Computation Focusing[*]

NICK RIOUX, University of Pennsylvania, USA
STEVE ZDANCEWIC, University of Pennsylvania, USA

Focusing is a technique from proof theory that exploits type information to prune inessential nondeterminism from proof search procedures. Viewed through the lens of the Curry-Howard correspondence, a focused typing derivation yields terms in normal form. This paper explores how to exploit focusing for reasoning about contextual equivalences and full abstraction. We present a focused polymorphic call-by-push-value calculus and prove a computational completeness result: for every well-typed term, there exists a focused term that is $\beta\eta$-equivalent to it. This completeness result yields a powerful way to refine the context lemmas for establishing contextual equivalences, cutting down the set that must be considered to just focused contexts. The paper demonstrates the application of focusing to establish program equivalences, including free theorems. It also uses focusing to prove full abstraction of a translation of the pure, total call-by-push-value language into a language with divergence and simple effect types, yielding a novel solution to a simple-to-state, but hitherto difficult to solve problem.

CCS Concepts: • **Theory of computation** → **Type theory**; *Proof theory*; **Logic and verification**.

Additional Key Words and Phrases: compiler verification, focusing, full abstraction, program equivalence, type systems

ACM Reference Format:
Nick Rioux and Steve Zdancewic. 2020. Computation Focusing. *Proc. ACM Program. Lang.* 4, ICFP, Article 95 (August 2020), 27 pages. https://doi.org/10.1145/3408977

1 INTRODUCTION

We often want to understand how a program component interacts with its environment. Compilers, for instance, ought to ensure that any use of a compiled component corresponds to some valid use of the corresponding source. If this were not the case, a target environment might be able to exploit the compiled component by using it in a way the programmer never expected to be possible, making it impossible for the programmer to reason about their program's behavior by using their knowledge of the source semantics. One of the strongest "compiler correctness" properties is *full abstraction*, which means that the compiler preserves and reflects contextual equivalences between the source and target languages. A fully abstract compiler ensures that all abstractions in a source language program are enforced in the compiled program.

Proving full abstraction results is notoriously challenging because it relies crucially on transporting contextual equivalences from one language to another. Two program components are *contextually equivalent* [Morris 1968] if there is no environment that can tell them apart: they must behave the same in any valid context. Unfortunately, the vast number of such contexts makes direct

[*]This paper's companion technical report [Rioux and Zdancewic 2020] contains a detailed appendix with the full proofs and definitions referenced here.

Authors' addresses: Nick Rioux, University of Pennsylvania, Philadelphia, Pennsylvania, USA, nrioux@cis.upenn.edu; Steve Zdancewic, University of Pennsylvania, Philadelphia, Pennsylvania, USA, stevez@cis.upenn.edu.

proofs of contextual equivalence difficult. We cannot practically enumerate each possible context and verify that two components behave identically in each case. For full abstraction, we have the additional challenge of relating source and target contexts.

As a consequence of the above challenges, much work in the area has lead to a wide variety of sound proof techniques that ease this burden. *Complete* methods, such as applicative bisimulation [Abramsky 1990; Abramsky and Ong 1993], environmental bisimulation [Sangiorgi et al. 2007], or "closed-instances of uses" (ciu) [Mason and Talcott 1991] and similar context lemmas[Milner 1977], cut down on the number of cases that must be considered, while still retaining the full power of contextual equivalence. *Incomplete* methods, such as normal form bisimulations [Lassen 1999, 2005] retain soundness, and often dispense with the quantification over contexts, but are not able to prove all the equivalences one might want.

For typed programming languages, there are more options. Type systems are a tried-and-tested method of enforcing a programmer's abstractions, and, as such, the type structure of a programming language can be exploited to make proving equivalences easier. For example, *logical relations* [Plotkin 1980; Statman 1985; Tait 1967] associate each type with a relation over closed terms that inhabit the type. The relations are defined inductively over the structure of the type, and soundness of the logical relation implies that two related terms are contextually equivalent. Again there is a tradeoff between *complete* logical relations methods, which can be quite involved to define, and more elegant, concise, but *incomplete* approaches, such as *operational extensionality* [Pitts 2004]. Intuitively, while logical relations can easily say a lot about what computations inhabit a type, it requires significantly more machinery (for instance TT-closure or *biorthogonality*) to characterize the contexts in which those computations run.

In this paper, we expore how *focusing*, a well-established, type-directed technique for cutting down the search space in proof search [Andreoli 1992; Chaudhuri et al. 2018; Simmons 2014], can be applied to the problem of proving contextual equivalences, and, by extension, full abstraction results. In focusing, the idea is to remove innessential nondeterministic choices from the search algorithm (by finding a β-normal, η-long proof term). For proof search, completeness of focusing says that "for each well-formed proof p of a proposition, there exists a focused proof p_f of the same proposition." For programs, we want to view focusing through the lens of the Curry-Howard correspondence, in which case the completeness maxim is "for each well-typed term M, there exists a focused term M_f that is $\beta\eta$-equivalent to M".

We can then see a proof of program equivalence as a search through all possible *focused* contexts, in which we demonstrate identical behavior for each context. Focusing lets us exploit type information to refine the usual ciu theorem to considerably cut down the space of contexts that we must consider. Informally, the usual ciu lemma defines a "use" to be an arbitrary evaluation context, but many evaluation contexts do unnecessary work—perhaps running a long computation that eventually evaluates to a constant—before getting around to probing the program under observation. Focusing complements this idea by exploiting type information to restrict the set of evaluation contexts to just those that immediately do the probing. Moreover, rather than considering all closing substitutions, as would be used in ciu and other techniques, we can instead close terms by substituting focused values, which again cuts down the search space.

The structure of a focused derivation yields strong inversion and induction principles that can be used to establish full abstraction results. We explore this possibility by proving full abstraction of a simple (almost trivial) compiler from a pure, total language to one that admits divergence but whose type system tracks that possibility. Despite the straightforward-sounding, and intuitively correct nature of the translation, which ensures that source programs are translated to the pure, terminating fragment of the target language, proving its correctness by traditional means is quite challenging (indeed, we are unaware of similar results in the literature). The crux of the matter is how to "back-translate" arbitrary target contexts, but here we can leverage focusing so that we need only back-translate focused contexts, which is easier to do.

Focusing computations is useful in other situations too. If we have a well-typed term and want to explore its possible behaviors, one approach is to use the classic "free theorems" [Wadler 1989], which have traditionally been proven using logical relations. Focusing provides an alternative. For instance, since the only closed, focused term of type $\forall\alpha.\alpha \to \alpha$ is the polymorphic identity function $\Lambda\alpha.\lambda x\!:\!\alpha.x$, completeness of focusing tells us that *any* term M of polymorphic identity type $\forall\alpha.\alpha \to \alpha$ is equivalent to the polymorphic identity function. We can extrapolate from this idea to prove equivalences of polymorphic types, including existential packages, by examining how focused contexts are restricted in interacting with the such types.

To summarize, in this paper we make these main contributions:

- We define a focused variant of Levy's call-by-push-value language with predicative polymorphism and prove its completeness with respect to the unfocused version.
- We demonstrate that focusing is useful for establishing program equivalences—it *refines* the usual ciu theorem to further cut down the set of contexts that must be considered—by which we can obtain results such as "free theorems." We also give an example, based on one found in the literature [Pitts 2004; Sumii and Pierce 2005], for which traditional logical relations and bisimulation arguments are insufficient, but in which focusing can be used effectively.
- We further demonstrate the utility of focusing via a minimal but demonstrative application in which we prove full abstraction for an embedding of a pure, total language into a language that admits divergence. To our knowledge, this is the first of its kind. Moreover, our proof uses distills full abstraction into compositionality and adequacy of a "back translation," which may be of independent interest.

We build our focusing machinery on top of Levy's call-by-push-value (CBPV) calculus [Levy 1999], a choice that suits our needs because focusing relies on a clear distinction between computations and values, which is made explicit by the CBPV type system. We use a predicative polymorphic variant of the language in the style of stratified system F [Eades III and Stump 2010], which admits good induction principles that simplify some of our proofs (we leave exploration of impredicative polymorphism to future work).

Section 2 reviews our variant of the CBPV language and establishes its basic metatheory. Section 3 gives the first main technical contribution of the paper, namely the definition of the focusing rules for CBPV and a proof of completeness. We next turn to several applications of focusing as developed here in Section 4, before moving on to our full abstraction result about a very simple compiler in Section 5. Finally, we wrap up the paper with a discussion of related work in Section 6.1.

2 CALL-BY-PUSH-VALUE

Figure 1 describes the syntax of a (predicative) polymorphic variant of Levy's simply-typed call-by-push-value (CBPV) language that we call CBPV$^\forall$. CBPV$^\forall$ makes evaluation order explicit by distinguishing values from computations both in the term syntax and in the type system. Values, V represent pure pieces of data, they include thunked computations, tagged (disjoint unions), products, and existential packages. Importantly, all term variables in CBPV$^\forall$ represent values. A computation, M perform some interesting, possibly effectful, operation. These include forcing the evaluation of a thunk, pattern matching against a value using the **pm** form, sequencing subcomputations via **bind**, lambda-abstracting a variable (which "pops" an argument from the stack), pushing a value onto the stack ($M\ V$), or constructing values. A lazy product $\langle M, N\rangle$ represents a choice of computations, one of which can be selected by the projection operations $M.1$ and $M.2$. Thus, the motto of call-by-push-value: "a value *is*, a computation *does*" [Levy 1999].

$$
\begin{array}{lll}
V & ::= & x \mid \textbf{thunk}\, N \mid (1, V) \mid (2, V) \\
& \mid & \textbf{unit} \mid (V_1, V_2) \mid (A, V) \mid (\underline{A}, V) \\
\\
M, N & ::= & \textbf{force}\, V \\
& \mid & \textbf{pm}\, V\, \textbf{as}\, \{(1, x_1).M_1, (2, x_2).M_2\} \\
& \mid & \textbf{pm}\, V\, \textbf{as}\, \textbf{unit}.N \\
& \mid & \textbf{pm}\, V\, \textbf{as}\, (x_1, x_2).N \\
& \mid & \textbf{pm}\, V\, \textbf{as}\, (X, x).N \\
& \mid & \textbf{pm}\, V\, \textbf{as}\, (\underline{X}, x).N \\
& \mid & \textbf{return}\, V \mid \textbf{bind}\, x \leftarrow M.N \\
& \mid & \lambda x : A.N \\
& \mid & \mid \lambda X : \kappa.N \lambda \underline{X} : \kappa.N \\
& \mid & M\ V \mid \langle M, N \rangle \mid N.1 \mid N.2 \\
& \mid & M\ A \mid M\ \underline{A} \\
\\
S & ::= & [\cdot] \mid \textbf{bind}\, x \leftarrow S.N \mid S\ V \\
& \mid & S\ A \mid S\ \underline{A} \mid S.1 \mid S.2
\end{array}
\qquad
\begin{array}{lll}
\kappa & ::= & \textbf{Type}^u \\
A, B & ::= & X \mid \textbf{U}\, \underline{A} \\
& \mid & \exists X : \kappa.A \mid \exists \underline{X} : \kappa.A \\
& \mid & A + B \mid \textbf{Unit} \mid A \times B \\
\\
\underline{A}, \underline{B} & ::= & \underline{X} \mid \textbf{F}\, A \\
& \mid & A \to \underline{B} \mid \forall X : \kappa.\underline{A} \\
& \mid & \forall \underline{X} : \kappa.\underline{A} \mid \underline{A}\ \&\ \underline{B} \\
\\
T & ::= & \underline{A} \mid \langle \underline{A} \rangle \\
\Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, x : \langle A \rangle \\
\Delta & ::= & \cdot \mid \Delta, X : \kappa \mid \Delta, \underline{X} : \kappa \\
\Omega & ::= & \cdot \mid x : A, \Omega
\end{array}
$$

Fig. 1. CBPV Terms: V values, M computations, and S stacks (left); Types and Contexts (right)

$\boxed{N \mapsto N'}$ *(small-step reduction)*

$$
\frac{N \mapsto N'}{S[N] \mapsto S[N']}
\qquad
\frac{}{\textbf{force thunk}\, N \mapsto N}
\qquad
\frac{}{\textbf{pm}\, (i, V)\, \textbf{as}\, \{(1, x_1).M_1, (2, x_2).M_2\} \mapsto M_i[V/x]}
$$

$$
\frac{}{\textbf{pm}\, (A, V)\, \textbf{as}\, (X, x).N \mapsto N[A/X][V/x]}
\qquad
\frac{}{\textbf{bind}\, x \leftarrow \textbf{return}\, V.N \mapsto N[V/x]}
\qquad
\frac{}{(\lambda x : A.N)\ V \mapsto N[V/x]}
$$

$$
\frac{}{(\lambda X : \kappa.N)\ A \mapsto N[A/X]}
$$

Fig. 2. Selected operational semantics

A stack S acts as an evaluation context, and the CBPV$^\forall$ operational semantics, shown in Figure 2, make uses of the stack explicit (the first rule). The remainder of the operational semantics rules correspond to the usual β reductions, where we use the notation $M[V/x]$ to mean the capture-avoiding substitution of the value V for the free occurrences of the variable x in M. We also use $A[\underline{B}/\underline{X}]$ and $A[\underline{B}/\underline{X}]$ to refer to the substitution of types for type variables in value types and similar notation for subtitution of type variables in computation types.

The right-hand side of Figure 1 shows the syntax of CBPV$^\forall$ types, which come in two varieties: value types, A, and computation types, $\underline{A}$. The lone kind $\textbf{Type}^u$ is parameterized by a natural number u that stratifies types into (ordered) universes to ensure predicativity [Leivant 1991]. A term environment, Γ, is an unordered mapping from term variables to value types. Type environments Δ similarly map type variables to kinds.

Variables in a Γ environment may also map to a suspended type $\langle A \rangle$. Suspensions are used only when describing focused programs and are not used in the conventional typing rules. The name suspension refers to the suspension of the focusing procedure. We delay the discussion of suspended types, along with ordered term environments Ω, to the description of focusing in Section 3.

