Elaborating Evaluation-Order Polymorphism

Jana Dunfield
University of British Columbia
Vancouver, Canada
jdl69@queensu.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 F.3.3 [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 $\lambda$-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 (\(\times^v, \times^n\)), product types (\(\langle^v, \langle^n\rangle\)), sum types (\(\langle^v, \langle^n\rangle\rangle\)) and recursive types (\(\mu^v, \mu^n\)). Using bidirectional typing, which distinguishes checking and inference, we can use information found in the types of functions to determine whether an unmarked $\lambda$ 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 $\bigwedge$ as a universal quantifier over evaluation orders. For example, the type $\forall a. \text{int} \rightarrow a$ int corresponds to (int $\rightarrow$ int) $\bigwedge$ (int $\rightarrow$ int). Thus, we can type code that is generic over evaluation orders. Datatype defini-
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 $\Delta a,\tau$ 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 $\triangleright S$.

(§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 $e ::= (\, | x \mid u \mid \lambda x. e \mid e_1 \odot e_2 \mid \text{fix } u. e \mid (\text{\textbackslash} \lambda x. e \mid e_1 \odot e_2 | \text{proj}_x e | \text{inj}_x e | \text{case}(e, x_1, e_1, x_2, e_2)$ |

| Evaluation order vars. $a$

| Evaluation orders $e ::= V \mid N \mid a$

| Type variables $\alpha$

| Valueneuses $\varphi ::= \text{val } | T$

| Source types $\tau ::= 1 \mid \alpha \mid \forall \alpha. \tau \mid \Delta a. \tau \mid \tau_1 \triangleright \tau_2$

| Source typing contexts $\gamma ::= \cdot \mid \gamma, x : \varphi \Rightarrow \tau \mid \gamma, u : \tau \Rightarrow \tau$

| 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 $e_1 \odot e_2$, fixed points $\text{fix } u. e$ with fixed point variables $u$, pairs and projections,
and sums inj\_k e with conditionals case\{e, x_1.e_1, x_2.e_2\} (short- hand for case e of inj\_1 x_1 \Rightarrow e_1 | 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 \((\tau \rightarrow \rho),\) the variable x should be considered a value (it will be replaced with a value at run time), but when checking against \((\rho \rightarrow \tau),\) 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 \(\lambda\)-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 →introduction rule

\[
\gamma, x : \tau_1 \vdash e : \tau_2 \quad \gamma \vdash \lambda x.e : (\tau_1 \rightarrow \tau_2)
\]

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

\[
\gamma, x : \tau_1 \vdash e : \tau_2 \quad \gamma \vdash \lambda x.e : (\tau_1 \rightarrow \tau_2)
\]

Products \(\ast\), sums \(+\), and recursive types \(\mu\) follow similarly.

We add a universal quantifier \(\forall\_a,\tau\) over evaluation orders.\(^2\) Its rules follow the usual type-assignment rules for \(\forall\): the introduction rule is parametric over an arbitrary evaluation order \(a\), and the elimination rule replaces \(\alpha\) with a particular evaluation order \(\epsilon\):

\[
\gamma, a \text{ evalorder } \vdash e : \tau \quad \gamma \vdash e : \forall\_a,\tau \quad \gamma \vdash e \text{ evalorder}
\]

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 \(\lambda\)-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 \(f \Theta e_2\) diverges depends on whether the type of \(f\) has \(V\) or \(N\). The above rules allow a compiler to make either choice. Polymorphism in the form of \(\forall\) aggravates the problem: it is tempting to infer for \(f\) the principal type \(\forall\_a, \ldots \rightarrow \ldots\); 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 \(\forall\_a\).

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 \(\text{check}\)ing and \(\text{synthesi}\)zation judgments. In the checking judgment \(e \in \tau\), we already know that \(e\) should have type \(\tau\), and are checking that \(e\) is consistent with this knowledge. In the synthesis judgment \(e \Rightarrow \tau\), we extract \(\tau\) 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 Pfening 2000; Dunfield and Pfening 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 Pfening (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 \(\forall\) or \(\forall\_a\) evaluation order can be introduced \textit{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 \((\lambda x.e) \Theta e_2\), the \(\lambda\) needs an annotation, because \(\lambda x.e\) is an introduction form in an elimination position: \([.\ldots]\) \(\Theta e_2\). In contrast, \(f \Theta (\lambda x.e_2)\) 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.

**Valuelessness.** Whether an expression is a value may depend on typing, so we put a valuelessness in the typing judgments: \( e : \vartheta \vdash S \) (or \( e : \vartheta \iff S \)) means that \( e \) at type \( S \) is definitely a value, while \( e : \tau \Rightarrow S \) (or \( e : \tau \iff S \)) means that \( e \) at type \( S \) is not known to be a value. In the style of abstract interpretation, we have a partial order \( \varrho \) such that \( \vartheta \subseteq \tau \). Then the \( \text{join} \) \( q_1 \sqcup q_2 \) is \( \vartheta \) when \( q_1 = q_2 = \vartheta \), and \( \top \) otherwise. \( \gamma \) Since valuelessness is just a projection of \( e \), we could formulate the system without it, using \( e \) to mark judgments as denoting values (\( \vartheta \)) or possible nonvalues (\( \varrho \)). But that seems prone to confusion: \( e \iff \) saying the expression is “by name” in some sense?

