star).\, x)\langle\star \to \star \Rightarrow \star \to \mathsf{Int}\rangle, a\rangle : (\star \to \mathsf{Int}) \times \mathsf{Ref}\ ((\mathsf{Int} \to \star) \times \star)\}$$

and the queue will contain the address a with the target type $(\mathsf{Int} \to \star) \times \star$. Subsequently, the address and its target type will be popped from the queue. The value at the address is cast from its RTTI to the greatest lower bound of the RTTI and the target type. This results in the following final state:

$$\{a \mapsto \langle(\lambda(x:\star).\, x)\langle\star \to \star \Rightarrow \star \to \mathsf{Int} \Rightarrow \mathsf{Int} \to \mathsf{Int}\rangle, a\rangle : (\mathsf{Int} \to \mathsf{Int}) \times \mathsf{Ref}\ ((\mathsf{Int} \to \star) \times \star)\}$$

Runtime structures

Values	$v \in \mathcal{V}$	$::=$	$\cdots \mid v\langle c \uparrow\rangle \mid a$
Heap	μ	$::=$	$\emptyset \mid \mu(a \mapsto v : T)$
Suspended casts	Ψ	$::=$	$\emptyset \mid \Psi; (a, T)$
Frames	F	$::=$	$\cdots \mid \square\langle c\rangle \mid \texttt{ref}\ \square@T \mid !\square@T \mid !\square \mid \square := e@T \mid v := \square@T$

Well-formedness of suspended casts $\boxed{\Sigma \vdash \Psi}$

$$\frac{\forall (a, T) \in \Psi \implies \exists T'.\ s.t.\ (a, T') \in \Sigma}{\Sigma \vdash \Psi}$$

FIGURE 1. λ_p^{ref} runtime structures.

Similar to λ_c^{ref}, λ_p^{ref} uses coercions to represent casts and does not address the problem of space-efficiency yet. The syntax and typing rules of coercions are the same as the ones for λ_c^{ref} (in Figure 1 in Chapter 4).

Figure 1 defined λ_p^{ref} runtime structures. Values and frames in λ_p^{ref} are defined in the same way they are defined in λ_c^{ref}. Furthermore, delayed casts are no longer needed because heaps can only store values. The queue of suspended casts Ψ is defined as a sequence of casts where each cast is a pair of an address and a type.

Figure 2 presents the apply-cast function which applies a coercion to a value. In the case of a reference coercion, apply-cast returns the input address as is along with a cast that has that input address and the type held by the coercion. This cast will be processed later by the state reduction rules. Furthermore, in the case of a pair coercion, apply-cast is called on both projections of the pair and the result queue from the call on the right projection is appended to the result queue from the call on left projection. Moreover, in the cases for identity, function, injection, and projection coercions, apply-cast returns an empty queue and the evaluation is standard (corresponds to the reduction relation $\longrightarrow_c$ for λ_c^{ref} defined in Figure 4 in Chapter 4). Finally, all other cases, including the failure coercion, evaluate to `error`.

Figure 2 also presents the program reduction relation $\longrightarrow_e$. The ALLOC, READ, DYNREAD, and WRITE rules are similar to the corresponding ones in λ_c^{ref} (defined in Figure 5 in Chapter 4). On the other hand, the DYNWRITE rule in λ_c^{ref} creates a delayed cast and writes it to the heap. To get rid of delayed casts, the new DYNWRITE rule casts the written value using the apply-cast function and writes the result value right away to the heap. The queue that apply-cast returns is now part of the configuration that the relation $\longrightarrow_e$ reduces to. If the call to apply-cast evaluates to `error`, DYNWRITEFAIL reduces to `error`.

$$\text{Cast result} \quad \mathsf{r} \in \mathcal{R} \quad ::= \quad (\Psi, v) \mid \texttt{error}$$

Cast application function $\boxed{\text{apply-cast} : \mathcal{V} \times \mathcal{C} \to \mathcal{R}}$

$$\begin{aligned}
\text{apply-cast}(v, c\uparrow) &= (\emptyset, v\langle c\uparrow\rangle) \\
\text{apply-cast}(a, \mathsf{Ref}\ T) &= (((a, T); \emptyset), a) \\
\text{apply-cast}(v, \iota) &= (\emptyset, v) \\
\text{apply-cast}((v\langle I!\rangle), J?) &= \text{apply-cast}(v, (I \Rightarrow J)) \\
\text{apply-cast}(\langle v_1, v_2\rangle, c \times d) &= (\Psi \oplus \Psi', \langle v_1', v_2'\rangle) \\
& \quad \text{if } \begin{array}{l} \text{apply-cast}(v_1, c) = (\Psi, v_1'), \\ \text{apply-cast}(v_2, d) = (\Psi', v_2') \end{array} \\
\text{apply-cast}(v, c) &= \texttt{error}
\end{aligned}$$

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

(Alloc) $\texttt{ref}\ v@T, \mu \longrightarrow_e a, \mu(a \mapsto v : T), \emptyset \quad \text{if } a \notin \text{dom}(\mu)$

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

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

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

(DynWrite)

$$a := v@T, \mu \longrightarrow_e \text{Unit}, \mu(a \mapsto v' : \mu(a)_{\text{rtti}}), \Psi \quad \text{if apply-cast}(v, (T \Rightarrow \mu(a)_{\text{rtti}})) = (\Psi, v')$$

(DynWriteFail)

$$a := v@T, \mu \longrightarrow_e \texttt{error}, \mu, \emptyset \quad \text{if apply-cast}(v, (T \Rightarrow \mu(a)_{\text{rtti}})) = \texttt{error}$$

$$\frac{M \longrightarrow_p N}{M, \mu \longrightarrow_e N, \mu, \emptyset} \qquad \frac{M, \mu \longrightarrow_e N, \mu', \Psi}{F[M], \mu \longrightarrow_e F[N], \mu', \Psi} \qquad \frac{}{F[\texttt{error}], \mu \longrightarrow_e \texttt{error}, \mu, \emptyset}$$

$$\textsc{Cast/Succeed}\ \frac{\text{apply-cast}(v, c) = (\Psi, v')}{v\langle c\rangle, \mu \longrightarrow_e v', \mu, \Psi} \qquad \textsc{Cast/fail}\ \frac{\text{apply-cast}(v, c) = \texttt{error}}{v\langle c\rangle, \mu \longrightarrow_e \texttt{error}, \mu, \emptyset}$$

State reduction rules $\boxed{e, \mu, \Psi \longrightarrow e, \mu, \Psi}$

$$\textsc{ProgReduce}\ \frac{e, \mu \longrightarrow_e e', \mu, \Psi}{e, \mu, \emptyset \longrightarrow e', \mu, \Psi} \qquad \textsc{UpdateHeap}\ \frac{\begin{array}{c} T' = T \sqcap \mu(a)_{\text{rtti}} \quad T' \neq \mu(a)_{\text{rtti}} \\ \text{apply-cast}(\mu(a)_{\text{val}}, (\mu(a)_{\text{rtti}} \Rightarrow T')) = (\Psi', v') \end{array}}{e, \mu, (a, T); \Psi \longrightarrow e, \mu(a \mapsto v' : T'), \Psi \oplus \Psi'}$$

$$\textsc{NoChange}\ \frac{T \sqcap \mu(a)_{\text{rtti}} = \mu(a)_{\text{rtti}}}{e, \mu, (a, T); \Psi \longrightarrow e, \mu, \Psi} \qquad \textsc{Error1}\ \frac{\begin{array}{c} T' = T \sqcap \mu(a)_{\text{rtti}} \quad T' \neq \mu(a)_{\text{rtti}} \\ \text{apply-cast}(\mu(a)_{\text{val}}, (\mu(a)_{\text{rtti}} \Rightarrow T')) = \texttt{error} \end{array}}{e, \mu, (a, T); \Psi \longrightarrow \texttt{error}, \mu, \Psi}$$

$$\textsc{Error2}\ \frac{T \nsim \mu(a)_{\text{rtti}}}{e, \mu, (a, T); \Psi \longrightarrow \texttt{error}, \mu, \Psi}$$

Figure 2. λ_p^{ref} dynamic semantics.

Moreover, the reduction relation for pure expressions $\longrightarrow_p$ is the same as the one defined for λ_c^{ref} in Figure 5 in Chapter 4. Furthermore, cast reduction rules Cast/Succeed and Cast/Fail are defined via calls to the apply-cast function. Finally, the congruence rule and the error propagation rule are standard.

The state reduction relation $\longrightarrow$, defined in Figure 2, pauses expression reduction if the queue is not empty and pops a suspended cast to reduce. For the suspended cast (a, T), the value stored at address a gets cast from $\mu(a)_{\text{rtti}}$ to $\mu(a)_{\text{rtti}} \sqcap T$ using the apply-cast function. apply-cast evaluates the cast down to a value and suspends any inner reference casts, returning another queue of suspended casts. The result value gets written to the heap and the new queue gets appended to the old one. This process continues until there are no more suspended casts in the queue. Now the queue is empty, the expression at hand can resume reduction. This way, only values are written to the heap. Note that the length of the queue is bounded by the size of the value being cast. This means that the queue cannot grow without bound, so the space overhead caused by the queue is always a constant factor. This new approach is easier to understand and is straightforward to implement in a runtime system for gradually typed languages. A step is taken via rule UpdateHeap if the suspended cast (a, T) is productive, i.e. $T \sqcap \mu(a)_{\text{rtti}} \neq \mu(a)_{\text{rtti}}$, and successful. The result value of the call to apply-cast gets written to the heap and the RTTI is updated accordingly. On the other hand, if the cast fails, the Error1 rule reduces to `error`. Moreover, if the cast is not productive, i.e. $T \sqcap \mu(a)_{\text{rtti}} = \mu(a)_{\text{rtti}}$, the NoChange rule drops it from the queue. Finally, if the target type of the suspended cast is not consistent with the RTTI, the Error2 rule reduces to `error`.

Example Now let's consider how the example program in Section 4.3 in Chapter 3 reduces with the dynamic semantics of λ_p^{ref}. The cast expression we want to reduce is:

$$r\langle \mathsf{Ref}\ ((\star \to \mathsf{Int}) \times (\mathsf{Ref}\ ((\mathsf{Int} \to \star) \times \star)))\rangle$$

and the heap is:

$$a \mapsto \langle f\langle (\star \to \star)! \rangle, a\langle (\mathsf{Ref}\ (\star \times \star))! \rangle\rangle : \star \times \star$$

Reduction will proceed via the Cast/Succeed rule reducing the expression to the address a and putting the cast in the queue. The reduced-to configuration is:

$$\mu = \boxed{a \mapsto \langle f\langle (\star \to \star)! \rangle, a\langle (\mathsf{Ref}\ (\star \times \star))! \rangle\rangle : \star \times \star}$$

$$\Psi = \boxed{\emptyset; (a, ((\star \to \mathsf{Int}) \times (\mathsf{Ref}\ ((\mathsf{Int} \to \star) \times \star))))}$$

Next, reduction proceeds via the UpdateHeap rule and the cast will get dequeued and then applied to the value in the heap. The result new value and the new RTTI will be written to the heap, and the inner reference cast will be suspended. The new configuration is:

$$\mu = \boxed{a \mapsto \langle f\langle(\iota \to (\mathsf{Int?}))\rangle, a\rangle : (\star \to \mathsf{Int}) \times (\mathsf{Ref}\ ((\mathsf{Int} \to \star) \times \star))}$$
$$\Psi = \boxed{\emptyset; (a, (\mathsf{Int} \to \star) \times \star)}$$

Finally, reduction will apply UpdateHeap to perform the cast in the queue. The final configuration will be:

$$\mu = \boxed{a \mapsto \langle f\langle(\iota \to (\mathsf{Int?}))\rangle\langle((\mathsf{Int!}) \to \iota)\rangle, a\rangle : (\mathsf{Int} \to \mathsf{Int}) \times (\mathsf{Ref}\ ((\mathsf{Int} \to \mathsf{Int}) \times \star))}$$
$$\Psi = \boxed{\emptyset}$$

Now the queue is empty, expression reduction will resume and the program will evaluate to 42.

It has been discussed throughout the section the relationship between λ_p^{ref} and λ_c^{ref}. However, the relationship between λ_p^{ref} and the monotonic machine is also interesting because both of them use a cast application function that returns values. However, the function in the monotonic machine writes the cast expression as a delayed cast to the heap, similar to what λ_c^{ref} does. This delayed cast gets read and reduced later using the same cast function and the process continues until a value is written to that heap cell. It is believed that storing delayed casts on the heap made the type safety proof less complicated. In the next section, it is shown that the typing of the heap in λ_p^{ref} depends on both the current heap typing and the side queue which complicates the type safety proof.

3. λ_p^{ref} Type Safety

3.1. Merging The Queue and Heap Typing. The queue of suspended casts plays a role in typing as well as in reduction. More specifically, the typing of values in the heap depends on both the heap typing and the queue of suspended casts. Both are combined using the following function, Φ.

Definition 5 (Merging Heap Typing and Suspended Casts Queue).

$$\begin{aligned}
\Phi(\Sigma, \emptyset) &= \Sigma \\
\Phi(\Sigma, ((a, T); \Psi)) &= \Phi((\Sigma(a \mapsto (T \sqcap \Sigma(a)))), \Psi) \quad \textit{if } \Sigma(a) \sim T \\
\Phi(\Sigma, ((a, T); \Psi)) &= \Phi(\Sigma, \Psi) \quad \textit{if } \Sigma(a) \nsim T
\end{aligned}$$

Lemma 11 shows that the result heap typing from Φ is more precise than the input one.

Lemma 11. *If* $\Sigma \vdash \Psi$*, and* $\Sigma' \sqsubseteq_p \Sigma$ *then,* $\Phi(\Sigma', \Psi) \sqsubseteq_p \Sigma'$

Proof. By induction on Ψ:

Case $\emptyset$: From $\Phi(\Sigma', \emptyset) = \Sigma'$ and the reflexivity of $\sqsubseteq_p$, we have $\Sigma' \sqsubseteq_p \Sigma'$.

Case $((a, T); \Psi)$: There are two cases:

Case $\Sigma(a) \nsim T$: By IH, we have $\Phi(\Sigma', \Psi) \sqsubseteq_p \Sigma'$.

Case $\Sigma(a) \sim T$: Consider the following:

(1) By the definition of $\sqcap$, we have $(T \sqcap \Sigma'(a) \sqsubseteq \Sigma'(a))$. From that we have $\Sigma'(a \mapsto T \sqcap \Sigma'(a)) \sqsubseteq_p \Sigma'$.

(2) By IH we have: $\Phi(\Sigma'(a \mapsto T \sqcap \Sigma'(a)), \Psi) \sqsubseteq_p \Sigma'(a \mapsto T \sqcap \Sigma'(a))$.

From 1 and 2, and the transitivity of $\sqsubseteq_p$, we have $\Phi(\Sigma'(a \mapsto T \sqcap \Sigma'(a)), \Psi) \sqsubseteq_p \Sigma'$.

□

Lemma 12 says that appending to the input queue can only make the result typing of Φ more precise.

Lemma 12 (Extending the queue). *If* $\Sigma \vdash \Psi$ *and* $\Sigma' \sqsubseteq_p \Sigma$*, then,* $\forall\ \Psi'$*, s.t.* $\Sigma \vdash \Psi'$*,* $\Phi(\Sigma', (\Psi \oplus \Psi')) \sqsubseteq_p \Phi(\Sigma', \Psi)$

Proof. By induction on Ψ:

Case $\emptyset$: By Lemma 11.

Case $((a, T); \Psi)$: There are two cases:

Case $\Sigma(a) \nsim T$: By IH, we have $\Phi(\Sigma', (\Psi \oplus \Psi')) \sqsubseteq_p \Phi(\Sigma', \Psi)$.

Case $\Sigma'(a) \sim T$: By IH, we have $\Phi(\Sigma'(a \mapsto T \sqcap \Sigma'(a)), (\Psi \oplus \Psi')) \sqsubseteq_p \Phi(\Sigma'(a \mapsto T \sqcap \Sigma'(a)), \Psi)$.

□

Lemma 13 shows that Φ preserves the append operation $\oplus$.

LEMMA 13 (Φ preserves $\oplus$). *If* $\Sigma \vdash \Psi$ *and* $\Sigma \vdash \Psi'$ *and* $\Sigma' \sqsubseteq_p \Sigma$*, then,* $\Phi(\Sigma', (\Psi \oplus \Psi')) = \Phi((\Phi(\Sigma', \Psi)), \Psi')$

PROOF. By induction on Ψ:

Case $\emptyset$**:** We need to show that $\Phi(\Sigma', \Psi') = \Phi(\Sigma', \Psi')$. By reflexivity.

Case $((a, T); \Psi)$**:** There are two cases:

Case $\Sigma'(a) \nsim T$**:** We need to show that $\Phi(\Sigma', (\Psi \oplus \Psi')) \sqsubseteq_p \Phi((\Phi(\Sigma', \Psi)), \Psi')$. By IH.

Case $\Sigma'(a) \sim T$**:** We need to show that $\Phi(\Sigma'(a \mapsto T \sqcap \Sigma'(a)), (\Psi \oplus \Psi')) \sqsubseteq_p \Phi((\Phi(\Sigma'(a \mapsto T \sqcap \Sigma'(a)), \Psi)), \Psi')$. By IH setting $\Sigma' = \Sigma'(a \mapsto T \sqcap \Sigma'(a))$.

□

Lemma 14 formalizes how adding a cast to the queue changes the typing of the address being cast.

LEMMA 14 (Φ soundness). *If* $\Sigma \vdash \Psi$ *and* $\Sigma' \sqsubseteq_p \Sigma$ *and* $(a, T) \in \Phi(\Sigma', \Psi)$*, then,* $(a, T \sqcap T') \in \Phi(\Sigma', (\Psi \oplus (\emptyset; (a, T'))))$

PROOF. Apply Lemma 13 by choosing $\Psi' = \emptyset; (a, T')$. Now we need to show that $(a, T \sqcap T') \in \Phi((\Phi(\Sigma', \Psi)), (\emptyset; (a, T')))$. Evaluating the outer function application gives us $(a, T \sqcap T') \in (\Phi(\Sigma', \Psi))(a \mapsto T \sqcap T')$. □

3.2. Heaps. A heap is typed using two heap typings. The first one specifies the types of the values currently living in the heap i.e. the RTTIs. The other typing is created by merging the current queue of suspended casts and the aforementioned typing and is used as the heap typing part of the values living in the heap.

DEFINITION 6 (Well-typed heaps). *A heap* μ *is well-typed with respect to heap typings* Σ' *and* Σ''*, written* $\Sigma' \mid \Sigma'' \vdash \mu$*, iff* $\forall a, T.\ \Sigma'(a) = T \implies \exists\ v$ *s.t.* $\Sigma'' \mid \emptyset \vdash v : T$ *and* $\mu(a) = v : T$*.*

If the queue of suspended casts is empty, then it must be the case that $\Sigma \mid \Sigma \vdash \mu$. If the queue Ψ is not empty, then it must be the case that $\Sigma' \mid \Phi(\Sigma', \Psi) \vdash \mu$ for some heap typing Σ'.

Lemma 15 formalizes how a heap gets cast.

LEMMA 15 (Heap cast). *If* $\Sigma \vdash \Psi$ *and* $\Sigma' \mid \Phi(\Sigma', ((a, T'); \Psi)) \vdash \mu$ *and* $(\Phi(\Sigma', ((a, T'); \Psi))) \mid \emptyset \vdash v : T$ *and* $\Sigma' \sqsubseteq_p \Sigma$ *and* $\mu(a) = v : T$*, then the heap cell address* a *points to can be cast to type*

$(T \sqcap T')$ *such that* $\Sigma'' = \Sigma'(a \mapsto (T \sqcap T'))$ *and* $\Sigma'' \mid \Phi(\Sigma'', (\Psi \oplus \Psi')) \vdash \mu(a \mapsto v' : (T \sqcap T'))$ *if such* v' *and* Ψ' *exist.*

PROOF. By cases on the result of apply-cast$(v, (T \Rightarrow (T \sqcap T')))$:

Case `error`: We conclude with a cast error.

Case (Ψ', v'): Let $\Sigma'' = \Sigma'(a \mapsto (T \sqcap T'))$. From $(T \sqcap T') \sqsubseteq T$ and $\Sigma' \mid (a, T'); \Psi \vdash \mu$ we have $(T \sqcap T') \sqsubseteq \Sigma'(a)$ which implies $\Sigma'' \sqsubseteq_p \Sigma'$. Also, by the definition of Φ, we have $\Phi(\Sigma'', \Psi) = \Phi(\Sigma', ((a, T'); \Psi))$, so from that and Lemma 12, we have that $\Phi(\Sigma'', (\Psi \oplus \Psi')) \sqsubseteq_p \Phi(\Sigma'', \Psi) \sqsubseteq_p \Phi(\Sigma', ((a, T'); \Psi))$. Next, Lemma 2 is applied to all values in the heap and to v', so we get $(\Phi(\Sigma'', (\Psi \oplus \Psi'))) \mid \emptyset \vdash v' : (T \sqcap T')$, thus $\Sigma'' \mid \Phi(\Sigma'', (\Psi \oplus \Psi')) \vdash \mu(a \mapsto v' : (T \sqcap T'))$.

□

3.3. Progress.

LEMMA 16 (Progress with suspended casts). *If* $\Sigma \vdash \Psi$ *and* $\Sigma' \sqsubseteq_p \Sigma$ *and* $\Sigma' \mid \Phi(\Sigma', ((a, T); \Psi)) \vdash \mu$ *and* $\Phi(\Sigma', ((a, T'); \Psi)) \mid \emptyset \vdash M : T$, *then* $\exists N, \Psi', \mu'$ *s.t.* $M, \mu, \Psi \longrightarrow N, \mu', (\Psi \oplus \Psi')$

PROOF. By cases on how T and $\mu(a)_{\text{rtti}}$ are related, a step is taken as follows:

$T \sim \mu(a)_{\mathbf{rtti}}$ and $T \sqcap \mu(a)_{\mathbf{rtti}} \neq \mu(a)_{\mathbf{rtti}}$: By cases on the result of (apply-cast$(\mu(a)_{\text{val}}, (\mu(a)_{\text{rtti}} \Rightarrow T)))$:

- **`error`:** A step is taken via ERROR1.
- **(v, Ψ''):** A step is taken via UPDATEHEAP.