Figure 3 details CBPV$^\forall$'s type system. The unit type, sums, and pairs are value types. We also note that they have positive polarity, that is, their elimination forms are pattern matching constructs. The thunk type $\textbf{U}\, \underline{A}$ is the type of a computation waiting to be evaluated. It can be executed with

$\boxed{\Delta \vdash_\kappa \underline{A} : \mathbf{Type}^u}$ *(computation type kinding)*

$$\frac{\Delta, X : \mathbf{Type}^v \vdash_\kappa \underline{A} : \mathbf{Type}^u}{\Delta \vdash_\kappa \forall X : \mathbf{Type}^v.\underline{A} : \mathbf{Type}^{\max(u,v)+1}}$$

$\boxed{\Delta \vdash_\kappa A : \mathbf{Type}^u}$ *(value type kinding)*

$$\frac{\Delta, X : \mathbf{Type}^v \vdash_\kappa A : \mathbf{Type}^u}{\Delta \vdash_\kappa \exists X : \mathbf{Type}^v.A : \mathbf{Type}^{\max(u,1)+1}}$$

$\boxed{\Delta; \Gamma \vdash_c M : \underline{B}}$ *(computation typing)*

$$\frac{\Delta; \Gamma \vdash_v V : \mathbf{U}\underline{A}}{\Delta; \Gamma \vdash_c \mathbf{force}\ V : \underline{A}} \qquad \frac{\Delta; \Gamma \vdash_v V : A \times B \qquad \Delta; \Gamma, x : A, y : B \vdash_c N : \underline{B}}{\Delta; \Gamma \vdash_c \mathbf{pm}\ V\ \mathbf{as}\ (x, y).N : \underline{B}}$$

$$\frac{\Delta; \Gamma \vdash_v V : \exists X : \kappa.B \qquad \Delta, X : \kappa; \Gamma, x : B \vdash_c N : \underline{B} \qquad X \notin \mathbf{FTV}(\underline{B})}{\Delta; \Gamma \vdash_c \mathbf{pm}\ V\ \mathbf{as}\ (X, x).N : \underline{B}} \qquad \frac{\Delta; \Gamma \vdash_v V : A}{\Delta; \Gamma \vdash_c \mathbf{return}\ V : \mathbf{F}A}$$

$$\frac{\Delta; \Gamma \vdash_c M : \mathbf{F}A \qquad \Delta; \Gamma, x : A \vdash_c N : \underline{B}}{\Delta; \Gamma \vdash_c \mathbf{bind}\ x \leftarrow M.N : \underline{B}} \qquad \frac{\Delta; \Gamma, x : A \vdash_c N : \underline{B}}{\Delta; \Gamma \vdash_c \lambda x : A.N : A \to \underline{B}} \qquad \frac{\Delta, X : \kappa; \Gamma \vdash_c N : \underline{B}}{\Delta; \Gamma \vdash_c \lambda X : \kappa.N : \forall X : \kappa.\underline{B}}$$

$$\frac{\Delta; \Gamma \vdash_c M : A \to \underline{B} \qquad \Delta; \Gamma \vdash_v V : A}{\Delta; \Gamma \vdash_c M\ V : \underline{B}} \qquad \frac{\Delta \vdash_\kappa A : \kappa \qquad \Delta; \Gamma \vdash_c M : \forall X : \kappa.\underline{B}}{\Delta; \Gamma \vdash_c M\ A : \underline{B}[A/X]}$$

$\boxed{\Delta; \Gamma \vdash_v V : B}$ *(value typing)*

$$\frac{\Delta; \Gamma \vdash_c N : \underline{A}}{\Delta; \Gamma \vdash_v \mathbf{thunk}\ N : \mathbf{U}\underline{A}} \qquad \frac{\Delta; \Gamma \vdash_v V_1 : A_1 \qquad \Delta; \Gamma \vdash_v V_2 : A_2}{\Delta; \Gamma \vdash_v (V_1, V_2) : A_1 \times A_2} \qquad \frac{\Delta \vdash_\kappa A : \kappa \qquad \Delta; \Gamma \vdash_v V : B[A/X]}{\Delta; \Gamma \vdash_v (A, V) : \exists X : \kappa.B}$$

Fig. 3. Selected kinding and typing rules

$\boxed{\Delta; \Gamma \vdash_c N \equiv N' : \underline{B}}$ *(computation equality)*

$$\frac{}{\Delta; \Gamma \vdash_c \mathbf{force}\ (\mathbf{thunk}\ M) \equiv M : \underline{B}} \qquad \frac{}{\Delta; \Gamma \vdash_c \mathbf{pm}\ (V_1, V_2)\ \mathbf{as}\ (x_1, x_2).M \equiv M[V_1/x_1][V_2/x_2] : \underline{B}}$$

$$\frac{}{\Delta; \Gamma \vdash_c \mathbf{pm}\ (A, V)\ \mathbf{as}\ (X, x).M \equiv M[A/X][V/x] : \underline{B}} \qquad \frac{}{\Delta; \Gamma \vdash_c \mathbf{bind}\ x \leftarrow \mathbf{return}\ V.M \equiv M[V/x] : \underline{B}}$$

$$\frac{}{\Delta; \Gamma \vdash_c (\lambda x : A.M)\ V \equiv M[V/x] : \underline{B}} \qquad \frac{}{\Delta; \Gamma \vdash_c (\lambda X : \kappa.M)\ A \equiv M[A/X] : \underline{B}}$$

$$\frac{}{\Delta; \Gamma, x : A_1 \times A_2 \vdash_c M \equiv \mathbf{pm}\ x\ \mathbf{as}\ (y_1, y_2).M[(y_1, y_2)/x] : \underline{B}} \qquad \frac{}{\Delta; \Gamma, x : \exists X : \kappa.A \vdash_c M \equiv \mathbf{pm}\ x\ \mathbf{as}\ (X, y).M[(X, y)/x] : \underline{B}}$$

$$\frac{}{\Delta; \Gamma \vdash_c M \equiv \mathbf{bind}\ x \leftarrow M.\mathbf{return}\ x : \underline{B}} \qquad \frac{}{\Delta; \Gamma \vdash_c M \equiv \lambda x : A.M\ x : A \to \underline{B}} \qquad \frac{}{\Delta; \Gamma \vdash_c M \equiv \lambda X : \kappa.M\ X : \forall X : \kappa.\underline{B}}$$

$$\frac{}{\Delta; \Gamma \vdash_c \mathbf{bind}\ y \leftarrow (\mathbf{bind}\ x \leftarrow M.N).N' \equiv \mathbf{bind}\ x \leftarrow M.\mathbf{bind}\ y \leftarrow N.N' : \underline{B}}$$

$$\frac{}{\Delta; \Gamma \vdash_c \mathbf{bind}\ x \leftarrow M.\lambda y : A.N \equiv \lambda y : A.\mathbf{bind}\ x \leftarrow M.N : A \to \underline{B}}$$

$$\frac{}{\Delta; \Gamma \vdash_c \mathbf{bind}\ x \leftarrow M.\lambda X : \kappa.N \equiv \lambda X : \kappa.\mathbf{bind}\ x \leftarrow M.N : \forall X : \kappa.\underline{B}}$$

$$\frac{}{\Delta; \Gamma \vdash_c \mathbf{bind}\ x \leftarrow M.\langle N_1, N_2\rangle \equiv \langle \mathbf{bind}\ x \leftarrow M.N_1, \mathbf{bind}\ x \leftarrow M.N_2\rangle : \underline{B}_1 \ \&\ \underline{B}_2}$$

$\boxed{\Delta; \Gamma \vdash_v V \equiv V' : B}$ *(value equality)*

$$\frac{}{\Delta; \Gamma \vdash_v V \equiv \mathbf{thunk}\ (\mathbf{force}\ V) : \mathbf{U}\underline{B}}$$

Fig. 4. Abbreviated equational theory

the **force** elimination form, making it the lone negative value type. Lazy products, and functions are negative computation types. The value returner type, $\mathbf{F}\,A$, is the type of computations that compute and return some value of type A. The value produced by a returner M can be accessed using the monadic bind construct $\mathbf{bind}\ x \leftarrow M.N$. This composes M and N, producing a larger computation. N may rely on x, the value returned by M. Returner types $\mathbf{F}\,A$ are the only positive computation types.

From these types, we can build up many other familiar types. We encode the type **Bool** as **Unit** + **Unit** where **true** and **false** refer to the injections of **unit**. **Maybe** A encodes to **Unit** + A. We refer to the constructors as **none** and **some** V. We occasionally find it convenient to use the fact that the types **Bool** and **Maybe Unit** are interchangeable, assuming **false** = **none** and **true** = **some unit**.

Equational Theory and Operational Properties. The separation of computations and values results in a rich equational theory. We write $\Delta; \Gamma \vdash_c M_1 \equiv M_2 : \underline{A}$ to mean M_1 and M_2 are equivalent according to the call-by-push-value equational theory. The $\equiv$ equivalence relation is a congruent closure of the β, η, associativity, and permutation laws given in figure 4. See Levy's work [1999] for further discussion.

As our goal is to establish equivalences as well as relationships between different languages, we take as given a few standard properties of the language. First, we assume that our equational theory recognizes syntactically different values as distinct.

Proposition 2.1 (Consistency of equational theory). *The computation* **return false** *is not equivalent to* **return true** *at type* **Bool** *under empty environments.*

Next, we characterize the behavior of whole programs in CBPV$^\forall$. Since CBPV$^\forall$is normalizing, every closed computation of type **F Bool** must return one of two values.

Proposition 2.2. *If* $\cdot; \cdot \vdash_c M : \mathbf{F\,Bool}$, *then either* $M \mapsto^* \mathbf{return\,false}$ *or* $M \mapsto^* \mathbf{return\,true}$.

As a consequence of these properties, as well as the fact that every step a program takes corresponds to a β rule, we have a correspondence between the operational semantics and the equational theory.

Lemma 2.3 (Adequacy of equational theory for operational semantics). *Given* $\cdot; \cdot \vdash_c M : \mathbf{F\,Bool}$ *and* $V \in \{\mathbf{false}, \mathbf{true}\}$ *where* $\cdot; \cdot \vdash_c M \equiv \mathbf{return}\ V : \mathbf{F\,Bool}$ *we have* $M \mapsto^* \mathbf{return}\ V$.

Contextual Equivalence. Contextual equivalence is the high-level, intuitive notion of equivalence that we generally wish to prove. It states that two terms are equivelent when placing them into closed contexts yields whole-programs that evaluate to the same value. In our definition below, we take *contexts* to be pairs of terms with a single free variable (the hole) and closing substitutions. We write $\Delta; \Gamma'' \vdash \gamma : \Gamma$ to refer to a mapping γ from variables in the domain of Γ to values whose types are given by Γ and environment by Δ and Γ'. We similarly write $\Delta' \vdash \delta : \Delta$ for a well-kinded substitution of types for type variables.

Definition 2.4 (Contextually equivalent values). V_1 and V_2 are said to be contextually equivalent, written $\Delta; \Gamma \vdash_v V_1 \approx_{ctx} V_2 : A$, when for all substitutions γ and δ and programs M such that

- $\cdot \vdash \delta : \Delta$
- $\cdot; \cdot \vdash \gamma : \delta(\Gamma)$
- $\cdot; x : \delta(A) \vdash_c M : \mathbf{F\,Bool}$

we have $M[\gamma(\delta(V_1))/x] \mapsto^* \mathbf{return\,true}$ iff $M[\gamma(\delta(V_2))/x] \mapsto^* \mathbf{return\,true}$.

Definition 2.5 (Contextually equivalent computations). M_1 and M_2 are said to be contextually equivalent, written $\Delta; \Gamma \vdash_c M_1 \approx_{ctx} M_2 : \underline{A}$, when **thunk** M_1 and **thunk** M_2 are contextually equivalent as values.

Our notion of contextual equivalence strays slightly from the usual notion. Viewing contexts as terms paired with closing substitutions spares us from having to define single-hole contexts and littering tedious reasoning about contexts and holes throughout our development. Although this is a non-trivial simplification, the coincidence of our definitions with the single-hole context definition is well-known and can be established using a standard technique such as logical relations, bisimulations, or the ciu theorem[Mason and Talcott 1991]. In any case, this choice is largely orthogonal to the point of this paper and we do not dwell on it further.

A common characterization of contextual equivalence is as the "greatest adequate congruence relation" [Pitts 2004]. As $\equiv$ is defined as a congruence closure and we have established adequacy in Lemma 2.3, it follows that the equational theory is included within contextual equivalence.

Lemma 2.6 (Soundness of equational theory). *If* $\Delta; \Gamma \vdash_c M_1 \equiv M_2 : \underline{A}$ *then* $\Delta; \Gamma \vdash_c M_1 \approx_{ctx} M_2 : \underline{A}$.

3 FOCUSING COMPUTATION

Traditionally, focusing has been employed as a proof search technique. Some work [Krishnaswami 2009; Zeilberger 2008b] does apply it through the lens of the Curry-Howard correspondence to study the design of programming language features. Though our ultimate goal is to extend this approach to enlist focusing as a tool for reasoning about program equivalences, we first review the standard approach to focusing through the lens of Curry-Howard. We explain the focusing rules for CBPV$^\forall$ from the point of view of a program synthesis technique as an aid to intuition. Previous work in functional type-directed program synthesis has alluded to the use of focusing to narrow down the synthesizer's search space [Frankle et al. 2016; Osera and Zdancewic 2015]. Synthesizers of stateful programs have also found its ideas useful [Polikarpova and Sergey 2019].

Consider the problem of synthesizing a program of type $\underline{A}$ under environment Γ that satisfies some specification S. We might attempt to search the space of well-typed expressions by repeatedly applying applicable typing rules until we have built a complete program and then attempt to verify that the program satisfies S. Unfortunately, this approach would lead us to visit numerous terms that are obviously equivalent, such as $(\lambda x : \mathbf{Bool}.\mathbf{return}\, x)$ $\mathbf{true}$ and $\mathbf{return}\,\mathbf{true}$.

Focusing cuts down the set of terms by eliminating such redundancy in the search space. By searching only through these focused terms, we visit fewer equivalent programs. At the same time, focusing manages to avoid sacrificing *completeness*. Completeness dictates that, given sufficient time, the algorithm should be able to discover *any* program, modulo equivalence.

3.1 Inversion

We now give a concrete presentation of focusing, heavily derived from that of [Simmons 2014]. Figure 5 inductively defines a focused subset of CBPV$^\forall$ terms in two phases. The first phase, *inversion*, is defined by the judgement $\Delta; \Gamma; \Omega \Vdash_{\text{inv}} N : \underline{A}$. This means that N is a focused computation of type $\underline{A}$ with type variables ascribed kinds by Δ, free variables of negative (thunk) type in the unordered environment Γ, and positive free variables in the ordered, suspension-free environment Ω. The ordering on Ω allows the focusing algorithm to process those variables in sequence, which reduces nondeterminism in the search. We write $\Omega \in \mathbf{ord}\,(\Gamma)$ when the entries in Ω are an ordering of those in Γ.

