MANIFEST CONTRACTS

Michael Greenberg

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2013

Supervisor of Dissertation

Benjamin C. Pierce
Henry Salvatori Professor of CIS

Graduate Group Chairperson

Val Tannen
Professor of CIS

Dissertation Committee
Stephanie Weirich (Associate Professor of CIS; Committee Chair)
Rajeev Alur (Zisman Family Professor of CIS)
Greg Morrisett (Allen B. Cutting Professor of CS at Harvard)
Steve Zdancewic (Associate Professor of CIS)

Acknowledgments

> I know you have faith in me; let me take it with me on the road before me and I will be able to go on forever.
>
> "Tang Qiaodi"
> *The Gourmet and Other Stories of Modern China*
> Lu Wenfu

Let me begin by thanking Benjamin Pierce for being a formative collaborator and ally—for his teaching, advice, guidance, and forbearance.

Many people—members of the Penn PL Club and otherwise—have provided helpful and insightful comments along the way: Brian Aydemir and João Belo; Amal Ahmed, Cătălin Hriţcu, Taro Sekiyama, Jianzhou Zhao; Ron Garcia, Fritz Henglein, Jeremy Siek, Phil Wadler, and Brent Yorgey. My fellow travelers Chris Casinghino and Adam Aviv have been welcome company. Nate Foster has, on several occasions, helped me remember to stay on the wagon. I hope I have been equally supportive of Arjun Ravi Narayan and Scott Kilpatrick.

Atsushi Igarashi and Stephanie Weirich have been a delight to work with as co-authors, and I have learned a great deal from both. Atsushi and his student, Taro Sekiyama, helped quash some serious bugs while being a pleasure to work with.

I am grateful for the attention, advice, and patience of my committee—Stephanie Weirich (again), Steve Zdancewic, Rajeev Alur, and Greg Morrisett. Their efforts have seriously improved the quality of my thesis.

I am interested in programming languages because of Dan Friedman. Dan insisted that I let loose and have fun in college, and perhaps I've taken his advice too much to heart, but he also insisted that I meet with Shriram Krishnamurthi, who formed so much of my early research taste—and helped me find Benjamin in turn.

My friends here in Philadelphia and beyond form a broad family who, along with my actual family, soothed and encouraged me. B2 was effectively my office, and its *baristi* kindly put up with me.

Finally, Hannah de Keijzer literally and figuratively brings music into my life every day, and there is no way for me to do that justice—not that I won't try.

ABSTRACT

MANIFEST CONTRACTS

Michael Greenberg

Benjamin Pierce

Eiffel [47] popularized *design by contract*, a software design philosophy where programmers specify the requirements and guarantees of functions via executable pre- and post-conditions written in code. Findler and Felleisen [26] brought contracts to higher-order programming, inspiring the PLT Racket implementation of contracts [56]. Existing approaches for runtime checking lack reasoning principles and stop short of their full potential—most Racket contracts check only simple types. Moreover, the standard algorithm for higher-order contract checking can lead to unbounded space consumption and can destroy tail recursion. In this dissertation, I develop so-called *manifest* contract systems which integrate more coherently in the type system, and relate them to Findler-and-Felleisen-style *latent* contracts. I extend a manifest system with type abstraction and relational parametricity, and also show how to integrate dynamic types and contracts in a *space efficient* way, i.e., in a way that doesn't destroy tail recursion. I put manifest contracts on a firm type-theoretic footing, showing that they support extensions necessary for real programming. Developing these principles is the first step in designing and implementing higher-order languages with contracts and refinement types.

My work has been supported by the National Science Foundation under grants 0534592 *Linguistic Foundations for XML View Update*, and 0915671 *Contracts for Precise Types*.

Some of the material in Chapter 4 was supported in part by the DARPA CRASH program through the United States Air Force Research Laboratory (AFRL) under Contract No. FA8650-10-C-7090. The views expressed are my own and do not reflect the official policy or position of the Department of Defense or the U.S. Government—and the more daylight between us, the better.

Contents

List of Tables

List of Figures

The technical development in this dissertation comprises three papers: Greenberg et al. [35] (itself an extended version of Greenberg et al. [34]), an extended and corrected version of Belo et al. [8], and a rejected POPL 2013 submission, Greenberg [33].

This document is typeset in LaTeX using Computer Modern. There would be even more errors had I not used Sewell and Zappa Nardelli's OTT tool [62]. I also used Aydemir and Weirich's LNGen tool [7] for the Coq development of parallel reduction in Chapter 2.

Chapter 1

Introduction

> Master, I've filled my contract, wrought in Thy many lands;
> Not by my sins wilt Thou judge me, but by the work of my hands.
>
> *The Song of the Wage-slave*
> Robert W. Service

Program specifications allow programmers to check their work. Data structure invariants for, e.g., red-black trees, help programmers write correct implementations; input/output specifications, also called pre- and post-conditions, offer a finer grained checking, at the level of individual functions, e.g., is this function an ϵ-approximation of the square-root function? Temporal logics allow programmers to specify the behavior of long-running and non-terminating programs, e.g., does every request packet sent to a server get a response?

It is folklore that the mere act of writing down a specification is enough to help with program correctness. Computer assisted checking adds a stronger guarantee: sound specification checkers can, in an assisted or even entirely automated way, prove that programs meet their specifications. Naturally, there are trade-offs: sound and complete checkers (of nontrivial properties) can't exist because of the halting problem. As a result, program specification languages exist on a spectrum, where less precise specifications are more robustly checkable—and, therefore, more often used.

Types are perhaps the most common form of program specification, even though their use isn't universal. Types occupy a sweet spot: they are easy for programmers to understand, allow for compositional reasoning, and are machine checkable, if not inferrable. Types also have a beautiful mathematical theory, and they are compelling to study in their own right.

Assertions and pre/post-conditions are also common forms of specification during the development cycle; they are very common when debugging, but perhaps less so in deployed code. Written in code, typically as part of the program itself, assertions and pre- and post-conditions allow for direct feedback at runtime. For example, the `pop` operation of a stack might have a pre-condition checking that the stack

argument is non-empty. When the program calls pop with an empty stack, an informative error message can be printed to the screen—or the debugger can kick in. Eiffel [47] makes pre- and post-conditions an essential part of program development in a method called *design by contract*. They use software contracts—pre- and post-conditions—extensively, stating precise properties as concrete predicates written in the same language as the rest of the program.

Eiffel's contracts are *first order*, designed for an imperative, object-oriented language. Findler and Felleisen [26] introduced "higher-order contracts" for functional languages. These can take one of two forms: predicate contracts (also known as *refinement types*) like $\{x{:}\mathsf{Int} \mid x > 0\}$, which denotes the positive numbers; and function contracts like $(x{:}\mathsf{Int}) \to \{y{:}\mathsf{Int} \mid y > x\}$, which denotes functions over the integers that return numbers larger than their inputs. Other examples might require that divisors are non-zero:

$$\mathsf{div} : \mathsf{Real} \to \{m{:}\mathsf{Real} \mid m \neq 0\} \to \mathsf{Real}$$

or that the square root function produces answers that are correct (up to some ϵ):

$$\mathsf{sqrt} : (x{:}\mathsf{Real}) \to \{y{:}\mathsf{Real} \mid \mathsf{abs}(x - y^2) < \epsilon\}$$

This last is a *very* strong contract; it entirely specifies the input/output behavior of the sqrt function. We could have given a similarly strong contract for division:

$$\mathsf{div} : (n{:}\mathsf{Real}) \to \{m{:}\mathsf{Real} \mid m \neq 0\} \to \{k{:}\mathsf{Real} \mid m \cdot k = n\}$$

One feature of contracts is that within the framework of contracts, it's possible to express a wide range of specifications at varying levels of precision.

In this dissertation, I **design higher-order languages combining conventional type systems, like the simply typed lambda calculus and System F, with contracts**. Designing type systems supporting contracts bolsters the power of abstractions defined by types. Before addressing my own work on the topic, I want to discuss the state of the art—PLT Racket's contract system [55, 56]—and its shortcomings.

PLT Racket contracts are overwhelmingly just simple types. Racket's large codebase has many contracts, but 82% of the contracts in their standard library merely recover simple types.[1] Findler and Felleisen developed their contract system to add a form of specification to PLT Racket. Racket is a Scheme [69], written in S-expression syntax and *dynamically typed*. (They have gone on to develop slightly more traditional type systems, Typed Racket [74] in particular.) **In using contracts for simple types, Racket contracts fall short of their full potential.**

As an anecdote to bolster and clarify the data I collected, consider a typical Racket file, shown in Figure 1.1. The file, adapted from the Racket standard library, defines a queue data structure, and then exports (in Racket parlance, "provides") several

[1] My own count, as of January 26th, 2011.

```
#lang racket/base

(struct queue (head tail) #:mutable)
(struct link (value [tail #:mutable]))

(define (make-queue) (queue #f #f))
(define (queue-empty? q) (not (queue-head q)))
(define (nonempty-queue? v) (and (queue? v) (queue-head v) #t))

(define (enqueue! q v)
  (unless (queue? q) (raise-type-error enqueue! "queue" 0 q))
  (let ([new (link v #f)])
    (if (queue-head q)
      (set-link-tail! (queue-tail q) new)
      (set-queue-head! q new))
    (set-queue-tail! q new)))

(define (dequeue! q)
  (unless (queue? q) (raise-type-error dequeue! "queue" 0 q))
  (let ([old (queue-head q)])
    (unless old (error 'dequeue! "empty queue"))
    (set-queue-head! q (link-tail old))
    (link-value old)))

(define queue/c (flat-named-contract "queue" queue?))
(define nonempty-queue/c (flat-named-contract "nonempty-queue" nonempty-queue?))

(provide/contract
 [queue/c flat-contract?]
 [nonempty-queue/c flat-contract?]
 [queue? (-> any/c boolean?)]
 [make-queue (-> queue/c)]
 [queue-empty? (-> queue/c boolean?)])
(provide enqueue! dequeue!)
```

Taken from `collects/data/queue.rkt`, as of November 6th 2011. Some comments have been removed.

Figure 1.1: A queue module in Racket

things. This is where contracts are typically placed in PLT Racket definitions, for reasons described below in Section 1.3. First, there are `queue/c` and `nonempty-queue/c`, which are "flat" contracts on values, i.e., predicate contracts a/k/a refinement types. Then the predicate `queue?` tests whether a given value is a queue: it has a function contract, written in prefix form, that says that `queue?` accepts any value (`any/c`) and returns a boolean (the predicate `boolean?`, which is automatically treated as a contract). The constructor `make-queue` is a function of zero arguments that returns a value that satisfies the contract `queue/c`. The predicate `queue-empty?` checks whether or not the queue is empty: its sole argument must be a queue (`queue/c`) and it returns a boolean. Notice that this interface is *dependent*: we provide `queue/c` and then use it in the contracts for `make-queue` and `queue-empty?`.

The queue module also provides two non-contracted identifiers, `enqueue!` and `dequeue!`.[2] Looking at their definitions, `enqueue!` expects a queue and returns whatever the structure mutator `set-queue-tail!` returns—a unit value. So a good contract for `enqueue!` would be `(-> queue/c any/c)`. Similarly, `dequeue!` takes in a queue and returns its head. What's more, `dequeue!` expects a *non-empty* queue—if given an empty queue, it will raise an error. So a good contract for `dequeue!` would be `(-> nonempty-queue/c any/c)`. The return contract of `any/c` makes sense, since we don't know anything about values in the queue.

As written, **the contracts on the queue interface don't enforce anything more than a simple type system would**, though they define a refined type (`nonempty-queue/c`). I would reiterate that this isn't cherry picking examples: four out of five Racket contracts enforce nothing more than simple types.

PLT Racket's use of contracts to recover simple types doesn't teach us much for the design of new languages: **it's already possible to check simple types completely statically**. While Racket's solution helps with their pre-existing code, why would a new language would follow suit? Why bother paying the runtime cost of contract checking at runtime when static checking is available?

Dynamically typed languages create a "contracts for simple types bias". Writing specifications can be hard work—as hard as programming. It's tempting to stop at "good enough"—which in a dynamically typed language, may mean only simple types. Moreover, dynamic typing and contracts leaves us without empty handed. What are the reasoning principles for contracts? How does contract checking affect execution? Are there limits on how much space and time contract checking can take?

In this dissertation, I explore **what contracts look like when we treat contracts as our type system, taking simple types for granted**. Suppose we start with a simply typed lambda calculus. Can we coherently integrate contracts into a type system? In terms of the PLT Racket example above, I'd like to treat `queue/c` as a type (just as it would be in ML)—and I'd also like to treat `nonempty-queue/c` as a type. Treating contracts as types has many benefits. First, it gets rid of the low bar for contracts in PLT Racket: if the programmer wants simple types, they

[2]The `!` are a Racket naming convention for functions that mutate their inputs.

have them as a static check. Next, having contracts as part of type signatures allows for operations on built-in abstract types to specify their abstractions totally; `dequeue!` is a partial operation, and the type `queue/c` $\to$ `unit` doesn't adequately capture that partiality; the type `nonempty-queue/c` $\to$ `unit` does. A similar case applies to built-in operations, e.g., the division operator. Finally, types and contracts combine to yield strong reasoning principles, giving better guarantees to clients and implementors of, e.g., the queue abstraction.

Two issues stand in the way of integrating contracts in types. What should the semantics of a type system with "manifest" contracts be? Are such semantics practical from an efficiency perspective?

The first issue, determining the semantics, is challenging because contracts contain code. If contracts are to be part of the type system, we are left with a *dependent* type system, one where treating `nonempty-queue/c` as a *type* forces the meaning of types to depend on evaluation. The metatheory of dependent languages is notoriously difficult, and is typically treated in rarified, "pure" settings, e.g., without nontermination or effects. But we are interested in writing specifications for *programs*, where programs may diverge or contracts may be unsatisfied—and therefore must signal errors. Early attempts at hybrid type/contract systems had serious issues with the well-foundedness of their definitions. **If contracts and types are to coexist, the metatheory must scale to impure core calculi.**

The second issue, efficiency, is in fact a challenge for all contract systems. Contract checking can change the asymptotic complexity of programs by accumulating too many checks on functions or on the stack. **If contracts are to be a core part of language design, they must not affect the asymptotic complexity of programs.**

In this dissertation, I (a) situate type/contract systems in the design space, (b) develop abstraction and reasoning principles (specifically, relational parametricity) for a type/contract system, and (c) resolve issues with space consumption affecting runtime checking of contracts. The rest of this introduction covers more specific technical details. At the end of the introduction, I remark briefly on notations and conventions (Section 1.5).

1.1 Latent and manifest contracts

Findler and Felleisen's work sparked a resurgence of interest in contracts, and in the intervening years a bewildering variety of related systems have been studied. Broadly, these come in two different sorts. In systems with *latent* contracts, types and contracts are orthogonal features. Examples of this style include Findler and Felleisen's original system, Hinze et al. [41], Blume and McAllester [11], Chitil and Huch [16], Guha et al. [37], and Tobin-Hochstadt and Felleisen [74]. By contrast, *manifest* contracts are integrated into the type system, which tracks, for each value, the most recently checked contract. *Hybrid types* [28] are a well-known example in this

style; others include the work of Ou et al. [51], Wadler and Findler [78], and Knowles et al. [45]. (I discuss related work at length in Chapter 5.)

Findler and Felleisen's latent calculus annotates function definitions with contracts; to actually check the contracts, they include terms like $\langle\{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\}\rangle^{l,l'}\ 1$, in which a boolean predicate, pos, is applied to a run-time value, 1. This term evaluates to 1, since pos 1 returns true. On the other hand, the term $\langle\{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\}\rangle^{l,l'}\ 0$ evaluates to *blame*, written $\Uparrow l$, signaling that a contract with label l has been violated. The other label on the contract, l', comes into play with *function contracts*, $c_1 \mapsto c_2$. For example, the term

$$\langle\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \mapsto \{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\}\rangle^{l,l'}\ (\lambda x{:}\mathsf{Int}.\ x - 1)$$

"wraps" the function $\lambda x{:}\mathsf{Int}.\ x - 1$ in a pair of checks: whenever the wrapped function is called, the argument is checked to see whether it is nonzero; if not, the blame term $\Uparrow l'$ is produced, signaling that the *context* of the contracted term violated the expectations of the contract. If the argument check succeeds, then the function is run and its result is checked against the contract pos x, raising $\Uparrow l$ if this fails (e.g., if the wrapped function is applied to 1).

The key feature of manifest systems is that descriptions like $\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\}$ are incorporated into the type system as *refinement types*. Values of refinement type are introduced via *casts* like $\langle\{x{:}\mathsf{Int} \mid \mathsf{true}\} \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\}\rangle^l\ n$, which has static type $\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\}$ and checks, dynamically, that n really is nonzero, raising $\Uparrow l$ otherwise. Similarly, $\langle\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\}\rangle^l\ n$ casts an integer that is statically known to be nonzero to one that is statically known to be positive.

The manifest analogue of function contracts is casts between function types. For example, consider:

$$f \quad = \quad \langle\lceil\mathsf{Int}\rceil \to \lceil\mathsf{Int}\rceil \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\} \to \{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\}\rangle^l\ (\lambda x{:}\lceil\mathsf{Int}\rceil.\ x - 1),$$

where $\lceil\mathsf{Int}\rceil = \{x{:}\mathsf{Int} \mid \mathsf{true}\}$. The sequence of events when f is applied to some argument n (of type P) is similar to what we saw before:

$$f\ n \longrightarrow_h \langle\lceil\mathsf{Int}\rceil \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\}\rangle^l\ ((\lambda x{:}\lceil\mathsf{Int}\rceil.\ x - 1)\ (\langle\{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\} \Rightarrow \lceil\mathsf{Int}\rceil\rangle^l\ n))$$

First, n is cast from $\{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\}$ to $\lceil\mathsf{Int}\rceil$ (it happens that in this case the cast cannot fail, since the target predicate is just true, but if it did, it would raise $\Uparrow l$); then the function body is evaluated; and finally its result is cast from $\lceil\mathsf{Int}\rceil$ to $\{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\}$, raising $\Uparrow l$ if this fails. The domain cast is contravariant and the codomain cast is covariant.

One point to note here is that casts in the manifest system have just one label, while contract checks in the latent system have two. This difference is not fundamental to the latent/manifest distinction—both latent and manifest systems can be given more or less rich algebras of blame—but rather a question of the pragmatics of assigning responsibility: contract checks (called *obligations* in Findler and Felleisen [26]) use

two labels, while casts use one. Informally, a function contract check $\langle c_1 \mapsto c_2 \rangle^{l,l'}\ f$ divides responsibility for f's behavior between its body and its environment: the programmer is saying "If f is ever applied to an argument that does not pass c_1, I refuse responsibility ($\Uparrow l'$), whereas if f's result for good arguments does not satisfy c_2, I accept responsibility ($\Uparrow l$)." In a system with casts, the programmer who writes $\langle R_1 \to R_2 \Rightarrow S_1 \to S_2 \rangle^l\ f$ is saying "Although all I know statically about f is that its results satisfy R_2 when it is applied to arguments satisfying R_1, I assert that it's okay to use it on arguments satisfying S_1 [because I believe that S_1 implies R_1] and that its results will always satisfy S_2 [because R_2 implies S_2]." In the latter case, the programmer is taking responsibility for *both* assertions (so $\Uparrow l$ makes sense in both cases), while the additional responsibility for checking that arguments satisfy S_1 will be discharged elsewhere (by another cast, with a different label).

We compare and contrast the latent and manifest semantics in Chapter 2. This lays the foundation: what are the semantics of contracts, and how do existing semantics in the literature relate to each other?

1.2 Contracts and abstraction

In Chapter 3, I study how contracts and type abstraction interact. Type abstraction is an essential part of language design, especially for higher-order languages. If we are going to design programming languages with manifest contracts, it's important that we know that type abstraction still follows our intuition.

It turns out that manifest contracts not only enjoy relational parametricity, the gold standard property of type abstraction, but that manifest contracts allow programmers to design particularly rich abstractions. Contracts and *polymorphism* make a natural combination: programmers can give strong contracts to abstract types, precisely stating pre- and post-conditions while hiding implementation details—for example, an abstract type of stacks might specify that the pop operation requires non-empty stacks as input, i.e., have the domain type $\{x{:}\alpha\ \mathsf{Stack} \mid \mathsf{not}\ (\mathsf{empty?}\ x)\}$. That is, just as the type abstraction of ML was used to force sound reasoning in proofs [32], we can use type abstraction to force sound uses of partial operations.

As one simple example, consider the following abstract type for natural numbers:

$$\begin{aligned}\mathsf{NAT} : \exists\alpha.\quad & (\mathsf{zero} : \alpha) \times (\mathsf{succ} : (\alpha \to \alpha)) \times (\mathsf{iszero} : (\alpha \to \mathsf{Bool})) \times \\ & (\mathsf{pred} : \{x{:}\alpha \mid \mathsf{not}\,(\mathsf{iszero}\,x)\} \to \alpha).\end{aligned}$$

Contracts have allowed us to specify a very precise type for pred—one which ensures the correctness of the function, since $\mathsf{pred}\ \mathsf{zero}$ isn't a natural. Since these are *manifest* contracts, the type system will track which values are known to be non-zero. In Section 3.1, I develop a more extended example of an embedded domain-specific language of transducers based on Boomerang [12].

Beyond these rich abstractions, subtyping and relational parametricity offer structured reasoning principles that are invaluable to compiler writers (for optimizations

$$
\begin{array}{ll}
 & \mathsf{odd}\ 3 \\
\longrightarrow & (\langle \mathsf{Dyn}\to\mathsf{Dyn} \Rightarrow \mathsf{Int}\to\mathsf{Bool}\rangle\ \mathsf{even})\ 2 \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\mathsf{even}\ (\langle \mathsf{Int} \Rightarrow \mathsf{Dyn}\rangle\ 2)) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\mathsf{even}\ 2_{\mathsf{Int!}}) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ ((\langle \mathsf{Int}\to\mathsf{Bool} \Rightarrow \mathsf{Dyn}\to\mathsf{Dyn}\rangle\ (\lambda x{:}\mathsf{Int}.\ \ldots))\ 2_{\mathsf{Int!}}) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)\ (\langle \mathsf{Dyn} \Rightarrow \mathsf{Int}\rangle\ 2_{\mathsf{Int!}}))) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)\ 2)) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ (\mathsf{odd}\ 1)) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ ((\langle \mathsf{Dyn}\to\mathsf{Dyn} \Rightarrow \mathsf{Int}\to\mathsf{Bool}\rangle\ \mathsf{even})\ 0)) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ (\langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\mathsf{even}\ (\langle \mathsf{Int} \Rightarrow \mathsf{Dyn}\rangle\ 0)))) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ (\langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\mathsf{even}\ 0_{\mathsf{Int!}}))) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ (\langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle \\
 & \quad ((\langle \mathsf{Int}\to\mathsf{Bool} \Rightarrow \mathsf{Dyn}\to\mathsf{Dyn}\rangle\ (\lambda x{:}\mathsf{Int}.\ \ldots))\ 0_{\mathsf{Int!}}))) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ (\langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle \\
 & \quad ((\lambda x{:}\mathsf{Int}.\ \ldots)\ (\langle \mathsf{Dyn} \Rightarrow \mathsf{Int}\rangle\ 0_{\mathsf{Int!}}))))) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ (\langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle \\
 & \quad ((\lambda x{:}\mathsf{Int}.\ \ldots)\ 0)))) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ (\langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ \mathsf{true}))) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ (\langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ \mathsf{true}_{\mathsf{Bool!}})) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ \mathsf{true}) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ \mathsf{true}_{\mathsf{Bool!}} \\
\longrightarrow & \mathsf{true}
\end{array}
$$

Figure 1.2: Contract checking destroys tail recursion

to representations, deforestation, etc.) and programmers (for formally supported reasoning). These principles are one of the payoffs of the manifest approach.

Finally, our development incorporates a significant technical improvement over earlier presentations of contracts (e.g., Flanagan [28] and Greenberg et al. [34], which is in effect Chapter 2): instead of introducing a denotational model to break a problematic circularity between typing, subtyping, and evaluation, I develop the metatheory of contracts in a completely syntactic fashion, omitting subtyping from the core system and recovering it *post facto* as a derived property.

1.3 Efficiency and contract checking

Since casts and checks have runtime effects, we must ask: what are the costs? Racket's contracts are essentially checked along the same lines as Findler and Felleisen's plan, as in the wrap function:[3]

```
(define (wrap ctc v)
  (if (flat-contract? ctc)
      (if ((flat-contract-pred ctc) v)
```

[3]For simplicity's sake, I've omitted blame.

```
      v
      (error (flat-contract-err-info ctc)))
    (let ([dom (fun-contract-dom ctc)]
          [cod (fun-contract-cod ctc)])
      (lambda (x)
        (wrap cod (v (wrap dom x)))))))
```

That is, flat contracts (like `nonempty-queue/c`) just check the predicate; function contracts η-expand their argument, wrapping the input in the domain contract and the return value in the codomain contract. For example, wrapping `queue?` with its `(-> any/c boolean?)` contract yields:

```
(lambda (x)
  (wrap boolean? (queue? (wrap any/c x))))
```

This η-expanded lambda is called a *function proxy*. In PLT Racket (and all other formulations of higher-order contracts), functions can accumulate an unbounded number of function proxies—with serious ramifications for asymptotic space bounds. Similarly, the codomain contract check in the function proxy can destroy tail recursion.

One longstanding problem with casts is space efficiency: contract checks, in their naïve formulation, can consume unbounded amounts of space at runtime both through excessive function proxies and through tail-recursion-breaking stack growth. This unbounded use of space can lead to changes in the asymptotic space efficiency of programs. Prior work [39, 40, 65, 68] offers space-efficient solutions exclusively in the domain of *gradual types* [67]—type systems integrating simple types and the dynamic type, Dyn.

As an example, take the mutually recursive definition of even and odd from Herman et al. [39]. Here odd is written in a simply typed style, but even is written in a more dynamically typed style.

$$
\begin{aligned}
&\mathsf{even} : \mathsf{Dyn}\to\mathsf{Dyn} = \langle \mathsf{Int}\to\mathsf{Bool} \Rightarrow \mathsf{Dyn}\to\mathsf{Dyn}\rangle\ \lambda x{:}\mathsf{Int}.\\
&\quad \mathsf{if}\ x = 0\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{odd}\ (x-1)\\
&\mathsf{odd} : \mathsf{Int}\to\mathsf{Bool} = \lambda x{:}\mathsf{Int}.\\
&\quad \mathsf{if}\ x = 0\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ (\langle \mathsf{Dyn}\to\mathsf{Dyn} \Rightarrow \mathsf{Int}\to\mathsf{Bool}\rangle\ \mathsf{even})\ (x-1)
\end{aligned}
$$

We show how odd 3 reduces in Figure 1.2. Note how casts between Bool and Dyn accumulate. Imagine writing even and odd in a surface language—how predictable is cast insertion? What was written as tail recursive—and ought to use only $O(1)$ space—instead takes $O(n)$ space.

This problem of contracts breaking tail recursion has very much informed the design philosophy of PLT Racket's contracts [56], where contracts are typically placed at module boundaries. Calls between functions in the same module aren't mediated by contracts, so intra-module tail recursion still works. Calls between modules do have contracts, though, so there's no guarantee about tail recursion. Naturally, this

scheme is somewhat limiting: what about libraries with higher-order functions and callbacks?

Following Herman et al. [39], I develop a space efficient scheme for a language with manifest contracts and type Dyn in Chapter 4. We show that space efficiency avoids some checks, failing and diverging less often than naïve calculi—but the two are otherwise observationally equivalent.

1.4 Summary

In this thesis, I show that **higher-order contracts are on a firm type-theoretic footing and that they support extensions necessary for real programming**. Developing these principles is the first step in designing and implementing higher-order languages with contracts and refinement types. In this dissertation, I:

- Study the relationship between the two families of core calculi for contracts (Chapter 2),
- Extend one family so that it admits type abstraction and enjoys relational parametricity (Chapter 3), and
- Resolve a show-stopping space-inefficiency (Chapter 4).

I discuss related work in Chapter 5 and conclude with a discussion of future work in Chapter 6.

1.5 Notation and other conventions

There are many different calculi in this dissertation: five in Chapter 2, one in Chapter 3, and three in Chapter 4. What's more, all but two of these calculi use manifest contracts, and all but two of those use casts. So there is a significant overlap in notation and names. It is easiest to interpret names as scoped by chapter: there are three different rules named E_App, but each is unique in its chapter. Well formedness and typing rules are named WF_... and T_... (though sometimes also S_...); evaluation rules are named E_..., but also F_... and G_....

We explain the notation in each chapter, but all of the calculi should be familiar as extensions of the simply typed lambda calculus.

This document is thoroughly hyperlinked—references to chapters, sections, figures, theorems, and citations are all linked. While this document reads best in color and on a device supporting linking, I have not marked the links in order to reduce visual noise.

Chapter 2

Contracts made manifest

> And something is happening here
> But you don't know what it is
>
> *Ballad of a Thin Man*
> Bob Dylan

Recall the distinction between latent and manifest systems made in Section 1.1: latent contracts are orthogonal to the type system (if it exists), while manifest contracts conflate contracts and types. While contract checks in latent systems may seem intuitively to be much the same thing as casts in manifest systems, the formal correspondence is not immediate. Manifest contracts used casts $\langle S_1 \Rightarrow S_2 \rangle^l$ with contravariant checking for functions; latent contracts used obligations $\langle c \rangle^{l,l'}$ with an invariant rule—but with polarized blame labels. How do these forms of checking and blame relate? These questions have led to some confusion in the community about the nature of contracts. Indeed, as we will see, matters become yet murkier in richer languages with features such as dependency.

Gronski and Flanagan [36] initiated a formal investigation of the connection between the latent and manifest worlds. They defined a core calculus, λ_{C}, capturing the essence of latent contracts in a simply typed lambda-calculus, and an analogous manifest calculus λ_{H}. I will follow the naming scheme of their treatment. Both calculi offer contract-like dynamic checks, but they differ in the exact structure of these checks, in their treatment of "blame," and in how dynamic checks and types interact.

In λ_{C}, dynamic checks occur when a contract is applied, as in $\langle c \rangle^{l,l'}\ t$. The contract checks to make sure that t is treated and behaves like a c. To achieve the same effect, λ_{H} applies a cast, $\langle S_1 \Rightarrow S_2 \rangle^l\ s$. This cast ensures that s—which used to behave like an S_1—is now treated and behaves like an S_2, where S_1 and S_2 are (very precise) types. For example, $\langle \{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\} \rangle^{l,l'}\ t$ ensures that t evaluates to a positive integer; $\langle \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\} \rangle^l\ s$ ensures that s—which we know to be nonzero—is also positive.

To compare these systems, they introduced a type-preserving translation ϕ from λ_{C} to λ_{H}. What makes ϕ interesting is that it relates the languages *feature for feature*:

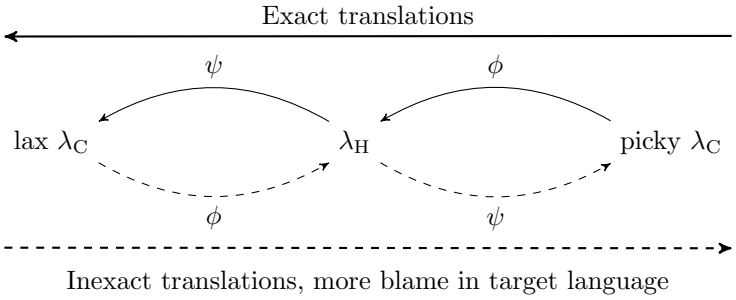


Figure 2.1: The axis of blame

predicate contracts over base types are mapped to casts between refinements of base type, and function contracts are mapped to function casts. The main result is that ϕ preserves behavior, in the sense that if a term t in λ_C evaluates to a constant k or blame $\Uparrow l$, then its translation $\phi(t)$ evaluates similarly.

In this chapter, I extend their work in two directions. First, I strengthen their main result by introducing a new feature-for-feature translation ψ from λ_H to λ_C and proving a similar correspondence theorem for ψ. (I also give a new, more detailed, proof of the correspondence theorem for ϕ.) These correspondences show that the manifest and latent approaches are effectively equivalent in the nondependent case.

Second, and more significantly, I extend the whole story to allow dependent function contracts in λ_C and dependent arrow types in λ_H. Dependency is extremely handy in contracts, as it allows for precise specifications of how the results of functions depend on their arguments. For example, here is a contract that we might use with an implementation of vector concatenation:

$$z_1{:}\mathsf{Vec} \mapsto z_2{:}\mathsf{Vec} \mapsto \{z_3{:}\mathsf{Vec} \mid \mathsf{vlen}\ z_3 = \mathsf{vlen}\ z_1 + \mathsf{vlen}\ z_2\}$$

Adding dependent contracts to λ_C is easy: the dependency is all in the contracts and the types stay simple. We have just one significant design choice: should domain contracts be rechecked when the bound variable appears in the codomain contract? This choice leads to two dialects of λ_C, one which does recheck (*picky* λ_C) and one which does not (*lax* λ_C). The choice is not clear, so I consider both. The question of which blame labels belong on this extra check is discussed at length in Dimoulas et al. [22], which introduces *indy* blame. Indy blame is a variant of picky. I do not consider it in depth here, since it does not affect *whether or not* blame is raised, only *which* blame. I discuss this point more in Section 2.6.3. In λ_H, on the other hand, dependency significantly complicates the metatheory, requiring the addition of a denotational semantics for types and kinds (a unary logical relation with a term model) to break a potential circularity in the definitions, plus an intricate sequence of technical lemmas involving parallel reduction to establish type soundness.

Surprisingly, the tight correspondence between λ_C and λ_H breaks down in the dependent case: the natural generalization of the translations does not preserve behavior exactly. Indeed, we can place λ_H between the two variants of λ_C on an "axis

of blame" (Figure 2.1), where evaluation behavior is preserved exactly when moving left on the axis (from picky λ_{C} to λ_{H} to lax λ_{C}), but translated terms can blame more than their pre-images when moving right.[1] It is still the case that when a pre-image raises blame, its translation blames as well—though not necessarily the same label. The discrepancy arises in the case of *abusive* contracts, such as

$$f{:}(\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \mapsto \{y{:}\mathsf{Int} \mid \mathsf{true}\}) \mapsto \{z{:}\mathsf{Int} \mid f\ 0 = 0\}$$

This rather strange contract has the form $f{:}c_1 \mapsto c_2$, where c_2 uses f in a way that violates c_1! In particular, if we apply it (in lax λ_{C}) to $\lambda f{:}\mathsf{Int} \to \mathsf{Int}.\ 0$ and then apply the result to $\lambda x{:}\mathsf{Int}.\ x$ and 5, the final result will be 5, since $\lambda x{:}\mathsf{Int}.\ x$ does satisfy the contract $\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \mapsto \{y{:}\mathsf{Int} \mid \mathsf{true}\}$ and 5 satisfies the contract $\{z{:}\mathsf{Int} \mid (\lambda x{:}\mathsf{Int}.\ x)\ 0 = 0\}$. However, running the translation of f in λ_{H} yields an extra check, wrapping the occurrence of f in the codomain contract with a cast from $\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \to \{y{:}\mathsf{Int} \mid \mathsf{true}\}$ to $\{x{:}\mathsf{Int} \mid \mathsf{true}\} \to \{y{:}\mathsf{Int} \mid \mathsf{true}\}$, which fails when the wrapped function is applied to 0. I discuss this phenomenon in greater detail in Section 2.3.

We should note at the outset that, like Gronski and Flanagan [36], we are interested in translations that relate λ_{C} and λ_{H} feature for feature, i.e., mapping predicate contracts to predicate contracts and function contracts to function contracts. Translations which don't map feature for feature can give an exact treatment of blame. Consider the following dependent version of the wrap operator from Findler and Felleisen [26]. There are two cases: one for refinements of base types B, one for dependent function contracts.

$$\begin{array}{rcl}
\phi(\langle\{x{:}B \mid t\}\rangle^{l,l'}) & = & \langle\lceil B\rceil \Rightarrow \{x{:}B \mid \phi(t)\}\rangle^{l} \\
\phi(\langle x{:}c_1 \mapsto c_2\rangle^{l,l'}) & = & \lambda f{:}\lceil x{:}c_1 \mapsto c_2\rceil. \\
& & \quad \lambda x{:}\lceil c_1\rceil. \\
& & \quad\ \phi(\langle c_2\rangle^{l,l'})\ (f\ (\phi(\langle c_1\rangle^{l',l})\ x))
\end{array}$$

We can define a similar mapping function that implements λ_{H}'s semantics as predicate contracts in lax or picky λ_{C}. It is unsurprising that an exact mapping exists: λ_{C} and λ_{H} are lambda calculi that feature, among other things, a way to conditionally raise exceptions. That these languages are interencodable is completely unsurprising. But translations like these do not relate function contracts to function casts at all, so they do not do much to tell us about how semantics of contracts and the semantics of casts relate.

In summary, the main contributions of this chapter are (a) the translation ψ and a symmetric version of Gronski and Flanagan's behavioral correspondence theorem, (b) the basic metatheory of (CBV, blame-sensitive) dependent λ_{H}, (c) dependent

[1] There might, in principle, be some other way of defining ϕ and ψ that (a) preserves types, (b) maps feature for feature, and (c) induces an exact behavioral equivalence. After considering a number of alternatives, we conjecture that no such ϕ and ψ exist.

$$
\begin{array}{rcll}
B & ::= & \mathsf{Bool} \mid \ldots & \text{base types} \\
k & ::= & \mathsf{true} \mid \mathsf{false} \mid \ldots & \text{first-order constants}
\end{array}
$$

Figure 2.2: Base types and constants for λ_{C} and λ_{H}

versions of ϕ and ψ and their properties with regard to λ_{H} and both dialects of λ_{C}, and (d) a weaker behavioral correspondence in the dependent case. I restrict my attention to strongly normalizing programs, though I believe the results should generalize readily to programs with recursion and nontermination. This chapter is an extended adaptation of Greenberg et al. [34] and Greenberg et al. [35].

2.1 The nondependent languages

We begin in this section by defining the nondependent versions of λ_{C} and λ_{H} and continue in Section 2.2 with the translations between them. The dependent languages, dependent translations, and their properties are developed in Sections 2.3, 2.5, and 2.6. Throughout this chapter, rules prefixed with an E or an F are operational rules for λ_{C} and λ_{H}, respectively. An initial T is used for λ_{C} typing rules; typing rules beginning with an S belong to λ_{H}.

All of the languages will share a set of base types and first-order constants, given in Figure 2.2. Let the set $\mathcal{K}_B$ contain constants of base type B. We assume that Bool is among the base types, with $\mathcal{K}_{\mathsf{Bool}} = \{\mathsf{true}, \mathsf{false}\}$.

2.1.1 The language λ_{C}

The language λ_{C} is the simply typed lambda calculus straightforwardly augmented with contracts. *Contracts* c come in two forms: predicate contracts $\{x{:}B \mid t\}$ over a base type B and higher-order contracts $c_1 \mapsto c_2$, which check the arguments and results of functions. We can use contracts in terms with the *contract obligation* $\langle c \rangle^{l,l'}$. Applying a contract obligation $\langle c \rangle^{l,l'}$ to a term t dynamically ensures that t and its surrounding context satisfy c. If t does not satisfy c, then the *positive* label l will be blamed and the whole term will reduce to $\Uparrow l$; on the other hand, if the context does not treat $\langle c \rangle^{l,l'}\ t$ as c demands, then the negative label l' will be blamed and the term will reduce to $\Uparrow l'$. In contexts where it is unambiguous, I refer to contract obligations simply as contracts.

The syntax and semantics of λ_{C} appears in Figure 2.3, with some common definitions (shared with λ_{H}) in Figure 2.2. Besides the contract term $\langle c \rangle^{l,l'}$, λ_{C} includes first-order constants k, blame, and *active checks* $\langle \{x{:}B \mid t_1\}, t_2, k \rangle^{l}$. Active checks do not appear in source programs; they are a technical artifact of the small-step operational semantics, as I explain below. Also, note that my contracts only allow refinements over base types B: we have function contracts, like $\{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\} \mapsto \{x{:}\mathsf{Int} \mid$

Syntax for $\lambda_{\mathbf{C}}$

$$
\begin{array}{rcll}
T & ::= & B \mid T_1 \to T_2 & \text{types} \\
c & ::= & \{x{:}B \mid t\} \mid c_1 \mapsto c_2 & \text{contracts} \\
\\
t & ::= & x \mid k \mid \lambda x{:}T_1.\ t_2 \mid t_1\ t_2 \mid & \text{terms} \\
 & & \Uparrow l \mid \langle c \rangle^{l,l'} \mid \langle \{x{:}B \mid t_1\}, t_2, k \rangle^{l} & \\
v & ::= & k \mid \lambda x{:}T_1.\ t_2 \mid \langle c \rangle^{l,l'} \mid \langle c_1 \mapsto c_2 \rangle^{l,l'}\ v & \text{values} \\
r & ::= & v \mid \Uparrow l & \text{results} \\
E & ::= & [\,]\,t \mid v\,[\,] \mid \langle \{x{:}B \mid t\}, [\,], k \rangle^{l} & \text{evaluation contexts}
\end{array}
$$

Operational semantics for $\lambda_{\mathbf{C}}$

$$
\begin{array}{rcll}
(\lambda x{:}T_1.\ t_2)\ v & \longrightarrow_c & t_2\{x := v\} & \text{E\_Beta} \\
k\ v & \longrightarrow_c & [\![k]\!](v) & \text{E\_Const} \\
\langle \{x{:}B \mid t\} \rangle^{l,l'}\ k & \longrightarrow_c & \langle \{x{:}B \mid t\}, t\{x := k\}, k \rangle^{l} & \text{E\_CCheck} \\
\langle \{x{:}B \mid t\}, \mathsf{true}, k \rangle^{l} & \longrightarrow_c & k & \text{E\_OK} \\
\langle \{x{:}B \mid t\}, \mathsf{false}, k \rangle^{l} & \longrightarrow_c & \Uparrow l & \text{E\_Fail} \\
(\langle c_1 \mapsto c_2 \rangle^{l,l'}\ v)\ v' & \longrightarrow_c & \langle c_2 \rangle^{l,l'}\ (v\ (\langle c_1 \rangle^{l',l}\ v')) & \text{E\_CDecomp} \\
E\,[\Uparrow l] & \longrightarrow_c & \Uparrow l & \text{E\_Blame} \\
E\,[t_1] & \longrightarrow_c & E\,[t_2] \quad \text{when } t_1 \longrightarrow_c t_2 & \text{E\_Compat}
\end{array}
$$

Typing rules for $\lambda_{\mathbf{C}}$

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

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T}\ \text{T\_Var} \qquad \frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.\ t_2 : T_1 \to T_2}\ \text{T\_Lam}$$

$$\frac{}{\Gamma \vdash k : \mathrm{ty_c}(k)}\ \text{T\_Const} \qquad \frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}\ \text{T\_App}$$

$$\frac{}{\Gamma \vdash \Uparrow l : T}\ \text{T\_Blame} \qquad \frac{\vdash c : T}{\Gamma \vdash \langle c \rangle^{l,l'} : T \to T}\ \text{T\_Contract}$$

$$\frac{\begin{array}{c}\emptyset \vdash k : B \qquad \emptyset \vdash t_2 : \mathsf{Bool} \qquad \vdash \{x{:}B \mid t_1\} : B \\ t_2 \longrightarrow_c^* \mathsf{true} \quad \text{implies} \quad t_1\{x := k\} \longrightarrow_c^* \mathsf{true}\end{array}}{\emptyset \vdash \langle \{x{:}B \mid t_1\}, t_2, k \rangle^{l} : B}\ \text{T\_Checking}$$

$\boxed{\vdash c : T}$

$$\frac{x{:}B \vdash t : \mathsf{Bool}}{\vdash \{x{:}B \mid t\} : B}\ \text{T\_BaseC} \qquad \frac{\vdash c_1 : T_1 \qquad \vdash c_2 : T_2}{\vdash c_1 \mapsto c_2 : T_1 \to T_2}\ \text{T\_FunC}$$

Figure 2.3: Syntax and semantics for λ_{C}

$\mathsf{nonzero}\ x\}$, but not predicate contracts over functions themselves, like $\{f{:}\mathsf{Bool} \to \mathsf{Bool} \mid f\ \mathsf{true} = f\ \mathsf{false}\}$.

Values v include constants, abstractions, contracts, and function contracts applied to values (more on these later); a *result* r is either a value or $\Uparrow l$ for some l. We interpret constants using two constructions: the type-assignment function $\mathrm{ty_c}$, which maps constants to first-order types of the form $B_1 \to B_2 \to \ldots \to B_n$ (and which is assumed to agree with $\mathcal{K}_B$); and the denotation function $[\![-]\!]$, which maps constants to functions from constants to constants (or blame, to allow for partiality). Denotations must agree with $\mathrm{ty_c}$, i.e., if $\mathrm{ty_c}(k) = B_1 \to B_2$, then $[\![k]\!](k_1) \in \mathcal{K}_{B_2}$ if $k_1 \in \mathcal{K}_{B_1}$.

The operational semantics is given in Figure 2.3. It includes six rules for basic (small-step, call-by-value) reductions, plus two rules that involve evaluation contexts E (Figure 2.3). The evaluation contexts implement left-to-right evaluation for function application. If $\Uparrow l$ appears in the active position of an evaluation context, it is propagated to the top level, like an uncatchable exception. As usual, values (and results) do not step.

The first two basic rules are standard, implementing primitive reductions and β-reductions for abstractions. In these rules, arguments must be values v. Since constants are first-order, we know that when E_Const reduces a well-typed application, the argument is not just a value, but a constant.

The rules E_CCheck, E_OK, E_Fail and E_CDecomp, describe the semantics of contracts. In E_CCheck, predicate contracts applied to constants step to an active check. Active checks include the original contract, the current state of the check, the constant being checked, and a label to blame if necessary. I hold on to the original contract as a technical device for the translation ϕ from λ_{C} to λ_{H}, since λ_{H} needs to know the target type of an active check. If the check evaluates to true, then E_OK returns the initial constant. If false, the check has failed and a contract has been violated, so E_Fail steps the term to $\Uparrow l$. Higher-order contracts on a value v wait to be applied to an additional argument. This is why function contracts applied to values are values. There is no substantial difference between this approach and expanding function contracts into new lambdas. When that argument has also been reduced to a value v', E_CDecomp decomposes the function cast: the argument value is checked with the argument part of the contract (switching positive and negative blame, since the context is responsible for the argument), and the result of the application is checked with the result contract.

The typing rules for λ_{C} (Figure 2.3) are mostly standard. We give types to constants using the type-assignment function $\mathrm{ty_c}$. Blame expressions have all types. Contracts are checked for well-formedness using the judgment $\vdash c : T$, comprising the rules T_BaseC, which requires that the checking term in a predicate contract return a boolean value when supplied with a term of the right type, and T_FunC. Note that the predicate t in a contract $\{x{:}B \mid t\}$ can contain at most x free, since we are considering only nondependent contracts for now. Contract application, like function application, is checked using T_App.

The T_CHECKING rule only applies in the empty context (active checks are only created at the top level during evaluation). The rule ensures that the contract $\{x{:}B \mid t_1\}$ has the right base type for the constant k, that the check expression t_2 has a boolean type, and that the check is actually checking the right contract. The latter condition is formalized by the implication: $t_2 \longrightarrow_c^* \mathsf{true}$ implies $t_1\{x := k\} \longrightarrow_c^* \mathsf{true}$ asserts that if t_2 evaluates to true, then the original check $t_1\{x := k\}$ must also evaluate to true. This requirement is needed for two reasons: first, nonsensical terms like $\langle\{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\}, \mathsf{true}, 0\rangle^l$ should not be well typed; and second, I use this property in showing that the translations are type preserving (see Section 2.5). One may think that this rule makes typechecking for the full "internal language" with checks undecidable—I certainly did—but in fact the entire language is strongly normalizing.[2] I could give a more precise condition—for example, that $t_1\{x := k\} \longrightarrow_c^* t_2$—but there is no need. We will find that this condition is more useful in the metatheory of Chapters 3 and 4.

The language enjoys standard preservation and progress theorems. Together, these ensure that evaluating a well-typed term to a normal form always yields a result r, which is either blame or a value of the appropriate type.

2.1.2 The language λ_{H}

The second calculus, nondependent λ_{H}, extends the simply typed lambda-calculus with *refinement types* and *cast expressions*. The definitions appear in Figure 2.4 (syntax and semantics) and Figure 2.5. Unlike λ_{C}, which separates contracts from types, λ_{H} combines them into refined base types $\{x{:}B \mid s_1\}$ and function types $S_1 \to S_2$. As for λ_{C}, I do not allow refinement types over functions nor refinements of refinements. (We add these features to a dependent λ_{H} in Chapter 3.) Unrefined base types B are *not* valid types; they must be wrapped in a trivial refinement, as the *raw* type $\{x{:}B \mid \mathsf{true}\}$. The terms of the language are mostly standard, including variables, the same first-order constants as λ_{C}, blame, abstractions, and applications. The cast expression $\langle S_1 \Rightarrow S_2\rangle^l$ dynamically checks that a term of type S_1 can be given type S_2. Like λ_{C}, active checks are used to give a small-step semantics to cast expressions.

The values of λ_{H} include constants, abstractions, casts, and function casts applied to values. Results are either values or blame. We give meaning to constants as we did in λ_{C}, reusing $[\![-]\!]$. Type assignment is via $\mathrm{ty_h}$, which we assume produces well-formed types (defined in Figure 2.5). To keep the languages in sync, I require that $\mathrm{ty_h}$ and $\mathrm{ty_c}$ agree on "type skeletons": if $\mathrm{ty_c}(k) = B_1 \to B_2$, then $\mathrm{ty_h}(k) = \{x{:}B_1 \mid s_1\} \to \{x{:}B_2 \mid s_2\}$.

The small-step, call-by-value semantics in Figure 2.4 comprises six basic rules and two rules involving evaluation contexts F. Each rule corresponds closely to its λ_{C} counterpart.

[2] As observed by Stephan Zdancewic at my defense.

Syntax for $\lambda_{\mathbf{H}}$

$$
\begin{array}{rcll}
S & ::= & \{x{:}B \mid s_1\} \mid S_1 \to S_2 & \text{types/contracts} \\
\\
s & ::= & x \mid k \mid \lambda x{:}S_1.\ s_2 \mid s_1\ s_2 \mid & \text{terms} \\
 & & \Uparrow l \mid \langle S_1 \Rightarrow S_2 \rangle^l \mid \langle \{x{:}B \mid s_1\}, s_2, k \rangle^l & \\
w & ::= & k \mid \lambda x{:}S_1.\ s_2 \mid \langle S_1 \Rightarrow S_2 \rangle^l \mid & \text{values} \\
 & & \langle S_{11} \to S_{12} \Rightarrow S_{21} \to S_{22} \rangle^l\ w & \\
q & ::= & w \mid \Uparrow l & \text{results} \\
F & ::= & [\,]\, s \mid w\, [\,] \mid \langle \{x{:}B \mid s\}, [\,], k \rangle^l & \text{evaluation contexts}
\end{array}
$$

Operational semantics for $\lambda_{\mathbf{H}}$

$$
\begin{array}{rcll}
(\lambda x{:}S_1.\ s_2)\ w_2 & \longrightarrow_h & s_2\{x := w_2\} & \text{F\_Beta} \\
k\ w & \longrightarrow_h & [\![k]\!](w) & \text{F\_Const} \\
\langle \{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid s_2\} \rangle^l\ k & \longrightarrow_h & \langle \{x{:}B \mid s_2\}, s_2\{x := k\}, k \rangle^l & \text{F\_CCheck} \\
\langle \{x{:}B \mid s\}, \mathsf{true}, k \rangle^l & \longrightarrow_h & k & \text{F\_OK} \\
\langle \{x{:}B \mid s\}, \mathsf{false}, k \rangle^l & \longrightarrow_h & \Uparrow l & \text{F\_Fail} \\
(\langle S_{11} \to S_{12} \Rightarrow S_{21} \to S_{22} \rangle^l\ w)\ w' & \longrightarrow_h & \langle S_{12} \Rightarrow S_{22} \rangle^l\ (w\ (\langle S_{21} \Rightarrow S_{11} \rangle^l\ w')) & \text{F\_CDecomp} \\
F\,[\Uparrow l] & \longrightarrow_h & \Uparrow l & \text{F\_Blame} \\
F\,[s_1] & \longrightarrow_h & F\,[s_2] \quad \text{when } s_1 \longrightarrow_h s_2 & \text{F\_Compat}
\end{array}
$$

Figure 2.4: Syntax and semantics for λ_{H}

Typing rules for $\lambda_{\mathbf{H}}$

$$\boxed{\Delta \vdash s : S}$$

$$\frac{x{:}S \in \Delta}{\Delta \vdash x : S}\ \text{S\_Var} \qquad \frac{\vdash S_1 \qquad \Delta, x{:}S_1 \vdash s_2 : S_2}{\Delta \vdash \lambda x{:}S_1.\ s_2 : S_1 \to S_2}\ \text{S\_Lam}$$

$$\frac{}{\Delta \vdash k : \mathrm{ty_h}(k)}\ \text{S\_Const} \qquad \frac{\Delta \vdash s_1 : S_1 \to S_2 \qquad \Delta \vdash s_2 : S_1}{\Delta \vdash s_1\ s_2 : S_2}\ \text{S\_App}$$

$$\frac{\vdash S}{\Delta \vdash \Uparrow l : S}\ \text{S\_Blame} \qquad \frac{\vdash S_1 \qquad \vdash S_2 \qquad \lfloor S_1 \rfloor = \lfloor S_2 \rfloor}{\Delta \vdash \langle S_1 \Rightarrow S_2 \rangle^{l} : S_1 \to S_2}\ \text{S\_Cast}$$

$$\frac{\Delta \vdash s : S_1 \qquad \vdash S_2 \qquad \vdash S_1 <: S_2}{\Delta \vdash s : S_2}\ \text{S\_Sub}$$

$$\frac{\begin{array}{l}\emptyset \vdash k : \{x{:}B \mid \mathsf{true}\} \qquad \emptyset \vdash s_2 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\} \qquad \vdash \{x{:}B \mid s_1\} \\ s_2 \longrightarrow_h^* \mathsf{true} \quad \text{implies} \quad s_1\{x := k\} \longrightarrow_h^* \mathsf{true}\end{array}}{\emptyset \vdash \langle \{x{:}B \mid s_1\}, s_2, k \rangle^{l} : \{x{:}B \mid s_1\}}\ \text{S\_Checking}$$

$$\boxed{\vdash S_1 <: S_2}$$

$$\frac{\forall k \in \mathcal{K}_B.\ (s_1\{x := k\} \longrightarrow_h^* \mathsf{true} \quad \text{implies} \quad s_2\{x := k\} \longrightarrow_h^* \mathsf{true})}{\vdash \{x{:}B \mid s_1\} <: \{x{:}B \mid s_2\}}\ \text{SSub\_Refine}$$

$$\frac{\vdash S_{21} <: S_{11} \qquad \vdash S_{12} <: S_{22}}{\vdash S_{11} \to S_{12} <: S_{21} \to S_{22}}\ \text{SSub\_Fun}$$

$$\boxed{\vdash S}$$

$$\frac{}{\vdash \{x{:}B \mid \mathsf{true}\}}\ \text{SWF\_Raw} \qquad \frac{\vdash S_1 \qquad \vdash S_2}{\vdash S_1 \to S_2}\ \text{SWF\_Fun}$$

$$\frac{x{:}\{x{:}B \mid \mathsf{true}\} \vdash s : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}}{\vdash \{x{:}B \mid s\}}\ \text{SWF\_Refine}$$

Figure 2.5: Typing rules for λ_{H}

Notice how the decomposition rules compare. In λ_{C}, the term $(\langle c_1 \mapsto c_2 \rangle^{l,l'}\ v)\ v'$ decomposes into two contract checks: c_1 checks the argument v' and c_2 checks the result of the application. In λ_{H} the term $(\langle S_{11} \to S_{12} \Rightarrow S_{21} \to S_{22} \rangle^{l}\ w)\ w'$ decomposes into two casts: a contravariant cast on the argument and a covariant cast on theresult. The contravariant cast $\langle S_{21} \Rightarrow S_{11} \rangle^{l}\ w'$ makes w' a suitable input for w, while $\langle S_{12} \Rightarrow S_{22} \rangle^{l}$ casts the result from w applied to (the cast) w'. Suppose $S_{21} = \{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\}$ and $S_{11} = \{x{:}B \mid \mathsf{nonzero}\ x\}$. Then the check on the argument ensures that $\mathsf{nonzero}\ x \longrightarrow_h^* \mathsf{true}$—not, as one might expect, that $\mathsf{pos}\ w' \longrightarrow_h^* \mathsf{true}$. While it is easy to read off from a λ_{C} contract exactly which checks will occur at runtime, a λ_{H} cast must be carefully inspected to see exactly which checks will take place. On the other hand, which label will be blamed is clearer with casts—there's only one!

The typing rules for λ_{H} (Figure 2.5) are also similar to those of λ_{C}. Just as the λ_{C} rule T_Contract checks to make sure that the contract has the right form, the λ_{H} rule S_Cast ensures that the two types in a cast are well-formed and have the same simple-type skeleton, defined as $\lfloor - \rfloor : S \to T$ (pronounced "erase S"):

$$\begin{aligned} \lfloor \{x{:}B \mid s\} \rfloor &= B \\ \lfloor S_1 \to S_2 \rfloor &= \lfloor S_1 \rfloor \to \lfloor S_2 \rfloor \end{aligned}$$

This prevents "stupid" casts, like $\langle \lceil \mathsf{Int} \rceil \Rightarrow \lceil \mathsf{Bool} \rceil \rangle^{l}$. I define a similar operator, $\lceil - \rceil : S \to S$ (pronounced "raw S"), which trivializes all refinements:

$$\begin{aligned} \lceil \{x{:}B \mid s\} \rceil &= \{x{:}B \mid \mathsf{true}\} \\ \lceil S_1 \to S_2 \rceil &= \lceil S_1 \rceil \to \lceil S_2 \rceil \end{aligned}$$

These operations apply to λ_{C} contracts and types in the natural way. Type well-formedness in λ_{H} is similar to contract well-formedness in λ_{C}, though the WF_Raw case needs to be added to get things off the ground.

The active check rule S_Checking plays a role analogous to the T_Checking rule in λ_{C}, again using an implication to guarantee that we only have sensible terms in the predicate position. Note that we retain the target type in the active check, and that S_Checking gives active checks that type—technical moves necessary for preservation.

An important difference is that λ_{H} has subtyping. The S_Sub rule allows an expression to be promoted to any well-formed supertype. Refinement types are supertypes if, for all constants of the base type, their condition evaluates to true whenever the subtype's condition evaluates to true. For function types, we use the standard contravariant subtyping rule. I do not consider source programs with subtyping, since subtyping makes type checking undecidable[3]; subtyping is just a technical device for ensuring type preservation. Consider the following reduction:

$$\langle \{x{:}\mathsf{Int} \mid \mathsf{true}\} \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\} \rangle^{l}\ 1 \longrightarrow_h^* 1$$

[3]Flanagan [28] and Knowles and Flanagan [44] discuss trade-offs between static and dynamic checking that allow for decidable type systems and subtyping.

The source term is well-typed at $\{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\}$. Since it evaluates to 1, we would like to have $\Delta \vdash 1 : \{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\}$. To have type preservation in general, though, $\mathrm{ty_h}(1)$ must be a subtype of $\{x{:}\mathsf{Int} \mid s\}$ *whenever* $s\{x := 1\} \longrightarrow_h^* \mathsf{true}$. That is, constants of base type must have "most-specific" types. One way to satisfy this requirement is to set $\mathrm{ty_h}(k) = \{x{:}B \mid x = k\}$ for $k \in \mathcal{K}_B$; then if $s\{x := k\} \longrightarrow_h^* \mathsf{true}$, we have $\vdash \mathrm{ty_h}(k) <: \{x{:}B \mid s\}$. This approach is taken in Knowles and Flanagan [44] and Ou et al. [51].

Standard progress and preservation theorems hold for λ_{H}. We can also obtain a semantic type soundness theorem as a restriction of the one for dependent λ_{H} (Theorem 2.3.12).

2.2 The nondependent translations

The latent and manifest calculi differ in a few respects. Obviously, λ_{C} uses contract application and λ_{H} uses casts. Second, λ_{C} contracts have two labels—positive and negative—where λ_{H} contracts have a single label. Finally, λ_{H} has a much richer type system than λ_{C}. The translation ψ from λ_{H} to λ_{C} and Gronski and Flanagan's ϕ from λ_{C} to λ_{H} must account for these differences while carefully mapping "feature for feature".

Since I give the translations for the dependent languages in full in Section 2.3, I merely sketch the main ideas here.

The interesting parts of the translations deal with contracts and casts. Everything else is translated homomorphically, though the type annotation on lambdas must be chosen carefully. The full definitions of these translations are in Section 2.4; the nondependent definitions are a straightforward restriction, so I omit them.

For ψ, translating from λ_{H}'s rich types to λ_{C}'s simple types is easy: we just erase the types to their simple skeletons. The interesting case is translating the cast $\langle S_1 \Rightarrow S_2 \rangle^l$ to a contract by translating the pair of types together, $\langle \psi(S_1, S_2) \rangle^{l,l}$. I define ψ as two mutually recursive functions: $\psi(s)$ translates λ_{H} terms to λ_{C} terms; $\psi(S_1, S_2)$ translates a pair of λ_{H} types—effectively, a cast—to a λ_{C} contract. The latter function is defined as follows:

$$\begin{array}{rcl} \psi(\{x{:}B \mid s_1\}, \{x{:}B \mid s_2\}) & = & \{x{:}B \mid \psi(s_2)\} \\ \psi(S_{11} \to S_{12}, S_{21} \to S_{22}) & = & \psi(S_{21}, S_{11}) \mapsto \psi(S_{12}, S_{22}) \end{array}$$

We use the single label on the cast in both the positive and negative positions of the resulting contract, i.e.:

$$\psi(\langle S_1 \Rightarrow S_2 \rangle^l) = \langle \psi(S_1, S_2) \rangle^{l,l}.$$

When we translate a pair of refinement types, we produce a contract that will check the predicate of the target type (like F_CCheck); when translating a pair of function types, we translate the domain contravariantly (like F_CDecomp). For example,

$$\langle \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \to \lceil \mathsf{Int} \rceil \Rightarrow \lceil \mathsf{Int} \rceil \to \{y{:}\mathsf{Int} \mid \mathsf{pos}\ y\} \rangle^l$$

translates to $\langle\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \mapsto \{y{:}\mathsf{Int} \mid \mathsf{pos}\ y\}\rangle^{l,l}$.

Translating from λ_{C} to λ_{H}, we are moving from a simple type system to a rich one. The translation ϕ (essentially the same as Gronski and Flanagan's) generates terms in λ_{H} with *raw* types—λ_{H} types with trivial refinements, corresponding to λ_{C}'s simple types. Since the translation targets raw types, the type preservation theorem is stated as "if $\Gamma \vdash t : T$ then $\lceil\Gamma\rceil \vdash \phi(t) : \lceil T\rceil$" (see Section 2.6.1).

Whereas the difficulty with ψ is ensuring that the checks match up, the difficulty with ϕ is ensuring that the terms in λ_{C} and λ_{H} will blame the same labels. I deal with this problem by translating a single contract with two blame labels into two separate casts. Intuitively, the cast carrying the negative blame label will run all of the checks in negative positions in the contract, while the cast with the positive blame label will run the positive checks. Let

$$\phi(\langle c\rangle^{l,l'}) = \lambda x{:}\lceil c\rceil.\ \langle\phi(c) \Rightarrow \lceil c\rceil\rangle^{l'}\ (\langle\lceil c\rceil \Rightarrow \phi(c)\rangle^{l}\ x),$$

where the translation of contracts to refined types is:

$$\begin{array}{rcl} \phi(\{x{:}B \mid t\}) & = & \{x{:}B \mid \phi(t)\} \\ \phi(c_1 \mapsto c_2) & = & \phi(c_1) \to \phi(c_2) \end{array}$$

The operation of casting into and out of raw types is a kind of "bulletproofing." Bulletproofing maintains the raw-type invariant: the positive cast takes the argument out of $\lceil c\rceil$ and the negative cast returns it there. For example,

$$\langle\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \mapsto \{y{:}\mathsf{Int} \mid \mathsf{pos}\ y\}\rangle^{l,l'}$$

translates to the λ_{H} term

$$\begin{array}{l} \lambda f{:}\lceil\mathsf{Int} \to \mathsf{Int}\rceil. \\ \quad \langle\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \to \{y{:}\mathsf{Int} \mid \mathsf{pos}\ y\} \Rightarrow \lceil\mathsf{Int} \to \mathsf{Int}\rceil\rangle^{l'} \\ \quad\ (\langle\lceil\mathsf{Int} \to \mathsf{Int}\rceil \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \to \{y{:}\mathsf{Int} \mid \mathsf{pos}\ y\}\rangle^{l}\ f). \end{array}$$

Unfolding the domain parts of the casts on f, the domain of the negative cast ensures that f's argument is nonzero with $\langle\lceil\mathsf{Int}\rceil \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\}\rangle^{l'}$; the domain of the positive cast does nothing, since $\langle\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \Rightarrow \lceil\mathsf{Int}\rceil\rangle^{l}$ has no effect. Similarly, the codomain of the negative cast does nothing while the codomain of the positive cast checks that the result is positive. Separating the checks allows λ_{H} to keep track of blame labels, mimicking λ_{C}. Put more generally, in the positive cast, the positive positions may fail because they are "down casts", whereas the negative positions are "up casts", so they cannot fail. The opposite is true of the negative cast. This embodies the idea of contracts as pairs of projections [25]. Note that bulletproofing is "overkill" at base type: for example, $\langle\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\}\rangle^{l,l'}$ translates to

$$\begin{array}{l} \lambda x{:}\lceil\mathsf{Int}\rceil. \\ \quad \langle\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \Rightarrow \lceil\mathsf{Int}\rceil\rangle^{l'} \\ \quad\ (\langle\lceil\mathsf{Int}\rceil \Rightarrow \{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\}\rangle^{l}\ x). \end{array}$$

Only the positive cast does anything—the negative cast into $\lceil \mathsf{Int} \rceil$ always succeeds. This asymmetry is consistent with λ_{C}, where predicate contracts also ignore the negative label. In Section 2.3 I extend the bulletproofing translation to dependent contracts—one of my main contributions.

Both ϕ and ψ preserve behavior in a strong sense: if $\Gamma \vdash t : B$, then either t and $\phi(t)$ both evaluate to the same constant k or they both raise $\Uparrow l$ for the same l; and conversely for ψ. Interestingly, we need to set up this behavioral correspondence *before* we can prove that the translations preserve well-typedness, because of the T_Checking and S_Checking rules.

2.3 The dependent languages

We now extend λ_{C} to dependent function contracts and λ_{H} to dependent functions. Very little needs to be changed in λ_{C}, since contracts and types barely interact; the changes to E_CDecomp and T_FunC are the important ones. Adding dependency to λ_{H} is more involved. In particular, adding contexts to the subtyping judgment entails adding contexts to SSub_Refine. To avoid a dangerous circularity, I define closing substitutions in terms of a separate type semantics. Additionally, the new F_CDecomp rule has a slightly tricky (but necessary) asymmetry, explained below.

2.3.1 Dependent λ_{C}

Dependent λ_{C} has been studied since Findler and Felleisen [26]; it received a very thorough treatment (with an untyped host language) in Blume and McAllester [11], was ported to Haskell by Hinze et al. [41] and Chitil and Huch [16], and was used as a specification language in Xu et al. [80]. Type soundness is not particularly difficult, since types and contracts are kept separate. My formulation follows Findler and Felleisen [26], with a few technical changes to make the proofs for ϕ easier.

We have marked the changed rules with a $\bullet$ next to their names. The new T_RefineC, T_FunC, and E_CDecomp rules in Figure 2.7 (typing) and Figure 2.6 (syntax and semantics) suffice to add dependency to λ_{C}. To help us work with the translations, we also make some small changes to the bindings in contexts, adding a new binding form to track the labels on a contract check throughout the contract well-formedness judgment. Note that T_FunC adds $x{:}{c_1}^{l',l}$ to the context when checking the codomain of a function contract, swapping blame labels. I add a new variable rule, T_VarC, that treats $x{:}c^{l,l'}$ as if it were its skeleton, $x{:}\lfloor c \rfloor$. While unnecessary for λ_{C}'s metatheory, this new binding form helps ϕ preserve types when translating from λ_{H} to picky λ_{C}; see Section 2.6.1.

Two different variants of the E_CDecomp rule can be found in the literature: they are *lax* and *picky*. The original rule in Findler and Felleisen [26] is lax (like most other contract calculi): it does not recheck c_1 when substituting v' into c_2. Blume and McAllester [11] used a picky semantics without observing their departure

Syntax for dependent $\lambda_{\mathbf{C}}$

$$
\begin{array}{rcll}
T & ::= & B \mid T_1 \to T_2 & \text{types} \\
c & ::= & \{x{:}B \mid t\} \mid x{:}c_1 \mapsto c_2 & \text{contracts}\bullet \\
\Gamma & ::= & \emptyset \mid \Gamma, x{:}T \mid \Gamma, x{:}c^{l,l'} & \text{typing contexts}\bullet \\
 & & & \\
t & ::= & x \mid k \mid \lambda x{:}T_1.\ t_2 \mid t_1\ t_2 \mid \Uparrow l \mid \langle c \rangle^{l,l'} \mid \langle \{x{:}B \mid t_1\}, t_2, k \rangle^{l} & \text{terms} \\
v & ::= & k \mid \lambda x{:}T_1.\ t_2 \mid \langle c \rangle^{l,l'} \mid \langle x{:}c_1 \mapsto c_2 \rangle^{l,l'}\ v & \text{values}\bullet \\
r & ::= & v \mid \Uparrow l & \text{results} \\
E & ::= & [\,]\,t \mid v\,[\,] \mid \langle \{x{:}B \mid t\}, [\,], k \rangle^{l} & \text{evaluation contexts}
\end{array}
$$

Operational semantics for dependent $\lambda_{\mathbf{C}}$ $\boxed{t_1 \longrightarrow_c t_2}$

$$
\begin{array}{rcll}
(\lambda x{:}T_1.\ t_2)\ v & \longrightarrow_c & t_2\{x := v\} & \text{E\_Beta} \\
k\ v & \longrightarrow_c & [\![k]\!](v) & \text{E\_Const} \\
\langle \{x{:}B \mid t\} \rangle^{l,l'}\ k & \longrightarrow_c & \langle \{x{:}B \mid t\}, t\{x := k\}, k \rangle^{l} & \text{E\_CCheck} \\
(\langle x{:}c_1 \mapsto c_2 \rangle^{l,l'}\ v)\ v' & \longrightarrow_{lax} & \langle c_2\{x := v'\} \rangle^{l,l'}\ (v\ (\langle c_1 \rangle^{l',l}\ v')) & \\
 & & & \text{E\_CDecompLax}\bullet \\
(\langle x{:}c_1 \mapsto c_2 \rangle^{l,l'}\ v)\ v' & \longrightarrow_{picky} & \langle c_2\{x := \langle c_1 \rangle^{l',l}\ v'\} \rangle^{l,l'}\ (v\ (\langle c_1 \rangle^{l',l}\ v')) & \\
 & & & \text{E\_CDecompPicky}\bullet \\
\langle \{x{:}B \mid t\}, \mathsf{true}, k \rangle^{l} & \longrightarrow_c & k & \text{E\_OK} \\
\langle \{x{:}B \mid t\}, \mathsf{false}, k \rangle^{l} & \longrightarrow_c & \Uparrow l & \text{E\_Fail} \\
E\,[\Uparrow l] & \longrightarrow_c & \Uparrow l & \text{E\_Blame} \\
E\,[t_1] & \longrightarrow_c & E\,[t_2] \quad \text{when } t_1 \longrightarrow_c t_2 & \text{E\_Compat}
\end{array}
$$

Contract erasure

$$\lfloor \{x{:}B \mid t\} \rfloor = B \qquad \lfloor x{:}c_1 \mapsto c_2 \rfloor = \lfloor c_1 \rfloor \to \lfloor c_2 \rfloor$$

Figure 2.6: Syntax and semantics for dependent λ_{C}

Typing rules for dependent $\lambda_{\mathbf{C}}$

$\boxed{\vdash \Gamma}$

$$\frac{}{\vdash \emptyset}\ \text{T\_Empty} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, x{:}T}\ \text{T\_ExtVarT}\bullet \qquad \frac{\vdash \Gamma}{\vdash \Gamma, x{:}T}\ \text{T\_ExtVarT}\bullet$$

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

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T}\ \text{T\_VarT} \qquad \frac{x{:}c^{l,l'} \in \Gamma}{\Gamma \vdash x : \lfloor c \rfloor}\ \text{T\_VarC}\bullet \qquad \frac{}{\Gamma \vdash k : \mathrm{ty_c}(k)}\ \text{T\_Const}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.\ t_2 : T_1 \to T_2}\ \text{T\_Lam} \qquad \frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}\ \text{T\_App}$$

$$\frac{\Gamma \vdash^{l,l'} c : T}{\Gamma \vdash \langle c \rangle^{l,l'} : T \to T}\ \text{T\_Contract}\bullet \qquad \frac{}{\Gamma \vdash \Uparrow l : T}\ \text{T\_Blame}$$

$$\frac{\begin{array}{l}\vdash \Gamma \qquad \emptyset \vdash k : B \qquad \emptyset \vdash t_2 : \mathsf{Bool} \qquad \emptyset \vdash^{l,l'} \{x{:}B \mid t_1\} : B \\ t_2 \longrightarrow_c{}^{*} \mathsf{true} \quad \text{implies} \quad t_1\{x := k\} \longrightarrow_c{}^{*} \mathsf{true}\end{array}}{\Gamma \vdash \langle \{x{:}B \mid t_1\}, t_2, k \rangle^{l} : B}\ \text{T\_Checking}$$

$\boxed{\Gamma \vdash^{l,l'} c : T}$

$$\frac{\Gamma, x{:}B \vdash t : \mathsf{Bool}}{\Gamma \vdash^{l,l'} \{x{:}B \mid t\} : B}\ \text{T\_RefineC} \qquad \frac{\Gamma \vdash^{l',l} c_1 : T_1 \qquad \Gamma, x{:}{c_1}^{l',l} \vdash^{l,l'} c_2 : T_2}{\Gamma \vdash^{l,l'} x{:}c_1 \mapsto c_2 : T_1 \to T_2}\ \text{T\_FunC}$$

Figure 2.7: Typing rules for dependent λ_{C}

from Findler and Felleisen; Hinze et al. [41] choose to be picky as well, substituting $\langle c_1 \rangle^{l',l}\ v'$ into c_2 because it makes their conjunction contract idempotent. We can show (straightforwardly) that both enjoy standard progress and preservation properties. Below, we consider translations to and from both dialects of λ_{C}: picky λ_{C} using only E_CDecompPicky in Sections 2.5.1 and 2.6.2, and lax λ_{C} using only E_CDecompLax in Sections 2.5.2 and 2.6.1. Accordingly, I give two sets of evaluation rules: $\longrightarrow_{lax}$ and $\longrightarrow_{picky}$. When I write $\longrightarrow_c$, the metavariable c ranges over *picky* and *lax*. I complete the type soundness proofs here generically, writing $\longrightarrow_c$ for the evaluation relation. For the translations in Section 2.4, I specify which evaluation relation we use.

I make a standard assumption about constant denotations being well typed: if $\Gamma \vdash k\ v : T$ then $\Gamma \vdash [\![k]\!](v) : T$.

2.3.1 Theorem [Progress]: If $\emptyset \vdash t : T$ then either $t \longrightarrow_c t'$ or $t = r$ (i.e., $t = v$ or $t = \Uparrow l$).

Proof: By induction on the typing derivation. □

For preservation, we prove confluence and substitution lemmas. Note that our substitution lemma must now also cover contracts, since they are no longer closed.

2.3.2 Lemma [Determinacy]: Let $\longrightarrow_c$ be either $\longrightarrow_{picky}$ or $\longrightarrow_{lax}$. If $t \longrightarrow_c t'$ and $t \longrightarrow_c t''$, then $t' = t'$.

2.3.3 Corollary [Cotermination]: If Let $\longrightarrow_c$ be either $\longrightarrow_{picky}$ or $\longrightarrow_{lax}$. $t \longrightarrow_c{}^* r$ and $t \longrightarrow_c{}^* t'$, then $t' \longrightarrow_c{}^* r$.

2.3.4 Lemma [Term and contract substitution]: If $\emptyset \vdash v : T'$, then

1. if $\Gamma, x{:}T', \Gamma' \vdash t : T$ then $\Gamma, \Gamma'\{x := v\} \vdash t\{x := v\} : T$, and
2. if $\Gamma, x{:}T', \Gamma' \vdash^{l,l'} c : T$ then $\Gamma, \Gamma'\{x := v\} \vdash^{l,l'} c\{x := v\} : T$.

Proof: By mutual induction on the typing derivations for t and c. □

We omit the proof for $x{:}c^{l,l'}$ bindings, which is similar.

2.3.5 Theorem [Preservation]: If $\emptyset \vdash t : T$ and $t \longrightarrow_c t'$ then $\emptyset \vdash t' : T$.

Proof: By induction on the typing derivation. This proof is straightforward because typing and contracts hardly interact. □

Syntax for dependent $\lambda_{\mathbf{H}}$

$$
\begin{array}{rcll}
S & ::= & \{x{:}B \mid s_1\} \mid x{:}S_1 \to S_2 & \text{types/contracts}\bullet \\
\Delta & ::= & \emptyset \mid \Delta, x{:}S & \text{typing contexts} \\
\\
s & ::= & x \mid k \mid \lambda x{:}S_1.\ s_2 \mid s_1\ s_2 \mid & \text{terms} \\
 & & \Uparrow l \mid \langle S_1 \Rightarrow S_2 \rangle^l \mid \langle \{x{:}B \mid s_1\}, s_2, k \rangle^l & \\
w & ::= & k \mid \lambda x{:}S_1.\ s_2 \mid \langle S_1 \Rightarrow S_2 \rangle^l \mid \langle x{:}S_{11} \to S_{12} \Rightarrow x{:}S_{21} \to S_{22} \rangle^l\ w & \text{values}\bullet \\
q & ::= & w \mid \Uparrow l & \text{results} \\
F & ::= & [\,]\ s \mid w\ [\,] \mid \langle \{x{:}B \mid s\}, [\,], k \rangle^l & \text{evaluation contexts}
\end{array}
$$

Operational semantics for dependent $\lambda_{\mathbf{H}}$ $\boxed{s_1 \rightsquigarrow_h s_2}$

$$
\begin{array}{rcll}
(\lambda x{:}S_1.\ s_2)\ w_2 & \rightsquigarrow_h & s_2\{x := w_2\} & \text{F\_Beta} \\
k\ w & \rightsquigarrow_h & [\![k]\!](w) & \text{F\_Const} \\
\langle \{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid s_2\} \rangle^l\ k & \rightsquigarrow_h & \langle \{x{:}B \mid s_2\}, s_2\{x := k\}, k \rangle^l & \text{F\_CCheck} \\
(\langle x{:}S_{11} \to S_{12} \Rightarrow x{:}S_{21} \to S_{22} \rangle^l\ w)\ w' & \rightsquigarrow_h & & \text{F\_CDecomp}\bullet \\
\multicolumn{4}{r}{\langle S_{12}\{x := \langle S_{21} \Rightarrow S_{11} \rangle^l\ w'\} \Rightarrow S_{22}\{x := w'\} \rangle^l\ (w\ (\langle S_{21} \Rightarrow S_{11} \rangle^l\ w'))} \\
\langle \{x{:}B \mid s\}, \mathsf{true}, k \rangle^l & \rightsquigarrow_h & k & \text{F\_OK} \\
\langle \{x{:}B \mid s\}, \mathsf{false}, k \rangle^l & \rightsquigarrow_h & \Uparrow l & \text{F\_Fail}
\end{array}
$$

$\boxed{s_1 \longrightarrow_h s_2}$

$$
\frac{s_1 \rightsquigarrow_h s_2}{s_1 \longrightarrow_h s_2}\ \text{F\_Reduce} \qquad \frac{s_1 \longrightarrow_h s_2}{F\,[s_1] \longrightarrow_h F\,[s_2]}\ \text{F\_Compat} \qquad \frac{}{F\,[\Uparrow l] \longrightarrow_h \Uparrow l}\ \text{F\_Blame}
$$

Figure 2.8: Syntax and operational semantics for dependent λ_{H}

2.3.2 Dependent $\lambda_{\mathbf{H}}$

Now we come to the challenging part: dependent λ_{H} and its proof of type soundness. These results require the most complex metatheory in this chapter—in fact, in the entirety of this dissertation—because we need some strong properties about call-by-value evaluation.[4] The full definitions are in Figures 2.8 and 2.9. As before, I have marked the changed rules with a • next to their names.

Now we can enrich the type system with dependent function types, $x{:}S_1 \to S_2$, where x may appear in S_2. The S_Cast rule and the proofs need a notion of type erasure, $\lfloor S \rfloor$; type height $|S|$ will also be used in the proofs.

$$
\begin{array}{rcl@{\qquad}rcl}
\lfloor - \rfloor & : & S \to T & |-| & : & S \to \mathbb{N} \\
\lfloor \{x{:}B \mid s\} \rfloor & = & B & |\{x{:}B \mid s\}| & = & 1 \\
\lfloor x{:}S_1 \to S_2 \rfloor & = & \lfloor S_1 \rfloor \to \lfloor S_2 \rfloor & |x{:}S_1 \to S_2| & = & 1 + |S_1| + |S_2|
\end{array}
$$

[4]The benefit of a CBV semantics is a better treatment of blame; I would avoid the marriage of lazy evaluation and exceptions found in Haskell, as I find it very confusing. By contrast, Knowles and Flanagan [44] cannot treat failed casts as exceptions because that would destroy confluence. They treat them as stuck terms. Readers familiar with the soundness proof of Knowles and Flanagan [44] will notice that my proof is significantly different from theirs. I discuss this in related work, Chapter 5.

A new dependent application rule, S_App, substitutes the argument into the result type of the application. I generalize WF_Refine to allow refinement-type predicates that use variables from the enclosing context. WF_Fun adds the bound variable to the context when checking the codomain of function types. In SSub_Fun, subtyping for dependent function types remains contravariant, but I also add the argument variable to the context with the smaller type. This is similar to the function subtyping rule of $F_{<:}$ [14].

We need to be careful when implementing higher-order dependent casts in the rule F_CDecomp. As the cast decomposes, the variables in the codomain types of such a cast must be replaced. However, this substitution is asymmetric; on one side, we cast the argument and on the other we do not. This behavior is required for type soundness. For suppose we have $\Delta \vdash x{:}S_{11} \to S_{12}$ and $\Delta \vdash x{:}S_{21} \to S_{22}$ with equal skeletons, and values $\Delta \vdash w : (x{:}S_{11} \to S_{12})$ and $\Delta \vdash w' : S_{21}$. Then $\Delta \vdash (\langle x{:}S_{11} \to S_{12} \Rightarrow x{:}S_{21} \to S_{22}\rangle^l\ w)\ w' : S_{22}\{x := w'\}$. When we decompose the cast, we must make *some* substitution into S_{12} and S_{22}, but which? It is clear that we must substitute w' into S_{22}, since the original application has type $S_{22}\{x := w'\}$. Decomposing the cast will produce the inner application $\Delta \vdash w\ (\langle S_{21} \Rightarrow S_{11}\rangle^l\ w') : S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ w'\}$; in order to apply the codomain cast to this term, we must substitute $\langle S_{21} \Rightarrow S_{11}\rangle^l\ w'$ into S_{12}. This calculation determines the form of F_CDecomp.

While the operational semantics changes only in F_CDecomp, we have split the evaluation relation into two parts: reductions $\rightsquigarrow_h$ and steps $\longrightarrow_h$. This is a technical change that allows me to factor the proofs more cleanly (particularly for the parallel reduction proofs).

The final change generalizes SSub_Refine to open terms. We must close these terms before we can compare their behavior, using closing substitutions σ and reading $\Delta \models \sigma$ as "σ satisfies Δ".

Care is needed here to prevent the typing rules from becoming circular: the typing rule S_Sub references the subtyping judgment, the subtyping rule SSub_Refine references the implication judgment, and the single implication rule S_Imp has $\Delta \models \sigma$ in a negative position. This circularity would cause the typing rules to be non-monotonic, and so the existence of a least or greatest fixed-point would not be immediately obvious—the type system would not be well defined! To avoid this circularity, $\Delta \models \sigma$ must not refer back to the other judgments. (The reader may wonder why this was not a problem in λ_C, but notice that in λ_C, implication is only used in T_Checking—which has no (real) context. If we only needed implication in the S_Checking rule, we would not need contexts here, either—we can ensure that active checks only occur at the top-level, with an empty context. But the SSub_Refine subtyping rule refers to S_Imp, and subtyping may be used in arbitrary contexts.)

We can avoid the circularity and ensure that the type system is well defined by building the syntactic rules on top of a denotational semantics for λ_H's types.[5] The

[5]Knowles and Flanagan [44] also introduce a type semantics, but theirs differs from mine in two

Typing rules

$\boxed{\vdash \Delta}$

$$\frac{}{\vdash \emptyset}\ \text{S\_Empty} \qquad \frac{\vdash \Delta \qquad \Delta \vdash S}{\vdash \Delta, x{:}S}\ \text{S\_ExtVar}$$

$\boxed{\Delta \vdash s : S}$

$$\frac{x{:}S \in \Delta}{\Delta \vdash x : S}\ \text{S\_Var} \qquad \frac{\Delta \vdash S_1 \qquad \Delta, x{:}S_1 \vdash s_2 : S_2}{\Delta \vdash \lambda x{:}S_1.\ s_2 : (x{:}S_1 \to S_2)}\ \text{S\_Lam}\bullet$$

$$\frac{}{\Delta \vdash k : \mathrm{ty_h}(k)}\ \text{S\_Const} \qquad \frac{\Delta \vdash s_1 : (x{:}S_1 \to S_2) \qquad \Delta \vdash s_2 : S_1}{\Delta \vdash s_1\ s_2 : S_2\{x := s_2\}}\ \text{S\_App}\bullet$$

$$\frac{\Delta \vdash S_1 \qquad \Delta \vdash S_2 \qquad \lfloor S_1 \rfloor = \lfloor S_2 \rfloor}{\Delta \vdash \langle S_1 \Rightarrow S_2 \rangle^l : S_1 \to S_2}\ \text{S\_Cast} \qquad \frac{\Delta \vdash s : S_1 \qquad \Delta \vdash S_2 \qquad \Delta \vdash S_1 <: S_2}{\Delta \vdash s : S_2}\ \text{S\_Sub}\bullet$$

$$\frac{\begin{array}{l}\vdash \Delta \\ \emptyset \vdash k : \{x{:}B \mid \mathsf{true}\} \qquad \emptyset \vdash s_2 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\} \\ \emptyset \vdash \{x{:}B \mid s_1\} \qquad \emptyset \vdash s_2 \supset s_1\{x := k\}\end{array}}{\Delta \vdash \langle \{x{:}B \mid s_1\}, s_2, k \rangle^l : \{x{:}B \mid s_1\}}\ \text{S\_Checking}\bullet$$

$\boxed{\Delta \vdash S}$

$$\frac{}{\Delta \vdash \{x{:}B \mid \mathsf{true}\}}\ \text{SWF\_Raw} \qquad \frac{\Delta \vdash S_1 \qquad \Delta, x{:}S_1 \vdash S_2}{\Delta \vdash x{:}S_1 \to S_2}\ \text{SWF\_Fun}$$

$$\frac{\Delta, x{:}\{x{:}B \mid \mathsf{true}\} \vdash s : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}}{\Delta \vdash \{x{:}B \mid s\}}\ \text{SWF\_Refine}\bullet$$

$\boxed{\Delta \vdash S_1 <: S_2}$

$$\frac{\Delta, x{:}\{x{:}B \mid \mathsf{true}\} \vdash s_1 \supset s_2}{\Delta \vdash \{x{:}B \mid s_1\} <: \{x{:}B \mid s_2\}}\ \text{SSub\_Refine}\bullet$$

$$\frac{\Delta \vdash S_{21} <: S_{11} \qquad \Delta, x{:}S_{21} \vdash S_{12} <: S_{22}}{\Delta \vdash x{:}S_{11} \to S_{12} <: x{:}S_{21} \to S_{22}}\ \text{SSub\_Fun}\bullet$$

$\boxed{\Delta \vdash s_1 \supset s_2}$

$$\frac{\forall \sigma.\ (\Delta \models \sigma\ \wedge\ \sigma(s_1) \longrightarrow_h^* \mathsf{true}) \quad \text{implies} \quad \sigma(s_2) \longrightarrow_h^* \mathsf{true}}{\Delta \vdash s_1 \supset s_2}\ \text{S\_Imp}\bullet$$

$$\Delta \models \sigma \iff \forall x \in \mathrm{dom}(\Delta).\ \sigma(x) \in [\![\sigma(\Delta(x))]\!]$$

Figure 2.9: Typing rules for dependent λ_{H}

Denotations of types

$$
\begin{array}{rcl}
s \in [\![\{x{:}B \mid s_0\}]\!] & \iff & s \longrightarrow_h^* \Uparrow l \vee (\exists k \in \mathcal{K}_B.\ s \longrightarrow_h^* k \wedge\ s_0\{x := k\} \longrightarrow_h^* \mathsf{true}) \\
s \in [\![x{:}S_1 \to S_2]\!] & \iff & \forall q \in [\![S_1]\!].\ s\ q \in [\![S_2\{x := q\}]\!]
\end{array}
$$

Denotations of kinds

$$
\begin{array}{rcl}
\{x{:}B \mid s\} \in [\![\star]\!] & \iff & \forall k \in \mathcal{K}_B.\ s\{x := k\} \in [\![\{x{:}\mathsf{Bool} \mid \mathsf{true}\}]\!] \\
x{:}S_1 \to S_2 \in [\![\star]\!] & \iff & S_1 \in [\![\star]\!] \wedge \forall q \in [\![S_1]\!].\ S_2\{x := q\} \in [\![\star]\!]
\end{array}
$$

Semantic judgments

$$
\begin{array}{rcl}
\Delta \models S_1 <: S_2 & \iff & \forall \sigma \text{ s.t. } \Delta \models \sigma, [\![\sigma(S_1)]\!] \subseteq [\![\sigma(S_2)]\!] \\
\Delta \models s : S & \iff & \forall \sigma \text{ s.t. } \Delta \models \sigma, \sigma(s) \in [\![\sigma(S)]\!] \\
\Delta \models S & \iff & \forall \sigma \text{ s.t. } \Delta \models \sigma, \sigma(S) \in [\![\star]\!]
\end{array}
$$

Figure 2.10: Type and kind semantics for dependent λ_{H}

idea is that the semantics of a type is a set of closed terms defined independently of the syntactic typing relation, but that turns out to contain all closed well-typed terms of that type. Thus, in the definition of $\Delta \models \sigma$, we quantify over a somewhat larger set than strictly necessary—not just the syntactically well-typed terms of appropriate type (which are all the ones that will ever appear in programs), but all semantically well-typed ones.

The type semantics appears in Figure 2.10. It is defined by induction on type skeletons. For refinement types, terms must either go to blame or produce a constant that satisfies (all instances of) the given predicate. For function types, well typed arguments must yield well-typed results. By construction, these sets include only terminating terms that do not get stuck. In order to show that casts inhabit the denotations of their types, we must also define a denotation of kinds. Since the only kind is $*$, its denotation $[\![\star]\!]$ directly defines semantic well-formedness in terms of the denotations of types.

We must again make the assumption that constants have most-specific types: if $\lfloor \mathrm{ty_h}(k) \rfloor = B$ and $s\{x := k\} \longrightarrow_h^* \mathsf{true}$ then $\emptyset \vdash \mathrm{ty_h}(k) <: \{x{:}B \mid s\}$. We make some other, more standard assumptions, as well. Constants must have closed, well-formed types, and the types assigned must be well-formed. We require that constants are semantically well typed: $k \in [\![\mathrm{ty_h}(k)]\!]$; this requirement is true by the "most-specific type" assumption at base types, but must be assumed at (first-

ways. First, because they cannot treat blame as an exception in their nondeterministic semantics, they must restrict the terms in the semantics to be those that only get stuck at failed casts. They do so by requiring the terms to be well-typed in the simply typed lambda calculus after all casts have been erased. Secondly, their type semantics does not require strong normalization. However, it is not clear whether their language actually admits nontermination—they include a fix constant, but their semantic type soundness proof appears to break down in that case. The problem is not insurmountable: either step indexing their semantics or a proof of unwinding as in Pitts [53] would resolve the issue.

order) functions types. Note that this rules out including fix as a constant, since my type semantics is inhabited only by strongly normalizing terms. We conjecture that expanding the denotation of refinement types to allow for divergence or a step-indexed logical relation [2] would allow us to consider nonterminating terms.

We introduce a few facts about the type semantics before proving semantic type soundness.

2.3.6 Lemma [Determinacy]: If $s \longrightarrow_h s'$ and $s \longrightarrow_h s''$, then $s' = s'$

2.3.7 Corollary [Cotermination]: If $s \longrightarrow_h^* s'$ and $s \longrightarrow_h^* q$, then $s' \longrightarrow_h^* q$.

2.3.8 Lemma [Expansion and contraction of $[\![S]\!]$]: If $s \longrightarrow_h^* s'$, then $s' \in [\![S]\!]$ iff $s \in [\![S]\!]$.

Proof: By induction on $|S|$. □

2.3.9 Lemma [Blame inhabits all types]: For all S, $\Uparrow l \in [\![S]\!]$.

Proof: By induction on $|S|$. □

2.3.10 Corollary [Nonemptiness]: For all S, there exists some q such that $q \in [\![S]\!]$.

The normal forms of $\longrightarrow_h^*$ are of the form $q = w$ or $\Uparrow l$.

2.3.11 Lemma [Strong normalization]: If $s \in [\![S]\!]$, then there exists a q such that $s \longrightarrow_h^* q$—i.e., either $s \longrightarrow_h^* w$ or $s \longrightarrow_h^* \Uparrow l$.

Proof: By induction on $|S|$.

$\underline{S = \{x{:}B \mid s_0\}}$: Suppose $s \in [\![\{x{:}B \mid s_0\}]\!]$. By definition, either $s \longrightarrow_h^* w$ or $s \longrightarrow_h^* \Uparrow l$, so s normalizes.

$\underline{S = x{:}S_1 \to S_2}$: Suppose $s \in [\![x{:}S_1 \to S_2]\!]$. We know that for any $q \in [\![S_1]\!]$ that $s\ q \in [\![S_2\{x := q\}]\!]$. Since $[\![S_1]\!]$ is nonempty (by Lemma 2.3.10), let $q \in [\![S_1]\!]$. By the IH, $s\ q \longrightarrow_h^* w$ or $s\ q \longrightarrow_h^* \Uparrow l$. By the definition of evaluation contexts and $\longrightarrow_h^*$, the function position is evaluated first. If the application reduces to a value (i.e., $s\ q \longrightarrow_h^* w$), then first $s\ q \longrightarrow_h^* w'\ q$, and so $s \longrightarrow_h^* w'$. Alternatively, the application could reduce to blame (i.e., $s\ q \longrightarrow_h^* \Uparrow l$). There are two ways for this to happen: either $s \longrightarrow_h^* \Uparrow l$, or $s \longrightarrow_h^* w'$ and $q \longrightarrow_h^* \Uparrow l$. In both cases, s normalizes. □ □

Unlike the rest of the chapter, I take a top-down approach to the rest of type soundness, to help motivate the steps. So: we are interested in relating the syntactic type system and the type semantics by *semantic type soundness*: if $\emptyset \vdash s : S$, then $s \in [\![S]\!]$. However, to prove this result, we must generalize it. In the bottom of Figure 2.10, I define three *semantic judgements* that correspond to each of the three typing judgments. (Note that the third one requires the definition of a *kind* semantics that picks out well-behaved types—those whose embedded terms belong to the type semantics.) We can then show that the typing judgments imply their semantic counterparts.

2.3.12 Theorem [Semantic type soundness]:

1. If $\Delta \vdash S_1 <: S_2$ then $\Delta \models S_1 <: S_2$.
2. If $\Delta \vdash s : S$ then $\Delta \models s : S$.
3. If $\Delta \vdash S$ then $\Delta \models S$.

Proof: Proof of (1) is in Lemma 2.3.14. Proof of (2) and (3) is in Lemma 2.3.21. □

The first part follows by induction on the subtyping judgment.

2.3.13 Lemma [Trivial refinements of constants]: If $k \in \mathcal{K}_B$, then $k \in [\![\{x{:}B \mid \mathsf{true}\}]\!]$.

2.3.14 Lemma [Semantic subtype soundness]: If $\Delta \vdash S_1 <: S_2$ then $\Delta \models S_1 <: S_2$.

Proof: By induction on the subtyping derivation.

SSub_Refine: We know $\Delta \vdash \{x{:}B \mid s_1\} <: \{x{:}B \mid s_2\}$, and must show the corresponding semantic subtyping. Inversion of this derivation gives us $\Delta, x{:}\{x{:}B \mid \mathsf{true}\} \vdash s_1 \supset s_2$, which means:

$$\forall\sigma.\ ((\Delta, x{:}\{x{:}B \mid \mathsf{true}\} \models \sigma \wedge \sigma(s_1) \longrightarrow_h^* \mathsf{true}) \quad \text{implies} \quad \sigma(s_2) \longrightarrow_h^* \mathsf{true}) \qquad (*)$$

We must show $\Delta \models \{x{:}B \mid s_1\} <: \{x{:}B \mid s_2\}$, i.e., that $\forall\sigma.\ \Delta \models \sigma$ implies $[\![\{x{:}B \mid s_1\}]\!] \subseteq [\![\{x{:}B \mid s_2\}]\!]$. Let σ be given such that $\Delta \models \sigma$. Suppose $s \in [\![\sigma(\{x{:}B \mid s_1\})]\!]$. By definition, either s goes to $\Uparrow l$, or it goes to $k \in \mathcal{K}_B$ such that $s_1\{x := k\} \longrightarrow_h^* \mathsf{true}$. In the former case, $\Uparrow l \in [\![\{x{:}B \mid s_2\}]\!]$ by definition. So consider the latter case, where $s \longrightarrow_h^* k$.

We already know that $k \in \mathcal{K}_B$, so it remains to see that: $\sigma(s_2)\{x := k\} \longrightarrow_h^* \mathsf{true}$. We know by assumption that $\sigma(s_1)\{x := k\} \longrightarrow_h^* \mathsf{true}$. By Lemma 2.3.13, $k \in [\![\{x{:}B \mid \mathsf{true}\}]\!]$.

Now observe that $\Delta, x{:}\{x{:}B \mid \mathsf{true}\} \models \sigma\{x := k\}$. Since $\sigma'(s_1) \longrightarrow_h^* \mathsf{true}$, we can conclude that $\sigma'(s_2) \longrightarrow_h^* \mathsf{true}$ by our assumption (*). This completes this case.

SSub_Fun: $\Delta \vdash (x{:}S_{11} \to S_{12}) <: (x{:}S_{21} \to S_{22})$; by the IH, we have $\Delta \models S_{21} <: S_{11}$ and $\Delta, x{:}S_{21} \models S_{12} <: S_{22}$. We must show that $\Delta \models (x{:}S_{11} \to S_{12}) <: (x{:}S_{21} \to S_{22})$.

Let $\Delta \models \sigma$ and $s \in [\![\sigma(x{:}S_{11} \to S_{12})]\!]$, for some σ. We must show, for all q, that if $q \in [\![\sigma(S_{21})]\!]$ then $s\ q \in [\![\sigma(S_{22})\{x := q\}]\!]$.

Let $q \in [\![\sigma(S_{21})]\!]$. Then $q \in [\![\sigma(S_{11})]\!]$. Since $s \in [\![\sigma(x{:}S_{11} \to S_{12})]\!]$, we know that $s\ q \in [\![\sigma(S_{12})\{x := q\}]\!]$. Finally, since $\Delta, x{:}S_{21} \models S_{12} <: S_{22}$ and $\Delta, x{:}S_{21} \models \sigma\{x := q\}$, we can conclude that $s\ q \in [\![\sigma(S_{22})\{x := q\}]\!]$, and so $s \in [\![\sigma(x{:}S_{21} \to S_{22})]\!]$. □

The proof semantic subtype soundess goes through easily, the first of the three parts of semantic soundness (Theorem 2.3.12). We run into some complications with semantic type and kind soundness, the second and third parts (which must be proven together). The crux of the difficulty lies with the S_App rule. Suppose the application $s_1\ s_2$ was well typed and $s_1 \in [\![x{:}S_1 \to S_2]\!]$ and $s_2 \in [\![S_1]\!]$. According to S_App, the application's type is $S_2\{x := s_2\}$. By the type semantics defined in Figure 2.10, if $s_1 \in [\![x{:}S_1 \to S_2]\!]$, then $s_1\ q \in [\![S_2\{x := q\}]\!]$ for any $q \in [\![S_1]\!]$. Sadly, s_2 is not necessarily a result! We do know, however, that $s_2 \in [\![S_1]\!]$, so $s_2 \longrightarrow_h^* q_2$ by strong normalization (Lemma 2.3.11). We need to ask, then, how the type semantics of $S_2\{x := s_2\}$ and $S_2\{x := q_2\}$ relate. (One might think that we can solve this by changing the type semantics to quantify over terms, not results. But this just pushes the problem to the S_Lam case.)

