Triangulating Context Lemmas

Citation for published version:

Digital Object Identifier (DOI):
10.1145/3167081

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Abstract
The idea of a context lemma spans a range of programming-language models: from Milner’s original through the CIU theorem to ‘CIU-like’ results for multiple language features. Each shows that to prove observational equivalence between program terms it is enough to test only some restricted class of contexts: applicative, evaluation, reduction, etc.

We formally reconstruct a distinctive proof method for context lemmas based on cyclic inclusion of three program approximations: by triangulating between ‘applicative’ and ‘logical’ relations we prove that both match the observational notion, while being simpler to compute. Moreover, the observational component of the triangle condenses a series of approximations covering variation in the literature around what variable-capturing structure qualifies as a ‘context’.

Although entirely concrete, our approach involves no term dissection or inspection of reduction sequences; instead we draw on previous context lemmas using operational logical relations and biorthogonality. We demonstrate the method for a fine-grained call-by-value presentation of the simply-typed lambda-calculus, and extend to a CIU result formulated with frame stacks.

All this is formalised and proved in Agda: building on work of Allais et al., we exploit dependent types to specify lambda-calculus terms as well-typed and well-scoped by construction. By doing so, we seek to dispel any lingering anxieties about the manipulation of concrete contexts when reasoning about bound variables, capturing substitution, and observational equivalences.

Keywords  Context lemma, CIU theorem, Agda, Dependent types, Observational equivalence, Logical relations

1 Introduction
The problem of determining equivalence between programs or program fragments (terms) is well-studied and a variety of approaches have been developed [11–13, 15, 16, 19, 27–30, 32]. Typically, the equivalence of interest is contextual or observational equivalence. Roughly speaking, this relates two terms if they behave equivalently in any program context. The motivation for working with observational equivalence is that it supports powerful equational reasoning about code, such as rewriting via chains of equivalence proofs and the substitution of equivalent sub-terms (‘equals for equals’) in larger programs.

The exact meaning of ‘program context’ and ‘behave equivalently’ may vary: for example, program contexts often involve some kind of free-variable capture; while for equivalence it may be enough that two programs both terminate, or that they return equal values at ground type. The quantification over program contexts from the language itself turns out to make observational equivalence robust — independent of the class of available observations — and self-regulating, as a language with more elaborate features has a correspondingly larger collection of testing contexts.

Sadly, this quantification over all possible enclosing contexts can also make observational equivalence difficult to prove directly. However, if it can be captured by some restricted form of contexts then the proof burden is reduced without sacrificing reasoning power. Such correspondence results are known as context lemmas, following Milner’s pioneering article [20], where he proved a context lemma for a typed combinatory logic with first-order function symbols.

Since then there have been many similar results published for a variety of calculi. The particular approach used to obtain the result varies, with many based on Howe’s coinductive ‘precongruence candidate’ method [12, 13]. In contrast, following earlier proofs for the nu-calculus [32] and ReFS [30], we isolate a key part of their argument for independent study: proving a context lemma by showing that a certain triangle of relations coincide. We propose this triangulation proof method as a general technique for proving context lemmas.

The triangle links three kinds of relation between terms:

Observational where terms are related by their behaviour in some class of contexts;

Applicative where function terms are related if they take equal arguments to related results;

Logical where function terms are related if they take related arguments to related results.

Rather than equivalences, we in fact work with inclusions between their associated approximation preorders; all results immediately carry over to equivalences.
While observational relations are defined by quantification over contexts, both applicative and logical relations are given by type structure. This structural character of these lower relations shapes the proofs along all three sides — in particular the implication from logical to observational arises from the fundamental theorem of logical relations.

The base of the triangle also shares out the distinctive properties of observational relations for powerful reasoning: applicative approximation is easily shown to be transitive, and logical approximation a congruence.

Observational relations require some chosen class of contexts, and in fact we see the triangle apex ramify into a chain of relations from behaviour in all contexts down through more restricted collections, with each relation naturally implying the next as the set of testing contexts becomes smaller. However, the triangulation result brackets all these observational relations between logical on the left and applicative on the right, proving that they collapse into each other. This collapse — that testing in all contexts is the same as testing over some restricted class — is the context lemma.

In this paper, we formally construct this triangulation proof method in a fine-grained call-by-value setting using the Agda interactive theorem prover. Leveraging state-of-the-art technology for dealing with well-typed and well-scoped terms [2], we show that, contrary to popularly held belief [15, 27], handling concrete contexts even in the formalised setting is straightforward, and moreover our method does not require inspection of reduction sequences [19, 32]. First, we apply the method to show the classical result for our simply-typed lambda calculus, namely that applicative contexts suffice (Section 3), using a big-step semantics. While we are not the first to undertake such a formalisation effort [3, 21], we believe our concrete, first-order representation to be more robust than previous attempts. To demonstrate the extensibility of the formalism, we generalise the triangle to the more powerful notion of CIU approximation (Section 4) by using frame stacks. Our intention is to extend our approach to more elaborate calculi involving effects, where such fine-grained stack-based approaches seem a better fit than big-step semantics. Demonstrating extensibility of our approach is moreover in the spirit of other proposals for mechanised metatheory developments [5]. In particular, we aim in future to scale the development to support core calculi and logical approximation a congruence.