The inversion phase exploits the observation that some typing rules can be applied whenever they are applicable and without regard for their ordering, without losing completeness. For example, whenever we need to build a program with a variable $x : A + B$ in the environment, we might as well immediately pattern match on that variable. The left rule for sums (SumL) reflects this

$\boxed{\Delta; \Gamma \Vdash_{\mathrm{c}} M : \underline{A}}$ *(canonical computations)*

$$\textsc{CompFocTop}\quad \frac{\Delta; \cdot; \Omega \Vdash_{\mathrm{inv}} M : T \qquad \Omega \in \mathbf{ord}\,(\Gamma)}{\Delta; \Gamma \Vdash_{\mathrm{c}} M : T}$$

$\boxed{\Delta; \Gamma; \Omega \Vdash_{\mathrm{inv}} M : T}$ *(inversion)*

$$\textsc{FR}\quad \frac{\Delta; \Gamma \Vdash_{\mathrm{foc}} V : [A]}{\Delta; \Gamma; \cdot \Vdash_{\mathrm{inv}} \mathbf{return}\, V : \mathbf{F}\, A}$$

$$\textsc{UL}\quad \frac{\Delta; \Gamma, x : \mathbf{U}\underline{A}; [\underline{A}] \Vdash_{\mathrm{foc}} S : T \qquad \mathbf{stable}\; T}{\Delta; \Gamma, x : \mathbf{U}\underline{A}; \cdot \Vdash_{\mathrm{inv}} S[\mathbf{force}\, x] : T}$$

$$\textsc{EtaL}\quad \frac{\Delta; \Gamma, x : \langle X \rangle; \Omega \Vdash_{\mathrm{inv}} N : T}{\Delta; \Gamma; x : X, \Omega \Vdash_{\mathrm{inv}} N : T}$$

$$\textsc{UMove}\quad \frac{\Delta; \Gamma, x : \mathbf{U}\underline{A}; \Omega \Vdash_{\mathrm{inv}} N : T}{\Delta; \Gamma; x : \mathbf{U}\underline{A}, \Omega \Vdash_{\mathrm{inv}} N : T}$$

$$\textsc{SumL}\quad \frac{\Delta; \Gamma; y : A, \Omega \Vdash_{\mathrm{inv}} M : T \qquad \Delta; \Gamma; y' : B, \Omega \Vdash_{\mathrm{inv}} N : T}{\Delta; \Gamma; x : A + B, \Omega \Vdash_{\mathrm{inv}} \mathbf{pm}\, x\, \mathbf{as}\, \{(1, y).M, (2, y').N\} : T}$$

$$\textsc{PairL}\quad \frac{\Delta; \Gamma; y_1 : A_1, y_2 : A_2, \Omega \Vdash_{\mathrm{inv}} M : T}{\Delta; \Gamma; x : A_1 \times A_2, \Omega \Vdash_{\mathrm{inv}} \mathbf{pm}\, x\, \mathbf{as}\, (y_1, y_2).M : T}$$

$$\textsc{ExistsValL}\quad \frac{\Delta, X : \kappa; \Gamma; y : A, \Omega \Vdash_{\mathrm{inv}} M : T \qquad X \notin \mathbf{FTV}\,(T)}{\Delta; \Gamma; x : \exists X : \kappa.A, \Omega \Vdash_{\mathrm{inv}} \mathbf{pm}\, x\, \mathbf{as}\, (X, y).M : T}$$

$$\textsc{EtaR}\quad \frac{\Delta; \Gamma; \cdot \Vdash_{\mathrm{inv}} N : \langle \underline{X} \rangle}{\Delta; \Gamma; \cdot \Vdash_{\mathrm{inv}} N : \underline{X}}$$

$$\textsc{FunR}\quad \frac{\Delta; \Gamma; x : A \Vdash_{\mathrm{inv}} N : \underline{B}}{\Delta; \Gamma; \cdot \Vdash_{\mathrm{inv}} \lambda x : A.N : A \to \underline{B}}$$

$$\textsc{ValTypeFunR}\quad \frac{\Delta, X : \kappa; \Gamma; \cdot \Vdash_{\mathrm{inv}} N : \underline{B}}{\Delta; \Gamma; \cdot \Vdash_{\mathrm{inv}} \lambda X : \kappa.N : \forall X : \kappa.\underline{B}}$$

$$\textsc{ProdR}\quad \frac{\Delta; \Gamma; \cdot \Vdash_{\mathrm{inv}} M : \underline{A} \qquad \Delta; \Gamma; \cdot \Vdash_{\mathrm{inv}} N : \underline{B}}{\Delta; \Gamma; \cdot \Vdash_{\mathrm{inv}} \langle M, N \rangle : \underline{A}\ \&\ \underline{B}}$$

$\boxed{\Delta; \Gamma \Vdash_{\mathrm{foc}} V : [A]}$ *(right focus)*

$$\textsc{IdPlus}\quad \frac{}{\Delta; \Gamma, x : \langle A \rangle \Vdash_{\mathrm{foc}} x : [A]}$$

$$\textsc{UR}\quad \frac{\Delta; \Gamma; \Omega \Vdash_{\mathrm{inv}} N : \underline{A}}{\Delta; \Gamma \Vdash_{\mathrm{foc}} \mathbf{thunk}\, N : [\mathbf{U}\underline{A}]}$$

$$\textsc{PairR}\quad \frac{\Delta; \Gamma \Vdash_{\mathrm{foc}} V_1 : [A_1] \qquad \Delta; \Gamma \Vdash_{\mathrm{foc}} V_2 : [A_2]}{\Delta; \Gamma \Vdash_{\mathrm{foc}} (V_1, V_2) : [A_1 \times A_2]}$$

$$\textsc{ExistsValR}\quad \frac{\Delta \vdash_{\kappa} A : \kappa \qquad \Delta; \Gamma \Vdash_{\mathrm{foc}} V : [B[A/X]]}{\Delta; \Gamma \Vdash_{\mathrm{foc}} (A, V) : [\exists X : \kappa.B]}$$

$$\textsc{SumR}\quad \frac{\Delta; \Gamma \Vdash_{\mathrm{foc}} V : [A_i]}{\Delta; \Gamma \Vdash_{\mathrm{foc}} (i, V) : [A_1 + A_2]}$$

$\boxed{\Delta; \Gamma; [\underline{A}] \Vdash_{\mathrm{foc}} S : T \text{ where } \mathbf{stable}\; T}$ *(left focus)*

$$\textsc{IdMinus}\quad \frac{}{\Delta; \Gamma; [\underline{A}] \Vdash_{\mathrm{foc}} [\cdot] : \langle \underline{A} \rangle}$$

$$\textsc{FL}\quad \frac{\Delta; \Gamma; x : A \Vdash_{\mathrm{inv}} N : T}{\Delta; \Gamma; [\mathbf{F}\, A] \Vdash_{\mathrm{foc}} \mathbf{bind}\, x \leftarrow [\cdot].N : T}$$

$$\textsc{FunL}\quad \frac{\Delta; \Gamma \Vdash_{\mathrm{foc}} V : [A] \qquad \Delta; \Gamma; [\underline{B}] \Vdash_{\mathrm{foc}} S : T}{\Delta; \Gamma; [A \to \underline{B}] \Vdash_{\mathrm{foc}} S[[\cdot]\ V] : T}$$

$$\textsc{ValTypeFunL}\quad \frac{\Delta \vdash_{\kappa} A : \kappa \qquad \Delta; \Gamma; [\underline{B}[A/X]] \Vdash_{\mathrm{foc}} S : T}{\Delta; \Gamma; [\forall X : \kappa.\underline{B}] \Vdash_{\mathrm{foc}} S[[\cdot]\ A] : T}$$

$$\textsc{ProdL}\quad \frac{\Delta; \Gamma; [\underline{A_i}] \Vdash_{\mathrm{foc}} S : T}{\Delta; \Gamma; [\underline{A_1}\ \&\ \underline{A_2}] \Vdash_{\mathrm{foc}} S[[\cdot]\ i] : T}$$

The predicate $\mathbf{stable}\; T$ holds if and only if $T = \mathbf{F}\, B$ or $T = \langle \underline{B} \rangle$.

The predicate $\mathbf{stable}\; \Gamma$ holds if and only if for every $x : T$ in Γ, $T = \mathbf{U}\underline{A}$ or $T = \langle A \rangle$.

Fig. 5. Selected focusing rules

observation, which is justified by the following eta law for sums.

$$\frac{\Delta; \Gamma, x : A_1 + A_2 \vdash_{\mathrm{c}} N : \underline{B}}{\Delta; \Gamma, x : A_1 + A_2 \vdash_{\mathrm{c}} N \equiv_{\eta} \mathbf{pm}\, x\, \mathbf{as}\, \{(1, x_1).N[(1, x_1)/x], (2, x_2).N[(2, x_2)/x]\} : \underline{B}}$$

Dually, the eta law for functions (given below) states that every CBPV$^{\forall}$ term of function type is equivalent to a lambda. Thus, whenever we need to build a program of function type, we are justified in assuming that it is a lambda expression. This idea is captured in the right rule for functions FunR .

$$\frac{\Delta; \Gamma \vdash_c N : A \to \underline{B}}{\Delta; \Gamma \vdash_c N \equiv_\eta \lambda x : A.N\ x : A \to \underline{B}}$$

In general, inversion proceeds as follows:

- When synthesizing a term containing a variable of positive type, we immediately pattern match on the variable and then synthesize the bodies of each case of the matching construct. This process is embodied by the "left" inversion rules, which extract information from the ordered context Ω.
- When synthesizing a term of negative type, we select an introduction form as the outermost term constructor and then synthesize its body. This process is embodied by the "right" inversion rules.

In some cases, two different inversion rules may seem to be applicable, potentially forcing the program synthesizer to make a nondeterministic choice between them. For example, the synthesizer may have the choice of pattern matching one of many positively-typed variables in the context. Or, it may need to decide between matching on a positive variable or producing an introduction form for a negative type. In these cases, it doesn't matter which choice the synthesizer makes: it may apply them in any order while maintaining completeness. We refer to this situation as *don't care* or *conjunctive* nondeterminism. In order to avoid unnecessary nondeterminism in our synthesis procedure, we here deal with don't care nondeterminism implicitly, by fixing an arbitrary order.

(1) We first match positive variables in the context Ω from left to right. Any negative variable $x : \mathbf{U}\,\underline{A}$ encountered during inversion is shifted into the "negative context" Γ by the UMove rule.
(2) We then apply introduction forms for negative types on the right hand side of the judgement. (Note that the "right" inversion rules require the context Ω to be empty).

Variables of suspended type provide a way to skip parts of the inversion phase. A variable $x : \langle A \rangle$ in Γ is never destructed. The computational intuition here is that suspended types on the left correspond to variables which are used parametrically: the overall computation may return them or a value containing them, but will not inspect them directly. The left η rule EtaL states that when we are trying to build a term with a variable whose type is a free type variable, we may proceed by suspending the type. This makes sense because values of abstract type are always used parametrically.

On the right hand side of the focusing judgement, we may also suspend computation types. A computation of suspended type cannot be built directly with term constructors but rather must execute some other computation obtained from the context. The rule EtaR allows the suspension of abstract types on the right side since there is no appropriate term constructor to use.

We write Γ° and T° to refer to a given type or environment with all suspensions erased. A judgement is *suspension-normal* when the only suspended types are type variables. Since the η rules allow the suspsension only of type variables, subderivations of a suspension-normal judgement are themselves suspension-normal.

3.2 Focusing

Eventually, the inversion phase reaches a situation in which we need to build a positive (returner) or suspended computation containing only variables of negative (thunk) or suspended type. For

$$\begin{array}{rcl}
\mathbf{force}\ V[S/\mathbf{nil}] & = & S[\mathbf{force}\ V] \\
(\mathbf{pm}\ V\ \mathbf{as}\ (x,y).N)[S/\mathbf{nil}] & = & \mathbf{pm}\ V\ \mathbf{as}\ (x,y).N[S/\mathbf{nil}] \\
(\mathbf{pm}\ V\ \mathbf{as}\ (X,x).N)[S/\mathbf{nil}] & = & \mathbf{pm}\ V\ \mathbf{as}\ (X,x).N[S/\mathbf{nil}] \\
\mathbf{return}\ V[S/\mathbf{nil}] & = & S[\mathbf{return}\ V] \\
(\mathbf{bind}\ x \leftarrow M.N)[S/\mathbf{nil}] & = & \mathbf{bind}\ x \leftarrow M.N[S/\mathbf{nil}] \\
(\lambda x : A.N)[S/\mathbf{nil}] & = & S[\lambda x : A.N] \\
(\lambda X : \kappa.N)[S/\mathbf{nil}] & = & S[\lambda X : \kappa.N] \\
(M\ V)[S/\mathbf{nil}] & = & S[M\ V] \\
(M\ A)[S/\mathbf{nil}] & = & S[M\ A]
\end{array}$$

Fig. 6. Stack substitution (abbreviated)

example, when we need to synthesize the term N where

$$\Delta; x_1 : \mathbf{U}\,\underline{A}_1, \ldots, x_n : \mathbf{U}\,\underline{A}_n, y_1 : \langle A_1\rangle, \ldots, y_m : \langle A_m\rangle; \cdot \Vdash_{\text{inv}} N : \mathbf{F}\,B$$

This marks the end of the inversion phase and we now say that the judgement is *stable*.

Our options for constructing N are to either immediately return a value of type B or to force one of the thunks in the context. This choice is an essential source of nondeterminism in our search. In the presence of effects, a term that immediately returns a value is not equivalent to one that runs an arbitrary computation provided by the context. This situation is known as *don't know* or *disjunctive* nondeterminism: we *don't know* which choice to make to find the program we desire, so the synthesizer must search all possibilities.

Choosing to return a value immediately leads us to apply the first inversion rule shown in Figure 5. In this scenario, we build the term $N = \mathbf{return}\ V$ using the rule in which V is synthesized according to the right focus judgement $\Delta; x_1 : \mathbf{U}\,\underline{A}_1, \ldots, x_n : \mathbf{U}\,\underline{A}_n, y_1 : \langle A_1\rangle, \ldots, y_m : \langle A_m\rangle \Vdash_{\text{foc}} V : [B]$. This indicates that we are focusing on the type B on the right side of the turnstile.

On the other hand, we could apply the second rule in order to use the ith thunk from the context.[1] In this case, $N = S[\mathbf{force}\ x_i]$ where we synthesize S according to the left focus judgement $\Delta; x_1 : \mathbf{U}\,\underline{A}_1, \ldots, x_n : \mathbf{U}\,\underline{A}_n; [\underline{A}_i] \Vdash_{\text{foc}} S : \mathbf{F}\,B$. The square brackets around $\underline{A}_i$ indicate we are focusing on the left and that $\underline{A}_i$ is the type of the computation that S consumes.

The focusing phase of synthesis consists of repeatedly applying rules that break down a type in a particular position (on the right or left). For example, the right focus sum rule allows us to break down the type $A_1 + A_2$ into the type A_i where i is 1 or 2. The choice of i is another source of "don't know" nondeterminism. The presence of nondeterminism is characteristic of rules in the focusing phase. In any case, the focus moves from $A_1 + A_2$ to A_i in the child derivation. The most significant aspect of focusing is that we do not need to care about what is in the context at this point: we can continue breaking down the type on the right side of the turnstile and shifting focus to its components for as long as possible. Focusing ends when either:

- we apply an identity rule, constructing a variable or empty stack,
- we must return to the inversion phase in order to build a computation in order to construct a thunk, or
- we must return to the inversion phase because we introduced a new variable to the context representing the value returned by a returner computation.

3.3 Completeness of Focusing

We now establish the basic correctness property of focusing—namely, that it is *complete* in the sense that, even though there are far fewer focused terms, we can, nevertheless find for every (well-typed) CBPV$^\forall$ term M a focused term M_f such that M is equivalent to M_f. To prove that

[1] In the rule we have the unordered context $\Gamma, x : \mathbf{U}\,\underline{A} = x_1 : \mathbf{U}\,\underline{A}_1, \ldots, x_n : \mathbf{U}\,\underline{A}_n$, so $x = x_i$, for some i.

result, we first need to build up some metatheory that explains how to perform type substitution on focused terms and how to substitute one focused term into another.

3.3.1 Basic Properties. Recall that a variable x of suspended type in a focused term is used only parametrically. As a result, variables of suspended type can be replaced with focused values using standard substitution and no redex is formed. Due to the variety of syntactic structures of the language, we arrive at several mutually recursive notions of substitution. The most unfamiliar one, stack substitution, is shown in Figure 6. The stack substitution $M[S/\mathbf{nil}]$ produces a computation that is η equivalent to directly plugging M into S, but it pushes the stack S as deep into M as possible. We also define the substitution of stacks into stacks $S[S'/\mathbf{nil}]$ in a very similar manner but leave the definition to the appendix.