$T \sim \mu(a)_{\mathbf{rtti}}$ and $T \sqcap \mu(a)_{\mathbf{rtti}} = \mu(a)_{\mathbf{rtti}}$: A step is taken via NOCHANGE.

$T \nsim \mu(a)_{\mathbf{rtti}}$: A step is taken via ERROR2.

□

If the queue of suspended casts is empty, then the progress lemma is standard.

LEMMA 17 (Progress with no suspended casts). *If* $\Sigma \mid \Sigma \vdash \mu$ *and* $\Sigma \mid \emptyset \vdash M : T$, *then either*

(1) M *is a value, or*

(2) $M =$ `error`, *or*

(3) $\exists\ \Psi, N, \mu'$ *s.t.* $M, \mu, \emptyset \longrightarrow N, \mu', \Psi$

From Lemmas 16 and 17, we assemble the full proof of progress by cases on whether the queue of suspended casts is empty:

Corollary 17.1 (Progress). *If* $\Sigma \vdash \Psi$ *and* $\Sigma' \mid \Phi(\Sigma', \Psi) \vdash \mu$ *and* $\Phi(\Sigma', \Psi) \mid \emptyset \vdash M : T$*, then either*

(1) M *is a value, or*

(2) $M = \texttt{error}$*, or*

(3) $\exists\ N, \mu'$ *s.t.* $M, \mu, \Psi \longrightarrow N, \mu', \Psi'$

3.4. Preservation. Lemma 18 asserts the well-typing of the apply-cast function.

Lemma 18 (apply-cast is well-typed). *If* $\Sigma \vdash \Psi$ *and* $\Sigma' \sqsubseteq_p \Sigma$ *and* $\Phi(\Sigma', \Psi) \mid \Gamma \vdash v : T$ *and* $c : T \Rightarrow T'$ *and apply-cast*$(v, c) = (\Psi', v')$*, then* $\Phi(\Sigma', \Psi \oplus \Psi') \mid \Gamma \vdash v' : T'$

Proof. By induction on c:

Case ι**:** By $\Psi \oplus \emptyset = \Psi$ and $v' = v$.

Case $T_2?$**:** It must be the case that $v = v_1\langle T_1!\rangle$. By IH and choosing $v = v_1$ and $c = (T_1 \Rightarrow T_2)$.

Case $T_1!$**:** By $\Psi \oplus \emptyset = \Psi$ and $v' = v\langle T_1!\rangle$.

Case $c \rightarrow d$**:** By $\Psi \oplus \emptyset = \Psi$ and $v' = v\langle c \rightarrow d\rangle$.

Case $\mathsf{Ref}\ T$**:** It must be the case that $v = a$ for some address a. By Lemma 14, we have $(a, (\Phi(\Sigma', \Psi))(a) \sqcap T) \in \Phi(\Sigma', (\Psi \oplus (\emptyset; (a, T))))$. By the definition of $\sqcap$, we have $(\Phi(\Sigma', \Psi))(a) \sqcap T \sqsubseteq T$. From that, we have $\Phi(\Sigma', \Psi \oplus (\emptyset; (a, T))) \mid \Gamma \vdash a : \mathsf{Ref}\ T$.

Case $c \times d$**:** Consider the following:

(1) There is only one valid value of a pair type, a pair, so we have $v = \langle v_1, v_2\rangle$.

(2) Assume that $c : T_1 \Longrightarrow T_1'$ and $\Phi(\Sigma', \Psi) \mid \Gamma \vdash v_1 : T_1$. From 1 and IH, we have apply-cast$(v_1, c) = (\Psi', v_1')$ and $\Sigma \vdash \Psi'$ and $\Phi(\Sigma', \Psi \oplus \Psi') \mid \Gamma \vdash v_1' : T_1'$.

(3) From Lemma 12, we have $\Phi(\Sigma', \Psi \oplus \Psi') \sqsubseteq_p \Phi(\Sigma', \Psi)$.

(4) From 1 and Lemma 2 in Chapter 4, we have $\Phi(\Sigma', \Psi \oplus \Psi') \mid \Gamma \vdash v_2 : T_2$.

(5) Similar to 2, assume that $d : T_2 \Longrightarrow T_2'$. From 4, we have $\Phi(\Sigma', \Psi \oplus \Psi') \mid \Gamma \vdash v_2 : T_2$. By IH, we have apply-cast$(v_2, d) = (\Psi'', v_2')$ and $\Sigma \vdash \Psi''$ and $\Phi(\Sigma', (\Psi \oplus \Psi') \oplus \Psi'') \mid \Gamma \vdash v_2' : T_2'$.

(6) From 5 and the associativity of $\oplus$, we have $\Phi(\Sigma', \Psi \oplus (\Psi' \oplus \Psi'')) \mid \Gamma \vdash v_2' : T_2'$.

(7) From Lemma 12, we have $\Phi(\Sigma', \Psi \oplus (\Psi' \oplus \Psi'')) \sqsubseteq_p \Phi(\Sigma', \Psi \oplus \Psi')$.

(8) From 7 and applying Lemma 2 in Chapter 4 to v_1', we have $\Phi(\Sigma', \Psi \oplus (\Psi' \oplus \Psi'')) \mid \Gamma \vdash v_1' : T_1$.

From 6 and 8, we have $\Phi(\Sigma', \Psi \oplus (\Psi' \oplus \Psi'')) \mid \Gamma \vdash \langle v_1', v_2' \rangle : T_1 \times T_2$

□

Type preservation has two cases. Lemma 19 formalizes the first case where the queue of suspended casts is empty and reduction proceeds by taking a step using the program reduction relation $\longrightarrow_e$.

LEMMA 19 (Preservation with no suspended casts). *If* $\Sigma \mid \Sigma \vdash \mu$ *and* $\Sigma \mid \emptyset \vdash M : T$ *and* $M, \mu \longrightarrow_e N, \mu', \Psi$, *then* $\exists\ \Sigma', \Sigma''$ *s.t.* $\Sigma' \vdash \Psi$ *and* $\Sigma' \sqsubseteq_{p/e} \Sigma$ *and* $\Phi(\Sigma', \Psi) \mid \emptyset \vdash N : T$ *and* $\Sigma' \mid \Phi(\Sigma', \Psi) \vdash \mu'$

PROOF. Standard induction on the derivation. Lemma 18 is applied in the case of writing into a partially typed address. □

The other case of type preservation is if the queue of suspended cast is not empty. Lemma 20 formalizes this second.

LEMMA 20 (Preservation with suspended casts). *If* $\Sigma \vdash \Psi$ *and* $\Sigma' \mid \Phi(\Sigma', ((a, T); \Psi)) \vdash \mu$ *and* $\Sigma' \sqsubseteq_p \Sigma$ *and* $\Phi(\Sigma', ((a, T); \Psi)) \mid \emptyset \vdash M : T$ *and* $M, \mu, \Psi \longrightarrow N, \mu', \Psi'$, *then* $\exists\ \Sigma''$ *s.t.* $\Sigma \vdash \Psi'$ *and* $\Sigma'' \sqsubseteq_p \Sigma'$ *and* $\Phi(\Sigma'', \Psi') \mid \emptyset \vdash N : T$ *and* $\Sigma'' \mid \Phi(\Sigma'', \Psi') \vdash \mu'$

PROOF. By cases on how T and $\mu(a)_{\text{rtti}}$ are related, a step is taken as follows:

$T \sim \mu(a)_{\mathbf{rtti}}$ and $T \sqcap \mu(a)_{\mathbf{rtti}} \neq \mu(a)_{\mathbf{rtti}}$: By cases on the result of (apply-cast($\mu(a)_{\text{val}}, (\mu(a)_{\text{rtti}} \Rightarrow (T \sqcap \mu(a)_{\text{rtti}}))$)):

`error`: A step is taken via ERROR1. $\Sigma'' = \Sigma'$ and $\Psi' = ((a, T'); \Psi)$ and $\mu' = \mu$ and $\Sigma'' \sqsubseteq_p \Sigma'$ by reflexivity.

(v, Ψ''): A step is taken via UPDATEHEAP. $\Sigma'' = \Sigma'(a \mapsto T \sqcap \mu(a)_{\text{rtti}})$. We know that $\Phi(\Sigma', ((a, T'); \Psi)) \mid \emptyset \vdash M : T$ and we need to show that $\Phi(\Sigma'', (\Psi \oplus \Psi')) \mid \emptyset \vdash M : T$. This can be shown using Lemma 2 but we need first to show that $\Phi(\Sigma'', (\Psi \oplus \Psi')) \sqsubseteq_p \Phi(\Sigma', ((a, T'); \Psi))$.

(1) $\Phi(\Sigma'(a \mapsto T \sqcap \mu(a)_{\text{rtti}}), \Psi) \sqsubseteq_p \Phi(\Sigma', ((a, T'); \Psi))$ holds by evaluating the outer function application.

(2) $\Phi(\Sigma'(a \mapsto T \sqcap \mu(a)_{\text{rtti}}), (\Psi \oplus \Psi')) \sqsubseteq_p \Phi(\Sigma'(a \mapsto T \sqcap \mu(a)_{\text{rtti}}), \Psi)$ holds by Lemma 12.

We conclude from (1) and (2) and the transitivity of $\sqsubseteq_p$.

$T \sim \mu(a)_{\mathbf{rtti}}$ and $T \sqcap \mu(a)_{\mathbf{rtti}} = \mu(a)_{\mathbf{rtti}}$: A step is taken via NoChange. $\Sigma'' = \Sigma'$ and $\Psi' = \Psi$ and $\mu' = \mu$ and $\Sigma'' \sqsubseteq_p \Sigma'$ by reflexivity.

$T \nsim \mu(a)_{\mathbf{rtti}}$: A step is taken via Error2. $\Sigma'' = \Sigma'$ and $\Psi' = \Psi$ and $\mu' = \mu$ and $\Sigma'' \sqsubseteq_p \Sigma'$ by reflexivity.

□

Lemma 21 combines Lemmas 19 and 20 into one lemma for type preservation.

Lemma 21 (Type preservation). *If $\Sigma \vdash \Psi$ and $\Sigma_1 \mid \Phi(\Sigma_1, \Psi) \vdash \mu$ and $\Phi(\Sigma_1, \Psi) \mid \emptyset \vdash M : T$ and $M, \mu, \Psi \longrightarrow N, \mu', \Psi'$, then $\exists\ \Sigma_2, \Sigma_3$ s.t. $\Sigma \vdash \Psi'$ and $\Sigma_3 \mid \emptyset \vdash N : T$ and $\Sigma_2 \mid \Sigma_3 \vdash \mu'$ and $\Sigma_2 \sqsubseteq_{p/e} \Sigma_1$*

Proof. By induction on the queue of suspended casts Ψ and applying Lemmas 19 and 20. □

From Corollary 17.1 and Lemma 21, we prove type safety.

Theorem 22 (Type safety). *If $\Sigma \mid \Sigma \vdash \mu$ and $\Sigma \mid \emptyset \vdash e : T$, then either*

(1) e diverges, or

(2) $e = \texttt{error}$, or

(3) $\exists\ v, \Sigma', \mu'$ s.t. $e, \mu, \emptyset \longrightarrow^* v, \mu', \emptyset$ and $\Sigma' \mid \emptyset \vdash v : T$ and $\Sigma' \mid \emptyset \vdash \mu'$

4. $\lambda_S^{\mathbf{ref}}$: Practical and Space-Efficient Semantics for Monotonic References

Proxies ensure that the type of a proxied value matches a certain type. Proxies are needed when it is hard to check immediately if the type of a value matches some type. For instance, it is in general impossible to check whether an arbitrary untyped function will return a Boolean without applying it first. With proxies, the function gets wrapped in a proxy that performs the desired checks at application sites. In this case, it checks if the incoming argument can be cast to the domain type of the underlying function, and also checks if the return value could be cast to Boolean.

An important consequence of that wrapping, however, is that values could accumulate multiple proxies when flowing through the boundaries between statically typed and dynamically typed

regions of code. Operations on such proxied values have to go through all the layers of proxies, performing all their checks in order to access the underlying value.

This problem was first observed in two higher-order, mutually recursive functions [Herman et al., 2007, 2010]. One of the functions expects a statically typed function as input and the other expects a dynamically typed function, so implicit function casts are inserted to mediate between the types. A function cast creates a run-time proxy wrapping the function being cast. Every time one of the functions is called, a new proxy is created. These proxies consume space proportional to the number of recursive calls.

With monotonic references there are no proxies on references, however neither the monotonic machine nor λ^{ref} integrated the solutions for space efficiency on other higher-order values such as functions. In this section, λ_p^{ref} is combined with space-efficiency for other values in a new semantics that is referred to as λ_S^{ref}.

4.1. Coercions in Normal Form for Monotonic References. A monotonic reference coercion holds just a type. To cast an address with some reference coercion $\mathsf{Ref}\ T$, the value at that address gets cast to the greatest lower bound of the RTTI and T. Casting an address with a sequence of two reference coercions amounts to taking the greatest lower bound of the RTTI and the types those two coercions hold. This suggests that composing two reference coercions should create a new reference coercion that holds a type that is the result of taking the greatest lower bound of the types held by the coercions being composed. However this definition causes subtle changes to the grammar and the typing rules of coercions.

λ_S^{ref} uses coercions in normal form to guarantee space-efficiency. Coercions in normal form are defined by a grammar consisting of three rules that enable coercion composition by a straightforward recursive function. This restricted grammar makes sure that the longest coercion will be one that starts with a projection, followed by a middle coercion, and ends with an injection. I adapt the normal form of [Kuhlenschmidt et al., 2019] (see the appendix in their auxiliary archive) by replacing the proxied reference coercion with one for monotonic references, and moving the failure coercion from final coercions to middle coercions. Figure 3 presents our coercions in normal form and how to create and compose them, which is described next.

The composition function $⨟$ takes two coercions in normal form as input and returns a coercion in normal form. It relies on three helper functions: $⨟^i_c$ that composes a final coercion with a coercion in normal form and returns a final coercion, $⨟^g_i$ that composes a middle coercion and a

Coercions (in normal form)	$c, d \in \mathcal{C}$	$::=$	$(I?\,;\,i) \mid i \mid \iota_\star$
Final coercions	i	$::=$	$(g\,;\,I!) \mid g$
Middle coercions	g	$::=$	$\iota \mid c \to d \mid c \times d \mid \mathsf{Ref}\ T \mid \bot$
Inert coercions	$c\uparrow$	$::=$	$i\uparrow$
Inert final coercions	$i\uparrow$	$::=$	$(g\,;\,I!) \mid g\uparrow$
Inert middle coercions	$g\uparrow$	$::=$	$c \to d$
Active coercions	$c\downarrow$	$::=$	$(I?\,;\,i) \mid i\downarrow \mid \iota_\star$
Active final coercions	$i\downarrow$	$::=$	$g\downarrow$
Active middle coercions	$g\downarrow$	$::=$	$\iota \mid c \times d \mid \mathsf{Ref}\ T \mid \bot$

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

$$\frac{}{\iota_\star : \star \Longrightarrow \star} \qquad \frac{}{\iota : I \Longrightarrow I} \qquad \frac{g : T \Longrightarrow I}{g\,;\,I! : T \Longrightarrow \star} \qquad \frac{i : I \Longrightarrow T}{I?\,;\,i : \star \Longrightarrow T} \qquad \frac{}{\bot : T \Longrightarrow T'}$$

$$\frac{c : T_1' \Longrightarrow T_1 \quad d : T_2 \Longrightarrow T_2'}{c \to d : T_1 \to T_2 \Longrightarrow T_1' \to T_2'} \qquad \frac{c : T_1 \Longrightarrow T_1' \quad d : T_2 \Longrightarrow T_2'}{c \times d : T_1 \times T_2 \Longrightarrow T_1' \times T_2'}$$

$$\frac{T \sqsubseteq T_2}{\mathsf{Ref}\ T : \mathsf{Ref}\ T_1 \Longrightarrow \mathsf{Ref}\ T_2}$$

Middle coercion creation $\boxed{(I \Rightarrow_g I) = i}$

$$(I \Rightarrow_g I) = \iota$$
$$(I \Rightarrow_g J) = \bot \qquad \text{if } I \not\sim J$$
$$(T_1 \to T_2 \Rightarrow_g T_1' \to T_2') = (T_1' \Rightarrow T_1) \to (T_2 \Rightarrow T_2')$$
$$(T_1 \times T_2 \Rightarrow_g T_1' \times T_2') = (T_1 \Rightarrow T_1') \times (T_2 \Rightarrow T_2')$$
$$(\mathsf{Ref}\ T \Rightarrow_g \mathsf{Ref}\ T') = \mathsf{Ref}\ T'$$

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

$$\begin{array}{rcl@{\qquad}rcl} (I \Rightarrow \star) &=& \iota\,;\,I! & (\star \Rightarrow \star) &=& \iota_\star \\ (\star \Rightarrow I) &=& I?\,;\,\iota & (I \Rightarrow J) &=& I \Rightarrow_g J \end{array}$$

Middle coercion composition $\boxed{g \fatsemi_g g = g}$

$$\begin{array}{rcl} \bot \fatsemi_g g &=& \bot \\ g \fatsemi_g \bot &=& \bot \\ \iota \fatsemi_g g &=& g \\ g \fatsemi_g \iota &=& g \\ c \to d \fatsemi_g c' \to d' &=& (c' \fatsemi c) \to (d \fatsemi d') \\ c \times d \fatsemi_g c' \times d' &=& (c \fatsemi c') \times (d \fatsemi d') \\ \mathsf{Ref}\ T \fatsemi_g \mathsf{Ref}\ T' &=& \bot \text{ if } T \not\sim T' \\ \mathsf{Ref}\ T \fatsemi_g \mathsf{Ref}\ T' &=& \mathsf{Ref}\ (T \sqcap T') \text{ if } T \sim T' \end{array}$$

Final and normal composition $\boxed{i \fatsemi_c^i c = i}$

$$\begin{array}{rcl} i \fatsemi_c^i \iota_\star &=& i \\ (g\,;\,I!) \fatsemi_c^i (J?\,;\,i) &=& g \fatsemi_c^i ((I \Rightarrow_g J) \fatsemi_c^i i) \\ g \fatsemi_c^i i &=& g \fatsemi_i^g i \end{array}$$

Middle and final composition $\boxed{g \fatsemi_i^g i = i}$

$$\begin{array}{rcl} g \fatsemi_i^g (g'\,;\,I!) &=& (g \fatsemi_g g')\,;\,I! \\ g \fatsemi_i^g g' &=& g \fatsemi_g g' \end{array}$$

Coercion composition $\boxed{c \fatsemi c = c}$

$$\iota_\star \fatsemi c = c \qquad (I?\,;\,i) \fatsemi c = I?\,;\,(i \fatsemi_c^i c) \qquad i \fatsemi c = i \fatsemi_c^i c$$

Figure 3. Coercions in normal form.

final coercion and returns a final coercion, and $\fatsemi_g$ that composes two middle coercions and returns a middle coercion.

Composing reference coercions

Composing two proxied reference coercions is done by composing the coercions for the read and the coercions for the write [Herman et al., 2007, 2010]. Composing two monotonic coercions is different because they hold types instead that are used to cast the heap.

The types held by the two coercions are merged by taking the greatest lower bound with respect to the precision relation $\sqsubseteq$ (using the function $\sqcap$). However, the function $\sqcap$ is partial and is undefined when the two input types are not consistent. In that case the failure coercion $\bot$ is returned instead, which is why failure coercions are moved into the grammar rule for middle coercions.

The coercion composition function maps two well-typed coercions to a well-typed coercion.

Lemma 23 (Well-typed Composition). *if* $c : T_1 \Longrightarrow T_2$ *and* $d : T_2 \Longrightarrow T_3$*, then* $c \fatsemi d : T_1 \Longrightarrow T_3$

Proof. By induction on c and d. For the case $\mathsf{Ref}\ T \fatsemi_g \mathsf{Ref}\ T'$: We know that $T' \sqsubseteq T_3$ by $\mathsf{Ref}\ T' : T_2 \Longrightarrow T_3$. We also know that $(T \sqcap T') \sqsubseteq T'$. By this and the transitivity of $\sqsubseteq$, we conclude that $(T \sqcap T') \sqsubseteq T_3$. □

Figure 4 presents size functions for types and coercions. The proof of termination for coercion composition relies on the fact that the sum of the sizes of the input coercions gets smaller at each recursive call. However, the case for composing injections and projections is complicated because a new coercion is created using the coercion creation function ($\Rightarrow_g$) and we need to reason about its size.

Lemma 24 (Size of created middle coercions). $size_T(T \Rightarrow_g T') \leq 2 * (size_T(T) + size_T(T'))$

Lemma 25 (Size of created coercions). $size_T(T \Rightarrow T') \leq 1 + 2 * (size_T(T) + size_T(T'))$

Note that the multiplication by 2 in the *size* function and in Lemmas 24 and 25 is needed to accommodate the case where a sequence of coercions is created, e.g. injections and projections.

Proposition 1 (Termination of Composition). $size(c \fatsemi d) \leq size(c) + size(d)$

Proof. By standard induction on the size in each case. Lemma 24 is applied in the case of composing an injection and a projection. □

Type size $\boxed{size_T(T)}$

$$size_T(\mathsf{Int}) = 1 \qquad size_T(()) = 1 \qquad size_T(\star) = 1 \qquad \|T_1 \to T_2\| = 1 + size_T(T_1) + size_T(T_2)$$
$$size_T(T_1 \times T_2) = 1 + size_T(T_1) + size_T(T_2) \qquad size_T(\mathsf{Ref}\ T) = 1 + size_T(T)$$

Coercion size $\boxed{size(c)}$

$$\begin{aligned} size(\iota_\star) &= 1 \\ size(I?\,;\,i) &= 1 + (2 * size_T(I)) + size_i(i) \\ size(i) &= 1 + size_i(i) \end{aligned}$$

Final coercion size $\boxed{size_i(i)}$

$$\begin{aligned} size_i(g\,;\,I!) &= 1 + (2 * size_T(I)) + size_g(g) \\ size_i(g) &= 1 + size_g(g) \end{aligned}$$

Middle coercion size $\boxed{size_g(g)}$

$$\begin{aligned} size_g(\iota) &= 1 \\ size_g(c \to d) &= 1 + size(c) + size(d) \\ size_g(c \times d) &= 1 + size(c) + size(d) \\ size_g(\mathsf{Ref}\ T) &= 1 + size_T(T) \\ size_g(\bot) &= 1 \end{aligned}$$

FIGURE 4. Sizes of types and coercions in normal form.

