Typing Strictness

DANIEL SAINATI, University of Pennsylvania, USA
JOSEPH W. CUTLER, University of Pennsylvania, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA
STEPHANIE WEIRICH, University of Pennsylvania, USA

Strictness analysis is critical to efficient implementation of languages with non-strict evaluation, mitigating much of the performance overhead of laziness. However, reasoning about strictness at the source level can be challenging and unintuitive. We propose a new definition of strictness that refines the traditional one by describing variable usage more precisely. We lay type-theoretic foundations for this definition in both call-by-name and call-by-push-value settings, drawing inspiration from the literature on type systems tracking effects and coeffects. We prove via a logical relation that the strictness attributes computed by our type systems accurately describe the use of variables at runtime, and we offer a strictness-annotation-preserving translation from the call-by-name system to the call-by-push-value one. All our results are mechanized in Rocq.

ACM Reference Format:
Daniel Sainati, Joseph W. Cutler, Benjamin C. Pierce, and Stephanie Weirich. 2025. Typing Strictness. 1, 1 (October 2025), 31 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Non-strict evaluation offers benefits over both fully lazy and fully strict evaluation strategies by allowing expressions to be evaluated at any point between when they are first encountered and when their value is needed. Unlike strict strategies, which *must* evaluate expressions as soon as they are encountered, non-strict strategies may delay evaluation arbitrarily, improving performance (e.g., by skipping unnecessary computation) and making it easier to work with codata like infinite lists. Unlike lazy strategies, which *must* wait until an expression is used before evaluating it, non-strict strategies may evaluate earlier to avoid creating thunks.

These benefits have led a number of languages to support non-strict evaluation, either as the default strategy, like Haskell [27] or R [21], or optionally via special annotations, like OCaml [46]. Non-strict evaluation is also critical for the performance of streaming libraries like FS2 in Scala [73] and code patterns like iterators in Rust [40], and it is a useful tool for automatic parallelization [9] and for opportunistic execution in scripting languages [57].

Non-strict evaluation also comes with a number of caveats. Overly lazy strategies suffer from drawbacks such as increased memory usage [25], unpredictable behavior [19] and space complexity [14, 15, 28], and security vulnerabilities [82]. To circumvent these issues, compilers like GHC [26] perform *strictness analysis* [60, 86], allowing them to reorder code to evaluate eagerly when they can prove that doing so will not change observable behavior. In pure settings, this analysis enables

Authors' Contact Information: Daniel Sainati, University of Pennsylvania, Philadelphia, USA, sainati@seas.upenn.edu; Joseph W. Cutler, University of Pennsylvania, Philadelphia, USA, jwc@seas.upenn.edu; Benjamin C. Pierce, University of Pennsylvania, Philadelphia, USA, bcpierce@seas.upenn.edu; Stephanie Weirich, University of Pennsylvania, Philadelphia, USA, sweirich@seas.upenn.edu.

ACM XXXX-XXXX/2025/10-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

performance optimizations that can massively reorder evaluation, but such improvements are not limited only to languages like Haskell; strictness analysis can still unlock speedups in impure languages when the compiler is able to ensure that it will not reorder any side effects [57].

Strictness analysis, however, comes with its own set of wrinkles. Allowing the compiler to reorder evaluation can result in unpredictable behavior, where modifying the usage of values in one place can change the performance [59] and even correctness [34] of code elsewhere. Additionally, the inherent imprecision of strictness analysis can result in expressions that seem as though they should evaluate strictly but do not, requiring the programmer to tweak their code to coax the analyzer into triggering the desired optimization [31]. Exacerbating these issues is the fact that the analysis' model of strictness is specialized to compilation and awkward as a reasoning tool: as we explain in Section 2, this model is overly extensional and can misrepresent how variables are used.

To clarify the traditional model's murkiness regarding variable usage, we propose a new, static notion of strictness, dubbed *intensional strictness*, that enables more direct reasoning about how programs use variables; any given occurrence of a variable can be either strict, lazy, or statically unknown. We present intensional strictness type-theoretically via a *type-and-effect* system [52], taking additional inspiration from the literature on *coeffects* [65]. Effect and coeffect annotations extend type systems to describe how programs interact with the world; the standard intuition is that effects describe what a program *does* to the world, while coeffects describe what a program *requires* from the world. The interplay between effects and coeffects is an active research area [13, 22, 55, 80], and we adopt mechanisms from these works to characterize the way that strict usage both imposes a demand on a program's environment (by requiring that a value be successfully produced), and modifies that environment (by triggering the evaluation of suspended computations).

Our new definition of strictness, while syntactic in nature like any type system, captures semantic notions of strict and lazy usage that should be intuitively familiar to users of Haskell and similar languages. Furthermore, it refines the extensional characterization of strictness offered by conventional strictness analysis, describing variable usage more precisely and decoupling the definition from questions of "demand" or observable behavior.

We formalize our new notion of strictness via CBN$^{\gamma}$, a call-by-name (CBN) calculus with strictness effects, and CBPV$^{\gamma}$, an extension of call-by-push-value (CBPV) [47] with similar annotations. CBPV is a convenient foundation for the study of strictness because it specifies explicitly where computation occurs by separating computations from values and including explicit constructs that suspend and resume execution. Furthermore, a canonical translation exists from CBN into CBPV [50], allowing us to use CBPV$^{\gamma}$ as a tool to understand strictness in CBN languages. As we argue in Section 5, this understanding further extends to call-by-need evaluation [44, 60, 88] as well.

Our primary motivation, throughout, is conceptual: our proposed definition is intended as a tool for better understanding and reasoning about strictness, not yet as a basis for practical implementation or surface-level design of non-strictly evaluated languages. All of our proofs [71] are mechanized in Rocq [79].

Concretely, our contributions are:

- We propose a new notion of *intensional strictness* and argue that it refines the extensional definition used by traditional strictness analysis (Section 2).
- We present CBN$^{\gamma}$, a *type-and-effect system* for a call-by-name language that embodies this new form of strictness (Section 3).
- We present CBPV$^{\gamma}$, a variant of CBPV that also embodies intensional strictness. We instrument a big-step operational semantics with strictness attributes and prove the soundness of CBPV$^{\gamma}$ typing with respect to this semantics, thus providing a sound semantics for CBN$^{\gamma}$ via a translation to CBPV$^{\gamma}$ that preserves strictness annotations (Section 4). CBPV$^{\gamma}$ allows

us to factor our proofs about CBN$^\gamma$ through a lower-level intermediary, simplifying the proofs themselves and potentially offering a setting for reasoning about strictness in other languages with translations into CBPV$^\gamma$.

- We prove that the strictness attributes computed by CBPV$^\gamma$ truly reflect strict and lazy usage: a well-typed program *can* be run in an environment without a binding for any lazily-used variable and *cannot* be run in an environment lacking any strictly-used variable. These proofs show that intensional strictness, as modeled by CBN$^\gamma$ and CBPV$^\gamma$, refines the original extensional definition (Section 5). The proofs make use of a pair of logical relations and involve a rather delicate treatment of variables and scoping.
- We enrich CBPV$^\gamma$ and CBN$^\gamma$ with *unused variable tracking*, an extension that is surprisingly simple in the former language and surprisingly complex in the latter. All the previously proven theorems hold for this extension (Section 6).

Section 7 discusses related work. Section 8 concludes and outlines possible future work.

2 What is Intensional Strictness?

In this section, we offer a new, *intensional* definition of strictness and examine how it differs from the extensional version used by strictness analysis. We build an intuition for how intensional strictness works and explain why it is natural to model it type-theoretically.

We describe each syntactic occurrence of a variable as either a *strict* or a *lazy* usage of that variable. A strict variable usage scrutinizes the value stored in that variable (e.g., it pattern matches on a pair or applies a function), while a lazy usage does not (e.g., it places the variable into a cons cell of a lazy list). A helpful analogy compares variables to boxes: a strict use opens the box to observe its contents, while a lazy use passes the box along unopened.

We might try to lift this terminology to call-by-name functions—"a strict function is one that scrutinizes its argument"—but this naïve definition is ambiguous about how the return values of functions are used. It is unclear from this definition what the strictness of the identity function $\lambda x.x$ should be, for example: it does not directly scrutinize its argument, but scrutinizing the result of a call to the identity *would* scrutinize the argument as well. Instead of considering functions in isolation, we need a more precise definition that allows us to reason about the contexts in which functions are called and the demands made on their results.

The strictness-analysis literature [60, 86] attempts to provide this precision by defining strict functions as those that fail to evaluate whenever their arguments fail to evaluate and lazy ones as those that (might) succeed even if their argument fails. This definition posits a hypothetical $\bot$ term, intuitively representing the result of an erroring or non-terminating computation, and says that a function f is strict if $f\bot = \bot$; that is, a strict function preserves failure. Because it is defined in terms of observable behavior, we refer to this notion of strictness as *extensional strictness*.

Strictness-analysis algorithms based on extensional strictness can mitigate much of the performance overhead of laziness [41, 75]. Compilers perform strictness analysis in an optimization pass, wherein they try to determine if it is safe to evaluate an expression eagerly and potentially save space by not allocating a thunk. By "safe," we mean that eager evaluation would not change the observable behavior of the program, compared to evaluating that same expression lazily. This use case shows why the strictness-analysis literature uses the definition it does: if $f\bot = \bot$, then f's argument can be evaluated eagerly, since, if the argument fails to evaluate, the call will too.

As an example of how strictness analysis can improve performance, consider an implementation of `sum` for lists using a tail-recursive fold in a Haskell-like language: `sum lst = foldl (+) 0 lst`. A programmer familiar with tail recursion might expect this function to use constant stack space, but, when evaluated lazily, it uses space linear in the length of the input.

```
-- x is strict, a is indeterminate
f1 x a = if x then a + 1 else 2

-- z is strict
f2 z = z

-- u is lazy
f3 u = loop

-- y is lazy
f4 y = Just y
```

Fig. 2. Simple examples of variable usage

To see why, let's step through a lazy evaluation of `sum` in Figure 1. Evaluation begins by applying `+` to the accumulator and the first element of `[1 .. n]`, the input to the **`foldl`**. However, rather than completing that application and yielding an integer value as the new accumulator, evaluation instead yields a thunk referencing both the old accumulator and the list element, resulting in an ever-growing chain of nested thunks. This chain is only forced once evaluation reaches the end of the list and the result of the call to `sum` is used; in the meantime, the entire contents of the list will be materialized within it.

```
foldl (+) 0 [1 .. n]                  - ->
foldl (+) (0 + 1) [2 .. n]            - ->
foldl (+) ((0 + 1) + 2) [3 .. n]      - ->
foldl (+) ((((0 + 1) + 2) + ...) + n) []
```

Fig. 1. Lazy evaluation of list sum using foldl

If we recognize that the whole contents of the list will eventually be scrutinized by the `+` operator, we can instead eagerly evaluate the application to a value and maintain constant space without changing observable behavior; if computation of any element of the list fails, we could never have computed the list's sum, regardless of when that computation occurred. Compilers like GHC use strictness analysis to improve performance by exploiting this realization to reorder evaluation [30].

However, despite its usefulness to compilers, extensional strictness is otherwise unsatisfying as a tool for human reasoning about programs. In particular, it does not directly say anything about how values are used! A function that always fails (i.e., always returns $\perp$), for example, is considered strict even if it never mentions its input, despite the fact that strictly evaluating a call to such a function can in fact change program behavior. This characterization is counterintuitive; to better understand strict and lazy usage, we would like a model of strictness that does not conflate a function's observable behavior with the way that it uses its argument.

To capture this additional nuance, we propose a definition of *intensional strictness* that directly describes how variables are used within the bodies of functions. We assume that the results of function calls are scrutinized; if the result of a call is used lazily, then the call will not be evaluated at all. Intensional strictness is presented informally here in terms of functions; later we will formalize it and generalize to open terms via the type systems in Sections 3 and 4.

Definition 2.1 (Intensional Strictness)**.** Functions can exhibit one of three kinds of strictness:

(1) A *strict function* is one that, on all possible execution paths through its body, scrutinizes its argument whenever its return value is scrutinized.
(2) A *lazy function* is one that does not scrutinize its argument, regardless of how its return value is used.
(3) An *indeterminate function* may or may not scrutinize its argument; the argument's usage is either statically unknown or varies depending on which execution path is taken when the function is called.

To get a feel for this definition, let's consider the code snippets in Figure 2. What is the strictness of each of the parameters of these four functions, assuming that their results are used strictly?

Scrutiny of `f1`'s result means we must know its value, and thus we must evaluate `x` to choose the correct arm of the conditional. Hence, `f1` is strict in `x`. Similarly, `f2` is strict in `z` because it returns its argument: if the result of a call to `f2` is scrutinized, then its argument is too. The extensional definition agrees with these labels: `f1`'s argument `x` can be evaluated eagerly without changing the

program's observable behavior, since its value will always be needed by `f1`'s body. Similarly, the identity function `f2` is extensionally strict: given $\bot$ as its argument, its result will also be $\bot$.

The definitions disagree, however, on the strictness of `u` in `f3`. The $\bot$ term used by the extensional definition represents the result of a failing or non-terminating computation, and, according to this model, `f3` always returns $\bot$, as it will never terminate. Accordingly, the extensional definition of strictness would describe `f3` as strict in its argument: if `f3 u` $= \bot$ for all `u`, then `f3` $\bot = \bot$ trivially. However, `f3` does not actually mention `u` anywhere in its body, let alone use it strictly, so the intensional definition would describe it as lazy, more accurately describing how the function interacts with its argument. This difference can matter in practice: consider an application of `f3` to an argument that throws an exception. Eagerly evaluating that argument would change the behavior of the function call, causing an error where the call would otherwise run forever. The extensional definition does not capture this nuance, and the additional precision afforded to us in this case by reasoning syntactically exemplifies the intensional nature of our new definition.

Our new definition of strictness also provides an opportunity to be more precise about what lazy usage means by splitting the "lazy" label into two separate characterizations. We describe `f4` as lazy in `y`, since constructors in non-strictly-evaluated languages do not scrutinize their arguments, and scrutiny of `f4`'s return value only requires evaluating to the top-level constructor (in Haskell and similar languages, strict usage requires evaluation to weak head normal form [29]). The traditional definition would also describe `f4` as lazy, but laziness in the traditional sense only means that `f4`'s argument *might* not be scrutinized and hence cannot be pre-evaluated. Definition 2.1, by comparison, provides more clarity by telling us that `f4`'s argument is *definitely* not scrutinized, and thus that a call to `f4` will still succeed even if its argument fails to produce a value.

In `f1`, on the other hand, the argument `a` is scrutinized by `+`, but only in the success branch of the conditional; in the failure branch, it is not used at all. Definition 2.1 tells us that `f1` is not strict in `a`, as `a` is not used strictly on all possible execution paths. However, `f1` is not lazy in `a` either, since `a` may be scrutinized if `f1`'s return value is. Traditional strictness analysis would label `f1` as lazy in `a`, since eagerly evaluating `a` might change the function's behavior. But this is misleading; some calls to `f1` may actually scrutinize `a`. It is more precise to say that `f1`'s strictness is *indeterminate* with respect to `a`, reflecting the fact that `a` may be used differently by different calls to `f1`. In general, the "indeterminate" classification used by intensional strictness provides the same information as the traditional "lazy" label, while the intensional "lazy" label provides additional precision.

These examples illustrate the ways in which intensional strictness refines the extensional definition embodied in existing strictness-analysis algorithms: it more precisely characterizes cases that the traditional definition would label lazy, splitting the usual notion of lazy usage into "indeterminate" use and "definitely lazy" use. Furthermore, it decouples reasoning about strictness from orthogonal questions of extensional behavior, describing variable use intensionally and distinguishing between the different ways that computation can fail.

Definition 2.1 is informal, however: we still need a rigorous and formal model to undergird it. To generalize our reasoning about variables and functions to arbitrary terms and positions, the model must be compositional and syntactic, so a type-theoretic approach is naturally suited for this task. However, it is not clear whether existing type systems like usage typing [24, 66, 81, 89, 90] and information-flow typing [16, 64, 83, 92, 93] are capable of modeling intensional strictness (for reasons detailed in Section 7), so we will need to develop a type system of our own.