Lemma 3.1 (Suspended substitution).

(1) *If* $\Delta; \Gamma, x : \langle A \rangle; \Omega \Vdash_{\text{inv}} M : T$ *and* $\Delta; \Gamma \Vdash_{\text{foc}} V : [A]$ *then* $\Delta; \Gamma; \Omega \Vdash_{\text{inv}} M[V/x] : T$
(2) *If* $\Delta; \Gamma, x : \langle A \rangle \Vdash_{\text{foc}} V : [B]$ *and* $\Delta; \Gamma \Vdash_{\text{foc}} V' : [A]$ *then* $\Delta; \Gamma \Vdash_{\text{foc}} V[V'/x] : [B]$
(3) *If* $\Delta; \Gamma, x : \langle A \rangle; [\underline{B}] \Vdash_{\text{foc}} S : T$ *and* $\Delta; \Gamma \Vdash_{\text{foc}} V : [A]$ *then* $\Delta; \Gamma; [\underline{B}] \Vdash_{\text{foc}} S[V/x] : T$
(4) *If* $\Delta; \Gamma; \Omega \Vdash_{\text{inv}} M : \langle \underline{B} \rangle$ *and* $\Delta; \Gamma; [\underline{B}] \Vdash_{\text{foc}} S : T$ *then* $\Delta; \Gamma; \Omega \Vdash_{\text{inv}} M[S/\mathbf{nil}] : T$
(5) *If* $\Delta; \Gamma; [\underline{A}] \Vdash_{\text{foc}} S : \langle \underline{B} \rangle$ *and* $\Delta; \Gamma; [\underline{B}] \Vdash_{\text{foc}} S' : T$ *then* $\Delta; \Gamma; [\underline{A}] \Vdash_{\text{foc}} S[S'/\mathbf{nil}] : T$

Proof. By routine induction on the focused derivation of the term being substituted into. □

3.3.2 Type Substitution. Given a focused derivation such as $\Delta, X : \kappa; \Gamma; \Omega \Vdash_{\text{inv}} M : T$ and a type $\Delta \vdash_{\kappa} A : \kappa$, we would like to be able to obtain a new focused term $M[\![A/X]\!]$ in which A has been substitued for the free type variable X. This operation must be more than a simple syntactic type substitution. Suppose we have $x : X \in \Gamma$ and $A = \mathbf{Bool}$. While M, according to the eta rule, can ignore x because it is of parametric type, $M[\![\mathbf{Bool}/X]\!]$ must pattern match on x because $\mathbf{Bool}$ is a positive type.

Thus, in addition to performing a standard type substitution, we define the focused type substitution operator as follows. It finds each use of the η rule for X in the focused derivation of M and replaces it with the derived η principle for A given by Lemma 3.3 (described below). As there is no syntax that represents where in a term the η rules are used, type substitution must work over derivations rather than terms. This makes a more formally written definition tedious so we omit it. It also means that writing $M[\![A/X]\!]$ is technically an abuse of notation. It is, however, unambiguous so long as we take care to ensure the intended derivation for M is clear from context.

Lemma 3.2 (Type substitution).

(1) *If* $\Delta, X : \kappa; \Gamma; \Omega \Vdash_{\text{inv}} M : T$ *and* $\Delta \vdash_{\kappa} A : \kappa$ *then* $\Delta; \Gamma[A/X]; \Omega[A/X] \Vdash_{\text{inv}} M[\![A/X]\!] : T[A/X]$
(2) *If* $\Delta, \underline{X} : \kappa; \Gamma; \Omega \Vdash_{\text{inv}} M : T$ *and* $\Delta \vdash_{\kappa} \underline{A} : \kappa$ *then* $\Delta; \Gamma[\underline{A}/\underline{X}]; \Omega[\underline{A}/\underline{X}] \Vdash_{\text{inv}} M[\![\underline{A}/\underline{X}]\!] : T[\underline{A}/\underline{X}]$
(3) *If* $\Delta, X : \kappa; \Gamma \Vdash_{\text{foc}} V : [B]$ *and* $\Delta \vdash_{\kappa} A : \kappa$ *then* $\Delta; \Gamma[A/X] \Vdash_{\text{foc}} V[\![A/X]\!] : [B[A/X]]$
(4) *If* $\Delta, \underline{X} : \kappa; \Gamma \Vdash_{\text{foc}} V : [B]$ *and* $\Delta \vdash_{\kappa} \underline{A} : \kappa$ *then* $\Delta; \Gamma[\underline{A}/\underline{X}] \Vdash_{\text{foc}} V[\![\underline{A}/\underline{X}]\!] : [B[\underline{A}/\underline{X}]]$
(5) *If* $\Delta, X : \kappa; \Gamma; [\underline{B}] \Vdash_{\text{foc}} S : T$ *and* $\Delta \vdash_{\kappa} A : \kappa$ *then* $\Delta; \Gamma[A/X]; [\underline{B}[A/X]] \Vdash_{\text{foc}} S[\![A/X]\!] : T[A/X]$
(6) *If* $\Delta, \underline{X} : \kappa; \Gamma; [\underline{B}] \Vdash_{\text{foc}} S : T$ *and* $\Delta \vdash_{\kappa} \underline{A} : \kappa$ *then* $\Delta; \Gamma[\underline{A}/\underline{X}]; [\underline{B}[\underline{A}/\underline{X}]] \Vdash_{\text{foc}} S[\![\underline{A}/\underline{X}]\!] : T[\underline{A}/\underline{X}]$

Proof. By routine mutual induction on the first derivation in each part. In the η cases, identity expansion (Lemma 3.3) is applied. □

3.3.3 Identity Expansion. Identity expansion, shown in Figure 7, is used during type substitution, as explained above. Intuitively, it η-expands the uses of a variable of a suspended value type in a term M, or η-expands M itself, when M is a computation type.

$$\begin{array}{rcl}
\eta_{x:X}(M) & = & M \\
\eta_{x:\mathrm{U}\,\underline{A}}(M) & = & M[\textbf{thunk}\ \eta_{\underline{A}}(\textbf{force}\ x)/x] \\
\eta_{x:A_1+A_2}(M) & = & \textbf{pm}\ x\ \textbf{as}\ \{(1,x_1).\eta_{x_1:A_1}(M[(1,x_1)/x]),(2,x_2).\eta_{x_2:A_2}(M[(2,x_2)/x])\} \\
\eta_{x:\mathbf{Unit}}(M) & = & \textbf{pm}\ x\ \textbf{as}\ \textbf{unit}.M[\textbf{unit}/x] \\
\eta_{x:A_1\times A_2}(M) & = & \textbf{pm}\ x\ \textbf{as}\ (x_1,x_2).\eta_{x_2:A_2}(\eta_{x_1:A_1}(M[(x_1,x_2)/x])) \\
\eta_{x:\exists X:\kappa.A}(M) & = & \textbf{pm}\ x\ \textbf{as}\ (X,y).\eta_{y:A}(M[(X,y)/x]) \\
\eta_{x:\exists \underline{X}:\kappa.A}(M) & = & \textbf{pm}\ x\ \textbf{as}\ (\underline{X},y).\eta_{y:A}(M[(\underline{X},y)/x]) \\
\\
\eta_{\underline{X}}(M) & = & M \\
\eta_{\mathrm{F}\,A}(M) & = & M[\textbf{bind}\ x \leftarrow [\cdot].\eta_{x:A}(\textbf{return}\ x)/\textbf{nil}] \\
\eta_{A\to\underline{B}}(M) & = & \lambda x : A.\eta_{x:A}(\eta_{\underline{B}}(M[[\cdot]\ x/\textbf{nil}])) \\
\eta_{\forall X:\kappa.\underline{B}}(M) & = & \lambda X : \kappa.\eta_{\underline{B}}(M[[\cdot]\ X/\textbf{nil}]) \\
\eta_{\forall \underline{X}:\kappa.\underline{B}}(M) & = & \lambda \underline{X} : \kappa.\eta_{\underline{B}}(M[[\cdot]\ \underline{X}/\textbf{nil}]) \\
\eta_{\underline{A}_1\ \&\ \underline{A}_2}(M) & = & \langle \eta_{\underline{A}_1}(M[[\cdot]\ 1/\textbf{nil}]), \eta_{\underline{A}_2}(M[[\cdot]\ 2/\textbf{nil}])\rangle
\end{array}$$

Fig. 7. Identity expansion operations

Lemma 3.3 (Identity Expansion).

(1) *If* $\Delta;\Gamma,x:\langle A\rangle;\Omega \Vdash_{\mathrm{inv}} M : T$ *then* $\Delta;\Gamma;x:A,\Omega \Vdash_{\mathrm{inv}} \eta_{x:A}(M) : T$ *and* $\Delta;\Gamma,\Gamma' \vdash_{\mathrm{c}} M \equiv \eta_{x:A}(M) : T^{\circ}$ *for* $\Omega \in \textbf{ord}\,(\Gamma')$.

(2) *If* $\Delta;\Gamma;\cdot \Vdash_{\mathrm{inv}} M : \langle\underline{A}\rangle$ *then* $\Delta;\Gamma;\cdot \Vdash_{\mathrm{inv}} \eta_{\underline{A}}(M) : \underline{A}$.

Proof. By routine induction on the suspended type. In each case, we must apply the supsended substitution lemma (Lemma 3.1) and note that the operations performed by the η operator are merely η-expansions. □

3.3.4 Term Substitution. When substituting for a variable x in a focused term M that does not have suspended type, we risk building an unfocused term: the usual substitution $M[V/x]$ may yield an unfocused term even though V and M are. For example, **if** x **then return unit else return unit** is focused but **if true then return unit else return unit** is not. The solution is to β-reduce and η-expand the term as we substitute in order to maintain focus, a process known as *hereditary* or *focused substitution* [Eades III and Stump 2010; Pfenning 1995; Simmons 2014]. We write $M[\![V/x]\!]$ for the focused substitution of V for x in M and $S[\![M]\!]$ for the focused hole-filling of M into S. The definitions, which are given in the appendix, rely upon auxiliary operations handling substitution of computations for variables of thunk type and binding of values from returner computations to variables in other computations. To establish the correctness of these operations, we need to show that they yield terms that are equivalent to those given by their usual, unfocused counterparts.

Lemma 3.4 (Focused substitution).
Assume Γ *and* T *are suspension-normal and* $\Omega \in \textbf{ord}\,(\Gamma'')$.

(1) *If* $\Delta;\Gamma \Vdash_{\mathrm{foc}} V : [A]$ *and* $\Delta;\Gamma,\Gamma';x:A,\Omega \Vdash_{\mathrm{inv}} N : T$, *then* $\Delta;\Gamma,\Gamma';\Omega \Vdash_{\mathrm{inv}} N[\![V/x]\!] : T$ *and* $\Delta;\Gamma^{\circ},\Gamma'^{\circ},\Gamma'' \vdash_{\mathrm{c}} N[V/x] \equiv N[\![V/x]\!] : T^{\circ}$

(2) *If* $\Delta;\Gamma;\cdot \Vdash_{\mathrm{inv}} M : \underline{A}$ *and* $\Delta;\Gamma,\Gamma';[\underline{A}] \Vdash_{\mathrm{foc}} S : \underline{B}$, *then* $\Delta;\Gamma,\Gamma';\cdot \Vdash_{\mathrm{inv}} S[\![M]\!] : \underline{B}$ *and* $\Delta;\Gamma^{\circ},\Gamma'^{\circ} \vdash_{\mathrm{c}} S[M] \equiv S[\![M]\!] : \underline{B}$.

(3) *If* $\Delta;\Gamma;\cdot \Vdash_{\mathrm{inv}} M : \underline{A}$ *and* $\Delta;\Gamma,\Gamma',x:\mathbf{U}\,\underline{A};\Omega \Vdash_{\mathrm{inv}} N : T$, *then* $\Delta;\Gamma,\Gamma';\Omega \Vdash_{\mathrm{inv}} N[\![M/\textbf{force}\ x]\!] : T$ *and* $\Delta;\Gamma^{\circ},\Gamma'^{\circ},\Gamma'' \vdash_{\mathrm{c}} N[\textbf{thunk}\ M/x] \equiv N[\![M/\textbf{force}\ x]\!] : T^{\circ}$.

(4) *If* $\Delta;\Gamma;\cdot \Vdash_{\mathrm{inv}} M : \underline{A}$ *and* $\Delta;\Gamma,\Gamma',x:\mathbf{U}\,\underline{A} \Vdash_{\mathrm{foc}} V : [B]$, *then* $\Delta;\Gamma,\Gamma' \Vdash_{\mathrm{foc}} V[\![M/\textbf{force}\ x]\!] : [B]$ *and* $\Delta;\Gamma^{\circ},\Gamma'^{\circ} \vdash_{\mathrm{v}} V[\textbf{thunk}\ M/x] \equiv V[\![M/\textbf{force}\ x]\!] : B$.

(5) *If* $\Delta;\Gamma;\cdot \Vdash_{\mathrm{inv}} M : \underline{A}$ *and* $\Delta;\Gamma,\Gamma',x:\mathbf{U}\,\underline{A};[\underline{B}] \Vdash_{\mathrm{foc}} S : T$, *then* $\Delta;\Gamma,\Gamma';[\underline{B}] \Vdash_{\mathrm{foc}} S[\![M/\textbf{force}\ x]\!] : T$ *and* $\Delta;\Gamma^{\circ},\Gamma'^{\circ};\underline{B} \vdash S[\textbf{thunk}\ M/x] \equiv S[\![M/\textbf{force}\ x]\!] : T^{\circ}$.

(6) *If* $\Delta;\Gamma;\Omega \Vdash_{\mathrm{inv}} M : \mathbf{F}\,A$ *and* $\Delta;\Gamma,\Gamma';x:A \Vdash_{\mathrm{inv}} N : T$ *where* $\textbf{stable}\ T$, *then* $\Delta;\Gamma,\Gamma';\Omega \Vdash_{\mathrm{inv}} N[\![x \leftarrow M]\!] : T$ *and* $\Delta;\Gamma^{\circ},\Gamma'^{\circ},\Gamma'' \vdash_{\mathrm{c}} \textbf{bind}\ x \leftarrow M.N \equiv N[\![x \leftarrow M]\!] : T^{\circ}$.

(7) *If* $\Delta; \Gamma; [\underline{B}] \Vdash_{\text{foc}} S : \mathbf{F}\,A$ *and* $\Delta; \Gamma, \Gamma'; x : A \Vdash_{\text{inv}} N : T$ *where* **stable** T*, then* $\Delta; \Gamma, \Gamma'; [\underline{B}] \Vdash_{\text{foc}} N[\![x \leftarrow S]\!] : T$.

Proof. We prove these claims by lexicographic induction on the following components:

- The universe level of the principal type.
- The formula size of the principal type.
- The "part size" (as in parts 1-7) decreases.
- In parts 3, 4, and 5, the focused derivation of the computation N, stack S, or value V being substituted into.
- In the case of the focused bind operators (parts 6 and 7), the focused derivation of the computation M or stack S being bound.

For cases 1-5, the principal type is the type of the value or computation being substituted or filled in. For the focused bind cases, the principal type is the type of the variable x being bound. □

3.3.5 *Completeness.*

We now prove our main completeness theorem simultaneously with a couple of lemmas. We first present Theorem 3.5, which is the top-level property claiming that for every well-typed term, there is an equivalent focused term. This result follows by mutual induction on the typing derivations of M (in Lemma 3.6 and Lemma 3.7) and V (in Lemma 3.8). The full proofs can be found in the appendix.

