

Elaborating Evaluation-Order Polymorphism

Joshua Dunfield

University of British Columbia
Vancouver, Canada
joshdunf@cs.ubc.ca

Abstract

We classify programming languages according to evaluation order: each language fixes one evaluation order as the default, making it transparent to program in that evaluation order, and troublesome to program in the other.

This paper develops a type system that is impartial with respect to evaluation order. Evaluation order is implicit in terms, and explicit in types, with by-value and by-name versions of type connectives. A form of intersection type quantifies over evaluation orders, describing code that is agnostic over (that is, polymorphic in) evaluation order. By allowing such generic code, programs can express the by-value and by-name versions of a computation without code duplication.

We also formulate a type system that only has by-value connectives, plus a type that generalizes the difference between by-value and by-name connectives: it is either a suspension (by name) or a “no-op” (by value). We show a straightforward encoding of the impartial type system into the more economical one. Then we define an elaboration from the economical language to a call-by-value semantics, and prove that elaborating a well-typed source program, where evaluation order is implicit, produces a well-typed target program where evaluation order is explicit. We also prove a simulation between evaluation of the target program and reductions (either by-value or by-name) in the source program.

Finally, we prove that typing, elaboration, and evaluation are faithful to the type annotations given in the source program: if the programmer only writes by-value types, no by-name reductions can occur at run time.

Categories and Subject Descriptors F3.3 [Mathematical Logic and Formal Languages]: Studies of Program Constructs—Type structure

Keywords evaluation order, intersection types, polymorphism

1. Introduction

It is customary to distinguish languages according to how they pass function arguments. We tend to treat this as a basic taxonomic distinction: for example, OCaml is a call-by-value language, while Haskell is call-by-need. Yet this taxonomy has been dubious from the start: Algol-60, in which arguments were call-by-name by default, also supported call-by-value. For the λ -calculus, Plotkin (1975) showed how to use *administrative reductions* to translate a

cbv program into one that behaves equivalently under cbn evaluation, and vice versa. Thus, one can write a call-by-name program in a call-by-value language, and a call-by-value program in a call-by-name language, but at the price of administrative burdens: creating and forcing thunks (to simulate call-by-name), or using special strict forms of function application, binding, etc. (to simulate call-by-value).

But programmers rarely want to encode an entire program into a different evaluation order. Rather, the issue is how to use the other evaluation order in *part* of a program. For example, game search can be expressed elegantly using a lazy tree, but in an ordinary call-by-value language one must explicitly create and force thunks. Conversely, a big advantage of call-by-value semantics is the relative ease of reasoning about cost (time and space); to recover some of this ease of reasoning, languages that are not call-by-value often have strict versions of function application and strictness annotations on types.

An impartial type system. For any given language, the language designers’ favourite evaluation order is the linguistically *unmarked* case. Programmers are not forced to use that order, but must do extra work to use another, even in languages with mechanisms specifically designed to mitigate these burdens, such as a *lazy* keyword (Wadler et al. 1998).

The first step we’ll take in this paper is to stop playing favourites: our source language allows each evaluation order to be used as easily as the other. Our *impartial type system* includes by-value and by-name versions of function types ($\overset{V}{\rightarrow}$, $\overset{N}{\rightarrow}$), product types ($*^V$, $*^N$), sum types ($+^V$, $+^N$) and recursive types (μ^V , μ^N). Using bidirectional typing, which distinguishes checking and inference, we can use information found in the types of functions to determine whether an unmarked λ or application should be interpreted as call-by-name or call-by-value.

What if we want to define the same operation over both evaluation orders, say *compose*, or *append* (that is, for strict and lazy lists)? Must we write two identical versions, with nearly-identical type annotations? No: We can use polymorphism based on intersection types. The abstruse reputation of intersection types is belied by a straightforward formulation as implicit products (Dunfield 2014), a notion also used by Chen et al. (2014) to express polymorphism over a finite set of levels (though without using the word “intersection”). In these papers’ type systems, elaboration takes a polymorphic source program and produces a target program explicitly specifying necessary, but tedious, constructs. For Dunfield (2014), the extra constructs introduce and eliminate the products that were implicit in the source language; for Chen et al. (2014), the extra constructs support a dynamic dependency graph for efficient incremental computation.

In this paper, we express the intersection type \wedge as a universal quantifier over evaluation orders. For example, the type $\forall a. \text{int} \overset{a}{\rightarrow} \text{int}$ corresponds to $(\text{int} \overset{V}{\rightarrow} \text{int}) \wedge (\text{int} \overset{N}{\rightarrow} \text{int})$. Thus, we can type code that is generic over evaluation orders. Datatype defini-

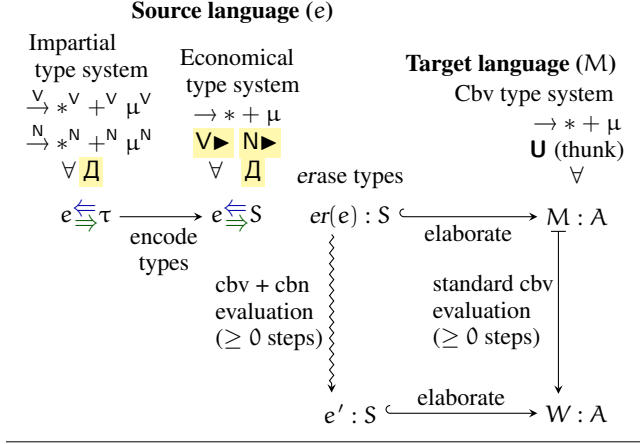


Figure 1. Encoding and elaboration

tions, expressed as recursive/sum types, can also be polymorphic in evaluation order; for example, operations on binary search trees can be written just once. Much of the theory in this paper follows smoothly from existing work on intersection types, particularly Dunfield (2014). However, since we only consider intersections equivalent to the quantified type $\Delta a. A$, our intersected types have parametric structure: they differ only in the evaluation orders decorating the connectives. This limitation, a cousin of the *refinement restriction* in datasort refinement systems (Freeman and Pfenning 1991; Davies 2005), avoids the need for a merge construct (Reynolds 1996; Dunfield 2014) and the issues that arise from it.

A simple, fine-grained type system. The source language just described meets our goal of impartiality, but the large number of connectives yields a slightly unwieldy type system. Fortunately, we can refine this system by abstracting out the differences between the by-name and by-value versions of each connective. That is, each by-name connective corresponds to a by-value connective with suspensions (thunks) added: the by-name function type $S_1 \xrightarrow{N} S_2$ corresponds to $(\mathbf{U} S_1) \rightarrow S_2$ where \rightarrow is by-value, whereas $S_1 \xrightarrow{V} S_2$ is simply $S_1 \rightarrow S_2$. Here, $\mathbf{U} S_1$ is a *thunk type*—essentially $\mathbf{1} \rightarrow S_1$. We realize this difference through a connective $\epsilon \triangleright S$, read “ ϵ suspend S ”, where $\mathbf{N} \triangleright S$ corresponds to $\mathbf{U} S$ and $\mathbf{V} \triangleright S$ is equivalent to S . This gives an economical type system with call-by-value versions of the usual connectives ($\rightarrow, *, +, \mu$), plus $\epsilon \triangleright S$. This type system is biased towards call-by-value (with call-by-name being “marked”), but we can easily encode the impartial connectives: $S_1 \xrightarrow{\epsilon} S_2$ becomes $(\epsilon \triangleright S_1) \rightarrow S_2$, the sum type $S_1 +^e S_2$ becomes $\epsilon \triangleright (S_1 + S_2)$, etc.

Another advantage of this type system is that, in combination with polymorphism, it is simple to define variants of data structures that mix different evaluation orders. For example, a single list definition can encompass lists with strict “next pointers” (so that “walking” the list is guaranteed linear time) and lazy elements (so that examining the element may not be constant time), as well as lists with lazy “next pointers” and strict contents (so that “walking” the list is not guaranteed linear—but once a cons cell has been produced, its element can be accessed in constant time).

Having arrived at this economical type system for source programs, in which evaluation order is implicit in terms, we develop an elaboration that produces a target program in which evaluation order is explicit: thunks are explicitly created and forced, and multiple versions of functions—by-value and by-name—are generated and selected explicitly.

Contributions. This paper makes the following contributions:

- (§2) We define an *impartial* source language and type system that are equally suited to call-by-value and call-by-name. Using a type $\Delta a. \tau$ that quantifies over evaluation orders a , programmers can define data structures and functions that are generic over evaluation order. The type system is bidirectional, alternating between checking an expression against a known type (derived from a type annotation) and synthesizing a type from an expression.
- (§3) Shifting to a call-by-value perspective, we abstract out the suspensions implicit in the by-name connectives, yielding a smaller *economical type system*, also suitable for a (non-impartial) source language. We show that programs well-typed in the impartial type system remain well-typed in the economical type system. Evaluation order remains implicit in terms, and is specified only in type annotations, using the *suspension point* $\epsilon \triangleright S$.
- (§5) We give *elaboration typing* rules from the economical type system into target programs with fully explicit evaluation order. We prove that, given a well-typed source program, the result of the translation is well-typed in a call-by-value target language (Section 4).
- (§6) We prove that the target program behaves like the source program: when the target takes a step from M to M' , the source program that elaborated to M takes some number of steps, yielding an expression that elaborates to M' . We also prove that if a program is typed (in the economical type system) without by-name suspensions, the source program can take only “by-value steps” possible in a cbv semantics. This result exploits a kind of subformula property of the bidirectional type system. Finally, we prove that if a program is impartially typed without using by-value connectives, it can be economically typed without by-name suspensions.

Figure 1 shows the structure of our approach.

Extended version with appendices. Proofs omitted from the main paper for space reasons can be found in Dunfield (2015).

2. Source Language and Impartial Type System

Program variables x
Source expressions $e ::= () \mid x \mid u \mid \lambda x. e \mid e_1 @ e_2 \mid \text{fix } u. e$
 $\mid \Lambda \alpha. e \mid e[\tau] \mid (e : \tau)$
 $\mid (e_1, e_2) \mid \text{proj}_k e$
 $\mid \text{inj}_k e \mid \text{case}(e, x_1.e_1, x_2.e_2)$

Figure 2. Impartial source language syntax

Evaluation order vars. a
Evaluation orders $\epsilon ::= \mathbf{V} \mid \mathbf{N} \mid a$
Type variables α
Valuenesses $\varphi ::= \text{val} \mid \top$
Source types $\tau ::= \mathbf{1} \mid \alpha \mid \forall \alpha. \tau \mid \Delta a. \tau \mid \tau_1 \xrightarrow{\epsilon} \tau_2$
 $\mid \tau_1 *^e \tau_2 \mid \tau_1 +^e \tau_2 \mid \mu^e \alpha. \tau$
Source typing contexts $\gamma ::= \cdot \mid \gamma, x \mid \varphi \Rightarrow \tau \mid \gamma, u \mid \tau \Rightarrow \tau$
 $\mid \gamma, a \text{ evalorder} \mid \gamma, \alpha \text{ type}$

Figure 3. Impartial types for the source language

In our source language (Figure 2), expressions e are the unit value $()$, variables x , abstraction $\lambda x. e$, application $e_1 @ e_2$, fixed points $\text{fix } u. e$ with fixed point variables u , pairs and projections,

and sums $\text{inj}_k e$ with conditionals $\text{case}(e, x_1.e_1, x_2.e_2)$ (shorthand for $\text{case } e \text{ of } \text{inj}_1 x_1 \Rightarrow e_1 \mid \text{inj}_2 x_2 \Rightarrow e_2$). Both of our type systems for this source language—the impartial type system in this section, and the economical type system of Section 3—have features not evident from the source syntax: polymorphism over evaluation orders, and recursive types.

2.1 Values

If we wanted a standard call-by-value language, we would give a grammar for values, and use values to define the operational semantics (and to impose a value restriction on polymorphism introduction). But we want an impartial language, which means that a function argument x is a value *only* if the function is being typed under call-by-value. That is, when checking $(\lambda x. e)$ against type $(\tau \xrightarrow{V} \tau)$, the variable x should be considered a value (it will be replaced with a value at run time), but when checking against $(\tau \xrightarrow{N} \tau)$, it should not be considered a value (it could be replaced with a non-value at run time). Since “valueness” depends on typing, our typing judgments will have to carry information about whether an expression should be considered a value.

We will also use valueness to impose a value restriction on polymorphism over evaluation orders, as well as polymorphism over types; see Section 2.5. In contrast, our operational semantics for the source language (Section 2.4), which permits two flavours (by-value and by-name) of reductions, will use a standard syntactic definition of values in the by-value reductions.

2.2 An impartial type system

In terms of evaluation order, the expressions in Figure 2 are a blank slate. You can imagine them as having whichever evaluation order you prefer. You can write down the typing rules for functions, pairs and sums, and you will get the same rules regardless of which evaluation order you chose. This is the conceptual foundation for many functional languages: start with the simply-typed λ -calculus, choose an evaluation order, and build up the language from there.¹ Our goal here is to allow different evaluation orders to be mixed. As a first approximation, we can try to put evaluation orders in the type system simply by decorating all the connectives. For example, in place of the standard \rightarrow -introduction rule

$$\frac{\gamma, x : \tau_1 \vdash e : \tau_2}{\gamma \vdash (\lambda x. e) : (\tau_1 \rightarrow \tau_2)}$$

we can decorate \rightarrow with an evaluation order ϵ (either V or N):

$$\frac{\gamma, x : \tau_1 \vdash e : \tau_2}{\gamma \vdash (\lambda x. e) : (\tau_1 \xrightarrow{\epsilon} \tau_2)}$$

Products $*$, sums $+$, and recursive types μ follow similarly.

We add a universal quantifier $\Delta a. \tau$ over evaluation orders². Its rules follow the usual type-assignment rules for \forall : the introduction rule is parametric over an arbitrary evaluation order a , and the

¹ The choice need not be easy. The first call-by-name language, Algol 60, also supported call-by-value. It seems that call-by-value was the language committee’s preferred default, but Peter Naur, the editor of the Algol 60 report, independently reversed that decision—which he said was merely one of a “few matters of detail” (Wexelblat 1981, p. 112). A committee member, F.L. Bauer, said this showed that Naur “had absorbed the Holy Ghost after the Paris meeting... there was nothing one could do... it was to be swallowed for the sake of loyalty.” (Wexelblat 1981, p. 130).

² The Cyrillic letter Δ , transliterated into English as D , bears some resemblance to an A (and thus to \forall); more interestingly, it is the first letter of the Russian word да (*da*). Many non-Russian speakers know that this word means “yes”, but another meaning is “and”, connecting it to intersection types.

elimination rule replaces a with a particular evaluation order ϵ :

$$\frac{\gamma, a \text{ evalorder} \vdash e : \tau}{\gamma \vdash e : \Delta a. \tau} \quad \frac{\gamma \vdash e : \Delta a. \tau \quad \gamma \vdash \epsilon \text{ evalorder}}{\gamma \vdash e : [\epsilon/a]\tau}$$

These straightforward rules have a couple of issues:

- Whether a program diverges can depend on whether it is run under call-by-value, or call-by-name. The simply-typed λ -calculus has the same typing rules for call-by-value and call-by-name, because those rules cannot distinguish programs that return something from programs that diverge. Since we want to elaborate to call-by-value or call-by-name depending on which type appeared, evaluation depends on the particular typing derivation. Suppose that evaluation of e_2 diverges, and that f is bound to $(\lambda x. e_1)$. Then whether $f @ e_2$ diverges depends on whether the type of f has \xrightarrow{V} or \xrightarrow{N} . The above rules allow a compiler to make either choice. Polymorphism in the form of Δ aggravates the problem: it is tempting to infer for f the principal type $\Delta a. \dots \xrightarrow{a} \dots$; the compiler can then choose how to instantiate a at each of f ’s call sites. Allowing such code is one of this paper’s goals, but only when the programmer knows that either evaluation order is sensible and has written an appropriate type annotation or module signature.

We resolve this through bidirectional typing, which ensures that quantifiers are introduced only via type annotation (a kind of subformula property). Internal details of the typing derivation still affect elaboration, and thus evaluation, but the internal details will be consistent with programmers’ expressed intent.

- If we extend the language with effects, we may need a value restriction in certain rules. For example, mutable references will break type safety unless we add a value restriction to the introduction rules for \forall and Δ .

A traditional value restriction (Wright 1995) would simply require changing e to v in the introduction rules, where v is a class of syntactic values. In our setting, whether a variable x is a value depends on typing, so a value restriction is less straightforward. We resolve this by extending the typing judgment with information about whether the expression is a value.

Bidirectional typing. We can refine the traditional typing judgment into *checking* and *synthesis* judgments. In the checking judgment $e \Leftarrow \tau$, we already know that e should have type τ , and are checking that e is consistent with this knowledge. In the synthesis judgment $e \Rightarrow \tau$, we extract τ from e itself (perhaps directly from a type annotation), or from assumptions available in a typing context.

The use of bidirectional typing (Pierce and Turner 2000; Dunfield and Krishnaswami 2013) is often motivated by the need to typecheck programs that use features Damas-Milner inference cannot handle, such as indexed and refinement types (Xi 1998; Davies and Pfenning 2000; Dunfield and Pfenning 2004) and higher-rank polymorphism. But decidability is not our motivation for using bidirectional typing. Rather, we want typing to remain predictable even though evaluation order is implicit. By following the approach of Dunfield and Pfenning (2004), in which “introduction forms check, elimination forms synthesize”, we ensure that the evaluation orders in typing match what programmers intended: a type connective with a V or N evaluation order can be introduced *only* by a checking judgment. Since the types in checking judgments are derived from type annotations, they match the programmer’s expressed intent.

Programmers must write annotations on expressions that are redexes: in $(\lambda x. e) @ e_2$, the λ needs an annotation, because $\lambda x. e$ is an introduction form in an elimination position: $[\] @ e_2$. In contrast, $f @ (\lambda x. e_2)$ needs no annotation, though the type of

f must be derived (if indirectly) from an annotation. Recursive functions $\text{fix } u. \lambda x. e$ “reduce” to their unfolding, so they also need annotations.

Valueness. Whether an expression is a value may depend on typing, so we put a *valueness* in the typing judgments: $e \text{ val} \Rightarrow S$ (or $e \text{ val} \Leftarrow S$) means that e at type S is definitely a value, while $e \text{ } \tau \Rightarrow S$ (or $e \text{ } \tau \Leftarrow S$) means that e at type S is not known to be a value. In the style of abstract interpretation, we have a partial order \sqsubseteq such that $\text{val} \sqsubseteq \top$. Then the *join* $\varphi_1 \sqcup \varphi_2$ is val when $\varphi_1 = \varphi_2 = \text{val}$, and \top otherwise. g Since valueness is just a projection of ϵ , we could formulate the system without it, using ϵ to mark judgments as denoting values (V) or possible nonvalues (N). But that seems prone to confusion: is $\text{N} \Leftarrow$ saying the expression is “by name” in some sense?

Types and typing contexts. In Figure 3 we show the grammar for evaluation orders ϵ , which are either by-value (V), by-name (N), or an evaluation order variable a . We have the unit type $\mathbf{1}$, type variables α , ordinary parametric polymorphism $\forall \alpha. \tau$, evaluation order polymorphism $\Delta a. \tau$, functions $\tau_1 \xrightarrow{\epsilon} \tau_2$, products $\tau_1 *^{\epsilon} \tau_2$, sums $\tau_1 +^{\epsilon} \tau_2$, and recursive types $\mu^{\epsilon} \alpha. \tau$.

A source typing context γ consists of variable declarations $x \text{ } \varphi \Rightarrow \tau$ denoting that x has type τ with valueness φ , fixed-point variable declarations $u \text{ } \tau \Rightarrow \tau$ (fixed-point variables are never values), evaluation-order variable declarations $a \text{ evalorder}$, and type variable declarations $\alpha \text{ type}$.

Impartial typing judgments. Figure 4 shows the bidirectional rules for impartial typing. The judgment forms are $\gamma \vdash_{\text{I}} e \text{ } \varphi \Leftarrow \tau$, meaning that e checks against τ (with valueness φ), and $\gamma \vdash_{\text{I}} e \text{ } \varphi \Rightarrow \tau$, meaning that e synthesizes type τ . The “I” on the turnstile stands for “impartial”.

Connective-independent rules. Rules **Ivar** and **Ifixvar** simply use assumptions stored in γ . Rule **Ifix** checks a fixed point $\text{fix } u. e$ against type τ by introducing the assumption $u \text{ } \tau \Rightarrow \tau$ and checking e against τ ; its premise has valueness φ because even if e is a value, $\text{fix } u. e$ is not (\top in the conclusion).

Rule **Sub** says that if e synthesizes τ then e checks against τ . For example, in the (ill-advised) fixed point expression $\text{fix } u. u$, the premise of **Ifix** tries to check u against τ , but **Ifixvar** derives a synthesis judgment, not a checking judgment; **Sub** bridges the gap.

Rule **Ianno** also mediates between synthesis and checking, in the opposite direction: if we can check an expression e against an annotated type τ , then $(e : \tau)$ synthesizes τ .

Introductions and eliminations. The rest of the rules are linked to type connectives. For easy reference, the figure shows each connective to the left of its introduction and elimination rules. We follow the recipe of Dunfield and Pfenning (2004): introduction rules check, and elimination rules synthesize. This recipe yields the smallest sensible set of rules, omitting some rules that are not absolutely necessary but can be useful in practice. For example, our rules never synthesize a type for an unannotated pair, because the pair is an introduction form.

Rule **I+Elim** follows the recipe, despite having a checking judgment in its conclusion: the connective being eliminated, $+^{\epsilon}$, is synthesized (in the first premise).

Functions. Rule **I→Intro** introduces the type $\tau_1 \xrightarrow{\epsilon} \tau_2$. Its premise adds an assumption $x \text{ valueness}(\epsilon) \Rightarrow \tau_1$, where $\text{valueness}(\epsilon)$ is val if $\epsilon = \text{V}$, and \top if ϵ is N or is an evaluation-order variable a . This rule thereby encompasses both variables that will be substituted with values ($\text{valueness}(\epsilon) = \text{val}$) and variables that might be substituted with non-values ($\text{valueness}(\epsilon) = \top$). Applying a function of type $\tau_1 \xrightarrow{\epsilon} \tau_2$ yields something of type τ_2 regardless of ϵ , so **I→Elim** ignores ϵ .

Consistent with the usual definition of syntactic values, **I→Intro**’s conclusion has val , while **I→Elim**’s conclusion has \top .

In rule **I→Elim**, the first premise has the connective to eliminate, so the first premise synthesizes $(\tau_1 +^{\epsilon} \tau_2)$. This provides the type τ_1 , so the second premise is a checking judgment; it also provides τ_2 , so the conclusion synthesizes.

Products. Rule **I*Intro** types a value if and only if both e_1 and e_2 are typed as values, so its conclusion has valueness $\varphi_1 \sqcup \varphi_2$.

Sums. Rule **I+Intro_k** is straightforward. In rule **I+Elim**, the assumptions added to γ in the branches say that x_1 and x_2 are values (val), because our by-name sum type is “by-name” on the *outside*. This point should become more clear when we see the translation of types into the economical system.

Recursive types. Rules **IμIntro** and **IμElim** have the same e in the premise and conclusion, without explicit “roll” and “unroll” constructs. In a non-bidirectional type inference system, this would be awkward since the expression doesn’t give direct clues about when to apply these rules. In this bidirectional system, the type tells us to apply **IμIntro** (since its conclusion is a checking judgment). Knowing when to apply **IμElim** is more subtle: we should try to apply it whenever we need to synthesize some *other* type connective. For instance, the first premise of **I+Elim** needs a $+$, so if we synthesize a μ -type we should apply **IμElim** in the hope of exposing a $+$.

The lack of explicit [un]rolls suggests that these are not iso-recursive but equi-recursive types (Pierce 2002, chapter 20). However, we don’t semantically equate a recursive type with its unfolding, so perhaps they should be called *implicitly* iso-recursive.

Note that an implementation would need to check that the type under the μ is guarded by a type connective that does have explicit constructs, to rule out types like $\mu^{\epsilon} \alpha. \alpha$, which is its own unfolding and could make the typechecker run in circles.

Explicit type polymorphism. In contrast to recursive types, we explicitly introduce and eliminate type polymorphism via the expressions $\Lambda \alpha. e$ and $M[\tau]$. This guarantees that a \forall can be instantiated with a type containing a particular evaluation order if and only if such a type appears in the source program.

Principality. Suppose $\gamma \vdash_{\text{I}} e_1 \text{ } \varphi \Rightarrow \Delta a. \tau_1 \rightarrow \tau_2$. Then, for any ϵ , we can derive $\gamma \vdash_{\text{I}} e_1 \text{ } \epsilon \Rightarrow \tau_2$. But we can’t use **IΔIntro** to derive the type $\Delta a'. [a'/a]\tau_2$, because $e_1 \text{ } \epsilon \Rightarrow \tau_2$. The only sense in which this expression has a principal type is if we have an evaluation-order variable in γ that we can substitute for a .