2 A Fine-Grained Call-By-Value Calculus

The work carried out here is performed with respect to a call-by-value calculus, $\lambda_{FG}^\rightarrow$, inspired by fine-grained call-by-value [17] (FGCBV) and a normal form for terms used by Pitts [29]. Figure 1 gives an informal context-free grammar for the calculus, with named $\lambda$ and let bindings. The distinguishing features are the separation between the phrase classes for values and terms (we systematically elide the explicit lifting of values into terms) and the restriction to one term constructor, let, to express sequencing. For convenience, we introduce the syntactic sugar $MV$ for application of a term $M$ to a value $V$, used in stating certain properties for $\lambda_{FG}^\rightarrow$. We consider a simple type discipline, where $r$ ranges over ground types ($r \in \{bool, int, unit\}$). For values, $x$ ranges over an infinite set of variables, $b$ ranges over $\{tt, ff\}$, and $n$ over $\mathbb{Z}$. In the definitions and proofs that follow, $=$ and $\equiv$ denote equalities that hold definitionally, and $\equiv$ denotes equalities which hold propositionally.

2.1 Formalising $\lambda_{FG}^\rightarrow$

Henceforth, reflecting our Agda [25] formalisation, we give $\lambda_{FG}^\rightarrow$ a de Bruijn presentation in a dependently-typed metalanguage. In particular, we emphasise the following view on our representation and proofs about it: namely that it is a formalisation in a dependently typed metalanguage, and therefore implementable in any implementation of such a type theory. For this paper, we used the Agda theorem prover to check our constructions, but any system such as Coq [34] or Idris [7] would do just as well, modulo pragmatics. Part of our purpose in insisting on this point is to encourage readers to develop similar representations in their own tool of choice. In fact, at the cost of losing meta-language typing guarantees, our development could be reprised in any other implemented theory supporting sufficiently rich inductive definitions, such as those of Isabelle/HOL [24]. However, our conviction is that a dependently-typed meta-language is a precision tool, acutely well-adapted to the demands of typed object-language representation. For the sake of exposition, we focus on the mathematics, rather than its rendering in Agda concrete syntax; readers interested in the latter should consult our source code. For their convenience, we provide a Rosetta Stone in Table 1 at the end of the paper, mediating between informal concepts, LaTeX rendering, and their Agda formalisation.

We follow the discipline advocated by Allais et al. [2], henceforth referred to as the ACMM framework, which allows us to represent the informal syntax of Figure 1 in terms of inductive families of types [10]. For each syntactic category of the object language — variables, values, terms — we
define such a family indexed by object-language type A and context Γ. Thus we express \( \lambda_{FG}^{-1} \) directly via its typing rules, as in Figure 2, so that values and terms are by construction well-typed and well-scoped. As a consequence, we sometimes need to introduce a renaming, in particular the special case of weakening, to ensure the type-and-scoping correctness of terms. For example, in the syntactic sugar for application, the argument is applied in an environment extended with the binding for the function:

\[
\frac{
\Gamma \vdash M : A \rightarrow B 
\quad \Gamma \vdash V : A 
\quad \Gamma \vdash U : A 
}{\Gamma \vdash V \ U : B}
\]

\[
\frac{
\Gamma \vdash M : A 
\quad \Gamma, A \vdash N : B 
}{\Gamma \vdash \text{let}_{A} \ M \ N : B}
\]

where weak takes Γ-terms to Γ,A-terms for any type A by shifting de Bruijn indices. From now on, for readability, we elide any explicit weakening in terms. The principal ACMM features we exploit are as follows.

- ACMM uses first-order abstract syntax (FOAS) in its term representation (rather than weak higher-order abstract syntax (HOAS), as used, for example, by Despeyroux et al. in [9]), specifically a de Bruijn index [8] variable representation: so \( \alpha \)-convertibility is syntactic equality; variables \( \text{var}_k \) are declared in terms of offsets \( k \) (generated by zero and successor) into an environment \( \Gamma \) consisting of a list of types; function values \( \lambda_{M} \) comprise a type label \( A \) and a well-scoped term body \( M \); and similarly for let-expressions \( \text{let}_{A} \ M \ N \).

In each case, we suppress object-language type labels where they are unambiguous in context.

- Whereas, for operations and proofs defined over terms, ACMM does use (weak) HOAS: thus renaming, capture-avoiding substitution, etc. are all instances of a generic Kripke-style traversal over the phrase classes of the language. Such traversals are specified by supplying type families \( V, M V, M T \) to interpret respectively each of variables, values, and terms, subject to suitable closure conditions. That for the \( \lambda \) constructor is (somewhat simplified) as follows:

\[
\frac{\Box (V \ A \Delta \rightarrow M^T \ B \Delta)}{M V (A \Rightarrow B) \Gamma}
\]

where the modality \( \Box \) quantifies over all context extensions \( \Gamma \subseteq \Delta \) by renamings. A particular practical consequence is that, by design, only value substitutions are expressible in the formalism — which in fact accords with usual informal practice when presenting fine-grained call-by-value.

Our choice of ACMM can be seen as contributing to a now decades-old debate on object-language representations, particularly with respect to binding structures. There is not space to do justice to all the issues here (for an interesting discussion, see for example [31]), nor is it the main focus of the paper. Nevertheless, some comments may be in order.

- Adequacy for a FOAS representation, by contrast with varieties of HOAS, is straightforward — even trivial — as part of grounds for belief in the formalisation. We view this as particularly effective in the case of formalising object-language contexts (in Section 2.2) in defining varieties of observational approximation, where the informal treatment of variables and binding is a potential minefield.