We can show that the two type semantics are in fact equal using a parallel reduction technique. I define a parallel reduction relation $\Rrightarrow$ on terms and types in Figure 2.11 that allows redexes in different parts of a term (or type) to be reduced in the same step, and then I prove that types that parallel-reduce to each other—like $S_2\{x := s_2\}$ and $S_2\{x := q_2\}$—have the same semantics. The definition of parallel reduction is standard, though we need to be careful to make it respect call-by-value reduction order: the β-redex $(\lambda x{:}S_1.\ s_1)\ s_2$ should not be contracted unless s_2 is a value, since doing so can change the order of effects. (Other redices within s_1 and s_2 can safely reduce.)[6] The proof requires a longish sequence of technical lemmas that essentially show that $\Rrightarrow$ commutes with $\longrightarrow_h^*$. Since the proofs require fussy symbol manipulation, we have done these proofs in Coq. My development is available at `http://www.cis.upenn.edu/~mgree/papers/lambdah_parred.tgz`. We restate the critical results here.

Lemma [Substitution of parallel-reducing terms, Lemma `A3` in `thy.v`]:
If $w \Rrightarrow w'$, then

1. if $s \Rrightarrow s'$ then $s\{x := w\} \Rrightarrow s'\{x := w'\}$, and
2. if $S \Rrightarrow S'$ then $S\{x := w\} \Rrightarrow S'\{x := w'\}$.

Lemma [Parallel reduction implies coevaulation, Lemma `A20` in `thy.v`]:
If $s_1 \Rrightarrow s_2$ then $s_1 \longrightarrow_h^* k$ iff $s_2 \longrightarrow_h^* k$. Similarly, $s_1 \longrightarrow_h^* \Uparrow l$ iff $s_2 \longrightarrow_h^* \Uparrow l$.

[6] I conjecture that the reflexive transitive closure of a similar "CBV-respecting" variant of full β-reduction could be used in place of parallel reduction. It is not clear whether it would lead to shorter proofs.

An alternative strategy would be to use $\Rrightarrow$ in the typing rules and $\longrightarrow_h$ in the operational semantics. This would simplify some of the metatheory, but it would complicate the specification of the language. Using $\longrightarrow_h$ in the typing rules gives a clearer intuition and keeps the core system small.

2.3.17 Lemma [Single parallel reduction preserves type semantics]: If $S_1 \Rrightarrow S_2$ then $[\![S_1]\!] = [\![S_2]\!]$.

Proof: By induction on $|S_1|$ (which is equal to $|S_2|$), with a case analysis on the final rule used to show $S_1 \Rrightarrow S_2$. □

2.3.18 Corollary [Parallel reduction preserves type semantics]: If $S_1 \Rrightarrow^* S_2$ then $[\![S_1]\!] = [\![S_2]\!]$.

2.3.19 Lemma [Partial semantic substitution]: If $\Delta_1, x{:}S', \Delta_2 \models s : S$, and $\Delta_1, x{:}S', \Delta_2 \models S$, and $\Delta_1 \models s' : S'$ then $\Delta_1, \Delta_2\{x := s'\} \models s\{x := s'\} : S\{x := s'\}$ and $\Delta_1, \Delta_2\{x := s'\} \models S\{x := s'\}$.

Proof: By the definition of $\Delta \models \sigma$. □

The semantic typing case for casts requires a separate induction.

2.3.20 Lemma [Semantic typing for casts]: If $\Delta \models S_1$ and $\Delta \models S_2$ and $\lfloor S_1 \rfloor = \lfloor S_2 \rfloor$, then $\Delta \models \langle S_1 \Rightarrow S_2 \rangle^l : x{:}S_1 \to S_2$ for fresh x.

Proof: By induction on $|S_1| = |S_2|$, going by cases on the shape of S_2. Let $\Delta \models \sigma$; we show that $\sigma(\langle S_1 \Rightarrow S_2 \rangle^l) \in [\![\sigma(S_1 \to S_2)]\!]$.

$\underline{S_2 = \{x{:}B \mid \sigma(s_2)\}}$: Let $q \in [\![\sigma(S_1)]\!]$. If $q = \Uparrow l'$, then the applied cast goes to $\Uparrow l'$, and we are done by Lemma 2.3.9. So $q = k \in \mathcal{K}_B$. By F_CCheck $\langle S_1 \Rightarrow \{x{:}B \mid \sigma(s_2)\}\rangle^l\ k \longrightarrow_h \langle \{x{:}B \mid \sigma(s_2)\}, \sigma(s_2)\{x := k\}, k\rangle^l$. By the well-kinding of S_2, we know that $\sigma(s_2)\{x := k\} \in [\![\{x{:}\mathsf{Bool} \mid \mathsf{true}\}]\!]$, so by strong normalization (Lemma 2.3.11), the predicate in the active check goes to blame or to a value. If it goes to blame, we are done. If it goes to a value, that value must be true or false. If it goes to false, then the whole term goes to blame and we are done. If it goes to true, then the check will step to k. But $\sigma(s_2)\{x := k\} \longrightarrow_h^* \mathsf{true}$, so $k \in [\![\sigma(\{x{:}B \mid s_2\})]\!]$ by definition. Expansion (Lemma 2.3.8) completes the proof.

$\underline{S_2 = x{:}S_{21} \to S_{22}}$: We must have $S_1 = x{:}S_{11} \to S_{12}$. Let $q \in [\![\sigma(S_1)]\!]$; if it is blame we are done by Lemma 2.3.9, so let it be a value w. Let $q' \in [\![\sigma(S_{21})]\!]$; if it is blame we are done, so let it be a value w'. By F_CDecomp:

$$\langle \sigma(S_{12})\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ w'\} \Rightarrow \sigma(S_{22})\{x := w'\}\rangle^l\ (w\ (\langle \sigma(S_{21}) \Rightarrow \sigma(S_{11})\rangle^l\ w'))$$

By the IH, $\langle \sigma(S_{21}) \Rightarrow \sigma(S_{11})\rangle^l$ is semantically well-typed, so we have $\langle \sigma(S_{21}) \Rightarrow \sigma(S_{11})\rangle^l\ w' \in [\![\sigma(S_{11})]\!]$. By strong normalization (Lemma 2.3.11), this term reduces (and therefore parallel reduces, by Lemma A4) to some q''.

$$\boxed{s_1 \Rrightarrow s_2}$$

$$\frac{}{s \Rrightarrow s}\ \text{FP\_Refl} \qquad \frac{w \Rrightarrow w'}{k\ w \Rrightarrow [\![k]\!](w')}\ \text{FP\_RConst}$$

$$\frac{s_{12} \Rrightarrow s'_{12} \qquad w_2 \Rrightarrow w'_2}{(\lambda x{:}S.\ s_{12})\ w_2 \Rrightarrow s'_{12}\{x := w'_2\}}\ \text{FP\_RBeta}$$

$$\frac{s_2 \Rrightarrow s'_2}{\langle \{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid s_2\} \rangle^l\ k \Rrightarrow \langle \{x{:}B \mid s'_2\}, s'_2\{x := k\}, k \rangle^l}\ \text{FP\_RCCheck}$$

$$\frac{}{\langle \{x{:}B \mid s\}, \mathsf{true}, k \rangle^l \Rrightarrow k}\ \text{FP\_ROK} \qquad \frac{}{\langle \{x{:}B \mid s\}, \mathsf{false}, k \rangle^l \Rrightarrow \Uparrow l}\ \text{FP\_RFail}$$

$$\frac{S_{11} \Rrightarrow S'_{11} \quad S_{12} \Rrightarrow S'_{12} \quad S_{21} \Rrightarrow S'_{21} \quad S_{22} \Rrightarrow S'_{22} \quad w_1 \Rrightarrow w'_1 \quad w_2 \Rrightarrow w'_2}{\begin{array}{l}(\langle x{:}S_{11} \to S_{12} \Rightarrow x{:}S_{21} \to S_{22} \rangle^l\ w_1)\ w_2 \Rrightarrow \\ \langle S'_{12}\{x := \langle S'_{21} \Rightarrow S'_{11} \rangle^l\ w'_2\} \Rightarrow S'_{22}\{x := w'_2\} \rangle^l\ (w'_1\ (\langle S'_{21} \Rightarrow S'_{11} \rangle^l\ w'_2))\end{array}}\ \text{FP\_RCDecomp}$$

$$\frac{S_1 \Rrightarrow S'_1 \qquad s_{12} \Rrightarrow s'_{12}}{\lambda x{:}S_1.\ s_{12} \Rrightarrow \lambda x{:}S'_1.\ s'_{12}}\ \text{FP\_Lam} \qquad \frac{s_1 \Rrightarrow s'_1 \qquad s_2 \Rrightarrow s'_2}{s_1\ s_2 \Rrightarrow s'_1\ s'_2}\ \text{FP\_App}$$

$$\frac{S_1 \Rrightarrow S'_1 \qquad S_2 \Rrightarrow S'_2}{\langle S_1 \Rightarrow S_2 \rangle^l \Rrightarrow \langle S'_1 \Rightarrow S'_2 \rangle^l}\ \text{FP\_Cast}$$

$$\frac{S \Rrightarrow S' \qquad s \Rrightarrow s'}{\langle S, s, k \rangle^l \Rrightarrow \langle S', s', k \rangle^l}\ \text{FP\_Check} \qquad \frac{}{F\ [\Uparrow l] \Rrightarrow \Uparrow l}\ \text{FP\_Blame}$$

$$\boxed{S_1 \Rrightarrow S_2}$$

$$\frac{}{S \Rrightarrow S}\ \text{FP\_SRefl} \qquad \frac{s \Rrightarrow s'}{\{x{:}B \mid s\} \Rrightarrow \{x{:}B \mid s'\}}\ \text{FP\_SRefine}$$

$$\frac{S_1 \Rrightarrow S'_1 \qquad S_2 \Rrightarrow S'_2}{x{:}S_1 \to S_2 \Rrightarrow x{:}S'_1 \to S'_2}\ \text{FP\_SFun}$$

Figure 2.11: Parallel reduction for dependent λ_{H}

We know that $w\ q'' \in [\![\sigma(S_{12})\{x := q''\}]\!]$ by assumption. Using parallel reduction (Corollary 2.3.18), we have $[\![\sigma(S_{12})\{x := q''\}]\!] = [\![\sigma(S_{12})\{x := \langle\sigma(S_{21}) \Rightarrow \sigma(S_{11})\rangle^l\ w'\}]\!]$.

Before applying the IH, we note that $\Delta \models S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ w'\}$ and $\Delta \models S_{22}\{x := w'\}$ by Lemma 2.3.19. Then, by the IH we see that $\langle\sigma(S_{12})\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ w'\} \Rightarrow \sigma(S_{22})\{x := w'\}\rangle^l$ is semantically well-kinded, so

$$\langle\sigma(S_{12})\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ w'\} \Rightarrow \sigma(S_{22})\{x := w'\}\rangle^l\ (w\ w'') \in [\![\sigma(S_{22})\{x := w'\}]\!] \quad \square$$

$\square$

2.3.21 Lemma [Semantic type soundness]:

1. If $\Delta \vdash s : S$ then $\Delta \models s : S$.
2. If $\Delta \vdash S$ then $\Delta \models S$.

Proof: By induction on the typing and well-formedness derivations, using Corollary 2.3.18 in the S_App case and Lemma 2.3.14 in the S_Sub case. $\square$

Theorem 2.3.12 gives us type soundness, and it combines with Lemma 2.3.11 for an even stronger result: well-typed programs always evaluate to values of appropriate (semantic) type.

While one can prove progress and preservation theorems, I omit them: we already have type soundness. Later proofs will require standard weakening and substitution lemmas, though, so we can prove them now.

2.3.22 Lemma [Weakening]: If $\Delta \vdash s : S$ and $\Delta \vdash S$, and $\mathrm{dom}(\Delta) \cap \mathrm{dom}(\Delta') = \emptyset$ with $\vdash \Delta, \Delta'$, then $\Delta, \Delta' \vdash s : S$ and $\Delta, \Delta' \vdash S$.

Proof: By straightforward induction on s and $|S|$; we reuse the (critical) context well-formedness derivation in the S_Checking case. $\square$

The substitution lemma has one complication: the operational judgment S_Imp requires the semantic type soundness theorem to show that a syntactically well-typed term can be used in a closing substitution. It is otherwise straightforward.

2.3.23 Lemma [Substitution (implication)]: If $\Delta_1, x{:}S, \Delta_2 \vdash s_1 \supset s_2$ and $\Delta_1 \vdash s : S$, then $\Delta_1, \Delta_2\{x := s\} \vdash s_1\{x := s\} \supset s_2\{x := s\}$.

Proof: Direct, unfolding the closing substitutions. $\square$

2.3.24 Lemma [Substitution (subtyping)]: If $\Delta_1, x{:}S, \Delta_2 \vdash S_1 <: S_2$ and $\Delta_1 \vdash s : S$, then $\Delta_1, \Delta_2\{x := s\} \vdash S_1\{x := s\} <: S_2\{x := s\}$.

Proof: By induction on the subtyping derivation. $\square$

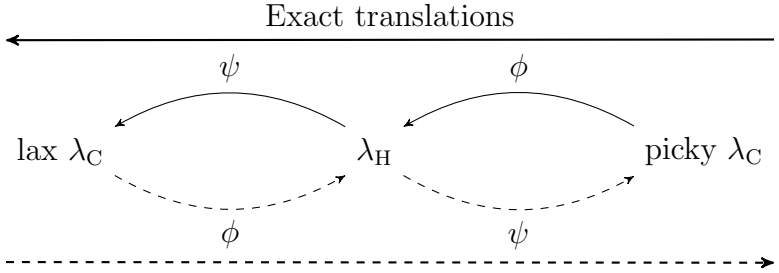
2.3.25 Lemma [Substitution (typing and well-formedness)]: If $\Delta_1 \vdash s : S$ then

1. if $\Delta_1, x{:}S, \Delta_2 \vdash s_1 : S_1$ then $\Delta_1, \Delta_2\{x := s\} \vdash s_1\{x := s\} : S_1\{x := s\}$,
2. if $\Delta_1, x{:}S, \Delta_2 \vdash S_1$ then $\Delta_1, \Delta_2\{x := s\} \vdash S_1\{x := s\}$, and
3. if $\vdash \Delta_1, x{:}S, \Delta_2$ then $\vdash \Delta_1, \Delta_2\{x := s\}$.

Proof: By mutual induction on the typing derivations. □

2.4 The translations

I divide treatment of the translations—ϕ from λ_{H} terms to λ_{C} terms and ψ from λ_{C} terms to λ_{H} terms—in two parts. I define the translations below. The axis of blame captures whether or not the translations exactly preserve semantics. The translations are *exact* when moving left on the axis of blame, but they are *inexact*—sometimes yielding more blame—moving right on the axis of blame.



Inexact translations, more blame in target language

I treat the exact translations in Section 2.5; the inexact translations are covered in Section 2.6.

Each translation proof follows the same basic schema. First, I define a logical relation between the two languages. Then we use the logical relation to prove a lemma relating the translation, contracts, and casts. Finally, we prove that the translation preserves evaluation behavior—that is, terms are logically related to their translations—and typing. All of the proofs make extensive use of expansion and contraction of evaluation and "cotermination" arguments. Every proof uses its own contract/cast logical relation. The proofs for the inexact translations of Section 2.6 demand custom term logical relations, too. I used σ to range over closing substitutions in λ_{H}; I use δ to range over dual closing substitutions in the logical relations.

Contexts $\boxed{\phi : (\vdash \Gamma) \to \Delta}$

$$
\begin{array}{rcl}
\phi(\vdash \emptyset) & = & \emptyset \\
\phi(\vdash \Gamma, x{:}T) & = & \phi(\vdash \Gamma), x{:}\lceil T \rceil \\
\phi(\vdash \Gamma, x{:}c^{l,l'}) & = & \phi(\vdash \Gamma), x{:}\phi(\Gamma \vdash^{l,l'} c : \lfloor c \rfloor)
\end{array}
$$

Terms $\boxed{\phi : (\Gamma \vdash t : T) \to s}$

$$
\begin{array}{rcl}
\phi(\Gamma_1, x{:}T, \Gamma_2 \vdash x : T) & = & x \\
\phi(\Gamma_1, x{:}c^{l,l'}, \Gamma_2 \vdash x : \lfloor c \rfloor) & = & \langle \phi(\Gamma_1 \vdash^{l,l'} c : \lfloor c \rfloor) \Rightarrow \lceil c \rceil \rangle^{l'}\ x \\
\phi(\Gamma \vdash k : T) & = & k \\
\phi(\Gamma \vdash \lambda x{:}T_1.\ t_2 : T_1 \to T_2) & = & \lambda x{:}\lceil T_1 \rceil.\ \phi(\Gamma, x{:}T_1 \vdash t_2 : T_2) \\
\phi(\Gamma \vdash t_1\ t_2 : T_2) & = & \phi(\Gamma \vdash t_1 : T_1 \to T_2)\ \phi(\Gamma \vdash t_2 : T_1) \\
\phi(\Gamma \vdash \Uparrow l : T) & = & \Uparrow l \\
\phi(\emptyset \vdash \langle c, t, k \rangle^l : B) & = & \langle \phi(\emptyset \vdash^{l,l'} c : B), \phi(\emptyset \vdash t : \mathsf{Bool}), k \rangle^l \\
\phi(\Gamma \vdash \langle c \rangle^{l,l'} : T) & = & \lambda x{:}\lceil c \rceil.\ \langle \phi(\Gamma \vdash^{l',l} c : T) \Rightarrow \lceil c \rceil \rangle^{l'} \\
 & & \quad (\langle \lceil c \rceil \Rightarrow \phi(\Gamma \vdash^{l,l'} c : T) \rangle^{l}\ x) \\
 & & \qquad \text{where } x \text{ is fresh}
\end{array}
$$

Types $\boxed{\phi : (\Gamma \vdash^{l,l'} c : T) \to S}$

$$
\begin{array}{rcl}
\phi(\Gamma \vdash^{l,l'} \{x{:}B \mid t\} : B) & = & \{x{:}B \mid \phi(\Gamma, x{:}B \vdash t : \mathsf{Bool})\} \\
\phi(\Gamma \vdash^{l,l'} x{:}c_1 \mapsto c_2 : T_1 \to T_2) & = & x{:}\phi(\Gamma \vdash^{l',l} c_1 : T_1) \to \phi(\Gamma, x{:}{c_1}^{l',l} \vdash^{l,l'} c_2 : T_2)
\end{array}
$$

Figure 2.12: The translation ϕ from dependent λ_{C} to dependent λ_{H}

2.4.1 Translating λ_{C} to λ_{H}: ϕ

We define the full ϕ for the dependent calculi in Figure 2.12. In the dependent case, we need to translate *derivations* of well-formedness and well-typing of λ_{C} contexts, terms, and contracts into λ_{H} contexts, terms, and types. I translate derivations to ensure type preservation, translating T_VarT and T_VarC derivations differently: I leave variables of simple type alone, but I cast variables bound to contracts.

To see why we need this distinction, consider the function contract $f{:}(x{:}\{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\} \mapsto \{y{:}\mathsf{Int} \mid \mathsf{true}\}) \mapsto \{z{:}\mathsf{Int} \mid f\ 0 = 0\}$. Note that this contract is well-formed in λ_{C}, but that the codomain "abuses" the bound variable. A naive translation will *not* be well-typed in λ_{H}. The term $f\ 0$ will not be typeable when f has type $x{:}\{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\} \to \lceil \mathsf{Int} \rceil$, since f only accepts positive arguments. The problem is that WF_Fun can add a (possibly refined) type to the context when checking the codomain, so we need to restore the "variables have raw types" invariant—something we can't always rely on subtyping to do, since types are not in general subtypes of their raw type. By tracking which variables were bound by contracts in λ_{C}, we can be sure to cast them to raw types when they're referenced. We therefore translate the contract above to $f{:}S \to \{z{:}\mathsf{Int} \mid (\langle S \Rightarrow \lceil \mathsf{Int} \to \mathsf{Int} \rceil \rangle^{l'}\ f)\ 0 = 0\}$, where $S = x{:}\{x{:}\mathsf{Int} \mid \mathsf{pos}\ x\} \to \lceil \mathsf{Int} \rceil$. This (partially) motivates the $x{:}c^{l,l'}$ binding form in dependent λ_{C}.

Bulletproofing uses raw types, defined here for the dependent system.

$$\begin{array}{rclcrcl} \lceil \{x{:}B \mid s\} \rceil & = & \{x{:}B \mid \mathsf{true}\} & & \lceil x{:}S_1 \to S_2 \rceil & = & \lceil S_1 \rceil \to \lceil S_2 \rceil \\ \lceil B \rceil & = & \{x{:}B \mid \mathsf{true}\} & & \lceil T_1 \to T_2 \rceil & = & \lceil T_1 \rceil \to \lceil T_2 \rceil \\ \lceil \{x{:}B \mid t\} \rceil & = & \{x{:}B \mid \mathsf{true}\} & & \lceil x{:}c_1 \mapsto c_2 \rceil & = & \lceil c_1 \rceil \to \lceil c_2 \rceil \end{array}$$

Note that dependency is eliminated.

We could write the translation on terms instead of derivations, defining

$$\phi(x{:}c_1 \mapsto c_2) = x{:}\phi(c_1) \to \phi(c_2)\{x := \langle \phi(c_1) \Rightarrow \lceil c_1 \rceil \rangle^l \; x\}$$

but the proofs are easier if we translate derivations.

Constants translate to themselves. One technical point: to maintain the raw type invariant, we need λ_{H}'s higher-order constants to have typings that can be seen as raw by the subtyping relation, i.e., $\Delta \vdash \mathrm{ty_h}(k) <: \lceil \mathrm{ty_c}(k) \rceil$. This can be proven at base types (since we have already assumed that $\mathrm{ty_h}(k)$ is the "most specific type" for each k), but must be assumed for first-order constant functions. This slightly restricts the types we might assign to constants, e.g., we cannot say $\mathrm{ty_h}(\mathsf{sqrt}) = x{:}\{x{:}\mathsf{Float} \mid x >= 0\} \to \{y{:}\mathsf{Float} \mid (y * y) = x\}$, since it is not the case that $\Delta \vdash \mathrm{ty_h}(\mathsf{sqrt}) <: \lceil \mathsf{Float} \to \mathsf{Float} \rceil$. Since its domain cannot be refined, $[\![\mathsf{sqrt}]\!]$ must be defined for all $k \in \mathcal{K}_{\mathsf{Float}}$, e.g., $[\![\mathsf{sqrt}]\!](-1)$ must be defined. We have already required that denotations be total over their simple types in λ_{C}, and λ_{H} uses the same denotation function $[\![-]\!]$, so this requirement does not seem too severe. In any case, we can define it to be equal to $\Uparrow l_0$, for some l_0. We could instead translate k to $\langle \mathrm{ty_h}(k) \Rightarrow \lceil \mathrm{ty_h}(k) \rceil \rangle^{l_0} \; k$; however, in this case the nondependent fragments of the languages would no longer correspond exactly.

When translating a term with ϕ, the image behaves the same as the pre-image when we interpret the pre-image using picky λ_{C} semantics (proved in Section 2.5.1). When the pre-image is interpreted in the lax λ_{C} semantics, however, we find that the image can raise blame when the pre-image returns a value (proved in Section 2.6.1).

2.4.2 Translating λ_{H} to λ_{C}: ψ

In this section, I formally define ψ for the dependent versions of lax λ_{C} and λ_{H}. I prove that ψ is type preserving and induces behavioral correspondence.

The full definition of ψ is in Figure 2.13. Most terms are translated homomorphically. In abstractions, the annotation is translated by erasing the refined λ_{H} type to its skeleton. As we mentioned in Section 2.2, the trickiest part is the translation of casts between function types: when generating the codomain contract from a cast between two function types, we perform the same asymmetric substitution as F_CDecomp. Since ψ inserts new casts, we need to pick a blame label: $\psi(\langle S_1 \Rightarrow S_2 \rangle^l)$ passes l as an index to $\psi^l(S_1, S_2)$.

I show that ψ preserves semantics exactly when its image is interpreted as lax λ_{C} in Section 2.5.2; I show that the picky λ_{C} interpretation of a term's ψ-image may blame more often than the original term in Secton 2.6.2.

Term translation $\boxed{\psi : s \to t}$

$$\begin{array}{rcl@{\qquad}rcl}
\psi(x) & = & x & \psi(k) & = & k \\
\psi(\lambda x{:}S.\ s) & = & \lambda x{:}\lfloor S \rfloor.\ \psi(s) & \psi(s_1\ s_2) & = & \psi(s_1)\ \psi(s_2) \\
\psi(\langle S_1 \Rightarrow S_2 \rangle^l) & = & \langle \psi^l(S_1, S_2) \rangle^{l,l} & \psi(\Uparrow l) & = & \Uparrow l \\
\psi(\langle \{x{:}B \mid s_1\}, s_2, k \rangle^l) & = & \langle \{x{:}B \mid \psi(s_1)\}, \psi(s_2), k \rangle^l & & &
\end{array}$$

Cast translation $\boxed{\psi : S \times S \times l \to T}$

$$\begin{array}{rcl}
\psi^l(\{x{:}B \mid s_1\}, \{x{:}B \mid s_2\}) & = & \{x{:}B \mid \psi(s_2)\} \\
\psi^l(x{:}S_{11} \to S_{12}, x{:}S_{21} \to S_{22}) & = & x{:}\psi^l(S_{21}, S_{11}) \mapsto \psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11} \rangle^l\ x\}, S_{22})
\end{array}$$

Figure 2.13: ψ mapping dependent λ_{H} to dependent λ_{C}

Result correspondence $r \approx q : T$

$$k \approx k : B \iff k \in \mathcal{K}_B$$

$$v \approx w : T_1 \to T_2 \iff \forall t \sim s : T_1.\ v\ t \sim w\ s : T_2$$

$$\Uparrow l \approx \Uparrow l : T$$

Term correspondence $\boxed{t \sim s : T}$

$$t \sim s : T \iff t \longrightarrow_c{}^* r \wedge s \longrightarrow_h^* q \wedge r \approx q : T$$

Figure 2.14: A blame-exact result/term correspondence

2.5 Exact translations

Translations moving left on the axis of blame—from picky λ_{C} to λ_{H}, and from λ_{H} to lax λ_{C}—are exact. That is, we can show a tight behavioral correspondence between terms and their translations (see Figure 2.14). I read $t \sim s : T$ as "t corresponds with s at type T".

The correspondence is a standard logical relation, defined in two intertwined parts: a relation on results, $r \approx q : T$ and its closure with respect to evaluation, $t \sim s : T$. The term correspondence is defined directly: terms correspond when they reduce to corresponding results. We write $\longrightarrow_c$ in this single definition: in Section 2.5.1 I use this definition taking $\longrightarrow_c$ to be $\longrightarrow_{picky}$; in Section 2.5.2 I use this definition taking $\longrightarrow_c$ to be $\longrightarrow_{lax}$. The result correspondence is defined inductively over λ_{C}'s simple types. Blame corresponds to itself at any type. At B, constants in $\mathcal{K}_B$ correspond to themselves; results at $T_1 \to T_2$ correspond when they applying them to corresponding *terms* yields corresponding *terms*. Stratifying the definition this way simplifies some of the proofs later. We call this correspondence *exact* because terms corresponding

Contract/type correspondence $\boxed{c \sim^{l,l'} S : T}$

$$\{x{:}B \mid t\} \sim^{l,l'} \{x{:}B \mid s\} : B \iff \forall k \in \mathcal{K}_B.\ t\{x := k\} \sim s\{x := k\} : \mathsf{Bool}$$

$$x{:}c_1 \mapsto c_2 \sim^{l,l'} x{:}S_1 \to S_2 : T_1 \to T_2 \iff c_1 \sim^{l',l} S_1 : T_1 \wedge \forall t \sim s : T_1.c_2\{x := \langle c_1\rangle^{l',l}\ t\} \sim^{l,l'} S_2\{x := \langle \lceil S_1 \rceil \Rightarrow S_1\rangle^{l'}\ s\} : T_2$$

Dual closing substitutions

$$\Gamma \models \delta \iff \begin{cases} \forall x{:}T \in \Gamma.\ \delta_1(x) \sim \delta_2(x) : T \\ \forall x{:}c^{l,l'} \in \Gamma.\ \delta_1(x) = \langle \delta_1(c)\rangle^{l,l'}\ t \wedge \delta_2(x) = \langle \lceil c \rceil \Rightarrow \delta_2(S)\rangle^{l}\ s \\ \text{where } S = \phi(\Gamma \vdash^{l,l'} c : \lfloor c \rfloor) \wedge t \sim s : \lfloor c \rfloor \end{cases}$$

Lifted to open terms

$$\begin{array}{rcl} \Gamma \vdash t \sim s : T & \iff & \forall \delta.\ (\Gamma \models \delta \ \text{ implies } \ \delta_1(t) \sim \delta_2(s) : T) \\ \Gamma \vdash c \sim^{l,l'} S : T & \iff & \forall \delta.\ (\Gamma \models \delta \ \text{ implies } \ \delta_1(c) \sim^{l,l'} \delta_2(S) : T) \end{array}$$

Figure 2.15: Blame-exact correspondence for ϕ from picky λ_{C}

at base type yield identical results.

Note that I define the correspondence here on closed (or harmlessly open) terms. In the following two sections, I define translation specific extensions of the correspondence to open terms and contracts.

2.5.1 Translating picky $\lambda_{\mathbf{C}}$ to $\lambda_{\mathbf{H}}$

We extend the term correspondence of Figure 2.14 to contracts and types, lifting the correspondences to open terms using dual closing substitutions. Recall that we interpret the term correspondence as using $\longrightarrow_{picky}$. For a binding $x{:}c^{l,l'} \in \Gamma$, we use ϕ to insert the negative cast (labelled with l') and closing substitutions (in Figure 2.15) to insert the positive cast (labelled with l). Do not be confused by the label used for function contract correspondence—this definition does, in fact, match up with closing substitutions. A binding $x{:}c^{l,l'} \in \Gamma$ must have come from the domain of an application of T_FunC, so the labels on the binding are *already* swapped when ϕ or $\Gamma \models \delta$ sees them. In the definition of function contract correspondence, we swap manually—whence the l' on the inserted cast. It helps to think of polarity in terms of position rather than the presence or absence of a prime.

2.5.1 Lemma [Expansion and contraction]: If $t \longrightarrow_{picky}{}^{*}\ t'$, and $s \longrightarrow_{h}^{*}\ s'$ then $t \sim s : T$ iff $t' \sim s' : T$.

2.5.2 Lemma [Constants self-correspond]: For all k, $k \sim k : \mathrm{ty_c}(k)$.

2.5.3 Lemma [Equivalence is closed under parallel reduction]: If $s \Rrightarrow s'$, then $t \sim s : T$ iff $t \sim s' : T$. Similarly, if $S \Rrightarrow S'$ then $c \sim^{l,l'} S : T$ iff $c \sim^{l,l'} S' : T$.

Proof: In both cases, by induction on T, using the first to prove the second. $\square$

2.5.4 Lemma [Trivial casts]: If $t \sim s : B$ and $\lfloor S \rfloor = B$, then $t \sim \langle S \Rightarrow \lceil B \rceil \rangle^l\ s : B$.

2.5.5 Lemma [Related base casts]: If $\{x{:}B \mid t\} \sim^{l_0,l_1} \{x{:}B \mid s\} : B$ and $t' \sim s' : B$ and $\lfloor S \rfloor = B$, then $\langle \{x{:}B \mid t\} \rangle^{l,l'}\ t' \sim \langle S \Rightarrow \{x{:}B \mid s\} \rangle^l\ s' : B$.

Proof: Direct. Note that l_0 and l_1 are entirely irrelevant. $\square$

2.5.6 Lemma [Bulletproofing]: If $t \sim s : T$ and $c \sim^{l,l'} S : T$ then $\langle c \rangle^{l,l'}\ t \sim \langle S \Rightarrow \lceil S \rceil \rangle^{l'}\ \langle \lceil S \rceil \Rightarrow S \rangle^l\ s : T$.

Proof: By induction on T. First, observe that either both t and s go to $\Uparrow l''$ or both t and s go to values related at T. In the former case, the outer terms also go to blame. So we only consider the case where $t \longrightarrow_{picky}{}^*\ v$, $s \longrightarrow_h^*\ w$, and $v \approx w : T$.

$\underline{T = B}$: So $c = \{x{:}B \mid t_1\}$ and $S = \{x{:}B \mid s_1\}$ and $S' = \{x{:}B \mid s_2\}$. By Lemma 2.5.5 we have $\langle c \rangle^{l,l'}\ t \sim \langle \lceil S \rceil \Rightarrow S \rangle^l\ s : B$. By Lemma 2.5.4 we can add the extra, trival cast $\langle S \Rightarrow \lceil S \rceil \rangle^{l'}$.

$\underline{T = T_1 \to T_2}$: We know that $c = x{:}c_1 \mapsto c_2$ and $S = x{:}S_1 \to S_2$. Let $t' \sim s' : T_1$. We only need to consider the case where $t' \longrightarrow_{picky}{}^*\ v'$ and $s' \longrightarrow_h^*\ w'$—if $t' \longrightarrow_{picky}{}^* \Uparrow l''$ and $s' \longrightarrow_h^* \Uparrow l''$ the outer terms correspond because both blame l''.

On the λ_{C} side, $(\langle c \rangle^{l,l'}\ t)\ t' \longrightarrow_{picky}{}^*\ \langle c_2\{x := \langle c_1 \rangle^{l',l}\ v'\} \rangle^{l,l'}\ (v\ (\langle c_1 \rangle^{l',l}\ v'))$. In λ_{H}, we can see

$$(\langle S \Rightarrow \lceil S \rceil \rangle^{l'}\ \langle \lceil S \rceil \Rightarrow S \rangle^l\ s)\ s' \longrightarrow_h^*$$
$$\langle S_2\{x := \langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ w'\} \Rightarrow \lceil S_2 \rceil \rangle^{l'}\ ((\langle \lceil S \rceil \Rightarrow S \rangle^l\ w)\ (\langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ w'))$$

We cannot determine where the redex is until we know the shape of T_1—does the negative argument cast step to an active check, or do we decompose the positive cast?

– $T_1 = B$. Since $v' \approx w' : B$, we must have $v' = w' = k \in \mathcal{K}_B$. By Lemma 2.5.5 and $c_1 \sim^{l',l} S_1 : B$, we know that $\langle c_1 \rangle^{l',l}\ v' \sim \langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ w' : B$. Both terms goes to blame or to the same value—which must be k, from inspection of the contract and cast evaluation rules.. The former case is immediate, since the outer terms then go to blame. So suppose $\langle c_1 \rangle^{l',l}\ k \longrightarrow_{picky}{}^*\ k$ and $\langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ k \longrightarrow_h^*\ k$. Now the terms evaluate like so:

$$\langle c_2\{x := \langle c_1 \rangle^{l',l}\ v'\} \rangle^{l,l'}\ (v\ (\langle c_1 \rangle^{l',l}\ v')) \longrightarrow_{picky}{}^*$$
$$\langle c_2\{x := \langle c_1 \rangle^{l',l}\ k\} \rangle^{l,l'}\ (v\ k)$$

$$\langle S_2\{x := \langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ k\} \Rightarrow \lceil S_2 \rceil \rangle^{l'}$$
$$\quad (\langle \lceil S \rceil \Rightarrow S \rangle^l\ w)\ (\langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ k)) \longrightarrow_h^*$$
$$\langle S_2\{x := \langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ k\} \Rightarrow \lceil S_2 \rceil \rangle^{l'}$$
$$\quad \langle \lceil S_2 \rceil \Rightarrow S_2\{x := k\} \rangle^l\ (w\ (\langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^l\ k))$$

By Lemma 2.5.4, $k \sim \langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^l\ k : B$, so $v\ k \sim w\ (\langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^l\ k) : T_2$.

We have by definition (and $k \sim k : B$) that $c_2\{x := \langle c_1 \rangle^{l',l}\ k\} \sim^{l,l'} S_2\{x := \langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ k\} : T_2$. Recall that $\langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ k \longrightarrow_h^* k$. This implies $\langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ k \Rrightarrow^* k$ (Lemma `A4` in the Coq). We can then see that $S_2\{x := \langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ k\} \Rrightarrow^* S_2\{x := k\}$ by Lemma `A1` in the Coq. By extension with the congruence rules:

$$\begin{array}{l}\langle S_2\{x := \langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ k\} \Rightarrow \lceil S_2 \rceil \rangle^{l'} \\ \quad \langle \lceil S_2 \rceil \Rightarrow S_2\{x := \langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ k\} \rangle^{l}\ (w\ (\langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^{l}\ k)) \Rrightarrow \\ \langle S_2\{x := \langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ k\} \Rightarrow \lceil S_2 \rceil \rangle^{l'} \\ \quad \langle \lceil S_2 \rceil \Rightarrow S_2\{x := k\} \rangle^{l}\ (w\ (\langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^{l}\ k))\end{array}$$

By the IH $\langle c_2\{x := \langle c_1 \rangle^{l',l}\ k\} \rangle^{l,l'}\ (v\ k)$ corresponds to the former, which means it is related to the latter by Lemma 2.5.3. We conclude the case with expansion (Lemma 2.5.1).

– $T_1 = T_{11} \to T_{12}$. We continue with an application of F_CDecomp in λ_{H}:

$$\begin{array}{l}\langle S_2\{x := \langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ w'\} \Rightarrow \lceil S_2 \rceil \rangle^{l'} \\ \quad ((\langle \lceil S \rceil \Rightarrow S \rangle^{l}\ w)\ (\langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ w')) \longrightarrow_h^* \\ \langle S_2\{x := \langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ w'\} \Rightarrow \lceil S_2 \rceil \rangle^{l'} \\ \quad \langle \lceil S_2 \rceil \Rightarrow S_2\{x := \langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ w'\} \rangle^{l} \\ \qquad (w\ (\langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^{l}\ (\langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ w')))\end{array}$$

By the IH on $c_1 \sim^{l',l} S_1 : T_1$ and $v' \sim w' : T_1$, we can find what we need for the domain: $\langle c_1 \rangle^{l',l}\ v' \sim \langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^{l}\ (\langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ w') : T_1$. By assumption, the results of applying v and w to these values correspond. (And they *are* values, since function contracts/casts applied to values are values.)

We have $c_2\{x := \langle c_1 \rangle^{l',l}\ v'\} \sim^{l,l'} S_2\{x := \langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l'}\ w'\} : T_2$ by assumption, so the IH tells us that the codomain contract and bulletproofing correspond. We conclude by expansion (Lemma 2.5.1). □ □

Having characterized how contracts and pairs of related casts relate, we show that terms correspond to their translation.

2.5.7 Theorem [Behavioral correspondence]: If $\vdash \Gamma$, then:

1. If $\phi(\Gamma \vdash t : T) = s$ then $\Gamma \vdash t \sim s : T$.
2. If $\phi(\Gamma \vdash^{l,l'} c : T) = S$ then $\Gamma \vdash c \sim^{l,l'} S : T$.

Proof: We simultaneously show both properties by induction on the depth of ϕ's recursion. □

We can now prove that ϕ preserves types, using Theorem 2.5.7 to show that ϕ preserves the implication judgment. As a preliminary, I use the behavioral correspondence to show that ϕ preserves the implication judgment.

2.5.8 Lemma: If $t_1 \longrightarrow_{picky}{}^* \mathsf{true}$ implies $t_2 \longrightarrow_{picky}{}^* \mathsf{true}$ then $\emptyset \vdash \phi(\emptyset \vdash t_1 : \mathsf{Bool}) \supset \phi(\emptyset \vdash t_1 : \mathsf{Bool})$.

Proof: By the logical relation. □

The type preservation proof is very similar to the correspondence proof of Theorem 2.5.7.

2.5.9 Theorem [Type preservation for ϕ]: If $\phi(\vdash \Gamma) = \Delta$, then:

1. $\vdash \Delta$.
2. If $\phi(\Gamma \vdash t : T) = s$ then $\Delta \vdash s : \lceil T \rceil$.
3. If $\phi(\Gamma \vdash^{l,l'} c : T) = S$ then $\Delta \vdash S$.

Proof: We prove all three properties simultaneously, by induction on the depth of ϕ's recursion.

The proof is by cases on the λ_{C} context well-formedness/term typing/contract well-formedness derivations, which determine the branch of ϕ taken. □

2.5.2 Translating λ_{H} to lax λ_{C}

We reuse the term correspondence $t \sim s : T$ (Figure 2.14), interpreting it as using $\longrightarrow_{lax}$. and define a new contract/cast correspondence $c \sim S_1 \Rightarrow^l S_2 : T$ (Figure 2.16), relating contracts and pairs of λ_{H} types—effectively, casts. This correspondence uses the term correspondence in the base type case and follows the pattern of F_CDecomp in the function case. Since it inserts a cast in the function case, I index the relation with a label, just like ψ. Note that the correspondence is blame-exact, relating λ_{C} and λ_{H} terms that either blame the same label or go to corresponding values.I define closing substitutions ignoring the contracts in the context; we lift the relation to open terms in the standard way.

We begin with some standard properties of the term correspondence relation.

2.5.10 Lemma [Expansion and contraction]: If $t \longrightarrow_{lax}{}^* t'$, and $s \longrightarrow_h^* s'$ then $t \sim s : T$ iff $t' \sim s' : T$.

2.5.11 Lemma [Blame corresponds to blame]: For all T, $\Uparrow l \sim \Uparrow l : T$.

2.5.12 Lemma [Constants self-correspond]: For all k, $k \approx k : \mathrm{ty_c}(k)$.

As a corollary of Lemma 2.5.11 and Lemma 2.5.10, if two terms evaluate to blame, then they correspond. This will be used extensively in the proofs below, as it allows us to eliminate many cases.

Contract/type correspondence

$$\boxed{c \sim S_1 \Rightarrow^l S_2 : T}$$

$$\{x{:}B \mid t\} \sim \{x{:}B \mid s_1\} \Rightarrow^l \{x{:}B \mid s_2\} : B \iff \forall k \in \mathcal{K}_B.\ t\{x := k\} \sim s_2\{x := k\} : \mathsf{Bool}$$

$$x{:}c_1 \mapsto c_2 \sim x{:}S_{11} \to S_{12} \Rightarrow^l S_{21} \to S_{22} : T_1 \to T_2 \iff c_1 \sim S_{21} \Rightarrow^l S_{11} : T_1 \wedge \forall t \sim s : T_1.c_2\{x := t\} \sim S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ s\} \Rightarrow^l S_{22}\{x := s\} : T_2$$

Dual closing substitutions

$$\Gamma \models \delta \iff \begin{cases} \forall x{:}T \in \Gamma, & \delta_1(x) \sim \delta_2(x) : T \\ \forall x{:}c^{l,l'} \in \Gamma, & \delta_1(x) \sim \delta_2(x) : \lfloor c \rfloor \end{cases}$$

Lifted to open terms

$$\begin{array}{rcl} \Gamma \vdash t \sim s : T & \iff & \forall \delta.\ (\Gamma \models \delta \ \text{ implies } \ \delta_1(t) \sim \delta_2(s) : T) \\ \Gamma \vdash c \sim S_1 \Rightarrow^l S_2 : T & \iff & \forall \delta.\ (\Gamma \models \delta \ \text{ implies } \ \delta_1(c) \sim \delta_2(S_1) \Rightarrow^l \delta_2(S_2) : T) \end{array}$$

Figure 2.16: Blame-exact correspondence for ψ into lax λ_{C}

2.5.13 Lemma [Corresponding terms coevaluate]: If $t \sim s : T$ then $t \longrightarrow_{lax}{}^* v \wedge s \longrightarrow_h^* w$ or $t \longrightarrow_{lax}{}^* \Uparrow l \wedge s \longrightarrow_h^* \Uparrow l$; moreover, $t \longrightarrow_{lax}{}^* r$ and $s \longrightarrow_h^* q$ such that $r \approx q : T$.

2.5.14 Lemma [Contract/cast correspondence]: If $c \sim S_1 \Rightarrow^l S_2 : T$ and $t \sim s : T$ then $\langle c\rangle^{l,l}\ t \sim \langle S_1 \Rightarrow S_2\rangle^l\ s : T$.

Proof: By induction on T. We reason via expansion (Lemma 2.5.10), showing that the initial terms reduce to corresponding terms.

$\underline{T = B}$: So $c = \{x{:}B \mid t_1\}$, $S_1 = \{x{:}B \mid s_1\}$, and $S_2 = \{x{:}B \mid s_2\}$. Since $t \sim s : B$, we know that they either both reduce to $k \in \mathcal{K}_B$ or $\Uparrow l'$. If the latter is the case, we are done. So suppose $t \longrightarrow_{lax}{}^* k$ along with $s \longrightarrow_h^* k$.

We can step our terms into active checks as follows, then:

$$\begin{array}{rcl} \langle\{x{:}B \mid t_1\}\rangle^{l,l}\ t & \longrightarrow_{lax}{}^* & \langle\{x{:}B \mid t_1\}, t_1\{x := k\}, k\rangle^l \\ \langle\{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid s_2\}\rangle^l\ s & \longrightarrow_h^* & \langle\{x{:}B \mid s_2\}, s_2\{x := k\}, k\rangle^l \end{array}$$

By inversion of the contract/cast correspondence, we know that $t_1\{x := k\} \sim s_2\{x := k\} : \mathsf{Bool}$, so these terms go to blame or to a Bool together. If they go to $\Uparrow l'$, we are done. If they go to false, then both the obligation and the cast will go to $\Uparrow l$. Finally, if they both go to true, then both terms will evaluate to k.

$\underline{T = T_1 \to T_2}$: $c = x{:}c_1 \mapsto c_2$, $S_1 = x{:}S_{11} \to S_{12}$, and $S_2 = x{:}S_{21} \to S_{22}$. We know by inversion of the contract/cast relation that $c_1 \sim S_{21} \Rightarrow^l S_{11} : T_1$ and that for all $t \sim s : T_1$, $c_2\{x := t\} \sim S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ s\} \Rightarrow^l S_{22}\{x := s\} : T_2$. We

want to prove that $\langle c \rangle^{l,l} \sim \langle S_1 \Rightarrow S_2 \rangle^l \; s : T_1 \to T_2$. First, we can assume $t \longrightarrow_{lax}{}^* v$ and $s \longrightarrow_h^* w$ where $v \sim w : T_1 \to T_2$—if not, both cast and contracted terms go to blame and we are done.

We show that the decomposition of the contract and cast terms correspond for all inputs. Let $t' \sim s' : T_1$. Again, we can assume that they reduce to $v' \sim w' : T_1$, or else we are done by blame lifting. On the λ_{C} side, we have

$$(\langle c \rangle^{l,l} \; t) \; t' \longrightarrow_{lax}{}^* \langle c_2\{x := v'\} \rangle^{l,l} \; (v \; (\langle c_1 \rangle^{l,l} \; v'))$$

In λ_{H}, we find

$$\begin{array}{l}(\langle S_1 \Rightarrow S_2 \rangle^l \; s) \; s' \longrightarrow_h^* \\ \langle S_{12}\{x := \langle S_{21} \Rightarrow S_{11} \rangle^l \; w\} \Rightarrow S_{22}\{x := w'\} \rangle^l \; (w \; (\langle S_{21} \Rightarrow S_{11} \rangle^l \; w'))\end{array}$$

By the IH, we know that $\langle c_1 \rangle^{l,l} \; v' \sim \langle S_{21} \Rightarrow S_{11} \rangle^l \; w' : T_1$. Since $v \sim w : T_1 \to T_2$, we have $v \; (\langle c_1 \rangle^{l,l} \; v') \sim w \; (\langle S_{21} \Rightarrow S_{11} \rangle^l \; w') : T_2$. Again by the IH, we can see that $\langle c_2\{x := v'\} \rangle^{l,l} \; (v \; (\langle c_1 \rangle^{l,l} \; v')) \sim \langle S_{12}\{x := \langle S_{21} \Rightarrow S_{11} \rangle^l \; w'\} \Rightarrow S_{22}\{x := w'\} \rangle^l \; w \; (\langle S_{21} \Rightarrow S_{11} \rangle^l \; w') : T_2$. □ □

We prove three more technical lemmas necessary for the behavioral and type correspondence.

2.5.15 Lemma [Skeletal equality of subtypes]: If $\Delta \vdash S_1 <: S_2$, then $\lfloor S_1 \rfloor = \lfloor S_2 \rfloor$.

2.5.16 Lemma: If $\lfloor S_1 \rfloor = \lfloor S_2 \rfloor = T$, then $\lfloor \psi^l(S_1, S_2) \rfloor = T$.

2.5.17 Lemma: If $\Delta_1 \vdash S_1$ and $\Delta_1 \vdash S_2$, where $\lfloor S_1 \rfloor = \lfloor S_2 \rfloor$ then

1. If $\Delta_1, x{:}S_1, \Delta_2 \vdash s : S$ then $\Delta_1, x{:}S_2, \Delta_2\{x := \langle S_2 \Rightarrow S_1 \rangle^l \; x\} \vdash s\{x := \langle S_2 \Rightarrow S_1 \rangle^l \; x\} : S\{x := \langle S_2 \Rightarrow S_1 \rangle^l \; x\}$, and
2. If $\Delta_1, x{:}S_1, \Delta_2 \vdash S$ then $\Delta_1, x{:}S_2, \Delta_2\{x := \langle S_2 \Rightarrow S_1 \rangle^l \; x\} \vdash S\{x := \langle S_2 \Rightarrow S_1 \rangle^l \; x\}$.

We use the correspondence relations to show that s and its translation $\psi(s)$ correspond—i.e., that ψ faithfully translates the λ_{H} semantics. We must choose the subject of induction carefully, however, to ensure that we can apply the IH in the case for function casts. An induction on the height of the well-formedness derivation is tricky because of the "extra" substitution that ψ does. Instead, we do induction on the depth of ψ's recursion, (and also derivation height, for the S_Sub case).

2.5.18 Theorem [Behavioral correspondence]:

1. If $\Delta \vdash s : S$ then $\lfloor \Delta \rfloor \vdash \psi(s) \sim s : \lfloor S \rfloor$.

2. If $\Delta \vdash S_1$ and $\Delta \vdash S_2$, where $\lfloor S_1 \rfloor = \lfloor S_2 \rfloor = \lfloor S \rfloor$, then $\lfloor \Delta \rfloor \vdash \psi^l(S_1, S_2) \sim S_1 \Rightarrow^l S_2 : \lfloor S \rfloor$ (for all l).

Proof: By induction on the lexicographically ordered pairs (m, n), where m is the depth of the recursion of the translation $\psi(s)$ (for part 1) or $\psi^l(S_1, S_2)$ (for part 2) and n is either $|\Delta \vdash s : S|$ (for part 1) or $|\Delta \vdash S_1| + |\Delta \vdash S_2|$ (for part 2). The first component decreases in all uses of the IH except for the S_Sub case, where only the second component decreases. Part (1) of the proof proceeds by case analysis on the final rule used in the typing derivation $\Delta \vdash s : S$. Which rule was used determines the shape of $\psi(s)$ in all cases but S_Sub.

We give only the most interesting cases for the first part: S_Cast, S_Checking, and S_Sub.

S_Cast: $\Delta \vdash \langle S_1 \Rightarrow S_2 \rangle^l : S_1 \to S_2$ and $\psi(\langle S_1 \Rightarrow S_2 \rangle^l) = \langle \psi^l(S_1, S_2) \rangle^{l,l}$. By inversion, $\Delta \vdash S_1$ and $\Delta \vdash S_2$, where $\lfloor S_1 \rfloor = \lfloor S_2 \rfloor$.

By the IH for proposition (2), $\lfloor \Delta \rfloor \vdash \psi^l(S_1, S_2) \sim S_1 \Rightarrow^l S_2 : \lfloor S_2 \rfloor$.

Let $\lfloor \Delta \rfloor \models \delta$; we must show $\delta_1(\langle \psi^l(S_1, S_2) \rangle^{l,l}) \sim \delta_2(\langle S_1 \Rightarrow S_2 \rangle^l) : \lfloor S_1 \rfloor \to \lfloor S_2 \rfloor$. Let $t \sim s : \lfloor S_1 \rfloor$. We have $\delta_1(\langle \psi^l(S_1, S_2) \rangle^{l,l})\ t \sim \delta_2(\langle S_1 \Rightarrow S_2 \rangle^l)\ s : \lfloor S_2 \rfloor$ by Lemma 2.5.14.

S_Checking: We have $\Delta \vdash \langle \{x{:}B \mid s_1\}, s_2, k \rangle^l : \{x{:}B \mid s_1\}$; translating yields $\psi(\langle \{x{:}B \mid s_1\}, s_2, k \rangle^l) = \langle \{x{:}B \mid \psi(s_1)\}, \psi(s_2), k \rangle^l$. Recall that the terms of the active check are closed. By inversion we have $\emptyset \vdash s_2 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}$ and $\emptyset \vdash k : \{x{:}B \mid \mathsf{true}\}$, so $k \in \mathcal{K}_B$.

By the IH, $\psi(s_2) \sim s_2 : \mathsf{Bool}$. These two terms coevaluate to blame or a boolean constant. There are three cases, all of which result in the active checks evaluating to $\approx$-corresponding values:

- If they go to $\Uparrow l'$, then the checks do too, and $\Uparrow l' \approx \Uparrow l' : B$.
- If they go to false, then the checks go to $\Uparrow l$, and $\Uparrow l \approx \Uparrow l : B$.
- If they go to true, then the checks go to $k \in \mathcal{K}_B$, and $k \approx k : B$.

S_Sub: $\Delta \vdash s : S$; we do not know anything about the shape of $\psi(s)$. By inversion, $\Delta \vdash s : S'$ and $\Delta \vdash S' <: S$. By Lemma 2.5.15, $\lfloor S' \rfloor = \lfloor S \rfloor$.

Since the sub-derivation $\Delta \vdash s : S'$ is smaller, by the IH $\lfloor \Delta \rfloor \vdash \psi(s) \sim s : \lfloor S' \rfloor$. But $\lfloor S' \rfloor = \lfloor S \rfloor$, so we are done.

Part (2) of this proof proceeds by cases on $\psi^l(S_1, S_2) = c$.

$\psi^l(S_1, \{x{:}B \mid s_2\}) = \{x{:}B \mid \psi(s_2)\}$: Note that $S_2 = \{x{:}B \mid s_2\}$. By inversion of $\Delta \vdash \{x{:}B \mid s_2\}$, we have $\Delta, x{:}\{x{:}B \mid \mathsf{true}\} \vdash s_2 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}$.

By the IH for proposition (1), $\lfloor \Delta \rfloor, x{:}B \vdash \psi(s_2) \sim s_2 : \mathsf{Bool}$.

We must show $\lfloor \Delta \rfloor \vdash \{x{:}B \mid \psi(s_2)\} \sim S_1 \Rightarrow^l \{x{:}B \mid s_2\} : B$. Let $\lfloor \Delta \rfloor \models \delta$; we prove that $\delta_1(\{x{:}B \mid \psi(s_2)\}) \sim \delta_2(S_1) \Rightarrow^l \delta_2(\{x{:}B \mid s_2\}) : B$, i.e., for all $k \in \mathcal{K}_B$, that $\delta_1(\psi(s_2))\{x := k\} \sim \delta_2(s_2)\{x := k\} : \mathsf{Bool}$. Since $k \sim k : B$, we can see this last by the IH.

$\underline{\psi^l(x{:}S_{11} \to S_{12}, x{:}S_{21} \to S_{22}) = x{:}\psi^l(S_{21}, S_{11}) \mapsto \psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ x\}, S_{22})}$:

We can see $S_2 = x{:}S_{21} \to S_{22}$ and so $S_1 = x{:}S_{11} \to S_{12}$, where $\lfloor S_{21} \rfloor = \lfloor S_{11} \rfloor$ and $\lfloor S_{22} \rfloor = \lfloor S_{12} \rfloor$. By inversion, we have the following well-formedness derivations:

$$\begin{array}{rr} \Delta \vdash S_{21} & \Delta \vdash S_{11} \\ \Delta, x{:}S_{21} \vdash S_{22} & \Delta, x{:}S_{22} \vdash S_{12} \end{array}$$

We can apply the IH (contravariantly) to see

$$\lfloor \Delta \rfloor \vdash \psi^l(S_{21}, S_{11}) \sim S_{21} \Rightarrow^l S_{11} : \lfloor S_{11} \rfloor \qquad (*)$$

By weakening (Lemma 2.3.22), we can see $\Delta, x{:}S_{21} \vdash S_{21}$ and $\Delta, x{:}S_{21} \vdash S_{11}$. We can reapply the IH to show $\lfloor \Delta \rfloor, x{:}\lfloor S_{21} \rfloor \vdash \psi^l(S_{21}, S_{11}) \sim S_{21} \Rightarrow^l S_{11} : \lfloor S_{11} \rfloor$. Now $\Delta, x{:}S_{21} \vdash \langle S_{21} \Rightarrow S_{11}\rangle^l : S_{21} \to S_{11}$ and $\Delta, x{:}S_{21} \vdash \langle S_{21} \Rightarrow S_{11}\rangle^l\ x : S_{11}$. By Lemma 2.5.17, we can substitute this last into $\Delta, x{:}S_{11} \vdash S_{12}$, finding $\Delta, x{:}S_{21} \vdash S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ x\}$.

We apply the IH for proposition (2) on $\Delta, x{:}S_{21} \vdash S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ x\}$ and $\Delta, x{:}S_{21} \vdash S_{22}$, showing

$$\begin{array}{rl} \lfloor \Delta \rfloor, x{:}\lfloor S_{21} \rfloor \vdash & \psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ x\}, S_{22}) \sim \\ & S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ x\} \Rightarrow^l S_{22} : \lfloor S_{22} \rfloor \qquad (**) \end{array}$$

We now combine (*) and (**) to show $\lfloor \Delta \rfloor \vdash \psi^l(x{:}S_{11} \to S_{12}, x{:}S_{21} \to S_{22}) \sim x{:}S_{11} \to S_{12} \Rightarrow^l x{:}S_{21} \to S_{22} : \lfloor S_2 \rfloor$. Let $\lfloor \Delta \rfloor \models \delta$. We can apply (*) to see $\delta_1(\psi^l(S_{21}, S_{11})) \sim \delta_2(S_{21}) \Rightarrow^l \delta_2(S_{11}) : \lfloor S_{11} \rfloor$. For the codomain we must show, for all $t \sim s : \lfloor S_{11} \rfloor$, that

$$\begin{array}{l} \delta_1(\psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ x\}, S_{22}))\{x := t\} \sim \\ \delta_2(S_{12})\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ s\} \Rightarrow^l \delta_2(S_{22})\{x := s\} : \lfloor S_{22} \rfloor \end{array}$$

Let $t \sim s : \lfloor S_{11} \rfloor$. Recalling that $\lfloor S_{11} \rfloor = \lfloor S_{21} \rfloor$, observe $\lfloor \Delta \rfloor, x{:}\lfloor S_{21} \rfloor \models \delta\{x := t, s\}$. Call this δ'. By (**) we see

$$\begin{array}{l} \delta'_1(\psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ x\}, S_{22})) \sim \\ \delta'_2(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ x\}) \Rightarrow^l \delta'_2(S_{22}) : \lfloor S_{22} \rfloor \end{array}$$

which we can rewrite to

$$\begin{array}{l} \delta_1(\psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ x\}, S_{22}))\{x := t\} \sim \\ \delta_2(S_{12})\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ s\} \Rightarrow^l \delta_2(S_{22})\{x := s\} : \lfloor S_{22} \rfloor \end{array}$$

This is exactly what we needed to finish the proof of correspondence. □

As a preliminary to type-preservation, we can use behavioral correspondence to show that the implication judgment is preserved.

2.5.19 Lemma: If $\emptyset \vdash s_1 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}$ and $\emptyset \vdash s_2 : \{x{:}\mathsf{Bool} \mid \mathsf{true}\}$ and $\emptyset \vdash s_1 \supset s_2$, then $\psi(s_1) \longrightarrow_{lax}{}^* \mathsf{true}$ implies $\psi(s_2) \longrightarrow_{lax}{}^* \mathsf{true}$.

Proof: By the logical relation. □

The type preservation proof is very similar to the correspondence proof of Theorem 2.5.18, though the function case of the type/contract correspondence is intricate.

2.5.20 Theorem [Type preservation for ψ]:

1. If $\Delta \vdash s : S$ then $\lfloor \Delta \rfloor \vdash \psi(s) : \lfloor S \rfloor$.
2. If $\Delta \vdash S_1$, $\Delta \vdash S_2$, where $\lfloor S_1 \rfloor = \lfloor S_2 \rfloor = T$, then $\lfloor \Delta \rfloor \vdash^{l,l'} \psi^l(S_1, S_2) : T$.

Proof: By induction on the lexicographically ordered pair containing (a) the depth of the recursion of the translation ψ or $\psi(s)$, and (b) $|\Delta \vdash s : S|$ or $|\Delta \vdash S_1| + |\Delta \vdash S_2|$.

Part (1) of the proof proceeds by case analysis on the final rule of $\Delta \vdash s : S$, which determines the shape of $\psi(s) = t$ in all cases but S_Sub. Part (2) of the proof proceeds by case analysis on $\psi^l(S_1, S_2) = c$.

$\underline{\psi^l(x{:}S_{11} \to S_{12}, x{:}S_{21} \to S_{22}) = x{:}\psi^l(S_{21}, S_{11}) \mapsto \psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ x\}, S_{22})}$:

We must have $S_2 = x{:}S_{21} \to S_{22}$ and $S_1 = x{:}S_{11} \to S_{12}$, where $\lfloor S_{21} \rfloor = \lfloor S_{11} \rfloor$ and $\lfloor S_{22} \rfloor = \lfloor S_{12} \rfloor$. By inversion, we have the following well-formedness derivations:

$$\begin{array}{rr} \Delta \vdash S_{21} & \Delta \vdash S_{11} \\ \Delta, x{:}S_{21} \vdash S_{22} & \Delta, x{:}S_{22} \vdash S_{12} \end{array}$$

By the IH $\lfloor \Delta \rfloor \vdash^{l',l} \psi^l(S_{21}, S_{11}) : \lfloor S_{11} \rfloor$. Note that $\lfloor \psi^l(S_{21}, S_{11}) \rfloor = \lfloor S_{21} \rfloor$.

By weakening, we can see $\Delta, x{:}S_{21} \vdash S_{21}$ and $\Delta, x{:}S_{21} \vdash S_{11}$. We can reapply the IH to show $\lfloor \Delta \rfloor, x{:}\lfloor S_{21} \rfloor \vdash^{l',l} \psi^l(S_{21}, S_{11}) : \lfloor S_{11} \rfloor$. Now $\Delta, x{:}S_{21} \vdash \langle S_{21} \Rightarrow S_{11}\rangle^l : S_{21} \to S_{11}$. Next $\Delta, x{:}S_{21} \vdash \langle S_{21} \Rightarrow S_{11}\rangle^l\ x : S_{11}$. By Lemma 2.5.17, we can substitute this last into $\Delta, x{:}S_{11} \vdash S_{12}$, finding $\Delta, x{:}S_{21} \vdash S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ x\}$.

By the IH for proposition (2) on $\Delta, x{:}S_{21} \vdash S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ x\}$ and $\Delta, x{:}S_{21} \vdash S_{22}$,

$$\lfloor \Delta \rfloor, x{:}\lfloor S_{21} \rfloor \vdash^{l,l'} \psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ x\}, S_{22}) : \lfloor S_{22} \rfloor$$

By Lemma 2.5.16, $\lfloor \psi^l(S_{21}, S_{11}) \rfloor = \lfloor S_{21} \rfloor$, so we can rewrite the above derivation to

$$\lfloor \Delta \rfloor, x{:}\psi^l(S_{21}, S_{11})^{l',l} \vdash^{l,l'} \psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ x\}, S_{22}) : \lfloor S_{22} \rfloor$$

Now by T_FunC

$$\lfloor \Delta \rfloor \vdash^{l,l'} x{:}\psi^l(S_{21}, S_{11}) \mapsto \psi^l(S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^l\ x\}, S_{22}) : \lfloor S_{21} \rfloor \to \lfloor S_{22} \rfloor \quad \square$$

□

2.6 Inexact translations

The same translations ϕ and ψ can be used to move right on the axis of blame (Figure 2.1). However, in this direction the images of these translations blame strictly more than their pre-images. We were able to use the same correspondence for both exact proofs in Section 2.5, but the following two proofs use custom correspondences: one where lax λ_{C} terms correspond to λ_{H} terms (with possibly more blame), and one where λ_{H} terms correspond to picky λ_{C} terms (with possibly more blame). In both cases, the λ_{C} terms will be on the left and the λ_{H} terms on the right.

2.6.1 Translating lax $\lambda_{\mathbf{C}}$ to $\lambda_{\mathbf{H}}$

Translating with ϕ from terms in picky λ_{C} to exactly corresponding terms in λ_{H} was a relatively straightforward generalization of the nondependent case; things get more interesting when we consider the translation ϕ from lax λ_{C} to dependent λ_{H}. We can prove that it preserves types (for terms without active checks), but we can only show a weaker behavioral correspondence: sometimes lax λ_{C} terms terminate with values when their ϕ-images go to blame. This weaker property is a consequence of bulletproofing, the asymmetrically substituting F_CDecomp rule, and the extra casts inserted for type preservation (i.e., for T_VarC derivations). This is not a weakness of my proof technique—I give a counterexample, a lax λ_{C} term $\emptyset \vdash t : T$ such that $t \longrightarrow_{lax}{}^{*} v$ and $\phi(\emptyset \vdash t : T) \longrightarrow_{h}^{*} \Uparrow l$.

We can show the behavioral correspondence using a blame-inexact logical relation, defined in Figure 2.17. The behavioral correspondence here, though weaker than before, is still pretty strong: if $t \sim_{\succ} s : B$ (read "t blames no more than s at type B"), then either $s \longrightarrow_{h}^{*} \Uparrow l$ or t and s both go to $k \in \mathcal{K}_B$. This correspondence differs slightly in construction from the earlier exact one—we define $\approx_{\succ}$ as a relation on *values*, while $\approx$ is a relation on *results*. Doing so simplifies the inexact treatment of blame—in particular, Lemma 2.6.2. We again use the term correspondence to relate contracts and λ_{H} types. We then lift the correspondences to open terms (Figure 2.17). Closing substitutions map variables to corresponding terms of appropriate type. Note that closing substitutions ignore the contract part of $x{:}c^{l,l'}$ bindings, treating them as if they were $x{:}\lfloor c \rfloor$.

2.6.1 Lemma [Expansion and contraction]: If $t \longrightarrow_{lax}{}^{*} t'$, and $s \longrightarrow_{h}^{*} s'$ then $t \sim_{\succ} s : T$ iff $t' \sim_{\succ} s' : T$.

Note that there are corresponding terms at every type. We can prove a much stronger lemma than we did for $\sim$ in Lemma 2.5.11, since the correspondence here is much weaker.

2.6.2 Lemma [Everything corresponds to blame]: For all t and T, $t \sim_{\succ} \Uparrow l' : T$.

Value correspondence $\boxed{v \approx_{\succ} w : T}$

$$k \approx_{\succ} k : B \iff k \in \mathcal{K}_B$$

$$v \approx_{\succ} w : T_1 \to T_2 \iff \forall t \sim_{\succ} s : T_1.\ v\ t \sim_{\succ} w\ s : T_2$$

Term correspondence $\boxed{t \sim_{\succ} s : T}$

$$t \sim_{\succ} s : T \iff s \longrightarrow_h^* \Uparrow l \vee (t \longrightarrow_{lax}{}^* v \wedge s \longrightarrow_h^* w \wedge v \approx_{\succ} w : T)$$

Contract/type correspondence $\boxed{c \sim_{\succ} S : T}$

$$\{x{:}B \mid t\} \sim_{\succ} \{x{:}B \mid s\} : B \iff \forall k \in \mathcal{K}_B.\ t\{x := k\} \sim_{\succ} s\{x := k\} : \mathsf{Bool}$$

$$x{:}c_1 \mapsto c_2 \sim_{\succ} x{:}S_1 \to S_2 : T_1 \to T_2 \iff c_1 \sim_{\succ} S_1 : T_1 \wedge \forall t \sim_{\succ} s : T_1.\ c_2\{x := t\} \sim_{\succ} S_2\{x := s\} : T_2$$

Dual closing substitutions

$$\Gamma \models_{\succ} \delta \iff \forall x \in \mathrm{dom}(\Gamma).\ \delta_1(x) \sim_{\succ} \delta_2(x) : \lfloor \Gamma(x) \rfloor$$

Lifted to open terms

$$\Gamma \vdash t \sim_{\succ} s : T \iff \forall \delta.\ (\Gamma \models_{\succ} \delta \ \text{ implies } \ \delta_1(t) \sim_{\succ} \delta_2(s) : T)$$
$$\Gamma \vdash c \sim_{\succ} S : T \iff \forall \delta.\ (\Gamma \models_{\succ} \delta \ \text{ implies } \ \delta_1(c) \sim_{\succ} \delta_2(S) : T)$$

Figure 2.17: Blame-inexact correspondence for ϕ from lax λ_{C}

2.6.3 Lemma [Constants self-correspond]: For all k, $k \approx_{\succ} k : \mathrm{ty_c}(k)$.

Proof: By induction on $\mathrm{ty_c}(k)$, recalling that constants are first order. $\square$

As a corollary of Lemma 2.6.2 and Lemma 2.6.1, if two terms evaluate to blame—or even just the λ_{H} side!—then they correspond. This will be used extensively in the proofs below, as it allows us to eliminate many cases.

We prove three lemmas about contracts and casts at base types. The first two characterize contracts and casts at base types.

2.6.4 Lemma [Trivial casts]: If $t \sim_{\succ} s : B$, then $t \sim_{\succ} \langle S \Rightarrow \lceil B \rceil \rangle^l \; s : B$ for all S.

2.6.5 Lemma [Related base casts]: If $\{x{:}B \mid t\} \sim_{\succ} \{x{:}B \mid s\} : B$ and $t' \sim_{\succ} s' : B$, then $\langle \{x{:}B \mid t\} \rangle^{l,l'} \; t' \sim_{\succ} \langle S \Rightarrow \{x{:}B \mid s\} \rangle^l \; s' : B$ for all S.

The third lemma shows that correspondence is closed under adding extra casts to the λ_{H} term, due to the inexactness of the behavioral correspondence. Since λ_{H} terms can go to blame more often than corresponding lax λ_{C} terms, we can add "extra" casts to λ_{H} terms. We formalize this in the following lemma, which captures the asymmetric treatment of blame by the $\sim_{\succ}$ relation. We use it to show that the cast substituted in the codomain by F_CDecomp does not affect behavioral correspondence. Note that the statement of the lemma requires that the types of the cast correspond to *some* contracts at the same type T, but we never use the contracts in the proof—they witness the well-formedness of the λ_{H} types.

2.6.6 Lemma [Extra casts]: If $t \sim_{\succ} s : T$ and $c_1 \sim_{\succ} S_1 : T$ and $c_2 \sim_{\succ} S_2 : T$, then $t \sim_{\succ} \langle S_1 \Rightarrow S_2 \rangle^l \; s : T$.

Proof: The proof is by induction on T. Note that we do not use c_1 or c_2 at all in the proof, but instead they are witnesses to the well-formedness of S_1 and S_2.

$\lfloor S_1 \rfloor = \lfloor S_2 \rfloor = T$. Either $s \longrightarrow_h^* \Uparrow l'$ or t and s both go to corresponding values at T. If $s \longrightarrow_h^* \Uparrow l'$, then $\langle S_1 \Rightarrow S_2 \rangle^l \; s \longrightarrow_h^* \Uparrow l'$ and $t \sim_{\succ} \Uparrow l' : T$ since everything is related to blame (Lemma 2.6.2).

Therefore, suppose that $t \longrightarrow_{lax}{}^* v$ and $s \longrightarrow_h^* w$ and $v \approx_{\succ} w : T$ in each of the following cases of the induction.

$\underline{T = B}$: So $S_2 = \{x{:}B \mid s_2\}$, and $c_2 = \{x{:}B \mid t_2\}$.

So $t \longrightarrow_{lax}{}^* k$ and $s \longrightarrow_h^* k$ for $k \in \mathcal{K}_B$. If t and s both go to k, then $\langle S_1 \Rightarrow S_2 \rangle^l \; s \longrightarrow_h^* \langle \{x{:}B \mid s_2\}, s_2\{x := k\}, k \rangle^l$. By $c_2 \sim_{\succ} S_2 : B$ we see (in particular) $t_2\{x := k\} \sim_{\succ} s_2\{x := k\} : \mathsf{Bool}$. So $s_2\{x := k\}$ either goes to $\Uparrow l'$ or $s_2\{x := k\}$ (and, irrelevantly, $t_2\{x := k\}$) go to some $k' \in \mathcal{K}_{\mathsf{Bool}}$. In the former case, $\langle \{x{:}B \mid s_2\}, s_2\{x := k\}, k \rangle^l \longrightarrow_h^* \Uparrow l'$ and we are done (by Lemma 2.6.2). In the latter case, the λ_{H} term either goes to $\Uparrow l$ (and everything is related to blame) or goes to k—but so does t, and $k \approx_{\succ} k : B$.

$\underline{T = T_1 \to T_2}$: We have:

$$\begin{array}{rclcrcl} S_1 & = & x{:}S_{11} \to S_{12} & & S_2 & = & x{:}S_{21} \to S_{22} \\ c_1 & = & x{:}c_{11} \mapsto c_{12} & & c_2 & = & x{:}c_{21} \mapsto c_{22} \end{array}$$

We have $t \longrightarrow_{lax}{}^* v$ and $s \longrightarrow_h^* w$, where $v \approx_\succ w : T_1 \to T_2$.

Let $t' \sim_\succ s' : T_1$; we wish to see that $v\ t' \sim_\succ (\langle S_1 \Rightarrow S_2 \rangle^l\ w)\ s' : T_2$. Either $s' \longrightarrow_h^* \Uparrow l'$ or both go to values. In the former case the whole cast goes to $\Uparrow l'$ we are done by Lemma 2.6.2, so let $t' \longrightarrow_{lax}{}^* v'$ and $s' \longrightarrow_h^* w'$.

Decomposing the cast in λ_{H},

$$\begin{array}{l} (\langle S_1 \Rightarrow S_2 \rangle^l\ w)\ s' \longrightarrow_h^* \\ \langle S_{12}\{x := \langle S_{21} \Rightarrow S_{11} \rangle^l\ w'\} \Rightarrow S_{22}\{x := w'\} \rangle^l\ (w\ (\langle S_{21} \Rightarrow S_{11} \rangle^l\ w')) \end{array}$$

We have $c_{21} \sim_\succ S_{21} : T_1$ and $c_{11} \sim_\succ S_{11} : T_1$, so $v' \sim_\succ \langle S_{21} \Rightarrow S_{11} \rangle^l\ w' : T_1$ by the IH. Since $v \approx_\succ w : T_1 \to T_2$, we can see that $v\ v' \sim_\succ w\ (\langle S_{21} \Rightarrow S_{11} \rangle^l\ w') : T_2$.

Furthermore, we know that for all $t'' \sim_\succ s'' : T_1$ that

- $c_{12}\{x := t''\} \sim_\succ S_{12}\{x := s''\} : T_2$ and
- $c_{22}\{x := t''\} \sim_\succ S_{22}\{x := s''\} : T_2$.

We know that $v' \sim_\succ w' : T_1$ and $v' \sim_\succ \langle S_{21} \Rightarrow S_{11} \rangle^l\ w' : T_1$, so we can see

- $c_{12}\{x := v'\} \sim_\succ S_{12}\{x := w'\} : T_2$ and
- $c_{22}\{x := v'\} \sim_\succ S_{22}\{x := \langle S_{21} \Rightarrow S_{11} \rangle^l\ w'\} : T_2$.

So by the IH,

$$v\ v' \sim_\succ \langle S_{12}\{x := \langle S_{21} \Rightarrow S_{11} \rangle^l\ w'\} \Rightarrow S_{22}\{x := w'\} \rangle^l\ (w\ (\langle S_{21} \Rightarrow S_{11} \rangle^l\ w')) : T_2$$

and we are done by expansion (Lemma 2.6.1). □ □

To apply the extra cast lemma, we'll need these "witness" contracts for raw types; to that end we define trivial contracts. These contracts are *lifted* from types, and are the λ_{C} correlate to λ_{H}'s raw types.

$$\begin{array}{rcl} B\uparrow & = & \{x{:}B \mid \mathsf{true}\} \\ (T_1 \to T_2)\uparrow & = & (T_1\uparrow) \mapsto (T_2\uparrow) \end{array}$$

2.6.7 Lemma [Lifted types logically relate to raw types]: For all T, $T\uparrow \sim_\succ \lceil T \rceil : T$.

The "bulletproofing" lemma is the key to the behavioral correspondence proof. I show that a contract application corresponds to bulletproofing with related types. Note that I allow for different types in the two casts. This is necessary due to an asymmetric substitution that occurs when $T = B \to T_2$.

2.6.8 Lemma [Bulletproofing]: If $t \sim_{\succ} s : T$ and $c \sim_{\succ} S : T$ and $c \sim_{\succ} S' : T$, then $\langle c \rangle^{l,l'}\ t \sim_{\succ} \langle S' \Rightarrow \lceil S' \rceil \rangle^{l'}\ \langle \lceil S \rceil \Rightarrow S \rangle^{l}\ s : T$.

Proof: By induction on T. First, observe that either $s \longrightarrow_h^* \Uparrow l''$ or both t and s go to values related at T. In the former case, $\langle S' \Rightarrow \lceil S' \rceil \rangle^{l'}\ \langle \lceil S \rceil \Rightarrow S \rangle^{l}\ s \longrightarrow_h^* \Uparrow l''$, and everything is related to blame (Lemma 2.6.2). So $t \longrightarrow_{lax}{}^*\ v$, $s \longrightarrow_h^*\ w$, and $v \approx_{\succ} w : T$.

$\underline{T = B}$: So $c = \{x{:}B \mid t_1\}$ and $S = \{x{:}B \mid s_1\}$ and $S' = \{x{:}B \mid s_2\}$. By Lemma 2.6.5 we have $\langle c \rangle^{l,l'}\ t \sim_{\succ} \langle \lceil S \rceil \Rightarrow S \rangle^{l}\ s : B$. By Lemma 2.6.4 we can add the extra, trivial cast.

$\underline{T = T_1 \to T_2}$: We know that $c = x{:}c_1 \mapsto c_2$, $S = x{:}S_1 \to S_2$ and $S' = x{:}S_1' \to S_2'$. Let $t' \sim_{\succ} s' : T_1$. By Lemma 2.6.2, we only need to consider the case where $t' \longrightarrow_{lax}{}^*\ v'$ and $s' \longrightarrow_h^*\ w'$—if $s' \longrightarrow_h^* \Uparrow l''$ we are done.

On the λ_{C} side, $(\langle c \rangle^{l,l'}\ t)\ t' \longrightarrow_{lax}{}^*\ \langle c_2\{x := v'\} \rangle^{l,l'}\ (v\ (\langle c_1 \rangle^{l',l}\ v'))$. In λ_{H}, we can see

$$(\langle S' \Rightarrow \lceil S' \rceil \rangle^{l'}\ \langle \lceil S \rceil \Rightarrow S \rangle^{l}\ s)\ s' \longrightarrow_h^*$$
$$\langle S_2'\{x := \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\ w'\} \Rightarrow \lceil S_2' \rceil \rangle^{l'}\ ((\langle \lceil S \rceil \Rightarrow S \rangle^{l}\ w)\ (\langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\ w'))$$

We cannot determine where the redex is until we know the shape of T_1—does the negative argument cast step to an active check, or do we decompose the positive cast?

– $T_1 = B$.

By Lemma 2.6.5 and $c_1 \sim_{\succ} S_1' : B$, we know that $\langle c_1 \rangle^{l',l}\ v' \sim_{\succ} \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\ w' : B$. The λ_{H} term goes to blame or both terms go to the same value, $v' = w' = k \in \mathcal{K}_B$. In the former case, the entire λ_{H} term goes to blame and we are done by Lemma 2.6.2. So suppose $\langle c_1 \rangle^{l',l}\ k \longrightarrow_{lax}{}^*\ k$ and $\langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\ w' \longrightarrow_h^*\ k$. Now the terms evaluate like so:

$$\langle c_2\{x := v'\} \rangle^{l,l'}\ (v\ (\langle c_1 \rangle^{l',l}\ v')) \longrightarrow_{lax}{}^*\ \langle c_2\{x := k\} \rangle^{l,l'}\ (v\ k)$$

$$\begin{array}{l}\langle S_2'\{x := \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\ w'\} \Rightarrow \lceil S_2' \rceil \rangle^{l'} \\ \qquad ((\langle \lceil S \rceil \Rightarrow S \rangle^{l}\ w)\ (\langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\ w')) \longrightarrow_h^* \\ \langle S_2'\{x := \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\ w'\} \Rightarrow \lceil S_2' \rceil \rangle^{l'} \\ \qquad \langle \lceil S_2 \rceil \Rightarrow S_2\{x := k\} \rangle^{l}\ (w\ (\langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^{l}\ k))\end{array}$$

By Lemma 2.6.4, $k \sim_{\succ} \langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^{l}\ k : B$, so $v\ k \sim_{\succ} w\ (\langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^{l}\ k) : T_2$.

Noting that $k \sim_{\succ} k : B$ and $k \sim_{\succ} \langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l}\ k : B$, we can see that $c_2\{x := k\} \sim_{\succ} S_2\{x := k\} : T_2$ and $c_2\{x := k\} \sim_{\succ} S_2'\{x := \langle \lceil S_1 \rceil \Rightarrow S_1 \rangle^{l}\ k\} : T_2$. Now the IH shows that $\langle c_2\{x := k\} \rangle^{l,l'}\ (v\ k) \sim_{\succ} \langle S_2'\{x := \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\ w'\} \Rightarrow \lceil S_2' \rceil \rangle^{l'}\ \langle \lceil S_2 \rceil \Rightarrow S_2\{x := k\} \rangle^{l}\ (w\ (\langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^{l}\ k)) : T_2$, and we conclude the case with expansion (Lemma 2.6.1).

– $T_1 = T_{11} \to T_{12}$. We can continue with an application of F_CDecomp in λ_{H} and find:

$$\begin{array}{l}\langle S_2'\{x := \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\ w'\} \Rightarrow \lceil S_2' \rceil \rangle^{l'} \\ \quad ((\langle \lceil S \rceil \Rightarrow S \rangle^{l}\ w)\ (\langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\ w')) \longrightarrow_h^* \\ \langle S_2'\{x := \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\ w'\} \Rightarrow \lceil S_2' \rceil \rangle^{l'} \\ \quad \langle \lceil S_2 \rceil \Rightarrow S_2\{x := \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\ w'\} \rangle^{l} \\ \quad\quad (w\ (\langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^{l}\ \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\ w'))\end{array}$$

By the IH, $\langle c_1 \rangle^{l',l}\ v' \sim_\succ \langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^{l}\ \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\ w' : T_1$. By assumption, the results of applying v and w to these values correspond. (And they *are* values, since function contracts/casts applied to values are values.)

We know $c_1 \sim_\succ S_1' : T_1$, and by Lemma 2.6.7 $T_1\uparrow \sim_\succ \lceil S_1' \rceil : T_1$. Since $v' \sim_\succ w' : T_1$, Lemma 2.6.6 shows $v' \sim_\succ \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\ w' : T_1$. This lets us see that $c_2\{x := v'\} \sim_\succ S_2'\{x := \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\ w'\} : T_2$ and $c_2\{x := v'\} \sim_\succ S_2\{x := \langle \lceil S_1' \rceil \Rightarrow S_1' \rangle^{l'}\ w'\} : T_2$. Now the IH and expansion (Lemma 2.6.1) complete the proof. □ □

Having characterized how contracts and pairs of related casts relate, we show that translated terms correspond to their sources.

2.6.9 Theorem [Behavioral correspondence]: If $\vdash \Gamma$, then:

1. If $\phi(\Gamma \vdash t : T) = s$ then $\Gamma \vdash t \sim_\succ s : T$.
2. If $\phi(\Gamma \vdash^{l,l'} c : T) = S$ then $\Gamma \vdash c \sim_\succ S : T$.

Proof: We simultaneously show both properties by induction on the depth of ϕ's recursion. To show $\Gamma \vdash t \sim_\succ s : T$, let $\Gamma \models \delta$—we will show $\delta_1(t) \sim_\succ \delta_2(s) : T$.

The proof proceeds by case analysis on the final rule of the translated typing and well-formedness derivations. □

We find a weak corollary: $\phi(\Gamma \vdash t : B) \longrightarrow_h^* k$ implies $t \longrightarrow_{lax}{}^* k$: if the λ_{H} term does *not* go to blame, then the original λ_{C} term must go to the same constant.

We can also show type preservation for terms not containing active checks. (We do not know that translated active checks are well typed, because Theorem 2.6.9 is not strong enough to preserve the implication judgment. We only expect these checks to occur at runtime, so this is good enough: ϕ preserves the types of source programs.)

2.6.10 Theorem [Type preservation for ϕ]: For programs without active checks, if $\phi(\vdash \Gamma) = \Delta$, then:

1. $\vdash \Delta$.
2. $\Delta \vdash \phi(\Gamma \vdash t : T) : \lceil T \rceil$.
3. $\Delta \vdash \phi(\Gamma \vdash^{l,l'} c : T)$.

Proof: We prove all three properties simultaneously, by induction on the depth of ϕ's recursion.

The proof is by cases on the λ_{C} context well-formedness/term typing/contract well-formedness derivations, which determine the branch of ϕ taken. $\square$

To see that the ϕ in Figure 2.12 does not give us exact blame, let us look at two counterexamples; in both cases, a lax λ_{C} term goes to a value while its translation goes to blame. In the first example, blame is raised in λ_{H} due to bulletproofing. In the second, blame is raised due to the extra cast from the translation of T_VarC. In both examples, the contracts are *abusive*: higher-order contracts where the codomain places a contradictory requirement on the domain. For the first counterexample, let

$$\begin{array}{rcl} c & = & f{:}(x{:}\{x{:}\mathsf{Int} \mid \mathsf{true}\} \mapsto \{y{:}\mathsf{Int} \mid \mathsf{nonzero}\ y\}) \mapsto \{z{:}\mathsf{Int} \mid f\ 0 = 0\} \\ S_1 & = & x{:}\{x{:}\mathsf{Int} \mid \mathsf{true}\} \to \{y{:}\mathsf{Int} \mid \mathsf{nonzero}\ y\} \\ S & = & \phi(\emptyset \vdash^{l,l} c : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}) \\ & = & f{:}S_1 \to \{z{:}\mathsf{Int} \mid (\langle S_1 \Rightarrow \lceil S_1 \rceil \rangle^l\ f)\ 0 = 0\}. \end{array}$$

Here, the contradiction comes when the codomain requires that $f\ 0$ yield 0, but f's contract says it will return a non-zero value. We find $\langle c \rangle^{l,l}\ (\lambda f.0)\ (\lambda x.0) \longrightarrow_{lax}{}^* 0$ but

$$(\lambda x{:}\lceil c \rceil.\ \langle S \Rightarrow \lceil S \rceil \rangle^l\ (\langle \lceil S \rceil \Rightarrow S \rangle^l\ x))\ (\lambda f.0)\ (\lambda x.0) \longrightarrow_h^* \Uparrow l.$$

For the second counterexample, let

$$\begin{array}{rcl} c' & = & f{:}(x{:}\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \mapsto \{y{:}\mathsf{Int} \mid \mathsf{true}\}) \mapsto \{z{:}\mathsf{Int} \mid f\ 0 = 0\} \\ S_1' & = & x{:}\{x{:}\mathsf{Int} \mid \mathsf{nonzero}\ x\} \to \{y{:}\mathsf{Int} \mid \mathsf{true}\} \\ S' & = & \phi(\emptyset \vdash^{l,l} c' : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}) \\ & = & f{:}S_1' \to \{z{:}\mathsf{Int} \mid (\langle S_1' \Rightarrow \lceil S_1' \rceil \rangle^l\ f)\ 0 = 0\}. \end{array}$$

This time, the contradiction comes from the codomain applying f to 0, while the domain contract requires that f's input be non-zero. We find $\langle c' \rangle^{l,l}\ (\lambda f.0)\ (\lambda x.0) \longrightarrow_{lax}{}^* 0$ but

$$(\lambda x{:}\lceil c' \rceil.\ \langle S' \Rightarrow \lceil c' \rceil \rangle^l\ (\langle \lceil S \rceil \Rightarrow \lceil c' \rceil \rangle^l\ x))\ (\lambda f.0)\ (\lambda x.0) \longrightarrow_h^* \Uparrow l.$$

The extra casts that ϕ inserts are all necessary—none can be removed. So while variations on this ϕ are possible, they can only add more casts, which won't resolve the problem that λ_{H} blames *more*.

2.6.2 Translating λ_{H} to picky λ_{C}

Terms in λ_{H} and their ψ-images in lax λ_{C} correspond exactly, as shown Section 2.5.2. When we change the operational semantics of λ_{C} to be *picky*, however, $\psi(s)$ blames (strictly) more often than s. Nevertheless, we can show an inexact correspondence, as we did for ϕ and lax λ_{C} in Section 2.6.1. I use a logical relation $\sim_\prec$ for ψ into picky λ_{C} (Figure 2.18). Here I have reversed the asymmetry: picky λ_{C} may blame

Value correspondence $\boxed{v \approx_{\prec} w : T}$

$$k \approx_{\prec} k : B \iff k \in \mathcal{K}_B$$

$$v \approx_{\prec} w : T_1 \to T_2 \iff \forall t \sim_{\prec} s : T_1.\ v\ t \sim_{\prec} w\ s : T_2$$

Term correspondence $\boxed{t \sim_{\prec} s : T}$

$$t \sim_{\prec} s : T \iff t \longrightarrow_{picky}{}^{*} \Uparrow l \vee t \longrightarrow_{picky}{}^{*} v \wedge s \longrightarrow_{h}^{*} w \wedge v \approx_{\prec} w : T$$

Contract/type correspondence $\boxed{c \sim_{\prec} S_1 \Rightarrow S_2 : T}$

$$\{x{:}B \mid t\} \sim_{\prec} \{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid s_2\} : B \iff \forall k \in \mathcal{K}_B.\ t\{x := k\} \sim_{\prec} s_2\{x := k\} : \mathsf{Bool}$$

$$x{:}c_1 \mapsto c_2 \sim_{\prec} x{:}S_{11} \to S_{12} \Rightarrow x{:}S_{21} \to S_{22} : T_1 \to T_2 \iff c_1 \sim_{\prec} S_{21} \Rightarrow S_{11} : T_1 \wedge \forall l.\forall t \sim_{\prec} s : T_1.\ c_2\{x := t\} \sim_{\prec} S_{12}\{x := \langle S_{21} \Rightarrow S_{11}\rangle^{l}\ s\} \Rightarrow S_{22}\{x := s\} : T_2$$

Dual closing substitutions

$$\Gamma \models \delta \iff \forall x \in \mathrm{dom}(\Gamma).\ \delta_1(x) \sim_{\prec} \delta_2(x) : \lfloor \Gamma(x) \rfloor$$

Lifted to open terms

$$\Gamma \vdash t \sim_{\prec} s : T \iff \forall \delta.\ (\Gamma \models \delta \ \text{ implies } \ \delta_1(t) \sim_{\prec} \delta_2(s) : T)$$
$$\Gamma \vdash c \sim_{\prec} S_1 \Rightarrow S_2 : T \iff \forall \delta.\ (\Gamma \models \delta \ \text{ implies } \ \delta_1(c) \sim_{\prec} \delta_2(S_1) \Rightarrow \delta_2(S) : T)$$

Figure 2.18: Blame-inexact correspondence for ψ into picky λ_{C}

more than λ_{H}. The proof follows the same general pattern: we first show that it is safe to add extra contract checks, then we show that contracts and casts correspond (inexactly), then the correspondence for well-typed terms. We can also show type preservation for source programs (excluding active checks).

2.6.11 Lemma [Expansion and contraction]: If $t \longrightarrow_{picky}{}^{*} t'$ and $s \longrightarrow_{h}^{*} s'$ then $t \sim_{\prec} s : T$ iff $t' \sim_{\prec} s' : T$.

2.6.12 Lemma [Blame corresponds to everything]: For all T, $\Uparrow l \sim_{\prec} s : T$.

2.6.13 Lemma [Constants self-correspond]: For all k, $k \approx_{\prec} k : \mathrm{ty_c}(k)$.

Proof: By induction on $\mathrm{ty_c}(k)$, recalling that constants are first order. □

As a corollary of Lemma 2.6.12 and Lemma 2.6.11, if a picky λ_{C} term evaluates to blame, then it corresponds to any λ_{H} term. This will be used extensively in the proofs below, as it allows us to eliminate many cases.

2.6.14 Lemma [Extra contracts]: If $t \sim_{\prec} s : T$ and $c \sim_{\prec} S_1 \Rightarrow S_2 : T$ then $\langle c \rangle^{l,l'}\ t \sim_{\prec} s : T$.

Proof: By induction on T. If $t \longrightarrow_{picky}{}^{*} \Uparrow l''$ we are done, so let $t \longrightarrow_{picky}{}^{*} v$ and $s \longrightarrow_{h}^{*} w$ such that $v \approx_{\prec} w : T$.

$\underline{T = B}$: So $c = \{x{:}B \mid t_2\}$ and $S_2 = \{x{:}B \mid s_2\}$. Moreover, $v = w = k \in \mathcal{K}_B$, since those are the only corresponding values at B.

We can step and see $\langle c \rangle^{l,l'}\ t \longrightarrow_{picky}{}^{*} \langle c, t_2\{x := k\}, k \rangle^{l}$. We know that $t_2\{x := k\} \sim_{\prec} s_2\{x := k\} : \mathsf{Bool}$. There are two possibilities: either $t_2\{x := k\} \longrightarrow_{picky}{}^{*} \Uparrow l''$ or both terms go to corresponding Bools. In the former case, the whole λ_{C} term goes to blame and we are done by Lemma 2.6.12. If both go to false, then the outer λ_{C} term evaluates to $\Uparrow l$ and we are done by Lemma 2.6.12 again. If both go to true, then both outer terms go to k, and $k \approx_{\prec} k : B$.

$\underline{T = T_1 \to T_2}$: So $c = x{:}c_1 \mapsto c_2$ and $S_1 = x{:}S_{11} \to S_{12}$ and $S_2 = x{:}S_{21} \to S_{22}$. Let $t' \sim_{\prec} s' : T_1$. If $t' \longrightarrow_{picky}{}^{*} \Uparrow l''$ we are done by Lemm 2.6.12, so let $t' \longrightarrow_{picky}{}^{*} v'$ and $s' \longrightarrow_{h}^{*} w'$, where $v' \approx_{\prec} w' : T_1$. We want to prove $(\langle c \rangle^{l,l'}\ t)\ t' \sim_{\prec} s\ s' : T_2$, which is true iff:

$$\langle c_2\{x := \langle c_1 \rangle^{l',l}\ v'\} \rangle^{l,l'}\ (v\ (\langle c_1 \rangle^{l',l}\ v')) \sim_{\prec} w\ w' : T_2$$

By the IH on $v \sim_{\prec} w' : T_1$ and $c_1 \sim_{\prec} S_{21} \Rightarrow S_{11} : T_1$, we have $\langle c_1 \rangle^{l',l}\ v' \sim_{\prec} w' : T_1$. By definition, applying v and w yields related terms at T_2. Since $\langle c_1 \rangle^{l',l}\ v' \sim_{\prec} w' : T_1$, we have $c_2\{x := \langle c_1 \rangle^{l',l}\ v'\} \sim_{\prec} S_{12}\{x := \langle S_{21} \Rightarrow S_{11} \rangle^{l''}\ w'\} \Rightarrow S_{22}\{x := w'\} : T_2$. We can now apply the IH and see:

$$\langle c_2\{x := \langle c_1 \rangle^{l',l}\ v'\} \rangle^{l,l'}\ (v\ (\langle c_1 \rangle^{l',l}\ v')) \sim_{\prec} w\ w' : T_2 \quad \square$$

□

2.6.15 Lemma [Contract/cast correspondence]: If $c \sim_\prec S_1 \Rightarrow S_2 : T$ and $t \sim_\prec s : T$ then $\langle c \rangle^{l,l'}\ t \sim_\prec \langle S_1 \Rightarrow S_2 \rangle^{l''}\ s : T$.

Proof: By induction on T. We reason via expansion (Lemma 2.6.11), showing that the initial terms reduce to corresponding terms.

$\underline{T = B}$: So $c = \{x{:}B \mid t_1\}$, $S_1 = \{x{:}B \mid s_1\}$, and $S_2 = \{x{:}B \mid s_2\}$. Since $t \sim_\prec s : B$, we know that they either both reduce to $k \in \mathcal{K}_B$ or $t \longrightarrow_{picky}{}^* \Uparrow l'$. If the latter is the case, we are done. So suppose $t \longrightarrow_{picky}{}^* k$ along with $s \longrightarrow_h^* k$.

We can step our terms into active checks as follows, then:

$$\begin{array}{rcl} \langle \{x{:}B \mid t_1\} \rangle^{l,l}\ t & \longrightarrow_{picky}{}^* & \langle \{x{:}B \mid t_1\}, t_1\{x := k\}, k \rangle^l \\ \langle \{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid s_2\} \rangle^l\ s & \longrightarrow_h^* & \langle \{x{:}B \mid s_2\}, s_2\{x := k\}, k \rangle^l \end{array}$$

By the contract/cast correspondence, we know that $t_1\{x := k\} \sim_\prec s_2\{x := k\} : \mathsf{Bool}$, so either $t_1\{x := k\}$ goes to blame or both terms go to a Bool together. In the former case, the outer λ_{C} term goes to blame and we are done by Lemma 2.6.12.. If they go to false, then both the active check goes to $\Uparrow l$ and we are done, again by Lemma 2.6.12. Finally, if they both go to true, then both terms will evaluate to $k \in \mathcal{K}_B$, and $k \approx_\prec k : B$.

$\underline{T = T_1 \to T_2}$: $c = x{:}c_1 \mapsto c_2$, $S_1 = x{:}S_{11} \to S_{12}$, and $S_2 = x{:}S_{21} \to S_{22}$. We know by inversion of the contract/cast relation that $c_1 \sim_\prec S_{21} \Rightarrow S_{11} : T_1$ and that for all l'' and $t \sim_\prec s : T_1$, $c_2\{x := t\} \sim_\prec S_{12}\{x := \langle S_{21} \Rightarrow S_{11} \rangle^{l''}\ s\} \Rightarrow S_{22}\{x := s\} : T_2$. We want to prove that $\langle c \rangle^{l,l} \sim_\prec \langle S_1 \Rightarrow S_2 \rangle^l\ s : T_1 \to T_2$. First, we can assume $t \longrightarrow_{picky}{}^* v$ and $s \longrightarrow_h^* w$ where $v \sim_\prec w : T_1 \to T_2$—if not, both the contracted term goes to blame and we are done by Lemma 2.6.12.