2.3 Programming with polymorphic evaluation order

Lists and streams. The impartial type system can express lists and (potentially terminating) streams in a single declaration:

$$\text{type List } a \alpha = \mu^a \beta. (\mathbf{1} +^a (\alpha *^a \beta))$$

Choosing $a = \text{V}$ yields $\mu^{\text{V}} \beta. (\mathbf{1} +^{\text{V}} (\alpha *^{\text{V}} \beta))$, which is the type of lists of elements α . Choosing $a = \text{N}$ yields $\mu^{\text{N}} \beta. (\mathbf{1} +^{\text{N}} (\alpha *^{\text{N}} \beta))$, which is the type of streams that may end—essentially, lazy lists. Since evaluation order is implicit in source expressions, we can write operations on $\text{List } a \alpha$ that work for lists *and* streams:

$$\begin{aligned} \text{map} &: \Delta a. \forall \alpha. (\alpha \xrightarrow{\text{V}} \beta) \xrightarrow{\text{V}} (\text{List } a \alpha) \xrightarrow{\text{V}} (\text{List } a \beta) \\ &= \Lambda \alpha. \text{fix map}. \lambda f. \lambda xs. \\ &\quad \text{case}(xs, x_1.\text{inj}_1(), \\ &\quad \quad x_2.\text{inj}_2(f @ (\text{proj}_1 x_2), \text{map } @ f @ (\text{proj}_2 x_2))) \end{aligned}$$

This sugar-free syntax bristles; in an implementation with conveniences like pattern-matching on tuples and named constructors, we could write

$\boxed{\text{valuenss}(\epsilon) = \varphi}$ Evaluation order ϵ maps to valuenss φ

$\text{valuenss}(\mathbf{V}) = \text{val}$
 $\text{valuenss}(\mathbf{N}) = \top$
 $\text{valuenss}(a) = \top$

$\boxed{\Gamma \vdash_I e \varphi \Leftarrow \tau}$ Source expression e checks against impartial type τ
 $\boxed{\Gamma \vdash_I e \varphi \Rightarrow \tau}$ Source expression e synthesizes impartial type τ

$$\begin{array}{c}
\frac{(x \varphi \Rightarrow \tau) \in \gamma}{\gamma \vdash_I x \varphi \Rightarrow \tau} \text{Ivar} \quad \frac{(u \top \Rightarrow \tau) \in \gamma}{\gamma \vdash_I u \top \Rightarrow \tau} \text{Ifixvar} \quad \frac{\gamma, u \top \Rightarrow \tau \vdash_I e \varphi \Leftarrow \tau}{\gamma \vdash_I (\text{fix } u. e) \top \Leftarrow \tau} \text{Ifix} \quad \frac{\gamma \vdash_I e \varphi \Rightarrow \tau}{\gamma \vdash_I e \varphi \Leftarrow \tau} \text{Isub} \quad \frac{\gamma \vdash_I e \varphi \Leftarrow \tau}{\gamma \vdash_I (e : \tau) \varphi \Rightarrow \tau} \text{Ianno} \\
\\
\mathbf{V} \quad \frac{\gamma, \alpha \text{ type } \vdash_I e \text{ val} \Leftarrow \tau}{\gamma \vdash_I \Lambda \alpha. e \text{ val} \Leftarrow \forall \alpha. \tau} \text{IVIntro} \quad \frac{\gamma \vdash_I e \varphi \Rightarrow \forall \alpha. \tau \quad \gamma \vdash \tau' \text{ type}}{\gamma \vdash_I e[\tau'] \varphi \Rightarrow [\tau'/\alpha]\tau} \text{IVElim} \quad \mathbf{1} \quad \frac{}{\gamma \vdash_I () \text{ val} \Leftarrow \mathbf{1}} \text{IIIntro} \\
\\
\mathbf{D} \quad \frac{\gamma, a \text{ evalorder } \vdash_I e \text{ val} \Leftarrow \tau}{\gamma \vdash_I e \text{ val} \Leftarrow \mathbf{D}a. \tau} \text{IDIntro} \quad \frac{\gamma \vdash_I e \varphi \Rightarrow \mathbf{D}a. \tau \quad \gamma \vdash \epsilon \text{ evalorder}}{\gamma \vdash_I e \varphi \Rightarrow [\epsilon/a]\tau} \text{IDElim} \\
\\
\rightarrow \quad \frac{\epsilon}{\gamma \vdash_I (\lambda x. e) \text{ val} \Leftarrow (\tau_1 \xrightarrow{\epsilon} \tau_2)} \text{I}\rightarrow\text{Intro} \quad \frac{\gamma \vdash_I e_1 \varphi_1 \Rightarrow (\tau_1 \xrightarrow{\epsilon} \tau_2) \quad \gamma \vdash_I e_2 \varphi_2 \Leftarrow \tau_1}{\gamma \vdash_I (e_1 @ e_2) \top \Rightarrow \tau_2} \text{I}\rightarrow\text{Elim} \\
\\
* \epsilon \quad \frac{\gamma \vdash_I e_1 \varphi_1 \Leftarrow \tau_1 \quad \gamma \vdash_I e_2 \varphi_2 \Leftarrow \tau_2}{\gamma \vdash_I (e_1, e_2) \varphi_1 \sqcup \varphi_2 \Leftarrow (\tau_1 *^{\epsilon} \tau_2)} \text{I*Intro} \quad \frac{\gamma \vdash_I e \varphi \Rightarrow (\tau_1 *^{\epsilon} \tau_2)}{\gamma \vdash_I (\text{proj}_k e) \top \Rightarrow \tau_k} \text{I*Elim}_k \\
\\
+ \epsilon \quad \frac{\gamma \vdash_I e \varphi \Leftarrow \tau_k}{\gamma \vdash_I (\text{inj}_k e) \varphi \Leftarrow (\tau_1 +^{\epsilon} \tau_2)} \text{I+Intro}_k \quad \frac{\gamma \vdash_I e \varphi_0 \Rightarrow (\tau_1 +^{\epsilon} \tau_2) \quad \gamma, (x_1 \text{ val} \Rightarrow \tau_1) \vdash_I e_1 \varphi_1 \Leftarrow \tau \quad \gamma, (x_2 \text{ val} \Rightarrow \tau_2) \vdash_I e_2 \varphi_2 \Leftarrow \tau}{\gamma \vdash_I \text{case}(e, x_1.e_1, x_2.e_2) \top \Leftarrow \tau} \text{I+Elim} \\
\\
\mu \epsilon \quad \frac{\gamma \vdash_I e \varphi \Leftarrow [(\mu^{\epsilon} \alpha. \tau)/\alpha]\tau}{\gamma \vdash_I e \varphi \Leftarrow \mu^{\epsilon} \alpha. \tau} \text{I}\mu\text{Intro} \quad \frac{\gamma \vdash_I e \varphi \Rightarrow \mu^{\epsilon} \alpha. \tau}{\gamma \vdash_I e \top \Rightarrow [(\mu^{\epsilon} \alpha. \tau)/\alpha]\tau} \text{I}\mu\text{Elim}
\end{array}$$

Figure 4. Impartial bidirectional typing for the source language

$$\begin{aligned}
\text{map } f \text{ xs} &: \mathbf{D}a. \forall \alpha. (\alpha \xrightarrow{\mathbf{V}} \beta) \xrightarrow{\mathbf{V}} (\text{List } a \ \alpha) \xrightarrow{\mathbf{V}} (\text{List } a \ \beta) \\
&= \text{case } \text{xs} \text{ of Nil} \Rightarrow \text{Nil} \\
&\quad | \text{Cons}(\text{hd}, \text{tl}) \Rightarrow \text{Cons}(f \ \text{hd}, \text{map } f \ \text{tl})
\end{aligned}$$

Note that, except for the type, this is standard code for *map*. Even this small example raises interesting questions:

- Must *all* the connectives in List have a? No. Putting a on either the μ or the $+$ and writing \mathbf{V} on the other connectives is enough to get stream behaviour when a is instantiated with \mathbf{N} : the only reason to eliminate (unroll) the μ is to eliminate (case on) the $+$; marking either connective will suspend the underlying computation. Marking both μ and $+$ induces a suspension of a suspension, where forcing the outer suspension immediately forces the inner one; one of the suspensions is superfluous.

Note that marking only $*$ with a, that is, $\mu^{\mathbf{V}}\beta. (\mathbf{1} +^{\mathbf{V}} (\alpha *^{\mathbf{a}} \beta))$, yields an “odd” data structure (Wadler et al. 1998), one that is not entirely lazy: we know immediately—without forcing a thunk—which injection we have (i.e. whether we have Nil or Cons).

- What evaluation orders should we use in the type of *map*? We used by-value ($\xrightarrow{\mathbf{V}}$), but we could use the same evaluation order as the list: $\mathbf{D}a. \forall \alpha. (\alpha \xrightarrow{\mathbf{a}} \beta) \xrightarrow{\mathbf{a}} (\text{List } a \ \alpha) \xrightarrow{\mathbf{a}} (\text{List } a \ \beta)$. This essentially gives “ML-ish” behaviour when $a = \mathbf{V}$, and “Haskell-ish” behaviour when $a = \mathbf{N}$. The type system, however, permits other variants—even the outlandishly generic

$$\mathbf{D}a_1, a_2, a_3, a_4, a_5. \forall \alpha. (\alpha \xrightarrow{a_1} \beta) \xrightarrow{a_2} (\text{List } a_3 \ \alpha) \xrightarrow{a_4} (\text{List } a_5 \ \beta)$$

We leave deeper investigation of these questions to future work: our purpose, in this paper, is to develop the type systems that make such questions matter.

Variations in being odd and even. The Standard ML type of “streams in odd style” (Wadler et al. 1998, Fig. 1), given by

$$\text{datatype } \alpha \text{ stream} = \text{Nil} \mid \text{Cons of } \alpha * \alpha \text{ stream susp}$$

where $\alpha \text{ stream susp}$ is the type of a thunk that yields an $\alpha \text{ stream}$, can be represented as the impartial type $\mu^{\mathbf{V}}\beta. (\mathbf{1} +^{\mathbf{V}} (\alpha *^{\mathbf{V}} (\mu^{\mathbf{N}}\gamma. \beta)))$. Note the slightly awkward $(\mu^{\mathbf{N}}\gamma. \beta)$, in which γ doesn’t occur; we can’t simply write $\mu^{\mathbf{N}}\beta$. on the outside, because that would suspend the entire sum. (In the economical type system in Section 3, it’s easy to put the suspension in either position.) This type differs subtly from another “odd” stream type, $\mu^{\mathbf{V}}\beta. (\mathbf{1} +^{\mathbf{V}} (\alpha *^{\mathbf{a}} \beta))$, which corresponds to the SML type

$$\text{datatype } \alpha \text{ stream} = \text{Nil} \mid \text{Cons of } \boxed{\alpha * \alpha \text{ stream}} \text{ susp}$$

Here, the contents α are under the suspension; given a value of this type, we immediately know whether we have Nil or Cons, but we must force a thunk to see what the value is, which will also reveal whether the tail is Nil or Cons.

We can also encode “streams in even style” (Wadler et al. 1998, Fig. 2): The SML declarations

$$\begin{aligned}
\text{datatype } \alpha \text{ stream}_- &= \text{Nil}_- \mid \text{Cons}_- \text{ of } \alpha * \alpha \text{ stream} \\
\text{withtype } \alpha \text{ stream} &= \alpha \text{ stream}_- \text{ susp}
\end{aligned}$$

correspond to $\mu^{\mathbf{N}}\beta. (\mathbf{1} +^{\mathbf{V}} (\alpha *^{\mathbf{V}} \beta))$, with the \mathbf{N} on μ playing the role of the *withtype* declaration.

Wadler et al. (1998) note that “streams in odd style” can be encoded with ease in SML, while “streams in even style” can be encoded with difficulty (see their Figure 2). In the impartial type system, both encodings are straightforward, and we would only need to write one (polymorphic) version of each of their functions over streams.

Source values $v ::= () \mid \lambda x. e \mid (v_1, v_2) \mid \text{inj}_k v$

By-value eval. contexts $C_V ::= []$
 $C_V @ e_2 \mid v_1 @ C_V$
 $(C_V, e_2) \mid (v_1, C_V) \mid \text{proj}_k C_V$
 $\text{inj}_k C_V \mid \text{case}(C_V, x_1.e_1, x_2.e_2)$

By-name eval. contexts $C_N ::= []$
 $C_N @ e_2 \mid e_1 @ C_N$
 $(C_N, e_2) \mid (e_1, C_N) \mid \text{proj}_k C_N$
 $\text{inj}_k C_N \mid \text{case}(C_N, x_1.e_1, x_2.e_2)$

$e \rightsquigarrow e'$ Source expression e steps to e'

$$\frac{e \rightsquigarrow_{RV} e'}{C_V[e] \rightsquigarrow C_V[e']} \text{SrcStepCtxV} \quad \frac{e \rightsquigarrow_{RN} e'}{C_N[e] \rightsquigarrow C_N[e']} \text{SrcStepCtxN}$$

$e \rightsquigarrow_{RV} e'$ e reduces to e' by value
 $e \rightsquigarrow_{RN} e'$ e reduces to e' by name

$(\lambda x. e_1) @ v_2 \rightsquigarrow_{RV} [v_2/x]e_1$	$\beta V \text{reduce}$
$(\lambda x. e_1) @ e_2 \rightsquigarrow_{RN} [e_2/x]e_1$	$\beta N \text{reduce}$
$(\text{fix } u. e) \rightsquigarrow_{RV} [(\text{fix } u. e)/u]e$	$\text{fix } V \text{reduce}$
$(\text{fix } u. e) \rightsquigarrow_{RN} [(\text{fix } u. e)/u]e$	$\text{fix } N \text{reduce}$
$\text{proj}_k (v_1, v_2) \rightsquigarrow_{RV} v_k$	$\text{proj } V \text{reduce}$
$\text{proj}_k (e_1, e_2) \rightsquigarrow_{RN} e_k$	$\text{proj } N \text{reduce}$
$\text{case}(\text{inj}_k v, x_1.e_1, x_2.e_2) \rightsquigarrow_{RV} [v/x_k]e_k$	$\text{case } V \text{reduce}$
$\text{case}(\text{inj}_k e, x_1.e_1, x_2.e_2) \rightsquigarrow_{RN} [e/x_k]e_k$	$\text{case } N \text{reduce}$

Figure 5. Source reduction

$er(e) = e'$ Source expression e erases to e'

$er(\lambda \alpha. e) = er(e)$	$er(()) = ()$
$er(e[S]) = er(e)$	$er(x) = x$
$er(e:S) = er(e)$	$er(e_1 @ e_2) = er(e_1) @ er(e_2)$
	etc.

Figure 6. Erasing types from source expressions

Binary trees. As with lists, we can define evaluation-order-polymorphic trees:

$\text{type Tree } a \alpha = \mu^a \beta. (\mathbf{1} +^V (\alpha *^V \beta *^V \beta))$

Here, only μ is polymorphic in a , to suppress redundant thinks.

2.4 Operational semantics for the source language

A source expression takes a step if a subterm in evaluation position can be reduced. We want to model by-value computation *and* by-name computation, so we define the source stepping relation \rightsquigarrow usings two notions of evaluation position and two notions of reduction. A *by-value evaluation context* C_V is an expression with a hole $[]$, where $C_V[e]$ is the expression with e in place of the $[]$. If e reduces by value to e' , written $e \rightsquigarrow_{RV} e'$, then $C_V[e] \rightsquigarrow C_V[e']$. For example, if $e_2 \rightsquigarrow_{RV} e'_2$ then $v_1 @ e_2 \rightsquigarrow v_1 @ e'_2$, because $v_1 @ []$ is a by-value evaluation context.

Dually, $C_N[e] \rightsquigarrow C_N[e']$ if $e \rightsquigarrow_{RN} e'$. Every by-value context is a by-name context, and every pair related by \rightsquigarrow_{RV} is also related by \rightsquigarrow_{RN} , but the converses do not hold. For instance, $e_1 @ []$ is a C_N but not a C_V , and $\text{proj}_2 (e_1, e_2) \rightsquigarrow_{RN} e_2$, but $\text{proj}_2 (e_1, e_2)$ reduces by value only when e_1 and e_2 are values.

Values, by-value evaluation contexts C_V , by-name evaluation contexts C_N , and the relations \rightsquigarrow , \rightsquigarrow_{RV} and \rightsquigarrow_{RN} are defined in

Figure 5. The definitions of v , C_V and \rightsquigarrow_{RV} , taken together, are standard for call-by-value; the definitions of C_N and \rightsquigarrow_{RN} are standard for call-by-name. The peculiarity is that \rightsquigarrow can behave either by value (rule SrcStepCtxV) or by name (rule SrcStepCtxN).

We assume that the expressions being reduced have been erased (Figure 6), so we omit a rule for reducing annotations. Alternatives are discussed in Section 6.1.

2.5 Value restriction

Our calculus excludes effects such as mutable references; however, to allow it to serve as a basis for larger languages, we impose a value restriction on certain introduction rules. Without this restriction, the system would be unsound in the presence of mutable references. Following Wright (1995), the rule IVIntro requires that its subject be a value, as in Standard ML (Milner et al. 1997). A similar value restriction is needed for intersection types (Davies and Pfenning 2000). The following example shows the need for the restriction on Δ :

$$\text{let } r : \text{ref } (\Delta a. \tau \overset{a}{\rightarrow} \tau) = \text{ref } f \text{ in}$$

$$r := g; \quad h(!r)$$

Assume we have $f : \Delta a. \tau \overset{a}{\rightarrow} \tau$ and $g : \tau \overset{N}{\rightarrow} \tau$ and $h : (\tau \overset{V}{\rightarrow} \tau) \overset{V}{\rightarrow} \tau$. By a version of IDIntro that doesn't require its subject to be a value, we have $r : \Delta a. \text{ref } (\tau \overset{a}{\rightarrow} \tau)$. By IDElim with N for a , we have $r : \text{ref } (\tau \overset{N}{\rightarrow} \tau)$, making the assignment $r := g$ well-typed. However, by IDElim with V for a , we have $r : \text{ref } (\tau \overset{V}{\rightarrow} \tau)$. It follows that the dereference $!r$ has type $\tau \overset{V}{\rightarrow} \tau$, so $!r$ can be passed to h . But $!r = g$ is actually call-by-name. If $h = \lambda x. x(e_2)$, we should be able to assume that e_2 will be evaluated exactly once, but $x = g$ is call-by-name, violating this assumption.

If we think of Δ as an intersection type, so that r has type $(\tau \overset{V}{\rightarrow} \tau) \wedge (\tau \overset{N}{\rightarrow} \tau)$, the example and argument closely follow Davies and Pfenning (2000) and, in turn, Wright (1995). (For union types, a similar problem arises, which can be solved by a dual solution—restricting the union-elimination rule to evaluation contexts (Dunfield and Pfenning 2003).)

2.6 Subtyping and η -expansion

Systems with intersection types often include subtyping. The strength of subtyping in intersection type systems varies, from syntactic approaches that emphasize simplicity (e.g. Dunfield and Pfenning (2003)) to semantic approaches that emphasize completeness (e.g. Frisch et al. (2002)). Generally, subtyping—at minimum—allows intersections to be transparently eliminated even at higher rank (that is, to the left of an arrow), so that the following function application is well-typed:

$$f : ((\tau_1 \wedge \tau'_1) \rightarrow \tau_2) \rightarrow \tau_3, \quad g : (\tau_1 \rightarrow \tau_2) \vdash f g : \tau_3$$

Through a subsumption rule, $g : (\tau_1 \rightarrow \tau_2)$ checks against type $(\tau_1 \wedge \tau'_1) \rightarrow \tau_2$, because a function that accepts all values of type τ_1 should also accept all values that have type τ_1 *and* type τ'_1 .

Using the analogy between intersection and Δ , in our impartial type system, we might expect to derive

$$f : ((\Delta a. \tau_1 \overset{a}{\rightarrow} \tau_1) \overset{V}{\rightarrow} \tau_2) \overset{V}{\rightarrow} \tau_3, \quad g : (\tau_1 \overset{N}{\rightarrow} \tau_1) \overset{V}{\rightarrow} \tau_2 \vdash f g : \tau_3$$

Here, f asks for a function of type $(\Delta a. \tau_1 \overset{a}{\rightarrow} \tau_1) \overset{V}{\rightarrow} \tau_2$, which works on all evaluation orders; but g 's type $(\tau_1 \overset{N}{\rightarrow} \tau_1) \overset{V}{\rightarrow} \tau_2$ says that g calls its argument only by name.

For simplicity, this paper excludes subtyping: our type system does not permit this derivation. But it would be possible to define a subtyping system, and incorporate subtyping into the subsumption rule Isub —either by treating Δ similarly to \forall (Dunfield and Krishnaswami 2013), or by treating Δ as an intersection type (Dunfield

and Pfenning 2003). A simple subtyping system could be derived from the typing rules that are *stationary*—where the premises type the same expression as the conclusion (Leivant 1986). For example, $\mathbb{I}\Delta\text{Elim}$ corresponds to

$$\frac{\Gamma \vdash e \text{ evalorder}}{\Gamma \vdash (\Delta a. \tau) \leq [\epsilon/a]\tau} \leq \Delta\text{-LEFT}$$

Alternatively, η -expansion can substitute for subtyping: even without subtyping and a subsumption rule, we can derive

$$\begin{aligned} f &: ((\Delta a. \tau_1 \xrightarrow{a} \tau_1) \rightarrow \tau_2) \rightarrow \tau_3, \\ g &: (\tau_1 \xrightarrow{N} \tau_1) \rightarrow \tau_2 \vdash f(\lambda x. g x) : \tau_3 \end{aligned}$$

This idea, developed by Barendregt et al. (1983), can be automated; see, for example, Dunfield (2014).

3. Economical Type System

$[\tau] = S$	Impartial type τ translates to economical type S
$[1] = \mathbf{1}$	$[\Delta a. \tau] = \Delta a. [\tau]$
$[\tau_1 \xrightarrow{\epsilon} \tau_2] = (\epsilon \blacktriangleright [\tau_1]) \rightarrow [\tau_2]$	$[\mu^\epsilon \alpha. \tau] = \mu \alpha. \epsilon \blacktriangleright [\tau]$
$[\tau_1 +^\epsilon \tau_2] = \epsilon \blacktriangleright ([\tau_1] + [\tau_2])$	$[\forall \alpha. \tau] = \forall \alpha. [\tau]$
$[\tau_1 *^\epsilon \tau_2] = (\epsilon \blacktriangleright [\tau_1]) * (\epsilon \blacktriangleright [\tau_2])$	$[\alpha] = \alpha$
$[\gamma] = \Gamma$	Impartial context γ translates to economical context Γ
$[\cdot] = \cdot$	$[\gamma, a \text{ evalorder}] = [\gamma], a \text{ evalorder}$
$[\gamma, \alpha \text{ type}] = [\gamma], \alpha \text{ type}$	$[\gamma, x \text{ val} \Rightarrow \tau] = [\gamma], x : \mathbf{V} \blacktriangleright [\tau]$
$[\gamma, u \tau \Rightarrow \tau] = [\gamma], u : [\tau]$	$[\gamma, x \tau \Rightarrow \tau] = [\gamma], x : \mathbf{N} \blacktriangleright [\tau]$
$[e] = e'$	Expression e with τ -annotations translates to expression e' with S -annotations
	$[(e : \tau)] = ([e] : [\tau])$
	$[e[\tau]] = [e][[\tau]]$
	$[e_1 @ e_2] = [e_1] @ [e_2]$
	etc.

Figure 7. Type translation into the economical language

The impartial type system directly generalizes a call-by-value system and a call-by-name system, but the profusion of connectives is unwieldy, and impartiality doesn't fit a standard operational semantics. Instead of elaborating the impartial system into our target language, we pause to develop an *economical* type system whose standard connectives (\rightarrow , $*$, $+$, μ) are by-value, but with a *suspension point* $\epsilon \blacktriangleright S$ to provide by-name behaviour. This intermediate system yields a straightforward elaboration. It also constitutes an alternative source language that, while biased towards call-by-value, conveniently allows call-by-name and evaluation-order polymorphism.

In the grammar in Figure 8, the economical types S are obtained from the impartial types τ by dropping all the ϵ decorations and adding a connective $\epsilon \blacktriangleright S$ (read “ ϵ suspend S ”). When ϵ is \mathbf{V} , this connective is a no-op: elaborating e at type $\mathbf{V} \blacktriangleright S$ and at type S yield the same term. But when ϵ is \mathbf{N} , elaborating e at type $\mathbf{N} \blacktriangleright S$ is like elaborating e at type $\mathbf{1} \rightarrow S$.

In economical typing contexts Γ , variables x denote values, so we replace the assumption form $x \varphi \Rightarrow \tau$ with $x : S$. Similarly, we replace $u \tau \Rightarrow \tau$ with $u : S$.

Dropping ϵ decorations means that—apart from the valuiness annotations—most of the economical rules in Figure 8 look fairly standard. The only new rules are for suspension points $\epsilon \blacktriangleright$, halfway down Figure 8. It would be nice to have only two rules (an introduction and an elimination), but we need to track whether e is a value,

which depends on the ϵ in $\epsilon \blacktriangleright S$: if we introduce the type $\mathbf{N} \blacktriangleright S$, then e will be elaborated to a thunk, which is a value; if we are eliminating $\mathbf{N} \blacktriangleright S$, the elaboration of e will have the form *force* \dots , which (like function application) is not a value.

3.1 Translating to economical types

To relate economical types to impartial types, we define a type translation $[\tau] = S$ that inserts suspension points (Figure 7). Given an impartially-typed source program e of type τ , we can show that $[e]$ has the economical type $[\tau]$ (Theorem 1).

Some parts of the translation are straightforward. Functions $\tau_1 \xrightarrow{\epsilon} \tau_2$ are translated to $(\epsilon \blacktriangleright [\tau_1]) \rightarrow [\tau_2]$ because when $\epsilon = \mathbf{N}$, we get the expected type $(\mathbf{N} \blacktriangleright [\tau_1]) \rightarrow [\tau_2]$ of a call-by-name function.