3 A Type System for Strictness

We present a call-by-name language, CBN$^{\gamma}$, that tracks intensional strictness in its type system using effects. The types and syntax are given in Figure 3. As a visual aid, we typeset the parts of CBN$^{\gamma}$ that are specific to strictness in orange, since these are the novel aspects of the system.

$$
\begin{array}{llll}
\textit{strictness attributes} & \alpha & ::= & \mathrm{S} \mid \mathrm{L} \mid ? \\
\textit{strictness effects} & \gamma & ::= & \cdot \mid \gamma, x : \alpha \\
\textit{types} & \tau & ::= & \mathtt{unit} \mid \tau_1^{\gamma_1} \times \tau_2^{\gamma_2} \mid \tau_1^{\gamma_1} + \tau_2^{\gamma_2} \mid (x :^{\alpha} \tau^{\gamma_1}) \xrightarrow{\gamma_2} \tau \\
\textit{contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau^{\gamma} \\
\\
\textit{expressions} & e & ::= & () \mid x \mid \mathbf{inl}\ e \mid \mathbf{inr}\ e \mid (e, e) \mid \lambda x.e \mid e_1\ e_2 \mid \mathbf{let}\ x = e\ \mathbf{in}\ e \mid \mathbf{sub}\ e \\
& & & \mid e; e \mid \mathbf{let}\ (x_1, x_2) = e\ \mathbf{in}\ e \mid \mathbf{case}\ e\ \mathbf{of}\ \mathbf{inl}\ x_1 \to e,\ \mathbf{inr}\ x_2 \to e
\end{array}
$$

Fig. 3. Types and Syntax of CBN$^{\gamma}$

Parts in black are standard. Along with the usual type introduction and elimination forms, the language includes a sequencing operation $e_1;\ e_2$, which strictly uses the `unit`-typed result of e_1 before continuing with e_2.

CBN$^{\gamma}$ assigns every variable one of three *strictness attributes* to track its usage, ordered according to the semilattice depicted in Figure 4. The metavariable α ranges over strictness attributes. The S attribute asserts that a variable is definitely used strictly at least once on every execution path. The L attribute asserts the opposite: that a variable is used only lazily (or not at all) on every execution path.[1] The ? attribute asserts nothing; a variable with this annotation may be used strictly, or lazily, or sometimes one and sometimes the other (on different execution paths), or not at all. We need the last of these because static strictness tracking (via types or otherwise) is inherently imprecise; in the presence of branching it is not always possible to be certain of how a variable will be used.

Fig. 4. Strictness semilattice

However, these attributes alone are not sufficient to track strictness. As McDermott and Mycroft [55] explain, scrutinizing variables in a non-strictly evaluated language can cause other code to be evaluated, and we must track how this evaluation uses variables as well. Accordingly, besides the attributes that appear on CBN$^{\gamma}$ types and variables to describe their usage, types also include *attribute vectors* (denoted with the metavariable γ) that describe how all in-scope variables are used by the suspended computations that inhabit a given type. These vectors γ are *effects* modeling the additional evaluation triggered by variable scrutiny; the type system tracks these effects by associating types and variables with γs describing what will happen when their values are scrutinized. (Further discussion of this terminology can be found in Section 4.6.) We write the vector that maps all in-scope variables to L as $\overline{\mathrm{L}}$, and, to reduce clutter, omit γs in examples when they are $\overline{\mathrm{L}}$. Additionally, we freely omit any variables with L attributes from γs. That is, $x : \mathrm{S}, y : \mathrm{L}$ and $x : \mathrm{S}$ denote the same vector.

3.1 A Few Examples

Before exploring the typing rules of CBN$^{\gamma}$ in detail, let's look at a few examples to build intuition. For each example term e, we'll describe how CBN$^{\gamma}$ types the term using a judgment of the form $\Gamma \vdash_{\mathsf{CBN}} e :^{\gamma} \tau$. The Γ in this judgment is a typing context, τ is the type of the term, and γ maps each free variable in e to a strictness attribute describing how it is used.

Addition. As an introductory example, consider the term $x + 1$. This term uses x strictly, as addition requires the value of x to be known in order to add to it. CBN$^{\gamma}$ checks this term using the judgment

$$x : \mathtt{Int} \vdash_{\mathsf{CBN}} x + 1 :^{x:\mathrm{S}} \mathtt{Int}.$$

[1] As we explain in Section 6, it is possible to be more precise here at the cost of some additional complexity.

The orange $x : \mathrm{S}$ that appears on the : is where we find the strictness information for the term being checked; x is assigned an S attribute, reflecting its strict usage.

Pairs and η-Laws. Next, we consider more complex data such as pairs. Because constructors in CBN$^\gamma$ do not require their arguments to be values, the types for such data must contain information about how their components use their free variables. Thus, CBN$^\gamma$'s types store separate γs for each of their components describing how they would use any in-scope variables if evaluated.

Consider a pair $(z, \mathtt{true})$ in a context where z has the type `Bool`. This term itself uses z lazily (i.e., with effect $z : \mathrm{L}$), but we also need to know how the components of this term would use z, were we to scrutinize them. CBN$^\gamma$ assigns this term the type $\mathtt{Bool}^{z:\mathrm{S}} \times \mathtt{Bool}$—let's call it P for short.

P describes pairs where each side contains a thunk that, when evaluated, will produce a boolean. The orange vectors in P tell us that the two sides of the product use the in-scope variable z differently. If P's first element is used strictly, z will be used strictly, but if its second element is used strictly, z will be used only lazily. CBN$^\gamma$ types this term with the judgment

$$z : \mathtt{Bool} \vdash_{\mathsf{CBN}} (z, \mathtt{true}) : \mathtt{Bool}^{z:\mathrm{S}} \times \mathtt{Bool}.$$

In general, the attributes on a judgment's : describe how a term uses its free variables *now*, while those in the term's type describe how it might use its free variables *later* as more of it is scrutinized.

Pairs and other structured types in CBN$^\gamma$ exhibit a crucial difference from other type systems for CBN calculi. Consider the terms x and $(\mathbf{fst}\ x, \mathbf{snd}\ x)$ in a context where x has type $\mathtt{Bool} \times \mathtt{Bool}$. In a pure CBN calculus, these two terms are considered η-equivalent and have the same type. In effectful settings like this one, however, the η-equivalence of these terms breaks down; any effects that would result from evaluating x occur only when scrutinizing the first term. The pair constructor in the second term suspends the uses of x, so x would not be evaluated at all if this term's top-level constructor were scrutinized. In short, the first term uses x strictly, while the second uses it lazily. CBN$^\gamma$ reflects this by requiring the two terms to be typed differently: x is typed with

$$x : \mathtt{Bool} \times \mathtt{Bool} \vdash_{\mathsf{CBN}} x :^{x:\mathrm{S}} \mathtt{Bool} \times \mathtt{Bool},$$

whereas $(\mathbf{fst}\ x, \mathbf{snd}\ x)$ is typed with

$$x : \mathtt{Bool} \times \mathtt{Bool} \vdash_{\mathsf{CBN}} (\mathbf{fst}\ x, \mathbf{snd}\ x) : \mathtt{Bool}^{x:\mathrm{S}} \times \mathtt{Bool}^{x:\mathrm{S}}.$$

While the first term uses x strictly, the second term uses x lazily and actually has a different type, one that also says that it uses x strictly in each of its components.

Contexts, Functions, and Latent Effects. So far, we have not considered the fact that, in non-strictly evaluated languages, accessing a variable may cause a thunk to be evaluated and thus other variables to be scrutinized. To track this deferred usage, typing contexts Γ map variables to both types τ and vectors γ; these γs describe the strictness with which any in-scope variables will be used if the thunked expression stored in a given variable is strictly used. Consider a program $\mathbf{let}\ x = \mathbf{if}\ y\ \mathbf{then}\ \mathtt{true}\ \mathbf{else}\ \mathtt{false}\ \mathbf{in}\ x$. Assuming y is in scope with type `Bool`, x is also a `Bool`, but one that uses y strictly if evaluated. Zooming in on the **let**'s body, CBN$^\gamma$ checks

$$y : \mathtt{Bool}, x : \mathtt{Bool}^{y:\mathrm{S}} \vdash_{\mathsf{CBN}} x :^{y:\mathrm{S}, x:\mathrm{S}} \mathtt{Bool}.$$

In the context, x is associated both with its type and with a vector $y : \mathrm{S}$, capturing the fact that any strict use of x will cause a strict use of y. This fact is also reflected in the judgment's output effect, which tells us that strict use of the term being checked will cause a strict use of y.

Extending this principle from open terms to functions, it is apparent that function types in CBN$^\gamma$ must also describe how their arguments use other variables. Function types in CBN$^\gamma$ thus have the shape $(x :^{\alpha} \tau_1^{\gamma_1}) \xrightarrow{\gamma_2} \tau_2$. The α (either S, L or ?) above the : describes how the function uses

its argument x, while the vector γ_2 describes the "latent effect" that will be produced when the function is called. The vector γ_1, on the other hand, describes the effect that the function's argument can have, i.e., how other variables will be used when the argument is evaluated. Note also that this function type is dependent, in the sense that τ_2 may mention x in its attribute vectors. A more detailed explanation of why this is necessary can be found in Section 3.2.

So, for example, one possible judgment for the identity function would be

$$y : \texttt{Bool} \vdash_{\mathsf{CBN}} \lambda x.\, x :^{y:\mathrm{L}} (x :^{\mathrm{S}} \texttt{Bool}^{y:\mathrm{S}}) \xrightarrow{y:\mathrm{S}} \texttt{Bool}.$$

This typing allows the identity to accept a boolean argument that uses y strictly when evaluated, and it tells us that the function uses its argument strictly and will use y strictly when called. The judgment also tells us that the function definition itself uses y lazily, since the use of the function's argument in the body is not evaluated if the function is not called.[2]

Unpacking a Pair. Consider the term **let** $(a, b) = x$ **in** b in a context where z has type Bool and x has type P (that is, $\texttt{Bool}^{z:\mathrm{S}} \times \texttt{Bool}^{z:\mathrm{L}}$). Assume also that x does not use z strictly when evaluated. The following term unpacks a pair and returns its second element, and CBN$^\gamma$ types it with

$$z : \texttt{Bool}, x : P \vdash_{\mathsf{CBN}} \textbf{let}\ (a, b) = x\ \textbf{in}\ b :^{z:\mathrm{L}, x:\mathrm{S}} \texttt{Bool}.$$

This judgment tells us that the term uses x strictly (since it is destructured by let-binding) but uses z lazily. This latter fact is a result of x's type P, which asserts that strict use of its second element results in lazy usage of z. When the term uses b (the second element of x) strictly, that usage therefore results in a lazy usage of z. Note that, as the variable b itself is local to this term, we do not need to report b's usage in the final typing judgment.

Now consider, instead, a term that returns x's first element: **let** $(a, b) = x$ **in** a. We could not type this term with the same judgment we previously used. P tells us that strict use of a causes a strict use of z, and therefore this term would have to be typed with the judgment

$$z : \texttt{Bool}, x : P \vdash_{\mathsf{CBN}} \textbf{let}\ (a, b) = x\ \textbf{in}\ a :^{z:\mathrm{S}, x:\mathrm{S}} \texttt{Bool}.$$

Indeterminate use. Finally, consider a term that returns either the first or second element of a P-typed variable x, depending on the value of another variable y:

$$\textbf{let}\ (a, b) = x\ \textbf{in}\ \textbf{if}\ y\ \textbf{then}\ a\ \textbf{else}\ b.$$

What type can we give to this term? Clearly it uses both y and x strictly, but the way it uses z depends on which branch of the **if** statement is taken. The success branch uses z strictly, while the failure branch uses it lazily, so we say that this term is *indeterminate* in its usage of z. As written, we cannot assign it a type, but we can make a small modification and produce another term like so:

$$\textbf{let}\ (a, b) = x\ \textbf{in}\ \textbf{if}\ y\ \textbf{then}\ (\textbf{sub}\ a)\ \textbf{else}\ (\textbf{sub}\ b).$$

The **sub** annotation makes explicit that we are using *subsumption* when typing this term; we forget information about how z is used in each branch (moving down the semilattice in Figure 4) to derive

$$z : \texttt{Bool}, y : \texttt{Bool}, x : P \vdash_{\mathsf{CBN}} \textbf{let}\ (a, b) = x\ \textbf{in}\ \textbf{if}\ y\ \textbf{then}\ (\textbf{sub}\ a)\ \textbf{else}\ (\textbf{sub}\ b) :^{y:\mathrm{S}, x:\mathrm{S}, z:?} \texttt{Bool}.$$

This judgment asserts that the term uses y and x strictly, but says that the usage of z is not known to be either strict or lazy. The explicit **sub** syntax is useful for the proof of Lemma 5.4, which requires

[2]Note that this typing also would mean that this identity function could not accept an argument that used y lazily, as the function's type would then be inaccurate about how its body uses y. Requiring function types to declare the effects associated with their arguments is restrictive, but prior work [55] has addressed this limitation via effect polymorphism [52, 70], and, as we discuss in Section 8, the same approach should work here. Polymorphism would be an orthogonal addition to CBN$^\gamma$, however, and we focus on monomorphic type systems here for clarity and simplicity.

$$\frac{x : \tau^{\gamma} \in \Gamma}{\Gamma \vdash_{\mathsf{CBN}} x :^{\gamma + x:\mathsf{S}} \tau} \text{ T-CBN-Var} \qquad \frac{\Gamma \vdash_{\mathsf{CBN}} e_1 :^{\gamma_1} \tau_1 \quad \Gamma \vdash_{\mathsf{CBN}} e_2 :^{\gamma_2} \tau_2}{\Gamma \vdash_{\mathsf{CBN}} (e_1, e_2) :^{\overline{\mathsf{L}}} \tau_1^{\gamma_1} \times \tau_2^{\gamma_2}} \text{ T-CBN-Pair}$$

$$\frac{\Gamma \vdash_{\mathsf{CBN}} e :^{\gamma'} \tau \quad \gamma \leq \gamma'}{\Gamma \vdash_{\mathsf{CBN}} \mathbf{sub}\ e :^{\gamma} \tau} \text{ T-CBN-Sub} \qquad \frac{\Gamma \vdash_{\mathsf{CBN}} e_1 :^{\gamma_1} \tau_1 \quad \Gamma, x : \tau_1^{\gamma_1} \vdash_{\mathsf{CBN}} e_2 :^{\gamma_2} \tau_2}{\Gamma \vdash_{\mathsf{CBN}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 :^{(\downarrow_x \gamma_2)} \downarrow_x \tau_2} \text{ T-CBN-Let}$$

$$\frac{\Gamma, x :^{\gamma_1} \tau_1 \vdash_{\mathsf{CBN}} e :^{\gamma_2, x:\alpha} \tau_2}{\Gamma \vdash_{\mathsf{CBN}} \lambda x.e :^{\overline{\mathsf{L}}} (x :^{\alpha} \tau_1^{\gamma_1}) \xrightarrow{\gamma_2} \tau_2} \text{ T-CBN-Abs} \qquad \frac{\Gamma \vdash_{\mathsf{CBN}} e_2 :^{\gamma_1} \tau_1 \quad \Gamma \vdash_{\mathsf{CBN}} e_1 :^{\gamma_3} (x :^{\alpha} \tau_1^{\gamma_1}) \xrightarrow{\gamma_2} \tau_2}{\Gamma \vdash_{\mathsf{CBN}} e_1\ e_2 :^{\gamma_2 + \gamma_3} \downarrow_x \tau_2} \text{ T-CBN-App}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{CBN}} e_1 :^{\gamma_1} \tau_1^{\gamma_1'} + \tau_2^{\gamma_2'} \qquad \tau = \downarrow_{x_1} \tau_1' = \downarrow_{x_2} \tau_2' \\ \Gamma, x_1 : \tau_1^{\gamma_1'}, \vdash_{\mathsf{CBN}} e_2 :^{\gamma_2, x_1:\alpha_1} \tau_1' \qquad \Gamma, x_2 : \tau_2^{\gamma_2'}, \vdash_{\mathsf{CBN}} e_3 :^{\gamma_2, x_2:\alpha_2} \tau_2'\end{array}}{\Gamma \vdash_{\mathsf{CBN}} \mathbf{case}\ e_1\ \mathbf{of\ inl}\ x_1 \to e_2,\ \mathbf{inr}\ x_2 \to e_3 :^{\gamma_1 + \gamma_2} \tau} \text{ T-CBN-Case}$$

Fig. 5. Typing rules for CBN$^{\gamma}$

inversion of the semantic judgment; without it the language would lose its syntax-directed character and would require additional work to make its semantic rules invertible. However, this syntax is included purely for technical convenience and does not change the system's expressiveness.

3.2 CBN$^{\gamma}$ Overview

Having worked through some examples to build intuition, we can begin to discuss the details of how CBN$^{\gamma}$ derives typing judgments. Strictness attributes are elements of a *preordered monoid* equipped with a $+$ operator to combine attributes and a $\leq$ comparator to order them. The $+$ operator has L as its identity and is defined in Table 1.

The $\leq$ comparator orders attributes according to the semilattice depicted in Figure 4, which is notably different from the order induced by $+$. Instead, $\leq$ orders attributes by information: one attribute is greater than another when it gives us more certainty about how a variable will be used.