Type height $\boxed{\|T\|}$

$$\|\mathsf{Int}\| = 1 \qquad \|()\| = 1 \qquad \|\star\| = 1 \qquad \|T_1 \to T_2\| = 1 + max(\|T_1\|, \|T_2\|)$$
$$\|T_1 \times T_2\| = 1 + max(\|T_1\|, \|T_2\|) \qquad \|\mathsf{Ref}\ T\| = 1 + \|T\|$$

Coercion height $\boxed{\|c\|}$

$$\|\iota_\star\| = 1 \qquad \|\iota\| = 1 \qquad \|\bot\| = 1 \qquad \|\mathsf{Ref}\ T\| = 1 + \|T\| \qquad \|c \to d\| = 1 + max(\|c\|, \|d\|)$$
$$\|c \times d\| = 1 + max(\|c\|, \|d\|) \qquad \|c\,;\,I!\| = max(\|c\|, \|I\|) \qquad \|I?\,;\,c\| = max(\|I\|, \|c\|)\|$$

FIGURE 5. Height functions for types and coercions in normal form.

Furthermore, to establish space-efficiency, we reason about the height of created and composed coercions. Figure 5 presents height functions for types and coercions. The definition of height I use is different from the one of Herman et al. [2007] because of the difference in semantics between Lazy-UD and Lazy-D. Injection and projection coercions have a height of 1 in the former and the height of the types they carry in the latter. The composition of Lazy-D injection and projection coercions creates a new coercion out of the types they carry, so the height of those types contributes to the height of the newly created coercion.

Uncoerced Values	u	$::=$	$k \mid \lambda(x:T).\, e \mid \langle v, v\rangle \mid a$
Values	v	$::=$	$u \mid u\langle c\uparrow\rangle$
Heap	μ	$::=$	$\emptyset \mid \mu(a \mapsto v : T)$
Suspended casts	Ψ	$::=$	$\emptyset \mid \Psi; (a, T)$
Frames	F	$::=$	$\square\, e \mid v\,\square \mid \langle\square, e\rangle \mid \langle v, \square\rangle \mid \mathtt{fst}\ \square \mid \mathtt{snd}\ \square \mid$ $\mathtt{ref}\ \square@T \mid\ !\square@T \mid\ !\square \mid \square := e@T \mid v := \square@T$
Cast result	$\mathsf{r} \in \mathcal{R}$	$::=$	$(\Psi, v) \mid \mathtt{error}$

Cast application function $\boxed{\text{apply-cast} : \mathcal{V} \times \mathcal{C} \to \mathcal{R}}$

$$\begin{aligned}
\text{apply-cast}((u\langle c\uparrow\rangle), d) &= \text{apply-cast}(u, (c \mathbin{⨾} d)) \\
\text{apply-cast}(u, c\uparrow) &= (\emptyset, u\langle c\uparrow\rangle) \\
\text{apply-cast}(a, \mathsf{Ref}\ T) &= (((a, T); \emptyset), a) \\
\text{apply-cast}(u, \iota) &= (\emptyset, u) \\
\text{apply-cast}(\langle v_1, v_2\rangle, c \times d) &= ((\Psi \oplus \Psi'), \langle v_1', v_2'\rangle) \\
&\quad \text{if } \begin{array}{l}\text{apply-cast}(v_1, c) = (\Psi, v_1'), \\ \text{apply-cast}(v_2, d) = (\Psi', v_2')\end{array} \\
\text{apply-cast}(v, c) &= \mathtt{error}
\end{aligned}$$

FIGURE 6. Runtime structures and the apply-cast function for λ_S^{ref}.

LEMMA 26 (Coercions height is bounded). $\|T_1 \Rightarrow T_2\| \leq \max(\|T_1\|, \|T_2\|)$

PROPOSITION 2 (Composition height is bounded). $\|c \mathbin{⨾} d\| \leq \max(\|c\|, \|d\|)$

PROOF. By Lemma 26 and induction on the structure of the compose function $c \mathbin{⨾} d$. □

A well-typed coercion in normal form consists of at most two sequences (three coercions or less), so a space-efficient coercion bounded in height is also bounded in size.

4.2. λ_S^{ref} Dynamic Semantics. Values, defined in Figure 6, are adjusted so that at most one layer of casts is allowed. Furthermore, the apply-cast function, defined in Figure 6, maintains space-efficiency by composing the old coercion on the value being cast and the new coercion using the coercion composition function $⨾$. Moreover, the reduction relation is restricted to also use the composition function to combine adjacent coercions first before reducing expressions underneath [Herman et al., 2007, 2010, Siek and Wadler, 2010, Siek et al., 2015a]. In particular, a new rule COMPOSE is added to compose adjacent coercions to the program reduction relation $\longrightarrow_e$, defined in Figure 7. Furthermore, $\longrightarrow_e$ is restricted such that reduction is disallowed under a sequence of two or more casts. Typically, this restriction is implemented by introducing a notion of cast-free evaluation context that does not have a cast application innermost and requiring the reduction of cast expressions to occur in that cast-free context. Instead, λ_S^{ref} imposes this restriction using frames, a simpler structure than evaluation contexts. The frame for cast expressions

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

(Alloc) $\texttt{ref}\ v@T, \mu \longrightarrow_e^{\text{ccd}} a, \mu(a \mapsto v : T), \emptyset \quad \text{if } a \notin \text{dom}(\mu)$

(Read) $!a, \mu \longrightarrow_e^{\text{ccd}} \mu(a)_{\text{val}}, \mu, \emptyset$

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

(Write) $a := v, \mu \longrightarrow_e^{\text{ccd}} \text{Unit}, \mu(a \mapsto v : \mu(a)_{\text{rtti}}), \emptyset$

(DynWrite) $a := v@T, \mu \longrightarrow_e^{\text{ccd}} \text{Unit}, \mu(a \mapsto v' : \mu(a)_{\text{rtti}}), \Psi$
$\quad \text{if apply-cast}(v, (T \Rightarrow \mu(a)_{\text{rtti}})) = (\Psi, v')$

(DynWriteFail) $a := v@T, \mu \longrightarrow_e^{\text{ccd}} \texttt{error}, \mu, \emptyset$
$\quad \text{if apply-cast}(v, (T \Rightarrow \mu(a)_{\text{rtti}})) = \texttt{error}$

(Compose) $M\langle c\rangle\langle d\rangle, \mu \longrightarrow_e^{\text{ccd}} M\langle c \fatsemi d\rangle, \mu, \emptyset$

$$\text{Switch}\ \frac{M, \mu \longrightarrow_e^{\text{ccd}} N, \mu, \Psi}{M, \mu \longrightarrow_e^{\text{cca}} N, \mu, \Psi} \qquad \text{Pure}\ \frac{M \longrightarrow_p N}{M, \mu \longrightarrow_e^{\text{ccd}} N, \mu, \emptyset}$$

$$\text{Cast/Succeed}\ \frac{\text{apply-cast}(v, c) = (\Psi, v')}{v\langle c\rangle, \mu \longrightarrow_e^{\text{ccd}} v', \mu, \Psi} \qquad \text{Cast/fail}\ \frac{\text{apply-cast}(v, c) = \texttt{error}}{v\langle c\rangle, \mu \longrightarrow_e^{\text{ccd}} \texttt{error}, \mu, \emptyset}$$

$$\text{Cong}\ \frac{M, \mu \longrightarrow_e^{\text{cca}} N, \mu', \Psi}{F[M], \mu \longrightarrow_e^{\text{ccd}} F[N], \mu', \Psi} \qquad \text{CongErr}\ \frac{}{F[\texttt{error}], \mu \longrightarrow_e^{\text{ccd}} \texttt{error}, \mu, \emptyset}$$

$$\text{CongCast}\ \frac{M, \mu \longrightarrow_e^{\text{ccd}} N, \mu', \Psi}{M\langle c\rangle, \mu \longrightarrow_e^{\text{cca}} N\langle c\rangle, \mu', \Psi} \qquad \text{CongCastE}\ \frac{}{\texttt{error}\langle c\rangle, \mu \longrightarrow_e^{\text{cca}} \texttt{error}, \mu, \emptyset}$$

State reduction rules $\boxed{e, \mu, \Psi \longrightarrow e, \mu, \Psi}$

$$\text{ProgReduce}\ \frac{e, \mu \longrightarrow_e e', \mu, \Psi}{e, \mu, \emptyset \longrightarrow e', \mu, \Psi} \qquad \text{UpdateHeap}\ \frac{T' = T \sqcap \mu(a)_{\text{rtti}} \quad T' \neq \mu(a)_{\text{rtti}} \quad \text{apply-cast}(\mu(a)_{\text{val}}, (\mu(a)_{\text{rtti}} \Rightarrow T')) = (\Psi', v')}{e, \mu, (a, T); \Psi \longrightarrow e, \mu(a \mapsto v' : T'), \Psi \oplus \Psi'}$$

$$\text{NoChange}\ \frac{T \sqcap \mu(a)_{\text{rtti}} = \mu(a)_{\text{rtti}}}{e, \mu, (a, T); \Psi \longrightarrow e, \mu, \Psi} \qquad \text{Error1}\ \frac{T' = T \sqcap \mu(a)_{\text{rtti}} \quad T' \neq \mu(a)_{\text{rtti}} \quad \text{apply-cast}(\mu(a)_{\text{val}}, (\mu(a)_{\text{rtti}} \Rightarrow T')) = \texttt{error}}{e, \mu, (a, T); \Psi \longrightarrow \texttt{error}, \mu, \Psi}$$

$$\text{Error2}\ \frac{T \nsim \mu(a)_{\text{rtti}}}{e, \mu, (a, T); \Psi \longrightarrow \texttt{error}, \mu, \Psi}$$

Figure 7. λ_S^{ref} dynamic semantics.

is removed from the grammar for frames, defined in Figure 6, and congruence rules for cast expressions is added to $\longrightarrow_e$. Furthermore, the reduction relation is indexed by a flag that indicates whether the cast congruence rule is allowed (cca) or disallowed (ccd) in the current context. The cast congruence rules, CastCong and CastCongE, require the flag cca, and they switch the flag

of the subexpression reduction to ccd. The rest of the rules are indexed by ccd. To allow rules indexed by ccd to be also available in cca contexts, a subsumption rule, Switch is added, that flips the flag from cca to ccd. This style is inspired by Siek [2019] formalization of the space-efficient gradually typed lambda calculus in Agda.

5. $\lambda_S^{\mathbf{ref}}$ Type Safety

Most proofs are omitted because they look very similar to the one for λ_p^{ref}. The reader can refer to Appendix A for the Agda proofs. Furthermore, the Φ function used to merge heap typing and the queue of suspended cast function in λ_p^{ref} is exactly the same for λ_S^{ref}.

Lemma 27 formalizes how a heap gets cast.

Lemma 27 (Heap cast). *If* $\Sigma \vdash \Psi$ *and* $\Sigma' \mid \Phi(\Sigma', ((a, T'); \Psi)) \vdash \mu$ *and* $(\Phi(\Sigma', ((a, T'); \Psi))) \mid \emptyset \vdash v : T$ *and* $\Sigma' \sqsubseteq_p \Sigma$ *and* $\mu(a) = v : T$*, then the heap cell address* a *points to can be cast to type* $(T \sqcap T')$ *such that* $\Sigma'' = \Sigma'(a \mapsto (T \sqcap T'))$ *and* $\Sigma'' \mid \Phi(\Sigma'', (\Psi \oplus \Psi')) \vdash \mu(a \mapsto v' : (T \sqcap T'))$ *if such* v' *and* Ψ' *exist.*

5.1. Progress.

Lemma 28 (Progress with suspended casts). *If* $\Sigma \vdash \Psi$ *and* $\Sigma' \sqsubseteq_p \Sigma$ *and* $\Sigma' \mid \Phi(\Sigma', ((a, T); \Psi)) \vdash \mu$ *and* $\Phi(\Sigma', ((a, T'); \Psi)) \mid \emptyset \vdash M : T$*, then* $\exists N, \Psi', \mu'$ *s.t.* $M, \mu, \Psi \longrightarrow N, \mu', (\Psi \oplus \Psi')$

As before, if the queue of suspended casts is empty, then the progress lemma is standard.

Lemma 29 (Progress with no suspended casts). *If* $\Sigma \mid \Sigma \vdash \mu$ *and* $\Sigma \mid \emptyset \vdash M : T$*, then either*

(1) M *is a value, or*

(2) $M = \texttt{error}$*, or*

(3) $\exists\ \Psi, N, \mu'$ *s.t.* $M, \mu, \emptyset \longrightarrow N, \mu', \Psi$

From Lemmas 28 and 29, we assemble the full proof of progress by cases on whether the queue of suspended casts is empty:

Corollary 29.1 (Progress). *If* $\Sigma \vdash \Psi$ *and* $\Sigma' \mid \Phi(\Sigma', \Psi) \vdash \mu$ *and* $\Phi(\Sigma', \Psi) \mid \emptyset \vdash M : T$*, then either*

(1) M *is a value, or*

(2) $M = \texttt{error}$*, or*

(3) $\exists\ N, \mu'$ *s.t.* $M, \mu, \Psi \longrightarrow N, \mu', \Psi'$

5.2. Preservation. Lemma 30 asserts the well-typing of the apply-cast function.

Lemma 30 (apply-cast is well-typed). *If* $\Sigma \vdash \Psi$ *and* $\Sigma' \sqsubseteq_p \Sigma$ *and* $\Phi(\Sigma', \Psi) \mid \Gamma \vdash v : T$ *and* $c : T \Rightarrow T'$ *and apply-cast*$(v, c) = (\Psi', v')$*, then* $\Phi(\Sigma', \Psi \oplus \Psi') \mid \Gamma \vdash v' : T'$

Type preservation has two cases. Lemma 31 formalizes the first case where the queue of suspended casts is empty and reduction proceeds by taking a step using the program reduction relation $\longrightarrow_e$.

Lemma 31 (Preservation with no suspended casts). *If* $\Sigma \mid \Sigma \vdash \mu$ *and* $\Sigma \mid \emptyset \vdash M : T$ *and* $M, \mu \longrightarrow_e N, \mu', \Psi$*, then* $\exists\ \Sigma', \Sigma''$ *s.t.* $\Sigma' \vdash \Psi$ *and* $\Sigma' \sqsubseteq_{p/e} \Sigma$ *and* $\Phi(\Sigma', \Psi) \mid \emptyset \vdash N : T$ *and* $\Sigma' \mid \Phi(\Sigma', \Psi) \vdash \mu'$

The other case of type preservation is if the queue of suspended cast is not empty. Lemma 32 formalizes this second.

Lemma 32 (Preservation with suspended casts). *If* $\Sigma \vdash \Psi$ *and* $\Sigma' \mid \Phi(\Sigma', ((a, T); \Psi)) \vdash \mu$ *and* $\Sigma' \sqsubseteq_p \Sigma$ *and* $\Phi(\Sigma', ((a, T); \Psi)) \mid \emptyset \vdash M : T$ *and* $M, \mu, \Psi \longrightarrow N, \mu', \Psi'$*, then* $\exists\ \Sigma''$ *s.t.* $\Sigma \vdash \Psi'$ *and* $\Sigma'' \sqsubseteq_p \Sigma'$ *and* $\Phi(\Sigma'', \Psi') \mid \emptyset \vdash N : T$ *and* $\Sigma'' \mid \Phi(\Sigma'', \Psi') \vdash \mu'$

Lemma 33 combines Lemmas 31 and 32 into one lemma for type preservation.

Lemma 33 (Type preservation). *If* $\Sigma \vdash \Psi$ *and* $\Sigma_1 \mid \Phi(\Sigma_1, \Psi) \vdash \mu$ *and* $\Phi(\Sigma_1, \Psi) \mid \emptyset \vdash M : T$ *and* $M, \mu, \Psi \longrightarrow N, \mu', \Psi'$*, then* $\exists\ \Sigma_2, \Sigma_3$ *s.t.* $\Sigma \vdash \Psi'$ *and* $\Sigma_3 \mid \emptyset \vdash N : T$ *and* $\Sigma_2 \mid \Sigma_3 \vdash \mu'$ *and* $\Sigma_2 \sqsubseteq_{p/e} \Sigma_1$

From Corollary 29.1 and Lemma 33, we prove type safety.

Theorem 34 (Type safety). *If* $\Sigma \mid \Sigma \vdash \mu$ *and* $\Sigma \mid \emptyset \vdash e : T$*, then either*

(1) e *diverges, or*

(2) $e = \texttt{error}$*, or*

(3) $\exists\ v, \Sigma', \mu'$ *s.t.* $e, \mu, \emptyset \longrightarrow^* v, \mu', \emptyset$ *and* $\Sigma' \mid \emptyset \vdash v : T$ *and* $\Sigma' \mid \emptyset \vdash \mu'$

Expression size $\boxed{size_e(e)}$

$$size_e(k) = 1 \qquad size_e(\lambda(x:T).\, e) = 1 + size_T(T) + size_e(e)$$
$$size_e(e_1\, e_2) = 1 + size_e(e_1) + size_e(e_2) \qquad size_e(\langle e_1, e_2\rangle) = 1 + size_e(e_1) + size_e(e_2)$$
$$size_e(\texttt{fst}\ e) = 1 + size_e(e) \qquad size_e(\texttt{snd}\ e) = 1 + size_e(e)$$
$$size_e(\texttt{ref}\ e@T) = 1 + size_T(T) + size_e(e) \qquad size_e(!e) = 1 + size_e(e)$$
$$size_e(!e@T) = 1 + size_T(T) + size_e(e) \qquad size_e(e_1 := e_2) = 1 + size_e(e_1) + size_e(e_2)$$
$$size_e(e_1 := e_2@T) = 1 + size_T(T) + size_e(e_1) + size_e(e_2) \qquad size_e(\texttt{error}) = 1$$
$$size_e(a) = 1 \qquad size_e(e\langle c\rangle) = 1 + size_e(e) + size(c) \qquad size(x) = 1$$

Store size $\boxed{size_\mu(\mu)}$

$$size_\mu(\mu) = \sum_{a\in \mathrm{dom}(\mu)} (1 + size_T(\mu(a)_{\mathrm{rtti}}) + size(\mu(a)_{\mathrm{val}}))$$

Queue size $\boxed{size_\Psi(\Psi)}$

$$size_\Psi(\Psi) = \sum_{(a,T)\in\Psi} (1 + size_T(T))$$

Configuration size $\boxed{size(e,\mu,\Psi)}$

$$size(e,\mu,\Psi) = size_e(e) + size_\mu(\mu) + size_\Psi(\Psi)$$

FIGURE 8. Sizes of configurations.

6. Space-Efficiency

In this section, I show that the total cost of maintaining coercions in λ_S^{ref} is bounded. I define the size of a configuration inductively as the sum of the sizes of its components (defined in Figure 8). To show that coercions occupy bounded space, I compare the size of configurations in reduction sequences to configurations in an "oracle" semantics where coercions do not occupy space at all. The oracular measure $size_{OR}$ is defined similarly to $size$, but without a cost for maintaining coercions; that is, $size_{OR}(c) = 0$. Theorem 35 states that the fraction of the configuration occupied by coercions is bounded in λ_S^{ref}.

THEOREM 35 (Space consumption). *If* $\Sigma \mid \emptyset \vdash M \hookrightarrow N : T$ *and* $N, \emptyset, \emptyset \longrightarrow N', \mu, \Psi$, *then there exists some type* T' *in the derivation of* $\Sigma \mid \emptyset \vdash M \hookrightarrow N : T$ *such that* $size(N', \mu, \Psi) \in O(2^{\|T\|} \times size_{OR}(N', \mu, \Psi))$.

PROOF SKETCH. Similar to the proof of Theorem 5 of Herman et al. [2010], during evaluation, the CONGCAST rule prevents nesting of adjacent coercions in any term in the evaluation context, redex, or store. Thus the number of coercions in the configuration is proportional to the size of the

configuration. The size of each coercion is bounded by its height and the height of the coercion is bounded by the height of the largest type in the typing of M. $\square$

CHAPTER 6

Grift: A Compiler for Gradual Typing

In this chapter I present a compiler, named Grift, that addresses the difficult challenge of efficient gradual typing for structural types. The input language includes a selection of difficult features: first-class functions, mutable arrays, and recursive types. The language is an extension of the Gradually Typed Lambda Calculus, abbreviated GTLC+.

Grift compiles the GTLC+ to C, then uses the Clang compiler to generate x86 executables. The Clang compiler provides low level optimizations. The GTLC+ language includes base types such as integers (fixnums), double precision floats, and Booleans but does not support implicit conversions between base types (i.e. no numeric tower). The GTLC+ also includes structural types such as n-ary tuples, mutable vectors, and higher-order functions. Figure 1 defines the syntax of the GTLC+. Grift does not yet implement space-efficient tail recursion, but implementation strategies for doing so are already known [Herman et al., 2007] and [Siek and Garcia, 2012]. This chapter presents a high level description of the techniques used to generate code for coercions. The code for Grift is available at the URL `https://github.com/Gradual-Typing/Grift`.