We show that the decomposition of the contract and cast terms correspond for all inputs. Let $t' \sim_\prec s' : T_1$. Again, we can assume that they reduce to $v' \sim_\prec w' : T_1$, or else we are done by blame lifting in λ_{C}. On the λ_{C} side, we have

$$(\langle c \rangle^{l,l'}\ t)\ t' \longrightarrow_{picky}{}^* \langle c_2\{x := \langle c_1 \rangle^{l',l}\ v'\} \rangle^{l,l'}\ (v\ (\langle c_1 \rangle^{l',l}\ v'))$$

In λ_{H}, we find

$$\begin{array}{l} (\langle S_1 \Rightarrow S_2 \rangle^{l''}\ s)\ s' \longrightarrow_h^* \\ \langle S_{12}\{x := \langle S_{21} \Rightarrow S_{11} \rangle^{l''}\ w\} \Rightarrow S_{22}\{x := w'\} \rangle^{l''}\ (w\ (\langle S_{21} \Rightarrow S_{11} \rangle^{l''}\ w')) \end{array}$$

By the IH, we know that $\langle c_1 \rangle^{l',l}\ v' \sim_\prec \langle S_{21} \Rightarrow S_{11} \rangle^{l''}\ w' : T_1$. Since $v \sim_\prec w : T_1 \to T_2$, we have $v\ (\langle c_1 \rangle^{l',l}\ v') \sim_\prec w\ (\langle S_{21} \Rightarrow S_{11} \rangle^{l''}\ w') : T_2$. By Lemma 2.6.14, $\langle c_1 \rangle^{l',l}\ v' \sim_\prec w' : T_1$. We can then see that $c_2\{x := \langle c_1 \rangle^{l',l}\ v'\} \sim_\prec S_{12}\{x := \langle S_{21} \Rightarrow S_{11} \rangle^{l''}\ w'\} \Rightarrow S_{22}\{x := w'\} : T_2$. By the IH, we therefore have

$$\begin{array}{l} \langle c_2\{x := \langle c_1 \rangle^{l',l}\ v'\} \rangle^{l,l'}\ (v\ (\langle c_1 \rangle^{l',l}\ v')) \sim_\prec \\ \langle S_{12}\{x := \langle S_{21} \Rightarrow S_{11} \rangle^{l''}\ w'\} \Rightarrow S_{22}\{x := w'\} \rangle^{l''}\ w\ (\langle S_{21} \Rightarrow S_{11} \rangle^{l''}\ w') : T_2 \quad \square \end{array}$$

$\square$

2.6.16 Theorem [Behavioral correspondence]:

1. If $\Delta \vdash s : S$ then $\lfloor\Delta\rfloor \vdash \psi(s) \sim_{\prec} s : \lfloor S\rfloor$.
2. If $\Delta \vdash S_1$ and $\Delta \vdash S_2$, where $\lfloor S_1\rfloor = \lfloor S_2\rfloor = \lfloor S\rfloor$, then $\lfloor\Delta\rfloor \vdash \psi^l(S_1, S_2) \sim_{\prec} S_1 \Rightarrow S_2 : \lfloor S\rfloor$.

Proof: By an induction similar to the proof Theorem 2.5.18. □ □

2.6.17 Theorem [Type preservation for ψ]: For programs with no active checks, if $\vdash \Delta$, then:

1. If $\Delta \vdash s : S$ then $\lfloor\Delta\rfloor \vdash \psi(s) : \lfloor S\rfloor$.
2. If $\Delta \vdash S_1$, $\Delta \vdash S_2$, where $\lfloor S_1\rfloor = \lfloor S_2\rfloor = T$, then $\lfloor\Delta\rfloor \vdash^{l,l'} \psi^l(S_1, S_2) : T$.

Proof: By an induction similar to the proof Theorem 2.5.20. □ □

Here is an example where a λ_{H} term reduces to a value while its ψ-image in picky λ_{C} term reduces to blame. As before, this counterexample uses an *abusive* contract: a higher-order contract where the codomain puts a contradictory requirement on the domain. Here, the contradiction is that f claims to return a non-zero value, but the codomain requires that it return 0.

$$\begin{array}{rcl}
S_1 & = & f{:}S_{11} \to S_{12} \\
 & = & f{:}(x{:}\lceil\mathsf{Int}\rceil \to \{y{:}\mathsf{Int} \mid \mathsf{nonzero}\ y\}) \to \lceil\mathsf{Int}\rceil \\
S_2 & = & f{:}S_{21} \to S_{22} \\
 & = & f{:}(x{:}\lceil\mathsf{Int}\rceil \to \lceil\mathsf{Int}\rceil) \to \{z{:}\mathsf{Int} \mid f\ 0 = 0\} \\
c & = & \psi^l(S_1, S_2) \\
 & = & f{:}\psi^l(S_{21}, S_{11}) \mapsto \psi^l(S_{12}\{f := \langle S_{21} \Rightarrow S_{11}\rangle^l\ f\}, S_{22}) \\
 & = & f{:}(x{:}\{x{:}\mathsf{Int} \mid \mathsf{true}\} \mapsto \{y{:}\mathsf{Int} \mid \mathsf{nonzero}\ y\}) \mapsto \{z{:}\mathsf{Int} \mid f\ 0 = 0\}
\end{array}$$

Let $w = (\lambda f{:}(x{:}\{x{:}\mathsf{Int} \mid \mathsf{true}\} \to \{y{:}\mathsf{Int} \mid \mathsf{nonzero}\ y\}).\ 0)$ and $w' = (\lambda x{:}\{x{:}\mathsf{Int} \mid \mathsf{true}\}.\ 0)$. The term is well typed: we can show $\emptyset \vdash w : S_1$ and $\emptyset \vdash w' : S_{21}$. Therefore $\emptyset \vdash (\langle S_1 \Rightarrow S_2\rangle^l\ w)\ w' : S_{22}\{f := w'\}$. Translating, we find

$$\psi((\langle S_1 \Rightarrow S_2\rangle^l\ w)\ w') = (\langle\psi^l(S_1, S_2)\rangle^{l,l}\ \psi(w))\ \psi(w') = (\langle c\rangle^{l,l}\ \lambda f{:}\mathsf{Int}.\ 0)\ \lambda x{:}\mathsf{Int}.\ 0.$$

On the one hand $(\langle S_1 \Rightarrow S_2\rangle^l\ w)\ w' \longrightarrow_h^* 0$, while $(\langle c\rangle^{l,l}\ \lambda f{:}\mathsf{Int}.\ 0)\ \lambda x{:}\mathsf{Int}.\ 0 \longrightarrow_{picky}^* \Uparrow l$. This means we cannot hope to use ψ as an exact correspondence between λ_{H} and picky λ_{C}. (Removing the extra cast ψ inserts into S_{12} does not affect the example, since ψ ignores S_{12} here.) For example,

$$\psi^l(\{z{:}\mathsf{Int} \mid \mathsf{true}\}\{f := \langle S_{21} \Rightarrow S_{11}\rangle^l\ f\}, \{z{:}\mathsf{Int} \mid f\ 0 = 0\}) = \{x{:}B \mid \psi(f\ 0 = 0)\}.$$

2.6.3 Alternative calculi

There are three alternative calculi I have not considered here: indy λ_{C} [22], superpicky λ_{H}, and nonterminating calculi. I describe them in detail below, but I leave them as future work.

Dimoulas et al. [22] add a third blame label to λ_{C}, representing the contract itself; I write it here as a subscript. They accordingly change the picky E_CDecomp rule:

$$(\langle x{:}c_1 \mapsto c_2 \rangle^{l,l'}_{l''}\ v_1)\ v_2 \longrightarrow_{indy} \langle c_2\{x := \langle c_1 \rangle^{l'',l}_{l''}\ v_2\}\rangle^{l,l'}_{l''}\ (v_1\ (\langle c_1 \rangle^{l',l}_{l''}\ v_2))$$

In the substitution in the codomain, note that the blame label on the domain contract uses the contract's blame label l''. The intuition here is that any problem arising in c_2 is in the contract's context (label l''), not the original negative context (label l'). I conjecture (but have not proven) that indy λ_{C} is in the same position on the axis of blame as picky λ_{C}. We should only need to change the labels on the contracts ϕ inserts to have an exact correspondence; however, ψ will remain inexact.

Superpicky λ_{H} reworks the F_CDecomp rule in an attempt to harmonize λ_{H} and picky λ_{C} semantics:[7]

$$\begin{array}{l}(\langle x{:}S_{11} \to S_{12} \Rightarrow x{:}S_{21} \to S_{22} \rangle^{l}\ w_1)\ w_2 \rightsquigarrow_h \\ \langle S_{12}\{x := \langle S_{21} \Rightarrow S_{11} \rangle^{l}\ w_2\} \Rightarrow S_{22}\{x := \langle S_{11} \Rightarrow S_{21} \rangle^{l}\ (\langle S_{21} \Rightarrow S_{11} \rangle^{l}\ w_2)\}\rangle^{l} \\ \qquad (w_1\ (\langle S_{21} \Rightarrow S_{11} \rangle^{l}\ w_2))\end{array}$$

This seems to resolve the problem with ψ into picky λ_{C}, but it poses problems in the proof of semantic type soundness for λ_{H}: how do $S_{22}\{x := w_2\}$ and $S_{22}\{x := \langle S_{11} \Rightarrow S_{21} \rangle^{l}\ (\langle S_{21} \Rightarrow S_{11} \rangle^{l}\ w_2)\}$ relate?

Finally, I have been careful to ensure that all of the calculi are strongly normalizing. I do not believe this to be essential, though I would have to change the logical relations—λ_{H}'s type semantics and the correspondences—to account for nontermination. Step-indexing [2] should suffice.

2.7 Conclusion

The main contributions of the work in this chapter are (1) the dependent translations ϕ and ψ and their properties, and (2) the formulation and metatheory of dependent λ_{H}. (Dependent λ_{C} is not a contribution on its own: many similar systems have been studied, and in any case its properties are simple.) The nondependent part of our ϕ translation essentially coincides with the one studied by Gronski and Flanagan [36], and our behavioral correspondence theorem is essentially the same as theirs (though our proof is substantially different—and more detailed). Our ψ translation completes their story for the nondependent case, establishing a tight connection between the systems. The full dependent forms of ϕ and ψ studied in this chapter are novel, as

[7] This idea is due to Jeremy Siek (personal communication, January 2010).

is the observation that the correspondence between the latent and manifest worlds is more problematic in this setting.

We can faithfully encode dependent λ_{H} into λ_{C}—the behavioral correspondence is tight. λ_{H}'s F_CDecomp rule forces us to accept a weaker behavioral correspondence when encoding λ_{C} into λ_{H}, so I conclude that the manifest and latent approaches are *not* equivalent in the dependent case. I do find, however, that the two approaches are entirely inter-encodable in the nondependent restriction.

Chapter 3

Polymorphic manifest contracts

> And even the Abstract Entities
> Circumambulate her charm;
> But our lot crawls between dry ribs
> To keep our metaphysics warm.
>
> ---
>
> *Whispers of Immortality*
> T.S. Eliot

Contracts are particularly useful at interfaces, allowing programmers to specify expectations and guarantees in code, rather than merely in comments. For example, consider an abstract datatype (ADT) modeling the natural numbers:

$$\mathsf{NAT} : \exists\alpha.(\mathsf{zero} : \alpha) \times (\mathsf{succ} : (\alpha \to \alpha)) \times (\mathsf{iszero} : (\alpha \to \mathsf{Bool})) \times (\mathsf{pred} : (\alpha \to \alpha))$$

The type NAT (with its existential quantification and products) can be encoded straightforwardly in System F. It is an *abstract* datatype because the actual representation of α is hidden: users of NAT interact with it through the constructors and operations provided. The zero constructor represents 0; the succ constructor takes a natural and produces its successor. The predicate iszero determines whether a given natural is zero. The pred operation takes a natural number and returns its predecessor. There are many ways to define an ADT matching NAT's signature, but we can suppose here that NAT is implemented using the standard Church encoding

$$\alpha = \forall\beta.\beta \to (\beta \to \beta) \to \beta.$$

Under the standard definition, $\mathsf{pred}\,\mathsf{zero} \longrightarrow^* \mathsf{zero}$, even though the mathematical natural-number predecessor operation isn't defined for zero. That is, the pred function is only *mostly* correct: $\mathsf{pred}\,n$ will return the same result as the mathematical natural-number predecessor so long as $n \neq 0$. Using contracts, we can specify this constraint explicitly, as we did in the introduction (Section 1.2):

$$\begin{aligned}\mathsf{NAT} : \exists\alpha.\ & (\mathsf{zero} : \alpha) \times (\mathsf{succ} : (\alpha \to \alpha)) \times (\mathsf{iszero} : (\alpha \to \mathsf{Bool})) \times \\ & (\mathsf{pred} : \{x{:}\alpha \mid \mathsf{not}\,(\mathsf{iszero}\,x)\} \to \alpha)\end{aligned}$$

In addition to requirements, we can also express guarantees. For example, the interface could indicate that successors are non-zero by giving succ the type $\alpha \to \{x{:}\alpha \mid \mathsf{not}\,(\mathsf{iszero}\,x)\}$. Both of these encodings rely on dependent types: we need to be able to refer back to iszero in the contracts for pred and succ. Fortunately, dependent functions can encode dependent sums:

$$\begin{array}{rcl}(x : T_1) \times T_2 & = & \forall\alpha.(x{:}T_1 \to T_2 \to \alpha) \to \alpha \\ (e_1, e_2) & = & \Lambda\alpha.\ \lambda f{:}(x{:}T_1 \to T_2 \to \alpha).\ f\ e_1\ e_2\end{array}$$

Finally, note that to put a refinement type on the codomain of succ, we would have to rearrange the dependent sum above so succ came after iszero.

This chapter presents a new core calculus for contracts and polymorphism, $\mathrm{F_H}$ (so named since it extends the manifest-contract core calculus, λ_{H}, with the type abstractions/impredicative polymorphism of System F). The example I've just given puts contracts inside types, so it is manifest—a sensible choice for combining contracts and abstract datatypes. Abstract datatypes already use the type system to mediate access to abstractions; manifest contracts allow types to exercise a still finer grained control.

$\mathrm{F_H}$ is a polymorphic manifest contract calculus. In designing $\mathrm{F_H}$, I take a new metatheoretical approach, since earlier manifest calculi don't scale up to polymorphism easily—for two reasons.

First, the metatheory for existing calculi has been too complicated to extend easily [34, 44]. Previous approaches proved semantic type soundness, using denotational techniques. The denotational semantics used are harder to scale than standard syntactic methods (i.e., progress and preservation). I explain the complexity in detail in related work (Section 5.2), but, briefly, the problem is subtyping. In both Greenberg et al. and Knowles and Flanagan, subtyping between refinement types is what ultimately requires denotational semantics. The denotational semantics is used to avoid a dangerous circularity in the mutually recursive definition of the typing and subtyping judgments. The heart of my new technique is to replace subtyping with a simpler conversion relation, which allows for a simpler, more scalable, syntactic metatheory. Eliminating subtyping doesn't lose useful reasoning principles: in Section 3.4, I define subtyping *post facto* and recover an "upcast" lemma from Knowles and Flanagan [44] showing that casts from subtypes to supertypes never fail.

Second, look again at our NAT abstract datatype. If we pare away the ADT's type abstraction to the underlying Church encoding, we find

$$\{x{:}\alpha \mid \mathsf{not}\,(\mathsf{iszero}\,x)\} = \{x{:}\forall\beta.\beta \to (\beta \to \beta) \to \beta \mid \mathsf{not}\,(\mathsf{iszero}\,x)\}.$$

If we can put contracts on type variables, then the contracts on our ADTs will need to allow refinements on function and forall types. These so-called "general refinements" aren't possible in existing manifest calculi, which restrict refinements to base types like Bool and Int. More specifically, earlier metatheory retains type preservation

by assigning constants most-specific types, but this approach doesn't scale to refinements of functions. My new metatheory for F_H takes a different approach to type preservation that supports general refinements. (For more detail, see the discussion of T_Exact in Section 3.2.)

In Section 3.1, I develop longer and more detailed examples. I describe F_H and prove type soundness in Section 3.2. I prove parametricity in Section 3.3 and the upcast lemma in Section 3.4.

3.1 Examples

In this section, I offer some longer examples, in order to develop an intuition for how contracts work in general and with polymorphism in particular. I introduce the mechanisms used to enforce contracts first; then I look at the NAT datatype in greater depth; I offer a "library as a language" as a final example.

Like other manifest calculi, F_H uses casts to dynamically enforce contracts. Applying a *cast* $\langle T_1 \Rightarrow T_2 \rangle^l$ to a value of type T_1 will ensure that the value behaves like a T_2. I call T_1 the source type and T_2 the target type. The l superscript is a *blame label*, used to differentiate between different casts and identify the source of failures. These failures are indicated by *blame*, an uncatchable exception with a blame label attached; I write this exception $\Uparrow l$ and pronounce it "blame l".

At base types, casts do one of two things: they either return the value they were applied to, or "raise blame". For example, consider a cast from integers Int to positive integers, $\{x{:}\mathsf{Int} \mid x > 0\}$. I write this cast $\langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\} \rangle^l$, picking an arbitrary label l. If we apply this cast to 5, we expect to get 5 back, since $5 > 0$. That is,

$$\langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\} \rangle^l \, 5 \longrightarrow^* 5.$$

On the other hand, suppose we apply the same cast to 0. This cast fails, since 0 is certainly not greater than itself. When the cast fails, it will raise blame with its label:

$$\langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\} \rangle^l \, 0 \longrightarrow^* \Uparrow l.$$

Checking predicate contracts with casts is easy. When we run $\langle T \Rightarrow \{x{:}T \mid e\} \rangle^l \, v$, we simply check if v satisfies the predicate, i.e., whether $e[v/x] \longrightarrow^* \mathsf{true}$. If it does, then the entire application goes to v; if not, then the program aborts, raising $\Uparrow l$. When checking predicate contracts, only the target type matters—the type system will guarantee that whatever value we have is well typed at the source type, i.e., satisfies any predicates it has. For example, we'll always have

$$\langle \{x{:}\mathsf{Int} \mid x > 0\} \Rightarrow \mathsf{Int} \rangle^l \, v \longrightarrow v$$

immediately, for all values v. The application will only be well typed if v actually *is* a positive number, but, operationally, the left-hand side doesn't matter from the perspective of function refinements.

Unlike casts between base types and refinements, casts between function types aren't checked immediately. Instead, casts at function types wrap their argument up, decomposing the function cast into two parts: one cast for the domain and one for the codomain. In this way, checking is deferred until the cast function is called. Suppose we have a function f of type $\mathsf{Int} \to \mathsf{Int}$ and we want to ensure that maps positives to numbers greater than 5, casting it to $\{x{:}\mathsf{Int} \mid x > 0\} \to \{y{:}\mathsf{Int} \mid y > 5\}$. The cast decomposes as follows:

$$\langle \mathsf{Int} \to \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\} \to \{y{:}\mathsf{Int} \mid y > 5\} \rangle^l f \quad \longrightarrow$$
$$\lambda x{:}\{x{:}\mathsf{Int} \mid x > 0\}.\ (\langle \mathsf{Int} \Rightarrow \{y{:}\mathsf{Int} \mid y > 5\} \rangle^l\, (f\, (\langle \{x{:}\mathsf{Int} \mid x > 0\} \Rightarrow \mathsf{Int} \rangle^l\, x))).$$

Both casts in the wrapper have the same blame label as the original cast. Notice that the domains of the function types are treated contravariantly; note the inner term in the wrapped term: $f\, (\langle \{x{:}\mathsf{Int} \mid x > 0\} \Rightarrow \mathsf{Int} \rangle^l\, x)$. In this case, the domain cast $\langle \{x{:}\mathsf{Int} \mid x > 0\} \Rightarrow \mathsf{Int} \rangle^l\, x$ will never fail: every positive integer is also an integer. The codomain cast is covariant: $\langle \mathsf{Int} \Rightarrow \{y{:}\mathsf{Int} \mid y > 5\} \rangle^l$ checks that f returns a number greater than 5. Since not every integer is greater than 5, this cast will fail if f returns a number less than or equal to 5.

Let's consider a few concrete choices of the function f. First, we cast the identity function to $\{x{:}\mathsf{Int} \mid x > 0\} \to \{y{:}\mathsf{Int} \mid y > 5\}$. We can safely apply the cast function to six:

$$\langle \mathsf{Int} \to \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\} \to \{y{:}\mathsf{Int} \mid y > 5\} \rangle^l\, (\lambda x{:}\mathsf{Int}.\ x)\, 6 \longrightarrow^* 6.$$

In general, the identity function does not take positives to numbers greater than five. But when we apply the cast function to 6, which happens to satisfy the codomain contract, no blame is raised. However, when we apply the cast identity function to a value that doesn't satisfy the codomain contract, say 2, blame will be raised:

$$\langle \mathsf{Int} \to \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\} \to \{y{:}\mathsf{Int} \mid y > 5\} \rangle^l\, (\lambda x{:}\mathsf{Int}.\ x)\, 2 \longrightarrow^* \Uparrow l.$$

Contrast this with the sorts of static checks offered by type systems: contract systems raise blame only when a violation is *detected*; type systems are usually conservative, signaling errors when a violation is *possible*.

Some functions will *never* work. If we cast the constant zero function to $\{x{:}\mathsf{Int} \mid x > 0\} \to \{y{:}\mathsf{Int} \mid y > 5\}$, it will raise blame for any value:

$$\langle \mathsf{Int} \to \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\} \to \{y{:}\mathsf{Int} \mid y > 5\} \rangle^l\, (\lambda x{:}\mathsf{Int}.\ 0)\, 6 \longrightarrow^* \Uparrow l.$$

The constant zero function, which *never* returns a value satisfying the codomain contract $\{y{:}\mathsf{Int} \mid y > 5\}$. No matter what value we apply the cast function to, it will always raise blame. Cast at function types are deferred, though: if we never call the cast function, it never has the opportunity to raise blame. Since casts check contracts dynamically, they only detect errors in parts of the program that are explored at runtime.

F_H's type system rules out directly applying a function with domain type $\{x{:}\mathsf{Int} \mid x > 0\}$ to 0. It is an important property of F_H that 0 doesn't have type $\{x{:}\mathsf{Int} \mid x > 0\}$! One can try to cast 0 from Int to $\{x{:}\mathsf{Int} \mid x > 0\}$, but this will always fail:

$$\begin{array}{c}\langle \mathsf{Int} \to \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\} \to \{y{:}\mathsf{Int} \mid y > 5\}\rangle^{l} \\ (\lambda x{:}\mathsf{Int}.\ 0)\,(\langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\}\rangle^{l'}\ 0)\end{array} \quad \longrightarrow^{*} \Uparrow l'$$

Finally, F_H supports dependent function types. For example, the type $x{:}\mathsf{Int} \to \{y{:}\mathsf{Int} \mid y > x\}$ is inhabited by functions over integers which produce results greater than their inputs. Dependent functions allow for very precise specifications. For example, $x{:}\mathsf{Float} \to \{y{:}\mathsf{Float} \mid |y^2 - x| < \epsilon\}$ specifies the square-root function. The exact unwinding rule for dependent functions is slightly subtle—see the discussion of E_Fun in Section 3.2.

With the basic operational ideas behind manifest contracts established, I offer two examples of contracts for abstract datatypes.

Contracts for abstract datatypes

The standard polymorphic encodings of existential and product types transfer over to F_H's System F-style impredicative polymorphism without a problem. Indeed, dependent functions allow us to go one step further and encode even dependent products such as $(x : \mathsf{Int}) \times \{y{:}\alpha\ \mathsf{List} \mid \mathsf{length}\, y = x\}$, which represents lists paired with their lengths.

$$\begin{array}{rcl}(x : T_1) \times T_2 & = & \forall\alpha.(x{:}T_1 \to T_2 \to \alpha) \to \alpha \\ (e_1, e_2)_{(x:T_1)\times T_2} & = & \Lambda\alpha.\ \lambda f{:}(x{:}T_1 \to T_2 \to \alpha).\ f\ e_1\ e_2\end{array}$$

As in pure System F, we need to specify types on the encoding of pairs—I'll omit these types when they are clear from context. I also use record projection notation and field names, to make the example clearer.

Let's return to our simple example combining contracts and polymorphism—an abstract datatype of natural numbers.

$$\begin{array}{c}\mathsf{NAT} : \exists\alpha.\ (\mathsf{zero} : \alpha) \times (\mathsf{succ} : (\alpha \to \alpha)) \times (\mathsf{iszero} : (\alpha \to \mathsf{Bool})) \times \\ (\mathsf{pred} : \{x{:}\alpha \mid \mathsf{not}\,(\mathsf{iszero}\,x)\} \to \alpha)\end{array}$$

I omit the implementation, a standard Church encoding, where $\alpha = \forall\beta.\beta \to (\beta \to \beta) \to \beta$. The constructors zero and succ are standard; the operator iszero determines whether a natural is zero; the operator pred yields the predecessor. As we saw above, the standard representation the naturals is *inadequate* with respect to the mathematical natural numbers, in particular with respect to pred . In math, $\mathsf{pred}\,\mathsf{zero}$ is undefined, but the implementation will return zero. The NAT interface hides our encoding of the naturals behind an existential type, but to ensure adequacy, we want

to ensure that pred is only ever applied to terms of type $\{x{:}\alpha \mid \mathsf{not}\,(\mathsf{iszero}\,x)\}$. With contracts, this is easy enough:

$$\begin{array}{rl}\mathsf{NAT} : \exists\alpha. & (\mathsf{zero} : \alpha) \times (\mathsf{succ} : (\alpha \to \alpha)) \times (\mathsf{iszero} : (\alpha \to \mathsf{Bool})) \times \\ & (\mathsf{pred} : \{x{:}\alpha \mid \mathsf{not}\,(\mathsf{iszero}\,x)\} \to \alpha).\end{array}$$

Recall that in the Church encoding, α will be instantiated with $\forall\beta.\beta \to (\beta \to \beta) \to \beta$. So the refinement $\{x{:}\alpha \mid \mathsf{not}\,(\mathsf{iszero}\,x)\}$ in the new type of pred is a refinement of a polymorphic function type. These general refinement types are available in $\mathrm{F_H}$, but they were not in earlier manifest calculi.

To see why this more specific type for pred is useful, consider the following expression.

$$\mathsf{unpack\ NAT} : \exists\alpha.\ I\ \mathsf{as}\ \alpha, n\ \mathsf{in}\ n.\mathsf{iszero}\,(n.\mathsf{pred}\,(n.\mathsf{zero})) : \mathsf{Bool}$$

Here I is the interface we specified for NAT. We've "unpacked" the ADT to make its type available as α; its constructors and operators are in the dependent pair n. We then ask if the predecessor of 0 is 0, running $n.\mathsf{iszero}\,(n.\mathsf{pred}\,(n.\mathsf{zero}))$. The inner application is *not well typed*! We have that $\mathsf{zero} : \alpha$, but the domain type of pred is $\{x{:}\alpha \mid \mathsf{not}\,(\mathsf{iszero}\,x)\}$. In order to make the application well typed, we must insert a cast:

$$\begin{array}{l}\mathsf{unpack\ NAT} : \exists\alpha.\ I\ \mathsf{as}\ \alpha, n\ \mathsf{in} \\ \quad n.\mathsf{iszero}\,(n.\mathsf{pred}\,(\langle\alpha \Rightarrow \{x{:}\alpha \mid \mathsf{not}\,(n.\mathsf{iszero}\,x)\}\rangle^{l}\,n.\mathsf{zero})) : \mathsf{Bool}\end{array}$$

Naturally, this cast will ultimately raise $\Uparrow l$, because $\mathsf{not}\,(n.\mathsf{iszero}\,n.\mathsf{zero}) \longrightarrow^{*} \mathsf{false}$.

The example so far imposes constraints only on the *use* of the abstract datatype, in particular on the use of pred. To have constraints imposed also on the *implementation* of the abstract datatype, consider the extension of the interface with a subtraction operation, sub, and a binary "less than or equal" operator, leq. Natural number subtraction $\mathsf{sub}\,x\,y$ is defined only when $\mathsf{leq}\,y\,x$; we can specify this pre-condition as before, by refining the type of sub's second argument. But subtraction comes with a guarantee, as well: $\mathsf{sub}\,x\,y$ will always be less than or equal to x. That is, the result $\mathsf{sub}\,x\,y$ has the refined type $\{z{:}\alpha \mid \mathsf{leq}\,z\,x\}$. We can specify both of these facts with the interface:

$$I' = I \times (\mathsf{leq} : \alpha \to \alpha \to \mathsf{Bool}) \times (\mathsf{sub} : (x{:}\alpha \to \{y{:}\alpha \mid \mathsf{leq}\,y\,x\} \to \{z{:}\alpha \mid \mathsf{leq}\,z\,x\}))$$

The sub function's contract requires that sub's second argument is less than or equal to the first; the contract requires that sub returns a result that is less than or equal to the first argument.

How can we write an implementation to meet this interface? By putting casts in the implementations. We can impose the contracts on pred and sub when we "pack up" the implementation NAT. Writing nat for the type of the Church encoding $\forall\beta.\beta \to (\beta \to \beta) \to \beta$, we define the exported pred and sub in terms of the standard, unrefined implementations, pred' and sub' .

$$\begin{array}{rcl}\mathsf{pred} & = & \langle\mathsf{nat} \to \mathsf{nat} \Rightarrow \{x{:}\mathsf{nat} \mid \mathsf{not}\,(\mathsf{iszero}\,x)\} \to \mathsf{nat}\rangle^{l}\,\mathsf{pred}' \\ \mathsf{sub} & = & \langle\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} \Rightarrow x{:}\mathsf{nat} \to \{y{:}\mathsf{nat} \mid \mathsf{leq}\,y\,x\} \to \{z{:}\mathsf{nat} \mid \mathsf{leq}\,z\,x\}\rangle^{l}\,\mathsf{sub}'\end{array}$$

Note, however, that the cast on pred' will never actually check anything at runtime: if we unfold the domain contract contravariantly, we see that $\langle\{x{:}\mathsf{nat} \mid \mathsf{not}\,(\mathsf{iszero}\,x)\} \Rightarrow \mathsf{nat}\rangle^l$ is a no-op, because we're casting out of a refinement. Instead, clients of NAT can only call pred with terms that are typed at $\{x{:}\mathsf{nat} \mid \mathsf{not}\,(\mathsf{iszero}\,x)\}$, i.e., by checking that values are nonzero with a cast into pred's input type. The story is the same for the contract on sub's second argument—the contravariant cast won't actually check anything. The codomain contract on sub, however, could fail if sub' mis-implemented subtraction.

I can sum up the situation for contracts in abstract datatype interfaces as follows: the positive parts of the interface type are checked by the datatype's contract and can raise blame—these parts are the responsibility of the ADT's implementation; the negative parts of the interface type are not checked by the datatype's contract—these parts are the responsibility of the ADT's clients. Distributing obligations in this way recalls Findler and Felleisen's seminal idea of client and server blame [26].

Contracts as type systems

Inasmuch as abstract datatypes are little languages, contracts for abstract datatypes are like type systems for little languages. In this section, I develop a toy combinator language of string transducers, which specify mappings between regular languages.[1] A type system for the transducer combinators translates naturally to a contracted abstract datatype implementation.

Each transducer $t : \mathcal{L}_1 \hookrightarrow \mathcal{L}_2$ maps from a (regular) domain $\mathsf{dom}\,t = \mathcal{L}_1$ to a (regular) range $\mathsf{rng}\,t = \mathcal{L}_2$.

$$\begin{array}{rcl} \mathcal{S} & ::= & \text{strings} \\ \mathcal{L} & ::= & \text{regular expressions} \\ t & ::= & \mathsf{copy}\,\mathcal{L} \mid \mathsf{delete}\,\mathcal{L} \mid \mathsf{concat}\,t_1\,t_2 \mid \mathsf{seq}\,t_1\,t_2 \end{array}$$

Let ϵ be the empty string, and let $\cdot$ be string concatenation. The combinators compose to specify transducers. For this example, we only need two primitive combinators: $\mathsf{copy}\,\mathcal{L}$, maps a string in the language $\mathcal{L}$ to itself; and $\mathsf{delete}\,\mathcal{L}$ which maps a string in the language $\mathcal{L}$ to the empty string, ϵ. We can combine transducers in two ways: $\mathsf{concat}\,t_1\,t_2$ runs t_1 on the first part of the input and t_2 on the second; $\mathsf{seq}\,t_1\,t_2$ runs t_2 on the output t_1. We can define a semantics for transducers easily enough, with a function $\mathsf{run}\,t$ that takes strings in $\mathsf{dom}\,t$ to strings in $\mathsf{rng}\,t$.

The $\mathsf{copy}\,\mathcal{L}$ transducer has the simplest semantics: it just copies strings in the given language, $\mathcal{L}$. Its typing rule is straightforward.

$$\mathsf{run}\,(\mathsf{copy}\,\mathcal{L})\,\mathcal{S} = \mathcal{S} \qquad\qquad \mathsf{copy}\,\mathcal{L} : \mathcal{L} \hookrightarrow \mathcal{L}$$

The $\mathsf{delete}\,\mathcal{L}$ transducer deletes a string in the language $\mathcal{L}$; its typing rule indicates that its range is the language containing only the empty string, $\{\epsilon\}$.

[1] This is effectively a unidirectional version of Boomerang [12].

$$\mathsf{run}\,(\mathsf{delete}\,\mathcal{L})\,\mathcal{S} = \epsilon \qquad\qquad \mathsf{delete}\,\mathcal{L} : \mathcal{L} \hookrightarrow \{\epsilon\}$$

The $\mathsf{concat}\ t_1\ t_2$ combinator splits its input between its two sub-transducers. Splitting up the input makes the semantics somewhat subtle: in general, $\mathcal{S} \in \mathsf{dom}\ t_1 \cdot \mathsf{dom}\ t_2$ does not imply that there is a *unique* way to split $\mathcal{S}$. When two regular languages always split uniquely, I say they are *unambiguously splittable*, written $\mathcal{L}_1 \cdot^! \mathcal{L}_2$. Unambiguous splittability of regular languages is decidable [12]; if we only concatenate transducers with unambiguously splittable domains, then the run function will be unambiguous.

$$\begin{array}{c}\mathsf{run}\,(\mathsf{concat}\ t_1\ t_2)\ (\mathcal{S}_1 \cdot \mathcal{S}_2) = \\ \mathsf{run}\ t_1\ \mathcal{S}_1 \cdot \mathsf{run}\ t_2\ \mathcal{S}_2 \\ \text{where } \mathcal{S}_i \in \mathsf{dom}\ t_i\end{array} \qquad\qquad \frac{\begin{array}{c}t_1 : \mathcal{L}_{11} \hookrightarrow \mathcal{L}_{12} \\ t_2 : \mathcal{L}_{21} \hookrightarrow \mathcal{L}_{22} \quad \mathcal{L}_{11} \cdot^! \mathcal{L}_{21}\end{array}}{\mathsf{concat}\ t_1\ t_2 : \mathcal{L}_{11} \cdot \mathcal{L}_{21} \hookrightarrow \mathcal{L}_{12} \cdot \mathcal{L}_{22}}$$

Finally, the $\mathsf{seq}\ t_1\ t_2$ combinator runs t_2 on the output of t_1. The typing rule requires that the two sub-transducers match: $\mathsf{rng}\ t_1$ and $\mathsf{dom}\ t_2$ must be the same language.

$$\mathsf{run}\,(\mathsf{seq}\ t_1\ t_2)\,\mathcal{S} = \mathsf{run}\ t_2\,(\mathsf{run}\ t_1\ \mathcal{S}) \qquad\qquad \frac{t_1 : \mathcal{L}_1 \hookrightarrow \mathcal{L}_2 \qquad t_2 : \mathcal{L}_2 \hookrightarrow \mathcal{L}_3}{\mathsf{seq}\ t_1\ t_2 : \mathcal{L}_1 \hookrightarrow \mathcal{L}_3}$$

In order to facilitate programming with these transducers, we may try to embed these combinators inside of a functional programming. It is rather difficult to generalize this combinator language—typing a lambda calculus with string regular expression types is not easy [72, 9, 10]. It is easy, however, to define an abstract datatype offering these operations, assuming we have a type String of strings and Regex of regular expressions, with appropriate decision procedures for unambiguous splittability and equality.

$$\begin{array}{rl}\mathsf{TRANS} : \exists\alpha.\ & (\mathsf{dom} : \alpha \to \mathsf{Regex}) \times (\mathsf{rng} : \alpha \to \mathsf{Regex}) \times \\ & (\mathsf{run} : (\alpha \to \mathsf{String} \to \mathsf{String})) \times \\ & (\mathsf{copy} : \mathsf{Regex} \to \alpha) \times (\mathsf{delete} : (\mathsf{Regex} \to \alpha)) \times \\ & (\mathsf{concat} : \alpha \to \alpha \to \alpha) \times (\mathsf{seq} : (\alpha \to \alpha \to \alpha))\end{array}$$

There is a problem, though: this datatype will let us write nonsensical combinators. In particular, we can give concat transducers that don't have unambiguously splittable domains, or we can give seq transducers which don't match up. For example, suppose $\epsilon \notin \mathcal{L}$ and $\mathcal{S} \in \mathcal{L}$. Let $t = \mathsf{seq}\,(\mathsf{delete}\,\mathcal{L})\,(\mathsf{copy}\,\mathcal{L})$. Then:

$$\mathsf{run}\,(\mathsf{seq}\,(\mathsf{delete}\,\mathcal{L})\,(\mathsf{copy}\,\mathcal{L}))\,\mathcal{S} = \epsilon$$

Since $\epsilon \notin \mathcal{L}$, this means that run took a value in $\mathsf{dom}\ t = \mathcal{L}$ and produced a value outside of $\mathsf{rng}\ t = \mathcal{L}$.

We are in a difficult position: we have a little language and a type system. But scaling our transducer language's type system up to the lambda calculus increases

complexity, and distracts us from what we'd like to be doing—writing transducer programs!

Contracts offer a middle way. By putting contracts on the TRANS interface, we can ensure that all transducers are well formed.

$$\begin{aligned}\mathsf{TRANS} : \exists\alpha.\ &(\mathsf{dom} : \alpha \to \mathsf{Regex}) \times (\mathsf{rng} : \alpha \to \mathsf{Regex}) \times \\ &(\mathsf{run} : (t{:}\alpha \to \{x{:}\mathsf{String} \mid x \in \mathsf{dom}\ t\} \to \{x{:}\mathsf{String} \mid x \in \mathsf{rng}\ t\})) \times \\ &(\mathsf{copy} : \mathsf{Regex} \to \alpha) \times (\mathsf{delete} : (\mathsf{Regex} \to \alpha)) \times \\ &(\mathsf{concat} : (t_1{:}\alpha \to \{t_2{:}\alpha \mid \mathsf{splittable}\,(\mathsf{dom}\ t_1)\,(\mathsf{dom}\ t_2)\} \to \alpha)) \times \\ &(\mathsf{seq} : (t_1{:}\alpha \to \{t_2{:}\alpha \mid \mathsf{rng}\ t_1 = \mathsf{dom}\ t_2\} \to \alpha))\end{aligned}$$

The TRANS abstract datatype defines an embedded domain-specific language—with its own domain-specific type system. For example, concat will only accept transducers with splittable domains; seq will only sequence transducers that match up. The interface given here is only one of many: it checks inputs but not outputs. For example, we could ensure that concat has our intended behavior by giving its codomain the type $\{t_3{:}\alpha \mid (\mathsf{dom}\ t_3 = (\mathsf{dom}\ t_1) \circ (\mathsf{dom}\ t_2)) \wedge (\mathsf{rng}\ t_3 = (\mathsf{rng}\ t_1) \circ (\mathsf{rng}\ t_2))\}$, where $\circ$ denotes regular expression concatenation.

The checks on run and seq are easy enough to build into our TRANS implementation. But there is a distinct advantage to making the contracts explicit in the interface type: the type system will keep track of unambiguous splittability checks. Programmers can track relevant information in refinements in client modules, and we can statically eliminate redundant checks (see Section 3.4).

In general, contracting abstract datatype interfaces allows for library designers to extend the language's type system with library-specific constraints. Clients then have two choices: propagate the library's contracts through their code, possibly avoiding redundant checks; or ignore the contracts within their own code, allowing the checks to happen whenever they call into the library. Either way, the library's users can rest assured that the contracts will guarantee the safety properties the library designers desired.

If programmers are careful to program in a "cover your ass" (CYA) style, wherein each library's interface uses contracts that are strong enough to guarantee that other libraries' contracts are satisfied, then error messages greatly improve. When libraries are stacked in a hierarchy several levels deep, CYA contracts in interfaces give programmers error messages earlier and at a higher level of abstraction.

As a final note before we begin the technical content: the foregoing is the current implementation strategy for Boomerang [12], a language of bidirectional string transducers called *lenses*. The semantic constraints on Boomerang combinators are decidable, but combining Boomerang's typing rules with the lambda calculus would be cumbersome—a hard open problem. Boomerang is a complex language, and I believe there is no room in the "complexity budget" for a statically checked type system: lenses can already be difficult to program with and understand, and the complicated constraints necessary for type checking will only add more to the programmer's burden. Instead, the Boomerang primitives have contracts that ensure

Terms and contexts

$$
\begin{array}{lcl}
T & ::= & B \mid \alpha \mid x{:}T_1 \to T_2 \mid \forall\alpha.T \mid \{x{:}T \mid e\} \\
\Gamma & ::= & \emptyset \mid \Gamma, x{:}T \mid \Gamma, \alpha
\end{array}
$$

Terms, values, results, and evaluation contexts

$$
\begin{array}{lcl}
e & ::= & x \mid k \mid \mathrm{op}\,(e_1, \ldots, e_n) \mid \lambda x{:}T.\ e \mid \Lambda\alpha.\ e \mid e_1\ e_2 \mid e\ T \mid \\
 & & \langle T_1 \Rightarrow T_2\rangle^l \mid \Uparrow l \mid \langle\{x{:}T \mid e_1\}, e_2, v\rangle^l \\
v & ::= & k \mid \lambda x{:}T.\ e \mid \Lambda\alpha.\ e \mid \langle T_1 \Rightarrow T_2\rangle^l \\
r & ::= & v \mid \Uparrow l \\
E & ::= & [\,]\ e_2 \mid v_1\ [\,] \mid [\,]\ T \mid \langle\{x{:}T \mid e\}, [\,], v\rangle^l \mid \mathrm{op}(v_1, \ldots, v_{i-1}, [\,], e_{i+1}, \ldots, e_n)
\end{array}
$$

Figure 3.1: Syntax for $\mathrm{F_H}$

that they produce sane bidirectional transformations. The Boomerang libraries built atop these primitives have contracts as well, in a CYA style. Even without optimizations to reduce the number of dynamic checks, this improvement in error handling has proved quite useful. Contracts are particularly suited to the phased nature of Boomerang, since the contracts on Boomerang's lens combinators are "quasi-static". Lenses are constructed only once and then run many times. Running a lens merely requires checking regular language membership, so higher cost one-time checks can be amortized over many lens runs.

3.2 Defining $\mathbf{F_H}$

The syntax of $\mathrm{F_H}$ is given in Figure 3.1. For unrefined types we have: base types B, which must include Bool; type variables α; dependent function types $x{:}T_1 \to T_2$ where x is bound in T_2; and universal types $\forall\alpha.T$, where α is bound in T. Aside from dependency in function types, these are just the types of the standard polymorphic lambda calculus. As usual, we write $T_1 \to T_2$ for $x{:}T_1 \to T_2$ when x does not appear free in T_2. We also have predicate contracts, or *refinement types*, written $\{x{:}T \mid e\}$. Conceptually, $\{x{:}T \mid e\}$ denotes values v of type T for which $e[v/x]$ reduces to true. For each B, I fix a set $\mathcal{K}_B$ of the constants in that type; I require the typing rules for constants and the typing and evaluation rules for operations to respect this set. I also require that $\mathcal{K}_{\mathsf{Bool}} = \{\mathsf{true}, \mathsf{false}\}$.

In the syntax of terms, the first line is standard for a call-by-value polymorphic language: variables, constants, several monomorphic first-order operations op (i.e., destructors of one or more base-type arguments), term and type abstractions, and term and type applications. The second line offers the standard constructs of a manifest contract calculus [28, 34, 44], with a few alterations, discussed below.

Casts are the distinguishing feature of manifest contract calculi. When applied to a value of type T_1, the cast $\langle T_1 \Rightarrow T_2\rangle^l$ ensures that its argument behaves—and is treated—like a value of type T_2. When a cast detects a problem, it raises blame, a label-indexed uncatchable exception written $\Uparrow l$. The label l allows us to trace blame back to a specific cast. (While labels here are drawn from an arbitrary set, in practice l will refer to a source-code location.) Finally, we use active checks $\langle\{x{:}T \mid e_1\}, e_2, v\rangle^l$

to support a small-step semantics for checking casts into refinement types. In an active check, $\{x{:}T \mid e_1\}$ is the refinement being checked, e_2 is the current state of checking, and v is the value being checked. The type in the first position of an active check isn't necessary for the operational semantics, but I keep it around as a technical aid to type soundness. If checking succeeds, the check will return v; if checking fails, the check will blame its label, raising $\Uparrow l$. Active checks and blame are not intended to occur in source programs—they are runtime devices. (In a real programming language based on this calculus, casts will probably not appear explicitly either, but will be inserted by an elaboration phase. The details of this process are beyond my present scope.)

The values in $\mathrm{F_H}$ are constants, term and type abstractions, and casts. We also define *results*, which are either values or blame. (Type soundness—a consequence of Theorems 3.2.25 and 3.2.27 below—will show that evaluation produces a result, but not necessarily a value.) In Chapter 2, casts between function types applied to values were themselves considered values. I make the other choice here: excluding applications from the possible syntactic forms of values simplifies our inversion lemmas. Instead, casts between function types will η-expand their argument. This makes the notion of "function proxy" explicit: the cast semantics adds many new closures. I address this issue in Chapter 4, where I make manifest contracts space-efficient.

There are two notable features relative to existing manifest calculi: first, *any* type (even a refinement type) can be refined, not just base types (as in [28, 34, 36, 44, 51]); second, the third part of the active check form $\langle\{x{:}T \mid e_1\}, e_2, v\rangle^l$ can be any value, not just a constant. Both of these changes are motivated by the introduction of polymorphism. In particular, to support refinement of type variables we must allow refinements of *all* types, since any type can be substituted for a variable.

Operational semantics

The call-by-value operational semantics in Figure 3.2 are given as a small-step relation, split into two sub-relations: one for reductions ($\rightsquigarrow$) and one for congruence and blame lifting ($\longrightarrow$).

The latter relation is standard. The E_Reduce rule lifts $\rightsquigarrow$ reductions into $\longrightarrow$; the E_Compat rule turns $\longrightarrow$ into a congruence over evaluation contexts; and the E_Blame rule lifts blame, treating it as an uncatchable exception. The reduction relation $\rightsquigarrow$ is more interesting. There are four different kinds of reductions: the standard lambda calculus reductions, structural cast reductions, cast staging reductions, and checking reductions.

The E_Beta, and E_TBeta rules should need no explanation—these are the standard call-by-value polymorphic lambda calculus reductions. The E_Op rule uses a denotation function $[\![-]\!]$ to give meaning to the first-order operations.

The E_Refl, E_Fun, and E_Forall rules reduce casts structurally. E_Refl eliminates a cast from a type to itself; intuitively, such a cast should always succeed anyway. (I discuss this rule more in Section 3.3.) When a cast between function types is applied to a value v, the E_Fun rule produces a new lambda, wrapping v with

Reduction rules $\boxed{e_1 \rightsquigarrow e_2}$

$$
\begin{array}{rcll}
\mathrm{op}\,(v_1, \ldots, v_n) & \rightsquigarrow & [\![\mathrm{op}]\!]\,(v_1, \ldots, v_n) & \text{E\_Op} \\
(\lambda x{:}T_1.\ e_{12})\,v_2 & \rightsquigarrow & e_{12}[v_2/x] & \text{E\_Beta} \\
(\Lambda\alpha.\ e)\,T & \rightsquigarrow & e[T/\alpha] & \text{E\_TBeta} \\
\\
\langle T \Rightarrow T\rangle^l\,v & \rightsquigarrow & v & \text{E\_Refl} \\
\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l\,v & \rightsquigarrow & & \text{E\_Fun} \\
\quad \lambda x{:}T_{21}.\ (\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\,x/x] \Rightarrow T_{22}\rangle^l\,(v\,(\langle T_{21} \Rightarrow T_{11}\rangle^l\,x))) & & & \\
& & \text{when } x{:}T_{11} \to T_{12} \neq x{:}T_{21} \to T_{22} & \\
\langle \forall\alpha.T_1 \Rightarrow \forall\alpha.T_2\rangle^l\,v & \rightsquigarrow & \Lambda\alpha.\ (\langle T_1 \Rightarrow T_2\rangle^l\,(v\,\alpha)) & \text{E\_Forall} \\
& & \text{when } \forall\alpha.T_1 \neq \forall\alpha.T_2 & \\
\\
\langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle^l\,v & \rightsquigarrow & \langle T_1 \Rightarrow T_2\rangle^l\,v & \text{E\_Forget} \\
& & \text{when } T_2 \neq \{x{:}T_1 \mid e\} \text{ and } T_2 \neq \{y{:}\{x{:}T_1 \mid e\} \mid e_2\} & \\
\langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l\,v & \rightsquigarrow & \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l\,(\langle T_1 \Rightarrow T_2\rangle^l\,v) & \text{E\_PreCheck} \\
& & \text{when } T_1 \neq T_2 \text{ and } T_1 \neq \{x{:}T' \mid e'\} & \\
\langle T \Rightarrow \{x{:}T \mid e\}\rangle^l\,v & \rightsquigarrow & \langle \{x{:}T \mid e\}, e[v/x], v\rangle^l & \text{E\_Check} \\
\\
\langle \{x{:}T \mid e\}, \mathsf{true}, v\rangle^l & \rightsquigarrow & v & \text{E\_OK} \\
\langle \{x{:}T \mid e\}, \mathsf{false}, v\rangle^l & \rightsquigarrow & \Uparrow l & \text{E\_Fail}
\end{array}
$$

Evaluation rules $\boxed{e_1 \longrightarrow e_2}$

$$
\frac{e_1 \rightsquigarrow e_2}{e_1 \longrightarrow e_2}\ \text{E\_Reduce} \qquad \frac{e_1 \longrightarrow e_2}{E\,[e_1] \longrightarrow E\,[e_2]}\ \text{E\_Compat} \qquad \frac{}{E\,[\Uparrow l] \longrightarrow \Uparrow l}\ \text{E\_Blame}
$$

Figure 3.2: Operational semantics for $\mathrm{F_H}$

a contravariant cast on the domain and covariant cast on the codomain. The extra substitution in the left-hand side of the codomain cast may seem suspicious, but in fact the rule must be this way in order for type preservation to hold (see [34] for an explanation). The E_Forall rule is similar to E_Fun, generating a type abstraction with the necessary covariant cast. Side conditions on E_Forall and E_Fun ensure that these rules apply only when E_Refl doesn't.

The E_Forget, E_PreCheck, and E_Check rules are cast-staging reductions, breaking a complex cast down to a series of simpler casts and checks. All of these rules require that the left- and right-hand sides of the cast be different—if they are the same, then E_Refl applies. The E_Forget rule strips a layer of refinement off the left-hand side; in addition to requiring that the left- and right-hand sides are different, the preconditions require that the right-hand side isn't a refinement of the left-hand side. The E_PreCheck rule breaks a cast into two parts: one that checks exactly one level of refinement and another that checks the remaining parts. We only

apply this rule when the two sides of the cast are different and when the left-hand side isn't a refinement. The E_Check rule applies when the right-hand side refines the left-hand side; it takes the cast value and checks that it satisfies the right-hand side. (We don't have to check the left-hand side, since that's the type we're casting *from*.)

Before explaining how these rules interact in general, I offer a few examples. First, here is a reduction using E_Check, E_Compat, E_Op, and E_OK:

$$\begin{array}{rcl} \langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x \geq 0\}\rangle^l\, 5 & \longrightarrow & \langle \{x{:}\mathsf{Int} \mid x \geq 0\}, 5 \geq 0, 5\rangle^l \\ & \longrightarrow & \langle \{x{:}\mathsf{Int} \mid x \geq 0\}, \mathsf{true}, 5\rangle^l \longrightarrow 5 \end{array}$$

A failed check will work the same way until the last reduction, which will use E_Fail rather than E_OK:

$$\begin{array}{rcl} \langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x \geq 0\}\rangle^l\, (-1) & \longrightarrow & \langle \{x{:}\mathsf{Int} \mid x \geq 0\}, -1 \geq 0, -1\rangle^l \\ & \longrightarrow & \langle \{x{:}\mathsf{Int} \mid x \geq 0\}, \mathsf{false}, -1\rangle^l \longrightarrow \Uparrow l \end{array}$$

Notice that the blame label comes from the cast that failed. Here is a similar reduction that needs some staging, using E_Forget followed by the first reduction we gave:

$$\begin{array}{rcl} \langle \{x{:}\mathsf{Int} \mid x = 5\} \Rightarrow \{x{:}\mathsf{Int} \mid x \geq 0\}\rangle^l\, 5 & \longrightarrow & \langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x \geq 0\}\rangle^l\, 5 \\ & \longrightarrow & \langle \{x{:}\mathsf{Int} \mid x \geq 0\}, 5 \geq 0, 5\rangle^l \longrightarrow^* 5 \end{array}$$

There are two cases where we need to use E_PreCheck. First, when multiple refinements are involved:

$$\begin{array}{rcl} \langle \mathsf{Int} \Rightarrow \{x{:}\{y{:}\mathsf{Int} \mid y \geq 0\} \mid x = 5\}\rangle^l\, 5 & \longrightarrow \\ \langle \{y{:}\mathsf{Int} \mid y \geq 0\} \Rightarrow \{x{:}\{y{:}\mathsf{Int} \mid y \geq 0\} \mid x = 5\}\rangle^l\, (\langle \mathsf{Int} \Rightarrow \{y{:}\mathsf{Int} \mid y \geq 0\}\rangle^l\, 5) & \longrightarrow^* \\ \langle \{y{:}\mathsf{Int} \mid y \geq 0\} \Rightarrow \{x{:}\{y{:}\mathsf{Int} \mid y \geq 0\} \mid x = 5\}\rangle^l\, 5 & \longrightarrow \\ \langle \{x{:}\{y{:}\mathsf{Int} \mid y \geq 0\} \mid x = 5\}, 5 = 5, 5\rangle^l & \longrightarrow^* \\ 5 & \end{array}$$

Second, when casting a function or universal type into a refinement of a *different* function or universal type.

$$\begin{array}{rcl} \langle \mathsf{Bool} \to \{x{:}\mathsf{Bool} \mid x\} \Rightarrow \{f{:}\mathsf{Bool} \to \mathsf{Bool} \mid f\,\mathsf{true} = f\,\mathsf{false}\}\rangle^l\, v & \longrightarrow \\ \langle \mathsf{Bool} \to \mathsf{Bool} \Rightarrow \{f{:}\mathsf{Bool} \to \mathsf{Bool} \mid f\,\mathsf{true} = f\,\mathsf{false}\}\rangle^l & \\ (\langle \mathsf{Bool} \to \{x{:}\mathsf{Bool} \mid x\} \Rightarrow \mathsf{Bool} \to \mathsf{Bool}\rangle^l\, v) & \end{array}$$

E_Refl is necessary for simple cases, like $\langle \mathsf{Int} \Rightarrow \mathsf{Int}\rangle^l\, 5 \longrightarrow 5$. Hopefully, such a silly cast would never be written, but it could arise as a result of E_Fun or E_Forall. (We also need E_Refl in our proof of parametricity; see Section 3.3.)

I offer two higher level ways to understand the interactions of these complicated cast rules. First, we can see the reduction rules as an unfolding of a recursive function,

choosing the first clause in case of ambiguity. That is, the operational semantics unfolds a cast $\langle T_1 \Rightarrow T_2 \rangle^l\, v$ like $\mathcal{C}^l(T_1, T_2, v)$:

$$\begin{array}{rcl}
\mathcal{C}^l(T, T, v) & = & v \\
\mathcal{C}^l(\{x{:}T_1 \mid e\}, T_2, v) & = & \mathcal{C}^l(T_1, T_2, v) \\
\mathcal{C}^l(T_1, \{x{:}T_2 \mid e\}, v) & = & \mathsf{let}\ x = \mathcal{C}^l(T_1, T_2, v)\ \mathsf{in}\ \langle \{x{:}T_2 \mid e\}, e, x \rangle^l \\
\mathcal{C}^l(\forall\alpha.T_1, \forall\alpha.T_2, v) & = & \Lambda\alpha.\ \mathcal{C}^l(T_1, T_2, v) \\
\mathcal{C}^l(x{:}T_{11} \to T_{12}, x{:}T_{21} \to T_{22}, v) & = & \\
\multicolumn{3}{r}{\lambda x{:}T_{21}.\ \mathcal{C}^l(T_{12}[\mathcal{C}^l(T_{21}, T_{11}, x)/x], T_{22}, v\, \mathcal{C}^l(T_{21}, T_{11}, x))}
\end{array}$$

Alternatively, the rules firing during the evaluation of a cast in the small-step semantics obeys the following regular schema:

$$\textsc{Refl} \mid (\textsc{Forget}^* \ (\textsc{Refl} \mid (\textsc{PreCheck}^* \ (\textsc{Refl} \mid \textsc{Fun} \mid \textsc{Forall})?\ \textsc{Check}^*)))$$

Let's consider the cast $\langle T_1 \Rightarrow T_2 \rangle^l\, v$. To simplify the following discussion, I define $\mathrm{unref}(T)$ as T without any outer refinements (though refinements on, e.g., the domain of a function would be unaffected); I write $\mathrm{unref}_n(T)$ when we remove only the n outermost refinements:

$$\mathrm{unref}(T) = \begin{cases} \mathrm{unref}(T') & \text{if } T = \{x{:}T' \mid e\} \\ T & \text{otherwise} \end{cases}$$

First, if $T_1 = T_2$, we can apply E_Refl and be done with it. If that doesn't work, we'll reduce by E_Forget until the left-hand side doesn't have any refinements. (N.B. we may not have to make any of these reductions.) Either all of the refinements will be stripped away from the source type, or E_Refl eventually applies and the entire cast disappears. Assuming E_Refl doesn't apply, we now have $\langle \mathrm{unref}(T_1) \Rightarrow T_2 \rangle^l\, v$. Next, we apply E_PreCheck until the cast is completely decomposed into one-step casts, once for each refinement in T_2:

$$\begin{array}{l}
\langle \mathrm{unref}_1(T_2) \Rightarrow T_2 \rangle^l (\langle \mathrm{unref}_2(T_2) \Rightarrow \mathrm{unref}_1(T_2) \rangle^l \\
\qquad (\ldots\ (\langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2) \rangle^l\, v)\ \ldots))
\end{array}$$

As our next step, we apply whichever structural cast rule applies to $\langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2) \rangle^l\, v$, one of E_Refl, E_Fun, or E_Forall. Now all that remains are some number of refinement checks, which can be dispatched by the E_Check rule (and other rules, of course, during the predicate checks themselves).

The E_Refl rule merits some more discussion. On the face of it, a cast $\langle T \Rightarrow T \rangle^l$ seems like it can't do anything: any value it applies must have already had type T, so what could go wrong during any checks? One might worry that adding such a cast will cause a different label to be blamed. In fact, we will find that such casts have no effect in Section 3.4, using the parametricity logical relation. But this safety proof is for a system where E_Refl was there in the first place! I haven't been able to prove parametricty for a system without E_Refl. Including it in the system, however, is a weaker assumption than including subsumption, as earlier systems did [34, 44]. It may be possible to further weaken the E_Refl by restricting it to base types.

Static typing

The type system comprises three mutually recursive judgments: context well formedness, type well formedness, and term well typing. The rules for contexts and types are unsurprising. The rules for terms are mostly standard. First, the T_App rule is dependent, to account for dependent function types. We have no need for a value restriction because F_H doesn't have effects; any effectful extension, even nontermination, would force such a restriction. The T_Cast rule is standard for manifest calculi, allowing casts between compatibly structured well formed types. Compatibility of type structures is defined in Figure 3.4; in short, compatible types erase to identical simple type skeletons. I discuss type compatibility and type conversion (as used in T_Conv) below. Note that I assign casts a non-dependent function type. The T_Op rule uses the ty function to assign (possibly dependent) monomorphic first-order types to operations; I require that $\mathsf{ty}(\mathrm{op})$ and $[\![\mathrm{op}]\!]$ agree.

Some of the typing rules—T_Check, T_Blame, T_Exact, T_Forget, and T_Conv—are "runtime only". These rules aren't needed to type check source programs, but we need them to guarantee preservation. T_Check, T_Exact, and T_Conv are excluded from source programs because I don't want the typing of source programs to rely on the evaluation relation; such an interaction is acceptable in this setting, but disrupts the phase distinction and is ultimately incompatible with nontermination and effects. I exclude T_Blame because programs shouldn't *start* with failures. Finally, I exclude T_Forget because I imagine that source programs have all type changes explicitly managed by casts. In explicitly tagged calculus, as described in Section 3.5 and Chapter 4, we are able to abandon the T_Forget rule in its entirety. Note that the conclusions of these rules use a context Γ, but their premises don't use Γ at all. Even though runtime terms and their typing rules should only ever occur in an empty context, the T_App rule substitutes terms into types—so a runtime term could end up under a binder. I therefore allow the runtime typing rules to apply in any well formed context, so long as the terms they type check are closed. The T_Blame rule allows us to give any type to blame—this is necessary for preservation. The T_Check rule types an active check, $\langle \{x{:}T \mid e_1\}, e_2, v\rangle^l$. Such a term arises when a term like $\langle T \Rightarrow \{x{:}T \mid e_1\}\rangle^l\, v$ reduces by E_Check. The premises of the rule are all intuitive except for $e_1[v/x] \longrightarrow^* e_2$, which is necessary to avoid nonsensical terms like $\langle \{x{:}T \mid x \geq 0\}, \mathsf{true}, -1\rangle^l$, where the wrong predicate gets checked. In Chapter 2, we have a similar but looser requirement: $e_2 \longrightarrow^* \mathsf{true}$ implies $e_1[v/x] \longrightarrow^* \mathsf{true}$. The proofs don't change radically, but I use the more syntactic requirement here because I aim to keep the system as syntactic as possible. The T_Exact rule allows us to retype a closed value of type T at $\{x{:}T \mid e\}$ if $e[v/x] \longrightarrow^* \mathsf{true}$. This typing rule guarantees type preservation for E_OK: $\langle \{x{:}T \mid e_1\}, \mathsf{true}, v\rangle^l \longrightarrow v$. If the active check was well typed, then we know that $e_1[v/x] \longrightarrow^* \mathsf{true}$, so T_Exact applies. Earlier systems used most-specific types and subtyping to show that the E_OK rule preserves typing. While the "most specific" requirement is abstract, a constant $k \in \mathcal{K}_B$ is typically given the *selfified* type $\mathsf{ty}(k) = \{x{:}B \mid x = k\}$ [51]. But functions don't

Context well formedness $\boxed{\vdash \Gamma}$

$$\frac{}{\vdash \emptyset}\ \text{WF\_Empty} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash T}{\vdash \Gamma, x{:}T}\ \text{WF\_ExtendVar} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, \alpha}\ \text{WF\_ExtendTVar}$$

Type well formedness $\boxed{\Gamma \vdash T}$

$$\frac{\vdash \Gamma}{\Gamma \vdash B}\ \text{WF\_Base} \qquad \frac{\vdash \Gamma \quad \alpha \in \Gamma}{\Gamma \vdash \alpha}\ \text{WF\_TVar} \qquad \frac{\Gamma, \alpha \vdash T}{\Gamma \vdash \forall \alpha. T}\ \text{WF\_Forall}$$

$$\frac{\Gamma \vdash T_1 \quad \Gamma, x{:}T_1 \vdash T_2}{\Gamma \vdash x{:}T_1 \to T_2}\ \text{WF\_Fun} \qquad \frac{\Gamma \vdash T \quad \Gamma, x{:}T \vdash e : \mathsf{Bool}}{\Gamma \vdash \{x{:}T \mid e\}}\ \text{WF\_Refine}$$

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

$$\frac{\vdash \Gamma \quad x{:}T \in \Gamma}{\Gamma \vdash x : T}\ \text{T\_Var} \qquad \frac{\vdash \Gamma}{\Gamma \vdash k : \mathsf{ty}(k)}\ \text{T\_Const} \qquad \frac{\emptyset \vdash T \quad \vdash \Gamma}{\Gamma \vdash \Uparrow l : T}\ \text{T\_Blame}$$

$$\frac{\Gamma \vdash T_1 \quad \Gamma, x{:}T_1 \vdash e_{12} : T_2}{\Gamma \vdash \lambda x{:}T_1.\ e_{12} : x{:}T_1 \to T_2}\ \text{T\_Abs} \qquad \frac{\Gamma \vdash e_1 : (x{:}T_1 \to T_2) \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2[e_2/x]}\ \text{T\_App}$$

$$\frac{\begin{array}{l}\vdash \Gamma \qquad \mathsf{ty}(\mathrm{op}) = x_1 : T_1 \to \ldots \to x_n : T_n \to T \\ \Gamma \vdash e_i : T_i[e_1/x_1, \ldots, e_{i-1}/x_{i-1}]\end{array}}{\Gamma \vdash \mathrm{op}\,(e_1, \ldots, e_n) : T[e_1/x_1, \ldots, e_n/x_n]}\ \text{T\_Op}$$

$$\frac{\Gamma, \alpha \vdash e : T}{\Gamma \vdash \Lambda\alpha.\ e : \forall \alpha. T}\ \text{T\_TAbs} \qquad \frac{\Gamma \vdash e_1 : \forall \alpha. T \quad \Gamma \vdash T_2}{\Gamma \vdash e_1\ T_2 : T[T_2/\alpha]}\ \text{T\_TApp}$$

$$\frac{\Gamma \vdash T_1 \quad \Gamma \vdash T_2 \quad T_1 \parallel T_2}{\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle^l : T_1 \to T_2}\ \text{T\_Cast}$$

$$\frac{\vdash \Gamma \quad \emptyset \vdash \{x{:}T \mid e_1\} \quad \emptyset \vdash v : T \quad \emptyset \vdash e_2 : \mathsf{Bool} \quad e_1[v/x] \longrightarrow^* e_2}{\Gamma \vdash \langle \{x{:}T \mid e_1\}, e_2, v \rangle^l : \{x{:}T \mid e_1\}}\ \text{T\_Check}$$

$$\frac{\vdash \Gamma \quad \emptyset \vdash e : T \quad \emptyset \vdash T' \quad T \equiv T'}{\Gamma \vdash e : T'}\ \text{T\_Conv} \qquad \frac{\emptyset \vdash v : \{x{:}T \mid e\} \quad \vdash \Gamma}{\Gamma \vdash v : T}\ \text{T\_Forget}$$

$$\frac{\vdash \Gamma \quad \emptyset \vdash v : T \quad \emptyset \vdash \{x{:}T \mid e\} \quad e[v/x] \longrightarrow^* \mathsf{true}}{\Gamma \vdash v : \{x{:}T \mid e\}}\ \text{T\_Exact}$$

Figure 3.3: Typing rules for $\mathrm{F_H}$

admit a decidable equality, so there isn't an obvious way to assign them most-specific types. T_Exact is a suitably extensional,syntactic, and subtyping-free replacement for the earlier semantic requirement: constants and functions can be assigned less specific types, but we can use T_Exact in the preservation proof to remember successful checks. Finally, the T_Conv rule allows us to retype expressions at convertible types: if $\emptyset \vdash e : T$ and $T \equiv T'$, then $\emptyset \vdash e : T'$ (or in any well formed context Γ). I define $\equiv$ as an explicitly substitutive but otherwise structural type conversion relation in Figure 3.4. The T_Conv rule is necessary to prove preservation in the case where $e_1\ e_2 \longrightarrow e_1\ e_2'$. Why? The first term is typed at $T_2[e_2/x]$ (by T_App), but reapplying T_App types the second term at $T_2[e_2'/x]$.

In the original ESOP 2011 work [8], we noted that $T_2[e_2/x]$ parallel reduces to $T_2[e_2'/x]$, so we took parallel reduction as type conversion. It turns out that parallel reduction doesn't quite have the properties we need. I discuss this further in Section 3.5. Here, I define a type conversion relation in Figure 3.4 that uses what I call *common subexpression reduction*, or CSR.. Belo et al. [8] also (falsely) claimed that symmetry wasn't necessary for type soundness or parametricity, but symmetry is in fact used in the proof of preservation (Lemma 3.2.27, when a term typed by T_App steps by E_Reduce/E_Refl).

The following proof of type soundness is (almost) entirely syntactic [79], offering a new approach to manifest calculi. In Chapter 2, we use subtyping instead of the $\equiv$ relation; one of the contributions in this work is the insight that subtyping—with its accompanying metatheoretical complications that prevent a simple syntactic proof of type soundness—is not an essential component of manifest calculi.

I say *almost* entirely because both type soundness and parametricty rest on a conjecture about a semantic property which I call *cotermination*.

3.2.1 Conjecture [Cotermination at true]: If $\sigma \longrightarrow^* \sigma'$ then $\sigma(e) \longrightarrow^* \mathsf{true}$ iff $\sigma'(e) \longrightarrow^* \mathsf{true}$.

That is, if σ and σ' are common subexpression reductions, then terms behave equivalently when substituted with either. We discuss this conjecture further in Section 3.5.

I define type compatibility and type conversion in Figure 3.4. It is critical to understand that type variables are compatible only with themselves and refinements of themselves. To understand why this is, recall that compatibility determines when it is permissible to cast between two types. We can't allow a cast between α and a base type B, a function type $x{:}T_1 \to T_2$, or a quantified type $\forall\beta.T$—we have no idea what type will fill in for α, and it's critical that compatibility is substitutive, i.e., that if $T_1 \parallel T_2$, then $T_1[e/x] \parallel T_2$ (Lemma 3.2.17). Moreover, we must avoid nontermination due to non-paremetric operations (e.g., Girard's J operator); it's imperative that a term like

$$\mathsf{let}\ \delta = \Lambda\alpha.\ \lambda x{:}\alpha.\ \langle \alpha \Rightarrow \forall\beta.\beta \to \beta \rangle^l\, \alpha\, x\ \mathsf{in}\ \delta\ \forall\beta.\beta \to \beta\ \delta$$

isn't well typed.

3.2.2 Lemma [Determinism]: If $e \longrightarrow e_1$ and $e \longrightarrow e_2$ then $e_1 = e_2$.

Type compatibility $\boxed{T_1 \parallel T_2}$

$$\frac{}{\alpha \parallel \alpha}\ \textsc{Sim\_Var} \qquad \frac{}{B \parallel B}\ \textsc{Sim\_Base}$$

$$\frac{T_1 \parallel T_2}{\{x{:}T_1 \mid e\} \parallel T_2}\ \textsc{Sim\_RefineL} \qquad \frac{T_1 \parallel T_2}{T_1 \parallel \{x{:}T_2 \mid e\}}\ \textsc{Sim\_RefineR}$$

$$\frac{T_{11} \parallel T_{21} \quad T_{12} \parallel T_{22}}{x{:}T_{11} \to T_{12} \parallel x{:}T_{21} \to T_{22}}\ \textsc{Sim\_Fun} \qquad \frac{T_1 \parallel T_2}{\forall\alpha.\, T_1 \parallel \forall\alpha.\, T_2}\ \textsc{Sim\_Forall}$$

Conversion $\boxed{\sigma_1 \longrightarrow^* \sigma_2}$ $\boxed{T_1 \equiv T_2}$

$$\sigma_1 \longrightarrow^* \sigma_2 \iff \begin{array}{l} \mathrm{dom}(\sigma_1) = \mathrm{dom}(\sigma_2) \wedge \\ \forall x \in \mathrm{dom}(\sigma_1).\ \sigma_1(x) \longrightarrow^* \sigma_2(x) \wedge \\ \forall \alpha \in \mathrm{dom}(\sigma_1).\ \sigma_1(\alpha) = \sigma_2(\alpha) \end{array}$$

$$\frac{}{\alpha \equiv \alpha}\ \textsc{C\_Var} \qquad \frac{}{B \equiv B}\ \textsc{C\_Base} \qquad \frac{\sigma_1 \longrightarrow^* \sigma_2 \quad T_1 \equiv T_2}{\{x{:}T_1 \mid \sigma_1(e)\} \equiv \{x{:}T_2 \mid \sigma_2(e)\}}\ \textsc{C\_Refine}$$

$$\frac{T_1 \equiv T_1' \quad T_2 \equiv T_2'}{x{:}T_1 \to T_2 \equiv x{:}T_1' \to T_2'}\ \textsc{C\_Fun} \qquad \frac{T \equiv T'}{\forall\alpha.\, T \equiv \forall\alpha.\, T'}\ \textsc{C\_Forall}$$

$$\frac{T_2 \equiv T_1}{T_1 \equiv T_2}\ \textsc{C\_Sym} \qquad \frac{T_1 \equiv T_2 \quad T_2 \equiv T_3}{T_1 \equiv T_3}\ \textsc{C\_Trans}$$

Figure 3.4: Type compatibility and conversion for $\mathrm{F_H}$

3.2.3 Lemma [Reflexivity of conversion]:
$T \equiv T$ for all T.

Proof: By induction on T.

$\underline{(T = \alpha)}$: By C_Var.

$\underline{(T = B)}$: By C_Base.

$\underline{(T = \{x{:}T' \mid e\})}$: By the IH on T', we have $T' \equiv T'$. With an empty σ, we have $\sigma \longrightarrow^* \sigma$ trivially. We are done by C_Refine.

$\underline{(T = x{:}T_1 \to T_2)}$: By the IH on T_1 and T_2 and C_Fun.

$\underline{(T = \forall\alpha.\, T)}$: By the IH on T and C_Forall. □

3.2.4 Lemma [Like-type arrow conversion]: If $x{:}T_{11} \to T_{12} \equiv T$ then $T = x{:}T_{21} \to T_{22}$.

Proof: By induction on the conversion relation. Only C_Fun applies, and C_Sym and C_Trans are resolved by the IH. □

3.2.5 Lemma [Conversion arrow inversion]: If $x{:}T_{11} \to T_{12} \equiv x{:}T_{21} \to T_{22}$ then $T_{11} \equiv T_{21}$ and $T_{12} \equiv T_{22}$.

Proof: By induction on the conversion derivation.

(C_Fun): Immediate.

(C_Sym): By the IH and C_Sym.

(C_Trans): It must be the case that T_2 is an arrow type (Lemma 3.2.4), so by the IH and C_Trans. □

3.2.6 Lemma [Like-type forall conversion]: If $\forall\alpha.\, T_1 \equiv T$ then $T = \forall\alpha.\, T_2$.

Proof: By induction on the conversion relation. Only C_Forall applies, and C_Sym and C_Trans are resolved by the IH. □

3.2.7 Lemma [Conversion forall inversion]: If $\forall\alpha.\, T_1 \equiv \forall\alpha.\, T_2$ then $T_1 \equiv T_2$.

Proof: By induction on the conversion derivation.

(C_Forall): Immediate.

(C_Sym): By the IH and C_Sym.

(C_Trans): By Lemma 3.2.6, we know that the intermediate type is a forall type, so by the IH and C_Trans. □

3.2.8 Lemma [Term substitutivity of conversion]:
If $T_1 \equiv T_2$ and $e_1 \longrightarrow^* e_2$ then $T_1[e_1/x] \equiv T_2[e_2/x]$.

3.2.9 Lemma [Type substitutivity of conversion]:
If $T_1 \equiv T_2$ then $T_1[T/\alpha] \equiv T_2[T/\alpha]$.

3.2.10 Lemma [Cotermination of refinement types]: If $\{x{:}T_1 \mid e_1\} \equiv \{x{:}T_2 \mid e_2\}$ then $T_1 \equiv T_2$ and $e_1[v/x] \longrightarrow^* \mathsf{true}$ iff $e_2[v/x]$, for all v.

Proof: By induction on the equivalence. There are three cases.

(C_Refine): We have $T_1 \equiv T_2$ by assumption. We know that $e_1 = \sigma_1(e)$ and $e_2 = \sigma_2(e)$ for $\sigma_1 \longrightarrow^* \sigma_2$. It is trivially true that $v \longrightarrow^* v$, so $\sigma_1[v/x] \longrightarrow^* \sigma_2[v/x]$. By cotermination (Lemma 3.2.1), we know that $\sigma_1(e)[v/x] \longrightarrow^* \mathsf{true}$ iff $\sigma_2(e)[v/x]$.

(C_Sym): By the IH.

(C_Trans): By the IHs and transitivity of $\equiv$ and cotermination. □

3.2.11 Lemma [Value inversion]: If $\emptyset \vdash v : T$ and $\mathrm{unref}_n(T) = \{x{:}T_n \mid e_n\}$ then $e_n[v/x] \longrightarrow^* \mathsf{true}$.

Proof: By induction on the height of the typing derivation; we list all the cases that could type values.

Proof:

(T_Const): By assumption of valid typing of constants.

(T_Abs): Contradictory—the type is wrong.

(T_TAbs): Contradictory—the type is wrong.

(T_Cast): Contradictory—the type is wrong.

(T_Conv): By applying Lemma 3.2.10 on the stack of refinements on T.

(T_Forget): By the IH on $\emptyset \vdash v : \{x{:}T \mid e\}$, adjusting each of the n down by one to cover the stack of refinements on T.

(T_Exact): By assumption for the outermost refinement; by the IH on $\emptyset \vdash v : T$ for the rest. □

3.2.12 Lemma [Term weakening]: If x is fresh and $\Gamma \vdash T'$ then

1. $\Gamma, \Gamma' \vdash e : T$ implies $\Gamma, x{:}T', \Gamma \vdash e : T$,
2. $\Gamma, \Gamma' \vdash T$ implies $\Gamma, x{:}T', \Gamma' \vdash T$, and
3. $\vdash \Gamma, \Gamma'$ implies $\vdash \Gamma, x{:}T', \Gamma'$.

Proof: By induction on e, T, and Γ'. The only interesting case is for terms where a runtime rule applies:

(T_Conv,T_Exact,T_Forget): The argument is the same for all terms, so: since $\vdash \Gamma, x{:}T', \Gamma'$, we can reapply T_Conv, T_Exact, or T_Forget, respectively. In the rest of this proof, we won't bother considering these rules. □

3.2.13 Lemma [Type weakening]: If α is fresh then

1. $\Gamma, \Gamma' \vdash e : T$ implies $\Gamma, \alpha, \Gamma \vdash e : T$,
2. $\Gamma, \Gamma' \vdash T$ implies $\Gamma, \alpha, \Gamma' \vdash T$, and
3. $\vdash \Gamma, \Gamma'$ implies $\vdash \Gamma, \alpha, \Gamma'$.

Proof: By induction on e, T, and Γ'. The proof is similar to term weakening, Lemma 3.2.12. □

3.2.14 Lemma [Compatibility is symmetric]: $T_1 \parallel T_2$ iff $T_2 \parallel T_1$.

Proof: By induction on $T_1 \parallel T_2$.

(Sim_Var): By Sim_Var.

(Sim_Base): By Sim_Base.

(Sim_RefineL): By Sim_RefineR and the IH.

(Sim_RefineR): By Sim_RefineL and the IH.

(Sim_Fun): By Sim_Fun and the IHs.

(Sim_Forall): By the IH and Sim_Forall. □

3.2.15 Lemma [Substitution preserves compatibility]:
If $T_1 \parallel T_2$, then $T_1[e/x] \parallel T_2$.

Proof: By induction on the compatibility relation.

(Sim_Var): By Sim_Var.

(Sim_Base): By Sim_Base.

(Sim_RefineL): By Sim_RefineL and the IH.

(Sim_RefineR): By Sim_RefineR and the IH.

(Sim_Fun): By Sim_Fun and the IHs.

(Sim_Forall): By Sim_Forall and the IH. □

3.2.16 Lemma [Term substitution]: If $\Gamma \vdash e' : T'$, then

1. if $\Gamma, x{:}T', \Gamma' \vdash e : T$ then $\Gamma, \Gamma'[e'/x] \vdash e[e'/x] : T[e'/x]$,
2. if $\Gamma, x{:}T', \Gamma' \vdash T$ then $\Gamma, \Gamma'[e'/x] \vdash T[e'/x]$, and
3. if $\vdash \Gamma, x{:}T', \Gamma'$ then $\vdash \Gamma, \Gamma'[e'/x]$.

Proof: By induction on e, T, and Γ'. In the first two clauses, we are careful to leave Γ' as long as it is well formed. □

3.2.17 Lemma [Type substitution preserves compatibility]: If $T_1 \parallel T_2$ then $T_1[T'/\alpha] \parallel T_2$.

Proof: By induction on the compatibility relation.

(Sim_Var): By Sim_Var.

(Sim_Base): By Sim_Base.

(Sim_RefineL): By Sim_RefineL and the IH.

(Sim_RefineR): By Sim_RefineR and the IH.

(Sim_Fun): By Sim_Fun and the IHs.

(Sim_Forall): By Sim_Forall and the IH.

□

3.2.18 Lemma [Type substitution]: If $\Gamma \vdash T'$ then

1. if $\Gamma, \alpha, \Gamma' \vdash e : T$, then $\Gamma, \Gamma'[T'/\alpha] \vdash e[T'/\alpha] : T[T'/\alpha]$,

2. if $\Gamma, \alpha, \Gamma' \vdash T$, then $\Gamma, \Gamma'[T'/\alpha] \vdash T[T'/\alpha]$, and

3. if $\vdash \Gamma, \alpha, \Gamma'$, then $\vdash \Gamma, \Gamma'[T'/\alpha]$.

Proof: By induction on e, T, and Γ'. □

As is standard for type systems with conversion rules, we must prove inversion lemmas in order to reason about typing derivations in a syntax-directed way.

3.2.19 Lemma [Lambda inversion]: If $\Gamma \vdash \lambda x{:}T_1.\ e_{12} : T$, then

1. $\Gamma \vdash T_1$,

2. $\Gamma, x{:}T_1 \vdash e_{12} : T_2$, and

3. $x{:}T_1 \to T_2 \equiv \mathrm{unref}(T)$.

Proof: By induction on the typing derivation. Cases not mentioned only apply to terms which are not lambdas.

(T_Abs): By inversion, we have $\Gamma \vdash T_1$ and $\Gamma, x{:}T_1 \vdash e_{12} : T_2$. We find conversion immediately by reflexivity (Lemma 3.2.3), since $\mathrm{unref}(T) = T = x{:}T_1 \to T_2$.

(T_Conv): We have $\Gamma \vdash \lambda x{:}T_1.\ e_{12} : T$; by inversion, $T \equiv T'$ and $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : T'$. By the IH on this second derivation, we find $\emptyset \vdash T_1$ and $x{:}T_1 \vdash e_{12} : T_2$ where, $\mathrm{unref}(T') \equiv x{:}T_1 \to T_2$. By weakening, we have $\Gamma \vdash T_1$ and $\Gamma, x{:}T_1 \vdash e_{12} : T_2$. Since $T' \equiv T$, we have $x{:}T_1 \to T_2 \equiv \mathrm{unref}(T') \equiv \mathrm{unref}(T)$ by C_Trans.

(T_Exact): $T = \{x{:}T' \mid e\}$, and we have $\Gamma \vdash \lambda x{:}T_1.\ e_{12} : \{x{:}T' \mid e\}$; by inversion, $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : T'$. By the IH, $\emptyset \vdash T_1$ and $x{:}T_1 \vdash e_{12} : T_2$, where $x{:}T_1 \to T_2 \equiv \mathrm{unref}(T')$. By weakening, $\Gamma \vdash T_1$ and $\Gamma, x{:}T_1 \vdash e_{12} : T_2$. Since $\mathrm{unref}(T') = \mathrm{unref}(\{x{:}T' \mid e\})$, we have the conversion by C_Trans): $x{:}T_1 \to T_2 \equiv \mathrm{unref}(T') = \mathrm{unref}(\{x{:}T' \mid e\})$.

(T_Forget): We have $\Gamma \vdash \lambda x{:}T_1.\ e_{12} : T$; by inversion, $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : \{x{:}T \mid e\}$. By the IH on this latter derivation, we $\emptyset \vdash T_1$ and $x{:}T_1 \vdash e_{12} : T_2$, where $x{:}T_1 \to T_2 \equiv \mathrm{unref}(\{x{:}T \mid e\})$. By weakening, $\Gamma \vdash T_1$ and $\Gamma, x{:}T_1 \vdash e_{12} : T_2$. Since $\mathrm{unref}(\{x{:}T \mid e\}) = \mathrm{unref}(T)$, we have by C_Trans that $x{:}T_1 \to T_{12} \equiv \mathrm{unref}(\{x{:}T \mid e\}) = \mathrm{unref}(T)$. □

3.2.20 Lemma [Cast inversion]: If $\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle^l : T$, then

1. $\Gamma \vdash T_1$,

2. $\Gamma \vdash T_2$,

3. $T_1 \parallel T_2$, and

4. $\_{:}T_1 \to T_2 \equiv \mathrm{unref}(T)$ (i.e., T_2 does not mention the dependent variable).

Proof: By induction on the typing derivation. Cases not mentioned only apply to syntactically distinct terms.

(T_Cast): $T = \_ : T_1 \to T_2$, and the derivation is by inversion. Conversion is by reflexivity (Lemma 3.2.3). T_2 does not mention the variable.

(T_Conv): $\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle^l : T$; by inversion, $\emptyset \vdash \langle T_1 \Rightarrow T_2 \rangle^l : T'$ and $T' \equiv T$. By the IH, we have $\emptyset \vdash T_1$ and $\emptyset \vdash T_2$ (which weaken to the derivations we want), as well as $T_1 \parallel T_2$ and $\_ : T_1 \to T_2 \equiv \mathrm{unref}(T')$. But $\mathrm{unref}(T') \equiv \mathrm{unref}(T)$, so we have the conversion we want by C_Trans.

(T_Exact): $\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle^l : \{x{:}T \mid e\}$; by inversion, $\emptyset \vdash \langle T_1 \Rightarrow T_2 \rangle^l : T$. By the IH, we have $\emptyset \vdash T_1$ and $\emptyset \vdash T_2$ (which weaken to the derivations we want). We also have $T_1 \parallel T_2$ and $\_ : T_1 \to T_2 \equiv \mathrm{unref}(T)$ — which is equal to $\mathrm{unref}(\{x{:}T \mid e\})$, so we're done.

(T_Forget): $\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle^l : T$; by inversion, $\emptyset \vdash \langle T_1 \Rightarrow T_2 \rangle^l : \{x{:}T \mid e\}$. By the IH, $\emptyset \vdash T_1$ and $\emptyset \vdash T_2$, so we find the derivations we want by weakening. We also have $T_1 \parallel T_2$ and $\_ : T_1 \to T_2 \equiv \mathrm{unref}(T)$, which is equal $\mathrm{unref}(\{x{:}T \mid e\})$, so this case is complete.

□

3.2.21 Lemma [Application inversion]: If $\Gamma \vdash e_1\ e_2 : T$, then

1. $\Gamma \vdash e_1 : (x{:}T_1 \to T_2)$,
2. $\Gamma \vdash e_2 : T_1$, and
3. $T_2[e_2/x] \equiv T$.

Proof: By induction on the typing derivation. Cases not mentioned only apply to syntactically distinct terms.

(T_App): We find the typings we want by inversion. Since $T = T_2[e_2/x]$, conversion is by reflexivity (Lemma 3.2.3).

(T_Conv): By the IH on $\emptyset \vdash e_1\ e_2 : T'$, we find the typings we want (up to weakening), and $T_2[e_2/x] \equiv T'$. By inversion, $T' \equiv T$, so by C_Trans we have $T_2[e_2/x] \equiv T$.

(T_Exact): Only applies to values, and $e_1\ e_2$ cannot be a value.

(T_Forget): Only applies to values, and $e_1\ e_2$ cannot be a value. □

3.2.22 Lemma [Type abstraction inversion]: If $\Gamma \vdash \Lambda\alpha.\ e : T$, then

1. $\Gamma, \alpha \vdash e : T'$ and
2. $\forall\alpha.T' \equiv \mathrm{unref}(T)$.

Proof: By induction on the typing derivation. Cases not mentioned only apply to syntactically distinct terms.

(T_TAbs): $T = \forall\alpha.\, T'$. Conversion is by reflexivity (Lemma 3.2.3).

(T_Conv): By the IH on $\emptyset \vdash \Lambda\alpha.\ e : T''$, using weakening to recover the typing derivation and using transitivity of conversion through unref to find $\text{unref}(T'') \equiv \text{unref}(T)$ by C_Trans.

(T_Exact): $T = \{x{:}T_0 \mid e_0\}$ for some T_0 and e_0. By the IH on $\emptyset \vdash \Lambda\alpha.\ e : T_0$, using weakening for the derivation and the fact that $\text{unref}(\{x{:}T_0 \mid e_0\}) = \text{unref}(T_0)$ to find conversion.

(T_Forget): By the IH on $\emptyset \vdash \Lambda\alpha.\ e : \{x{:}T \mid e_0\}$, using weakening for the derivation and the fact that $\text{unref}(\{x{:}T \mid e_0\}) = \text{unref}(T)$ to find conversion. □

Inversion lemmas in hand, we prove a canonical forms lemma to support a proof of progress. The canonical forms proof is "modulo" the unref function: the shape of the values of type $\{x{:}T \mid e\}$ are determined by the inner type T.

3.2.23 Lemma [Conversion of unrefined types]: If $T_1 \equiv T_2$ then $\text{unref}(T_1) \equiv \text{unref}(T_2)$.

Proof: By induction on the derivation of $T_1 \equiv T_2$. □

3.2.24 Lemma [Canonical forms]: If $\emptyset \vdash v : T$, then:

1. If $\text{unref}(T) = B$ then $v = k \in \mathcal{K}_B$ for some v
2. If $\text{unref}(T) = x{:}T_1 \to T_2$ then v is
   (a) $\lambda x{:}T_1'.\ e_{12}$ and $T_1' \equiv T_1$ for some x, T_1' and e_{12}, or
   (b) $\langle T_1' \Rightarrow T_2' \rangle^l$ and $T_1' \equiv T_1$ and $T_2' \equiv T_2$ for some T_1', T_2', and l
3. If $\text{unref}(T) = \forall\alpha.\, T'$ then v is $\Lambda\alpha.\ e$ for some e.

Proof: By induction on the typing derivation.

(T_Var): Contradictory: variables are not values.

(T_Const): $\emptyset \vdash k : T$ and $\text{unref}(T) = B$; we are in case 1. By assumption, $k \in \mathcal{K}_B$.

(T_Op): Contradictory: $\text{op}\,(e_1, \ldots, e_n)$ is not a value.

(T_Abs): $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : T$ and $T = \text{unref}(T) = x{:}T_1 \to T_2$; we are in case 2a. Conversion is by reflexivity (Lemma 3.2.3).

(T_App): Contradictory: $e_1\ e_2$ is not a value.

(T_TAbs): $\emptyset \vdash \Lambda\alpha.\ e : \forall\alpha.\, T$; we are in case 3. It is immediate that $v = \Lambda\alpha.\ e$, and conversion is by reflexivity (Lemma 3.2.3).

(T_TApp): Contradictory: $e\ T$ is not a value.

(T_Cast): $\emptyset \vdash \langle T_1 \Rightarrow T_2 \rangle^l : \_ : T_1 \to T_2$; we are in case 2b. It is immediate that $v = \langle T_1 \Rightarrow T_2 \rangle^l$. Conversion is by reflexivity (Lemma 3.2.3).

(T_Check): Contradictory: $\langle \{x{:}T \mid e_1\}, e_2, v \rangle^l$ is not a value.

(T_Blame): Contradictory: $\Uparrow l$ is not a value.

(T_Conv): $\emptyset \vdash v : T$; by inversion, $\emptyset \vdash v : T'$ and $T' \equiv T$. We find an appropriate form for $\text{unref}(T')$ by the IH on $\emptyset \vdash v : T'$. We go by cases, in each case reproving whatever case was found in the IH and finding conversions by C_Trans.

> Case 1: $\text{unref}(T) = B$ and $v = k \in \mathcal{K}_B$. Since $\text{unref}(T') \equiv \text{unref}(T)$, we know that $\text{unref}(T') = B$, which is all we needed to show.
>
> Case 2a: $\text{unref}(T) = x{:}T_1 \to T_2$ and $v = \lambda x{:}T_1''.\ e_{12}$ and $T_1'' \equiv T_1$. Since $T' \equiv T$, we have $\text{unref}(T') \equiv \text{unref}(T)$ (Lemma 3.2.23) and so $\text{unref}(T') = x{:}T_1' \to T_2'$ for some T_1' and T_2' such that $T_1' \equiv T_1$ (Lemma 3.2.5); by C_Trans, we have $T_1'' \equiv T_1'$.
>
> Case 2b: $\text{unref}(T) = x{:}T_1 \to T_2$ and $v = \langle T_1' \Rightarrow T_2' \rangle^l$ and $T_1' \equiv T_1$ and $T_2' \equiv T_2$. Since $T' \equiv T$, we have $\text{unref}(T') \equiv \text{unref}(T)$ (Lemma 3.2.23) and so $\text{unref}(T') = x{:}T_1'' \to T_2''$ for some T_1'' and T_2'' such that $T_1'' \equiv T_1$ and $T_2'' \equiv T_2$ (Lemma 3.2.5); by C_Trans, we have $T_1' \equiv T_1''$ and $T_2' \equiv T_2''$ as required.
>
> Case 3: $\text{unref}(T) = \forall \alpha.\, T_0$ and v is $\Lambda \alpha.\ e$. Since $T' \equiv T$, then $\text{unref}(T') \equiv \text{unref}(T)$ (Lemma 3.2.23).

(T_Exact): $\emptyset \vdash v : \{x{:}T \mid e\}$; by inversion, $\emptyset \vdash v : T$. Noting that $\text{unref}(\{x{:}T \mid e\}) = \text{unref}(T)$, we apply the IH. Unlike the previous case, we need not change the conversion—it's in terms of the unrefined type.

(T_Forget): $\emptyset \vdash v : T$; by inversion $\emptyset \vdash v : \{x{:}T \mid e\}$. By the IH (noting $\text{unref}(\{x{:}T \mid e\}) = \text{unref}(T)$), so we use the IH's conversion directly. □

3.2.25 Theorem [Progress]: If $\emptyset \vdash e : T$, then either

1. $e \longrightarrow e'$, or
2. e is a result r, i.e., a value or blame.

Proof: By induction on the typing derivation.

(T_Var): Contradictory: there is no derivation $\emptyset \vdash x : T$.

(T_Const): $\emptyset \vdash k : \mathsf{ty}(k)$. In this case, $e = k$ is a result.

(T_Op): $\emptyset \vdash \mathrm{op}\,(e_1, \ldots, e_n) : \sigma(T)$, where $\mathsf{ty}(\mathrm{op}) = x_1 : T_1 \to \ldots \to x_n : T_n \to T$. By inversion, $\emptyset \vdash e_i : \sigma(T_i)$. Applying the IH from left to right, each of the e_i either steps or is a result.

Suppose everything to the left of e_i is a value. Then either e_i steps or is a result. If $e_i \longrightarrow e_i'$, then $\mathrm{op}\,(v_1, \ldots, v_{i-1}, e_i, \ldots, e_n) \longrightarrow \mathrm{op}(v_1, \ldots, v_{i-1}, e_i', \ldots, e_n)$ by E_Compat. One the other hand, if e_i is a result, there are two cases. If $e_i = \Uparrow l$, then the original expression steps to $\Uparrow l$ by E_Blame. If e_i is a value, we can continue this process for each of the operation's arguments. Eventually, all of the operations arguments are values. By value inversion (Lemma 3.2.11), we know that we can type each of these values at the exact refinement types we need by T_Exact. We assume that if $\mathrm{op}\,(v_1, \ldots, v_n)$ is well defined on values satisfying the refinements in its type, so E_Op applies.

(T_Abs): $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : (x{:}T_1 \to T_2)$. In this case, $e = \lambda x{:}T_1.\ e_{12}$ is a result.

(T_App): $\emptyset \vdash e_1\ e_2 : T_2[e_2/x]$; by inversion, $\emptyset \vdash e_1 : (x{:}T_1 \to T_2)$ and $\emptyset \vdash e_2 : T_1$.

By the IH on the first derivation, e_1 steps or is a result. If e_1 steps, then the entire term steps by E_Compat. In the latter case, if e_1 is blame, we step by E_Blame. So e_1 is a value, v_1.

By the IH on the second derivation, e_2 steps or is a result. If e_2 steps, then by E_Compat. Otherwise, if e_2 is blame, we step by E_Blame. So e_2 is a value, v_2.

By canonical forms (Lemma 3.2.24) on $\emptyset \vdash e_1 : (x{:}T_1 \to T_2)$, there are two cases:

> ($e_1 = \lambda x{:}T_1'.\ e_{12}$ and $T_1' \equiv T_1$): In this case, $(\lambda x{:}T_1'.\ e_{12})\ v_2 \longrightarrow e_{12}[v_2/x]$ by E_Beta.
>
> ($e_1 = \langle T_1' \Rightarrow T_2' \rangle^l$ and $T_1' \equiv T_1$ and $T_2' \equiv T_2$): We know that $T_1' \parallel T_2'$ by cast inversion (Lemma 3.2.20). We determine which step is taken by cases on T_1' and T_2'.
>
> > ($T_1' = B$):
> >
> > > ($T_2' = B'$): It must be the case that $B = B'$, since $B \parallel B'$. By E_Refl, $\langle B \Rightarrow B \rangle^l\ v_2 \longrightarrow v_2$.
> > >
> > > ($T_2' = \alpha$ or $x{:}T_{21} \to T_{22}$ or $\forall \alpha.\, T_{22}$): Incompatible; contradictory.
> > >
> > > ($T_2' = \{x{:}T_2'' \mid e\}$): If $T_2'' = B$, then by E_Check, $\langle B \Rightarrow \{x{:}B \mid e\} \rangle^l\ v_2 \longrightarrow \langle \{x{:}B \mid e\}, e[v_2/x], v_2 \rangle^l$. Otherwise, by E_PreCheck, we have:
> >
> > $$\langle B \Rightarrow \{x{:}T_2'' \mid e\} \rangle^l\ v_2 \longrightarrow \langle T_2'' \Rightarrow \{x{:}T_2'' \mid e\} \rangle^l\ (\langle B \Rightarrow T_2'' \rangle^l\ v_2)$$
>
> > ($T_1' = \alpha$):
> >
> > > ($T_2' = \alpha'$): It must be the case that $\alpha = \alpha'$, since $\alpha \parallel \alpha'$. By E_Refl, $\langle \alpha \Rightarrow \alpha \rangle^l\ v_2 \longrightarrow v_2$.
> > >
> > > ($T_2' = B$ or $x{:}T_{21} \to T_{22}$ or $\forall \alpha.\, T_{22}$): Incompatible; contradictory.

$\underline{(T_2' = \{x{:}T_2'' \mid e\})}$: If $T_2'' = \alpha$, then by E_Check, $\langle \alpha \Rightarrow \{x{:}\alpha \mid e\}\rangle^l\, v_2 \longrightarrow \langle \{x{:}\alpha \mid e\}, e[v_2/x], v_2\rangle^l$. Otherwise,

$$\langle \alpha \Rightarrow \{x{:}T_2'' \mid e\}\rangle^l\, v_2 \longrightarrow \langle T_2'' \Rightarrow \{x{:}T_2'' \mid e\}\rangle^l\, (\langle \alpha \Rightarrow T_2''\rangle^l\, v_2)$$

by E_PreCheck.

$\underline{(T_1' = x{:}T_{11} \to T_{12})}$:

$\underline{(T_2' = B \text{ or } \alpha \text{ or } \forall\alpha.T_{22})}$: Incompatible; contradictory.

$\underline{(T_2' = x{:}T_{21} \to T_{22})}$: If $T_1' = T_2'$, then $\langle T_1' \Rightarrow T_1'\rangle^l\, v_2 \longrightarrow v_2$ by E_Refl. If not, then

$$\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l\, v_2 \longrightarrow \\ \lambda x{:}T_{21}.\ (\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \Rightarrow T_{22}\rangle^l\, (v_2\, (\langle T_{21} \Rightarrow T_{11}\rangle^l\, x)))$$

by E_Fun.

$\underline{(T_2' = \{x{:}T_2'' \mid e\})}$: If $T_1' = T_2''$, then $\langle T_1' \Rightarrow \{x{:}T_1' \mid e\}\rangle^l\, v_2 \longrightarrow \langle \{x{:}T_1' \mid e\}, e[v_2/x], v_2\rangle^l$ by E_Check. If not, then

$$\langle T_1' \Rightarrow \{x{:}T_2'' \mid e\}\rangle^l\, v_2 \longrightarrow \langle T_2'' \Rightarrow \{x{:}T_2'' \mid e\}\rangle^l\, (\langle T_1' \Rightarrow T_2''\rangle^l\, v_2)$$

by E_PreCheck.

$\underline{(T_1' = \forall\alpha.T_{12})}$:

$\underline{(T_2' = B \text{ or } \alpha \text{ or } x{:}T_{21} \to T_{22})}$: Incompatible; contradictory.

$\underline{(T_2' = \forall\alpha.T_{22})}$: If $T_1' = T_2'$, then $\langle T_1' \Rightarrow T_2'\rangle^l\, v_2 \longrightarrow v_2$ by E_Refl. If not, then $\langle \forall\alpha.T_{11} \Rightarrow \forall\alpha.T_{22}\rangle^l\, v_2 \longrightarrow \Lambda\alpha.\ (\langle T_{11} \Rightarrow T_{22}\rangle^l\, (v_2\, \alpha))$ by E_Forall.