$+$	S	?	L
S	S	S	S
?	S	?	?
L	S	?	L

Table 1. Definition of $+$

We lift $+$ and $\leq$ to operate over vectors γ pointwise, noting that $\leq$ forms a preorder over γs, where a vector γ_1 is only considered $\leq$ a vector γ_2 if every variable in γ_1 has an attribute that is $\leq$ the corresponding attribute in γ_2. So, for example, the vectors $x : ?, y : \mathsf{L}$ and $x : \mathsf{L}, y : ?$ are unrelated.[3]

CBN$^{\gamma}$'s type system incorporates effects γ into its typing judgment to track the strictness of each variable in the context; a selection of the typing rules can be found in Figure 5. We give a named presentation of these typing rules here, but our Rocq mechanization [71] uses a de Bruijn representation.

The simplest rule of interest is T-CBN-Var, which looks up x in the context Γ and produces the effect γ that was latent there. In addition to this effect, the rule also adds a strict usage of x to its output effect to reflect that the value stored in x was strictly used.

[3]This still endows γs with sufficient structure to be elements of a preordered monoid, a requirement for them to behave like effects [37].

Next, since let-binding is lazy, the T-CBN-Let rule does not realize the γ_1 effects of the let-bound expression e_1. Rather, it associates those effects with the bound variable x in the context, to be produced if that variable is strictly used later. The rule realizes only the effects γ_2 associated with e_2, as any strict uses of x in e_2 will already have produced γ_1, which will be included in γ_2.

Also of note is how T-CBN-Let handles scoping in its result. During the checking of e_2, we must have an effect in the context for x. However, once x goes out of scope, its usage is no longer of interest, and we drop its attributes from γ_2 and τ_2 via the $\downarrow_x$ operator. This operator traverses the structure of a type and removes any mappings for the variable x from all vectors in the type. To understand why this is reasonable, consider that our goal in modeling Definition 2.1 is to know the strictness of each function's argument—that is, whether an argument is used strictly or lazily by a function's body. Given this, the strict or lazy usage of a function's local variable is not of external interest and can be discarded when that variable goes out of scope. Similarly, once we leave the scope of a let expression, we no longer care to track the strictness of the variable it binds. A detailed example of this behavior can be found in Section 4.2.

The function introduction and elimination rules, T-CBN-Abs and T-CBN-App, are where we achieve the primary desideratum of the system: the ability to annotate arrow types with the strictness of the functions they classify. In the T-CBN-Abs rule, the body of the function is checked in a context where the input effect γ_1 is associated with the parameter x. This will realize effects for the body, which we can split into the portion γ_2 not mentioning x plus whatever α is associated with x directly, which tells us how x is used in the function body.

The function type used in this rule is dependent because CBN$^\gamma$ function types need to mention the variables they close over. Consider the function $\lambda x.\ \lambda y.\ x$, where x has type τ_1 and y type τ_2. This function is lazy in its argument; when partially applied it will yield the function $\lambda y.x$ and will not evaluate its argument. However, the inner function is strict in its free variable x, and its type must reflect that. It therefore must have type $(y :^{\mathrm{L}} \tau_2) \xrightarrow{x:\mathrm{S}} \tau_1$, which then requires us to type the overall function at $(x :^{\mathrm{L}} \tau_1) \to (y :^{\mathrm{L}} \tau_2) \xrightarrow{x:\mathrm{S}} \tau_1$. We see here that the return type of the overall function (that is, the type of the inner function) must be able to mention the variable that the overall function abstracts in order to properly describe its usage. On the other hand, x does *not* need to appear on the $\to$ portion of the outer function type, as x's lazy usage in the outer function body is already reflected in the L associated with the argument.

The T-CBN-App rule enforces that the effects realized by the argument expression e_2, namely γ_1, match the effects expected by the function. These effects are already included in the latent effects γ_2 of the function type, however. Therefore, as in the T-CBN-Let rule, they do not need to be added to the resulting effects for this rule. Instead, the rule adds together only the effects realized by the body of the function and the effects realized by the computation of the function e_1.

The T-CBN-App rule also manages its scope similarly to the T-CBN-Let rule by dropping the abstracted variable from the result of the function's return type after the call. The α on the applied function type does not appear in the rule's conclusion for similar reasons: the calling scope does not need to track how the function body uses its argument.

The last point of note concerns effect subsumption. Some imprecision is unavoidable in CBN$^\gamma$ as a result of the branching in the **case** expression. We could soundly restrict this imprecision to the T-CBN-Case rule, but we instead allow subsumption to happen anywhere via the T-CBN-Sub rule, as it allows us to type more programs.

Consider, for example, the standard Church encoding of the booleans, $\lambda x.\lambda y.x$ and $\lambda x.\lambda y.y$ [11]. To be a *usable* encoding of the booleans, both of these functions must inhabit the same type, but without subsumption we would be forced to assign them the types

$$\lambda x.\lambda y.x \in (x :^{\mathrm{L}} A) \to (y :^{\mathrm{L}} A) \xrightarrow{x:\mathrm{S}} A \quad \text{and} \quad \lambda x.\lambda y.y \in (x :^{\mathrm{L}} A) \to (y :^{\mathrm{S}} A) \xrightarrow{x:\mathrm{L}} A.$$

$$
\begin{array}{llll}
\textit{strictness attributes} & \alpha & ::= & \mathrm{S} \mid \mathrm{L} \mid ? \\
\textit{attribute vectors} & \gamma & ::= & \cdot \mid \gamma, x : \alpha \\
\textit{value types} & A & ::= & \mathtt{unit} \mid \mathbf{U}_\gamma B \mid A_1 \times A_2 \mid A_1 + A_2 \\
\textit{computation types} & B & ::= & A^\alpha \to B \mid \mathbf{F} A \\
\textit{contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A \\
 & & & \\
\textit{values} & V & ::= & () \mid x \mid \{M\} \mid \mathbf{inl}\ V \mid \mathbf{inr}\ V \mid (V_1, V_2) \\
\textit{computations} & M & ::= & \lambda x.\, M \mid M\ V \mid V! \mid x \leftarrow M_1\ \mathbf{in}\ M_2 \mid (x_1, x_2) \leftarrow V\ \mathbf{in}\ M \mid \mathbf{sub}\ M \\
 & & & \mid \mathbf{ret}\ V \mid V; M \mid \mathbf{case}\ V\ \mathbf{of}\ \mathbf{inl}\ x_1 \to M_1,\ \mathbf{inr}\ x_2 \to M_2
\end{array}
$$

Fig. 6. Syntax of CBPV$^\gamma$

The outer function type uses x lazily in both cases because the inner function suspends all usages in its body; if, for example, we were to partially apply the first function, this would not result in a strict usage of x because the result of that partial application would be a function that uses x in its body. But the types disagree on the strictness of y and on how the inner function uses x in its body.

The difference makes these types unsuitable for a boolean encoding. With subsumption, however, we can encode the Church booleans as $\lambda x.\lambda y.\mathbf{sub}\ x$ and $\lambda x.\lambda y.\mathbf{sub}\ y$ and assign both the less exact, but more useful, type[4]

$$(x :^{\mathrm{L}} A) \to (y :^{?} A) \xrightarrow{x:?} A.$$

The other rules (available in the extended version of this paper [72]) are straightforward, keeping in mind that in CBN$^\gamma$ all constructors are lazy. Hence, all the introduction rules have an $\overline{\mathrm{L}}$ effect and store any latent effects on the type of the value they produce.

4 Strictness in Call-By-Push-Value

We would like to prove that the type system of CBN$^\gamma$ is sound and yields the desired guarantees about usage. However, carrying out these proofs directly in CBN$^\gamma$ is awkward: many language features in call-by-name languages can suspend computation, so proofs about CBN$^\gamma$ will involve significant duplicated work. In particular, the logical relations used in Sections 4.4 and 5 must enforce certain invariants relating types and values whenever a computation is thunked. Instead of repeating these checks for every type, we would rather carry out our metatheoretic analysis in a language that isolates the thunking operation in one syntactic construct with a dedicated type to represent it. We can then translate CBN$^\gamma$ to this language in a type-preserving way.

Levy's call-by-push-value (CBPV) [47–51] has precisely the features we need. CBPV makes an explicit distinction between values and computations, separating the two into different syntactic classes according to the slogan "a value is, a computation does" [49]. In particular, it features a thunk value former, written $\{M\}$, that suspends a computation M as a value, and a forcing operation, written $V!$, that forces a thunk V by running its suspended computation. $\mathbf{U}B$ is the type of thunks that produce computations of type B when forced. The thunking construct allows CBPV, despite being strictly evaluated, to model lazily evaluated languages like CBN$^\gamma$.

This section describes CBPV$^\gamma$, an extension of CBPV with strictness tracking, and establishes the metatheoretic properties that make it a useful model of intensional strictness.

4.1 Syntax of CBPV$^\gamma$

The primary difference between CBPV$^\gamma$, given in Figure 6, and standard presentations of CBPV is the presence of an attribute vector γ on the $\mathbf{U}$ type former, tracking how variables are used when a

[4] An effect-polymorphic extension of CBN$^\gamma$ could instead type both terms polymorphically.

thunk is forced. A variable annotated with an L on a **U** type will be used lazily if a thunk inhabiting that type is forced, while a variable with an S will be used strictly. Function types also carry an attribute α describing how arguments are used when functions are applied.

Beyond the **U** type, CBPV$^\gamma$ features two classes of types to go with its two classes of terms. *Positive types* (ranged over by A) describe values, while *negative types* (written B) describe computations. Value types include **U** types along with the unit type, sums, and products. Computation types include function types and **F** types, which are dual to **U** types. The *returner type* $\mathbf{F}\ A$ describes computations returning values of type A. As before, for technical expedience we extend the usual presentation of CBPV with a **sub** term handling attribute subsumption.

4.2 Typing Rules for CBPV$^\gamma$

CBPV$^\gamma$ has two typing judgments: $\gamma \cdot \Gamma \vdash V : A$ for values and $\gamma \cdot \Gamma \vdash M : B$ for computations. The shapes of these judgments are somewhat different from the CBN$^\gamma$ typing judgment; instead of appearing on the : like an effect, γ is zipped with the context Γ on the left of the $\vdash$.[5] We can zip and unzip $\gamma \cdot \Gamma$ freely, i.e., we write $\gamma \cdot \Gamma, x :^\alpha A$ interchangeably with $(\gamma, x : \alpha) \cdot (\Gamma, x : A)$. This is different from how CBN$^\gamma$ contexts work, as those associate variables with both types and γs. This difference reflects the fact that CBPV$^\gamma$ is evaluated strictly, rather than lazily like CBN$^\gamma$, so the use of a variable in CBPV$^\gamma$ does not itself trigger any evaluation. Instead, this triggering occurs in the thunking and forcing rules, which suspend and resume computation. Accordingly, whereas in CBN$^\gamma$ all type constructors must be graded with attributes because all type constructors suspend computation, in CBPV only the **U** type is graded because only the **U** type suspends computation. We can think of CBN$^\gamma$ types as implicitly containing **U** types (as is made explicit by translation in Figure 11), and therefore all CBN$^\gamma$ types need to keep track of strictness attributes.

Note also that the definition of contexts requires that variables be bound to value types; binding a computation to a variable requires it to be explicitly thunked.

As we consider the rules, recall the second example from Section 3.1, where we described how CBN$^\gamma$ breaks the usual η-equivalence between x and $(\mathbf{fst}\ x, \mathbf{snd}\ x)$. The pair constructor in CBN$^\gamma$ implicitly thunks its two arguments; this is the reason why x is lazy in the latter term but not the former. To capture this behavior in CBPV$^\gamma$, the thunk value must change the attributes associated with the computation it suspends, breaking the η-equivalence between V and $\{V!\}$. In particular, we want CBPV$^\gamma$ to describe the term x as using the variable x strictly but $\{x!\}$ as using it lazily.

To type x properly, we must ensure that variable lookup produces an S attribute. This leads directly to the T-Var rule in Figure 7. The rule requires that only the variable being used has a strict attribute, isolating any imprecision in CBPV$^\gamma$ to the T-Sub rule.

To type $\{x!\}$, we need the thunking operation to make the use of x inside the thunk lazy. Accordingly, the T-Thunk rule suspends all the uses of the variables in M represented by γ and packages them into the **U** type, to be released later if the thunk is forced. The vector of attributes on the thunk itself is then set to the lazy $\overline{\mathrm{L}}$ vector, since a thunk value uses nothing strictly. So, for example, the value $\{\mathbf{ret}\ x\}$ has the type $\mathbf{U}_{x:\mathrm{S}}\ \mathbf{F}\ A$ when $x :^\mathrm{L} A \in \gamma \cdot \Gamma$.

The T-Force rule is more complex. Intuitively, the vector of attributes γ_1 is used to produce V, which has the type $\mathbf{U}_{\gamma_2} B$. This γ_2 represents the usages that will occur when V is forced, so those attributes are "replayed" here into the context by adding them to γ_1. So, were we to add an ! to our example from before to produce the computation $\{\mathbf{ret}\ x\}!$, this new program would have the type $\mathbf{F}\ A$ when $x :^\mathrm{S} A \in \gamma \cdot \Gamma$. The attributes from the thunk's type have been added to those used to produce it (in this case $\overline{\mathrm{L}}$).

[5] Some readers may notice a similarity to coeffect systems in the judgment's shape; we discuss this similarity in Section 4.6.

$\boxed{\gamma \cdot \Gamma \vdash V : A}$ *(Value typing)*

$$\frac{}{\overline{\mathsf{L}} \cdot \Gamma_1, x :^{\mathsf{S}} A, \overline{\mathsf{L}} \cdot \Gamma_2 \vdash x : A}\ \textsc{T-Var} \qquad \frac{\gamma \cdot \Gamma \vdash M : B}{\overline{\mathsf{L}} \cdot \Gamma \vdash \{M\} : \mathbf{U}_\gamma B}\ \textsc{T-Thunk}$$

$\boxed{\gamma \cdot \Gamma \vdash M : B}$ *(Computation typing)*

$$\frac{\gamma \cdot \Gamma \vdash V : A}{\gamma \cdot \Gamma \vdash \mathbf{ret}\ V : \mathbf{F}A}\ \textsc{T-Return} \qquad \frac{\gamma' \cdot \Gamma \vdash M : B \quad \gamma \leq \gamma'}{\gamma \cdot \Gamma \vdash \mathbf{sub}\ M : B}\ \textsc{T-Sub}$$

$$\frac{\gamma_1 \cdot \Gamma \vdash V : \mathbf{U}_{\gamma_2} B}{\gamma_1 + \gamma_2 \cdot \Gamma \vdash V! : B}\ \textsc{T-Force} \qquad \frac{\gamma_1 \cdot \Gamma \vdash M_1 : \mathbf{F}A \quad \gamma_2 \cdot \Gamma, x :^{\alpha} A \vdash M_2 : B}{(\gamma_1 + \gamma_2) \cdot \Gamma \vdash x \leftarrow M_1\ \mathbf{in}\ M_2 : \downarrow_x B}\ \textsc{T-Let}$$

$$\frac{\gamma_1 \cdot \Gamma \vdash V : \mathtt{unit} \quad \gamma_2 \cdot \Gamma \vdash M : B}{(\gamma_1 + \gamma_2) \cdot \Gamma \vdash V; M : B}\ \textsc{T-Seq}$$

$$\frac{\gamma \cdot \Gamma, x :^{\alpha} A \vdash M : B}{\gamma \cdot \Gamma \vdash \lambda x.M : A^{\alpha} \to \downarrow_x B}\ \textsc{T-Abs} \qquad \frac{\gamma_1 \cdot \Gamma \vdash M : A^{\alpha} \to B \quad \gamma_2 \cdot \Gamma \vdash V : A}{\gamma_1 + \gamma_2 \cdot \Gamma \vdash M\ V : B}\ \textsc{T-App}$$

Fig. 7. Main typing rules for CBPV$^{\gamma}$

With these three rules, we can see how CBPV$^{\gamma}$ differentiates x from $\{x!\}$. Assuming x is bound to a thunk value of type $\mathbf{U}B$, CBPV$^{\gamma}$ types the two programs with the derivations

$$x :^{\mathsf{S}} \mathbf{U}B \vdash x : \mathbf{U}_{x:\mathsf{L}} B \qquad \text{and} \qquad x :^{\mathsf{L}} \mathbf{U}B \vdash \{x!\} : \mathbf{U}_{x:\mathsf{S}} B$$

respectively. The strict use of x has been moved from the context in the former program onto the $\mathbf{U}$ type in the latter, describing how the use of x has been suspended by the thunk.