Theorem 3.5 (Completeness). *Assume* $\Delta; \Gamma \vdash_{\text{c}} M : \underline{B}$*. Then there exists* M_f *such that* $\Delta; \Gamma \Vdash_{\text{c}} M_f : \underline{B}$ *and* $\Delta; \Gamma \vdash_{\text{c}} M \equiv M_f : \underline{B}$.

Proof. Let $\Omega \in \mathbf{ord}\,(\Gamma)$. By instantiating Lemma 3.6 with an empty stack and substitution, we get that there exists M_f equivalent to M such that $\Delta; \cdot; \Omega \Vdash_{\text{inv}} M_f : \langle \underline{B} \rangle$. From identity expansion, we get $\eta_{\underline{B}}(M_f)$ a term equivalent to M_f (and thus also M) such that $\Delta; \cdot; \Omega \Vdash_{\text{inv}} \eta_{\underline{B}}(M_f) : \underline{B}$. This completes the proof. □

The lemma structure largely follows that of the focusing rules. Lemma 3.6 corresponds to the inversion phase. Consider an unfocused computation M with a variable x of type $A_1 \times A_2$. A focused term must immediately pattern match on x. We would like to η-expand M into the equivalent term $\mathbf{pm}\,x\,\mathbf{as}\,(x_1, x_2).M[(x_1, x_2)/x]$ and then continue by focusing the computation $M[(x_1, x_2)/x]$ in the body of the matching construct. Unfortunately, this term is not strictly smaller than M, so a naive approach does not satisfy our induction scheme.

We can, however, make the observation that the value (x_1, x_2) is already focused in an environment where x_1 and x_2 have suspension type. Rather than substituting (x_1, x_2) directly into M, then, we are able to keep around a mapping ω from variables to focused values. Dually, we keep track of a stack S which is used in the same way when η-expanding computations of negative type.

In the base cases, after all possible η-expansion has completed and the judgement is stable, Lemma 3.7 is used to focus the resulting term.

In all of the following claims, suppose that $\Omega_1 \in \mathbf{ord}\,(\Gamma_1)$ and $\Omega_2 \in \mathbf{ord}\,(\Gamma_2)$ where Γ_1 and Γ_2 are suspension-free. Furthermore, assume Γ is stable and suspension-normal.

Lemma 3.6. *Assume*

- $\Delta; \Gamma^\circ, \Gamma_1, \Gamma_2 \vdash_{\text{c}} M : \underline{B}$,
- $\Delta; \Gamma, \langle \Gamma_2 \rangle; [\underline{B}] \Vdash_{\text{foc}} \omega(S) : T$
- $\Delta; \Gamma, \langle \Gamma_2 \rangle \Vdash_{\text{c}} \omega : \Omega_1$*, and*

Then there exists M_f *such that* $\Delta; \Gamma; \Omega_2 \Vdash_{\text{inv}} M_f : T^\circ$ *and* $\Delta; \Gamma, \Gamma_2 \vdash_{\text{c}} \omega(M[S/\mathbf{nil}]) \equiv M_f : T^\circ$.

Given a computation which is typed under a stable judgement, Lemma 3.7 beta reduces it as much as possible and proceeds recursively into subexpressions to obtain a focused term.

Lemma 3.7. *Assume*

- $\Delta; \Gamma^\circ, \Gamma_1 \vdash_c M : \underline{B}$,
- $\Delta; \Gamma \Vdash_c \omega : \Omega_1$,
- $\Delta; \Gamma; [\underline{B}] \Vdash_{foc} \omega(S) : T$ *where* **stable** Γ *and* **stable** T.

Then there exists M_f *such that* $\Delta; \Gamma; \cdot \Vdash_{inv} M_f : T$ *and* $\Delta; \Gamma^\circ \vdash_c \omega(M[S/\mathbf{nil}]) \equiv M_f : T^\circ$.

Lemma 3.8. *Assume* $\Delta; \Gamma^\circ, \Gamma_1 \vdash_v V : B$ *and* $\Delta; \Gamma \Vdash_c \omega : \Omega_1$. *Then there exists* V_f *such that* $\Delta; \Gamma \Vdash_{foc} V_f : [B]$ *and* $\Delta; \Gamma^\circ \vdash_v \omega(V) \equiv V_f : B$.

4 EXAMPLE APPLICATIONS OF COMPUTATION FOCUSING

In this section, we illustrate some of the uses for computation focusing, which has interesting applications even in the simple predicative, polymorphic CBPV$^\forall$ language defined above.

4.1 "Free Theorems": Polymorphic Identity Function

As a first example, we show how computation focusing can replicate some of the "free theorems" [Wadler 1989] that would typically be proved directly via a logical relations arguments. For instance, consider a closed term M of type $\forall X : \kappa.X \to \mathbf{F}\,X$. A classic free theorem for this type tells us that M must be contextually equivalent to the polymorphic identity function. Rather than using a logical relations argument, examining the possible derivations that we may obtain from focusing M yields a particularly simple alternative proof of this property.

Completeness of focusing (Theorem 3.5) states that there must exist a focused term M_f which is equivalent to M. Simply by inversion on the derivation of $\cdot; \cdot; \cdot \Vdash_{inv} M_f : \forall X : \kappa.X \to \mathbf{F}\,X$ we find $M_f = \lambda X : \kappa.\lambda x : X.\mathbf{return}\, x$. Thus, M is equivalent to the polymorphic identity function.

4.2 Corollary to Completeness: Context Lemmas

As a stepping stone to more interesting applications of focusing, we next show that computational focusing indeed does cut down the search space of contexts that we must consider to prove contextual equivalence. Intuitively, to prove that two CBPV$^\forall$ computations are contextually equivalent it suffices to examine their behaviors in all *focused* stacks, rather than arbitrary contexts. Similarly, to show that two values are contextually equivalent, it suffices to show that they yield similar behaviors when placed into each *focused* term, rather than an arbitrary context.

These intuitions are captured in the following lemmas.

Lemma 4.1. *Assume* $\Omega \in \mathbf{ord}\,(\Gamma)$.
Suppose that for all M_f, ω, *and* δ *such that* $\cdot; x : A \Vdash_c M_f : \mathbf{F\,Bool}$ *and* $\cdot; \cdot \Vdash_c \omega : \Omega$ *and* $\cdot \vdash \delta : \Delta$ *we have* $M_f[\omega(\delta(V_1))/x] \mapsto^* \mathbf{return\,true}$ *iff* $M_f[\omega(\delta(V_2))/x] \mapsto^* \mathbf{return\,true}$.
Then, $\Delta; \Gamma \vdash_v V_1 \approx_{ctx} V_2 : A$

Lemma 4.2. *Assume* $\Omega \in \mathbf{ord}\,(\Gamma)$.
Suppose that for all M_f, ω, *and* δ *such that* $\cdot; x : \mathbf{U}\,\underline{A} \Vdash_c M_f : \mathbf{F\,Bool}$ *and* $\cdot; \cdot \Vdash_c \omega : \Omega$ *and* $\cdot \vdash \delta : \Delta$ *we have* $M_f[\mathbf{thunk}\ \omega(\delta(M_1))/x] \mapsto^* \mathbf{return\,true}$ *iff* $M_f[\mathbf{thunk}\ \omega(\delta(M_2))/x] \mapsto^* \mathbf{return\,true}$.
Then, $\Delta; \Gamma \vdash_c M_1 \approx_{ctx} M_2 : \underline{A}$

To prove each of these, we start by considering an arbitrary context M and closing substitution γ with the goal of showing that substituting some values closed by γ into M yields equivalent results. Our premise allows us to substitute these values into any *focused* context and get equivalent results. We simply apply completeness of focusing (Theorem 3.5) to the context M and γ to obtain equivalent, focused versions of these.

4.3 Tricky Existential Packages

As a more sophisticated example of computation focusing, we can use it to prove the contextual equivalence of existential packages, even in cases where standard proof techniques, such as logical relations, fall short. We show a (somewhat contrived) but small and illustrative example of this phenomenon, adapted to our CBPV$^\forall$ setting from similar examples from the literature [Pitts 2004; Sumii and Pierce 2005]. Note that the proof relies essentially on the ability to do induction on focused derivations.

Let $\mathbb{Z}_3$ be the type **Unit** + **Unit** + **Unit**. We refer to the inhabitants of this type as 0, 1, and 2. Consider the type $A = \exists X : \mathbf{Type}^u.\mathbf{U}\,(\mathbf{U}\,(X \to \mathbf{F}\,X) \to \mathbf{F}\,\mathbf{Bool})$ and the following two closed values of this type:

$$\begin{array}{lcl} V_1 & = & (\mathbf{Bool}, \mathbf{thunk}\ (\lambda f : \mathbf{U}\,(\mathbf{Bool} \to \mathbf{F}\,\mathbf{Bool}).\mathbf{force}\,f\ \mathbf{true} \wedge \neg\,(\mathbf{force}\,f\ \mathbf{false}))) \\ V_2 & = & (\mathbb{Z}_3, \mathbf{thunk}\ \lambda f : \mathbf{U}\,(\mathbb{Z}_3 \to \mathbf{F}\,\mathbb{Z}_3).\mathbf{bind} \\ & & \qquad x_0 \leftarrow \mathbf{force}\,f\ 0, x_1 \leftarrow \mathbf{force}\,f\ 1, x_2 \leftarrow \mathbf{force}\,f\ 2. \\ & & \qquad x_0 = 0 \wedge x_1 = 1 \wedge x_2 = 2) \end{array}$$

The packages V_1 and V_2 are equivalent because the existential ensures that the only argument that can be passed in for f is the identity function—the context does not have any knowledge about X and no means of constructing or destructing values of type X, so, intuitively, it is forced to provide the identity function. In the case that f is instantiated with the identity, then the functions in both V_1 and V_2 always return true.

Directly applying the ciu theorem for this example would allow us to narrow down the contexts we need to consider to **bind** $w \leftarrow [\cdot].M$ where

$$\cdot; w : \exists X : \mathbf{Type}^u.\mathbf{U}\,(\mathbf{U}\,(X \to \mathbf{F}\,X) \to \mathbf{F}\,\mathbf{Bool}) \vdash_c M : \mathbf{F}\,\mathbf{Bool}$$

It remains to show that

$$\mathbf{bind}\ w \leftarrow \mathbf{return}\ V_1.M \equiv M[V_1/w]$$

evaluates to the same value as

$$\mathbf{bind}\ w \leftarrow \mathbf{return}\ V_2.M \equiv M[V_2/w]$$

We would like to take advantage of the earlier intuition that the only interesting thing M might do is break apart w via pattern matching into (X, y) and then apply y to some value morally equivalent to the identity function. Sadly for us, the ciu theorem is not particularly helpful here since it gives us no information about how the variable w is used by M.

Instead, we use focusing to prove the equivalence. Consider an arbitrary focused context M_f, where $\cdot; x : A \Vdash_c M_f : \mathbf{F}\,\mathbf{Bool}$. By Lemma 4.1, it suffices to show that there exists $V \in \{\mathbf{true}, \mathbf{false}\}$ such that $M_f[V_1/x] \mapsto^* \mathbf{return}\ V$ and $M_f[V_2/x] \mapsto^* \mathbf{return}\ V$.

Inverting the focusing derivation, we obtain $M_f = \mathbf{pm}\ x\ \mathbf{as}\ (X, y).M'$ where

$$X : \mathbf{Type}^u; \cdot; y : \mathbf{U}\,(\mathbf{U}\,(X \to \mathbf{F}\,X) \to \mathbf{F}\,\mathbf{Bool}) \Vdash_{\mathrm{inv}} M' : \mathbf{F}\,\mathbf{Bool}$$

It now suffices to show

$$M'[\mathbf{Bool}/X][\mathbf{thunk}\ (\lambda f : \mathbf{U}\,(\mathbf{Bool} \to \mathbf{F}\,\mathbf{Bool}).(\mathbf{force}\,f)\ \mathbf{true} \wedge \neg\,((\mathbf{force}\,f)\ \mathbf{false}))/y]$$

returns the same value as

$$\begin{array}{ll} M'[\mathbb{Z}_3/X][\mathbf{thunk}\ \lambda f & : \mathbf{U}\,(\mathbb{Z}_3 \to \mathbf{F}\,\mathbb{Z}_3).\mathbf{bind} \\ & x_0 \leftarrow \mathbf{force}\,f\ 0, x_1 \leftarrow \mathbf{force}\,f\ 1, x_2 \leftarrow \mathbf{force}\,f\ 2. \\ & x_0 = 0 \wedge x_1 = 1 \wedge x_2 = 2/x] \end{array}$$

From here, we proceed by induction on the focused derivation of M'.

Case: $M' = \mathbf{return\,true}$ or $M' = \mathbf{return\,false}$
In these cases, M' ignores the function it is given, so we immediately have that both programs return the same value.

Case: $M' = \mathbf{bind}\ z \leftarrow \mathbf{force}\ y\ \mathbf{thunk}\ (\lambda w : X.M'').\mathbf{if}\ z\ \mathbf{then}\ N_1\ \mathbf{else}\ N_2$

By inversion we have

- $X : \mathbf{Type}^u; y : \mathbf{U}\,(\mathbf{U}\,(X \to \mathbf{F}\,X) \to \mathbf{F}\,\mathbf{Bool}), w : \langle X \rangle; \cdot \Vdash_{\text{inv}} M'' : \mathbf{F}\,X$ and
- $X : \mathbf{Type}^u; y : \mathbf{U}\,(\mathbf{U}\,(X \to \mathbf{F}\,X) \to \mathbf{F}\,\mathbf{Bool}); \cdot \Vdash_{\text{inv}} N_1 : \mathbf{F}\,\mathbf{Bool}$.

In this case, M' is calling y with a function $\mathbf{thunk}\ (\lambda w : X.M'')$ of type $\mathbf{U}\,(X \to \mathbf{F}\,X)$. The crux of this proof is showing that this argument must be the identity function. In other words, M'' must return w. The reason this is trickier than our earlier proof of the polymorphic identity function is that the variable y is free in M''. We will prove this in Lemma 4.3, whose proof we defer.