We are less constrained in how to translate other connectives:

- We could translate $\tau_1 +^\epsilon \tau_2$ to $(\epsilon \blacktriangleright [\tau_1]) + (\epsilon \blacktriangleright [\tau_2])$. But then $\mathbf{1} +^{\mathbf{N}} \mathbf{1}$ —presumably intended as a non-strict boolean type—would be translated to $(\mathbf{N} \blacktriangleright \mathbf{1}) + (\mathbf{N} \blacktriangleright \mathbf{1})$, which exposes which injection was used (whether the boolean is true or false) without forcing the (spurious) thunk around the unit value. Thus, we instead place the thunk around the entire sum, so that $\mathbf{1} +^{\mathbf{N}} \mathbf{1}$ translates to $\mathbf{N} \blacktriangleright (\mathbf{1} + \mathbf{1})$.
- We could translate $\tau_1 *^\epsilon \tau_2$ to $\epsilon \blacktriangleright ([\tau_1] * [\tau_2])$ —which corresponds to how we decided to translate sum types. Instead, we translate it to $(\epsilon \blacktriangleright [\tau_1]) * (\epsilon \blacktriangleright [\tau_2])$, so that, when $\epsilon = \mathbf{N}$, we get a pair of thunks; accessing one component of the pair (by forcing its thunk) won't cause the other component to be forced.
- Finally, in translating $\mu^\epsilon \alpha. \tau$, we could put a suspension on each occurrence of α in τ , rather than a single suspension on the outside of τ . Since τ is often a sum type, writing $+^\epsilon$ already puts a thunk on τ ; we don't need a thunk around a thunk. But by the same token, suspensions around the occurrences of α can also lead to double thunks: translating the type of lazy natural numbers $\mu^{\mathbf{N}} \alpha. (\mathbf{1} +^{\mathbf{N}} \alpha)$ would give $\mu \alpha. (\mathbf{N} \blacktriangleright (\mathbf{1} + \mathbf{N} \blacktriangleright \alpha))$, which expands to $\mathbf{N} \blacktriangleright (\mathbf{1} + \mathbf{N} \blacktriangleright \mathbf{N} \blacktriangleright (\mathbf{1} + \dots))$.

The rationales for our translation of products and recursive types are less clear than the rationale for sum types; it's possible that different encodings would be preferred in practice.

The above translation does allow programmers to use the alternative encodings, though awkwardly. For example, a two-thunk variant of $\tau_1 *^\epsilon \tau_2$ can be obtained by writing $(\mu^\epsilon \beta. \tau_1) *^{\mathbf{V}} (\mu^\epsilon \beta. \tau_2)$, where β doesn't occur; the only purpose of μ here is to insert a suspension. (This suggests a kind of ill-founded argument for our chosen translation of μ : it enables us to insert suspensions, albeit awkwardly.)

3.2 Programming with economical types

We can translate the list/stream example from Section 2.3 to the economical system:

$$\text{type List } a \ \alpha = \mu \beta. a \blacktriangleright (\mathbf{1} + (\alpha * \beta))$$

The body of *map* is the same; only the type annotation is different.

$$\begin{aligned} \text{map} &: \Delta a. \forall \alpha. (\alpha \rightarrow \beta) \rightarrow (\text{List } a \ \alpha) \rightarrow (\text{List } a \ \beta) \\ &= \lambda \alpha. \text{fix } \text{map}. \lambda f. \lambda xs. \\ &\quad \text{case}(xs, x_1.\text{inj}_1(), \\ &\quad \quad x_2.\text{inj}_2(f @ (\text{proj}_1 x_2), \text{map} @ f @ (\text{proj}_2 x_2))) \end{aligned}$$

The above type for *map* corresponds to the impartial type with $\xrightarrow{\mathbf{V}}$. At the end of Section 2.3, we gave a very generic type for *map*, which we can translate to the economical system:

$$\begin{aligned} &\Delta a_1, a_2, a_3, a_4, a_5. \\ &\forall \alpha. \left(a_2 \blacktriangleright ((a_1 \blacktriangleright \alpha) \rightarrow \beta) \right) \rightarrow (a_4 \blacktriangleright (\text{List } a_3 \ \alpha)) \rightarrow (\text{List } a_5 \ \beta) \end{aligned}$$

Economical types $S ::= \mathbf{1} \mid \alpha \mid \forall \alpha. S \mid \Delta a. S \mid \epsilon \triangleright S$ Econ. typing contexts $\Gamma ::= \cdot \mid \Gamma, x : S \mid \Gamma, u : S \mid \Gamma, a \text{ evalorder} \mid \Gamma, \alpha \text{ type}$
 $\mid S_1 \rightarrow S_2 \mid S_1 * S_2 \mid S_1 + S_2 \mid \mu \alpha. S$ Econ. source expressions $e ::= \dots \mid \Lambda \alpha. e \mid e[S] \mid (e : S)$

$\Gamma \vdash_E e \Leftarrow S$ Source expression e checks against economical type S
 $\Gamma \vdash_E e \Rightarrow S$ Source expression e synthesizes economical type S

$$\begin{array}{c}
\frac{(x : S) \in \Gamma}{\Gamma \vdash_E x \text{ val} \Rightarrow S} \mathbf{Evar} \quad \frac{(u : S) \in \Gamma}{\Gamma \vdash_E u \top \Rightarrow S} \mathbf{Efixvar} \quad \frac{\Gamma, u : S \vdash_E e \Leftarrow S}{\Gamma \vdash_E (\text{fix } u. e) \top \Leftarrow S} \mathbf{Efix} \quad \frac{\Gamma \vdash_E e \Rightarrow S}{\Gamma \vdash_E e \Leftarrow S} \mathbf{Esub} \quad \frac{\Gamma \vdash_E e \Leftarrow S}{\Gamma \vdash_E (e : S) \Rightarrow S} \mathbf{Eanno} \\
\forall \quad \frac{\Gamma, \alpha \text{ type} \vdash_E e \text{ val} \Leftarrow S}{\Gamma \vdash_E \Lambda \alpha. e \text{ val} \Leftarrow \forall \alpha. S} \mathbf{E}\forall\text{Intro} \quad \frac{\Gamma \vdash_E e \Rightarrow \forall \alpha. S \quad \Gamma \vdash S' \text{ type}}{\Gamma \vdash_E e[S'] \Rightarrow [S'/\alpha]S} \mathbf{E}\forall\text{Elim} \quad \mathbf{1} \quad \frac{}{\Gamma \vdash_E () \text{ val} \Leftarrow \mathbf{1}} \mathbf{E1Intro} \\
\Delta \quad \frac{\Gamma, a \text{ evalorder} \vdash_E e \text{ val} \Leftarrow S}{\Gamma \vdash_E e \text{ val} \Leftarrow \Delta a. S} \mathbf{E}\Delta\text{Intro} \quad \frac{\Gamma \vdash_E e \Rightarrow \Delta a. S \quad \Gamma \vdash e \text{ evalorder}}{\Gamma \vdash_E e \Rightarrow [e/a]S} \mathbf{E}\Delta\text{Elim} \\
\epsilon \triangleright \quad \frac{\Gamma \vdash_E e \Leftarrow S}{\Gamma \vdash_E e \Leftarrow \epsilon \triangleright S} \mathbf{E}\epsilon\text{Intro} \quad \frac{\Gamma \vdash_E e \Rightarrow \forall \triangleright S}{\Gamma \vdash_E e \Rightarrow S} \mathbf{E}\epsilon\text{Elim}_\forall \quad \frac{\Gamma \vdash_E e \Rightarrow \epsilon \triangleright S}{\Gamma \vdash_E e \top \Rightarrow S} \mathbf{E}\epsilon\text{Elim}_e \\
\rightarrow \quad \frac{\Gamma, x : S_1 \vdash_E e \Leftarrow S_2}{\Gamma \vdash_E (\lambda x. e) \text{ val} \Leftarrow (S_1 \rightarrow S_2)} \mathbf{E}\rightarrow\text{Intro} \quad \frac{\Gamma \vdash_E e_1 \Rightarrow (S_1 \rightarrow S_2) \quad \Gamma \vdash_E e_2 \Leftarrow S_1}{\Gamma \vdash_E (e_1 @ e_2) \top \Rightarrow S_2} \mathbf{E}\rightarrow\text{Elim} \\
* \quad \frac{\Gamma \vdash_E e_1 \Leftarrow S_1 \quad \Gamma \vdash_E e_2 \Leftarrow S_2}{\Gamma \vdash_E (e_1, e_2) \Leftarrow (S_1 * S_2)} \mathbf{E}*Intro \quad \frac{\Gamma \vdash_E e \Rightarrow (S_1 * S_2)}{\Gamma \vdash_E (\text{proj}_k e) \top \Rightarrow S_k} \mathbf{E}*Elim_k \\
+ \quad \frac{\Gamma \vdash_E e \Leftarrow S_k}{\Gamma \vdash_E (\text{inj}_k e) \Leftarrow (S_1 + S_2)} \mathbf{E}+Intro_k \quad \frac{\Gamma \vdash_E e \Rightarrow (S_1 + S_2) \quad \Gamma, x_1 : S_1 \vdash_E e_1 \Leftarrow S \quad \Gamma, x_2 : S_2 \vdash_E e_2 \Leftarrow S}{\Gamma \vdash_E \text{case}(e, x_1.e_1, x_2.e_2) \top \Leftarrow S} \mathbf{E}+Elim \\
\mu \quad \frac{\Gamma \vdash_E e \Leftarrow [(\mu \alpha. S)/\alpha]S}{\Gamma \vdash_E e \Leftarrow \mu \alpha. S} \mathbf{E}\mu\text{Intro} \quad \frac{\Gamma \vdash_E e \Rightarrow \mu \alpha. S}{\Gamma \vdash_E e \top \Rightarrow [(\mu \alpha. S)/\alpha]S} \mathbf{E}\mu\text{Elim}
\end{array}$$

Figure 8. Economical bidirectional typing

This type might not look economical, but makes redundant suspensions more evident: List $a_3 \alpha$ is $\mu \dots a_3 \triangleright \dots$, so the suspension controlled by a_4 is never useful, showing that a_4 is unnecessary.

3.3 Economizing

The main result of this section is that impartial typing derivations can be transformed into economical typing derivations. The proof (Dunfield 2015, Appendix B.3) relies on a lemma that converts typing assumptions with $\forall \triangleright S'$ to assumptions with S' .

Theorem 1 (Economizing).

- (1) If $\gamma \vdash_I e \Rightarrow \tau$ then $[\gamma] \vdash_E [e] \Rightarrow [\tau]$.
- (2) If $\gamma \vdash_I e \Leftarrow \tau$ then $[\gamma] \vdash_E [e] \Leftarrow [\tau]$.

4. Target Language

Our target language (Figure 9) has by-value \rightarrow , $*$, $+$ and μ connectives, \forall , and a **U** connective (for thunks).

The \forall connective has explicit introduction and elimination forms $\Lambda _ . M$ and $M[_]$. This “type-free” style is a compromise between having no explicit forms for \forall and having explicit forms that contain types ($\Lambda \alpha. M$ and $A[M]$). Having no explicit forms would complicate some proofs; including the types would mean that target terms contain types, giving a misleading impression that operational behaviour is influenced by types.

The target language also has an explicit introduction form $\text{roll } M$ and elimination form $\text{unroll } M$ for μ types.

As with \forall , we distinguish thunks to simplify some proofs: Source expressions typed with the $\mathbf{N}\triangleright$ connective are elaborated to $\text{thunk } M$, rather than to a λ with an unused bound variable.

Target terms	$M ::= () \mid x \mid \lambda x. M \mid M_1 M_2$ $\mid u \mid \text{fix } u. M \mid \Lambda _ . M \mid M[_]$ $\mid \text{thunk } M \mid \text{force } M$ $\mid (M_1, M_2) \mid \text{proj}_k M$ $\mid \text{inj}_k M \mid \text{case}(M, x_1.M_1, x_2.M_2)$ $\mid \text{roll } M \mid \text{unroll } M$
Values	$W ::= () \mid x \mid \lambda x. M \mid \Lambda _ . M$ $\mid \text{thunk } M \mid (W_1, W_2)$ $\mid \text{inj}_k W \mid \text{roll } W$
Valuables	$\tilde{V} ::= () \mid x \mid \lambda x. M \mid \Lambda _ . \tilde{V} \mid \tilde{V}[_]$ $\mid \text{thunk } M \mid (\tilde{V}_1, \tilde{V}_2)$ $\mid \text{proj}_k \tilde{V} \mid \text{inj}_k \tilde{V} \mid \text{roll } \tilde{V} \mid \text{unroll } \tilde{V}$
Eval. contexts	$\mathcal{C} ::= [] \mid \mathcal{C} @ M_2 \mid W_1 @ \mathcal{C} \mid \mathcal{C}[_]$ $\mid \text{force } \mathcal{C}$ $\mid (\mathcal{C}, M_2) \mid (W_1, \mathcal{C}) \mid \text{proj}_k \mathcal{C}$ $\mid \text{inj}_k \mathcal{C} \mid \text{case}(\mathcal{C}, x_1.M_1, x_2.M_2)$ $\mid \text{roll } \mathcal{C} \mid \text{unroll } \mathcal{C}$
Target types	$A, B ::= \mathbf{1} \mid \alpha \mid \forall \alpha. A \mid A_1 \rightarrow A_2 \mid \mathbf{U} A_1$ $\mid A_1 * A_2 \mid A_1 + A_2 \mid \mu \alpha. A$
Typing contexts	$G ::= \cdot \mid G, x : A \mid G, \alpha \text{ type}$

Figure 9. Syntax of the target language

Dually, eliminating $\mathbf{N}\triangleright$ results in a target term $\text{force } M$, rather than to $M()$.

4.1 Typing rules

Figure 10 shows the typing rules for our target language. These are standard except for the $\mathbf{T}\forall\text{Intro}$ rule and the rules for thunks:

$\boxed{G \vdash_T M : A}$	Target term M has target type A	$\frac{}{G \vdash_T () : \mathbf{1}}$	T1Intro
$\frac{(x : A) \in G}{G \vdash_T x : A}$	Tvar	$\frac{(u : A) \in G}{G \vdash_T u : A}$	Tfixvar
$\frac{G, u : A \vdash_T e : A}{G \vdash_T (\text{fix } u. e) : A}$			Tfix
$\forall \frac{G, \alpha \text{ type} \vdash_T \tilde{V} : A}{G \vdash_T \bigwedge _. \tilde{V} : \forall \alpha. A}$	TVIntro	$\frac{G \vdash_T M : \forall \alpha. A}{G \vdash_T M[_] : [A'/\alpha]A}$	TVElim
$\rightarrow \frac{G, x : A \vdash_T M : B}{G \vdash_T (\lambda x. M) : A \rightarrow B}$	TIntro	$\frac{G \vdash_T M_1 : A \rightarrow B}{G \vdash_T (M_1 M_2) : B}$	TElim
$\mathbf{U} \frac{G \vdash_T M : B}{G \vdash_T \text{thunk } M : \mathbf{U} B}$	TUIntro	$\frac{G \vdash_T M_1 : \mathbf{U} B}{G \vdash_T \text{force } M_1 : B}$	TUElim
$* \frac{G \vdash_T M_1 : A_1}{G \vdash_T (M_1, M_2) : A_1 * A_2}$	T*Intro	$\frac{G \vdash_T M : A_1 * A_2}{G \vdash_T \text{proj}_k M : A_k}$	T*Elim_k
$\dagger \frac{G \vdash_T M : A_k}{G \vdash_T \text{inj}_k M : A_1 + A_2}$	T+Intro_k	$\frac{G \vdash_T M : A_1 + A_2}{G, x_1 : A_1 \vdash_T M_1 : A}$	T+Elim
$\mu \frac{}{G \vdash_T \text{case}(M, x_1.M_1, x_2.M_2) : A}$		$G, x_2 : A_2 \vdash_T M_2 : A$	
$\frac{G \vdash_T M : [\mu\alpha. A/\alpha]A}{G \vdash_T \text{roll } M : \mu\alpha. A}$	TμIntro	$\frac{G \vdash_T M : \mu\alpha. A}{G \vdash_T \text{unroll } M : [\mu\alpha. A/\alpha]A}$	TμElim

Figure 10. Target language type system

Valuability restriction. Though we omit mutable references from the target language, we want the type system to accommodate them. Using the standard syntactic value restriction (Wright 1995) would spoil this language as a target for our elaboration: when source typing uses *elab*TVIntro, it requires that the source expression be a value (not syntactically, but according to the source typing derivation). Yet if that source value is typed using *elab*TElim, it will elaborate to a projection, which is not a syntactic value. So we use a valuability restriction in TVIntro. A target term is a *valuable* \tilde{V} if it is a value (e.g. $\lambda x. M$) or is a projection, injection, roll or unroll of something that is valuable (Figure 9). Later, we’ll prove that if a source expression is a value (according to the source typing derivation), its elaboration is valuable (Lemma 6).

Thunks. We give *thunk* M the type $\mathbf{U} B$ for “thUnk B ” (if M has type B); *force* M eliminates this connective.

4.2 Operational semantics

The target operational semantics has two relations: $M \mapsto_R M'$, read “ M reduces to M' ”, and $M \mapsto M'$, read “ M steps to M' ”. The latter has only one rule, StepContext, which says that $\mathcal{C}[M] \mapsto \mathcal{C}[M']$ if $M \mapsto_R M'$, where \mathcal{C} is an evaluation context (Figure 9). The rules for \mapsto_R (Figure 11) reduce a λ applied to a value; a force of a thunk; a fixed point; a type application; a projection of a pair of values; a case over an injected value; and an unroll of a rolled value. Apart from force (*thunk* M), which we can view as strange syntax for $(\lambda x. M)()$, this is all standard: these definitions use values W , not valuables \tilde{V} .

4.3 Type safety

Lemma 2 (Valuability). If $\tilde{V} \mapsto M'$ or $\tilde{V} \mapsto_R M'$ then M' is valuable, that is, there exists $\tilde{V}' = M'$.

Lemma 3 (Substitution). If $G, x : A', G' \vdash_T M : A$ and $G \vdash_T W : A'$ then $G, G' \vdash_T [W/x]M : A$.

$\boxed{M \mapsto M'}$	Target term M steps (by-value) to target term M'	$\frac{M \mapsto_R M'}{\mathcal{C}[M] \mapsto \mathcal{C}[M']}$	StepContext
$\boxed{M \mapsto_R M'}$	Target redex M reduces (by-value) to M'		
$(\lambda x. M) @ W \mapsto_R [W/x]M$		βReduce	
$\text{force}(\text{thunk } M) \mapsto_R M$		forceReduce	
$(\text{fix } u. M) \mapsto_R [(\text{fix } u. M)/u]M$		fixReduce	
$(\bigwedge _. M)[_] \mapsto_R M$		tyappReduce	
$\text{proj}_k((W_1, W_2)) \mapsto_R W_k$		projReduce	
$\text{case}(\text{inj}_k W, x_1.M_1, x_2.M_2) \mapsto_R [W/x_k]M_k$		caseReduce	
$\text{unroll}(\text{roll } W) \mapsto_R W$		unrollReduce	

Figure 11. Target language operational semantics

$\boxed{ S = A}$	Economical type S elaborates to target type A	$ \mathbf{1} = \mathbf{1}$	$ \mathbf{V}\triangleright S = S $
$ S_1 \rightarrow S_2 = S_1 \rightarrow S_2 $		$ S_1 + S_2 = S_1 + S_2 $	$ \mathbf{N}\triangleright S = \mathbf{U} S $
$ \alpha = \alpha$		$ \forall \alpha. S = \forall \alpha. S $	$ \Delta a. S = [V/a]S * [N/a]S $
$ \mu\alpha. S = \mu\alpha. S $			
$\boxed{ \Gamma = G}$	Economical typing context Γ elaborates to target typing context G	$ \cdot = \cdot$	$ \Gamma, x : S = \Gamma , x : S $
		$ \Gamma, \alpha \text{ type} = \Gamma , \alpha \text{ type}$	$ \Gamma, u : S = \Gamma , u : S $
		$ \Gamma, a \text{ evalorder} $	undefined

Figure 12. Translation from economical types to target types

Theorem 4 (Type safety). If $\vdash_T M : A$ then either M is a value, or $M \mapsto M'$ and $G \vdash_T M' : A$.

Proof. By induction on the derivation of $G \vdash_T M : A$, using Lemma 3 and standard inversion lemmas, which we omit. \square

5. Elaboration

Now we extend the economical typing judgment with an output M , a *target term*: $\Gamma \vdash e_\varphi : S \hookrightarrow M$. The target term M should be well-typed using the typing rules in Figure 10, but what type should it have? We answer this question by defining another translation on types. This function, defined by a function $|S| = A$, translates an economical source type S to a target type A .

We will show that if $e_\varphi : S \hookrightarrow M$ then $M : A$, where $A = |S|$; this is Theorem 10. Our translation follows a similar approach to Dunfield (2014). However, that system had general intersection types $A_1 \wedge A_2$, where A_1 and A_2 don’t necessarily have the same structure. In contrast, we have $\Delta a. A$ which corresponds to $([V/a]A) \wedge ([N/a]A)$. We also differ in having recursive types; since these are explicitly rolled (or *folded*) and unrolled in our target language, our rules *elab*μIntro and *elab*μElim add these constructs.

Not bidirectional. We want to relate the operational behaviour of a source expression to the operational behaviour of its elaboration. Since our source operational semantics is over type-erased source expressions, it will be convenient for elaboration to work on erased source expressions. Without type annotations, we can collapse the bidirectional judgments into a single judgment (with “ \cdot ” in place of \Leftarrow/\Rightarrow); this obviates the need for elaboration versions of **Esub** and **Eanno**, which merely switch between \Leftarrow and \Rightarrow .

$$\boxed{\Gamma \vdash e_\varphi : S \hookrightarrow M} \text{ Erased source expression } e \text{ elaborates at type } S \text{ to target term } M$$

$$\begin{array}{c}
\frac{(\lambda : S) \in \Gamma}{\Gamma \vdash x_{\text{val}} : S \hookrightarrow x} \text{elabvar} \quad \frac{(u : S) \in \Gamma}{\Gamma \vdash u_\tau : S \hookrightarrow u} \text{elabfixvar} \quad \frac{\Gamma, u : S \vdash e_\varphi : S \hookrightarrow M}{\Gamma \vdash (\text{fix } u. e)_\tau : S \hookrightarrow (\text{fix } u. M)} \text{elabfix} \quad \frac{}{\Gamma \vdash ()_{\text{val}} : \mathbf{I} \hookrightarrow ()} \text{elab1Intro} \\
\bigvee \frac{\Gamma, \alpha \text{ type} \vdash e_{\text{val}} : S \hookrightarrow M}{\Gamma \vdash e_{\text{val}} : \forall \alpha. S \hookrightarrow \Lambda _ . M} \text{elab}\forall\text{Intro} \quad \frac{\Gamma \vdash e_\varphi : \forall \alpha. S \hookrightarrow M \quad \Gamma \vdash S' \text{ type}}{\Gamma \vdash e_\varphi : [S'/\alpha]S \hookrightarrow M[_]} \text{elab}\forall\text{Elim} \\
\bigtriangleleft \frac{\Gamma \vdash e_{\text{val}} : [V/a]S \hookrightarrow M_1 \quad \Gamma \vdash e_{\text{val}} : [N/a]S \hookrightarrow M_2}{\Gamma \vdash e_{\text{val}} : (\Delta a. S) \hookrightarrow (M_1, M_2)} \text{elab}\Delta\text{Intro} \quad \frac{\Gamma \vdash e_\varphi : (\Delta a. S) \hookrightarrow M}{\Gamma \vdash e_\varphi : [V/a]S \hookrightarrow (\text{proj}_1 M) \quad \Gamma \vdash e_\varphi : [N/a]S \hookrightarrow (\text{proj}_2 M)} \text{elab}\Delta\text{Elim} \\
\blacktriangleright \frac{\Gamma \vdash e_\varphi : S \hookrightarrow M}{\Gamma \vdash e_\varphi : V \blacktriangleright S \hookrightarrow M} \text{elab}\blacktriangleright\text{Intro} \quad \frac{\Gamma \vdash e_\varphi : V \blacktriangleright S \hookrightarrow M}{\Gamma \vdash e_\varphi : S \hookrightarrow M} \text{elab}\blacktriangleright\text{Elim}_V \quad \frac{\Gamma \vdash e_\varphi : N \blacktriangleright S \hookrightarrow M}{\Gamma \vdash e_\tau : S \hookrightarrow (\text{force } M)} \text{elab}\blacktriangleright\text{Elim}_N \\
\rightarrow \frac{\Gamma, x : S_1 \vdash e_\varphi : S_2 \hookrightarrow M}{\Gamma \vdash (\lambda x. e)_{\text{val}} : (S_1 \rightarrow S_2) \hookrightarrow \lambda x. M} \text{elab}\rightarrow\text{Intro} \quad \frac{\Gamma \vdash e_{1 \varphi_1} : (S_1 \rightarrow S_2) \hookrightarrow M_1 \quad \Gamma \vdash e_{2 \varphi_2} : S_1 \hookrightarrow M_2}{\Gamma \vdash (e_1 @ e_2)_\tau : S_2 \hookrightarrow (M_1 @ M_2)} \text{elab}\rightarrow\text{Elim} \\
* \frac{\Gamma \vdash e_{1 \varphi_1} : S_1 \hookrightarrow M_1 \quad \Gamma \vdash e_{2 \varphi_2} : S_2 \hookrightarrow M_2}{\Gamma \vdash (e_1, e_2)_{\varphi_1 \sqcup \varphi_2} : (S_1 * S_2) \hookrightarrow (M_1, M_2)} \text{elab}* \text{Intro} \quad \frac{\Gamma \vdash e_\varphi : (S_1 * S_2) \hookrightarrow M}{\Gamma \vdash (\text{proj}_k e)_\tau : S_k \hookrightarrow (\text{proj}_k M)} \text{elab}* \text{Elim}_k \\
+ \frac{\Gamma \vdash e_\varphi : S_k \hookrightarrow M}{\Gamma \vdash (\text{inj}_k e)_\varphi : (S_1 + S_2) \hookrightarrow (\text{inj}_k M)} \text{elab}+ \text{Intro}_k \quad \frac{\Gamma \vdash e_{\varphi_0} : (S_1 + S_2) \hookrightarrow M_0 \quad \Gamma, x_1 : S_1 \vdash e_{1 \varphi_1} : S \hookrightarrow M_1 \quad \Gamma, x_2 : S_2 \vdash e_{2 \varphi_2} : S \hookrightarrow M_2}{\Gamma \vdash \text{case}(e, x_1.e_1, x_2.e_2)_\tau : S \hookrightarrow \text{case}(M_0, x_1.M_1, x_2.M_2)} \text{elab}+ \text{Elim} \\
\mu \frac{\Gamma \vdash e_\varphi : [(\mu \alpha. S)/\alpha]S \hookrightarrow M}{\Gamma \vdash e_\varphi : \mu \alpha. S \hookrightarrow (\text{roll } M)} \text{elab}\mu\text{Intro} \quad \frac{\Gamma \vdash e_\varphi : \mu \alpha. S \hookrightarrow M}{\Gamma \vdash e_\tau : [(\mu \alpha. S)/\alpha]S \hookrightarrow (\text{unroll } M)} \text{elab}\mu\text{Elim}
\end{array}$$

Figure 13. Elaboration

Elaboration rules. We are elaborating the economical type system, which has by-value connectives, into the target type system, which also has by-value connectives. Most of the elaboration rules just map source constructs into the corresponding target constructs; for example, *elabvar* elaborates x to x , and *elab \rightarrow Intro* elaborates $\lambda x. e$ to $\lambda x. M$ where e elaborates to M .