- There is no doubting the convenience of HOAS in terms of definability of operations such as substitution; nor, as the ACMM authors point out, in terms of flexibility of 'logical relations' arguments about the behaviour of such operations: we make heavy use of such results. Arguably, given our use of 'logical' definitions of relations such as in Section 3.2, we could perhaps have done more to leverage this aspect of ACMM. We leave this to future work.

**Definition 2.1** (Simultaneous Substitution). For typing environments \( \Gamma \) and \( \Delta \) let \( \theta : \Gamma \vdash \Delta \) denote a simultaneous substitution \( \theta \) of variables in environment \( \Gamma \) by \( \lambda_{FG}^{-1} \) values in environment \( \Delta \). Informally, \( \theta \) takes Γ-terms to Δ-terms: given \( \Gamma \vdash M : A \) we may apply the substitution to \( M \) and obtain \( \Delta \vdash \theta(M) : A \). In the special case of \( \theta : \Gamma, A \vdash \Gamma \) given
at \( \text{var}_0 \) by value \( \Gamma \vdash V : A \) and everywhere else the identity we write \( \theta(M) \) as \( M[V] \) in the usual way. For the special case of \( \theta : \Gamma \vdash \cdot \) we say that \( \theta \) is a \( \Gamma \)-closing substitution. If \( \theta : \Gamma \vdash \Delta \) and \( \theta' : \Delta \vdash \Theta \) then \( \theta' \circ \theta : \Gamma \vdash \Theta \) denotes the composition of the simultaneous substitutions.

**Definition 2.2.** Given \( \theta : \Gamma \vdash \Delta \), define \( \theta_0 \) to be the value \( \Delta \vdash V : A \) specified for the variable \( \Gamma \vdash \text{var}_k : A \). Then, \( \theta_0 \vdash \text{var}_k \) denotes the value specified for \( \text{var}_k \). For \( \theta : \Gamma \vdash \Delta \), define the simultaneous substitution \( \text{succ} \theta : \Gamma \vdash \Delta \) by its behaviour on variables \( \Gamma \vdash \text{var}_k : A \), for all indices \( k \):

\[
(\text{succ} \theta)_k = \theta(\text{succ}_k).
\]

Moreover, any \( \theta : \Gamma, \Delta \vdash \Delta \) is extensionally equal to that of the substitution obtained by extending \( \text{succ} \) by \( \theta_0 \).

We define a small-step operational semantics for \( \lambda^*_{FG} \) via a primitive reduction on closed terms, in Figure 3, which is then contained within the general reductions of Figure 4. The separation of these two relations is inspired by Pitts [29] and is used when we define frame stack evaluation in Section 4. Both relations are type-correct by construction; the proof of type preservation is immediate.

Figure 5 presents \( \lambda^*_{FG} \) evaluation semantics as a type-indexed relation between closed terms and values. We omit typing annotations where unambiguous. Evaluation is closed under the reduction relations: proof is straightforward by induction over derivations.

**Lemma 2.3 (Reduction Respects Evaluation).**

If \( M \rightsquigarrow M' \) or \( M \Rightarrow M' \) then \( M \Downarrow V \) if and only if \( M' \Downarrow V \).

### 2.2 Formalising Contexts

As a step towards formalising observational approximation, we treat contexts as an extension of \( \lambda^*_{FG} \) syntax; and thanks to ACMM the grammar of these concrete contexts is directly expressible as an inductive definition. A term context \( C \) is a possibly-open term of arbitrary type containing zero or more occurrences of a well-typed and well-scoped hole. Analogously, a value context \( V \) is a possibly-open value of arbitrary type containing zero or more occurrences of such a hole. Naturally enough our formalisation enforces well-typed and well-scoped contexts, by construction, in the same way it does for terms and values.

The usual notion of a variable-capturing context (VCC) allows a context to capture variables occurring free in the term that fills the hole. In Figure 6 we define VCCs for \( \lambda^*_{FG} \), with \( C, D, E \) ranging over term contexts and \( V, W \) over value contexts. The form of a well-typed and well-scoped context \( C \) is \( \Delta \vdash C : \langle\langle \Gamma \vdash A \rangle\rangle^{\langle\langle V \rangle\rangle}B \), which says that term context \( C \) has type \( B \) in environment \( \Delta \) and contains zero or more occurrences of a hole of type \( A \) in an environment \( \Gamma \). For such a well-typed term context \( C \), there is an operation, instantiation, denoted by \( C\langle\langle M \rangle\rangle^{\langle\langle V \rangle\rangle} \) which fills the holes in the term context \( C \) with \( M \). We make a customary abuse...
of notation in overloading chevrons ‘⟨⟨⟩⟩’ to represent both the occurrence of a hole in a context and the operation of instantiation. Instantiation is (yet) another structural traversal, where the only (base) case of interest is that of the hole constructor, replacing the hole with $M$:

$$\langle\langle-\rangle\rangle\langle\langle M\rangle\rangle^{\text{VCC}} = M.$$  

By virtue of our de Bruijn encoding, we avoid traditional concerns (as in [15, 27, for example]) associated with $\alpha$-equivalence of VCCs. However, VCCs are still a little inconvenient to work with because they force holes to match exactly the enclosing scope of their context, even up to the ordering of the variables captured (see the vccHOLE rule in Figure 6).