Applying Lemma 4.3, $X : \mathbf{Type}^u; y : \mathbf{U}\,(\mathbf{U}\,(X \to \mathbf{F}\,X) \to \mathbf{F}\,\mathbf{Bool}), w : X \vdash_c M'' \equiv \mathbf{return}\ w : \mathbf{F}\,X$. Now that we have shown that y is passed the identity function, we can examine the functions in V_1 and V_2 to confirm that whenever the identity function is passed in for f, they both return **true**. Applying this observation, it suffices to show

$$\begin{array}{l}
\mathbf{bind}\ z \leftarrow \mathbf{return\,true}.\\
\quad \mathbf{if}\ z\ \mathbf{then}\\
\qquad N_1[\mathbf{Bool}/X][\mathbf{thunk}\ (\lambda f : \mathbf{U}\,(\mathbf{Bool} \to \mathbf{F}\,\mathbf{Bool}).(\mathbf{force}\,f)\ \mathbf{true} \land \neg\,((\mathbf{force}\,f)\ \mathbf{false}))/y]\\
\quad \mathbf{else}\\
\qquad N_2[\mathbf{Bool}/X][\mathbf{thunk}\ (\lambda f : \mathbf{U}\,(\mathbf{Bool} \to \mathbf{F}\,\mathbf{Bool}).(\mathbf{force}\,f)\ \mathbf{true} \land \neg\,((\mathbf{force}\,f)\ \mathbf{false}))/y]\\
\mapsto^*\\
N_1[\mathbf{Bool}/X][\mathbf{thunk}\ (\lambda f : \mathbf{U}\,(\mathbf{Bool} \to \mathbf{F}\,\mathbf{Bool}).(\mathbf{force}\,f)\ \mathbf{true} \land \neg\,((\mathbf{force}\,f)\ \mathbf{false}))/y]
\end{array}$$

returns the same value as

$$\begin{array}{l}
\mathbf{bind}\ z \leftarrow \mathbf{return\,true}.\\
\quad \mathbf{if}\ z\ \mathbf{then}\\
\qquad N_1[\mathbb{Z}_3/X][\mathbf{thunk}\ \lambda f : \mathbf{U}\,(\mathbb{Z}_3 \to \mathbf{F}\,\mathbb{Z}_3).\mathbf{bind}\ x_0 \leftarrow \mathbf{force}\,f\ 0, x_1 \leftarrow \mathbf{force}\,f\ 1,\\
\qquad\qquad x_2 \leftarrow \mathbf{force}\,f\ 2.x_0 = 0 \land x_1 = 1 \land x_2 = 2/y]\\
\quad \mathbf{else}\\
\qquad N_2[\mathbb{Z}_3/X][\mathbf{thunk}\ \lambda f : \mathbf{U}\,(\mathbb{Z}_3 \to \mathbf{F}\,\mathbb{Z}_3).\mathbf{bind}\ x_0 \leftarrow \mathbf{force}\,f\ 0, x_1 \leftarrow \mathbf{force}\,f\ 1,\\
\qquad\qquad x_2 \leftarrow \mathbf{force}\,f\ 2.x_0 = 0 \land x_1 = 1 \land x_2 = 2/y]\\
\mapsto^*\\
N_1[\mathbb{Z}_3/X][\mathbf{thunk}\ \lambda f : \mathbf{U}\,(\mathbb{Z}_3 \to \mathbf{F}\,\mathbb{Z}_3).\mathbf{bind}\ x_0 \leftarrow \mathbf{force}\,f\ 0, x_1 \leftarrow \mathbf{force}\,f\ 1,\\
\qquad x_2 \leftarrow \mathbf{force}\,f\ 2.x_0 = 0 \land x_1 = 1 \land x_2 = 2/y]
\end{array}$$

But this follows thanks to the induction hypothesis for N_1, whose derivation is a subderivation of that of M'.

Now all that remains is to return to the proof of Lemma 4.3, the key parametricity property we relied on. Unlike the reasoning we just described, the proof of this lemma could be given using a logical relation. However, as we already have the machinery set up for a focusing-based proof, we opt to continue with this approach instead.

Lemma 4.3. *For $n \geq 1$ assume:*

- $X : \mathbf{Type}^u; y : \mathbf{U}\,(\mathbf{U}\,(X \to \mathbf{F}\,X) \to \mathbf{F}\,\mathbf{Bool}), w_1 : \langle X \rangle, \ldots, w_n : \langle X \rangle; \cdot \Vdash_{\text{inv}} M : \mathbf{F}\,X$ *and*
- $\cdot; \cdot \vdash_v V : \mathbf{U}\,(\mathbf{U}\,(A \to \mathbf{F}\,A) \to \mathbf{F}\,\mathbf{Bool})$

Then, there exists some j such that

$$X : \mathbf{Type}^u; w_1 : X, \ldots, w_n : X \vdash_c M[A/X][V/y] \equiv \mathbf{return}\ w_j : \mathbf{F}\,A$$

Proof. We proceed by induction on the focused derivation of M. Applying theorem 3.5, let V_f be a focused value equivalent to V.

Case: $M = \mathbf{return}\ w_i$

This case is immediate since $(\mathbf{return}\ w_i)[A/X][V/y] = \mathbf{return}\ w_i$.

Case: $M = \mathbf{bind}\ z \leftarrow \mathbf{force}\ y\ (\mathbf{thunk}\ \lambda w : X.N).\mathbf{if}\ z\ \mathbf{then}\ N_1\ \mathbf{else}\ N_2$

We are required to show

$$\begin{aligned} \cdot; \Gamma \vdash_c\ & \mathbf{bind}\ z \leftarrow (\mathbf{force}\ y\ (\mathbf{thunk}\ \lambda w : X.N))[A/X][V/y]. \\ & \quad \mathbf{if}\ z\ \mathbf{then}\ N_1[A/X][V/y]\ \mathbf{else}\ N_2[A/X][V/y] \equiv \\ & \mathbf{return}\ w_i : \mathbf{F}\,A \end{aligned}$$

By inversion,

$$y : \mathbf{U}\,(\mathbf{U}\,(X \to \mathbf{F}\,X) \to \mathbf{F}\,\mathbf{Bool}), w_1 : \langle X \rangle, \ldots, w_n : \langle X \rangle; \cdot \Vdash_{X:\mathbf{Type}^u} \\ \mathbf{force}\ y\ (\mathbf{thunk}\ \lambda w : X.N) : \mathbf{F}\,\mathbf{Bool}$$

Applying focused substitution (Lemma 3.4),

$$\cdot; w_1 : \langle A \rangle, \ldots, w_n : \langle A \rangle; \cdot \Vdash_{\mathrm{inv}} (\mathbf{force}\ y\ (\mathbf{thunk}\ \lambda w : X.N))[\![A/X]\!][\![V_f/y]\!] : \mathbf{F}\,\mathbf{Bool}$$

Inverting this derivation, we see that $(\mathbf{force}\ y\ (\mathbf{thunk}\ \lambda w : X.N))[\![A/X]\!][\![V_f/y]\!]$ must be equal to either **return true** or **return false**. The two cases of the proof are analagous; we will proceed assuming the former. This lets us determine:

$$\begin{aligned} \Delta; \Gamma \vdash_c\ & \mathbf{bind}\ z \leftarrow (\mathbf{force}\ y\ (\mathbf{thunk}\ \lambda w : X.N))[A/X][V_f/y]. \\ & \quad \mathbf{if}\ z\ \mathbf{then}\ N_1[A/X][V/y]\ \mathbf{else}\ N_2[A/X][V/y] \\ \equiv\ & \mathbf{bind}\ z \leftarrow \mathbf{return}\ \mathbf{true}. \\ & \quad \mathbf{if}\ z\ \mathbf{then}\ N_1[A/X][V/y]\ \mathbf{else}\ N_2[A/X][V/y] \\ \equiv\ & N_1[A/X][V/y] : \mathbf{F}\,A \end{aligned}$$

By the induction hypothesis, there exists some i such that $N_1[A/X][V/y]$ is equivalent to $\mathbf{return}\ w_i$, completing the proof. □

5 COMPILER CORRECTNESS: FULL ABSTRACTION

As a more extensive demonstration of how computation focusing can be applied, we show in this section how it can help in proving full abstraction results.

5.1 Full Abstraction

Full abstraction is a strong compiler correctness property. It requires that two programs are contextually equivalent in the source language if and only if their compilations are equivalent in the target. In other words, equivalence must be preserved and reflected. Full abstraction is compositional in that it provides guarantees about the behavior of a compiled component regardless of the code it is linked with. It is not restricted to whole-program compilers and enables the linking of a component with other components that may be written in entirely different source languages.

Proofs of full abstraction are about comparing the observations that can be made about a source language term with those that can be made in the target language on the compilation of the source term. If some target language context is able to make an observation that distinguishes two programs which were equivalent in the source, then the compiler does not enforce the source language's abstractions and the programmer cannot reason about a component's behavior using their knowledge of the language it was written in. In this case, full abstraction fails because the compiler does not preserve equivalence. On the other hand, if two different source-language

$$\begin{array}{rcl} A, B & ::= & \mathbf{U}\,\underline{A} \mid A + B \mid \mathbf{Unit} \mid A \times B \\ \underline{A}, \underline{B} & ::= & \mathbf{F}\,A \mid A \to \underline{B} \mid \underline{A} \,\&\, \underline{B} \end{array}$$

Fig. 8. CBPV$_s$: source language types

programs are compiled to the same target language program (i.e. equivalence is not reflected), the compiler is changing the behavior of some programs.

Designing compilers to be fully abstract often requires careful consideration of how source-language features are encoded into the target. This is particularly important when compiling less expressive source languages to more expressive target languages. Using the target language's type system to regulate the observations which can be made about the compiler's output is, in many cases, crucial.

We draw the correspondence between target and source language contexts through a *back-translation* relation which relates a target-level context for a compiled component to a corresponding source-language context. The idea is that if there is some target-language context which can distinguish two compiled components, its back-translation can distinguish the original source-level components.

If the target language is more expressive than the source language, then we cannot hope to back-translate every target context. Since the hole of the back-translated context must have a type that is a translation of some source language type, we can narrow down the set of contexts that concern us. As we have seen before, when we have type information describing a context, focusing can generally be applied to enumerate the possible shapes of the context. In the case of proving full abstraction, this means that fewer contexts need be back-translated.

For example, the consistency of dependently-typed languages like Coq hinges on the fact that that every program terminates. Languages like OCaml place no such restrictions on their programs. OCaml programmers, however, may wish to link their programs with highly-reliable verified Coq components. However, building a compiler from Coq to OCaml with a compositional correctness theorem poses many challenges. An important first step in this direction is to design a way for a component written in a total language to be compiled fully-abstractly to an intermediate language in which it can link with potentially nonterminating components. In the remainder of this section, we establish full abstraction for the extremely simplified case in which the source language is a terminating, simply-typed language and the target is CBPV$^{\forall}$plus a divergent term and termination-sensitive typing.

We study a primitive compiler because even in this idealized setting, current proof techniques fall short. The challenge of full abstraction proofs generally has more to do with capturing the relationship between the source and target language than the translation itself. And, unfortunately, current techniques are limited in their ability to reason about the operations that target-level contexts may perform on a compiled component. The simplicity of the embedding only makes it all the more frustrating that we have hitherto been unable to prove this essential compiler correctness property.

5.2 Setup

To this end, we introduce CBPV$_s$, a simply typed subset of CBPV$^{\forall}$ and CBPV$_t^{\forall\bullet}$, which includes both polymorphism and a diverging computation **diverge**. Note that naively embedding CBPV$_s$ into CBPV$_t^{\forall\bullet}$ is not fully abstract. In a terminating language, no context can differentiate between $M_1 = \lambda x : \mathbf{U}\,(\mathbf{F}\,\mathbf{Unit}).\mathbf{force}\,x$, a function that forces a thunk of unit type provided by the context and $M_2 = \lambda x : \mathbf{U}\,(\mathbf{F}\,\mathbf{Unit}).\mathbf{return}\,\mathbf{unit}$, a function that ignores the thunk and simply returns the

$$\begin{array}{rcl} \epsilon & ::= & \bullet \mid \circ \\ \underline{A}, \underline{B} & ::= & \ldots \mid \mathbf{F}_\epsilon A \\ M, N & ::= & \ldots \mid \mathbf{diverge} \end{array} \qquad \eta_{\mathbf{F}_\epsilon A}(M) \;=\; M[\mathbf{bind}\, x \leftarrow [\cdot].\eta_{x:A}(\mathbf{return}\, x)/\mathbf{nil}]$$

$$\frac{}{\Delta; \Gamma \vdash_{\mathrm{c}} \mathbf{bind}\, x \leftarrow \mathbf{diverge}.M \equiv \mathbf{diverge} : \mathbf{F}_\bullet A} \qquad \frac{}{\mathbf{diverge} \mapsto \mathbf{diverge}}$$

$$\frac{\Delta \vdash_\kappa A : \kappa}{\Delta; \Gamma; \cdot \Vdash_{\mathrm{inv}} \mathbf{diverge} : \mathbf{F}_\bullet A} \qquad \frac{\Delta; \Gamma \Vdash_{\mathrm{foc}} V : [A]}{\Delta; \Gamma; \cdot \Vdash_{\mathrm{inv}} \mathbf{return}\, V : \mathbf{F}_\epsilon A} \qquad \frac{\Delta; \Gamma; x : A \Vdash_{\mathrm{inv}} N : T \quad \mathbf{lub}\,(T) \geq \epsilon}{\Delta; \Gamma; [\mathbf{F}_\epsilon A] \Vdash_{\mathrm{foc}} \mathbf{bind}\, x \leftarrow [\cdot].N : T}$$

Fig. 9. CBPV$_t^{\forall\bullet}$: syntax and semantics differences from CBPV$^\forall$and key focusing definitions

unit value. On the other hand, the CBPV$_t^{\forall\bullet}$ stack $S = [\cdot]$ ($\mathbf{thunk\ diverge}$) distinguishes M_1 and M_2 by providing a diverging thunk as an argument. Thus, a fully abstract compiler must ensure that no non-terminating CBPV$_t^{\forall\bullet}$ function is ever passed to compiled CBPV$_s$ code.

To statically enforce this separation, the type system of CBPV$_t^{\forall\bullet}$ extends that of CBPV$^\forall$ in order to track which computations may diverge. The CBPV$_t^{\forall\bullet}$ returner type $\mathbf{F}_\epsilon A$ is annotated with an effect ϵ, where $\epsilon = \bullet$ indicates the computation may diverge and $\epsilon = \circ$ indicates a total computation. The key definitions for the language, its focused forms, and extending the equational theory are given in Figure 9. The crucial rule is the focusing rule for bind, which says that the whole term might diverge if the computation bound to x might. Note that $\circ \leq \bullet$ and $\mathbf{lub}(T)$ traverses T, going only on the right-hand side of function types, and computes the least upper bound of the effect annotations in the types it visits.

We aim to prove fully abstract a simple compiler that embeds CBPV$_s$ into the pure subset of CBPV$_t^{\forall\bullet}$. The judgements $\Delta \vdash_\kappa \underline{A} : \mathbf{Type}^u \rightsquigarrow \underline{A}'$ and $\Delta \vdash_\kappa A : \mathbf{Type}^u \rightsquigarrow A'$ describe the type translation for this compiler. The key point is that returner computation types in the source are translated into *total* returner types in the target according to the following rule:

$$\frac{\Delta \vdash_\kappa A : \mathbf{Type}^u \rightsquigarrow A'}{\Delta \vdash_\kappa \mathbf{F} A : \mathbf{Type}^u \rightsquigarrow \mathbf{F}_\circ A'}$$

The rest of the rules defining these judgements act homomorphically, following the syntax of types and translating each source type constructor to the analogous type constructor in the target. We leave them to the appendix. The computation translation $\Delta; \Gamma \vdash_{\mathrm{c}} M : \underline{A} \rightsquigarrow M'$ and value translation $\Delta; \Gamma \vdash_{\mathrm{v}} V : A \rightsquigarrow V'$ are similarly straightforward, so we omit them here as well. Note that the translation is defined over the entire source language, including unfocused terms.

To understand why, informally, we should expect equivalences to be preserved between these languages, consider a target language context M whose hole is of translation type. The fact that all computations of translation type are total means that M cannot pass any impure term to a compiled component in a way that might break the source language's abstractions. That is not to say M itself is pure, however. M may well perform a calculation involving its hole and, based on the result, decide to diverge or not.