Elaborating \forall . Rule *elab \forall Intro* elaborates e (which is type-erased and thus has no explicit source construct) to the target type abstraction $\Lambda _ . M$; rule *elab \forall Elim* elaborates to a target type application $M[_]$.

Elaborating Δ . Rule *elab Δ Intro* elaborates an e at type $\Delta a. S$ to a pair with the elaborations of e at type $[V/a]S$ and at $[N/a]S$. Note that unlike the corresponding rule *E Δ Intro* in the non-elaborating economical type system, which introduces a variable a into Γ and types e parametrically, *elab Δ Intro* substitutes concrete evaluation orders V and N for a . Consequently, the Γ in the elaboration judgment never contains a *evalorder* declarations.

Rule *elab Δ Elim* elaborates to the appropriate projection.

Elaborating \blacktriangleright . Rule *elab \blacktriangleright Intro* has two conclusions. The first conclusion elaborates at type $V \blacktriangleright S$ as if elaborating at type S . The second conclusion elaborates at $N \blacktriangleright S$ to a thunk. Correspondingly, rule *elab \blacktriangleright Elim $_V$* ignores the V suspension, and rule *elab \blacktriangleright Elim $_N$* forces the thunk introduced via *elab \blacktriangleright Intro*.

5.1 Elaboration type soundness

The main result of this section (Theorem 10) is that, given a non-elaborating economical typing derivation $\Gamma \vdash_{\mathbf{E}} e_\varphi \Leftarrow S$, we can derive $\Gamma \vdash \text{er}(e)_\varphi : S \hookrightarrow M$ such that the target term M is well-typed. The erasure function $\text{er}(e)$, defined in Figure 6, removes type annotations, type abstractions, and type applications.

It will be useful to relate various notions of valuiness. First, if e elaborates to a syntactic target value W , then the elaboration rules deem e to be a (source) value.

Lemma 5. If $\Gamma \vdash e_\varphi : S \hookrightarrow W$ then $\varphi = \text{val}$.

Second, if e is a value according to the source typing rules, its elaboration M is valuable (but not necessarily a syntactic target value).

Lemma 6 (Elaboration valuability).

If $\Gamma \vdash e_{\text{val}} : S \hookrightarrow M$ then M is valuable, that is, there exists \tilde{V} such that $M = \tilde{V}$.

Several substitution lemmas are required. The first is for the non-elaborating economical type system; we'll use it in the *E Δ Intro* case of the main proof to remove a *evalorder* declarations.

Lemma 7 (Substitution—Evaluation orders).

- (1) If $\Gamma, a \text{ evalorder}, \Gamma' \vdash S \text{ type}$ and $\Gamma \vdash e \text{ evalorder}$ then $\Gamma, [e/a]\Gamma' \vdash [e/a]S \text{ type}$.
- (2) If \mathcal{D} derives $\Gamma, a \text{ evalorder}, \Gamma' \vdash_{\mathbf{E}} e_\varphi \Leftarrow S$ and $\Gamma \vdash e \text{ evalorder}$ then \mathcal{D}' derives $\Gamma, [e/a]\Gamma' \vdash_{\mathbf{E}} e_\varphi \Leftarrow [e/a]S$ where \mathcal{D}' is not larger than \mathcal{D} .
- (3) If \mathcal{D} derives $\Gamma, a \text{ evalorder}, \Gamma' \vdash_{\mathbf{E}} e_\varphi \Rightarrow S$ and $\Gamma \vdash e \text{ evalorder}$, then \mathcal{D}' derives $\Gamma, [e/a]\Gamma' \vdash_{\mathbf{E}} e_\varphi \Rightarrow [e/a]S$ where \mathcal{D}' is not larger than \mathcal{D} .

Next, we show that an expression e_1 can be substituted for a variable x , provided e_1 elaborates to a target value W .

Lemma 8 (Expression substitution).

- (1) If $\Gamma \vdash e_{1 \varphi_1} : S_1 \hookrightarrow W$ and $\Gamma, x : S_1, \Gamma' \vdash e_{2 \varphi_2} : S \hookrightarrow M$ then $\Gamma, \Gamma' \vdash [e_1/x]e_{2 \varphi_2} : S \hookrightarrow [W/x]M$.
- (2) If $\Gamma \vdash \text{fix } u. e_1_\tau : S_1 \hookrightarrow \text{fix } u. M_1$ and $\Gamma, u : S_1, \Gamma' \vdash e_{2 \varphi_2} : S \hookrightarrow M$ then $\Gamma, \Gamma' \vdash [(\text{fix } u. e_1)/u]e_{2 \varphi_2} : S \hookrightarrow [(\text{fix } u. M_1)/u]M$.

Lemma 9 (Type translation well-formedness).

If $\Gamma \vdash S$ type then $|\Gamma| \vdash |S|$ type.

We can now state the main result of this section:

Theorem 10 (Elaboration type soundness).

If $\Gamma \vdash_{\mathbf{E}} e \varphi \Leftarrow S$ or $\Gamma \vdash_{\mathbf{E}} e \varphi \Rightarrow S$

where $\Gamma \vdash S$ type and Γ contains no *evalorder* declarations

then there exists M such that $\Gamma \vdash_{\mathbf{E}} \text{er}(e) \varphi' : S \hookrightarrow M$

where $\varphi' \sqsubseteq \varphi$ and $|\Gamma| \vdash_{\mathbf{T}} M : |S|$.

The proof is in Dunfield (2015, Appendix B.5). In this theorem, the resulting elaboration judgment has a valueness φ' that can be more precise than the valueness φ in the non-elaborating judgment. Suppose that, inside a derivation of a *evalorder* $\vdash_{\mathbf{E}} e \text{val} \Leftarrow S$, we have

$$\frac{\text{a evalorder } \vdash_{\mathbf{E}} e' \text{val} \Leftarrow \text{a} \triangleright S'}{\text{a evalorder } \vdash_{\mathbf{E}} e' \tau \Leftarrow S'} \mathbf{E} \triangleright \text{Elim}_e$$

The valueness in the conclusion must be \top , because we might substitute N for a , which is elaborated to a *force*, which is not a value. Now suppose we substitute V for a . We need to construct an elaboration derivation, and the only rule that works is *elab* \triangleright *Elim* $_V$:

$$\frac{\cdot \vdash e' \text{val} : \mathbf{V} \triangleright S' \hookrightarrow M}{\cdot \vdash e' \text{val} : S' \hookrightarrow M} \mathbf{elab} \triangleright \text{Elim}_V$$

This says e' is a value (*val*), where the original (parametric) economical typing judgment had \top : Substituting a concrete object (here, V) for a variable a increases information, refining \top (“I cannot prove this is a value”) into *val*. In the introduction rules, substituting N for a can replace \top with *val*, because we know we’re elaborating to a *think*, which is a value.

6. Consistency

Our main result in this section, Theorem 15, says that if e elaborates to a target term M , and M steps (zero or more times) to a target value W , then e steps (zero or more times) to some e' that elaborates to W . The source language stepping relation (Figure 5) allows both by-value and (more permissive) by-name reductions, raising the concern that a call-by-value program might elaborate to a call-by-name target program, that is, one taking steps that correspond to by-name reductions in the source program. So we strengthen the statement, showing that if M is completely free of by-name constructs, then all the steps taken in the source program are by-value.

That still leaves the possibility that we messed up our elaboration rules, such that a call-by-value source program elaborates to an M that contains by-name constructs. So we prove (Theorem 18) that if the source program is completely free of by-name constructs, its elaboration M is also free of by-name constructs. Similarly, we prove (Theorem 17) that creating an economical typing derivation from an impartial typing derivation preserves N-freeness.

Proofs can be found in Dunfield (2015, Appendix B.6).

6.1 Source-side consistency?

A source expression typed by name won’t get stuck if a by-value reduction is chosen, but it may diverge instead of terminating.

Suppose we have typed $(\lambda x. x)$ against $\tau \xrightarrow{N} \tau$. Taking only a by-name reduction, we have

$$(\lambda x. ())(\text{fix } u. u) \rightsquigarrow [(\text{fix } u. u)/x] () = () \text{ using } \beta N \text{reduce}$$

However, if we “contradict” the typing derivation by taking by-value reductions, we diverge:

$$\begin{aligned} (\lambda x. ())(\text{fix } u. u) &\rightsquigarrow (\lambda x. ())([(\text{fix } u. u)/u]u) \text{ using } \text{fix } V \text{reduce} \\ &= (\lambda x. ())(\text{fix } u. u) \rightsquigarrow \dots \end{aligned}$$

We’re used to type safety being “up to” nontermination in the sense that we either get a value or diverge, without getting stuck, but this is worse: divergence depends on which reductions are chosen.

To get a source type safety result that is both direct (without appealing to elaboration and target reductions) and useful, we’d need to give a semantics of “reduction with respect to a typing derivation”, or else reduction *of* a typing derivation. Such a semantics would support reasoning about local transformations of source programs. It should also lead to a converse of the consistency result in this section: if a source expression reduces with respect to a typing derivation, and that typing derivation corresponds to an elaboration derivation, then the target program obtained by elaboration can be correspondingly reduced.

6.2 Defining N-freeness

Definition 1 (N-freeness—impartial).

- (1) An impartial type τ is *N-free* iff (i) for each ϵ appearing in S , the evaluation order ϵ is V ; and (ii) τ has no Δ quantifiers.
- (2) A judgment $\gamma \vdash_{\mathbf{I}} e \varphi \Leftarrow \tau$ or $\gamma \vdash_{\mathbf{I}} e \varphi \Rightarrow \tau$ is *N-free* iff: (a) γ has no *evalorder* declarations; (b) in each declaration $x \varphi \Rightarrow \tau$ in γ , the valueness φ is *val* and the type τ is N-free; (c) all types appearing in e are N-free; and (d) τ is N-free.

Definition 2 (N-freeness—economical).

- (1) An economical type S is *N-free* iff (i) for each $\epsilon \triangleright S_0$ appearing in S , the evaluation order ϵ is V ; and (ii) S has no Δ quantifiers.
- (2) A judgment $\Gamma \vdash_{\mathbf{E}} e \varphi \Leftarrow S$ or $\Gamma \vdash_{\mathbf{E}} e \varphi \Rightarrow S$ is *N-free* iff: (a) Γ has no *evalorder* declarations; (b) all types S' in Γ are N-free; (c) all types appearing in e are N-free; and (d) S is N-free.

Definition 3 (N-freeness—target). A target term M is *N-free* iff it contains no *think* and *force* constructs.

6.3 Lemmas for consistency

An inversion lemma allows types of the form $\mathbf{V} \triangleright \dots \mathbf{V} \triangleright S$, a generalization needed for the *elab* \triangleright *Elim* $_V$ case; when we use the lemma in the consistency proof, the type is not headed by $\mathbf{V} \triangleright$:

Lemma 11 (Inversion). Given $\cdot \vdash e \varphi : \underbrace{\mathbf{V} \triangleright \dots \mathbf{V} \triangleright}_0 \text{ or more } S \hookrightarrow M$:

- (0) If $M = (\lambda x. M_0)$ and $S = (S_1 \rightarrow S_2)$ then $e = (\lambda x. e_0)$ and $\cdot, x : S_1 \vdash e_0 \varphi' : S_2 \hookrightarrow M_0$.
- (1) If $M = (W_1, W_2)$ and $S = (\Delta a. S_0)$ then $\cdot \vdash e \varphi : [V/a]S_0 \hookrightarrow W_1$ and $\cdot \vdash e \varphi : [N/a]S_0 \hookrightarrow W_2$.
- (2) If $M = \text{think } M_0$ and $S = \mathbf{N} \triangleright S_0$ then $\cdot \vdash e \varphi' : S_0 \hookrightarrow M_0$.

Parts (3)–(6), for \forall , $+$, μ and $*$, are stated in the appendix.

Previously, we showed that if a source expression elaborates to a target value, source typing says the expression is a value ($\varphi = \text{val}$); here, we show that if a source expression elaborates to a target value that is N-free (ruling out *think* M produced by the second conclusion of *elab* \triangleright *Intro*), then e is a *syntactic* value.

Lemma 12 (Syntactic values).

If $\Gamma \vdash e \text{val} : S \hookrightarrow W$ and W is N-free then e is a syntactic value.

The next lemma just says that the \mapsto relation doesn’t produce *thunks* and *forces* out of thin air.

Lemma 13 (Stepping preserves N-freeness). If M is N-free and $M \mapsto M'$ then M' is N-free.

The proof is by cases on the derivation of $M \mapsto M'$, using the fact that if M_0 and M_1 are N-free, then $[M_0/x]M_1$ is N-free.

6.4 Consistency results

Theorem 14 (Consistency).

If $\cdot \vdash e \varphi : S \hookrightarrow M$ and $M \mapsto M'$ then there exists e' such that $e \rightsquigarrow^* e'$ and $\cdot \vdash e' \varphi' : S \hookrightarrow M'$ and $\varphi' \sqsubseteq \varphi$.

Moreover: (1) If $\varphi = \text{val}$ then $e' = e$. (2) If M is N-free then $e \rightsquigarrow^* e'$ can be derived without using *SrcStepCtxN*.

Result (1), under “moreover”, amounts to saying that values don’t step. Result (2) stops us from lazily sneaking in uses of SrcStepCtxN instead of showing that, given N -free M , we can always find a by-value evaluation context for use in SrcStepCtxV .

Theorem 15 (Multi-step consistency).

If $\cdot \vdash e \varphi : S \hookrightarrow M$ and $M \mapsto^* W$ then there exists e' such that $e \rightsquigarrow^* e'$ and $\cdot \vdash e'_{\text{val}} : S \hookrightarrow W$. Moreover, if M is N -free then we can derive $e \rightsquigarrow^* e'$ without using SrcStepCtxN .

6.5 Preservation of N -freeness

Lemma 16. If $\Gamma \vdash_{\mathbf{E}} e \varphi \Rightarrow S$ and S is *not* N -free then it is not the case that both Γ and e are N -free.

Theorem 17 (Economizing preserves N -freeness).

If $\gamma \vdash_{\mathbf{I}} e \varphi \Leftarrow \tau$ (resp. \Rightarrow) where the judgment is N -free (Definition 1 (2)) then $[\Gamma] \vdash_{\mathbf{E}} [e] \varphi \Leftarrow [\tau]$ (resp. \Rightarrow) where this judgment is N -free (Definition 2 (2)).

Theorem 18 (Elaboration preserves N -freeness).

If $\Gamma \vdash_{\mathbf{E}} e \varphi \Leftarrow S$ (or \Rightarrow) where the judgment is N -free (Definition 2 (2)) then $\Gamma \vdash_{\text{er}}(e) \varphi : S \hookrightarrow M$ such that M is N -free.

7. Related Work

History of evaluation order. In the λ -calculus, normal-order (leftmost-outermost) reduction seems to have preceded anything resembling call-by-value, but Bernays (1936) suggested requiring that the term being substituted in a reduction be in normal form. In programming languages, Algol-60 originated call-by-name and also provided call-by-value (Naur et al. 1960, 4.7.3); while the decision to make the former the default is debatable, direct support for two evaluation orders made Algol-60 an improvement on many of its successors. Plotkin (1975) related cbv and cbn to the λ -calculus, and developed translations between them.

Call-by-need or *lazy* evaluation was developed in the 1970s with the goal of doing as little computational work as possible, under which we can include the unbounded work of not terminating (Wadsworth 1971; Henderson and Morris 1976; Friedman and Wise 1976).

Laziness in call-by-value languages. Type-based support for selective lazy evaluation has been developed for cbv languages, including Standard ML (Wadler et al. 1998) and Java (Warth 2007). These approaches allow programmers to conveniently switch to another evaluation order, but don’t allow polymorphism over evaluation orders. Like our economical type system, these approaches are biased towards one evaluation order.

General coercions. General approaches to typed coercions were explored by Breazu-Tannen et al. (1991) and Barthe (1996). Swamy et al. (2009) developed a general typed coercion system for a simply-typed calculus, giving thunks as an example. In addition to annotations on all λ arguments, their system requires thunks (but not forces) to be written explicitly.

Intersection types. While this paper avoids the notation of intersection types, the quantifier Δ is essentially an intersection type of a very specific form. Theories of intersection types were originally developed by Coppo et al. (1981), among others; Hindley (1992) gives a useful introduction and survey. Intersections entered programming languages—as opposed to λ -calculus—when Reynolds (1996) put them at the heart of the Forsythe language. Subsequently—Reynolds’s paper describes ideas he developed in the 1980s—Freeman and Pfenning (1991) started a line of research on *refinement* intersections, where both parts of an intersection must refine the same base type (essentially, the same ML type).

The Δ intersection in this paper mixes features of general intersection and refinement intersection: the V and N instantiations

have close-to-identical structure, but cbv and cbn functions aren’t refinements of some “order-agnostic” base type. Our approach is descended mainly from the system of Dunfield (2014), which elaborates (general) intersection and union types into ordinary product and sum types. We differ in not having a source-level ‘merge’ construct $e_1 \,, e_2$, where the type system can select either e_1 or e_2 , ignoring the other component. Since e_1 and e_2 are not prevented from having the same type, the type system may elaborate either expression, resulting in unpredictable behaviour. In our type systems, we can think of $@$ in the source language as a merge ($@^V \,, @^N$), but the components have incompatible types. Moreover, the components must behave the same apart from evaluation order (evoking a standard property of systems of refinement intersection).

Alternative target languages. The impartial type system for our source language suggests that we should consider targeting an impartial, but more explicit, target language. In an untyped setting, Asperti (1990) developed a calculus with call-by-value and call-by-name λ -abstractions; function application is disambiguated at run time. In a typed setting, call-by-push-value (Levy 1999) systematically distinguishes values and computations; it has a thunk type \mathbf{U} (whence our notation) but also a dual, “lift” \mathbf{F} , which constructs a computation out of a value type. Early in the development of this paper, we tried to elaborate directly from the impartial type system to cbpv, without success. Levy’s elegant *pair* of translations from cbv and from cbn don’t seem to fit together easily; our feeling is that a combined translation would be either complicated, or prone to generating many redundant forces and thunks.

Zeilberger (2009) defined a polarized type system with positive and negative forms of each standard connective. In that system, \downarrow and \uparrow connectives alternate between polarities, akin to \mathbf{U} and \mathbf{F} in call-by-push-value. Zeilberger’s system has a symmetric function type, rather than the asymmetric function type found in cbpv. We guess that a translation into this system would have similar issues as with call-by-push-value.

8. Future Work

This paper develops type systems with multiple evaluation orders and polymorphism over evaluation orders, opening up the design space. More work is needed to realize these ideas in practice.

Implicit polymorphism. We made type polymorphism explicit, to prevent the type system from guessing evaluation orders. A practical system should find polymorphic instances without guessing, perhaps based on existential type variables (Dunfield and Krishnaswami 2013). We could also try to use some form of (lexically scoped?) default evaluation order. Such a default could also be useful for deciding whether some language features, such as let-expressions, should be by-value or by-name.

Exponential expansion. Our rules elaborate a function typed with $n \Delta$ quantifiers into 2^n instantiations. Only experience can demonstrate whether this is a problem in practice, but we have reasons to be optimistic.

First, we need the right point of comparison. The alternative to elaborating *map* into, say, 8 instantiations is to write 8 copies of *map* by hand. Viewed this way, elaboration maintains the size of the target program, while allowing an exponentially shorter source program! (This is the flipside of a sleight-of-hand from complexity theory, where you can make an algorithm look faster by inflating the input: Given an algorithm that takes 2^n time, where n is the number of bits in the input integer, we can get a purportedly polynomial algorithm by encoding the input in unary.)

Second, a compiler could analyze the source program and generate only the instances actually used, similar to monomorphization of \forall -polymorphism in MLton (mlton.org).

Other evaluation orders. Our particular choice of evaluation orders is not especially practical: the major competitor to call-by-value is call-by-need, not call-by-name. We chose call-by-name for simplicity (for example, in the source reduction rules), but many of our techniques should be directly applicable to call-by-need: elaboration would produce thunks in much the same way, just for a different dynamic semantics. Moreover, our approach could be extended to more than two evaluation orders, using an n -way intersection that elaborates to an n -tuple.

One could also take “order” very literally, and support left-to-right and right-to-left call-by-value. For low-level reasons, OCaml uses the former when compiling to native code, and the latter when compiling to bytecode. Being able to specify order of evaluation via type annotations could be useful when porting code from Standard ML (which uses left-to-right call-by-value).

Program design. We also haven’t addressed questions about when to use what evaluation order. Such questions seem to have been lightly studied, perhaps because of social factors: a programmer may choose a strict language because they tend to solve problems that don’t need laziness—which is self-reinforcing, because laziness is less convenient in a strict language. However, Chang (2014) developed tools, based on both static analysis and dynamic profiling, that suggest where laziness is likely to be helpful.

Existential quantification. By analogy to union types (Dunfield 2014), an existential quantifier would elaborate to a sum type. For example, the sum tag on a function of type $\exists a. \tau \xrightarrow{a} \tau$ would indicate, at run time, whether the function was by-value or by-name. This might resemble a typed version of the calculus of Asperti (1990).

Acknowledgments

The ICFP reviewers made suggestions and asked questions that have (I believe) improved the paper. The Max Planck Institute for Software Systems supported the early stages of this work. Dmitry Chistikov suggested the symbol Δ .

References

A. Asperti. Integrating strict and lazy evaluation: the λ_{sl} -calculus. In *Programming Language Implementation and Logic Programming*, volume 456 of *LNCS*, pages 238–254. Springer, 1990.

H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. Symbolic Logic*, 48(4):931–940, 1983.

G. Barthe. Implicit coercions in type systems. In *Proc. TYPES ’95*, volume 1158 of *LNCS*, pages 1–15, 1996.

P. Bernays. Review of “Some Properties of Conversion” by Alonzo Church and J.B. Rosser. *J. Symbolic Logic*, 1:74–75, 1936.

V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, 1991.

S. Chang. *On the Relationship Between Laziness and Strictness*. PhD thesis, Northeastern University, 2014.

Y. Chen, J. Dunfield, M. A. Hammer, and U. A. Acar. Implicit self-adjusting computation for purely functional programs. *J. Functional Programming*, 24(1):56–112, 2014.

M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift f. math. Logik und Grundlagen d. Math.*, 27:45–58, 1981.

R. Davies. *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University, 2005. CMU-CS-05-110.

R. Davies and F. Pfenning. Intersection types and computational effects. In *ICFP*, pages 198–208, 2000.

J. Dunfield. Elaborating intersection and union types. *J. Functional Programming*, 24(2–3):133–165, 2014.

J. Dunfield. Elaborating evaluation-order polymorphism, 2015. Extended version with appendices. arXiv:1504.07680 [cs.PL].

J. Dunfield and N. R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*, 2013. arXiv:1306.6032 [cs.PL].

J. Dunfield and F. Pfenning. Type assignment for intersections and unions in call-by-value languages. In *FoSSaCS*, pages 250–266, 2003.

J. Dunfield and F. Pfenning. Tridirectional typechecking. In *Principles of Programming Languages*, pages 281–292, 2004.

T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, pages 268–277, 1991.