**Types and typing contexts.** In Figure 3 we show the grammar for evaluation orders \( e \), which are either by-value (\( \vartheta \)), by-name (\( \varrho \)), or an evaluation order variable \( a \). We have the unit type \( I \), type variables \( \alpha \), ordinary parametric polymorphism \( \forall \alpha . \tau \), evaluation order polymorphism \( \text{fix} \vartheta \tau \), functions \( \tau_1 \twoheadrightarrow \tau_2 \), products \( \tau_1 \times^* \tau_2 \), sums \( \tau_1 \oplus^* \tau_2 \), and recursive types \( \mu \alpha . \tau \). A source typing context \( \gamma \) consists of variable declarations \( x : \tau \Rightarrow \gamma \) denoting that \( x \) has type \( \tau \) with valuelessness \( \varrho \), fixed-point variable declarations \( \gamma \Rightarrow \tau \) (fixed-point variables are never values), evaluation-order variable declarations a evalorder, and type variable declarations \( \alpha \tau \).

**Impartial typing judgments.** Figure 4 shows the bidirectional rules for impartial typing. The judgment forms are \( \gamma \vdash_{\vartheta} \epsilon \varrho \Rightarrow \gamma \), meaning that \( e \) checks against \( \tau \) (with valuelessness \( \varrho \)), and \( \gamma \vdash_{\varrho} \epsilon \vartheta \Rightarrow \tau \), meaning that \( e \) synthesizes type \( \tau \). The “I” on the turnstile stands for “impartial.”

**Connective-independent rules.** Rules Ivar and Ifixvar simply use assumptions stored in \( \gamma \). Rule Fix checks a fixed point \( \text{fix} \ u \ . \ e \) against type \( \tau \) by introducing the assumption \( \gamma \Rightarrow \tau \) and checking \( e \) against \( \tau \); its premise has valuelessness \( \varrho \) because even if \( e \) is a value, \( \text{fix} \ u \ . \ e \) is not \( \top \) in the conclusion. Rule Isub says that if \( e \) synthesizes \( \gamma \) then \( e \) checks against \( \gamma \). For example, in the ill-advised fixed point expression \( \text{fix} \ u \ . \ u \), the premise of Fix tries to check \( u \) against \( \gamma \), but Ifixvar derives a synthesis judgment, not a checking judgment; Isub 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 \( \gamma \), then \( e : \gamma \) synthesizes \( \gamma \).

**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-Intro follows the recipe, despite having a checking judgment in its conclusion: the connective being eliminated, \( + \gamma \), is synthesized (in the first premise).

**Functions.** Rule I-Intro introduces the type \( \tau_1 \twoheadrightarrow \tau_2 \). Its premise adds an assumption \( x : \text{valuelessness}(e) \Rightarrow \tau_1 \), where \( \text{valuelessness}(e) \) is \( \vartheta \) if \( e : \vartheta \), and \( \top \) if \( e : \top \) or \( e \) is an evaluation-order variable \( a \). This rule thereby encompasses both variables that will be substituted with values (\( \text{valuelessness}(e) = \vartheta \)) and variables that might be substituted with non-values (\( \text{valuelessness}(e) = \top \)). Applying a function of type \( \tau_1 \twoheadrightarrow \tau_2 \) yields something of type \( \tau_2 \) regardless of \( e \), so I-Intro eliminates ignores \( e \).

Consistent with the usual definition of syntactic values, \( \text{I-Intro}'s \) conclusion has \( \vartheta \), while I-\text{Elim}'s conclusion has \( \top \).

In rule I-\text{Elim}, the first premise has the connective to eliminate, so the first premise synthesizes \( (\tau_1 \twoheadrightarrow \tau_2) \). This provides the type \( \tau_1 \), so the second premise is a checking judgment; it also provides \( \tau_2 \), so the conclusion synthesizes.

**Products.** Rule I-Intro \( e \) types a value if and only if both \( e_1 \) and \( e_2 \) are typed as values, so its conclusion has valuelessness \( \vartheta \sqcup \varrho \).

**Sums.** Rule I-\text{Intro}_a is straightforward. In rule I-\text{Elim}, the assumptions added to \( \gamma \) 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 IUnintro and IUnelim 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 IUnintro (since its conclusion is a checking judgment). Knowing when to apply IUnelim is more subtle: we should try to apply it whenever we need to synthesize some other type connective. For instance, the first premise of IUnelim needs \( a \), so if we synthesize a \( \mu \)-type we should apply IUnelim in the hope of exposing a \( a \).

The lack of explicit \( [\text{un} ] \text{rolls} \) suggests that these are not iso-reductive 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 \( \mu \) is guarded by a type connective that does have explicit constructs, to rule out types like \( \mu x . \alpha \), 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 \( \Lambda \alpha . \epsilon \) and \( M \epsilon \). This guarantees that a \( \lambda \) 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 \( \gamma \vdash_{\varrho} \epsilon_1 \vartheta \Rightarrow \Delta \alpha . \tau_1 \Rightarrow \tau_2 \). Then, for any \( e \), we can derive \( \gamma \vdash_{\varrho} \epsilon_1 \tau_2 \Rightarrow \epsilon_2 \tau_1 \). But we can’t use IUnintro to derive the type \( \Delta \alpha . (\epsilon_1 \tau_2 \tau_1 \epsilon_2 \tau_2) \), because \( e_1 \tau_2 \). The only sense in which this expression has a principal type is if we have an evaluation-order variable in \( \gamma \) that we can substitute for \( a \).