The first step in the Grift compiler is to translate to an intermediate language with explicit casts. This process is standard [Siek and Taha, 2006, Siek, 2008, Herman et al., 2010] except that a local optimization is added to avoid unnecessary casts in untyped code. The standard cast insertion algorithm [Siek et al., 2015b] can cause unnecessary overhead in untyped regions of code. Consider the function `(lambda ([f : Dyn]) (f 42))` which applies a dynamically typed value `f` as a function. The standard algorithm would compile it to:

```
(lambda ([f : Dyn])
  ((cast f Dyn (Dyn -> Dyn) L) (cast 42 Int Dyn L)))
```

The cast on `f` will allocate a function proxy if the source type of `f` is anything but `(Dyn -> Dyn)`. Although the proxy is important to obtain the desired semantics, the allocation is unnecessary in this case because the proxy is used right away and never used again. Instead, Grift specializes these

```
Variables      x  ::=  (lisp style identifiers)
Characters     c  ::=  (lisp style character literals)
Integers       i  ::=  (signed 61 bit integers)
Floats         f  ::=  (double precision floating point numbers)
Blame Label    l  ::=  (double quoted strings)
Types          T  ::=  x | Dyn | Unit | Bool | Int | Char | Float | (T ...-> T) | (Tuple T ...)
                       | (Ref T)|(Vect T)|(Rec x T)
Literals       V  ::=  () | #f | #t | c | i | f
Operators      O  ::=  + | - | * | / | < | <= | = | >= | > | fl+ | fl- | fl* | fl/ | fl< | fl<=
                   |   fl= | fl>= | fl> | int->char | char->int | float->int | int->float
                   |   print-int | read-int | print-float | print-char | read-char
Parameters     F  ::=  x | (x : T)
Expressions    E  ::=  V | (O E ...) | (ann E T l) | (if E E E) | (time E) | x
                   |   (lambda (F ...) : T E) | (E E ...) | (let ([x : T E] ...)E ...)
                   |   (letrec ([x : T E] ...) E ...) | (tuple E ...) | (tuple-proj E i)
                   |   (repeat (x E E) [(x E)] E) | (begin E ... E) | (box E) | (unbox E)
                   |   (box-set! E E) | (make-vector E E)(vector-ref E E)
                   |   (vector-set! E E E) | (vector-length E)
Definitions    D  ::=  (define x : T E) | (define (x F ...) : T E ...) | E
Program        P  ::=  D ...
```

FIGURE 1. The syntax of the GTLC+ as supported by Grift. This grammar shows every major syntactic form available in GTLC+, and presents a handful of the operators. Most type annotations can be omitted by dropping the preceding ":". The syntax for the omitting type annotations in the exception, formal parameters, is shown in the grammar.

cases by generating code that does what a proxy would do without actually allocating one. Grift applies this optimization to proxied references and tuples as well.

The next step in the compiler exposes the runtime functions that implement casts. The representation of values is described in Section 1. The implementation of coercions is described in Section 2. After lowering casts, Grift performs closure conversion using a flat representation [Cardelli, 1983, 1984, Appel, 1992], and translates all constructors and destructors to memory allocations, reads, writes, and numeric manipulation.

For memory allocation and reclamation, Grift uses the Boehm-Demers-Weiser conservative garbage collector [Boehm and Weiser, 1988, Demers et al., 1990]. Grift optimizes closures, for example, translating some closure applications into direct function calls using the techniques of Keep et al. [2012]. Grift does not perform any other general-purpose or global optimizations, such as type inference and specialization, constant folding, copy propagation, or inlining. On the other hand, the compiler does specialize casts based on their source and target type and it specializes operations on proxies.

1. Value Representation

Values are represented according to their type. An Int value is a 61-bit integer stored in 64-bits. A Bool value is also stored in 64-bits, using the C encoding of 1 for true and 0 for false. A function value is a pointer to one of two different kinds of closures; the lowest bit of the pointer indicates which kind. The first kind, for regular functions, is a flat closure that consists of 1) a function pointer, 2) a pointer to a function for casting the closure, and 3) the values of the free variables. The second kind of closure, which I call a *proxy closure*, is for functions that have been cast. It consists of 1) a function pointer (to a "wrapper" function), 2) a pointer to the underlying closure, and 3) a pointer to a coercion.

A value of reference type is a pointer to the data or to a proxy. The lowest bit of the pointer indicates which. A proxy consists of a reference and a pointer to a reference coercion. A value of type $\star$ is a 64-bit integer, with the 3 lowest bits indicating the type of the value that has been *injected* (i.e. cast into the $\star$ type). For types whose values can fit in 61 bits (e.g. Int and Bool), the injected value is stored inline. For types with larger values, the 61 bits are a pointer to a pair of 64-bit items that contain the injected value and its type. In the following section, the macros for allocating and accessing values have all uppercase names to distinguish them from C functions. The macro definitions are listed in Appendix 1.

2. Implementation of Coercions

Coercions are represented by heap allocated values. In Grift, the coercions that are statically known are allocated once at the start of the program. The runtime function `coerce`, described below, implements coercion application. To do so, it interprets the coercion and performs the actions it represents to the value.

Coercions are represented as 64-bit values where the lowest 3 bits indicate whether the coercion is a projection, injection, sequence, failure, or identity. For an identity coercion, the remaining 61 bits are not used. For the other coercions, the 61 bits store a pointer to heap-allocated structures that I describe below. Because the number of pointer tags is limited, the function, reference, tuple, and recursive coercions are differentiated by a secondary tag stored in the first word of their heap-allocated structure. The C type definitions for coercions are included in Appendix 1.

- Projection coercions ($T?p$) cast from $\star$ to a type T. The runtime representation is 2×64 bits: the first word is a pointer to the type T and the second is a pointer to the blame label p.
- Injection coercions ($T!$) cast from an arbitrary type to $\star$. The runtime representation is 64 bits, holding a pointer to the type T.
- Function coercions ($c_1, \ldots, c_n \to d$) cast between two function types of the same arity. A coercion for a function with n parameters is represented in $64 \times (n + 2)$ bits, where the first word stores the secondary tag and arity, the second stores a coercion on the return, and the remaining words store n coercions for the arguments.
- Reference coercions ($\mathsf{Ref}\ c\ d$) cast between reference or vector types and are represented as 3×64 bits, including the secondary tag, a coercion for writing, and another coercion for reading.
- Tuple coercions cast between two n-tuple types and are represented as $64 \times (n + 1)$ bits, including the secondary tag, the length of the tuple, and a coercion for each element of the tuple.
- Recursive coercions ($\mathsf{Rec}\ x.c$) serve as targets for back edges in "infinite" coercions created by casting between equirecursive types. They are represented in 2×64 bits for a secondary tag and a pointer to a coercion whose subcoercions can contain a pointer to this coercion.
- Sequences coercions ($c\,;\,d$) apply coercion c then coercion d and store two coercions in 2×64 bits.
- Failure coercions ($\bot^p$) immediately halt the program and are represented in 64 bits to store a pointer to the blame label.

Applying a Coercion

The application of a coercion to a value is implemented by a C function named `coerce`, shown in Figure 2, that takes a value and a coercion and either returns a value or signals an error. The `coerce` function dispatches on the coercion's tag. Identity coercions return the value unchanged. Sequence coercions apply the first coercion and then the second coercion. Injection coercions build a value of type $\star$. Projection coercions take a value of type $\star$ and build a new coercion from the runtime type to the target of the projection, which it applies to the underlying value.

Coercing a reference type (i.e. box or vector) builds a proxy that stores two coercions, one for reading and one for writing, and the pointer to the underlying reference. In case the reference has

```
obj coerce(obj v, crcn c) {
  switch(TAG(c)) {
  case ID_TAG: return v;
  case SEQUENCE_TAG:
    sequence seq = UNTAG_SEQ(c);
    return coerce(coerce(v, seq->fst), seq->snd);
  case PROJECT_TAG:
    projection proj = UNTAG_PRJ(c);
    crcn c2 = mk_crcn(TYPE(v), proj->type, proj->lbl);
    return coerce(UNTAG(v), c2);
  case INJECT_TAG:
    injection inj = UNTAG_INJECT(c);
    return INJECT(v, inj->type);
  case HAS_2ND_TAG: {
    switch (UNTAG_2ND(c)->second_tag) {
    case REF_COERCION_TAG:
      if (TAG(v) != REF_PROXY) {
        return MK_REF_PROXY(v, c);
      } else {
        ref_proxy p = UNTAG_REF_PROXY(v);
        crcn c2 = compose(p->coercion, c);
        return MK_REF_PROXY(p->ref, c2);  }
    case FUN_COERCION_TAG:
      if (TAG(v) != FUN_PROXY) {
        return UNTAG_FUN(v).caster(v, c);
      } else {
        fun_proxy p = UNTAG_FUN_PROXY(v);
        crcn c2 = compose(p->coercion, c);
        return MK_FUN_PROXY(p->wrap, p->clos, c2); }
    case TUPLE_COERCION_TAG:
      int n = TUPLE_COERCION_SIZE(c);
      obj t = MK_TUPLE(n);
      for (int i = 0; i < n; i++) {
        obj  e = t.tup->elem[i];
        crcn d = TUPLE_COERCION_ELEM(c, i);
        t.tup->elem[i] = coerce(e, d); }
      return t;
    case REC_COERCION_TAG:
      return coerce(v, REC_COERCION_BODY(c)); }}
  case FAIL_TAG: raise_blame(UNTAG_FAIL(c)->lbl); }}
```

FIGURE 2. The `coerce` function applies a coercion to a value.

already been coerced, the old and new coercions are composed via `compose`, so that there will only ever be one proxy on a proxied reference, which ensures space efficiency.

When coercing a function, `coerce` checks whether the function has previously been coerced. If it has not been previously coerced, then there is no proxy, and we invoke its function pointer for casting, passing it the function and the coercion to be applied. This "caster" function allocates and initializes a proxy closure. If the function has been coerced, Grift builds a new proxy closure containing the underlying closure, but its coercion is the result of composing the proxy's coercion with the coercion being applied via `compose` (the code for this function is in Appendix 1). The call to `compose` is what maintains space efficiency.

Coercing a tuple allocates a new tuple whose elements are the result of recurring on each of the elements of the original tuple with each of the corresponding subcoercions of the original tuple coercion. A recursive coercion is simply a target for specifying a recursive coercion. As such, it is ignored and the body of the recursive coercion is applied instead to value. Failure coercions halt execution and report an error.

Applying Functions

Because the coercion implementation distinguishes between regular closures and proxy closures, one might expect closure call sites to branch on the type of closure being applied. However, this is not the case because Grift ensures that the memory layout of a proxy closure is compatible with the regular closure calling convention. The only change to the calling convention of functions is that the lowest bit of the pointer to the closure, which distinguishes proxy closures from regular closures, has to be cleared. This representation is inspired by a technique of Siek and Garcia [2012] which itself is inspired by Findler and Felleisen [2002b].

Reading and Writing to Proxied References

To handle reads and writes on proxied references, Grift generates code that branches on whether the reference is proxied or not (by checking the tag bits on the pointer). If the reference is proxied the read or write coercion from the proxy is applied to the value read from or written to the reference. To ensure space efficiency, there can be at most one proxy on each reference. If the reference isn't proxied, the operation is a simple machine read or write.

3. Credits

The Grift project started in 2014 at Indiana University by Andre Kuhlenschmidt and Jeremy Siek. I joined the project a year later and worked on a handful new features, optimizations, diagnostics and profiling, and other compiler engineering tasks. My work in Grift includes adding proxied vectors and tuples, type hoisting and hashconsing, adding experimental memory allocators, adding a cast profiler, and building most of our benchmarking infrastructure. Furthermore, I implemented monotonic references and vectors in Grift and that will be described in detail in Chapter 7 . On the other hand, Andre Kuhlenschmidt created the core of the compiler pipeline starting from the parser all the way down to the C backend. In particular, he added support for both type-based casts and coercions. He also added support for different representations of function proxies that I will show how they perform alongside monotonic references in Chapter 8. Furthermore, Andre added support for closure optimizations and optimizations on casts based on known type information at compile-time and moved Grift to adapt the Boehm garbage collector. Finally, he added support for strings, floats, proxied references, and recursive types and created Static Grift which plays an important role in our performance evaluation studies.

CHAPTER 7

Implementation of Monotonic References in Grift

I have implemented monotonic references and monotonic vectors (based on the semantics of λ_S^{ref}) in Grift. The runtime system in Grift can represent casts using coercions in normal form to guarantee space-efficiency, which makes it a perfect host for implementing λ_S^{ref} that is also based on coercions in normal form.

1. Runtime Representations

Values Monotonic references and vectors have only one type of values, 64-bit addresses. In the case of references, the address points to a sequence of 16 bytes on the heap. The RTTI is stored in the first 8 bytes, and the pointed-to value is stored in the second 8 bytes. In the case of vectors of n elements, the address points to a sequence of $16 + (n * 8)$ bytes, where the RTTI is stored in the first 8 bytes, the length n is stored in the next 8 bytes, and finally, the n elements are stored in the subsequent $n * 8$ bytes.

Coercions As explain in Chapter 6, Grift coercions are represented as 64-bit values where the lowest 3 bits indicate whether the coercion is a projection, injection, sequence, failure, or identity. For an identity coercion, the remaining 61 bits are not used. For the other coercions, the 61 bits store a pointer to heap-allocated structures that stores relevant information. Because the number of pointer tags is limited, function, reference, vector, tuple, and recursive coercions are differentiated by a secondary tag stored in the first word of their heap-allocated structure. Reference and vector coercions share the same representation which is a tagged 64-bit pointer to a 16 byte sequence on the heap where the secondary tag is stored in the first 8 bytes and the type that will be used to cast the heap cell is stored in the other 8 bytes.

Queue of suspended casts Figure 1 presents C implementation of the queue in Grift. It is implemented as a global cyclic queue on top of a dynamically-resizing array. The queue is defined as a struct that has five members. The `capacity` variable store the size of the underlying array `casts` while `size` tracks the number of elements currently in the queue. Furthermore, `front` and

```c
typedef struct {
  int capacity, size, front, rear;
  suspended_cast* casts;
} cast_queue;
cast_queue* allocate_cast_queue() {
  cast_queue *cq = GC_NEW(cast_queue);
  cq->casts = GC_MALLOC(sizeof(suspended_cast[CAST_QUEUE_INIT_CAPACITY]));
  cq->capacity = CAST_QUEUE_INIT_CAPACITY;
  cq->front = 0; cq->rear = 0; cq->size = 0;
  return cq;
}
void cast_queue_enqueue(cast_queue* cq, int64_t addr, int64_t ty){
  if (cq->size == cq->capacity) cast_queue_resize(cq);
  cq->casts[cq->rear] = (suspended_cast) { .address = addr, .type = ty };
  cq->rear = (cq->rear + 1) % cq->capacity;
  cq->size++;
}
void cast_queue_dequeue(cast_queue* cq){
  suspended_cast rv = cq->casts[cq->front];
  cq->casts[cq->front] = (suspended_cast) { .address = 0, .type = 0 };
  cq->front = (cq->front + 1) % cq->capacity;
  cq->size--;
}
void cast_queue_resize(cast_queue* cq){
  int capacity = CAST_QUEUE_CAPACITY_MULTIPLE_FACTOR * cq->capacity;
  suspended_cast* casts = GC_MALLOC(sizeof(suspended_cast[capacity]));
  for(int i = 0; i < cq->capacity; ++i)
    casts[i] = cq->casts[(cq->front + i) % cq->capacity];
  if(cq->casts) GC_FREE(cq->casts);
  cq->casts = casts;
  cq->rear = cq->capacity;
  cq->capacity = capacity;
  cq->front = 0;
}
int64_t cast_queue_peek_address(cast_queue* cq) {
  return suspended_cast_get_address(&(cq->casts[cq->front]));
}
int64_t cast_queue_peek_type(cast_queue* cq) {
  return suspended_cast_get_type(&(cq->casts[cq->front]));
}
int cast_queue_is_not_empty(cast_queue* cq) { return cq->size != 0; }
```

FIGURE 1. C implementation of the queue of suspended casts as a cyclic queue on top of a dynamically-resizing array.

```
int64_t greatest_lower_bound(int64_t type1, int64_t type2);
int64_t apply_cast(int64_t value, int64_t type1, int64_t type2, bool suspend_ref_casts);

int64_t * apply_reference_cast(int64_t * address, int64_t type, bool suspend_ref_casts) {
    if (suspend_ref_casts) {
        cast_queue_enqueue(ref_cq, address, type);
    } else {
        int64_t old_rtti = address[0];
        int64_t new_rtti = greatest_lower_bound(old_rtti, type);
        if (old_rtti != new_rtti) {
            int64_t old_value = address[1];
            int64_t new_value = apply_cast(old_value, old_rtti, new_rtti, true);
            address[0] = new_rtti;
            address[1] = new_value;
            apply_suspended_casts();
        }
    }
    return address;
}
```

FIGURE 2. The C code for the `apply_reference_cast` function that casts an address using a type carried by a coercion. Key called functions are forward declared.

`rear` are the indices representing the current front and rear of the queue in `casts`. Finally, `casts` is an array of pairs where elements are 16-byte wide, a pointer to the heap cell that will be cast is stored in the first 8 bytes, and the type that will be used for casting is stored in the other 8. `casts` is dynamically resized every time a new cast is about to be inserted but there is no room for it, i.e. `size == capacity`.

This design is chosen to enable fast enqueue and dequeue operations and to avoid the overhead of queue allocations at every cast site. Alternatively, a runtime optimized for space rather than runtime could choose to represent the queue as a linked list and allocates a new one at every cast site.

Using data structures other than FIFO queues could affect when errors happen, changing the observable behavior.

2. Compiling Referece Operations

Coercion Application Figure 2 presents the `apply_reference_cast` function that casts an address with a type carried by a reference coercion. There are two cases, depending on whether the

```c
void apply_suspended_casts() {
    while (cast_queue_is_not_empty(ref_cq)) {
        int64_t * suspended_cast_address = cast_queue_peek_address(ref_cq);
        int64_t old_rtti = suspended_cast_address[0];
        int64_t suspended_cast_type = cast_queue_peek_type(ref_cq);
        int64_t new_rtti = greatest_lower_bound(old_rtti, suspended_cast_type);
        cast_queue_dequeue(ref_cq);
        if (old_rtti != new_rtti) {
            int64_t old_value = suspended_cast_address[1];
            int64_t new_value = apply_cast(old_value, old_rtti, new_rtti, true);
            suspended_cast_address[0] = new_rtti;
            suspended_cast_address[1] = new_value;
        }
    }
}
```

FIGURE 3. The C code for the `apply_suspended_casts` function that applies all suspended casts in the queue.

current cast was reached from another application of a reference coercion and they are distinguished using the boolean parameter `suspend_ref_casts`. If the current coercion application was indeed reached from an application of another reference coercion, the address and the type carried by the coercion are enqueued and the address is returned. On the other hand, if the application of the reference coercion was not reached from another one, i.e. it is the first reference coercion to be applied at this cast site, then the coercion application proceeds normally by reading the current value from the heap, casting it, and finally writing the new value along with the new RTTI back to the heap. Before returning the address as a result, all suspended casts in the queue are applied until the queue is empty using the `apply_suspended_casts` function (defined in Figure 3). The corresponding functions for applying vector coercions are listed in Appendix C.

Reading and Writing There are two kinds of read and write expressions in λ_S^{ref}. Compiling reference operations from the source language to the appropriate expression in λ_S^{ref} is type-directed and is done during cast insertion (defined in Figure 2 in Chapter 4). Reading and Writing into a statically typed reference is implemented by just `read_value = address[1];` and `address[1] = new_value;`.

On the other hand, reading from and writing into a partially typed reference needs to perform a cast between the RTTI and the type of the reference. Figure 4 presents the functions `read` and `write` that implement that behavior following the reduction specified by DYNREAD and DYNWRITE

```c
int64_t read(int64_t * address, int64_t type) {
    int64_t rtti = address[0];
    int64_t value = address[1];
    return apply_cast(value, rtti, type, false);
}

void write(int64_t * address, int64_t value, int64_t type) {
    int64_t rtti = address[0];
    int64_t new_value = apply_cast(value, type, rtti, true);
    address[1] = value;
    apply_suspended_casts();
}
```

FIGURE 4. The C code for writing a value to a partially-typed address.

rules in λ_S^{ref}. In the case of a write, the written value is cast to the RTTI type and that could cause the addition of suspended casts to the queue. After writing the new value to the heap, suspended casts in the queue are applied until the queue is empty.