D. P. Friedman and D. S. Wise. CONS should not evaluate its arguments. In *ICALP*, pages 257–284. Edinburgh Univ. Press, 1976.

A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *Logic in Computer Science*, 2002.

P. Henderson and J. H. Morris, Jr. A lazy evaluator. In *Principles of Programming Languages*, pages 95–103. ACM, 1976.

J. R. Hindley. Types with intersection: An introduction. *Formal Aspects of Computing*, 4:470–486, 1992.

D. Leivant. Typing and computational properties of lambda expressions. *Theoretical Computer Science*, 44(0):51–68, 1986.

P. B. Levy. Call-by-push-value: A subsuming paradigm. In *Typed Lambda Calculi and Applications*, pages 228–243. Springer, 1999.

R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

P. Naur et al. Report on the algorithmic language ALGOL 60. *Comm. ACM*, 3(5):299–314, 1960.

B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Prog. Lang. Systems*, 22:1–44, 2000.

G. Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

J. C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, 1996.

N. Swamy, M. Hicks, and G. M. Bierman. A theory of typed coercions and its applications. In *ICFP*, pages 329–340, 2009.

P. Wadler, W. Taha, and D. MacQueen. How to add laziness to a strict language without even being odd. In *Workshop on Standard ML*, 1998. <http://homepages.inf.ed.ac.uk/wadler/papers/lazyinstruct/lazyinstruct.ps>.

C. Wadsworth. *Semantics and Pragmatics of the lambda-Calculus*. PhD thesis, University of Oxford, 1971.

A. Warth. LazyJ: Seamless lazy evaluation in Java. In *FOOL*, 2007. foolwood07.cs.uchicago.edu/program/warth.pdf.

R. L. Wexelblat, editor. *History of Programming Languages I*. ACM, 1981.

A. K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

N. Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University, 2009. CMU-CS-09-122.

Supplemental material for “Elaborating Evaluation-Order Polymorphism”

This section of the extended version (Dunfield 2015) contains the (straightforward) rules for type well-formedness (Appendix A), proofs about economical typing that belong to Section 3 (Appendix B.3), proofs about elaboration typing that belong to Section 5 (Appendix B.5), and consistency proofs that belong to Section 6 (Appendix B.6).

A. Type Well-formedness

$\gamma \vdash \epsilon \text{ evalorder}$	Evaluation order ϵ is well-formed		
		$\frac{}{\gamma \vdash \mathbf{V} \text{ evalorder}}$	$\frac{(a \text{ evalorder}) \in \gamma}{\gamma \vdash a \text{ evalorder}}$
		$\frac{}{\gamma \vdash \mathbf{N} \text{ evalorder}}$	
$\gamma \vdash \tau \text{ type}$	Impartial type τ is well-formed		
		$\frac{}{\gamma \vdash \mathbf{1} \text{ type}}$	$\frac{(\alpha \text{ type}) \in \gamma}{\gamma \vdash \alpha \text{ type}}$
		$\frac{\gamma, \alpha \text{ type} \vdash \tau \text{ type}}{\gamma \vdash (\forall \alpha. \tau) \text{ type}}$	$\frac{\gamma, a \text{ evalorder} \vdash \tau \text{ type}}{\gamma \vdash (\Delta a. \tau) \text{ type}}$
		$\frac{\gamma \vdash \epsilon \text{ evalorder} \quad \gamma \vdash \tau_1 \text{ type} \quad \gamma \vdash \tau_2 \text{ type}}{\gamma \vdash (\tau_1 \xrightarrow{\epsilon} \tau_2) \text{ type}}$	
		$\frac{\gamma \vdash \epsilon \text{ evalorder} \quad \gamma \vdash (\tau_1 *^{\epsilon} \tau_2) \text{ type}}{\gamma \vdash (\tau_1 +^{\epsilon} \tau_2) \text{ type}}$	$\frac{\gamma \vdash \epsilon \text{ evalorder} \quad \gamma, \alpha \text{ type} \vdash \tau \text{ type}}{\gamma \vdash (\mu^{\epsilon} \alpha. \tau) \text{ type}}$

Figure 14. Type well-formedness in the impartial type system

$\Gamma \vdash \epsilon \text{ evalorder}$	Evaluation order ϵ is well-formed		
		$\frac{}{\Gamma \vdash \mathbf{V} \text{ evalorder}}$	$\frac{(a \text{ evalorder}) \in \Gamma}{\Gamma \vdash a \text{ evalorder}}$
		$\frac{}{\Gamma \vdash \mathbf{N} \text{ evalorder}}$	
$\Gamma \vdash S \text{ type}$	Economical type S is well-formed		
		$\frac{}{\Gamma \vdash \mathbf{1} \text{ type}}$	$\frac{(\alpha \text{ type}) \in \Gamma}{\Gamma \vdash \alpha \text{ type}}$
		$\frac{\Gamma, \alpha \text{ type} \vdash S \text{ type}}{\Gamma \vdash (\forall \alpha. S) \text{ type}}$	$\frac{\Gamma, a \text{ evalorder} \vdash S \text{ type}}{\Gamma \vdash (\Delta a. S) \text{ type}}$
		$\frac{\Gamma \vdash S_1 \text{ type} \quad \Gamma \vdash S_2 \text{ type}}{\Gamma \vdash (S_1 \rightarrow S_2) \text{ type}}$	
		$\frac{\Gamma \vdash \epsilon \text{ evalorder} \quad \Gamma \vdash S \text{ type}}{\Gamma \vdash (\epsilon \blacktriangleright S) \text{ type}}$	$\frac{\Gamma \vdash (S_1 * S_2) \text{ type}}{\Gamma \vdash (S_1 + S_2) \text{ type}}$
			$\frac{\Gamma, \alpha \text{ type} \vdash S \text{ type}}{\Gamma \vdash (\mu \alpha. S) \text{ type}}$

Figure 15. Type well-formedness in the economical type system

$G \vdash A \text{ type}$	Target type A is well-formed		
		$\frac{}{G \vdash \mathbf{1} \text{ type}}$	$\frac{(\alpha \text{ type}) \in G}{G \vdash \alpha \text{ type}}$
		$\frac{G \vdash A_1 \text{ type} \quad G \vdash A_2 \text{ type}}{G \vdash (A_1 \rightarrow A_2) \text{ type}}$	$\frac{G, \alpha \text{ type} \vdash A \text{ type}}{G \vdash (\forall \alpha. A) \text{ type}}$
		$\frac{G \vdash A \text{ type}}{G \vdash (\mathbf{U} A) \text{ type}}$	$\frac{G, \alpha \text{ type} \vdash A \text{ type}}{G \vdash (\mu \alpha. A) \text{ type}}$
		$\frac{G \vdash (A_1 * A_2) \text{ type}}{G \vdash (A_1 + A_2) \text{ type}}$	

Figure 16. Type well-formedness in the target type system

B. Proofs

Notation

We present some proofs in a line-by-line style, with the justification for each claim in the rightmost column. We highlight with \blacksquare what we needed to show; this is most useful when trying to prove statements with several conclusions, like “if... then Q1 and Q2 and Q3”, where we might derive Q2 early (say, directly from the induction hypothesis) but need several more steps to show Q1 and Q3.

- | | | |
|---|--|--------------------------------------|
| | $\gamma, \alpha \text{ type} \vdash_{\mathbf{I}} e_0 \text{ val} \Leftarrow \tau_0$ | Subderivation |
| | $[\gamma, \alpha \text{ type}] \vdash_{\mathbf{E}} [e_0] \text{ val} \Leftarrow [\tau_0]$ | By i.h. |
| | $[\gamma], \alpha \text{ type} \vdash_{\mathbf{E}} [e_0] \text{ val} \Leftarrow [\tau_0]$ | By def. of $[-]$ |
| | $[\gamma] \vdash_{\mathbf{E}} \Lambda \alpha. [e_0] \text{ val} \Leftarrow \forall \alpha. [\tau_0]$ | By E\forallIntro |
| ☞ | $[\gamma] \vdash_{\mathbf{E}} [\Lambda \alpha. e_0] \text{ val} \Leftarrow [\forall \alpha. \tau_0]$ | By def. of $[-]$ |
- **Case** $\frac{\gamma \vdash_{\mathbf{I}} e_0 \varphi \Rightarrow \forall \alpha. \tau_0 \quad \gamma \vdash \tau' \text{ type}}{\gamma \vdash_{\mathbf{I}} e_0 [\tau'] \varphi \Rightarrow [\tau'/\alpha] \tau_0} \text{IVElim}$
- | | | |
|---|--|---|
| | $\gamma \vdash_{\mathbf{I}} e_0 \varphi \Rightarrow \forall \alpha. \tau_0$ | Subderivation |
| | $[\gamma] \vdash_{\mathbf{E}} [e_0] \varphi \Rightarrow [\forall \alpha. \tau_0]$ | By i.h. |
| | $[\gamma] \vdash_{\mathbf{E}} [e_0] \varphi \Rightarrow \forall \alpha. [\tau_0]$ | By def. of $[-]$ |
| | $\gamma \vdash \tau' \text{ type}$ | Subderivation |
| | $[\gamma] \vdash [\tau'] \text{ type}$ | By Lemma 20 |
| | $[\gamma] \vdash_{\mathbf{E}} [e_0] [[\tau']] \varphi \Rightarrow [[\tau'/\alpha] \tau_0]$ | By E\forallElim |
| ☞ | $[\gamma] \vdash_{\mathbf{E}} [e_0 [\tau']] \varphi \Rightarrow [[\tau'/\alpha] \tau_0]$ | By properties of $[-]$ and substitution |
- **Case** $\frac{\gamma, a \text{ evalorder} \vdash_{\mathbf{I}} e \text{ val} \Leftarrow \tau_0}{\gamma \vdash_{\mathbf{I}} e \text{ val} \Leftarrow \Delta a. \tau_0} \text{IDIntro}$
- | | | |
|---|---|-------------------------------------|
| | $\gamma, a \text{ evalorder} \vdash_{\mathbf{I}} e \text{ val} \Leftarrow \tau_0$ | Subderivation |
| | $[\gamma, a \text{ evalorder}] \vdash_{\mathbf{E}} [e] \text{ val} \Leftarrow [\tau_0]$ | By i.h. |
| | $[\gamma], a \text{ evalorder} \vdash_{\mathbf{E}} [e] \text{ val} \Leftarrow [\tau_0]$ | By def. of $[-]$ |
| | $[\gamma] \vdash_{\mathbf{E}} [e] \text{ val} \Leftarrow \Delta a. [\tau_0]$ | By EΔIntro |
| ☞ | $[\gamma] \vdash_{\mathbf{E}} [e] \text{ val} \Leftarrow [\Delta a. \tau_0]$ | By def. of $[-]$ |
- **Case** $\frac{\gamma \vdash_{\mathbf{I}} e \varphi \Rightarrow \Delta a. \tau_0 \quad \gamma \vdash \epsilon \text{ evalorder}}{\gamma \vdash_{\mathbf{I}} e \varphi \Rightarrow [\epsilon/a] \tau_0} \text{IDElim}$
- | | | |
|---|--|---|
| | $\gamma \vdash_{\mathbf{I}} e \varphi \Rightarrow \Delta a. \tau_0$ | Subderivation |
| | $[\gamma] \vdash_{\mathbf{E}} [e] \varphi \Rightarrow [\Delta a. \tau_0]$ | By i.h. |
| | $\gamma \vdash \epsilon \text{ evalorder}$ | Subderivation |
| | $[\gamma] \vdash \epsilon \text{ evalorder}$ | By Lemma 21 |
| | $[\gamma] \vdash_{\mathbf{E}} [e] \varphi \Rightarrow [\epsilon/a] \tau_0$ | By EΔElim |
| ☞ | $[\gamma] \vdash_{\mathbf{E}} [e] \varphi \Rightarrow [[\epsilon/a] \tau_0]$ | By properties of $[-]$ and substitution |
- **Case** $\frac{(x \varphi \Rightarrow \tau) \in \gamma}{\gamma \vdash_{\mathbf{I}} x \varphi \Rightarrow \tau} \text{Ivar}$
- $(x \varphi \Rightarrow \tau) \in \gamma$ Premise
- We distinguish cases of φ :
- If $\varphi = \text{val}$, then:

	$(x : \mathbf{V}\blacktriangleright[\tau]) \in [\gamma]$	By def. of $[-]$
	$[\gamma] \vdash_{\mathbf{E}} x \text{ val} \Rightarrow \mathbf{V}\blacktriangleright[\tau]$	By Evar
☞	$[\gamma] \vdash_{\mathbf{E}} x \text{ val} \Rightarrow [\tau]$	By E\blacktrianglerightElim\mathbf{V}
 - If $\varphi = \top$, then:

	$(x : \mathbf{N}\blacktriangleright[\tau]) \in [\gamma]$	By def. of $[-]$
	$[\gamma] \vdash_{\mathbf{E}} x \text{ val} \Rightarrow \mathbf{N}\blacktriangleright[\tau]$	By Evar
☞	$[\gamma] \vdash_{\mathbf{E}} x \top \Rightarrow [\tau]$	By E\blacktrianglerightElimϵ
- **Case** $\frac{(u \top \Rightarrow \tau) \in \gamma}{\gamma \vdash_{\mathbf{I}} u \top \Rightarrow \tau} \text{Ifixvar}$
- | | | |
|---|--|-------------------|
| | $(u : [\tau]) \in [\gamma]$ | By def. of $[-]$ |
| ☞ | $[\gamma] \vdash_{\mathbf{E}} u \top \Rightarrow [\tau]$ | By Efixvar |

• Case
$$\frac{\gamma, u \top \Rightarrow \tau \vdash_{\mathbf{I}} e_0 \varphi \Leftarrow \tau}{\gamma \vdash_{\mathbf{I}} (\text{fix } u. e_0) \top \Leftarrow \tau} \text{Ifix}$$

$$\begin{array}{ll} [\gamma, u \top \Rightarrow \tau] \vdash_{\mathbf{E}} e_0 \varphi \Leftarrow [\tau] & \text{By i.h.} \\ [\gamma], u : [\tau] \vdash_{\mathbf{E}} e_0 \varphi \Leftarrow [\tau] & \text{By def. of } [-] \\ \vDash [\gamma] \vdash_{\mathbf{E}} (\text{fix } u. e_0) \top \Leftarrow [\tau] & \text{By Efix} \end{array}$$

• Case
$$\frac{\gamma \vdash_{\mathbf{I}} e \varphi \Rightarrow \tau}{\gamma \vdash_{\mathbf{I}} e \varphi \Leftarrow \tau} \text{Isub}$$

By i.h. and Esub.

• Case
$$\frac{\gamma \vdash_{\mathbf{I}} e_0 \varphi \Leftarrow \tau}{\gamma \vdash_{\mathbf{I}} (e_0 : \tau) \varphi \Rightarrow \tau} \text{Ianno}$$

By i.h. and Eanno.

• Case
$$\frac{\gamma \vdash_{\mathbf{I}} e_1 \varphi_1 \Leftarrow \tau_1 \quad \gamma \vdash_{\mathbf{I}} e_2 \varphi_2 \Leftarrow \tau_2}{\gamma \vdash_{\mathbf{I}} (e_1, e_2) \varphi_1 \sqcup \varphi_2 \Leftarrow (\tau_1 *^e \tau_2)} \text{I*Intro}$$

$$\begin{array}{ll} \gamma \vdash_{\mathbf{I}} e_1 \varphi_1 \Leftarrow \tau_1 & \text{Subderivation} \\ [\gamma] \vdash_{\mathbf{E}} [e_1] \varphi_1 \Leftarrow [\tau_1] & \text{By i.h.} \\ [\gamma] \vdash_{\mathbf{E}} [e_1] \varphi_1 \Leftarrow \epsilon \blacktriangleright [\tau_1] & \text{By E}\blacktriangleright\text{Intro} \\ \\ [\gamma] \vdash_{\mathbf{E}} [e_2] \varphi_2 \Leftarrow \epsilon \blacktriangleright [\tau_2] & \text{Similar} \\ \\ [\gamma] \vdash_{\mathbf{E}} ([e_1], [e_2]) \varphi_1 \sqcup \varphi_2 \Leftarrow (\epsilon \blacktriangleright [\tau_1]) * (\epsilon \blacktriangleright [\tau_2]) & \text{By E*Intro} \\ \vDash [\gamma] \vdash_{\mathbf{E}} [(e_1, e_2)] \varphi_1 \sqcup \varphi_2 \Leftarrow [\tau_1 *^e \tau_2] & \text{By def. of } [-] \end{array}$$

• Case
$$\frac{\gamma \vdash_{\mathbf{I}} e_0 \varphi \Rightarrow (\tau_1 *^e \tau_2)}{\gamma \vdash_{\mathbf{I}} (\text{proj}_k e_0) \top \Rightarrow \tau_k} \text{I*Elim}_k$$

$$\begin{array}{ll} \gamma \vdash_{\mathbf{I}} e_0 \varphi \Rightarrow (\tau_1 *^e \tau_2) & \text{Subderivation} \\ [\gamma] \vdash_{\mathbf{E}} [e_0] \varphi \Rightarrow [\tau_1 *^e \tau_2] & \text{By i.h.} \\ [\gamma] \vdash_{\mathbf{E}} [e_0] \varphi \Rightarrow (\epsilon \blacktriangleright [\tau_1]) * (\epsilon \blacktriangleright [\tau_2]) & \text{By def. of } [-] \\ [\gamma] \vdash_{\mathbf{E}} (\text{proj}_k [e_0]) \top \Rightarrow (\epsilon \blacktriangleright [\tau_k]) & \text{By E*Elim}_k \\ \vDash [\gamma] \vdash_{\mathbf{E}} [\text{proj}_k e_0] \top \Rightarrow [\tau_k] & \text{By E}\blacktriangleright\text{Elim}_e \text{ and def. of } [-] \end{array}$$

• Case
$$\frac{\gamma \vdash_{\mathbf{I}} e_0 \varphi \Leftarrow \tau_k}{\gamma \vdash_{\mathbf{I}} (\text{inj}_k e_0) \varphi \Leftarrow (\tau_1 +^e \tau_2)} \text{I+Intro}_k$$

$$\begin{array}{ll} \gamma \vdash_{\mathbf{I}} e_0 \varphi \Leftarrow \tau_k & \text{Subderivation} \\ [\gamma] \vdash_{\mathbf{E}} [e_0] \varphi \Leftarrow [\tau_k] & \text{By i.h.} \\ [\gamma] \vdash_{\mathbf{E}} (\text{inj}_k [e_0]) \varphi \Leftarrow [\tau_1] + [\tau_2] & \text{By E+Intro}_k \\ [\gamma] \vdash_{\mathbf{E}} (\text{inj}_k [e_0]) \varphi \Leftarrow \epsilon \blacktriangleright ([\tau_1] + [\tau_2]) & \text{By E}\blacktriangleright\text{Intro (first conclusion)} \\ \vDash [\gamma] \vdash_{\mathbf{E}} [\text{inj}_k e_0] \varphi \Leftarrow [\tau_1] + [\tau_2] & \text{By def. of } [-] \end{array}$$

• Case
$$\frac{\gamma, x_1 \text{val} \Rightarrow \tau_1 \vdash_{\mathbf{I}} e_1 \varphi_1 \Leftarrow \tau \quad \gamma, x_2 \text{val} \Rightarrow \tau_2 \vdash_{\mathbf{I}} e_2 \varphi_2 \Leftarrow \tau}{\gamma \vdash_{\mathbf{I}} e_0 \varphi_0 \Rightarrow (\tau_1 +^e \tau_2) \quad \gamma \vdash_{\mathbf{I}} \text{case}(e_0, x_1.e_1, x_2.e_2) \top \Leftarrow \tau} \text{I+Elim}$$

$\gamma \vdash_I e_0 \varphi_0 \Rightarrow (\tau_1 +^\epsilon \tau_2)$	Subderivation
$[\gamma] \vdash_E [e_0] \varphi_0 \Rightarrow [\tau_1 +^\epsilon \tau_2]$	By i.h.
$[\gamma] \vdash_E [e_0] \varphi_0 \Rightarrow \mathbf{e}\blacktriangleright([\tau_1] + [\tau_2])$	By def. of $[-]$
$[\gamma] \vdash_E [e_0] \top \Rightarrow ([\tau_1] + [\tau_2])$	By $\mathbf{E}\blacktriangleright\mathbf{Elim}_\epsilon$

$\gamma, x_1 \text{ val} \Rightarrow \tau_1 \vdash_I e_1 \varphi_1 \Rightarrow \tau$	Subderivation
$[\gamma], x_1 : \mathbf{V}\blacktriangleright[\tau_1] \vdash_E [e_1] \varphi_1 \Rightarrow [\tau]$	By i.h. and def. of $[-]$
$[\gamma], x_1 : [\tau_1] \vdash_E [e_1] \varphi_1 \Rightarrow [\tau]$	By Lemma 19

$[\gamma], x_2 : [\tau_2] \vdash_E [e_2] \varphi_2 \Rightarrow [\tau]$ Similarly

\vDash $[\gamma] \vdash_E [\text{case}(e_0, x_1.e_1, x_2.e_2)] \top \Rightarrow [\tau]$ By $\mathbf{E}+\mathbf{Elim}$

• **Case** $\frac{\gamma \vdash_I e \varphi \Leftarrow [(\mu^\epsilon \alpha. \tau_0) / \alpha] \tau_0}{\gamma \vdash_I e \varphi \Leftarrow \mu^\epsilon \alpha. \tau_0}$ $\mathbf{I}\mu\mathbf{Intro}$

$\gamma \vdash_I e \varphi \Leftarrow [(\mu^\epsilon \alpha. \tau_0) / \alpha] \tau_0$	Subderivation
$[\gamma] \vdash_E [e] \varphi \Leftarrow [[(\mu^\epsilon \alpha. \tau_0) / \alpha] \tau_0]$	By i.h.
$[\gamma] \vdash_E [e] \varphi \Leftarrow [[\mu^\epsilon \alpha. \tau_0] / \alpha] [\tau_0]$	By a property of substitution/ $[-]$
$[\gamma] \vdash_E [e] \varphi \Leftarrow [(\mu \alpha. \mathbf{e}\blacktriangleright[\tau_0]) / \alpha] [\tau_0]$	By def. of $[-]$

$[\gamma] \vdash_E [e] \varphi \Leftarrow \mu \alpha. \mathbf{e}\blacktriangleright[\tau_0]$	By $\mathbf{E}\mu\mathbf{Intro}$
\vDash $[\gamma] \vdash_E [e] \varphi \Leftarrow [\mu^\epsilon \alpha. \tau_0]$	By def. of $[-]$

• **Case** $\frac{\gamma \vdash_I e \varphi_0 \Rightarrow \mu^\epsilon \alpha. \tau_0}{\gamma \vdash_I e \top \Rightarrow [(\mu^\epsilon \alpha. \tau_0) / \alpha] \tau_0}$ $\mathbf{I}\mu\mathbf{Elim}$

$\gamma \vdash_I e \varphi_0 \Rightarrow \mu^\epsilon \alpha. \tau_0$	Subderivation
$[\gamma] \vdash_E [e] \varphi_0 \Rightarrow [\mu^\epsilon \alpha. \tau_0]$	By i.h.
$[\gamma] \vdash_E [e] \varphi_0 \Rightarrow \mu \alpha. \mathbf{e}\blacktriangleright[\tau_0]$	By def. of $[-]$

$[\gamma] \vdash_E [e] \top \Rightarrow [(\mu \alpha. \mathbf{e}\blacktriangleright[\tau_0]) / \alpha] \mathbf{e}\blacktriangleright[\tau_0]$	By $\mathbf{E}\mu\mathbf{Elim}$
$[\gamma] \vdash_E [e] \top \Rightarrow [[\mu^\epsilon \alpha. \tau_0] / \alpha] \mathbf{e}\blacktriangleright[\tau_0]$	By def. of $[-]$
$[\gamma] \vdash_E [e] \top \Rightarrow \mathbf{e}\blacktriangleright[[\mu^\epsilon \alpha. \tau_0] / \alpha] [\tau_0]$	By a property of substitution
$[\gamma] \vdash_E [e] \top \Rightarrow [[\mu^\epsilon \alpha. \tau_0] / \alpha] [\tau_0]$	By $\mathbf{E}\blacktriangleright\mathbf{Elim}_\epsilon$

\vDash $[\gamma] \vdash_E [e] \top \Rightarrow [(\mu^\epsilon \alpha. \tau_0) / \alpha] \tau_0$ By a property of substitution/ $[-]$ □

B.5 Elaboration

Lemma 5. If $\Gamma \vdash e \varphi : S \leftrightarrow W$ then $\varphi = \text{val}$.

Proof. By induction on the given derivation.