### 2.3 Programming with Polymorphic Evaluation Order

#### Lists and streams.

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

\[
\text{type List} \; a \; \alpha = \mu \beta . \; \{ I + \gamma (\alpha + \beta) \}
\]

Choosing \( a = V \) yields \( \mu V \)(\( I + V (\alpha + V) \)), which is the type of lists of elements \( \alpha \). Choosing \( a = N \) yields \( \mu N \beta . \; \{ I + N (\alpha + \beta) \} \), 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 List \( a \) that work for lists and streams:

\[
\begin{align*}
\text{map} : \Delta \alpha \times \epsilon (\alpha + \beta) \times (\epsilon (\text{List} \; a \; \alpha) \Rightarrow (\text{List} \; a \; \beta)) &= \Lambda \alpha \times \epsilon \text{map} \; \text{Af} \times \text{Lx} \\
\text{case} : \{ \epsilon x \times \epsilon x \times \epsilon x \times \epsilon x \times \epsilon \} \\
&= \epsilon \times \text{x \times x \times x} (f @ \text{proj} \times x \times x) \Rightarrow \epsilon \times \text{map} \; f @ \text{proj} \times x \times x
\end{align*}
\]

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

| $\text{evaluation order } e \text{ maps to valuation } \varphi$ |
|---------------------------------------------------------------|
| $\Gamma \vdash_1 e \varphi \iff \tau$ Source expression $e$ checks against impartial type $\tau$ |
| $\Gamma \vdash_1 e \varphi \Rightarrow \tau$ Source expression $e$ synthesizes impartial type $\tau$ |

$$
\begin{aligned}
\text{value}(e) &= \varphi \\
\text{evaluation order } e \text{ maps to valuation } \varphi \\
\Gamma \vdash_1 e \varphi \iff \tau \\
\Gamma \vdash_1 e \varphi \Rightarrow \tau \\
\end{aligned}
$$

| $\text{value}(V) = \text{val}$ |
|-------------------------------|
| $\text{value}(N) = \text{T}$ |
| $\text{value}(a) = \text{T}$ |

### Figure 4. Impartial bidirectional typing for the source language

| $\text{value}(\epsilon) = \varphi$ |
|-------------------------------|
| $\text{evaluation order } e \text{ maps to valuation } \varphi$ |
| $\Gamma \vdash_1 e \varphi \iff \tau$ Source expression $e$ checks against impartial type $\tau$ |
| $\Gamma \vdash_1 e \varphi \Rightarrow \tau$ Source expression $e$ synthesizes impartial type $\tau$ |

$$
\begin{aligned}
\forall \gamma, \alpha. \text{type} \vdash_1 \nu \varphi \iff \tau \\
\forall \gamma, \alpha. \text{type} \vdash_1 \lambda \varphi \iff \nu \varphi \iff \tau \\
\forall \gamma, \alpha. \text{type} \vdash_1 \exists \varphi \iff \tau \\
\forall \gamma, \alpha. \text{type} \vdash_1 \forall \varphi \iff \tau \\
\forall \gamma, \alpha. \text{type} \vdash_1 \exists \varphi \iff \tau \\
\forall \gamma, \alpha. \text{type} \vdash_1 \forall \varphi \iff \tau \\
\forall \gamma, \alpha. \text{type} \vdash_1 \nu \varphi \iff \tau \\
\forall \gamma, \alpha. \text{type} \vdash_1 \lambda \varphi \iff \nu \varphi \iff \tau \\
\forall \gamma, \alpha. \text{type} \vdash_1 \exists \varphi \iff \tau \\
\forall \gamma, \alpha. \text{type} \vdash_1 \forall \varphi \iff \tau \\
\forall \gamma, \alpha. \text{type} \vdash_1 \nu \varphi \iff \tau \\
\forall \gamma, \alpha. \text{type} \vdash_1 \lambda \varphi \iff \nu \varphi \iff \tau \\
\forall \gamma, \alpha. \text{type} \vdash_1 \exists \varphi \iff \tau \\
\forall \gamma, \alpha. \text{type} \vdash_1 \forall \varphi \iff \tau \\
\forall \gamma, \alpha. \text{type} \vdash_1 \nu \varphi \iff \tau \\
\forall \gamma, \alpha. \text{type} \vdash_1 \lambda \varphi \iff \nu \varphi \iff \tau \\
\forall \gamma, \alpha. \text{type} \vdash_1 \exists \varphi \iff \tau \\
\forall \gamma, \alpha. \text{type} \vdash_1 \forall \varphi \iff \tau \\
\end{aligned}
$$

### Variations in being odd and even.

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

$$
datatype \alpha \text{ stream} = \text{Nil} | \text{Cons} \ (\alpha \times \alpha \text{ stream}) \text{ susp}$$

where $\text{stream susp}$ is the type of a thunk that yields an $\alpha$ stream, can be represented as the imperative type $\mu\alpha.\beta. (1 + V (\alpha \times \alpha \text{ stream susp}))$. Note the slightly awkward $\mu\alpha.\beta. (1 + V (\alpha \times \alpha \text{ stream susp}))$, in which $\gamma$ doesn’t occur; we can’t simply write $\mu\alpha.\beta. (1 + V (\alpha \times \alpha \text{ stream susp}))$, as 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, $\mu\alpha.\beta. (1 + V (\alpha \times \alpha \text{ stream susp}))$, which corresponds to the SML type

$$
datatype \alpha \text{ stream} = \text{Nil} | \text{Cons} \ (\alpha \times \alpha \text{ stream}) \text{ susp}$$

Here, the contents $\alpha$ 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

$$
datatype \alpha .\text{ stream} = \text{Nil} _- | \text{Cons} .\alpha \times \alpha \text{ stream susp}$$

where

$$
datatype \alpha .\text{ stream susp} = \text{Nil} _- | \text{Cons} .\alpha \times \alpha \text{ stream susp}$$

where

$$
\mu\alpha.\beta. (1 + V (\alpha \times \alpha \text{ stream susp}))$$

correspond to $\mu\alpha.\beta. (1 + V (\alpha \times \alpha \text{ stream susp}))$, with the $N$ on $\mu$ playing the role of the $\mu$type 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 imperative 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 \( v ::= () | \lambda x. e | (v_1, v_2) | \text{inj}_k v \)