To manage such concerns we employ a different notion of value-substituting contexts (VSCs), where each occurrence of a hole carries an appropriate (well-typed) substitution, as in rule vscHOLE below. This ensures that the instantiation operation $\mathcal{C}\langle\langle M\rangle\rangle^{\text{VSC}}$ is itself well-typed and well-sorted, and again definable by a simple traversal. Typing judgements for VSCs are the same as for VCCs except we replace the vccHOLE rule with the following.

$$\frac{}{\Delta \vdash \theta : \Gamma \not\vdash \Delta} \quad \frac{}{\Delta \vdash \langle\langle \theta-\rangle\rangle : \langle\langle \Gamma + A\rangle\rangle^{\text{VSC}}A}$$

Instantiation for VSCs is as with VCCs except for the base case, where a hole is filled by applying the substitution provided.

$$\langle\langle \theta-\rangle\rangle\langle\langle M\rangle\rangle^{\text{VSC}} = \theta(M)$$

Every VCC is trivially a VSC, by annotating each hole with an identity substitution. For the rest of the paper we use unadorned chevrons for instantiation when it is clear from the context which version of contexts is being used.

Lassen defines a notion of variable-capturing context [16] that is slightly different to ours, equivalent to each hole carrying a renaming rather than a general value substitution. This lies strictly between our VCCs and VSCs, as any renaming is naturally a substitution. Later, we shall see that relations based on either VCCs or VSCs coincide, and so also with ones based on Lassen’s variant.

### 2.3 Observational Approximation

This section formally defines the notion of observational approximation, alluded to in the introduction, for $\lambda^{FG}_{\Gamma C}$. Its definition can be traced back to Morris [23] and builds on the account of term contexts and substitutions in the previous section.

We begin with a relation transformer lifting any relation on values to a relation on terms.

**Definition 2.4.** For closed terms $M, N$ of type $A$ and $\mathcal{R}$ a binary relation on closed values of type $A$,

$$M [\mathcal{R}] \quad N \not\vdash V. M \downarrow_A V \implies \exists U. N \downarrow_A U \land V \mathcal{R} U$$

This relation transformer is closed under primitive reduction and expansion. We state and sketch a proof for left-closure under $\leadsto$.

**Lemma 2.5** ([ ]$^T$ left-closed under $\leadsto$). For all closed terms $M, N$ of type $A$ and $\mathcal{R}$ a binary relation on closed values of type $A$, the following properties hold.

1. If $M \leadsto P$ and $M [\mathcal{R}]^T N$ then $P [\mathcal{R}]^T N$
2. If $M \leadsto P$ and $P [\mathcal{R}]^T N$ then $M [\mathcal{R}]^T N$

**Proof.** By the definition of $[ ]^T$ and Lemma 2.3.

We next define a basic relation that drives all our other notions of approximation. This aims to capture what values can be directly distinguished, without further context or evaluation.

**Definition 2.6** (Ground equivalence). Let $\equiv_A$ be the equivalence relation defined on closed values of type $A$ by:

- $V \equiv_r U \iff V = U$, for all $V, U : r$,
- $\lambda_A M \equiv_{A\rightarrow B} \lambda_A N$ for all $M, N$

One reason to factor this out in our development is the potential for incorporating non-trivial equivalences on ground values (such as for adding primitive operations and their definitions) in a similar fashion to Johann et al. [14] (see Section 5 for more on this).

Although this particular relation is symmetric, the definitions we build on it are biased to give approximation preorders. In particular this anticipates calculi with nontermination, where termination is a basic observable.

Now we are in a position to define observational approximation in terms of ground equivalence. First, we have a definition of approximation parametric in the notion of contexts we consider (VSCs, VCCs, . . .). Let $\mathcal{K}$ range over these notions of contexts. A *program* is a closed term of arbitrary type. A *program $\mathcal{K}$-context* is a closed term $\mathcal{K}$-context ranged over by $P, Q$, i.e. $\vdash P : (\Gamma \vdash A)^{\mathcal{K}} B$; where for convenience we usually omit the empty environment.

**Definition 2.7** ($\mathcal{K}$ Approximation). If $M, N$ are terms of type $A$ in context $\Gamma$ then

$$\Gamma \vdash M \ll_{\mathcal{K}} A N$$

asserts that for all program contexts $P : (\Gamma \vdash A)^{\mathcal{K}} B$,

$$P [\langle\langle M\rangle\rangle^{\mathcal{K}} \equiv_{\mathcal{K}} B]^T P [\langle\langle N\rangle\rangle^{\mathcal{K}}]$$

From all $\mathcal{K}$-contextual approximations, we choose as primary the one with the largest class of contexts — thereby able to make the finest discrimination between terms.

**Definition 2.8** (Observational Approximation). Let observational approximation, written $\ll_{\text{VSC}}$, be Definition 2.7 instantiated with $\mathcal{K} = \text{VSC}$.

Now we can establish our first inclusion of approximations.
Lemma 2.9. If Γ ⊢ M ≲ VSC A N then Γ ⊢ M ≲ ASC A N.

Proof. Assume given terms Γ ⊢ M, N : A with Γ ⊢ M ≲ VSC A N and VCC P : ⟨⟨Γ ⊢ A⟩⟩VCC B. From P we can build matching VSC Q where each hole carries the identity substitution. The result follows by using our VSC approximation with M and N, and the property that for all terms Γ ⊢ M’ : A, P ⟨⟨M’⟩⟩VCC ≡ Q ⟨⟨M’⟩⟩VSC (proved by induction on the structure of P). □

Definition 2.7 can readily be extended to an equivalence, denoted by ≈K:
Γ ⊢ M ≲ K A N ≜ Γ ⊢ M ≲ K A N ∧ Γ ⊢ N ≲ K A M.