For any rule whose conclusion has val , we already have our result. This takes care of elab1Intro , $\text{elab}\forall\mathbf{Intro}$, $\text{elab}\mathbf{\Delta}\mathbf{Intro}$, the second conclusion of $\text{elab}\blacktriangleright\mathbf{Intro}$, elabvar , and $\text{elab}\rightarrow\mathbf{Intro}$. Rules whose conclusions have target terms that can never be a value are impossible, which takes care of $\text{elab}\forall\mathbf{Elim}$, $\text{elab}\mathbf{\Delta}\mathbf{Elim}$, $\text{elab}\blacktriangleright\mathbf{Elim}_\mathbb{N}$, elabfixvar , elabfix , $\text{elab}\rightarrow\mathbf{Elim}$, $\text{elab}\ast\mathbf{Elim}_k$, $\text{elab}+\mathbf{Elim}$, and $\text{elab}\mu\mathbf{Elim}$. We are left with:

- **Case $\text{elab}\blacktriangleright\mathbf{Intro}$** (first conclusion): The result follows by i.h. and $\text{elab}\blacktriangleright\mathbf{Intro}$.
- **Case $\text{elab}\ast\mathbf{Intro}$** : We have $W = (W_1, W_2)$. By i.h. twice, $\varphi_1 = \text{val}$ and $\varphi_2 = \text{val}$. Applying $\text{elab}\ast\mathbf{Intro}$ gives the result (using $\text{val} \sqcup \text{val} = \text{val}$).
- **Cases $\text{elab}\blacktriangleright\mathbf{Elim}_\forall$, $\text{elab}+\mathbf{Intro}_k$, $\text{elab}\mu\mathbf{Intro}$** : The result follows by i.h. and applying the same rule. □

Lemma 6 (Elaboration valuability).

If $\Gamma \vdash e \text{val} : S \leftrightarrow M$ then M is valuable, that is, there exists \tilde{V} such that $M = \tilde{V}$.

Proof. By induction on the given derivation.

- **Cases elabvar , elab1Intro , $\text{elab}\rightarrow\mathbf{Intro}$** : Immediate.
- **Cases $\text{elab}\blacktriangleright\mathbf{Intro}$** (N conclusion), $\text{elab}\blacktriangleright\mathbf{Elim}_\mathbb{N}$, elabfix , elabfixvar , $\text{elab}\rightarrow\mathbf{Elim}$, $\text{elab}\ast\mathbf{Elim}_k$, $\text{elab}+\mathbf{Elim}$, $\text{elab}\mu\mathbf{Elim}$: Impossible: these rules cannot elaborate values.

- **Case $elab\Delta$ Intro**: By i.h., M_1 and M_2 are valuable; therefore (M_1, M_2) is valuable.
- **Case $elab\Delta$ Elim**: By i.h., M_0 is valuable; therefore $\text{proj}_1 M_0$ and $\text{proj}_2 M_0$ are valuable.
- **Case $elab*$ Intro**: Similar to the $elab\Delta$ Intro case.
- **Cases $elab\forall$ Intro, $elab\forall$ Elim**: By i.h., M_0 is valuable; therefore $\bigwedge_.$ M_0 and $M[_]_0$ are valuable.
- **Cases $elab\blacktriangleright$ Intro (\vee conclusion), $elab\blacktriangleright$ Elim \vee** : By i.h.
- **Case $elab+$ Intro κ** : By i.h., M_0 is valuable; therefore $\text{inj}_\kappa M_0$ is valuable.
- **Case $elab\mu$ Intro**: By i.h., M_0 is valuable; therefore $\text{roll } M_0$ is valuable. □

Lemma 7 (Substitution—Evaluation orders).

- (1) If $\Gamma, a \text{ evalorder}, \Gamma' \vdash S \text{ type}$ and $\Gamma \vdash e \text{ evalorder}$
then $\Gamma, [\epsilon/a]\Gamma' \vdash [\epsilon/a]S \text{ type}$.
- (2) If \mathcal{D} derives $\Gamma, a \text{ evalorder}, \Gamma' \vdash_{\mathbf{E}} e \varphi \Leftarrow S$ and $\Gamma \vdash e \text{ evalorder}$
then \mathcal{D}' derives $\Gamma, [\epsilon/a]\Gamma' \vdash_{\mathbf{E}} e \varphi \Leftarrow [\epsilon/a]S$ where \mathcal{D}' is not larger than \mathcal{D} .
- (3) If \mathcal{D} derives $\Gamma, a \text{ evalorder}, \Gamma' \vdash_{\mathbf{E}} e \varphi \Rightarrow S$ and $\Gamma \vdash e \text{ evalorder}$,
then \mathcal{D}' derives $\Gamma, [\epsilon/a]\Gamma' \vdash_{\mathbf{E}} e \varphi \Rightarrow [\epsilon/a]S$ where \mathcal{D}' is not larger than \mathcal{D} .

Proof. Part (1): By induction on the first derivation. Part (1) does not depend on the other parts.
Parts (2) and (3): By induction on the given derivation, using part (1):

- **Case $\mathbf{E}\forall$ Intro**: By i.h. and $\mathbf{E}\forall$ Intro.

- **Case**
$$\frac{\Gamma, a \text{ evalorder}, \Gamma' \vdash_{\mathbf{E}} e \varphi \Rightarrow \forall \alpha. S_0 \quad \Gamma, a \text{ evalorder}, \Gamma' \vdash S' \text{ type}}{\Gamma, a \text{ evalorder}, \Gamma' \vdash_{\mathbf{E}} e \varphi \Rightarrow [S'/\alpha]S_0} \mathbf{E}\forall\text{Elim}$$

$\Gamma, a \text{ evalorder}, \Gamma' \vdash_{\mathbf{E}} e \varphi \Rightarrow \forall \alpha. S_0$	Subderivation
$\Gamma, [\epsilon/a]\Gamma' \vdash_{\mathbf{E}} e \varphi \Rightarrow [\epsilon/a](\forall \alpha. S_0)$	By i.h.
$\Gamma, [\epsilon/a]\Gamma' \vdash_{\mathbf{E}} e \varphi \Rightarrow \forall \alpha. [\epsilon/a]S_0$	By def. of subst.
$\Gamma, a \text{ evalorder}, \Gamma' \vdash S'$	Subderivation
$\Gamma, [\epsilon/a]\Gamma' \vdash [\epsilon/a]S'$	By part (1)
$\Gamma, [\epsilon/a]\Gamma' \vdash_{\mathbf{E}} e \varphi \Rightarrow [[\epsilon/a]S'/\alpha][\epsilon/a]S_0$	By $\mathbf{E}\forall\text{Elim}$
$\Gamma, [\epsilon/a]\Gamma' \vdash_{\mathbf{E}} e \varphi \Rightarrow [\epsilon/a][S'/\alpha]S_0$	By def. of subst.

- **Case**
$$\frac{(\chi : S) \in (\Gamma, a \text{ evalorder}, \Gamma')}{\Gamma, a \text{ evalorder}, \Gamma' \vdash_{\mathbf{E}} \chi_{\text{val}} \Rightarrow S} \mathbf{E}\text{var}$$

Follows from the definition of substitution on contexts.

- **Case $\mathbf{E}\text{fixvar}$** : Similar to the $\mathbf{E}\text{var}$ case.

The remaining cases are straightforward, using the i.h. and properties of substitution. □

Lemma 22 (Type substitution).

- (1) If $\Gamma \vdash S' \text{ type}$ and $\Gamma, \alpha \text{ type} \vdash S \text{ type}$ then $\Gamma \vdash [S'/\alpha]S \text{ type}$.
- (2) If $\Gamma \vdash S' \text{ type}$ and $\Gamma, \alpha \text{ type} \vdash e \varphi : S \leftrightarrow M$ then $\Gamma \vdash e \varphi : [S'/\alpha]S \leftrightarrow M$.

Proof. In each part, by induction on the second derivation. In part (2), the $elab\forall$ Elim case uses part (1). □

Lemma 8 (Expression substitution).

- (1) If $\Gamma \vdash e_1 \varphi_1 : S_1 \leftrightarrow W$ and $\Gamma, \chi : S_1, \Gamma' \vdash e_2 \varphi_2 : S \leftrightarrow M$
then $\Gamma, \Gamma' \vdash [e_1/\chi]e_2 \varphi_2 : S \leftrightarrow [W/\chi]M$.
- (2) If $\Gamma \vdash \text{fix } u. e_1 \top : S_1 \leftrightarrow \text{fix } u. M_1$
and $\Gamma, u : S_1, \Gamma' \vdash e_2 \varphi_2 : S \leftrightarrow M$
then $\Gamma, \Gamma' \vdash [(\text{fix } u. e_1)/u]e_2 \varphi_2 : S \leftrightarrow [(\text{fix } u. M_1)/u]M$.

Proof. Part (1): By induction on the given derivation. In the $elab\text{var}$ case, use Lemma 5 to get $\Gamma \vdash e_1 \text{val} : S_1 \leftrightarrow W$. By weakening, $\Gamma, \Gamma' \vdash e_1 \text{val} : S_1 \leftrightarrow W$, which is $\Gamma, \Gamma' \vdash [e_1/\chi]\chi \text{val} : S \leftrightarrow [W/\chi]M$.

Part (2): By induction on the given derivation. Note that in the $elab\text{fixvar}$ case, $\varphi_2 = \top$. □

Theorem 10 (Elaboration type soundness).

If $\Gamma \vdash_{\mathbf{E}} e \varphi \Leftarrow S$ or $\Gamma \vdash_{\mathbf{E}} e \varphi \Rightarrow S$
 where $\Gamma \vdash S$ type and Γ contains no a *evalorder* declarations
 then there exists M such that $\Gamma \vdash \text{er}(e)_{\varphi'} : S \leftrightarrow M$
 where $\varphi' \sqsubseteq \varphi$ and $|\Gamma| \vdash_{\mathbf{T}} M : |S|$.

Proof. By induction on the size of the given derivation. If $\varphi' = \varphi$, we often don't bother to state $\varphi \sqsubseteq \varphi$ explicitly.

- **Case** $\frac{(x : S) \in \Gamma}{\Gamma \vdash_{\mathbf{E}} x \text{ val} \Rightarrow S} \mathbf{Evar}$
 - $\text{er}(x)_{\varphi} : S \leftrightarrow x$ Premise
 - $\Gamma \vdash \text{er}(x)_{\varphi} : S \leftrightarrow x$ By *elabvar*
 - $(x : |S|) \in |\Gamma|$ By def. of $|-|$
 - $|\Gamma| \vdash_{\mathbf{T}} x : |S|$ By *Tvar*
- **Case** *Efixvar*: Similar to the *Evar* case.
- **Case** $\frac{\Gamma, u : S \vdash_{\mathbf{E}} e_0 \varphi_0 \Leftarrow S}{\Gamma \vdash_{\mathbf{E}} (\text{fix } u. e_0)_{\top} \Leftarrow S} \mathbf{Efix}$
 - $\Gamma, u : S \vdash_{\mathbf{E}} e_0 \varphi_0 \Leftarrow S$ Subderivation
 - $\Gamma, u : S \vdash \text{er}(e_0)_{\varphi'_0} : S \leftrightarrow M_0$ By i.h.
 - $|\Gamma, u : S| \vdash_{\mathbf{T}} M_0 : |S|$ "
 - $|\Gamma|, u : |S| \vdash_{\mathbf{T}} M_0 : |S|$ By def. of $|-|$
 - $\Gamma, u : S \vdash \text{er}(e_0)_{\top} : S \leftrightarrow \text{fix } u. M_0$ By *elabfix*
 - $|\Gamma| \vdash_{\mathbf{T}} (\text{fix } u. M_0) : |S|$ By *Tfix*
- **Case** $\frac{\Gamma \vdash_{\mathbf{E}} e \varphi \Rightarrow S}{\Gamma \vdash_{\mathbf{E}} e \varphi \Leftarrow S} \mathbf{Esub}$
 - $\Gamma \vdash_{\mathbf{E}} e \varphi \Rightarrow S$ Subderivation
 - $\Gamma \vdash \text{er}(e)_{\varphi'} : S \leftrightarrow M$ By i.h.
 - $\varphi' \sqsubseteq \varphi$ "
 - $|\Gamma| \vdash_{\mathbf{T}} M : |S|$ "
- **Case** $\frac{\Gamma \vdash_{\mathbf{E}} e_0 \varphi \Leftarrow S}{\Gamma \vdash_{\mathbf{E}} (e_0 : S)_{\varphi} \Rightarrow S} \mathbf{Eanno}$
 - $\Gamma \vdash_{\mathbf{E}} e_0 \varphi \Leftarrow S$ Subderivation
 - $\Gamma \vdash \text{er}(e_0)_{\varphi'} : S \leftrightarrow M$ By i.h.
 - $\varphi' \sqsubseteq \varphi$ "
 - $|\Gamma| \vdash_{\mathbf{T}} M : |S|$ "
 - $\Gamma \vdash \text{er}((e_0 : S))_{\varphi'} : S \leftrightarrow M$ By def. of $\text{er}(-)$
- **Case** $\frac{}{\Gamma \vdash_{\mathbf{E}} () \text{ val} \Leftarrow \mathbf{1}} \mathbf{E1Intro}$
 - $\Gamma \vdash \text{er}(()_{\varphi}) : \mathbf{1} \leftrightarrow ()$ By *elab1Intro*
 - $|\Gamma| \vdash_{\mathbf{T}} () : \mathbf{1}$ By *T1Intro*
 - $|\Gamma| \vdash_{\mathbf{T}} () : |\mathbf{1}|$ By def. of $|-|$
- **Case** $\frac{\Gamma, a \text{ evalorder} \vdash_{\mathbf{E}} e \text{ val} \Leftarrow S_0}{\Gamma \vdash_{\mathbf{E}} e \text{ val} \Leftarrow \Delta a. S_0} \mathbf{E\Delta Intro}$

$\Gamma, a \vdash_{\mathbf{E}} e_{\text{val}} \Leftarrow S_0$	Subd.
$\Gamma \vdash_{\mathbf{E}} e_{\text{val}} \Leftarrow [V/a]S_0$	By Lemma 7 (2)
$\Gamma \vdash \text{er}(e)_{\text{val}}: [V/a]S_0 \leftrightarrow M_V$	By i.h.
$ \Gamma \vdash_{\mathbf{T}} M_V : [V/a]S_0 $	"
$\Gamma \vdash_{\mathbf{E}} e_{\text{val}} \Leftarrow [N/a]S_0$	By Lemma 7 (2)
$\Gamma \vdash \text{er}(e)_{\text{val}}: [N/a]S_0 \leftrightarrow M_N$	By i.h.
$ \Gamma \vdash_{\mathbf{T}} M_N : [N/a]S_0 $	"
$\vDash \Gamma \vdash \text{er}(e)_{\text{val}}: \Delta a. S_0 \leftrightarrow (M_V, M_N)$	By <i>elab</i> Δ <i>Intro</i>
$ \Gamma \vdash_{\mathbf{T}} (M_V, M_N) : S_1 * S_2 $	By $\mathbf{T}*$ <i>Intro</i>
$\vDash \Gamma \vdash_{\mathbf{T}} (M_V, M_N) : \Delta a. S_0 $	By def. of $ - $

• **Case** $\frac{\Gamma \vdash_{\mathbf{E}} e_{\varphi} \Rightarrow \Delta a. S_0 \quad \Gamma \vdash \epsilon \text{ evalorder}}{\Gamma \vdash_{\mathbf{E}} e_{\varphi} \Rightarrow [\epsilon/a]S_0} \mathbf{E}\Delta\mathbf{Elim}$

$\Gamma \vdash \text{er}(e)_{\varphi}': \Delta a. S_0 \leftrightarrow M_0$	By i.h.
$\vDash \varphi' \sqsubseteq \varphi$	"
$ \Gamma \vdash_{\mathbf{T}} M_0 : [V/a]S_0 * [N/a]S_0 $	"

If $\epsilon = V$ then:

$\vDash \Gamma \vdash \text{er}(e)_{\varphi}': [V/a]S_0 \leftrightarrow \text{proj}_1 M_0$	By <i>elab</i> Δ <i>Elim</i>
$\vDash \Gamma \vdash_{\mathbf{T}} \text{proj}_1 M_0 : [V/a]S_0 $	By $\mathbf{T}*$ <i>Elim</i> ₁

Otherwise, $\epsilon \neq V$. It is given that Γ contains no a -declarations, and we also have $\Gamma \vdash \epsilon \text{ evalorder}$. It follows that ϵ cannot be a variable a . Therefore $\epsilon = N$.

$\vDash \Gamma \vdash \text{er}(e)_{\varphi}': [N/a]S_0 \leftrightarrow \text{proj}_2 M_0$	By <i>elab</i> Δ <i>Elim</i>
$\vDash \Gamma \vdash_{\mathbf{T}} \text{proj}_2 M_0 : [N/a]S_0 $	By $\mathbf{T}*$ <i>Elim</i> ₂

• **Case** $\frac{\Gamma \vdash_{\mathbf{E}} e_{\varphi} \Leftarrow S_0}{\Gamma \vdash_{\mathbf{E}} e_{\varphi} \Leftarrow \epsilon \blacktriangleright S_0} \mathbf{E}\blacktriangleright\mathbf{Intro}$ (first conclusion)

$\Gamma \vdash_{\mathbf{E}} e_{\varphi} \Leftarrow S_0$	Subderivation
$\Gamma \vdash \text{er}(e)_{\varphi}': S_0 \leftrightarrow M_0$	By i.h.
$\varphi' \sqsubseteq \varphi$	"
$ \Gamma \vdash_{\mathbf{T}} M_0 : S_0 $	"

By similar reasoning as in the $\mathbf{E}\Delta\mathbf{Elim}$ case, either $\epsilon = V$ or $\epsilon = N$.

If $\epsilon = V$:

$ S_0 = V\blacktriangleright S_0 $	By def. of $ - $
Let $M = M_0$.	
$\vDash \Gamma \vdash \text{er}(e)_{\varphi}': V\blacktriangleright S_0 \leftrightarrow M$	By <i>elab</i> \blacktriangleright <i>Intro</i> (first conclusion)
$\vDash \varphi' \sqsubseteq \varphi$	Above
$\vDash \Gamma \vdash_{\mathbf{T}} M : V\blacktriangleright S_0 $	By above equality

If $\epsilon = N$:

$\mathbf{U} S_0 = N\blacktriangleright S_0 $	By def. of $ - $
Let $M = \text{thunk } M_0$.	
$\vDash \Gamma \vdash \text{er}(e)_{\text{val}}: N\blacktriangleright S_0 \leftrightarrow \text{thunk } M_0$	By <i>elab</i> \blacktriangleright <i>Intro</i> (second conclusion)
$\vDash \text{val} \sqsubseteq \varphi$	By def. of \sqsubseteq
$ \Gamma \vdash_{\mathbf{T}} \text{thunk } M_0 : \mathbf{U} S_0 $	By $\mathbf{T}\rightarrow$ <i>Intro</i>
$\vDash \Gamma \vdash_{\mathbf{T}} M : N\blacktriangleright S_0 $	By above equalities

• **Case** $\frac{\Gamma \vdash_{\mathbf{E}} e_{\varphi} \Leftarrow S_0}{\Gamma \vdash_{\mathbf{E}} e_{\text{val}} \Leftarrow N\blacktriangleright S_0} \mathbf{E}\blacktriangleright\mathbf{Intro}$ (second conclusion)

$\Gamma \vdash_{\mathbf{E}} e_{\varphi'} \Leftarrow S_0$	Subderivation
$\Gamma \vdash \text{er}(e)_{\varphi'} : S_0 \leftrightarrow M_0$	By i.h.
$\varphi' \sqsubseteq \varphi$	"
$ \Gamma \vdash_{\mathbf{T}} M_0 : S_0 $	"
$\mathbf{U} S_0 = \mathbf{N}\blacktriangleright S_0 $	By def. of $ - $

Let $M = \text{thunk } M_0$.

$\Gamma \vdash \text{er}(e)_{\text{val}} : \mathbf{N}\blacktriangleright S_0 \leftrightarrow \text{thunk } M_0$	By <i>elab</i> ►Intro (second conclusion)
$\text{val} \sqsubseteq \varphi$	By def. of \sqsubseteq
$ \Gamma \vdash_{\mathbf{T}} \text{thunk } M_0 : \mathbf{U} S_0 $	By $\mathbf{T}\rightarrow$ Intro
$ \Gamma \vdash_{\mathbf{T}} M : \mathbf{N}\blacktriangleright S_0 $	By above equalities

• Case $\frac{\Gamma \vdash_{\mathbf{E}} e_{\varphi} \Rightarrow \mathbf{V}\blacktriangleright S}{\Gamma \vdash_{\mathbf{E}} e_{\varphi} \Rightarrow S} \mathbf{E}\blacktriangleright\text{Elim}_V$

$\Gamma \vdash_{\mathbf{E}} e_{\varphi} \Rightarrow \mathbf{V}\blacktriangleright S$	Subderivation
$\Gamma \vdash \text{er}(e)_{\varphi'} : \mathbf{V}\blacktriangleright S \leftrightarrow M_0$	By i.h.
$\varphi' \sqsubseteq \varphi$	"
$ \Gamma \vdash_{\mathbf{T}} M_0 : \mathbf{V}\blacktriangleright S $	"
$ \mathbf{V}\blacktriangleright S = S $	By def. of $ - $

Let $M = M_0$.

$\Gamma \vdash \text{er}(e)_{\varphi'} : S \leftrightarrow M$	By <i>elab</i> ►Elim _V
$ \Gamma \vdash_{\mathbf{T}} M : S $	By above equalities

• Case $\frac{\Gamma \vdash_{\mathbf{E}} e_{\varphi'} \Rightarrow \epsilon \blacktriangleright S}{\Gamma \vdash_{\mathbf{E}} e_{\top} \Rightarrow S} \mathbf{E}\blacktriangleright\text{Elim}_{\epsilon}$

By similar reasoning as in the $\mathbf{E}\blacktriangleright\text{Elim}$ case, either $\epsilon = \mathbf{V}$ or $\epsilon = \mathbf{N}$.

If $\epsilon = \mathbf{V}$, follow the $\mathbf{E}\blacktriangleright\text{Elim}_V$ case above.

If $\epsilon = \mathbf{N}$:

$\Gamma \vdash_{\mathbf{E}} e_{\varphi'} \Rightarrow \mathbf{N}\blacktriangleright S$	Subderivation
$\Gamma \vdash \text{er}(e)_{\varphi'} : \mathbf{N}\blacktriangleright S \leftrightarrow M_0$	By i.h.
$ \Gamma \vdash_{\mathbf{T}} M_0 : \mathbf{N}\blacktriangleright S $	"
$ \mathbf{N}\blacktriangleright S = \mathbf{U} S $	By def. of $ - $
Let $M = (\text{force } M_0)$.	
$\Gamma \vdash \text{er}(e)_{\top} : S \leftrightarrow \text{force } M_0$	By <i>elab</i> ►Elim _N
$\top \sqsubseteq \varphi$	By def. of \sqsubseteq
$ \Gamma \vdash_{\mathbf{T}} M_0 : \mathbf{U} S $	Above ($ \mathbf{N}\blacktriangleright S = \mathbf{U} S $)
$ \Gamma \vdash_{\mathbf{T}} \text{force } M_0 : S $	By $\mathbf{T}\mathbf{U}\text{Elim}$

• Case $\frac{\Gamma \vdash_{\mathbf{E}} e_1 \varphi_1 \Leftarrow S_1 \quad \Gamma \vdash_{\mathbf{E}} e_2 \varphi_2 \Leftarrow S_2}{\Gamma \vdash_{\mathbf{E}} (e_1, e_2)_{\varphi_1 \sqcup \varphi_2} \Leftarrow (S_1 * S_2)} \mathbf{E}\blacktriangleright\text{Intro}$

$\Gamma \vdash \text{er}(e_1)_{\varphi} : S_1 \leftrightarrow M_1$	By i.h.
$\varphi'_1 \sqsubseteq \varphi_1$	"
$ \Gamma \vdash_{\mathbf{T}} M_1 : S_1 $	"
$\Gamma \vdash \text{er}(e_2)_{\varphi} : S_2 \leftrightarrow M_2$	By i.h.
$\varphi'_2 \sqsubseteq \varphi_2$	"
$ \Gamma \vdash_{\mathbf{T}} M_2 : S_2 $	"
$\Gamma \vdash (\text{er}(e_1), \text{er}(e_2))_{\varphi'_1 \sqcup \varphi'_2} : (S_1 * S_2) \leftrightarrow (M_1, M_2)$	By <i>elab</i> *Intro
$\varphi'_1 \sqcup \varphi'_2 \sqsubseteq \varphi_1 \sqcup \varphi_2$	$\varphi'_1 \sqsubseteq \varphi_1$ and $\varphi'_2 \sqsubseteq \varphi_2$
$ \Gamma \vdash_{\mathbf{T}} (M_1, M_2) : S_1 * S_2 $	By $\mathbf{T}\blacktriangleright$ Intro
$ \Gamma \vdash_{\mathbf{T}} (M_1, M_2) : S_1 * S_2 $	By def. of $ - $

- **Case**
$$\frac{\Gamma \vdash_{\mathbf{E}} e_0 \varphi_0 \Rightarrow (S_1 * S_2)}{\Gamma \vdash_{\mathbf{E}} (\text{proj}_k e_0) \top \Rightarrow S_k} \mathbf{E*Elim}_k$$
 - $\Gamma \vdash \text{er}(e_0)_{\varphi'_0} : (S_1 * S_2) \hookrightarrow M_0$ By i.h.
 - $|\Gamma| \vdash_{\mathbf{T}} M_0 : |S_1 * S_2|$ "
 - $|\Gamma| \vdash_{\mathbf{T}} M_0 : |S_1| * |S_2|$ By def. of $|-|$
 - ☞ $\Gamma \vdash (\text{proj}_k \text{er}(e_0)) \top : S_k \hookrightarrow (\text{proj}_k M_0)$ By *elab*Elim*_k
 - ☞ $|\Gamma| \vdash_{\mathbf{T}} (\text{proj}_k M_0) : |S_k|$ By *T*Elim*_k