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

By-name eval. contexts \( C_n ::= [C_n e_2 v_1 C_n] \)
\( (C_n, e_2) | (v_1, C_n) | \text{proj}_k C_n \)
\( \text{inj}_k C_n | \text{case}(C_n, x_1, e_1, x_2.e_2) \)

Figure 5. Source reduction

\[ \text{er}(e) = e' \]
Source expression e erases to e'

\[ \text{er}(\lambda x. e) = \text{er}(e) \]
\[ \text{er}(e[S]) = \text{er}(e) \]
\[ \text{er}((e:S)) = \text{er}(e) \]

Figure 6. Erasing types from source expressions

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 \( \mu \) by-name computation, so we define the source stepping relation \( \Rightarrow \) using 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 \([\_]\). If \( e \) reduces by value to \( e' \), written \( e \Rightarrow_{\text{rv}} e' \), then \( C_v[e] \Rightarrow_{\text{rv}} C_v[e'] \). For example, if \( e_2 \Rightarrow_{\text{rv}} e_2' \) then \( v_1 @ e_2 \Rightarrow_{\text{rv}} v_1 @ e_2' \), because \( v_1 @ [\_] \) is a by-value evaluation context.

Dually, \( C_n[e] \Rightarrow_{\text{rn}} C_n[e'] \) if \( e \Rightarrow_{\text{rn}} e' \). Every by-value context is a by-name context, and every pair related by \( \Rightarrow_{\text{rv}} \) is also related by \( \Rightarrow_{\text{rn}} \), but the converses do not hold. For instance, \( e_1 @ [\_] \) is a \( C_n \) but not a \( C_v \), and \( \text{proj}_2(e_1, e_2) \Rightarrow_{\text{rn}} e_2 \), 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 \( \Rightarrow_{\text{rv}}, \Rightarrow_{\text{vn}}, \Rightarrow_{\text{rn}} \) are defined in Figure 5. The definitions of \( \nu \), \( C_v \), and \( \Rightarrow_{\text{rv}} \), taken together, are standard for call-by-value; the definitions of \( C_n \) and \( \Rightarrow_{\text{rn}} \) are standard for call-by-name. The peculiarity is that \( \Rightarrow \) can behave either by value (rule \( \text{SrcStepCtxtV} \)) or by name (rule \( \text{SrcStepCtxtN} \)).

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 \( \text{IVintro} \) 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 \( D \).

\[ \text{let } r : ref \{ [D_a. \tau \Rightarrow \tau] \} = \text{ref } f \text{ in } \]
\[ r := g; \quad h(r) \]

Assume we have \( f : D_a. \tau \Rightarrow \tau \) and \( g : \tau \Rightarrow N \tau \) and \( h : (\tau \Rightarrow \nu) \Rightarrow \tau \). By a version of \( \text{IDintro} \) that doesn’t require its subject to be a value, we have \( r : ref (\tau \Rightarrow \nu) \). By \( \text{IDelim} \) with \( N \) for \( a \), we have \( r : ref (\tau \Rightarrow \nu) \), making the assignment \( r := g \) well-typed. However, by \( \text{IDelim} \) with \( V \) for \( a \), we have \( r : ref (\tau \Rightarrow \nu) \). It follows that the dereference \( r \) has type \( \tau \Rightarrow \nu \), so \( r \) can be passed to \( h \). But \( h \) is actually call-by-name. If \( h = \lambda x. e_2 \), 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 \( D \) as an intersection type, so that \( r \) has type \( (\tau \Rightarrow \nu) \land (\tau \Rightarrow \nu) \), 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 \( \eta \)-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:

\[ f : ((\tau_1 \land \tau_1') \Rightarrow \tau_2) \Rightarrow \tau_3 \]
\[ g : (\tau_1 \Rightarrow \tau_2) \Rightarrow f : g : \tau_3 \]

Through a subsumption rule, \( g : (\tau_1 \Rightarrow \tau_2) \) checks against type \( (\tau_1 \land \tau_1') \Rightarrow \tau_2 \), because a function that accepts all values of type \( \tau_1 \) should also accept all values that have type \( \tau_1 \) and type \( \tau_1' \).

Using the analogy between intersection and \( D \), in our impartial type system, we might expect to derive

\[ f : (D_a. \tau_1 \Rightarrow \tau_1') \Rightarrow \tau_2 \]
\[ g : (\tau_1 \Rightarrow \tau_1') \Rightarrow \tau_2 \Rightarrow f : g : \tau_3 \]

Here, \( f \) asks for a function of type \( (D_a. \tau_1 \Rightarrow \tau_1') \Rightarrow \tau_2 \), which works on all evaluation orders; but \( g \)’s type \( (\tau_1 \Rightarrow \tau_1') \Rightarrow \tau_2 \) 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 \( \text{Isub} \)—either by treating \( D \) similarly to \( \nu \) (Dunfield and Krishnaswami 2013), or by treating \( D \) as an intersection type (Dunfield
This idea, developed by Barendregt et al. (1983), can be automated; alternatively, \( \eta \)-expansion can substitute for subtyping: even without subtyping and a subsumption rule, we can derive

\[
f : (\langle Da, \tau_1 \rightarrow \tau_1 \rangle \rightarrow \tau_2) \rightarrow \tau_3,
g : (\tau_1 \rightarrow \tau_2 \rightarrow f \langle x, g \ x \rangle : \tau_3
\]

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

3. Economical Type System

![Figure 7. Type translation into the economical language](image)

The economical 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 (\( \rightarrow \), \( + \), \( \ast \), \( \mu \)) are by-value, but with a suspension point \( e \vdash S \) 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 \( \tau \) by dropping all the \( e \) decorations and adding a connective \( e \vdash S \) (read “\( e \) suspend \( S \)”). When \( e \) is \( V \), this connective is a no-op; elaborating \( e \) at type \( \mathbf{V} \vdash S \) and at type \( S \) yield the same term. But when \( e \) is \( N \), elaborating \( e \) at type \( \mathbf{N} \vdash S \) is like elaborating \( e \) at type \( \mathbf{I} \rightarrow S \).