The T-Abs and T-App rules handle function type introduction and elimination. Lambda abstraction does not suspend the attributes of the body, so the γ necessary to check the M in $\lambda x.M$ "passes through" the function (e.g., $\lambda x.$ **ret** y is considered to use y strictly). Instead, since functions need to be thunked to be bound to variables, their attributes will be suspended by the thunk rule. This treatment of attributes in functions may appear surprising, but it is common practice in the literature on effects and coeffects in CPBV [35, 36, 80] and has the benefit of isolating all reasoning about the suspension and resumption of attributes to the rules for the $\mathbf{U}$ type. This behavior also means that, unlike CBN$^{\gamma}$, CBPV$^{\gamma}$'s arrows do not include their argument in the scope of their return type. Consider, for example, the function $\lambda x.\ \lambda y.$ **ret** x. The inner function uses x strictly, and, because this strict usage passes through the inner function to the outer, we can assign the latter the type $A_1^{\mathsf{S}} \to A_2^{\mathsf{L}} \to \mathbf{F}\ A_1$. Accordingly, when producing the result type in the T-Abs rule, we remove x from the arrow's return type B.

For a more interesting example, a function $\lambda x.(x; \mathbf{ret}\ y)$ can be typed as $\mathtt{unit}^{\mathsf{S}} \to \mathbf{F}\ A$ in a context where y has the type A and attribute S, since it uses both its argument and y strictly. Conversely, a function $\lambda x.$ **ret** $\{x; \mathbf{ret}\ y\}$ can be assigned type $\mathtt{unit}^{\mathsf{L}} \to \mathbf{F}\ \mathbf{U}_{y:\mathsf{S}}\ \mathbf{F}A$, since it uses its argument and y lazily and produces a thunk that will use y strictly if forced. As discussed above, the usage of x is not mentioned in the return type of the function; an external caller only cares about how the result of the function uses the variables that exist in the calling scope.

The T-Let rule has a different structure from its CBN$^{\gamma}$ counterpart, since CBPV$^{\gamma}$ evaluates strictly while CBN$^{\gamma}$ does not. Unlike T-CBN-Let, the vector γ_1 used in the evaluation of M_1 is not implicitly

$$
\begin{array}{lrcl}
\textit{closed terminal values} & W & ::= & () \mid (W, W) \mid \mathbf{inl}\ W \mid \mathbf{inr}\ W \mid \{\gamma, \rho, M\} \\
\textit{closed terminal computations} & T & ::= & \mathbf{ret}\ W \mid \langle\!\langle \gamma, \rho, \lambda x.M \rangle\!\rangle \\
\textit{environments} & \rho & ::= & \cdot \mid \rho, x \mapsto W
\end{array}
$$

Fig. 8. Closed terminal values, computations and environments in CBPV$^{\gamma}$

present in the context for the evaluation of M_2, so the T-Let rule must explicitly add it to the vector γ_2 used by M_2 to yield the resulting attributes for the entire term. The rule otherwise behaves like the T-CBN-Let rule, including in its handling of scoping.

To better understand this scoping, consider the program $z \leftarrow (x \leftarrow \mathbf{ret}\ ()\ \mathbf{in}\ \mathbf{ret}\ \{x; \mathbf{ret}\ y\})\ \mathbf{in}\ z!$, assuming y has type A. We would like this program to have the type $\mathbf{F}\ A$ and assign an S attribute to y. At the beginning of the program, the variable x is assigned a $()$ value, and hence will type at `unit`. Thus, the thunk value $\{x; \mathbf{ret}\ y\}$ has the type $\mathbf{U}_{x:\mathrm{S}, y:\mathrm{S}}\ \mathbf{F}\ A$ when x and y have an L attribute. However, by the time this thunk is forced, x has gone out of scope. Bearing in mind our goal of assigning S to y, however, we can see that x's usage inside the thunk is irrelevant to y; we can forget x from the type $\mathbf{U}_{x:\mathrm{S}, y:\mathrm{S}}\ \mathbf{F}\ A$ to yield $\mathbf{U}_{y:\mathrm{S}}\ \mathbf{F}\ A$, which has exactly the behavior we want.

The other rules are straightforward; they can be found in the extended version of this paper [72].

4.3 Big-Step Semantics of CBPV$^{\gamma}$

To prove that evaluation of CBPV$^{\gamma}$ reflects the strictness attributes computed by the type system, we enrich the standard semantics of CBPV$^{\gamma}$ with strictness-attribute tracking. We present the semantics in a big-step style, defining terminal values W and computations T in Figure 8, along with environments ρ. Like Γs, ρs can freely be zipped and unzipped with γs.

CBPV$^{\gamma}$'s two evaluation judgments have the form $\gamma \cdot \rho \vdash V \Downarrow W$ for values and $\gamma \cdot \rho \vdash M \Downarrow T$ for computations. The rules of particular relevance to strictness tracking are given in Figure 9.

Most of the complexity in the semantics of CBPV$^{\gamma}$ arises from how thunk values handle the scopes of their captured attribute vectors. In the E-Thunk rule, where thunk values are created, the existing environment is captured and a γ is chosen for the result. This rule is unusual in that it allows the choice to be any γ—to see why, remember that we are adding γ to the semantics only to enable metatheoretic reasoning; the semantics here is free to select whichever γ is necessary to evaluate this thunk, should it later be forced. However, by the time a thunk *is* forced, the scope of the vector it captured may no longer be the same as the scope in which it is being forced.

Concretely, in the E-Force rule, there are two different scopes in play. The outer scope, where γ_1 and ρ live, is the same scope in which the typing judgment will view a force computation. The inner scope, where γ_2 and ρ' live, is internal to the thunk value and may be arbitrarily different from the outer scope. Accordingly, the scope of γ_2 must be adjusted when adding it to γ_1 in the rule's conclusion; this adjustment is exactly the restriction of γ_2 to the domain of γ_1 (written $\gamma_2 \mid_{dom\ \gamma_1}$).

As an example, recall the program from earlier: $z \leftarrow (x \leftarrow \mathbf{ret}\ ()\ \mathbf{in}\ \mathbf{ret}\ \{x; \mathbf{ret}\ y\})\ \mathbf{in}\ z!$. Assuming ρ maps y to some value W, the bound term $\{x; \mathbf{ret}\ y\}$ reduces to a thunk value capturing its environment: $\{(y : \mathrm{S}, x : \mathrm{S}), (x \mapsto (), y \mapsto W), x; \mathbf{ret}\ y\}$. When that value is forced, however, only y is in scope, so we take the restriction of $y : \mathrm{S}, x : \mathrm{S}$ (i.e., $y : \mathrm{S}$) and add it to the attributes used to produce the thunk ($\overline{\mathrm{L}}$) to get the result $y : \mathrm{S}$, which agrees with the typing rules, as desired. Intuitively, x is local to the thunk here, so we do not care about its usage when forcing the thunk.

It is also worth pointing out the E-App rule, which mirrors prior work in CBPV but is potentially confusing nonetheless. As mentioned previously, CBPV$^{\gamma}$ functions do not suspend their attributes (also seen here in the E-Abs rule). Therefore, the application rule does not need to include the attributes γ_3 from the closure in the result of the application, as they will already be included in the attributes γ_1 needed to produce the closure in the first place.

$\boxed{\gamma \cdot \rho \vdash V \Downarrow W}$ *(Value semantics)*

$$\frac{}{\overline{\mathsf{L}} \cdot \rho_1, x \mapsto^{\mathsf{S}} W, \overline{\mathsf{L}} \cdot \rho_2 \vdash x \Downarrow W}\ \text{E-Var} \qquad \frac{}{\overline{\mathsf{L}} \cdot \rho \vdash \{M\} \Downarrow \{\gamma, \rho, M\}}\ \text{E-Thunk}$$

$\boxed{\gamma \cdot \rho \vdash M \Downarrow T}$ *(Computation semantics)*

$$\frac{\gamma' \cdot \rho \vdash M \Downarrow T \quad \gamma \leq \gamma'}{\gamma \cdot \rho \vdash \mathbf{sub}\ M \Downarrow T}\ \text{E-Sub} \qquad \frac{\gamma \cdot \rho \vdash V \Downarrow W}{\gamma \cdot \rho \vdash \mathbf{ret}\ V \Downarrow \mathbf{ret}\ W}\ \text{E-Return}$$

$$\frac{\gamma_1 \cdot \rho \vdash M_1 \Downarrow \mathbf{ret}\ W \quad \gamma_2 \cdot \rho, x \mapsto^{\alpha} W \vdash M_2 \Downarrow T}{(\gamma_1 + \gamma_2) \cdot \rho \vdash x \leftarrow M_1\ \mathbf{in}\ M_2 \Downarrow T}\ \text{E-Let} \qquad \frac{\gamma_1 \cdot \rho \vdash V \Downarrow \{\gamma_2, \rho', M\} \quad \gamma_2 \cdot \rho' \vdash M \Downarrow T}{\gamma_1 + (\gamma_2 \mid_{dom\ \gamma_1}) \cdot \rho \vdash V! \Downarrow T}\ \text{E-Force}$$

$$\frac{\gamma_1 \cdot \rho \vdash V \Downarrow () \quad \gamma_2 \cdot \rho \vdash M \Downarrow T}{(\gamma_1 + \gamma_2) \cdot \rho \vdash V; M \Downarrow T}\ \text{E-Seq} \qquad \frac{}{\gamma \cdot \rho \vdash \lambda x.M \Downarrow \langle\!\langle \gamma, \rho, \lambda x.M \rangle\!\rangle}\ \text{E-Abs}$$

$$\frac{\gamma_1 \cdot \rho \vdash M \Downarrow \langle\!\langle \gamma_3, \rho', \lambda x.M' \rangle\!\rangle \quad \gamma_2 \cdot \rho \vdash V \Downarrow W \quad \gamma_3 \cdot \rho', x \mapsto^{\alpha} W \vdash M' \Downarrow T}{\gamma_1 + \gamma_2 \cdot \rho \vdash M\ V \Downarrow T}\ \text{E-App}$$

Fig. 9. Main semantic rules for CBPV$^{\gamma}$

The other rules are straightforward; they can be found in the extended version of this paper [72].

4.4 Soundness of CBPV$^{\gamma}$

We can now prove that the attributes carried through the semantics reflect the attributes computed by the typing judgment. The proofs in this section, along with all those that follow, have been mechanized in Rocq [71]. These proofs require a surprising amount of scope bookkeeping and are most tractable with a logical relation. This relation, presented in Figure 10, resembles the one used by Torczon et al. [80] to prove coeffect soundness, differing only in the thunk and function cases.

In the thunk case, we need to handle the fact that the scopes in the **U** type and in the thunk value may have become "misaligned"; that is, variables may have been introduced to (or removed from) the type that are not present (or are still present) in the thunk value. However, recalling the earlier examples we used when defining the evaluation rule E-Force, we see that strictness tracking for values is only concerned with the attributes that exist in the scope using a value, not the scope that defined it. Thus, for the purposes of semantic typing, we care only to check that the attributes in the value agree with those in the type, when restricted to the domain that they share.[6]

With this logical relation in hand, we can define the usual notion of semantic typing in Figure 10. We state and prove the fundamental lemma for this relation and derive soundness as a corollary.

Lemma 4.1 (Fundamental Lemma: Soundness)**.** For all γ and Γ, if $\gamma \cdot \Gamma \vdash V : A$, then $\gamma \cdot \Gamma \vDash V : A$, and if $\gamma \cdot \Gamma \vdash M : B$, then $\gamma \cdot \Gamma \vDash M : B$.

[6]The mechanization of this proof uses well-scoped de Bruijn syntax, and as a result needs somewhat more elaborate bookkeeping in this case, since in a nameless setting it is not possible to determine the overlap between two arbitrary scopes. Instead, the scopes of values must be tracked explicitly and adjusted as they flow through the program. More details about this can be found in the artifact associated with this paper [71].

$$
\begin{array}{rcl}
\mathcal{W}[\![\texttt{unit}]\!] & = & \{()\} \\
\mathcal{W}[\![A_1 \times A_2]\!] & = & \{(W_1, W_2) \mid W_1 \in \mathcal{W}[\![A_1]\!] \text{ and } W_2 \in \mathcal{W}[\![A_2]\!]\} \\
\mathcal{W}[\![A_1 + A_2]\!] & = & \{\mathbf{inl}\ W_1 \mid W_1 \in \mathcal{W}[\![A_1]\!]\} \cup \{\mathbf{inr}\ W_2 \mid W_2 \in \mathcal{W}[\![A_2]\!]\} \\
\mathcal{W}[\![\mathbf{U}_\gamma B]\!] & = & \{\{\gamma', \rho, M\} \mid \gamma \mid_{dom\ \gamma'} = \gamma' \mid_{dom\ \gamma} \text{ and } \gamma' \cdot \rho \vdash M \Downarrow T \text{ and } T \in \mathcal{T}[\![B]\!]\} \\
\mathcal{T}[\![\mathbf{F}A]\!] & = & \{\mathbf{ret}\ W \mid W \in \mathcal{W}[\![A]\!]\} \\
\mathcal{T}[\![A^\alpha \to B]\!] & = & \{\langle\!\langle \gamma, \rho, \lambda x.M \rangle\!\rangle \mid \text{ for all } W \in \mathcal{W}[\![A]\!], \gamma \cdot \rho, x \mapsto^\alpha W \vdash M \Downarrow T \text{ and } T \in \mathcal{T}[\![B]\!]\}
\end{array}
$$

$$
\begin{array}{rcl}
\Gamma \vDash \rho & \triangleq & x : A \in \Gamma \implies x \mapsto W \in \rho \text{ and } W \in \mathcal{W}[\![A]\!] \\
\gamma \cdot \Gamma \vDash V : A & \triangleq & \Gamma \vDash \rho \implies \gamma \cdot \rho \vdash V \Downarrow W \text{ and } W \in \mathcal{W}[\![A]\!] \\
\gamma \cdot \Gamma \vDash M : B & \triangleq & \Gamma \vDash \rho \implies \gamma \cdot \rho \vdash M \Downarrow T \text{ and } T \in \mathcal{T}[\![B]\!]
\end{array}
$$

Fig. 10. Logical Relation and Semantic Typing for Soundness

Proof. By mutual induction on the typing derivations for values and computations. □

Theorem 4.2 (Soundness)**.** Given γ, Γ, and ρ such that $\Gamma \vDash \rho$,

(1) if $\gamma \cdot \Gamma \vdash V : A$, then there exists some W such that $\gamma \cdot \rho \vdash V \Downarrow W$;
(2) if $\gamma \cdot \Gamma \vdash M : B$, then there exists some T such that $\gamma \cdot \rho \vdash M \Downarrow T$.

Proof. Follows directly from the fundamental lemma. □

We can also show that the αs we place on arrow types accurately describe the strictness of function arguments. In other words, if a computation has a function type, then it must evaluate to a terminal closure. This closure, furthermore, will successfully evaluate in an extended environment where the α on the function type marks the strictness of the argument.

Theorem 4.3 (Function Type Soundness)**.** Given γ, Γ, A, B, α, and ρ such that $\Gamma \vDash \rho$, if $\gamma \cdot \Gamma \vdash M : A^\alpha \to B$, then there exists some γ', ρ' and M' such that $\gamma \cdot \rho \vdash M \Downarrow \langle\!\langle \gamma', \rho', \lambda x.M' \rangle\!\rangle$. Additionally, for any W such that $W \in \mathcal{W}[\![A]\!]$, there exists some T such that $\gamma' \cdot \rho', x \mapsto^\alpha W \vdash M' \Downarrow T$.

Proof. Follows directly from the fundamental lemma. □

4.5 CBN$^\gamma$ Translation

Having proved the soundness of CBPV$^\gamma$, we argue that it can be used as a model of strictness in CBN$^\gamma$, the call-by-name system from Section 3. We give semantics to CBN$^\gamma$ via Levy's standard translation [51] from the CBN lambda calculus to CBPV, which we extend to CBN$^\gamma$ and CBPV$^\gamma$.

The CBN$^\gamma$ typing rules are related those of CBPV$^\gamma$ by the translation in Figure 11, which is identical to Levy's save for the inclusion of γs and some additional bookkeeping in the type translation. Because of the difference in strictness tracking behavior between arrow types in CBN$^\gamma$ and CBPV$^\gamma$, we need to track the attributes suspended by the CBN$^\gamma$ arrow type separately, so we place them on any outer **U** that may occur in the type translation. Therefore, the type translation is a function $\tau \to B \times \gamma$, where the γ tracks any effects suspended in the CBN$^\gamma$ types that become unsuspended vectors in the resulting CBPV$^\gamma$ derivation.