- **Case**
$$\frac{\Gamma, x : S_1 \vdash_{\mathbf{E}} e_0 \varphi_0 \Leftarrow S_2}{\Gamma \vdash_{\mathbf{E}} (\lambda x. e_0) \text{val} \Leftarrow (S_1 \rightarrow S_2)} \mathbf{E\rightarrow Intro}$$
 - $\Gamma, x : S_1 \vdash \text{er}(e_0)_{\varphi'_0} : S_2 \hookrightarrow M_0$ By i.h.
 - $|\Gamma, x : S_1| \vdash_{\mathbf{T}} M_0 : |S_2|$ "
 - $|\Gamma, x : S_1| = (|\Gamma|, x : |S_1|)$ By def. of $|-|$
 - $\Gamma, x : S_1 \vdash \text{er}(e_0)_{\varphi'_0} : S_2 \hookrightarrow M_0$ Above
 - ☞ $\Gamma \vdash (\lambda x. e_0) \text{val} : (S_1 \rightarrow S_2) \hookrightarrow (\lambda x. M_0)$ By *elab→Intro*
 - $|\Gamma|, x : |S_1| \vdash_{\mathbf{T}} M_0 : |S_2|$ Above
 - $|\Gamma|, x : |S_1| \vdash_{\mathbf{T}} (\lambda x. M_0) : |S_1| \rightarrow |S_2|$ By *T→Intro*
 - ☞ $|\Gamma|, x : |S_1| \vdash_{\mathbf{T}} (\lambda x. M_0) : |S_1 \rightarrow S_2|$ By def. of $|-|$

- **Case**
$$\frac{\Gamma \vdash_{\mathbf{E}} e_1 \varphi_1 \Rightarrow (S_1 \rightarrow S) \quad \Gamma \vdash_{\mathbf{E}} e_2 \varphi_2 \Leftarrow S_1}{\Gamma \vdash_{\mathbf{E}} (e_1 @ e_2) \top \Rightarrow S} \mathbf{E\rightarrow Elim}$$
 - $\Gamma \vdash \text{er}(e_1)_{\varphi'_1} : (S' \rightarrow S) \hookrightarrow M_1$ By i.h.
 - $|\Gamma| \vdash_{\mathbf{T}} M_1 : |S' \rightarrow S|$ "
 - $|\Gamma| \vdash_{\mathbf{T}} M_1 : |S'| \rightarrow |S|$ By def. of $|-|$
 - $\Gamma \vdash \text{er}(e_2)_{\varphi'_2} : S' \hookrightarrow M_2$ By i.h.
 - $|\Gamma| \vdash_{\mathbf{T}} M_2 : |S'|$ "
 - ☞ $\Gamma \vdash \text{er}(e_1 @ e_2) \top : (S' \rightarrow S) \hookrightarrow (M_1 M_2)$ By *elab→Elim*
 - ☞ $|\Gamma| \vdash_{\mathbf{T}} (M_1 M_2) : |S|$ By *T→Elim*

- **Case**
$$\frac{\Gamma, \alpha \text{ type} \vdash_{\mathbf{E}} e_0 \text{val} \Leftarrow S_0}{\Gamma \vdash_{\mathbf{E}} \Lambda \alpha. e_0 \text{val} \Leftarrow \forall \alpha. S_0} \mathbf{E\forall Intro}$$
 - $\Gamma, \alpha \text{ type} \vdash_{\mathbf{E}} e_0 \text{val} \Leftarrow S_0$ Subderivation
 - $\Gamma, \alpha \text{ type} \vdash \text{er}(e_0)_{\text{val}} : S_0 \hookrightarrow M_0$ By i.h.
 - $|\Gamma, \alpha \text{ type}| \vdash_{\mathbf{T}} M_0 : |S_0|$ "
 - $|\Gamma|, \alpha \text{ type} \vdash_{\mathbf{T}} M_0 : |S_0|$ By def. of $|-|$
 - ☞ $\Gamma \vdash \text{er}(e_0)_{\text{val}} : \forall \alpha. S_0 \hookrightarrow \Lambda _ . M_0$ By *elab∀Intro*
 - ☞ $\Gamma \vdash \text{er}(\Lambda \alpha. e_0)_{\text{val}} : \forall \alpha. S_0 \hookrightarrow \Lambda _ . M_0$ By def. of *er(-)*
 - $|\Gamma| \vdash_{\mathbf{T}} \Lambda _ . M_0 : |\forall \alpha. S_0|$ By *T∀Intro*
 - ☞ $|\Gamma| \vdash_{\mathbf{T}} \Lambda _ . M_0 : |\forall \alpha. S_0|$ By def. of subst.

- **Case**
$$\frac{\Gamma \vdash_{\mathbf{E}} e_0 \varphi \Rightarrow \forall \alpha. S_0 \quad \Gamma \vdash S' \text{ type}}{\Gamma \vdash_{\mathbf{E}} e_0 [S'] \varphi \Rightarrow [S'/\alpha] S_0} \mathbf{E\forall Elim}$$

	$\Gamma \vdash_{\mathbf{E}} e_0 \varphi \Rightarrow \forall \alpha. S_0$	Subderivation
	$\Gamma \vdash \text{er}(e_0) \varphi': \forall \alpha. S_0 \hookrightarrow M_0$	By i.h.
☞	$\varphi' \sqsubseteq \varphi$	"
	$ \Gamma \vdash_{\mathbf{T}} M_0 : \forall \alpha. S_0 $	"
	$\Gamma \vdash S' \text{ type}$	Subderivation
	$\Gamma \vdash \text{er}(e_0) \varphi': [S'/\alpha]S_0 \hookrightarrow M_0[_]$	By <i>elab</i> Elim
☞	$\Gamma \vdash \text{er}(e_0[S']) \varphi': [S'/\alpha]S_0 \hookrightarrow M_0[_]$	By def. of $\text{er}(-)$
	$ \Gamma \vdash S' $	By Lemma 9
	$ \Gamma \vdash_{\mathbf{T}} M_0 : \forall \alpha. S_0 $	By def. of $ - $
	$ \Gamma \vdash_{\mathbf{T}} M_0[_] : [S'/\alpha]S_0 $	By $\mathbf{T}\forall$ Elim
	$[S'/\alpha] S_0 = [S'/\alpha]S_0 $	From def. of subst.
☞	$ \Gamma \vdash_{\mathbf{T}} M_0[_] : [S'/\alpha]S_0 $	By above equality

• **Case**
$$\frac{\Gamma \vdash_{\mathbf{E}} e_0 \varphi \Leftarrow S_k}{\Gamma \vdash_{\mathbf{E}} (\text{inj}_k e_0) \varphi \Leftarrow (S_1 + S_2)} \mathbf{E+Intro}_k$$

	$\Gamma \vdash_{\mathbf{E}} e_0 \varphi \Leftarrow S_k$	Subderivation
	$\Gamma \vdash \text{er}(e_0) \varphi': S_k \hookrightarrow M_0$	By i.h.
☞	$\varphi' \sqsubseteq \varphi$	"
	$ \Gamma \vdash_{\mathbf{T}} M_0 : S_k $	"
☞	$\Gamma \vdash \text{inj}_k \text{er}(e_0) \varphi': (S_1 + S_2) \hookrightarrow \text{inj}_k M_0$	By <i>elab</i> +Intro _k
	$ \Gamma \vdash_{\mathbf{T}} \text{inj}_k M_0 : S_1 + S_2 $	By $\mathbf{T+Intro}_k$
☞	$ \Gamma \vdash_{\mathbf{T}} \text{inj}_k M_0 : S_1 + S_2 $	By def. of $ - $

• **Case**
$$\frac{\Gamma \vdash_{\mathbf{E}} e_0 \varphi_0 \Rightarrow (S_1 + S_2) \quad \Gamma, x_1 : S_1 \vdash_{\mathbf{E}} e_1 \varphi_1 \Leftarrow S \quad \Gamma, x_2 : S_2 \vdash_{\mathbf{E}} e_2 \varphi_2 \Leftarrow S}{\Gamma \vdash_{\mathbf{E}} \text{case}(e_0, x_1.e_1, x_2.e_2) \top \Leftarrow S} \mathbf{E+Elim}$$

	$\Gamma \vdash_{\mathbf{E}} e_0 \varphi_0 \Rightarrow S_1 + S_2$	Subderivation
	$\Gamma \vdash \text{er}(e_0) \varphi'_0 : (S_1 + S_2) \hookrightarrow M_0$	By i.h.
	$ \Gamma \vdash_{\mathbf{T}} M_0 : S_1 + S_2 $	"
	$ \Gamma \vdash_{\mathbf{T}} M_0 : S_1 + S_2 $	By def. of $ - $
	$\Gamma, x_1 : S_1 \vdash_{\mathbf{E}} e_1 \varphi_1 \Leftarrow S$	Subderivation
	$\Gamma, x_1 : S_1 \vdash \text{er}(e_1) \varphi'_1 : S \hookrightarrow M_1$	By i.h.
	$ \Gamma, x_1 : S_1 \vdash_{\mathbf{T}} M_1 : S $	"
☞	$ \Gamma , x_1 : S_1 \vdash_{\mathbf{T}} M_1 : S $	By def. of $ - $
	$\Gamma, x_2 : S_2 \vdash \text{er}(e_2) \varphi'_2 : S \hookrightarrow M_2$	Similar to above
	$ \Gamma , x_2 : S_2 \vdash_{\mathbf{T}} M_2 : S $	"
☞	$\Gamma \vdash \text{er}(\text{case}(e_0, x_1.e_1, x_2.e_2)) \top : S \hookrightarrow \text{case}(M_0, x_1.M_1, x_2.M_2)$	By <i>elab</i> +Elim
☞	$ \Gamma \vdash_{\mathbf{T}} \text{case}(M_0, x_1.M_1, x_2.M_2) : S $	By $\mathbf{T+Elim}$

• **Case**
$$\frac{\Gamma \vdash_{\mathbf{E}} e \varphi \Leftarrow [(\mu \alpha. S_0)/\alpha] S_0}{\Gamma \vdash_{\mathbf{E}} e \varphi \Leftarrow \mu \alpha. S_0} \mathbf{E}\mu\text{Intro}$$

	$\Gamma \vdash_{\mathbf{E}} e \Leftarrow [(\mu\alpha. S_0)/\alpha] S_0$	Subderivation
	$\Gamma \vdash \text{er}(e) \varphi' : [(\mu\alpha. S_0)/\alpha] S_0 \leftrightarrow M_0$	By i.h.
⊢	$\varphi' \sqsubseteq \varphi$	"
	$ \Gamma \vdash_{\mathbf{T}} M_0 : [(\mu\alpha. S_0)/\alpha] S_0 $	"
⊢	$\Gamma \vdash \text{er}(e) \varphi' : (\mu\alpha. S_0) \leftrightarrow (\text{roll } M_0)$	By <i>elabμIntro</i>
	$ [(\mu\alpha. S_0)/\alpha] S_0 = [\mu\alpha. S_0/\alpha] S_0 $	From def. of $ _ $
	$ \Gamma \vdash_{\mathbf{T}} M_0 : [\mu\alpha. S_0/\alpha] S_0 $	By above equality
	$ \Gamma \vdash_{\mathbf{T}} (\text{roll } M_0) : \mu\alpha. S_0 $	By $\mathbf{T}\mu\text{Intro}$
⊢	$ \Gamma \vdash_{\mathbf{T}} (\text{roll } M_0) : \mu\alpha. S_0 $	By def. of subst.

- **Case** $\frac{\Gamma \vdash_{\mathbf{E}} e \Rightarrow \mu\alpha. S_0}{\Gamma \vdash_{\mathbf{E}} e_{\mathbf{T}} \Rightarrow [(\mu\alpha. S_0)/\alpha] S_0} \mathbf{E}\mu\text{Elim}$

Broadly similar to the $\mathbf{E}\mu\text{Intro}$ case. □

B.6 Consistency

Lemma 11 (Inversion). Given $\cdot \vdash e \varphi : \underbrace{\mathbf{V}\blacktriangleright \dots \mathbf{V}\blacktriangleright}_{0 \text{ or more}} S \leftrightarrow M$:

- (0) If $M = (\lambda x. M_0)$ and $S = (S_1 \rightarrow S_2)$
then $e = (\lambda x. e_0)$ and $\cdot, x : S_1 \vdash e_0 \varphi' : S_2 \leftrightarrow M_0$.
- (1) If $M = (W_1, W_2)$ and $S = (\Delta a. S_0)$
then $\cdot \vdash e \varphi : [V/a]S_0 \leftrightarrow W_1$ and $\cdot \vdash e \varphi : [N/a]S_0 \leftrightarrow W_2$.
- (2) If $M = \text{thunk } M_0$ and $S = \mathbf{N}\blacktriangleright S_0$ then $\cdot \vdash e \varphi' : S_0 \leftrightarrow M_0$.
- (3) If $M = \Lambda _ . M_0$ and $S = (\forall \alpha. S_0)$
then $\cdot, \alpha \text{ type} \vdash e_{\text{val}} : S_0 \leftrightarrow M_0$.
- (4) If $M = (\text{inj}_k W)$ and $S = (S_1 + S_2)$
then $e = (\text{inj}_k e')$ and $\cdot \vdash e' \varphi : S_k \leftrightarrow W$.
- (5) If $M = (\text{roll } W)$ and $S = (\mu\alpha. S_0)$
then $\cdot \vdash e \varphi : [(\mu\alpha. S_0)/\alpha] S_0 \leftrightarrow W$.
- (6) If $M = (W_1, W_2)$ and $S = (S_1 * S_2)$
then $\cdot \vdash e_1 \varphi_1 : S_1 \leftrightarrow W_1$ and $\cdot \vdash e_2 \varphi_2 : S_2 \leftrightarrow W_2$
where $e = (e_1, e_2)$ and $\varphi = \varphi_1 \sqcup \varphi_2$.

Proof. By induction on the given derivation.

For some rules, the proof cases are the same for all parts:

- **Cases $\text{elab}\blacktriangleright\text{Intro}$ (V conclusion), $\text{elab}\blacktriangleright\text{Elim}_V$:**
The result follows by i.h. In the $\text{elab}\blacktriangleright\text{Intro}$ case, we apply the i.h. with one less $\mathbf{V}\blacktriangleright$; in the $\text{elab}\blacktriangleright\text{Elim}_V$ case, we have one more $\mathbf{V}\blacktriangleright$.

For part (0):

- **Case $\text{elab}\rightarrow\text{Intro}$:** The subderivation gives the result.

For part (1):

- **Case $\text{elab}\Delta\text{Intro}$:** The subderivations give the result.

For part (2):

- **Case $\text{elab}\blacktriangleright\text{Intro}$ (N conclusion):** The subderivation gives the result.

For part (3):

- **Case $\text{elab}\forall\text{Intro}$:** The subderivation gives the result.

For part (4):

- **Case $\text{elab}+\text{Intro}_k$:** The subderivation gives the result.

For part (5):

- **Case $elab\mu\text{Intro}$:** The subderivation gives the result.

For part (6):

- **Case $elab*\text{Intro}$:** The subderivations give the result.

All other cases are impossible: either M has the wrong form, or S has the wrong form. □

Lemma 12 (Syntactic values).

If $\Gamma \vdash e_{\text{val}} : S \leftrightarrow W$ and W is N-free then e is a syntactic value.

Proof. By induction on the given derivation.

- **Cases $elab1\text{Intro}$, $elabvar$, $elab\rightarrow\text{Intro}$:** Immediate: the rule requires that e is a syntactic value.
- **Cases $elab\blacktriangleright\text{Elim}_N$, $elabfixvar$, $elabfix$, $elab\rightarrow\text{Elim}$, $elab*\text{Elim}_K$, $elab+\text{Elim}$:**
Impossible: these rules require that val be \top .
- **Case $elab\blacktriangleright\text{Intro}$ (N-conclusion):** Impossible: $\text{thunk } M_0$ is not N-free.
- **Case $elab\mu\text{Elim}$:** Impossible: $\text{unroll } M_0$ is not a value W .
- **Cases $elab\forall\text{Intro}$, $elab\forall\text{Elim}$, $elab\blacktriangleright\text{Intro}$ (V-conclusion), $elab\blacktriangleright\text{Elim}_V$:**
Apply the i.h. to the subderivation.
- **Cases $elab*\text{Intro}$, $elab+\text{Intro}_K$, $elab\mu\text{Intro}$:**
Apply the i.h. to the subderivation(s).
- **Case $elab\sqsupset\text{Intro}$:** Apply the i.h. to the $\Gamma \vdash e_{\text{val}} : [W/a]S_0 \leftrightarrow W_1$ subderivation.
- **Case $elab\sqsupset\text{Elim}$:** Impossible: W must be a projection, but projections are not values. □

Theorem 14 (Consistency).

If $\cdot \vdash e_{\varphi} : S \leftrightarrow M$ and $M \mapsto M'$ then there exists e' such that $e \rightsquigarrow^* e'$ and $\cdot \vdash e'_{\varphi'} : S \leftrightarrow M'$ and $\varphi' \sqsubseteq \varphi$.
Moreover: (1) If $\varphi = \text{val}$ then $e' = e$. (2) If M is N-free then $e \rightsquigarrow^* e'$ can be derived without using SrcStepCtxN .

Proof. By induction on the derivation of $\cdot \vdash e_{\varphi} : S \leftrightarrow M$.

- **Cases $elabvar$, $elabfixvar$:** Impossible, because the typing context is empty.

- **Case**

$$\frac{\cdot, u : S \vdash e_0_{\varphi} : S \leftrightarrow M_0}{\cdot \vdash (\text{fix } u. e_0)_{\top} : S \leftrightarrow (\text{fix } u. M_0)} \text{elabfix}$$

$$\begin{array}{ll} \cdot, u : S \vdash e_0_{\varphi} : S \leftrightarrow M_0 & \text{Subderivation} \\ (\text{fix } u. M_0) \mapsto M' & \text{Given} \\ M' = [(\text{fix } u. M_0)/u]M_0 & \text{By inversion on rule fixReduce} \\ \text{④ } (\text{fix } u. e_0) \rightsquigarrow [(\text{fix } u. e_0)/u]e_0 & \text{By fixVreduce and SrcStepCtxV} \\ \cdot \vdash (\text{fix } u. e_0)_{\top} : S \leftrightarrow (\text{fix } u. M_0) & \text{Given} \\ \cdot, u : S \vdash e_0_{\varphi} : S \leftrightarrow M_0 & \text{Subderivation} \\ \text{④ } \cdot \vdash [(\text{fix } u. e_0)/u]e_0_{\varphi} : S \leftrightarrow [(\text{fix } u. M_0)/u]M_0 & \text{By Lemma 8 (2)} \\ \text{(1) ④ } & \text{(holds vacuously)} \quad \varphi = \top \\ \text{(2) ④ } & \text{Derivation does not use SrcStepCtxN} \end{array}$$

- **Case**

$$\frac{}{\cdot \vdash ()_{\text{val}} : \mathbf{1} \leftrightarrow ()} \text{elab1Intro}$$

Impossible, since $M = ()$ but $() \mapsto M'$ is not derivable.

- **Case**

$$\frac{\cdot, x : S_1 \vdash e_0_{\varphi} : S_2 \leftrightarrow M_0}{\cdot \vdash (\lambda x. e_0)_{\text{val}} : (S_1 \rightarrow S_2) \leftrightarrow \lambda x. M_0} \text{elab}\rightarrow\text{Intro}$$

Impossible, since $M = \lambda x. M_0$ but $(\lambda x. M_0) \mapsto M'$ is not derivable.

- **Case**
 - $\cdot \vdash e_1 \varphi_1: (S_1 \rightarrow S) \leftrightarrow M_1$
 - $\cdot \vdash e_2 \varphi_2: S_1 \leftrightarrow M_2$
 - $\frac{\cdot \vdash (e_1 @ e_2) \top: S \leftrightarrow (M_1 M_2)}{\cdot \vdash (e_1 @ e_2) \top: S \leftrightarrow (M_1 M_2)} \text{elab} \rightarrow \text{Elim}$

First, note that $\varphi = \top$ so “moreover” part (1) is vacuously satisfied.

We have $(M_1 M_2) \mapsto M'$. By inversion on StepContext, $M = (M_1 M_2) = \mathcal{C}[M_0]$ and $M' = \mathcal{C}[M'_0]$. From $(M_1 M_2) = \mathcal{C}[M_0]$ and the definition of \mathcal{C} , either $\mathcal{C} = []$, or $\mathcal{C} = (\mathcal{C}_1 M_2)$, or $\mathcal{C} = (M_1 C_2)$ with M_1 a value.

- If $\mathcal{C} = []$, then $M = M_0$ and $M' = M'_0$. By inversion on β Reduce with $(M_1 M_2) \mapsto_R M'$, we have $M_1 = (\lambda x. M \text{body})$ and $M_2 = W$ and $M' = [W/x]M \text{body}$.

If $M_1 M_2$ is *not* N-free, then:

$$\begin{array}{ll}
\cdot \vdash e_1 \varphi_1: (S_1 \rightarrow S) \leftrightarrow (\lambda x. M \text{body}) & \text{Subderivation} \\
e_1 = (\lambda x. e \text{body}) & \text{By Lemma 11 (0)} \\
\cdot, x: S_1 \vdash e \text{body}_{\varphi''}: S \leftrightarrow M \text{body} & \text{"} \\
\cdot \vdash e_2 \varphi_2: S_1 \leftrightarrow W & \text{Subderivation } (M_2 = W) \\
\Rightarrow \cdot \vdash [e_2/x]e \text{body}_{\varphi'}: S \leftrightarrow [W/x]M \text{body} & \text{By Lemma 8 (1)} \\
\Rightarrow \varphi' \sqsubseteq \varphi & \text{"} \\
(\lambda x. e \text{body}) @ e_2 \rightsquigarrow_{\text{RN}} [e_2/x]e \text{body} & \text{By } \beta \text{Nreduce} \\
\Rightarrow (\lambda x. e \text{body}) @ e_2 \rightsquigarrow^* [e_2/x]e \text{body} & \text{By SrcStepCtxN}
\end{array}$$

If $M_1 M_2$ is N-free, then:

$$\begin{array}{ll}
\cdot \vdash e_1 \varphi_1: (S_1 \rightarrow S) \leftrightarrow (\lambda x. M \text{body}) & \text{Subderivation} \\
e_1 = (\lambda x. e \text{body}) & \text{By Lemma 11 (0)} \\
\cdot, x: S_1 \vdash e \text{body}_{\varphi''}: S \leftrightarrow M \text{body} & \text{"} \\
\cdot \vdash e_2 \varphi_2: S_1 \leftrightarrow W & \text{Subderivation } (M_2 = W) \\
\cdot \vdash e_2 \text{val}: S_1 \leftrightarrow W & \text{By Lemma 5} \\
W \text{ is N-free} & M_1 W \text{ is N-free} \\
\cdot \vdash v \text{val}: S_1 \leftrightarrow W & \text{By Lemma 12} \\
\Rightarrow \cdot \vdash [v/x]e \text{body}_{\varphi'}: S \leftrightarrow [W/x]M \text{body} & \text{By Lemma 8 (1)} \\
\Rightarrow \varphi' \sqsubseteq \varphi & \text{"} \\
(\lambda x. e \text{body}) @ v \rightsquigarrow_{\text{RN}} [v/x]e \text{body} & \text{By } \beta \text{Vreduce} \\
\Rightarrow (\lambda x. e \text{body}) @ v \rightsquigarrow^* [v/x]e \text{body} & \text{By SrcStepCtxV}
\end{array}$$

- If $\mathcal{C} = (\mathcal{C}_1 M_2)$, then:

$$\begin{array}{ll}
M_1 M_2 \mapsto M' & \text{Given} \\
\underbrace{\mathcal{C}_1[M_R]}_{M_1} M_2 \mapsto \underbrace{\mathcal{C}_1[M'_R]}_{M'_1} M_2 & \text{By inversion on rule StepContext} \\
M_R \mapsto_R M'_R & \text{By inversion on rule StepContext} \\
\mathcal{C}_1[M_R] \mapsto \mathcal{C}_1[M'_R] & \text{By StepContext} \\
M_1 \mapsto M'_1 & \text{By known equalities} \\
\cdot \vdash e_1 \varphi_1: (S_1 \rightarrow S) \leftrightarrow M_1 & \text{Subderivation} \\
e_1 \rightsquigarrow^* e'_1 & \text{By i.h.} \\
\cdot \vdash e'_1 \varphi'_1: (S_1 \rightarrow S) \leftrightarrow M'_1 & \text{"} \\
\Rightarrow e_1 @ e_2 \rightsquigarrow^* e'_1 @ e_2 & \text{By SrcStepCtxV} \\
\Rightarrow \cdot \vdash e'_1 @ e_2 \top: S \leftrightarrow M'_1 M_2 & \text{By } \text{elab} \rightarrow \text{Elim}
\end{array}$$

If M is N-free, then M_1 is N-free and the i.h. is sufficient for “moreover” part (2).

- If $\mathcal{C} = (M_1 C_2)$ where M_1 is a value, then we have $M_2 \mapsto M'_2$.