In particular, we face the following challenges in demonstrating full abstraction:

(1) Our proof must make use of the modal type system in CBPV$_t^{\forall\bullet}$ in order to enforce the invariant that a component compiled from CBPV$_s$ is never passed a nonterminating computation.

$$\frac{\Delta; \Gamma; \cdot \Vdash_{\text{foc}} V : [A] \twoheadrightarrow V'}{\Delta; \Gamma; \cdot \Vdash_{\text{inv}} \mathbf{return}\, V : \mathbf{F}_\epsilon A \twoheadrightarrow \text{return}_\epsilon V'} \qquad \frac{\Delta \vdash_\kappa A : \kappa \twoheadrightarrow A'}{\Delta; \Gamma; \cdot \Vdash_{\text{inv}} \mathbf{diverge} : \mathbf{F}_\bullet A \twoheadrightarrow \mathbf{return\,none}}$$

$$\frac{\Delta; \Gamma; x : A \Vdash_{\text{inv}} N : T \twoheadrightarrow N'}{\Delta; \Gamma; [\mathbf{F}_\epsilon A] \Vdash_{\text{foc}} \mathbf{bind}\, x \leftarrow [\cdot].N : T \twoheadrightarrow \text{bind}_\epsilon x \leftarrow [\cdot].N'}$$

$$\begin{array}{rcl}
\text{return}_\circ V & = & \mathbf{return}\, V \\
\text{return}_\bullet V & = & \mathbf{return}\,(\mathbf{some}\, V) \\
\text{bind}_\circ x \leftarrow S.N & = & \mathbf{bind}\, x \leftarrow S.N \\
\text{bind}_\bullet x \leftarrow S.N & = & \mathbf{bind}\, y \leftarrow S.\mathbf{pm}\, y\, \mathbf{as}\, \{\mathbf{none}.\mathbf{return\,none}, \mathbf{some}\, x.N\}
\end{array}$$

Fig. 10. Selected back-translation rules and macro definitions.

(2) Even this observation however, does not mean that programs consisting of a compiled CBPV$_s$ component in a native CBPV$_t^{\forall\bullet}$ context must always terminate: a computation internal to the context may still diverge. We need to represent this behavior in the source language.
(3) Finally, a CBPV$_t^{\forall\bullet}$ context may internally use other data types such as polymorphic functions that have no equivalent in CBPV$_s$.

The literature has visited these challenges before, so we briefly describe some related work before showing how focusing is a natural addition to the picture.

5.3 Background

New, Bowman, and Ahmed [New et al. 2016] are able to prove full abstraction for a compiler from a language without exceptions but with recursive types to one with exceptions tracked by a modal type system. They find a *universal type* in the source language which is able to represent all of the dynamic behavior of the target language. They then write an interpreter in the source which is used to interpret back-translated, type-erased, target-language contexts. This is very similar to the case we are investigating, except that we seek to avoid interpreting all of the CBPV$_t^{\forall\bullet}$ language in CBPV$_s$, a strategy which would imply that to meet our eventual goal, an OCaml interpreter would need to be written in Coq.

Like us, Ahmed and Blume [Ahmed and Blume 2011] compile to a target language with polymorphic types from a source language without. In their case, both languages are pure. Since no quantified type can be passed into a source-language function, they show that any use of polymorphic types can be partially evaluated away during back-translation. Unfortunately, this makes the well-foundedness of the back-translation dependent upon counting steps of evaluation, and has not been shown to extend to languages with nontermination. Instead of partial evalutation during back-translation, we suggest focalization as a principled, type-directed means of normalization *before* back-translation.

5.4 Focusing and Full Abstraction

To establish equivalence preservation, we need to show that for any two CBPV$_s$ programs M_1 and M_2 and CBPV$_t^{\forall\bullet}$ context N' which can distinguish their compilations, we can find a CBPV$_s$ context which distinguishes M_1 and M_2. In other words, we must define a *back-translation* of target language contexts to source language contexts. This is where focusing comes in: thanks to completeness, we need only concern ourselves with translating focused target contexts. The interesting part of the back-translation is given in Figure 10. Its definition follows the structure of derivations of focused terms. However, since we ultimately only need to back-translate contexts whose holes expect a term compiled from CBPV$_s$, the back-translation need only be defined over terms whose types and variables' types are in the image of the type translation. For this reason, we never need to back-translate polymorphic functions or terms which instantiate polymorphic functions. This is important, since CBPV$_s$ is simply typed and there is no obvious back-translation for such terms.

Most back-translation rules simply translate type and term constructors in the target language to their analogs in the source. Of course this is not possible in the case of rules involving the type $\mathbf{F}_\bullet A$. Instead, we back-translate this type to $\mathbf{F}\,(\mathbf{Maybe}\ A')$ where A back-translates to A'. The idea is that diverging computations in the target will back-translate to returners that return **none** in the source. Computations of partial type that do return a value V will back-translate to computations that return **some** V' where V' is the back-translation of V. This is handled by the back-translation rules for **return** V and **diverge**. In order to define and prove properties of the back-translation of returner computations uniformly, we define the back-translation of **return** V and **bind** $x \leftarrow [\cdot].N$ in terms of macros $\mathrm{return}_\epsilon\, V'$ and $\mathrm{bind}_\epsilon\, x \leftarrow [\cdot].N'$. These macros represent the return and bind operations for the $\mathbf{F}\,(\mathbf{Maybe}\ A)$ monad and are defined in Figure 10.

Notice that our back-translation is a retraction of compilation. That is, if M' is the translation of M, then M is the back-translation of M'. This property, together with the *adequacy* and *compositionality* properties that follow, is the key to proving full abstraction.

Lemma 5.1 (Adequacy of back-translation). *If* $\cdot;\cdot \Vdash_c M : \mathbf{F}_\bullet\mathbf{Unit} \twoheadrightarrow M'$ *then either*

- $M = \mathbf{diverge}$ *and* $M' = \mathbf{return\ none}$ *or*
- $M = \mathbf{return\ unit}$ *and* $M' = \mathbf{return}\,(\mathbf{some\ unit})$.

Proof. By case analysis on the back-translation of M. □

Adequacy states that we back-translate complete programs correctly. That is, closed partial target programs of type $\mathbf{F}_\bullet\mathbf{Unit}$ back-translate to **return none** when they loop and **return (some unit)** when they terminate. This is easy to prove thanks to the fact that we restrict the back-translation to focused terms: there are only two computations of this type that are both closed and focused.

Lemma 5.2 (Compositionality of back-translation).
Assume Γ *and* T *are suspension-normal and* $\Omega \in \mathbf{ord}\,(\Gamma'')$.

(1) *If* $\Delta;\Gamma;\cdot \Vdash_{\mathrm{foc}} V : [A] \twoheadrightarrow V'$ *and* $\Delta;\Gamma,\Gamma';x : A,\Omega \Vdash_{\mathrm{inv}} N : T \twoheadrightarrow N'$*, then* $\Delta;\Gamma,\Gamma';\Omega \Vdash_{\mathrm{inv}} N[\![V/x]\!] : T \twoheadrightarrow N'[\![V'/x]\!]$.
(2) *If* $\Delta;\Gamma;\cdot \Vdash_{\mathrm{inv}} M : \underline{A} \twoheadrightarrow M'$ *and* $\Delta;\Gamma,\Gamma';[\underline{A}] \Vdash_{\mathrm{foc}} S : \underline{B} \twoheadrightarrow S'$*, then* $\Delta;\Gamma,\Gamma';\cdot \Vdash_{\mathrm{inv}} S[\![M]\!] : \underline{B} \twoheadrightarrow S'[\![M']\!]$.
(3) *If* $\Delta;\Gamma;\cdot \Vdash_{\mathrm{inv}} M : \underline{A} \twoheadrightarrow M'$ *and* $\Delta;\Gamma,\Gamma',x : \mathbf{U}\underline{A};\Omega \Vdash_{\mathrm{inv}} N : T \twoheadrightarrow N'$*, then* $\Delta;\Gamma,\Gamma';\Omega \Vdash_{\mathrm{inv}} N[\![M/\mathbf{force}\ x]\!] : T \twoheadrightarrow N'[\![M'/\mathbf{force}\ x]\!]$.
(4) *If* $\Delta;\Gamma;\cdot \Vdash_{\mathrm{inv}} M : \underline{A} \twoheadrightarrow M'$ *and* $\Delta;\Gamma,\Gamma',x : \mathbf{U}\underline{A};\cdot \Vdash_{\mathrm{foc}} V : [B] \twoheadrightarrow V'$*, then* $\Delta;\Gamma,\Gamma';\cdot \Vdash_{\mathrm{foc}} V[\![M/\mathbf{force}\ x]\!] : [B] \twoheadrightarrow V'[\![M'/\mathbf{force}\ x]\!]$.
(5) *If* $\Delta;\Gamma;\cdot \Vdash_{\mathrm{inv}} M : \underline{A} \twoheadrightarrow M'$ *and* $\Delta;\Gamma,\Gamma',x : \mathbf{U}\underline{A};[\underline{B}] \Vdash_{\mathrm{foc}} S : T \twoheadrightarrow S'$*, then* $\Delta;\Gamma,\Gamma';[\underline{B}] \Vdash_{\mathrm{foc}} S[\![M/\mathbf{force}\ x]\!] : T \twoheadrightarrow S'[\![M'/\mathbf{force}\ x]\!]$.
(6) *If* $\Delta;\Gamma;\Omega \Vdash_{\mathrm{inv}} M : \mathbf{F}_\epsilon A \twoheadrightarrow M'$ *and* $\Delta;\Gamma,\Gamma';x : A \Vdash_{\mathrm{inv}} N : T \twoheadrightarrow N'$ *where* **stable** T*, then* $\Delta;\Gamma,\Gamma';\Omega \Vdash_{\mathrm{inv}} N[\![x \leftarrow M]\!] : T \twoheadrightarrow N'[\![x \leftarrow M']\!]$.
(7) *If* $\Delta;\Gamma;[\underline{B}] \Vdash_{\mathrm{foc}} S : \mathbf{F}_\epsilon A \twoheadrightarrow S'$ *and* $\Delta;\Gamma,\Gamma';x : A \Vdash_{\mathrm{inv}} N : T \twoheadrightarrow N'$ *where* **stable** T*, then* $\Delta;\Gamma,\Gamma';[\underline{B}] \Vdash_{\mathrm{foc}} N[\![x \leftarrow S]\!] : T \twoheadrightarrow N'[\![x \leftarrow S']\!]$.

Compositionality states that our back-translation is substitutive. That is, back-translations of incomplete programs can be composed via focused substitution. This lemma and its proof take essentially the same form as the focused substitution lemma (Lemma 3.4) required for proving completeness.

With these properties established, we are almost ready to tackle full abstraction. At heart, full abstraction is simply a corollary to the existence of a back-translation that is adequate, compositional, and a retraction of the forward translation. However, since we have defined a back-translation only over focused terms, we need to put in some additional effort to both focus target language computations that need to be back-translated and show that this process does not disrupt our compiler correctness property. In doing so, we make use of the fact that the compiler pass preserves the equational theory and focusing derivations.

Lemma 5.3 (Equivalence preservation). *Suppose* $\Delta; \Gamma \vdash_c N_1 \approx_{ctx} N_2 : \underline{A}$ *and* $\Delta \vdash_\kappa \underline{A} : \kappa \rightsquigarrow \underline{A}'$ *and* $\Delta \vdash_\Gamma \Gamma \rightsquigarrow \Gamma'$ *and* $\Delta; \Gamma \vdash_c N_1 : \underline{A} \rightsquigarrow N_1'$ *and* $\Delta; \Gamma \vdash_c N_2 : \underline{A} \rightsquigarrow N_2'$.
Then $\Delta; \Gamma' \vdash_c N_1' \approx_{ctx} N_2' : \underline{A}'$

Proof. We begin by obtaining N_{f_1} and N_{f_2}, the focused forms of N_1 and N_2 respectively. We then translate these to N_{f_1}' and N_{f_2}'. The compiler preserves the equational theory, so we know that N_{f_1}' is contextually equivalent to N_1'. Likewise for N_{f_2}' and N_2'.

It now suffices to show that N_{f_1}' is contextually equivalent to N_{f_2}'. To do so, we consider an arbitrary focused context M_f and focused substitution γ and will show that

$$M_f'[\![\gamma'(\!|N_{f_1}'|\!)/\textbf{force } x]\!] = M_f'[\![\gamma'(\!|N_{f_2}'|\!)/\textbf{force } x]\!]$$

We will need to use the fact that N_{f_1} and N_{f_2} are contextually equivalent in the source language. To obtain a source-level context, we back-translate M_f' to M_f and γ' to γ. We obtain the source language equality

$$M_f[\![\gamma(\!|N_{f_1}|\!)/\textbf{force } x]\!] = M_f[\![\gamma(\!|N_{f_2}|\!)/\textbf{force } x]\!]$$

For this to be useful, we need to somehow relate it to the terms in the target-level equality that we are trying to prove. Fortunately, our adequacy lemma bridges the gap by reducing our burden to simply showing that the computation on each side of the target language equality back-translates to the term on the same side of this source language equality.

Note that the back-translation is a retraction of the translation. Therefore, N_{f_1}' back-translates to N_{f_1} and N_{f_2}' back-translates to N_{f_2}. All that remains is to apply our compositionality lemma to determine that

- $M_f'[\![\gamma'(\!|N_{f_1}'|\!)/\textbf{force } x]\!]$ back-translates to $M_f[\![\gamma(\!|N_{f_1}|\!)/\textbf{force } x]\!]$ and
- $M_f'[\![\gamma'(\!|N_{f_2}'|\!)/\textbf{force } x]\!]$ back-translates to $M_f[\![\gamma(\!|N_{f_2}|\!)/\textbf{force } x]\!]$.

□

Lemma 5.4 (Equivalence reflection). *Suppose* $\Delta; \Gamma \vdash_c N_1' \approx_{ctx} N_2' : \underline{A}$ *and* $\Delta \vdash_\kappa \underline{A} : \kappa \rightsquigarrow \underline{A}'$ *and* $\Delta \vdash_\Gamma \Gamma \rightsquigarrow \Gamma'$ *and* $\Delta; \Gamma \vdash_c N_1 : \underline{A} \rightsquigarrow N_1'$ *and* $\Delta; \Gamma \vdash_c N_2 : \underline{A} \rightsquigarrow N_2'$.
Then $\Delta; \Gamma' \vdash_c N_1 \approx_{ctx} N_2 : \underline{A}'$.

Proof. Similar to the preceding proof. See the appendix for details. □

Equivalence preservation and reflection imply full abstraction.

Theorem 5.5 (Full abstraction). *Suppose* $\Delta \vdash_\kappa \underline{A} : \kappa \rightsquigarrow \underline{A}'$ *and* $\Delta \vdash_\Gamma \Gamma \rightsquigarrow \Gamma'$ *and* $\Delta; \Gamma \vdash_c N_1 : \underline{A} \rightsquigarrow N_1'$ *and* $\Delta; \Gamma \vdash_c N_2 : \underline{A} \rightsquigarrow N_2'$.
Then $\Delta; \Gamma \vdash_c N_1 \approx_{ctx} N_2 : \underline{A}$ *iff* $\Delta; \Gamma' \vdash_c N_1' \approx_{ctx} N_2' : \underline{A}'$.