In economical typing contexts \( \Gamma \), 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 \( e \) decorations means that—apart from the valueness annotations—most of the economical rules in Figure 8 look fairly standard. The only new rules are for suspension points \( e \vdash \), 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 \( e \) in \( e \vdash S \): if we introduce the type \( \mathbf{N} \vdash S \), then \( e \) will be elaborated to a thunk, which is a value; if we are eliminating \( \mathbf{N} \vdash S \), the elaboration of \( e \) will have the form \( \mathbf{force} \ldots \), 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 \( \langle \tau \rangle = S \) that inserts suspension points (Figure 7). Given an impartially-typed source program \( e \) of type \( \tau \), we can show that \( \langle e \rangle \) has the economical type \( \langle \tau \rangle \) (Theorem 1).

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

We are less constrained in how to translate other connectives:

- We could translate \( \tau_1 \rightarrow^e \tau_2 \) to \( \langle e \vdash [\tau_1] \rangle \rightarrow [\tau_2] \). But then \( 1 \vdash \mathbf{N} \rightarrow 1 \) — presumably intended as a non-strict boolean type — would be translated to \( \mathbf{N} \rightarrow \mathbf{N} \), 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 \( 1 \vdash \mathbf{N} \rightarrow [\tau_1] \).
- We could translate \( \mu \alpha.\tau \) to \( \mathbf{mu} \alpha.\langle e \vdash [\tau_1] \rangle \rightarrow [\tau_2] \) — which corresponds to how we decided to translate sum types. Instead, we translate it to \( \langle e \vdash [\tau_1] \rangle \rightarrow (\langle e \vdash [\tau_2] \rangle) \), so that, when \( e = N \), 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 \( \mu \alpha.\tau \), we could put a suspension on each occurrence of \( \alpha \) in \( \tau \), rather than a single suspension on the outside of \( \tau \). Since \( \tau \) is often a sum type, writing \( ^e+ \) already puts a thunk on \( \tau \); we don’t need a thunk around a thunk. But by the same token, suspensions around the occurrences of \( \alpha \) can also lead to double thunks: translating the type of lazy natural numbers \( \mu \alpha.\langle 1 \vdash \mathbf{N} \rangle \) would give \( \mu \alpha.\langle \mathbf{N} \rightarrow \mathbf{N} \rangle \), which expands to \( \mathbf{N} \rightarrow \mathbf{N} \rightarrow \mathbf{N} \rightarrow \ldots \).

The rationale 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 \( \tau_1 \rightarrow^e \tau_2 \) can be obtained by writing \( (\mu \beta.\langle \tau_1 \rangle) \rightarrow^{e^V} (\mu \beta.\tau_2) \), where \( \beta \) doesn’t occur; the only purpose of \( \mu \) here is to insert a suspension. (This suggests a kind of ill-founded argument for our chosen translation of \( \mu \); 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:

\[
\text{type List } a = \mu \beta. \mathbf{a} \Rightarrow (1 + (a + \beta))
\]

The body of \( \text{map} \) is the same; only the type annotation is different.

\[
\text{map} : \mathbf{Aa} \rightarrow \forall \alpha. (a \rightarrow \alpha) \rightarrow (\text{List } a \rightarrow \text{List } \alpha) = \mathbf{Ax} \rightarrow \text{map } a \rightarrow \text{map } a
\]

\[
\text{case}(x : s, x_1, \mathbf{inj}_1, y), x_2, \mathbf{inj}_2 (x \rightarrow \text{map } y (\text{proj}_1 x, \text{map } x \rightarrow \text{map } x (\text{proj}_2 x)))
\]

The above type for \( \text{map} \) corresponds to the impartial type with \( \forall \).

At the end of Section 2.3, we gave a very generic type for \( \text{map} \), which we can translate to the economical system:

\[
\mathbf{DA} a_1, a_2, a_3, a_4, a_5.
\]

\[
\forall \alpha. (a_1 \Rightarrow (a_1 \Rightarrow \alpha) \Rightarrow a_1 \Rightarrow (\text{List } a_2 \Rightarrow \alpha) \rightarrow \text{List } a_5 \beta)
\]

262
Economical types $S := \mathbf{1} | \alpha \mid \forall \alpha. S \mid \Delta a. S \mid e : S$

Economical bidirectional typing

This type might not look economical, but makes redundant suspensions more evident: List $a_1 \mid \alpha$ is $\mu a_1 \mid \alpha \cdot a_1 \cdot \cdots$, so the suspension controlled by $a_2$ 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 $\mathcal{V} \mathcal{R} S'$ to assumptions with $S'$.

Theorem 1 (Economizing).

(1) If $\gamma \vdash_1 e \varphi \Rightarrow \tau$ then $[\gamma] \vdash_{\mathcal{E}} [e] \varphi \Rightarrow [\tau]$.