We now state and prove our main lemma:

Lemma 4.4 (CBN$^\gamma$ Translation Correctness)**.** If $\Gamma \vdash_{\mathsf{CBN}} e :^\gamma \tau$ and $[\![\tau]\!] = (B, \gamma')$, then $\gamma + \gamma' \cdot [\![\Gamma]\!] \vdash [\![e]\!] : B$.

Proof. By induction on the CBN$^\gamma$ typing derivation. □

$$[\![\texttt{unit}]\!] = (\mathbf{F}\,\texttt{unit}, \overline{\mathrm{L}})$$

$$[\![(x :^{\alpha} \tau_1^{\gamma_1}) \xrightarrow{\gamma_2} \tau_2]\!] = ((\mathbf{U}_{\gamma_1+\gamma_1'} B_1)^{\alpha+\gamma_2'(x)} \to \downarrow_x B_2, \gamma_2 + \downarrow_x \gamma_2') \text{ where } [\![\tau_1]\!] = (B_1, \gamma_1') \text{ and } [\![\tau_2]\!] = (B_2, \gamma_2')$$

$$[\![\tau_1^{\gamma_1} \times \tau_2^{\gamma_2}]\!] = (\mathbf{F}\,(\mathbf{U}_{\gamma_1+\gamma_1'} B_1 \times \mathbf{U}_{\gamma_2+\gamma_2'} B_2), \overline{\mathrm{L}}) \text{ where } [\![\tau_1]\!] = (B_1, \gamma_1') \text{ and } [\![\tau_2]\!] = (B_2, \gamma_2')$$

$$[\![\tau_1^{\gamma_1} + \tau_2^{\gamma_2}]\!] = (\mathbf{F}\,(\mathbf{U}_{\gamma_1+\gamma_1'} B_1 + \mathbf{U}_{\gamma_2+\gamma_2'} B_2), \overline{\mathrm{L}}) \text{ where } [\![\tau_1]\!] = (B_1, \gamma_1') \text{ and } [\![\tau_2]\!] = (B_2, \gamma_2')$$

$$[\![\cdot]\!] = \cdot \qquad [\![\Gamma, x :^{\gamma} \tau]\!] = [\![\Gamma]\!], x : \mathbf{U}_{\gamma+\gamma'} B \text{ where } [\![\tau]\!] = (B, \gamma')$$

$$\begin{array}{lll}
[\![()]\!] & = \mathbf{ret}\ () & [\![x]\!] \quad = x! \\
[\![\lambda x.e]\!] & = \lambda x.[\![e]\!] & [\![e_1\ e_2]\!] \quad = [\![e_1]\!]\{[\![e_2]\!]\} \\
[\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2]\!] & = x \leftarrow \mathbf{ret}\ \{[\![e_1]\!]\}\ \mathbf{in}\ [\![e_2]\!] & [\![e_1; e_2]\!] \quad = x \leftarrow [\![e_1]\!]\ \mathbf{in}\ x; [\![e_2]\!] \\
[\![(e_1, e_2)]\!] & = \mathbf{ret}\ (\{[\![e_1]\!]\}, \{[\![e_2]\!]\}) & [\![\mathbf{sub}\ e]\!] \quad = \mathbf{sub}\ [\![e]\!] \\
[\![\mathbf{inl}\ e]\!] & = \mathbf{ret}\ (\mathbf{inl}\ \{[\![e]\!]\}) & [\![\mathbf{inr}\ e]\!] \quad = \mathbf{ret}\ (\mathbf{inr}\ \{[\![e]\!]\})
\end{array}$$

$$[\![\mathbf{let}\ (x_1, x_2) = e_1\ \texttt{in}\ e_2]\!] = x \leftarrow [\![e_1]\!]\ \mathbf{in}\ (x_1, x_2) \leftarrow x\ \mathbf{in}\ [\![e_2]\!]$$

$$[\![\mathbf{case}\ e_1\ \mathbf{of\ inl}\ x_1 \to e_2\ \mathbf{inr}\ x_2 \to e_3]\!] = x \leftarrow [\![e_1]\!]\ \mathbf{in\ case}\ x\ \mathbf{of\ inl}\ x_1 \to [\![e_2]\!]\ \mathbf{inr}\ x_2 \to [\![e_3]\!]$$

Fig. 11. Translation from CBN$^{\gamma}$ to CBPV$^{\gamma}$

The difference in the behavior of functions between CBN$^{\gamma}$ and CBPV$^{\gamma}$ causes a slight discrepancy in the translation: as seen in the statement of Lemma 4.4, the γ' from the translation of τ must be added to the γ in the CBN$^{\gamma}$ typing derivation to yield the strictness computed by CBPV$^{\gamma}$. However, inspection of the translation for types in Figure 11 shows that, at returner types, γ' is always $\overline{\mathrm{L}}$, the unit of our effect algebra. We thus state our main theorem at **F** types.

Theorem 4.5 (CBN$^{\gamma}$ Translation Preserves Strictness)**.** If $\Gamma \vdash_{\mathsf{CBN}} e :^{\gamma} \tau$ and $[\![\tau]\!] = (\mathbf{F}\ A, \gamma')$, then $\gamma \cdot [\![\Gamma]\!] \vdash [\![e]\!] : \mathbf{F}\ A$.

Proof. Using Lemma 4.4 and case analysis on τ. □

We can also show that the semantics of CBPV$^{\gamma}$ "interpret" CBN$^{\gamma}$ in a γ-preserving way.

Theorem 4.6 (CBN$^{\gamma}$ Interpretation)**.** If $\Gamma \vdash_{\mathsf{CBN}} e :^{\gamma} \tau$ and $[\![\tau]\!] = (\mathbf{F}\ A, \gamma')$, then, for any ρ such that $\rho \vDash [\![\Gamma]\!]$, there is some W such that $\gamma \cdot \rho \vdash [\![e]\!] \Downarrow \mathbf{ret}\ W$.

Proof. Simple composition of Lemma 4.1 and Theorem 4.5. □

With this, we have a strong claim that the type system we have developed for CBPV$^{\gamma}$ can model the CBN$^{\gamma}$ type-and-effect system we laid out in Section 3.

4.6 Aside: Are Attribute Vectors Effects?

One unusual facet of the relationship between CBN$^{\gamma}$ and CBPV$^{\gamma}$ is that γs behave like effects in CBN$^{\gamma}$ but lose this flavor when translated to CBPV$^{\gamma}$. This seems strange on its face, and it is worth discussing why we describe γs as effects in CBN$^{\gamma}$ but not in CBPV$^{\gamma}$.

Effects in CBN$^{\gamma}$. CBN languages generally prefer to isolate effects via monads to maintain purity, so effect systems for such languages are relatively uncommon. CBN$^{\gamma}$ does, however, resemble the call-by-name effect system Name described by McDermott and Mycroft [55]. In particular, while Name is a general effect system, if one were to instantiate that system with the γ effects we have described, the result would be strikingly similar to CBN$^{\gamma}$. Much like CBN$^{\gamma}$, Name has

typing contexts that associate variables with both types and effects, produces latent effects in its variable rule, and has effect annotations on both function parameters and bodies. The main difference between the systems is that the T-CBN-Var rule marks its variable as strict in the effect it produces—necessary for the specific effect that CBN$^\gamma$ is tracking.

Additionally, as noted above, CBN$^\gamma$'s γs form a preordered monoid, which, according to Katsumata [37] is necessary for them to be effects. The preorder is required to compare effects, while the monoid operator is necessary to combine them in function applications.

Coeffects in CBPV$^\gamma$? The rules of CBPV$^\gamma$, on the other hand, mostly do not resemble effect rules. The most obvious reason for this is that the typing judgment for values contains γs; recalling Levy's slogan for CBPV—"a value is, a computation does"—it would be quite strange for a value to produce an effect. On the other hand, the rules for CBPV$^\gamma$ do resemble the coeffect rules in prior work by Torczon et al. [80], except that CBPV$^\gamma$ tracks γs on the **U** type rather than the **F**.

Evidence for this connection can be found in the way our type system degenerates when instantiated for a call-by-value language without thunks. Since CBPV$^\gamma$ can also model CBV evaluation [47], one might wonder what the pre-image of the translation of CBV into CBPV$^\gamma$ would look like. However, since CBV does not suspend computation, a type system tracking lazy usage devolves into a relevance type system, which is known to be an instance of a coeffect system [2, 7, 66].

Another technical difference between general graded type systems and ours relates to the structure of the associated modality. In CBPV, the graded Box and Diamond modalities can be derived by composing the **U** and **F** types. Prior work [80] grades coeffects using the **F** type and effects using the **U** type. In contrast, our system tracks variable usage like a coeffect but grades the **U** type like an effect. It could be, however, that there is flexibility where grades appear (e.g. other prior work [69] grades the **F** type with effects), so the relationship is not completely clear.

Lastly, coeffect algebras feature a multiplication operator to scale coeffects, along with an addition operator to combine them, but CBPV$^\gamma$ lacks such a scaling operation. This is because the strictness algebra of CBPV$^\gamma$ is *non-quantitative* [5] (or *informational* [1]) and defines $+$ idempotently. Multiplication becomes degenerate in such a system, since it is just iterated addition. To make CBPV$^\gamma$'s rules more closely resemble coeffect rules, we could define a scaling operation $\cdot$ with S as its unit and L as its annihilator. However, including explicit scaling would not change any attributes computed by CBPV$^\gamma$, so there does not seem to be much point.

5 Semantic Properties of CBPV$^\gamma$

Our next job is to show that the intensional, type-theoretic characterization of strictness provided by CBPV$^\gamma$ refines the commonly understood extensional definition. To do this, we allow programs to be evaluated in environments that may not have bindings for every variable in the typing context. Such missing variables cannot be scrutinized in any valid semantic derivation tree (since the premise of the E-Var rule cannot be satisfied), so programs cannot use an unbound value strictly. (We deliberately remain agnostic as to how such variables may end up missing from the environment, so a variable without a binding can be used to model one given a $\bot$ value in the extensional model.)

We establish two different properties: *lazy soundness* and *strict failure*. First, any well-typed program can be run in an environment lacking bindings for any variables with an L attribute and still produce a result. Second, a well-typed program with an **F** type is guaranteed to fail when run in an environment lacking a value for any variable with an S attribute.

5.1 Lazy Soundness

To prove lazy soundness—that unbound lazy variables do not prevent successful execution—we use the logical relation in Figure 12, which is indexed by a set of variables X and a vector $\gamma_{\mathtt{Obs}}$. The set X

$$
\begin{aligned}
\mathcal{W}[\![\mathtt{unit}]\!]_X^{\gamma_{\mathsf{Obs}}} &= \{()\} \\
\mathcal{W}[\![A_1 \times A_2]\!]_X^{\gamma_{\mathsf{Obs}}} &= \{(W_1, W_2) \mid W_1 \in \mathcal{W}[\![A_1]\!]_X^{\gamma_{\mathsf{Obs}}} \text{ and } W_2 \in \mathcal{W}[\![A_2]\!]_X^{\gamma_{\mathsf{Obs}}}\} \\
\mathcal{W}[\![A_1 + A_2]\!]_X^{\gamma_{\mathsf{Obs}}} &= \{\mathbf{inl}\ W_1 \mid W_1 \in \mathcal{W}[\![A_1]\!]_X^{\gamma_{\mathsf{Obs}}}\} \cup \{\mathbf{inr}\ W_2 \mid W_2 \in \mathcal{W}[\![A_2]\!]_X^{\gamma_{\mathsf{Obs}}}\} \\
\mathcal{W}[\![\mathbf{U}_\gamma B]\!]_X^{\gamma_{\mathsf{Obs}}} &= \{\{\gamma', \rho, M\} \mid \gamma \mid_{dom\ \gamma'} = \gamma' \mid_{dom\ \gamma} \text{ and} \\
&\qquad (\gamma_{\mathsf{Obs}} \mid_X \leq_S \gamma \mid_X \implies \gamma' \cdot \rho \vdash M \Downarrow T \text{ and } T \in \mathcal{T}[\![B]\!]_X^{\gamma_{\mathsf{Obs}}})\} \\
\mathcal{T}[\![\mathbf{F}A]\!]_X^{\gamma_{\mathsf{Obs}}} &= \{\mathbf{ret}\ W \mid W \in \mathcal{W}[\![A]\!]_X^{\gamma_{\mathsf{Obs}}}\} \\
\mathcal{T}[\![A^\alpha \to B]\!]_X^{\gamma_{\mathsf{Obs}}} &= \{\langle\!\langle \gamma, \rho, \lambda x.M \rangle\!\rangle \mid \text{for all } W \in \mathcal{W}[\![A]\!]_X^{\gamma_{\mathsf{Obs}}}, \gamma \cdot \rho, x \mapsto^\alpha W \vdash M \Downarrow T \text{ and } T \in \mathcal{T}[\![B]\!]_X^{\gamma_{\mathsf{Obs}}}\}
\end{aligned}
$$

Fig. 12. Lazy Logical Relation

$$
\begin{aligned}
\gamma \cdot \Gamma \vDash_X^{\gamma_{\mathsf{Obs}}} \rho &\triangleq x :^\alpha A \in (\gamma \cdot \Gamma) \text{ and } \alpha \neq \mathrm{L} \implies x \mapsto W \in \rho \text{ and } W \in \mathcal{W}[\![A]\!]_X^{\gamma_{\mathsf{Obs}}} \\
\gamma \cdot \Gamma \vDash^{\mathrm{L}} \rho &\triangleq \gamma \cdot \Gamma \vDash_{dom\ \gamma}^{\gamma} \rho \\
X \vDash^{\mathrm{S}} \gamma_{\mathsf{Obs}} &\triangleq x \notin X \implies x : \mathrm{S} \in \gamma_{\mathsf{Obs}} \\
\gamma \cdot \Gamma \vDash^{\mathrm{L}} V : A &\triangleq \text{for all } X \text{ and } \gamma_{\mathsf{Obs}}, (\gamma_{\mathsf{Obs}} \cdot \Gamma \vDash_X^{\gamma_{\mathsf{Obs}}} \rho \text{ and } \gamma_{\mathsf{Obs}} \mid_X \leq_S \gamma \mid_X \text{ and } X \vDash^{\mathrm{S}} \gamma_{\mathsf{Obs}}) \implies \\
&\qquad \gamma \cdot \rho \vdash V \Downarrow W \text{ and } W \in \mathcal{W}[\![A]\!]_X^{\gamma_{\mathsf{Obs}}} \\
\gamma \cdot \Gamma \vDash^{\mathrm{L}} M : B &\triangleq \text{for all } X \text{ and } \gamma_{\mathsf{Obs}}, (\gamma_{\mathsf{Obs}} \cdot \Gamma \vDash_X^{\gamma_{\mathsf{Obs}}} \rho \text{ and } \gamma_{\mathsf{Obs}} \mid_X \leq_S \gamma \mid_X \text{ and } X \vDash^{\mathrm{S}} \gamma_{\mathsf{Obs}}) \implies \\
&\qquad \gamma \cdot \rho \vdash M \Downarrow T \text{ and } T \in \mathcal{T}[\![B]\!]_X^{\gamma_{\mathsf{Obs}}}
\end{aligned}
$$

Fig. 13. Lazy Semantic Typing

represents the outermost scope of the program, i.e., the scope of the top-level program declaration. This is the smallest scope that will ever exist during evaluation, since variables that are free at the top level can never go out of scope. We define also a relation $\leq_S$ on vectors that compares attributes pointwise and relates two vectors if the attributes in the lesser vector are "stricter" than those in the greater. That is, anywhere the greater vector is non-lazy, the lesser must be as well.

The γ_{Obs} functions like an "observer" context, describing how a variable can be used (as opposed to how it is actually used). This gives rise to the requirement that γ_{Obs} always be stricter than the γ on the type in the $\mathbf{U}$ case of the relation. Thunks in this relation may not evaluate to a value, but, if they do, they will use variables less strictly than what the γ_{Obs} vector allows. For example, the thunk $\{\mathbf{ret}\ x\} \in \mathcal{W}[\![\mathbf{U}_{x:\mathrm{S}}\mathbf{F}A]\!]_{\{x\}}^{x:\mathrm{L}}$, even if x is missing a value. The observer vector in this case tells us that x will not be used strictly at any point, so this thunk is never forced and accordingly is still "well typed" by this relation because $x : \mathrm{L} \not\leq_S x : \mathrm{S}$.

We define semantic typing for environments, values, and computations in Figure 13. Intuitively, an environment ρ is semantically well typed—by a vector γ and context Γ at a scope X and observer γ_{Obs}—if all the values associated with variables that have attributes other than L are well typed at their associated types in Γ at the same scope and observer.

A value or computation is semantically well typed by vector γ and context Γ if for every observer γ_{Obs}, environment ρ, and scope X, the value or computation can be evaluated to a well-typed result as long as ρ is well typed, γ_{Obs} is stricter than the vector γ of attributes used by evaluation, and every variable in γ_{Obs} that is not present in X is assigned an attribute of S, allowing it to be used.