3. Optimizations

Optimized Type Representation With monotonic references, new types could be created at runtime, using the greatest lower bound function, and are often checked for equality (e.g. line 10 in Figure 2). Performing structure equality on types takes, in the worst case, a linear time with respect to the number of nodes in the type. To speed up this operation, types are hashconsed [Allen, 1978] instead and structural equality is replaced with pointer equality which is a $O(1)$ operation. To enable hashconsing, the memory layout of a structural type is updated as follows: for $\star$ and base types (atomic types), no need to change anything because they are represented as constant naturals. For structural types of n subparts, one extra field is added to the front. For instance, the function type $(\mathsf{Int}, \mathsf{Int}) \rightarrow \mathsf{Bool}$ is now represented using the following array:

hash code
|

0	1	2	3	4
h	2	Bool	Int	Int

Figure 5 presents the `hashcons_type` function that takes a type and hashcons it. At line 35, atomic types are returned as is. The hash code for other types is computed using a simple hashing

```
intptr_t compute_type_hash_code(intptr_t ty) {
    tag = ty & TYPE_TAG_MASK;
    int64_t * untagged_ty_ptr = (int64_t *) (ty ^ tag);
    switch (tag) {
        case TYPE_REF_TAG:
            int64_t arg_ty = untagged_ty_ptr[TYPE_REF_ARG_INDEX];
            return 1 + get_type_hashcode(arg_ty) * 19;
        case TYPE_VECT_TAG:
            int64_t arg_ty = untagged_ty_ptr[TYPE_VECT_ARG_INDEX];
            return 2 + get_type_hashcode(arg_ty) * 19;
        case TYPE_FN_TAG:
            int64_t return_ty = untagged_ty_ptr[TYPE_FN_RETURN_TYPE_INDEX];
            int64_t hash_code = get_type_hashcode(arg_ty);
            int64_t count = untagged_ty_ptr[TYPE_FN_ARGS_COUNT_INDEX];
            for (int i = TYPE_FN_ARGS_OFFSET; i < count + TYPE_FN_ARGS_OFFSET; ++i) {
                hash_code += get_type_hashcode(untagged_ty_ptr[i]);
                hash_code *= 19;
            }
            return hash_code + 3;
        case TYPE_TUPLE_TAG:
            int64_t hash_code = 0;
            int64_t count = untagged_ty_ptr[TYPE_TUPLE_COUNT_INDEX];
            for (int i = TYPE_TUPLE_ARGS_OFFSET; i < count + TYPE_TUPLE_ARGS_OFFSET; ++i) {
                hash_code += get_type_hashcode(untagged_ty_ptr[i]);
                hash_code *= 19;
            }
            return hash_code + 4;
        case TYPE_MU_TAG:
            int64_t body_ty = untagged_ty_ptr[TYPE_MU_BODY_INDEX];
            return 5 + get_type_hashcode(body_ty) * 19;
    }
}

intptr_t hashcons_type(intptr_t ty) {
    if (ty <= TYPE_MAX_ATOMIC_REP) return ty;
    int64_t hash_code = compute_type_hash_code(ty);
    int64_t htype = hashcons(types_hash_table, ty, hash_code);
    if (ty == htype) {
        int64_t tag = ty & TYPE_TAG_MASK;
        ((int64_t *)(htype ^ tag))[TYPE_HASHCODE_INDEX] = hcode;
    }
    return htype;
}
```

FIGURE 5. C implementation of type hashconsing.

$$\text{Cast result} \quad \mathsf{r} \in \mathcal{R} \quad ::= \quad (\Psi, v) \mid \texttt{error}$$

Cast application function $\boxed{\text{apply-cast}_T : \mathcal{V} \times \mathcal{T} \times \mathcal{T} \to \mathcal{R}}$

$$\begin{aligned}
\text{apply-cast}_T((u\langle c \to d\rangle), (T_1 \to T_2), (T_1' \to T_2')) &= (\emptyset, u\langle (c \to d) \,\text{;}\, ((T_1 \to T_2) \Rightarrow (T_1' \to T_2'))\rangle) \\
\text{apply-cast}_T(u\langle g \,;\, I!\rangle, \star, I') &= \text{apply-cast}(u, g \,\text{;}\, (I \Rightarrow I')) \\
\text{apply-cast}_T(u, T_1 \to T_2, T_1' \to T_2') &= (\emptyset, u\langle T_1 \to T_2 \Rightarrow T_1' \to T_2'\rangle) \\
\text{apply-cast}_T(u\langle g \,;\, I!\rangle, \star, \star) &= (\emptyset, u\langle g \,;\, I!\rangle) \\
\text{apply-cast}_T(a, \mathsf{Ref}\ T, \mathsf{Ref}\ T') &= (((a, T'); \emptyset), a) \\
\text{apply-cast}_T(\langle v_1, v_2\rangle, T_1 \times T_2, T_1' \times T_2') &= ((\Psi \oplus \Psi'), \langle v_1', v_2'\rangle) \\
&\quad \text{if } \begin{array}{l} \text{apply-cast}_T(v_1, T_1, T_1') = (\Psi, v_1'), \\ \text{apply-cast}_T(v_2, T_2, T_2') = (\Psi', v_2') \end{array} \\
\text{apply-cast}_T(v, T, T') &= \texttt{error}
\end{aligned}$$

FIGURE 6. Modified apply-cast to work on types instead of coercions.

algorithm implemented by the `compute_type_hash_code` function. The type is then inserted into the hashconsing table `types_hash_table` using the `hashcons` function. `types_hash_table` is implemented by a hash table on top of a dynamically-resizing array and uses the chaining strategy for collision resolution. The implementation is in Appendix D. The `get_type_hashcode` function retrieves the hash code of a type from an already hashconsed type.

Lazy Coercions When casting an address, a new coercion is created from the old and new RTTIs, only to be applied immediately after and discarded right away. To avoid this unnecessary allocation, types are used instead for casting (e.g. line 12 in Figure 2) and a coercion is created only if there is a need to store the coercion or to compose it with another one.

Figure 6 presents the definition of the apply-cast$_T$ function that expects type arguments instead of a coercion. In the case of casting a proxied function to another function type, apply-cast$_T$ creates a coercion between the input types and composes it with the old coercion in the proxy. Furthermore, in the case of casting from $\star$ to an injectable type, apply-cast$_T$ creates a new coercion between the type held by the injection and the target type and composes it with the other coercion g in the sequence in the injected value. The result of composition is used to coerce the injected value using the apply-cast function. Finally, in the case of casting an unproxied function, apply-cast$_T$ creates a coercion out of the input types and wraps the function in a proxy with that coercion. All other cases are standard.

CHAPTER 8

Performance Evaluation

In this performance evaluation, I seek to answer a number of research questions regarding the runtime overheads associated with gradual typing.

(1) **What is the time cost of achieving space efficiency with coercions?** (Section 2)
(2) **What is the overhead of gradual typing?** (Sec. 3) I subdivide this question into the overheads on programs that are (a) statically typed, (b) untyped, and (c) partially typed.
(3) **Do monotonic references eliminate overheads associated with gradual typing from statically typed code?** (Section 5)
(4) **What is the overhead of using monotonic references in partially typed and untyped code?** (Sections 6 and 7)

Of course, to answer research question 2 definitively we would need to consider all possible implementations of gradual typing. Instead, I only answer this question for the concrete implementation Grift.

1. Experimental Methodology for Evaluating Space-Efficiency

Benchmarks from a number of sources are used: the well-known Scheme benchmark suite (R6RS) used to evaluate the Larceny [Hansen and Clinger, 2002] and Gambit [Feeley, 2014] compilers, the PARSEC benchmarks [Bienia et al., 2008], the Computer Language Benchmarks Game (CLBG), and the Gradual Typing Performance Benchmarks [GTP, 2018]. I do not use all of the benchmarks from these suites due to the limited number of language features currently supported by the Grift compiler. In addition to the above benchmarks, two textbook algorithms are included: matrix multiplication and quicksort. I chose quicksort in particular because it exhibits catastrophic overheads. The benchmarks that are used:

sieve: This program finds prime numbers using the Sieve of Eratosthenes. The program includes a library for streams implemented using higher-order functions and represents streams using equirecursive types.

n-body: Models the orbits of Jovian planets, using a simple symplectic-integrator.

tak: This benchmark, originally a Gabriel benchmark, is a triply recursive integer function related to the Takeuchi function. It is a test of function calls and arithmetic.

ray: Ray tracing a scene, 20 iterations. It is a test of floating point arithmetic.

blackscholes: This benchmark calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation. There is no closed-form expression for the Black-Scholes equation and as such it must be computed numerically.

matmult: A triply-nested loop for matrix multiplication, with integer elements.

quicksort: The quicksort algorithm on already-sorted (worst-case) input, with integer arrays.

fft: Fast Fourier Transform on real-valued points. A test of floating point numbers.

Porting the Benchmarks.

The benchmarks are ported to the GTLC+, OCaml, Typed Racket, Chez Scheme, and Gambit. For the R6RS and CLBG benchmarks, types annotations have been added and tail recursive loops have been converted to an iterative form. For the Blackscholes benchmark, the GTLC+ types are used which are the closest analog to the representation used in the original C benchmark. In some cases the choice of representation in GTLC+ has a more specialized representation than in its original source language, in these cases the original benchmark are altered to make the comparison as close as possible. For Chez Scheme and Gambit the safe variants of fixnum and floating point specialized math operations are used, but for Racket and Typed-Racket there is only the option of safe and well-performing floating point operators. For fixnums the polymorphic math operations are used. In OCaml, the `int` and `float` types are used which correspond to unboxed 63 bit integers and boxed double precision floating point numbers respectively. In all languages, internal timing is used so that any differences in runtime initialization do not count towards runtime measurements. No attempt is made to account for the difference in garbage collection between the languages. Note that Grift uses an off the shelf version of the Boehm-Demers-Weiser conservative garbage collector that implements a generational mark-sweep algorithm [Boehm and Weiser, 1988, Demers et al., 1990]. The source code for all benchmarks is available at URL:`https://github.com/Gradual-Typing/benchmarks`.

Experimental Setup.

The experiments were conducted on an unloaded machine with a 4-core Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz processor with 8192 KB of cache and 16 GB of RAM running Red Hat

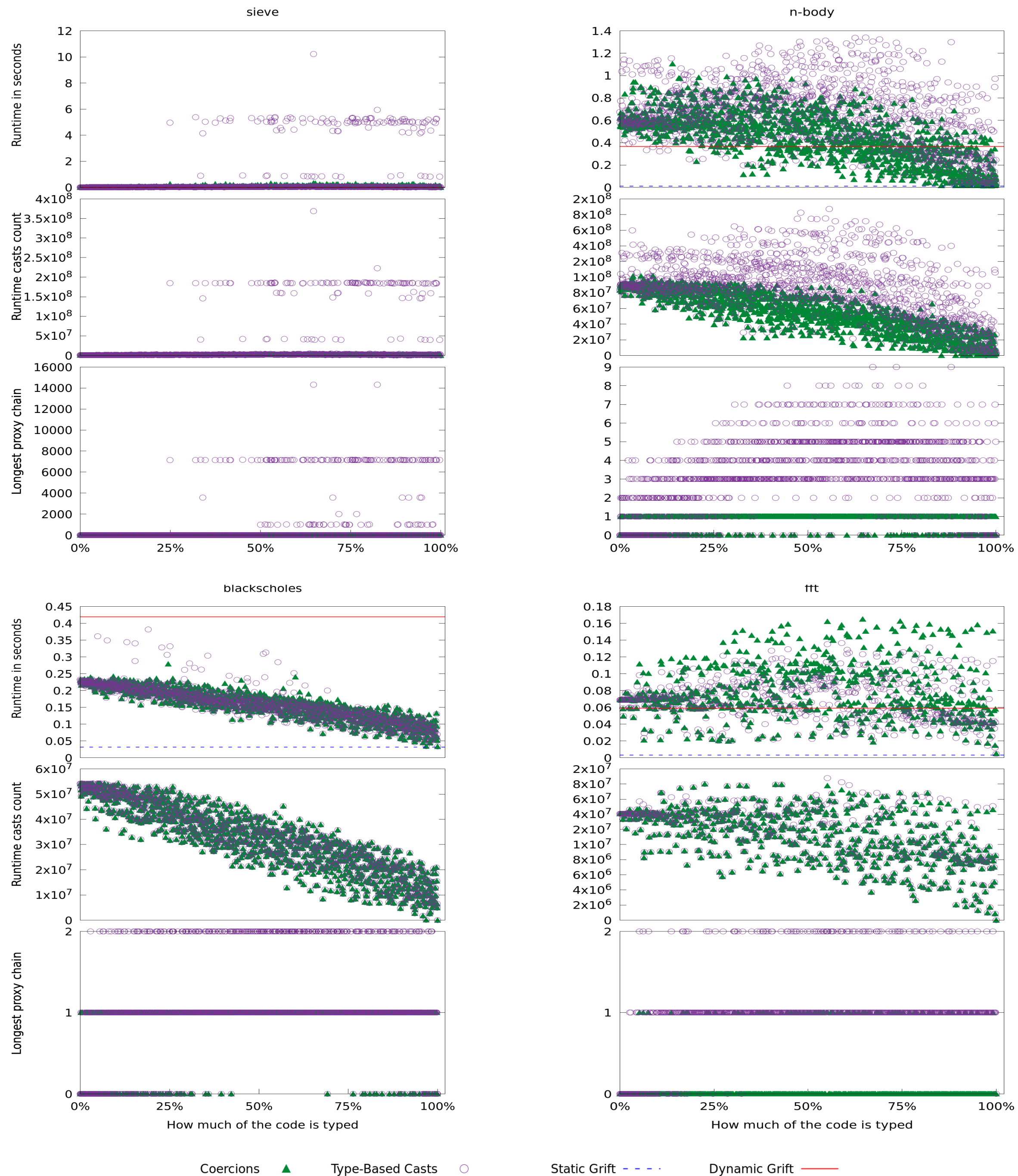


FIGURE 1. Grift with coercions is compared to Grift with type-based casts across partially typed-programs to evaluate the cost of space-efficiency.

4.8.5-16. The C compiler was Clang 7.0.1, the Gambit compiler is version 4.9.3, Racket is version 7.2, and Chez Scheme is version 9.5.3. All time measurements use real time and I report the mean of 5 repeated measurements.

Measuring the Performance Lattice.

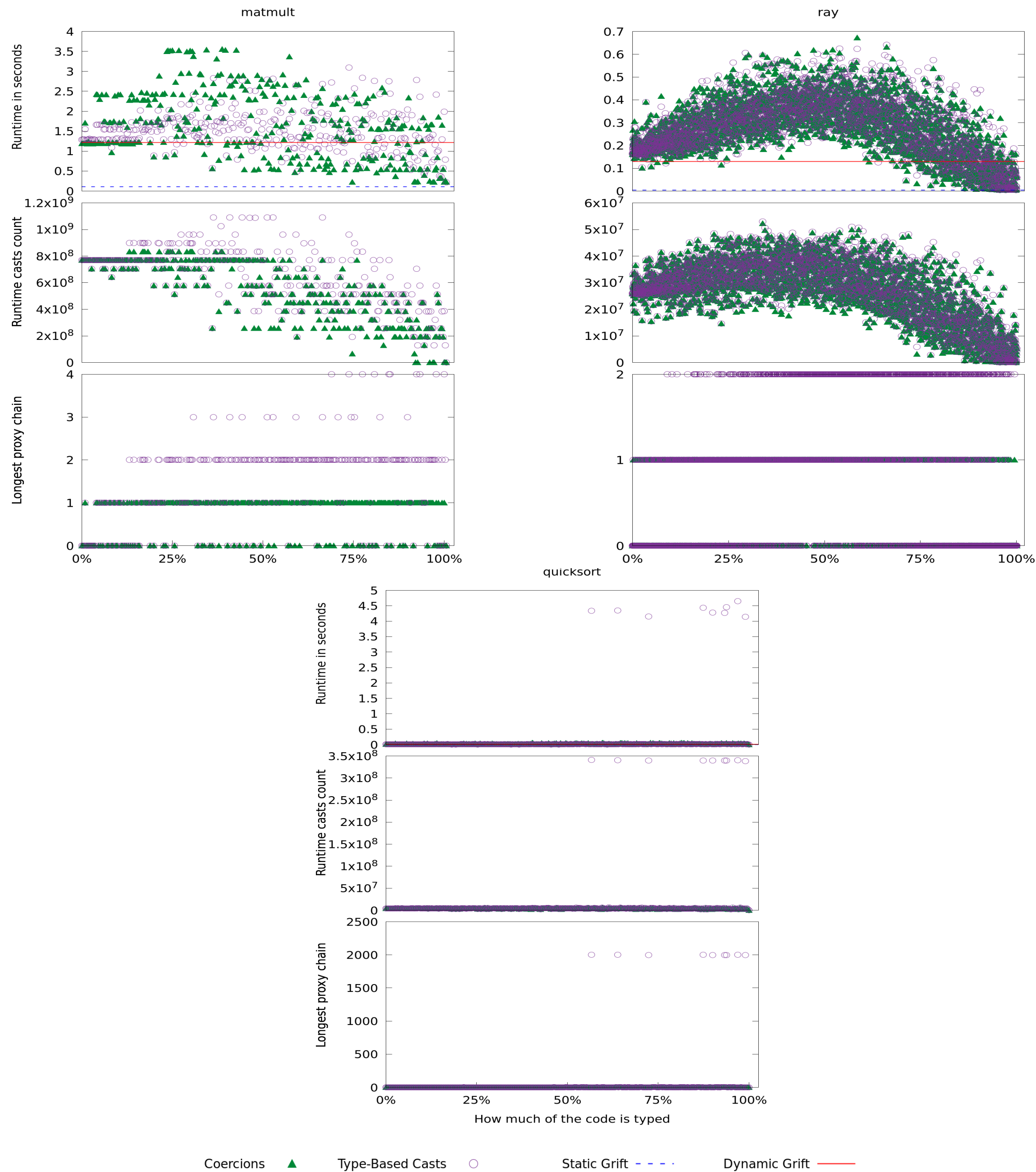


FIGURE 1. Grift with coercions is compared to Grift with type-based casts across partially typed-programs to evaluate the cost of space-efficiency.

It was observed that evaluating the performance of implementations of gradually typed languages is challenging because one needs to consider not just one configuration of each program, but the many configurations of a program that can be obtained by adding/removing type annotations [Takikawa et al., 2016]. For languages with coarse-grained gradual typing, one considers all the combinations of making each module typed or untyped, so there are 2^m configurations of

Benchmark	Input description	Partially typed	Partially typed (coarse)	Typed/Untyped
sieve	Max number	1999	500	$9,999$
n-body	Cycles count	$100,000$	$100,000$	$100,000$
tak	(x, y, z)	$(40, 20, 12)$	$(40, 20, 12)$	$(40, 20, 12)$
ray	Scene size	100×100	100×100	100×100
blackscholes	Options count	$4,096$	$4,096$	$16,384$
matmult	Matrices size	400×400	400×400	400×400
quicksort	Array length	$1,000$	$1,000$	$10,000$
fft	Sequence length	$65,536$	$65,536$	$16,777,216$

FIGURE 2. List of inputs to each benchmark.

the m modules. The situation for languages with fine-grained gradual typing, as is the case for GTLC+, is considerably more difficult because any type annotation, and even any node within a type annotation, may be changed to `Dyn`, so there are millions of ways to add type annotations to these benchmarks.

Greenman and Migeed [2018] show that sampling even a linear number of configurations (with respect to program size) gives an accurate characterization of the performance of the exponential configuration space. For our experiments on partially typed programs, I follow the same approach and show the results for a linear number of randomly sampled configurations for each benchmark.

My sampling algorithm takes as inputs a statically-typed program, the number of samples, and the number of bins to be uniformly sampled. It creates a list of associations between source locations and type annotations, and shuffles the list to ensure randomness. The algorithm then proceeds to pick new gradual configurations of each static type, but constrains the overall program's type precision to fall within a desired bin. These newly generated gradual types are then used to generate a gradual configuration of the original program by inserting the gradual types at the source locations where the static types where originally found. The algorithm iterates selecting an equal number of samples for each bin until the desired number of samples have been generated.

There are many partially typed configurations where each one is ran multiple times, and it is often the case that the performance of partially typed configurations is worse than the statically typed one, so to make our evaluation feasible (i.e. to finish in days instead of weeks), the input sizes to partially typed configurations are smaller than the ones to the statically typed configuration. Figure 2 presents the inputs to each benchmark in each case.

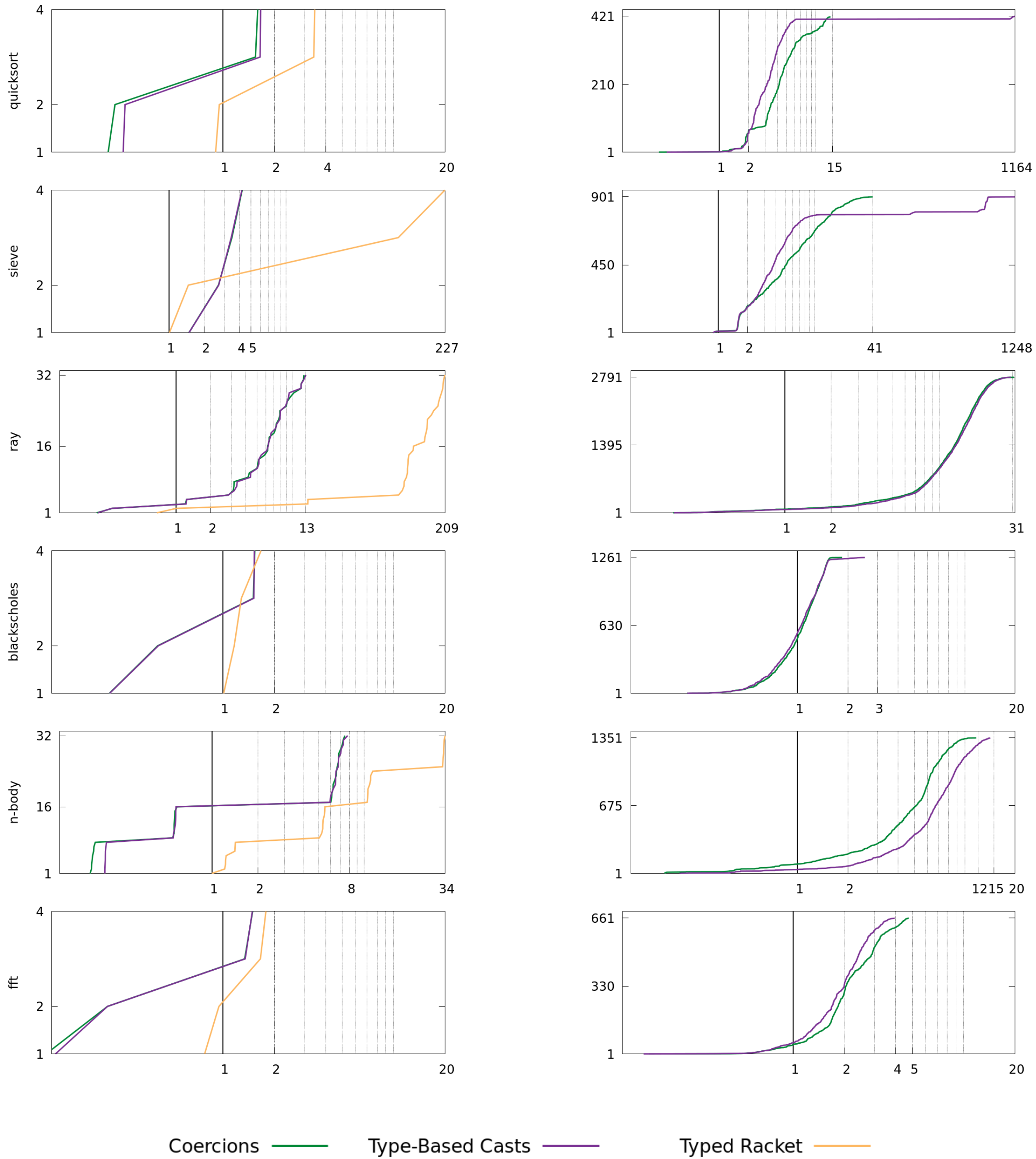


FIGURE 3. The cumulative performance of Grift and Typed Racket on partially typed configurations. The x-axis represents the slowdowns with respect to Racket. The y-axis is the total number of configurations. The plots on the left are for coarse-grained configurations whereas the plots on the right are for fine-grained configurations (so it is a different view on the same data as in Figure 1). These results show that coercions eliminate the catastrophic overheads (quicksort, sieve) of type-based casts and that Grift has less incidental overhead than Typed Racket (sieve, ray, n-body).