(2) If $\gamma \vdash_1 e \varphi \Leftarrow \tau$ then $[\gamma] \vdash_{\mathcal{E}} [e] \varphi \Leftarrow [\tau]$.

4. Target Language

Our target language (Figure 9) has by-value $\to$, $\ast$, $+$ and $\mu$ connectives, $\forall$, and a $\mathcal{U}$ connective (for thunks).

The $\forall$ connective has explicit introduction and elimination forms $\forall \alpha. M$ and $M[\_]$. This "type-free" style is a compromise between having no explicit forms for $\forall$ and having explicit forms that contain types $\forall \alpha. M$ and $A[\mu M]$. 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 $\mu$ types.

As with $\forall$, we distinguish thunks to simplify some proofs: Source expressions typed with the $\mathcal{V}R$ connective are elaborated to thunk $M$, rather than to a $\lambda$ with an unused bound variable.

Target terms $M := (\_ | \lambda x. M | M_1 M_2 | u \mid \text{fix} u. M | \Delta a. M | M[\_] | \text{thunk} M | \text{force} M | (M_1, M_2) | \text{proj}_k M | \text{inj}_i M | \text{case}(M, x_1, M_1, x_2, M_2) | \text{roll} M | \text{unroll} M$

Values $W := (\_ | \lambda x. M | \Delta a. M | \text{thunk} M | (W_1, W_2) | \text{inj}_i W | \text{roll} W$

Valuables $V := (\_ | \lambda x. M | \Delta a. V | V[\_] | \text{thunk} M | (V_1, V_2) | \text{proj}_k V | \text{inj}_i V | \text{roll} V | \text{unroll} V$

Eval. contexts $C := (\_ | C \mathcal{R} M_2 | W_1 \mathcal{R} C | C[\_] | \text{force} C | (C, M_1, M_2) | (W_1, C) | \text{proj}_k C | \text{inj}_i C | \text{case}(C, x_1, M_1, x_2, M_2) | \text{roll} C | \text{unroll} C$

Target types $A, B := (\_ | \alpha \mid \forall \alpha. A | A_1 \Rightarrow A_2 | U A_1 | A_1 + A_2 | A_1 \cdot A_2 | \mu \alpha. A$

Typing contexts $G := (\_ | G, x : A | G, \alpha$ type

Dually, eliminating $\mathcal{N}R$ results in a target term 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 $\mathcal{V}R\text{intro}$ rule and the rules for thunks:
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 evalIntro, 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 evalElim, 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 valuability V if it is a value (e.g. λx. M) or is a projection, injection, roll or unroll of something that is valuiable (Figure 9). Later, we’ll prove that if a source expression is a value (according to the source typing derivation), its elaboration is valuiable (Lemma 6).

Thunks. We give thunk M the type 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 ⊢_R M', read “M reduces to M'”, and M →_R M', read “M steps to M'”. The latter has only one rule, StepContext, which says that C[M] →_R C[M'] if M →_R M'.

The rules for →_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 (λx. M) Q, this is all standard: these definitions use values W, not valuables V.

4.3 Type Safety

Lemma 2 (Valueability). If V →_R M' or V →_R M' then M' is valuiable, that is, there exists V' = M'.

Lemma 3 (Substitution). If G, x : A', G' ⊢_R M : A and G ⊢_W : A' then G, G' [W/x]M : A.

4.4 Elaboration

Now we extend the economical typing judgment with an output M, a target term: Γ ⊢ e : ϕ; S ⊢ 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 : ϕ; S ⊢ M then A : S; this is Theorem 10. Our translation follows a similar approach to Dunfield (2014). However, that system had general intersection types A₁ ∧ A₂, where A₁ and A₂ don’t necessarily have the same structure. In contrast, we have DA, which corresponds to (|V/a|A) ∧ (|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 evalIntro and evalElim 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 “⇐” in place of ⊢_", this obviates the need for elaboration versions of Esub and Eanno, which merely switch between ⇐ and ⇒.

Figure 10. Target language operational semantics

Figure 11. Target language operational semantics

Figure 12. Translation from economical types to target types

Theorem 4 (Type safety). If Γ ⊢ e : ϕ; S ⊢ M then either M is a value, or M →_R M' and G ⊢_R M' : A.

Proof. By induction on the derivation of G ⊢_R M : A, using Lemma 3 and standard inversion lemmas, which we omit.

5. Elaboration

Now we extend the economical typing judgment with an output M, a target term: Γ ⊢ e : ϕ; S ⊢ 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 : ϕ; S ⊢ M then A : S; this is Theorem 10. Our translation follows a similar approach to Dunfield (2014). However, that system had general intersection types A₁ ∧ A₂, where A₁ and A₂ don’t necessarily have the same structure. In contrast, we have DA, which corresponds to (|V/a|A) ∧ (|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 evalIntro and evalElim 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 “⇐” in place of ⊢_", this obviates the need for elaboration versions of Esub and Eanno, which merely switch between ⇐ and ⇒.
Erased source expression e elaborates at type S to target term M

\[
\Gamma \vdash e : S \mapsto M
\]

Erased source expression e elaborates at type S to target term M

\[
\Gamma \vdash e : S \mapsto M
\]

\[
\Gamma, \alpha \text{type} e : S \mapsto M
\]

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, \( \text{elabvar} \) elaborates x to x, and \( \text{elab} \rightarrow \text{Intro} \) elaborates \( \lambda x. e \) to \( \lambda x. \text{elab} \).

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

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

Rule \( \text{elab} \rightarrow \text{Elim} \) elaborates to the appropriate projection.