This last requirement deserves some additional explanation. As mentioned, X describes the outermost scope of a program, and in the proof of Theorem 5.2 we will choose X to be this scope. For the purposes of induction, however, we must generalize to an arbitrary scope that exactly

$$\frac{}{() \equiv ()} \qquad \frac{W \equiv W'}{\mathbf{inl}\ W \equiv \mathbf{inl}\ W'} \qquad \frac{W \equiv W'}{\mathbf{inr}\ W \equiv \mathbf{inr}\ W'} \qquad \frac{W_1 \equiv W_1' \quad W_2 \equiv W_2'}{(W_1, W_2) \equiv (W_1', W_2')}$$

$$\frac{\rho \equiv \rho'}{\{\gamma, \rho, M\} \equiv \{\gamma', \rho', M\}} \qquad \frac{W \equiv W'}{\mathbf{ret}\ W \equiv \mathbf{ret}\ W'} \qquad \frac{\rho \equiv \rho'}{\langle\!\langle \gamma, \rho, M \rangle\!\rangle \equiv \langle\!\langle \gamma', \rho', M \rangle\!\rangle} \qquad \frac{\forall x, \rho(x) \equiv \rho'(x)}{\rho \equiv \rho'}$$

Fig. 14. Equivalence modulo γ

describes the variables that might be missing values in ρ. Every variable not in X must have been added to ρ during evaluation, and thus cannot be missing and can have an S attribute in γ_{Obs}.

Now we can state and prove the fundamental lemma, from which soundness follows directly.

Lemma 5.1 (Lazy Fundamental Lemma)**.** For all γ and Γ, if $\gamma \cdot \Gamma \vdash V : A$, then $\gamma \cdot \Gamma \vDash^{\mathrm{L}} V : A$, and if $\gamma \cdot \Gamma \vdash M : B$, then $\gamma \cdot \Gamma \vDash^{\mathrm{L}} M : B$.

Proof. By induction on the typing derivation. □

Theorem 5.2 (Lazy Soundness)**.** For all γ, Γ, and ρ such that $\gamma \cdot \Gamma \vDash^{\mathrm{L}} \rho$ and all x such that $x : \mathrm{L} \in \gamma$,

(1) if $\gamma \cdot \Gamma \vdash V : A$, then there is some W such that $\gamma \cdot (\rho - x) \vdash V \Downarrow W$;
(2) if $\gamma \cdot \Gamma \vdash M : B$, then there is some T such that $\gamma \cdot (\rho - x) \vdash M \Downarrow T$,

where $\rho - x$ denotes the environment equivalent to ρ with the binding for x removed.

Proof. Using the fundamental lemma, choosing γ_{Obs} to be γ and X to be $dom\ \gamma$. We also rely on the fact that $\gamma \cdot \Gamma \vDash^{\mathrm{L}} \rho - x$ when $x : \mathrm{L} \in \gamma$. □

This theorem tells us that if CBPV$^{\gamma}$ assigns a lazy attribute to a variable in some program, we can still evaluate that program in an environment where that variable is not bound to a value.

5.2 Strict Failure

To show that strictly using a missing value always causes evaluation to fail, we need a different, more complex logical relation. We also need a precise notion of what it means for evaluation to fail. In particular, it is possible for a semantic derivation in CBPV$^{\gamma}$ to fail because the wrong attributes were chosen for a non-deterministic rule like E-Thunk or E-Sub, rather than because a value is absent. This kind of failure is uninteresting, in the sense that evaluation could simply have made a different choice and produced a valid derivation tree; we want to argue that strict usage of a variable whose value is missing cannot *ever* produce a valid semantic derivation, regardless of how attributes are chosen. We denote this kind of failure with the $\lightning$ symbol.

We formalize a notion of derivations that are equivalent up to their choice of attributes by defining a relation $\equiv$ that relates two closed terminals if they are equivalent modulo their attributes. This definition is given in Figure 14. We then use this notion of equivalence modulo attributes to precisely express what it means for there to be no possible valid derivation tree for a given expression and environment.

Definition 5.3 (Semantic Failure)**.**

$$\rho \vdash V\lightning \quad \triangleq \quad \textit{for all } \gamma \textit{ and } \rho', \ \textit{if } \rho \equiv \rho' \ \textit{ then } \ \nexists W, \gamma \cdot \rho' \vdash V \Downarrow W$$
$$\rho \vdash M\lightning \quad \triangleq \quad \textit{for all } \gamma \textit{ and } \rho', \ \textit{if } \rho \equiv \rho' \ \textit{ then } \ \nexists T, \gamma \cdot \rho' \vdash M \Downarrow T$$

Intuitively, this definition says that a value or computation fails to evaluate in an environment ρ if, for any choice of attributes—both those in the derivation itself and also those appearing in the environment—there is no closed terminal that can be produced by any derivation. This rules out

$$
\begin{aligned}
\mathcal{W}[\![\mathtt{unit}]\!]_x &= \{()\} \\
\mathcal{W}[\![A_1 \times A_2]\!]_x &= \{(W_1, W_2) \mid W_1 \in \mathcal{W}[\![A_1]\!]_x \text{ and } W_2 \in \mathcal{W}[\![A_2]\!]_x\} \\
\mathcal{W}[\![A_1 + A_2]\!]_x &= \{\mathbf{inl}\ W_1 \mid W_1 \in \mathcal{W}[\![A_1]\!]_x\} \cup \{\mathbf{inr}\ W_2 \mid W_2 \in \mathcal{W}[\![A_2]\!]_x\} \\
\mathcal{W}[\![\mathbf{U}_\gamma B]\!]_x &= \{\{\gamma', \rho, M\} \mid \gamma \mid_{dom\ \gamma'} = \gamma' \mid_{dom\ \gamma} \text{ and } (\gamma', \rho, M) \in \mathcal{M}[\![B]\!]_x\} \\
\mathcal{T}[\![\mathbf{F}A]\!]_x &= \{\mathbf{ret}\ W \mid W \in \mathcal{W}[\![A]\!]_x\} \\
\mathcal{T}[\![A^\alpha \to B]\!]_x &= \{\langle\!\langle \gamma, \rho, \lambda x.M \rangle\!\rangle \mid \text{ for all } W \in \mathcal{W}[\![A]\!]_x, \\
&\qquad \rho, x \mapsto W \vdash M\lightning \text{ or } \gamma \cdot \rho, x \mapsto^\alpha W \vdash M \Downarrow T \text{ and } T \in \mathcal{T}[\![B]\!]_x\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{M}[\![\mathbf{F}A]\!]_x = \{(\gamma, \rho, M) \mid x : \mathrm{S} \in \gamma &\implies \rho \vdash M\lightning \text{ and} \\
x : \mathrm{S} \notin \gamma &\implies \rho \vdash M\lightning \text{ or } \gamma \cdot \rho \vdash M \Downarrow T \text{ and } T \in \mathcal{T}[\![\mathbf{F}A]\!]_x\} \\
\mathcal{M}[\![A^\alpha \to B]\!]_x = \{(\gamma, \rho, M) \mid x : \mathrm{S} \in \gamma &\implies \rho \vdash M\lightning \text{ or } (\gamma \cdot \rho \vdash M \Downarrow T \text{ and } T \in \mathcal{F}[\![\alpha, A, B]\!]_x) \text{ and} \\
x : \mathrm{S} \notin \gamma &\implies \rho \vdash M\lightning \text{ or } \gamma \cdot \rho \vdash M \Downarrow T \text{ and } T \in \mathcal{T}[\![A^\alpha \to B]\!]_x\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F}[\![(\alpha, A, \mathbf{F}A')]\!]_x &= \{\langle\!\langle \gamma, \rho, \lambda x.M \rangle\!\rangle \mid \text{ for all } W \in \mathcal{W}[\![A]\!]_x, \quad \rho, x \mapsto W \vdash M\lightning\} \\
\mathcal{F}[\![(\alpha, A, A'^{\alpha'} \to B)]\!]_x &= \{\langle\!\langle \gamma, \rho, \lambda x.M \rangle\!\rangle \mid \text{ for all } W \in \mathcal{W}[\![A]\!]_x, \rho, x \mapsto W \vdash M\lightning \text{ or} \\
&\qquad \gamma \cdot \rho, x \mapsto^\alpha W \vdash M \Downarrow T \text{ and } T \in \mathcal{F}[\![(\alpha', A', B)]\!]_x\}
\end{aligned}
$$

Fig. 15. Strict Logical Relation

trivial failures of evaluation that occur due to an incorrect choice of attributes and instead tells us about terms that fail to evaluate because they attempt to use a missing or ill-typed value.

We define the logical relation for the proof of strict failure in Figure 15. Unlike the lazy relation, the strict relation is indexed not by an entire attribute vector but rather by a single variable. We can understand this logical relation as searching the environment for the site of execution failure: the x variable indexing the relation is the location of the variable missing its value, and the relation guarantees that thunk values that are in it will necessarily fail when forced, if their attributes use x strictly. If a thunk's attributes are not S for x, the relation instead guarantees that forcing the thunk will either fail or produce a result that itself is in the relation.

Most of the complexity in this relation comes from the fact that attributes "pass through" lambda abstractions in CBPV$^\gamma$. This has precedent elsewhere in the CBPV literature [35, 36, 80], but it comes with an unfortunate consequence: despite the fact that, say, $\lambda x.$ **ret** y typechecks with attribute $y : \mathrm{S}$, evaluation of this program will not actually fail. Instead, it will produce a closure. In practice, to bind a function to a variable in CBPV$^\gamma$, it must be suspended via a thunk, which will restore the behavior we want. Therefore, we take the standard approach in the CBPV literature, stating and proving our top-level soundness theorem at returner types $\mathbf{F}A$ only. However, for purposes of induction the actual relation itself must be fully general.

To achieve this generality, we define two different helper relations $\mathcal{M}[\![\cdot]\!]_x$ (for *maybe*) and $\mathcal{F}[\![\cdot, \cdot, \cdot]\!]_x$ (for *failure*). The former describes computations that *may* fail depending on their usage of x, while the latter describes closures that *definitely* fail due to their usage of x. On function types returning **F** types, this latter relation guarantees that evaluation of the closed-over function body actually fails, while on function types returning other functions it instead guarantees that evaluation of the function body will either fail or produce another closure that is also in $\mathcal{F}[\![\cdot, \cdot, \cdot]\!]_x$.

$$
\begin{array}{rcl}
\gamma \cdot \Gamma \vDash^{\lightning}_{x} \rho & \triangleq & x : \mathrm{S} \in \gamma \text{ and } x \notin \rho \text{ and } (y \neq x \implies \rho(y) \in \mathcal{W}[\![\Gamma(y)]\!]_x) \\
\gamma \cdot \Gamma \vDash^{?}_{x} \rho & \triangleq & x : \mathrm{S} \notin \gamma \text{ and } x \notin \rho \text{ and } (y \neq x \implies \rho(y) \in \mathcal{W}[\![\Gamma(y)]\!]_x) \\
\gamma \cdot \Gamma \vDash^{\mathrm{S}} V : A & \triangleq & \gamma \cdot \Gamma \vDash^{\lightning}_{x} \rho \implies \rho \vdash V \lightning \text{ and} \\
 & & \gamma \cdot \Gamma \vDash^{?}_{x} \rho \implies \rho \vdash V \lightning \text{ or } \gamma \cdot \rho \vdash V \Downarrow W \text{ and } W \in \mathcal{W}[\![A]\!]_x \\
\gamma \cdot \Gamma \vDash^{\mathrm{S}} M : \mathbf{F}A & \triangleq & \gamma \cdot \Gamma \vDash^{\lightning}_{x} \rho \implies \rho \vdash M \lightning \text{ and} \\
 & & \gamma \cdot \Gamma \vDash^{?}_{x} \rho \implies \rho \vdash M \lightning \text{ or } \gamma \cdot \rho \vdash M \Downarrow T \text{ and } T \in \mathcal{T}[\![\mathbf{F}A]\!]_x \\
\gamma \cdot \Gamma \vDash^{\mathrm{S}} M : A^{\alpha} \to B & \triangleq & \gamma \cdot \Gamma \vDash^{\lightning}_{x} \rho \implies \rho \vdash M \lightning \text{ or } \gamma \cdot \rho \vdash M \Downarrow T \text{ and } T \in \mathcal{F}[\![A^{\alpha} \to B]\!]_x \text{ and} \\
 & & \gamma \cdot \Gamma \vDash^{?}_{x} \rho \implies \rho \vdash M \lightning \text{ or } \gamma \cdot \rho \vdash M \Downarrow T \text{ and } T \in \mathcal{T}[\![A^{\alpha} \to B]\!]_x
\end{array}
$$

Fig. 16. Strict Semantic Typing

Figure 16 defines strict semantic typing and formalizes the intuition for the logical relation. An environment can either be $\vDash^{\lightning}_{x}$ (well typed, but missing a strictly-used value at x) or $\vDash^{?}_{x}$ (well typed, but missing a possibly-used value at x) for a given γ and Γ. A semantically well-typed value or computation will definitely fail (or be in the failure relation, in the case of functions) if run with an environment that is missing a strictly-used value at x, and it will either fail or produce a well-typed result if evaluated with an environment that is missing a possibly-used value at x.

As in the lazy case, the fundamental lemma derives the main theorem as a corollary.

Lemma 5.4 (Strict Fundamental Lemma)**.** For all γ and Γ, if $\gamma \cdot \Gamma \vdash V : A$ then $\gamma \cdot \Gamma \vDash^{\mathrm{S}} V : A$, and if $\gamma \cdot \Gamma \vdash M : B$ then $\gamma \cdot \Gamma \vDash^{\mathrm{S}} M : B$.

Proof. By induction on the typing derivation. □

Theorem 5.5 (Strict Failure)**.** For all x, γ, Γ, and ρ such that $\gamma \vDash^{\lightning}_{x} \rho$,

(1) if $\gamma \cdot \Gamma \vdash V : A$ then $\rho \vdash V \lightning$;
(2) if $\gamma \cdot \Gamma \vdash M : \mathbf{F}\, A$ then $\rho \vdash M \lightning$.

Proof. Follows directly from the Fundamental Lemma 5.4. □

This gives us the opposite guarantee from Theorem 5.2, telling us that any well-typed program will fail when run in an environment where one of its strictly-used variables is missing a value. We can also use this theorem to argue that arguments to strict functions can be evaluated eagerly. Informally, if some strict function is called with a thunk argument and produces a result, Theorem 5.5 tells us by contraposition that the function necessarily uses the value of its argument—the value cannot be missing. Hence, forcing the thunked argument must produce a result, and forcing this argument before applying the function would correspond to a strict function call.

5.3 A More Precise Characterization of Strictness

Having proved that CBPV$^{\gamma}$'s type-theoretic characterization of intensional strictness also captures the traditional, extensional notion, we can move on to consider the *new* insight the intensional definition affords us: the ability to distinguish between functions that fail when "called on bottom" because their argument is used strictly and those that fail independently of their argument.

The traditional abstract interpretation [60, 84] and projection-based definitions [86] of extensional strictness assert that a function f is strict if $f\bot = \bot$, where $\bot$ denotes the result of a non-terminating

or aborting computation. Usually no distinction is made between the possible ways $\bot$ can be produced; the definition says only that a strict function fails whenever its argument fails.

The properties described by Theorem 5.2 and Theorem 5.5 are clearly related to this definition. As no $\bot$ value can ever actually exist, we can represent it with an environment that is missing a binding for the variable to which the traditional approach would ascribe a $\bot$ value. Theorem 5.5 then gives us a definition of strictness that implies $f\bot = \bot$: a strict function whose argument is missing its value will fail to evaluate. Likewise, Theorem 5.2 gives us a definition of laziness that implies that lazy functions do not scrutinize their arguments: a lazy function that is given an argument whose value is missing will still evaluate.

However, intensional strictness as described by CBPV$^\gamma$'s type system has some extra precision compared to extensional strictness. Consider a hypothetical program $y \leftarrow \mathbf{ret}\ \{\ldots error \ldots\}\ \mathbf{in}\ \lambda x.y!$. The function produced by this program—call it f—would be considered strict, extensionally, even though it does not use its argument at all. This characterization is useful to an optimizing compiler performing strictness analysis (i.e., it would allow such a compiler to evaluate this function's argument eagerly), but it fails to describe how f actually uses its argument.