In the remainder of this paper results will be stated for the approximations only but can be straightforwardly lifted to equivalences [30, 32].

3 Big Steps for a Context Lemma

In this section we define three approximation relations and prove a triangle of implications which together lead to a context lemma inspired by Milner’s original for combinatory logic [20].

As noted in the introduction, proving observational approximation directly can be arduous owing to the requirement to consider all possible program contexts. Happily, Milner [20] showed that for typed combinatory logic it is enough to consider only contexts where the hole occurs in function position applied to some arguments. The following definition captures such contexts in our setting.

Definition 3.1 (Applicative Substituting Contexts). The applicative substituting contexts (ASCs) are the restricted class of contexts defined by the following inference rules:

\[
\begin{align*}
\text{ASC-Hole} & : \theta : \Gamma \vdash A \\
\Delta & : \langle\langle\theta\rangle\rangle : \langle\langle\Gamma \vdash A\rangle\rangle^{ASC} A \\
\Delta & : \langle\langle\theta\rangle\rangle : \langle\langle\Gamma \vdash A\rangle\rangle^{ASC} B \rightarrow C \quad \Delta \vdash V : B \\
\Delta & : \langle\langle\theta\rangle\rangle : \langle\langle\Gamma \vdash A\rangle\rangle^{ASC} C
\end{align*}
\]

Definition 3.2 (ASC Approximation). Let ASC approximation be Definition 2.7 with \(K = \text{ASC}\).

Having introduced the required notation we may now formally state the context lemma we aim to prove in this section. Firstly, in our setting, Milner’s original context lemma states that for all open \(\lambda_{FG}\) terms \(\Gamma ⊢ M, N : A\),
\[\Gamma ⊢ M \Leftrightarrow \text{ASC} \quad \Gamma ⊢ M \Leftrightarrow \text{VSC}\]

However, we have taken our notion of VSCs as primary, so our context lemma becomes
\[\Gamma ⊢ M \Leftrightarrow \text{ASC} \quad \Gamma ⊢ M \Leftrightarrow \text{VSC}\]

Before proving the above result we define some auxiliary notions.

\[
\begin{align*}
\text{betaV-Ref} & : M \rightarrow^{\beta V} M \\
\text{betaV-Step} & : M \rightarrow^{\beta V} (\lambda A N) V \\
\text{betaV} & : M \rightarrow^{\beta} N \quad \text{betaL} \quad M \rightarrow^{\beta} M' \\
\text{let} & : M N \rightarrow^{\beta} \text{let}_A M N \equiv M[\text{let}_A N[V]]
\end{align*}
\]

Figure 7. \(\lambda_{FG}\) iterated beta-reduction with respect to sequencing.

Definition 3.3 (Beta-Redexes from Substitution). Let \(\theta\) be a \(\Delta\)-closing substitution. For \(\Delta ⊢ D : \langle\langle\Gamma \vdash A\rangle\rangle^{VCC} B\), define \(D^\theta\) by induction on the size of \(\Delta\):
\[D^\theta = \begin{cases} D & \Delta = \cdot \\ ((\lambda C D) \theta_0)^{\text{succ}} & \Delta = \Delta', C \end{cases}\]

\(D^\theta\) represents a new (closed) VCC obtained from \(D\) by constructing a sequence of beta-redexes on the outside of \(D\) from the substitution \(\theta\). For \(\Delta ⊢ M : A\), let \(M^\theta\) denote the analogous operation for constructing a closed term from \(M\) and \(\theta\).

Figure 7 defines iterated beta-reduction with respect to term sequencing. Both relations satisfy analogous properties to Lemmas 2.3 and 2.5. Then we have the obvious result

Lemma 3.4. For \(\Delta ⊢ M : A\), and \(\Delta\)-closing substitution \(\theta\), then we have

\[M^\theta \rightarrow^{\beta V} \theta(M)\]

Definition 3.5. Define the operation \(\star : \text{ASC} \rightarrow \text{VCC}\), which transforms a closed ASC into a closed VCC, by induction on the structure of the ASC:
\[
\star(\langle\langle\theta\rangle\rangle) = \langle\langle\theta^\theta\rangle\rangle \\
\star(P V) = \star(P) V
\]

where the ascApp case uses our syntactic sugar, extended to \(K\) contexts.

Lemma 3.6. For all \(\Gamma ⊢ M : A\), \(\Delta ⊢ C : \langle\langle\Gamma \vdash A\rangle\rangle^{VCC} B\), and \(\Delta\)-closing substitutions, \(\theta\), we have
\[(C^\theta)\langle\langle M\rangle\rangle = \langle\langle C\langle\langle M\rangle\rangle\rangle\rangle^\theta\]

Proof. By induction on the size of the environment \(\Delta\). □

Lemma 3.7. If \(\Gamma ⊢ M \Leftrightarrow \text{VSC} A N\) then \(\Gamma ⊢ M \Leftrightarrow \text{ASC} A N\).

Proof. Assume given terms \(\Gamma ⊢ M, N : A\), \(\Gamma ⊢ M \Leftrightarrow \text{VSC} A N\) and \(\text{ASC} P : \langle\langle\Gamma \vdash A\rangle\rangle^{ASC} B\). The proof proceeds by induction on the structure of \(P\).