**Elaborating \( \rightarrow \).** Rule \( \text{elab} \rightarrow \text{Intro} \) has two conclusions. The first conclusion elaborates at type \( V \rightarrow S \) as if elaborating at type S. The second conclusion elaborates at \( N \rightarrow S \) to a thunk. Correspondingly, rule \( \text{elab} \rightarrow \text{Elim} \) ignores the V suspension, and rule \( \text{elab} \rightarrow \text{Elim} \) forces the thunk introduced via \( \text{elab} \rightarrow \text{Intro} \).

### 5.1 Elaboration Type Soundness

The main result of this section (Theorem 10) is that, given a non-elaborating economical typing derivation \( \Gamma \vdash e : S \mapsto M \), we can derive \( \Gamma \vdash e : e \mapsto S \mapsto M \) such that the target term M is well-typed. The erasure function \( e \), defined in Figure 6, removes type annotations, type abstractions, and type applications.

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

**Lemma 5.** If \( \Gamma \vdash e : S \mapsto W \) then \( e = \text{val} \).

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

**Lemma 6 (Elaboration valueness).** If \( \Gamma \vdash e : S \mapsto M \) then M is valuable, that is, there exists \( V \) such that \( M = V \).

Several substitution lemmas are required. The first is for the non-elaborating economical type system; we’ll use it in the \( \text{E} \rightarrow \text{Intro} \) case of the main proof to remove an evalorder declarations.

**Lemma 7 (Substitution—Evaluation orders).**

1. If \( \Gamma, e \vdash \text{evalorder} \), \( \Gamma' \vdash e : S \mapsto T \) and \( \Gamma' \vdash e : S \mapsto T \) then \( \Gamma, e [a/\alpha] \vdash \text{evalorder} \).
2. If \( \Gamma' \vdash e : S \mapsto T \) and \( \Gamma' \vdash e : S \mapsto T \) then \( \Gamma' \vdash e [a/\alpha] \vdash \text{evalorder} \).
3. If \( \Gamma' \vdash e : S \mapsto T \) and \( \Gamma' \vdash e : S \mapsto T \) then \( \Gamma' \vdash e [a/\alpha] \vdash \text{evalorder} \).

Next, we show that an expression e1 can be substituted for a variable x, provided e1 elaborates to a target value W.

**Lemma 8 (Expression substitution).**

1. If \( \Gamma \vdash \text{fix} u, e : T \vdash \text{fix} u, M_1 \) and \( \Gamma \vdash \text{fix} x, S_1 \vdash e \mapsto S \mapsto M_2 \)

Next, we show that an expression e1 can be substituted for a variable x, provided e1 elaborates to a target value W.

**Lemma 9 (Expression substitution).**

1. If \( \Gamma \vdash \text{fix} u, e : T \vdash \text{fix} u, M_1 \) then \( \Gamma \vdash \text{fix} u, e [a/\alpha] \vdash \text{fix} u, M_2 \)
Lemma 9 (Type translation well-formedness).
If \( \Gamma \vdash T \) type then \( [\Gamma] \vdash [T] \) type.

We can now state the main result of this section:

Theorem 10 (Elaboration type soundness).
If \( \Gamma \vdash e \preceq S \) or \( \Gamma \vdash e \Rightarrow S \)
where \( \Gamma \vdash T \) type and \( \Gamma \) contains no a evalorder declarations
then there exists \( M \) such that \( \Gamma \vdash e \leadsto e' \vdash: S \leftarrow M \)
where \( q' \subseteq q \) and \( [\Gamma] \vdash T : [S] \).

The proof is in Dunfield (2015, Appendix B.5). In this theorem,
the resulting elaboration judgment has a valueness \( q' \) that can be
more precise than the valueness \( q \) in the non-elaborating judgment.
Suppose that, inside a derivation of a evalorder \( e \vdash e \preceq S \), we have

\[
\begin{align*}
\Gamma & \vdash e \preceq S' \\
\Gamma & \vdash e \preceq S' \\
\end{align*}
\]

The valueness in the conclusion must be \( T \), 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
e elaboration derivation, and the only rule that works is elab \( \Rightarrow \) Elimv:

\[
\begin{align*}
\Gamma & \vdash e \preceq S' \\
\Gamma & \vdash e \preceq S' \\
\end{align*}
\]

This says \( e' \) is a value (val), where the original (parametric) econo-
mical typing judgment had \( T \): Substituting a concrete object
(here, \( V \)) for a variable a increases information, refining \( T \) ("I can-
not prove this is a value") into val. In the introduction rules, substi-
tuting \( N \) for a can replace \( T \) with val, because we know we're
e elaborating to a thunk, 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 elabo-
rates 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 con-
structs, then all the steps taken in the source program are by-value.

That still leaves the possibility that we messed up our elabora-
tion rules, such that a call-by-name 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,
it 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 \( T \). Taking only a by-
name reduction, we have:

\[
(\lambda x. O)(\text{fix } u. u) \leadsto (\text{fix } u. u)/x\langle \rangle = \langle \rangle \text{ using } \beta\text{Nreduce}
\]

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

\[
(\lambda x. O)(\text{fix } u. u) \leadsto (\lambda x. O)((\text{fix } u. u)/u)\langle u \rangle \text{ using } \beta\text{Nreduce}
\]

\[
\leadsto (\lambda x. O)(\text{fix } u. u) \leadsto \ldots
\]

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 ap-
pealing to elaboration and target reductions) and useful, we’d need
to give a semantics of “reduction with respect to a typing derive-
tion”, or else reduction of a typing derivation. Such a semantics
would support reasoning about local transformations of source pro-
grams. It should also lead to a converse of the consistency result
in this section: if a source expression reduces with respect to a typings
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 \( T \) is N-free if (i) for each \( e \) appearing in
\( S \), the evaluation order \( e \) is \( V \); and (ii) \( T \) has no \( D \) quantifiers.
(2) A judgment \( \Gamma \vdash e \preceq S \) is N-free if: (a) \( \Gamma \)
has no a evalorder declarations; (b) in each declaration \( x \preceq T \)
in \( \gamma \), the valueness \( q \) is \( V \) and the type \( T \) is N-free; (c) all types
appearing in \( e \) are \( N \)-free; and (d) \( T \) is N-free.