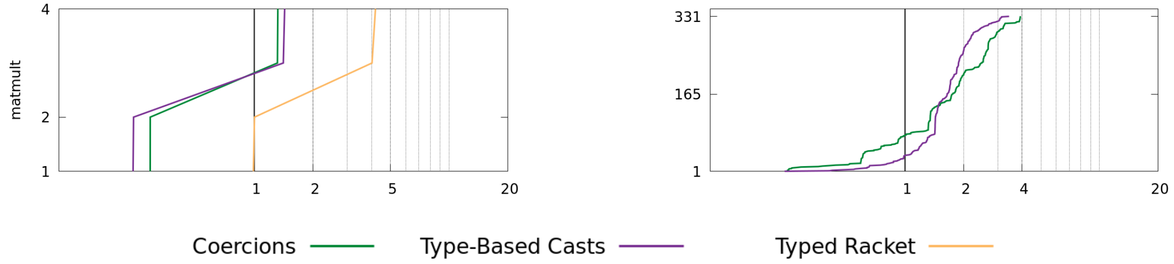


FIGURE 3. The cumulative performance of Grift and Typed Racket on partially typed configurations. The x-axis represents the slowdowns with respect to Racket. The y-axis is the total number of configurations. The plots on the left are for coarse-grained configurations whereas the plots on the right are for fine-grained configurations (so it is a different view on the same data as in Figure 1). These results show that coercions eliminate the catastrophic overheads (quicksort, sieve) of type-based casts and that Grift has less incidental overhead than Typed Racket (sieve, ray, n-body).

2. The Runtime Cost of Space Efficiency

Figure 1 compares the performance of Grift with type-based casts to Grift with coercions, to measure the runtime cost (or savings) of using coercions to obtain space efficiency. The comparison between the two approaches is on partially typed configurations of the benchmarks.

In Figure 1, for each benchmark there are three plots that share the same x-axis, which varies the amount of type annotations in the program, from 0% on the left to 100% on the right. In the first plot, the y-axis is the absolute runtime in seconds. In the second, it is the number of runtime casts that were executed, and in the third plot, the y-axis it is the length of the longest chain of proxies that was accessed at runtime. The line marked Dynamic Grift indicates the performance of Grift (using coercions) on untyped code. That is, on code in which every type annotation is $\star$ and every constructed value (e.g. integer constant) is explicitly cast to $\star$. The line marked Static Grift is the performance of the Static Grift compiler on fully typed code. Static Grift is a variant of Grift that is statically typed, with no support for (or overhead from) gradual typing (see Section 3).

The sieve and quicksort benchmarks elicits very long chains of proxies on some configurations, which in turn causes catastrophic overhead for type-based casts. Indeed, the plot concerning the longest proxy chains for sieve in Figure 1, shows that the sieve configurations with catastrophic performance are the ones that accumulate proxy chains of length greater than $2,000$. Likewise, Takikawa et al. [2016] report over $100\times$ overhead on sieve for Typed Racket. In contrast, the coercion-based approach successfully eliminates these catastrophic slowdowns in sieve and quicksort. The scale of the figures makes it hard to judge their performance in detail as the runtimes are so

low relative to the type-based casts. Coercions are 0.32× to 87× faster than type-based casts on sieve.

The n-body benchmark is interesting in that it elicits only mild space efficiency problems, with proxy chains up to length 9, and this corresponds to a mild increase in performance of coercions relative to type-based casts. Coercions are 0.38× to 39× faster than type-based casts on n-body.

In benchmarks that do not elicit space efficiency problems, there is a general trend that coercions are roughly equal in performance to type-based casts.

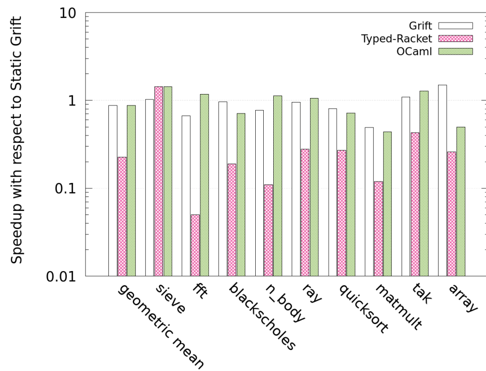
> **Answer to research question** (1)**:** On benchmarks that do not induce long proxy chains, there is a mild speedup and sometimes a mild slowdown for coercions compared to type-based casts. However, on benchmarks with long proxy chains, coercions eliminate the catastrophic overheads.

3. Gradual Typing Overhead and Comparisons

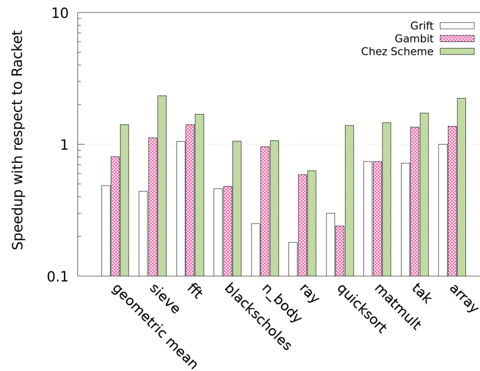
The purpose of this section is to answer research question 2, i.e., what is the overhead of gradual typing? We want to understand which overheads are *inherent* (an necessary part of sound gradual typing) as opposed to *incidental* (unnecessary overheads that could be removed). *incidental* overheads are identified by comparing Grift to other implementations, and reasoning about what these comparisons say about gradual tying as a whole, and our implementations of Gradual Typing.

In Section 3.0.1 Grift is compared to statically typed programming languages. To isolate the overheads associated with gradual typing from the rest of our implementation, a variant of the Grift compiler is added, named Static Grift, that requires the input program to be statically typed and does not generate any code in support of gradual typing. Comparing Grift to Static Grift shows that gradual typing introduces some overhead (though the comparison doesn't say if the overhead is inherent to gradual typing), and comparing Static Grift to OCaml and Typed Racket shows that Static Grift has reasonable performance for a statically typed language.

In Section 3.0.2 overheads of gradual typing on dynamically typed code are examined. Grift is compared against Racket, Gambit, and Chez Scheme. It shows that while Grift is in the ball park of dynamically typed programming languages, it experiences some incidental overheads. The overhead is conjectured to be incidental because Typed Racket avoids similar overheads, in their



(A) Statically Typed Programs



(B) Untyped Programs

FIGURE 4. A comparison of the speedup on typed and untyped programs of Grift. For typed programs, speedup is measured wrt. Static Grift and Grift is compared against OCaml and Typed Racket. Grift shows some overhead compared to Static Grift but is on par with OCaml and significantly faster than Typed Racket. For untyped programs, speedup is measured wrt. Racket and Grift is compared to Gambit and Chez Scheme. Grift's performance is roughly half that of Racket, Gambit, and Chez Scheme's.

dynamically typed implementation. This is as expected because Grift does not implement the many general purpose optimizations that are in these other systems.

Section 3.0.3 inspects the overheads of gradual typing on partially typed code. Grift shows that space-efficient coercions avoid the catastrophic overheads associated with gradual typing. This demonstrates that these catastrophic overheads are incidental to gradual typing. On the other hand, I conjecture that constant-factor overheads associated with composing coercions is inherent for gradually typed programming languages with structural types. However, I also think there is still some remaining constant-factor overhead that is incidental and could be eliminated.

3.0.1. *Evaluation on Statically Typed Programs.* Figure 4a shows the results of evaluating the speedup of Grift with respect to Static Grift on statically typed benchmarks. The performance of Grift sometimes dips to 0.49× that of Static Grift. To put the performance of Grift in context, it is comparable to OCaml and better than fully static Typed Racket.

> **Answer to research question (2 a):** the performance of Grift on statically typed code is often reasonable and is on par with OCaml but can dip to 0.49× with respect to Static Grift on array-intensive benchmarks.

I think that most of the differences between Grift and Static Grift can be attributed to the checks for proxies in operations on mutable arrays and I conjecture that this performance overhead is inherent to the standard semantics for gradual typing.

3.0.2. *Evaluation on Untyped Programs.* Figure 4b shows the results of evaluating the speedup of Grift with respect to Racket on untyped configurations of the benchmarks. The figure also includes results for Gambit and Chez Scheme. The performance of Grift is generally lower than Racket, Gambit, and Chez Scheme on these benchmarks, which is unsurprising because Grift does not perform any general-purpose optimizations. This experiment does not tease apart which of these performance differences are due to gradual typing per se and which of them are due to orthogonal differences in implementation, e.g., ahead-of-time versus JIT compilation, quality of general-purpose optimizations, etc. Thus I can draw only the following conservative conclusion.

> **Answer to research question (2 b)** the overhead of Grift on untyped code is currently reasonable but there are still some constant-factor improvements to be made.

3.0.3. *Evaluation on Partially Typed Programs.* To answer research question (2 c), i.e., "what is the overhead of gradual typing for partially typed code?", consider the results in Figure 3. The left-hand column shows the performance of Grift (with coercions and type-based) and for Typed Racket on coarse-grained configurations, in which each module is either typed or untyped. The right-hand column shows the performance results for Grift on fine-grained configurations.

The cumulative performance plots shown in Figure 3 indicate how many partially typed configurations perform within a certain performance range. The x-axis is log-scale slowdown with respect to Racket and the y-axis is the total number of configurations. For instance, to determine how many configurations perform within a 2× slowdown of Racket, read the y-axis where the corresponding line crosses 2 on the x-axis. Lines that climb steeply as they go to the right exhibit good performance on most configurations whereas lines that climb slowly signal poorer performance.

The first observation, based on the right-hand side of Figure 3, that we take away is that coercions reduce overheads in the benchmarks that cause long chains of proxies (quicksort, sieve, and n-body). This can be seen in the way the green line for coercions is to the left, sometimes far to the left, of the purple line for type-based casts. This demonstrates that catastrophic overheads are incidental (just a property of type-based casts and related technologies), and not inherent to gradual typing per se.

The second observation, based on the left-hand side of Figure 3, is that Typed Racket, with its contract-based runtime checks, and even with collapsible contracts[Feltey et al., 2018], incurs significant incidental overheads. The yellow line for Typed Racket is far to the right of Grift on sieve, ray, and n-body.

Third, it is interesting to compare the left-hand column to the right-hand column. Many researchers have speculated regarding whether fine-grained or coarse-grained gradual typing elicit more runtime overhead. The data suggests that there is not a simple answer to this question. From a syntactic point of view, it is certainly true that coarse-grained yields fewer opportunities for casts to be inserted. However, sometimes a single cast can have a huge impact on runtime, especially if it appears in a hot code region or if it wraps a proxy around a value that is used later in a hot code region. For example, in the sieve, ray, and n-body benchmarks, there is considerable overheads for Typed Racket on many coarse-grained configurations. On the other hand, fine-grained gradual typing provides many more opportunities for configurations to elicit more runtime overhead. For example, compare the left and right-hand sides for quicksort and sieve, in which there are catastrophic slowdowns for type-based casts in the fine-grained configurations but not in the coarse-grained configurations.

> **Answer to research question (2 c):** the overhead of Grift on partially typed code is no longer catastrophic, but there is still room for improvement.

4. Experimental Methodology for Evaluating Monotonic References

In Section 5, the performance of monotonic references is evaluated on statically typed programs.

Grift with proxied and monotonic references is compared against Static Grift to measure the runtime overhead caused by proxied references and to verify the claim that monotonic references eliminate such overheads. The results of this experiment show that proxied references causes mild runtime overheads on statically typed code and monotonic references do not cause any such overheads.

In Section 6, the performance of monotonic references is compared against proxied references on partially typed programs. The results of this experiment show that, on average, monotonic references perform better than proxied references.

Figure 5 presents the inputs to each benchmark in each case.

Experimental Setup.

Benchmark	Input description	Partially typed	Statically Typed
sieve	Max number	500	$9,999$
n-body	Cycles count	$2,000$	$100,000$
tak	(x, y, z)	$(40, 20, 12)$	$(40, 20, 11)$
ray	Scene size	100×100	1000×1000
blackscholes	Options count	$4,096$	$16,384$
matmult	Matrices size	400×400	400×400
quicksort	Array length	$1,000$	$10,000$
fft	Sequence length	$65,536$	$16,777,216$

FIGURE 5. List of inputs to each benchmark.

The experiments were conducted on an unloaded machine with a 4-core Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz processor with 8192 KB of cache and 16 GB of RAM running Red Hat 7.7. The C compiler was Clang 9.0.1 and Grift is configured to use coercions by default to represent casts efficiently. All time measurements use real time and the mean of 5 repeated measurements is reported.

5. Efficient Statically Typed Code

Figure 6 shows the slowdown of different configurations of Grift with respect to *Static Grift*. Static Grift is used as the baseline because its performance is the best performance one could hope for from Grift on statically typed programs. Grift configurations vary in which reference semantics is used (Monotonic vs Proxied) and whether function proxies are represented as closures [Siek and Garcia, 2012] (UniformClosure vs NoUniformClosure). As discussed earlier in the dissertation, representing function proxies as closures enables uniform calling convention for functions and proxies, so dynamic dispatch is no longer needed. Grift with monotonic references and closure representation of function proxies matches the performance of Static Grift while Grift with proxied references is up to 1.56× slower than Static Grift.

> **Answer to research question** (3)**:** On all benchmarks, the runtime of Grift with monotonic references, and with closure representation of function proxies, is roughly the same as the runtime of Static Grift, so yes monotonic references eliminate overheads associated with gradual typing from statically typed code.

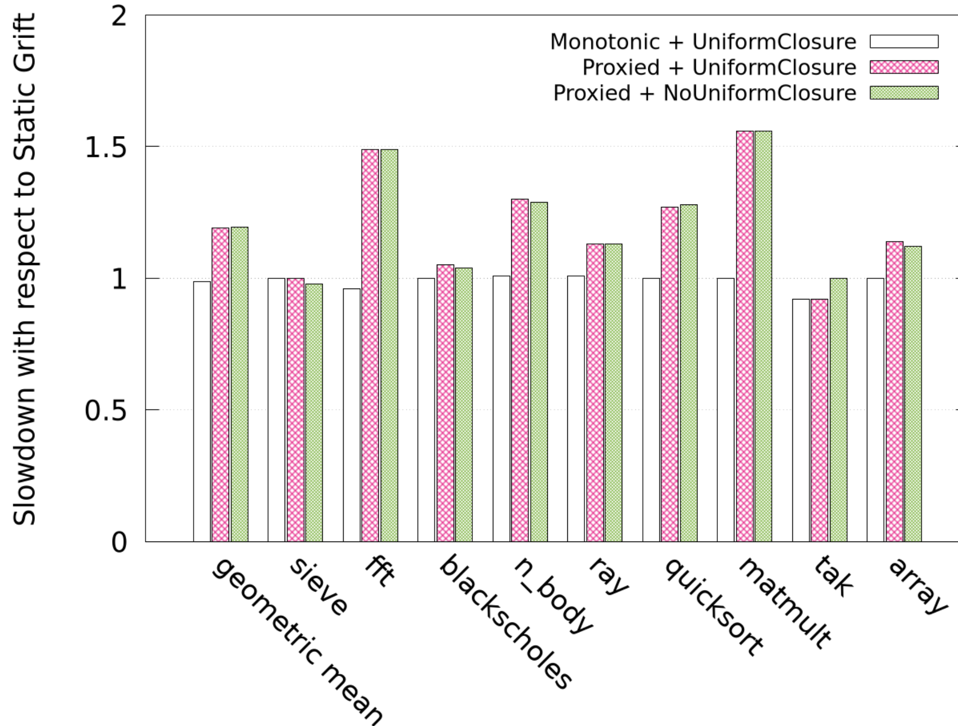


FIGURE 6. A comparison of the slowdown on statically typed Grift programs wrt Static Grift. The performance of Static Grift is matched by that of Grift with monotonic references and closure representation of function proxies.

6. Evaluation of Monotonic References on Partially Typed Programs

To answer research question 4, i.e., "What is the overhead of using monotonic references in partially typed code?", consider the results in Figures 7 and 8. In Figure 7, each plot shows the average runtime (in seconds) per a partially typed configuration on the Y-axis and the percentage of typed code regions in it on the X-axis. In addition to plotting the runtimes of monotonic and proxied references on partially typed configurations, two more lines are shown. The first one is the red line and it represents the average runtime of the fully untyped configuration compiled using Grift with proxied references. The other line is the dashed blue and represents the average runtime of Static Grift on the statically typed configuration. Grift is configured with the UniformClosure option (described in Section 5) in both configurations. In Figure 8, the Y-axis represents the summation of the average runtimes of all partially typed configurations.

The first observation is that monotonic and proxied references have roughly the same performance on partially typed configurations of ray and fft and our take away is that in some cases there is no significant performance difference between using either of them on partially typed code.

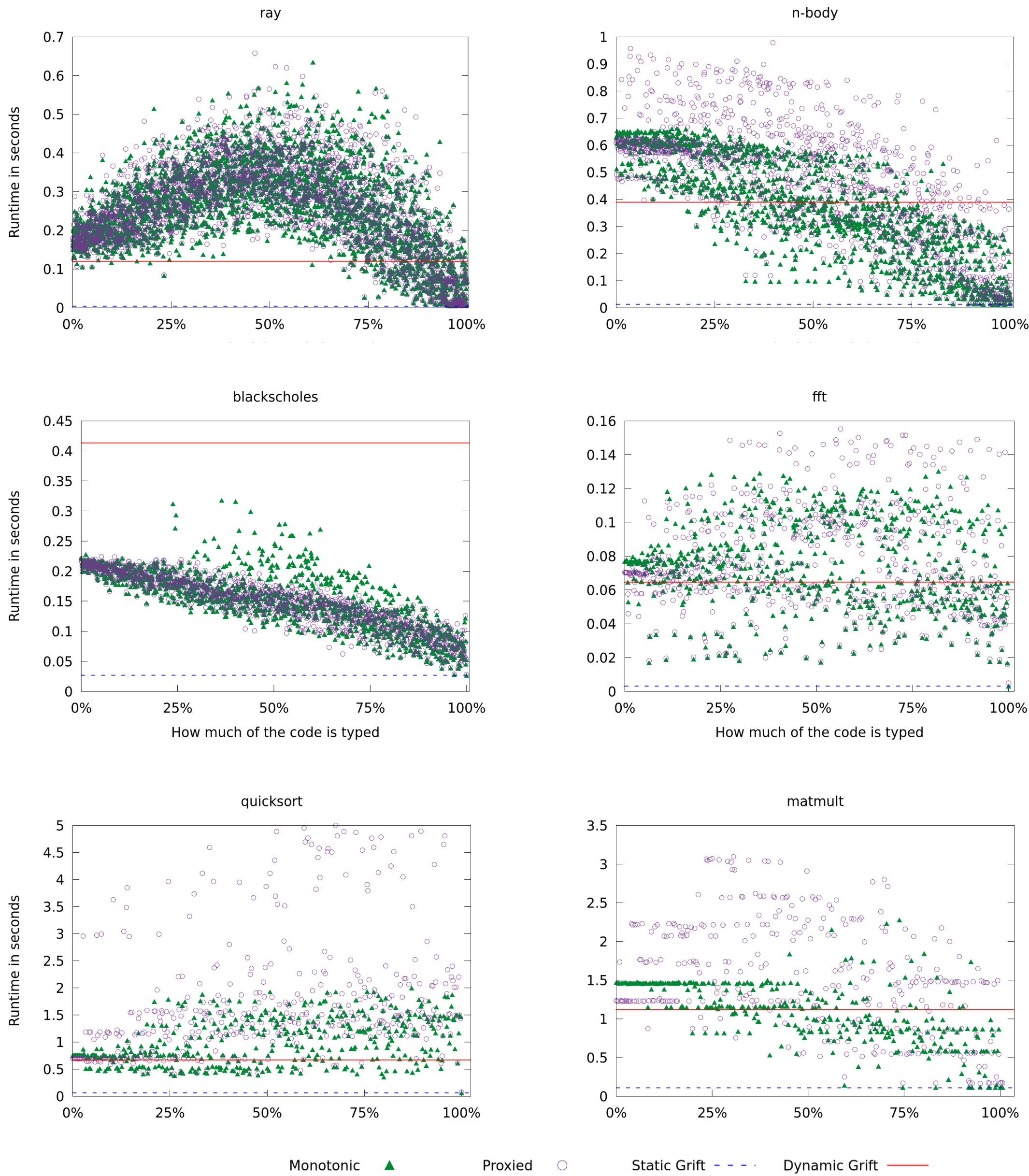


FIGURE 7. Grift with monotonic references is compared against Grift with proxied references across partially typed-programs to evaluate the overhead of monotonic references. Monotonic references performs better on n-body, quicksort, and matmult. They are about the same on fft and ray.

Furthermore, monotonic references is significantly faster than proxied references on partially typed configurations of quicksort, n-body, and matmult. On the other hand, monotonic references is slightly slower than proxied references on partially typed configurations of blackscholes. However, it is still much faster than the fully untyped configuration. Our take away is that monotonic references is faster than proxied references in most cases.

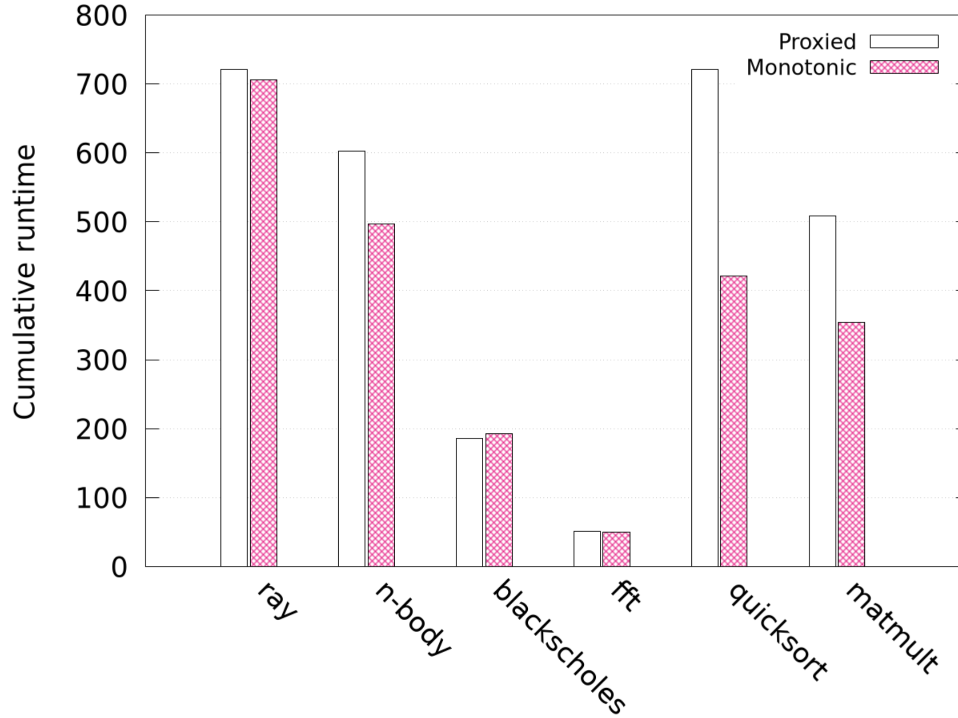


FIGURE 8. Comparison between Grift with monotonic references and proxied references. The Y-axis is the cumulative runtime in seconds across all partially typed configurations. Monotonic references is slightly slower than proxied references on blackscholes and is about the same or faster on the rest.