If \(P = \langle\langle\theta\rangle\rangle\), for some \(\Gamma\)-closing substitution \(\theta\), then the result follows from Lemmas 3.6 and 3.4, using \(\theta\)'s left- and right-closure under \(\rightarrow^{\beta V}\).
The contextual approximations for VSCs, VCCs and ASCs form the apex of the triangle, while \( \preceq \) is applicative approximation and \( \preceq \) is logical approximation, both to be defined shortly. We have already shown the implications within the dotted ellipse (Lemmas 2.9 and 3.7), and the next two sections will address those forming the triangle.

### 3.1 Applicative Approximation

The definition of applicative approximation roughly follows Stark’s definition [32].

**Definition 3.8 (Applicative Approximation).** We define applicative approximation \( \preceq \) using two mutually recursive definitions, one for values and one for terms.

Define the relation \( V_1 \preceq_{val} V_2 \) for closed values \( V_1, V_2 \) inductively on the structure of type \( A \):

\[
\begin{align*}
&\text{AppGndApx} & V_1 \equiv V_2 & \quad \forall V : A. M_1[V] \preceq_{trm} M_2[V] \\
&\text{AppAbsApx} & V_1 \preceq_{val} V_2 & \quad \lambda_A M_1 \preceq_{val} A \rightarrow B \lambda_A M_2
\end{align*}
\]

Define the relation \( M_1 \preceq_{trm} M_2 \) for closed terms \( M_1, M_2 \) of type \( A \) by the following definition:

\[
M_1 \preceq_{trm} A \quad M_2 \quad \triangleq \quad M_1 [\mathit{val}]^T M_2
\]

Just like for observational approximation, applicative approximation can be stated for open terms. We employ simultaneous substitutions to close over the environment.

**Definition 3.9 (Open Applicative Approximation).** Given \( \Gamma \vdash M_1 : A \) and \( \Gamma \vdash M_2 : A \), we say that \( M_1 \) applicatively approximates \( M_2 \), written \( \Gamma \vdash M_1 \preceq_{A} M_2 \), if and only if for all \( \Gamma \)-closing substitutions \( \theta \) we have \( \theta(M_1) \preceq_{A} \theta(M_2) \).

A relationship between ASC approximation and applicative approximation can be established from which we obtain one side of the triangle.

\[
\begin{align*}
&\text{VCC} \quad \preceq \quad \text{VSC} & \triangleq \quad \text{ASC} \\
&\text{ASC} & \triangleq \quad \text{VSC} \quad \preceq \quad \text{VCC}
\end{align*}
\]

The contextual approximations for VSCs, VCCs and ASCs form the apex of the triangle, while \( \preceq \) is applicative approximation and \( \preceq \) is logical approximation, both to be defined shortly. We have already shown the implications within the dotted ellipse (Lemmas 2.9 and 3.7), and the next two sections will address those forming the triangle.

### 3.2 Logical Approximation

Logical approximation differs with respect to applicative approximation only in its handling of functions.

**Definition 3.12 (Logical Approximation).** Define logical approximation by the mutually recursive relations \( \preceq_{val} \) and \( \preceq_{trm} \). The relation \( V_1 \preceq_{val} V_2 \) for closed values \( V_1, V_2 \) is defined recursively on the structure of type \( A \):

\[
\begin{align*}
&\text{LogGndApx} & V_1 \equiv_{trm} V_2 & \quad \forall V : A. M_1[V] \preceq_{trm} M_2[V] \\
&\text{LogAbsApx} & V_1 \preceq_{val} V_2 & \quad \lambda_A M_1 \preceq_{val} A \rightarrow B \lambda_A M_2
\end{align*}
\]

The relation \( M_1 \preceq_{trm} M_2 \) for closed terms \( M_1, M_2 \) of type \( A \) is defined using \( [\cdot]^T \) and \( \preceq_{val} \):

\[
M_1 \preceq_{trm} A \quad M_2 \quad \triangleq \quad M_1 [\mathit{val}]^T M_2
\]

To extend logical approximation to open terms we define the notion of logical approximation of substitutions.

**Definition 3.13 (Logical Approximation of Substitutions).** For \( \Gamma \)-closing substitutions, \( \theta, \theta' \), \( \theta \preceq \theta' \) holds if and only if, for all \( k : (A \in \Gamma) \), \( \theta_k \preceq \theta'_k \). For \( \theta, \theta' : \Gamma \vdash \Delta \), \( \beta \preceq \beta' \) holds if and only if, for all \( \Delta \)-closing substitutions, \( \theta_1, \theta_2 \) such that \( \theta_1 \preceq \theta_2 \) holds then \( \theta_1 \circ \beta \theta_2 \circ \beta' \) holds.

**Definition 3.14 (Open Logical Approximation).** Assume given \( \Gamma \vdash M_1 : A \) and \( \Gamma \vdash M_2 : A \), then \( M_1 \) logically approximates \( M_2 \), written \( \Gamma \vdash M_1 \preceq_{A} M_2 \), if and only if, for all \( \Gamma \)-closing substitutions \( \theta_1, \theta_2 \) if \( \theta_1 \preceq \theta_2 \) then \( \theta_1(M_1) \preceq_{A} \theta_2(M_2) \).

Logical approximation is closed under primitive reduction, and expansion.