Definition 2 (N-freeness—economical).
(1) An economical type \( S \) is N-free if (i) for each \( e \preceq S_0 \)
appearing in \( S \), the evaluation order \( e \) is \( V \); and (ii) \( S \) has no \( D \) quantifiers.
(2) A judgment \( \Gamma \vdash e \preceq S \) is N-free if: (a) \( \Gamma \)
has no a evalorder declarations; (b) all types \( S' \) in \( \Gamma \) 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 if it
contains no thunk and force constructs.

6.3 Lemmas for Consistency

An inversion lemma allows types of the form \( V \bullet \ldots \bullet V \bullet S \),
a generalization needed for the elab \( \Rightarrow \) Elimv case; when we use the lemma
in the consistency proof, the type is not headed by \( V \):

Lemma 11 (Inversion). Given \( \Gamma, \cdot \vdash e \preceq S_0 \).

(0) If \( M = (\lambda x. M_0) \) and \( S = (S_1 \rightarrow S_2) \)
then \( e = (\lambda x. e_0) \) and \( \cdot \vdash e_0 \preceq S_2 \rightarrow M_0 \).
(1) If \( M = (W_1, W_2) \) and \( S = (\lambda a. S_0) \)
then \( e \vdash (N[a]S_0) \rightarrow W_1 \) and \( e \vdash (N[a]S_0) \rightarrow W_2 \).
(2) If \( M = \text{thunk } M_0 \) and \( S = V \bullet S_0 \)
then \( e \vdash e_0 \preceq S_0 \rightarrow M_0 \).

Parts (3)–(6), for \( V \), \( + \), \( * \), and \( \times \), 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 \( (q = val) \);
here, we show that if a source expression elaborates to a target
value that is N-free (ruling out thunk \( M \) produced by the second
conclusion of elab \( \Rightarrow \) Intro), then \( e \) is a syntactic value.

Lemma 12 (Syntactic values).
If \( \Gamma, \cdot \vdash e \preceq V \bullet S \rightarrow W \) and \( W \) is N-free then \( e \) is a syntactic value.

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

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

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

6.4 Consistency Results

Theorem 14 (Consistency).
(1) If \( \cdot \vdash e \preceq S \rightarrow M \) and \( M \rightarrow M' \) then there exists \( e' \) such that
\( e \leadsto e' \) and \( \cdot \vdash e' \preceq S \rightarrow M' \) and \( q' \subseteq q \).
Moreover: (1) If \( q = val \) then \( e' = e \). (2) If \( M \) is N-free then
\( e \leadsto e' \) 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 \textit{SrcStepCtxN} instead of showing that, given N-free M, we can always find a by-value evaluation context for use in \textit{SrcStepCtxV}.

**Theorem 15** (Multi-step consistency).
If \( \vdash e : \phi \); \( S \rightarrow M \) and \( M \xrightarrow{\ast} W \) then there exists \( e' \) such that \( e \xrightarrow{\ast} e' \) and \( \vdash e' : \psi \); \( S \rightarrow W \). Moreover, if M is N-free then we can derive \( e \xrightarrow{\ast} e' \) without using \textit{SrcStepCtxN}.

### 6.5 Preservation of N-Freeness

**Lemma 16.** If \( \Gamma \vdash e : \phi \Rightarrow S \) and S is not N-free then it is the case that both \( \Gamma \) and e are N-free.

**Theorem 17** (Economizing preserves N-freeness).
If \( \gamma \vdash e : \phi \Rightarrow \tau \) (resp. \( \Rightarrow \)) where the judgment is N-free (Definition 1 (2)) then \( \Gamma \vdash e : \phi \Rightarrow [\tau] \) (resp. \( \Rightarrow \)) where this judgment is N-free (Definition 2 (2)).

**Theorem 18** (Elaboration preserves N-freeness).
If \( \Gamma \vdash e : \phi \Leftrightarrow S \) (or \( \Rightarrow \)) where the judgment is N-free (Definition 2 (2)) then \( \Gamma \vdash e \Rightarrow e : \phi \Rightarrow S \rightarrow M \) such that M is N-free.

### 7. Related Work

**History of evaluation order.** In the \( \lambda \)-calculus, normal-order (leftmost-outmost) 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, 47.3); 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 \( \lambda \)-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 \( \lambda \) 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 \( \Delta \) 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 (1997) gives a useful introduction and survey. Intersections entered programming languages—as opposed to \( \lambda \)-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 Pfennig (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 \( \Delta \) 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 \( e_1 \times e_2 \), 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 \( \emptyset \) in the source language as a merge \( (\emptyset^1, \emptyset^0) \), 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 \( \lambda \)-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 \( U \) (whence our notation) but also a dual, “lift” 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 \textit{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 U and 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 \( \Pi \Delta \) 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 \textit{map} into, say, 8 instantiations is to write 8 copies of \textit{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 at all 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 $\exists a. \tau \to \tau$ 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 $\exists$. 

References
A. Asperti. Integrating strict and lazy evaluation: the $\Lambda_\mathbf{d}$-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.