If M is not N-free, then:

$\cdot \vdash e_2 \varphi_2: S_1 \hookrightarrow M_2$	Subderivation
$e_2 \rightsquigarrow^* e'_2$	By i.h.
$\cdot \vdash e'_2 \varphi'_2: S_1 \hookrightarrow M'_2$	"
$\dashv\vdash e_1 @ e_2 \rightsquigarrow^* e_1 @ e'_2$	By SrcStepCtxN
$\dashv\vdash \cdot \vdash e_1 @ e'_2 \top: S \hookrightarrow M_1 M'_2$	By <i>elab</i> →Elim

If M is N-free, then:

$\cdot \vdash e_1 \varphi_1: (S_1 \rightarrow S) \hookrightarrow M_1$	Subderivation
$\cdot \vdash e_1 \text{val}: (S_1 \rightarrow S) \hookrightarrow M_1$	By Lemma 5
$e_1 = v_1$	By Lemma 12
$\cdot \vdash e_2 \varphi_2: S_1 \hookrightarrow M_2$	Subderivation
$e_2 \rightsquigarrow^* e'_2$	By i.h.
$\cdot \vdash e'_2 \varphi'_2: S_1 \hookrightarrow M'_2$	"
$\dashv\vdash v_1 @ e_2 \rightsquigarrow^* v_1 @ e'_2$	By SrcStepCtxV
$\dashv\vdash \cdot \vdash v_1 @ e'_2 \top: (S_1 \rightarrow S) \hookrightarrow M'_1 M_2$	By <i>elab</i> →Elim

- **Case** $\frac{\cdot \vdash e_{\text{val}}: [V/a]S_0 \hookrightarrow M_1 \quad \cdot \vdash e_{\text{val}}: [N/a]S_0 \hookrightarrow M_2}{\cdot \vdash e_{\text{val}}: (\Delta a.S_0) \hookrightarrow (M_1, M_2)}$ *elab*ΔIntro

By inversion on $(M_1, M_2) \mapsto M'$, either $M' = (M'_1, M_2)$ and $M_1 \mapsto M'_1$, or $M' = (M_1, M_2)$ and $M_2 \mapsto M'_2$.
In the first case:

$\cdot \vdash e_{\text{val}}: [V/a]S_0 \hookrightarrow M_1$	Subderivation
$M_1 \mapsto M'_1$	Above
$\cdot \vdash e_{\text{val}}: [V/a]S_0 \hookrightarrow M'_1$	By i.h. ($\varphi = \text{val}$ so $e' = e$)
$\cdot \vdash e_{\text{val}}: [N/a]S_0 \hookrightarrow M_2$	Subderivation
$\dashv\vdash \cdot \vdash e_{\text{val}}: (\Delta a.S_0) \hookrightarrow (M'_1, M_2)$	By <i>elab</i> ΔIntro
$\dashv\vdash \mathcal{D} :: e \rightsquigarrow^* e'$	
(1) $\dashv\vdash e' = e$	Above
(2) $\dashv\vdash \mathcal{D}$ does not use SrcStepCtxN	Zero steps in $e \rightsquigarrow^* e'$

The second case is similar.

- **Case** $\frac{\cdot \vdash e_{\varphi}: (\Delta a.S_0) \hookrightarrow M_0}{\cdot \vdash e_{\varphi}: [V/a]S_0 \hookrightarrow (\text{proj}_1 M_0) \quad \cdot \vdash e_{\varphi}: [N/a]S_0 \hookrightarrow (\text{proj}_2 M_0)}$ *elab*ΔElim

First conclusion:

$(\text{proj}_1 M_0) \mapsto M'$ Given

Either $M' = \text{proj}_1 M'_0$ where $M_0 \mapsto M'_0$, or $M' = W_1$ and $M_0 = (W_1, W_2)$.

▪ In the first case:

$\cdot \vdash e_{\varphi}: (\Delta a.S_0) \hookrightarrow M_0$	Subderivation
$M_0 \mapsto M'_0$	Above
$\cdot \vdash e'_{\varphi}: (\Delta a.S_0) \hookrightarrow M'_0$	By i.h.
$\dashv\vdash \mathcal{D} :: e \rightsquigarrow^* e'$	"
(1) $\dashv\vdash$ If $\varphi = \text{val}$ then $e = e'$	"
If M_0 is N-free then \mathcal{D} does not use SrcStepCtxN	"
(2) $\dashv\vdash$ If $(\text{proj}_1 M_0)$ is N-free then \mathcal{D} does not use SrcStepCtxN	Definition of N-free
$\dashv\vdash \cdot \vdash e'_{\varphi}: [V/a]S_0 \hookrightarrow M'_0$	By <i>elab</i> ΔElim

▪ In the second case:

	$\cdot \vdash e_\varphi: (\Delta a. S_0) \leftrightarrow (W_1, W_2)$	Subderivation
☞	$\cdot \vdash e_\varphi: [V/a]S_0 \leftrightarrow W_1$	By Lemma 11 (1)
	$\text{proj}_1 (W_1, W_2) \mapsto W_1$	Given
(1)☞	Let $e' = e.$	
☞	$\mathcal{D} :: e \rightsquigarrow^* e'$	$e' = e$
(2)☞	\mathcal{D} does not use SrcStepCtxN	Zero steps in $e \rightsquigarrow^* e'$

Second conclusion:

Either $M' = \text{proj}_2 M'_0$ where $M_0 \mapsto M'_0$, or $M' = W_1$ and $M_0 = (W_1, W_2)$.

- In the first case: similar to the first subcase of the $[V/a]$ part above.
- In the second case: similar to the second subcase of the $[V/a]$ part above.

• **Case** $\cdot, \alpha \vdash e_{\text{val}}: S \leftrightarrow M$
 $\frac{\cdot, \alpha \vdash e_{\text{val}}: S \leftrightarrow M}{\cdot \vdash e_{\text{val}}: \forall \alpha. S \leftrightarrow \Lambda _ . M} \text{elab}\forall\text{Intro}$

This case is impossible, because $(\Lambda _ . M) \mapsto M'$ is not derivable.

• **Case** $\cdot \vdash e_\varphi: \forall \alpha. S_0 \leftrightarrow M_0$ $\cdot \vdash S' \text{ type}$
 $\frac{\cdot \vdash e_\varphi: \forall \alpha. S_0 \leftrightarrow M_0 \quad \cdot \vdash S' \text{ type}}{\cdot \vdash e_\varphi: [S'/\alpha]S_0 \leftrightarrow M_0[_]} \text{elab}\forall\text{Elim}$
 $(M_0[_]) \mapsto M'$ Given
 $M_0 = (\Lambda _ . M')$ By inversion
 $\cdot \vdash e_\varphi: \forall \alpha. S_0 \leftrightarrow M_0$ Subderivation
 $\cdot \vdash e_\varphi: \forall \alpha. S_0 \leftrightarrow (\Lambda _ . M')$ By above equality
 $\cdot, \alpha \text{ type} \vdash e_\varphi: S_0 \leftrightarrow M'$ By Lemma 11 (3)
☞ $\cdot \vdash e_\varphi: [S'/\alpha]S_0 \leftrightarrow M'$ By Lemma 22
☞ $e \rightsquigarrow^* e$ Zero steps

“Moreover” parts (1) and (2) are immediately satisfied, because $e' = e$.

• **Case** $\cdot \vdash e_\varphi: S_0 \leftrightarrow M_0$
 $\frac{\cdot \vdash e_\varphi: S_0 \leftrightarrow M_0}{\cdot \vdash e_\varphi: V \blacktriangleright S_0 \leftrightarrow M_0} \text{elab}\blacktriangleright\text{Intro}$
 $\cdot \vdash e_{\text{val}}: N \blacktriangleright S_0 \leftrightarrow \text{thunk } M_0$

The second conclusion is not possible, because $(\text{thunk } M_0) \mapsto M'$ is not derivable.

For the first conclusion: We have $M_0 = M$.

	$\cdot \vdash e_\varphi: S_0 \leftrightarrow M$	Subderivation
☞	$\mathcal{D} :: e \rightsquigarrow^* e'$	By i.h.
	$\cdot \vdash e'_{\varphi'}: S_0 \leftrightarrow M'$	"
☞	$\varphi' \sqsubseteq \varphi$	"
(1)☞	If $\varphi = \text{val}$ then $e = e'$	"
(2)☞	If M is N-free then \mathcal{D} does not use SrcStepCtxN	"
☞	$\cdot \vdash e'_{\varphi'}: V \blacktriangleright S_0 \leftrightarrow M'$	By $\text{elab}\blacktriangleright\text{Intro}$

• **Case** $\cdot \vdash e_\varphi: V \blacktriangleright S \leftrightarrow M$
 $\frac{\cdot \vdash e_\varphi: V \blacktriangleright S \leftrightarrow M}{\cdot \vdash e_\varphi: S \leftrightarrow M} \text{elab}\blacktriangleright\text{Elim}_V$

By i.h. and $\text{elab}\blacktriangleright\text{Elim}_V$.

• **Case** $\cdot \vdash e_{\varphi_0}: N \blacktriangleright S \leftrightarrow M_0$
 $\frac{\cdot \vdash e_{\varphi_0}: N \blacktriangleright S \leftrightarrow M_0}{\cdot \vdash e_\top: S \leftrightarrow (\text{force } M_0)} \text{elab}\blacktriangleright\text{Elim}_N$

We have $(\text{force } M_0) \mapsto M'$. If $M_0 \mapsto M'_0$, use the i.h. and then apply $\text{elab}\blacktriangleright\text{Elim}_N$. Otherwise, $M_0 = \text{thunk } M'$.

	$\cdot \vdash e_{\varphi_0}: N \blacktriangleright S \leftrightarrow \text{thunk } M'$	Subderivation
☞	$\cdot \vdash e_{\varphi'_0}: S \leftrightarrow M'$	By Lemma 11 (2)
☞	$\varphi'_0 \sqsubseteq \top$	By def. of \sqsubseteq
☞	$e \rightsquigarrow^* e$	Zero steps
(1)☞	(holds vacuously)	$\varphi = \top$
(2)☞	Derivation does not use SrcStepCtxN	Zero steps

$$\bullet \text{ Case } \frac{\cdot \vdash e_1 \varphi: S_1 \leftrightarrow M_1 \quad \cdot \vdash e_2 \varphi: S_2 \leftrightarrow M_2}{\cdot \vdash (e_1, e_2) \varphi: (S_1 * S_2) \leftrightarrow (M_1, M_2)} \text{elab*Intro}$$

Apply the i.h. to the appropriate subderivation, then apply *elab*Intro* and *SrcStepCtxV*.

“Moreover” part (1):

If $\varphi = \text{val}$, the i.h. shows that $e'_1 = e_1$ (or $e'_2 = e_2$ if $M_2 \mapsto M'_2$); thus, $(e'_1, e_2) = (e_1, e_2)$ (or $(e_1, e'_2) = (e_1, e_2)$).

“Moreover” part (2):

If (M_1, M_2) is N-free, then M_1 and M_2 are N-free, and the i.h. shows that $\mathcal{D}_0 :: e_k \rightsquigarrow^* e'_k$ does not use *SrcStepCtxN*. Therefore $(e_1, e_2) \rightsquigarrow^* \dots$ does not use *SrcStepCtxN*.

$$\bullet \text{ Case } \frac{\cdot \vdash e_0 \varphi_0: (S_1 * S_2) \leftrightarrow M_0}{\cdot \vdash (\text{proj}_k e_0) \top: S_k \leftrightarrow (\text{proj}_k M_0)} \text{elab*Elim}_k$$

We have $(\text{proj}_k M_0) \mapsto M'$.

If $M_0 \mapsto M'_0$ then use the i.h. and apply *elab*Elim_k*.

Otherwise, $M_0 = (W_1, W_2)$ and $M' = W_k$.

- If M is *not* N-free, we can use *projNreduce*:

$$\begin{array}{l} \text{⊢} \\ \text{⊢ } e_k \varphi_k: S_k \leftrightarrow W_k \quad \text{By Lemma 11 (6)} \\ e_0 = (e_1, e_2) \quad \text{"} \end{array}$$

$$\text{⊢ } \text{proj}_k (e_1, e_2) \rightsquigarrow e_k \quad \text{By projNreduce}$$

“Moreover” part (2): M is not N-free.

- If M is N-free, we have the obligation not to use *projNreduce*.

$$\begin{array}{l} \cdot \vdash e_0 \varphi_0: (S_1 * S_2) \leftrightarrow (W_1, W_2) \quad \text{Subderivation} \\ \cdot \vdash e_0 \text{val}: (S_1 * S_2) \leftrightarrow (W_1, W_2) \quad \text{By Lemma 5} \\ \cdot \vdash v \text{val}: (S_1 * S_2) \leftrightarrow (W_1, W_2) \quad \text{By Lemma 12} \\ \cdot \vdash (v_1, v_2) \text{val}: (S_1 * S_2) \leftrightarrow (W_1, W_2) \quad \text{By Lemma 11 (6)} \\ \text{⊢ } \cdot \vdash v_k \text{val}: S_k \leftrightarrow W_k \quad \text{"} \end{array}$$

$$\text{⊢ } \text{proj}_k (v_1, v_2) \rightsquigarrow^* v_k \quad \text{By projVreduce and SrcStepCtxV}$$

“Moreover” part (2): we did not use *SrcStepCtxN*.

“Moreover” part (1): $\varphi = \top$.

$$\bullet \text{ Case } \frac{\cdot \vdash e_0 \varphi: S_k \leftrightarrow M_0}{\cdot \vdash (\text{inj}_k e_0) \varphi: (S_1 + S_2) \leftrightarrow (\text{inj}_k M_0)} \text{elab+Intro}_k$$

$$\begin{array}{l} (\text{inj}_k M_0) \mapsto M' \quad \text{Given} \\ M' = (\text{inj}_k M'_0) \text{ and } M_0 \mapsto M'_0 \quad \text{By inversion} \\ \cdot \vdash e_0 \varphi: S_k \leftrightarrow M_0 \quad \text{Subderivation} \\ \cdot \vdash e'_0 \varphi': S_k \leftrightarrow M'_0 \quad \text{By i.h.} \\ \text{⊢ } \varphi' \sqsubseteq \varphi \quad \text{"} \\ e_0 \rightsquigarrow^* e'_0 \quad \text{"} \\ \text{⊢ } (\text{inj}_k e_0) \rightsquigarrow^* (\text{inj}_k e'_0) \\ \text{⊢ } \cdot \vdash (\text{inj}_k e'_0) \varphi': (S_1 + S_2) \leftrightarrow (\text{inj}_k M'_0) \quad \text{By elab+Intro}_k \end{array}$$

“Moreover” part (1) follows from the i.h.

“Moreover” part (2) follows from the i.h.: If $\text{inj}_k M_0$ is N-free, then M_0 is N-free; if $e_0 \rightsquigarrow^* e'_0$ does not use *SrcStepCtxN*, we can derive $(\text{inj}_k e_0) \rightsquigarrow^* (\text{inj}_k e'_0)$ without *SrcStepCtxN*.

$$\bullet \text{ Case } \frac{\cdot, x_1: S_1 \vdash e_1 \varphi_1: S \leftrightarrow M_1 \quad \cdot, x_2: S_2 \vdash e_2 \varphi_2: S \leftrightarrow M_2}{\cdot \vdash e_0 \varphi_0: (S_1 + S_2) \leftrightarrow M_0 \quad \cdot, x_2: S_2 \vdash e_2 \varphi_2: S \leftrightarrow M_2} \text{elab+Elim}$$

$$\cdot \vdash \text{case}(e_0, x_1.e_1, x_2.e_2) \top: S \leftrightarrow \text{case}(M_0, x_1.M_1, x_2.M_2)$$

First note that “Moreover” part (1) is vacuously satisfied, since $\varphi = \top$.

We have $\text{case}(M_0, x_1.M_1, x_2.M_2) \mapsto M'$. Either (1) $M_0 \mapsto M'_0$ and $M' = \text{case}(M'_0, x_1.M_1, x_2.M_2)$ or (2) $M_0 = (\text{inj}_k W)$ and $M' = [W/x_k]M_k$.

For (1), apply the i.h. to $\cdot \vdash e_0 \varphi: (S_1 + S_2) \leftrightarrow M_0$ and apply *elab+Elim*. “Moreover” part (2) follows from the i.h.

For (2) if M is *not* N-free, we can use *SrcStepCtxN*:

$\cdot \vdash e_0 \varphi_0: (S_1 + S_2) \leftrightarrow (\text{inj}_k W)$	Subderivation
$e_0 = \text{inj}_k e'_0$	By Lemma 11 (4)
$\cdot \vdash e'_0 \varphi'_0: S_k \leftrightarrow W$	"
$\cdot, x_k: S_k \vdash e_k \varphi_k: S \leftrightarrow M_k$	Subderivation
$\cdot \vdash [e'_0/x_k]e_k \varphi'_k: S \leftrightarrow [W/x_k]M_k$	By Lemma 8 (1)
$e_0 = \text{inj}_k e'_0$	Above
$\text{case}(\text{inj}_k e'_0, x_1.e_1, x_2.e_2) \rightsquigarrow_{\text{RN}} [e'_0/x_k]e_k$	By caseNreduce
$\text{case}(e_0, x_1.e_1, x_2.e_2) \rightsquigarrow^* [e'_0/x_k]e_k$	By SrcStepCtxN

For (2) if M is N-free, we can show $\cdot \vdash [e'_0/x_k]e_k \varphi'_k: S \leftrightarrow [W/x_k]M_k$ as in the case when M is not N-free, but we have an obligation ("Moreover" part (2)) not to use caseNreduce.

$\cdot \vdash e_0 \varphi_0: (S_1 + S_2) \leftrightarrow \text{inj}_k W$	Subderivation
$\cdot \vdash e_0 \text{val}: (S_1 + S_2) \leftrightarrow \text{inj}_k W$	By Lemma 5
$e_0 = v$	By Lemma 12
$\cdot \vdash v \text{val}: (S_1 + S_2) \leftrightarrow \text{inj}_k W$	By above equality
$v_0 = \text{inj}_k v'_0$	By Lemma 11 (4)
$e \rightsquigarrow [v'_0/x]e_k$	By caseVreduce and SrcStepCtxV

• **Case** $\frac{\cdot \vdash e \varphi: [(\mu\alpha. S_0)/\alpha]S_0 \leftrightarrow M_0}{\cdot \vdash e \varphi: \mu\alpha. S_0 \leftrightarrow (\text{roll } M_0)}$ *elabμIntro*

By inversion, $M_0 \mapsto M'_0$ and $M' = (\text{roll } M'_0)$.

$\cdot \vdash e \varphi: [(\mu\alpha. S_0)/\alpha]S_0 \leftrightarrow M_0$	Subderivation
$\cdot \vdash e' \varphi: [(\mu\alpha. S_0)/\alpha]S_0 \leftrightarrow M'_0$	By i.h.
$e \rightsquigarrow^* e'$	"
$\cdot \vdash e' \varphi: \mu\alpha. S_0 \leftrightarrow (\text{roll } M'_0)$	By <i>elabμIntro</i>

"Moreover" parts (1) and (2) follow from the i.h.

• **Case** $\frac{\cdot \vdash e \varphi_0: \mu\alpha. S_0 \leftrightarrow M_0}{\cdot \vdash e \tau: [(\mu\alpha. S_0)/\alpha]S_0 \leftrightarrow (\text{unroll } M_0)}$ *elabμElim*

We have $(\text{unroll } M_0) \mapsto M'$. Either (1) $M' = (\text{unroll } M'_0)$ and $M_0 \mapsto M'_0$ or (2) $M_0 = (\text{roll } W)$ and $M' = W$.

If (1), similar to the *elabμIntro* case.

If (2):

$\cdot \vdash e \varphi_0: \mu\alpha. S_0 \leftrightarrow (\text{roll } W)$	Subderivation
$\cdot \vdash e' \varphi': [(\mu\alpha. S_0)/\alpha]S_0 \leftrightarrow W$	By Lemma 11 (5)
$e \rightsquigarrow^* e'$	"

"Moreover" part (1) is vacuously satisfied; part (2) follows from the i.h. □

Theorem 15 (Multi-step consistency).

If $\cdot \vdash e \varphi: S \leftrightarrow M$ and $M \mapsto^* W$ then there exists e' such that $e \rightsquigarrow^* e'$ and $\cdot \vdash e' \text{val}: S \leftrightarrow W$. Moreover, if M is N-free then we can derive $e \rightsquigarrow^* e'$ without using SrcStepCtxN.

Proof. By induction on the derivation of $M \mapsto^* W$.

If $M = W$ then let e' be e . By Lemma 5, $\cdot \vdash e' \text{val}: S \leftrightarrow W$. The source expression e steps to itself in zero steps, so $e \rightsquigarrow^* e$, i.e. $e \rightsquigarrow^* e'$. We did not use SrcStepCtxN.

Otherwise, we have $M \mapsto M'$ and $M' \mapsto^* W$ for some M' . By Theorem 14, $\cdot \vdash e_1 \varphi: S \leftrightarrow M'$, where $e \rightsquigarrow^* e_1$; also, if M is N-free, then Theorem 14 showed that we did not use SrcStepCtxN. If M is N-free, then by Lemma 13, M' is N-free. By i.h., there exists e' such that $e_1 \rightsquigarrow^* e'$ and $\cdot \vdash e' \text{val}: S \leftrightarrow W$. It follows that $e \rightsquigarrow^* e'$. □

If a source type, economical typing judgment, or target term is not N-free, we say it is *N-tainted*.

Lemma 16. If $\Gamma \vdash_{\mathbf{E}} e \varphi \Rightarrow S$ and S is *not* N-free then it is not the case that both Γ and e are N-free.

Proof. By induction on the given derivation.

• **Case E∀Elim:** If S' is not N-free, then $e = e_0[S']$ is not N-free. Otherwise, we have that $S = [S'/\alpha]S_0$ is not N-free; since S' is N-free, S_0 must not be N-free, which lets us apply the i.h., giving the result.

- Cases **E Δ Elim**, **E \blacktriangleright Elim \vee** , **E \blacktriangleright Elim ϵ** : The i.h. gives the result.
- Cases **Evar**, **Efixvar**: The type S appears in Γ , so Γ is N-tainted.
- Case **Eanno**: The type S appears in $e = (e_0 : S)$, so e is N-tainted.
- Case **E \rightarrow Elim**: If S is N-tainted then $S_1 \rightarrow S$ is N-tainted, and the result follows by i.h.
- Cases **E*Elim**, **E μ Elim**: Similar to the **E \rightarrow Elim** case. □

Theorem 17 (Economizing preserves N-freeness).

If $\gamma \vdash_{\mathbf{I}} e \Leftarrow \tau$ (resp. \Rightarrow) where the judgment is N-free (Definition 1 (2)) then $[\Gamma] \vdash_{\mathbf{E}} [e] \Leftarrow [\tau]$ (resp. \Rightarrow) where this judgment is N-free (Definition 2 (2)).

Proof. By induction on the given derivation. We can simply follow the proof of Theorem 1, observing that if the given impartial judgment is N-free, the resulting economical judgment is N-free. For example, in the **I \rightarrow Intro** case, we have $\tau = (\tau_1 \xrightarrow{\epsilon} \tau_2)$. Since we know that τ is N-free, $\epsilon = \vee$, so the translation of τ is $(\vee \blacktriangleright [\tau_1]) \rightarrow [\tau_2]$, which is N-free. Note that Definition 1 (2)(b) bars $\chi \tau \Rightarrow \tau$ declarations—which would result in $\chi : \mathbf{N} \blacktriangleright \dots$ —from γ . □

Theorem 18 (Elaboration preserves N-freeness).

If $\Gamma \vdash_{\mathbf{E}} e \Leftarrow S$ (or \Rightarrow) where the judgment is N-free (Definition 2 (2)) then $\Gamma \vdash \text{er}(e) \Leftarrow S \leftrightarrow M$ such that M is N-free.

Proof. By induction on the given derivation.

- Case **E1Intro**: Apply *elab1Intro*.
- Case **E Δ Intro**: Impossible: $S = \Delta a. S_0$, which is not N-free (Definition 2 (1)(ii)).
- Case **E Δ Elim**:
We have $\Gamma \vdash_{\mathbf{E}} e \Leftarrow \Delta a. S_0$, where $S = [S'/a]S_0$.
By Definition 2 (1)(ii), the type $\Delta a. S_0$ is N-tainted. So, by Lemma 16, at least one of Γ and e is N-tainted. But it was given that the judgment $\Gamma \vdash_{\mathbf{E}} e \Leftarrow S$ is N-free, which means that Γ and e are N-free. We have a contradiction: this case is impossible.
- Case **E \blacktriangleright Intro** (first conclusion): Use the i.h. and apply rule *elab \blacktriangleright Intro* (first conclusion).
- Case **E \blacktriangleright Intro** (second conclusion): Impossible: $S = \mathbf{N} \blacktriangleright S_0$, which is not N-free.
- Case **E \blacktriangleright Elim \vee** : Use the i.h. and apply rule *elab \blacktriangleright Elim \vee* .
- Case **E \blacktriangleright Elim ϵ** :
We have $\Gamma \vdash_{\mathbf{E}} e \Leftarrow \epsilon \blacktriangleright S$.
If $\epsilon = \vee$ then use the i.h., apply rule *elab \blacktriangleright Elim \vee* .
Otherwise, $\epsilon \blacktriangleright S$ is not N-free. As in the **E Δ Elim** case, we can use Lemma 16 to reach a contradiction.
- Cases **Evar**, **Efixvar**, **Efix**, **E \forall Intro**, **E \forall Elim**, **E \rightarrow Intro**, **E \rightarrow Elim**, **E*Elim**, **E+Intro κ** , **E+Elim**, **E μ Intro**, **E μ Elim**:
Use the i.h. on all subderivations (if any) and apply the corresponding elaboration rule, e.g. in the **Efix** case, apply *elabfix*.
- Cases **Esub**, **Eanno**: Use the i.h.
- Case **E*Intro**: Use the i.h. on each subderivation, and apply *elab*Intro*. □