$\underline{(T_2' = \{x{:}T_2'' \mid e\})}$: If $T_1' = T_2''$, then $\langle T_1' \Rightarrow \{x{:}T_1' \mid e\}\rangle^l\, v_2 \longrightarrow \langle \{x{:}T_1' \mid e\}, e[v_2/x], v_2\rangle^l$ by E_Check. If not, then $\langle T_1' \Rightarrow \{x{:}T_2'' \mid e\}\rangle^l\, v_2 \longrightarrow \langle T_2'' \Rightarrow \{x{:}T_2'' \mid e\}\rangle^l\, (\langle T_1' \Rightarrow T_2''\rangle^l\, v_2)$ by E_PreCheck.

$\underline{(T_1' = \{x{:}T_1'' \mid e_1'\})}$:

$\underline{(T_2' = B \text{ or } \alpha \text{ or } x{:}T_{21} \to T_{22} \text{ or } \forall\alpha.T_{22})}$: We see

$$\langle \{x{:}T_1'' \mid e_1'\} \Rightarrow T_2'\rangle^l\, v_2 \longrightarrow \langle T_1'' \Rightarrow T_2'\rangle^l\, v_2$$

by E_Forget.

$\underline{(T_2' = \{x{:}T_2'' \mid e_2'\})}$: If $T_1' = T_2'$, then we immediately have $\langle T_1' \Rightarrow T_2'\rangle^l\, v_2 \longrightarrow v_2$ by E_Refl. If $T_1' = T_2''$, then

$$\langle T_1' \Rightarrow \{x{:}T_1' \mid e_2'\}\rangle^l\, v_2 \longrightarrow \langle \{x{:}T_1' \mid e_2'\}, e_2'[v_2/x], v_2\rangle^l$$

by E_Check. Otherwise,

$$\langle \{x{:}T_1'' \mid e_1'\} \Rightarrow \{x{:}T_2'' \mid e_2'\}\rangle^l\, v_2 \longrightarrow \langle T_1'' \Rightarrow \{x{:}T_2'' \mid e_2'\}\rangle^l\, v_2$$

by E_Forget.

(T_TAbs): $\emptyset \vdash \Lambda\alpha.\ e' : \forall\alpha.T$. In this case, $\Lambda\alpha.\ e'$ is a result.

(T_TApp): $\emptyset \vdash e_1\ T_2 : T_1[T_2/\alpha]$; by inversion, $\emptyset \vdash e_1 : \forall\alpha.T_1$ and $\emptyset \vdash T_2$. By the IH on the first derivation, e_1 steps or is a result. If $e_1 \longrightarrow e_1'$, then $e_1\ T_2 \longrightarrow e_1'\ T_2$ by E_Compat. If $e_1 = \Uparrow l$, then $\Uparrow l\ T_2 \longrightarrow \Uparrow l$ by E_Blame.

If $e_1 = v_1$, then we know that $v_1 = \Lambda\alpha.\ e_1'$ by canonical forms (Lemma 3.2.24). We can see $(\Lambda\alpha.\ e_1')\ T_2 \longrightarrow e_1'[T_2/\alpha]$ by E_TBeta.

(T_Cast): $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l : T_1 \to T_2$. In this case, $\langle T_1 \Rightarrow T_2\rangle^l$ is a result.

(T_Check): $\emptyset \vdash \langle\{x{:}T \mid e_1\}, e_2, v\rangle^l : \{x{:}T \mid e_1\}$; by inversion, $\emptyset \vdash e_2 : \mathsf{Bool}$. By the IH, either $e_2 \longrightarrow e_2'$ steps or $e_2 = r_2$. In the first case, $\langle\{x{:}T \mid e_1\}, e_2, v\rangle^l \longrightarrow \langle\{x{:}T \mid e_1\}, e_2', v\rangle^l$ by E_Compat. In the second case, either $r_2 = \Uparrow l$ or $r_2 = v_2$. If we have blame, then the entire term steps by E_Blame. If we have a value, then we know that v_2 is either true or false, since it's typed at Bool. If it's true, we step by E_OK. Otherwise we step by E_Fail.

(T_Blame): $\emptyset \vdash \Uparrow l : T$. In this case, $\Uparrow l$ is a result.

(T_Conv): $\emptyset \vdash e : T'$; by inversion, $\emptyset \vdash e : T$. By the IH, we see that $e \longrightarrow e'$ or $e = r$.

(T_Exact): $\emptyset \vdash v : \{x{:}T \mid e\}$. Here, v is a result by assumption.

(T_Forget): $\emptyset \vdash v : T$. Again, v is a result by assumption. □

□

The following regularity property formalizes an important property of the type system: all contexts and types involved are well formed. This is critical for the proof of preservation: when a term raises blame, we must show that the blame is well typed. With regularity, we can immediately know that the original type is well formed.

3.2.26 Lemma [Context and type well formedness]:
1. If $\Gamma \vdash e : T$, then $\vdash \Gamma$ and $\Gamma \vdash T$.
2. If $\Gamma \vdash T$ then $\vdash \Gamma$.

Proof: By induction on the typing and well formedness derivations. □

3.2.27 Theorem [Preservation]: If $\emptyset \vdash e : T$ and $e \longrightarrow e'$, then $\emptyset \vdash e' : T$.

Proof: By induction on the typing derivation.

(T_Var): Contradictory—we can't have $\emptyset \vdash x : T$.

(T_Const): $\emptyset \vdash k : \mathsf{ty}(k)$. Contradictory—values don't step.

(T_Op): $\emptyset \vdash \mathrm{op}\,(e_1, \ldots, e_n) : \sigma(T)$. By cases on the step taken:

> (E_Reduce/E_Op): $\mathrm{op}\,(v_1, \ldots, v_n) \longrightarrow [\![\mathrm{op}]\!]\,(v_1, \ldots, v_n)$. This case is by assumption.

(E_Blame): $e_i = \Uparrow l$, and everything to its left is a value. By context and type well formedness (Lemma 3.2.26), $\emptyset \vdash \sigma(T)$. So by T_Blame, $\emptyset \vdash \Uparrow l : \sigma(T)$.

(E_Compat): Some $e_i \longrightarrow e_i'$. By the IH and T_Op, using T_Conv to show that $\sigma(T) \equiv \sigma'(T)$ (Lemma 3.2.8).

(T_Abs): $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : (x{:}T_1 \to T_2)$. Contradictory—values don't step.

(T_App): $\emptyset \vdash e_1\, e_2 : T_2'[e_2/x]$, with $\emptyset \vdash e_1 : (x{:}T_1' \to T_2')$ and $\emptyset \vdash e_2 : T_1'$, by inversion. By cases on the step taken.

(E_Reduce/E_Beta): $(\lambda x{:}T_1.\ e_{12})\, v_2 \longrightarrow e_{12}[v_2/x]$. First, we have $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : (x{:}T_1' \to T_2')$. By inversion for lambdas (Lemma 3.2.19), $x{:}T_1 \vdash e_{12} : T_2$. Moreover, $x{:}T_1 \to T_2 \equiv x{:}T_1' \to T_2'$, which means $T_2 \equiv T_2'$ (Lemma 3.2.5).

By substitution, $\emptyset \vdash e_{12}[v_2/x] : T_2[v_2/x]$. We then see that $T_2[v_2/x] \equiv T_2'[v_2/x]$ (Lemma 3.2.8), so T_Conv completes this case.

(E_Reduce/E_Refl): $\langle T \Rightarrow T\rangle^l\, v_2 \longrightarrow v_2$. By cast inversion (Lemma 3.2.20), $\_{:}T \to T \equiv x{:}T_1' \to T_2'$ and $\emptyset \vdash T$. In particular, we have $T \equiv T_2'$ and $T \equiv T_1'$ (Lemma 3.2.5). By substitutivity of conversion (Lemma 3.2.8), $T[v_2/x] \equiv T_2'[v_2/x]$. Since T is closed, we really know that $T \equiv T_2'[v_2/x]$.

By C_Sym and C_Trans, we have $T_1' \equiv T \equiv T_2'[v_2/x]$. By T_Conv on $\emptyset \vdash v_2 : T_1'$, we have $\emptyset \vdash v : T_2'[v_2/x]$.

(E_Reduce/E_Forget): $\langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle^l\, v_2 \longrightarrow \langle T_1 \Rightarrow T_2\rangle^l\, v_2$. We restate the typing judgment and its inversion:

$$\begin{array}{l}\emptyset \vdash \langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle^l\, v_2 : T_2'[v_2/y] \\ \emptyset \vdash \langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle^l : (y{:}T_1' \to T_2') \\ \emptyset \vdash v_2 : T_1'\end{array}$$

By cast inversion (Lemma 3.2.20), we know that $\emptyset \vdash T_1$ and $\emptyset \vdash T_2$—as well as $\_{:}\{x{:}T_1 \mid e\} \to T_2 \equiv y{:}T_1' \to T_2'$ and $\{x{:}T_1 \mid e\} \parallel T_2$. Inverting this conversion (Lemma 3.2.5), finding $\{x{:}T_1 \mid e\} \equiv T_1'$ and $T_2 \equiv T_2'$. Then by T_Conv and C_Sym, $\emptyset \vdash v_2 : \{x{:}T_1 \mid e\}$; by T_Forget, $\emptyset \vdash v_2 : T_1$.

By T_Cast, we have $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l : y{:}T_1 \to T_2$, with $T_1 \parallel T_2$ iff $\{x{:}T_1 \mid e\} \parallel T_2$. (Note, however, that y does not appear in T_2—we write it to clarify the substitutions below.)

By T_App, we find $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l\, v_2 : T_2[v_2/y]$. Since $T_2 \equiv T_2'$, we have $T_2[v_2/y] \equiv T_2'[v_2/y]$ by Lemma 3.2.8. We are done by T_Conv.

(E_Reduce/E_PreCheck):

$$\begin{array}{l}\langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l\, v_2 \longrightarrow \\ \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l\, (\langle T_1 \Rightarrow T_2\rangle^l\, v_2)\end{array}$$

We restate the typing judgment and its inversion:

$$\begin{array}{l}\emptyset \vdash \langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l\, v_2 : T_2'[v_2/y] \\ \emptyset \vdash \langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l : y{:}T_1' \to T_2' \\ \emptyset \vdash v_2 : T_1'\end{array}$$

By cast inversion (Lemma 3.2.20), $\emptyset \vdash T_1$ and $\emptyset \vdash \{x{:}T_2 \mid e\}$, and $y{:}T_1 \to \{x{:}T_2 \mid e\} \equiv y{:}T_1' \to T_2'$. Also, $T_1 \parallel \{x{:}T_2 \mid e\}$.

By inversion on $\emptyset \vdash \{x{:}T_2 \mid e\}$, we find $\emptyset \vdash T_2$. Next, $T_1 \parallel T_2$ iff $T_1 \parallel \{x{:}T_2 \mid e\}$. Now by T_Cast, we find $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l : y{:}T_1 \to T_2$. Note, however, that y doesn't occur in T_2.

We can take the convertible function types and see that their parts are convertible: $T_1 \equiv T_1'$ and $\{x{:}T_2 \mid e\} \equiv T_2'$. Using the first conversion, we find $\emptyset \vdash v_2 : T_1$ by T_Conv. By T_App, $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l\, v_2 : T_2[v_2/y]$, where $T_2[v_2/y] = T_2$.

By C_Refl and C_RefineR, $T_2 \parallel \{x{:}T_2 \mid e\}$. We have well formedness derivations for both types, as well, so $\emptyset \vdash \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l : y{:}T_2 \to \{x{:}T_2 \mid e\}$ by T_Cast. Again, y does not appear in e or T_2. By T_App, we have $\emptyset \vdash \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l\, (\langle T_1 \Rightarrow T_2\rangle^l\, v_2) : \{x{:}T_2 \mid e\}[\langle T_1 \Rightarrow T_2\rangle^l\, v_2/y]$.

Since y isn't in $\{x{:}T_2 \mid e\}$, we can see:

$$\{x{:}T_2 \mid e\}[\langle T_1 \Rightarrow T_2\rangle^l\, v_2/y] = \{x{:}T_2 \mid e\} = \{x{:}T_2 \mid e\}[v_2/y]$$

By substitutivity of conversion (Lemma 3.2.8), we have $\{x{:}T_2 \mid e\}[v_2/y] \equiv T_2'[v_2/y]$. We can now apply T_Conv to find $\emptyset \vdash \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l\, (\langle T_1 \Rightarrow T_2\rangle^l\, v_2) : T_2'[v_2/y]$.

(E_Reduce/E_Check): $\langle T \Rightarrow \{x{:}T \mid e\}\rangle^l\, v_2 \longrightarrow \langle \{x{:}T \mid e\}, e[v_2/x], v_2\rangle^{l'}$.
We restate the typing judgment with its inversion:

$$\begin{array}{l}\emptyset \vdash \langle T \Rightarrow \{x{:}T \mid e\}\rangle^l\, v_2 : T_2'[v_2/y] \\ \emptyset \vdash \langle T \Rightarrow \{x{:}T \mid e\}\rangle^l : y{:}T_1' \to T_2' \\ \emptyset \vdash v_2 : T_1'\end{array}$$

By cast inversion (Lemma 3.2.20), $\emptyset \vdash \{x{:}T \mid e\}$ and $\emptyset \vdash T$. Moreover, $y{:}T \to \{x{:}T \mid e\} \equiv y{:}T_1' \to T_2'$, where y doesn't occur in $\{x{:}T \mid e\}$. This means that $T \equiv T_1'$ and $\{x{:}T \mid e\} \equiv T_2'$.

Using T_Conv and C_Sym with the first conversion shows $\emptyset \vdash v_2 : T$. By inversion on $\emptyset \vdash \{x{:}T \mid e\}$, we see $x{:}T \vdash e : \mathsf{Bool}$. By term substitution (Lemma 3.2.16), we find $\emptyset \vdash e[v_2/x] : \mathsf{Bool}$. Finally, $e[v_2/x] \longrightarrow^* e[v_2/x]$ by reflexivity (Lemma 3.2.3).

T_Check (with WF_Empty) shows $\emptyset \vdash \langle \{x{:}T \mid e\}, e[v_2/x], v_2\rangle^l : \{x{:}T \mid e\}$. By substitutivity of conversion (Lemma 3.2.8), $\{x{:}T \mid e\}[v_2/y] \equiv T_2[v_2/y]$.

Since y doesn't occur in $\{x{:}T \mid e\}$, we know that $\{x{:}T \mid e\}[v_2/y] = \{x{:}T \mid e\}$, so we can show that $\{x{:}T \mid e\} \equiv T_2[v_2/y]$ by C_Sym, and now $\emptyset \vdash \langle \{x{:}T \mid e\}, e[v_2/x], v_2 \rangle^l : T_2[v_2/y]$ by T_Conv.

(E_Reduce/E_Fun):

$$\begin{array}{l}\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22} \rangle^l \, v_2 \longrightarrow \\ \lambda x{:}T_{21}.\ (\langle T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l \, x/x] \Rightarrow T_{22} \rangle^l \, (v_2 \, (\langle T_{21} \Rightarrow T_{11} \rangle^l \, x)))\end{array}$$

We restate the typing judgment with its inversion:

$$\begin{array}{l}\emptyset \vdash \langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22} \rangle^l \, v_2 : T_2'[v_2/y] \\ \emptyset \vdash \langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22} \rangle^l : (y{:}T_1' \to T_2') \\ \emptyset \vdash v_2 : T_1'\end{array}$$

By cast inversion on the first derivation:

$$\begin{array}{ll}\emptyset \vdash x{:}T_{11} \to T_{12} & \emptyset \vdash x{:}T_{21} \to T_{22} \\ x{:}T_{11} \to T_{12} \parallel x{:}T_{21} \to T_{22} & \\ \_{:}(x{:}T_{11} \to T_{12}) \to (x{:}T_{21} \to T_{22}) \equiv y{:}T_1' \to T_2' & \end{array}$$

By inversion of this last (Lemma 3.2.5):

$$x{:}T_{11} \to T_{12} \equiv T_1' \qquad\qquad x{:}T_{21} \to T_{22} \equiv T_2'$$

So by T_Conv and C_Sym, we have $\emptyset \vdash v_2 : x{:}T_{11} \to T_{12}$. By weakening (Lemma 3.2.12), $x{:}T_{21} \vdash v_2 : x{:}T_{11} \to T_{12}$.

By inversion of the well formedness of the function types:

$$\emptyset \vdash T_{11} \qquad x{:}T_{11} \vdash T_{12} \qquad\qquad \emptyset \vdash T_{21} \qquad x{:}T_{21} \vdash T_{22}$$

By weakening (Lemma 3.2.12), we find $x{:}T_{21} \vdash T_{11}$ and $x{:}T_{21} \vdash T_{21}$. By compatibility:

$$T_{11} \parallel T_{21} \qquad\qquad T_{12} \parallel T_{22}$$

By T_Cast, $x{:}T_{21} \vdash \langle T_{21} \Rightarrow T_{11} \rangle^l : (\_{:}T_{21} \to T_{11})$ (notice that compatibility is symmetric, per Lemma 3.2.14). By T_App and T_Var, we can see $x{:}T_{21} \vdash \langle T_{21} \Rightarrow T_{11} \rangle^l \, x : T_{11}[x/\_] = T_{11}$. Again by T_App, we have $x{:}T_{21} \vdash v_2 \, (\langle T_{21} \Rightarrow T_{11} \rangle^l \, x) : T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l \, x/x]$. By weakening (Lemma 3.2.12) and substitution (Lemma 3.2.16), we have the following two derivations:

$$x{:}T_{21} \vdash T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l \, x/x] \qquad\qquad x{:}T_{21} \vdash T_{22}$$

By T_Cast and Lemma 3.2.15:

$$x{:}T_{21} \vdash \langle T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l \, x/x] \Rightarrow T_{22} \rangle^l : (y{:}T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l \, x/x] \to T_{22})$$

Noting that y is free here. By T_App:

$$\begin{array}{rl} x{:}T_{21} \vdash & \langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \Rightarrow T_{22}\rangle^l\,(v_2\,(\langle T_{21} \Rightarrow T_{11}\rangle^l\, x)) \\ : & T_{22}[v_2\,(\langle T_{21} \Rightarrow T_{11}\rangle^l\, x)/y] (= T_{22}) \end{array}$$

Finally, by T_Abs:

$$\emptyset \vdash \lambda x{:}T_{21}.\ \langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \Rightarrow T_{22}\rangle^l\,(v_2\,(\langle T_{21} \Rightarrow T_{11}\rangle^l\, x)) : x{:}T_{21} \to T_{22}$$

Since y isn't in $x{:}T_{21} \to T_{22}$, we can see that $x{:}T_{21} \to T_{22} = (x{:}T_{21} \to T_{22})[v_2/y]$. Using this fact with substitutivity of conversion (Lemma 3.2.8), we find $x{:}T_{21} \to T_{22} \equiv T_2'[v_2/y]$. So—finally—by T_Conv we have:

$$\emptyset \vdash \lambda x{:}T_{21}.\ \langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \Rightarrow T_{22}\rangle^l\,(v_2\,(\langle T_{21} \Rightarrow T_{11}\rangle^l\, x)) : T_2'[v_2/y]$$

(E_Reduce/E_Forall): $\langle \forall\alpha.T_1 \Rightarrow \forall\alpha.T_2\rangle^l\, v_2 \longrightarrow (\Lambda\alpha.\ \langle T_1 \Rightarrow T_2\rangle^l\,(v\,\alpha))$
We restate the typing and its inversion:

$$\begin{array}{l} \emptyset \vdash \langle \forall\alpha.T_1 \Rightarrow \forall\alpha.T_2\rangle^l\, v_2 : T_2'[v_2/x] \\ \emptyset \vdash \langle \forall\alpha.T_1 \Rightarrow \forall\alpha.T_2\rangle^l : x{:}T_1' \to T_2' \\ \emptyset \vdash v_2 : T_1' \end{array}$$

By cast inversion (Lemma 3.2.20):

$$\begin{array}{ll} \emptyset \vdash \forall\alpha.T_1 & \emptyset \vdash \forall\alpha.T_2 \\ \forall\alpha.T_1 \parallel \forall\alpha.T_2 & \\ \_{:}(\forall\alpha.T_1) \to (\forall\alpha.T_2) \equiv x{:}T_1' \to T_2' & \end{array}$$

By inversion of this last $\forall\alpha.T_1 \equiv T_1'$ and $\forall\alpha.T_2 \equiv T_2'$ (Lemma 3.2.5). By T_Conv and C_Sym, $\emptyset \vdash v_2 : \forall\alpha.T_1$. By type variable weakening, WF_TVar, and T_TApp, we have:

$$\alpha \vdash v_2\,\alpha : T_1[\alpha/\alpha] = T_1$$

By inversion of the universal types's well formedness and compatibility:

$$\alpha \vdash T_1 \qquad \alpha \vdash T_2 \qquad T_1 \parallel T_2$$

So by T_Cast, $\alpha \vdash \langle T_1 \Rightarrow T_2\rangle^l : (x{:}T_1 \to T_2)$, noting that x doesn't occur in T_2. By T_App, $\alpha \vdash \langle T_1 \Rightarrow T_2\rangle^l\,(v_2\,\alpha) : T_2[v_2\,\alpha/x] = T_2$. By T_TAbs, $\emptyset \vdash \Lambda\alpha.\ (\langle T_1 \Rightarrow T_2\rangle^l\,(v\,\alpha)) : \forall\alpha.T_2$.

We know that $\forall\alpha.T_2 \equiv T_2'$, so by type variable substitutivity of conversion (Lemma 3.2.9), $(\forall\alpha.T_2)[v_2/x] \equiv T_2'[v_2/x]$. Since x isn't in $\forall\alpha.T_2$, we know that $\forall\alpha.T_2 \equiv T_2'[v_2/x]$ (by way of Lemma 3.2.8).. Now we can see by T_Conv that $\emptyset \vdash \Lambda\alpha.\ (\langle T_1 \Rightarrow T_2\rangle^l\,(v\,\alpha)) : T_2'[v_2/x]$.

(E_Compat): $E\,[e] \longrightarrow E\,[e']$ when $e \longrightarrow e'$ By cases on E:

($E = [\,]\ e_2,\ e_1 \longrightarrow e_1'$): By the IH and T_App.

($E = v_1\ [\,],\ e_2 \longrightarrow e_2'$): By the IH, T_App, and T_Conv, since $T_2[e_2/x] \equiv T_2[e_2'/x]$ by reflexivity (Lemma 3.2.3) and substitutivity (Lemma 3.2.8).

(E_Blame): $E\,[\Uparrow l] \longrightarrow \Uparrow l$ $\emptyset \vdash E\,[\Uparrow l] : T$ by assumption. By type well formedness (Lemma 3.2.26), we know that $\emptyset \vdash T$. We then have $\emptyset \vdash \Uparrow l : T$ by T_Blame.

(T_TAbs): $\emptyset \vdash \Lambda\alpha.\ e : \forall\alpha.T$. This case is contradictory—values don't step.

(T_TApp): $\emptyset \vdash e\ T : T'[T/\alpha]$. By cases on the step taken.

(E_Reduce/E_TBeta): $(\Lambda\alpha.\ e')\ T \longrightarrow e'[T/\alpha]$ We restate the typing derivation and its inversion:

$$\emptyset \vdash (\Lambda\alpha.\ e')\ T : T'[T/\alpha] \qquad \emptyset \vdash \Lambda\alpha.\ e' : \forall\alpha.T' \qquad \emptyset \vdash T$$

By type abstraction inversion (Lemma 3.2.22): $\alpha \vdash e' : T''$ and $\forall\alpha.T'' \equiv \forall\alpha.T'$; by inversion of this last (Lemma 3.2.7), $T'' \equiv T'$.

By type variable substitution (Lemma 3.2.18), $\emptyset \vdash e'[T/\alpha] : T''[T/\alpha]$. By type substitutivity of conversion (Lemma 3.2.9), $T''[T/\alpha] \equiv T'[T/\alpha]$. T_Conv gives us $\emptyset \vdash e'[T/\alpha] : T'[T/\alpha]$ as desired.

(E_Compat): $E\,[e] \longrightarrow E\,[e']$, where $E = [\,]\ T$. By the IH and T_TApp.

(E_Blame): $E\,[\Uparrow l] \longrightarrow \Uparrow l$. $\emptyset \vdash E\,[\Uparrow l] : T$ by assumption. By type well formedness (Lemma 3.2.26), we know that $\emptyset \vdash T$. So we see $\emptyset \vdash \Uparrow l : T$ by T_Blame.

(T_Cast): $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l : T_1 \to T_2$. This case is contradictory—values don't step.

(T_Check): $\emptyset \vdash \langle\{x{:}T \mid e_1\}, e_2, v\rangle^l : \{x{:}T \mid e_1\}$. By cases on the step taken.

(E_Reduce/E_OK): $\langle\{x{:}T \mid e_1\}, \mathsf{true}, v\rangle^l \longrightarrow v$. By inversion, $\emptyset \vdash v : T$ and $\emptyset \vdash \{x{:}T \mid e\}$; we also have $e_1[v/x] \longrightarrow^* \mathsf{true}$. By WF_Empty and the assumption that $e[v/x] \longrightarrow^* \mathsf{true}$, we can find $\emptyset \vdash v : \{x{:}T \mid e\}$ by T_Exact.

(E_Reduce/E_Fail): $\langle\{x{:}T \mid e_1\}, \mathsf{false}, v\rangle^l \longrightarrow \Uparrow l$ We have $\emptyset \vdash \{x{:}T \mid e\}$ by inversion. By WF_Empty and T_Blame, $\emptyset \vdash \Uparrow l : \{x{:}T \mid e\}$.

(E_Compat): $E\,[e] \longrightarrow E\,[e']$, where $E = \langle\{x{:}T \mid e_1\}, [\,], v\rangle^l$. By the IH on e, we know that $\emptyset \vdash e' : \mathsf{Bool}$. We still have $\emptyset \vdash \{x{:}T \mid e_1\}$ and $\emptyset \vdash v : T$ from our original derivation. Since $e_1[v/x] \longrightarrow^* e$ and $e \longrightarrow e'$, then $e_1[v/x] \longrightarrow^* e'$. Therefore, $\emptyset \vdash \langle\{x{:}T \mid e_1\}, e', v\rangle^l : \{x{:}T \mid e_1\}$ by T_Check.

(E_Blame): $E[\Uparrow l] \longrightarrow \Uparrow l$. $\emptyset \vdash E[\Uparrow l] : T$ by assumption. By type well formedness (Lemma 3.2.26), we know that $\emptyset \vdash T$. So $\emptyset \vdash \Uparrow l : T$ by T_Blame.

(T_Blame): $\emptyset \vdash \Uparrow l : T$. This case is contradictory—blame doesn't step.

(T_Conv): $\emptyset \vdash e : T'$; by inversion we have $\emptyset \vdash e : T$ and $T \equiv T'$ and $\emptyset \vdash T'$ (and, trivially, $\vdash \emptyset$). By the IH on the first derivation, we know that $\emptyset \vdash e' : T$. By T_Conv, we can see that $\emptyset \vdash e' : T'$.

(T_Exact): $\emptyset \vdash v : \{x{:}T \mid e\}$. This case is contradictory—values don't step.

(T_Forget): $\emptyset \vdash v : T$. This case is contradictory—values don't step. □

□

Requiring standard weakening, substitution, and inversion lemmas, the syntactic proof of type soundness is straightforward. It is easy to restrict F_H to a simply typed calculus with a similar type soundness proof. In fact, after cutting out universal types and restricting refinements to base types, it's possible to simplify the operational semantics and to do away with the T_Forget rule. I don't give the proof here because it is subsumed by type soundness in F_H; the Cast system in Chapter 4 follows this principle, though it uses explicit tags and also has the dynamic type Dyn.

3.3 Parametricity

I prove relational parametricity for two reasons: (1) it yields powerful reasoning techniques such as free theorems [76] and our subtyping proof in Section 3.4, and (2) it indicates that contracts don't interfere with type abstraction, i.e., that F_H supports polymorphism in the same way that System F does. The proof is mostly standard: I define a (syntactic) logical relation on terms and types, where each type is interpreted as a relation on terms and the relation at type variables is given as a parameter. In the next section, I define a subtype relation and show that an upcast—a cast whose source type is a subtype of the target type—is logically related to the identity function. I conjecture that logically related terms are *contextually equivalent* and upcasts can be eliminated without changing the meaning of a program.

We begin by defining two relations: $r_1 \sim r_2 : T; \theta; \delta$ relates closed results, defined by induction on types; $e_1 \simeq e_2 : T; \theta; \delta$ relates closed expressions which evaluate to results in the first relation. The definitions are shown in Figure 3.5.[2] Both relations have three indices: a type T, a substitution θ for type variables, and a substitution δ for term variables. A type substitution θ, which gives the interpretation of free type variables in T, maps a type variable to a triple (R, T_1, T_2) comprising a binary relation R on terms and two closed types T_1 and T_2. A term substitution δ maps

[2]To save space, I write $\Uparrow l \sim \Uparrow l : T; \theta; \delta$ separately instead of manually adding such a clause for each type.

Closed terms $\boxed{r_1 \sim r_2 : T; \theta; \delta}$ $\qquad$ $\boxed{e_1 \simeq e_2 : T; \theta; \delta}$

$$
\begin{array}{rcl}
k \sim k : B; \theta; \delta & \iff & k \in \mathcal{K}_B \\
v_1 \sim v_2 : \alpha; \theta; \delta & \iff & \exists R\, T_1\, T_2,\ \alpha \mapsto R,\, T_1,\, T_2 \in \theta \wedge v_1\ R\ v_2 \\
v_1 \sim v_2 : (x{:}T_1 \to T_2); \theta; \delta & \iff & \forall v_1' \sim v_2' : T_1; \theta; \delta,\ v_1\ v_1' \simeq v_2\ v_2' : T_2; \theta; \delta[v_1', v_2'/x] \\
v_1 \sim v_2 : \forall \alpha. T; \theta; \delta & \iff & \forall R\, T_1\, T_2,\ v_1\ T_1 \simeq v_2\ T_2 : T; \theta[\alpha \mapsto R, T_1, T_2]; \delta \\
v_1 \sim v_2 : \{x{:}T \mid e\}; \theta; \delta & \iff & v_1 \sim v_2 : T; \theta; \delta \wedge \\
 & & \theta_1(\delta_1(e))[v_1/x] \longrightarrow^* \mathsf{true} \wedge \theta_2(\delta_2(e))[v_2/x] \longrightarrow^* \mathsf{true} \\
\Uparrow l \sim \Uparrow l : T; \theta; \delta & & \\
 & & \\
e_1 \simeq e_2 : T; \theta; \delta & \iff & \exists r_1\, r_2,\, e_1 \longrightarrow^* r_1 \wedge e_2 \longrightarrow^* r_2 \wedge r_1 \sim r_2 : T; \theta; \delta \\
\multicolumn{3}{c}{R_{T,\theta,\delta} = \{(r_1, r_2) \mid r_1 \sim r_2 : T; \theta; \delta\}}
\end{array}
$$

Types $\boxed{T_1 \simeq T_2 : *; \theta; \delta}$

$$
\begin{array}{rcl}
B \simeq B : *; \theta; \delta & & \\
\alpha \simeq \alpha : *; \theta; \delta & & \\
x{:}T_{11} \to T_{12} \simeq x{:}T_{21} \to T_{22} : *; \theta; \delta & \iff & T_{11} \simeq T_{21} : *; \theta; \delta \wedge \\
 & & \forall v_1 \sim v_2 : T_{11}; \theta; \delta, \\
 & & \quad T_{12} \simeq T_{22} : *; \theta; \delta[v_1, v_2/x] \\
\forall \alpha. T_1 \simeq \forall \alpha. T_2 : *; \theta; \delta & \iff & \forall R\, T_1'\, T_2',\ T_1 \simeq T_2 : *; \theta[\alpha \mapsto R, T_1', T_2']; \delta \\
\{x{:}T_1 \mid e_1\} \simeq \{x{:}T_2 \mid e_2\} : *; \theta; \delta & \iff & T_1 \simeq T_2 : *; \theta; \delta \wedge \\
 & & \forall v_1 \sim v_2 : T_1; \theta; \delta,\ \theta_1(\delta_1(e_1))[v_1/x] \simeq \theta_2(\delta_2(e_2))[v_2/x] : \mathsf{Bool}; \theta; \delta
\end{array}
$$

Open terms and types $\boxed{\Gamma \vdash \theta; \delta}$ $\qquad$ $\boxed{\Gamma \vdash e_1 \simeq e_2 : T}$ $\qquad$ $\boxed{\Gamma \vdash T_1 \simeq T_2 : *}$

$$
\begin{array}{rcl}
\Gamma \vdash \theta; \delta & \iff & \forall x{:}T \in \Gamma,\ \theta_1(\delta_1(x)) \simeq \theta_2(\delta_2(x)) : T; \theta; \delta \wedge \\
 & & \forall \alpha \in \Gamma, \exists R\, T_1\, T_2,\ \alpha \mapsto R,\, T_1,\, T_2 \in \theta \\
\Gamma \vdash e_1 \simeq e_2 : T & \iff & \forall \Gamma \vdash \theta; \delta,\ \theta_1(\delta_1(e_1)) \simeq \theta_2(\delta_2(e_2)) : T; \theta; \delta \\
\Gamma \vdash T_1 \simeq T_2 : * & \iff & \forall \Gamma \vdash \theta; \delta,\ T_1 \simeq T_2 : *; \theta; \delta
\end{array}
$$

Figure 3.5: The logical relation for parametricity

from variables to pairs of closed values. I write projections δ_i $(i = 1, 2)$ to denote projections from this pair. I similarly write θ_i $(i = 1, 2)$ for a substitution that maps a type variable α to T_i where $\theta(\alpha) = (R, T_1, T_2)$.

With these definitions out of the way, the term relation is mostly straightforward. First, $\Uparrow l$ is related to itself at every type. A base type B gives the identity relation on $\mathcal{K}_B$, the set of constants of type B. A type variable α simply uses the relation assumed in the substitution θ. Related functions map related arguments to related results. Type abstractions are related when their bodies are parametric in the interpretation of the type variable. Finally, two values are related at a refinement type when they are related at the underlying type and both satisfy the predicate; here, the predicate e gets closed by applying the substitutions. I require that both values satisfy their refinements rather than having the first satisfy the predicate iff the second does because I want to know that values related at refinement types *actually inhabit those types*, i.e., actually satisfy the predicates of the refinement. The $\sim$ relation on results is extended to the relation $\simeq$ on closed terms in a straightforward manner:

terms are related if and only if they both terminate at related results. Divergent terms are not related to each other—though we will discover that divergent terms do not exist in F_H. I extend the relation to open terms, written $\Gamma \vdash e_1 \simeq e_2 : T$, relating open terms that are related when closed by any "Γ-respecting" pair of substitutions θ and δ (written $\Gamma \vdash \theta; \delta$, also defined in Figure 3.5).

To show that (well-typed) casts yield related results when applied to related inputs, we also need a relation on types $T_1 \simeq T_2 : *; \theta; \delta$; I define this relation in Figure 3.5. We can use the logical relation on terms to handle the arguments of function types and refinement types. Note that the T_1 and T_2 in this relation are not necessarily closed; terms in refinement types, which should be related at Bool, are closed by applying substitutions. In the function and refinement type cases, the relation on a smaller type is universally quantified over logically related values. There are two choices of the type at which they should be related (for example, the second line of the function type case could change T_{11} to T_{21}). It does not really matter which side we choose, since they are related types. Here, we are "left-leaning";[3] in the proof, I justify this choice by proving a "type exchange" lemma (Lemma 3.4.2) that lets us replace a type index T_1 in the term relation by T_2 when $T_1 \simeq T_2 : *$. Finally, we lift the type relation to open terms, writing $\Gamma \vdash T_1 \simeq T_2 : *$ when two types are equivalent for any Γ-respecting substitutions.

It is worth discussing two points peculiar to this formulation: terms in the logical relation are untyped, and the type indices are open.

I allow any relation on terms to be used in θ; terms related at T need not be well typed at T. The standard formulation of a logical relation is well typed throughout, requiring that the relation R in every triple be well typed, only relating values of type T_1 to values of type T_2 (e.g., [54]). I suspect that part of the reason this proof of parametricity works is its similarity to Girard's untyped reducibility candidates. I have two motivations for leaving the relations untyped. First, functions of type $x{:}T_1 \to T_2$ must map related values ($v_1 \sim v_2 : T_1$) to related results...but at what type? While $T_2[v_1/x]$ and $T_2[v_2/x]$ are related in the type relation, terms that are well typed at one type won't necessarily be well typed at the other, whether definitions are left- or right-leaning. Second, we prove in Section 3.4 that upcasts have no effect: if $T_1 <: T_2$, then $\langle T_1 \Rightarrow T_2 \rangle^l \sim \lambda x{:}T_1.\ x : T_1 \to T_2$. That is, I want a cast $\langle T_1 \Rightarrow T_2 \rangle^l$, of type $T_1 \to T_2$, to be related to the identity $\lambda x{:}T_1.\ x$, of type $T_1 \to T_1$. There is one small hitch: $\lambda x{:}T_1.\ x$ has type $T_1 \to T_1$, not $T_1 \to T_2$! I therefore don't demand that two expressions related at T be well typed at T, and I allow *any* relation to be chosen as R.

The type indices of the term relation are not necessarily closed. Instead, just as the interpretation of free type variables in the logical relation's type index are kept in a substitution θ, we keep δ as a substitution for the free term variables that can appear in type indices. Keeping this substitution separate avoids a problem in defining the logical relation at function types. Consider a function type $x{:}T_1 \to T_2$:

[3] I, too, lean quite far to the left.

the *logical* relation says that values v_1 and v_2 are related at this type when they take related values to related results, i.e. if $v_1' \sim v_2' : T_1; \theta; \delta$, then we should be able to find $v_1\, v_1' \simeq v_2\, v_2'$ at some type. The question here is which type index we should use. If we keep type indices closed (with respect to term variables), we cannot use T_2 on its own—we have to choose a binding for x! Knowles and Flanagan [44] deal with this problem by introducing the "wedge product" operator, which merges two types—one with v_1' substituted for x and the other with v_2' for x—into one. Instead of substituting eagerly, we put both bindings in δ and apply them when needed—the refinement type case. I think this formulation is more uniform with regard to free term/type variables, since eager substitution is a non-starter for type variables, anyway.

As Atsushi and I developed the proof, we found that the E_Refl rule $\langle T \Rightarrow T \rangle^l\, v \rightsquigarrow v$ is not just a convenient way to skip decomposing a trivial cast into smaller trivial casts (when T is a polymorphic or dependent function type); E_Refl is, in fact, crucial to obtaining parametricity in this syntactic setting. On the one hand, the evaluation of well-typed programs never encounters casts with uninstantiated type variables—a key property of our evaluation relation. On the other hand, by parametricity, we expect every value of type $\forall\alpha.\alpha \to \alpha$ to behave the same as the polymorphic identity function. One of the values of this type is $\Lambda\alpha.\ \langle \alpha \Rightarrow \alpha \rangle^l$. Without E_Refl, however, applying this type abstraction to a compound type, say $\mathsf{Bool} \to \mathsf{Bool}$, and a function f of type $\mathsf{Bool} \to \mathsf{Bool}$ would return, by E_Fun, a wrapped version of f that is syntactically different from the f we passed in—that is, the function broke parametricity! We expect the returned value should behave the same as the input, though—the results are just *syntactically* different. With E_Refl, $\langle T \Rightarrow T \rangle^l$ returns the input immediately, regardless of T—just as the identity function. So, this rule is a technical necessity, ensuring that casts containing type variables behave parametrically.

Now we can set about proving parametricity (Lemma 3.3.7). We begin with compositionality theorems relating the closing substitutions θ and δ to substitution in terms (Lemma 3.3.1) and types (Lemma 3.3.3); convertibility shows that our logical relation relates terms at convertible types (Lemma 3.3.4); after some lemmas about casts and a separate induction relating casts between related types (Lemma 3.3.6), we prove parametricity.

3.3.1 Lemma [Term compositionality]: If $\delta_1(e) \longrightarrow^* e_1$ and $\delta_2(e) \longrightarrow^* e_2$ then $r_1 \sim r_2 : T; \theta; \delta[e_1, e_2/x]$ iff $r_1 \sim r_2 : T[e/x]; \theta; \delta$.

Proof: By induction on the (simple) structure of T, proving both directions simultaneously. We treat the case where $r_1 = r_2 = \Uparrow l$ separately from the induction, since it's the same easy proof in all cases: $\Uparrow l \sim \Uparrow l : T; \theta; \delta$ irrespective of T and δ. So for the rest of proof, we know $r_1 = v_1$ and $r_2 = v_2$. Only the refinement case is interesting.

($\underline{T = \{y{:}T' \mid e'\}}$): We show both directions simultaneously, where $x \neq y$, i.e., y is fresh. By the IH for T', we know that

$$v_1 \sim v_2 : T'; \theta; \delta[e_1, e_2/x] \text{ iff } v_1 \sim v_2 : T'[e/x]; \theta; \delta.$$

It remains to show that the values satisfy their refinements.

That is, we must show:

$$\theta_1(\delta_1(e'[e_1/x][v_1/y])) \longrightarrow^* \mathsf{true} \text{ iff } \theta_1(\delta_1(e'[e/x][v_1/y])) \longrightarrow^* \mathsf{true}$$

$$\theta_2(\delta_2(e'[e_2/x][v_2/y])) \longrightarrow^* \mathsf{true} \text{ iff } \theta_2(\delta_2(e'[e/x][v_2/y])) \longrightarrow^* \mathsf{true}$$

So let:

$$\begin{array}{rclcrcl} \sigma_1 & = & \theta_1\delta_1[\delta_1(e)/x, v_1/y] & \longrightarrow^* & \theta_1\delta_1[e_1/x, v_1/y] & = & \sigma_1' \\ \sigma_2 & = & \theta_2\delta_2[\delta_2(e)/x, v_2/y] & \longrightarrow^* & \theta_2\delta_2[e_2/x, v_2/y] & = & \sigma_2' \end{array}$$

We have $\sigma_1 \longrightarrow^* \sigma_1'$ by reflexivity except for $\delta_1(e) \longrightarrow^* e_1$, which we have by assumption; likewise, we have $\sigma_2 \longrightarrow^* \sigma_2'$. Then $\sigma_i(e')$ and $\sigma_i'(e')$ coterminate (Lemma 3.2.1), and we are done. □

3.3.2 Lemma [Term Weakening/Strengthening]: If $x \notin T$, then $r_1 \sim r_2 : T; \theta; \delta[e_1, e_2/x]$ iff $r_1 \sim r_2 : T; \theta; \delta$.

Proof: Similar to Lemma 3.3.1. □

3.3.3 Lemma [Type compositionality]:
$r_1 \sim r_2 : T; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$ iff $r_1 \sim r_2 : T[T'/\alpha]; \theta; \delta$.

Proof: By induction on the (simple) structure of T, proving both directions simultaneously. As for Lemma 3.3.1, we treat the case where $r_1 = r_2 = \Uparrow l$ separately from the induction, since it's the same easy proof in all cases: $\Uparrow l \sim \Uparrow l : T; \theta; \delta$ irrespective of T and δ. So for the rest of proof, we know $r_1 = v_1$ and $r_2 = v_2$. Here, the interesting case is for function types, where we must deal with some asymmetries in the definition of the logical relation. I also include the case for quantified types.

($\underline{T = x{:}T_1 \to T_2}$): There are two cases:

> ($\underline{\Rightarrow}$): Given $v_1 \sim v_2 : (x{:}T_1 \to T_2); \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$, we wish to show that $v_1 \sim v_2 : (x{:}T_1 \to T_2)[T'/\alpha]; \theta; \delta$. Let $v_1' \sim v_2' : T_1[T'/\alpha]; \theta; \delta$. We must show that $v_1\, v_1' \simeq v_2\, v_2' : T_2[T'/\alpha]; \theta; \delta[v_1', v_2'/x]$. By the IH on T_1, $v_1' \sim v_2' : T_1; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$. By assumption,
>
> $$v_1\, v_1' \simeq v_2\, v_2' : T_2; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta[v_1', v_2'/x].$$
>
> These normalize to $r_1' \sim r_2' : T_2; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta[v_1', v_2'/x]$. Since $x \notin T'$, Lemma 3.3.2 gives $R_{T',\theta,\delta} = R_{T',\theta,\delta[v_1',v_2'/x]}$ and so
>
> $$r_1' \sim r_2' : T_2; \theta[\alpha \mapsto R_{T',\theta,\delta[v_1',v_2'/x]}, \theta_1(\delta_1(T'[v_1'/x])), \theta_2(\delta_2(T'[v_2'/x]))]; \delta[v_1', v_2'/x].$$
>
> By the IH on T_2, $r_1' \sim r_2' : T_2[T'/\alpha]; \theta; \delta[v_1', v_2'/x]$. By expansion, $v_1\, v_1'' \simeq v_2\, v_2'' : T_2[T'/\alpha]; \theta; \delta[v_1', v_2'/x]$.

($\Leftarrow$): This case is similar: Given $v_1 \sim v_2 : (x{:}T_1 \to T_2)[T'/\alpha]; \theta; \delta$, we wish to show that $v_1 \sim v_2 : (x{:}T_1 \to T_2); \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$. Let $v_1' \sim v_2' : T_1; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$. We must show that

$$v_1\, v_1' \simeq v_2\, v_2' : T_2; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta[v_1', v_2'/x].$$

By the IH on T_1, $v_1' \sim v_2' : T_1[T'/\alpha]; \theta; \delta$. By assumption, $v_1\, v_1' \simeq v_2\, v_2' : T_2[T'/\alpha]; \theta; \delta[v_1', v_2'/x]$. These normalize to $r_1' \simeq r_2' : T_2[T'/\alpha]; \theta; \delta[v_1', v_2'/x]$. By the IH on T_2,

$$\begin{aligned} r_1' \simeq r_2' :\ & T_2[T'/\alpha]; \\ & \theta[\alpha \mapsto R_{T',\theta,\delta[v_1',v_2'/x]}, \theta_1(\delta_1(T'[v_1'/x])), \theta_2(\delta_2(T'[v_2'/x]))]; \\ & \delta[v_1', v_2'/x]. \end{aligned}$$

Since $x \notin T'$, Lemma 3.3.2 gives

$$r_1' \simeq r_2' : T_2[T'/\alpha]; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta[v_1', v_2'/x].$$

Finally, by expansion

$$\begin{aligned} v_1\, v_1' \simeq v_2\, v_2' :\ & T_2[T'/\alpha]; \\ & \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \\ & \delta[v_1', v_2'/x]. \end{aligned}$$

($T = \forall\alpha'.T_0$): There are two cases:

($\Rightarrow$): Given $v_1 \sim v_2 : \forall\alpha'.T_0; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$, we wish to show that $v_1 \sim v_2 : \forall\alpha'.(T_0[T'/\alpha]); \theta; \delta$. Let a relation R and closed types T_1 and T_2 be given. By assumption, we know that $v_1\, T_1 \simeq v_2\, T_2 : T_0; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))][\alpha' \mapsto R, T_1, T_2]; \delta$. They normalize to $r_1' \sim r_2' : T_0; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))][\alpha' \mapsto R, T_1, T_2]; \delta$. By the IH, $r_1' \sim r_2' : T_0[T'/\alpha]; \theta[\alpha' \mapsto R, T_1, T_2]; \delta$. By expansion, $v_1\, T_1 \simeq v_2\, T_2 : T_0[T'/\alpha]; \theta[\alpha' \mapsto R, T_1, T_2]; \delta$. Then, $v_1 \sim v_2 : \forall\alpha'.(T_0[T'/\alpha]); \theta; \delta$.

($\Leftarrow$): This case is similar: given $v_1 \sim v_2 : \forall\alpha'.(T_0[T'/\alpha]); \theta; \delta$, we wish to show that $v_1 \sim v_2 : \forall\alpha'.T_0; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$. Let a relation R and closed types T_1 and T_2 be given. By assumption, we know that $v_1\, T_1 \simeq v_2\, T_2 : T_0[T'/\alpha]; \theta[\alpha' \mapsto R, T_1, T_2]; \delta$. They normalize to $r_1' \sim r_2' : T_0[T'/\alpha]; \theta[\alpha' \mapsto R, T_1, T_2]; \delta$. By the IH, $r_1' \sim r_2' : T_0; \theta[\alpha' \mapsto R, T_1, T_2][\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$. By expansion, $v_1\, T_1 \simeq v_2\, T_2 : T_0; \theta[\alpha' \mapsto R, T_1, T_2][\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$. Then, $v_1 \sim v_2 : \forall\alpha'.T_0; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$.

□

□

Complexity of casts

$$
\begin{array}{lcl}
cc(\langle T \Rightarrow T\rangle^l) & = & 1 \\
cc(\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l) & = & cc(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \Rightarrow T_{22}\rangle^l) + \\
 & & cc(\langle T_{21} \Rightarrow T_{11}\rangle^l) + 1 \\
cc(\langle \forall\alpha.T_1 \Rightarrow \forall\alpha.T_2\rangle^l) & = & cc(\langle T_1 \Rightarrow T_2\rangle^l) + 1 \\
cc(\langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle^l) & = & cc(\langle T_1 \Rightarrow T_2\rangle^l) + 1 \\
 & & (\text{if } T_2 \neq \{x{:}T_1 \mid e\} \text{ and } T_2 \neq \{y{:}\{x{:}T_1 \mid e\} \mid e'\}) \\
cc(\langle T_1 \Rightarrow \{x{:}T_1 \mid e\}\rangle^l) & = & 1 \\
cc(\langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l) & = & cc(\langle T_1 \Rightarrow T_2\rangle^l) + 2 \\
 & & (\text{if } T_1 \neq T_2 \text{ and } T_1 \text{ is not a refinement type})
\end{array}
$$

Figure 3.6: Complexity of casts

3.3.4 Lemma [Convertibility]: If $T_1 \equiv T_2$ then $r_1 \sim r_2 : T_1;\theta;\delta$ iff $r_1 \sim r_2 : T_2;\theta;\delta$.

Proof: By induction on the conversion relation, leaving θ and δ general. The case where $r_1 = r_2 = \Uparrow l$ is immediate, so we only need to consider the case where $r_1 = v_1$ and $r_2 = v_2$.

(C_Var): It must be that $T_1 = T_2 = \alpha$, so we are done immediately.

(C_Base): It must be that $T_1 = T_2 = B$, so we are done immediately.

(C_Refine): We have that $T_1 = \{x{:}T_1' \mid \sigma_1(e)\}$ and $T_2 = \{x{:}T_2' \mid \sigma_2(e)\}$, where $T_1' \equiv T_2'$ and $\sigma_1 \longrightarrow^* \sigma_2$.

We have $\theta_1\delta_1\sigma_1 \longrightarrow^* \theta_1\delta_1\sigma_2$ and $\theta_2\delta_2\sigma_1 \longrightarrow^* \theta_2\delta_2\sigma_2$ (by reflexivity of $\longrightarrow^*$ for θ_i and δ_i and assumption for $\sigma_1 \longrightarrow^* \sigma_2$), so by cotermination (Lemma 3.2.1):

$$
\begin{array}{l}
\theta_1(\delta_1(\sigma_1(e)))[v_1/x] \longrightarrow^* \mathsf{true} \text{ iff } \theta_1(\delta_1(\sigma_2(e)))[v_1/x] \longrightarrow^* \mathsf{true} \\
\theta_2(\delta_2(\sigma_1(e)))[v_2/x] \longrightarrow^* \mathsf{true} \text{ iff } \theta_2(\delta_2(\sigma_2(e)))[v_2/x] \longrightarrow^* \mathsf{true}
\end{array}
$$

(C_Fun): We have that $T_1 = x{:}T_{11} \to T_{12} \equiv x{:}T_{21} \to T_{22} = T_2$.

Let $v_1' \sim v_2' : T_{21};\theta;\delta$ be given; we must show that $v_1\, v_1' \simeq v_2\, v_2' : T_{22};\theta;\delta[v_1', v_2'/x]$.

By the IH, we know that $v_1' \sim v_2' : T_{11};\theta;\delta$, so we know that $v_1\, v_1' \simeq v_2\, v_2' : T_{12};\theta;\delta[v_1', v_2'/x]$. We are done by another application of the IH.

The other direction is similar.

(C_Forall): We have that $T_1 = \forall\alpha.T_1' \equiv \forall\alpha.T_2' = T_2$.

Let R, T, and T' be given. We must show that $v_1\, T \simeq v_2\, T' : T_2';\theta[\alpha \mapsto R, T, T'];\delta$. We know that $v_1\, T \simeq v_2\, T' : T_1';\theta[\alpha \mapsto R, T, T'];\delta$, so we are done by the IH.

The other direction is similar.

(C_Sym): By the IH.

(C_Trans): By the IHs. □

Before we can show parametricity (Lemma 3.3.7), we prove in a separate induction that casts between related types are related (Lemma 3.3.6). Before we can prove *that* lemma, we need an auxiliary fact about casts and substitution.

3.3.5 Lemma [Cast substitution]: If $\vdash \Gamma, x{:}T_1$, $\Gamma, x{:}T_1 \vdash \langle T_1 \Rightarrow T_2\rangle^l \simeq \langle T_1 \Rightarrow T_2\rangle^l : T_2$, and $\Gamma, x{:}T_2 \vdash e_1 \simeq e_2 : T$ then $\Gamma_1, x{:}T_1, \Gamma_2 \vdash e_1[\langle T_1 \Rightarrow T_2\rangle^l\, x/x] \simeq e_2[\langle T_1 \Rightarrow T_2\rangle^l\, x/x] : T$.

Proof: Let $\Gamma \vdash \theta; \delta$. We must show that

$$\theta_1(\delta_1(e_1[\langle T_1 \Rightarrow T_2\rangle^l\, x/x])) \simeq \theta_2(\delta_2(e_2[\langle T_1 \Rightarrow T_2\rangle^l\, x/x])) : T; \theta; \delta.$$

Now $\theta_1(\delta_1(x)) \simeq \theta_2(\delta_2(x)) : T_1; \theta; \delta$ by definition. By assumption, $\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l\, x)) \simeq \theta_2(\delta_2(\langle T_1 \Rightarrow T_2\rangle^l\, x)) : T_2; \theta; \delta$. So let δ' be $\delta[\delta_1(\langle T_1 \Rightarrow T_2\rangle^l\, x), \delta_2(\langle T_1 \Rightarrow T_2\rangle^l\, x)/x]$. We have $\Gamma, x{:}T_2 \vdash \theta; \delta'$, so by assumption we have that $\theta_1(\delta'_1(e_1)) \simeq \theta_2(\delta'_2(e_2)) : T; \theta; \delta$, which is the same as $\theta_1(\delta_1(e_1[\langle T_1 \Rightarrow T_2\rangle^l\, x/x])) \simeq \theta_2(\delta_2(e_2[\langle T_1 \Rightarrow T_2\rangle^l\, x/x])) : T; \theta; \delta$, and we are done. □

We show that (well typed) casts relate to themselves by induction a cast complexity metric, *cc*, defined in Figure 3.6. The complexity of a cast is the number of steps it and its subparts can take. This definition is carefully dependent on our definition of type compatibility and our cast reduction rules. Doing induction on this metric greatly simplifies the proof: we show that stepping casts at related types yields either related non-casts, or lower complexity casts between related types.

3.3.6 Lemma [Cast Reflexivity]: If $\vdash \Gamma$, $T_1 \parallel T_2$, $\Gamma \vdash T_1 \simeq T_1 : *$, and $\Gamma \vdash T_2 \simeq T_2 : *$, then $\Gamma \vdash \langle T_1 \Rightarrow T_2\rangle^l \simeq \langle T_1 \Rightarrow T_2\rangle^l : T_1 \to T_2$.

Proof: By induction on $cc(\langle T_1 \Rightarrow T_2\rangle^l)$.

$\underline{(T_1 = T_2)}$: Given $\Gamma \vdash \theta; \delta$, we wish to show that

$$\langle \theta_1(\delta_1(T_1)) \Rightarrow \theta_1(\delta_1(T_1))\rangle^l \sim \langle \theta_2(\delta_2(T_1)) \Rightarrow \theta_2(\delta_2(T_1))\rangle^l : T_1 \to T_1; \theta; \delta.$$

Let $v_1 \sim v_2 : T_1; \theta; \delta$. We must show that

$$\begin{array}{l}\langle \theta_1(\delta_1(T_1)) \Rightarrow \theta_1(\delta_1(T_1))\rangle^l\, v_1 \simeq \\ \langle \theta_2(\delta_2(T_1)) \Rightarrow \theta_2(\delta_2(T_1))\rangle^l\, v_2 : T_1; \theta; \delta[v_1, v_2/z]\end{array}$$

for fresh z. By E_Refl, these normalize to $v_1 \sim v_2 : T_1; \theta; \delta[v_1, v_2/z]$. Lemma 3.3.2 finishes the case.

$\underline{(T_1 = x{:}T_{11} \to T_{12} \text{ and } T_2 = x{:}T_{21} \to T_{22} \text{ and } T_1 \neq T_2)}$: Then, we have:

$$\begin{array}{c} T_{11} \parallel T_{21} \qquad T_{21} \parallel T_{22} \\ \Gamma \vdash T_{11} \\ \Gamma, x{:}T_{11} \vdash T_{12} \quad \Gamma, x{:}T_{21} \vdash T_{22}. \end{array}$$

Given $\Gamma \vdash \theta; \delta$, we wish to show that

$$\theta_1(\delta_1(\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l)) \sim \theta_1(\delta_1(\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l)) \\ : (x{:}T_{11} \to T_{12}) \to (x{:}T_{21} \to T_{22}); \theta; \delta.$$

Let $v_1 \sim v_2 : x{:}T_{11} \to T_{12}; \theta; \delta$. We must show that

$$\theta_1(\delta_1(\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l))\, v_1 \simeq \\ \theta_1(\delta_1(\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l))\, v_2 : x{:}T_{21} \to T_{22}; \theta; \delta[v_1, v_2/z]$$

for fresh z. Let

$$\begin{array}{rcl} v_1' & = & \theta_1(\delta_1(\lambda x{:}T_{21}.\ (\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \Rightarrow T_{22}\rangle^l\, (v_1\, (\langle T_{21} \Rightarrow T_{11}\rangle^l\, x))))) \\ v_2' & = & \theta_2(\delta_2(\lambda x{:}T_{21}.\ (\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, x/x] \Rightarrow T_{22}\rangle^l\, (v_2\, (\langle T_{21} \Rightarrow T_{11}\rangle^l\, x))))). \end{array}$$

Since

$$\begin{array}{rcl} \theta_1(\delta_1(\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l))\, v_1 & \longrightarrow & v_1' \\ \theta_2(\delta_2(\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l))\, v_2 & \longrightarrow & v_2', \end{array}$$

it suffices to show that

$$v_1' \sim v_2' : x{:}T_{21} \to T_{22}; \theta; \delta[v_1, v_2/z].$$

Let $v_1'' \sim v_2'' : T_{21}; \theta; \delta[v_1, v_2/z]$. We must show that

$$v_1'\, v_1'' \simeq v_2'\, v_2'' : T_{22}; \theta; \delta[v_1, v_2/z][v_1'', v_2''/x].$$

Since

$$\begin{array}{rcl} v_1'\, v_1'' & \longrightarrow & \theta_1(\delta_1(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, v_1''/x] \Rightarrow T_{22}\rangle^l\, (v_1\, (\langle T_{21} \Rightarrow T_{11}\rangle^l\, v_1'')))) \\ v_2'\, v_2'' & \longrightarrow & \theta_2(\delta_2(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, v_2''/x] \Rightarrow T_{22}\rangle^l\, (v_2\, (\langle T_{21} \Rightarrow T_{11}\rangle^l\, v_2'')))), \end{array}$$

it suffices to show that

$$\theta_1(\delta_1(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, v_1''/x] \Rightarrow T_{22}\rangle^l\, (v_1\, (\langle T_{21} \Rightarrow T_{11}\rangle^l\, v_1'')))) \\ \simeq \theta_2(\delta_2(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, v_2''/x] \Rightarrow T_{22}\rangle^l\, (v_2\, (\langle T_{21} \Rightarrow T_{11}\rangle^l\, v_2'')))) \\ : T_{22}; \theta; \delta[v_1, v_2/z][v_1'', v_2''/x].$$

Since $\Gamma \vdash T_1 \simeq T_1 : *$ and $\Gamma \vdash T_2 \simeq T_2 : *$, we have $\Gamma \vdash T_{11} \simeq T_{11} : *$ and $\Gamma \vdash T_{21} \simeq T_{21} : *$. Then, by the IH, we have

$$\theta_1(\delta_1(\langle T_{21} \Rightarrow T_{11}\rangle^l)) \sim \theta_2(\delta_2(\langle T_{21} \Rightarrow T_{11}\rangle^l)) : T_{21} \to T_{11}; \theta; \delta.$$

By Lemma 3.3.2 and assumption,

$$\theta_1(\delta_1(\langle T_{21} \Rightarrow T_{11}\rangle^l))\, v_1'' \simeq \theta_2(\delta_2(\langle T_{21} \Rightarrow T_{11}\rangle^l))\, v_2'' : T_{11}; \theta; \delta[v_1, v_2/z][v_1'', v_2''/z']$$

for fresh z'. These normalize to $r_1 \simeq r_2 : T_{11}; \theta; \delta[v_1, v_2/z][v_1'', v_2''/z']$. If $r_1 = r_2 = \Uparrow l$ for some l, then we also have $v_1\ v_1'' \longrightarrow^* \Uparrow l$ and $v_2\ v_2'' \longrightarrow^* \Uparrow l$ and, by expansion,

$$v_1\ v_1'' \simeq v_2\ v_2'' : T_{22}; \theta; \delta[v_1, v_2/z][v_1'', v_2''/x].$$

Otherwise, let $v_1''' = r_1$ and $v_2''' = r_2$. By Lemma 3.3.2 and definition,

$$v_1\ v_1''' \simeq v_2\ v_2''' : T_{12}; \theta; \delta[v_1''', v_2'''/x].$$

These normalize to

$$r_1' \sim r_2' : T_{12}; \theta; \delta[v_1''', v_2'''/x].$$

If $r_1' = r_2' = \Uparrow l'$ for some l', then, again, we have $v_1\ v_1'' \longrightarrow^* \Uparrow l$ and $v_2\ v_2'' \longrightarrow^* \Uparrow l$ and, by expansion,

$$v_1\ v_1'' \simeq v_2\ v_2'' : T_{22}; \theta; \delta[v_1, v_2/z][v_1'', v_2''/x].$$

Otherwise, let $v_1'''' = r_1'$ and $v_2'''' = r_2'$. By Lemma 3.3.2 and 3.3.1 and the fact that

$$\begin{array}{lll} \langle T_{21} \Rightarrow T_{11} \rangle^l\ v_1'' & \longrightarrow^* & v_1''' \\ \langle T_{21} \Rightarrow T_{11} \rangle^l\ v_2'' & \longrightarrow^* & v_2''' \end{array}$$

we have

$$v_1'''' \sim v_2'''' : T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l\ x/x]; \theta; \delta[v_1'', v_2''/x].$$

From $\Gamma \vdash T_1 \simeq T_1 : *$, we have $\Gamma, x{:}T_{11} \vdash T_{12} \simeq T_{12} : *$ by definition. Furthermore, we have $\Gamma, x{:}T_{21} \vdash T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l\ x/x] \simeq T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l\ x/x] : *$ by Lemma 3.3.5.

Since $T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l\ x/x] \parallel T_{22}$, by the IH,

$$\begin{array}{rl} \Gamma, x{:}T_{21} \vdash & \langle T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l\ x/x] \Rightarrow T_{22} \rangle^l \simeq \\ & \langle T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l\ x/x] \Rightarrow T_{22} \rangle^l : T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l\ x/x] \to T_{22}. \end{array}$$

Since $\Gamma, x{:}T_{21} \vdash \theta; \delta[v_1'', v_2''/x]$ and $x \notin T_{21}, T_{11}$, we have

$$\begin{array}{l} \theta_1(\delta_1(\langle T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l\ v_1''/x] \Rightarrow T_{22} \rangle^l)) \sim \\ \theta_2(\delta_2(\langle T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l\ v_2''/x] \Rightarrow T_{22} \rangle^l)) \\ : T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l\ x/x] \to T_{22}; \theta; \delta[v_1'', v_2''/x]. \end{array}$$

By definition,

$$\begin{array}{r} (\theta_1(\delta_1(\langle T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l\ v_1''/x] \Rightarrow T_{22} \rangle^l)))\ v_1'''' \\ \simeq (\theta_2(\delta_2(\langle T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l\ v_2''/x] \Rightarrow T_{22} \rangle^l)))\ v_2'''' \\ : T_{22}; \theta; \delta[v_1'', v_2''/x][v_1'''', v_2''''/z''] \end{array}$$

for fresh z''. These normalize to $r_1'' \sim r_2'' : T_{22}; \theta; \delta[v_1'', v_2''/x][v_1'''', v_2''''/z'']$. By expansion and Lemma 3.3.2, $v_1'\ v_1'' \simeq v_2'\ v_2'' : T_{22}; \theta; \delta[v_1, v_2/z][v_1'', v_2''/x]$.

$\underline{(T_1 = \forall\alpha.\, T_1' \text{ and } T_2 = \forall\alpha.\, T_2' \text{ and } T_1 \neq T_2)}$: Given $\Gamma \vdash \theta; \delta$, we wish to show that

$$\langle \forall\alpha.\theta_1(\delta_1(T_1')) \Rightarrow \forall\alpha.\theta_1(\delta_1(T_2'))\rangle^l \sim \langle \forall\alpha.\theta_2(\delta_2(T_1')) \Rightarrow \forall\alpha.\theta_2(\delta_2(T_2'))\rangle^l \\ : (\forall\alpha.\, T_1') \to (\forall\alpha.\, T_2'); \theta; \delta.$$

Let $v_1 \sim v_2 : \forall\alpha.\, T_1'; \theta; \delta$. We must show that

$$\langle \forall\alpha.\theta_1(\delta_1(T_1')) \Rightarrow \forall\alpha.\theta_1(\delta_1(T_2'))\rangle^l\, v_1 \simeq \langle \forall\alpha.\theta_2(\delta_2(T_1')) \Rightarrow \forall\alpha.\theta_2(\delta_2(T_2'))\rangle^l\, v_2 \\ : (\forall\alpha.\, T_2'); \theta; \delta[v_1, v_2/z]$$

for fresh z. Since

$$\langle \forall\alpha.\theta_1(\delta_1(T_1')) \Rightarrow \forall\alpha.\theta_1(\delta_1(T_2'))\rangle^l\, v_1 \longrightarrow \Lambda\alpha.\ \langle \theta_1(\delta_1(T_1')) \Rightarrow \theta_1(\delta_1(T_2'))\rangle^l\, (v_1\, \alpha) \\ \langle \forall\alpha.\theta_2(\delta_2(T_1')) \Rightarrow \forall\alpha.\theta_2(\delta_2(T_2'))\rangle^l\, v_2 \longrightarrow \Lambda\alpha.\ \langle \theta_2(\delta_2(T_1')) \Rightarrow \theta_2(\delta_2(T_2'))\rangle^l\, (v_2\, \alpha),$$

it suffices to show that

$$\Lambda\alpha.\ \langle \theta_1(\delta_1(T_1')) \Rightarrow \theta_1(\delta_1(T_2'))\rangle^l\, (v_1\, \alpha) \sim \\ \Lambda\alpha.\ \langle \theta_2(\delta_2(T_1')) \Rightarrow \theta_2(\delta_2(T_2'))\rangle^l\, (v_2\, \alpha) \\ : (\forall\alpha.\, T_2'); \theta; \delta[v_1, v_2/z].$$

Let R, T_1'', T_2'' be given. We will show that

$$\Lambda\alpha.\ \langle \theta_1(\delta_1(T_1')) \Rightarrow \theta_1(\delta_1(T_2'))\rangle^l\, (v_1\, \alpha)\ T_1'' \sim \\ \Lambda\alpha.\ \langle \theta_2(\delta_2(T_1')) \Rightarrow \theta_2(\delta_2(T_2'))\rangle^l\, (v_2\, \alpha)\ T_2'' \\ : T_2'; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta[v_1, v_2/z].$$

These terms normalize to

$$\theta_1(\delta_1(\langle T_1' \Rightarrow T_2'\rangle^l[T_1''/\alpha]))\, (v_1\ T_1'') \text{ and} \\ \theta_2(\delta_2(\langle T_1' \Rightarrow T_2'\rangle^l[T_2''/\alpha]))\, (v_2\ T_2''),$$

and we will show

$$\theta_1(\delta_1(\langle T_1' \Rightarrow T_2'\rangle^l[T_1''/\alpha]))\, (v_1\ T_1'') \simeq \\ \theta_2(\delta_2(\langle T_1' \Rightarrow T_2'\rangle^l[T_2''/\alpha]))\, (v_2\ T_2'') \\ : T_2'; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta[v_1, v_2/z].$$

By assumption,

$$v_1\ T_1'' \simeq v_2\ T_2'' : T_1'; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta.$$

These normalize to

$$r_1 \sim r_2 : T_1'; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta.$$

If $r_1 = r_2 = \Uparrow l$ for some l, we have

$$\Lambda\alpha.\ \langle \theta_1(\delta_1(T_1')) \Rightarrow \theta_1(\delta_1(T_2'))\rangle^l\, (v_1\, \alpha)\ T_1'' \ \longrightarrow^*\ \Uparrow l \\ \Lambda\alpha.\ \langle \theta_2(\delta_2(T_1')) \Rightarrow \theta_2(\delta_2(T_2'))\rangle^l\, (v_2\, \alpha)\ T_2'' \ \longrightarrow^*\ \Uparrow l,$$

finishing the case. Otherwise, we have $r_1 = v_1'$ and $r_2 = v_2'$ and

$$v_1' \sim v_2' : T_1'; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta.$$

It is easy to show that $\Gamma, \alpha \vdash T_1' \simeq T_1' : *$ and $\Gamma, \alpha \vdash T_2' \simeq T_2' : *$ from the assumptions $\Gamma, \alpha \vdash T_1 \simeq T_1 : *$ and $\Gamma, \alpha \vdash T_2 \simeq T_2 : *$. Then, by the IH, we have

$$\Gamma, \alpha \vdash \langle T_1' \Rightarrow T_2' \rangle^l \simeq \langle T_1' \Rightarrow T_2' \rangle^l : T_1' \to T_2'.$$

Since $\Gamma, \alpha \vdash \theta[\alpha \mapsto R, T_1'', T_2'']; \delta$, we have

$$\begin{aligned} \theta_1(\delta_1(\langle T_1'[T_1''/\alpha] \Rightarrow T_2'[T_1''/\alpha] \rangle^l)) \sim \theta_2(\delta_2(\langle T_1'[T_2''/\alpha] \Rightarrow T_2'[T_2''/\alpha] \rangle^l)) \\ : T_1' \to T_2'; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta. \end{aligned}$$

By definition,

$$\begin{aligned} \theta_1(\delta_1(\langle T_1' \Rightarrow T_2' \rangle^l [T_1''/\alpha])) \, v_1' \simeq \theta_2(\delta_2(\langle T_1' \Rightarrow T_2' \rangle^l [T_2''/\alpha])) \, v_2' \\ : T_2'; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta[v_1', v_2'/z'] \end{aligned}$$

for fresh z'. By Lemma 3.3.2,

$$\theta_1(\delta_1(\langle T_1' \Rightarrow T_2' \rangle^l [T_1''/\alpha])) \, v_1' \simeq \theta_2(\delta_2(\langle T_1' \Rightarrow T_2' \rangle^l [T_2''/\alpha])) \, v_2' : T_2'; \theta[\alpha \mapsto R, T_1'', T_2'']; \delta.$$

And now, by expansion,

$$\begin{aligned} \theta_1(\delta_1(\langle T_1' \Rightarrow T_2' \rangle^l [T_1''/\alpha])) \, (v_1 \; T_1'') &\longrightarrow^* \theta_1(\delta_1(\langle T_1' \Rightarrow T_2' \rangle^l [T_1''/\alpha])) \, v_1' \\ \theta_2(\delta_2(\langle T_1' \Rightarrow T_2' \rangle^l [T_2''/\alpha])) \, (v_2 \; T_2'') &\longrightarrow^* \theta_2(\delta_2(\langle T_1' \Rightarrow T_2' \rangle^l [T_2''/\alpha])) \, v_2' \end{aligned}$$

finish the case.

$\underline{(T_1 = \{x{:}T_1' \mid e\} \text{ and } T_1 \neq T_2 \text{ and } T_2 \neq \{y{:}T_1 \mid e'\})}$: Given $\Gamma \vdash \theta; \delta$, we wish to show that

$$\theta_1(\delta_1(\langle \{x{:}T_1' \mid e\} \Rightarrow T_2 \rangle^l)) \sim \theta_2(\delta_2(\langle \{x{:}T_1' \mid e\} \Rightarrow T_2 \rangle^l)) : \{x{:}T_1' \mid e\} \to T_2; \theta; \delta.$$

Let $v_1 \sim v_2 : \{x{:}T_1' \mid e\}; \theta; \delta$, that is, $v_1 \sim v_2 : T_1'; \theta; \delta$ and

$$\begin{aligned} \theta_1(\delta_1(e))[v_1/x] &\longrightarrow^* \mathsf{true} \\ \theta_2(\delta_2(e))[v_2/x] &\longrightarrow^* \mathsf{true} \end{aligned}$$

We have to show that

$$\begin{aligned} &\theta_1(\delta_1(\langle \{x{:}T_1' \mid e\} \Rightarrow T_2 \rangle^l)) \, v_1 \simeq \\ &\theta_2(\delta_2(\langle \{x{:}T_1' \mid e\} \Rightarrow T_2 \rangle^l)) \, v_2 : T_2; \theta; \delta[v_1, v_2/z] \end{aligned}$$

for fresh z. Since

$$\begin{aligned} \theta_1(\delta_1(\langle \{x{:}T_1' \mid e\} \Rightarrow T_2 \rangle^l)) \, v_1 &\longrightarrow \theta_1(\delta_1(\langle T_1' \Rightarrow T_2 \rangle^l)) \, v_1 \\ \theta_2(\delta_2(\langle \{x{:}T_1' \mid e\} \Rightarrow T_2 \rangle^l)) \, v_2 &\longrightarrow \theta_2(\delta_2(\langle T_1' \Rightarrow T_2 \rangle^l)) \, v_2 \end{aligned}$$

by E_Forget, it suffices to show that

$$\theta_1(\delta_1(\langle T_1' \Rightarrow T_2\rangle^l))\ v_1 \simeq \theta_2(\delta_2(\langle T_1' \Rightarrow T_2\rangle^l))\ v_2 : T_2; \theta; \delta[v_1, v_2/z].$$

Since $\Gamma \vdash T_1 \simeq T_1 : *$, we have $\Gamma \vdash T_1' \simeq T_1' : *$ by definition. We also have $T_1' \parallel T_2$, by inversion. Then, by the IH

$$\theta_1(\delta_1(\langle T_1' \Rightarrow T_2\rangle^l)) \sim \theta_2(\delta_2(\langle T_1' \Rightarrow T_2\rangle^l)) : T_1' \to T_2; \theta; \delta.$$

And finally, by assumption,

$$\theta_1(\delta_1(\langle T_1' \Rightarrow T_2\rangle^l))\ v_1 \simeq \theta_2(\delta_2(\langle T_1' \Rightarrow T_2\rangle^l))\ v_2 : T_2; \theta; \delta[v_1, v_2/z].$$

$\underline{(T_2 = \{x{:}T_1 \mid e\})}$: Given $\Gamma \vdash \theta; \delta$, we wish to show that

$$\theta_1(\delta_1(\langle T_1 \Rightarrow \{x{:}T_1 \mid e\}\rangle^l)) \sim \theta_2(\delta_2(\langle T_1 \Rightarrow \{x{:}T_1 \mid e\}\rangle^l)) : T_1 \to \{x{:}T_1 \mid e\}; \theta; \delta.$$

Let $v_1 \sim v_2 : T_1; \theta; \delta$. We have to show that

$$\begin{array}{l}\theta_1(\delta_1(\langle T_1 \Rightarrow \{x{:}T_1 \mid e\}\rangle^l))\ v_1 \simeq \\ \theta_2(\delta_2(\langle T_1 \Rightarrow \{x{:}T_1 \mid e\}\rangle^l))\ v_2 : \{x{:}T_1 \mid e\}; \theta; \delta[v_1, v_2/z]\end{array}$$

for fresh z. Since

$$\begin{array}{lcl}\theta_1(\delta_1(\langle T_1 \Rightarrow \{x{:}T_1 \mid e\}\rangle^l))\ v_1 & \longrightarrow & \langle \{x{:}T_1 \mid e\}, e[v_1/x], v_1\rangle^l \\ \theta_2(\delta_2(\langle T_1 \Rightarrow \{x{:}T_1 \mid e\}\rangle^l))\ v_2 & \longrightarrow & \langle \{x{:}T_1 \mid e\}, e[v_2/x], v_2\rangle^l\end{array}$$

by E_Check, it suffices to show that

$$\langle \{x{:}T_1 \mid e\}, e[v_1/x], v_1\rangle^l \simeq \langle \{x{:}T_1 \mid e\}, e[v_2/x], v_2\rangle^l : \{x{:}T_1 \mid e\}; \theta; \delta[v_1, v_2/z].$$

By the assumption $\Gamma \vdash T_2 \simeq T_2 : *$, we have $T_2 \simeq T_2 : *; \theta; \delta$. By definition, $T_1 \simeq T_1 : *; \theta; \delta$ and

$$\theta_1(\delta_1(e))[v_1'/x] \simeq \theta_2(\delta_2(e))[v_2'/x] : \mathsf{Bool}; \theta; \delta$$

for any $v_1' \sim v_2' : T_1; \theta; \delta$. So, in particular,

$$\theta_1(\delta_1(e))[v_1/x] \simeq \theta_2(\delta_2(e))[v_2/x] : \mathsf{Bool}; \theta; \delta.$$

These reduce to $r_1 \sim r_2 : \mathsf{Bool}; \theta; \delta$.

We have three cases:

$\underline{(r_1 = r_2 = \Uparrow l' \text{ for some } l')}$: Then the case is immediate by:

$$\begin{array}{lcl}\langle \{x{:}T_1 \mid e\}, e[v_1/x], v_1\rangle^l & \longrightarrow^* & \Uparrow l' \\ \langle \{x{:}T_1 \mid e\}, e[v_2/x], v_2\rangle^l & \longrightarrow^* & \Uparrow l'.\end{array}$$

($r_1 = r_2 = \mathsf{true}$): Then,

$$\begin{array}{lcl} \langle \{x{:}T_1 \mid e\}, e[v_1/x], v_1 \rangle^l & \longrightarrow^* & v_1 \\ \langle \{x{:}T_1 \mid e\}, e[v_2/x], v_2 \rangle^l & \longrightarrow^* & v_2. \end{array}$$

By assumption and Lemma 3.3.2, $v_1 \sim v_2 : T_1; \theta; \delta[v_1, v_2/z]$, finishing the case.

($r_1 = r_2 = \mathsf{false}$): Then we conclude by,

$$\begin{array}{lcl} \langle \{x{:}T_1 \mid e\}, e[v_1/x], v_1 \rangle^l & \longrightarrow^* & \Uparrow l \\ \langle \{x{:}T_1 \mid e\}, e[v_2/x], v_2 \rangle^l & \longrightarrow^* & \Uparrow l. \end{array}$$

(T_1 is not a refinement type and $T_2 = \{x{:}T_2' \mid e\}$ and $T_1 \neq T_2'$): Given $\Gamma \vdash \theta; \delta$, we wish to show that

$$\theta_1(\delta_1(\langle T_1 \Rightarrow \{x{:}T_2' \mid e\}\rangle^l)) \sim \theta_2(\delta_2(\langle T_1 \Rightarrow \{x{:}T_2' \mid e\}\rangle^l)) : T_1 \to \{x{:}T_2' \mid e\}; \theta; \delta.$$

Let $v_1 \sim v_2 : T_1; \theta; \delta$. We have to show that

$$\begin{array}{l} \theta_1(\delta_1(\langle T_1 \Rightarrow \{x{:}T_2' \mid e\}\rangle^l))\, v_1 \simeq \\ \theta_2(\delta_2(\langle T_1 \Rightarrow \{x{:}T_2' \mid e\}\rangle^l))\, v_2 : \{x{:}T_2' \mid e\}; \theta; \delta[v_1, v_2/z] \end{array}$$

for fresh z. Since

$$\begin{array}{l} \theta_1(\delta_1(\langle T_1 \Rightarrow \{x{:}T_2' \mid e\}\rangle^l))\, v_1 \longrightarrow \\ \theta_1(\delta_1(\langle T_2' \Rightarrow \{x{:}T_2' \mid e\}\rangle^l))\, (\theta_1(\delta_1(\langle T_1 \Rightarrow T_2'\rangle^l))\, v_1) \end{array}$$

$$\begin{array}{l} \theta_2(\delta_2(\langle T_1 \Rightarrow \{x{:}T_2' \mid e\}\rangle^l))\, v_2 \longrightarrow \\ \theta_2(\delta_2(\langle T_2' \Rightarrow \{x{:}T_2' \mid e\}\rangle^l))\, (\theta_2(\delta_2(\langle T_1 \Rightarrow T_2'\rangle^l))\, v_2) \end{array}$$

by E_PreCheck, it suffices to show that

$$\begin{array}{r} \theta_1(\delta_1(\langle T_2' \Rightarrow \{x{:}T_2' \mid e\}\rangle^l))\, (\theta_1(\delta_1(\langle T_1 \Rightarrow T_2'\rangle^l))\, v_1) \\ \simeq \theta_2(\delta_2(\langle T_2' \Rightarrow \{x{:}T_2' \mid e\}\rangle^l))\, (\theta_2(\delta_2(\langle T_1 \Rightarrow T_2'\rangle^l))\, v_2) \\ : \{x{:}T_2' \mid e\}; \theta; \delta[v_1, v_2/z]. \end{array}$$

Since $\Gamma \vdash T_2 \simeq T_2 : *$, we have $\Gamma \vdash T_2' \simeq T_2' : *$ by definition. Noting that

$$\begin{array}{rcl} cc(\langle T_1 \Rightarrow \{x{:}T_2' \mid e\}\rangle^l) & = & cc(\langle T_1 \Rightarrow T_2'\rangle^l) + 2 \;>\; cc(\langle T_1 \Rightarrow T_2'\rangle^l) \\ cc(\langle T_1 \Rightarrow \{x{:}T_2' \mid e\}\rangle^l & = & cc(\langle T_1 \Rightarrow T_2'\rangle^l) + 2 \;>\; 1 = cc(\langle T_2' \Rightarrow \{x{:}T_2' \mid e\}\rangle^l). \end{array}$$

we can apply the IH to see

$$\begin{array}{l} \Gamma \vdash \langle T_1 \Rightarrow T_2'\rangle^l \simeq \langle T_1 \Rightarrow T_2'\rangle^l : T_1 \to T_2', \text{ and} \\ \Gamma \vdash \langle T_2' \Rightarrow \{x{:}T_2' \mid e\}\rangle^l \simeq \langle T_2' \Rightarrow \{x{:}T_2' \mid e\}\rangle^l : T_2' \to \{x{:}T_2' \mid e\}. \end{array}$$

Then we can easily see

$$\begin{array}{r} \theta_1(\delta_1(\langle T_2' \Rightarrow \{x{:}T_2' \mid e\}\rangle^l))\, (\theta_1(\delta_1(\langle T_1 \Rightarrow T_2'\rangle^l))\, v_1) \\ \simeq \theta_2(\delta_2(\langle T_2' \Rightarrow \{x{:}T_2' \mid e\}\rangle^l))\, (\theta_2(\delta_2(\langle T_1 \Rightarrow T_2'\rangle^l))\, v_2) \\ : \{x{:}T_2' \mid e\}; \theta; \delta[v_1, v_2/z]. \end{array}$$

□

Finally, we can prove relational parametricity—every well-typed term (under Γ) is related to itself for any Γ-respecting substitutions.

3.3.7 Theorem [Parametricity]: 1. If $\Gamma \vdash e : T$ then $\Gamma \vdash e \simeq e : T$, and

2. If $\Gamma \vdash T$ then $\Gamma \vdash T \simeq T : *$.

Proof: By simultaneous induction on the derivations with case analysis on the last rule used.

(T_Var): Let $\Gamma \vdash \theta; \delta$. We wish to show that $\theta_1(\delta_1(x)) \simeq \theta_2(\delta_2(x)) : T; \theta; \delta$, which follows from the assumption.

(T_Const): By the assumption that constants are assigned correct types.

(T_Op): By the assumption that operators are assigned correct types (and the IHs for the operator's arguments).

(T_Abs): We have $e = \lambda x{:}T_1.\ e_{12}$ and $T = x{:}T_1 \to T_2$ and $\Gamma, x{:}T_1 \vdash e_{12} : T_2$. Let $\Gamma \vdash \theta; \delta$. We wish to show that

$$\theta_1(\delta_1(\lambda x{:}T_1.\ e_{12})) \sim \theta_2(\delta_2(\lambda x{:}T_1.\ e_{12})) : (x{:}T_1 \to T_2); \theta; \delta.$$

Let $v_1 \sim v_2 : T_1; \theta; \delta$. We must show that

$$(\lambda x{:}\theta_1(\delta_1(T_1)).\ \theta_1(\delta_1(e_{12})))\ v_1 \simeq (\lambda x{:}\theta_2(\delta_2(T_1)).\ \theta_2(\delta_2(e_{12})))\ v_2 : T_2; \theta; \delta[v_1, v_2/x].$$

Since

$$\begin{array}{rcl} (\lambda x{:}\theta_1(\delta_1(T_1)).\ \theta_1(\delta_1(e_{12})))\ v_1 & \longrightarrow & \theta_1(\delta_1(e_{12}))[v_1/x] \\ (\lambda x{:}\theta_2(\delta_2(T_1)).\ \theta_2(\delta_2(e_{12})))\ v_2 & \longrightarrow & \theta_2(\delta_2(e_{12}))[v_2/x], \end{array}$$

it suffices to show

$$\theta_1(\delta_1(e_{12}))[v_1/x] \simeq \theta_2(\delta_2(e_{12}))[v_2/x] : T_2; \theta; \delta[v_1, v_2/x].$$

By the IH, $\Gamma, x{:}T_1 \vdash e_{12} \simeq e_{12} : T_2$. The fact that $\Gamma, x{:}T_1 \vdash \theta; \delta[v_1, v_2/x]$ finishes the case.

(T_App): We have $e = e_1\ e_2$ and $\Gamma \vdash e_1 : x{:}T_1 \to T_2$ and $\Gamma \vdash e_2 : T_1$ and $T = T_2[e_2/x]$. Let $\Gamma \vdash \theta; \delta$. We wish to show that

$$\theta_1(\delta_1(e_1\ e_2)) \simeq \theta_2(\delta_2(e_1\ e_2)) : T_2[e_2/x]; \theta; \delta.$$

By the IH,

$$\begin{array}{l} \theta_1(\delta_1(e_1)) \simeq \theta_2(\delta_2(e_2)) : x{:}T_1 \to T_2; \theta; \delta, \text{ and} \\ \theta_1(\delta_1(e_2)) \simeq \theta_2(\delta_2(e_2)) : T_1; \theta; \delta. \end{array}$$

These normalize to $r_{11} \sim r_{12} : x{:}T_1 \to T_2; \theta; \delta$ and $r_{21} \simeq r_{22} : T_1; \theta; \delta$, respectively. If $r_{11} = r_{12} = \Uparrow l$ or $r_{21} = r_{22} = \Uparrow l$ for some l, then we are done:

$$\begin{array}{rcl} \theta_1(\delta_1(e_1\ e_2)) & \longrightarrow^* & \Uparrow l \\ \theta_2(\delta_2(e_1\ e_2)) & \longrightarrow^* & \Uparrow l. \end{array}$$

So let $r_{ij} = v_{ij}$. By definition,

$$v_{11}\ v_{21} \simeq v_{12}\ v_{22} : T_2; \theta; \delta[v_{21}, v_{22}/x].$$

These normalize to $r_1' \sim r_2' : T_2; \theta; \delta[v_{21}, v_{22}/x]$. By Lemma 3.3.1,

$$r_1' \sim r_2' : T_2[e_2/x]; \theta; \delta.$$

By expansion, we can then see

$$\theta_1(\delta_1(e_1\ e_2)) \simeq \theta_2(\delta_2(e_1\ e_2)) : T_2[e_2/x]; \theta; \delta.$$

(T_TABS): We have $e = \Lambda\alpha.\ e_0$ and $T = \forall\alpha.T_0$ and $\Gamma, \alpha \vdash e_0 : T_0$. Let $\Gamma \vdash \theta; \delta$. We wish to show that

$$\theta_1(\delta_1(\Lambda\alpha.\ e_0)) \sim \theta_2(\delta_2(\Lambda\alpha.\ e_0)) : \forall\alpha.T_0; \theta; \delta.$$

Let R, T_1, T_2 be given. We must show that

$$\theta_1(\delta_1(\Lambda\alpha.\ e_0))\ T_1 \simeq \theta_2(\delta_2(\Lambda\alpha.\ e_0))\ T_2 : T_0; \theta[\alpha \mapsto R, T_1, T_2]; \delta.$$

Since

$$\begin{array}{rcl} \theta_1(\delta_1(\Lambda\alpha.\ e_0))\ T_1 & \longrightarrow & \theta_1(\delta_1(e_0))[T_1/\alpha] \\ \theta_2(\delta_2(\Lambda\alpha.\ e_0))\ T_2 & \longrightarrow & \theta_2(\delta_2(e_0))[T_2/\alpha] \end{array}$$

it suffices to show that

$$\theta_1(\delta_1(e_0))[T_1/\alpha] \simeq \theta_2(\delta_2(e_0))[T_2/\alpha] : T_0; \theta[\alpha \mapsto R, T_1, T_2]; \delta.$$

Since $\Gamma, \alpha \vdash \theta[\alpha \mapsto R, T_1, T_2]; \delta$, the IH finishes the case with $\Gamma, \alpha \vdash e_0 \simeq e_0 : T_0$.

(T_TAPP): We have $e = e_1\ T_2$ and $\Gamma \vdash e_1 : \forall\alpha.T_0$ and $\Gamma \vdash T_2$ and $T = T_0[T_2/\alpha]$. Let $\Gamma \vdash \theta; \delta$. We wish to show that

$$\theta_1(\delta_1(e_1\ T_2)) \simeq \theta_2(\delta_2(e_1\ T_2)) : T_0[T_2/\alpha]; \theta; \delta.$$

By the IH,

$$\theta_1(\delta_1(e_1)) \simeq \theta_2(\delta_2(e_1)) : \forall\alpha.T_0; \theta; \delta.$$

These normalize to $r_1 \sim r_2 : \forall\alpha.\, T_0; \theta; \delta$. If both results are blame, $\theta_1(\delta_1(e_1\ T_2))$ and $\theta_2(\delta_2(e_1\ T_2))$ also normalize to blame, and we are done. So let $r_1 = v_1$ and $r_2 = v_2$. Then, by definition,

$$v_1\ T_1' \simeq v_2\ T_2' : T_0; \theta[\alpha \mapsto R,\, T_1',\, T_2']; \delta$$

for any R, T_1', T_2'. In particular,

$$v_1\, \theta_1(\delta_1(T_2)) \simeq v_2\, \theta_2(\delta_2(T_2)) : T_0; \theta[\alpha \mapsto R_{T_2,\theta,\delta}, \theta_1(\delta_1(T_2)), \theta_2(\delta_2(T_2))]; \delta.$$

These normalize to

$$r_1' \sim r_2' : T_0; \theta[\alpha \mapsto R_{T_2,\theta,\delta}, \theta_1(\delta_1(T_2)), \theta_2(\delta_2(T_2))]; \delta.$$

By Lemma 3.3.3, $r_1' \sim r_2' : T_0[T_2/\alpha]; \theta; \delta$. By expansion,

$$\theta_1(\delta_1(e_1\ T_2)) \simeq \theta_2(\delta_2(e_1\ T_2)) : T_0[T_2/\alpha]; \theta; \delta.$$

(T_Cast): We have $e = \langle T_1 \Rightarrow T_2 \rangle^l$ and $\vdash \Gamma$ and $T_1 \parallel T_2$ and $\Gamma \vdash T_1$, $\Gamma \vdash T_2$ and $T = T_1 \to T_2$. By the IH, $\Gamma \vdash T_1 \simeq T_1 : *$ and $\Gamma \vdash T_2 \simeq T_2 : *$. By Lemma 3.3.6,

$$\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle^l \simeq \langle T_1 \Rightarrow T_2 \rangle^l : T_1 \to T_2.$$

(T_Blame): Immediate.

(T_Check): We have $e = \langle \{x{:}T_1 \mid e_1\}, e_2, v \rangle^l$ and $\emptyset \vdash v : T_1$ and $\emptyset \vdash e_2 : \mathsf{Bool}$, $\vdash \Gamma$ and $\emptyset \vdash \{x{:}T_1 \mid e_1\}$ and $e_1[v/x] \longrightarrow^* e_2$ and $T = \{x{:}T_1 \mid e_1\}$. Let $\Gamma \vdash \theta; \delta$. We wish to show that

$$\theta_1(\delta_1(\langle \{x{:}T_1 \mid e_1\}, e_2, v \rangle^l)) \simeq \theta_2(\delta_2(\langle \{x{:}T_1 \mid e_1\}, e_2, v \rangle^l)) : \{x{:}T_1 \mid e_1\}; \theta; \delta.$$

By the IH,

$$\theta_1(\delta_1(e_2)) \simeq \theta_2(\delta_2(e_2)) : \mathsf{Bool}; \theta; \delta$$

and these normalize to the same result. If the result is false or $\Uparrow l'$ for some l', then, for some l'',

$$\begin{array}{rcl} \theta_1(\delta_1(\langle \{x{:}T_1 \mid e_1\}, e_2, v \rangle^l)) & \longrightarrow^* & \Uparrow l'' \\ \theta_2(\delta_2(\langle \{x{:}T_1 \mid e_1\}, e_2, v \rangle^l)) & \longrightarrow^* & \Uparrow l''. \end{array}$$

Otherwise, the result is true. Then, by the IH, $v \sim v : T_1; \theta; \delta$ and $\emptyset \vdash \{x{:}T_1 \mid e_1\} \simeq \{x{:}T_1 \mid e_1\} : *$. By definition,

$$\theta_1(\delta_1(e_1))[v/x] \simeq \theta_2(\delta_2(e_1))[v/x] : \mathsf{Bool}; \theta; \delta[v, v/x].$$

Then, we have

$$\begin{array}{rcl} \theta_1(\delta_1(e_1))[v/x] = e_1[v/x] & \longrightarrow^* & \mathsf{true} \\ \theta_2(\delta_2(e_1))[v/x] = e_1[v/x] & \longrightarrow^* & \mathsf{true}. \end{array}$$

By definition, $v \simeq v : \{x{:}T_1 \mid e_1\}; \theta; \delta$. By expansion,

$$\theta_1(\delta_1(\langle \{x{:}T_1 \mid e_1\}, e_2, v \rangle^l)) \simeq \theta_2(\delta_2(\langle \{x{:}T_1 \mid e_1\}, e_2, v \rangle^l)) : \{x{:}T_1 \mid e_1\}; \theta; \delta.$$

(T_Conv): By Lemma 3.3.4.

(T_Exact): We have $e = v$ and $\emptyset \vdash v : T$ and $\emptyset \vdash \{x{:}T_0 \mid e_0\}$ and $e_0[v/x] \longrightarrow^* \mathsf{true}$ and $T = \{x{:}T_0 \mid e_0\}$. Let $\Gamma \vdash \theta; \delta$. We wish to show that

$$v \sim v : \{x{:}T_0 \mid e_0\}; \theta; \delta.$$

By the IH, $v \sim v : T_0; \theta; \delta$. Since $\emptyset \vdash \{x{:}T_0 \mid e_0\}$, the only free variable in e_0 is x and

$$\begin{aligned} \theta_1(\delta_1(e_0))[v/x] = e_0[v/x] &\longrightarrow^* \mathsf{true} \\ \theta_2(\delta_2(e_0))[v/x] = e_0[v/x] &\longrightarrow^* \mathsf{true}. \end{aligned}$$

By definition, $v \sim v : \{x{:}T_0 \mid e_0\}; \theta; \delta$.

(T_Forget): By the IH, $\emptyset \vdash v \simeq v : \{x{:}T \mid e\}$, which implies $\Gamma \vdash v \simeq v : T$.

(WF_Base): Trivial.

(WF_TVar): Trivial.

(WF_Fun): By the IH.

(WF_Forall): By the IH.

(WF_Refine): By the IH. □

I refer readers to [61] for a significantly expanded account of parametricity for $\mathrm{F_H}$ *with recursion*. There, they have proved that their logical relation based on $\top\top$-closure [53] is sound[4] with respect to contextual equivalence. Since the details of their technical developments are different from what I present here, I can only conjecture that my logical relation is also sound with respect to contextual equivalence.

We do have that logically related programs are by definition *behaviorally equivalent*: if $\emptyset \vdash e_1 \simeq e_2 : T$, then e_1 and e_2 coterminate at related results. When the results are constants or blame, the results are not only logically related, but equal.

3.4 Subtyping and Upcast Elimination

Knowles and Flanagan [44] define a subtyping relation for their manifest calculus, λ_{H}, as a primitive notion of the system. Furthermore, they prove that upcast elimination is sound: if $T_1 <: T_2$, then $\langle T_1 \Rightarrow T_2 \rangle^l$ is equivalent to the identity function. Upcast elimination is, at heart, an optimization: since the cast can never fail, there is no point in running it. I define a subtyping relation for $\mathrm{F_H}$ and prove that upcast elimination is sound. To be clear, the type system of $\mathrm{F_H}$ doesn't have subtyping or a subsumption rule at all; we simply show that upcasts are logically related—and therefore behaviorally equivalent—to the identity.

[4] And also complete, under certain conditions.

Subtyping $\boxed{\Gamma \vdash T_1 <: T_2}$

$$\frac{}{\Gamma \vdash B <: B}\ \text{S\_Base} \qquad \frac{}{\Gamma \vdash \alpha <: \alpha}\ \text{S\_TVar} \qquad \frac{\Gamma, \alpha \vdash T_1 <: T_2}{\Gamma \vdash \forall \alpha. T_1 <: \forall \alpha. T_2}\ \text{S\_Forall}$$

$$\frac{\Gamma \vdash T_{21} <: T_{11} \quad \Gamma, x{:}T_{21} \vdash T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l\, x/x] <: T_{22}}{\Gamma \vdash x{:}T_{11} \to T_{12} <: x{:}T_{21} \to T_{22}}\ \text{S\_Fun}$$

$$\text{casts}(T) = \begin{cases} \langle T' \Rightarrow \{x{:}T' \mid e\} \rangle^l \circ \text{casts}(T') & \text{if } T = \{x{:}T' \mid e\} \\ \lambda x{:}T.\ x & \text{otherwise} \end{cases}$$

$$\frac{\begin{array}{l}\Gamma \vdash \text{unref}(T_1) <: \text{unref}(T_2) \\ \Gamma, x{:}\,\text{unref}(T_1) \vdash \text{casts}(T_1)\, x\ \supset\ \text{casts}(T_2)\, (\langle \text{unref}(T_1) \Rightarrow \text{unref}(T_2) \rangle^l\, x)\end{array}}{\Gamma \vdash T_1 <: T_2}\ \text{S\_Refine}$$

Implication $\boxed{\Gamma \vdash e_1\ \supset\ e_2}$

$$\frac{\forall \Gamma \vdash \theta; \delta.\ (\exists v.\ \theta_1(\delta_1(e_1)) \longrightarrow^*\ v)\, \text{implies}\, (\exists v.\ \theta_1(\delta_1(e_2)) \longrightarrow^*\ v)}{\Gamma \vdash e_1\ \supset\ e_2}\ \text{Imp}$$

Figure 3.7: Subtyping, implication, and closing substitutions

I define subtyping in Figure 3.7. My subtyping rules are similar to those in λ_{H}. The first three rules are standard. The rule for dependent function types is mostly usual: contravariant on argument types and covariant on return types. Here, we need to be careful about the type of x. Return types T_{12} and T_{22} should be compared under the assumption that x has T_{21}, which is a subtype of the other argument type T_{11} [6]. However, x in T_{12} has a different type, i.e., T_{11}, so we need to insert a cast to keep the subtyping relation well typed—F_{H} doesn't have subsumption! This extra cast is necessary in the S_Fun case when proving the upcast lemma, but the well typedness of the relation is necessary so we can apply parametricity (Lemma 3.3.7) in the S_Refine case.

The rule for subtyping of refinements differs substantially from λ_{H}'s, mostly because F_{H} allows refinements of arbitrary types, while λ_{H} only refines base types. The S_Refine rule essentially says T_1 is a subtype of T_2 if (1) T_1 without the (outermost) refinements is a subtype of T_2 without the (outermost) refinements, and (2) for any v of type $\mathrm{unref}(T_1)$, if $\mathrm{casts}(T_1)\, v$ reduces to a value, so does $\mathrm{casts}(T_2)\, \langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2) \rangle^l\, v$, for any l. The intuition behind the second condition is that, for T_1 to be a subtype of T_2, the predicates in T_1 (combined by conjunction) should be stronger than those in T_2. Recall that $\mathrm{casts}(T)$ is defined in Figure 3.7 as the composition of casts necessary to cast from $\mathrm{unref}(T)$ to T. So, if application of $\mathrm{casts}(T)$ to a value of type $\mathrm{unref}(T)$ does not raise blame, then the value can be typed at T by repeated use of T_Exact.