At this point we have shown that the embedding of CBPV$_s$ into CBPV$_t^{\forall\bullet}$ is fully abstract. Using focusing, we were able to take advantage of the fact that the type translation ensures CBPV$_t^{\forall\bullet}$ contexts of translation type do not have any more discriminating power than CBPV$_s$ contexts, even though CBPV$_t^{\forall\bullet}$ is impure. Furthermore, focusing lets us avoid the need to back-translate polymorphic functions despite their presence in CBPV$_t^{\forall\bullet}$. It also simplified the proof of full abstraction: for example, the proof lemma 5.1 is a short and straightforward case analysis thanks to the fact that there are only two closed and focused programs of type $\mathbf{F}_{\bullet}\mathbf{Unit}$ in $\mathrm{CBPV}_t^{\forall\bullet}$.

6 CONCLUSION

6.1 Related Work

Focusing and Normalization. Focusing has a long history, originating with Andreoli's [1992] study of it in the context of linear logic. More recent work has studied its connection to programming languages. Zeilberger [2008a; 2008b] uses focused, polarized logic as a guide to develop a language in which, as in CBPV, evaluation order is reflected in the type system. Brock-Nannestad and Schürmann [2010] describe focused natural deduction systems, which Espírito Santo [2016] examines carefully in order to devise an elegant syntax and study focused CBPV. Krishnaswami [2009] investigates pattern matching in a language based on the focused sequent calculus. Abel and Sattler [2019] study normalization by evaluation for CBPV and obtain normal forms close to ours.

However, the above-mentioned work does not explore focusing's relationship to reasoning about programs and contextual equivalence. None prove a *computational* completeness property like theorem 3.5, which is the springboard for our investigation.

Scherer [2017] describes an algorithm for deciding equivalence in the simply typed lambda calculus (STLC) with sums and an empty type. This approach uses a relatively complex (compared to ours) normalization procedure to reduce two terms of interest, obtaining canonical forms that are syntactically identical if and only if the terms of interest are contextually equivalent. The normalization process relies upon both the purity of the language under consideration and its simple type system.

On the other hand, we do not desire an algorithm for deciding equivalence but only a useful proof technique for demonstrating it. To this end, we are willing to accept that our normal forms are not canonical in exchange for a simpler and more widely applicable normalization procedure. This in turn yields a more straightforward description of our normal forms. We obtain results despite relatively lax normal forms because rather than normalizing only the terms we wish to prove equivalent, we also normalize the contexts that they may appear in.

Fully Abstract Compilation. Research in secure compilation has gained steam in recent years. Full abstraction is a particularly desirable property because of its strength and composability. Proofs of it have a long history; we refer the reader to the work of Patrignani, Ahmed, and Clarke [Patrignani et al. 2019] for a detailed discussion.

The most closely related line of work to our full abstraction proof is that of Ahmed and her collaborators. Ahmed and Blume proved the full abstraction of closure conversion [Ahmed and Blume 2008] and CPS translation [Ahmed and Blume 2011]. The latter work combines the source and target languages into a single "multi-language" [Matthews and Findler 2007] with boundaries that mediate between source and target terms. Their target language is more expressive than their source language in that it supports System F-style polymorphism. They recognize that the back-translation is needed only for terms of translation type. Thus, features that appear in the target language but not the source must exist only in intermediate terms that can be partially evaluated away, making back-translation possible. The inductive definition of the back-translation is quite complicated, however, and involves counting steps of remaining execution. This approach does not seem to extend to languages with divergence.

Our technique of defining the back-translation only over focused terms is similar in spirit to back-translation by partial evaluation. Focused terms are fully β reduced, so our back-translation definition does not need to handle intermediate terms in the target. We believe this positions focusing as a promising alternative to partial evaluation in defining back-translations.

New, Bowman, and Ahmed [New et al. 2016] prove full abstraction by essentially building an interpreter for the target language in the source language. Their target language has a modal type system which tracks exceptions while exceptions are not supported in the source language. This situation is very similar to our own; however, we are interested in compiling from potentially less expressive *total* source languages to more expressive target languages that admit divergence. In such a scenario, we cannot hope to interpret the target language from the source.

Context Lemmas. We do not need to examine two programs' behaviors in every context to prove them equivalent: only a more manageable subset of contexts will suffice. This is the basic observation made by context lemmas. The ciu theorem is one such lemma which terms M and N as equivalent when M and N evaluate to identical values after all free variables have been closed off with well-typed values and the computation is placed inside of an evaluation context that produces a base type.

We can take advantage of the fact that the ciu theorem allows reasoning to take place inside of a context in which M and N are being evaluated to prove program equivalences by reasoning about how they evaluate. The ciu theorem is less helpful, though, when it comes to reasoning about how M and N interact with their environment. In a call-by-name setting, the type of M and N allow us to determine the shape of the context they run in. However, in a call-by-value setting there are many more possible evaluation contexts, which makes the ciu theorem less useful. Dually, in call-by-value, free variables are closed off with closed values. The types of those values make it possible to significantly narrow down the outermost term constructors used to build them. In call-by-name, though, variables are closed with expressions, which we know less about. This means that in call-by-value, the ciu theorem is lacking when it comes to reasoning about the continuation a program's output is passed to, while in call-by-name, the ciu theorem does not help with reasoning about how a program interacts with its inputs. With focusing, we both narrow the set of contexts and narrow the set of closing values.

On the other hand, the ciu theorem narrows the contexts we need to consider to prove computations equivalent to those that evaluate a computation exactly once. Using focused terms with a free variable of thunk type, we must consider situations in which the thunks are forced multiple times. Thus there is no direct comparison between the contexts used for ciu equivalence and the contexts we describe as focused. Instead, the two techniques may be seen as complementary.

6.2 Future Work

Focusing has traditionally been applied in the context of logics [Andreoli 1992] and pure programming languages [Scherer 2017; Scherer and Rémy 2015]. Thus, much work remains to be done in applying computation focusing to languages with more complex types and effects.

Impredicativity. We would expect the focusing rules for a System F-like language to look essentially unchanged from this paper. However, proving completeness of focusing is more challenging. We make essential use of an induction principle that relies on predicativity in proving lemmas 3.4 and 5.2. It looks unlikely that syntactic completeness proofs like ours will scale to impredicative languages, but a more semantic logical-relation style approach should succeed here. Work in normalization by evaluation for impredicative languages [Abel 2008] exemplifies a step in this direction.

Linear Types. As focusing was developed in the setting of linear logic, there is no fundamental challenge in supporting linear types. In fact, linear languages have very useful normal forms. Consequently, we believe that focusing is a promising approach for proving free theorems about linear types, especially in the presence of effects. Prior work on establishing relational parametricity for linear lambda calculus using "open" logical relations [Zhao et al. 2010] is effective in pure languages, but does not take advantage of the *intensional* properties of linear functions that are needed to fully reason about the interaction between linear types and effects. Perhaps surprisingly, logical relations seem ill-suited for the task and focusing may be a better match for proving such results.

Recursion. The definition of the set of focused terms is not hard to adjust to accommodate the addition of a term-level fixed point operator to $\mathrm{CBPV}_t^{\forall\bullet}$. The price to pay is that our focused terms are less canonical: they may contain unfoldable uses of the fixed point operator. For the purposes of proving equivalence, however, we should not need to consider contexts that make arbitrary use of the fixed point operator since the execution of a terminating program unfolds fixed points a finite number of times. Similarly, proving our full abstraction example should only require back-translation of finite unfoldings of fixed points. To incorporate this idea, we would need a syntactic fixed point induction principle [Debakker and Scott 1969; Pitts 2004]. Pitts's proof of this property is based on long and tedious inductions on the operational semantics, but we believe focusing may simplify the argument.

Zeilberger [2008a; 2008b] presents a focused language in which pattern matching forms are represented not by traditional syntax, but by meta-level maps from patterns to expressions. In this setting, recursive types are a simple addition because these maps are capable of matching against an infinite number of patterns. When trying to define a focused subset of a language with finitary syntax, however, it seems one could only hope to describe finite approximations of focused terms even for simple recursive types like natural numbers. Moreover, our induction principle for the substitution lemma will not suffice as it relies upon induction on the structure of types.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and the Penn PL Club for their feedback and suggestions for improvements to earlier versions of this paper. The first author would additionally like to thank Max New, Gabriel Scherer, and Amal Ahmed for many insightful conversations on call-by-push-value, focusing, and proofs of program equivalences.

This work was funded by the National Science Foundation's Expedition in Computing *The Science of Deep Specification* award 1521539 with additional support by the ONR grant *REVOLVER* award N00014-17-1-2930. The first author is supported by a National Science Foundation Graduate Research Fellowship under Grant No. 1845298. 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 National Science Foundation or the ONR.

REFERENCES

Andreas Abel. 2008. Weak $\beta\eta$-Normalization and Normalization by Evaluation for System F. In *15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2008, 22-27 November 2008, Doha, Qatar, Proceedings (Lecture Notes in Artificial Intelligence)*, Illiano Cervesato, Helmut Veith, and Andrei Voronkov (Eds.), Vol. 5330. Springer-Verlag, 497–511.

Andreas Abel and Christian Sattler. 2019. Normalization by Evaluation for Call-By-Push-Value and Polarized Lambda Calculus. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages 2019 (PPDP '19)*. Association for Computing Machinery, New York, NY, USA, Article 3, 12 pages.

Samson Abramsky. 1990. The Lazy λ- Calculus. In *Research topics in functional programming*. Addison Wesley, 65–116.

Samson Abramsky and C-H Luke Ong. 1993. Full abstraction in the lazy lambda calculus. *Information and Computation* 105, 2 (1993), 159–267.

Amal Ahmed and Matthias Blume. 2008. Typed Closure Conversion Preserves Observational Equivalence. *SIGPLAN Not.* 43, 9 (2008), 157–168.

Amal Ahmed and Matthias Blume. 2011. An Equivalence-preserving CPS Translation via Multi-language Semantics. *SIGPLAN Not.* 46, 9 (2011), 431–444.

Jean-Marc Andreoli. 1992. Logic programming with focusing proofs in linear logic. *Journal of logic and computation* 2, 3 (1992), 297–347.

Taus Brock-Nannestad and Carsten Schürmann. 2010. Focused Natural Deduction. In *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10).* Springer-Verlag, Berlin, Heidelberg, 157–171.

Kaustuv Chaudhuri, Frank Pfenning, and Greg Price. 2018. A logical characterization of forward and backward chaining in the inverse method. *Journal of Automated Reasoning* 40, 2-3 (2018), 133–177.

J. W. Debakker and Dana S. Scott. 1969. A theory of programs. (1969).

Harley D Eades III and Aaron Stump. 2010. Hereditary Substitution for Stratified System F. *International Workshop on Proof-Search in Type Theories* 10 (2010).

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-Directed Synthesis: A Type-Theoretic Interpretation. *SIGPLAN Not.* 51, 1 (Jan. 2016), 802–815.

Neelakantan R. Krishnaswami. 2009. Focusing on Pattern Matching. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09).* Association for Computing Machinery, New York, NY, USA, 366–378.

Søren B Lassen. 1999. Bisimulation in Untyped Lambda Calculus: Böhm Trees and Bisimulation up to Context. *Electronic Notes in Theoretical Computer Science* 20 (1999), 346–374.

Søren B. Lassen. 2005. Eager Normal Form Bisimulation. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05).* 345–354.

Daniel Leivant. 1991. Finitely Stratified Polymorphism. *Inf. Comput.* 93, 1 (July 1991), 93–113.

Paul Blain Levy. 1999. Call-by-Push-Value: A Subsuming Paradigm. In *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings (Lecture Notes in Computer Science)*, Jean-Yves Girard (Ed.), Vol. 1581. Springer, 228–242.

Ian Mason and Carolyn Talcott. 1991. Equivalence in functional languages with effects. *Journal of functional programming* 1, 3 (1991), 287–327.

Jacob Matthews and Robert Bruce Findler. 2007. Operational Semantics for Multi-language Programs. *SIGPLAN Not.* 42, 1 (2007), 3–10.

Robin Milner. 1977. Fully abstract models of typed λ-calculi. *Theoretical Computer Science* 4, 1 (1977), 1 – 22.

J. H. Morris. 1968. *Lambda Calculus Models of Programming Languages.* Ph.D. Dissertation. Massachusets Institute of Technology.

Max S. New, William J. Bowman, and Amal Ahmed. 2016. Fully Abstract Compilation via Universal Embedding. *SIGPLAN Not.* 51, 9 (Sept. 2016), 103–116.

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. *SIGPLAN Not.* 50, 6 (June 2015), 619–630.

Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work. *ACM Comput. Surv.* 51, 6, Article 125 (2019), 36 pages.

Frank Pfenning. 1995. Structural Cut Elimination. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS '95).* IEEE Computer Society, USA, 156.

Andrew Pitts. 2004. Typed Operational Reasoning. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). The MIT Press, Chapter 7.

Gordon D Plotkin. 1980. Lambda-definability in the full type hierarchy. *To HB Curry: essays on combinatory logic, lambda calculus and formalism* (1980), 363–373.

Nadia Polikarpova and Ilya Sergey. 2019. Structuring the Synthesis of Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 72 (Jan. 2019), 30 pages.

Nick Rioux and Steve Zdancewic. 2020. *Computation Focusing (Technical Report).* Technical Report MS-CIS-20-04. Department of Computer and Information Science, University of Pennsylvania.

Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. 2007. Environmental bisimulations for higher-order languages. In *Logic in Computer Science, 2007. LICS 2007. 22nd Annual IEEE Symposium on.* IEEE, 293–302.

José Espírito Santo. 2016. The Polarized λ-calculus. *Electr. Notes Theor. Comput. Sci.* 332 (2016), 149–168.

Gabriel Scherer. 2017. Deciding Equivalence with Sums and the Empty Type. *SIGPLAN Not.* 52, 1 (2017), 374–386.

Gabriel Scherer and Didier Rémy. 2015. Which Simple Types Have a Unique Inhabitant?. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015).* Association for Computing Machinery, New York, NY, USA, 243–255.

Robert J Simmons. 2014. Structural focalization. *ACM Transactions on Computational Logic (TOCL)* 15, 3 (2014), 21.

Richard Statman. 1985. Logical relations and the typed λ-calculus. *Information and Control* 65, 2-3 (1985), 85–97.

Eijiro Sumii and Benjamin C. Pierce. 2005. A Bisimulation for Type Abstraction and Recursion. *SIGPLAN Not.* 40, 1 (2005), 63–74.

William W. Tait. 1967. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic* 32, 2 (1967), 198–212.

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the 4th International Symposium on Functional Programming and Computer Architecture.*

Noam Zeilberger. 2008a. Focusing and Higher-order Abstract Syntax. *SIGPLAN Not.* 43, 1 (2008), 359–369.

Noam Zeilberger. 2008b. On the unity of duality. *Annals of Pure and Applied Logic* 153, 1 (2008), 66 – 96.

Jianzhou Zhao, Qi Zhang, and Steve Zdancewic. 2010. Relational Parametricity for Polymorphic Linear Lambda Calculus. In *Proceedings of the Eighth ASIAN Symposium on Programming Languages and Systems (APLAS).*