To understand the root causes of the slowdowns, let's revisit the casting semantics of monotonic and proxied vectors. Casting a monotonic vector is a $O(n)$ operation where a new RTTI gets computed and every element in the vector gets cast to it. Applying multiple casts to a monotonic vector could be expensive but once the RTTI becomes a fully static type, the vector can no longer be cast. On the other hand, casting a proxied vector is a much cheaper operation where either the cast vector gets wrapped in a proxy if it was not already, or the old and new proxies are composed.

Answer to research question (4)**:** In most benchmarks Grift with monotonic references matches or has a better performance than proxied references on partially typed code. Occasionally there is some overhead, but it does not cause worse performance than that of the untyped configuration.

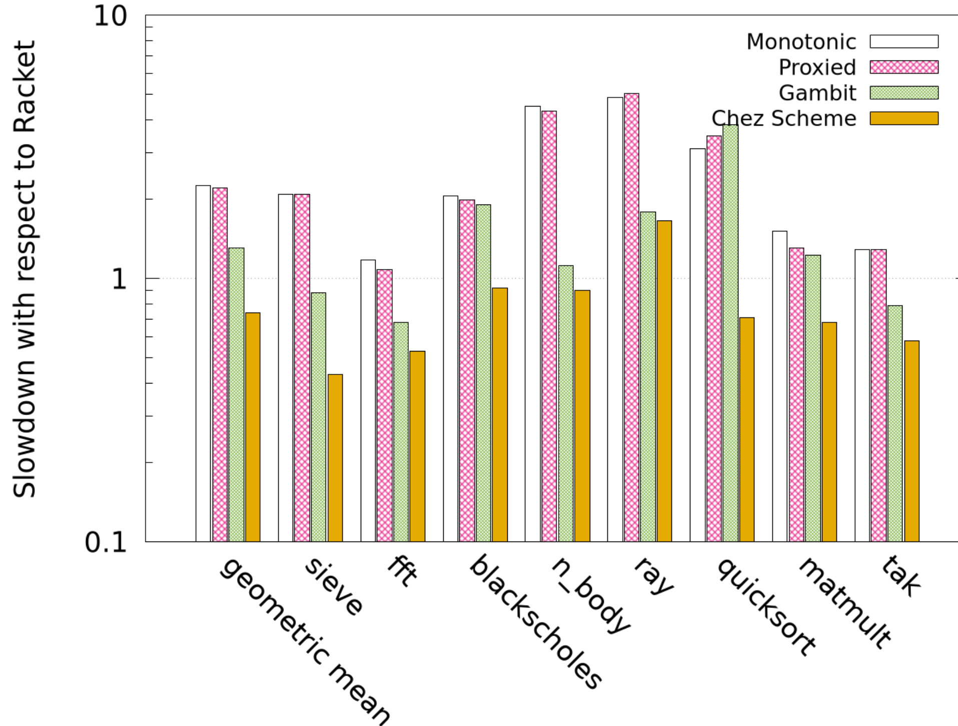


FIGURE 9. Comparison between Grift with monotonic references and proxied references on untyped programs. Furthermore, the comparison include Racket, Gambit Scheme, and Chez Scheme as representatives of dynamically typed languages. The Y-axis is the slowdown with respect to Racket. Monotonic and Proxied references have roughly the same performance on untyped programs.

7. Evaluation of Monotonic References on Untyped Programs

To answer the second part of the research question 4, i.e., "What is the overhead of using monotonic references in untyped code?", consider the results in Figures 9. The figure shows the results of evaluating the slowdown of Grift with monotonic references with respect to Racket on untyped configurations of the benchmarks. The figure also includes results for Grift with proxied references, Gambit, and Chez Scheme. The performance of monotonic references matches that of proxied references but is generally lower than Racket, Gambit, and Chez Scheme on these benchmarks, which is, again, unsurprising because Grift does not perform any general-purpose optimizations. Thus, monotonic references does not introduce additional overheads in untyped programs.

Answer to research question (4)**:** Monotonic references perform roughly as well as proxied references on untyped code.

8. Threats to Validity

One concern with these experiments is that the GTLC+ is a small language compared to other programming languages. For example, both Typed Racket and OCaml support separate compilation, tail-call optimization, unions, and polymorphism.The Grift compiler supports none of these. This is likely one of the reasons that Static Grift has performance on par with OCaml. Extending Grift to support these features will likely introduce overheads. The question relevant to this paper is whether there will be any additional overheads that arise from the interactions between gradual typing and the new features. For example, adding polymorphism with relational parametricity would require runtime sealing, which could incur significant overhead.

Another concern with being a small language is that the language features available in GTLC+ limit benchmarks that are supported. This has led to a numerically leaning suite of benchmarks. Of the 8 benchmarks presented in this paper, 6 feature a significant amount of arithmetic. There is a possibility that Grift performs really well on arithmetic benchmarks, but not as well on other types of benchmarks.

CHAPTER 9

Conclusions

Monotonic references is a design for gradually-typed mutable references that improves over the traditional design as they incur zero-overhead for reference operations in statically-typed code regions while maintaining the full expressiveness of gradual typing. However, prior work on monotonic references presented reduction semantics that writes cast expressions to the heap that reduce using a small-step reduction relation. It is not straightforward to implement such a process efficiently in a compiler and runtime system for gradually-typed languages. Furthermore, earlier work did not guarantee space-efficiency, a key property to ensure runtime efficiency in implementations of gradually-typed languages.

In this dissertation, I present novel dynamic semantics for monotonic references that is space-efficient and that only writes values to the heap. The former is accomplished by identifying a normal form for coercions with monotonic references and by extending the composition function accordingly. Moreover, in my dynamic semantics, a cast is applied to a value using a simple recursive function that suspends inner casts of references by putting them in a queue. This process is straightforward to implement in a runtime system for a gradually-typed language. Furthermore, I describe my implementation in Grift, a compiler for a gradually-typed programming language, along with a few related optimizations. Finally, performance evaluation shows that my implementation eliminates all overheads associated with gradually typed references from statically typed code without adding significant overhead in partially typed or untyped code.

Bibliography

2017. URL https://flow.org/en/.

Gradual typing performance benchmarks, October 2018. URL https://pkgs.racket-lang.org/package/gtp-benchmarks.

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for All. In *Symposium on Principles of Programming Languages*, January 2011.

Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for free for free: Parametricity, with and without types. In *International Conference on Functional Programming*, ICFP, September 2017.

John Allen. *Anatomy of LISP*. McGraw-Hill, Inc., New York, NY, USA, 1978. ISBN 0-07-001115-X.

Christopher Anderson and Sophia Drossopoulou. BabyJ - from object based to class based programming via types. In *WOOD '03*, volume 82. Elsevier, 2003.

Andrew W. Appel. *Compiling with continuations*. Cambridge University Press, New York, NY, USA, 1992. ISBN 0-521-41695-7.

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.

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University, January 2008.

Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to c\#. In Theo D'Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, pages 76–100, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14107-2.

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.

Małgorzata Biernacka and Olivier Danvy. Towards compatible and interderivable semantic specifications for the Scheme programming language, Part II: Reduction semantics and abstract machines. In *Semantics and Algebraic Specification: Essays dedicated to Peter D. Mosses on the occasion of his 60th birthday*, pages 186–206, 2009.

Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: Robust, concurrent, extensible scripting on the jvm. In *ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 117–136, 2009.

Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, September 1988. ISSN 0038-0644. doi: 10.1002/spe.4380180902. URL `http://dx.doi.org/10.1002/spe.4380180902`.

Gilad Bracha. Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. Migrating gradual types. *Proc. ACM Program. Lang.*, 2(POPL):15:1–15:29, December 2017. ISSN 2475-1421. doi: 10.1145/3158103. URL `http://doi.acm.org/10.1145/3158103`.

Luca Cardelli. The functional abstract machine. Technical Report TR-107, AT&T Bell Laboratories, 1983.

Luca Cardelli. Compiling a functional language. In *ACM Symposium on LISP and Functional Programming*, LFP '84, pages 208–217. ACM, 1984.

Giuseppe Castagna and Victor Lanvin. Gradual typing with union and intersection types. In *International Conference on Functional Programming*, 2017.

Olaf Chitil. Practical typed lazy contracts. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 67–76, New York, NY, USA, 2012. ACM.

Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 261–269, New York, NY, USA, 1990. ACM. ISBN 0-89791-343-4. doi: 10.1145/96709.96735. URL `http://doi.acm.org/10.1145/96709.96735`.

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.

Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In *ESOP*, 2012.

Marc Feeley. Gambit-c: A portable implementation of scheme. Technical Report v4.7.2, Universite de Montreal, February 2014.

Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. Collapsible contracts: Fixing a pathology of gradual typing. *Proc. ACM Program. Lang.*, 2 (OOPSLA):133:1–133:27, October 2018. ISSN 2475-1421. doi: 10.1145/3276503. URL `http://doi.acm.org/10.1145/3276503`.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming*, ICFP, pages 48–59, October 2002a.

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. Technical Report NU-CCS-02-05, Northeastern University, 2002b.

Cormac Flanagan. Hybrid type checking. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 245–256, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2. doi: 10.1145/1111037.1111059. URL `http://doi.acm.org/10.1145/1111037.1111059`.

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. ACM, 2015.

Kathryn E Gray and Matthew Flatt. Compiling java to plt scheme. In *Proc. 5th Workshop on Scheme and Functional Programming*, pages 53–61, 2004.

Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 231–245, New York, NY, USA, 2005. ACM Press.

Michael Greenberg. Space-efficient manifest contracts. *CoRR*, abs/1410.2813, 2014. URL `http://arxiv.org/abs/1410.2813`.

Michael Greenberg. Space-efficient manifest contracts. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 181–194, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2676967. URL `http://doi.acm.org/10.1145/2676726.2676967`.

Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *Principles of Programming Languages (POPL) 2010*, 2010.

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`.

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.

Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric polymorphic contracts. In *Dynamic Languages Symposium*, 2007.

Lars T. Hansen and William D. Clinger. An experimental study of renewal-older-first garbage collection. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 247–258, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8. doi: 10.1145/581478.581502. URL `http://doi.acm.org/10.1145/581478.581502`.

Anders Hejlsberg. Introducing TypeScript. Microsoft Channel 9 Blog, 2012.

Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22 (3):197–230, June 1994.

David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *Trends in Functional Prog. (TFP)*, page XXVIII, April 2007.

David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23(2):167–189, 2010.

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. On polymorphic gradual typing. In *International Conference on Functional Programming*, ICFP. ACM, 2017.

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.

Andrew W. Keep, Alex Hearn, and R. Kent Dybvig. Optimizing closures in O(0)-time. In *Proceedings of the 2012 Workshop on Scheme and Functional Programming*, Scheme '12, 2012.

Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. Toward efficient gradual typing for structural types via coercions. In *Conference on Programming Language Design and Implementation*, PLDI. ACM, June 2019.

G. Lagorio and E. Zucca. Introducing safe unknown types in java-like languages. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 1429–1434, New York, NY, USA, 2006. ACM. ISBN 1-59593-108-2. doi: 10.1145/1141277.1141609. URL `http://doi.acm.org/10.1145/1141277.1141609`.

Xavier Leroy and Michel Mauny. *Dynamics in ML*, pages 406–426. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991. ISBN 978-3-540-47599-6. doi: 10.1007/3540543961_20. URL `http://dx.doi.org/10.1007/3540543961_20`.

Jacob Matthews and Amal Ahmed. Parametric polymorphism through run-time sealing, or, theorems for low, low prices! In *Proceedings of the 17th European Symposium on Programming (ESOP'08)*, March 2008.

Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *The 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2007.

Zeina Migeed and Jens Palsberg. What is decidable about gradual types? *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi: 10.1145/3371097. URL `https://doi.org/10.1145/3371097`.

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`.

Guilherme Ottoni. Hhvm jit: A profile-guided, region-based compiler for php and hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 151–165, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192374. URL `http://doi.acm.org/10.1145/3192366.3192374`.

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for typescript. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 167–180, New

York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2676971. URL `http://doi.acm.org/10.1145/2676726.2676971`.

Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete types for typescript. In *European Conference on Object-Oriented Programming*, ECOOP, 2015.

Gregor Richards, Ellen Arteca, and Alexi Turcotte. The vm already knew that: Leveraging compile-time knowledge to optimize gradual typing. *Proc. ACM Program. Lang.*, 1(OOPSLA):55:1–55:27, October 2017. ISSN 2475-1421. doi: 10.1145/3133879. URL `http://doi.acm.org/10.1145/3133879`.

Taro Sekiyama, Soichiro Ueda, and Atsushi Igarashi. Shifting the blame - A blame calculus with delimited control. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*, pages 189–207, 2015. doi: 10.1007/978-3-319-26529-2_11. URL `https://doi.org/10.1007/978-3-319-26529-2_11`.

Jeremy Siek, 2019. URL `https://github.com/jsiek/gradual-typing-in-agda`.

Jeremy G. Siek. Space-efficient blame tracking for gradual types. April 2008.

Jeremy G. Siek. Type safety in five easy lemmas, August 2012a. URL `https://siek.blogspot.com/2012/08/type-safety-in-five-easy-lemmas.html`.

Jeremy G. Siek. Rationale for "type safety in five", August 2012b. URL `https://siek.blogspot.com/2012/08/rationale-for-type-safety-in-five.html`.

Jeremy G. Siek and Ronald Garcia. Interpretations of the gradually-typed lambda calculus. In *Scheme and Functional Programming Workshop*, 2012.

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

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.

Jeremy G. Siek and Sam Tobin-Hochstadt. The recursive union of some gradual types. In Sam Lindley, Conor McBride, Don Sannella, and Phil Trinder, editors, *Wadler Fest*, LNCS. Springer, 2016.

Jeremy G. Siek and Manish Vachharajani. Gradual typing and unification-based inference. In *DLS*, 2008.

Jeremy G. Siek and Michael M. Vitousek. Monotonic references for gradual typing. *CoRR*, abs/1312.0694, 2013.

Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *Symposium on Principles of Programming Languages*, POPL, pages 365–376, January 2010.

Jeremy G. Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. In *European Symposium on Programming*, ESOP, pages 17–31, March 2009.

Jeremy G. Siek, Peter Thiemann, and Philip Wadler. Blame and coercion: Together again for the first time. In *Conference on Programming Language Design and Implementation*, PLDI, June 2015a.

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 2015b.

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 2015c.

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.

Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual typing embedded securely in JavaScript. In *ACM Conference on Principles of Programming Languages (POPL)*, January 2014.

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *Principles of Programming Languages*, POPL. ACM, January 2016.

The Dart Team. *Dart Programming Language Specification*. Google, 1.2 edition, March 2014.

Satish Thatte. Quasi-static typing. In *POPL 1990*, pages 367–381, New York, NY, USA, 1990. ACM Press.

Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium*, 2006.

Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *Symposium on Principles of Programming Languages*, January 2008.

Julien Verlaguet and Alok Menghrajani. Hack: a new programming langauge for HHVM, 2014. URL `https://code.facebook.com/posts/264544830379293/`

hack-a-new-programming-language-for-hhvm/.

Michael Vitousek, Cameron Swords, and Jeremy G. Siek. Big types in little runtime. In *Symposium on Principles of Programming Languages*, POPL, 2017.

Michael M. Vitousek, Jeremy G. Siek, Andrew Kent, and Jim Baker. Design and evaluation of gradual typing for Python. In *Dynamic Languages Symposium*, 2014.

Haiping Zhao, Iain Proctor, Minghui Yang, Xin Qi, Mark Williams, Qi Gao, Guilherme Ottoni, Andrew Paroski, Scott MacVicar, Jason Evans, and Stephen Tu. The hiphop compiler for php. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 575–586, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384658. URL http://doi.acm.org/10.1145/2384616.2384658.

APPENDIX A

Agda Mechanization

The agda proofs are hosted on github at the following url: `https://github.com/deyaaeldeen/monotonic/tree/dissertation`. Figure 1 presents the mapping between the formal statements in this dissertation and corresponding Agda definitions.

1. Agda Formalization

There are differences between the semantics presented in the dissertation and the semantics formalized in Agda. In addition to the ones described earlier, there is also other fundamental differences discussed below.

intrinsic vs extrinsic The semantics in this dissertation is presented in Curry style where terms exist independent of types and typing rules assign types to those terms. I refer to this approach as *extrinsic*. On the other hand, the semantics in the Agda files follows the Church style where terms and typing rules are fused. I refer to this approach as *intrinsic*.

Variable Representation Variables in the semantics in this dissertation are named while the Agda semantics uses de Bruijn indices to represent variables. The dissertation did not provide a definition of the substitution operation $M[x/N]$ but it is trivial in the sense that it avoids accidental capturing by restricting N to be a closed expression.

Formal Statement	Top Level Agda Module	Agda Definition Name
Definition 1	MonoRef.Dynamics.Store.Precision	$\sqsubseteq_h$
Definition 2	MonoRef.Dynamics.Store.Extension	$\sqsubseteq$
Definition 3	MonoRef.Dynamics.Store.TypingProgress	StoreTypingProgress
Lemma 2	MonoRef.Dynamics.Store.Precision	typeprecise-strenthen-expr
Lemma 3	MonoRef.Dynamics.Store.Extension	prefix-weaken-expr
Definition 4	MonoRef.Dynamics.Store.Evolving.StoreDef	Store
Corollary 3.2	MonoRef.Dynamics.Store.Evolving.Base	ν-cast
Corollary 3.3	MonoRef.Dynamics.Store.Evolving.Base	store
Lemma 4	MonoRef.Dynamics.Simple.EvStore.ActiveCastProgress	$\longrightarrow^c$progress-active
Lemma 5	MonoRef.Dynamics.Simple.EvStore.EvolvingStoreProgress	$\longrightarrow^c$progress-scv
Corollary 5.1	MonoRef.Dynamics.Simple.EvStore.CastProgress	$\longrightarrow^c$progress
Lemma 6	MonoRef.Dynamics.Simple.EvStore.Properties	scv$\longrightarrow^c\Longrightarrow$cv$'$
Lemma 7	MonoRef.Dynamics.Progress.EvolvingStore	progress-evolving-store
Lemma 8	MonoRef.Dynamics.Simple.EvStore.NormalStoreProgress	progress-normal-store
Corollary 8.1	MonoRef.Dynamics.Simple.EvStore.TypeSafety	progress
Lemma 9	MonoRef.Dynamics.Simple.EvStore.Reduction	$\longrightarrow$
Theorem 10	MonoRef.Dynamics.Simple.EvStore.TypeSafety	type-safety
Definition 5	MonoRef.Dynamics.Reduction.StdStore.SuspendedCast	merge$'$
Lemma 11	MonoRef.Dynamics.Reduction.StdStore.SuspendedCast	merge$'$
Lemma 12	MonoRef.Dynamics.Reduction.StdStore.SuspendedCast	merge-respects-$\sqsubseteq_h'$
Lemma 13	MonoRef.Dynamics.Reduction.StdStore.SuspendedCast	merge$'$-respects-++$'$
Lemma 14	MonoRef.Dynamics.Reduction.StdStore.SuspendedCast	merge-soundness
Definition 6	MonoRef.Dynamics.Store.Std.StoreDef	Store
Lemma 15	MonoRef.Dynamics.Reduction.StdStore.StateReduction	μ-cast
Lemma 16	MonoRef.Dynamics.Progress.StdStore	suspended-cast-progress
Lemma 17	MonoRef.Dynamics.Simple.StdStore.NormalStoreProgress	progress-normal-store
Lemma 17.1	MonoRef.Dynamics.Simple.StdStore.TypeSafety	progress
Lemma 18	MonoRef.Dynamics.Simple.StdStore.ApplyCast	apply-cast
Lemma 19	MonoRef.Dynamics.Simple.StdStore.Reduction	$\longrightarrow_e$
Lemma 20	MonoRef.Dynamics.Reduction.StdStore.StateReduction	$\longrightarrow$
Lemma 21	MonoRef.Dynamics.Simple.StdStore.Reduction	$\longrightarrow$
Lemma 23	MonoRef.Coercions.NormalForm.Plain.Compose	compose-normal-form
Lemma 24	MonoRef.Coercions.NormalForm.Plain.Compose	mk-fcoercion-size
Lemma 25	MonoRef.Coercions.NormalForm.Plain.Compose	mk-nfcoercion-size
Proposition 1	MonoRef.Coercions.NormalForm.Plain.Compose	compose-normal-form
Lemma 26	MonoRef.Coercions.NormalForm.Plain.Height.Lemmas	mk-nfcoercion-height
Proposition 2	MonoRef.Coercions.NormalForm.Plain.Height	compose-normal-form-height
Lemma 27	MonoRef.Dynamics.Reduction.StdStore.StateReduction	μ-cast
Lemma 28	MonoRef.Dynamics.Progress.StdStore	suspended-cast-progress
Lemma 29	MonoRef.Dynamics.Efficient.StdStore.NormalStoreProgress	progress-normal-store
Lemma 29.1	MonoRef.Dynamics.Efficient.StdStore.TypeSafety	progress
Lemma 30	MonoRef.Dynamics.Efficient.StdStore.ApplyCast	apply-cast
Lemma 31	MonoRef.Dynamics.Efficient.StdStore.Reduction	$\longrightarrow_e$
Lemma 32	MonoRef.Dynamics.Reduction.StdStore.StateReduction	$\longrightarrow$
Lemma 33	MonoRef.Dynamics.Efficient.StdStore.Reduction	$\longrightarrow$

FIGURE 1. Mapping formal statements in the dissertation to Agda definitions.