If the implication in S_Refine holds for a value v of type $\mathrm{unref}(T_1)$, then either: (1) v did not pass the checks in $\mathrm{casts}(T_1)$, so this value is not in T_1; or (2) v passed the checks in $\mathrm{casts}(T_1)$ and $\langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2) \rangle^l\, v$ passed all of the checks in $\mathrm{casts}(T_2)$. So, if (1) or (2) hold for all values of type $\mathrm{unref}(T_1)$, then it means that all values of type T_1 can be safely treated as if they had type T_2, i.e., T_1 a subtype of T_2.

Finally, we need a source of closing substitutions to compare the evaluation of the two casts. I use the closing substitutions from the logical relation at T as the source of "values of type T". (Arbitrarily, we take the values and types from the left.) There is a similar situation in the manifest calculi of Knowles and Flanagan [44] and Greenberg, Pierce, and Weirich [34]. They both define a separate denotational semantics for use in their refinement subtyping rule—a unary logical relation unrelated to the logical relations used in their respective proofs. But they *need* to do so, in order to avoid a circularity. F_{H} has no such issues, and I make the decision because it is expedient.

I formulate the implication judgment in terms of cotermination at values rather than cotermination at true (as in [34, 44]) because we have to contend with multiple layers of refinement in types—using cotermination at values reduces the amount of predicate bookkeeping we have to do.

Having defined subtyping, we are able to show that upcast elimination is sound.

3.4.1 Lemma [LR substitutivity]: If $\Gamma, x{:}T_1, \Gamma' \vdash T_2$ and $\Gamma \vdash e_1 \simeq e_2 : T_1$, then

$\Gamma, \Gamma'[e_1/x] \vdash T_2[e_1/x] \simeq T_2[e_2/x] : *$.

Proof: By induction on $\Gamma, x{:}T_1, \Gamma' \vdash T_2$. □

3.4.2 Lemma [LR type exchange]: If $T_1 \simeq T_2 : *; \theta; \delta$ and $T_2 \simeq T_1 : *; \theta; \delta$, then $v_1 \sim v_2 : T_1; \theta; \delta$ if and only if $v_1 \sim v_2 : T_2; \theta; \delta$.

Proof: By induction on the size of T_1 and T_2.

$\underline{(T_1 = T_2 = B)}$: Trivial.

$\underline{(T_1 = T_2 = \alpha)}$: Trivial.

$\underline{(T_1 = x{:}T_{11} \to T_{12} \text{ and } T_2 = x{:}T_{21} \to T_{22})}$: By definition, $T_{11} \simeq T_{21} : *; \theta; \delta$ and $T_{21} \simeq T_{11} : *; \theta; \delta$ and

$$\begin{array}{l} \forall v_1' \sim v_2' : T_{11}; \theta; \delta.\ T_{12} \simeq T_{22} : *; \theta; \delta[v_1', v_2'/x], \text{ and} \\ \forall v_1' \sim v_2' : T_{21}; \theta; \delta.\ T_{22} \simeq T_{12} : *; \theta; \delta[v_1', v_2'/x]. \end{array}$$

Then:

$$\begin{array}{rl} & v_1 \sim v_2 : T_1; \theta; \delta \\ \iff & \forall (v_1' \sim v_2' : T_{11}; \theta; \delta),\ v_1\, v_1' \simeq v_2\, v_2' : T_{12}; \theta; \delta[v_1', v_2'/x] \\ \text{(by the IH)} \iff & \forall (v_1' \sim v_2' : T_{21}; \theta; \delta),\ v_1\, v_1' \simeq v_2\, v_2' : T_{22}; \theta; \delta[v_1', v_2'/x] \\ \iff & v_1 \sim v_2 : T_2; \theta; \delta. \end{array}$$

$\underline{(T_1 = \forall \alpha.\, T_1' \text{ and } T_2 = \forall \alpha.\, T_2')}$: By definition,

$$\begin{array}{ll} \forall R\, T_1''\, T_2'', & T_1' \simeq T_2' : *; \theta[\alpha \mapsto R,\, T_1'',\, T_2'']; \delta \\ \forall R\, T_1''\, T_2'', & T_2' \simeq T_1' : *; \theta[\alpha \mapsto R,\, T_1'',\, T_2'']; \delta. \end{array}$$

Then:

$$\begin{array}{rl} & v_1 \sim v_2 : T_1; \theta; \delta \\ \iff & \forall R\, T_1''\, T_2'',\ v_1\, T_1'' \simeq v_2\, T_2'' : T_1'; \theta[\alpha \mapsto R,\, T_1'',\, T_2'']; \delta \\ \text{(by the IH)} \iff & \forall R\, T_1''\, T_2'',\ v_1\, T_1'' \simeq v_2\, T_2'' : T_2'; \theta[\alpha \mapsto R,\, T_1'',\, T_2'']; \delta \\ \iff & v_1 \sim v_2 : T_2; \theta; \delta. \end{array}$$

$\underline{(T_1 = \{x{:}T_1' \mid e_1\} \text{ and } T_2 = \{x{:}T_2' \mid e_2\})}$: By definition, $T_1' \simeq T_2' : *; \theta; \delta$ and $T_2' \simeq T_1' : *; \theta; \delta$. And also:

$$\forall (v_1 \sim v_2 : T_1'; \theta; \delta),\ \theta_1(\delta_1(e_1))[v_1/x] \simeq \theta_2(\delta_2(e_2))[v_2/x] : \mathsf{Bool}; \theta; \delta \tag{3.1}$$

$$\forall (v_1 \sim v_2 : T_2'; \theta; \delta),\ \theta_1(\delta_1(e_2))[v_1/x] \simeq \theta_2(\delta_2(e_1))[v_2/x] : \mathsf{Bool}; \theta; \delta. \tag{3.2}$$

Then:

$$\begin{array}{rl} & v_1 \sim v_2 : \{x{:}T_1' \mid e_1\}; \theta; \delta \\ \iff & \left\{ \begin{array}{l} v_1 \sim v_2 : T_1'; \theta; \delta \\ \theta_1(\delta_1(e_1))[v_1/x] \longrightarrow^* \mathsf{true} \\ \theta_2(\delta_2(e_1))[v_2/x] \longrightarrow^* \mathsf{true} \end{array} \right. \\ \left(\begin{array}{l} \text{by the IH} \\ \text{by (3.1)} \\ \text{by (3.2)} \end{array} \right) \iff & \left\{ \begin{array}{l} v_1 \sim v_2 : T_2'; \theta; \delta \\ \theta_2(\delta_2(e_2))[v_2/x] \longrightarrow^* \mathsf{true} \\ \theta_1(\delta_1(e_2))[v_1/x] \longrightarrow^* \mathsf{true} \end{array} \right. \\ \iff & v_1 \sim v_2 : \{x{:}T_2' \mid e_2\}; \theta; \delta. \end{array}$$

□

3.4.3 Lemma [Upcast lemma]: If $\Gamma \vdash T_1 <: T_2$ and $\Gamma \vdash T_1$ and $\Gamma \vdash T_2$, then $\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle^l \simeq \lambda x{:}T_1.\ x : T_1 \to T_2$.

Proof: By induction on the subtyping derivation.

(S_Base): Easy. We have $T_1 = B$ and $T_2 = B$. Let $\Gamma \vdash \theta; \delta$. We wish to show that

$$\theta_1(\delta_1(\langle B \Rightarrow B \rangle^l)) \simeq \theta_2(\delta_2(\lambda x{:}T_1.\ x)) : B \to B; \theta; \delta.$$

Let $v_1 \sim v_2 : B; \theta; \delta$. We must show that

$$\theta_1(\delta_1(\langle B \Rightarrow B \rangle^l))\ v_1 \simeq \theta_2(\delta_2(\lambda x{:}T_1.\ x))\ v_2 : B; \theta; \delta[v_1, v_2/z]$$

for fresh z, but these normalize to $v_1 \sim v_2 : B; \theta; \delta[v_1, v_2/z]$. Lemma 3.3.2 finishes the case.

(S_TVar): Similar to the case for S_Base.

(S_Fun): We have $T_1 = y{:}T_{11} \to T_{12}$ and $T_2 = y{:}T_{21} \to T_{22}$ and $\Gamma \vdash T_{21} <: T_{11}$ and $\Gamma, y{:}T_{21} \vdash T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l\ y/y] <: T_{22}$.

Let $\Gamma \vdash \theta; \delta$. We wish to show that

$$\begin{array}{rl} \theta_1(\delta_1(\langle y{:}T_{11} \to T_{12} \Rightarrow y{:}T_{21} \to T_{22} \rangle^l)) & \simeq \\ \theta_2(\delta_2(\lambda x{:}T_1.\ x)) & : (y{:}T_{11} \to T_{12}) \to (y{:}T_{21} \to T_{22}); \theta; \delta. \end{array}$$

Let $v_1 \sim v_2 : y{:}T_{11} \to T_{12}; \theta; \delta$. We must show that

$$\begin{array}{rl} \theta_1(\delta_1(\langle y{:}T_{11} \to T_{12} \Rightarrow y{:}T_{21} \to T_{22} \rangle^l))\ v_1 & \simeq \\ \theta_2(\delta_2(\lambda x{:}T_1.\ x))\ v_2 & : y{:}T_{21} \to T_{22}; \theta; \delta[v_1, v_2/z] \end{array}$$

for fresh z.

If $T_1 = T_2$, then we are done, since

$$\begin{array}{rcl} \theta_1(\delta_1(\langle y{:}T_{11} \to T_{12} \Rightarrow y{:}T_{21} \to T_{22} \rangle^l))\ v_1 & \longrightarrow & v_1 \\ \theta_2(\delta_2(\lambda x{:}T_1.\ x))\ v_2 & \longrightarrow & v_2. \end{array}$$

Otherwise,

$$\begin{array}{l} \theta_1(\delta_1(\langle y{:}T_{11} \to T_{12} \Rightarrow y{:}T_{21} \to T_{22} \rangle^l))\ v_1 \longrightarrow \\ \theta_1(\delta_1(\lambda y{:}T_{21}.\ \langle T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l\ y/y] \Rightarrow T_{21} \rangle^l\ (v_1\ (\langle T_{21} \Rightarrow T_{11} \rangle^l\ y)))) \end{array}$$

$$\theta_2(\delta_2(\lambda x{:}T_1.\ x))\ v_2 \longrightarrow v_2.$$

So, it suffices to show that

$$\begin{array}{r} \theta_1(\delta_1(\lambda y{:}T_{21}.\ \langle T_{12}[\langle T_{21} \Rightarrow T_{11} \rangle^l\ y/y] \Rightarrow T_{21} \rangle^l\ (v_1\ (\langle T_{21} \Rightarrow T_{11} \rangle^l\ y)))) \sim v_2 \\ : y{:}T_{21} \to T_{22}; \theta; \delta[v_1, v_2/z]. \end{array}$$

Let $v_1' \sim v_2' : T_{21}; \theta; \delta[v_1, v_2/z]$. We will show that

$$\theta_1(\delta_1(\lambda y{:}T_{21}.\ \langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, y/y] \Rightarrow T_{21}\rangle^l\, (v_1\, (\langle T_{21} \Rightarrow T_{11}\rangle^l\, y)))))\ v_1' \simeq v_2\ v_2' : T_{22}; \theta; \delta[v_1, v_2/z][v_1', v_2'/y].$$

By the IH, $\Gamma \vdash \langle T_{21} \Rightarrow T_{11}\rangle^l \simeq \lambda x{:}T_{21}.\ x : T_{21} \to T_{11}$. So,

$$\theta_1(\delta_1(\langle T_{21} \Rightarrow T_{11}\rangle^l))\ v_1' \simeq (\lambda x{:}\theta_1(\delta_1(T_{21}).\ x)\ v_2' : T_{11}; \theta; \delta[v_1', v_2'/z']$$

for fresh z''. Then, for some v_1'', $\theta_1(\delta_1(\langle T_{21} \Rightarrow T_{11}\rangle^l))\ v_1' \longrightarrow v_1''$ and $v_1'' \simeq v_2' : T_{11}; \theta; \delta[v_1', v_2'/z']$. By assumption,

$$v_1\ v_1'' \simeq v_2\ v_2' : T_{12}; \theta; \delta[v_1'', v_2'/y],$$

which normalize to $v_1''' \simeq v_2''' : T_{12}; \theta; \delta[v_1'', v_2'/y]$. Now, we can show that

$$\Gamma, y{:}T_{21} \vdash \langle T_{21} \Rightarrow T_{11}\rangle^l\, y \simeq y : T_{11}, \text{ and}$$
$$\Gamma \vdash \langle T_{21} \Rightarrow T_{11}\rangle^l \simeq \lambda x{:}T_{21}.\ x : T_{21} \to T_{11}.$$

Noting that $T_{12} = T_{12}[y/y]$, Lemma 3.4.1 obtains

$$\Gamma, y{:}T_{21} \vdash T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, y/y] \simeq T_{12} : *.$$

Then, by Lemma 3.4.2 (note that it is easy to show that $\Gamma \vdash T_1 \simeq T_2 : *$ if and only if $\Gamma \vdash T_2 \simeq T_1 : *$), we have

$$v_1''' \simeq v_2''' : T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, y/y]; \theta; \delta[v_1'', v_2'/y].$$

On the other hand, by the other IH,

$$\Gamma, y{:}T_{21} \vdash \langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, y/y] \Rightarrow T_{22}\rangle^l \simeq \lambda x{:}(T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, y/y]).\ x : T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, y/y] \to T_{22}.$$

Since $\Gamma, y{:}T_{21} \vdash \theta; \delta[v_1', v_2'/y]$,

$$\theta_1(\delta_1(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, v_1'/y] \Rightarrow T_{22}[v_1'/y]\rangle^l)) \sim \theta_2(\delta_2(\lambda x{:}T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, v_2'/y].\ x)) : T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, y/y] \to T_{22}; \theta; \delta[v_1', v_2'/y].$$

So,

$$\theta_1(\delta_1(\langle T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, v_1'/y] \Rightarrow T_{22}[v_1'/y]\rangle^l))\ v_1''' \simeq \theta_2(\delta_2((\lambda x{:}T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, v_2'/y].\ x)))\ v_2''' : T_{22}; \theta; \delta[v_1', v_2'/y][v_1''', v_2'''/z'''].$$

for fresh z'''. They normalize to

$$v_1'''' \sim v_2''' : T_{22}; \theta; \delta[v_1', v_2'/y][v_1''', v_2'''/z'''].$$

Now, letting $T'_{12} = T_{12}[\langle T_{21} \Rightarrow T_{11}\rangle^l\, y/y]$, we have:

$$\begin{array}{rl} & \theta_1(\delta_1(\lambda y{:}T_{21}.\ (\langle T'_{12} \Rightarrow T_{22}\rangle^l\,(v_1\,(\langle T_{21} \Rightarrow T_{11}\rangle^l\, y))))))\, v'_1 \\ \longrightarrow & \theta_1(\delta_1(\langle T'_{12}[v'_1/y] \Rightarrow T_{22}[v'_1/y]\rangle^l\,(v_1\,(\langle T_{21} \Rightarrow T_{11}\rangle^l\, v'_1))))) \\ \longrightarrow^* & \theta_1(\delta_1(\langle T'_{12}[v'_1/y] \Rightarrow T_{22}[v'_1/y]\rangle^l))\,(v_1\, v''_1) \\ \longrightarrow^* & \theta_1(\delta_1(\langle T'_{12}[v'_1/y] \Rightarrow T_{22}[v'_1/y]\rangle^l))\, v'''_1 \\ \longrightarrow^* & v''''_1 \end{array}$$

and $v_2\, v'_2 \longrightarrow^* v'''_2$ and

$$v''''_1 \sim v'''_2 : T_{22}; \theta; \delta[v'_1, v'_2/y][v'''_1, v'''_2/z'''],$$

which concludes this case by expansion.

(S_Forall): We have $T_1 = \forall\alpha.\, T'_1$ and $T_2 = \forall\alpha.\, T'_2$ and $\Gamma, \alpha \vdash T'_1 <: T'_2$.
Let $\Gamma \vdash \theta; \delta$. We wish to show that

$$\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l)) \simeq \theta_2(\delta_2(\lambda x{:}T_1.\ x)) : T_1 \to T_2; \theta; \delta.$$

Let $v_1 \sim v_2 : T_1; \theta; \delta$. We must show that

$$\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\, v_1 \simeq \theta_2(\delta_2(\lambda x{:}T_1.\ x))\, v_2 : T_2; \theta; \delta[v_1, v_2/z]$$

for fresh z. If $T_1 = T_2$, then we are done:

$$\begin{array}{rcl} \theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\, v_1 & \longrightarrow & v_1 \\ \theta_2(\delta_2(\lambda x{:}T_1.\ x))\, v_2 & \longrightarrow & v_2. \end{array}$$

Otherwise,

$$\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\, v_1 \longrightarrow \Lambda\alpha.\ (\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\,(v_1\,\alpha)).$$

So, it suffices to show that

$$\Lambda\alpha.\ (\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\,(v_1\,\alpha)) \simeq v_2 : T_2; \theta; \delta[v_1, v_2/z].$$

Let R, T''_1, T''_2 be given. We must show that

$$\Lambda\alpha.\ (\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\,(v_1\,\alpha))\, T''_1 \simeq v_2\, T''_2 : T'_2; \theta[\alpha \mapsto R, T''_1, T''_2]; \delta[v_1, v_2/z].$$

Since

$$\Lambda\alpha.\ (\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\,(v_1\,\alpha))\, T''_1 \longrightarrow \theta_1(\delta_1(\langle T_1[T''_1/\alpha] \Rightarrow T_2[T''_1/\alpha]\rangle^l))\,(v_1\, T''_1),$$

it suffices to show that

$$\theta_1(\delta_1(\langle T_1[T''_1/\alpha] \Rightarrow T_2[T''_1/\alpha]\rangle^l))\,(v_1\, T''_1) \simeq v_2\, T''_2 : T'_2; \theta[\alpha \mapsto R, T''_1, T''_2]; \delta[v_1, v_2/z].$$

By assumption

$$v_1\ T_1'' \simeq v_2\ T_2'' : T_1; \theta[\alpha \mapsto R,\, T_1'',\, T_2'']; \delta,$$

which normalize to

$$v_1' \sim v_2' : T_1; \theta[\alpha \mapsto R,\, T_1'',\, T_2'']; \delta.$$

By the IH,

$$\Gamma, \alpha \vdash \langle T_1' \Rightarrow T_2' \rangle^l \simeq \lambda x{:}T_1'.\ x : T_1' \to T_2'$$

and so,

$$\theta_1(\delta_1(\langle T_1[T_1''/\alpha] \Rightarrow T_2[T_1''/\alpha] \rangle^l)) \sim \lambda x{:}T_1'[T_2''/\alpha].\ x : T_1' \to T_2'; \theta[\alpha \mapsto R,\, T_1'',\, T_2'']; \delta.$$

Then, by definition,

$$\begin{aligned}\theta_1(\delta_1(\langle T_1[T_1''/\alpha] \Rightarrow T_2[T_1''/\alpha] \rangle^l))\ v_1' \simeq (\lambda x{:}T_1'[T_2''/\alpha].\ x)\ v_2' \\ : T_2'; \theta[\alpha \mapsto R,\, T_1'',\, T_2'']; \delta[v_1', v_2'/z']\end{aligned}$$

for fresh z'. They normalize to

$$v_1'' \sim v_2' : T_2'; \theta[\alpha \mapsto R,\, T_1'',\, T_2'']; \delta[v_1', v_2'/z'],$$

and by expansion we are done:

$$\theta_1(\delta_1(\langle T_1[T_1''/\alpha] \Rightarrow T_2[T_1''/\alpha] \rangle^l))\ (v_1\ T_1'') \simeq v_2\ T_2'' : T_2'; \theta[\alpha \mapsto R,\, T_1'',\, T_2'']; \delta[v_1, v_2/z].$$

(S_Refine): We have $\Gamma \vdash \mathrm{unref}(T_1) <: \mathrm{unref}(T_2)$ and

$$\Gamma, x{:}\,\mathrm{unref}(T_1) \vdash \mathrm{casts}(T_1)\ x\ \supset\ \mathrm{casts}(T_2)\ (\langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2) \rangle^l\ x).$$

Let $\Gamma \vdash \theta; \delta$. We wish to show that

$$\theta_1(\delta_1(\langle T_1 \Rightarrow T_2 \rangle^l)) \simeq \theta_2(\delta_2(\lambda x{:}T_1.\ x)) : T_1 \to T_2; \theta; \delta.$$

Let $v_1 \sim v_2 : T_1; \theta; \delta$. We must show that

$$\theta_1(\delta_1(\langle T_1 \Rightarrow T_2 \rangle^l))\ v_1 \simeq \theta_2(\delta_2(\lambda x{:}T_1.\ x))\ v_2 : T_2; \theta; \delta[v_1, v_2/z]$$

for fresh z.

We have three cases according to how the LHS reduces.

($T_2 = \mathrm{unref}_1^j(T_1)$ for some j): Then, we have

$$\begin{array}{lll} & & \theta_1(\delta_1(\langle T_1 \Rightarrow T_2 \rangle^l))\ v_1 \\ (\text{by E\_Forget}) & \longrightarrow^* & \theta_1(\delta_1(\langle T_2 \Rightarrow T_2 \rangle^l))\ v_1 \\ (\text{by E\_Refl}) & \longrightarrow & v_1. \end{array}$$

Since $\theta_2(\delta_2(\lambda x{:}T_1.\ x))\ v_2 \longrightarrow v_2$, we have $v_1 \sim v_2 : T_1; \theta; \delta[v_1, v_2/z]$. Since $\mathrm{unref}_1^j(T_1) = T_2$, we also have $v_1 \sim v_2 : T_2; \theta; \delta[v_1, v_2/z]$. So, by expansion,

$$\theta_1(\delta_1(\langle T_1 \Rightarrow T_2 \rangle^l))\ v_1 \simeq \theta_2(\delta_2(\lambda x{:}T_1.\ x))\ v_2 : T_2; \theta; \delta[v_1, v_2/z].$$

($T_1' = \mathrm{unref}_1^j(T_1)$ for some j and $T_2 = \{x{:}T_1' \mid e_2\}$): Then, we have

$$\begin{array}{lll} & & \theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\, v_1 \\ (\text{by E\_Forget}) & \longrightarrow^* & \theta_1(\delta_1(\langle T_1' \Rightarrow \{x{:}T_1' \mid e_2\}\rangle^l))\, v_1 \\ (\text{by E\_Check}) & \longrightarrow & \langle \theta_1(\delta_1(\{x{:}T_1' \mid e_2\})), \theta_1(\delta_1(e_2))[v_1/x], v_1\rangle^l. \end{array}$$

By assumption, $v_1 \simeq v_2 : \mathrm{unref}(T_1); \theta; \delta$ and $\theta_1(\delta_1(\mathrm{casts}(T_1)))\, v_1 \longrightarrow^* v_1$. By Imp, for some v_1' we have,

$$\theta_1(\delta_1(\mathrm{casts}(T_2)))\, (\theta_1(\delta_1(\langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2)\rangle^l))\, v_1) \longrightarrow^* v_1'.$$

By inspecting the reduction sequence, we have

$$\begin{array}{lll} & & \mathrm{casts}(T_2)\, (\langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2)\rangle^l\, v_1) \\ (\text{by E\_Refl}) & \longrightarrow & \mathrm{casts}(T_2)\, v_1 \\ & \longrightarrow^* & \langle T_1' \Rightarrow \{x{:}T_1' \mid e_2\}\rangle^l\, v_1 \\ (\text{by E\_Check}) & \longrightarrow & \langle \theta_1(\delta_1(\{x{:}T_1' \mid e_2\})), \theta_1(\delta_1(e_2))[v_1/x], v_1\rangle^l \\ & \longrightarrow^* & \langle \theta_1(\delta_1(\{x{:}T_1' \mid e_2\})), \mathsf{true}, v_1\rangle^l \\ & \longrightarrow & v_1. \end{array}$$

Since $\theta_2(\delta_2(\lambda x{:}T_1.\ x))\, v_2 \longrightarrow v_2$, we have

$$v_1 \sim v_2 : T_1; \theta; \delta[v_1, v_2/z].$$

Since $\mathrm{unref}_1^j(T_1) = T_1'$, we also have

$$v_1 \sim v_2 : T_1'; \theta; \delta[v_1, v_2/z].$$

Finally, $\theta_1(\delta_1(e_2))[v_1/x] \longrightarrow^* \mathsf{true}$ gives $v_1 \sim v_2 : \{x{:}T_1' \mid e_2\}; \theta; \delta[v_1, v_2/z]$. Finally, by expansion

$$\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\, v_1 \simeq \theta_2(\delta_2(\lambda x{:}T_1.\ x))\, v_2 : T_2; \theta; \delta[v_1, v_2/z].$$

(Otherwise): We have

$$\begin{array}{lll} & & \theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\, v_1 \\ (\text{by E\_Forget}) & \longrightarrow^* & \theta_1(\delta_1(\langle \mathrm{unref}(T_1) \Rightarrow T_2\rangle^l))\, v_1 \\ (\text{by E\_Precheck}) & \longrightarrow^* & e \end{array}$$

where

$$\begin{array}{rl} e = & \theta_1(\delta_1(\langle \mathrm{unref}_1(T_2) \Rightarrow T_2\rangle^l \\ & \quad (\langle \mathrm{unref}_2(T_2) \Rightarrow \mathrm{unref}_1(T_2)\rangle^l \\ & \quad\quad (\cdots\ (\langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2)\rangle^l\, v_1))))). \end{array}$$

By the IH,

$$\Gamma \vdash \langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2)\rangle^l \simeq \lambda x{:}\,\mathrm{unref}(T_1).\ x : (\mathrm{unref}(T_1) \to \mathrm{unref}(T_2)).$$

That is,

$$\theta_1(\delta_1(\langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2)\rangle^l))\, v_1 \simeq \theta_1(\delta_1(\lambda x{:}\,\mathrm{unref}(T_1).\ x))\, v_2 : \mathrm{unref}(T_2); \theta; \delta[v_1, v_2/z].$$

They normalize to $v_1' \sim v_2 : \mathrm{unref}(T_2); \theta; \delta[v_1, v_2/z]$.

By assumption, $v_1 \simeq v_2 : \mathrm{unref}(T_1); \theta; \delta$ and $\theta_1(\delta_1(\mathrm{casts}(T_1)))\, v_1 \longrightarrow^* v_1$. By IMP, for some v_1'' we have:

$$\theta_1(\delta_1(\mathrm{casts}(T_2)\,((\langle \mathrm{unref}(T_1) \Rightarrow \mathrm{unref}(T_2)\rangle^l)\, v_1))) \longrightarrow \theta_1(\delta_1(\mathrm{casts}(T_2)))\, v_1' \longrightarrow^* v_1''.$$

Inspecting this reduction sequence gives $v_1'' = v_1'$.

By Lemma 3.3.7, $\Gamma \vdash T_2 \simeq T_2 : *$, which shows $\theta_2(\delta_2(\mathrm{casts}(T_2)))\, v_2 \longrightarrow^* v_2'$ for some v_2'—that is, running all of the refinements doesn't produce blame, which means the checks must go to true. We can then conclude that $v_1' \sim v_2 : T_2; \theta; \delta[v_1, v_2/z]$, and so, by expansion,

$$\theta_1(\delta_1(\langle T_1 \Rightarrow T_2\rangle^l))\, v_1 \simeq \theta_2(\delta_2(\lambda x{:}T_1.\ x))\, v_2 : T_2; \theta; \delta[v_1, v_2/z].$$

□

□

Other optimizations

We can use subtyping to prove other optimizations correct. For example, we can show that $\{y{:}\{x{:}T \mid e_1\} \mid e_2\}$ and $\{x{:}T \mid e_1 \wedge e_2[x/y]\}$ are equivalent. To do so, we must first show that subtyping is reflexive. In order to show reflexivity, we must show that subtyping respects contextual equivalence of types—two types are contextually equivalent if their type parts are identical and the matching term parts are contextually equivalent. (This implies that contextually equivalent types have contextually equivalent sub-parts.) We lift this to contexts in the natural way.

3.4.4 Lemma [Subtyping respects contextual equivalence]: If T_1 is contextually equivalent to T_1', then (1) $\Gamma \vdash T_1 <: T_2$ iff $\Gamma \vdash T_1' <: T_2$ and (2) $\Gamma \vdash T_2 <: T_1$ iff $\Gamma \vdash T_2 <: T_1'$.

Proof: By induction on the subtyping derivations. □

3.4.5 Lemma [Reflexivity of subtyping]: If $\Gamma \vdash T$, then $\Gamma \vdash T <: T$.

Proof: By induction on the height of $\Gamma \vdash T$, using Lemma 3.4.4 to account for the cast in the domain of S_FUN. □

3.4.6 Lemma: If $\Gamma \vdash \{y{:}\{x{:}T \mid e_1\} \mid e_2\}$, then $\Gamma \vdash \{y{:}\{x{:}T \mid e_1\} \mid e_2\} <: \{x{:}T \mid e_1 \wedge e_2[x/y]\}$ and $\Gamma \vdash \{x{:}T \mid e_1 \wedge e_2[x/y]\} <: \{y{:}\{x{:}T \mid e_1\} \mid e_2\}$.

Proof: By inversion, we know that the inner types are well formed ($\Gamma \vdash \{x{:}T \mid e_1\}$ and $\Gamma \vdash T$) and that the predicates are well formed ($\Gamma, x{:}T \vdash e_1 : \mathsf{Bool}$ and $\Gamma, y{:}\{x{:}T \mid e_1\} \vdash e_2 : \mathsf{Bool}$).

We want to apply S_Refine to prove both subtypings. By reflexivity of subtyping (Lemma 3.4.5), we know that $\Gamma \vdash \mathrm{unref}(\{y{:}\{x{:}T \mid e_1\} \mid e_2\}) <: \mathrm{unref}(\{x{:}T \mid e_1 \wedge e_2[x/y]\})$ and vice versa. We must then show, forall $\Gamma, x{:}T \vdash \theta; \delta$, that if

$$\theta_1(\delta_1(\langle T \Rightarrow \{x{:}T \mid e_1\}\rangle^l\,(\langle\{x{:}T \mid e_1\} \Rightarrow \{y{:}\{x{:}T \mid e_1\} \mid e_2\}\rangle^l\,x)))$$

reduces to a value iff

$$\theta_1(\delta_1(\langle T \Rightarrow \{x{:}T \mid e_1 \wedge e_2[x/y]\}\rangle^l\,x))$$

does, too.

Let $\theta_1(\delta_1(x)) = v$. The former steps to a value when $e_1[v/x] \longrightarrow^* \mathsf{true}$ and $e_2[v/y] \longrightarrow^* \mathsf{true}$. The latter steps to a value $(e_1 \wedge e_2[x/y])[v/x] \longrightarrow^* \mathsf{true}$, i.e., when $e_1[v/x] \longrightarrow^* \mathsf{true}$ and $e_2[v/y] \longrightarrow^* \mathsf{true}$. □

3.5 Type conversion: parallel reduction vs. common subexpression reduction

Belo et al. [8] used parallel reduction (defined in Figure 3.8) as the conversion relation between types, while here I've used a structural conversion relation with common subexpression reduction (CSR). In this section, I explain the need for a conversion relation, why I've made the change, and how the two approaches differ.

First, why do we need a type conversion relation at all? Suppose we are trying to prove preservation, in order to find syntactic type soundness. Consider the term $v_1\ e_2$, where v_1 has the type $x{:}T_1 \to T_2$ and e_2 has the type T_1. According to T_App, the type of $v_1\ e_2$ is $T_2[e_2/x]$. What happens when $e_2 \longrightarrow e_2'$? The syntactic type system gives us $v_1\ e_2 : T_2[e_2'/x]$. To finish such a preservation proof, we must know how $T_2[e_2/x]$ and $T_2[e_2'/x]$ relate. Intuitively, they ought to be inhabited by the same values: since our evaluation semantics is deterministic, any value that satisfies the checks in $T_2[e_2/x]$ should also satisfy the checks in $T_2[e_2'/x]$, since the latter type is just a few extra steps along. We must modify the definition of our syntactic type system to make it respect this equivalence in a formal way.

In λ_{H} in Chapter 2, we observe that while $T_2[e_2/x]$ may not reduce to $T_2[e_2'/x]$—types don't reduce at all, in fact—we can relate them as subtypes of each other. For this reason (among others), the λ_{H} system introduces a subtyping relation. But it turns out that subtyping in that language introduces a vicious cycle (see Section 5.2.2), forcing us to adopt a semantic approach to types soundness. I end up

Parallel term reduction $\boxed{e_1 \Rrightarrow e_2}$

$$\frac{v_i \Rrightarrow v_i'}{\mathrm{op}\,(v_1,\,\ldots,\,v_n) \Rrightarrow [\![\mathrm{op}]\!]\,(v_1',\,\ldots,\,v_n')}\ \textsc{EP\_ROp} \qquad \frac{e_{12} \Rrightarrow e_{12}' \quad v_2 \Rrightarrow v_2'}{(\lambda x{:}T.\ e_{12})\ v_2 \Rrightarrow e_{12}'[v_2'/x]}\ \textsc{EP\_RBeta}$$

$$\frac{e \Rrightarrow e' \quad T_2 \Rrightarrow T_2'}{(\Lambda\alpha.\ e)\ T_2 \Rrightarrow e'[T_2'/\alpha]}\ \textsc{EP\_RTBeta} \qquad \frac{v \Rrightarrow v'}{\langle T \Rightarrow T\rangle^l\, v \Rrightarrow v'}\ \textsc{EP\_RRefl}$$

$$\frac{T_2 \neq \{x{:}T_1 \mid e\} \quad T_2 \neq \{y{:}\{x{:}T_1 \mid e\} \mid e_2\} \quad T_1 \Rrightarrow T_1' \quad T_2 \Rrightarrow T_2' \quad v \Rrightarrow v'}{\langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle^l\, v \Rrightarrow \langle T_1' \Rightarrow T_2'\rangle^l\, v'}\ \textsc{EP\_RForget}$$

$$\frac{T_1 \neq T_2 \quad T_1 \neq \{x{:}T \mid e\} \quad T_1 \Rrightarrow T_1' \quad T_2 \Rrightarrow T_2' \quad e \Rrightarrow e' \quad v \Rrightarrow v'}{\langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l\, v \Rrightarrow \langle T_2' \Rightarrow \{x{:}T_2' \mid e'\}\rangle^l\, (\langle T_1' \Rightarrow T_2'\rangle^l\, v')}\ \textsc{EP\_RPreCheck}$$

$$\frac{T \Rrightarrow T' \qquad e \Rrightarrow e' \qquad v \Rrightarrow v'}{\langle T \Rightarrow \{x{:}T \mid e\}\rangle^l\, v \Rrightarrow \langle \{x{:}T' \mid e'\}, e'[v'/x], v'\rangle^l}\ \textsc{EP\_RCheck}$$

$$\frac{v \Rrightarrow v'}{\langle \{x{:}T \mid e_1\}, \mathsf{true}, v\rangle^l \Rrightarrow v'}\ \textsc{EP\_ROK} \qquad \frac{}{\langle \{x{:}T \mid e_1\}, \mathsf{false}, v\rangle^l \Rrightarrow \Uparrow l}\ \textsc{EP\_RFail}$$

$$\frac{\begin{array}{c} x{:}T_{11} \to T_{12} \neq x{:}T_{21} \to T_{22} \\ T_{11} \Rrightarrow T_{11}' \qquad T_{12} \Rrightarrow T_{12}' \qquad T_{21} \Rrightarrow T_{21}' \qquad T_{22} \Rrightarrow T_{22}' \qquad v \Rrightarrow v' \end{array}}{\begin{array}{l} \langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l\, v \Rrightarrow \\ \qquad \lambda x{:}T_{21}'.\ (\langle T_{12}'[\langle T_{21}' \Rightarrow T_{11}'\rangle^l\, x/x] \Rightarrow T_{22}'\rangle^l\, (v'\, (\langle T_{21}' \Rightarrow T_{11}'\rangle^l\, x))) \end{array}}\ \textsc{EP\_RFun}$$

$$\frac{\forall\alpha.T_1 \neq \forall\alpha.T_2 \qquad T_1 \Rrightarrow T_1' \qquad T_2 \Rrightarrow T_2' \qquad v \Rrightarrow v'}{\langle \forall\alpha.T_1 \Rightarrow \forall\alpha.T_2\rangle^l\, v \Rrightarrow \Lambda\alpha.\ (\langle T_1' \Rightarrow T_2'\rangle^l\, (v'\,\alpha))}\ \textsc{EP\_RForall}$$

$$\frac{}{e \Rrightarrow e}\ \textsc{EP\_Refl} \qquad \frac{T_1 \Rrightarrow T_1' \quad e_{12} \Rrightarrow e_{12}'}{\lambda x{:}T_1.\ e_{12} \Rrightarrow \lambda x{:}T_1'.\ e_{12}'}\ \textsc{EP\_Abs} \qquad \frac{e_1 \Rrightarrow e_1' \quad e_2 \Rrightarrow e_2'}{e_1\ e_2 \Rrightarrow e_1'\ e_2'}\ \textsc{EP\_App}$$

$$\frac{e \Rrightarrow e'}{\Lambda\alpha.\ e \Rrightarrow \Lambda\alpha.\ e'}\ \textsc{EP\_TAbs} \qquad \frac{e_1 \Rrightarrow e_1' \quad T_2 \Rrightarrow T_2'}{e_1\ T_2 \Rrightarrow e_1'\ T_2'}\ \textsc{EP\_TApp}$$

$$\frac{e_i \Rrightarrow e_i'}{\mathrm{op}\,(e_1,\,\ldots,\,e_n) \Rrightarrow \mathrm{op}\,(e_1',\,\ldots,\,e_n')}\ \textsc{EP\_Op} \qquad \frac{T_1 \Rrightarrow T_1' \quad T_2 \Rrightarrow T_2'}{\langle T_1 \Rightarrow T_2\rangle^l \Rrightarrow \langle T_1' \Rightarrow T_2'\rangle^l}\ \textsc{EP\_Cast}$$

$$\frac{T \Rrightarrow T' \quad e \Rrightarrow e'}{\langle T, e, k\rangle^l \Rrightarrow \langle T', e', k\rangle^l}\ \textsc{EP\_Check} \qquad \frac{}{E\,[\Uparrow l] \Rrightarrow \Uparrow l}\ \textsc{EP\_Blame}$$

Parallel type reduction $\boxed{T_1 \Rrightarrow T_2}$

$$\frac{}{T \Rrightarrow T}\ \textsc{EP\_TRefl} \qquad \frac{\sigma_1 \longrightarrow^* \sigma_2 \quad T_1 \Rrightarrow T_2}{\{x{:}T_1 \mid \sigma_1(e)\} \Rrightarrow \{x{:}T_2 \mid \sigma_2(e)\}}\ \textsc{EP\_TRefine}$$

$$\frac{T_1 \Rrightarrow T_1' \quad T_2 \Rrightarrow T_2'}{x{:}T_1 \to T_2 \Rrightarrow x{:}T_1' \to T_2'}\ \textsc{EP\_TFun} \qquad \frac{T \Rrightarrow T'}{\forall\alpha.T \Rrightarrow \forall\alpha.T'}\ \textsc{EP\_TForall}$$

Figure 3.8: Parallel reduction

Conversion $\boxed{\sigma_1 \longrightarrow^* \sigma_2}$ $\qquad$ $\boxed{T_1 \equiv T_2}$

$$\sigma_1 \longrightarrow^* \sigma_2 \iff \begin{array}{l} \mathrm{dom}(\sigma_1) = \mathrm{dom}(\sigma_2) \wedge \\ \forall x \in \mathrm{dom}(\sigma_1).\ \sigma_1(x) \longrightarrow^* \sigma_2(x) \wedge \\ \forall \alpha \in \mathrm{dom}(\sigma_1).\ \sigma_1(\alpha) = \sigma_2(\alpha) \end{array}$$

$$\frac{}{\alpha \equiv \alpha}\ \text{C\_Var} \qquad \frac{}{B \equiv B}\ \text{C\_Base} \qquad \frac{\sigma_1 \longrightarrow^* \sigma_2 \quad T_1 \equiv T_2}{\{x{:}T_1 \mid \sigma_1(e)\} \equiv \{x{:}T_2 \mid \sigma_2(e)\}}\ \text{C\_Refine}$$

$$\frac{T_1 \equiv T_1' \quad T_2 \equiv T_2'}{x{:}T_1 \to T_2 \equiv x{:}T_1' \to T_2'}\ \text{C\_Fun} \qquad \frac{T \equiv T'}{\forall \alpha.\, T \equiv \forall \alpha.\, T'}\ \text{C\_Forall}$$

$$\frac{T_2 \equiv T_1}{T_1 \equiv T_2}\ \text{C\_Sym} \qquad \frac{T_1 \equiv T_2 \quad T_2 \equiv T_3}{T_1 \equiv T_3}\ \text{C\_Trans}$$

Figure 3.9: Type conversion via common subexpression reduction

showing that that $T_2[e_2/x]$ *parallel reduces* to $T_2[e_2'/x]$. That is, we can take some number of reduction steps in parallel (one for each free occurrence of x) from $T_2[e_2/x]$ to $T_2[e_2'/x]$. I then show that the denotations of such types are equal (Lemma 2.3.17), and then prove *semantic* type soundness for λ_{H} with respect to those denotations. Note that Lemma 2.3.17 depends on a long Coq development showing *cotermination*: if $e_1 \Rrightarrow e_2$, then $e_1 \longrightarrow^* v_1$ iff $e_2 \longrightarrow^* v_2$ such that $v_1 \Rrightarrow v_2$ (Lemma `A20` in `thy.v`)—more on this later.

The situation for λ_{H} is somewhat unsatisfying. We set out to prove syntactic type soundness and ended up proving semantic type soundness along the way. While not a serious burden for a language as small as λ_{H}, having to use semantic techniques throughout makes adding some features—polymorphism, state and other effects, concurrency—difficult. For example, a semantic proof of type soundness for F_{H} is very close to a proof of parametricity—must we prove parametricity while proving type soundness?

In originally doing the work in this chapter (Belo et al. [8]), we observed that we could get rid of subtyping and explicitly use the symmetric, transitive closure of parallel reduction as the conversion relation. (Parallel reduction is reflexive by definition.) Between that (and a few other changes), we found subtyping no longer necessary. We still, however, needed cotermination for parallel reduction. We also explicitly depended on *substitutivity*: if $e_1 \Rrightarrow e_2$ and $e_1' \Rrightarrow e_2'$ then $e_1[e_1'/x] \Rrightarrow e_2[e_2'/x]$. It turns out that the proof of cotermination for λ_{H} also needs substitutivity, Lemma `A3` in `thy.v`, but we needed it for the substitution in types discussed above. Unfortunately, parallel reduction in F_{H} is *not* substitutive [60]. There are two counterexamples, both in Figure 3.10.

Why doesn't substitutivity hold in F_{H}, when it did (so easily) in λ_{H}? There are two reasons. First, the cast semantics of F_{H} is much more complicated than that of λ_{H} (six rules, as opposed to two). The F_{H} rules depend on upon certain (syntactic) equalities between types—both counterexamples in Figure 3.10 take advantage these

Counterexample 1

Let T be a type with a free variable x.

$$\begin{array}{rcl} e_1 & = & \langle T \Rightarrow \{y{:}T[5/x] \mid \mathsf{true}\}\rangle^l\, 0 \\ e_2 & = & \langle T[5/x] \Rightarrow \{y{:}T[5/x] \mid \mathsf{true}\}\rangle^l\, (\langle T \Rightarrow T[5/x]\rangle^l\, 0) \\ e_1' = e_2' & = & 5 \end{array}$$

Observe that $e_1' \Rrightarrow e_2'$ (by EP_Refl) and $e_1 \Rrightarrow e_2$ (by EP_RPreCheck) but $e_1[5/x] = \langle T[5/x] \Rightarrow \{y{:}T[5/x] \mid \mathsf{true}\}\rangle^l\, 0 \Rrightarrow \langle \{y{:}T[5/x] \mid \mathsf{true}\}, \mathsf{true}, 0\rangle^l$ by EP_RCheck, not $e_2[5/x]$.

Counterexample 2

Let T_2 be a type with a free variable x.

$$\begin{array}{rcl} e_1 & = & \langle T_1 \to T_2 \Rightarrow T_1 \to T_2[5/x]\rangle^l\, v \\ e_2 & = & \lambda y{:}T_1.\ \langle T_2 \Rightarrow T_2[5/x]\rangle^l\, (v\, (\langle T_1 \Rightarrow T_1\rangle^l\, y)) \\ e_1' = e_2' & = & 5 \end{array}$$

Observe that $e_1' \Rrightarrow e_2'$ (by EP_Refl) and $e_1 \Rrightarrow e_2$ (by EP_RFun). We have $e_1[5/x] = \langle T_1 \to T_2[5/x] \Rightarrow T_1 \to T_2[5/x]\rangle^l\, v \Rrightarrow v[5/x]$ by EP_RRefl, not $e_2[5/x]$.

Figure 3.10: Counterexamples to substitutivity of parallel reduction in $\mathrm{F_H}$

equalities to break substitutivity. Second, λ_{H} treats $\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l\, v$ as a value, while it is a redex in $\mathrm{F_H}$. This syntactic coincidence makes an exact cotermination lemma possible. But as the second counterexample shows, in $\mathrm{F_H}$ it's possible to have a substitution introduce a function proxy in e_2 but not e_1. While I conjecture that e_1 and e_2 are *contextually* equivalent, they won't yield values that parallel reduce to each other. The rules that are the source of the problem for substitutivity of parallel reduction are the EP_R... rules, where a reduction in the outer term happens at the same time as parallel reductions deep inside the term. Note that both counterexamples make use of such rules.

The semantics of $\mathrm{F_H}$ as published in ESOP 2011 are wrong. To fix them, I introduced a simpler conversion relation, defined in Figure 3.9. Instead of allowing full parallel reduction, I restrict convertible types to the CSR $\sigma_1 \longrightarrow^* \sigma_2$, i.e., substitutions over the same set of term and type variables where (a) every term binding in σ_1 reduces to its corresponding binding in σ_2, and (b) the type bindings are identical. My conversion relation is essentially the symmetric, transitive closure[5] of parallel reduction—*without* these reducing rules.

Phrasing the conversion relation in terms of CSR gives us substitutivity nearly automatically, but cotermination remains an issue. In Conjecture 3.2.1, I suggest that terms related by CSR coterminate at true; this is enough to prove type soundness and parametricity of $\mathrm{F_H}$.

It is unclear if cotermination of parallel reduction holds in $\mathrm{F_H}$ despite the absence

[5]I prove that my relation is reflexive in Lemma 3.2.3.

of substitutivity, i.e., whether if $e_1 \Rrightarrow e_2$ then $e_1 \longrightarrow^* v_1$ iff $e_2 \longrightarrow v_2$ such that $v_1 \Rrightarrow v_2$. But it turns out that we can get by with a simpler property: cotermination at true, rather than at arbitrary values. This property is a corollary of cotermination, since $\mathsf{true} \Rrightarrow \mathsf{true}$, but it is less likely to be interfered with by function proxies which may be introduced as in the second counterexample. The intuition that leads me to believe that cotermination at true holds for CSR is that in a well typed program, any extra checks or function proxies introduced due to differing substitutions must eventually disappear if the type of the final expression is Bool.

I believe that weak bisimulation is a promising proof technique: it's syntactic enough to avoid issues of circularity with typing, but semantic enough to relate terms that are related by $\longrightarrow^*$ reductions. I think *weak* bisimulations in particular are appropriate, because of the need for $\longrightarrow^*$ reductions on both sides of the relation. The system as defined may make such a proof slightly difficult. Recalling the first counterexample to substitutivity, we will need to have $e_1[e_1'/x]$ and $e_2[e_2'/x]$ in the relation, but how can the relation "remember" that any checks that occur on one side but not the other inevitably succeed? Introducing explicit tagging, as I do in Chapter 4, is an attractive approach to solving this technical problem. In an explicitly tagged manifest contract system, the only values inhabiting refinement types are tagged as such, e.g., $(v, \{x{:}T \mid e\})$; the operational semantics then manages tags on values, tagging in E_CheckOK and untagging in E_Forget. Explicit tagging has several advantages: it clarifies the staging of the operational semantics; it eliminates the need for a T_Forget rule; it gives value inversion directly (Lemma 3.2.11). Finally, any proof of cotermination at true (Conjecture 3.2.1) must be careful to *not* rely on type soundness, preservation, or substitution properties. Those theorems in $\mathrm{F_H}$ rely on Conjecture 3.2.1, so we can't use them in its proof.

Finally: what kind of calculus *wouldn't* have cotermination at true? In a non-deterministic language, CSR may make one choice with σ_1 and another with σ_2. Fortunately, $\mathrm{F_H}$ is deterministic. In a deterministic language, cotermination at true may not hold for CSR if the evaluation relation abuses equalities that are violated by reduction. $\mathrm{F_H}$'s semantics *does* use equalities that are violated by term reduction; I believe that "abuse" means using an equality on part of a term to determine which step to take, but then ignoring that part of the term later in evaluation. Since $\mathrm{F_H}$ *doesn't* do that, I am confident enough to conjecture that cotermination at true holds.

3.6 Conclusion

This chapter presented a simpler approach to manifest contract calculi, which I applied to defining $\mathrm{F_H}$, a parametrically polymorphic manifest contract calculus. When I say "parametrically" polymorphic, I mean in particular that the relation R used to related terms at type variables in the logical relation is a *parameter* of the logical re-

lation, which admits any instantiation of R.[6] I offered the first operational semantics for *general* refinements, where refinements can apply to any type, not just base types. Finally, I defined a *post facto* subtyping relation, proving that "upcasts" from subtypes to supertypes always succeed in $\mathrm{F_H}$, i.e., that subtyping is sound. This recovers the reasoning principles lost when we left subtyping out of the language definition.

[6]Earlier versions [8] only admit relations that respect parallel reduction, but that restriction has been relaxed.

Chapter 4

Space-efficient manifest contracts

> Space and Time! now I see it is true, what I guessd at,
> What I guessd when I loafd on the grass,
> What I guessd while I lay alone in my bed,
> And again as I walkd the beach under the paling stars of the morning.
>
> *Song of Myself*
> Walt Whitman

There has been a great deal of investigation into so-called *full-spectrum* programming languages, combining dynamic types, simple types, and contract-style refinements. The promise of a single language admitting the full development cycle—from a small script to a more manageable, statically typed program to a robust, verified system—has held great allure for some time. Prior attempts to fulfill this promise have attacked the problem piecemeal: script to program, and program to verified program. This dissertation so far is no exception: we have studied contracts as an extension on top of simple types (Chapter 2) and System F (Chapter 3).

I find it useful to think of one axis of the design space as a spectrum of expressivity, ranging from dynamic types on the left to dependent refinement types on the right (illustrated in Figure 4.1). I call it the "dyn/refine spectrum". On the one hand, work in the script-to-program category goes back at least to Abadi et al.'s work with type Dynamic, with more recent work falling under Siek and Taha's rubric of "gradual typing" [1, 73, 67, 46, 74, 4, 65, 23, 18, 39, 68]. On the other hand, program verification is an enormous field in its own right; in this dissertation, I focus on dynamic enforcement methods in general and contracts in particular. We have seen that some more recent work takes a type-oriented, or *manifest*, approach to contracts, allowing so-called *refinement types* of the form $\{x{:}T \mid e\}$, inhabited by values v that satisfy the *predicate* e, i.e. $e[v/x] \longrightarrow^* \mathsf{true}$.

Over the last decade, the state of the art combining these two paradigms—in gradual types and manifest contracts in particular—has steadily progressed [10, 45, 78, 51]. Starting with Sage [45] and continuing with Wadler and Findler [78], many

dynamic types $\longleftrightarrow$ simple types $\longleftrightarrow$ refinement types $\longleftrightarrow$ dependency

Figure 4.1: The dyn/refine spectrum of cast expressiveness

languages have expressed the interactions between dynamic and simple static types and between simple static types and refinement types using a single syntactic form: the cast. The previous two chapters are no exception.

Casts are promising. They offer a unified view of changes in type information, have straightforward operational semantics, and enjoy a fruitful relationship with subtyping (see [78, 65, 44, 34, 68, 8]). The space-efficient full-spectrum language I develop in this chapter derives its semantics from a full-spectrum language with casts. Before we can continue, I must explain how casts work with Dyn, the dynamic type.

The first consequence of introducing Dyn to our language is the introduction of nontermination: in earlier chapters, the typing regime prevented divergence. But it is easy to write a nonterminating program using Dyn—we must simply deal with it. In any case, this is a theoretical difficulty; pragmatically, I *want* languages which have general recursion (and, so, the potential for nontermination).

As for casts into and out of Dyn, a cast of the form $\langle \mathsf{Int} \Rightarrow \mathsf{Dyn} \rangle\ 5$ asks for the number 5, which has type Int, to be treated as a value of type Dyn, the dynamic type. The operational semantics of such a cast will mark the value with a tag, as in $5_{\mathsf{Int!}}$. Similarly, a cast of the form $\langle \mathsf{Dyn} \Rightarrow \mathsf{Int} \rangle\ 5_{\mathsf{Int!}}$ will project the tagged integer out of type dynamic, yielding the original value 5. In the case where the cast's argument isn't tagged correctly, the cast must raise an error. Consider the term $\langle \mathsf{Dyn} \Rightarrow \mathsf{Int} \rangle\ \mathsf{true}_{\mathsf{Bool!}}$. It tries to project an Int out of type dynamic, but the dynamic value is really a Bool—a type error. All we can reasonably do is abort the program, evaluating to the uncatchable exception fail.[1]

We have already seen that casts on functions are the most interesting casts: values with functional casts on them, like $\langle T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22} \rangle\ v_1$, are themselves values; they are *wrapped* with a *function proxy*. When such wrapped values are applied to a value v_2, the cast unfolds:

$$(\langle T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22} \rangle\ v_1)\ v_2 \longrightarrow \langle T_{12} \Rightarrow T_{22} \rangle\ (v_1\ (\langle T_{21} \Rightarrow T_{11} \rangle\ v_2))$$

Note that this rule is contravariant in the domain. I use a different semantics in Chapter 3, where I introduce new lambdas. Since our subject in this chapter is space efficiency, introducing extra closures for explicit function proxies is a non-starter.

One problem common to calculi with casts is the problem of *space efficiency*. In particular, casts can accumulate in an unbounded way: redundant casts can grow the stack arbitrarily; casting functions can introduce an arbitrary number of function

[1] I forgo blame in this chapter, leaving it as future work. Since my final result is an inexact relationship between the naïve and space-efficient calculi, tracking blame doesn't seem particularly useful—it won't match up.

proxies. This unbounded growth of casts can, in the extreme, change the asymptotic complexity of programs. To see why the naïve treatment isn't space efficient, recall the mutually recursive definition of even and odd from Section 1.3, adapted from Herman et al. [39]:

$$
\begin{array}{l}
\mathsf{even} : \mathsf{Dyn}{\to}\mathsf{Dyn} = \langle \mathsf{Int}{\to}\mathsf{Bool} \Rightarrow \mathsf{Dyn}{\to}\mathsf{Dyn} \rangle \ \lambda x{:}\mathsf{Int}. \\
\quad \mathsf{if}\ x = 0\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{odd}\ (x - 1) \\
\mathsf{odd} : \mathsf{Int}{\to}\mathsf{Bool} = \lambda x{:}\mathsf{Int}. \\
\quad \mathsf{if}\ x = 0\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ (\langle \mathsf{Dyn}{\to}\mathsf{Dyn} \Rightarrow \mathsf{Int}{\to}\mathsf{Bool} \rangle\ \mathsf{even})\ (x - 1)
\end{array}
$$

In this example, even is written in a more dynamically typed style than odd. (While this example is contrived, it is easy to imagine mutually recursive modules with dynamic typing and refinement types. For example, in PLT Racket [55], Typed Racket [74] modules interoperate with a number of untyped Racket modules. In those cases, it's possible to accumulate an unbounded number of redundant checks on the stack or as function proxies, e.g., on continuations or callbacks.) The cast $\langle \mathsf{Int}{\to}\mathsf{Bool} \Rightarrow \mathsf{Dyn}{\to}\mathsf{Dyn} \rangle\ (\lambda x{:}\mathsf{Int}.\ \ldots)$ in the definition of even will (a) check that even's dynamically typed arguments are in fact Ints and (b) tag the resulting booleans into the dynamic type. The symmetric cast on even in the definition of odd serves to make the function even behave as if it were typed. This cast will ultimately cast the integer value $n-1$ into the dynamic type, Dyn, as well as projecting even's result out of type Dyn and into type Bool. Now consider the reduction sequence in Figure 4.2, observing how the number of coercions grows (redexes are highlighted).

While the operational semantics doesn't have an explicit stack, we can still see the accumulation of pending casts. What *were* tail-recursive calls are accumulating extra work in the continuation.[2] What's more, the work is redundant: we must tag and untag true twice. In short, casts have taken an algorithm that should use a constant amount of stack space and turned it into an algorithm that uses $O(n)$ stack space. Such a large space overhead is impractical: casts aren't space efficient.

The same holds true for casts into and out of refinement types. Consider a library of drawing primitives based around painters, functions of type Canvas→Canvas. An underlying graphics library offers basic functions for manipulating canvases and functions over canvases, e.g., primFlipH : (Canvas→Canvas)→(Canvas→Canvas) flips an image on its horizontal axis. A wrapper library may add derived functions while re-exporting the underlying functions with refinement types specifying a square canvas dimensions, where $\mathsf{SquareCanvas} = \{x{:}\mathsf{Canvas} \mid \mathsf{width}(x) = \mathsf{height}(x)\}$:

$$
\begin{array}{lcl}
\mathsf{flipH}\ p & = & \langle \mathsf{Canvas}{\to}\mathsf{Canvas} \Rightarrow \mathsf{SquareCanvas}{\to}\mathsf{SquareCanvas} \rangle \\
& & \quad (\mathsf{primFlipH}\ (\langle \mathsf{SquareCanvas}{\to}\mathsf{SquareCanvas} \Rightarrow \mathsf{Canvas}{\to}\mathsf{Canvas} \rangle\ p))
\end{array}
$$

The wrapper library only accepts painters with appropriately refined types, but must strip away these refinements before calling the underlying implementation—which de-

[2] The call to even in the else branch of odd doesn't look like a tail call, but if the coercions are inserted automatically—as in Herman et al.—then it can be very difficult to tell what is and isn't a tail call. See related work (Chapter 5) for a discussion of cast insertion.

$$
\begin{array}{ll}
 & \mathsf{odd}\ 3 \\
\longrightarrow & (\langle \mathsf{Dyn}\to\mathsf{Dyn} \Rightarrow \mathsf{Int}\to\mathsf{Bool}\rangle\ \mathsf{even})\ 2 \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\mathsf{even}\ (\langle \mathsf{Int} \Rightarrow \mathsf{Dyn}\rangle\ 2)) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\mathsf{even}\ 2_{\mathsf{Int!}}) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ ((\langle \mathsf{Int}\to\mathsf{Bool} \Rightarrow \mathsf{Dyn}\to\mathsf{Dyn}\rangle\ (\lambda x{:}\mathsf{Int}.\ \ldots))\ 2_{\mathsf{Int!}}) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)\ (\langle \mathsf{Dyn} \Rightarrow \mathsf{Int}\rangle\ 2_{\mathsf{Int!}}))) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)\ 2)) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ (\mathsf{odd}\ 1)) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ ((\langle \mathsf{Dyn}\to\mathsf{Dyn} \Rightarrow \mathsf{Int}\to\mathsf{Bool}\rangle\ \mathsf{even})\ 0)) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ (\langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\mathsf{even}\ (\langle \mathsf{Int} \Rightarrow \mathsf{Dyn}\rangle\ 0)))) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ (\langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\mathsf{even}\ 0_{\mathsf{Int!}}))) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ (\langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle \\
 & \quad ((\langle \mathsf{Int}\to\mathsf{Bool} \Rightarrow \mathsf{Dyn}\to\mathsf{Dyn}\rangle\ (\lambda x{:}\mathsf{Int}.\ \ldots))\ 0_{\mathsf{Int!}}))) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ (\langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle \\
 & \quad ((\lambda x{:}\mathsf{Int}.\ \ldots)\ (\langle \mathsf{Dyn} \Rightarrow \mathsf{Int}\rangle\ 0_{\mathsf{Int!}}))))) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ (\langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle \\
 & \quad ((\lambda x{:}\mathsf{Int}.\ \ldots)\ 0)))) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ (\langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ \mathsf{true}))) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ (\langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ \mathsf{true}_{\mathsf{Bool!}})) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ (\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle\ \mathsf{true}) \\
\longrightarrow & \langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle\ \mathsf{true}_{\mathsf{Bool!}} \\
\longrightarrow & \mathsf{true}
\end{array}
$$

Figure 4.2: Space-inefficient reduction

mands Canvas→Canvas painters. The wrapper library then has to cast these modified functions *back* to the refined types. Calling flipH (flipH p) will yield:

$$
\begin{array}{l}
\langle \mathsf{Canvas}\to\mathsf{Canvas} \Rightarrow \mathsf{SquareCanvas}\to\mathsf{SquareCanvas}\rangle \\
\quad (\mathsf{primFlipH} \\
\quad\quad (\langle \mathsf{SquareCanvas}\to\mathsf{SquareCanvas} \Rightarrow \mathsf{Canvas}\to\mathsf{Canvas}\rangle \\
\quad\quad\quad (\langle \mathsf{Canvas}\to\mathsf{Canvas} \Rightarrow \mathsf{SquareCanvas}\to\mathsf{SquareCanvas}\rangle\ (\mathsf{primFlipH}\ p'))))
\end{array}
$$

where $\langle \mathsf{SquareCanvas}\to\mathsf{SquareCanvas} \Rightarrow \mathsf{Canvas}\to\mathsf{Canvas}\rangle\ p \longrightarrow^* p'$

That is, we first cast p to a plain painter and return a new painter p'. We then cast p' into and then immediately out of the refined type, before continuing on to flip p'. All the while, we are accumulating many function proxies beyond the wrapping that the primitive library is doing. A space-efficient scheme for manifest contracts avoids accumulating these extra function proxies on p'.

Two of the existing approaches to space efficiency in the world of gradual typing [39, 65, 68] factor casts into their constituent *coercions*, adapting Henglein's system [38]. I take a similar approach. For example, the cast $\langle \mathsf{Dyn} \Rightarrow \mathsf{Bool}\rangle$, which checks that a dynamic value is a boolean and then untags it, is written as the coercion Bool?; the cast $\langle \mathsf{Bool} \Rightarrow \mathsf{Dyn}\rangle$, which tags a boolean into type dynamic, is written Bool!. Most importantly, coercions can be composed, so $\langle \mathsf{Bool?}\rangle\ (\langle \mathsf{Bool!}\rangle\ e) \longrightarrow \langle \mathsf{Bool!}; \mathsf{Bool?}\rangle\ e$.

Herman et al. normalize the coercion $\mathsf{Bool!};\mathsf{Bool?}$ into the no-op coercion Id. This normalization process is how they achieve space efficiency. For example:

$$\langle \mathsf{Bool?} \rangle\ (\langle \mathsf{Bool!} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)\ (\langle \mathsf{Int?} \rangle\ 2_{\mathsf{Int!}}))) \longrightarrow \langle \mathsf{Id} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)\ (\langle \mathsf{Int?} \rangle\ 2_{\mathsf{Int!}}))$$

This composition and normalization of pending coercions allows them to prove a bound on the size of any coercion that occurs during the run of a given program. This bound effectively restores the possibility of tail-call optimization.

However, it isn't obvious how to extend Herman et al.'s [39] coercion system to refinement types. When do we test that values satisfy predicates? How do refinement type coercions normalize? I show that refinement types should have a checking coercion $\{x{:}T \mid e\}?$ and an (un)tagging coercion $\{x{:}T \mid e\}!$; the key insight for space efficiency is that the composition $\{x{:}T \mid e\}?;\{x{:}T \mid e\}!$ should normalize to Id, i.e., checks that are immediately forgotten should be thrown away. Throwing away checks sounds dangerous, but the calculus is still sound—values typed at refinement types must satisfy their refinements. On the one hand, this is great news—space-efficiency is not only more practical, but there are fewer errors! On the other hand, the space-efficient semantics aren't *exactly* equivalent to the naïve semantics. Whether or not this is good news, these dropped checks are part and parcel of space efficiency. I develop this idea in detail in Section 4.5; I show that this means that space-efficient programs fail less often than their naïve counterparts in Section 4.6.

In this chapter, I make several contributions, extending the existing solutions in a number of dimensions. In particular, I:

- Present a new approach to coercions that extends the earlier work to refinement types in a novel formulation (Section 4.5);

- Introduce what is, at present, the most expressive full-spectrum contract language, offering type dynamic as well as refinements of both base types and the dynamic type (Section 4.3 and Section 4.5); this new language narrowly edges out Wadler and Findler [78] by including refinements of type dynamic;

- Show that this language is space efficient, i.e., there are a bounded number of coercions in any program, and those coercions are bounded in size (Section 4.7); and

- Offer proof my new language is sound with respect to the naïve, inefficient semantics: if the naïve semantics reaches a value, so does the space-efficient one, but occasionally, the naïve semantics will fail when the space-efficient one succeeds (Section 4.6).

Outline

In this long chapter, I begin by briefly overviewing my design philosophy in Section 4.1. I then define a cast calculus, Cast, with dynamic and refinement types (Section 4.2), followed by a naïve, space-inefficient coercion calculus, Naive, with

dynamic and refinement types (Section 4.3). I translate CAST terms into behaviorally equivalent NAIVE terms in Section 4.4. This proof justifies my eventual claim that I've made *manifest contracts* space-efficient: since CAST terms have exactly corresponding NAIVE terms, which are simulated by EFFICIENT terms, we know that EFFICIENT terms simulate CAST terms. If I didn't relate CAST and NAIVE, I would have invented invented a system that enjoys space efficiency—and looks like manifest contracts, if you squint. In Section 4.5 I make the naïve calculus into a space-efficient one, which I call EFFICIENT. In Section 4.6, I relate the two coercion calculi, showing that the two are *mostly* observationally equivalent: if the naïve semantics reduces a term to a value, so will the space efficient one; sometimes the naïve semantics will diverge or produce a failure when the space-efficient one succeeds. I give formal justification for my claim of space efficiency in Section 4.7.

4.1 Design philosophy

My design philosophy has three principles:

1. Base values have simple types; e.g., all integers are typed at Int.
2. We give operations types precise enough to guarantee totality; e.g., division has a type at least as precise as $\mathsf{Int}{\to}\{x{:}\mathsf{Int} \mid x \neq 0\}{\to}\mathsf{Int}$.
3. Values satisfy their refinement types; e.g., if $\emptyset \vdash v : \{x{:}T \mid e\}$, then $e[v/x] \longrightarrow^* \mathsf{true}$.

Giving base values simple types by avoids some of the technical problems seen in previous refinement type systems that gave constants most specific types [51, 28, 45, 44, 34]. These other systems set $\mathsf{ty}(k)$ such that if $e[k/x] \longrightarrow^* \mathsf{true}$, then $\mathsf{ty}(k) <: \{x{:}T \mid e\}$. We wish to avoid subtyping—and its concomitant circularities [34, 44, 8]. Beyond technical issues, simple types are a de facto default in functional programming; Hindley-Milner is a sweet spot in the design space. I believe that taking this sweet spot as the default programming paradigm will increase the usability and practicality of manifest contracts.

There is one exception to assigning values simple types: it makes sense to give null or $\mathsf{undefined}$ a dynamic type; attempting to coerce such a value to a simple type would result in an error. I omit such values—it's not clear that they're a good idea [42]. One alternative, due to Stephanie Weirich, is to have these dummy values inhabit *all* types, in which case they would never need to be tagged or checked at all, and elimination forms would raise errors.

To me, keeping operations total is the main goal of refinement types. In fact, the original use of refinement types by Freeman and Pfenning [30] was to make partial pattern matching total. The calculi here don't have algebraic datatypes, but partial operations are just as undesirable as partial pattern matches. This point of view is nice for implementations: primitive partial operations can be exposed at types strong

enough to guarantee safety; the coercion algorithm will insert appropriate checks which may or may not be optimized away. (This is *hybrid type checking* [28].)

The flipside of using refinements to protect partial operations is that refinement checks that don't end up protecting partial operations aren't as important as those that do. In Efficient in Section 4.5, we occasionally skip checking whether or not a value satisfies a refinement. We only do this when the program's next step would be to untag the value as having satisfied the refinement—why bother checking if we don't care about the result? This lenient approach to refinement type checking means that the naïve and space-efficient calculi don't behave exactly the same: some programs raise errors in the former but not the latter, precisely because we skip checks.

One might ask: if it's okay to have a ψ rule skip refinement type checks, why not do the same for checks when moving into and out of Dyn? The simplest explanation is that we *must* have a ψ rule for refinement types—without it, we won't have space efficiency. But the (safe) ϕ rule suffices when we consider Dyn. Since we *can* have the gradual typing parts of Naive and Efficient behave the same, we do. As a philosophical difference, I hold the line at the structure of values, i.e., simple types. If simple types are the default paradigm of the language, we want to avoid errors in this fragment as much as possible. As Siek and Wadler [68] say:

> [the] boundary between static and dynamic typing regions require[s] certain run-time checks to maintain the integrity of the static region.

The refined world has the structure of the simply typed world plus a predicate—it has more type information. On the other hand, dynamic types have less information, so we must be stricter with them. Put another way, simple types are about the broad structure of values, while refinements are about the safety of operations. If we never end up running the operation whose safety is ensured by a refinement type, no problem; the programmer's specification may be too tight. Polymorphism aside, there is no such thing as a simple type that is "too tight".

My third principle is a compromise between stringency and lenience. If a value is typed at a refinement type, then it satisfies its predicate. All well-typed refinement calculi have value inversion—it is a consequence of subject reduction. Having only value inversion instead of stronger reasoning principles (as in the semantics of Xu et al. [80]) is a compromise because some checks can be skipped (lenience), but once the program reduces to a value it must actually inhabit its type (stringency). This *value inversion* principle is valuable, and one that we have used in practice when programming with refinement types. In call-by-value (CBV) languages (including all of the calculi in this thesis), value inversion lets programmers reason easily about functions which take refined inputs.

Value inversion is the *least* that a refinement type can mean in a sound system: any less and type soundness won't hold. While every sound refinement type system has value inversion, I find it desirable to be able to directly invert the typing derivation to find that a value satisfies the predicate of its refinement type, rather than applying a

Types and base types

$$
\begin{array}{lcl}
T & ::= & B \mid T_1 \to T_2 \mid \{x{:}B \mid e\} \mid \mathsf{Dyn} \mid \{x{:}\mathsf{Dyn} \mid e\} \\
B & ::= & \mathsf{Bool} \mid \mathsf{Int} \mid \ldots
\end{array}
$$

Terms, results, values, and pre-values

$$
\begin{array}{lcl}
e & ::= & x \mid r \mid op(e_1, \ldots, e_n) \mid e_1\ e_2 \mid \langle T_1 \Rightarrow T_2 \rangle\ e \mid \langle \{x{:}T \mid e_1\}, e_2, v \rangle \\
r & ::= & v \mid \mathsf{fail} \\
v & ::= & u_{\mathsf{Id}} \mid v_{B!} \mid v_{\mathbf{Fun!}} \mid v_{\{x:T \mid e\}?} \mid v_{\langle T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22} \rangle} \\
u & ::= & k \mid \lambda x{:}T.\ e
\end{array}
$$

Typing contexts

$$
\begin{array}{lcl}
\Gamma & ::= & \emptyset \mid \Gamma, x{:}T
\end{array}
$$

Figure 4.3: Cast syntax

more complicated or general reasoning principle. I believe that having value inversion directly is easier for programmers to understand and use.

4.2 A cast calculus

In this section, we define Cast, a cast calculus with dynamic and refinement types. It is a small extension of Wadler and Findler's system [78].

I define the syntax and static semantics in Section 4.2.1; the operational semantics are in Section 4.2.2. I prove type soundness in Section 4.2.3.

Rule naming conventions

Before I begin my technical work in earnest, a word about conventions. We are defining three calculi: one with casts (Cast) and two with coercions (Naive and Efficient). The latter two largely share syntax and typing rules; all typing rules will be of the form T_Name. In general, we will rely on context to differentiate terms. The evaluation rules for Cast, in this section, are named G_Name, using a subscripted arrow $\longrightarrow_c$. The evaluation rules for the naïve calculus, Naive, in Section 4.3 are named F_Name, using a subscripted arrow $\longrightarrow_n$ for the reduction relation; the space-efficient evaluation rules for Efficient are in Section 4.5 are named E_Name using a plain arrow $\longrightarrow$.

4.2.1 Syntax and typing

I give the syntax of Cast in Figure 4.3. The syntax is much like those of Chapters 2 and 3: I extend the simply typed lambda calculus with Dyn, refinements of base types and dynamic, casts, active checks, and failures. I sometimes write $\{x{:}T \mid e\}$ when it doesn't matter whether the underlying type is Dyn or B. Notice that

Well formed contexts and types $\boxed{\vdash \Gamma}$ $\boxed{\vdash T}$

$$\frac{}{\vdash \emptyset}\ \text{WF\_Empty} \qquad \frac{\vdash \Gamma \quad \vdash T}{\vdash \Gamma, x{:}T}\ \text{WF\_Extend}$$

$$\frac{}{\vdash B}\ \text{WF\_Base} \qquad \frac{}{\vdash \mathsf{Dyn}}\ \text{WF\_Dyn} \qquad \frac{\vdash T_1 \quad \vdash T_2}{\vdash T_1 \to T_2}\ \text{WF\_Fun}$$

$$\frac{x{:}T \vdash e : \mathsf{Bool} \quad T = B \text{ or } T = \mathsf{Dyn}}{\vdash \{x{:}T \mid e\}}\ \text{WF\_Refine}$$

Similar types $\boxed{\vdash T_1 \parallel T_2}$

$$\frac{T \neq T_1 \to T_2}{\vdash T \parallel T}\ \text{P\_Id} \qquad \frac{}{\vdash \mathsf{Dyn} \parallel T}\ \text{P\_DynL} \qquad \frac{}{\vdash T \parallel \mathsf{Dyn}}\ \text{P\_DynR}$$

$$\frac{\vdash T_{11} \parallel T_{21} \quad \vdash T_{12} \parallel T_{22}}{\vdash T_{11} \to T_{12} \parallel T_{21} \to T_{22}}\ \text{P\_Fun}$$

$$\frac{\vdash T_1 \parallel T_2}{\vdash \{x{:}T_1 \mid e\} \parallel T_2}\ \text{P\_RefineL} \qquad \frac{\vdash T_1 \parallel T_2}{\vdash T_1 \parallel \{x{:}T_2 \mid e\}}\ \text{P\_RefineR}$$

Figure 4.4: Typing for Cast, part 1

refinements of dynamic indirectly include refinements of functions. At the cost of having even more canonical coercions in Section 4.5.1, we could add refinements of functions. I omit them because they would have brought complexity without new insights. Going beyond refinements of functions to the general refinements of $\mathrm{F_H}$ in Chapter 3, however, is challenging future work (see Chapter 6). The values in Cast are structured slightly differently from what I have presented so far—I separate *values* v from *pre-values* u. In particular, values have tags while pre-values are constants and lambdas.

The typing rules for Cast are in Figures 4.4 and 4.5. Most of the rule are standard. I follow Wadler and Findler [78] in tracking completed casts with explicit tags—this approach plays particularly well with the coercion-based approach in the sequel.

First, the well formedness rules for contexts and types are straightforward. In WF_Refine, we ensure that we only ever refine base types and Dyn; as for other refinement type rules, we must make sure that the refinement predicate is well typed. As we did for $\mathrm{F_H}$ (in Figure 3.4), we must define type similarity. Note that Dyn is similar to every type. We restrict the P_Id rule so that the notion of compatibility

Well typed terms and values $\boxed{\Gamma \vdash u : T}$ $\boxed{\Gamma \vdash e : T}$

$$\frac{\vdash \Gamma}{\Gamma \vdash k : \mathsf{ty}(k)}\ \text{T\_Const} \qquad \frac{\vdash T_1 \quad \Gamma, x{:}T_1 \vdash e_{12} : T_2}{\Gamma \vdash \lambda x{:}T_1.\ e_{12} : T_1 \to T_2}\ \text{T\_Abs}$$

$$\frac{\Gamma \vdash u : T}{\Gamma \vdash u_{\mathsf{Id}} : T}\ \text{T\_PreVal} \qquad \frac{\Gamma \vdash v : B}{\Gamma \vdash v_{B!} : \mathsf{Dyn}}\ \text{T\_TagB} \qquad \frac{\Gamma \vdash v : \mathsf{Dyn} \to \mathsf{Dyn}}{\Gamma \vdash v_{\mathbf{Fun}!} : \mathsf{Dyn}}\ \text{T\_TagFun}$$

$$\frac{\vdash \Gamma \quad \emptyset \vdash v : T \quad \vdash \{x{:}T \mid e\} \quad e[v/x] \longrightarrow_c^* \mathsf{true}_{\mathsf{Id}}}{\Gamma \vdash v_{\{x:T \mid e\}?} : \{x{:}T \mid e\}}\ \text{T\_TagRefine}$$

$$\frac{\Gamma \vdash v : T_{11} \to T_{12} \quad \vdash T_{21} \to T_{22} \quad \vdash T_{11} \to T_{12} \parallel T_{21} \to T_{22}}{\Gamma \vdash v_{\langle T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22} \rangle} : T_{21} \to T_{22}}\ \text{T\_Wrap}$$

$$\frac{\vdash \Gamma \quad x{:}T \in \Gamma}{\Gamma \vdash x : T}\ \text{T\_Var} \qquad \frac{\vdash T \quad \vdash \Gamma}{\Gamma \vdash \mathsf{fail} : T}\ \text{T\_Fail}$$

$$\frac{\Gamma \vdash e : T_1 \quad \vdash T_2 \quad \vdash T_1 \parallel T_2}{\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle\ e : T_2}\ \text{T\_Cast} \qquad \frac{\Gamma \vdash e_1 : (T_1 \to T_2) \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}\ \text{T\_App}$$

$$\frac{\mathsf{ty}(op) = T_1 \to \ldots \to T_n \to T \quad \Gamma \vdash e_i : T_i}{\Gamma \vdash op(e_1, \ldots, e_n) : T}\ \text{T\_Op}$$

$$\frac{\vdash \Gamma \quad \vdash \{x{:}T \mid e_1\} \quad \emptyset \vdash v : T \quad \emptyset \vdash e_2 : \mathsf{Bool} \quad e_1[v/x] \longrightarrow_c^* e_2}{\Gamma \vdash \langle \{x{:}T \mid e_1\}, e_2, v \rangle : \{x{:}T \mid e_1\}}\ \text{T\_CheckCast}$$

Figure 4.5: Typing for Cast, part 2

lines up more neatly with the operational semantics for casts.

The typing rules for terms are mostly standard. We have a specialized treatment of values to account for tags. T_PreVal types pre-values tagged with Id. T_TagB and T_TagFun type values that have been injected into the dynamic type, Dyn. T_TagRefine types values that have been checked as satisfying refinement types. Note that we structure this rule to apply only to closed values, but with a context to allow for weakening—as in Chapter 3. T_Wrap types values with function proxies. Otherwise, the rules should be familiar: T_Cast and T_CheckCast are standard, based on what we have seen so far.

It is worth taking a moment to comment on the type assignment functions $\mathsf{ty}(k)$ and $\mathsf{ty}(op)$ used in the T_Const and T_Op rules. In line with my philosophy (Section 4.1), the rule for constants gives them base types: $\mathsf{ty}(k) = B$. I require that no constants have, by default, type dynamic or a refinement type. By the same philosophy, the operator type assignment function $\mathsf{ty}(op)$ takes operations to first-order types that ensure totality. If $\mathsf{ty}(op) = T_1 \to \ldots \to T_n \to T$, then the operation's denotation $[\![op]\!]$ is a total function from $(T_1, \ldots, T_n)$ to T. If, for example, division is expressed as the operator div, then $\mathsf{ty}(\mathsf{div}) = \mathsf{Int} \to \{x{:}\mathsf{Int} \mid x \neq 0_{\mathsf{Id}}\} \to \mathsf{Int}$ or some similarly exact type. This property of operator types is critical: I believe that refinement types are meant to help programmers avoid failures in general, and failures of (fundamentally partial) primitive operations in particular.

To be clear, constants must have simple types, but operations on dynamic values are permitted—so long as they only look at the tag, not the underlying value. In particular, the proofs work for tag testing operations like isFun and $\mathsf{is}B$, but not for operations that extract and then apply functions tagged with **Fun**!.

4.2.2 Operational semantics

The operational semantics are defined in Figure 4.6 (for core rules) and Figure 4.7 (for cast rules). The rules implement straightforward tagging and checking operations.

The core rules should be unsurprising. G_Fun captures the behavior of function proxies in application positions. Much more interesting are the cast rules of Figure 4.7. Even though we don't have general refinements, I still adopt the cast staging techniques we used for $\mathrm{F_H}$ in Chapter 3. That is, G_CastCheck, G_CastPreCheck, and G_CastPredPred (which could just as easily be named G_CastForget) should be familiar. G_CastId is the reflexivity rule for casts, but limited to *not* apply for functions. This is done to mirror how coercions behave on function types; addressing this issue is future work (and discussed in Chapter 6).

G_CastFun and G_CastB handle injections into Dyn: the simply attach a type tag to the value. Note that G_CastFun only applies to functions of type $\mathsf{Dyn} \to \mathsf{Dyn}$. We use G_CastFunDyn to inject functions of other types into Dyn by stages.

G_CastFunFun and G_CastBB handle successful projections back out of Dyn; failed projections—mismatches between the target type of the cast and the tag on

$\boxed{e_1 \longrightarrow_c e_2}$

$$\frac{}{(\lambda x{:}T.\ e_{12})_{\mathsf{ld}}\ v_2 \longrightarrow_c e_{12}[v_2/x]}\ \text{G\_Beta} \qquad \frac{}{op(v_1, \ldots, v_n) \longrightarrow_c [\![op]\!]\,(v_1, \ldots, v_n)}\ \text{G\_Op}$$

$$\frac{}{v_{1\langle T_{11}\to T_{12}\Rightarrow T_{21}\to T_{22}\rangle}\ v_2 \longrightarrow_c \langle T_{12} \Rightarrow T_{22}\rangle\ (v_1\ (\langle T_{21} \Rightarrow T_{11}\rangle\ v_2))}\ \text{G\_Fun}$$

$$\frac{e_1 \longrightarrow_c e_1'}{e_1\ e_2 \longrightarrow_c e_1'\ e_2}\ \text{G\_AppL} \qquad \frac{e_2 \longrightarrow_c e_2'}{v_1\ e_2 \longrightarrow_c v_1\ e_2'}\ \text{G\_AppR}$$

$$\frac{e_i \longrightarrow_c e_i'}{op(v_1, \ldots, v_{i-1}, e_i, \ldots, e_n) \longrightarrow_c op(v_1, \ldots, v_{i-1}, e_i', \ldots, e_n)}\ \text{G\_OpInner}$$

$$\frac{e \longrightarrow_c e'}{\langle T_1 \Rightarrow T_2\rangle\ e \longrightarrow_c \langle T_1 \Rightarrow T_2\rangle\ e'}\ \text{G\_CastInner}$$

$$\frac{e_2 \longrightarrow_c e_2'}{\langle \{x{:}T \mid e_1\}, e_2, v\rangle \longrightarrow_c \langle \{x{:}T \mid e_1\}, e_2', v\rangle}\ \text{G\_CheckInner}$$

$$\frac{}{\mathsf{fail}\ e_2 \longrightarrow_c \mathsf{fail}}\ \text{G\_AppRaiseL} \qquad \frac{}{v_1\ \mathsf{fail} \longrightarrow_c \mathsf{fail}}\ \text{G\_AppRaiseR}$$

$$\frac{}{op(v_1, \ldots, v_{i-1}, \mathsf{fail}, \ldots, e_n) \longrightarrow_c \mathsf{fail}}\ \text{G\_OpRaise}$$

$$\frac{}{\langle T_1 \Rightarrow T_2\rangle\ \mathsf{fail} \longrightarrow_c \mathsf{fail}}\ \text{G\_CastRaise} \qquad \frac{}{\langle \{x{:}T \mid e\}, \mathsf{fail}, v\rangle \longrightarrow_c \mathsf{fail}}\ \text{G\_CheckRaise}$$

Figure 4.6: Cast operational semantics (core rules)

$\boxed{e_1 \longrightarrow_c e_2}$

$$\frac{}{\langle T \Rightarrow \{x{:}T \mid e\}\rangle\ v \longrightarrow_c \langle \{x{:}T \mid e\}, e[v/x], v\rangle}\ \text{G\_CastCheck}$$

$$\frac{}{\langle \{x{:}T \mid e\}, \mathsf{true}_{\mathsf{Id}}, v\rangle \longrightarrow_c v_{\{x:T|e\}?}}\ \text{G\_CheckOK}$$

$$\frac{}{\langle \{x{:}T \mid e\}, \mathsf{false}_{\mathsf{Id}}, v\rangle \longrightarrow_c \mathsf{fail}}\ \text{G\_CheckFail} \qquad \frac{T \neq T_1 \to T_2}{\langle T \Rightarrow T\rangle\ v \longrightarrow_c v}\ \text{G\_CastId}$$

$$\frac{}{\langle \mathsf{Dyn} \Rightarrow T_1 \to T_2\rangle\ v_{B!} \longrightarrow_c \mathsf{fail}}\ \text{G\_CastFunFailB}$$

$$\frac{}{\langle \mathsf{Dyn} \Rightarrow T_1 \to T_2\rangle\ v_{\mathbf{Fun}!} \longrightarrow_c \langle \mathsf{Dyn} \to \mathsf{Dyn} \Rightarrow T_1 \to T_2\rangle\ v}\ \text{G\_CastFunFun}$$

$$\frac{}{\langle B \Rightarrow \mathsf{Dyn}\rangle\ v \longrightarrow_c v_{B!}}\ \text{G\_CastB} \qquad \frac{}{\langle \mathsf{Dyn} \to \mathsf{Dyn} \Rightarrow \mathsf{Dyn}\rangle\ v \longrightarrow_c v_{\mathbf{Fun}!}}\ \text{G\_CastFun}$$

$$\frac{}{\langle \mathsf{Dyn} \Rightarrow B\rangle\ v_{B!} \longrightarrow_c v}\ \text{G\_CastBB} \qquad \frac{B \neq B'}{\langle \mathsf{Dyn} \Rightarrow B\rangle\ v_{B'!} \longrightarrow_c \mathsf{fail}}\ \text{G\_CastBFailB}$$

$$\frac{}{\langle \mathsf{Dyn} \Rightarrow B\rangle\ v_{\mathbf{Fun}!} \longrightarrow_c \mathsf{fail}}\ \text{G\_CastBFailFun}$$

$$\frac{}{\langle T_{11} \to T_{12} \Rightarrow \mathsf{Dyn}\rangle\ v \longrightarrow_c \langle \mathsf{Dyn} \to \mathsf{Dyn} \Rightarrow \mathsf{Dyn}\rangle\ (\langle T_{11} \to T_{12} \Rightarrow \mathsf{Dyn} \to \mathsf{Dyn}\rangle\ v)}\ \text{G\_CastFunDyn}$$

$$\frac{}{\langle T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22}\rangle\ v \longrightarrow_c v_{\langle T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22}\rangle}}\ \text{G\_CastFunWrap}$$

$$\frac{T_2 \neq \{x{:}T_1 \mid e\}}{\langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle\ v_{\{x:T_1|e\}?} \longrightarrow_c \langle T_1 \Rightarrow T_2\rangle\ v}\ \text{G\_CastPredPred}$$

$$\frac{T_1 \neq T_2 \quad T_1 \neq \{x{:}T_1' \mid e'\}}{\langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle\ v \longrightarrow_c \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle\ (\langle T_1 \Rightarrow T_2\rangle\ v)}\ \text{G\_CastPreCheck}$$

Figure 4.7: Cast operational semantics (cast rules)

		Target type			
Source type		$\{x{:}\mathsf{Dyn} \mid e\}$	Dyn	$\mathsf{Dyn}{\to}\mathsf{Dyn}$	$T_1{\to}T_2$
	$\{x{:}\mathsf{Dyn} \mid e\}$	I | (PP, PC, C)	PP	PP, FFB | FF, I	PP, FFB | (FF, FW)
	Dyn	C	I	FFB | (FF, I)	FFB | (FF, FW)
	$\mathsf{Dyn}{\to}\mathsf{Dyn}$	PC, F, C	F	FD, FW, F	FW
	$T_1{\to}T_2$	PC, FD, FW, F, C	FD, FW, F	FW	FW
	B	PC, B, C	B	not well typed	
	$\{x{:}B \mid e\}$	PP, PC, B, C	PP, B	not well typed	

		Target type	
Source type		B	$\{x{:}B \mid e\}$
	$\{x{:}\mathsf{Dyn} \mid e\}$	PP, FFB | BB	PP, PC, BFF | BFB | (BB, C)
	Dyn	BB | BFB | BFF	PC, BFF | BFB | (BB, C)
	$\mathsf{Dyn}{\to}\mathsf{Dyn}$	not well typed	
	$T_1{\to}T_2$	not well typed	
	B	I	C
	$\{x{:}B \mid e\}$	PP	I | (PP, PC, C)

The vertical axis is source types, the horizontal axis is target types. Rule names are abbreviated, where I means G_CastId, C means G_CastCheck, PC means G_CastPreCheck, etc. I use commas for sequencing and | for disjunction.

Table 4.1: Cast cast reductions, by type

the value—are handled by G_CastBFailB, G_CastBFailFun (for casts into base types B), and G_CastFunFailB (for casts into $\mathsf{Dyn}{\to}\mathsf{Dyn}$).

The G_CastFunWrap rule wraps functions with function proxies. This is the same as treating $\langle T_{11}{\to}T_{12} \Rightarrow T_{21}{\to}T_{22}\rangle\ v$ as a value, as we did in Chapter 2. We use a separate tag for symmetry's sake, since function proxies on values must be represented similarly in the sequel.

There are enough cast rules that it may not be immediately clear which rules apply when. In Table 4.1, we show how casts between each relevant type reduce. (I split out $\mathsf{Dyn}{\to}\mathsf{Dyn}$ because it is treated specially.) In order to fit the table easily on a single page, I use abbreviated rule names: a rule G_CastRuleName is written RN: I drop the G_Cast prefix and then take the initials of the rule. I use a regular expressions-like syntax: commas are sequential concatenation and | is disjunction. For example, a cast from $\{x{:}\mathsf{Dyn} \mid e\}$ to $\{x{:}B \mid e\}$ will run PP, PC, BFF | BFB | (BB, C): that is, it first runs G_CastPredPred, then G_CastPreCheck. We then check the tag, running G_CastBFailFun or G_CastBFailB if the tag is wrong. If the type in the case matches the tag on the value being cast, we run G_CastBB and then G_CastCheck.

4.2.3 Proofs

The proofs are entirely standard, culminating in a syntactic type soundness proof (Theorem 4.2.7) by progress and preservation (Lemmas 4.2.3 and 4.2.6).

4.2.1 Lemma [Determinism]: If $e \longrightarrow_c e_1$ and $e \longrightarrow_c e_2$ then $e_1 = e_2$.

Proof: By induction on $e \longrightarrow_c e_1$, observing that in each case the same rule must have applied to find $e \longrightarrow_c e_2$. □

4.2.2 Lemma [Canonical forms]: If $\Gamma \vdash v : T$, then:

- $T = \mathsf{Dyn}$ implies that $v = v'_{B!}$ or $v = v'_{\mathbf{Fun}!}$ for some v'.
- $T = B$ implies that $v = k_{\mathsf{Id}}$.
- $T = T_1 \to T_2$ implies that $v = \lambda x{:}T_1.\ e_{\mathsf{Id}}$ or $v = v'_{\langle T'_1 \to T'_2 \Rightarrow T_1 \to T_2 \rangle}$ for some v'.
- $T = \{x{:}T' \mid e\}$ implies that $v = v'_{\{x:T|e\}?}$ for some v'.

Proof: By case analysis on the typing derivation. □

4.2.3 Lemma [Progress]: If $\emptyset \vdash e : T$ then there exists an e' such that $e \longrightarrow_c e'$ or e is a result.

Proof: By induction on the typing derivation.

(T_PreVal) u_{Id} is a result.

(T_TagB) $v_{B!}$ is a result.

(T_TagFun) $v_{\mathbf{Fun}!}$ is a result.

(T_TagRefine) $v_{\{x:T|e\}?}$ is a result.

(T_Wrap) $v_{\langle T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22} \rangle}$ is a result.

(T_Var) Contradictory—variables aren't well typed in the empty context.

(T_Fail) fail is a result.

(T_Cast) By the IH on $\emptyset \vdash e : T_1$, either $e \longrightarrow_c e'$ or e is a result. In the former case, we step by G_CastInner. If e is a result, then it is either fail or a value v. In the former case, we step by G_CastRaise. Otherwise, we go by cases on T_1 and T_2.

($T_1 = \mathsf{Dyn}$) By cases on T_2.

($T_2 = \mathsf{Dyn}$) We step by G_CastId.

($T_2 = B$) By canonical forms (Lemma 4.2.2), v is either ${v'}_{B'!}$ or ${v'}_{\mathbf{Fun!}}$. If $B = B'$, we step by G_CastBB. If not, we step by G_CastBFailB or G_CastBFailFun.

($T_2 = T_{21} \to T_{22}$) By canonical forms (Lemma 4.2.2), $v = {v'}_{B'!}$ or $v = {v'}_{\mathbf{Fun!}}$. We step by G_CastFunFailB or G_CastFunFun.

($T_2 = \{x{:}T \mid e\}$) If $T = \mathsf{Dyn}$, we step by G_CastCheck. If $T = B$ (the only other option), we step by G_CastPreCheck.

($T_1 = B$) By cases on T_2.

($T_2 = \mathsf{Dyn}$) We step by G_CastB.

($T_2 = B'$) By inversion of $\vdash B \parallel B'$, we have $B = B'$. We step by G_CastId.

($T_2 = T_{21} \to T_{22}$) Contradictory, since it is not the case that $\vdash B \parallel T_{21} \to T_{22}$.

($T_2 = \{x{:}T \mid e\}$) If $T = B'$ (and so $B' = B$), we step by G_CastCheck. Otherwise we step by G_CastPreCheck.

($T_1 = T_{11} \to T_{12}$) By cases on T_2.

($T_2 = \mathsf{Dyn}$) We step by G_CastFun or G_CastFunDyn.

($T_2 = B$) Contradictory, since it is not the case that $\vdash T_{21} \to T_{22} \parallel B$.

($T_2 = T_{21} \to T_{22}$) We step by G_CastFunWrap.

($T_2 = \{x{:}T \mid e\}$) It must be that $T = \mathsf{Dyn}$, since it is not the case that $\vdash T_{21} \to T_{22} \parallel B$. We step by G_CastPreCheck.

($T_1 = \{x{:}T \mid e\}$) By cases on T_2.

($T_2 = \mathsf{Dyn}$) We step by G_CastPredPred.

($T_2 = B$) We step by G_CastPredPred.

($T_2 = T_{21} \to T_{22}$) We step by G_CastPredPred.

($T_2 = \{x{:}T' \mid e'\}$) We step by G_CastPredPred or G_CastId.

(T_App) We have $\emptyset \vdash e_1\ e_2 : T_2$. By the IH on $\emptyset \vdash e_1 : T_1 \to T_2$, either e_1 steps, or it is a result. In the former case, we go by G_AppL. In the latter, e_1 is either fail (and we step by G_AppRaiseL) or e_1 is a value.

Similarly, by the IH on $\emptyset \vdash e_2 : T_1$, either e_2 steps or is a result. We can apply G_AppR or G_AppRaiseR (using that e_1 is a value), unless e_2 is some value v_2. In that case, we use canonical forms to see that e_1 is either $\lambda x{:}T_1.\ e'_{1\,\mathsf{Id}}$ or $v_{1\langle T'_1 \to T'_2 \Rightarrow T_1 \to T_2 \rangle}$. We step by G_Beta and G_Fun, respectively.

(T_Op) By induction on n, applying the IH to step by either G_OpInner or G_OpFail. If all of the arguments are values, we step by G_Op.

(T_CheckCast) By the IH, we can step the active check by G_CastInner or G_CastRaise. If it's a value, we have $\emptyset \vdash v_2 : \mathsf{Bool}$, so v_2 is either $\mathsf{true}_{\mathsf{Id}}$ or $\mathsf{false}_{\mathsf{Id}}$. We step by G_CheckOK and G_CheckFail, respectively.

□

4.2.4 Lemma [Regularity]:

- If $\Gamma \vdash e : T$ then $\vdash \Gamma$ and $\vdash T$.
- If $\Gamma \vdash u : T$ then $\vdash \Gamma$ and $\vdash T$.

Proof: By mutual induction on the derivations.

(T_Var) $\vdash \Gamma$ by assumption, which gives us $\vdash T$.

(T_Const) $\vdash \Gamma$ By assumption, and we assume that $\vdash \mathsf{ty}(k)$.

(T_Abs) We have $\vdash T_1$ and $\Gamma, x{:}T_1 \vdash e_{12} : T_2$ by assumption. By the IH, $\vdash \Gamma, x{:}T_1$ and $\vdash T_2$. By inversion, $\vdash \Gamma$. We have $\vdash T_1 \to T_2$ by WF_Fun.

(T_PreVal) By the IH.

(T_TagB) By the IH and WF_Dyn.

(T_TagFun) By the IH and WF_Dyn.

(T_TagRefine) By the IH and assumption.

(T_Wrap) By the IH and assumption.

(T_Cast) By the IH and assumption.

(T_Fail) By assumption.

(T_App) By the IH.

(T_Op) By the IH and the assumption that operators have well formed types.

(T_CheckCast) By assumption.

□

4.2.5 Lemma [Substitution]: If $\emptyset \vdash v : T$ then:

- If $\Gamma_1, x{:}T, \Gamma_2 \vdash e : T'$ then $\Gamma_1, \Gamma_2 \vdash e[v/x] : T'$.
- If $\Gamma_1, x{:}T, \Gamma_2 \vdash u : T'$ then $\Gamma_1, \Gamma_2 \vdash u[v/x] : T'$.

Proof: By mutual induction on the typing derivations for terms and pre-values, leaving Γ_2 general.

(T_Const) Immediate by T_Const.

(T_Abs) By T_Abs, using the IH on $\Gamma_1, x{:}T, \Gamma_2, y{:}T_1 \vdash e : T_2$.

(T_Var) If x is the variable in question, then by weakening. If not, then by T_Var.

(T_PreVal) By the IH and T_PreVal.

(T_TagB) By the IH and T_TagB.

(T_TagFun) By the IH and T_TagFun.

(T_TagRefine) Immediate by T_TagRefine, since the terms themselves are actually closed.

(T_Wrap) By the IH and T_Wrap.

(T_Cast) By the IH and T_Cast.

(T_Fail) Immediate by T_Fail.

(T_App) By the IH and T_App.

(T_Op) By the IH and T_Op.

(T_CheckCast) Immedate by T_CheckCast, since the terms themselves are actually closed.

□

4.2.6 Lemma [Preservation]: If $\emptyset \vdash e : T$ and $e \longrightarrow_c e'$, then $\emptyset \vdash e' : T$.

Proof: By induction on the evaluation derivation.

(G_Beta) By inversion, $x{:}T_1 \vdash e_1 : T_2$ and $\emptyset \vdash v_2 : T_1$. By substitution (Lemma 4.2.5).

(G_Fun) By inversion of T_Wrap, we know that $\vdash T_{11} \rightarrow T_{12} \parallel T_{21} \rightarrow T_{22}$. By T_Cast, T_App, and T_Cast; we find the similarities necessary for the T_Cast rules by inversion, since only P_Fun could have applied.

(G_Op) By assumption.

(G_CastId) Immediate.

(G_CastFunFailB) We have $\vdash T_1 \rightarrow T_2$ by inversion; we are done by T_Fail.

(G_CastFunFun) We have $\vdash \mathsf{Dyn} \parallel T_i$ by P_DynL; by P_Fun, T_Cast, and assumption.

(G_CastB) By assumption and T_TagB.

(G_CastFun) By assumption and T_TagFun.

(G_CastBB) By assumption.

(G_CastBFailB) We have $\vdash B$ immediately; by T_Fail.

(G_CastBFailFun) We have $\vdash B$ immediately; by T_Fail.