**Lemma 3.15** (\( \preceq_{trm} \) left-closed under \( \rightarrow \)). If \( M \rightarrow P \) and \( M \preceq_{A} N \) then \( P \preceq_{A} N \), and if \( M \rightarrow P \) and \( P \preceq_{A} N \) then \( M \preceq_{A} N \).

**Lemma 3.16** (\( \preceq_{trm} \) right-closed under \( \rightarrow \)). If \( N \rightarrow P \) and \( M \preceq_{A} N \) then \( M \preceq_{A} P \), and if \( N \rightarrow P \) and \( M \preceq_{A} P \) then \( M \preceq_{A} N \).

The following lemma follows directly from the above definition of open logical approximation.
with the following generalised transitivity property [29, 30]:

\[
\begin{align*}
\text{LOGAPXIfThenElse} & \quad B \preceq^\text{val}_{\text{base}} B' \quad L \preceq^\text{trm} L' \quad R \preceq^\text{trm} R' \\
\text{if } B \text{ then } L \text{ else } R & \preceq^\text{trm} \text{ if } B' \text{ then } L' \text{ else } R'
\end{align*}
\]

**Proof.** By simultaneous induction on the structure of type 

3.3 Completing the Triangle

**Lemma 3.18 (Logical Apx. Term Closure).** Logical approximation is closed under the compound term formers (Figure 8).

**Proof.** By Lemmas 3.15 and 3.16, and the \textbf{let} rule for big-step evaluation.

**Lemma 3.19 (Fundamental Theorem of Logical Relations).** Logical approximation is reflexive:

1. \( \Gamma \vdash V \preceq^\text{val} V \)
2. \( \Gamma \vdash M \preceq^\text{trm} M \)

**Proof.** Perform simultaneous induction on the typing derivations \( \Gamma \vdash V : A \) and \( \Gamma \vdash M : A \) using Lemma 3.18.

4 Extension to Frame Stacks

Since Milner’s original context lemma, a number of similar results have appeared for more complex calculi. In particular, for calculi with effects (such as state) applicative contexts are not sufficient and one requires a more powerful notion known as Closed Instantiations of Uses (CIU) approximation [19].

In this section we consider a different triangle of approximations involving frame stacks [29]. We parallel the ReFS proofs of Pitts and Stark [30] recasting the applicative and logical approximations described in Section 3 in terms of frame stack evaluation. First though, we define the correct contexts and relate them to VCCs just like that for ASCs. We omit some proofs since they follow similar reasoning. For an alternative account of a similar development, the reader may wish to consult the chapter by Pitts [29] in Pierce’s book [26].

**Definition 4.1 (CIU Contexts).** Let \textbf{CIU contexts} be the restricted class of contexts defined by the following inference rules:

\[
\begin{align*}
\text{ciuHole} & \quad \theta : \Gamma \vdash \Delta \\
\Delta \vdash \langle \theta \rangle : \langle \Gamma \vdash A \rangle^\text{CIU} & \quad \Delta, B \vdash N : C \\
\text{ciuApp} & \quad \Delta \vdash D : \langle \Gamma \vdash A \rangle^\text{CIU} B \\
\Delta, B \vdash N : C & \quad \Delta \vdash \text{let}_B D N : \langle \Gamma \vdash A \rangle^\text{CIU} C
\end{align*}
\]

It is worth stressing the differences between ASCs and CIU contexts. ASCs consist of a well-typed and well-scoped hole applied to a sequence of closed values. CIU contexts form a nested sequence of let bindings (culminating in the hole) with a term body.
We may extend the @ operation to operate on closed CIU contexts instead of terms, denoted by − ⟨⟨@⟩−⟩ − , replacing the case for ConsFrmTy above with:

$$(S \circ_{A} N)⟨@⟩P = S⟨@⟩(\text{let } P N)$$

The type of the holes in the context are left unchanged by the operation. The following lemma establishes that the action on a frame stack commutes with context instantiation.

**Lemma 4.5.** $(S⟨@⟩P)⟨M⟩_{CIU} = S⟨@⟩(P⟨M⟩)_{CIU}$

**Proof.** By induction on the frame stack $S$. □

The following lemma relates frame stack and big-step evaluation using the action over a frame stack.

**Lemma 4.6.** $(S, M) \downarrow V \iff S@M \downarrow V$.

**Proof.** For $\Rightarrow$, the proof follows by the properties:

1. $M \downarrow V \Rightarrow \langle Id, M \rangle \downarrow V$
2. $\langle Id, S@M \rangle \downarrow U \Rightarrow \langle S, M \rangle \downarrow U$

where (1) is by induction on the derivation of $M \downarrow V$ and (2) is by induction on $S$.

For $\Leftarrow$, the proof follows from a standardisation argument [33]: evaluation can be decomposed into evaluation of a focussed term $M$ to value $W$ and evaluation of the surrounding context $S$ filled with $W$.

3. $S@M \downarrow V \Rightarrow \exists W.M \downarrow W @ S@W \downarrow V$
4. $M \downarrow W \wedge S@W \downarrow V \Rightarrow S@M \downarrow V$

Both are proved simultaneously by induction on $S$. □

We define a relation transformer that lifts a relation on values to a relation between frame stack configurations ($\prec, \prec$).