Unlike the extensional definition of strictness, CBPV$^\gamma$ does distinguish between functions that fail due to strict use of their arguments and those that fail for other reasons. CBPV$^\gamma$ would describe f as being lazy in its argument, rather than strict, and also tell us that f uses y strictly. In a situation where a call to f fails, this additional nuance allows us to reason about *why* it has failed: CBPV$^\gamma$ tells us that this failure cannot be the result of the function's use of its argument (due to Theorem 5.2) and instead tells us that it must be the result of its use of y (due to Theorem 5.5). This is useful information for programmers to have; if, say, a call to a lazy function were to throw an exception, a programmer could be certain that the source of that exception was in the function body, not in the argument to the call.

This increased precision regarding sources of failure also makes intensional strictness better suited to reasoning about impure languages where exceptions are not encapsulated in monads as in Haskell. In such settings, the extensional definition fails to provide a satisfying characterization of strictness because it conflates aborting and non-terminating programs; it would thus falsely identify a call to f as an opportunity for eager evaluation, when such an optimization may in fact change observable behavior if the argument does not terminate.

It is also worth noting that we can apply these insights to the call-by-need [44, 88] evaluation strategy used by languages like Haskell, as call-by-need is equivalent to call-by-name [60]: the former is a more efficient implementation of the latter that uses values with the same strictness.

6 Tracking Unused Variables

The L attribute described so far is not as precise as it could be. Unlike the S attribute, which guarantees that variables are definitely used strictly, the L attribute describes both variables that are used lazily and those that are not used at all. It would be more accurate to describe the L attribute as asserting that a variable is definitely *not* used strictly, as opposed to definitely used lazily.

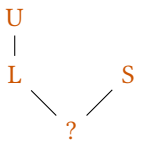

Fig. 17. Extended semilattice

In this section, we show how to add additional precision to the system by extending CBPV$^\gamma$ and CBN$^\gamma$ with a new attribute U to track variables that are known to be *unused*. This fourth attribute is related to the notion of *absence* described by *absence analysis* [75]. It extends the existing set to produce a new semilattice, depicted in Figure 17. The L, S, and ? attributes have the same meaning as before, and the intuition for $\leq$ remains the same as well: U tells us strictly more information about a variable's usage (or lack thereof) than L.

$$\frac{\gamma \cdot \Gamma \vdash M : B}{\mathbb{L}(\gamma) \cdot \Gamma \vdash \{M\} : \mathbf{U}_\gamma B}\ \text{T-Thunk-Ext} \qquad \frac{}{\mathbb{L}(\gamma) \cdot \rho \vdash \{M\} \Downarrow \{\gamma, \rho, M\}}\ \text{E-Thunk-Ext}$$

Fig. 18. Static and semantic rules for thunks in CBPV$^\gamma$ with unused variable tracking

6.1 Extending CBPV$^\gamma$

We present the extension of CBPV$^\gamma$ first, as it is simple. U, rather than L, becomes the new identity of the $+$ operator, while the result of adding any two of the previously existing attributes remains the same as in Table 1. We introduce the shorthand $\overline{\mathrm{U}}$ for the vector mapping all variables to U.

We do, however, need to introduce a new operation on αs and γs. When suspending a computation in the T-Thunk rule of the original CBPV$^\gamma$, it was sufficient to simply use the lazy vector $\overline{\mathrm{L}}$ for the required attributes, as no variables are used strictly by a thunked computation. If we wish to be more precise, however, it is inaccurate to say that a thunk that does not use a variable x in its body is lazy in x; we must instead check the thunk construct with a vector that both is lazy with respect to the thunk body and does not introduce any new usages.

To achieve this, we introduce a new operation $\mathbb{L}(\cdot)$ (read "lazify") that makes a usage lazy. $\mathbb{L}(\alpha) = \mathrm{L}$ for all α except U; instead, $\mathbb{L}(\mathrm{U}) = \mathrm{U}$ because suspending an unused variable does not introduce a usage of that variable. This new operation lifts pointwise to attribute vectors. We use this new operation to define new rules for thunks, presented in Figure 18. All the other rules remain unchanged, save for replacing $\overline{\mathrm{L}}$ with $\overline{\mathrm{U}}$ in the rules for variables and $()$.

This extension to CBPV$^\gamma$ enjoys all the metatheoretic properties proven previously, without significant changes to any proofs or definitions. We therefore elide these proofs for brevity, noting only that Theorem 5.2 also applies to variables with a U attribute in addition to those with an L.

The simplicity of this extension is a compelling demonstration of the benefits of reasoning about variable usage and strictness in CBPV, rather than in CBN directly.

6.2 Extending CBN$^\gamma$

Tracking unused variables in CBN$^\gamma$ requires significantly more effort due to implicit thunking. In particular, types in the extended CBN$^\gamma$ must satisfy a certain well-formedness condition: a type cannot claim that it uses more variables than those used by the derivation that produces it.

Consider the term (x, y). Assuming x and y both have the type `unit`, we can assign this term type $\texttt{unit}^{x:\mathrm{S},\, y:\mathrm{U}} \times \texttt{unit}^{x:\mathrm{U},\, y:\mathrm{S}}$, producing effect $x : \mathrm{L}, y : \mathrm{L}$. This makes intuitive sense: despite the fact that each side of the pair only uses one of the two variables, the pair as a whole uses both x and y. It would be impossible, on the other hand, to produce this type while not using either of the two variables; to create either side of this pair type, we would need an expression that uses one of the two strictly. In general, typing derivations that look like $\Gamma \vdash e : {}^{x:\mathrm{U},\ldots}\, \tau_1^{x:\mathrm{L},\ldots} \times \tau_2^{\gamma}$ are not possible: there is no way to produce a type that uses x without using x in its derivation.

Types produced by the CBN$^\gamma$ typing rules should have this property, but unfortunately there are a handful of rules where types are chosen without an accompanying subderivation, such as the rules for **inl** or **inr**. In these cases, the rules must enforce that any such types are well formed and satisfy particular usage conditions to ensure that all derivations permitted by the CBN$^\gamma$ typing judgment contain well-formed types. The formal definition of these well-formedness requirements, along with a selection of the extended rules for CBN$^\gamma$, can be found in the extended version of this paper [72].

The translation between CBN$^\gamma$ and CBPV$^\gamma$ needs only a small adjustment: the type translation presented in Figure 11 must lazify all the intermediate effects on each type rather than returning $\overline{\mathrm{L}}$

for any non-function types. With this modification, the translation enjoys the same correctness properties described in Section 4.5 for all well-formed CBN$^{\gamma}$ derivations.

7 Related Work

Strictness analysis was first presented by Mycroft [60]. His initial work laid the foundations for strictness analysis using abstract interpretation [12], but it was limited to programs working with simple data for which it sufficed to use an abstract domain containing just a $\top$ (defined) and a $\bot$ (undefined) element. The original formulation was extended to a four-point domain to better handle list programs [84]. In addition to fully defined and undefined values, the four-point domain enabled reasoning about data that was itself defined but that might contain subcomponents that were not (e.g., a fully-defined list whose elements were undefined).

Wadler and Hughes [86] improved upon earlier techniques using *projections* [33] to handle significantly more complex programs and bypass the need for abstract interpretation. Projection-based analysis involves reasoning backwards to determine how defined a program's inputs need to be, based on how its outputs are used; it was first implemented in practice by Kubiak et al. [41] and it now forms the foundation for strictness analysis in GHC [75]. Other approaches have been found that outperform GHC, such as the "Optimistic Evaluation" strategy [18], but these have proven too complex for practical adoption.

Demand analysis [45] generalizes strictness analysis by allowing compilers to reason about "how much" of a value is used (or "demanded"). It has been adapted to provide fine-grained analysis of performance in non-strictly evaluated languages [6, 91] and to guide compiler optimizations [76]. CBN$^{\gamma}$'s types also describe demand beyond the top-level constructor; the suspended γs that appear in types describe how variables get used as more of the result of a term is demanded.

Strictness in Types. Usage type systems [81, 89, 90] such as linear types [4, 24, 53, 85] are used to track how variables and values are used in programs. Attempting to apply such systems to modeling strictness, however, breaks down almost immediately when considering variables that appear in the bodies of functions. For example, such systems would just describe `f4` from the examples in Figure 2 as using its argument y, failing to capture the fact that `y` is not strictly used.

Approaches based on information flow [16, 64, 83, 92, 93] have the opposite problem. Unlike usage typing, which underapproximates strictness, information-flow typing overapproximates it. While these systems can distinguish the ways in which `f1` and `f4` in Figure 2 use their arguments, they also distinguish between variable usages that *should* be considered equivalent from a strictness perspective. For example, an information-flow system would say that **if** `y` **then** `1` **else** `2` leaks information about `y` and that `y` `` `seq` `` `1` does not, since the latter produces the same result regardless of `y`'s value. It is clear, however, that both of these programs use `y` strictly, so information flow is not sufficient for modeling strictness either.

There are, however, three existing source-level approaches to tracking strictness. Kuo and Mishra [42, 43] describe a constraint-gathering static analysis they call "strictness types." Schrijvers and Mycroft [74] describe an effect system for modeling strictness and use it to guide a handful of program optimizations. Both of these approaches, however, are limited to analysis of flat data (i.e., numbers or other base types) and thus are not sufficiently expressive for our purposes.

Barendsen and Smetsers [3] and Smetsers and van Eekelen [77] outline a type system modeling strictness using attributes annotated on types. Their system features a two-point lattice containing a ! attribute for strict usage and a ? attribute for uses lacking strictness information. This approach allows function types to be annotated with usage information (e.g., a strict function from A to B would be typed $A^{!} \to B$), but it only tells half the story. Guarantees can only be made about strict

usage (i.e., types for which a ! attribute is inferred) whereas CBN$^\gamma$ and CBPV$^\gamma$ can make guarantees about the evaluation of lazy functions as well.

Effects and Coeffects. Type-and-effect systems are well-studied tools for modeling side effects at the level of type systems [37, 52, 78, 87]. The study of coeffects is relatively newer, having been first introduced by Petricek et al. [65] in 2013, but it has been an area of active study for the past decade [8, 10, 23, 63, 66]. Coeffects have been used to track properties like differential privacy [68], error bounds [39], irrelevance in dependent type theories [2], and resource usage [4, 5, 10, 80].

The design of CBN$^\gamma$ and CBPV$^\gamma$ resembles coeffect-graded systems, such as those described by Bianchini et al. [5] and Choudhury et al. [10]. However, while grade algebras can describe properties similar to laziness and strictness, it is not clear whether coeffect-graded systems generalize our work. In particular, CBN$^\gamma$ types describe how variables will be used if the values inhabiting those types are used. In a graded system, on the other hand, grades describe only how values are used, not the downstream effects of such usage. CBN$^\gamma$ is more detailed because it annotates types with an entire grade vector instead of a single grade annotation on a graded modal type, but it is possible that a clever choice of grade algebra may be able to capture the same properties. Further work may provide a more rigorous understanding of the relationship between these systems.

The interaction between effects and coeffects is an active research area [13, 22]. Nanevski [61] presents a type system that uses modal types to model local and global state in a manner analogous to effects and coeffects. Of particular relevance to our work is that the $\Box$ modality behaves like the **U** type in CBPV$^\gamma$, in the sense that it both suspends computation and tracks local variable usage within the type, which has the flavor of a coeffect. Gaboardi et al. [22] use graded modal types to describe a type system in which effects and coeffects can interact with one another according to distributive laws, while Hirsch and Tate [32] investigate how effects and coeffects interact when distributive laws between them do not exist. The latter work is especially relevant because their choice of how to layer effects and coeffects semantically results in either strict or lazy evaluation, although they do not surface this information statically. Lastly, McDermott and Mycroft [55] blend coeffects with effects to implement an effect system for a call-by-need language similarly to CBN$^\gamma$.

Call-By-Push-Value. Call-by-push-value and its translations from call by name and call by value are due to Levy [47–51]. It is frequently used as a substrate for studying effects [35, 36, 54, 69] and their interplay with other systems, including evaluation order [56], dependent types [67], and time complexity [38]. Torczon et al. [80] extend CBPV with support for both effects and coeffects; their presentation and mechanization of this system heavily influenced this paper. They, in turn, drew from Forster et al. [20]'s mechanization of CBPV in Rocq.

8 Conclusion and Future Work

We have presented a new intensional definition of strictness that refines the usual extensional definition by describing usage more precisely. We introduced CBN$^\gamma$ and CBPV$^\gamma$, type systems that embody our definition using effects, related these systems via a type-preserving translation, and proved that they capture appropriate semantic notions of strict and lazy usage.

In the future, we hope to investigate whether CBN$^\gamma$ could be used in practice to typecheck real programs in languages like Haskell. In its current presentation, the annotation burden in the CBN$^\gamma$ type system is quite severe—expecting programmers to annotate types with the usage of every variable in scope does not seem reasonable! Prior work by Wansbrough [89], however, describes implementing a type system with a similar annotation burden practically in GHC; we plan to explore whether the principles guiding that work can be applied to CBN$^\gamma$.

To work for realistic Haskell programs, a few refinements to CBN$^\gamma$'s type system would be needed. As mentioned in Section 3, the requirement that CBN$^\gamma$ function types declare the effects

that they allow their arguments to produce is quite restrictive. Existing work on systems similar to CBN$^\gamma$ [55] addresses this limitation using effect polymorphism [52, 70], and we believe the same approach would work here. This would involve building out the theory and metatheory of effect-polymorphic extensions of CBN$^\gamma$ and CBPV$^\gamma$. Rioux and Zdancewic [69] describe CBPV$^\forall$, a type-polymorphic extension to CBPV, providing a promising foundation for this endeavor.

Further, our type system would need to support more complex datatypes, such as lists. Indeed, we believe the main principles should extend relatively straightforwardly. Aside from the usual **nil** and **cons** constructors for introducing lists, we would add a **fold** f *lst acc* operation to consume them. The strictness annotations on the type inferred for f would describe how the list and accumulator are used: if f has an S attribute for both arguments, then the list contents are all strictly used (e.g., the `sum` example from Figure 1), whereas an L attribute for the list element argument and an S attribute for the accumulator would result in only the "spine" of the list being strictly used (e.g., the `length` function). An L attribute for the accumulator would mean that the fold does not even use the spine of the list strictly (e.g., `map`).

We would also like to explore whether CBPV$^\gamma$ can be used to guide compiler optimizations. CBPV is often used as a reasoning tool for intermediate languages [17, 58, 62], so the groundwork has already been laid. We plan to build a compiler for a fragment of Haskell into CBPV$^\gamma$ to investigate what transformations can be applied based on intensional strictness in this setting.

Additionally, as mentioned previously, the call-by-name insights provided by this work extend to call-by-need evaluation, as the two strategies use variables with the same strictness. However, there is interesting future work in actually formalizing this relationship, and in particular in working out the details of how to model call-by-need evaluation in CBPV. One option would be to use the Extended Call-By-Push-Value system of McDermott and Mycroft [56], which changes the evaluation strategy of CBPV itself to be call-by-need. Another option would be to encode call-by-need evaluation via translation to CBPV by extending the latter with mutable references. In such a translation, we conjecture that all value types A would be translated to references to sum types $\mathbf{ref}(\mathbf{U}\mathbf{F}A + A)$, and evaluating thunks the first time would translate to storing their resulting value in the reference in place of the previously thunked expression.

Beyond strictness, the characterization of γs in CBN$^\gamma$ and CBPV$^\gamma$ deserves further attention: we would like to study more formally the connection between CBPV$^\gamma$'s γs and coeffects. We have also noted the potential relationship between these types and the logic of modal necessity, and we are eager to learn more about the extent and nature of this relationship.

Acknowledgements

We thank the anonymous reviewers for their feedback and suggestions. We also thank José Manuel Calderón Trilla and Simon Peyton-Jones for their valuable suggestions about how to best frame and explain the contributions of the paper. Lastly, we thank Jonathan Chan, Yiyun Liu, Nick Rioux, Cassia Torczon, and PLClub at large for feedback on the paper and help with formalization.

This material is based upon work supported by the U.S. National Science Foundation under Grants No. 2313998 and 2402449. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. National Science Foundation.