(G_CastFunDyn) We use P_DynR, P_Fun in two applications of T_Cast.

(G_CastFunWrap) By T_Wrap.

(G_CastPredPred) By assumption and T_Cast, using $\vdash T_1 \parallel T_2$ from the inversion of $\vdash \{x{:}T_1 \mid e\} \parallel T_2$.

(G_CastPreCheck) By assumption and T_Cast. We use $\vdash T_1 \parallel T_2$ from the inversion of $\vdash T_1 \parallel \{x{:}T_2 \mid e\}$ in the first case and P_Id (it can't be a function type!) with P_RefineR in the second.

(G_CastCheck) By T_CheckCast, using $e[v/x] \longrightarrow_c^* e[v/x]$.

(G_CheckOK) By T_TagRefine, using $e[v/x] \longrightarrow_c^* \mathsf{true}_{\mathsf{Id}}$.

(G_CheckFail) By T_Fail, using the assumption that $\vdash \{x{:}T \mid e\}$.

(G_AppL) By T_App and the IH.

(G_AppR) By T_App and the IH.

(G_OpInner) By T_Op and the IH.

(G_CastInner) By T_Cast and the IH.

(G_CheckInner) By T_CheckCast and the IH, extending $e[v/x] \longrightarrow_c^* e_2 \longrightarrow_c e_2'$.

(G_AppRaiseL) By regularity (Lemma 4.2.4) and T_Fail.

(G_AppRaiseR) By regularity (Lemma 4.2.4) and T_Fail.

(G_OpRaise) By regularity (Lemma 4.2.4) and T_Fail.

(G_CastRaise) By inversion, $\vdash T_2$; then, by T_Fail.

(G_CheckRaise) By inversion, $\vdash \{x{:}T \mid e\}$; then, by T_Fail.

□

4.2.7 Theorem [Type soundness]: If $\emptyset \vdash e : T$, then either $e \longrightarrow^* r$ such that $\emptyset \vdash r : T$ or e diverges.

Proof: Using progress (Lemma 4.2.3) and preservation (Lemma 4.2.6). Unsurprisingly, this is not a constructive proof. □

Types and base types

$$
\begin{array}{lcl}
T & ::= & B \mid T_1 \to T_2 \mid \{x{:}B \mid e\} \mid \mathsf{Dyn} \mid \{x{:}\mathsf{Dyn} \mid e\} \\
B & ::= & \mathsf{Bool} \mid \mathsf{Int} \mid \ldots
\end{array}
$$

Coercions, primitive coercions, and type tags

$$
\begin{array}{lcl}
c & ::= & d_1; \ldots; d_n \\
d & ::= & D! \mid D? \mid c_1 \mapsto c_2 \mid \mathsf{Fail} \\
D & ::= & B \mid \mathbf{Fun} \mid \{x{:}B \mid e\} \mid \{x{:}\mathsf{Dyn} \mid e\}
\end{array}
$$

Terms, results, values, and pre-values

$$
\begin{array}{lcl}
e & ::= & x \mid r \mid op(e_1, \ldots, e_n) \mid e_1\ e_2 \mid \langle c \rangle\ e \mid \\
 & & \langle \{x{:}T \mid e_1\}, e_2, v \rangle \\
r & ::= & v \mid \mathsf{fail} \\
v & ::= & u_{\mathsf{Id}} \mid v_{B!} \mid v_{\mathbf{Fun}!} \mid v_{\{x:T \mid e\}?} \mid v_{c_1 \mapsto c_2} \\
u & ::= & k \mid \lambda x{:}T.\ e
\end{array}
$$

Typing contexts

$$
\begin{array}{lcl}
\Gamma & ::= & \emptyset \mid \Gamma, x{:}T
\end{array}
$$

Figure 4.8: NAIVE syntax

4.3 A naïve coercion calculus

In this section, we define a naïve coercion calculus, NAIVE. Its syntax is in Figure 4.8, its typing rules are in Figure 4.9 (both in Section 4.3.1), and its operational semantics are in Figure 4.12. The type soundness proof appears in Section 4.3.3; we show how to translate CAST terms into behaviorally equivalent NAIVE terms in Section 4.4.

This language adheres to a design philosophy of "simple types by default, dynamic and refinement types by coercion". I am the first to articulate a typing philosophy for full-spectrum languages. My design philosophy, elaborated more fully in Section 4.1, has three principles. First, base values have simple types; e.g., all integers are typed at Int. Second, we give operations types precise enough to guarantee totality; e.g., division has a type at least as precise as $\mathsf{Int} \to \{x{:}\mathsf{Int} \mid x \neq 0\} \to \mathsf{Int}$. I understand refinement types as being designed for protecting partial operations (the original name is due to a method for protecting partial pattern matches [30]); giving operations types that make them total means that any reasoning about runtime errors can entirely revolve around cast (here, coercion) failures. And third, values satisfy their refinement types; e.g., if $\emptyset \vdash v : \{x{:}T \mid e\}$, then $e[v/x] \longrightarrow^* \mathsf{true}$. Every sound refinement type calculus has this, but ours will have it as an inversion of the typing rule. Since we are defining call-by-value (CBV) languages, this means that functions can assume their inputs actually satisfy their refinement types.

4.3.1 Syntax and typing

Most of the terms here are standard parts of the lambda calculus. The most pertinent extension here is the coercion term, $\langle c \rangle\ e$; I describe my language of coercions in greater detail below. Evaluation returns *results*: either a value or a failure fail. The term fail represents coercion failure. Coercion failures can occur when the predicate fails—i.e., $e_1[v/x] \longrightarrow_n^* \mathsf{false}_{\mathsf{ld}}$ (see F_CheckFail)—or when dynamically typed values don't match their type—e.g., $\langle \mathsf{Bool?} \rangle\ 5_{\mathsf{Int!}}$ (see F_TagFail). I treat fail as an uncatchable exception. Just like for Cast, values in Naive are split in two parts: *pre-values* u are the typical values of other languages: constants and lambdas; *values* v are pre-values with a stack of primitive coercions. (See below for an explanation of the different kinds of coercions.) Technically, a value is either a pre-value tagged with the identity coercion, u_{ld}, or an inner value *tagged* with an extra coercion, v_d. That is, in this language every value has a list of coercions. Values are introduced in source programs with the identity coercion, u_{ld}. Keeping a coercion on *every* value is a slight departure from prior formulations. Doing so is technically expedient—simplifying the structure of the language and clearly differentiating terms with pending coercions and values with tags.

The terms of the calculus are otherwise fairly unremarkable: we have variables, application, and a fixed set of built-in operations. We have two additional runtime terms. The *active check* $\langle \{x{:}T \mid e_1\}, e_2, v \rangle$ represents an ongoing check that the value v satisfies the predicate e_1; it is invariant that $e_1[v/x] \longrightarrow_n^* e_2$. The second term, fail, represents the uncatchable exception thrown when a check fails. Adding exception handling facilities would completely destroy any obvious formal equivalence between Naive and Efficient, though I conjecture that an adequately instrumented semantics would be able to determine the precise point at which a Naive program diverged from its Efficient translation.

The calculus has: simple types, where B is a base type and $T_1 \to T_2$ is the standard function type; the dynamic type Dyn; and refinements of both base types and type dynamic. The base refinement $\{x{:}B \mid e\}$ includes all constants k of type B such that $e[k_{\mathsf{ld}}/x] \longrightarrow_n^* \mathsf{true}_{\mathsf{ld}}$. Similarly, the dynamic refinement $\{x{:}\mathsf{Dyn} \mid e\}$ includes all *values* v such that v has type Dyn and $e[v/x] \longrightarrow_n^* \mathsf{true}_{\mathsf{ld}}$. When defining inference rules in this fashion (e.g., T_TagValRefine), I treat such a rule as a rule schema that expands into two separate rules. I find it useful to think of three different "domains" of types: the base domain with the types B and $\{x{:}B \mid e\}$; the functional domain with the types of the form $T_1 \to T_2$, with special attention paid to functions on dynamic values, of type $\mathsf{Dyn} \to \mathsf{Dyn}$; and the dynamic domain with the types Dyn and $\{x{:}\mathsf{Dyn} \mid e\}$.

Well formedness judgments of types, and contexts are defined in Figure 4.9. It is worth noting, however, that well formedness of refinements refers back to the term typing judgment. Coercion typing is in defined in Figure 4.10.

Term typing (also defined in Figure 4.9) is mostly standard. Readers should find the rules for constants (T_Const), variables (T_Var), functions (T_Abs), failure

Well formed contexts and types $\boxed{\vdash \Gamma}$ $\boxed{\vdash T}$

$$\frac{}{\vdash \emptyset}\ \text{WF\_Empty} \qquad \frac{\vdash \Gamma \quad \vdash T}{\vdash \Gamma, x{:}T}\ \text{WF\_Extend}$$

$$\frac{}{\vdash B}\ \text{WF\_Base} \qquad \frac{}{\vdash \mathsf{Dyn}}\ \text{WF\_Dyn} \qquad \frac{\vdash T_1 \quad \vdash T_2}{\vdash T_1 \to T_2}\ \text{WF\_Fun}$$

$$\frac{x{:}T \vdash e : \mathsf{Bool} \quad T = B \text{ or } T = \mathsf{Dyn}}{\vdash \{x{:}T \mid e\}}\ \text{WF\_Refine}$$

Well typed terms and values $\boxed{\Gamma \vdash u : T}$ $\boxed{\Gamma \vdash e : T}$

$$\frac{\vdash \Gamma}{\Gamma \vdash k : \mathsf{ty}(k)}\ \text{T\_Const} \qquad \frac{\vdash T_1 \quad \Gamma, x{:}T_1 \vdash e_{12} : T_2}{\Gamma \vdash \lambda x{:}T_1.\ e_{12} : T_1 \to T_2}\ \text{T\_Abs}$$

$$\frac{\Gamma \vdash u : T}{\Gamma \vdash u_{\mathsf{Id}} : T}\ \text{T\_PreVal} \qquad \frac{\begin{array}{c} d \neq \{x{:}T \mid e\}? \quad d \neq \mathsf{Fail} \\ \Gamma \vdash v : T_1 \quad \vdash d : T_1 \rightsquigarrow T_2 \end{array}}{\Gamma \vdash v_d : T_2}\ \text{T\_TagVal}$$

$$\frac{\vdash \Gamma \quad \emptyset \vdash v : T \quad \vdash \{x{:}T \mid e\} \quad e[v/x] \longrightarrow_n^* \mathsf{true}_{\mathsf{Id}}}{\Gamma \vdash v_{\{x:T \mid e\}?} : T}\ \text{T\_TagValRefine}$$

$$\frac{\vdash \Gamma \quad x{:}T \in \Gamma}{\Gamma \vdash x : T}\ \text{T\_Var} \qquad \frac{\vdash T \quad \vdash \Gamma}{\Gamma \vdash \mathsf{fail} : T}\ \text{T\_Fail} \qquad \frac{\vdash c : T_1 \rightsquigarrow T_2 \quad \Gamma \vdash e : T_1}{\Gamma \vdash \langle c \rangle\ e : T_2}\ \text{T\_Coerce}$$

$$\frac{\mathsf{ty}(op) = T_1 \to \ldots \to T_n \to T \quad \Gamma \vdash e_i : T_i}{\Gamma \vdash op(e_1, \ldots, e_n) : T}\ \text{T\_Op}$$

$$\frac{\Gamma \vdash e_1 : (T_1 \to T_2) \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}\ \text{T\_App}$$

$$\frac{\vdash \Gamma \quad \vdash \{x{:}T \mid e_1\} \quad \emptyset \vdash v : T \quad \emptyset \vdash e_2 : \mathsf{Bool} \quad e_1[v/x] \longrightarrow_n^* e_2}{\Gamma \vdash \langle \{x{:}T \mid e_1\}, e_2, v \rangle : \{x{:}T \mid e_1\}}\ \text{T\_CheckNaive}$$

Figure 4.9: Typing for Naive

Well typed coercions $\boxed{\vdash c : T_1 \rightsquigarrow T_2}$ $\boxed{\vdash d : T_1 \rightsquigarrow T_2}$

$$\frac{\vdash T}{\vdash \mathsf{Id} : T \rightsquigarrow T}\ \text{C\_Id} \qquad \frac{\vdash d_1 : T_1 \rightsquigarrow T' \quad \vdash \ldots; d_n : T' \rightsquigarrow T_2}{\vdash d_1; \ldots; d_n : T_1 \rightsquigarrow T_2}\ \text{C\_Compose}$$

$$\frac{\vdash T_1 \quad \vdash T_2}{\vdash \mathsf{Fail} : T_1 \rightsquigarrow T_2}\ \text{C\_Fail} \qquad \frac{\vdash c_1 : T_{21} \rightsquigarrow T_{11} \quad \vdash c_2 : T_{12} \rightsquigarrow T_{22}}{\vdash c_1 \mapsto c_2 : (T_{11} \to T_{12}) \rightsquigarrow (T_{21} \to T_{22})}\ \text{C\_Fun}$$

$$\frac{\vdash \mathbf{src}(D)}{\vdash D! : \mathbf{src}(D) \rightsquigarrow \mathbf{tgt}(D)}\ \text{C\_Tag} \qquad \frac{\vdash \mathbf{src}(D)}{\vdash D? : \mathbf{tgt}(D) \rightsquigarrow \mathbf{src}(D)}\ \text{C\_Untag}$$

Source and target types $\boxed{\mathbf{src}(D) : T}$ $\boxed{\mathbf{tgt}(D) : T}$

$$\begin{array}{rclrcl} \mathbf{src}(B) & = & B & \mathbf{tgt}(B) & = & \mathsf{Dyn} \\ \mathbf{src}(\mathbf{Fun}) & = & \mathsf{Dyn} \to \mathsf{Dyn} & \mathbf{tgt}(\mathbf{Fun}) & = & \mathsf{Dyn} \\ \mathbf{src}(\{x{:}T \mid e\}) & = & \{x{:}T \mid e\} & \mathbf{tgt}(\{x{:}T \mid e\}) & = & T \end{array}$$

Figure 4.10: Coercion typing

(T_Fail), application (T_App), and built-in operations (T_Op) familiar.

Pre-values are typed by T_Const and T_Abs; pre-values tagged with the Id coercion are typed as values by T_PreVal. T_TagVal types values that are tagged with anything but a refinement type, for which we use a separate rule. We want all values at a refined type to satisfy their refinement—a key property and part of my philosophy of refinement types. The T_TagValRefine rule ensures that values typed at a refinement type actually satisfy their refinement. In the metatheory, the typing rule for active check forms, T_CheckNaive, holds onto a trace of the evaluation of the predicate. If the check succeeds, the trace can then be put directly into a T_TagValRefine derivation. Naturally, none of these rules with premises concerning evaluation (T_TagValRefine, T_CheckNaive) are necessary for source programs—they are technicalities for the proofs of preservation (Lemma 4.3.8 below) and equivalence (Section 4.6).

The coercions are the essence of this calculus: they represent the step-by-step checks that are done to move values between dynamic, simple, and refinement types. The syntax of coercions in Figure 4.8 splits coercions into three parts: composite coercions c, primitive coercions d, and tags D. The typing rules for coercions are written in Figure 4.9; the types of primitive coercions are shown graphically in Figure 4.11. When it is clear from context whether I mean a composite or a primitive coercion, I will simply call either a "coercion". A composite coercion c is a list of primitive coercions. I write the empty coercion—the composite coercion comprising zero primitive coercions—as Id. When I write $c; d$ or $d; c$ in a rule and it matches against a coercion

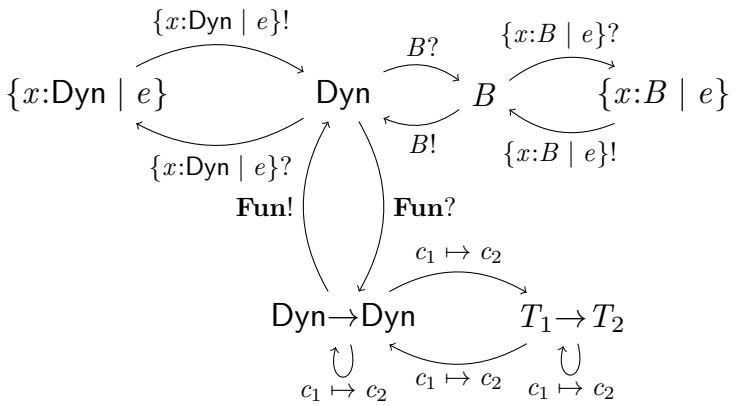


Figure 4.11: Primitive coercions

with a single primitive coercion—that is, when we match $c; B!$ against $B!$—we let $c = \mathsf{Id}$. This is a slight departure from earlier coercion systems; this construction avoids messing around too much with re-association of coercion composition. I compare my coercions to other formulations in related work (Chapter 5). There are four kinds of primitive coercions: failures Fail, *tag* coercions $D!$, *checking* coercions $D?$, and *functional* coercions $c_1 \mapsto c_2$. (Note that c_1 and c_2 are composite.) Finally, the tags D are a flattening of the type space: each base type B has a corresponding tag (which I also write B); functions have a single tag **Fun**. Intuitively, these are the type tags that are commonly used in dynamically typed languages. We also have refinement tags for both types of refinement, which I write the same as the corresponding types: $\{x{:}B \mid e\}$ for refinements of base types and $\{x{:}\mathsf{Dyn} \mid e\}$ for refinements of type dynamic.

Failure coercions are present only for showing the equivalence with the space-efficient calculus; rule F_Fail gives a semantics for Fail. (I discuss the operational semantics more fully below, in Section 4.3.2.) It is worthwhile to contrast my treatment of failure with that of Herman et al. [39]. Whereas Henglein treats mismatched tag/untag operations, such as $B!; \mathbf{Fun}?$, as stuck, I follow Herman et al. [39] in having an explicit failure coercion, Fail, which leads to an uncatchable program failure, fail. The F_Fail rule causes the program to fail when a failure coercion appears. (I carefully keep Fail out of the tags placed on pre-values and values.) In fact, F_Fail will never apply when evaluating sensible source programs—no sane program will start with Fail in it, and no Naive evaluation rule generates Fail. Instead, failures arise in the Naive when the other rules with Fail in their name fire. We include F_Fail as a technicality for the soundness proof (Theorem 4.6.10) in Section 4.6. In Herman et al.'s calculus, $\langle \mathsf{Fail} \rangle\ v$ is a value—the program won't actually fail until this value reaches an elimination form. While systems with lazy error detection have been proposed [43], here $\langle \mathsf{Fail} \rangle\ e$ raises a program-terminating exception immediately, before stepping e further. Eager error detection is more in line with standard error behavior, particularly other calculi where failed casts result in

blame [26, 28, 41, 67, 16, 36, 74, 78, 65, 34, 68, 8, 4, 22].

The *tagging* coercions $D!$ and *checking* coercions $D?$ fall into two groups: those which move values into type dynamic and those which deal with refinements (of either base types or type dynamic). They are typed by C_Tag and C_Untag; note that $\vdash \mathbf{tgt}(D)$ for all D, but we need the $\vdash \mathbf{src}(D)$ premise for $D = \{x{:}T \mid e\}$ in particular, to ensure that the predicate is well typed.

The tags B and **Fun** are used to move values to and from the dynamic type Dyn. The tagging coercions $B!$ and **Fun**! mark a base value (typed B) or functional value (typed $\mathsf{Dyn}{\to}\mathsf{Dyn}$) as having the dynamic type Dyn. The checking coercions $B?$ and **Fun**? are the corresponding *untagging* coercions, taking a dynamic value and checking its tag. If the tags match, the original typed value is returned: $\langle \mathsf{Bool}? \rangle\ \mathsf{true}_{B!} \longrightarrow_n \mathsf{true}_{\mathsf{Id}}$. If the tags don't match, the program fails: $\langle \mathsf{Bool}? \rangle\ 5_{\mathsf{Int}!} \longrightarrow_n^* \mathsf{fail}$.

The tags $\{x{:}B \mid e\}$ and $\{x{:}\mathsf{Dyn} \mid e\}$ are used for refinements. The checking coercion $\{x{:}T \mid e\}?$ checks that a value v satisfies the predicate e, i.e., that $e[v/x] \longrightarrow_n \mathsf{true}_{\mathsf{Id}}$; see F_CheckOK and F_CheckFail below. The coercion $\{x{:}T \mid e\}!$ is correspondingly used to 'forget' refinement checks.

Note that in both the dynamic and the refinement cases, the checking coercions are the ones that might fail. Given my philosophy of values starting out simply typed, the two types of coercions differ in that tagging coercions are applied first when moving from simple types to dynamic typing, but checking coercions are applied first when moving to refinements. Any simply typed value is just fine as an appropriately tagged dynamic value, but a simply typed value must be checked to see if it satisfies a refinement.

Finally, while the **Fun**! and **Fun**? coercions injecting and project functions on $\mathsf{Dyn}{\to}\mathsf{Dyn}$ into type Dyn, there is a separate structural coercion that works on typed functions: $c_1 \mapsto c_2$, typed by the rule C_Fun. Note that the C_Fun rule is contravariant; see the F_Fun rule below.

Only certain so-called *value coercions* can appear on values: Id appears on *all* values; $B!$ and **Fun**! tag values into the dynamic type; $\{x{:}T \mid e\}?$ marks successful refinement checks; and $c_1 \mapsto c_2$ wraps a value with pending checks, creating a function proxy. I revisit the notion of value coercions below, in the space efficient calculus of Section 4.5.

4.3.2 Operational semantics

The rules are adapted from the evaluation contexts used in Herman et al. [39]. The core rules in Figure 4.12 are standard CBV rules (F_Beta and F_Op), as are most of the congruence and exception raising rules (F_AppL, F_AppR, F_OpInner, F_AppRaiseL, F_AppRaiseR, F_OpRaise). F_Fun and F_Merge manage function proxies and pending coercions on the stack; the tag management rules appear in Figure 4.13. Before discussing the coercion evaluation rules, it is worth taking a moment to talk about the denotation of operators. In particular, $[\![op]\!]\,(v_1, \ldots, v_n)$ must (a) be total when applied to correctly typed values and (b) ignore the tags on its

$\boxed{e_1 \longrightarrow_n e_2}$

$$\frac{}{(\lambda x{:}T.\ e_{12})_{\mathsf{Id}}\ v_2 \longrightarrow_n e_{12}[v_2/x]}\ \text{F\_Beta} \qquad \frac{}{v_{1\,(c_1 \mapsto c_2)}\ v_2 \longrightarrow_n \langle c_2 \rangle\ (v_1\ (\langle c_1 \rangle\ v_2))}\ \text{F\_Fun}$$

$$\frac{}{op(v_1, \ldots, v_n) \longrightarrow_n [\![op]\!]\,(v_1, \ldots, v_n)}\ \text{F\_Op}$$

$$\frac{e \neq \langle c' \rangle\ e'}{\langle \mathsf{Fail}; c \rangle\ e \longrightarrow_n \mathsf{fail}}\ \text{F\_Fail} \qquad \frac{e_1 \longrightarrow_n e_1'}{e_1\ e_2 \longrightarrow_n e_1'\ e_2}\ \text{F\_AppL} \qquad \frac{e_2 \longrightarrow_n e_2'}{v_1\ e_2 \longrightarrow_n v_1\ e_2'}\ \text{F\_AppR}$$

$$\frac{e_i \longrightarrow_n e_i'}{op(v_1, \ldots, v_{i-1}, e_i, \ldots, e_n) \longrightarrow_n op(v_1, \ldots, v_{i-1}, e_i', \ldots, e_n)}\ \text{F\_OpInner}$$

$$\frac{e \neq \langle c' \rangle\ e'' \quad e \longrightarrow_n e'}{\langle c \rangle\ e \longrightarrow_n \langle c \rangle\ e'}\ \text{F\_CoerceInner}$$

$$\frac{}{\langle c_1 \rangle\ (\langle c_2 \rangle\ e) \longrightarrow_n \langle c_2; c_1 \rangle\ e}\ \text{F\_Merge}$$

$$\frac{e_2 \longrightarrow_n e_2'}{\langle \{x{:}T \mid e_1\}, e_2, v \rangle \longrightarrow_n \langle \{x{:}T \mid e_1\}, e_2', v \rangle}\ \text{F\_CheckInner}$$

$$\frac{}{\langle c \rangle\ \mathsf{fail} \longrightarrow_n \mathsf{fail}}\ \text{F\_CoerceRaise} \qquad \frac{}{op(v_1, \ldots, v_{i-1}, \mathsf{fail}, \ldots, e_n) \longrightarrow_n \mathsf{fail}}\ \text{F\_OpRaise}$$

$$\frac{}{\mathsf{fail}\ e_2 \longrightarrow_n \mathsf{fail}}\ \text{F\_AppRaiseL} \qquad \frac{}{v_1\ \mathsf{fail} \longrightarrow_n \mathsf{fail}}\ \text{F\_AppRaiseR}$$

$$\frac{}{\langle \{x{:}T \mid e\}, \mathsf{fail}, v \rangle \longrightarrow_n \mathsf{fail}}\ \text{F\_CheckRaise}$$

Figure 4.12: Naive operational semantics (core rules)

$$\frac{}{\langle \mathsf{Id} \rangle\ v \longrightarrow_n v}\ \text{F\_TagId}$$

$$\frac{}{\langle \{x{:}T \mid e\}?; c \rangle\ v \longrightarrow_n \langle c \rangle\ \langle \{x{:}T \mid e\}, e[v/x], v \rangle}\ \text{F\_Check}$$

$$\frac{}{\langle \{x{:}T \mid e\}, \mathsf{true}_{\mathsf{Id}}, v \rangle \longrightarrow_n v_{\{x:T|e\}?}}\ \text{F\_CheckOK}$$

$$\frac{}{\langle \{x{:}T \mid e\}, \mathsf{false}_{\mathsf{Id}}, v \rangle \longrightarrow_n \mathsf{fail}}\ \text{F\_CheckFail}$$

$$\frac{}{\langle B!; c \rangle\ v \longrightarrow_n \langle c \rangle\ v_{B!}}\ \text{F\_TagB} \qquad \frac{}{\langle \mathbf{Fun}!; c \rangle\ v \longrightarrow_n \langle c \rangle\ v_{\mathbf{Fun}!}}\ \text{F\_TagFun}$$

$$\frac{}{\langle B?; c \rangle\ v_{B!} \longrightarrow_n \langle c \rangle\ v}\ \text{F\_TagBB} \qquad \frac{}{\langle \mathbf{Fun}?; c \rangle\ v_{\mathbf{Fun}!} \longrightarrow_n \langle c \rangle\ v}\ \text{F\_TagFunFun}$$

$$\frac{}{\langle \mathbf{Fun}?; c \rangle\ v_{B!} \longrightarrow_n \mathsf{fail}}\ \text{F\_TagFunFailB}$$

$$\frac{B \neq B'}{\langle B?; c \rangle\ v_{B'!} \longrightarrow_n \mathsf{fail}}\ \text{F\_TagBFailB} \qquad \frac{}{\langle B?; c \rangle\ v_{\mathbf{Fun}!} \longrightarrow_n \mathsf{fail}}\ \text{F\_TagBFailFun}$$

$$\frac{}{\langle (c_1 \mapsto c_2); c \rangle\ v \longrightarrow_n \langle c \rangle\ v_{c_1 \mapsto c_2}}\ \text{F\_TagFunWrap}$$

$$\frac{}{\langle \{x{:}T \mid e\}!; c \rangle\ v_{\{x:T|e\}?} \longrightarrow_n \langle c \rangle\ v}\ \text{F\_TagPredPred}$$

Figure 4.13: Naive operational semantics (coercion rules)

inputs. This disallows some potentially useful operators—e.g., testing or projecting the tag from a dynamic value—but greatly simplifies the technicalities relating the two calculi in Section 4.6. I don't believe that adding such tag-dependent operators would break anything in a deep way, but I omit them for simplicity's sake. While on the subject of tags, I want to stress that the dynamic type tags are *not* erasable, since the evaluation rules depend on them. I conjecture that the refinement tags $\{x{:}T \mid e\}?$ are in fact erasable, but have not proven so.

Most of the rules for coercions take a term of the form $\langle d; c\rangle\ v$ and somehow apply the primitive coercion d to v. The rest cover more structural uses of coercions. I cover these structural rules first and then explain the "tagging" rules. F_TagId (in Figure 4.13) applies when we have used up all of the primitive coercions, in which case we simply drop the coercion form.

The F_Merge and F_CoerceInner rules coordinate coercion merging and congruence. The F_Merge rule simply concatenates two adjacent coercions. This concatenation isn't space efficient—in the space-efficient calculus, we normalize the concatenation to a canonical coercion of bounded size. F_CoerceInner steps congruently inside a coerced term—we are careful to ensure that it can only apply after F_Merge has fired. Carefully staging F_CoerceInner after F_Merge helps maintain determinism(Lemma 4.3.4)—if we didn't force F_Merge to apply first, the number of coercions might grow out of control.[3] Note that in F_Merge and F_Fail, the innermost term need not be a value. If I formulated the semantics as an abstract machine, we could have an explicit stack of coercions; instead, we combine them as they collide in the term.

The remaining coercion rules have Tag in their name and work on a term $\langle d; c\rangle\ v$ by combining d and v. F_TagB and F_TagFun tag base values and functions (of type $\mathsf{Dyn}{\to}\mathsf{Dyn}$) into type dynamic, using the tagging coercions $B!$ and $\mathbf{Fun}!$, respectively. F_TagBB and F_TagFunFun apply $B?$ and $\mathbf{Fun}?$ to values that have matching $B!$ and $\mathbf{Fun}!$ tags; the effect is to simply strip the tag off the value. The F_TagFunFailB, F_TagBFailFun, and F_TagBFailB rules cause the program to fail when it tries to strip a tag off with a checking coercion that doesn't match.

F_Check starts the *active check* for a refinement check. An active check $\langle\{x{:}T \mid e_1\}, e_2, v\rangle$ is a special kind of condition: if $e_2 \longrightarrow_n^* \mathsf{true}_{\mathsf{Id}}$, then it returns $v_{\{x:T|e_1\}}?$ (rule F_CheckOK); if, on the other hand, $e_2 \longrightarrow_n^* \mathsf{false}_{\mathsf{Id}}$, then the active check returns fail (rule F_CheckFail). Note that the typing rules for active checks make sure that $e_1[v/x] \longrightarrow_n^* e_2$, i.e., that the active check is actually checking whether or not the value satisfies the predicate. The F_TagPredPred rule is similar the untagging rules F_TagBB and F_TagFunFun, though there is no chance of failure here. Having $\{x{:}T \mid e\}!$ eliminate the tag $\{x{:}T \mid e\}?$ is reminiscent of the coercion normalization rule given in the introduction—which we said occasionally skips checks. But here in the Naive, F_TagPredPred only applies when removing a tag from a

[3] Siek and Wadler [68] pointed out that Herman et al.'s evaluation contexts introduce nondeterminism; explicit congruence rules avoid this problem.

value, i.e., when the check has already been done. In the space-efficient semantics in Section 4.5, coercion normalization will actually skip checks.

It is worth emphasizing here: F_TagPredPred is not an optimization. It is simply the natural translation of G_CastPredPred into coercions:

$$\langle \{x{:}T_1 \mid e\} \Rightarrow T_2 \rangle \; v_{\{x:T_1|e\}?} \longrightarrow_c \langle T_1 \Rightarrow T_2 \rangle \; v$$

When a value is tagged as having been checked but must be cast out of the refinement type, we simply drop the tag on the floor. Rephrasing this in coercions, we find:

$$\langle \{x{:}T \mid e\}!; c \rangle \; v_{\{x:T|e\}?} \longrightarrow_n \langle c \rangle \; v$$

Without a rule like this, there would never be a way to untag a value;[4] suppose $x{:}\{x{:}\mathsf{Int} \mid x \neq 0_{\mathsf{Id}}\}$. We need coercions c_1 and c_2 such that

$$(\lambda x{:}\{x{:}\mathsf{Int} \mid x \neq 0_{\mathsf{Id}}\}.\ \mathsf{if}\ \ldots\ \mathsf{then}\ \mathsf{div}(5_{\mathsf{Id}}, x)\ \mathsf{else}\ \mathsf{plus}(1_{\mathsf{Id}}, \langle c_1 \rangle\ x))_{\mathsf{Id}}\ (\langle c_2 \rangle\ \ldots)$$

is well typed. Whether we write them with question marks or exclamation points is irrelevant: we need *some* coercions such that c_2 and c_1 cancel each other out. I choose to have ? represent checking operations that may fail (testing a dynamic type tag, adding a refinement tag) and ! to represent operations that won't fail (adding a dynamic type tag, dropping a refinement tag).

The F_TagFunWrap rule wraps a value in a functional coercion. The F_Fun rule unwinds applications of wrapped values, coercing the wrapped function's argument and result.

Readers particularly familiar with contracts will recognize that these coercions are a lower level account of the steps taken in running casts (see Belo et al. [8] for an account); alternatively, these coercions are a lower level formulation of the projections underlying contracts [25]. See Greenberg et al. [34] for a comparison.

In Figure 4.14, I translate the cast example from the introduction (Figure 4.2). It is easy to see that this calculus isn't space efficient, either:, coercions can consume an unbounded amount of space. As the function evaluates, a stack of coercions builds up—here, proportional to the size of the input. Again, I highlight redexes; note that the whole term is highlighted when the outermost coercions merge. The casts of the earlier example match the coercions here, e.g. the cast $\langle \mathsf{Dyn} \Rightarrow \mathsf{Bool} \rangle\ e$ is just like the coercion $\langle \mathsf{Bool}? \rangle\ e$.

4.3.3 Proofs

Naive is type sound, i.e, it enjoys progress and preservation.

4.3.1 Lemma: For all D, $\vdash \mathbf{tgt}(D)$.

[4] Short of accumulating refinements of values, which is obviously not space efficient. We could leave tags on, adding a rule like T_Forget in Chapter 3, at the expense of implicitly dropping a tag in other rules. That wins nothing: we still have to drop the tag, and the rules are more confusing.

$$
\begin{array}{ll}
 & \mathsf{odd}\ 3_{\mathsf{Id}} \\
\longrightarrow_n & \mathsf{even}_{\mathsf{Int!}\mapsto\mathsf{Bool?}}\ 2_{\mathsf{Id}} \\
\longrightarrow_n & \langle \mathsf{Bool?} \rangle\ (\mathsf{even}\ (\langle \mathsf{Int!} \rangle\ 2_{\mathsf{Id}})) \\
\longrightarrow_n & \langle \mathsf{Bool?} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathsf{Id}})_{\mathsf{Int?}\mapsto\mathsf{Bool!}}\ (2_{\mathsf{Id}})_{\mathsf{Int!}}) \\
\longrightarrow_n & \langle \mathsf{Bool?} \rangle\ (\langle \mathsf{Bool!} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathsf{Id}}\ (\langle \mathsf{Int?} \rangle\ (2_{\mathsf{Id}})_{\mathsf{Int!}}))) \\
\longrightarrow_n & \langle \mathsf{Bool!}; \mathsf{Bool?} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathsf{Id}}\ (\langle \mathsf{Int?} \rangle\ (2_{\mathsf{Id}})_{\mathsf{Int!}})) \\
\longrightarrow_n & \langle \mathsf{Bool!}; \mathsf{Bool?} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathsf{Id}}\ 2_{\mathsf{Id}}) \\
\longrightarrow_n & \langle \mathsf{Bool!}; \mathsf{Bool?} \rangle\ (\mathsf{odd}\ 1_{\mathsf{Id}}) \\
\longrightarrow_n & \langle \mathsf{Bool!}; \mathsf{Bool?} \rangle\ (\mathsf{even}_{\mathsf{Int!}\mapsto\mathsf{Bool?}}\ 0_{\mathsf{Id}}) \\
\longrightarrow_n & \langle \mathsf{Bool!}; \mathsf{Bool?} \rangle\ (\langle \mathsf{Bool?} \rangle\ (\mathsf{even}\ (\langle \mathsf{Int!} \rangle\ 0_{\mathsf{Id}}))) \\
\longrightarrow_n & \langle \mathsf{Bool?}; \mathsf{Bool!}; \mathsf{Bool?} \rangle\ (\mathsf{even}\ (\langle \mathsf{Int!} \rangle\ 0_{\mathsf{Id}})) \\
\longrightarrow_n & \langle \mathsf{Bool?}; \mathsf{Bool!}; \mathsf{Bool?} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathsf{Id}})_{\mathsf{Int?}\mapsto\mathsf{Bool!}}\ (0_{\mathsf{Id}})_{\mathsf{Int!}}) \\
\longrightarrow_n & \langle \mathsf{Bool?}; \mathsf{Bool!}; \mathsf{Bool?} \rangle\ (\langle \mathsf{Bool!} \rangle \\
 & \quad ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathsf{Id}}\ (\langle \mathsf{Int?} \rangle\ (0_{\mathsf{Id}})_{\mathsf{Int!}}))) \\
\longrightarrow_n & \langle \mathsf{Bool!}; \mathsf{Bool?}; \mathsf{Bool!}; \mathsf{Bool?} \rangle \\
 & \quad ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathsf{Id}}\ (\langle \mathsf{Int?} \rangle\ (0_{\mathsf{Id}})_{\mathsf{Int!}})) \\
\longrightarrow_n & \langle \mathsf{Bool!}; \mathsf{Bool?}; \mathsf{Bool!}; \mathsf{Bool?} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathsf{Id}}\ 0_{\mathsf{Id}}) \\
\longrightarrow_n & \langle \mathsf{Bool!}; \mathsf{Bool?}; \mathsf{Bool!}; \mathsf{Bool?} \rangle\ \mathsf{true}_{\mathsf{Id}} \\
\longrightarrow_n & \langle \mathsf{Bool?}; \mathsf{Bool!}; \mathsf{Bool?} \rangle\ (\mathsf{true}_{\mathsf{Id}})_{\mathsf{Bool!}} \\
\longrightarrow_n & \langle \mathsf{Bool!}; \mathsf{Bool?} \rangle\ \mathsf{true}_{\mathsf{Id}} \\
\longrightarrow_n & \langle \mathsf{Bool?} \rangle\ (\mathsf{true}_{\mathsf{Id}})_{\mathsf{Bool!}} \\
\longrightarrow_n & \langle \mathsf{Id} \rangle\ \mathsf{true}_{\mathsf{Id}} \\
\longrightarrow_n & \mathsf{true}_{\mathsf{Id}}
\end{array}
$$

Figure 4.14: NAIVE reduction

Proof: By cases on D, we use either WF_Dyn or WF_Base. □

4.3.2 Lemma [Regularity of coercion typing]:

- If $\vdash c : T_1 \rightsquigarrow T_2$ then $\vdash T_1$ and $\vdash T_2$.
- If $\vdash d : T_1 \rightsquigarrow T_2$ then $\vdash T_1$ and $\vdash T_2$.

Proof: By mutual induction on the typing derivations.

(C_Id) By inversion.

(C_Fail) By inversion.

(C_Compose) By the IH.

(C_Untag) By assumption on the left; by Lemma 4.3.1 on the right.

(C_Tag) By Lemma 4.3.1 on the left; by assumption on the right.

(C_Fun) By the IH and WF_Fun.

□

4.3.3 Lemma [Regularity]: If $\Gamma \vdash e : T$ then $\vdash T$, and if $\Gamma \vdash u : T$ then $\vdash T$.

Proof: By induction on the typing derivation.

(T_Const) By assumption.

(T_Abs) By the assumption, the IH, and WF_Fun.

(T_PreVal) By the IH.

(T_TagVal) By regularity of coercion typing (Lemma 4.3.2).

(T_TagValRefine) By assumption.

(T_Var) By induction on $\vdash \Gamma$.

(T_Fail) By assumption.

(T_Coerce) By regularity of coercion typing (Lemma 4.3.2).

(T_Op) By assumption on operation typing.

(T_App) By the IH

(T_CheckNaive) By assumption.

□

4.3.4 Lemma [Determinism]: If $e \longrightarrow_n e_1$ and $e \longrightarrow_n e_2$ then $e_1 = e_2$.

Proof: By induction on $e \longrightarrow_n e_1$, observing that in each case the same rule must have applied to find $e \longrightarrow_n e_2$. □

4.3.5 Lemma [Canonical forms]: If $\emptyset \vdash v : T$, then:

- If $T = \mathsf{Bool}$, then v is either $\mathsf{true}_{\mathsf{Id}}$ or $\mathsf{false}_{\mathsf{Id}}$.
- If $T = T_1 \to T_2$, then v is either $\lambda x{:}T_1.\ e_{12\mathsf{Id}}$ or ${v'}_{c_1 \mapsto c_2}$.
- If $T = \mathsf{Dyn}$, then v is either ${v'}_{B!}$ or ${v'}_{\mathbf{Fun}!}$.
- If $T = \{x{:}T' \mid e\}$, then v is ${v'}_{\{x:T'|e\}?}$.

Proof: By induction on the typing derivation $\emptyset \vdash v : T$.

Proof:

(T_PreVal) Constants have base types, so we must only consider the case where $T = \mathsf{Bool}$; if $\mathsf{ty}(k) = \mathsf{Bool}$ then k is either true or false, and we are done.

(T_TagVal) $\emptyset \vdash {v'}_d : T_2$ and $\vdash d : T_1 \rightsquigarrow T_2$. It must be that d is one of $B!$ or $\mathbf{Fun}!$ or $c_1 \mapsto c_2$. In the first two cases, we fulfill the $T_2 = \mathsf{Dyn}$ case. In the latter case, we fulfill the arrow case.

(T_TagValRefine) $\emptyset \vdash {v'}_{\{x:T|e\}?} : \{x{:}T \mid e\}$, which fulfills the refinement type case. □

4.3.6 Lemma [Progress]: If $\emptyset \vdash e : T$ then either e is a result or $e \longrightarrow_n e'$.

Proof: By induction on the typing derivation.

(T_PreVal) u_{Id} is a result.

(T_TagVal) v_d is a result.

(T_TagValRefine) $v_{\{x:T|e\}?}$ is a result.

(T_Var) Contradictory—no such typing derivation.

(T_Fail) fail is a result.

(T_Coerce) $\emptyset \vdash \langle c \rangle\ e : T_2$, where $\vdash c : T_1 \rightsquigarrow T_2$ and $\emptyset \vdash e : T_1$. We go by cases on the IH for e:

($e = v$) We go by cases on c.

($c = \mathsf{Id}$) We step by F_TagId.

($c = \mathsf{Fail}; c'$) We step by F_Fail.

($c = B!; c'$) We step by F_TagB.

($c = B?; c'$) By canonical forms (Lemma 4.3.5), v is either ${v'}_{B'!}$ or ${v'}_{\mathbf{Fun}!}$. In the former case, we step by either F_TagBB or F_TagBFailB, depending on whether $B = B'$. In the latter case, we step by F_TagBFailFun.

($c = \mathbf{Fun}!; c'$) We step by F_TagFun.

($c = \mathbf{Fun}?; c'$) By canonical forms (Lemma 4.3.5), v is either ${v'}_{B'!}$ or ${v'}_{\mathbf{Fun}!}$. In the former case, we step by F_TagFunFailB. In the latter case, we step by F_TagFunFun.

($c = \{x{:}T \mid e\}?; c'$) We step by F_Check.

($c = \{x{:}T \mid e\}!; c'$) By canonical forms (Lemma 4.3.5), v must be of the form ${v'}_{\{x:T|e\}?}$, so we step by F_TagPredPred.

($c = (c_1 \mapsto c_2); c'$) We step by F_TagFunWrap.

($e = \mathsf{fail}$) We step by F_CoerceFail.

($e \longrightarrow_n e'$) If $e = \langle c' \rangle\ e'$, we ignore the step from the IH and instead step by F_Merge. Otherwise, we step by F_CoerceInner.

(T_Op) $\emptyset \vdash op(e_1, \ldots, e_n) : T$. We go by cases on the IH of each e_i, from left to right. If any of the e_i steps, we step by F_OpInner. If any of e_i is fail, we step by F_OpRaise. Finally, if all of the e_i are values, we step by F_Op.

(T_App) We have $\emptyset \vdash e_1\ e_2 : T_2$, where $\emptyset \vdash e_1 : T_1 \to T_2$ and $\emptyset \vdash e_2 : T_1$. We go by cases on IH for e_1.

($e_1 = v_1$) We go by cases on the IH for $\emptyset \vdash e_2 : T_1$.

($e_2 = v_2$) By canonical forms (Lemma 4.3.5), v_1 is either $\lambda x{:}T_1.\ e_{12\mathsf{Id}}$ or ${v'_1}_{c_1 \mapsto c_2}$. We step by F_Beta or F_Fun, respectively.

($e_2 = \mathsf{fail}$) We step by F_AppRaiseR.

($e_2 \longrightarrow_n e_2'$) We step by F_AppR.

($e_1 = \mathsf{fail}$) We step by F_AppRaiseL.

(otherwise) By the IH on $\emptyset \vdash e_1 : T_1 \to T_2$, we have $e_1 \longrightarrow_n e_1'$, and we step by F_AppL.

(T_CheckNaive) $\emptyset \vdash \langle \{x{:}T \mid e_1\}, e_2, v \rangle : \{x{:}T \mid e_1\}$. We go by cases on the IH for $\emptyset \vdash e_2 : \mathsf{Bool}$.

($e_2 = v_2$) By canonical forms (Lemma 4.3.5), v_2 is either $\mathsf{true}_{\mathsf{Id}}$ or $\mathsf{false}_{\mathsf{Id}}$. We step by F_CheckOK or F_CheckFail, respectively.

($e_2 = \mathsf{fail}$) We step by F_CheckRaise.

($e_2 \longrightarrow_n e_2'$) We step by F_CheckInner.

□

4.3.7 Lemma [Substitution]: If $\Gamma_1, x{:}T, \Gamma_2 \vdash e : T'$ and $\emptyset \vdash v : T$ then $\Gamma_1, \Gamma_2 \vdash e[v/x] : T'$. Similarly, if $\Gamma_1, x{:}T, \Gamma_2 \vdash u : T'$ and $\emptyset \vdash v : T$ then $\Gamma_1, \Gamma_2 \vdash u[v/x] : T'$.

Proof: By induction on the typing derivation, leaving Γ_2 general.

(T_Const) By WF_Const.

(T_Abs) By WF_Abs and the IH.

(T_PreVal) By T_PreVal and the IH.

(T_TagVal) By T_TagVal and the IH.

(T_TagValRefine) By T_TagValRefine and the IH, noting that all terms involved are actually closed.

(T_Var) Either by assumption (if the two variables are the same) or by T_Var and weakening.

(T_Fail) By T_Fail.

(T_Coerce) By T_Coerce and the IH.

(T_Op) By T_Op and the IH.

(T_App) By T_App and the IH.

(T_CheckNaive) By T_CheckNaive, noting that all terms involved are actually closed.

□

4.3.8 Lemma [Preservation]: If $\emptyset \vdash e : T$ and $e \longrightarrow_n e'$ then $\emptyset \vdash e' : T$.

Proof: By induction on the typing derivation $\emptyset \vdash e : T$.

(T_PreVal) Contradictory—doesn't step.

(T_TagVal) Contradictory—doesn't step.

(T_TagValRefine) Contradictory—doesn't step.

(T_Var) Contradictory—there is no such derivation.

(T_Fail) Contradictory—doesn't step.

(T_Coerce) By cases on the step taken.

(F_Check) $\emptyset \vdash \langle \{x{:}T_1 \mid e\}?; c\rangle\ v : T_2$ and $\langle \{x{:}T_1 \mid e\}?; c\rangle\ v \longrightarrow_n \langle c\rangle\ \langle \{x{:}T_1 \mid e\}, e[v/x], v\rangle$. By inversion, $\vdash c : \{x{:}T_1 \mid e\} \rightsquigarrow T_2$. We are done by T_CheckNaive and T_Coerce.

(F_TagId) $\emptyset \vdash \langle \mathsf{Id}\rangle\ v : T$ and $\langle \mathsf{Id}\rangle\ v \longrightarrow_n v$. By assumption.

(F_TagFunFailB) $\emptyset \vdash \langle \mathbf{Fun}?; c\rangle\ v_{B!} : T$ and $\langle \mathbf{Fun}?; c\rangle\ v_{B!} \longrightarrow_n \mathsf{fail}$. By regularity of coercion typing (Lemma 4.3.2) we can apply T_Fail.

(F_TagFunFun) $\emptyset \vdash \langle \mathbf{Fun}?; c\rangle\ v_{\mathbf{Fun}!} : T$ and $\langle \mathbf{Fun}?; c\rangle\ v_{\mathbf{Fun}!} \longrightarrow_n \langle c\rangle\ v$. By inversion $\vdash c : (\mathsf{Dyn}{\to}\mathsf{Dyn}) \rightsquigarrow T$, and we are done by assumption and T_Coerce.

(F_TagB) $\emptyset \vdash \langle B!; c\rangle\ v : T$ and $\langle B!; c\rangle\ v \longrightarrow_n \langle c\rangle\ v_{B!}$. By inversion, $\vdash c : \mathsf{Dyn} \rightsquigarrow T$ and $\emptyset \vdash v : B$. We finish by T_TagVal and T_Coerce.

(F_TagFun) $\emptyset \vdash \langle \mathbf{Fun}!; c\rangle\ v : T$ and $\langle \mathbf{Fun}!; c\rangle\ v \longrightarrow_n \langle c\rangle\ v_{\mathbf{Fun}!}$. By inversion, $\vdash c : \mathsf{Dyn} \rightsquigarrow T$ and $\emptyset \vdash v : \mathsf{Dyn}{\to}\mathsf{Dyn}$. We finish by T_TagVal and T_Coerce.

(F_TagBB) $\emptyset \vdash \langle B?; c\rangle\ v_{B!} : T$ and $\langle B?; c\rangle\ v_{B!} \longrightarrow_n \langle c\rangle\ v$. By inversion, $\vdash c : B \rightsquigarrow T$ and $\emptyset \vdash v : B$. We finish by T_Coerce.

(F_TagBFailB) $\emptyset \vdash \langle B?; c\rangle\ v_{B'!} : T$ and $\langle B?; c\rangle\ v_{B'!} \longrightarrow_n \mathsf{fail}$. By regularity of coercion typing (Lemma 4.3.2) and T_Fail.

(F_TagBFailFun) $\emptyset \vdash \langle B?; c\rangle\ v_{\mathbf{Fun}!} : T$ and $\langle B?; c\rangle\ v_{\mathbf{Fun}!} \longrightarrow_n \mathsf{fail}$. By regularity of coercion typing (Lemma 4.3.2) and T_Fail.

(F_TagFunWrap) $\emptyset \vdash \langle (c_1 \mapsto c_2); c\rangle\ v : T$ and $\langle (c_1 \mapsto c_2); c\rangle\ v \longrightarrow_n \langle c\rangle\ v_{c_1 \mapsto c_2}$. By inversion, $\vdash c : (T_{21}{\to}T_{22}) \rightsquigarrow T$ and $\emptyset \vdash v : T_{11}{\to}T_{12}$. By T_TagVal and T_Coerce.

(F_TagPredPred) We have:

$$\emptyset \vdash \langle \{x{:}T_1 \mid e\}!; c\rangle\ v_{\{x:T_1|e\}?} : T_2$$
$$\langle \{x{:}T_1 \mid e\}!; c\rangle\ v_{\{x:T_1|e\}?} \longrightarrow_n \langle c\rangle\ v$$

By inversion, $\vdash c : T_1 \rightsquigarrow T_2$ and $\emptyset \vdash v : T_1$. By T_Coerce.

(F_Fail) $\emptyset \vdash \langle \mathsf{Fail}; c\rangle\ e : T$ and $\langle \mathsf{Fail}; c\rangle\ e \longrightarrow_n \mathsf{fail}$. By regularity of coercion typing (Lemma 4.3.2), $\vdash T$, so we are done by T_Fail.

(F_CoerceInner) $\emptyset \vdash \langle c\rangle\ e : T_2$ and $\langle c\rangle\ e \longrightarrow_n \langle c\rangle\ e'$, where $e \longrightarrow_n e'$. By inversion, $\vdash c : T_1 \rightsquigarrow T_2$ and $\emptyset \vdash e : T_1$. By the IH on $\emptyset \vdash e : T_1$, $\emptyset \vdash e' : T_1$. We are done by T_Coerce.

(F_Merge) $\emptyset \vdash \langle c_1\rangle\ (\langle c_2\rangle\ e) : T_3$ and $\langle c_1\rangle\ (\langle c_2\rangle\ e) \longrightarrow_n \langle c_2; c_1\rangle\ e$. By inversion, $\vdash c_1 : T_2 \rightsquigarrow T_3$ and $\vdash c_2 : T_1 \rightsquigarrow T_2$ and $\emptyset \vdash e : T_1$. By C_Compose, $\vdash c_2; c_1 : T_1 \rightsquigarrow T_3$, so we are done by T_Coerce.

(F_CoerceRaise) $\emptyset \vdash \langle c\rangle\ \mathsf{fail} : T$ and $\langle c\rangle\ \mathsf{fail} \longrightarrow_n \mathsf{fail}$. By regularity of coercion typing (Lemma 4.3.2), we have $\vdash T$. We are done by T_Fail.

(T_Op) We go by cases on the step taken.

(F_Op) By assumption on the denotations of operations.

(F_OpInner) By the IH and T_OpInner.

(F_OpRaise) By regularity (Lemma 4.3.3) and T_Fail.

(T_App) We go by cases on the step taken.

(F_Beta) $\emptyset \vdash \lambda x{:}T_1.\ e_{12}{}_{\mathsf{Id}}\ v_2 : T_2$ and $\lambda x{:}T_1.\ e_{12}{}_{\mathsf{Id}}\ v_2 \longrightarrow_n e_{12}[v_2/x]$. By inversion, $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : T_1 \to T_2$; by further inversion, $x{:}T_1 \vdash e_1 : T_2$. By substitution (Lemma 4.3.7), $\emptyset \vdash e_1[v_2/x] : T_{22}$.

(F_Fun) $\emptyset \vdash v_1{}_{c_1 \mapsto c_2}\ v_2 : T_{22}$ and $v_1{}_{c_1 \mapsto c_2}\ v_2 \longrightarrow_n \langle c_2 \rangle\ (v_1\ (\langle c_1 \rangle\ v_2))$. By inversion, $\emptyset \vdash v_1{}_{c_1 \mapsto c_2} : T_{21} \to T_{22}$ and $\emptyset \vdash v_2 : T_{21}$; by further inversion, $\emptyset \vdash v_1 : T_{11} \to T_{12}$ and $\vdash c_1 : T_{21} \rightsquigarrow T_{11}$ and $\vdash c_2 : T_{12} \rightsquigarrow T_{22}$.

By T_Coerce, $\emptyset \vdash \langle c_1 \rangle\ v_2 : T_{11}$. By T_App, $\emptyset \vdash v_1\ (\langle c_1 \rangle\ v_2) : T_{12}$. By T_Coerce, $\emptyset \vdash \langle c_2 \rangle\ (v_1\ (\langle c_1 \rangle\ v_2)) : T_{22}$, and we are done.

(F_AppL) $\emptyset \vdash e_1\ e_2 : T_2$ and $e_1\ e_2 \longrightarrow_n e_1'\ e_2$ where $e_1 \longrightarrow_n e_1'$. By inversion, $\emptyset \vdash e_1 : T_1 \to T_2$ and $\emptyset \vdash e_2 : T_2$. By the IH on the former derivation, $\emptyset \vdash e_1' : T_1 \to T_2$, and we are done by T_App.

(F_AppR) $\emptyset \vdash v_1\ e_2 : T_2$ and $v_1\ e_2 \longrightarrow_n v_1\ e_2'$ where $e_2 \longrightarrow_n e_2'$. By inversion, $\emptyset \vdash v_1 : T_1 \to T_2$ and $\emptyset \vdash e_2 : T_2$. By the IH on the latter derivation, $\emptyset \vdash e_2' : T_2$, and we are done by T_App.

(F_AppRaiseL) $\emptyset \vdash \mathsf{fail}\ e_2 : T$. By regularity (Lemma 4.3.3), $\vdash T$, so we can apply T_Fail and be done.

(F_AppRaiseR) $\emptyset \vdash v_1\ \mathsf{fail} : T$. By regularity (Lemma 4.3.3), $\vdash T$, so we can apply T_Fail and be done.

(T_CheckNaive) We go by cases on the step taken.

(F_CheckOK) We have:

$$\emptyset \vdash \langle \{x{:}T \mid e\}, \mathsf{true}_{\mathsf{Id}}, v \rangle : \{x{:}T \mid e\}$$
$$\langle \{x{:}T \mid e\}, \mathsf{true}_{\mathsf{Id}}, v \rangle \longrightarrow_n v_{\{x:T|e\}}?$$

By inversion, $\emptyset \vdash v : T$ and $e[v/x] \longrightarrow_n^* \mathsf{true}_{\mathsf{Id}}$, so we are done by T_TagValRefine.

(F_CheckFail) We have:

$$\emptyset \vdash \langle \{x{:}T \mid e\}, \mathsf{false}_{\mathsf{Id}}, v \rangle : \{x{:}T \mid e\}$$
$$\langle \{x{:}T \mid e\}, \mathsf{false}_{\mathsf{Id}}, v \rangle \longrightarrow_n \mathsf{fail}$$

By inversion, $\vdash \{x{:}T \mid e\}$, and we are done by T_Fail.

(F_CheckInner) We have:

$$\emptyset \vdash \langle \{x{:}T \mid e_1\}, e_2, v \rangle : \{x{:}T \mid e_1\}$$
$$\langle \{x{:}T \mid e_1\}, e_2, v \rangle \longrightarrow_n \langle \{x{:}T \mid e_1\}, e_2', v \rangle$$

where $e_2 \longrightarrow_n e_2'$. By inversion, we have $\vdash \{x{:}T \mid e_1\}$ and $\emptyset \vdash v : T$ and $\emptyset \vdash e_2 : \mathsf{Bool}$ and $e_1[v/x] \longrightarrow_n^* e_2$. We now have $e_1[v/x] \longrightarrow_n^* e_2'$, and $\emptyset \vdash e_2' : \mathsf{Bool}$ by the IH, so we are done by T_CheckNaive.

(F_CheckRaise) $\emptyset \vdash \langle \{x{:}T \mid e\}, \mathsf{fail}, v\rangle : \{x{:}T \mid e\}$ and $\langle \{x{:}T \mid e\}, \mathsf{fail}, v\rangle \longrightarrow_n \mathsf{fail}$. By inversion, $\vdash \{x{:}T \mid e\}$, so we can apply T_Fail and be done.

□

4.3.9 Theorem [Type soundness]: If $\emptyset \vdash e : T$ then either $e \longrightarrow_n^* r$ such that $\emptyset \vdash r : T$ or e diverges.

Proof: Using progress (Lemma 4.3.6) and preservation (Lemma 4.3.8). Naturally the proof is not constructive! □

Before concluding the section, I establish *coercion congruence*, a property of evaluation that will be critical in relating Naive and Efficient in Section 4.6. We say e_1 and e_2 coterminate when (a) e_1 diverges iff e_2 diverges, and (b) $e_1 \longrightarrow_n^* v$ iff $e_2 \longrightarrow_n^* v$. Cotermination is obviously reflexive, symmetric, and transitive.

When I say that e diverges, we mean that for all e' such that $e \longrightarrow_n^* e'$, there exists an e'' such that $e' \longrightarrow_n e''$. Combined with the fact that values don't step, we can see that this definition coincides with another standard definition of divergence: e never reduces to a value, i.e., for all e' such that $e \longrightarrow_n^* e'$, it is never the case that e' is a value.

4.3.10 Lemma: If $e \longrightarrow_n e'$ then e and e' coterminate.

Proof: By soundness (Theorem 4.3.9), either e diverges or reduces a result.

By assumption, e takes at least one step, so it isn't a value. So if $e \longrightarrow_n^* e''$ to diverge or reduce to a value, then $e \longrightarrow_n e' \longrightarrow_n^* e''$ by determinism (Lemma 4.3.4). So e' behaves the same way. □

4.3.11 Corollary: $\langle c_1\rangle\ (\langle c_2\rangle\ e)$ and $\langle c_2; c_1\rangle\ e$ coterminate.

Proof: By Lemma 4.3.10 and F_Merge. □

4.3.12 Lemma [Coercion divergence congruence]: If e diverges then $\langle c\rangle\ e$ diverges.

Proof: It is never the case that $e \longrightarrow_n^* v$, and $\langle c\rangle\ e$ is never a value for any c or e. □

4.3.13 Lemma [Coercion congruence]: If $e \longrightarrow_n^* r$ then $\langle c\rangle\ r$ and $\langle c\rangle\ e$ coterminate.

Proof: We instead prove that there exists an e' such that $\langle c\rangle\ r \longrightarrow^* e'$ and $\langle c\rangle\ e \longrightarrow^* e'$, which implies coterminationby way of Lemma 4.3.10. We go by induction on $e \longrightarrow_n^* r$.

($r \longrightarrow_n^0 r$) Immediate.

($e \longrightarrow_n e' \longrightarrow_n^* r$) By cases on $e \longrightarrow_n e'$. Rules without coercions (the core rules F_Beta, F_Fun, F_Op, F_CheckOK, F_CheckFail; the congruence rules F_AppL, F_AppR, F_OpInner, F_CheckInner; and the exception raising rules F_AppRaiseL, F_AppRaiseR, F_OpRaise, and F_CheckRaise) are relatively easy: there is no merging. These cases work by first applying F_CoerceInner and the original rule to find $\langle c \rangle\ e \longrightarrow_n \langle c \rangle\ e'$, and then we are done by the IH, since $e \longrightarrow_n e'$ and $e \longrightarrow_n^* r$ implies $e' \longrightarrow_n^* r$ by determinism (Lemma 4.3.4).

The remaining cases must confront some degree of merging between coercions in e and the coercion c. The general thrust is to observe that if $e = \langle c' \rangle\ e \longrightarrow_n \langle c'' \rangle\ e'$, then $\langle c \rangle\ (\langle c' \rangle\ e) \longrightarrow_n \langle c'; c \rangle\ e \longrightarrow_n \langle c''; c \rangle\ e''$. We can then apply the IH on $\langle c'' \rangle\ e' \longrightarrow_n^* r$ to find an e'' such that $\langle c \rangle\ (\langle c'' \rangle\ e') \longrightarrow_n^* e''$ and $\langle c \rangle\ r \longrightarrow_n^* e''$. Finally, the former term steps by F_Merge to $\langle c''; c \rangle\ e'$, which must in turn reduce to e''—so we are done.

(F_TagId) $\langle \mathsf{Id} \rangle\ e \longrightarrow_n e$, so $\langle c \rangle\ (\langle \mathsf{Id} \rangle\ e) \longrightarrow_n \langle \mathsf{Id}; c \rangle\ e = \langle c \rangle\ e$. By the IH on $e \longrightarrow_n^* r$, we know that there exists an e' such that $\langle c \rangle\ e \longrightarrow_n^* e'$ and $\langle c \rangle\ r \longrightarrow_n^* e'$.

(F_TagFunFailB) $\langle \mathbf{Fun}?; c' \rangle\ v_{0\,B!} \longrightarrow_n \mathsf{fail}$, so

$$\langle c \rangle\ (\langle \mathbf{Fun}?; c' \rangle\ v_{0\,B!}) \longrightarrow_n \langle \mathbf{Fun}?; c'; c \rangle\ v_{0\,B!} \longrightarrow_n \mathsf{fail}$$

by F_Merge and F_TagFunFailB.

So $e' = \mathsf{fail}$, since $\langle c \rangle\ \mathsf{fail} \longrightarrow_n \mathsf{fail}$ by F_CoerceRaise.

(F_TagFunFun) $\langle \mathbf{Fun}?; c' \rangle\ v_{0\,\mathbf{Fun}!} \longrightarrow_n \langle c' \rangle\ v_0$, so

$$\langle c \rangle\ (\langle \mathbf{Fun}?; c' \rangle\ v_{0\,\mathbf{Fun}!}) \longrightarrow_n \langle \mathbf{Fun}?; c'; c \rangle\ v_{0\,\mathbf{Fun}!} \longrightarrow_n \langle c'; c \rangle\ v_0$$

by F_Merge and F_TagFunFun. We know that $\langle c' \rangle\ v_0 \longrightarrow_n^* v$, and by the IH there exists an e' such that $\langle c \rangle\ (\langle c' \rangle\ v_0) \longrightarrow_n^* e'$ and $\langle c \rangle\ r \longrightarrow_n^* e'$. We conclude by applying F_Merge on the former term along with Corollary 4.3.11.

(F_TagB) $\langle B!; c' \rangle\ v_0 \longrightarrow_n \langle c \rangle\ v_{0\,B!}$, so $\langle c \rangle\ (\langle B!; c' \rangle\ v_0) \longrightarrow_n \langle B!; c'; c \rangle\ v_0 \longrightarrow_n \langle c'; c \rangle\ v_{0\,B!}$. Since $\langle c' \rangle\ v_{0\,B!} \longrightarrow_n^* v$, we know that there exists an e' such that $\langle c \rangle\ (\langle c' \rangle\ v_{0\,B!}) \longrightarrow_n^* e'$ and $\langle c \rangle\ r \longrightarrow_n^* e'$ by the IH. We conclude by applying F_Merge on the former term along with Corollary 4.3.11.

(F_TagFun) As for F_TagB.

(F_TagBB) As for F_TagFunFun.

(F_TagBFailB) As for F_TagFunFailB.

(F_TagBFailFun) As for F_TagBFailFun.

(F_TagFunWrap) $\langle (c_1 \mapsto c_2); c' \rangle\ v_0 \longrightarrow_n \langle c' \rangle\ v_{0\,c_1 \mapsto c_2}$, so

$$\langle c \rangle\ (\langle (c_1 \mapsto c_2); c' \rangle\ v_0) \longrightarrow_n \langle (c_1 \mapsto c_2); c'; c \rangle\ v_0 \longrightarrow_n \langle c'; c \rangle\ v_{0\,c_1 \mapsto c_2}$$

Since $\langle c' \rangle\ v_{0\,c_1 \mapsto c_2} \longrightarrow_n^* v$, we know by the IH that there exists an e' such that $\langle c \rangle\ (\langle c' \rangle\ v_{0\,c_1 \mapsto c_2}) \longrightarrow_n^* e'$ and $\langle c \rangle\ r \longrightarrow_n^* e'$. We conclude by applying F_Merge on the former term along with Corollary 4.3.11.

(F_TagPredPred) As for F_TagFunFun and F_TagBB.

(F_Check) $\langle \{x{:}T \mid e\}?; c' \rangle\ v_0 \longrightarrow_n \langle c' \rangle\ \langle \{x{:}T \mid e\}, e[v_0/x], v_0 \rangle$, so

$$\langle c \rangle\ (\langle \{x{:}T \mid e\}?; c' \rangle\ v_0) \longrightarrow_n \langle \{x{:}T \mid e\}?; c'; c \rangle\ v_0 \longrightarrow_n \langle c'; c \rangle\ \langle \{x{:}T \mid e\}, e[v_0/x], v_0 \rangle$$

by F_Merge and F_Check. Since $\langle c' \rangle\ \langle \{x{:}T \mid e\}, e[v_0/x], v_0 \rangle \longrightarrow_n^* r$, the IH gives us an e' such that $\langle c \rangle\ (\langle c' \rangle\ \langle \{x{:}T \mid e\}, e[v_0/x], v_0 \rangle) \longrightarrow_n^* e'$ and $\langle c \rangle\ r \longrightarrow_n^* e'$; we are done by applying F_Merge and Corollary 4.3.11 on the left.

(F_CoerceInner) $\langle c' \rangle\ e \longrightarrow_n \langle c' \rangle\ e'$, so $\langle c \rangle\ (\langle c' \rangle\ e) \longrightarrow_n \langle c'; c \rangle\ e \longrightarrow_n \langle c'; c \rangle\ e'$. Since $\langle c' \rangle\ e' \longrightarrow_n^* r$, the IH gives us an e'' such that $\langle c \rangle\ (\langle c' \rangle\ e') \longrightarrow_n^* e''$. This last steps immediately by F_Merge to $\langle c'; c \rangle\ e'$, so we know that term goes to e''', too.

(F_Merge) $\langle c_1 \rangle\ (\langle c_2 \rangle\ e) \longrightarrow_n \langle c_2; c_1 \rangle\ e$, so

$$\langle c \rangle\ (\langle c_1 \rangle\ (\langle c_2 \rangle\ e)) \longrightarrow_n \langle c_1; c \rangle\ (\langle c_2 \rangle\ e) \longrightarrow_n \langle c_2; c_1; c \rangle\ e$$

We know that $\langle c_2; c_1 \rangle\ e \longrightarrow_n^* r$, so the IH gives us an e' such that $\langle c \rangle\ (\langle c_2; c_1 \rangle\ e) \longrightarrow_n^* e'$ and $\langle c \rangle\ r \longrightarrow_n^* e'$. But the former term steps by F_Merge to $\langle c_2; c_1; c \rangle\ e$, so we are done by Corollary 4.3.11.

(F_Fail) $\langle \mathsf{Fail}; c' \rangle\ e \longrightarrow_n \mathsf{fail}$, so $\langle c \rangle\ (\langle \mathsf{Fail}; c' \rangle\ e) \longrightarrow_n \langle \mathsf{Fail}; c'; c \rangle\ e \longrightarrow_n \mathsf{fail}$. We have $e' = r = \mathsf{fail}$, since $\langle c \rangle\ \mathsf{fail} \longrightarrow_n^* \mathsf{fail}$ by F_CoerceRaise.

(F_CoerceRaise) $\langle c' \rangle\ \mathsf{fail} \longrightarrow_n \mathsf{fail}$, so $\langle c \rangle\ (\langle c' \rangle\ \mathsf{fail}) \longrightarrow_n \langle c'; c \rangle\ \mathsf{fail} \longrightarrow_n \mathsf{fail}$, and again $e' = r = \mathsf{fail}$ as in F_Fail, so we are done.

□

4.4 Soundness of Naive with regard to Cast

I use a step-indexed logical relation (defined in Figure 4.16) to show that the terms translated from Cast to Naive by way of the function coerce (defined in Figure 4.15) are behaviorally equivalent. We are forced to use step indices to account for the type structure of dynamic types, i.e., that the "universal" type Dyn includes (tagged values from) the types Dyn→Dyn and B.

The proof strategy is the same as that of Greenberg et al. [34] and Belo et al. [8]: we separately relate the contract portions of the two languages (Figure 4.17). After showing that related contract checks (here, casts and coercions) behave in related ways on related values, we can show that Cast terms are related to their Naive translations.

This proof is a simpler version of the proof relating Naive and Efficient in Section 4.6.

Translating casts $\boxed{\mathsf{coerce}(T_1, T_2) : T}$

$$
\begin{array}{rcl}
\mathsf{coerce}(T, T) & = & \mathsf{Id} \\
 & & \text{when } T \neq T_1 \to T_2 \\
\mathsf{coerce}(\mathsf{Dyn}, T_1 \to T_2) & = & \mathbf{Fun}?; \mathsf{coerce}(\mathsf{Dyn} \to \mathsf{Dyn}, T_1 \to T_2) \\
\mathsf{coerce}(\mathsf{Dyn}, B) & = & B? \\
\mathsf{coerce}(B, \mathsf{Dyn}) & = & B! \\
\mathsf{coerce}(T_1 \to T_2, \mathsf{Dyn}) & = & \mathsf{coerce}(T_1 \to T_2, \mathsf{Dyn} \to \mathsf{Dyn}); \mathbf{Fun}! \\
\mathsf{coerce}(T_{11} \to T_{12}, T_{21} \to T_{22}) & = & \mathsf{coerce}(T_{21}, T_{11}) \mapsto \mathsf{coerce}(T_{12}, T_{22}) \\
\mathsf{coerce}(\{x{:}T_1 \mid e\}, T_2) & = & \{x{:}T_1 \mid \mathsf{coerce}(e)\}!; \mathsf{coerce}(T_1, T_2) \\
 & & \text{when } T_2 \neq \{x{:}T_1 \mid e\} \\
\mathsf{coerce}(T_1, \{x{:}T_2 \mid e\}) & = & \mathsf{coerce}(T_1, T_2); \{x{:}T_2 \mid \mathsf{coerce}(e)\}? \\
 & & \text{when } T_1 \neq T_2 \text{ and } T_1 \neq \{x{:}T_1' \mid e'\}
\end{array}
$$

Translating pre-values and terms $\boxed{\mathsf{coerce}(u) : u}$ $\boxed{\mathsf{coerce}(e) : e}$

$$
\begin{array}{rcl}
\mathsf{coerce}(k) & = & k \\
\mathsf{coerce}(\lambda x{:}T.\ e) & = & \lambda x{:}\mathsf{coerce}(T).\ \mathsf{coerce}(e)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{coerce}(x) & = & x \\
\mathsf{coerce}(u_{\mathsf{Id}}) & = & \mathsf{coerce}(u)_{\mathsf{Id}} \\
\mathsf{coerce}(v_{B!}) & = & \mathsf{coerce}(v)_{B!} \\
\mathsf{coerce}(v_{\mathbf{Fun}!}) & = & \mathsf{coerce}(v)_{\mathbf{Fun}!} \\
\mathsf{coerce}(v_{\langle T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22} \rangle}) & = & \mathsf{coerce}(v)_{(\mathsf{coerce}(T_{21}, T_{11}) \mapsto \mathsf{coerce}(T_{12}, T_{22}))} \\
\mathsf{coerce}(v_{\{x:T \mid e\}?}) & = & \mathsf{coerce}(v)_{\{x:T \mid \mathsf{coerce}(e)\}?} \\
\mathsf{coerce}(\langle T_1 \Rightarrow T_2 \rangle\ e) & = & \langle \mathsf{coerce}(T_1, T_2) \rangle\ \mathsf{coerce}(e) \\
\mathsf{coerce}(e_1\ e_2) & = & \mathsf{coerce}(e_1)\ \mathsf{coerce}(e_2) \\
\mathsf{coerce}(op(e_1, \ldots, e_n)) & = & op(\mathsf{coerce}(e_1), \ldots, \mathsf{coerce}(e_n)) \\
\mathsf{coerce}(\langle \{x{:}T \mid e_1\}, e_2, v \rangle) & = & \langle \mathsf{coerce}(\{x{:}T \mid e_1\}), \mathsf{coerce}(e_2), \mathsf{coerce}(v) \rangle
\end{array}
$$

Translating types and contexts $\boxed{\mathsf{coerce}(T) : T}$ $\boxed{\mathsf{coerce}(\Gamma) : \Gamma}$

$$
\begin{array}{rcl}
\mathsf{coerce}(\mathsf{Dyn}) & = & \mathsf{Dyn} \\
\mathsf{coerce}(B) & = & B \\
\mathsf{coerce}(T_1 \to T_2) & = & \mathsf{coerce}(T_1) \to \mathsf{coerce}(T_2) \\
\mathsf{coerce}(\{x{:}T \mid e\}) & = & \{x{:}T \mid \mathsf{coerce}(e)\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{coerce}(\emptyset) & = & \emptyset \\
\mathsf{coerce}(\Gamma, x{:}T) & = & \mathsf{coerce}(\Gamma), x{:}\mathsf{coerce}(T)
\end{array}
$$

Figure 4.15: Translating from CAST to NAIVE

Value rules $\boxed{v_1 \sim^j v_2 : T}$

$$\begin{array}{rcl}
\forall j.\ k_{\mathsf{Id}} \sim^j k_{\mathsf{Id}} : B & \iff & \mathsf{ty}(k) = B \\
v_{11} \sim^j v_{21} : T_1 \to T_2 & \iff & \\
\forall m < j.\ \forall v_{12} \sim^m v_{22} : T_1.\ v_{11}\ v_{12} \simeq^m v_{21}\ v_{22} : T_2 &&
\end{array}$$

$$\begin{array}{rcl}
v_{1\,B!} \sim^j v_{2\,B!} : \mathsf{Dyn} & \iff & v_1 \sim^j v_2 : B \\
v_{1\,\mathbf{Fun!}} \sim^j v_{2\,\mathbf{Fun!}} : \mathsf{Dyn} & \iff & v_1 \sim^j v_2 : \mathsf{Dyn} \to \mathsf{Dyn}
\end{array}$$

$$\begin{array}{c}
v_{1\,\{x:T|e_1\}?} \sim^j v_{2\,\{x:T|e_2\}?} : \{x{:}T \mid e_1\} \\
\iff \\
\forall m < j.\ v_1 \sim^m v_2 : T \wedge \{x{:}T \mid e_1\} \sim^m \{x{:}T \mid e_2\}
\end{array}$$

Term rules $\boxed{e_1 \simeq^j e_2 : T}$

$$\begin{array}{c}
e_1 \simeq^j e_2 : T \iff \\
e_1 \text{ diverges} \vee \\
\forall m < j. \quad e_1 \longrightarrow_c^m \mathsf{fail} \implies e_2 \longrightarrow_c^* \mathsf{fail} \\
\wedge \quad e_1 \longrightarrow_c^m v_1 \implies e_2 \longrightarrow_c^* v_2 \wedge v_1 \sim^{(j-m)} v_2 : T
\end{array}$$

Type rules $\boxed{T_1 \sim^j T_2}$

$$\begin{array}{rcl}
B \sim^j B & & \mathsf{Dyn} \sim^j \mathsf{Dyn} \\
T_{11} \to T_{12} \sim^j T_{21} \to T_{22} & \iff & T_{11} \sim^j T_{21} \wedge T_{12} \sim^j T_{22} \\
\{x{:}T \mid e_1\} \sim^j \{x{:}T \mid e_2\} & \iff & \\
\forall m < j.\ \forall v_1 \sim^m v_2 : T.\ e_1[v_1/x] \simeq^m e_2[v_2/x] : \mathsf{Bool} &&
\end{array}$$

Closing substitutions and open terms $\boxed{\Gamma \models^j \delta}$ $\boxed{\Gamma \vdash e_1 \simeq e_2 : T}$

$$\begin{array}{rcl}
\Gamma \models^j \delta & \iff & \forall x \in \mathrm{dom}(\Gamma).\ \delta_1(x) \sim^j \delta_2(x) : T \\
\Gamma \vdash e_1 \simeq e_2 : T & \iff & \forall j \geq 0.\ \forall \Gamma \models^j \delta.\ \delta_1(e_1) \simeq^j \delta_2(e_2) : T
\end{array}$$

Figure 4.16: Relating Cast and Naive

4.4.1 Lemma [Preservation for coerce]: Assuming that no refinement tags or active checking forms are present:

1. If $\Gamma \vdash e : T$ then $\mathsf{coerce}(\Gamma) \vdash \mathsf{coerce}(e) : \mathsf{coerce}(T)$.
2. If $\Gamma \vdash u : T$ then $\mathsf{coerce}(\Gamma) \vdash \mathsf{coerce}(u) : \mathsf{coerce}(T)$.
3. If $\vdash T_1$ and $\vdash T_2$ and $\vdash T_1 \parallel T_2$ then $\vdash \mathsf{coerce}(T_1, T_2) : \mathsf{coerce}(T_1) \rightsquigarrow \mathsf{coerce}(T_2)$.
4. If $\vdash T$ then $\vdash \mathsf{coerce}(T)$.
5. If $\vdash \Gamma$ then $\vdash \mathsf{coerce}(\Gamma)$.

Proof: By simultaneous induction on the typing/well formedness/similarity derivations.

(WF_Empty) By WF_Empty.

(WF_Extend) By WF_Extend, using IHs (4) and (5).

(WF_Dyn) By WF_Dyn.

(WF_Base) By WF_Base.

(WF_Fun) By WF_Fun and IH (4).

(WF_Refine) By WF_Refine and IH (1).

(T_Const) By WF_Const and IH (5).

(T_Abs) By WF_Abs and IHs (4) and (1).

(T_Var) By WF_Var and IH (5).

(T_PreVal) By WF_PreVal and IH (2).

(T_TagB) By IH (1), $\mathsf{coerce}(\Gamma) \vdash \mathsf{coerce}(v) : B$. By C_Tag, $\vdash B! : B \rightsquigarrow \mathsf{Dyn}$. We are done by T_TagVal.

(T_TagFun) By IH (1), $\mathsf{coerce}(\Gamma) \vdash \mathsf{coerce}(v) : \mathsf{Dyn}{\to}\mathsf{Dyn}$. By C_Tag, we have $\vdash \mathbf{Fun}! : (\mathsf{Dyn}{\to}\mathsf{Dyn}) \rightsquigarrow \mathsf{Dyn}$. We are done by T_TagVal.

(T_TagRefine) Contradiction—we assumed that refinement tags do not appear.

(T_Wrap) By IH (1), $\mathsf{coerce}(\Gamma) \vdash \mathsf{coerce}(v) : \mathsf{coerce}(T_{11} \to T_{12})$. By Naive regularity (Lemma 4.3.3) and IH (4), we have $\vdash \mathsf{coerce}(T_{11} \to T_{12})$ and $\vdash \mathsf{coerce}(T_{21} \to T_{22})$. Since we also have $\vdash T_{11} \to T_{12} \parallel T_{21} \to T_{22}$, we can apply IH (3) to find that

$$\vdash \mathsf{coerce}(T_{11} \to T_{12}, T_{21} \to T_{22}) : \mathsf{coerce}(T_{11} \to T_{12}) \rightsquigarrow \mathsf{coerce}(T_{21} \to T_{22})$$

Recall that $\mathsf{coerce}(T_{11} \to T_{12}, T_{21} \to T_{22}) = \mathsf{coerce}(T_{21}, T_{11}) \mapsto \mathsf{coerce}(T_{12}, T_{22})$. So by WF_TagVal, we are done.

(T_Fail) By WF_Fail and IHs (4) and (5).

(T_Cast) By WF_Coerce, Naive regularity (Lemma 4.3.3), and IHs (1), (4), and (3).

(T_App) By WF_App and IH (1).

(T_Op) By WF_Op and IH (1).

(T_CheckCast) Contradictory—we assumed that there were no active checks.

(P_Id) Since T can't be a function type, $\mathsf{coerce}(T, T) = \mathsf{Id}$, and we are done by C_Id.

(P_DynL) If T_2 is a function, then by C_Compose with C_Untag and IH (3). If it is B, then by C_Untag. Finally, if it is a refinement, then by C_Untag and IH (4) along with C_Compose and IH (3).

(P_DynR) By C_Tag and IH (4), C_Compose and IH (3).

(P_Fun) By C_Fun and IH (3).

(P_RefineL) By C_Compose and IH (3), with C_Tag and IH (4).

(P_RefineR) By C_Compose and IH (3), with C_Untag and IH (4).

□

4.4.2 Lemma: If $\vdash T_1 \Rightarrow T_2 \equiv c$, then for all $v_1 \sim^j v_2 : T_1$, we have $\langle T_1 \Rightarrow T_2 \rangle\ v_1 \simeq^j \langle \mathsf{coerce}(T_1, T_2) \rangle\ v_2 : T_2$.

Proof: By induction on the derivation of $\vdash T_1 \Rightarrow T_2 \equiv c$. In all cases, we begin by letting $m < j$ be given such that $\langle T_1 \Rightarrow T_2 \rangle\ v_1 \longrightarrow_c^m r_1$. (If the left-hand side diverges, we are done.)

(RC_Id) We must show that $\langle T \Rightarrow T \rangle\ v_1 \simeq^m \langle \mathsf{Id} \rangle\ v_2 : T$, where T is not a function type. The left-hand side must step by G_CastId to v_1. The right-hand side steps by F_TagId to v_2. Since $v_1 \sim^j v_2 : T$ and $m < j$, we are done.

$\boxed{\vdash T_1 \Rightarrow T_2 \equiv c}$

$$\frac{T \neq T_1 \to T_2}{\vdash T \Rightarrow T \equiv \mathsf{Id}}\ \text{RC\_Id} \qquad \frac{}{\vdash \mathsf{Dyn} \Rightarrow B \equiv B?}\ \text{RC\_DynB} \qquad \frac{}{\vdash B \Rightarrow \mathsf{Dyn} \equiv B!}\ \text{RC\_BDyn}$$

$$\frac{\vdash \mathsf{Dyn}\to\mathsf{Dyn} \Rightarrow T_1 \to T_2 \equiv c}{\vdash \mathsf{Dyn} \Rightarrow T_1 \to T_2 \equiv \mathbf{Fun}?; c}\ \text{RC\_DynFun} \qquad \frac{\vdash T_1 \to T_2 \Rightarrow \mathsf{Dyn}\to\mathsf{Dyn} \equiv c}{\vdash T_1 \to T_2 \Rightarrow \mathsf{Dyn} \equiv c; \mathbf{Fun}!}\ \text{RC\_FunDyn}$$

$$\frac{\vdash T_{21} \Rightarrow T_{11} \equiv c_1 \quad \vdash T_{12} \Rightarrow T_{22} \equiv c_2}{\vdash T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22} \equiv c_1 \mapsto c_2}\ \text{RC\_Fun}$$

$$\frac{T_2 \neq \{x{:}T_1 \mid e_1\} \quad x{:}T_1 \vdash e_1 \simeq e_2 : \mathsf{Bool} \quad \vdash T_1 \Rightarrow T_2 \equiv c}{\vdash \{x{:}T_1 \mid e_1\} \Rightarrow T_2 \equiv \{x{:}T_1 \mid e_2\}!; c}\ \text{RC\_PredPred}$$

$$\frac{T_1 \neq \{x{:}T_1' \mid e'\} \quad \vdash T_1 \Rightarrow T_2 \equiv c \quad x{:}T_2 \vdash e_1 \simeq e_2 : \mathsf{Bool}}{\vdash T_1 \Rightarrow \{x{:}T_2 \mid e_1\} \equiv c; \{x{:}T_2 \mid e_2\}?}\ \text{RC\_PreCheck}$$

Figure 4.17: Relating casts and Naive coercions

(RC_DynB) We must show that $\langle \mathsf{Dyn} \Rightarrow B \rangle\ v_1 \simeq^m \langle B? \rangle\ v_2 : B$. There are two ways to have $v_1 \sim^j v_2 : \mathsf{Dyn}$: either they are both tagged with base type $B'!$ or with as functions with **Fun**!.

If they are tagged as functions, then both sides step to fail. (The left-hand side steps by G_CastBFailFun, the right by F_TagBFailFun.) If they are tagged with $B' \neq B$, then both sides step to fail, on the left by G_CastBFailB and by F_TagBFailB on the right.

If they are tagged with $B' = B$, then they will reduce to v_1 and v_2 respectively, by G_CastBB on the left and F_TagBB and F_TagId on the right. Since we have $v_1 \sim^j v_2 : B$, we can find $v_1 \sim^{(j-m)} v_2 : B$ and be done.

(RC_BDyn) We must show that $\langle B \Rightarrow \mathsf{Dyn} \rangle\ v_1 \simeq^m \langle B! \rangle\ v_2 : \mathsf{Dyn}$. The left-hand side steps by G_CastB and the right-hand side by F_TagB and F_TagId, leaving us to prove $v_{1\,B!} \sim^{(j-m)} v_{2\,B!} : \mathsf{Dyn}$, which we have immediately by assumption (since $m < j$).

(RC_DynFun) We must show that $\langle \mathsf{Dyn} \Rightarrow T_1 \to T_2 \rangle\ v_1 \simeq^m \langle \mathbf{Fun}?; c \rangle\ v_2 : T_1 \to T_2$. First, we go by cases as for the RC_DynB case above. Easily solving the cases when v_1 and v_2 are tagged with $B!$. So $v_1 = v_{1\,\mathbf{Fun!}}'$ and $v_2 = v_{2\,\mathbf{Fun!}}'$. The left-hand side steps by G_CastFunFun and the right hand side steps by F_TagFunFun, giving us $\langle \mathsf{Dyn}\to\mathsf{Dyn} \Rightarrow T_1 \to T_2 \rangle\ v_1'$ on the left and $\langle c \rangle\ v_2'$ on the right. We are done by the IH.

(RC_FunDyn) We must show that $\langle T_1 \to T_2 \Rightarrow \mathsf{Dyn} \rangle\ v_1 \simeq^m \langle c; \mathbf{Fun}! \rangle\ v_2 : \mathsf{Dyn}$,

where $\vdash \mathsf{Dyn}{\to}\mathsf{Dyn} \Rightarrow T_1{\to}T_2 \equiv c$.

The left-hand side must step by G_CastFunDyn, giving us $\langle \mathsf{Dyn}{\to}\mathsf{Dyn} \Rightarrow \mathsf{Dyn}\rangle\ (\langle T_1{\to}T_2 \Rightarrow \mathsf{Dyn}{\to}\mathsf{Dyn}\rangle\ v_1)$. We know that $\langle T_1{\to}T_2 \Rightarrow \mathsf{Dyn}{\to}\mathsf{Dyn}\rangle\ v_1 \simeq^j \langle c\rangle\ v_2 : \mathsf{Dyn}{\to}\mathsf{Dyn}$ by the IH.

Now we consider for a moment $\langle T_1{\to}T_2 \Rightarrow \mathsf{Dyn}{\to}\mathsf{Dyn}\rangle\ v_1 \simeq^j \langle c\rangle\ v_2 : \mathsf{Dyn}{\to}\mathsf{Dyn}$. If the left-hand side diverges, we are done. Similarly, if both sides reduce to fail, then so do our original terms, and we are done. So suppose they reduce to $v_1' \sim^{(j-m')} v_2' : \mathsf{Dyn}{\to}\mathsf{Dyn}$ for some $m' < j$. By coercion congruence (Lemma 4.3.13), we know that $\langle c; \mathbf{Fun!}\rangle\ v_2 \longrightarrow_n^* \langle \mathbf{Fun!}\rangle\ v_2'$

We can expand the right-hand side to $\langle \mathbf{Fun!}\rangle\ (\langle c\rangle\ v_2) \longrightarrow_n \langle c; \mathbf{Fun!}\rangle\ v_2$, so we can now reduce on the left and right to find $\langle \mathsf{Dyn}{\to}\mathsf{Dyn} \Rightarrow \mathsf{Dyn}\rangle\ v_1'$ on the left and $\langle \mathbf{Fun!}\rangle\ v_2'$ on the right. The left steps by G_CastFun and the right by F_TagFun and F_TagId. We then have $v_1'_{\mathbf{Fun!}} \sim^{(j-m'-1)} v_2'_{\mathbf{Fun!}} : \mathsf{Dyn}$. If $j - m' - 1$ is below m, we are done with a smaller evaluation derivation. If it's larger, we are done trivially, as determinism (Lemma 4.2.1) guarantees that this is the only evaluation derivation.

(RC_Fun) We must show that $\langle T_{11}{\to}T_{12} \Rightarrow T_{21}{\to}T_{22}\rangle\ v_1 \simeq^m \langle c_1 \mapsto c_2\rangle\ v_2 : T_{21}{\to}T_{22}$, given that $\vdash T_{21} \Rightarrow T_{11} \equiv c_1$ and $\vdash T_{12} \Rightarrow T_{22} \equiv c_2$.

The left-hand side steps by G_CastFunWrap to $v_1{}_{\langle T_{11}\to T_{12} \Rightarrow T_{21}\to T_{22}\rangle}$; the right-hand side steps by F_TagFunWrap and F_TagId to $v_1{}_{c_1 \mapsto c_2}$.

Let $m' < m$ such that $v_1' \sim^{m'} v_2' : T_{21}$. We must show that

$$v_1{}_{\langle T_{11}\to T_{12} \Rightarrow T_{21}\to T_{22}\rangle}\ v_1' \simeq^{m'} v_2{}_{c_1 \mapsto c_2}\ v_2' : T_{21}{\to}T_{22}$$

The left-hand side steps by G_Fun to $\langle T_{12} \Rightarrow T_{22}\rangle\ (v_1\ (\langle T_{21} \Rightarrow T_{11}\rangle\ v_1'))$; the right-hand side steps by F_Fun to $\langle c_2\rangle\ (v_2\ (\langle c_1\rangle\ v_2'))$. By the IH on $\vdash T_{21} \Rightarrow T_{11} \equiv c_1$, we can find either (a) divergence on the left, (b) failure on the left and right, or (c) a value $v_1'' \sim^j v_2'' : T_{11}$.

In this last case, we know that $v_1\ v_1'' \simeq^{m''} v_2\ v_2'' : T_{12}$ for all $m'' < j$, so we again find either (a) divergence on the left, (b) failure on the left and right, or (c) a value $v_1''' \sim^j v_2''' : T_{12}$.

In this last case, we know by the IH on $\vdash T_{12} \Rightarrow T_{22} \equiv c_2$ that $\langle T_{12} \Rightarrow T_{22}\rangle\ v_1''' \simeq^j \langle c_2\rangle\ v_2''' : T_{22}$, and we are done.

(RC_PredPred) We must show $\langle \{x{:}T_1 \mid e_1\} \Rightarrow T_2\rangle\ v_1 \simeq^m \langle \{x{:}T_1 \mid e_2\}!; c\rangle\ v_2 : T_2$, given $x{:}T_1 \vdash e_1 \simeq e_2 : \mathsf{Bool}$ and $\vdash T_1 \Rightarrow T_2 \equiv c$.

The only way to have $v_1 \sim^j v_2 : \{x{:}T_1 \mid e_1\}$ is to have both values tagged as $v_1'{}_{\{x:T_1|e_1\}}?$ and $v_2'{}_{\{x:T_1|e_2\}}?$. (Anything else would be ill typed.)

So the left-hand side steps by G_CastPredPred and the right-hand steps by F_TagPredPred, giving us $\langle T_1 \Rightarrow T_2\rangle\ v_1'$ and $\langle c\rangle\ v_2'$, respectively. By assumption we have $v_1' \sim^j v_2' : T_1$, so we are done by the IH.

(RC_PreCheck) We must show $\langle T_1 \Rightarrow \{x{:}T_2 \mid e_1\}\rangle\ v_1 \simeq^m \langle c; \{x{:}T_2 \mid e_2\}?\rangle\ v_2 : \{x{:}T_2 \mid e_1\}$ given $\vdash T_1 \Rightarrow T_2 \equiv c$ and $x{:}T_2 \vdash e_1 \simeq e_2 : \mathsf{Bool}$.

In Cast we step by G_CastPreCheck, yielding $\langle T_2 \Rightarrow \{x{:}T_2 \mid e_1\}\rangle\ (\langle T_1 \Rightarrow T_2\rangle\ v_1)$.

As in RC_FunDyn, we consider the inner term for a moment. We know by the IH that $\langle T_1 \Rightarrow T_2\rangle\ v_1 \simeq^j \langle c\rangle\ v_2 : T_2$. If the left-hand side diverges or both go to fail, we are done—our whole term will behave the same way, in the same number of steps. So we only need to consider the case where both sides reduce to $v_1' \sim^{(j-m')} v_2' : T_2$ for some $m' < j$. By coercion congruence (Lemma 4.3.13), we know that $\langle c; \{x{:}T \mid e_2\}?\rangle\ v_2 \longrightarrow_n^* \langle \{x{:}T \mid e_2\}?\rangle\ v_2'$.

Now we can see that both sides reduce again: on the left, by G_CastCheck; on the right, by F_TagCheck.

Since we have $x{:}T_2 \vdash e_1 \simeq e_2 : \mathsf{Bool}$, we know that $e_1[v_1'/x] \simeq^j e_2[v_2'/x] : \mathsf{Bool}$, so the checks behave the same: either the left-hand side diverges, both return fail, or they return related values at type Bool—that is, they both return $\mathsf{true}_{\mathsf{Id}}$ or $\mathsf{false}_{\mathsf{Id}}$.

In this last case, both sides step: to $v_{1\{x:T|e_1\}}'?$ and $v_{2\{x:T|e_2\}}'?$ (by G_CheckOK or F_CheckOK and then F_TagId), or to fail (by G_CheckFail on the left and by F_CheckFail and F_CoerceRaise on the right). In either case, we may or may not have enough steps left in our index. Either way we are done: if we do have enough steps, we have found a derivation to related values. If not, we are done trivially, as determinism (Lemma 4.2.1) guarantees that this is the only evaluation derivation.

□

4.4.3 Theorem [Soundness]: 1. If $\Gamma \vdash u : T$ then $\Gamma \vdash u_{\mathsf{Id}} \simeq \mathsf{coerce}(u_{\mathsf{Id}}) : T$.

2. If $\Gamma \vdash v : T$ then $\Gamma \vdash v \simeq \mathsf{coerce}(v) : T$.

3. If $\Gamma \vdash e : T$ then $\Gamma \vdash e \simeq \mathsf{coerce}(e) : T$.

4. If $\vdash T_1$ and $\vdash T_2$ and $\vdash T_1 \parallel T_2$ then $\vdash T_1 \Rightarrow T_2 \equiv \mathsf{coerce}(T_1, T_2)$.

5. If $\vdash T$ then $\forall j.\ T \sim^j \mathsf{coerce}(T)$.

Proof: By simultaneous induction on the typing/well formedness/similarity derivations.

(T_Const) Immediate, by definition: $k \sim^j k : B$ for all j.

(T_Abs) We must show that $\Gamma \vdash (\lambda x{:}T_1.\ e)_{\mathsf{Id}} \simeq (\lambda x{:}\mathsf{coerce}(T_1).\ \mathsf{coerce}(e))_{\mathsf{Id}} : T_1 \to T_2$, given that $\Gamma, x{:}T_1 \vdash e \simeq \mathsf{coerce}(e) : T_2$ by IH (3).

Let j be given such that $\Gamma \models^j \delta$. Unfolding some definitions, we must show that $\delta_1(\lambda x{:}T_1.\ e)_{\mathsf{Id}} \sim^j \delta_2(\lambda x{:}\mathsf{coerce}(T_1).\ \mathsf{coerce}(e))_{\mathsf{Id}} : T_1 \to T_2$. Let $m < j$ be given such that $v_1 \sim^m v_2 : T$. We must show that

$$(\lambda x{:}T_1.\ \delta_1(e))_{\mathsf{Id}}\ v_1 \simeq^m (\lambda x{:}\mathsf{coerce}(T_1).\ \delta_2(\mathsf{coerce}(e)))_{\mathsf{Id}}\ v_2 : T_2$$

Stepping each side, we see we must show that $\delta_1(e[v_1/x]) \simeq^{(m-1)} \delta_2(\mathsf{coerce}(e)[v_2/x]) : T_2$. We have $\Gamma \models^{(m-1)} \delta[v_1, v_2/x]$, so we are done by instantiating IH (3) at $m-1$.

(T_PreVal) By IH (1).

(T_TagB) We must show that $\Gamma \vdash v_{B!} \simeq \mathsf{coerce}(v)_{B!} : \mathsf{Dyn}$. Let j be given such that $\Gamma \models^j \delta$; we must show that $\delta_1(v_{B!}) \sim^j \delta_2(\mathsf{coerce}(v)_{B!}) : \mathsf{Dyn}$.

By IH (2), we know that $\Gamma \vdash v \simeq \mathsf{coerce}(v) : B$; instantiating this proposition with j and δ, we are done.

(T_TagFun) We must show that $\Gamma \vdash v_{\mathbf{Fun}!} \simeq \mathsf{coerce}(v)_{\mathbf{Fun}!} : \mathsf{Dyn}$. Let j be given such that $\Gamma \models^j \delta$; we must show that $\delta_1(v_{\mathbf{Fun}!}) \sim^j \delta_2(\mathsf{coerce}(v)_{\mathbf{Fun}!}) : \mathsf{Dyn}$.

By IH (2), we know that $\Gamma \vdash v \simeq \mathsf{coerce}(v) : \mathsf{Dyn} \to \mathsf{Dyn}$; instantiating this proposition with j and δ, we are done.

(T_TagRefine) We must show that $\Gamma \vdash v_{\{x:T|e\}?} \simeq \mathsf{coerce}(v)_{\{x:T|\mathsf{coerce}(e)\}?} : \{x{:}T \mid e\}$. Let j be given such that $\Gamma \models^j \delta$; we must show that

$$\delta_1(v_{\{x:T|e\}?}) \sim^j \delta_2(\mathsf{coerce}(v)_{\{x:T|\mathsf{coerce}(e)\}?}) : \{x{:}T \mid e\}$$

Let $m < j$ be given; we must show that $v \sim^m \mathsf{coerce}(v) : T$ and $\{x{:}T \mid e\} \sim^m \{x{:}T \mid \mathsf{coerce}(e)\}$. We find this by instantiating IH (2) and IH (5) at m and δ.