APPENDIX B

Grift Macros

1. Values, Macros, and Compose

Figures 1, 2, 3, and 4 lists the C structs used to represent values and macros used in the code examples to manipulate them. Figure 5 shows the code for the `compose` runtime function. Figure 5 gives the interface for an associative map/stack used in `compose` to recognize when we have already composed a recursive coercion that could be used for a particular pair of coercions.

```
1  #define TAG(value) (((int64_t)value)&0b111)
2  #define UNTAG_INT(value) (((int64_t)value)&~0b111)
3  #define TAG_INT(value, tag) (((int64_t)value)|tag)
4  #define UNTAG_REF(ref) ((obj*)UNTAG_INT(ref))
```

FIGURE 1. All allocated values have 3 bits that can be used for tagging.

```
#define TYPE_DYN_RT_VALUE 7
#define TYPE_INT_RT_VALUE 15
#define TYPE_BOOL_RT_VALUE 23
#define TYPE_UNIT_RT_VALUE 31
#define TYPE_FLOAT_RT_VALUE 39
#define TYPE_CHAR_RT_VALUE 47
typedef struct {int64_t index; int64_t hash } type_summary;
typedef struct {type_summary summary; type* body} ref_type;
typedef struct {type_summary summary;
  int64_t arity; type ret; type args[]} fun_type;
typedef struct {type_summary summary;
  int64_t  size; type elems[]} tup_type;
typedef struct {type_summary summary;
  type* body} rec_type;
typedef union {
  int64_t atm;
  ref_type* ref; fun_type* fun;
  tup_type* tup; rec_type* rec} type;
```

Figure 2. Runtime types are either 64 bit integers or a pointer to a heap allocated type. Heap allocated types are hoisted and shared at runtime so that structural equality is equivalent to pointer equality. The lowest 3 bits of each type are used to distinguish between heap allocated types and atomic types. The `summary` field of heap allocated types is used in implementation hashconsing at runtime for the monotonic references implementation.

```
typedef char* blame;
#define ID NULL
typedef struct {type type; blame info;} project_crcn;
typedef struct {type type} inject_crcn;
typedef struct {crcn fst; crcn snd} seq_crcn;
typedef struct {blame info} fail_crcn;
#define UNTAG_2ND(c) \
   ((struct {snd_tag second_tag;}*)UNTAG_INT(c))
typedef struct {
  snd_tag second_tag;
  int32_t arity; crcn ret;
  crcn args[] } fun_crcn;
typedef struct {
  snd_tag second_tag;
  crcn write; crcn read} ref_crcn;
typedef struct {
  snd_tag second_tag;
  int64_t size; crcn elems[] } tup_crcn;
typedef int64_t snd_tag;
typedef struct {
  snd_tag second_tag;
  crcn body[] } rec_crcn;
typedef union {
  project_crcn* prj;
  inject_crcn* inj;
  seq_crcn* seq;
  fail_crcn* fail;
  fun_crcn* fun;
  ref_crcn* ref;
  tup_crcn* tup;
  rec_crcn* rec} crcn;
// UNTAG_PRJ, UNTAG_FAIL, UNTAG_SEQ are similar to UNTAG_INJ
#define UNTAG_INJ(inj) ((inject_crcn)UNTAG_INT(inj))
// MK_SEQ, MK_PROJECTION, MK_INJECTION are similar
#define MK_REF_COERCION(r, w) \
   (tmp_rc = (ref_crcn*)GC_MALLOC(RC_SIZE),\
    tmp_rc->second_tag=REF_COERCION_TAG, \
    tmp_rc->read=r, tmp_rc->write=w,\
    (crcn)(TAG_INT(tmp_rc, HAS_2ND_TAG)))
```

FIGURE 3. Coercions are represented as directed graphs. The only back edges are recursive coercion nodes (`rec_coercion`). We maintain the normal form that isn't enforced by these types. Furthermore, we maintain the invariant that `rec_crcn` are only allocated if referenced by a subcoercion.

```c
typedef struct {
  void* code;
  (obj)(*caster)(obj, type, type, blame);
  obj fvs[]; } closure;
typedef struct{obj elems[]} tuple;
#define MK_TUPLE(n) ((tuple*)GC_MALLOC(sizeof(obj) * n))
typedef struct{obj elem} box;
typedef struct{int64_t length; obj elems[]} vector;
#ifdef TYPE_BASED_CASTS
typedef struct {
  obj* ref;
  type source;
  type target;
  blame info;} ref_proxy;
#define MK_REF_PROXY(v, s, t, l) \
   (tmp_rp = (ref_proxy*)GC_MALLOC(RP_SIZE),\
    tmp_rp->value=v, tmp_rp->source=s,\
    tmp_rp->target=t, tmp_rp->info=l,\
    (obj)TAG_INT(tmp_rp, REF_PROXY_TAG)
#define UNTAG_FUN(fun) ((closure*)(fun)))
#elseif COERCIONS
#define UNTAG_FUN(fun) ((closure*)UNTAG_INT(fun))
typedef struct {obj* ref; crcn cast;} ref_proxy
#define MK_REF_PROXY(v, c) \
   (tmp_rp = (ref_proxy*)GC_MALLOC(RP_SIZE),\
    tmp_rp->value=v, tmp_rp->coerce=c,\
    (obj)TAG_INT(tmp_rp, REF_PROXY_TAG)
#endif
typedef struct {obj value; type source} nonatomic_dyn;
#define UNTAG_NONATOMIC(value) \
   ((nonatomic_dyn)UNTAG_INT(value))
typedef union {
  int64_t atomic;
  nonatomic_dyn* boxed} dynamic;
#define UNTAG(v) \
   ((TAG(v) == INT_TAG) ? (obj)(UNTAG_INT(v)>>3) : \
    (TAG(v) == UNIT_TAG) ? (obj)UNIT_CONSTANT : \
    ... (obj)UNTAG_NONATOMIC(v).value)
#define TYPE(v) \
   ((TAG(v) == INT_TAG) ? (type)INT_TYPE : \
    (TAG(v) == UNIT_TAG) ? (type)UNIT_TYPE :\
    ... UNTAG_NONATOMIC(v)->source)
```

Figure 4. Value Representation

```
#define INJECT(v, s) \
   ((s==INT_TYPE) ? TAG_INT(v<<3, INT_TAG) :\
    (s==UNIT_TYPE) ? DYN_UNIT_CONSTANT : ... \
    ... (tmp_na = (nonatomic_dyn*)GC_MALLOC(NA_DYN_SIZE),\
    tmp_na->value=value, tmp_na->source=s, (obj)tmp_na)
typedef union {
  int64_t fixnum; double  flonum; dynamic dyn;
  closure* clos; tuple* tuple;
  box* box; vector* vec} obj;
```

Figure 4. Value Representation

```
typedef struct {
  crcn fst;
  crcn snd;
  crcn mapsto } assoc_triple;
typedef struct {
  unsigned int size;
  unsigned int next;
  assoc_triple *triples;} assoc_stack;

// Push a new triple on the assoc_stack
void
assoc_stack_push(assoc_stack *m, crcn f, crcn s, crcn t);

// return index of association of f and s
// returns -1 if the association isn't found
int grift_assoc_stack_find(assoc_stack *m, obj f, obj s);

// pop the most recent triple, return the mapsto value
obj
grift_assoc_stack_pop(assoc_stack *m);

// return the mapsto of the ith association
obj
grift_assoc_stack_ref(assoc_stack *m, int i);

// update the mapsto of the ith association
void
grift_assoc_stack_set(assoc_stack *m, int i, obj v);
```

Figure 5. The association stack is used `compose` to compose recursive equations at runtime. It associates two pointers as a key to another pointer.

```c
assoc_stack *as;
crcn compose(crcn fst, crcn snd, bool* id_eqv, int* fvs) {
  if (fst == ID) {if (snd == ID) return ID; else { *id_eqv = false; return snd; }}
  else if (snd == ID) { return fst; }
  else if (TAG(fst) == SEQUENCE_TAG) {
    sequence s1 = UNTAG_SEQ(fst);
    if (TAG(s1->fst) == PROJECT_TAG) {
      *id_eqv = false; return MK_SEQ(s1->fst, compose(s1->snd, snd, id_eqv, fvs)); }
    else if (TAG(snd) == FAIL_TAG) { *id_eqv = false; return snd; }
    else { sequence s2 = UNTAG_SEQ(snd);
      type src = UNTAG_INJ(s1->snd)->type; type tgt = UNTAG_PRJ(s2->fst)->type;
      blame lbl = UNTAG_PRJ(s2->fst)->lbl; crcn c = mk_crcn(src, tgt, lbl);
      bool unused = true;
      return compose(compose(seq->fst, c, &unused, fvs), s2->snd, id_eqv, fvs); }
  } else if (TAG(snd) == SEQUENCE_TAG) {
    if (TAG(fst) == FAIL) { *id_eqv = false; return fst; }
    else {
      crcn c = compose(fst, s2->fst, id_eqv, fvs);
      *id_eqv = false; return MK_SEQ(c, UNTAG_SEQ(seq)->snd); }
  } else if (TAG(snd) == FAIL) {
      *id_eqv = false; return (TAG(fst) == FAIL ? fst : snd); }
  } else if (TAG(fst) == HAS_2ND_TAG) {
    snd_tag tag1 = UNTAG_2ND(fst)->second_tag; snd_tag tag2 = UNTAG_2ND(fst)->second_tag;
    if (tag1 = REC_COERCION_TAG  tag2 = REC_COERCION_TAG) {
      int i = assoc_stack_find(as, c1, c2);
      if (i < 0) {
        assoc_stack_push(as, c1, c2, NULL); new_id_eqv = true;
        crcn c = (tag1 == REC_COERCION_TAG) ?
                    compose(REC_COERCION_BODY(c1), c2, &new_id_eqv, fvs):
                    compose(c1, REC_COERCION_BODY(c2), &new_id_eqv, fvs);
        crcn mu = assoc_stack_pop(as);
        if (!*new_id_eqv) *id_eqv = false;
        if (mu == NULL) return c;
        *fvs -= 1;
        if (*fvs = 0 && new_id_eqv) { return ID }
        else { REC_COERCION_BODY_INIT(mu, c); return mu; }
      else { crcn mu = assoc_stack_ref(as, i);
        if (mu = NULL) {*fvs += 1; mu = MK_REC_CRCN();
          assoc_stack_set(as, i, mu); return mu; }
        else { return mu; }}}
    else if (tag1 == FUN_COERCION_TAG) {
      return compose_fun(fst, snd); }
```

```
    else if (tag1 == REF_COERCION_TAG) {
      ref_crcn r1 = UNTAG_REF(fst);
      ref_crcn r2 = UNTAG_REF(snd);
      if (read == ID && write == ID) return ID;
      else {
        crcn c1 = compose(r1->read, r2->read);
        crcn c2 = compose(r2->write, r1->write);
        return MK_REF_COERCION(c1, c2); }}
    else { // Must be tuple coercions
      return compose_tuple(fst, snd); }}
  else { raise_blame(UNTAG_FAIL(fst)->lbl); }
}
```

FIGURE 5. The `compose` function for normalizing coercions.

APPENDIX C

Implementation of Monotonic Vectors in Grift

```
1  int64_t apply_cast(int64_t value, int64_t type1, int64_t type2, bool suspend_ref_casts,
2                     suspend_vect_casts);
3
4  int64_t * apply_vector_cast(int64_t * address, int64_t type, bool
5                              suspend_ref_casts, bool suspend_vect_casts) {
6      if (suspend_vect_casts) {
7          cast_queue_enqueue(vect_cq, address, type);
8      } else {
9          int64_t old_rtti = address[0];
10         int64_t new_rtti = greatest_lower_bound(old_rtti, type);
11         if (old_rtti != new_rtti) {
12             int64_t vect = address[1];
13             int64_t n = vector_length(vect);
14             for (int i = 0; i < n; ++i) {
15                 int64_t old_value_i = address[i+2];
16                 int64_t new_value_i = apply_cast(old_value_i, old_rtti,
17                                                  new_rtti, suspend_ref_casts, true);
18                 address[i+2] = new_value_i;
19             }
20             address[0] = new_rtti;
21             apply_suspended_vector_casts();
22         }
23     }
24     return address;
25 }
```

FIGURE 1. The C code for the `apply_vector_cast` function that casts an address using a type carried by a coercion. Key called functions are forward declared.

```c
void apply_suspended_vector_casts(bool suspend_ref_casts) {
    while (cast_queue_is_not_empty(vect_cq)) {
        int64_t * suspended_cast_address = cast_queue_peek_address(vect_cq);
        int64_t old_rtti = suspended_cast_address[0];
        int64_t suspended_cast_type = cast_queue_peek_type(vect_cq);
        int64_t new_rtti = greatest_lower_bound(old_rtti, suspended_cast_type);
        cast_queue_dequeue(vect_cq);
        if (old_rtti != new_rtti) {
            int64_t vect = suspended_cast_address[1];
            int64_t n = vector_length(vect);
            for (int i = 0; i < n; ++i) {
                int64_t old_value_i = suspended_cast_address[i+2];
                int64_t new_value_i = apply_cast(old_value_i, old_rtti,
                                                 new_rtti, suspend_ref_casts, true);
                suspended_cast_address[i+2] = new_value_i;
            }
            suspended_cast_address[0] = new_rtti;
        }
    }
}
```

FIGURE 2. The C code for the `apply_suspended_vector_casts` function that applies all suspended casts in the queue.

```c
int64_t read(int64_t * address, int64_t index, int64_t type) {
    int64_t rtti = address[0];
    int64_t value = address[2+index];
    return apply_cast(value, rtti, type, false);
}

void vector_write(int64_t * address, int64_t index, int64_t value, int64_t type) {
    int64_t rtti = address[0];
    int64_t new_value = apply_cast(value, type, rtti, false, true);
    address[2+index] = new_value;
    apply_suspended_vector_casts(false);
}
```

FIGURE 3. The C code for writing a value to a partially-typed vector.

APPENDIX D

Type Hashconsing

```
struct list {
  int64_t data;
  struct list* next;
};
typedef struct list* list;

struct chain {
  list list;
};

typedef struct chain* chain;

struct table {
  int64_t slots;
  int64_t num_elems;
  float load_factor;
  chain* array;
};

typedef struct table* table;

table alloc_hash_table(int64_t slots, float load_factor)
{
  table ht = GC_MALLOC(8 * 4);
  ht->slots = slots;
  ht->num_elems = 0;
  ht->load_factor = load_factor;
  ht->array = GC_MALLOC(8 * slots);
  return ht;
}
```

```
void ht_resize(table ht) {
  int64_t old_slots = ht->slots;
  chain* old_array = ht->array;
  int64_t new_slots = old_slots * 2;
  ht->slots = new_slots;
  ht->array = GC_MALLOC(8 * new_slots);
  int i;
  for (i = 0; i < old_slots; ++i) {
    chain C = old_array[i];
    if (C != NULL) {
      list p = C->list;
      while (p != NULL) {
        types_reinsert(ht, p->data);
        p = p->next;
      }
    }
  }
  if (old_array) GC_FREE(old_array);
}
```

```
int64_t hashcons(table ht, int64_t e, int64_t hcode)
{
  float current_load = (float) ht->num_elems/(float) ht->slots;
  if (current_load > ht->load_factor) {
    ht_resize(ht);
  }
  int h = hcode % ht->slots;
  h = h < 0 ? h + ht->slots : h;
  chain C = ht->array[h];
  if (C == NULL) {
    C = GC_MALLOC(8 * 1);
    ht->array[h] = C;
    list new_item = GC_MALLOC(8 * 2);
    new_item->data = e;
    new_item->next = NULL;
    C->list = new_item;
    ht->num_elems++;
    return e;
  }
  list p = C->list;
  while (p != NULL) {
    if (types_equal(e, p->data))
      return p->data;
    p = p->next;
  }
  list new_item = GC_MALLOC(8 * 2);
  new_item->data = e;
  new_item->next = C->list;
  C->list = new_item;
  ht->num_elems++;
  return e;
}
```

```c
int types_equal(int64_t t1, int64_t t2)
{
  int64_t tag1 = (t1 & TYPE_TAG_MASK);
  int64_t tag2 = (t2 & TYPE_TAG_MASK);
  int64_t count, untagged_t1, untagged_t2, i;
  if (tag1 == tag2) {
    untagged_t1 = (t1 ^ tag1);
    untagged_t2 = (t2 ^ tag1);
    switch (tag1) {
    case TYPE_MU_TAG:
      return ((int64_t*)untagged_t1)[TYPE_MU_BODY_INDEX] ==
             ((int64_t*)untagged_t2)[TYPE_MU_BODY_INDEX];
    case TYPE_GREF_TAG ... TYPE_MVECT_TAG:
      // the type index is the same for gref,gvect,mref, and mvect.
      return ((int64_t*)untagged_t1)[TYPE_GREF_TYPE_INDEX] ==
             ((int64_t*)untagged_t2)[TYPE_GREF_TYPE_INDEX];
      break;
    case TYPE_TUPLE_TAG:
      count = ((int64_t*)untagged_t1)[TYPE_TUPLE_COUNT_INDEX];
      // the loop checks count along with the elements
      for (i = TYPE_TUPLE_ELEMENTS_OFFSET-1;
           i < (count + TYPE_TUPLE_ELEMENTS_OFFSET);
           ++i) {
        if (((int64_t*)untagged_t1)[i] != ((int64_t*)untagged_t2)[i]) {
          return false;
        }
      }
      return true;
      break;
```

```
    case TYPE_FN_TAG:
      count = ((int64_t*)untagged_t1)[TYPE_FN_ARITY_INDEX];
      // the loop checks the arity and the return type along with the elements
      for (i = TYPE_FN_FMLS_OFFSET-2; i < (count + TYPE_FN_FMLS_OFFSET); ++i) {
        if (((int64_t*)untagged_t1)[i] != ((int64_t*)untagged_t2)[i]) {
          return false;
        }
      }
      return true;
      break;
    default:
      printf("grift internal runtime error:\n"
             "    location: hashcons.c/types-equal\n"
             "    cause: unrecognized type tag: %" PRId64 "\n",
             tag1);
      exit(1);
    }
  }
  return false;
}
```

```c
void types_reinsert(table ht, int64_t ty)
{
  int64_t tag = (ty & TYPE_TAG_MASK);
  int64_t untagged_ty = (ty ^ tag);
  int64_t h;
  switch (tag) {
  case TYPE_FN_TAG ... TYPE_MU_TAG:
    h = ((int64_t*)untagged_ty)[TYPE_HASHCONS_HASHCODE_INDEX] % ht->slots;
    h = h < 0 ? h + ht->slots : h;
    chain C = ht->array[h];
    if (C == NULL) {
      C = GC_MALLOC(8 * 1);
      ht->array[h] = C;
      C->list = NULL;
    }
    list new_item = GC_MALLOC(8 * 2);
    new_item->data = ty;
    new_item->next = C->list;
    C->list = new_item;
    break;
  default:
    printf("grift internal runtime error:\n"
           "    location: hashcons.c/types-reinsert\n"
           "    cause: unrecognized type tag: %" PRId64 "\n",
           tag);
    exit(1);
  }
```

FIGURE 0. The C implementation of type hashconsing.

Deyaaeldeen Almahallawi

Email : dalmahal@indiana.edu

http://deyaaeldeen.github.io

Education

- **Indiana University** — Bloomington, IN
  Ph.D. in Computer Science — *Aug. 2013 – June 2020*
  M.Sc. in Computer Science; GPA: 3.7 — *Aug. 2013 – May 2015*
- **Helwan University** — Cairo, Egypt
  B.Sc. in Computer Science; GPA: 3.34 — *Aug. 2007 – May 2011*

Experience

- **Bloomberg LP** — New York, NY
  Compiler Engineer Intern — *Summer 2019*
  - **Automatic Differentiation**: Designed and implemented a DSL for differentiable functions where generated code can be distributed as a shared library. Used for differentiating monte-carlo simulations efficiently. Haskell, Bash
- **The MathWorks, Inc** — Natick, MA
  Compiler Engineer Intern — *Summer 2018*
  - **Matlab**: Automatic loop optimization based on the polyhedral model to increase data locality in cache. C++
- **LeapYear Technologies** — Berkeley, CA
  Software Engineer Intern — *Summer 2017*
  - **Query Compilation**: Privacy-preserving query engine and machine learning platform on top of Apache Spark. Haskell, Java, Python.
- **Indiana University** — Bloomington, IN
  Research and Teaching Assistant, and Lecturer — *Aug. 2013 - Present*
  - **Research Assistant (Adviser: Prof. Jeremy Siek)**: Working on Grift, ahead-of-time compiler for gradual typing with efficient runtime system and C and LLVM backends.
  - **Lecturer and Teaching Assistant**: Lectured Compilers, and taught labs and graded for Introduction to Computer Science, Data Structures, and Computer Networks.
- **MESC Labs** — Cairo, Egypt
  Software Engineer — *Aug. 2011 - Aug. 2013*
  - **LibCAD**: Breast cancer detection from mammograms service. Increased the detection accuracy by investigating many novel feature extraction techniques. Matlab, C++.

Selected Publications

- Space-Efficient Monotonic References (Workshop on Gradual Typing) 2020.
- Toward Efficient Gradual Typing for Structural Types via Coercions (PLDI) 2019.
- An Efficient Compiler for the Gradually Typed Lambda Calculus (Scheme) 2018.

Other Projects

- **Distributed Reversible Computing 2013**: Worked on a language for distributed reversible computing that provides dynamic channel creation and treats ”speculation” as first class construct.

Awards

- Honor roll of top students at Faculty of Computers and Information, Helwan University, 2008

Class Projects

- **Scheme Compiler**: Optimizing Compiler from a subset of Scheme R6RS to x86_64.
- **Dynamizer**: Compiler from a statically-typed language to gradually-typed configurations of a program.
- **Distributed Privacy-Preserving Machine Learning**: using differential privacy and fully homomorphic encryption.

Programming Skills

- **Languages**: Agda, Haskell; Racket; C++; C; X86-64; Java
- **Tools**: emacs; git; docker; valgrind; gdb; tmux