References

[1] Andreas Abel and Jean-Philippe Bernardy. 2020. A unified view of modalities in type systems. 4 (2020), 1–28. Issue ICFP. doi:10.1145/3408972

[2] Andreas Abel, Nils Anders Danielsson, and Oskar Eriksson. 2023. A Graded Modal Dependent Type Theory with a Universe and Erasure, Formalized. *Proc. ACM Program. Lang.* 7, ICFP, Article 220 (Aug. 2023), 35 pages. doi:10.1145/

3607862
[3] Erik Barendsen and Sjaak Smetsers. 2007. Strictness Analysis via Resource Typing. https://www.cs.ru.nl/barendregt60/essays/barendsen_smetsers/art02_barendsen_smetsers.pdf
[4] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.* 2, POPL, Article 5 (Dec. 2017), 29 pages. doi:10.1145/3158093
[5] Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. 2023. Resource-Aware Soundness for Big-Step Semantics. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 267 (Oct. 2023), 29 pages. doi:10.1145/3622843
[6] Bror Bjerner and S. Holmström. 1989. A composition approach to time analysis of first order lazy functional programs. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture* (Imperial College, London, United Kingdom) *(FPCA '89)*. Association for Computing Machinery, New York, NY, USA, 157–165. doi:10.1145/99370.99382
[7] Edwin Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming (ECOOP 2021)* (2021). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 9:1–9:26. doi:10.4230/LIPIcs.ECOOP.2021.9
[8] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag, Berlin, Heidelberg, 351–370. doi:10.1007/978-3-642-54833-8_19
[9] José Manuel Calderón Trilla. 2015. *Improving Implicit Parallelism*. Ph. D. Dissertation. University of York. http://jmct.cc/thesis.pdf
[10] Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. 2021. A graded dependent type system with a usage-aware semantics. 5 (2021), 1–32. Issue POPL. doi:10.1145/3434331
[11] Alonzo Church. 1932. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics* 33, 2 (1932), 346–366. doi:10.2307/1968337
[12] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. doi:10.1145/512950.512973
[13] Ugo Dal Lago and Francesco Gavazzo. 2022. A relational theory of effects and coeffects. *Proc. ACM Program. Lang.* 6, POPL, Article 31 (Jan. 2022), 28 pages. doi:10.1145/3498692
[14] Edkso de Vries. 2017. Visualizing lazy evaluation. https://www.well-typed.com/blog/2017/09/visualize-cbn/
[15] Edkso de Vries. 2020. Being lazy without getting bloated. https://well-typed.com/blog/2020/09/nothunks/
[16] Dorothy E. Denning and Peter J. Denning. 1977. Certification of programs for secure information flow. *Commun. ACM* 20, 7 (July 1977), 504–513. doi:10.1145/359636.359712
[17] Paul Downen, Zena M. Ariola, Simon Peyton Jones, and Richard A. Eisenberg. 2020. Kinds are calling conventions. *Proc. ACM Program. Lang.* 4, ICFP, Article 104 (Aug. 2020), 29 pages. doi:10.1145/3408986
[18] Robert Ennals and Simon Peyton Jones. 2003. Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. *SIGPLAN Not.* 38, 9 (Aug. 2003), 287–298. doi:10.1145/944746.944731
[19] Olivier Flückiger, Guido Chari, Jan Ječmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. 2019. R melts brains: an IR for first-class environments and lazy effectful arguments. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages* (Athens, Greece) *(DLS 2019)*. Association for Computing Machinery, New York, NY, USA, 55–66. doi:10.1145/3359619.3359744
[20] Yannick Forster, Steven Schäfer, Simon Spies, and Kathrin Stark. 2019. Call-by-push-value in Coq: operational, equational, and denotational theory. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New York, NY, USA, 2019-01-14) *(CPP 2019)*. Association for Computing Machinery, 118–131. doi:10.1145/3293880.3294097
[21] The R Foundation. 1999. https://www.r-project.org/
[22] Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining effects and coeffects via grading. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara Japan, 2016-09-04). ACM, 476–489. doi:10.1145/2951913.2951939
[23] Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag, Berlin, Heidelberg, 331–350. doi:10.1007/978-3-642-54833-8_18
[24] Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101. doi:10.1016/0304-3975(87)90045-4
[25] Aviral Goel and Jan Vitek. 2019. On the design, implementation, and use of laziness in R. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 153 (Oct. 2019), 27 pages. doi:10.1145/3360579
[26] Haskell.org. 1996. The Glasgow Haskell Compiler. https://www.haskell.org/ghc/

[27] Haskell.org. 1996. Haskell Language. https://www.haskell.org/

[28] HaskellWiki. 2019. Foldr Foldl Foldl'. https://wiki.haskell.org/Foldr_Foldl_Foldl'

[29] HaskellWiki. 2021. Weak head normal form. https://wiki.haskell.org/Weak_head_normal_form

[30] HaskellWiki. 2022. Performance/Strictness. https://wiki.haskell.org/Performance/Strictness

[31] HaskellWiki. 2025. seq. https://wiki.haskell.org/index.php?title=Seq

[32] Andrew K. Hirsch and Ross Tate. 2018. Strict and lazy semantics for effects: layering monads and comonads. *Proc. ACM Program. Lang.* 2, ICFP, Article 88 (July 2018), 30 pages. doi:10.1145/3236783

[33] John Hughes. 1989. Projections for polymorphic strictness analysis. In *Category Theory and Computer Science*, David H. Pitt, David E. Rydeheard, Peter Dybjer, Andrew M. Pitts, and Axel Poigné (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 82–100.

[34] Patricia Johann and Janis Voigtländer. 2004. Free theorems in the presence of seq. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 99–110. doi:10.1145/964001.964010

[35] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 145–158. doi:10.1145/2500365.2500590

[36] Ohad Kammar and Gordon D. Plotkin. 2012. Algebraic foundations for effect-dependent optimisations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 349–360. doi:10.1145/2103656.2103698

[37] Shin-ya Katsumata. 2014. Parametric effect monads and semantics of effect systems. *SIGPLAN Not.* 49, 1 (Jan. 2014), 633–645. doi:10.1145/2578855.2535846

[38] G. A. Kavvos, Edward Morehouse, Daniel R. Licata, and Norman Danner. 2019. Recurrence extraction for functional programs through call-by-push-value. *Proc. ACM Program. Lang.* 4, POPL, Article 15 (Dec. 2019), 31 pages. doi:10.1145/3371083

[39] Ariel E. Kellison, Laura Zielinski, David Bindel, and Justin Hsu. 2025. Bean: A Language for Backward Error Analysis. *Proc. ACM Program. Lang.* 9, PLDI, Article 221 (June 2025), 25 pages. doi:10.1145/3729324

[40] Steve Klabnik, Carol Nichols, and Chris Krycho. 2024. Processing a Series of Items with Iterators. https://doc.rust-lang.org/book/ch13-02-iterators.html#processing-a-series-of-items-with-iterators

[41] Ryszard Kubiak, John Hughes, and John Launchbury. 1992. Implementing Projection-based Strictness Analysis. In *Functional Programming, Glasgow 1991*, Rogardt Heldal, Carsten Kehler Holst, and Philip Wadler (Eds.). Springer, London, 207–224. doi:10.1007/978-1-4471-3196-0_17

[42] T.-M. Kuo and P. Mishra. 1987. On strictness and its analysis. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Munich, West Germany) *(POPL '87)*. Association for Computing Machinery, New York, NY, USA, 144–155. doi:10.1145/41625.41638

[43] Tsung-Min Kuo and Prateek Mishra. 1989. Strictness analysis: a new perspective based on type inference. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture* (Imperial College, London, United Kingdom) *(FPCA '89)*. Association for Computing Machinery, New York, NY, USA, 260–272. doi:10.1145/99370.99390

[44] John Launchbury. 1993. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '93*. ACM Press, Charleston, South Carolina, United States, 144–154. doi:10.1145/158511.158618

[45] John Launchbury and Gebreselassie Baraki. 1996. Representing Demand by Partial Projections. *Journal of Functional Programming* 6 (12 1996). doi:10.1017/S0956796800001878

[46] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2025. https://ocaml.org/manual/5.3/api/Lazy.html

[47] Paul Blain Levy. 1999. Call-by-Push-Value: A Subsuming Paradigm. In *Typed Lambda Calculi and Applications* (Berlin, Heidelberg, 1999), Jean-Yves Girard (Ed.). Springer, 228–243. doi:10.1007/3-540-48959-2-17

[48] Paul Blain Levy. 2001. Call-By-Push-Value. https://qmro.qmul.ac.uk/xmlui/handle/123456789/4742

[49] Paul Blain Levy. 2003. *Call-By-Push-Value.* Springer Netherlands, Dordrecht. doi:10.1007/978-94-007-0954-6

[50] Paul Blain Levy. 2006. Call-by-push-value: Decomposing call-by-value and call-by-name. 19, 4 (2006), 377–414. doi:10.1007/s10990-006-0480-6

[51] Paul Blain Levy. 2022. Call-by-push-value. *ACM SIGLOG News* 9, 2 (May 2022), 7–29. doi:10.1145/3537668.3537670

[52] J. M. Lucassen and D. K. Gifford. 1988. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '88* (San Diego, California, United States, 1988). ACM Press, 47–57. doi:10.1145/73560.73564

[53] John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. 1995. Call-by-name, Call-by-value, Call-by-need, and the Linear Lambda Calculus. *Electronic Notes in Theoretical Computer Science* 1 (1995), 370–392. doi:10.1016/S1571-

0661(04)00022-2 MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference.
[54] Dylan McDermott. 2025. Grading call-by-push-value, explicitly and implicitly. doi:10.4230/LIPIcs.FSCD.2025.25
[55] Dylan McDermott and Alan Mycroft. 2018. Call-by-need effects via coeffects. *Open Computer Science* 8, 1 (2018), 93–108. doi:doi:10.1515/comp-2018-0009
[56] Dylan McDermott and Alan Mycroft. 2019. Extended Call-by-Push-Value: Reasoning About Effectful Programs and Evaluation Order.. In *ESOP*. 235–262.
[57] Stephen Mell, Konstantinos Kallas, Steve Zdancewic, and Osbert Bastani. 2025. Opportunistically Parallel Lambda Calculus. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 365 (Oct. 2025), 27 pages. doi:10.1145/3763143
[58] Henry Mercer, Cameron Ramsay, and Neel Krishnaswami. 2022. Implicit Polarized F: local type inference for impredicativity. (03 2022). doi:10.48550/arXiv.2203.01835
[59] Neil Mitchell. 2013. Destroying Performance with Strictness. https://neilmitchell.blogspot.com/2013/08/destroying-performance-with-strictness.html
[60] Alan Mycroft. 1980. The Theory and Practice of Transforming Call-by-need into Call-by-value. In *Proceedings of the Fourth 'Colloque International sur la Programmation' on International Symposium on Programming*. Springer-Verlag, Berlin, Heidelberg, 269–281.
[61] Aleksandar Nanevski. 2003. From dynamic binding to state via modal possibility. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming* (Uppsala Sweden, 2003-08-27). ACM, 207–218. doi:10.1145/888251.888271
[62] Max S. New. 2023. Compiling with Call-by-push-value. (2023). https://maxsnew.com/docs/mfps2023-slides.pdf
[63] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.* 3, ICFP, Article 110 (July 2019), 30 pages. doi:10.1145/3341714
[64] Jens Palsberg and Patrick O'Keefe. 1995. A type system equivalent to flow analysis. *ACM Trans. Program. Lang. Syst.* 17, 4 (July 1995), 576–599. doi:10.1145/210184.210187
[65] Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2013. Coeffects: Unified Static Analysis of Context-Dependence. In *Automata, Languages, and Programming* (Berlin, Heidelberg, 2013), Fedor V. Fomin, Rūsiņš Freivalds, Marta Kwiatkowska, and David Peleg (Eds.). Springer, 385–397. doi:10.1007/978-3-642-39212-2-35
[66] Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: a calculus of context-dependent computation. *SIGPLAN Not.* 49, 9 (Aug. 2014), 123–135. doi:10.1145/2692915.2628160
[67] Pierre-Marie Pédrot and Nicolas Tabareau. 2019. The fire triangle: how to mix substitution, dependent elimination, and effects. 4 (2019), 1–28. Issue POPL. doi:10.1145/3371126
[68] Jason Reed and Benjamin C. Pierce. 2010. Distance makes the types grow stronger: a calculus for differential privacy. *SIGPLAN Not.* 45, 9 (Sept. 2010), 157–168. doi:10.1145/1932681.1863568
[69] Nick Rioux and Steve Zdancewic. 2020. Computation focusing. 4 (2020), 95:1–95:27. Issue ICFP. doi:10.1145/3408977
[70] Lukas Rytz, Martin Odersky, and Philipp Haller. 2012. Lightweight Polymorphic Effects. In *ECOOP 2012 – Object-Oriented Programming* (Berlin, Heidelberg, 2012), James Noble (Ed.). Springer, 258–282. doi:10.1007/978-3-642-31057-7-13
[71] Daniel Sainati, Joseph W. Cutler, Benjamin C. Pierce, and Stephanie Weirich. 2025. Artifact associated with "Typing Strictness". doi:10.5281/zenodo.17279039
[72] Daniel Sainati, Joseph W. Cutler, Benjamin C. Pierce, and Stephanie Weirich. 2025. Typing Strictness (Extended Version). doi:10.48550/arXiv.2510.16133
[73] Typelevel Scala. 2013. FS2: Functional, effectful, concurrent streams for Scala. https://fs2.io/#/
[74] Tom Schrijvers and Alan Mycroft. 2010. Strictness Meets Data Flow. In *Static Analysis*, Radhia Cousot and Matthieu Martel (Eds.). Springer, Berlin, Heidelberg, 439–454. doi:10.1007/978-3-642-15769-1_27
[75] Ilya Sergey, Simon Peyton Jones, and Dimitrios Vytiniotis. 2014. Theory and practice of demand analysis in Haskell. (2014). https://www.microsoft.com/en-us/research/publication/theory-practice-demand-analysis-haskell/
[76] Ilya Sergey, Dimitrios Vytiniotis, and Simon Peyton Jones. 2014. Modular, higher-order cardinality analysis in theory and practice. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. Association for Computing Machinery, New York, NY, USA, 335–347. doi:10.1145/2535838.2535861
[77] Sjaak Smetsers and Marko van Eekelen. 2013. Higher-Order Strictness Typing. In *Trends in Functional Programming*, Hans-Wolfgang Loidl and Ricardo Peña (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 85–100.
[78] J.-P. Talpin and P. Jouvelot. 1992. The type and effect discipline. In *[1992] Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science* (1992-06). 162–173. doi:10.1109/LICS.1992.185530
[79] Rocq Team. 2025. The Rocq Prover. https://rocq-prover.org/
[80] Cassia Torczon, Emmanuel Suárez Acevedo, Shubh Agrawal, Joey Velez-Ginorio, and Stephanie Weirich. 2024. Effects and Coeffects in Call-by-Push-Value. 8 (2024), 1108–1134. Issue OOPSLA2. doi:10.1145/3689750
[81] David N. Turner, Philip Wadler, and Christian Mossin. 1995. Once upon a type. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture* (La Jolla, California, USA) *(FPCA '95)*.

Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/224164.224168

[82] Marco Vassena, Joachim Breitner, and Alejandro Russo. 2017. Securing Concurrent Lazy Programs Against Information Leakage. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. 37–52. doi:10.1109/CSF.2017.39

[83] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A sound type system for secure flow analysis. *J. Comput. Secur.* 4, 2–3 (Jan. 1996), 167–187.

[84] Philip Wadler. 1987. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In *Abstract Interpretation of Declarative Languages*. Halsted Press.

[85] Philip Wadler. 1993. Linear Types Can Change the World! (10 1993).

[86] Philip Wadler and R. J. M. Hughes. 1987. Projections for strictness analysis. In *Proceedings of the Functional Programming Languages and Computer Architecture*. Springer-Verlag, Berlin, Heidelberg, 385–407.

[87] Philip Wadler and Peter Thiemann. 2003. The marriage of effects and monads. *ACM Trans. Comput. Logic* 4, 1 (Jan. 2003), 1–32. doi:10.1145/601775.601776

[88] C.P. Wadsworth. 1971. *Semantics and Pragmatics of the Lambda-calculus*. University of Oxford. https://books.google.com/books?id=kl1QIQAACAAJ

[89] Keith Wansbrough. 2005. *Simple polymorphic usage analysis*. Technical Report UCAM-CL-TR-623. University of Cambridge, Computer Laboratory. doi:10.48456/tr-623

[90] Keith Wansbrough and Simon Peyton Jones. 1999. Once upon a polymorphic type. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 15–28. doi:10.1145/292540.292545

[91] Li-yao Xia, Laura Israel, Maite Kramarz, Nicholas Coltharp, Koen Claessen, Stephanie Weirich, and Yao Li. 2024. Story of Your Lazy Function's Life: A Bidirectional Demand Semantics for Mechanized Cost Analysis of Lazy Programs. *Proceedings of the ACM on Programming Languages* 8, ICFP (Aug. 2024), 30–63. doi:10.1145/3674626

[92] Steve Zdancewic and Andrew C. Myers. 2001. Secure Information Flow and CPS. In *Programming Languages and Systems*, David Sands (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 46–61.

[93] Steve Zdancewic and Andrew C. Myers. 2002. Secure Information Flow via Linear Continuations. *Higher-Order and Symbolic Computation* 15, 2 (Sept. 2002), 209–234. doi:10.1023/A:1020843229247