(T_Wrap) We must show that

$$\Gamma \vdash \mathsf{coerce}(v)_{\langle T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22} \rangle} \simeq \mathsf{coerce}(v)_{\mathsf{coerce}(T_{21}, T_{11}) \mapsto \mathsf{coerce}(T_{12}, T_{22})} : T_{21} \to T_{22}$$

Let j be given such that $\Gamma \models^j \delta$; we must show that

$$\delta_1(v)_{\langle T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22} \rangle} \simeq^j \delta_2(\mathsf{coerce}(v))_{\mathsf{coerce}(T_{21}, T_{11}) \mapsto \mathsf{coerce}(T_{12}, T_{22})} : T_{21} \to T_{22}$$

Let $m < j$ be given such that $v_1 \sim^m v_2 : T_{21}$. We must show (after stepping each side) that

$$\begin{array}{rl} \langle T_{12} \Rightarrow T_{22} \rangle\ (\delta_1(v)\ (\langle T_{21} \Rightarrow T_{11} \rangle\ v_1)) & \simeq^(\ m-1) \\ \langle \mathsf{coerce}(T_{12}, T_{22}) \rangle\ (\delta_2(\mathsf{coerce}(v))\ (\langle \mathsf{coerce}(T_{21}, T_{11}) \rangle\ v_2)) & : T_{22} \end{array}$$

By IH (4) and Lemma 4.4.2, we know that

$$\langle T_{21} \Rightarrow T_{11} \rangle\ v_1 \simeq^{j'} \langle \mathsf{coerce}(T_{21}, T_{11}) \rangle\ v_2 : T_{11}$$

for all j'. Instantiating this at $m-1$, we see that either: (a) the left-hand side diverges, and so therefore does the whole term; (b) both sides reduce to fail, and so therefore does the whole term; or (c) both sides reduce to $v_1' \sim^{(m-1-m')} v_2' : T_{11}$.

In all but the last case, we are done. In the last case, we know by the IH that $\Gamma \vdash v \simeq \mathsf{coerce}(v) : T_{11} \to T_{12}$, so we can instantiate this at $m-1-m'$ to find a similar set of cases. Again, we only need to consider the case where $v\ v_1'$ and $\mathsf{coerce}(v)\ v_2'$ reduce to values $v_1'' \sim^{(m-1-m'-m'')} v_2'' : T_{12}$ in m'' steps.

We now reason as in the domain cast, using IH (4) and Lemma 4.4.2 to finally conclude that $\langle T_{12} \Rightarrow T_{22} \rangle\ v_1'' \simeq^{(m-1-m'-m'')} \langle \mathsf{coerce}(T_{12}, T_{22}) \rangle\ v_2'' : T_{22}$.

(T_Var) If $\Gamma \vdash x \simeq x : T$, then $\delta_1(x) \simeq^j \delta_2(x) : T$ for all j by definition.

(T_Fail) We immediately have $\mathsf{fail} \simeq^j \mathsf{fail} : T$ for all j and T.

(T_Cast) We must show that $\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle\ e \simeq \langle \mathsf{coerce}(T_1, T_2) \rangle\ \mathsf{coerce}(e) : T_2$. Let j be given such that $\Gamma \models^j \delta$.

By IH (3), we know that $\Gamma \vdash e \simeq \mathsf{coerce}(e) : T_1$. We can instantiate this at j and δ to find that either: (a) the left side diverges, and so does the cast term; (b) both sides reduce to fail, and so do the cast/coercion terms in one more step; or (c) both sides reduce in m steps to $v_1 \sim^{(j-m)} v_2 : T_1$. By coercion congruence (Lemma 4.3.13), we know that $\langle \mathsf{coerce}(T_1, T_2) \rangle\ \delta_2(\mathsf{coerce}(e)) \longrightarrow_n^* \langle \mathsf{coerce}(T_1, T_2) \rangle\ v_2$.

By IH (4), we know that $\vdash T_1 \Rightarrow T_2 \equiv \mathsf{coerce}(T_1, T_2)$. By Lemma 4.4.2, we know that $\langle T_1 \Rightarrow T_2 \rangle\ v_1 \simeq^j \langle \mathsf{coerce}(T_1, T_2) \rangle\ v_2 : T_2$ for all j—and for $m < j$ in particular, so we are done.

(T_App) We must show that $\Gamma \vdash e_1\ e_2 \simeq \mathsf{coerce}(e_1)\ \mathsf{coerce}(e_2) : T_2$. Let j be given such that $\Gamma \models^j \delta$. We must show $\delta_1(e_1)\ \delta_1(e_2) \simeq^j \delta_2(\mathsf{coerce}(e_1))\ \delta_2(\mathsf{coerce}(e_2)) : T_2$.

By IH (3), we know that $\Gamma \vdash e_1 \simeq \mathsf{coerce}(e_1) : T_1 \to T_2$; by instantiating at j and δ, we have either: a left-side divergence (and are done); fail in $m < j$ steps on both sides (and are done in $m+1$ steps); or $v_1 \sim^{(j-m)} v_2 : T_1 \to T_2$.

Again by IH (3), $\Gamma \vdash e_2 \simeq \mathsf{coerce}(e_2) : T_1$. Instantiating at $j-m$, we again have three possibilities: left-side divergence, fail on both sides, or reduction to $v_1' \sim^{(j-m-m')} v_2' : T_1$. In the former two cases we are done. In the latter, we know by definition that $v_1\ v_1' \simeq^{(j-m-m')} v_2\ v_2' : T_2$, and we are done.

(T_Op) Using IH (3), we can reduce all of the arguments of the operation to values. Since operations are first-order and don't use dynamic types, we know that the arguments must all be base values or refinements of base values, so the related values are *equal* constants. The G_Op/F_Op will then produce identical outputs, which must also be related.

(T_CheckCast) We must show that

$$\Gamma \vdash \langle \{x{:}T \mid e_1\}, e_2, v\rangle \simeq \langle \{x{:}T \mid \mathsf{coerce}(e_1)\}, \mathsf{coerce}(e_2), \mathsf{coerce}(v)\rangle : \{x{:}T \mid e_1\}$$

Let j and $\Gamma \models^j \delta$ be given. By IH (3) instantiated at j, we know that $e_2 \simeq^j \mathsf{coerce}(e_2) : \mathsf{Bool}$. If this diverges, so does the whole term, and we are done. If both sides go to fail in m steps, we are done in $m+1$ steps. If both sides go to $v_1 \sim^{j-m} v_2 : \mathsf{Bool}$ in m steps, then there are two possibilities.

If $v_1 = v_2 = \mathsf{true}_{\mathsf{Id}}$, then both sides step by G_CheckOK/F_CheckOK to $\delta_1(v)_{\{x:T|e_1\}}?$ and $\delta_2(\mathsf{coerce}(v))_{\{x:T|\mathsf{coerce}(e_1)\}}?$. Let $m' < j - m - 1$ be given; We already have $\delta_1(v) \sim^{m'} \delta_2(\mathsf{coerce}(v)) : T$ by IH (2), and we are done by IH (5), finding:

$$\{x{:}T \mid e_1\} \sim^{m'} \{x{:}T \mid \mathsf{coerce}(e_1)\}$$

(P_Id) We have $\vdash T \Rightarrow T \equiv \mathsf{Id}$ by RC_Id immediately.

(P_DynL) We have $\vdash \mathsf{Dyn} \parallel T$. We go by cases on T:

($T = \mathsf{Dyn}$) Immediate, by RC_Id.

($T = B$) By RC_DynB.

($T = T_1 \to T_2$) We can find $\vdash \mathsf{Dyn}\to\mathsf{Dyn} \parallel T_1 \to T_2$ by an application of P_Fun and two applications of P_DynL. Then by IH (4), we have $\vdash \mathsf{Dyn}\to\mathsf{Dyn} \Rightarrow T_1 \to T_2 \equiv \mathsf{coerce}(\mathsf{Dyn}\to\mathsf{Dyn}, T_1 \to T_2)$, and then by RC_DynFun we have $\vdash \mathsf{Dyn} \Rightarrow T_1 \to T_2 \equiv \mathbf{Fun}?; \mathsf{coerce}(\mathsf{Dyn}\to\mathsf{Dyn}, T_1 \to T_2)$.

($T = \{x{:}T' \mid e\}$) We have $\vdash \mathsf{Dyn} \parallel T'$ by P_DynL immediately, so by IH (4) we have $\vdash \mathsf{Dyn} \Rightarrow T' \equiv \mathsf{coerce}(\mathsf{Dyn}, T')$. By IH (5), $\{x{:}T' \mid e\} \sim^j \{x{:}T' \mid \mathsf{coerce}(e)\}$ for all j; unfolding this definition, we can find $x{:}T' \vdash e \simeq \mathsf{coerce}(e) : \mathsf{Bool}$. So now we have $\vdash \mathsf{Dyn} \Rightarrow \{x{:}T' \mid e\} \equiv \mathsf{coerce}(\mathsf{Dyn}, T'); \{x{:}B \mid \mathsf{coerce}(e)\}?$ by RC_PreCheck.

(P_DynR) We have $\vdash T \parallel \mathsf{Dyn}$. We go by cases on T:

($T = \mathsf{Dyn}$) Immediate, by RC_Id.

($T = B$) By RC_BDyn.

($T = T_1 \to T_2$) We can find $\vdash T_1 \to T_2 \parallel \mathsf{Dyn}\to\mathsf{Dyn}$ by an application of P_Fun and two applications of P_DynR. Then by IH (4), we have $\vdash T_1 \to T_2 \Rightarrow \mathsf{Dyn}\to\mathsf{Dyn} \equiv \mathsf{coerce}(T_1 \to T_2, \mathsf{Dyn}\to\mathsf{Dyn})$, and then by RC_DynFun we have $\vdash T_1 \to T_2 \Rightarrow \mathsf{Dyn} \equiv \mathsf{coerce}(T_1 \to T_2, \mathsf{Dyn}\to\mathsf{Dyn}); \mathbf{Fun}!$.

($T = \{x{:}T' \mid e\}$) We have $\vdash T' \parallel \mathsf{Dyn}$ by P_DynL immediately, so by IH (4) we have $\vdash T' \Rightarrow \mathsf{Dyn} \equiv \mathsf{coerce}(T', \mathsf{Dyn})$. By IH (5), we can unfold $\{x{:}T' \mid e\} \sim^j \{x{:}T' \mid \mathsf{coerce}(e)\}$ into $x{:}T' \vdash e \simeq \mathsf{coerce}(e) : \mathsf{Bool}$. We are now done by RC_PredPred.

(P_Fun) We have $\vdash T_{11} \to T_{12} \parallel T_{21} \to T_{22}$. By IH (4), we find:

$$\vdash T_{21} \Rightarrow T_{11} \equiv \mathsf{coerce}(T_{21}, T_{11}) \vdash T_{12} \Rightarrow T_{22} \equiv \mathsf{coerce}(T_{12}, T_{22})$$

We find $\vdash T_{11} \to T_{12} \Rightarrow T_{21} \to T_{22} \equiv \mathsf{coerce}(T_{21}, T_{11}) \mapsto \mathsf{coerce}(T_{12}, T_{22})$ by RC_Fun.

(P_RefineL) We have $\vdash \{x{:}T_1 \mid e\} \parallel T_2$; by inversion, $\vdash T_1 \parallel T_2$. If $T_2 = \{x{:}T_1 \mid e\}$, then we are done by RC_Id. Otherwise, we use RC_PredPred and IHs (4) and (5).

(P_RefineR) We have $\vdash T_1 \parallel \{x{:}T_2 \mid e\}$; by inversion, $\vdash T_1 \parallel T_2$. If $T_1 = \{x{:}T_2 \mid e\}$, then we are done by RC_Id. Otherwise, we use RC_PreCheck and IHs (4) and (5).

(WF_Dyn) By definition.

(WF_Base) By definition.

(WF_Fun) By IH (5).

(WF_Refine) By IH (3) and rearranging quantifiers, recalling that e_1 (and so $\mathsf{coerce}(e_1)$) are both closed.

□

4.5 A space-efficient coercion calculus

Having developed the naïve semantics in Naive, I now turn to space efficiency. In this section, I define Efficient, a space-efficient coercion calculus.

There are two loci of inefficiency: coercion merges and function proxies (functional coercions). When F_Merge applies, it merely concatenates two coercions: $\langle c_1 \rangle\ (\langle c_2 \rangle\ e) \longrightarrow_n \langle c_2; c_1 \rangle\ e$. Efficient's semantics combines c_2 and c_1 to eliminate redundant checks. To bound the number of function proxies, I ensure that coercion merging combines adjacent functional coercions, and I change the tagging scheme on values—while Naive allows an arbitrary stack of tags values, Efficient will keep the size of value tags bounded. The solution to both of these problems lies in *canonical coercions* and the *merge* algorithm. Before I describe them below in Section 4.5.1, I discuss changes to the syntax of values and to the typing rules.

I make changes to both the syntax (Figure 4.18) and typing rules (Figure 4.19) of Naive from Section 4.3; I define an entirely new operational semantics for Efficient in Section 4.5.2 (see Figure 4.22). Changes are marked with highlighting and/or a bullet marker, •. The proof of type soundness of this development is in Section 4.5.3; it relies on my development of canonical coercions in Section 4.5.1. Throughout the new typing rules, I assume that coercions are canonical (see below).

Types

$$
\begin{array}{lcl}
T & ::= & B \mid T_1 \to T_2 \mid \{x{:}B \mid e\} \mid \mathsf{Dyn} \mid \{x{:}\mathsf{Dyn} \mid e\} \\
B & ::= & \mathsf{Bool} \mid \mathsf{Int} \mid \ldots
\end{array}
$$

Coercions, primitive coercions, and type tags

$$
\begin{array}{lcl}
\bullet c & ::= & d_1; \ldots; d_n \mid \mathsf{Fail} \\
\bullet d & ::= & D! \mid D? \mid c_1 \mapsto c_2 \\
D & ::= & B \mid \mathbf{Fun} \mid \{x{:}B \mid e\} \mid \{x{:}\mathsf{Dyn} \mid e\}
\end{array}
$$

Terms, values, pre-values, and results

$$
\begin{array}{lcl}
e & ::= & x \mid r \mid op(e_1, \ldots, e_n) \mid e_1\ e_2 \mid \langle c \rangle\ e \mid \\
 & & \langle \{x{:}T \mid e_1\}, e_2, v \rangle \\
r & ::= & v \mid \mathsf{fail} \\
\bullet v & ::= & u_c \\
u & ::= & k \mid \lambda x{:}T.\ e
\end{array}
$$

Typing contexts

$$
\Gamma ::= \emptyset \mid \Gamma, x{:}T
$$

Figure 4.18: Updated syntax for EFFICIENT

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

$$\frac{c \neq \mathsf{Fail} \quad c \neq c'; \{x{:}T \mid e\}? \quad c \text{ is canonical} \quad \Gamma \vdash u : T_1 \quad \vdash c : T_1 \rightsquigarrow T_2}{\Gamma \vdash u_c : T_2}\ \text{T\_VAL}$$

$$\frac{\emptyset \vdash u_c : T \quad \vdash \{x{:}T \mid e\}? : T \rightsquigarrow \{x{:}T \mid e\} \quad c; \{x{:}T \mid e\}? \text{ is canonical} \quad \vdash \Gamma \quad e[u_c/x] \longrightarrow^* \mathsf{true}_{\mathsf{Id}}}{\Gamma \vdash u_{c;\{x:T \mid e\}?} : \{x{:}T \mid e\}}\ \text{T\_VALREFINE}$$

$$\frac{\vdash \Gamma \quad \vdash \{x{:}T \mid e_1\} \quad \emptyset \vdash v : T \quad \emptyset \vdash e_2 : \mathsf{Bool} \quad e_1[v/x] \longrightarrow^* e_2}{\Gamma \vdash \langle \{x{:}T \mid e_1\}, e_2, v \rangle : \{x{:}T \mid e_1\}}\ \text{T\_CHECK}$$

$$\frac{\vdash c : T_1 \rightsquigarrow T_2 \qquad \Gamma \vdash e : T_1 \qquad c \text{ is canonical}}{\Gamma \vdash \langle c \rangle\ e : T_2}\ \text{T\_COERCE}$$

Figure 4.19: Updated typing rules for EFFICIENT

In NAIVE, tagging is stacked: a value is either a pre-value tagged with Id or a value tagged with a single primitive coercion. In EFFICIENT, I collapse this stack: values are pre-values tagged with a composite coercion, u_c. While NAIVE typed values with stacks of coercions using T_PREVAL, T_TAGVAL, and T_TAGVALREFINE, we now use rules T_VAL and T_VALREFINE to type values with a single, composite coercion on them.

I also change the structure of coercions slightly, treating Fail as a composite coercion. I do this because coercion normalization doesn't allow composite coercions with Fail at the top level—such coercions are normalized to just Fail itself.

The changes to the typing rules aren't major: T_VAL and T_VALREFINE account for flattened values: T_VAL will apply to u_c unless the coercion c ends in $\{x{:}T \mid e\}?$, in which case the typing derivation for the value will be T_VALREFINE around T_VAL. I separate the two rules to make sure we have value inversion, as outlined in my philosophy (Section 4.1). That is, I want to ensure that if a value has a refinement check tag on it, it satisfies that refinement. The T_CHECK rule changes to use the space-efficient semantics, but it remains a technical rule for supporting the evaluation of programs. Finally, I change T_COERCE to require that coercions appearing in the program source are *canonical*. Before discussing the new evaluation rules, I discuss my space-efficient coercions and what I mean by a canonical coercion.

4.5.1 Space-efficient coercions

I define a set of *canonical coercions*, further subdivided into value coercions for constants and for functions. I list these coercions in Table 4.2 below; I prove that they are in fact *the* normal coercions for a standard set of rewrite rules given in Figure 4.20 in Lemma 4.5.5 and Lemma 4.5.7. Next, I define a set of rules for merging coercions, proving that merging two well typed canonical coercions yields a well typed canonical coercion—no bigger than the previous two combined. This is how I show space efficiency: where a naïve implementation would accumulate and discharge all checks, EFFICIENT will keep its coercions in canonical form for which we have a bounded size (see Table 4.2).

Henglein and Herman et al. define coercions with slightly different structure: for me, Id is notation for the empty composite coercion, but for them it is a coercion in its own right. Henglein and Herman et al.'s systems work by taking a single rewrite rule, the so-called ϕ rule:

$$D!; D? \longrightarrow \mathsf{Id} \qquad (\phi)$$

The ϕ rule stands in distinction to the ψ rule:

$$D?; D! \longrightarrow \mathsf{Id} \qquad (\psi)$$

Henglein observed that for ϕ and ψ rules, the right-hand side is "more efficient". The ϕ rules are *efficient* and *harmless*: nothing can fail (if a value is already a D-typed value, then tagging as D into Dyn and then immediately untagging can't fail, so you

$\boxed{c \longrightarrow c}$

$$
\begin{array}{rlll}
\mathsf{Fail}; c & \longrightarrow \mathsf{Fail} & (\text{F\_FAIL}) & \\
c; \mathsf{Fail} & \longrightarrow \mathsf{Fail} & & \\
(c_{11} \mapsto c_{12}); (c_{21} \mapsto c_{22}) & \longrightarrow (c_{21}; c_{11}) \mapsto (c_{12}; c_{22}) & & \\
B!; B? & \longrightarrow \mathsf{Id} & (\text{F\_TAGBB}) & (\phi) \\
B!; B'? & \longrightarrow \mathsf{Fail} \text{ when } B \neq B' & & \\
 & & (\text{F\_TAGBFAILB}) & \\
\mathbf{Fun}!; \mathbf{Fun}? & \longrightarrow \mathsf{Id} & (\text{F\_TAGFUNFUN}) & (\phi) \\
B!; \mathbf{Fun}? & \longrightarrow \mathsf{Fail} & (\text{F\_TAGBFAILFUN}) & \\
\mathbf{Fun}!; B? & \longrightarrow \mathsf{Fail} & (\text{F\_TAGFUNFAILB}) & \\
\{x{:}T \mid e\}?; \{x{:}T \mid e\}! & \longrightarrow \mathsf{Id} & (\text{F\_TAGPREDPRED}) & (\psi) \\
\{x{:}T \mid e\}!; \{x{:}T \mid e\}? & \longrightarrow \mathsf{Id} & & (\phi)
\end{array}
$$

$$\frac{d_{i-1}; d_i \longrightarrow c}{d_1; \ldots; d_{i-1}; d_i; \ldots; d_n \longrightarrow d_1; \ldots; c; \ldots; d_n} \quad \textsc{Compat}$$

$$\frac{c_1 \longrightarrow c_1'}{d_1; \ldots; (c_1 \mapsto c_2); \ldots; d_n \longrightarrow d_1; \ldots; (c_1' \mapsto c_2); \ldots; d_n} \quad \textsc{FunDom}$$

$$\frac{c_2 \longrightarrow c_2'}{d_1; \ldots; (c_1 \mapsto c_2); \ldots; d_n \longrightarrow d_1; \ldots; (c_1 \mapsto c_2'); \ldots; d_n} \quad \textsc{FunCod}$$

Figure 4.20: Coercion rewriting rules

might as well just run Id). But the right-hand side of a ψ rule is "safer"—the left-hand side of a ψ rule may fail at runtime, but the right-hand side won't. Checking that a value of type Dyn holds a D-typed value and then immediately retagging it might fail, since the value may actually have a different type. Put another way, ϕ rules don't affect the extensional behavior of the program, just the (intensional) cost of running it; ψ rules affect extensional behavior, possibly hiding errors.

Henglein and Herman et al. define term rewriting systems with ϕ rules modulo an equational theory obtained by completing the following rules with reflexivity, symmetry, transitivity, and compatibility:

$$\begin{array}{rcl} \mathsf{Id}; c & = & c \\ c; \mathsf{Id} & = & c \\ c_{11} \mapsto c_{12}; c_{21} \mapsto c_{22} & = & (c_{21}; c_{11}) \mapsto (c_{12}; c_{22}) \\ (c_1; c_2); c_3 & = & c_1; (c_2; c_3) \end{array}$$

My system has ϕ rules for dynamic types and for manifest contracts, but it also has a ψ rule for manifest contracts. To make the technicalities are simpler, I directly define a rewrite system in Figure 4.20, where I lift the two-coercion rewrite rules to composite coercions with the rules Compat, FunDom, and FunCod. Rules that are instances of ϕ or ψ rules are marked as such. When reading the rules, recall that $c = \mathsf{Id}; c = c; \mathsf{Id}$. I expand out the ϕ rule for clarity here (and to ease some technicalities below).

Note that many of these rules correspond to reduction rules in Naive's operational semantics; I've written the reduction rule names next to such rewrite rules. The rules for predicates over base types and type dynamic are new. Just as $B?$ takes a less specific type, Dyn, to a more specific type B while performing a check, we have $\{x{:}B \mid e\}?$ take a less specific type, B to a more specific type $\{x{:}B \mid e\}$ while performing a check. That is, $\mathbf{src}(D)$ is more specific than $\mathbf{tgt}(D)$. By the same analogy, $\{x{:}B \mid e\}!$ takes a more specific type to a less specific one. The rules for refinements don't have just a ϕ rule—they must have a ψ rule, too:

$$\{x{:}T \mid e\}?; \{x{:}T \mid e\}! \longrightarrow \mathsf{Id}$$

Note that this is a *syntactic* equivalence on refinement types and their predicate terms. This may lead to some implementation issues; see Chapter 6.

To see why we need a ψ rule, consider the rule F_TagPredPred from the naïve operational semantics (Figure 4.12): we must have a ψ rule if $\{x{:}T \mid e\}!$ is going to untag $v_{\{x:T|e\}?}$. But if $\{x{:}T \mid e\}?; \{x{:}T \mid e\}! \longrightarrow \mathsf{Id}$ on tags, it must also hold for coercions on the stack, if we want space efficiency. That is, the coercion

$$\{x{:}T \mid e_1\}?; \{x{:}T \mid e_1\}!; \{x{:}T \mid e_2\}?; \{x{:}T \mid e_2\}!; \ldots; \{x{:}T \mid e_n\}?; \{x{:}T \mid e_n\}!$$

will accumulate checks unless each pair of e_i can be annihilated.

Space-efficient refinement checking *must* drop some checks on the floor. The ϕ rule for refinements is an optimization, unnecessary for soundness and space efficiency. Removing this rule would add one extra canonical coercion. Suppose

$\{x{:}T \mid e\}!;\{x{:}T \mid e\}?; c$ was canonical. What can c be that is (a) well typed, and (b) doesn't reduce? There is no such coercion; the only new canonical coercion would be $\{x{:}T \mid e\}!;\{x{:}T \mid e\}?$. Dropping the ϕ rule doesn't affect soundness, either. In fact, having it complicates matters a tiny bit:

$$\begin{array}{ll}
 & \langle\{x{:}T \mid e\}?\rangle\ (\langle\{x{:}T \mid e\}!\rangle\ v_{\{x:T|e\}?}) \\
\longrightarrow_n & \langle\{x{:}T \mid e\}!;\{x{:}T \mid e\}?\rangle\ v_{\{x:T|e\}?} \\
\longrightarrow_n & \langle\{x{:}T \mid e\}?\rangle\ v \\
\longrightarrow_n & \langle \mathsf{Id}\rangle\ \langle\{x{:}T \mid e\}, e[v/x], v\rangle \\
\longrightarrow_n^* & \langle \mathsf{Id}\rangle\ \langle\{x{:}T \mid e\}, \mathsf{true}_{\mathsf{Id}}, v\rangle \quad \text{(by typing of } v_{\{x:T|e\}?} \\
\longrightarrow_n & \langle \mathsf{Id}\rangle\ v_{\{x:T|e\}?} \\
\longrightarrow_n & v_{\{x:T|e\}?}
\end{array}$$

And in the space-efficient calculus with N_PHI for $D = \{x{:}T \mid e\}$, the merge equivalent of the ϕ rule:

$$\begin{array}{ll}
 & \langle\{x{:}T \mid e\}?\rangle\ (\langle\{x{:}T \mid e\}!\rangle\ u_{c;\{x:T|e\}?}) \\
\longrightarrow & \langle\{x{:}T \mid e\}!;\{x{:}T \mid e\}?\rangle\ u_{c;\{x:T|e\}?} \\
\longrightarrow & \langle \mathsf{Id}\rangle\ u_{c;\{x:T|e\}?} \\
\longrightarrow & u_{c;\{x:T|e\}?}
\end{array}$$

And without the ϕ rule:

$$\begin{array}{ll}
 & \langle\{x{:}T \mid e\}?\rangle\ (\langle\{x{:}T \mid e\}!\rangle\ u_{c;\{x:T|e\}?}) \\
\longrightarrow & \langle\{x{:}T \mid e\}!;\{x{:}T \mid e\}?\rangle\ u_{c;\{x:T|e\}?} \\
\longrightarrow & \langle\{x{:}T \mid e\}?\rangle\ u_c \\
\longrightarrow & \langle \mathsf{Id}\rangle\ \langle\{x{:}T \mid e\}, e[u_c/x], u_c\rangle \\
\longrightarrow^* & \langle \mathsf{Id}\rangle\ \langle\{x{:}T \mid e\}, e[u_c/x], u_c\rangle \quad \text{(by typing of } u_{c;\{x:T|e\}?} \\
\longrightarrow & \langle \mathsf{Id}\rangle\ u_{c;\{x:T|e\}?} \\
\longrightarrow & u_{c;\{x:T|e\}?}
\end{array}$$

The evaluation derivation is closer to that of the naive system without the ϕ rule for refinements—including the ϕ rule merely skips rechecking the predicate. I do not do so here, but I conjecture that EFFICIENT terms are behaviorally equivalent with or without the ϕ rule, though the runtime costs will be different.

Finally, one may wonder: we have ψ rule for refinements, but no such rule for tagging into and checking from Dyn. Why not? Henglein pointed out that ψ rules don't preserve the behavior of the original program exactly, possibly hiding errors. It turns out that I *must* have a ψ rule to make manifest contracts space efficient. But gradual types don't need the ψ rule to be space efficient. Since I'd like the behavior of NAIVE and EFFICIENT to be as close as possible, I minimize the mismatch by only using ψ rules for manifest contracts, not dynamic contracts—even if it doesn't give us a stronger theorem.

The situation is neatly dual: manifest contracts must have ψ rules, but need not have ϕ rules; dynamic types must have ϕ rules, but need not have ψ rules.

4.5.1 Lemma [Preservation for rewriting]: If $\vdash c : T_1 \rightsquigarrow T_2$ and $c \longrightarrow c'$ then $\vdash c' : T_1 \rightsquigarrow T_2$.

Proof: By case analysis on the reduction step taken. □

4.5.2 Lemma [Confluence]: Given a well typed coercion c_1, if $c_1 \longrightarrow c_2$ and $c_1 \longrightarrow c_2'$ then $c_2 \longrightarrow^* c_3$ and $c_2' \longrightarrow^* c_3$.

Proof: By case analysis on the critical pairs. If the reductions take place in non-overlapping parts of the term, then we simply make the appropriate, other, reduction step in both terms.

If the reduction takes place in an overlapping critical pair, we must reason more carefully. If one of the steps is any of the Id or Fail cases, we are done: simply take the appropriate step in the other term.

The remaining critical pairs are multiple functional coercions in a row and overlapping refinement coercions.

Suppose we have $c_1 = c; (c_{11} \mapsto c_{12}); (c_{21} \mapsto c_{22}); (c_{31} \mapsto c_{32}); c'$ and

$$\begin{array}{rcl} c_1 & \longrightarrow & c_2 = c; ((c_{21}; c_{11}) \mapsto (c_{12}; c_{22})); (c_{31} \mapsto c_{32}); c' \\ c_1 & \longrightarrow & c_2' = c; (c_{11} \mapsto c_{12}); ((c_{31}; c_{21}) \mapsto (c_{22}; c_{32})); c'. \end{array}$$

We can simply reduce each side once more to find $c; (c_{31}; c_{21}; c_{11}) \mapsto (c_{12}; c_{22}; c_{32}); c'$ on both sides.

Suppose we have $c_1 = c; \{x{:}T \mid e\}?; \{x{:}T \mid e\}!; \{x{:}T \mid e\}?; c'$ and

$$\begin{array}{rcl} c_1 & \longrightarrow & c_2 = c; \mathsf{Id}; \{x{:}T \mid e\}?; c' \\ c_1 & \longrightarrow & c_2' = c; \{x{:}T \mid e\}?; \mathsf{Id}; c'. \end{array}$$

Note that the refinements must be the same to have a well typed critical pair where the either reduction rule could fire. In this case, we can simply reduce the Id on either side to find confluence. The case for $c_1 = c; \{x{:}T \mid e\}!; \{x{:}T \mid e\}?; \{x{:}T \mid e\}!; c'$ is symmetric.

□

4.5.3 Lemma: The rewrite system of Figure 4.20 is normalizing.

Proof: We define coercion size straightforwardly, as follows:

$$\begin{array}{rcl} \mathrm{size}(\mathsf{Id}) = \mathrm{size}(\mathsf{Fail}) & = & 1 \\ \mathrm{size}(D!) = \mathrm{size}(D?) & = & 1 \\ \mathrm{size}(d_1; \ldots; d_n) & = & \sum \mathrm{size}(d_i) \\ \mathrm{size}(c_1 \mapsto c_2) & = & 1 + \mathrm{size}(c_1) + \mathrm{size}(c_2) \end{array}$$

We show that if $c_1 \longrightarrow c_2$, then $\mathrm{size}(c_1) > \mathrm{size}(c_2)$ (by inspection). The only subtle case is $(c_{11} \mapsto c_{12}); (c_{21} \mapsto c_{22}) \longrightarrow (c_{21}; c_{11}) \mapsto (c_{12}; c_{22})$, wherein the size reduces by exactly 1, by eliminating one of the functional coercion constructors. □

4.5.4 Lemma: The rewrite system of Figure 4.20 is strongly normalizing on well typed coercions.

Proof: This follows directly from the fact that the rewrite system is normalizing (Lemma 4.5.3) and confluent (Lemma 4.5.2). □

4.5.5 Lemma: The canonical coercions are normal.

Proof: By inspection. □

4.5.6 Lemma: If $d_1; \ldots; d_n$ is canonical, then any prefix $d_1; \ldots$ or suffix $\ldots; d_n$ is also canonical.

Proof: By inspection of Figure 4.11. □

4.5.7 Lemma: If $\vdash c : T_1 \rightsquigarrow T_2$ and c is normal, then c is canonical.

Proof: By induction on the derivation of $\vdash c : T_1 \rightsquigarrow T_2$. Consulting Table 4.2 will speed up the case analysis, since the possible typings quickly circumscribe the possibilities.

(C_Id) Id is canonical.

(C_Fail) Fail is canonical.

(C_Compose) We have $\vdash d; c : T_1 \rightsquigarrow T_2$; by inversion we have $\vdash d : T_1 \rightsquigarrow T'$ and $\vdash c : T' \rightsquigarrow T_2$. Moreover, if $d; c$ is normal, then so is c—so by the IH, we know that c is canonical.

We will consider various possibilities for c, but in all cases we may rule out Id—because each of the primitive coercions is canonical—and Fail—because $d; \mathsf{Fail}$ is not normal. So in the following cases, we are looking at a coercion $d; c$ where c is non-empty. We go by cases on the former derivation:

(C_Untag) By cases on d:

($d = B?$) It must be the case that $T' = B$, so c is one of $B!$ or $B!; \{x{:}\mathsf{Dyn} \mid e\}?$ or $\{x{:}B \mid e\}?$. In all cases, we have immediately that $d; c$ is canonical.

($d = \mathbf{Fun}?$) It must be the case that $T' = (\mathsf{Dyn} \to \mathsf{Dyn})$, so c is one of $\mathbf{Fun}!$ or $\mathbf{Fun}!; \{x{:}\mathsf{Dyn} \mid e\}?$ or one of the $(c_1 \mapsto c_2); c'$ coercions. In all cases, $d; c$ is canonical.

($d = \{x{:}T \mid e\}?$) The only possibility is that $c = \{x{:}T \mid e\}!; c'$, but this would not be normal—a contradiction. (That is, the only canonical coercion beginning $\{x{:}T \mid e\}?$ is exactly $\{x{:}T \mid e\}?$.)

(C_Tag) By cases on d:

($d = B!$) It must be the case that $T' = \mathsf{Dyn}$. It cannot be the case that $c = \mathbf{Fun}?; c'$, for then $d; c$ wouldn't be normal. By the same token, we can't have that $c = B?; c'$. The only remaining possibility is that $c = \{x{:}\mathsf{Dyn} \mid e\}?$, and $B!; \{x{:}\mathsf{Dyn} \mid e\}?$ is canonical.

$\boxed{c_1 * c_2 \Rightarrow c_3}$

$$\frac{c_1; c_2 \text{ is canonical}}{c_1 * c_2 \Rightarrow (c_1; c_2)} \quad \text{N\_Canonical}$$

$$\frac{c_{21} * c_{11} \Rightarrow c_{31} \quad c_{12} * c_{22} \Rightarrow c_{32} \quad c_1 * (c_{31} \mapsto c_{32}); c_2 \Rightarrow c}{c_1; (c_{11} \mapsto c_{12}) * (c_{21} \mapsto c_{22}); c_2 \Rightarrow c} \quad \text{N\_Fun}$$

$$\frac{}{\mathsf{Fail} * c \Rightarrow \mathsf{Fail}} \quad \text{N\_FailL} \qquad \frac{c \neq \mathsf{Fail}}{c * \mathsf{Fail} \Rightarrow \mathsf{Fail}} \quad \text{N\_FailR} \qquad \frac{c_1 * c_2 \Rightarrow c}{c_1; D! * D?; c_2 \Rightarrow c} \quad \text{N\_Phi}$$

$$\frac{B \neq B'}{c_1; B! * B'?; c_2 \Rightarrow \mathsf{Fail}} \quad \text{N\_BFailB} \qquad \frac{}{c_1; B! * \mathbf{Fun}?; c_2 \Rightarrow c} \quad \text{N\_BFailFun}$$

$$\frac{}{c_1; \mathbf{Fun}! * B?; c_2 \Rightarrow \mathsf{Fail}} \quad \text{N\_FunFailB} \qquad \frac{c_1 * c_2 \Rightarrow c}{c_1; \{x{:}T \mid e\}? * \{x{:}T \mid e\}!; c_2 \Rightarrow c} \quad \text{N\_PredPred}$$

Figure 4.21: Merging coercions

($d = \mathbf{Fun}!$) It must be the case that $T' = \mathsf{Dyn}$. It cannot be the case that $c = \mathbf{Fun}?; c'$ or $c = B!; c'$, because then $d; c$ wouldn't be normal. We can therefore conclude that $c = \{x{:}\mathsf{Dyn} \mid e\}?$, and $\mathbf{Fun}!; \{x{:}\mathsf{Dyn} \mid e\}?$ is canonical.

($d = \{x{:}B \mid e\}!$) c is either $B!$ or $B!; \{x{:}\mathsf{Dyn} \mid e'\}?$ or $\{x{:}B \mid e'\}?$. For the first two, we have that $d; c$ is canonical. In the last case, $d; c$ is normal iff $e \neq e'$, in which case $d; c$ is canonical.

($d = \{x{:}\mathsf{Dyn} \mid e\}!$) c must come from Dyn. If it is $\{x{:}\mathsf{Dyn} \mid e'\}?$, we must have $e \neq e'$ for $d; c$ to be normal, but then $d; c$ is canonical. If $c = B?; c'$, then $\{x{:}\mathsf{Dyn} \mid e\}!; B?; c'$ is canonical. The same is true when $c = \mathbf{Fun}?; c'$.

(C_Fun) $d = c_1 \mapsto c_2$. It must be the case that c_1 and c_2 are normal, so by the IH they are also canonical. It can't be the case that $c = (c_1' \mapsto c_2'); c'$—then $d; c$ wouldn't be normal.

So the only possibility is that (epending on the types of c_1 and c_2) the coercion c could be $\mathbf{Fun}!$ or $\mathbf{Fun}!; \{x{:}\mathsf{Dyn} \mid e\}?$. In both of these cases, $d; c$ is canonical.

□

4.5.8 Corollary: All well typed coercions rewrite to a canonical coercion.

Proof: By strong normalization (Lemma 4.5.4), preservation (Lemma 4.5.1), and the fact well typed normal coercions are canonical (Lemma 4.5.7). □

Having developed the rewrite rules for my (somewhat relaxed) coercions, I define a merge algorithm in Figure 4.21 that takes two coercions and merges them from the

"inside out": when merging $d_{11}; \ldots; d_{1\,n}$ and $d_{21}; \ldots; d_{2\,i}$, we first merge $d_{1\,n}$ and d_{21}, and then $d_{1\,n-1}$ and d_{22}, and so on. We will only ever merge *canonical* coercions, greatly simplifying the algorithm. We are justified in doing this by Corollary 4.5.8—every coercion reduces (i.e., is equivalent to) some normalized, canonical form. My merging relation implements the ϕ rule for all tags in N_PHI; since we have ψ rule only for refinements, we implement it separately in N_PREDPRED. We implement failure rules in N_BFAILB, N_BFAILFUN, and N_FUNFAILB. I relate my coercion system to others in Chapter 5.

Looking at Table 4.2, we can see why T_VALREFINE only needs to check the last coercion on a tagged pre-value: if $\{x{:}T \mid e\}?$ appears in a canonical coercion, it appears at the end. Similarly, the observation that certain coercions are *value coercions*, i.e., are the only coercions that can be applied to values, can be made based on typing: if constants are typed at simple types and lambdas are assigned functional types, then all value coercions must come from B or $T_1 \to T_2$.

I write $c_1 \Downarrow c_2$ (read "merge c_1 and c_2") for canonical coercions c_1 and c_2 to mean the coercion c such that $c_1 * c_2 \Rightarrow c$. This notation is justified by Lemma 4.5.11, which shows that merging is an operator on canonical coercions.

4.5.9 Lemma [Preservation for merge]: If $\vdash c_1 : T_1 \rightsquigarrow T_2$ and $\vdash c_2 : T_2 \rightsquigarrow T_3$ and $c_1 * c_2 \Rightarrow c_3$ then $\vdash c_3 : T_1 \rightsquigarrow T_3$.

Proof: By induction on the derivation of $c_1 * c_2 \Rightarrow c_3$.

(N_CANONICAL) By induction on the length of c_1, using C_COMPOSE.

(N_FAILL) We have $\mathsf{Fail} * c_2 \Rightarrow \mathsf{Fail}$. By regularity (Lemma 4.3.2), $\vdash T_3$. By inversion we have $\vdash T_1$, so by C_FAIL we have $\vdash \mathsf{Fail} : T_1 \rightsquigarrow T_3$.

(N_FAILR) We have $c_1 * \mathsf{Fail} \Rightarrow \mathsf{Fail}$. By regularity (Lemma 4.3.2), $\vdash T_1$. By inversion we have $\vdash T_3$, so by C_FAIL we have $\vdash \mathsf{Fail} : T_1 \rightsquigarrow T_3$.

(N_PHI) We have $\vdash c_1; D! : T_1 \rightsquigarrow \mathbf{tgt}(D)$ and $\vdash D?; c_2 : \mathbf{tgt}(D) \rightsquigarrow T_3$. By inversion, $\vdash c_1 : T_1 \rightsquigarrow \mathbf{src}(D)$ and $\vdash c_2 : \mathbf{src}(D) \rightsquigarrow T_3$. We can now apply the IH to find that $\vdash c : T_1 \rightsquigarrow T_3$.

(N_BFAILB) We have $\vdash c_1; B! : T_1 \rightsquigarrow \mathsf{Dyn}$ and $\vdash B'?; c_2 : \mathsf{Dyn} \rightsquigarrow T_3$. By regularity (Lemma 4.3.2), we know that $\vdash T_1$ and $\vdash T_3$. So we can conclude by C_FAIL that $\vdash \mathsf{Fail} : T_1 \rightsquigarrow T_3$.

(N_BFAILFUN) We have $\vdash c_1; B! : T_1 \rightsquigarrow \mathsf{Dyn}$ and $\vdash \mathbf{Fun}?; c_2 : \mathsf{Dyn} \rightsquigarrow T_3$. By regularity (Lemma 4.3.2), we know that $\vdash T_1$ and $\vdash T_3$. So we can conclude by C_FAIL that $\vdash \mathsf{Fail} : T_1 \rightsquigarrow T_3$.

(N_FUNFAILB) We have $\vdash c_1; \mathbf{Fun}! : T_1 \rightsquigarrow \mathsf{Dyn}$ and $\vdash B?; c_2 : \mathsf{Dyn} \rightsquigarrow T_3$. By regularity (Lemma 4.3.2), we know that $\vdash T_1$ and $\vdash T_3$. So we can conclude by C_FAIL that $\vdash \mathsf{Fail} : T_1 \rightsquigarrow T_3$.

Coercion		Type		
Id	:	T	$\rightsquigarrow$	T
Fail	:	T	$\rightsquigarrow$	T'
$\{x{:}\mathsf{Dyn} \mid e\}?$	:	Dyn	$\rightsquigarrow$	$\{x{:}\mathsf{Dyn} \mid e\}$
$B?$	:	Dyn	$\rightsquigarrow$	B
$B?; B!$	:	Dyn	$\rightsquigarrow$	Dyn
$B?; B!; \{x{:}\mathsf{Dyn} \mid e\}?$	:	Dyn	$\rightsquigarrow$	$\{x{:}\mathsf{Dyn} \mid e\}$
$B?; \{x{:}B \mid e\}?$	:	Dyn	$\rightsquigarrow$	$\{x{:}B \mid e\}$
$\mathbf{Fun}?$	:	Dyn	$\rightsquigarrow$	$\mathsf{Dyn}{\to}\mathsf{Dyn}$
$\mathbf{Fun}?; \mathbf{Fun}!$	:	Dyn	$\rightsquigarrow$	Dyn
$\mathbf{Fun}?; \mathbf{Fun}!; \{x{:}\mathsf{Dyn} \mid e\}?$	:	Dyn	$\rightsquigarrow$	$\{x{:}\mathsf{Dyn} \mid e\}$
$\mathbf{Fun}?; c_1 \mapsto c_2$	:	Dyn	$\rightsquigarrow$	$T_{21}{\to}T_{22}$
$\mathbf{Fun}?; c_1 \mapsto c_2; \mathbf{Fun}!$	:	Dyn	$\rightsquigarrow$	Dyn
$\mathbf{Fun}?; c_1 \mapsto c_2; \mathbf{Fun}!; \{x{:}\mathsf{Dyn} \mid e\}?$	:	Dyn	$\rightsquigarrow$	$\{x{:}\mathsf{Dyn} \mid e\}$
$B!$	:	B	$\rightsquigarrow$	Dyn
$B!; \{x{:}\mathsf{Dyn} \mid e\}?$	:	B	$\rightsquigarrow$	$\{x{:}\mathsf{Dyn} \mid e\}$
$\{x{:}B \mid e\}?$	:	B	$\rightsquigarrow$	$\{x{:}B \mid e\}$
$\mathbf{Fun}!$	:	$(\mathsf{Dyn}{\to}\mathsf{Dyn})$	$\rightsquigarrow$	Dyn
$\mathbf{Fun}!; \{x{:}\mathsf{Dyn} \mid e\}?$	:	$(\mathsf{Dyn}{\to}\mathsf{Dyn})$	$\rightsquigarrow$	$\{x{:}\mathsf{Dyn} \mid e\}$
$c_1 \mapsto c_2$	:	$(T_{11}{\to}T_{12})$	$\rightsquigarrow$	$(T_{21}{\to}T_{22})$
$c_1 \mapsto c_2; \mathbf{Fun}!$	:	$(T_{11}{\to}T_{12})$	$\rightsquigarrow$	Dyn
$c_1 \mapsto c_2; \mathbf{Fun}!; \{x{:}\mathsf{Dyn} \mid e\}?$	:	$(T_{11}{\to}T_{12})$	$\rightsquigarrow$	$\{x{:}\mathsf{Dyn} \mid e\}$
$\{x{:}\mathsf{Dyn} \mid e\}!$	:	$\{x{:}\mathsf{Dyn} \mid e\}$	$\rightsquigarrow$	Dyn
$\{x{:}\mathsf{Dyn} \mid e\}!; \{x{:}\mathsf{Dyn} \mid e'\}?$	:	$\{x{:}\mathsf{Dyn} \mid e\}$	$\rightsquigarrow$	Dyn where $e \neq e'$
$\{x{:}\mathsf{Dyn} \mid e\}!; B?$	:	$\{x{:}\mathsf{Dyn} \mid e\}$	$\rightsquigarrow$	B
$\{x{:}\mathsf{Dyn} \mid e\}!; B?; B!$	:	$\{x{:}\mathsf{Dyn} \mid e\}$	$\rightsquigarrow$	Dyn
$\{x{:}\mathsf{Dyn} \mid e\}!; B?; B!; \{x{:}\mathsf{Dyn} \mid e'\}?$	:	$\{x{:}\mathsf{Dyn} \mid e\}$	$\rightsquigarrow$	$\{x{:}\mathsf{Dyn} \mid e'\}$
$\{x{:}\mathsf{Dyn} \mid e\}!; B?; \{x{:}B \mid e'\}?$	:	$\{x{:}\mathsf{Dyn} \mid e\}$	$\rightsquigarrow$	$\{x{:}B \mid e'\}$
$\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}?$	:	$\{x{:}\mathsf{Dyn} \mid e\}$	$\rightsquigarrow$	$(\mathsf{Dyn}{\to}\mathsf{Dyn})$
$\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}?; \mathbf{Fun}!$	:	$\{x{:}\mathsf{Dyn} \mid e\}$	$\rightsquigarrow$	Dyn
$\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}?; \mathbf{Fun}!; \{x{:}\mathsf{Dyn} \mid e'\}?$	:	$\{x{:}\mathsf{Dyn} \mid e\}$	$\rightsquigarrow$	$\{x{:}\mathsf{Dyn} \mid e'\}$
$\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}?; c_1 \mapsto c_2$	:	$\{x{:}\mathsf{Dyn} \mid e\}$	$\rightsquigarrow$	$T_{21}{\to}T_{22}$
$\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}?; c_1 \mapsto c_2; \mathbf{Fun}!$	:	$\{x{:}\mathsf{Dyn} \mid e\}$	$\rightsquigarrow$	Dyn
$\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}?; c_1 \mapsto c_2; \mathbf{Fun}!; \{x{:}\mathsf{Dyn} \mid e'\}?$	:	$\{x{:}\mathsf{Dyn} \mid e\}$	$\rightsquigarrow$	$\{x{:}\mathsf{Dyn} \mid e'\}$
$\{x{:}B \mid e\}!$	:	$\{x{:}B \mid e\}$	$\rightsquigarrow$	B
$\{x{:}B \mid e\}!; B!$	:	$\{x{:}B \mid e\}$	$\rightsquigarrow$	Dyn
$\{x{:}B \mid e\}!; B!; \{x{:}\mathsf{Dyn} \mid e'\}?$	:	$\{x{:}B \mid e\}$	$\rightsquigarrow$	$\{x{:}\mathsf{Dyn} \mid e'\}$
$\{x{:}B \mid e\}!; \{x{:}B \mid e'\}?$	:	$\{x{:}B \mid e\}$	$\rightsquigarrow$	$\{x{:}B \mid e'\}$ where $e \neq e'$

Rows with a blue background are *value coercions*, and are the only coercions that can appear as tags on pre-values. Horizontal rules mark a change of initial primitive coercion.

Table 4.2: Canonical coercions

(N_PredPred) We have $\vdash c_1; \{x{:}T \mid e\}? : T_1 \rightsquigarrow \{x{:}T \mid e\}$ and $\vdash \{x{:}T \mid e\}!; c_2 : \{x{:}T \mid e\} \rightsquigarrow T_3$, where T is either B or Dyn. By inversion, $\vdash c_1 : T_1 \rightsquigarrow T$ and $\vdash c_2 : T \rightsquigarrow T_3$. We can now apply the IH to find that $\vdash c : T_1 \rightsquigarrow T_3$.

(N_Fun) We have:

$$\begin{array}{l} \vdash c_1; (c_{11} \mapsto c_{12}) : T_1 \rightsquigarrow (T_{21} \to T_{22}) \\ \vdash (c_{21} \mapsto c_{22}); c_2 : (T_{21} \to T_{22}) \rightsquigarrow T_3 \end{array}$$

By inversion, we know that:

$$\begin{array}{ll} \vdash c_1 : & T_1 \rightsquigarrow (T_{11} \to T_{12}) \\ \vdash c_{11} : & T_{21} \rightsquigarrow T_{11} \\ \vdash c_{12} : & T_{12} \rightsquigarrow T_{22} \\ \vdash c_{21} : & T_{31} \rightsquigarrow T_{21} \\ \vdash c_{22} : & T_{22} \rightsquigarrow T_{32} \\ \vdash c_2 : & (T_{31} \to T_{32}) \rightsquigarrow T_3 \end{array}$$

We know $c_{21} * c_{11} \Rightarrow c_{31}$ and $c_{12} * c_{22} \Rightarrow c_{32}$, so by the IH we have that $\vdash c_{31} : T_{31} \rightsquigarrow T_{11}$ and $\vdash c_{32} : T_{12} \rightsquigarrow T_{32}$. By C_Fun, $\vdash c_{31} \mapsto c_{32} : (T_{11} \to T_{12}) \rightsquigarrow (T_{31} \to T_{32})$. We now have enough typing to apply the IH on $c_1 * (c_{31} \mapsto c_{32}); c_2 \Rightarrow c$ and find that $\vdash c : T_1 \rightsquigarrow T_3$.

□

4.5.10 Lemma [Merge is a function]: Given canonical coercions c_1 and c_2, if $c_1 * c_2 \Rightarrow c_3$ and $c_1 * c_2 \Rightarrow c_3'$, then $c_3 = c_3'$.

Proof: By induction on the derivation of $c_1 * c_2 \Rightarrow c_3$, observing in each case that whatever rule applied to form the derivation must be the one used to form $c_1 * c_2 \Rightarrow c_3'$. The only tricky case is N_Canonical. But if $c_1; c_2$ is canonical, then none of the N_... rules can apply. □

4.5.11 Lemma [Merge is an operator]: Given canonical coercions $\vdash c_1 : T_1 \rightsquigarrow T_2$ and $\vdash c_2 : T_2 \rightsquigarrow T_3$, then there exists a unique canonical coercion c such that $c_1 * c_2 \Rightarrow c$.

Proof: By induction on $\text{size}(c_1) + \text{size}(c_2)$, with a long case analysis. Uniqueness is by Lemma 4.5.10; we use Lemma 4.7.1 to apply the IH when merging two functional coercions. We can use Table 4.2 and types to narrow the search.

First, we can rule out cases where either coercion is Id (we just get the other coercion, which is already canonical) or Fail (we just get Fail, which is canonical). Now we go by analysis on the left coercion c_1. In many cases we will reduce out a few primitive coercions and then use the fact that prefixes and suffixes of canonical coercions are canonical (Lemma 4.5.6) to apply the IH.

($\{x{:}\mathsf{Dyn} \mid e\}?$) The only coercions that can apply are of the form $\{x{:}\mathsf{Dyn} \mid e\}!; c_2'$. In all cases, we will first apply N_PredPred, leaving us with Id on the left and c_2' on the right. We will have just c_2', which is canonical since it is a suffix of a canonical coercion (Lemma 4.5.6).

($B?$) The only canonical coercions that can apply are:

1. $B!$, where $B?; B!$ is canonical;
2. $B!; \{x{:}\mathsf{Dyn} \mid e\}?$, where $B?; B!; \{x{:}\mathsf{Dyn} \mid e\}?$ is canonical; and
3. $\{x{:}B \mid e\}?$, where $B?; \{x{:}B \mid e\}?$ is canonical.

($B?; B!$) Here c_2 can be any canonical coercion from Dyn. The possibilities are:

1. $\{x{:}\mathsf{Dyn} \mid e\}?$, where $B?; B!; \{x{:}\mathsf{Dyn} \mid e\}?$ is canonical;
2. $B?; c_2'$, with $B? * c_2' \Rightarrow c_3$ by the IH and Lemma 4.5.6, and we can then apply N_PredPred;
3. $B'?; c_2'$, where $B \neq B'$ and we have Fail;
4. $\mathbf{Fun}?; c_2'$, where we have Fail.

($B?; B!; \{x{:}\mathsf{Dyn} \mid e\}?$) Here c_2 can be any canonical coercion from $\{x{:}\mathsf{Dyn} \mid e\}?$. All of these are of the form $\{x{:}\mathsf{Dyn} \mid e\}!; c_2'$. By the IH, $B?; B! * c_2' \Rightarrow c$ for some canonical c; then we can apply N_PredPred.

($B?; \{x{:}B \mid e\}?$) Here c_2 can be any canonical coercion from $\{x{:}B \mid e\}$. All of these are of the form $\{x{:}B \mid e\}!; c_2'$. By the IH, $B? * c_2' \Rightarrow c$ for some canonical c; then we can apply N_PredPred.

($\mathbf{Fun}?$) Here c_2 can be any canonical coercion from $\mathsf{Dyn}{\to}\mathsf{Dyn}$, to which is either of the form $\mathbf{Fun}!; c_2'$ or $c_1 \mapsto c_2; c_2'$. In either case, $\mathbf{Fun}?; c_2$ is already canonical, so by N_Canonical we are done.

($\mathbf{Fun}?; \mathbf{Fun}!$) Here c_2 can be any canonical coercion from Dyn. The possibilities are:

1. $\{x{:}\mathsf{Dyn} \mid e\}?$, where $\mathbf{Fun}?; \mathbf{Fun}!; \{x{:}\mathsf{Dyn} \mid e\}?$ is canonical;
2. $B?; c_2'$, where we have Fail;
3. $\mathbf{Fun}?; c_2'$, where we can step once by N_PredPred after observing that $\mathbf{Fun}? * c_2' \Rightarrow c$ for some canonical c by the IH and Lemma 4.5.6.

($\mathbf{Fun}?; \mathbf{Fun}!; \{x{:}\mathsf{Dyn} \mid e\}?$) Here c_2 comes from $\{x{:}\mathsf{Dyn} \mid e\}$. These are all of the form $\{x{:}\mathsf{Dyn} \mid e\}!; c_2'$. If we are to step by N_PredPred, we must consider $\mathbf{Fun}?; \mathbf{Fun}!$ and c_2' (canonical by Lemma 4.5.6). Since their combined size is smaller than our original coercions, we know by the IH that $\mathbf{Fun}?; \mathbf{Fun}! * c_2' \Rightarrow c$ for some canonical c, and can step.

(**Fun**?; $c_{11} \mapsto c_{12}$) The coercion c_2 must come from a functional type that agrees with the type of $\vdash c_{11} \mapsto c_{12} : (T_{11} \to T_{12}) \rightsquigarrow (T_{21} \to T_{22})$, i.e., $\vdash c_2 : (T_{21} \to T_{22}) \rightsquigarrow T_3$. These coercions either begin **Fun**!; c_2' or $c_{21} \mapsto c_{22}; c_2'$. In the first case, **Fun**?; $c_{11} \mapsto c_{12}$; **Fun**!; c_2' is canonical if **Fun**!; c_2' is canonical. In the second case, we will step by N_Phi: we first apply the IH on the smaller coercions **Fun**? and $c_{31} \mapsto c_{32}; c_2'$ (using Lemma 4.7.1) to find a canonical c.

(**Fun**?; $c_{11} \mapsto c_{12}$; **Fun**!) Here c_2 comes from type Dyn. If it is of the form $B?; c_2'$, we have Fail, which is canonical. If $c_2 = \{x{:}\mathsf{Dyn} \mid e\}?$, then their concatenation, **Fun**?; $c_{11} \mapsto c_{12}$; **Fun**!; $\{x{:}\mathsf{Dyn} \mid e\}?$, is canonical. If it is of the form **Fun**?; c_2', then we can step by N_Phi: we must show how **Fun**?; $c_{11} \mapsto c_{12}$ and c_2' merge. The latter is canonical by Lemma 4.5.6. Since both are canonical and their combined size is smaller than the original pair of coercions (by Lemma 4.7.1), we can apply the IH to find a canonical c that they merge to.

(**Fun**?; $c_{11} \mapsto c_{12}$; **Fun**!; $\{x{:}\mathsf{Dyn} \mid e\}?$) Here c_2 comes from $\{x{:}\mathsf{Dyn} \mid e\}?$, so it must be a canonical coercion of the form $\{x{:}\mathsf{Dyn} \mid e\}!; c_2'$. We can combine them by N_PredPred, using Lemma 4.5.6 and the IH.

($B!$) In this case, c_2 comes from Dyn. If it is $\{x{:}\mathsf{Dyn} \mid e\}?$, we are done—$B!; \{x{:}\mathsf{Dyn} \mid e\}?$ is canonical. If it begins **Fun**!; c_2', we get Fail, which is canonical. If it begins $B?; c_2'$, we get c_2', which is canonical by Lemma 4.5.6.

($B!; \{x{:}\mathsf{Dyn} \mid e\}?$) Here $c_2 = \{x{:}\mathsf{Dyn} \mid e\}!; c_2'$, for canonical c_2' (Lemma 4.5.6). We can apply the IH to find $B! * c_2' \Rightarrow c$ for some canonical c, stepping by N_PredPred.

($\{x{:}B \mid e\}?$) Here $c_2 = \{x{:}B \mid e\}!; c_2'$, for canonical c_2' (Lemma 4.5.6). By N_PredPred and N_Canonical, we find that $\{x{:}B \mid e\}? * \{x{:}B \mid e\}!; c_2' \Rightarrow c_2'$.

(**Fun**!) In this case, c_2 comes from Dyn. If it is $\{x{:}\mathsf{Dyn} \mid e\}?$, we are done; **Fun**!; $\{x{:}\mathsf{Dyn} \mid e\}?$ is canonical. If it begins $B!; c_2'$, we get Fail, which is canonical. If it begins **Fun**?; c_2', we get c_2', which is canonical by Lemma 4.5.6.

(**Fun**!; $\{x{:}\mathsf{Dyn} \mid e\}?$) Here $c_2 = \{x{:}\mathsf{Dyn} \mid e\}!; c_2'$, with c_2' canonical (Lemma 4.5.6). We can apply the IH to find **Fun**! $* c_2' \Rightarrow c$ for some canonical c.

($c_{11} \mapsto c_{12}$) The coercion c_2 must come from a functional type that agrees with the type of $\vdash c_{11} \mapsto c_{12} : (T_{11} \to T_{12}) \rightsquigarrow (T_{21} \to T_{22})$, i.e., $\vdash c_2 : (T_{21} \to T_{22}) \rightsquigarrow T_3$. These coercions either begin **Fun**!; c_2' or $c_{21} \mapsto c_{22}; c_2'$. In the first case, $c_{11} \mapsto c_{12}$; **Fun**!; c_2' is canonical if **Fun**!; c_2' is canonical.

In the second case, we use the IH to find that $c_{21} * c_{11} \Rightarrow c_{31}$ and $c_{12} * c_{22} \Rightarrow c_{32}$. If $c_{21} \mapsto c_{22}; c_2'$ was canonical, so must be $c_{31} \mapsto c_{32}; c_2'$, which is what we are left with after stepping by N_Fun and by N_Canonical.

($c_{11} \mapsto c_{12}; \mathbf{Fun}!$) In this case, c_2 comes from Dyn. If it is $\{x{:}\mathsf{Dyn} \mid e\}?$, we are done; $\mathbf{Fun}!; \{x{:}\mathsf{Dyn} \mid e\}?$ is canonical. If it begins $B!; c_2'$, we get Fail, which is canonical. If it begins $\mathbf{Fun}?; c_2'$, we can apply the IH (since c_2' is canonical by Lemma 4.5.6) to show that $c_{11} \mapsto c_{12} * c_2' \Rightarrow c$ for some canonical c, stepping by N_Phi.

($c_{11} \mapsto c_{12}; \mathbf{Fun}!; \{x{:}\mathsf{Dyn} \mid e\}?$) Here $c_2 = \{x{:}\mathsf{Dyn} \mid e\}!; c_2'$, where c_2' is canonical by Lemma 4.5.6. We can apply the IH to find $c_{11} \mapsto c_{12}; \mathbf{Fun}! * c_2' \Rightarrow c$ for some canonical c, stepping by N_PredPred.

($\{x{:}\mathsf{Dyn} \mid e\}!$) Here c_2 must come from Dyn.

If it is $\{x{:}\mathsf{Dyn} \mid e\}?$, then we step by N_Phi and are left with Id. If it is $\{x{:}\mathsf{Dyn} \mid e'\}?$ for $e \neq e'$, then $\{x{:}\mathsf{Dyn} \mid e\}!; \{x{:}\mathsf{Dyn} \mid e\}?$ is canonical.

If it is $B!; c_2'$ or $\mathbf{Fun}!; c_2'$, then again $\{x{:}\mathsf{Dyn} \mid e\}!; c_2$ is canonical.

($\{x{:}\mathsf{Dyn} \mid e\}!; \{x{:}\mathsf{Dyn} \mid e'\}?$) The coercion c_2 must be of the form $\{x{:}\mathsf{Dyn} \mid e''\}!; c_2'$, where c_2' is canonical by Lemma 4.5.6. So by the IH, $\{x{:}\mathsf{Dyn} \mid e\}! * c_2' \Rightarrow c$ for some canonical c, and we can step by N_PredPred.

($\{x{:}\mathsf{Dyn} \mid e\}!; B?$) Here c_2 must be either $B!; c_2'$ or $\{x{:}B \mid e'\}?$. In either case, the concatenation of the two is canonical.

($\{x{:}\mathsf{Dyn} \mid e\}!; B?; B!$) In this case, c_2 comes from Dyn. If it is $\{x{:}\mathsf{Dyn} \mid e'\}?$, we are done—$\{x{:}\mathsf{Dyn} \mid e\}!; B?; B!; \{x{:}\mathsf{Dyn} \mid e'\}?$ is canonical. If it begins $\mathbf{Fun}!; c_2'$, we get Fail, which is canonical. If it begins $B?; c_2'$, we can apply the IH to find $\{x{:}\mathsf{Dyn} \mid e\}!; B? * c_2' \Rightarrow c$, since c_2' is canonical by Lemma 4.5.6. We can then apply N_Phi.

($\{x{:}\mathsf{Dyn} \mid e\}!; B?; B!; \{x{:}\mathsf{Dyn} \mid e'\}?$) Here $c_2 = \{x{:}\mathsf{Dyn} \mid e'\}!; c_2'$, where c_2' is canonical by Lemma 4.5.6. We can apply the IH to find $\{x{:}\mathsf{Dyn} \mid e\}!; B?; B! * c_2' \Rightarrow c$ for some canonical c, stepping by N_PredPred.

($\{x{:}\mathsf{Dyn} \mid e\}!; B?; \{x{:}B \mid e'\}?$) Here $c_2 = \{x{:}B \mid e'\}!; c_2'$, where c_2' is canonical by Lemma 4.5.6. We can apply the IH to find $\{x{:}\mathsf{Dyn} \mid e\}!; B? * c_2' \Rightarrow c$ for some canonical c, stepping by N_PredPred.

($\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}?$) Here c_2 can be any canonical coercion from $\mathsf{Dyn}{\to}\mathsf{Dyn}$, i.e., of the form $\mathbf{Fun}!; c_2'$ or $c_1 \mapsto c_2; c_2'$ where c_2' is canonical. In either case, $\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}?; c_2$ is already canonical, so by N_Canonical we are done.

($\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}?; \mathbf{Fun}!$) In this case, c_2 comes from Dyn. If it is $\{x{:}\mathsf{Dyn} \mid e\}?$, we are done—$\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}?; \mathbf{Fun}!; \{x{:}\mathsf{Dyn} \mid e\}?$ is canonical. If it begins $B!; c_2'$, we get Fail, which is canonical. If it begins $\mathbf{Fun}?; c_2'$, we can apply the IH (since c_2' is canonical by Lemma 4.5.6) to show that $\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}? * c_2' \Rightarrow c$ for some canonical c, stepping by N_Phi.

($\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}?; \mathbf{Fun}!; \{x{:}\mathsf{Dyn} \mid e'\}?$) Here $c_2 = \{x{:}\mathsf{Dyn} \mid e'\}!; c_2'$, where c_2' is canonical by Lemma 4.5.6. We can apply the IH to find $\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}?; \mathbf{Fun}! * c_2' \Rightarrow c$ for some canonical c, stepping by N_PredPred.

($\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}?; c_{11} \mapsto c_{12}$) The coercion c_2 must come from a functional type that agrees with the type of $\vdash c_{11} \mapsto c_{12} : (T_{11} \to T_{12}) \rightsquigarrow (T_{21} \to T_{22})$, i.e., $\vdash c_2 : (T_{21} \to T_{22}) \rightsquigarrow T_3$. These coercions either begin $\mathbf{Fun}!; c_2'$ or $c_{21} \mapsto c_{22}; c_2'$. In the first case, $c_{11} \mapsto c_{12}; \mathbf{Fun}!; c_2'$ is canonical if $\mathbf{Fun}!; c_2'$ is canonical.

In the second case, the IH gives us that $c_{21} * c_{11} \Rightarrow c_{31}$ and $c_{12} * c_{22} \Rightarrow c_{32}$ (using Lemma 4.7.1 for the size argument). If $c_{21} \mapsto c_{22}; c_2'$ was canonical, so must be $(c_{31} \mapsto c_{32}); c_2'$. By the IH again, $\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}? * (c_{31} \mapsto c_{32}); c_2' \Rightarrow c$ for some canonical c, so we can step by N_Fun.

($\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}?; c_{11} \mapsto c_{12}; \mathbf{Fun}!$) In this case, c_2 comes from Dyn. If it is $\{x{:}\mathsf{Dyn} \mid e'\}?$, we are done—$\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}?; c_{11} \mapsto c_{12}; \mathbf{Fun}!; \{x{:}\mathsf{Dyn} \mid e'\}?$ is canonical. If it begins $B!; c_2'$, we get Fail, which is canonical. If it begins $\mathbf{Fun}?; c_2'$, we can apply the IH (since c_2' is canonical by Lemma 4.5.6) to show that $\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}?; c_{11} \mapsto c_{12} * c_2' \Rightarrow c$ for some canonical c, stepping by N_Phi.

($\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}?; c_{11} \mapsto c_{12}; \mathbf{Fun}!; \{x{:}\mathsf{Dyn} \mid e'\}?$) Here $c_2 = \{x{:}\mathsf{Dyn} \mid e'\}!; c_2'$, where c_2' is canonical by Lemma 4.5.6. We can apply the IH to find $\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}?; c_{11} \mapsto c_{12}; \mathbf{Fun}! * c_2' \Rightarrow c$ for some canonical c, stepping by N_PredPred.

($\{x{:}B \mid e\}!$) Here $c_2 = \{x{:}B \mid e\}?; c_2'$, with c_2' canonical by Lemma 4.5.6. We merge to c_2' by N_Phi and N_Canonical.

($\{x{:}B \mid e\}!; B!$) Here c_2 comes from Dyn. If c_2 is $\{x{:}\mathsf{Dyn} \mid e'\}?$, then we are done, since $\{x{:}B \mid e\}!; B!; \{x{:}\mathsf{Dyn} \mid e'\}?$ is canonical. If it begins $\mathbf{Fun}!; c_2'$, we get Fail, which is canonical. If it begins $B?; c_2'$, we can use the IH to show that $\{x{:}B \mid e\}! * c_2' \Rightarrow c$ for a canonical c (since c_2' is canonical by Lemma 4.5.6). Then we can step by N_Phi.

($\{x{:}B \mid e\}!; B!; \{x{:}\mathsf{Dyn} \mid e'\}?$) Here $c_2 = \{x{:}\mathsf{Dyn} \mid e'\}!; c_2'$. We know that c_2' is canonical (Lemma 4.5.6), so by the IH we have $\{x{:}B \mid e\}!; B! * c_2' \Rightarrow c$ for some canonical c. We step by N_PredPred.

($\{x{:}B \mid e\}!; \{x{:}B \mid e'\}?$) Here $c_2 = \{x{:}B \mid e'\}!; c_2'$. We know that c_2' is canonical (Lemma 4.5.6), so by the IH we have $\{x{:}B \mid e\}! * c_2' \Rightarrow c$ for some canonical c. We step by N_PredPred.

□

4.5.12 Lemma: For all canonical coercions c_1 and c_2, if $c_1 * c_2 \Rightarrow c_3$ then $c_1; c_2 \longrightarrow^* c_3$.

Proof: By induction on the derivation of $c_1 * c_2 \Rightarrow c_3$.

(N_Canonical) Since $c_3 = c_1; c_2$, we are done immediately by reflexivity.

(N_FailL) We have $\mathsf{Fail} * c_2 \Rightarrow \mathsf{Fail}$. By induction on the length of c_2, we have $\mathsf{Fail}; c_2 \longrightarrow^* \mathsf{Fail}$.

(N_FailR) We have $c_1 * \mathsf{Fail} \Rightarrow \mathsf{Fail}$. By induction on the length of c_1, we have $c_1; \mathsf{Fail} \longrightarrow^* \mathsf{Fail}$.

(N_Phi) We have $c_1 * c_2 \Rightarrow c_3$. We have $c_1; D!; D?; c_2 \longrightarrow c_1; c_2$ for all D, so we are done by the IH.

(N_BFailB) We have $c_1; B!; B'?; c_2 \longrightarrow c_1; \mathsf{Fail}; c_2$. By induction on the length of c_1 and c_2, we can conclude that $c_1; \mathsf{Fail}; c_2 \longrightarrow^* \mathsf{Fail}$.

(N_BFailFun) We have $c_1; B!; \mathbf{Fun}?; c_2 \longrightarrow c_1; \mathsf{Fail}; c_2$. By induction on the length of c_1 and c_2, we know that $c_1; \mathsf{Fail}; c_2 \longrightarrow^* \mathsf{Fail}$.

(N_FunFailB) We have $c_1; \mathbf{Fun}!; B?; c_2 \longrightarrow c_1; \mathsf{Fail}; c_2$. By induction on the length of c_1 and c_2, we know that $c_1; \mathsf{Fail}; c_2 \longrightarrow^* \mathsf{Fail}$.

(N_PredPred) We have $c_1 * c_2 \Rightarrow c_3$ and $c_1; \{x{:}T \mid e\}?; \{x{:}T \mid e\}!; c_2 \longrightarrow c_1; c_2$, so we have $c_1; c_2 \longrightarrow^* c_3$ by the IH.

(N_Fun) We know that $c_{21} * c_{11} \Rightarrow c_{31}$ and $c_{12} * c_{22} \Rightarrow c_{32}$ and $c_1 * (c_{31} \mapsto c_{32}); c_2 \Rightarrow c_3$.

We can rewrite $c_1; (c_{11} \mapsto c_{12}); (c_{21} \mapsto c_{22}); c_2 \longrightarrow c_1; ((c_{21}; c_{11}) \mapsto (c_{12}; c_{22})); c_2$. By the IH, we know that $c_{21}; c_{11} \longrightarrow^* c_{31}$ and $c_{12}; c_{22} \longrightarrow^* c_{32}$, so we can rewrite to $c_1; (c_{31} \mapsto c_{32}); c_2$. But we know by the IH that this rewrites to c_3, so we are done.

□

4.5.13 Lemma [Merge is associative]: $c_1 \Downarrow (c_2 \Downarrow c_3) = (c_1 \Downarrow c_2) \Downarrow c_3$ for all canonical coercions c_1, c_2, and c_3.

Proof: Consider the term $c_1; c_2; c_3$. On the one hand, we can say that $c_1; c_2; c_3 \longrightarrow^* c_1; c_2 \Downarrow c_3 \longrightarrow^* c_1 \Downarrow c_2 \Downarrow c_3$ by Lemma 4.5.12. On the other hand, we also have $c_1; c_2; c_3 \longrightarrow^* c_1 \Downarrow c_2; c_3 \longrightarrow^* c_1 \Downarrow c_2 \Downarrow c_3$.

Recall that results of merges are canonical forms (Lemma 4.5.11), which are normal (Lemma 4.5.5). Since rewriting is confluent (Lemma 4.5.2), it must be that case that $c_1 \Downarrow (c_2 \Downarrow c_3) = (c_1 \Downarrow c_2) \Downarrow c_3$. □

$\boxed{e_1 \longrightarrow e_2}$

$$\frac{}{(\lambda x{:}T.\ e_{12})_{\mathsf{Id}}\ v_2 \longrightarrow e_{12}[v_2/x]}\ \text{E\_Beta} \qquad \frac{}{u_{1\,(c_1 \mapsto c_2)}\ v_2 \longrightarrow \langle c_2 \rangle\ (u_{1\,\mathsf{Id}}\ (\langle c_1 \rangle\ v_2))}\ \text{E\_Fun}$$

$$\frac{}{op(v_1, \ldots, v_n) \longrightarrow [\![op]\!]\,(v_1, \ldots, v_n)}\ \text{E\_Op}$$

$$\frac{}{\langle \{x{:}T \mid e\}?; c \rangle\ v \longrightarrow \langle c \rangle\ \langle \{x{:}T \mid e\}, e[v/x], v \rangle}\ \text{E\_Check}$$

$$\frac{}{\langle \{x{:}T \mid e\}, \mathsf{true}_{\mathsf{Id}}, u_c \rangle \longrightarrow u_{c \Downarrow \{x:T \mid e\}?}}\ \text{E\_CheckOK}\bullet$$

$$\frac{}{\langle \{x{:}T \mid e\}, \mathsf{false}_{\mathsf{Id}}, v \rangle \longrightarrow \mathsf{fail}}\ \text{E\_CheckFail} \qquad \frac{d_1 \neq \{x{:}T \mid e\}? \qquad c * d_1 \Rightarrow \mathsf{Fail}}{\langle d_1; c_2 \rangle\ u_c \longrightarrow \mathsf{fail}}\ \text{E\_TagFail}\bullet$$

$$\frac{d_1 \neq \{x{:}T \mid e\}? \qquad c * d_1 \Rightarrow c' \qquad c' \neq \mathsf{Fail}}{\langle d_1; c_2 \rangle\ u_c \longrightarrow \langle c_2 \rangle\ u_{c'}}\ \text{E\_Tag}\bullet$$

$$\frac{}{\langle \mathsf{Id} \rangle\ v \longrightarrow v}\ \text{E\_TagId} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ \text{E\_AppL} \qquad \frac{e_2 \longrightarrow e_2'}{v_1\ e_2 \longrightarrow v_1\ e_2'}\ \text{E\_AppR}$$

$$\frac{e_i \longrightarrow e_i'}{op(v_1, \ldots, v_{i-1}, e_i, \ldots, e_n) \longrightarrow op(v_1, \ldots, v_{i-1}, e_i', \ldots, e_n)}\ \text{E\_OpInner}$$

$$\frac{}{\langle c_1 \rangle\ (\langle c_2 \rangle\ e) \longrightarrow \langle c_2 \Downarrow c_1 \rangle\ e}\ \text{E\_Merge}\bullet \qquad \frac{e \neq \langle c' \rangle\ e'' \quad c \neq \mathsf{Fail} \quad e \longrightarrow e'}{\langle c \rangle\ e \longrightarrow \langle c \rangle\ e'}\ \text{E\_CoerceInner}$$

$$\frac{e_2 \longrightarrow e_2'}{\langle \{x{:}T \mid e_1\}, e_2, v \rangle \longrightarrow \langle \{x{:}T \mid e_1\}, e_2', v \rangle}\ \text{E\_CheckInner}$$

$$\frac{e \neq \langle c \rangle\ e'}{\langle \mathsf{Fail} \rangle\ e \longrightarrow \mathsf{fail}}\ \text{E\_Fail} \qquad \frac{}{\langle c \rangle\ \mathsf{fail} \longrightarrow \mathsf{fail}}\ \text{E\_CoerceRaise}$$

$$\frac{}{\mathsf{fail}\ e_2 \longrightarrow \mathsf{fail}}\ \text{E\_AppRaiseL} \qquad \frac{}{v_1\ \mathsf{fail} \longrightarrow \mathsf{fail}}\ \text{E\_AppRaiseR}$$

$$\frac{}{op(v_1, \ldots, v_{i-1}, \mathsf{fail}, \ldots, e_n) \longrightarrow \mathsf{fail}}\ \text{E\_OpRaise} \qquad \frac{}{\langle \{x{:}T \mid e\}, \mathsf{fail}, v \rangle \longrightarrow \mathsf{fail}}\ \text{E\_CheckRaise}$$

Figure 4.22: Efficient operational semantics

$$
\begin{array}{ll}
 & \mathsf{odd}\ 3_{\mathsf{Id}} \\
\longrightarrow & \mathsf{even}_{\mathsf{Int!}\mapsto\mathsf{Bool?}}\ 2_{\mathsf{Id}} \\
\longrightarrow & \langle \mathsf{Bool?} \rangle\ (\mathsf{even}\ (\langle \mathsf{Int!} \rangle\ 2_{\mathsf{Id}})) \\
\longrightarrow & \langle \mathsf{Bool?} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathsf{Int?}\mapsto\mathsf{Bool!}}\ 2_{\mathsf{Int!}}) \\
\longrightarrow & \langle \mathsf{Bool?} \rangle\ (\langle \mathsf{Bool!} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathsf{Id}}\ (\langle \mathsf{Int?} \rangle\ 2_{\mathsf{Int!}}))) \\
\longrightarrow & \langle \mathsf{Id} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathsf{Id}}\ (\langle \mathsf{Int?} \rangle\ 2_{\mathsf{Int!}})) \\
\longrightarrow & \langle \mathsf{Id} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathsf{Id}}\ 2_{\mathsf{Id}}) \\
\longrightarrow & \langle \mathsf{Id} \rangle\ (\mathsf{odd}\ 1_{\mathsf{Id}}) \\
\longrightarrow & \langle \mathsf{Id} \rangle\ (\mathsf{even}_{\mathsf{Int!}\mapsto\mathsf{Bool?}}\ 0_{\mathsf{Id}}) \\
\longrightarrow & \langle \mathsf{Id} \rangle\ (\langle \mathsf{Bool?} \rangle\ (\mathsf{even}\ (\langle \mathsf{Int!} \rangle\ 0_{\mathsf{Id}}))) \\
\longrightarrow & \langle \mathsf{Bool?} \rangle\ (\mathsf{even}\ (\langle \mathsf{Int!} \rangle\ 0_{\mathsf{Id}})) \\
\longrightarrow & \langle \mathsf{Bool?} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathsf{Int?}\mapsto\mathsf{Bool!}}\ 0_{\mathsf{Int!}}) \\
\longrightarrow & \langle \mathsf{Bool?} \rangle\ (\langle \mathsf{Bool!} \rangle\ (\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathsf{Id}}\ (\langle \mathsf{Int?} \rangle\ 0_{\mathsf{Int!}})) \\
\longrightarrow & \langle \mathsf{Id} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathsf{Id}}\ (\langle \mathsf{Int?} \rangle\ 0_{\mathsf{Int!}})) \\
\longrightarrow & \langle \mathsf{Id} \rangle\ ((\lambda x{:}\mathsf{Int}.\ \ldots)_{\mathsf{Id}}\ 0_{\mathsf{Id}}) \\
\longrightarrow & \langle \mathsf{Id} \rangle\ \mathsf{true}_{\mathsf{Id}} \\
\longrightarrow & \mathsf{true}_{\mathsf{Id}}
\end{array}
$$

Figure 4.23: Space-efficient reduction

4.5.2 Operational semantics

I give the changed operational semantics in Figure 4.22. The biggest change to the operational semantics is that E_Merge explicitly merges the two coercions. Herman et al. simply say that they keep their coercions in normal form—that is, we should interpret normalization happening automatically when E_Merge applies, even though they write E_Merge as directly concatenating the two coercions into $c_2; c_1$. My semantics explicitly normalizes the coercions (rule E_Merge), possibly stopping the program on the next step (the no-longer-useless rule E_Fail).

Otherwise, the rules are largely the same as the naïve semantics, though we're now able to use merges to distill the tag rules into a few possibilities: E_Tag replaces all of the successful F_Tag* rules; E_TagFail replaces all of the failing F_Tag* rules. E_CheckOK is essentially F_CheckOK, though it uses a merge instead of concatenation (though the typing rules mean that the merge will apply N_Canonical every time).

We can finally observe that our reduction is space efficient: the coercions in Figure 4.23 don't grow with the size of the input like the coercions in Figure 4.14 or the casts in Figure 4.2. I discuss this claim in more detail in Section 4.7.

4.5.3 Proofs

Efficient enjoys type soundness; we can show as much using standard syntactic methods. We must assume that $\mathsf{ty}(k)$ and $\mathsf{ty}(op)$ are always well formed.

4.5.14 Lemma [Regularity]: 1. If $\Gamma \vdash e : T$, then $\vdash T$.

2. If $\Gamma \vdash u : T$, then $\vdash T$.

3. If $\vdash \Gamma$, then $\vdash T$ for all $x{:}T \in \Gamma$.

Proof: By mutual induction on the typing derivations. □

4.5.15 Lemma [Determinism]: If $e \longrightarrow e_1$ and $e \longrightarrow e_2$, then $e_1 = e_2$.

Proof: By induction on $e \longrightarrow e_1$, observing that in each case the same rule must have applied to find $e \longrightarrow e_2$. In the E_Merge case, we rely on the fact that merge is an operator on canonical coercions (Lemma 4.5.11). □

4.5.16 Lemma [Canonical forms]: If $\emptyset \vdash v : T$ then:

- If $T = \mathsf{Bool}$, then $v = \mathsf{true}_{\mathsf{Id}}$ or $v = \mathsf{false}_{\mathsf{Id}}$.
- If $T = T_1 \to T_2$, then $v = \lambda x{:}T_1'.\ e_c$ where $c = \mathsf{Id}$ or $c = c_1 \mapsto c_2$.

Proof: By cases on the typing derivation. We observe that the only coercions from B to B are Id and Fail, and $\mathsf{true}_{\mathsf{Fail}}$ isn't well formed.

In the function case, the only coercions from $T_{11} \to T_{12}$ to $T_{21} \to T_{22}$ are Id and Fail and $c_1 \mapsto c_2$. Since u_{Fail} isn't well formed, c must be either the identity or a functional coercion. □

4.5.17 Lemma [Progress]: If $\emptyset \vdash e : T$ then either e is a result, or there exists an e' such that $e \longrightarrow e'$.

Proof: By induction on the typing derivation $\emptyset \vdash e : T$. I give only the interesting cases.

(T_Coerce) We have $\emptyset \vdash \langle c \rangle\ e : T_2$, where $\vdash c : T_1 \rightsquigarrow T_2$ and $\emptyset \vdash e : T_1$. If the inner term is of the form $\langle c' \rangle\ e'$, we step by E_Merge (in which case we are well typed by Lemma 4.5.9). If $c = \mathsf{Fail}$, we step E_Fail (in which case we are well typed by Lemma 4.5.14 and T_Fail). If not, it either steps by E_CoerceInner or e is a result. If $e = \mathsf{fail}$, we step by E_CoerceRaise. If $e = u_{c'}$, we step by E_Tag, E_TagFail, or E_Check, depending on what the leftmost coercion in c is. If there is no leftmost coercion, i.e., we have $\langle \mathsf{Id} \rangle\ v$, then we step by E_TagId.

By inversion, $\emptyset \vdash e_2 : \mathsf{Bool}$. By canonical forms (Lemma 4.5.16), if e_2 is a value, then it is either $\mathsf{true}_{\mathsf{Id}}$ or $\mathsf{false}_{\mathsf{Id}}$. In the former case, we step by E_CheckOK; in the latter case, we step by E_CheckFail.

□

4.5.18 Lemma [Weakening]: If $\Gamma_1, \Gamma_2 \vdash e : T$ and $x \notin \mathrm{dom}(\Gamma)$ and $\vdash T'$, then $\Gamma_1, x{:}T', \Gamma_2 \vdash e : T$.

Proof: By induction on e. □

4.5.19 Lemma [Substitution]: If $\emptyset \vdash v : T'$:

- if $\Gamma_1, x{:}T', \Gamma_2 \vdash e : T$ then $\Gamma_1, \Gamma_2 \vdash e[v/x] : T$.
- if $\Gamma_1, x{:}T', \Gamma_2 \vdash u : T$ then $\Gamma_1, \Gamma_2 \vdash u[v/x] : T$.

Proof: By induction on the typing derivation of e. □

4.5.20 Lemma [Preservation]: If $\emptyset \vdash e : T$ and $e \longrightarrow e'$, then $\emptyset \vdash e' : T$.

Proof: By induction on the typing derivation. I give only the interesting cases.

(T_Coerce) We go by cases on the step taken:

(E_CoerceInner) By the IH.

(E_Merge) By T_Coerce and Lemma 4.5.9.

(E_Fail) By T_Fail and regularity (Lemma 4.5.14).

(E_CoerceRaise) By T_Fail and regularity (Lemma 4.5.14).

(E_Tag) By T_Coerce, T_Val, and C_Compose, knowing that the result of the merge isn't a fail or $\{x{:}T \mid e\}?$ for T_Val.

(E_TagFail) By T_Fail and regularity (Lemma 4.5.14).

(E_TagId) By the typing assumption on the value.

(E_Check) By T_Coerce and T_Check.

(T_Check) We go by cases on the step taken:

(E_CheckInner) By the IH.

(E_CheckOK) By T_ValRefine and C_Compose, knowing that the merge can't be a fail—predicates never introduce failures. We can take the evaluation we need directly from the T_Check derivation.

(E_CheckFail) By T_Fail and regularity (Lemma 4.5.14).

(E_CheckRaise) By T_Fail and regularity (Lemma 4.5.14).

□

4.5.21 Theorem [Type soundness]: If $\emptyset \vdash e : T$ then either $e \longrightarrow^* r$ such that $\emptyset \vdash r : T$ or e diverges.

Proof: Using progress (Lemma 4.5.17) and preservation (Lemma 4.5.20). Naturally the proof is not constructive! □

In Section 4.3, I define cotermination and show that if $e \longrightarrow_n^* r$ then $\langle c\rangle\ e$ and $\langle c\rangle\ r$ coterminate, and if e diverges so does $\langle c\rangle\ e$ (Lemma 4.3.13). Unfortunately, neither of these exactly hold in Efficient. For the former, consider $e = \langle\{x{:}\mathsf{Bool} \mid \mathsf{false}_{\mathsf{Id}}\}?\rangle\ \mathsf{true}_{\mathsf{Id}} \longrightarrow^* \mathsf{fail}$. The term $\langle\{x{:}\mathsf{Bool} \mid \mathsf{false}_{\mathsf{Id}}\}!\rangle\ e \longrightarrow^* \mathsf{true}_{\mathsf{Id}}$. For the latter, suppose that $\mathsf{diverge}$ is a closed, divergent term, such as

$$(\lambda x{:}(\mathsf{Dyn}{\to}\mathsf{Dyn}).\ x\ (\langle\mathbf{Fun}!\rangle\ x))_{\mathsf{Id}}\ (\langle\mathbf{Fun}?\mapsto\mathsf{Id}\rangle\ (\lambda x{:}(\mathsf{Dyn}{\to}\mathsf{Dyn}).\ x\ (\langle\mathbf{Fun}!\rangle\ x))_{\mathsf{Id}})$$

On the one hand

$$\begin{array}{ll} & \langle\{x{:}\mathsf{Bool} \mid \mathsf{diverge}\}!\rangle\ (\langle\{x{:}\mathsf{Bool} \mid \mathsf{diverge}\}?\rangle\ \mathsf{true}_{\mathsf{Id}}) \\ \longrightarrow & \langle\mathsf{Id}\rangle\ \mathsf{true}_{\mathsf{Id}} \\ \longrightarrow & \mathsf{true}_{\mathsf{Id}} \end{array}$$

but $\langle\{x{:}\mathsf{Bool} \mid \mathsf{diverge}\}!\rangle\ \mathsf{true}_{\mathsf{Id}}$ diverges. The theorem doesn't hold for when e diverges or evaluates to fail—but it *does* hold for values.

4.5.22 Lemma: If $e \longrightarrow^* v$ and $\langle c\rangle\ v \longrightarrow^* v'$ then $\langle c\rangle\ e \longrightarrow^* v'$.

Proof: By induction on $e \longrightarrow^* v$, and then by cases on the step taken. (The base case ($e = v$) is immediate.)

If the step doesn't have an exposed coercion, we can just reapply the step taken. The core rules (E_Beta, E_Fun, E_Op, E_CheckOK, E_CheckFail), the congruence rules (E_AppL, E_AppR, E_OpInner, E_CheckInner), and the exception raising rules (E_AppRaiseL, E_AppRaiseR, E_OpRaise, E_CheckRaise) all fit this rubric. The remaining cases must merge coercions in e with c. The general thrust is as for Lemma 4.3.13: we will merge and normalize, take some small tag manipulation step (determined by analyzing how the canonical merge went), and then reproduce that for a stepped e.

(E_TagId) We step by E_Merge (using N_Canonical) and apply the IH.

(E_Tag) We have $e_2 = \langle d_1; c_2\rangle\ u_{c_2'} \longrightarrow \langle c_2\rangle\ u_{c_2' \Downarrow d_1}$ with $d_1 \neq \{x{:}T \mid e\}?$.

Instead we step to $\langle d_1; c_2 \Downarrow c\rangle\ u_{c_2'}$ by E_Merge.

If d_1 remains unaffected by the merge, we are done easily: we step by E_Tag and can apply the IH on $\langle c\rangle\ (\langle c_2\rangle\ u_{c'})$.

If d_1 is affected, then all of $d_1; c_2$ must have disappeared. If $d_1; c_2 \Downarrow c = c$, then we are done by assumption. So instead it must be the case that $d_1; c_2 \Downarrow c$ is some suffix of c.

By the IH, $\langle c\rangle\ (\langle c_2\rangle\ u_{c_2' \Downarrow d_1}) \longrightarrow^* v'$, which steps to $\langle c_2 \Downarrow c\rangle\ u_{c_2' \Downarrow d_1}$. Whatever coercion was left in $c_2 \Downarrow c$ that eliminated d_1 must now be exposed, so we can step by E_Tag to find $\langle d_1; c_2 \Downarrow c\rangle\ u_{c_2'} \longrightarrow^* v'$.

(E_TagFail) Contradictory—fail isn't a value.

(E_Check) We have $e_2 = \langle \{x{:}T \mid e\}?; c' \rangle\ v_2' \longrightarrow \langle c' \rangle\ \langle \{x{:}T \mid e\}, e[v_2'/x], v_2' \rangle$; since this steps to some value v', it must be the case that $e[v_2'/x] \longrightarrow^* \mathsf{true}_{\mathsf{Id}}$.

The combined term steps by E_Merge to $\langle \{x{:}T \mid e\}?; c' \Downarrow c \rangle\ v_2'$.

Now, the result of $\{x{:}T \mid e\}?; c' \Downarrow c$ either has the same refinement checking coercion on the front or it doesn't. If it does, we can use the evaluation we found above to step to $\langle c' \Downarrow c \rangle\ v_2'$ (using Lemma 4.5.13).

If it doesn't, we're already at $\langle c' \Downarrow c \rangle\ v_2'$. In either case, that term is equivalent to $\langle c \rangle\ (\langle c' \rangle\ v_2')$, and we can apply the IH on $\langle c' \rangle\ v_2' \longrightarrow^* v_2$.

(E_CoerceInner) We have $e_2 = \langle c_1 \rangle\ (\langle c_2 \rangle\ e_2') \longrightarrow \langle c_2 \Downarrow c_1 \rangle\ e_2'$. We'll step by E_Merge twice to find $\langle c_2 \Downarrow c_1 \Downarrow c \rangle\ e_2'$. By associativity of merges, that term is equal to $\langle c_1 \Downarrow c_2 \Downarrow c \rangle\ e_2$.

By the IH we know that $\langle c \rangle\ (\langle c_2 \Downarrow c_1 \rangle\ e_2') \longrightarrow^* v'$, so we are done.

(E_Fail) Contradictory—fail isn't a value.

(E_Merge) We step by E_Merge and then by E_Merge again to find the same result, and we are done.

□

4.6 Soundness of Efficient with regard to Naive

We will never get an exact semantic match between the naïve and space-efficient semantics: the ψ rule for refinements in the space-efficient semantics means that some checks will happen in the naïve semantics that won't happen in the space-efficient semantics. If those checks fail or diverge, then the Naive term won't behave the same as its translation into Efficient. All we can hope for is that *if* Naive produces a value, then Efficient will produce a similar one.

I adapt the asymmetric logical relations from Chapter 2 to show that the two calculi behave mostly the same, with Naive diverging and failing more often. My proof method largely follows the early one, as well: I augment the logical relation with a separate relation $\vdash c_1 \equiv c_2$ (read "c_1 is equivalent to c_2") on coercions (Figure 4.26). Just as the $\vdash T_1 \Rightarrow T_2 \equiv c$ relation from Section 4.4 was an invariant relating casts to coercions, $\vdash c_1 \equiv c_2$ relates casts to their normal forms. With this relation in hand, we first prove that applying related coercions to related values yields related values (Lemma 4.6.8). Then we show that well typed terms are related to their translations and that coercions are related to their translations (Theorem 4.6.10).

We are forced into a *step-indexed* logical relation because of the type Dyn. In particular, the more common, fixpoint-on-types definition wouldn't work for the function tag case, since we need to take values at Dyn and relate them at the type $\mathsf{Dyn} \to \mathsf{Dyn}$. I define the step-indexed logical relation in Figure 4.24; when I say $e \longrightarrow_n^m e'$, I mean

Value rules $\boxed{v_1 \precsim^j v_2 : T}$

$$\begin{array}{rcl}
\forall j.\ k_{\mathsf{Id}} \precsim^j k_{\mathsf{Id}} : B & \iff & \mathsf{ty}(k) = B \\
v_{11} \precsim^j v_{21} : T_1 \to T_2 & \iff & \\
\forall m < j.\ \forall v_{12} \precsim^m v_{22} : T_1.\ v_{11}\ v_{12} \mathrel{\precsim\!\!\!\approx}^m v_{21}\ v_{22} : T_2 &&
\end{array}$$

$$\begin{array}{rcl}
v_{1\,B!} \precsim^j u_{2\,c \Downarrow B!} : \mathsf{Dyn} & \iff & v_1 \precsim^j u_{2\,c} : B \\
v_{1\,\mathbf{Fun}!} \precsim^j u_{2\,c \Downarrow \mathbf{Fun}!} : \mathsf{Dyn} & \iff & v_1 \precsim^j u_{2\,c} : \mathsf{Dyn} \to \mathsf{Dyn}
\end{array}$$

$$\begin{array}{c}
v_{1\,\{x:T \mid e_1\}?} \precsim^j u_{c \Downarrow \{x:T \mid e_2\}?} : \{x{:}T \mid e_1\} \\
\iff \\
\forall m < j.\ v_1 \precsim^m u_{2\,c} : T \wedge \{x{:}T \mid e_1\} \precsim^m \{x{:}T \mid e_2\}
\end{array}$$

Term rules $\boxed{e_1 \mathrel{\precsim\!\!\!\approx}^j e_2 : T}$

$$\begin{array}{l}
e_1 \mathrel{\precsim\!\!\!\approx}^j e_2 : T \iff \\
e_1 \text{ diverges} \vee \\
\forall m < j. \quad e_1 \longrightarrow_n^m \mathsf{fail} \\
\qquad \vee \quad e_1 \longrightarrow_n^m v_1 \implies e_2 \longrightarrow^* v_2 \wedge v_1 \precsim^{(j-m)} v_2 : T
\end{array}$$

Type rules $\boxed{T_1 \precsim^j T_2}$

$$\begin{array}{rcl}
B \precsim^j B \qquad \mathsf{Dyn} \precsim^j \mathsf{Dyn} & & \\
T_{11} \to T_{12} \precsim^j T_{21} \to T_{22} & \iff & T_{11} \precsim^j T_{21} \wedge T_{12} \precsim^j T_{22} \\
\{x{:}T \mid e_1\} \precsim^j \{x{:}T \mid e_2\} & \iff & \\
\forall m < j.\ \forall v_1 \precsim^m v_2 : T.\ e_1[v_1/x] \mathrel{\precsim\!\!\!\approx}^m e_2[v_2/x] : \mathsf{Bool} &&
\end{array}$$

Closing substitutions and open terms $\boxed{\Gamma \models^j_{\succ} \delta}$ $\boxed{\Gamma \vdash e_1 \mathrel{\precsim\!\!\!\approx} e_2 : T}$

$$\begin{array}{rcl}
\Gamma \models^j_{\succ} \delta & \iff & \forall x \in \mathrm{dom}(\Gamma).\ \delta_1(x) \precsim^j \delta_2(x) : T \\
\Gamma \vdash e_1 \mathrel{\precsim\!\!\!\approx} e_2 : T & \iff & \forall j \geq 0.\ \forall \Gamma \models^j_{\succ} \delta.\ \delta_1(e_1) \mathrel{\precsim\!\!\!\approx}^j \delta_2(e_2) : T
\end{array}$$

Figure 4.24: Relating Naive and Efficient

that e steps to e' in *exactly* m steps. The definitions begin by defining a relation $v_1 \gtrsim^j v_2 : T$ for closed values and a relation $e_1 \gtrapprox^j e_2 : T$ for closed terms as a fixpoint on the index j.[5] I lift the definitions to open terms by defining *dual closing value substitutions* δ; if $\Gamma \models^j_\succ \delta$ and $x{:}T \in \Gamma$, then $\delta_1(x) \gtrsim^j \delta_2(x) : T$.

Naive terms are on the left of the relation, while Efficient terms are on the right. We require that both sides be well typed. We obtain the asymmetry we seek by saying that *when* the naïve semantics yields a value, then the space-efficient yields a similar one—but otherwise, the naïve semantics will fail or diverge. This definition still allows Efficient to diverge or to fail, but then the naïve semantics must also diverge or fail—but note that it's possible for the left-hand side of the relation to diverge and the right-hand side to fail, and vice versa. This is possible because the naïve semantics could run a check that diverges, while the space-efficient semantics skips that check and instead runs a failing one.

Step-indexed logical relations are commonly asymmetric, for a separate reason: only one side needs the index; for us, it is particularly convenient to put the step-index on the naïve side, allowing us to skip reasoning about how step indices and merges interact.

We could try to be more specific in the relation: either you get the same thing, or you get divergence or a failure—that can be traced back exactly to a check that happened in Naive but not in Efficient. I omit this more precise tracking for simplicity's sake.

The value relation $v_1 \gtrsim^j v_2 : T$ is subtler than usual for this logical relation: the definitions at Dyn and $\{x{:}T \mid e\}$ must shuffle some tags around. In particular, the rule for type Dyn is split into cases by the underlying tag of values. The case for refinements $\{x{:}T \mid e\}$ requires that the values be related at the underlying type T (recalling that $T = B$ or $T = \mathsf{Dyn}$) and also that the values be tagged as satisfying the predicate (or a related predicate in Efficient). For well typed terms, this is enough to ensure that the underlying values satisfy their refinement predicates.

My proof works by showing that a well-typed naïve term e is related to its translation into the space-efficient term $\mathsf{canonical}(e)$. I define $\mathsf{canonical}$ in Figure 4.25, omitting most of the cases since they are homomorphic. In particular, $\mathsf{canonical}(v)$ must unfold the stacked tags on a Naive value and merge them into a single coercion.

Before we continue, I must justify that $\mathsf{canonical}$ is a function—in particular, is the case for composite coercions valid?

4.6.1 Lemma: $\mathsf{canonical}$ is a well defined function that produces canonical coercions.

Proof: By induction on c, using Lemma 4.5.11 to show that $c_1 \Downarrow c_2$ is a unique canonical coercion when c_1 and c_2 are canonical. □

[5]The definition for terms is just fancy notation for a proposition involving the relation on values; this stratified definition conveniently separates the definition of related values and evaluation, while avoiding the need for $\top\top$-closure [54].

Composite coercions $\boxed{\mathsf{canonical}(c) : c}$

$$\begin{array}{rcl}
\mathsf{canonical}(\mathsf{Id}) & = & \mathsf{Id} \\
\mathsf{canonical}(d_1; \ldots; d_n) & = & \mathsf{canonical}(d_1) \Downarrow \mathsf{canonical}(d_2; \ldots; d_n)
\end{array}$$

Primitive coercions $\boxed{\mathsf{canonical}(d) : d}$

$$\begin{array}{rcl}
\mathsf{canonical}(\mathsf{Fail}) & = & \mathsf{Fail} \\
\mathsf{canonical}(D!) & = & \mathsf{canonical}(D)! \\
\mathsf{canonical}(D?) & = & \mathsf{canonical}(D)? \\
\mathsf{canonical}(c_1 \mapsto c_2) & = & \mathsf{canonical}(c_1) \mapsto \mathsf{canonical}(c_2)
\end{array}$$

Tags $\boxed{\mathsf{canonical}(D) : D}$

$$\begin{array}{rcl}
\mathsf{canonical}(B) & = & B \\
\mathsf{canonical}(\mathbf{Fun}) & = & \mathbf{Fun} \\
\mathsf{canonical}(\{x{:}T \mid e\}) & = & \{x{:}T \mid \mathsf{canonical}(e)\}
\end{array}$$

Pre-values $\boxed{\mathsf{canonical}(u) : u}$

$$\begin{array}{rcl}
\mathsf{canonical}(k) & = & k \\
\mathsf{canonical}(\lambda x{:}T.\ e) & = & \lambda x{:}\mathsf{canonical}(T).\ \mathsf{canonical}(e)
\end{array}$$

Expressions $\boxed{\mathsf{canonical}(e) : e}$

$$\begin{array}{rcl}
\mathsf{canonical}(x) & = & x \\
\mathsf{canonical}(u_{\mathsf{Id}}) & = & \mathsf{canonical}(u)_{\mathsf{Id}} \\
\mathsf{canonical}(v_d) & = & u_{c \Downarrow d} \text{ where } \mathsf{canonical}(v) = u_c \\
\mathsf{canonical}(\mathsf{fail}) & = & \mathsf{fail} \\
\mathsf{canonical}(op(e_1, \ldots, e_n)) & = & op(\mathsf{canonical}(e_1), \ldots, \mathsf{canonical}(e_n)) \\
\mathsf{canonical}(e_1\ e_2) & = & \mathsf{canonical}(e_1)\ \mathsf{canonical}(e_2) \\
\mathsf{canonical}(\langle c \rangle\ e) & = & \langle \mathsf{canonical}(c) \rangle\ \mathsf{canonical}(e) \\
\mathsf{canonical}(\langle \{x{:}T \mid e_1\}, e_2, v \rangle) & = & \langle \mathsf{canonical}(\{x{:}T \mid e_1\}), \mathsf{canonical}(e_2), \mathsf{canonical}(v) \rangle
\end{array}$$

Types $\boxed{\mathsf{canonical}(T) : T}$

$$\begin{array}{rcl}
\mathsf{canonical}(B) & = & B \\
\mathsf{canonical}(T_1 \to T_2) & = & \mathsf{canonical}(T_1) \to \mathsf{canonical}(T_2) \\
\mathsf{canonical}(\mathsf{Dyn}) & = & \mathsf{Dyn} \\
\mathsf{canonical}(\{x{:}T \mid e\}) & = & \{x{:}T \mid \mathsf{canonical}(e)\}
\end{array}$$

Contexts $\boxed{\mathsf{canonical}(\Gamma) : \Gamma}$

$$\begin{array}{rcl}
\mathsf{canonical}(\emptyset) & = & \emptyset \\
\mathsf{canonical}(\Gamma, x{:}T) & = & \mathsf{canonical}(\Gamma), x{:}\mathsf{canonical}(T)
\end{array}$$

Figure 4.25: Canonicalizing NAIVE terms

In order to put $\mathsf{canonical}(e)$ in the relation, we must know that it is well typed. Since some of the runtime typing terms require facts about derivations that will be hard to translate—in particular, T_TagValRefine and T_CheckNaive—we'll exclude them from the proof. Normal "source" terms shouldn't have any of these, anyway. Later on, when we've proved that naïve terms are logically related to their canonical translations, we'll have the evaluation derivations after all.

4.6.2 Lemma [Preservation for canonical]: Assuming that no refinement tags or active checking forms are present:

1. If $\vdash c : T_1 \rightsquigarrow T_2$ then $\vdash \mathsf{canonical}(c) : \mathsf{canonical}(T_1) \rightsquigarrow \mathsf{canonical}(T_2)$.
2. If $\vdash d : T_1 \rightsquigarrow T_2$ then $\vdash \mathsf{canonical}(d) : \mathsf{canonical}(T_1) \rightsquigarrow \mathsf{canonical}(T_2)$.
3. If $\Gamma \vdash u : T$ then $\mathsf{canonical}(\Gamma) \vdash \mathsf{canonical}(u) : \mathsf{canonical}(T)$.
4. If $\Gamma \vdash e : T$ then $\mathsf{canonical}(\Gamma) \vdash \mathsf{canonical}(e) : \mathsf{canonical}(T)$.
5. If $\vdash T$ then $\vdash \mathsf{canonical}(T)$.
6. If $\vdash \Gamma$ then $\vdash \mathsf{canonical}(\Gamma)$.

Proof: By simultaneous induction on the derivations, using Lemma 4.5.9 when merging coercions.

(C_Id) By C_Id.

(C_Compose) By IH (2) and IH (1) and Lemma 4.5.9.

(C_Fail) By C_Fail.

(C_Untag) By C_Untag and the IH.

(C_Tag) By C_Tag and the IH.

(C_Fun) By IH (1) and C_Fun.

(T_Const) By T_Const and IH (6).

(T_Abs) By IH (5) and IH (4), we can reapply T_Abs.

(T_Var) By IH (6) and T_Var.

(T_PreVal) By IH (3) and T_Val.

(T_TagVal) By IH (4), Lemma 4.5.9, and T_Val.

(T_TagValRefine) Contradictory—we assumed these weren't present.

(T_Op) By IH (4) and T_Op.

(T_App) By IH (4) and T_App.

(T_Coerce) By IH (1) and IH (4) and T_Coerce.

(T_Fail) By IH (6) and IH (5), we can reapply T_Fail.

(T_CheckNaive) Contradictory—we assumed these weren't present.

(WF_Base) By WF_Base.

(WF_Dyn) By WF_Dyn.

(WF_Fun) By IH (5) and WF_Fun.

(WF_Refine) By IH (4) and WF_Refine, noting that T is either B or Dyn, so $\mathsf{canonical}(T) = T$.

(WF_Empty) Immediate.

(WF_Extend) By IH (6) and IH (5), we can reapply WF_Extend.

□

The ultimate goal is soundness: if $\Gamma \vdash e : T$ then $\Gamma \vdash e \gtrsim\!\!\approx \mathsf{canonical}(e) : T$. The proof works in a few stages: first I define relations $\vdash c$ **ignorable** (coercions which are equivalent to Id or Fail) and $\vdash c$ **failable** (coercions which are equivalent to Fail). I define these relations in Figure 4.26. We then prove lemmas that allow us to easily work with ignorable and failable coercions (Lemma 4.6.6 and Lemma 4.6.7, respectively). Then we relate non-canonical coercions to canonical ones (using a separate inductive relation $\vdash c_1 \equiv c_2$, defined in Figure 4.26). We show that such related coercions take related values to related values (Lemma 4.6.8). We then prove a separate lemma showing that related coercions are logically related on related terms (Lemma 4.6.9). This is not a trivial extension of the similar lemma for values, due to coercion merges. With those lemmas to hand, we can finally prove soundness: that terms are related to their translations (Theorem 4.6.10). Don't worry–I'll explain the proof less tersely as we go.

We begin by establishing standard properties of the logical relation. We will use closure under evaluation (Lemmas 4.6.3 and 4.6.4), fast-path failure (Lemma 4.6.5), and determinism (Lemmas 4.3.4 and 4.5.15) extensively. If you've already read Chapter 2, the proof techniques here should be old hat.

4.6.3 Lemma [Expansion]: If $e_1 \longrightarrow_n^* e_1'$ and $e_2 \longrightarrow^* e_2'$ then $e_1' \gtrsim\!\!\approx^j e_2 : T$ implies $e_1 \gtrsim\!\!\approx^j e_2 : T$.

Proof: Let $m < j$. If e_1' diverges, so does e_1 by determinism (Lemma 4.3.4) and we are done.

Otherwise, we have $e_1' \longrightarrow_n^m r$. So there exists an m' such that $e_1 \longrightarrow_n^{m'} r$. If $m' > j$, then we are done vacuously; if $m' < j$ then we are done by assumption. □

4.6.4 Lemma [Contraction]: If $e_1 \longrightarrow_n^* e_1'$ and $e_2 \longrightarrow^* e_2'$ then $e_1 \gtrapprox^j e_2 : T$ implie $e_1' \gtrapprox^j e_2 : T$.

Proof: Let $m < j$. If e_1 diverges, so does e_1' by determinism (Lemma 4.3.4), and we are done.

Otherwise, we have $e_1 \longrightarrow_n^m r$. But since $e_1 \longrightarrow_n^* e_1'$, we know that $e_1' \longrightarrow_n^{m'} r$ for $m' < m$ by determinism (Lemma 4.3.4), so we are done by assumption. □

4.6.5 Lemma: Forall indices j, if $e_1 \longrightarrow_n^m \mathsf{fail}$ then $e_1 \gtrapprox^j e_2 : T$.

Proof: If e_1 goes to fail in $m \geq j$ steps, then we are done vacuously. If $m < j$, then are done by definition. □

Ignorable coercions can be freely added or removed to naïve terms while preserving logical relation to space-efficient terms. The I_Id rule obviously fits this bill; I_Fail is also acceptable, when we realize that Naive can fail more often than Efficient. I_BB captures the case of an injection of a base-value into type dynamic, with some possible extra coercions in the middle. I_FunFun and I_PredSame are similar. Note that I_PredPred can't have extra coercions in the middle—the coercion typing rules ensure that the only non-Fail coercion that can come after $\{x{:}T \mid e\}?$ is $\{x{:}T \mid e\}!$. The I_* rules try to capture the logic of similarly named N_* rules.

4.6.6 Lemma: If $\langle c_1; c_2\rangle\ v_1 \gtrapprox^j e_2 : T$ and $\vdash c_1$ **ignorable** then $\langle c_2\rangle\ v_1 \gtrapprox^j e_2 : T$.

Proof: By induction on $\vdash c_1$ **ignorable**.

Let $m < j$.

(I_Id) By definition, $\mathsf{Id}; c_2 = c_2$, so we are done immediately.

(I_Fail) $\langle \mathsf{Fail}; c_2\rangle\ v_1 \longrightarrow_n \mathsf{fail}$, and $\mathsf{fail} \gtrapprox^j e_2 : T$ by Lemma 4.6.5.

(I_BB) $\langle B!; c_1'; B?; c_2\rangle\ v_1 \longrightarrow_n \langle c_1'; B?; c_2\rangle\ v_{1\,B!}$ by F_TagB; the reduced term is then related to e_2 at T (Lemma 4.6.4). Since $\vdash c_1'$ **ignorable**, we can apply the IH and find that $\langle B?; c_2\rangle\ v_{1\,B!} \gtrapprox^j e_2 : T$. By F_TagBB and Lemma 4.6.4, we have $\langle c_2\rangle\ v_1 \gtrapprox^j e_2 : T$¿

(I_FunFun) $\langle \mathbf{Fun}!; c_1'; \mathbf{Fun}?; c_2\rangle\ v_1 \longrightarrow_n \langle c_1'; \mathbf{Fun}?; c_2\rangle\ v_{1\,\mathbf{Fun}!}$ by F_TagFun; the reduced term is then related to e_2 at T (Lemma 4.6.4). Since $\vdash c_1'$ **ignorable**, we can apply the IH and find that $\langle \mathbf{Fun}?; c_2\rangle\ v_{1\,B!} \gtrapprox^j e_2 : T$. By F_TagFunFun and Lemma 4.6.4, we have $\langle c_2\rangle\ v_1 \gtrapprox^j e_2 : T$¿

Ignorable coercions $\boxed{\vdash c\ \mathbf{ignorable}}$

$$\frac{}{\vdash \mathsf{Id}\ \mathbf{ignorable}}\ \text{I\_Id} \qquad \frac{}{\vdash \mathsf{Fail}\ \mathbf{ignorable}}\ \text{I\_Fail} \qquad \frac{\vdash c\ \mathbf{ignorable}}{\vdash B!; c; B?\ \mathbf{ignorable}}\ \text{I\_BB}$$

$$\frac{\vdash c\ \mathbf{ignorable}}{\vdash \mathbf{Fun}!; c; \mathbf{Fun}?\ \mathbf{ignorable}}\ \text{I\_FunFun} \qquad \frac{}{\vdash \{x{:}T \mid e\}?; \{x{:}T \mid e\}!\ \mathbf{ignorable}}\ \text{I\_PredPred}$$

$$\frac{\vdash c\ \mathbf{ignorable}}{\vdash \{x{:}T \mid e\}!; c; \{x{:}T \mid e\}?\ \mathbf{ignorable}}\ \text{I\_PredSame}$$

$$\frac{\vdash c_1\ \mathbf{ignorable} \quad \vdash c_2\ \mathbf{ignorable}}{\vdash c_1; c_2\ \mathbf{ignorable}}\ \text{I\_Concat}$$

Failable coercions $\boxed{\vdash c\ \mathbf{failable}}$

$$\frac{}{\vdash \mathsf{Fail}\ \mathbf{failable}}\ \text{L\_Fail} \qquad \frac{B \neq B' \quad \vdash c\ \mathbf{ignorable}}{\vdash B!; c; B'?\ \mathbf{failable}}\ \text{L\_BB}$$

$$\frac{\vdash c\ \mathbf{ignorable}}{\vdash B!; c; \mathbf{Fun}?\ \mathbf{failable}}\ \text{L\_BFun} \qquad \frac{\vdash c\ \mathbf{ignorable}}{\vdash \mathbf{Fun}!; c; B?\ \mathbf{failable}}\ \text{L\_FunB}$$

Relating coercions $\boxed{\vdash c_1 \equiv c_2}$ $\boxed{\vdash d_1 \equiv d_2}$ $\boxed{\vdash D_1 \equiv D_2}$

$$\frac{\vdash c_i'\ \mathbf{ignorable} \quad \vdash c_i \equiv d_i}{\vdash c_0'; c_1; c_1'; c_2; \ldots; c_{n-1}'; c_n; c_n' \equiv d_1; \ldots; d_n}\ \text{R\_Composite} \qquad \frac{\vdash c\ \mathbf{ignorable}}{\vdash c \equiv \mathsf{Id}}\ \text{R\_Id}$$

$$\frac{\begin{array}{l}\vdash c_i'\ \mathbf{ignorable}\\ \vdash (c_{1\,n\,1}; \ldots; c_{111}) \mapsto (c_{112}; \ldots; c_{1\,n\,2}) \equiv c_{21} \mapsto c_{22}\end{array}}{\vdash (c_{111} \mapsto c_{112}); c_1'; \ldots; c_n'; (c_{1\,n\,1} \mapsto c_{1\,n\,2}) \equiv c_{21} \mapsto c_{22}}\ \text{R\_Fun} \qquad \frac{\vdash c_1\ \mathbf{failable}}{\vdash c_1'; c_1; c_2' \equiv c_2}\ \text{R\_Fail}$$

$$\frac{\vdash D \equiv D}{\vdash D! \equiv D!}\ \text{R\_Tag} \qquad \frac{\vdash D \equiv D}{\vdash D? \equiv D?}\ \text{R\_Untag}$$

$$\frac{}{\vdash B \equiv B}\ \text{R\_DB} \qquad \frac{}{\vdash \mathbf{Fun} \equiv \mathbf{Fun}}\ \text{R\_DFun} \qquad \frac{x{:}\mathsf{Bool} \vdash e_1 \gtrapprox e_2 : \mathsf{Bool}}{\vdash \{x{:}T \mid e_1\} \equiv \{x{:}T \mid e_2\}}\ \text{R\_DPred}$$

Figure 4.26: Relating Naive coercions to canonical Efficient coercions

(I_PredPred) $\langle \{x{:}T \mid e\}?; \{x{:}T \mid e\}!; c_2\rangle\ v_1 \longrightarrow_n \langle \{x{:}T \mid e\}!; c_2\rangle\ \langle \{x{:}T \mid e\}, e[v_1/x], v_1\rangle$ by F_Check. By type soundness (Theorem 4.3.9), we know that $e[v_1/x]$ either reduces a to value, reduces to fail, or diverges. In either of the last two cases, we are done by the definition of the logical relation or Lemma 4.6.5, respectively.

Suppose that $e[v_1/x] \longrightarrow_n^* v$; the value v must be either $\mathsf{true}_{\mathsf{Id}}$ or $\mathsf{false}_{\mathsf{Id}}$, since $\emptyset \vdash e[v_1/x] : \mathsf{Bool}$. In the latter case we are done by Lemma 4.6.5, just like for when the whole term reduced to fail.

So then $\langle \{x{:}T \mid e\}!; c_2\rangle\ \langle \{x{:}T \mid e\}, \mathsf{true}_{\mathsf{Id}}, v_1\rangle \longrightarrow_n \langle \{x{:}T \mid e\}!; c_2\rangle\ v_{1\,\{x:T|e\}?}$ by F_CheckOK. We can then step by F_TagPredPred and apply Lemma 4.6.4 to find that $\langle c_2\rangle\ v_1 \gtrsim^j e_2 : T$.

(I_PredSame) $\langle \{x{:}T \mid e\}!; c_1'; \{x{:}T \mid e\}?; c_2\rangle\ v_1 \gtrsim^j e_2 : T$. By inversion of the typing of v_1, we know that it must be of the form $v_{1\,\{x:T|e\}?}'$, and furthermore $e[v_1/x] \longrightarrow_n^* \mathsf{true}_{\mathsf{Id}}$.

Lemma 4.6.4 and F_PredPred give us $\langle c_1'; \{x{:}T \mid e\}?; c_2\rangle\ v_1' \gtrsim^j e_2 : T$, so we can apply the IH on $\vdash c_1'$ **ignorable** to find $\langle \{x{:}T \mid e\}?; c_2\rangle\ v_1' \gtrsim^j e_2 : T$. We then step by F_TagPredPred, then by F_Check. Since we know that $e[v_1'/x] \longrightarrow_n^* \mathsf{true}_{\mathsf{Id}}$, we can step the whole term using that relation and F_CheckOK to find $\langle c_2\rangle\ v_{1\,\{x:T|e\}?}' = \langle c_2\rangle\ v_1 \gtrsim^j e_2 : T$ by Lemma 4.6.4.

(I_Concat) $\langle c_{11}; c_{12}; c_2\rangle\ v_1 \gtrsim^j e_2 : T$. By the IH on $\vdash c_{11}$ **ignorable**, we find that

$$\langle c_{12}; c_2\rangle\ v_1 \gtrsim^j e_2 : T$$

By the IH on $\vdash c_{12}$ **ignorable**, we find that $\langle c_2\rangle\ v_1 \gtrsim^j e_2 : T$.

□

We prove a similar lemma that *failable* coercions always fail. The L_* rules try to capture the logic of similarly named N_Fail* rules.

4.6.7 Lemma: If $\vdash c_1$ **failable**, then $\langle c_1'; c_1; c_2'\rangle\ v_1 \gtrsim^j e_2 : T$.

Proof: We begin by using type soundness (Theorem 4.3.9) to find that either $\langle c_1'; c_1; c_2'\rangle\ v_1 \longrightarrow_n^* \langle c_1; c_2'\rangle\ v_1'$, the whole left-hand side reduces to fail (and we are done by Lemma 4.6.5), or the whole left-hand side diverges (and we are done by definition). In the last two cases we are done, so we consider the first case.

We proceed by induction on $\vdash c_1$ **failable**.

(L_Fail) We step by F_Fail, and have $\mathsf{fail} \gtrsim^j e_2 : T$ by Lemma 4.6.5.

(L_BB) We step by F_TagB followed by F_TagBFailB, and then we have $\mathsf{fail} \gtrsim^j e_2 : T$ by definition; we finish by expansion (Lemma 4.6.3).

(L_BFun) We step by F_TagB followed by F_TagBFailFun, and we have $\mathsf{fail} \gtrsim^j e_2 : T$ by Lemma 4.6.5; we finish by expansion (Lemma 4.6.3).

(L_FunB) We step by F_TagFun followed by F_TagFunFailB, and we have $\mathsf{fail} \gtrsim^j e_2 : T$ by Lemma 4.6.5; we finish by expansion (Lemma 4.6.3)

□

With ignorable and failable coercions, we can characterize *all* non-canonical coercions, relating them to canonical coercions. The relation $\vdash c_1 \equiv c_2$ relates a non-canonical coercion c_1 to a canonical coercion c_2. Note that this inductively defined relation isn't the same thing as the logical relation—it's a separate invariant relation, used to relate coercions to their canonical forms. Only when relating refinements does this invariant relation dip into the logical relation, to relate the predicates. First we show that well typed coercions c in Naive are related to $\mathsf{canonical}(c)$ in Efficient. Then we'll use this general relation to relate *in the logical relation* how coercion forms work on logically related values.

We now show that *any* coercions $\vdash c_1 \equiv c_2$ yield related results when applied to related values. I defined the relation $\vdash c_1 \equiv c_2$ because this lemma is easier to prove on the relation than on the canonical function itself.

4.6.8 Lemma [Relating canonical coercions]:
If $v_1 \gtrsim^j v_2 : T_1$ and $\vdash c_1 : T_1 \rightsquigarrow T_2$ and $\vdash c_1 \equiv c_2$, then $\langle c_1 \rangle\ v_1 \gtrsim^j \langle c_2 \rangle\ v_2 : T_2$.

Proof: By induction on $\vdash c_1 \equiv c_2$ using ignorability (Lemma 4.6.6) and failability (Lemma 4.6.7) extensively.

(R_Id) Observe that $T_1 = T_2$. By ignorability (Lemma 4.6.6), $\langle c_1 \rangle\ v_1 \longrightarrow_n^* \langle \mathsf{Id} \rangle\ v_1$; by F_TagId, $\langle \mathsf{Id} \rangle\ v_1 \longrightarrow_n v_1$. On the right-hand side, E_TagId steps $\langle \mathsf{Id} \rangle\ v_2 \longrightarrow v_2$, and we have $v_1 \gtrsim^j v_2 : T_1$ by assumption. We are done by expansion (Lemma 4.6.3).

(R_Fail) By failability (Lemma 4.6.7).

(R_Composite) We want to show

$$\langle c_0'; c_1; c_1'; c_2; \ldots; c_{n-1}'; c_n; c_n' \rangle\ v_1 \gtrsim^j \langle d_1; \ldots; d_n \rangle\ v_2 : T_2$$

given that $\vdash c_i'$ **ignorable** and $\vdash c_i \equiv d_i$.

We go by induction on n. The $n = 0$ case is already covered by R_Id above.

We first use ignorability (Lemma 4.6.6), and we need to show:

$$\langle c_1; c_1'; c_2; \ldots; c_{n-1}'; c_n; c_n' \rangle\ v_1 \gtrsim^j \langle d_1; \ldots; d_n \rangle\ v_2 : T_2$$

We now go by cases on $\vdash c_1 \equiv d_1$ for the first hypothesis.

(R_Fail) By failability (Lemma 4.6.7).

(R_Tag) By well typing of the LR, the expressions must have the same type $T_1 = B$ (if the tag is $B!$), $T_1 = \mathsf{Dyn}{\to}\mathsf{Dyn}$ (if the tag is **Fun**!), or $T_1 = \{x{:}T_1 \mid e\}$ (if the tag is $\{x{:}T_1 \mid e\}!$).

Whatever the case, we know by $v_1 \gtrsim^j v_2 : T_1$ that v_1 and v_2 are similarly tagged, so we can step each side by F_TagB, F_TagFun, or F_PredPred on the left (and E_Tag on the right). We then finish by the first IH and expansion (Lemma 4.6.3).

(R_Untag) By well typing of the LR, the expressions must have the same type $T_1 = \mathsf{Dyn}$ (if the tag is $B?$ or **Fun**?), $T_1 = B$ (if the tag is $\{x{:}B \mid e\}?$), or $T_1 = \mathsf{Dyn}$ (if the tag is $\{x{:}\mathsf{Dyn} \mid e\}?$).

In the first case, we step by one of the following pairs on each side:

F_TagBB/E_Tag
F_TagFunFun/E_Tag
F_TagBFailB/E_TagFail
F_TagBFailFun/E_TagFail
F_TagFunFailB/E_TagFail

For the first two, we are done by value relation we have and the first IH and expansion (Lemma 4.6.3); for the latter three, we are done by having fail on the left (Lemma 4.6.5).

In the last two cases, both sides step to checking forms, which we know coterminate at *all* indices, and we finish by the first IH.

(R_Fun) We prove by a separate induction that: If

* $v_1 \gtrsim^j u_{1\mathsf{Id}} : T_1{\to}T_2$,
* $\vdash ((c_{1\,n\,1}; \ldots; c_{111}) \mapsto (c_{112}; \ldots; c_{1\,n\,2})) : (T_1{\to}T_2) \rightsquigarrow (T_1'{\to}T_2')$, and
* $\vdash ((c_{1\,n\,1}; \ldots; c_{111}) \mapsto (c_{112}; \ldots; c_{1\,n\,2})) \equiv c_{21} \mapsto c_{22}$

then $v_{1(c_{111}\mapsto c_{112})\ldots(c_{1\,n\,1}\mapsto c_{1\,n\,2})} \gtrsim^j u_{1\,c_{21}\mapsto c_{22}} : T_1'{\to}T_2'$. We prove it by induction on n, repeatedly unwrapping the tags on the left-hand side, applying F_Merge, and then the outer IH.

We step through the ignorable coercions (Lemma 4.6.6), eventually yielding a value such that the second IH relates the stacked function tags on the left to the merged ones on the right. We then finish by expansion (Lemma 4.6.3).

□

In Chapter 2, a similar characterization of casts is sufficient: once we'd related λ_C contracts and λ_H casts, we had what we needed to handle the corresponding cases of the final proof. But that strategy won't work here: congruent reductions under coercions may introduce coercions on the outside; these extra coercions will merge into c_1 and c_2 (by F_Merge or E_Merge), possibly disrupting $\vdash c_1 \equiv c_2$. Consider the T_Merge case of the proof of soundness. Even if we have $\langle c_1 \rangle\ v_1 \gtrsim^j \langle c_2 \rangle\ v_2 : T$

and $e_1 \longrightarrow_n^* v_1$ and $e_2 \longrightarrow^* v_2$, we can't just put the derivations in and be done: what if e_1 or e_2 produce terms like $\langle c_1' \rangle\ e_1'$ or $\langle c_2' \rangle\ e_2'$ as they evaluate? Then F_Merge or E_Merge will fire, and we won't know anything about the related coercions, nor do we know how many extra steps may have been taken. Accounting for the steps, it turns out, is not particularly hard: if too many new steps are added, the terms are vacuously in the relation; if not, then we merely need to account for the extra merged coercions.

In short: having Lemma 4.6.8 on values doesn't immediately tell us anything about how coercions work on arbitrary terms; we must prove that separately.

4.6.9 Lemma [Relating coercions with merges]: If $e_1 \gtrapprox^j e_2 : T_1$ and $\vdash c_1 : T_1 \rightsquigarrow T_2$ and $\vdash c_1 \equiv c_2$, then $\langle c_1 \rangle\ e_1 \gtrapprox^j \langle c_2 \rangle\ e_2 : T_2$.

Proof: First, we can ignore the cases where $e_1 \longrightarrow_n^m \mathsf{fail}$ or e_1 diverges—those are immediately related, since $\langle c_1 \rangle\ e_1$ also diverges or goes to fail(by Lemma 4.3.13).

So $e_1 \longrightarrow_n^m v_1$ and $e_2 \longrightarrow^* v_2$, and by definition, $v_1 \gtrapprox^j v_2 : T_1$. By coercion congruence(Lemma 4.3.13 and Lemma 4.5.22), there exist e_1' and e_2' such that (a) $\langle c_1 \rangle\ e_1 \longrightarrow_n^* e_1'$ and $\langle c_1 \rangle\ v_1 \longrightarrow_n^* e_1'$, and (b) $\langle c_2 \rangle\ e_2 \longrightarrow^* e_2'$ and $\langle c_2 \rangle\ v_2 \longrightarrow^* e_2'$. But we know that $\langle c_1 \rangle\ v_1 \gtrapprox^j \langle c_2 \rangle\ v_2 : T_2$ by Lemma 4.6.8, so we are done by contraction and expansion(Lemma 4.6.4 and Lemma 4.6.3). □

4.6.10 Theorem [Soundness]:

- If $\Gamma \vdash u : T$ then $\Gamma \vdash u_{\mathsf{Id}} \gtrsim \mathsf{canonical}(u_{\mathsf{Id}}) : T$.
- If $\Gamma \vdash e : T$ then $\Gamma \vdash e \gtrsim \mathsf{canonical}(e) : T$.
- If $\vdash c : T_1 \rightsquigarrow T_2$ then $\vdash c \equiv \mathsf{canonical}(c)$.
- If $\vdash T$ then $\forall j.\ T \gtrsim^j \mathsf{canonical}(T)$.

Proof: By lexicographic induction on the typing derivation and the size of the term (v or e, respectively), using Lemma 4.6.9 in the T_Coerce case. We use Lemma 4.6.8 in the T_TagVal, T_TagValRefine, and coercion cases.

Let a j be given. In all cases, we begin by letting $\Gamma \models_\succ^j \delta$, so we must show that $\delta_1(e) \gtrapprox^j \delta_2(\mathsf{canonical}(e)) : T$.

(T_Const) Immediate—$k_{\mathsf{Id}} \gtrsim^j k_{\mathsf{Id}} : B$ for any j.

(T_Abs) We must show that

$$\delta_1(\lambda x{:}T_1.\ e_{1\mathsf{Id}}) \gtrsim^j \delta_2(\lambda x{:}\mathsf{canonical}(T_1).\ \mathsf{canonical}(e_1)_{\mathsf{Id}}) : T_1 \to T_2$$

given that $\Gamma, x{:}T_1 \vdash e_1 : T_2$. Let $m < j$, and let $v_1 \gtrsim^m v_2 : T_1$. We have

$$\delta_1(\lambda x{:}T_1.\ e_{1\mathsf{Id}})\ v_1 \longrightarrow_n \delta_1(e_1)[v_1/x]$$
$$\delta_2(\lambda x{:}\mathsf{canonical}(T_1).\ \mathsf{canonical}(e_1)_{\mathsf{Id}})\ v_2 \longrightarrow \delta_2(\mathsf{canonical}(e_1))[v_2/x]$$

We must now show that these two terms are related at m. We can apply IH (4.6.10) on $\Gamma, x{:}T_1 \vdash e_1 : T_2$ at the index m using the closing substitution $\Gamma, x{:}T_1 \models_\succ^m \delta[v_1, v_2/x]$.

(T_PreVal) By IH (4.6.10).

(T_TagVal) We must show that

$$\delta_1(v_{1\,d}) \gtrsim^j \delta_2(u_{1\,c \Downarrow \mathsf{canonical}(d)}) : T_2$$

where $\mathsf{canonical}(v_1) = u_{1\,c}$ given that $\delta_1(v_1) \gtrapprox^j \delta_2(\mathsf{canonical}(u_1)_c) : T_1$, knowing that $d \neq \{x{:}T_1 \mid e\}$? and $\vdash d : T_1 \rightsquigarrow T_2$

By IH (4.6.10), we find that $\vdash d \equiv \mathsf{canonical}(d)$. We can then apply Lemma 4.6.8 to find that $\langle d \rangle\ \delta_1(v_1) \gtrapprox^j \langle \mathsf{canonical}(d) \rangle\ \delta_2(\mathsf{canonical}(u_1)_c) : T_2$. Since we originally had $v_{1\,d}$, we know that d must be a value tag, we can step both sides (by F_TagB, F_TagFun, F_TagFunWrap or F_TagPredPred on the left; E_Tag or E_Fun on the right) to find

$$\begin{array}{l} \langle d \rangle\ \delta_1(v_1) \longrightarrow_n^* \delta_1(v_{1\,d}) \\ \langle \mathsf{canonical}(d) \rangle\ \delta_2(\mathsf{canonical}(u_1)_c) \longrightarrow^* \delta_2(\mathsf{canonical}(u_1)_{c \Downarrow \mathsf{canonical}(d)}) \end{array}$$

We are then done by expansion (Lemma 4.6.3).

(T_TagValRefine) As for the previous case, stepping through related checking forms to find related results. (Or, possibly, by finding fail on the left and ignoring the right entirely, by Lemma 4.6.5.)

(T_Var) By definition, $\delta_1(x) \gtrapprox^j \delta_2(x) : T$.

(T_Op) By IH (4.6.10) we can either find divergence or failure on the left, or each argument reduces to a value. In this case, we know that operations are first-order and don't take dynamic values, so we get the exact same output on both sides—which must then be related.

(T_App) By IH (4.6.10) and the definition of the logical relation at function types.

(T_Coerce) By Lemma 4.6.9 and IH (4.6.10).

(T_Fail) Immediate by definition.

(T_Check) By IH (4.6.10).

(C_Id) By R_Id.

(C_Fail) By R_Fail, L_Fail, and I_Id.

(C_Compose) We have

$$\begin{array}{l} \vdash d_1 \equiv \mathsf{canonical}(d_1) \\ \vdash d_2; \ldots; d_n \equiv \mathsf{canonical}(d_2; \ldots; d_n) \end{array}$$

It remains to show that concatenation on the left is related to merging on the right.

We go by cases on d_1. Throughout the analysis, we will examine the rule used to find $\vdash d_2; \ldots; d_n \equiv \mathsf{canonical}(d_2; \ldots; d_n)$—when the rule was R_Fail, we are immediately done, since the merge will produce Fail on the right and the left will always satisfy R_Fail.

($d_1 = B!$) Either $\mathsf{canonical}(d_1) \Downarrow \mathsf{canonical}(d_2; \ldots; d_n)$ begins with $B!$ or it doesn't. If it does, then we invert $\vdash d_2; \ldots; d_n \equiv \mathsf{canonical}(d_2; \ldots; d_n)$:

(R_Id) We are done by R_Composite.

(R_Fail) We are done by R_Fail.

(R_Composite) We are done by R_Composite, with $c_0' = \mathsf{Id}$ and $c_1 = B!$ and $c_1'; \ldots; c_n' = d_2; \ldots; d_n$.

If it doesn't, then we go by inversion of the derivation of

$$\mathsf{canonical}(d_1) * \mathsf{canonical}(d_2; \ldots; d_n) \Rightarrow \ldots$$

We exclude obviously contradictory cases (N_Canonical, N_FailL, N_Phi with $D = \mathbf{Fun}$ or $D = \{x{:}T \mid e\}$, N_FunFailB, N_PredPred, N_Fun).

(N_FailR) The only relation rule that could have applied is R_Fail, so we are done again by R_Fail.

(N_Phi with $D = B$) By R_Composite, since $\vdash B!; c; B?\,\mathbf{ignorable}$.

(N_BFailB) By R_Fail and L_BB.

(N_BFailFun) By R_Fail and L_BFun.

($d_1 = \mathbf{Fun}!$) As for $B!$.

($d_1 = \{x{:}T \mid e\}!$) If d_1 isn't at the beginning of the coercion, then it must be that $\{x{:}T \mid e\}?$ begins $\mathsf{canonical}(d_2; \ldots; d_n)$.

If d_1 remains at the beginning, then we go by R_Composite or R_Fail, depending on how we found the derivation of $\vdash d_2; \ldots; d_n \equiv \mathsf{canonical}(d_2; \ldots; d_n)$: we use the former if R_Id or R_Composite was used, the latter if R_Fail was used.

We go by cases on the derivation of $\vdash d_2; \ldots; d_n \equiv \mathsf{canonical}(d_2; \ldots; d_n)$.

(R_Id) Contradictory—we assumed that the final merge result began with $\{x{:}T \mid e\}?$.

(R_Fail) By R_Fail.

(R_Composite) By R_Composite, noting that

$$\vdash \{x{:}T \mid e\}!; c_0'; \{x{:}T \mid e\}?\,\mathbf{ignorable}$$

by I_PredSame.

($d_1 = B?$) It must be the case that the merged coercion starts with $B?$, since no merge eliminates $B?$ on the left.

If R_Id or R_Composite derived $\vdash d_2; \ldots; d_n \equiv \mathsf{canonical}(d_2; \ldots; d_n)$, we are done by R_Composite. If R_Fail was used, we're done immediately.

($d_1 = \mathbf{Fun}?$) As for $B?$.

($d_1 = \{x{:}T \mid e\}?$) The outermost rule applying to find $\{x{:}T \mid \mathsf{canonical}(e)\}? \Downarrow \mathsf{canonical}(d_2; \ldots; d_n)$ must be either N_FailR, N_Canonical, or N_PredPrd. In the first two cases, we are done (by R_Composite, R_Tag, and R_DPred—or by R_Fail). In the third case, we are done by the IH with R_Composite and R_Tag, with $c_0' = \mathsf{Id}$. In the fourth case, we are done by the IH (since $\vdash \{x{:}T \mid e\}?; \{x{:}T \mid e\}!\,\mathbf{ignorable}$, and we can use either R_Id or R_Composite with I_Concat).

($d_1 = c_{11} \mapsto c_{12}$) The outermost rule applying in $\mathsf{canonical}(c_{11}) \mapsto \mathsf{canonical}(c_{12}) \Downarrow \mathsf{canonical}(d_2; \ldots; d_n)$ must be one of N_FailR, N_Canonical, or N_Fun.

In the first two cases we are done, either by R_Composite with R_Fun or by R_Fail. In the third case, we are done by the IH with R_Composite and R_Fun. In the fourth case, we are done by the IH, noticing that the relation on the right comes from R_Composite with an initial R_Fun, so we can fold $c_{11} \mapsto c_{12}$ into that R_Fun derivation and reconstruct a new R_Composite derivation.

(WF_Dyn) Immediate

(WF_Base) Immediate

(WF_Fun) By IH (4.6.10).

(WF_Refine) We have $T \gtrsim^j T$ immediately. As IH (4.6.10) on $x{:}T \vdash e : \mathsf{Bool}$, we have $x{:}T \vdash e \gtrapprox \mathsf{canonical}(e) : \mathsf{Bool}$. Let $m < j$ and $v_1 \gtrsim^m v_2 : T$ be given; we can then find $e[v_1/x] \gtrapprox^m \mathsf{canonical}(e)[v_2/x] : \mathsf{Bool}$ by instantiating at m and δ.

□

The definition of $\emptyset \vdash e_1 \gtrapprox e_2 : T$ gives us approximate observational equivalence: either e diverges, e reduces to fail, or $e \longrightarrow_n^* v_1$ and $\mathsf{canonical}(e) \longrightarrow^* v_2$ such that $v_1 \gtrsim^j v_2 : T$ for arbitrary j. (I write this result without indices because the definition of $\Gamma \vdash e_1 \gtrapprox e_2 : T$ quantifies over all indices.) Note that for base values, we have exactly the same result on both sides.

4.7 Space efficiency

The structure of my space-efficiency proof is largely the same as in prior work. Coercion size is broken down by the order of the types involved; the maximum size of any coercion is $|\text{largest coercion}| \cdot 2^{\text{tallest type}}$. Inspecting the canonical coercions, the largest is $\{x{:}\mathsf{Dyn} \mid e\}!; \mathbf{Fun}?; c_1 \mapsto c_2; \mathbf{Fun}!; \{x{:}\mathsf{Dyn} \mid e'\}?$, with a size of 5. The largest possible canonical coercion therefore has size $M = 5 \cdot 2^h$.

Formally, observe that merging canonical coercions c_1 and c_2 either produces a smaller coercion or $c_1; c_2$ is canonical(and has size $\text{size}(c_1) + \text{size}(c_2)$).

4.7.1 Lemma [Merge reduces size]: If $c_1 * c_2 \Rightarrow c_3$, then either:

- $\text{size}(c_1) + \text{size}(c_2) > \text{size}(c_3)$, or
- $c_3 = c_1; c_2$ is canonical.

Proof: By induction on the derivation of $c_1 * c_2 \Rightarrow c_3$. Note that in either case, $\text{size}(c_1) + \text{size}(c_2) \geq \text{size}(c_3)$.

(N_CANONICAL) $c_1; c_2$ is canonical.

(N_FAILL) We have $\mathsf{Fail} * c_2 \Rightarrow \mathsf{Fail}$, with $1 + \text{size}(c_2) > 1$.

(N_FAILR) We have $c_1 * \mathsf{Fail} \Rightarrow \mathsf{Fail}$, with $\text{size}(c_1) + 1 > 1$.

(N_PHI) We have $\vdash c_1; D! : T_1 \rightsquigarrow \mathbf{tgt}(D)$ and $\vdash D?; c_2 : \mathbf{tgt}(D) \rightsquigarrow T_3$. By the IH, $\text{size}(c_1) + \text{size}(c_2) \geq \text{size}(c_3)$, so we immediately have $\text{size}(c_1) + 1 + \text{size}(c_2) + 1 > \text{size}(c_3)$.

(N_BFAILB) We have $\vdash c_1; B! : T_1 \rightsquigarrow \mathsf{Dyn}$ and $\vdash B'?; c_2 : \mathsf{Dyn} \rightsquigarrow T_3$. It is immediate that size(Fail) is smaller.

(N_BFAILFUN) We have $\vdash c_1; B! : T_1 \rightsquigarrow \mathsf{Dyn}$ and $\vdash \mathbf{Fun}?; c_2 : \mathsf{Dyn} \rightsquigarrow T_3$. It is immediate that size(Fail) is smaller.

(N_FUNFAILB) We have $\vdash c_1; \mathbf{Fun}! : T_1 \rightsquigarrow \mathsf{Dyn}$ and $\vdash B?; c_2 : \mathsf{Dyn} \rightsquigarrow T_3$. It is immediate that size(Fail) is smaller.

(N_PREDPRED) We have $\vdash c_1; \{x{:}T \mid e\}? : T_1 \rightsquigarrow \{x{:}T \mid e\}$ and $\vdash \{x{:}T \mid e\}!; c_2 : \{x{:}T \mid e\} \rightsquigarrow T_3$, where T is either B or Dyn. By the IH, $\text{size}(c_1) + \text{size}(c_2) \geq \text{size}(c_3)$, so we immediately have $\text{size}(c_1) + 1 + 1 + \text{size}(c_2) \geq \text{size}(c_3)$.

(N_FUN) We have:

$$\begin{array}{l} \vdash c_1; (c_{11} \mapsto c_{12}) : T_1 \rightsquigarrow (T_{21} \to T_{22}) \\ \vdash (c_{21} \mapsto c_{22}); c_2 : (T_{21} \to T_{22}) \rightsquigarrow T_3 \end{array}$$

By the IH:

$$\begin{array}{rcl} \text{size}(c_{21}) + \text{size}(c_{11}) & \geq & \text{size}(c_{31}) \\ \text{size}(c_{12}) + \text{size}(c_{22}) & \geq & \text{size}(c_{33}) \\ \text{size}(c_1) + \text{size}((c_{31} \mapsto c_{32}); c_2) & \geq & \text{size}(c_3) \end{array}$$

Now we can see that:

$$\begin{array}{rcl} \text{size}(c_{11} \mapsto c_{12}) + \text{size}(c_{21} \mapsto c_{22}) & = & 1 + \text{size}(c_{11}) + \text{size}(c_{12}) + \\ & & 1 + \text{size}(c_{21}) + \text{size}(c_{22}) \\ & > & 1 + \text{size}(c_{31}) + \text{size}(c_{32}) \end{array}$$

Now we can finally conclude that:

$$\begin{array}{rcl} \text{size}(c_1; (c_{11} \mapsto c_{12})) + \text{size}((c_{21} \mapsto c_{22}); c_2) & > & \text{size}(c_1) + \text{size}((c_{31} \mapsto c_{32}); c_2) \\ & > & \text{size}(c_3) \end{array}$$

□

Rules with merges (and E_Merge in particular) don't increase the size of the largest coercion in the program. Applying this lemma across an evaluation $e \longrightarrow^* e'$, we can see that no coercion ever exceeds the size of the largest coercion in e. If M is the size of the largest coercion, then there is at most an M-fold space overhead of coercions. But this size bound is galactic; I find it hard to believe that this overhead is observable in practice. A much more interesting notion of space efficiency—not studied here—is to determine implementation schemes for space-efficient layout of coercions in memory and time-efficient merges of coercions. I believe that explicitly enumerating the canonical coercions is a step towards this goal: the canonical coercions in Table 4.2 are exactly those which must be represented.

4.8 Conclusion

Space-efficiency is attainable for contract languages that cover the whole spectrum, from dynamic types to refinement types. While canonical coercions may skip checks—and behave slightly differently from naïve, inefficient implementations—we still have the guarantee that when the naïve implementation produces a value, the space-efficient one will produce a behaviorally equivalent one. Resolving the space inefficiency of naïve contract systems is an important first step in making contracts amenable to pervasive use.

Chapter 5

Related work

> Literature does not exist in a vacuum.
>
> *ABC of Reading*
> Ezra Pound

We begin by surveying the field in Section 5.1, with a focus on λ_{H} in Section 5.1.2. We then focus more closely on work related to F_{H} and space-efficiency in Sections 5.2 and 5.3, respectively.

5.1 Contracts: a survey

Conferences in recent years have seen a profusion of papers on higher-order contracts and related features. This is all to the good, but for newcomers to the area it can be a bit overwhelming, especially given the great variety of technical approaches. To help reduce the level of confusion, in Table 5.1 I summarize the important points of comparison between a number of systems that are closely related to ours. This table is an updated version of those in Greenberg et al. [34, 35]. A similar comparison for gradual typed systems—which necessarily has some overlap with the material in this table—is in Table 5.2.

The largest difference is between latent and manifest treatments of contracts—i.e., whether contract checking (under whatever name) is a completely dynamic matter or whether it leaves a "trace" that the type system can track.

Another major distinction (labeled "dep" in the figure) is the presence of dependent contracts or, in manifest systems, dependent function types. Latent systems with dependent contracts also vary in whether their semantics is lax or picky.

Next, most contract calculi use a standard call-by-value order of evaluation ("eval order" in the figure). Notable exceptions include those of Hinze et al. [41], which is embedded in Haskell, Flanagan [28], which uses a variant of call-by-name, and Knowles and Flanagan [44], which uses full β-reduction (more on this below).

Latent systems

	FF02 (1)	HJL06 (2)	GF07 λ_C (3)	BM06 (4)	DFFF11 (5)	λ_C (Ch. 2)
dep (6)	✓ lax	✓ picky	×	(7)	✓ indy	✓ either
eval order	CBV	lazy	CBV	CBV	CBV	CBV
blame (8)	$\Uparrow l$	$\Uparrow l$	$\Uparrow l$	$\Uparrow l$ or $\bot$	$\Uparrow l$	$\Uparrow l$
checking (9)	if	if	◯	active	active	active
typing (10)	✓	✓	✓	n/a	✓	✓
any con (11)	✓	✓	✓	✓	✓	✓

Manifest systems

	GF07 λ_H (3)	F06 (12)	KF10 (13)	WF09 (14)	OTMW04 (15)	BGIP11 (16)	λ_H (Ch. 2)	F_H (Ch. 3)	Eff. (Ch. 4)
dep (6)	×	✓	✓	×	✓	✓	✓	✓	×
eval order	CBV	CBN(17)	full β	CBV	CBV	CBV	CBV	CBV	CBV
blame (8)	$\Uparrow l$	stuck	stuck	$\Uparrow l$	$\Uparrow$	$\Uparrow l$	$\Uparrow l$	$\Uparrow l$	$\Uparrow$
checking (9)	◯	◯	active	active	if	active	active	active	active
typing (10)	×	×	✓	✓	✓	✓	✓	✓	✓
any con (11)	✓	✓	✓	✓	×	✓	✓	✓	✓
poly (18)	×	×	×	×	×	✓	×	✓	×
space (19)	×	×	×	×	×	×	×	×	✓

(1) Findler and Felleisen [26]. (2) Hinze et al. [41]. (3) Gronski and Flanagan [36]. (4) Blume and McAllester [11]. (5) Dimoulas et al. [22]. (12) Flanagan [28]. (13) Knowles and Flanagan [44]. (14) Wadler and Findler [78]. (15) Ou et al. [51]. (16) Belo et al. [8]. This is not the same as the F_H offered in Chapter 3. I omit Cast and Naive from this analysis. (6) Does the system include dependent contracts or function types (✓) or not (×) and, for latent systems, is the semantics lax or picky? (See below for more on "indy" checking.) (7) An "unusual" form of dependency, where negative blame in the codomain results in nontermination. (17) A nondeterministic variant of CBN. (8) Do failed contracts raise labeled blame ($\Uparrow l$), raise blame without a label ($\Uparrow$), get stuck, or sometimes raise blame and sometimes diverge ($\bot$)? (9) Is contract or cast checking performed using an "active check" syntactic form (active), an "if" construct with a refined typing rule (if), or "inlined" by making the operational semantics refer to its own reflexive and transitive closure (◯)? (10) Is the typing relation monotonic, i.e., is the typing relation known to be uniquely defined? (11) Are arbitrary user-defined boolean functions allowed as contracts or refinements (✓), or only built-in ones (×)? (18) Does the type system support polymorphism? (19) Is the operational semantics space efficient?

Table 5.1: Comparison between contract systems

Another point of variation ("blame" in the figure) is how contract violations or cast failures are reported—by raising an exception or by getting stuck. I return to this below.

The next two rows in the table ("checking" and "typing") concern more technical points in the papers most closely related to the work in Chapter 2. In both Gronski and Flanagan [36] and Flanagan [28], the operational semantics checks casts "all in one go":

$$\frac{s_2\{x := k\} \to_h^* \mathsf{true}}{\langle\{x{:}B \mid s_1\} \Rightarrow \{x{:}B \mid s_2\}\rangle^l\ k \to_h k}$$

Such rules are formally awkward, and in any case they violate the spirit of a small-step semantics. Also, the formal definitions of λ_{H} in both Gronski and Flanagan [36] and Flanagan [28] involve a circularity between the typing, subtyping, and implication relations. Knowles and Flanagan [44] improve the technical presentation of λ_{H} in both respects. In particular, they avoid circularity (as I do) by introducing a denotational interpretation of types and maintain small-step evaluation by using a new syntactic form of "partially evaluated casts" (like most of the other systems).

5.1.1 Refinement types and contracts

Refinement types, in fact, first referred to refinements of datatypes in particular [30], though the term has been appropriated to mean types of the form $\{x{:}T \mid e\}$ in general. We discuss the connection between our manifest contracts and datatypes more in Section 5.3.

Findler and Felleisen's contracts—and those in this dissertation—have runtime effects and are *not* erasable in general. That is, the default semantics is to check all of these specifications as the program runs. Runtime checking contrasts markedly with refinement type theories that use SMT (satisfiability modulo theories) solvers, where there is little or no checking at runtime. The work on liquid types is one example [58, 59]. Others go further, including type Dyn: F* [71, 29] and DJS [19, 17]. All of these languages use SMT solvers to resolve refinement types written as propositions in a logic—while I use code. The logics of many of these languages allow first-order quantification, which rules out run-time checking as a possibility. In general, languages with propositional refinement types begin to push into the territory of program verification and program logics. This is a broad field in its own right, and I do not attempt to survey it here.

I should contrast the SMT-solver approach with the Curry–Howard approach taken in, e.g., Coq [21]. There are many differences, but the chief one is in the nature of evidence: refinement types and contracts use SMT solvers or other extensional "proof on the side" methods, while Coq and its brethren have intensional proof languages where the programmer constructs proofs like programs (and vice versa). Introducing impurities to the Curry–Howard style is an active area of research, but the current state of the art favors the extensional approach (e.g., F*).

Some work has been done on optimizing away unnecessary contract checks. In Chapter 3, I prove the soundness of *upcast elimination* (Lemma 3.4.3); other such optimizations have been the subject of some study already [44, 8]. In Chapter 4, I optimize the space usage of contracts in a way that requires us to (soundly) skip some checks. Findler et al. [27] (discussed more in Section 6.1.2) reduce re-checking of contracts on data structures.

5.1.2 Situating λ_{H}

The formulation of λ_{H} in Chapter 2 is most comparable to that of Knowles and Flanagan [44], but there are some significant differences. First, my cast-checking constructs are equipped with labels, and failed casts go to explicit blame—i.e., they raise labeled exceptions. In the λ_{H} of Knowles and Flanagan (though not the earlier one of Gronski and Flanagan), failed casts are simply stuck terms—their progress theorem says "If a well-typed term cannot step, then either it is a value or it contains a stuck cast." Second, their operational semantics uses full, non-deterministic β-reduction, rather than specifying a particular order of reduction, as I have done. This significantly simplifies parts of the metatheory by allowing them to avoid introducing parallel reduction. I prefer standard call-by-value reduction because I consider blame to be an exception—a computational effect—and I want to be able to reason about *which* blame will be raised by expressions involving many casts. At first glance, it might seem that my theorems follow directly from the results for Knowles and Flanagan's language, since CBV is a restriction of full β-reduction. However, the reduction relation is used in the type system (in rule S_Imp), so the type systems for the two languages are not the same. For example, suppose the term *bad* contains a cast that fails. In my system $\{y{:}B \mid \mathsf{true}\}$ is not a subtype of $\{y{:}B \mid (\lambda x{:}S.\ \mathsf{true})\ bad\}$ because the contract evaluates to blame. However, the subtyping does hold in the Knowles and Flanagan system because the predicate reduces to true.

The system studied by Ou et al. [51] is also close in spirit to my λ_{H}. The main difference is that, because their system includes general recursion, they restrict the terms that can appear in contracts to just applications involving predefined constants: only "pure" terms can be substituted into types, and these do not include lambda-abstractions. My system (like all of the others in Table 5.1—see the row labeled "any con") allows arbitrary user-defined boolean functions to be used as contracts.

My description of λ_{C} is ultimately based on λ_{Con} [26], though my presentation is slightly different in its use of checks. Hinze et al. [41] adapted Findler and Felleisen-style contracts to a location-passing implementation in Haskell, using picky dependent function contracts.

My λ_{H} type semantics in Section 2.3.2 is effectively a semantics of contracts. Blume and McAllester [11] offers a semantics of contracts that is slightly different—my semantics includes blame at every type, while theirs explicitly excludes it. Xu et al. [80] is also similar, though their "contracts" have no dynamic semantics at all: they are simply specifications.

Dimoulas et al. [22] introduce a new dialect of picky λ_C, where contract checks in the codomain are given a distinct negative label. If labels represent "contexts" for values, then this treats the contract as an independent context. "Indy" λ_C and picky λ_C will raise exactly the same *amount* of blame, but they will blame different labels.

Chapter 3, a corrected extension of Belo et al. [8], at once simplifies and extends the CBV λ_H given in Chapter 2. The type system is redesigned to avoid subtyping and closing substitutions, so type soundness is proved with easy syntactic methods [79]. The language also allows *general refinements*—refinements of any type, not just base types—and extends the type system to polymorphism. This can be seen as completing some of the future work of Greenberg et al. [34].

I have discussed only a small sample of the many papers on contracts and related ideas. I refer the reader to Knowles and Flanagan [44] for a more comprehensive survey. Another useful resource is Wadler and Findler [77] (technically superceded by Wadler and Findler [78], but with a longer related work section), which surveys work combining contracts with type Dyn and related features.

There are also *many* other systems that employ various kinds of precise types, but in a completely static manner. One notable example is the work of Xu et al. [80], which uses user-defined boolean predicates to classify values (justifying their use of the term "contracts") but checks statically that these predicates hold.

Sage [45] and Knowles and Flanagan [44] both support mixed static and dynamic checking of contracts, using, e.g., a theorem prover. I have not addressed this aspect of their work, since I have chosen to work directly with the core calculus λ_H, which for them was the target of an elaboration function.

5.2 F_H: polymorphism and manifest metatheory

I discuss work related to F_H (Chapter 3) in two parts. First, I distinguish my work from the untyped contract systems that enforce parametric polymorphism *dynamically*, rather than statically as F_H does. Then I discuss how F_H differs from existing manifest contract calculi in greater detail.

5.2.1 Dynamically checked polymorphism

The F_H type system enforces parametricity with type abstractions and type variables, while refinements are dynamically checked. Another line of work omits refinements, seeking instead to dynamically enforce parametricity—typically with some form of sealing (à la Pierce and Sumii [52]).

Guha et al. [37] define contracts with polymorphic signatures, maintaining abstraction with sealed "coffers"; they do not prove parametricity. Matthews and Ahmed [46] prove parametricity for a polymorphic multi-language system with a similar policy. Neis et al. [50] use dynamic type generation to restore parametricity in the presence of intensional type analysis. F_H's contracts are subordinate to the type system, so

the parametricity result does not require dynamic type generation. Ahmed et al. [3] prove parametricity for a gradual typing [67] calculus which enforces polymorphism with a set of global runtime seals. Ahmed et al. [4] define a polymorphic calculus for gradual typing, using local syntactic "barriers" instead of global seals. I believe that it is possible to combine F_H with the barrier calculus of Ahmed et al., yielding a polymorphic blame calculus [78]. I leave this to future work.

5.2.2 F_H and other manifest calculi

For a coarse comparison, please refer back to Table 5.1. In this section, I give a more technical comparison with the closest related work. Four existing manifest calculi with dependent function types ([28, 34, 44, 51]) use subtyping and theorem provers as part of the definition of their type systems. All four of these calculi have complicated metatheory. Ou et al. [51] restrict refinements and arguments of dependent functions to a conservative approximation of pure terms; they also place strong requirements on their prover. Knowles and Flanagan [44] as well as Greenberg, Pierce, and Weirich [34] use denotational semantics to give a firm foundation to earlier work [28]. I consider three systems in more detail: Knowles and Flanagan's λ_H (KF); Chapter 2's λ_H (which is the same as Greenberg, Pierce, and Weirich's λ_H, which I write here as GPW); and F_H. The rest of this subsection addresses the differences between KF, GPW, and F_H.

What made KF and GPW so complicated? Both systems share the same two impediments in the preservation proof: preservation after active checks and after congruence steps in the argument position of applications. KF and GPW use subtyping to resolve these issues. First, subtyping helps preserve types when evaluating casts with predicate contracts: if $\langle \mathsf{Int} \Rightarrow \{x{:}\mathsf{Int} \mid x > 0\}\rangle^l\, n \longrightarrow^* n$, then we need to type n at $\{x{:}\mathsf{Int} \mid x > 0\}$. KF and GPW use a rule like the following for refinement subtyping:[1]

$$\frac{\forall \Gamma, x{:}\{x{:}B \mid \mathsf{true}\} \vdash \sigma.\ \sigma(e_1) \longrightarrow^* \mathsf{true} \,\mathrm{implies}\, \sigma(e_2) \longrightarrow^* \mathsf{true}}{\Gamma \vdash \{x{:}B \mid e_1\} <: \{x{:}B \mid e_2\}}$$

Combined with a "constants get most specific types" requirement—for example, assigning n the type $\{x{:}\mathsf{Int} \mid x = n\}$—subtyping allows n to be typed at any predicate contract it satisfies. Second, KF and GPW use subtyping to show the equivalence of types with different but related term substitutions. Consider the standard dependent-function application rule:

$$\frac{\Gamma \vdash e_1 : (x{:}T_1 \to T_2) \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2[e_2/x]}$$

If $e_2 \longrightarrow e_2'$, how do $T_2[e_2/x]$ and $T_2[e_2'/x]$ relate? (An important question when proving preservation!) Both KF and GPW relate reduction and subtyping, showing

[1] Readers familiar with the systems will recognize that I've folded the implication judgment into the relevant subtyping rule.

that types that reduce to each other are mutual subtypes. KF use full beta reduction throughout their system. GPW use call-by-value reduction in their operational semantics, showing that parallel reducing types are mutual subtypes, separately relating CBV and parallel reduction. Once these two difficulties are resolved, both preservation proofs are standard, given appropriate subtyping inversion lemmas.

So much for subtyping. Why do KF and GPW need denotational semantics? Spelled out pedantically, the subtyping rule above has the following premise:

$$\forall \sigma.\ \Gamma, x{:}\{x{:}B \mid \mathsf{true}\} \vdash \sigma \text{ implies } (\sigma(e_1) \longrightarrow^* \mathsf{true} \text{ implies } \sigma(e_2) \longrightarrow^* \mathsf{true})$$

That is, the well formedness of the closing substitution σ is in a negative position. Where do closing substitutions come from? We cannot use the typing judgment itself, as this would be ill-defined: term typing requires subtyping via subsumption; subtyping requires closing substitutions in a negative position via the refinement case; but closing substitutions require typing. We need another source of values: hence, denotational semantics. Both KF and GPW define syntactic term models of types to use as a source of values for closing substitutions, though the specifics differ.

After adding subtyping and denotational semantics, both KF and GPW are well defined and have syntactic proofs of type soundness. But in the process of proving syntactic type soundness, both languages proved semantic soundness theorems:

$$\Gamma \vdash e : T \text{ implies } \forall \Gamma \vdash \sigma,\ \sigma(e) \in [\![\sigma(T)]\!]$$

This theorem suffices for soundness of the language... so why bother with a syntactic proof? In light of this, GPW only proves semantic soundness. The situation in KF and GPW is unsatisfying: the syntactic proof of type soundness motivated subtyping, which motivated denotational semantics, which obviated the need for syntactic proof. Beyond this, the proofs are hard to scale: adding in polymorphism or state is a non-trivial task, since we must—before defining the type system!—construct an appropriate denotational semantics, which itself depends on the evaluation relation.

$\mathrm{F_H}$ solves the problem by avoiding subtyping—which is what forced the presence of closing substitutions and denotational semantics in the first place. The first issue in preservation—that of preserving refinement types after checks have finished—was resolved in KF and GPW with subtyping. Instead, I resolve it with a runtime rule that allows us to type values with any refinement they satisfy:

$$\frac{\vdash \Gamma \qquad \emptyset \vdash v : T \qquad \emptyset \vdash \{x{:}T \mid e\} \qquad e[v/x] \longrightarrow^* \mathsf{true}}{\Gamma \vdash v : \{x{:}T \mid e\}}\ \textsc{T\_Exact}$$

Adding this rule eliminates one use of subtyping as well as the "most-specific type" restriction. If we "bit the bullet" and allowed non-empty contexts in T_Exact, then we would need to apply a closing substitution to $e[v/x]$ before checking if it reduces to true. But the circularity in subtyping alluded to at the beginning of this chapter was caused by closing substitutions; we must avoid them! The second issue

in preservation—that of conversion between $T_2[e_2/x]$ and $T_2[e_2'/x]$—can be resolved in a similar fashion. We define another runtime rule that allows us to convert types:

$$\frac{\vdash \Gamma \qquad \emptyset \vdash e : T \qquad \emptyset \vdash T' \qquad T \equiv T'}{\Gamma \vdash e : T'} \text{ T\_Conv}$$

The conversion we use, $\equiv$, is defined as the symmetric, transitive closure of CBV-respecting parallel reduction. This is only as much equivalence as we need: if $e_2 \longrightarrow e_2'$, then $T_2[e_2/x] \equiv T_2[e_2'/x]$. These two rules suffice to keep subtyping out of $\mathrm{F_H}$, which in turn avoids denotational semantics.

Other consequences of subtyping

The $\mathrm{F_H}$ operational semantics is essentially a superset of λ_{H} from Chapter 2, barring some slight differences in the function cast decomposition rule. The type system, however, is not a superset: λ_{H} types some programs $\mathrm{F_H}$ does not. In particular, λ_{H} builds in subsumption, while $\mathrm{F_H}$ only has a subsumption principle *post facto*. We can, however, take a λ_{H} typing derivation and eliminate every occurrence of subsumption: by the upcast lemma, the two programs are equivalent, even if one of them is not well typed. That is, I have taken subsumption out of the type system and proved subsumption safe as an optimization—and, in doing so, greatly simplified the type system.

5.3 Space efficiency and gradual types

There are two threads of work related to the development of Chapter 4: a more recent line of work on gradual types, refinement types, and full-spectrum programming languages; and an older, more general line of work on coercions, which may or may not have runtime semantics. Space efficiency and representation have been studied in both settings.

5.3.1 Space efficiency, gradual typing, and refinement types

We give an overview of the field of gradual typing in 5.2. The commonality between all systems is the presence of a type like Dyn.

In Siek and Taha's seminal work on gradual typing [67], space efficiency is already a concern—they point out that the canonical forms lemma has implications for which values can be unboxed (the typed ones). Herman, Tomb, and Flanagan [39] compiled a language like Siek and Taha's into a calculus with Henglein's coercions [38], proving a space-efficiency result with a galactic bound similar to mine. Herman et al. stop at proving that their compilation is type preserving without proving soundness of their compilation. (I compare my system to Herman et al.'s in greater detail below.) Siek, Garcia, and Taha [65] explore the design space around Herman et al.'s result,

	ACPP89 (1)	T90 (2)	CF91 (3)	H94 (4)	KTGFF06 (5)	ST06 (6)	HTF07/10 (7)	SGT09 (8)	WF09 (9)
blame (10)	total	$\Uparrow$	$\Uparrow$	stuck	$\Uparrow$	$\Uparrow$	$\Uparrow$	$\Uparrow l$	$\Uparrow l$
con (11)	×	×	×	×	✓?	×	×	×	✓
poly (12)	×	×	✓	×	✓	×	×	×	×
space (13)	×	×	×	✓	×	×	×	✓+	×

	BGHL10 (14)	SW10 (15)	AFSW11 (16)	SG12 (17)	G13 (18)	Eff. (Ch. 4)
blame (10)	$\Uparrow$	$\Uparrow l$	$\Uparrow l$	$\Uparrow l$	$\Uparrow l$	$\Uparrow$
con (11)	✓(FO)	×	×	×	×	✓
poly (12)	×	×	✓	×	×	×
space (13)	×	✓	×	✓	✓+	✓+

(1) Abadi et al. [1]. (2) Thatte [73] (3) Cartwright and Fagan [15] (4) Henglein [38] (5) Knowles et al. [45] (6) Siek and Taha [67] (7) Herman et al. [39, 40] (8) Siek et al. [65] (9) Wadler and Findler [78] (14) Bierman et al. [10] (15) Siek and Wadler [68] (16) Ahmed et al. [4] (17) Siek and Garcia [66] (18) Garcia [31] (10) Do failed contracts raise labeled blame ($\Uparrow l$), raise blame without a label ($\Uparrow$), or get stuck? NB that (1) requires that typecases be total, so there are no errors. (11) Are there contracts or refinements types? NB that (5) lacks a soundness proof, and (14) is a first-order language. (12) Does the type system support polymorphism? (13) Is the operational semantics space efficient? If so (✓), is there a proof of soundness relating the space-efficient calculus to a naïve semantics (✓+)?

Table 5.2: Comparison between gradual typing systems

this time with an observational equivalence theorem exactly relating two coercion semantics.

Siek and Wadler [68] study an alternative, cast-based formulation of space efficiency, proving tighter bounds than Herman et al. [39] and an exact observational equivalence. Their insight is that casts can be factored not merely as a "twosome" $\langle S \Rightarrow T \rangle$, but rather as a threesome: $\langle S \overset{R}{\Rightarrow} T \rangle$. They maintain the invariant that S downcasts to R, and R upcasts to T; merging casts amounts to calculating a greatest lower bound. They come up with an elegant theory of merging casts, with a detailed accounting for blame. While the mathematics is beautiful, I believe that their algorithm is overkill: Herman et al.'s journal article [40] cleanly enumerates the recursive structure of the canonical coercions for dynamic and simple types, with only 17 possible structures at the top level. Siek and Wadler's theory is the theory of these 17 structures. Many of the solutions can be simply pre-computed and looked up in a table at runtime. I have 37 canonical coercions. I don't study the question here, but I believe that a careful analysis would allow for very compact representations with very fast merges—by pointer comparison and table lookup when functional coercions aren't involved. I discuss this issue further in future work (Chapter 6).

Before considering other full-spectrum languages, I compare this work to the most closely related work: Herman, Tomb, Flanagan [39, 40], Garcia [31], Siek and Garcia [66], and Henglein [38]. Henglein is trying to reason carefully about programs written in a dynamic style, rather than thinking about multi-paradigm programming (though it is clear that he knows that his work applies to "dynamic typing in a static language"). His theory of coercions has no Fail coercion and treats Id slightly differently at function types. Herman et al. adapt his calculus to match the setting of gradual types, though they never rebuild his theory. Henglein develops a general theory *characterizing* canonical coercions, but I *enumerate* them, as in Herman et al. [40].

Perhaps the biggest difference is that Henglein and Herman et al. formulate coercions as having arbitrary composition: $c_1; c_2$ is a coercion that can be used freely. As a consequence, it is somewhat difficult to reason directly about coercions in the calculus: what should $\langle (c_1 \mapsto c_2; \mathbf{Fun}!); \{x{:}\mathsf{Dyn} \mid e\}? \rangle\ v$ do? Their solution is to work with coercions up to an equivalence relation that includes associativity of coercion composition; coercions normalize in a term rewriting system modulo this equivalence relation. Henglein studies some algorithmic rewriting systems. But Herman et al. don't develop the rewriting system at all, never showing that their rewriting system is strongly normalizing, and even when they enumerate canonical coercions in their journal version [40], they do so without proof. I feel that term rewriting modulo equational theories is insufficient for guiding an implementation of a coercion calculus: the compiler needs a concrete representation for coercions and a concrete algorithm for merging them. I accordingly adopt a constrained form of coercion composition out of a desire to aid implementation, but also out of expedience: I don't need to worry about associativity at all. I don't believe that free composition buys anything,

anyway: I don't expect programmers to be writing coercions by hand, so ease of expression in the coercion language isn't particularly important.

Herman et al.'s calculus is a little odd: the value $\langle \mathsf{Id} \rangle\ v$ takes a step to v. My approach makes a clearer distinction between terms and the results that they produce. Siek and Wadler noticed a separate problem with nondeterminism at cast merges, most likely due to a mistake in defining evaluation contexts.

Siek and Garcia [66] study various interpretations and implementations of gradual typing. Their `seq-lazy` function is very similar to my merge algorithm, though our treatments of associativity are different.

Garcia's work [31] is remarkably similar to mine, though restricted to gradual types: starting with a gradually typed calculus with casts, he develops coercions and then threesomes as a series of derivations. His supercoercions are remarkably similar to my canonical coercions, though mine are complicated by the presence of refinement types. Our approaches differ, though; he says:

> One might try to devise an ad hoc reassociation scheme or represent a sequence of coercions as lists, but it would necessarily involve pairwise comparisons, bidirectional search, and splicing into the middle of complex coercion expressions.

My representation is very nearly a "sequence of coercions as lists", though I abandon the standard linked-list structure. My merge algorithm roughly fits his description: I do pairwise comparisons from the inside out and concatenation, though I do not splice things in the middle nor do I do bidirectional search. Coercions and threesomes are two representations of the same idea. Which of the two implementations is best in practice is an open question. Coercion merges are amenable to table lookup, but the threesome merge operator can be memoized. Which is faster, which—if any—has a more compact representation? Which is easier to implement?

The work discussed so far consisted of calculi devised expressly for space-efficient gradual typing. Findler et al. [27] discuss space efficiency from the perspective of an implementation in PLT Racket (then PLT Scheme). Their setting—latent contracts, no type system—is rather different from the foregoing systems; they address datatypes, while the foundational calculi omit datatypes.

Considering the wider world of full-spectrum programming languages, we summarize existing solutions. None of the following are space efficient; I am the first to combine space efficiency, gradual types, and refinement types. Ou et al. [51] cover the spectrum and include dependent types, but allow only a constrained set of refinement predicates; Sage [45] covers the entire spectrum and also includes dependent types, but lacks a soundness proof; Wadler and Findler's [78] development covers dynamic types through refinements of base types; Bierman et al. [10] cover the whole spectrum but (also with dependency) only for first-order types.

5.3.2 Coercions

There are many other systems that use coercions to other ends. Henglein gives an excellent summary of work up to 1994 in the related work section of his article [38]. One of the classic uses of coercions is subtyping [13, 48]; more recent work relates subtyping and polymorphism [20]. Work on unboxing [63, 49] confronts similar issues of space efficiency. Many of these works carefully ensure that coercions are erasable, while my coercions are definitely not.

Swamy, Hicks, and Bierman [70] study coercion insertion in general, showing that their framework can encode gradual types. I haven't studied coercion insertion at all, though Swamy et al.'s framework would be a natural one to use. I am not aware of work on how coercion insertion algorithms affect space consumption, though experience with the implementation of Boomerang [12] shows that small changes in coercion insertion can affect the efficiency of checking.

Chapter 6

Conclusion and future work

> Why mince words, anyway, since you are not completely real Dreyfuses, Edisons, and Napoleons? You have assumed these names vicariously, for lack of anything better. Now you will swell the numbers of many of your predecessors, those anonymous Garibaldis, Bismarcks, and MacMahons who wander in their thousands, unacknowledged, all over the world.
>
> *Sanatorium under the Sign of the Hourglass*
> Bruno Schulz

> Now at last without fantasies or self-deception, cut off from the mistakes and confusion of the past, grave and simple, carrying a small suitcase, getting on a bus, like girls in movies leaving home, convents, lovers, I supposed I would get started on my real life.
>
> *Lives of Girls and Women*
> Alice Munro

Manifest contracts, as opposed to latent contracts, are the way forward for investigating strong specifications in general purpose programming languages. The existing uses of latent contracts have been lackluster: PLT Racket uses contracts to recover simple types. I suspect that the lack of a type discipline has prevented pervasive use of strong specifications. Taking programming in a type discipline as a baseline may avoid the "contracts for types are good enough" under-specification problem found in PLT Racket.

In this dissertation, I have developed the field of manifest contracts with three contributions:

- Characterized manifest contracts in relation to latent contracts (Chapter 2);
- Developed F_{H}, a manifest contract calculus with a powerful reasoning principle: relational parametricity (Chapter 3); and

• Shown how to resolve the unbounded space consumption in function proxies and on the stack that can accrue with contract checking (Chapter 4).

Each of these contributions serves language designers in different ways, but all of this goes to support a firm theoretical basis for manifest contracts. Obtaining reasoning principles and eliminating the gross inefficiencies in naïve formulations are necessary first steps in designing and implementing higher-order languages with pervasive manifest contracts.

The characterization of latent and manifest contracts outlines the field: what options does the language designer have, and how do they relate? This allows for informed choices—and, as in the example of Dimoulas et al. [22], further study. The proof technique for translation correctness—logical relations extended with an inductively defined invariant for contracts—offers theoreticians a powerful tool for metatheoretical work with contracts; I use it again in Chapter 4.

The F_{H} metatheory of Chapter 3 offers several things. First, its metatheory is much simpler than that of Chapter 2's λ_{H}, as it eschews denotational techniques.[1] The F_{H} metatheory offers a cleaner and easier way to design core calculi for manifest contracts, a natural first step in language design. The F_{H} parametricity relation offers powerful reasoning principles to implementors and programmers alike: the upcast lemma of Section 3.4, for example, could drive an upcast elimination optimization in a compiler; the parametricity relation could be used to reason about different representations of abstract types.

Finally, the space efficiency work is necessary for a real implementation of designs that use contracts pervasively, rather than just at module boundaries like in PLT Racket. Moreover, my work in Chapter 4 reveals fundamental trade-offs in the design space: space-efficiency can be had for manifest contracts, but at the price of skipping some checks. Finally, I offer a different perspective on and new algorithms for coercions and threesomes.

I hope that language designers can, after reading this dissertation, begin to think about what it would mean to include manifest contracts in their systems: how the operational semantics might ensure space efficiency; how the type system might be kept small enough to admit easy analysis, while offering good reasoning principles; what equalities the reasoning principles permit. I view my work as digging—not even pouring—foundation for languages with manifest contracts. I have surveyed the field, and I offer workably efficient semantics and powerful principles for reasoning and type abstraction.

[1] The original, flawed conception of Belo et al. [8] was still simpler—it didn't "need" a coermination theorem; however, it did need correction.

6.1 Future work

The work in this dissertation aims to put manifest contracts on a firm type-theoretical footing. There are several avenues of future work on the road to putting manifest contracts on a firm *pragmatic* footing. I conclude by discussing future work.

6.1.1 State and effects

The biggest subject missing from this dissertation is *state* and other forms of effects, beyond uncatchable exceptions (blame) and nontermination. Adding state to a manifest calculus is problematic. Consider the type $\{x{:}\mathsf{Ref\ Int} \mid !x > !y\}$. When a value v has this type, what does the programmer know? When must the contract be checked? If either x or y update, we may need to recheck the contract. But what if x and y need to *both* be updated? What if there are circular dependencies between refined references? The folklore consensus is that *some* sort of limits are necessary on stateful contracts. Some work has been done on this question, in particular Shinnar [64]. Shinnar uses a notion of delimited (transactional memory) transactions to (a) check contracts at the end of transactions, and (b) roll back changes contracts make to state. Unfortunately for our purposes, he emphasizes the delimited transactions in favor of a detailed study of stateful contracts. Dimoulas et al. [22] offer a latent semantics without a study of what the semantics actually *means*. The temporal contracts due to Disney et al. [24] may seem unrelated, but there is no real difference between a temporal contract and a stateful contract. Phrased as stateful contracts, their temporal contracts are essentially limited to advancing the state of a state machine—they forbid contracts from reading or writing to state, or even calling functions.

6.1.2 Datatypes

In Chapter 3, I investigated how contracts interact with abstract types and polymorphism; Sekiyama and Igarashi [61] have extended that work to include a language with fixpoints. It remains to address datatypes; very little study has been done on how contracts and concrete implementations of algebraic datatypes interact. I am aware only of Findler et al. [27]; Rehof [57] studies gradual typing calculi with sums, but within the framework of dynamic types, and without an eye to implementation. While the programming language community normally contents itself with Church encodings (as in Belo et al. [8]), they are insufficient here—as Findler et al. demonstrate, we must finely control abstract datatype representations in order to check contracts without changing asymptotic time complexity. Supporting space-efficient datatypes in addition to space-efficient functions and stacks would complement the work in Chapter 4.

I believe it would be very interesting future work to try to combine my refinement types, protecting partial operations (space efficiently), with classic refinements of datatypes [30] protecting partial matches (also space efficiently). As a first step,

adding pairs and sums presents a design choice: is tag merging deep, as it is for functions, or is it shallow and deferred upon projection? The deep choice is more obviously space efficient, but both options should be explored.

6.1.3 Coercion insertion

While some of the early gradual typing papers give coercion insertion algorithm, very little has been done in terms of comparative study; I am aware only of Swamy et al. [70] and Allende et al. [5]. Can different coercion insertion strategies affect the space and time efficiency of a program?

I am able to answer an emphatic "yes" to those questions based on experience in Boomerang [12]. Consider a partial binary operation $\mathsf{op} : (x{:}T) \to \{y{:}T \mid \mathsf{P}\ x\ y\} \to T$ (Recall seq and concat from Chapter 3.) Suppose we have a nested application $\mathsf{op}\ (\mathsf{op}\ e_1\ e_2)\ e_3$ where each of the e_i is typed at T. It is natural to insert a cast $\langle T \Rightarrow \{y{:}T \mid \mathsf{P}\ x\ y\}\rangle$ on e_2. Well, almost—we actually need to cast to $\{y{:}T \mid \mathsf{P}\ e_1\ y\}$, to account for the application of the dependent function type. So the new inner term is $e_{inner} = (\mathsf{op}\ e_1\ (\langle T \Rightarrow \{y{:}T \mid \mathsf{P}\ e_1\ y\}\rangle\ e_2))$. If we apply the coercion insertion similarly on the outer term, we need to cast e_3 to $\{y{:}T \mid \mathsf{P}\ e_{inner}\ y\}$—which will force us to completely rerun the check on e_2 and the application of op. Even with parser tricks to keep expression trees balanced, this is so inefficient as to be a non-starter.

Our solution in Boomerang was to evaluate terms with inserted checks in an *aliased* call-by-value fashion: terms are evaluated in a call-by-value order, but there is some sharing in the expression tree. This seriously convoluted the definition of our interpreter—we implemented it with reference cells in OCaml—but led to an enormous speedup. While this problem arises in every language with a dependent application rule, it is particular dire for us because the terms in types are sometimes evaluated!

6.1.4 Theoretical curiosities

I am curious to see what free theorems and reasoning principles come derive from relational parametricity for manifest calculi. I also wonder whether or not we can soundly perform type erasure in $\mathrm{F_H}$—the careful treatment of compatibility at type variables seems to indicate yes, but it remains to be shown.

With the introduction of abstract types, there is room to draw connections between the client/server blame from the ADTs of Section 3.1 and the classic Findler and Felleisen-style client/server blame. On a similar note, I believe that an interesting connection can be drawn between blame labels and stack traces.

On the one hand, Typed Racket née Scheme [74] has a "static analysis" feel in its flow-sensitive type system; later work by Tobin-Hochstadt and Van Horn [75] develops a static analysis. On the other hand, most of the work on manifest contracts uses subtyping, relying on SMT solvers to resolve implications between predicates [44, 45,

28, 10, 59, 58, 51]. How do static analysis and subtyping compare at eliminating unnecessary checks and/or detecting failures?

Knowles and Flanagan [44] give a cast insertion algorithm with a three-valued theorem prover to compile source-level programs into λ_{H} terms with casts: if the prover says "yes", then subtyping holds and no cast is necessary; if the prover says "maybe", then subtyping may or may not hold, so a cast is inserted; if the prover says "no", then the program is rejected. Rejecting such programs is tempting, but too conservative: a cast that isn't an upcast doesn't *always* fail, it just *may* fail. A program should be rejected not when a non-upcast is needed, but when attempting to cast between types that don't share any values.

6.1.5 Extensions for Efficient

The obvious next step for the space-efficiency work is to add blame [26]. Phil Wadler berated me about the omission, and insists that I prove a blame theorem [78], which would allow me to claim that blame comes from less specific types. Siek and Wadler [68] were the first space-efficient calculus to have blame, which they obtain with some effort—their threesome merging takes place outside in, making it hard to compute which label to blame. Later work [66, 31] simplifies the treatment of blame somewhat. I conjecture that my inside-out coercion merge algorithm offers an easy way to compute blame: blame comes from left to right. The `seq-lazy` definition from Siek and Garcia [66] bolsters my confidence in this conjecture.

In Cast, the rule G_CastId doesn't apply to casts between function types, which instead use G_CastFunWrap. Given the cast $\langle T_1 \to T_2 \Rightarrow T_1 \to T_2 \rangle\ v$, we will wrap v with redundant checks: when the T_i aren't arrow types, the casts immediately disappear. That is, we η-expand all function contracts, even trivial ones. To make matters worse, G_CastFunDyn and G_CastFunFun introduce function casts. In Efficient, each function will only ever have a single function proxy on it, so the cost is not so great. Even so, an implementation would want to avoid this unnecessary η-expansion. The corresponding rewrite rule would be $(\mathsf{Id} \mapsto \mathsf{Id}) \longrightarrow \mathsf{Id}$. I would need a merge rule similar to FunDom to handle this case. Henglein includes the equality $(\mathsf{Id} \mapsto \mathsf{Id}) = \mathsf{Id}$ along with associativity. Relatedly, I could consider eager semantics, where $(\mathsf{Fail} \mapsto c_2) = (c_1 \mapsto \mathsf{Fail}) = \mathsf{Fail}$, as discussed in the literature [65, 66, 31].

If checks are expensive, predicting when checks happen could be important. When will the N_PredPred merge rule apply, and how can programmers predict performance? There are further implementation concerns: the coercion merging algorithm needs to compare refinement types, which, as Greg Morrisett pointed out, amounts to comparing closures. Comparing closures is dangerous business: optimizers may disrupt programmer expectations. If I were to introduce dependencies, the comparison of closures would include the comparison of environments containing functions; extensionally equivalent functions may not be intensionally equal, leading to still more unexpected behavior. A more nominal approach may serve here. Finally, there is an open question about calling conventions, since tagging introduces a second kind of

closure: when calling a function, do we need to run coercions or not? Jeremy Siek suggested a "smart closure" which holds the logic for branching inside its own code; this may support better branch prediction than an indirect jump or branching at call sites.

Extending the calculus to general refinements, where any type T can be refined to $\{x{:}T \mid e\}$, would be a challenging but important step towards adding polymorphism. (You can't allow refinement of type variables unless *any* type can be refined, since there's no way to know what type will be substituted in for the variable.) It wouldn't be too difficult to add function refinements $\{x{:}(T_1 \to T_2) \mid e\}$ to this calculus, but refinements of refinements seem to break space efficiency: if $\{x{:}B \mid e\}?$ is canonical, so is $\{x{:}B \mid e\}?;\{x{:}\{x{:}B \mid e\} \mid e'\}?$—there are an infinite number of canonical coercions. In a monomorphic calculus, the number of canonical coercions can be bounded by the types in the original program, but not so in a polymorphic calculus. Prior work relating dynamic types and polymorphism will apply here [57, 46, 4]—though attaining a relational parametricity proof remains a hard open problem.

Other systems have treated failure more eagerly, e.g., $\mathsf{Fail} \mapsto \mathsf{Id} * c \Rightarrow \mathsf{Fail}$. This would further disrupt the connection between the naïve and space-efficient semantics.

Adding dependent functions to the coercion calculus above would complicate matters significantly, but would also add a great deal of expressiveness. Adding the type $(x : T_1) \to T_2$ is straightforward enough: we should be able to prove type soundness using entirely syntactic techniques, adapting work on FH, a polymorphic calculus with manifest contracts and general refinements [8]. Designing the coercions is a challenge, though. Dependency means that in the coercion $(x{:}c_1) \mapsto c_2$, the variable x is bound in c_2. Coercion well formedness now needs a context to keep track of such bound variables:

$$\frac{\Gamma \vdash c_1 : T_{11} \rightsquigarrow T_{21} \qquad \Gamma, x{:}T_{11} \vdash c_2 : T_{12} \rightsquigarrow T_{22}}{\Gamma \vdash (x{:}c_1) \mapsto c_2 : ((x{:}T_{21}) \to T_{12}) \rightsquigarrow ((x{:}T_{11}) \to T_{22})}$$

How do dependent functions and their corresponding coercions affect coercion normalization? The following structural equivalence rule, derived from the for dependent functions looks improbable, though its asymmetry echoes the asymmetry of the dependent function cast rule in other manifest calculi, e.g., F_H and λ_H.

$$\begin{array}{l}((x{:}c_1) \mapsto c_2);((y{:}c_1') \mapsto c_2') = \\ (y{:}(c_1';c_1)) \mapsto (c_2\{\langle c_1' \rangle\ y/x\};c_2')\end{array}$$

The metatheory surrounding dependent functions in coercion calculi will be difficult. In fact, the rule above isn't obvious: trying to use $x{:}T_{21}$ as the binding for x will raise difficulties in typing the equational rule.

Programs with dependent types have a potentially infinite set of types (and, so, coercions) which may appear as the program evaluates. In the type $(x : \mathsf{Real}) \to \{y{:}\mathsf{Real} \mid |x - y^2| < \epsilon\}$, there are potentially infinitely many different codomain types: one for each Real value of x. But even with an infinite number of possible coercions,

the set of canonical coercions shouldn't change (beyond the addition of dependency to functions).

Set semantics for refinement types extend refinement types to have a set of predicates, rather than a single one. The N_Phi merge rule with $D = \{x{:}T \mid e\}$ skips a check when we would have projected out of and then back into a refinement type. If refinements were sets, we could broaden this optimization to allow the space-efficient calculus to avoid even more redundant checks.(This was, in fact, the original motivation for this work.)

In an extension of the system of Section 4.5 with so-called "general" refinements [8], the coercion $\{x{:}\mathsf{Int} \mid x > 0\}?; \{x{:}\{x{:}\mathsf{Int} \mid x > 0\} \mid x > 5\}?$ is well typed but $\{x{:}\mathsf{Int} \mid x > 0\}?; \{x{:}\mathsf{Int} \mid x > 5\}?$ isn't. Furthermore, $\{x{:}\{x{:}\mathsf{Int} \mid x > 0\} \mid x > 5\}?$ and $\{x{:}\{x{:}\mathsf{Int} \mid x > 5\} \mid x > 0\}?$ are totally different coercions, even though the underlying predicates in the refinements are the same. Could we convert $\{x{:}T \mid e_1\}?; \{x{:}\{x{:}T \mid e_1\} \mid e_2\}?$ to a single coercion? Simply changing the coercion to $\{x{:}T \mid e_1 \wedge e_2\}?$ disrupts typing, but the upcast lemma (Lemma 3.4.3 in Chapter 3) shows that this is still behaviorally equivalent. Treating a refinement as being flat with a single set of predicates, rather than a tower of separate refinements, would resolve these ordering issues. That is, we could have a single type that represents both orderings of refinement: $\{x{:}\mathsf{Int} \mid \{x > 5, x > 0\}\}$. Casts into and out of refinement types could be simplified to adding and removing refinements from the set. When a value is coerced into a refinement type with a set of predicates, the type system remembers all of the predicates equally, acting as a cache of multiple satisfied contracts. The utility of the set semantics is that helps address the library problem: when writing a list library, what refinements are important—emptiness, sortedness, length? When refinements are treated as sets of predicates, libraries can deal only with their own predicates, treating extra client predicates parametrically.

For example, remembering that a list is both non-empty and sorted might be useful for a sorted-list representation of sets. When the set code needs to take the head of a list (which happens to be a minimal member of the set), it can do so directly. Similarly, when calling the $\mathsf{insertSorted}$ function to add an element to the set, it knows that both its original representation and the extended one are still valid, sorted representations.

Another example where the set semantics allows multiple predicates to interact is a value $v : \{x{:}\mathsf{Int} \mid \{x \neq 0, \mathsf{prime}\ x\}\}$. Since $v \neq 0$, we can use it as the divisor with $\mathsf{div} : \mathsf{Int} \to \{x{:}\mathsf{Int} \mid x \neq 0\} \to \mathsf{Int}$; since $\mathsf{prime}\ v$, we can use it as half of a private key.

Set semantics was my initial motivation for revisiting space-efficient coercions: I was interested in ways of remembering contract checks on values to reduce redundant checking.

Bibliography

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Principles of Programming Languages (POPL)*, 1989.

[2] Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming (ESOP)*, 2006.

[3] Amal Ahmed, Robert Bruce Findler, Jacob Matthews, and Philip Wadler. Blame for all. In *Workshop on Script-to-Program Evolution (STOP)*, 2009.

[4] Amal Ahmed, Robert Bruce Findler, Jeremy Siek, and Philip Wadler. Blame for all. In *Principles of Programming Languages (POPL)*, 2011.

[5] Esteban Allende, Johan Fabry, and Eric Tanter. Cast insertion strategies for gradually-typed objects. In *Dynamic Languages Symposium (DLS)*, 2013.

[6] David Aspinall and Adriana Compagnoni. Subtyping dependent types. *IFIP Conference on Theoretical Computer Science (TCS)*, September 2001.

[7] Brian Aydemir and Stephanie Weirich. LNgen: Tool support for locally nameless representations. Technical Report MS-CIS-10-24, University of Pennsylvania, June 2010.

[8] João Filipe Belo, Michael Greenberg, Atsushi Igarashi, and Benjamin C. Pierce. Polymorphic contracts. In *European Symposium on Programming (ESOP)*, 2011.

[9] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. In *International Conference on Functional Programming (ICFP)*, pages 51–63, New York, NY, USA, 2003. ACM. ISBN 1-58113-756-7. doi:10.1145/944705.944711.

[10] Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, and David Langworthy. Semantic subtyping with an SMT solver. In *International Conference on Functional Programming (ICFP)*, 2010.

[11] Matthias Blume and David A. McAllester. Sound and complete models of contracts. *Journal of Functional Programming (JFP)*, 2006.

[12] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: resourceful lenses for string data. In *Principles of Programming Languages (POPL)*, 2008. doi:10.1145/1328438.1328487.

[13] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172 – 221, 1991. ISSN 0890-5401. doi:http://dx.doi.org/10.1016/0890-5401(91)90055-7. Selections from 1989 {IEEE} Symposium on Logic in Computer Science.

[14] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. In *Information and Computation*, 1991.

[15] Robert Cartwright and Mike Fagan. Soft typing. In *Programming Language Design and Implementation (PLDI)*, pages 278–292, New York, NY, USA, 1991. ACM. ISBN 0-89791-428-7. doi:10.1145/113445.113469.

[16] Olaf Chitil and Frank Huch. Monadic, prompt lazy assertions in haskell. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2007.

[17] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for javascript. In *OOPSLA*, pages 587–606, 2012.

[18] Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. Nested refinements: a logic for duck typing. In *Principles of Programming Languages (POPL)*, POPL '12, pages 231–244, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi:10.1145/2103656.2103686.

[19] Ravi Chugh, Patrick Maxim Rondon, and Ranjit Jhala. Nested refinements: a logic for duck typing. In *Principles of Programming Languages (POPL)*, pages 231–244, 2012.

[20] Julien Cretin and Didier Rémy. On the power of coercion abstraction. In *Principles of Programming Languages (POPL)*, POPL '12, pages 361–372, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi:10.1145/2103656.2103699.

[21] Coq development team. The Coq proof assistant reference manual, version 8.2, August 2009. `http://coq.inria.fr/`.

[22] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: no more scapegoating. In *Principles of Programming Languages (POPL)*, 2011. doi:10.1145/1926385.1926410.

[23] Tim Disney and Cormac Flanagan. Gradual information flow typing. In *Workshop on Script-to-Program Evolution (STOP)*, 2011.

[24] Tim Disney, Cormac Flanagan, and Jay McCarthy. Temporal higher-order contracts. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 176–188, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0865-6. doi:10.1145/2034773.2034800.

[25] Robert Bruce Findler. Contracts as pairs of projections. In *Symposium on Logic Programming*, 2006.

[26] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*, 2002.

[27] Robert Bruce Findler, Shu-Yu Guo, and Anne Rogers. Lazy contract checking for immutable data structures. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *Implementation and Application of Functional Languages*, pages 111–128. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-85372-5. doi:10.1007/978-3-540-85373-2_7.

[28] Cormac Flanagan. Hybrid type checking. In *Principles of Programming Languages (POPL)*, 2006.

[29] Cédric Fournet, Nikhil Swamy, Juan Chen, Pierre-Évariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to javascript. In *Principles of Programming Languages (POPL)*, pages 371–384, 2013.

[30] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Programming Language Design and Implementation (PLDI)*, June 1991.

[31] Ronald Garcia. Calculating threesomes, with blame. In *International Conference on Functional Programming (ICFP)*, 2013.

[32] Mike Gordon. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, language, and interaction*, chapter From LCF to HOL: a short history, pages 169–185. MIT Press, Cambridge, MA, USA, 2000. ISBN 0-262-16188-5.

[33] Michael Greenberg. Space-efficient manifest contracts. Rejected from POPL, 2013.

[34] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *Principles of Programming Languages (POPL)*, 2010.

[35] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. *JFP*, 22(3):225–274, May 2012.

[36] Jessica Gronski and Cormac Flanagan. Unifying hybrid types and contracts. In *Trends in Functional Programming (TFP)*, 2007.

[37] Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric polymorphic contracts. In *Dynamic Languages Symposium (DLS)*, 2007.

[38] Fritz Henglein. Dynamic typing: Syntax and proof theory. *Sci. Comput. Program.*, 22(3):197–230, 1994.

[39] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *Trends in Functional Programming (TFP)*, pages 404–419, April 2007.

[40] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher Order Symbol. Comput.*, 23(2):167–189, June 2010. ISSN 1388-3690. doi:10.1007/s10990-011-9066-z.

[41] Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *Functional and Logic Programming (FLOPS)*, 2006.

[42] Tony Hoare. Null references: The billion dollar mistake. QCon talk, 2009.

[43] Catalin Hritcu, Michael Greenberg, Ben Karel, Benjamin C. Pierce, and Greg Morrisett. All your IFCException are belong to us. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 3–17, 2013. doi:10.1109/SP.2013.10. The author is deeply embarassed by the title of this paper.

[44] Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Transactions on Programming Languages and Systems*, 32:6:1–6:34, 2010.

[45] Kenneth Knowles, Aaron Tomb, Jessica Gronski, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, 2006.

[46] Jacob Matthews and Amal Ahmed. Parametric polymorphism through run-time sealing or, theorems for low, low prices! In *European Symposium on Programming (ESOP)*, 2008.

[47] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., 1992. ISBN 0-13-247925-7.

[48] Yasuhiko Minamide. Runtime behavior of conversion interpretation of subtyping. In *Selected Papers from the 13th International Workshop on Implementation of Functional Languages*, IFL '02, pages 155–167, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43537-9.

[49] Yasuhiko Minamide and Jacques Garrigue. On the runtime complexity of type-directed unboxing. In *International Conference on Functional Programming (ICFP)*, ICFP '98, pages 1–12, New York, NY, USA, 1998. ACM. ISBN 1-58113-024-4. doi:10.1145/289423.289424.

[50] Georg Neis, Derek Dreyer, and Andreas Rossberg. Non-parametric parametricity. In *International Conference on Functional Programming (ICFP)*, 2009.

[51] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *IFIP Conference on Theoretical Computer Science (TCS)*, 2004.

[52] Benjamin Pierce and Eijiro Sumii. Relating cryptography and polymorphism, July 2000.

[53] A. M. Pitts. Typed operational reasoning. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7. The MIT Press, 2005.

[54] A. M. Pitts. Typed operational reasoning. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7, pages 245–289. The MIT Press, 2005. ISBN 0-262-16228-8.

[55] PLT. PLT Racket, 2013. URL `http://racket-lang.org`.

[56] PLT. PLT Racket contract system, 2013. URL `http://pre.plt-scheme.org/docs/html/guide/contracts.html`.

[57] Jakob Rehof. Polymorphic dynamic typing: Aspects of proof theory and inference. Master's thesis, DIKU, 1995. DIKU Technical Report D-249.

[58] Patrick M. Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *Programming Language Design and Implementation (PLDI)*, 2008.

[59] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Low-level liquid types. In *Principles of Programming Languages (POPL)*, 2010.

[60] Taro Sekiyama and Atsushi Igarashi. Personal communication, November 2013.

[61] Taro Sekiyama and Atsushi Igarashi. Logical relations for a manifest calculus, fixed. In preparation for submission., 2013.

[62] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Tom Ridge, Susmit Sarkar, and Rok Strnisa. Ott: effective tool support for the working semanticist. In *International Conference on Functional Programming (ICFP)*, 2007.

[63] Zhong Shao. Flexible representation analysis. In *International Conference on Functional Programming (ICFP)*, pages 85–98, Amsterdam, The Netherlands, June 1997.

[64] Avraham Shinnar. *Safe and Effective Contracts*. PhD thesis, Harvard University, May 2011.

[65] Jeremy Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. In Giuseppe Castagna, editor, *Programming Languages and Systems*, volume 5502 of *Lecture Notes in Computer Science*, pages 17–31. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-00589-3. doi:10.1007/978-3-642-00590-9_2.

[66] Jeremy G Siek and Ronald Garcia. Interpretations of the gradually-typed lambda calculus. In *Scheme and Functional Programming (SFP)*, 2012.

[67] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, September 2006.

[68] Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *Principles of Programming Languages (POPL)*, pages 365–376, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. doi:10.1145/1706299.1706342.

[69] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler, and Jacob Matthews, editors. *Revised [6] Report on the Algorithmic Language Scheme*. Cambridge University Press, Cambridge, UK, June 2010. ISBN 9780521193993.

[70] Nikhil Swamy, Michael Hicks, and Gavin M. Bierman. A theory of typed coercions and its applications. In *International Conference on Functional Programming (ICFP)*, ICFP '09, pages 329–340, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi:10.1145/1596550.1596598. URL `http://doi.acm.org/10.1145/1596550.1596598`.

[71] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In *Programming Language Design and Implementation (PLDI)*, pages 387–398, 2013.

[72] Naoshi Tabuchi, Eijiro Sumii, and Akinori Yonezawa. Regular expression types for strings in a text processing language. *Electronic Notes in Theoretical Computer Science*, 2003. doi:10.1016/S1571-0661(04)80781-3. International Workshop in Types in Programming.

[73] Satish Thatte. Quasi-static typing. In *Principles of Programming Languages (POPL)*, 1990.

[74] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *Principles of Programming Languages (POPL)*, 2008.

[75] Sam Tobin-Hochstadt and David Van Horn. Higher-order symbolic execution via contracts. In *OOPSLA*, OOPSLA '12, pages 537–554, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. doi:10.1145/2384616.2384655.

[76] Philip Wadler. Theorems for free! In *Conference on Functional Programming and Computer Architecture (FPCA)*, 1989.

[77] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Scheme and Functional Programming Workshop*, 2007.

[78] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*, 2009.

[79] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

[80] Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static contract checking for haskell. In *Principles of Programming Languages (POPL)*, 2009.