**Definition 4.7.** For frame stacks $S_1, S_2 : A \prec B$, closed terms $M_1, M_2 : A$, and binary relation $R$ on closed values of type $B$, relation $S_1, M_1 \vdash [R] F S_2, M_2$ holds if and only if

$$\forall V. (S_1, M_1) \downarrow V \Rightarrow \exists U. (S_2, M_2) \downarrow U \wedge V \uparrow R \uparrow U.$$
We now define the frame stack analogue to logical approxi-
mation. We implicitly lift the value arguments to terms.

As with applicative frame approximation,

The relation \( \langle \cdot \rangle^\text{val}_{\Lambda} \) is defined to be

The logical frame approximation is defined by three mutually recursive

where the approximation \( \langle \cdot \rangle_{\text{stk}}^\Lambda \) is defined to be

The result follows from Lemma 3.2.

Lemma 4.10. Observational approximation implies applica-
tive frame approximation.

Proof. By Lemmas 2.9, 4.4 and 4.9. \( \square \)

4.3 Logical Frame Approximation

We now define the frame stack analogue to logical approxi-
mation from Section 3.2, using \( T T \)-lifting / biorthogonality
for logical relations as demonstrated by Pitts and Stark [30].

Definition 4.11 (Logical Frame Approximation). Logical
frame approximation is defined by three mutually recursive
relations on pairs of (closed) values, terms and stacks. De-
define \( V_1, V_2 \) for closed values \( V_1, V_2 \) inductively by the
structure of type \( \Lambda \):

The relation \( \langle \cdot \rangle^\text{stk}_{\Lambda} \) is defined by the \( T T \)-lifting of \( \langle \cdot \rangle^\text{val}_{\Lambda} \) over \( \langle \cdot \rangle^\text{stk}_{\Lambda} \): for closed terms \( M_1 \) and \( M_2 \), approximation
\( M_1 \langle \cdot \rangle^\text{stk}_{\Lambda} M_2 \) is defined to be

As with applicative frame approximation, \( \langle \cdot \rangle^\text{stk}_{\Lambda} \) uses the \( \equiv \)
relation to relate the final values. In the definition of \( \langle \cdot \rangle^\text{stk}_{\Lambda} \), we implicitly lift the value arguments to terms.

As in Lemmas 3.15, 3.16, and 3.22, logical frame approxi-
mation is closed under primitive reduction and expansion,
and closed under all VSCs.

Lemma 4.12 (Logical Frame Apx. Lifts). Logical frame ap-
proximation is closed under the compound term formers.

Proof. The only deviation from the proof of Lemma 3.18 is for \( \lambda e, T \),
where we extend the approximation of stacks from
\( S_1 \langle \cdot \rangle^\text{stk}_{\Lambda_{\Lambda \rightarrow B}} S_2 \) to
\( S_1 \circ_{\Lambda} N \langle \cdot \rangle^\text{stk}_{\Lambda_{\Lambda \rightarrow C \rightarrow B}} S_2 \circ_{\Lambda} N' \).

Lemma 4.13. Logical frame approximation is reflexive:

1. \( \Gamma \vdash V \langle \cdot \rangle^\text{val}_{\Lambda} V \)

2. \( \Gamma \vdash M \langle \cdot \rangle^\text{frm}_{\Lambda} M \)

3. \( S \langle \cdot \rangle^\text{stk}_{\Lambda} S \)

Proof. For (1) and (2) the proof is similar to Lemma 3.19
using Lemma 4.12. For (3), perform induction on the well-
typed derivation of \( S \) using (2) for consValue.

Lemma 4.14. If \( \Gamma \vdash M_1 \langle \cdot \rangle^\text{frm}_{\Lambda} M_2 \)
and \( \Delta \vdash C : \langle \Gamma \rangle_A B \) then
\( \Delta \vdash C \langle M_1 \rangle_B \langle \cdot \rangle^\text{frm}_{\Lambda} C \langle M_2 \rangle_B \)

4.4 Establishing the Triangle

We can now proceed to the following analogue of Lemma 3.23.

Lemma 4.15. Logical frame approximation implies observa-
tional approximation:

\( \Gamma \vdash M \langle \cdot \rangle^\text{frm}_{\Lambda} N \implies \Gamma \vdash M \langle \cdot \rangle^\text{VSC}_{\Lambda} N \)

Proof. By definition, using Lemma 4.14. \( \square \)

Lemma 4.16. Applicative and logical frame approximation
satisfy the following transitivity property:

\( M \langle \cdot \rangle^\text{frm}_{\Lambda} N \) and \( N \langle \cdot \rangle^\text{frm}_{\Lambda} P \) then
\( M \langle \cdot \rangle^\text{frm}_{\Lambda} P. \)

Proof. By induction on the structure of type A. \( \square \)

We establish the final implication of the triangle: that
applicative frame approximation implies logical frame approxi-
mation.

Lemma 4.17.

\( \Gamma \vdash M \langle \cdot \rangle^\text{frm}_{\Lambda} N \implies \Gamma \vdash M \langle \cdot \rangle^\text{frm}_{\Lambda} N \)

Proof. By Lemmas 4.13 and 4.16. \( \square \)

Once again, all three notions coincide.

Lemma 4.18 (Frame Stack Triangulation). Observational,
applicative frame and logical frame approximation coincide.

Proof. Using Lemmas 4.10, 4.17 and 4.15. \( \square \)

For a simply-typed lambda calculus like \( \lambda_{FG} \), it is not in
fact essential to consider CIU approximation — we already
know from Section 3 that applicative contexts are enough
to distinguish terms. Differences, though, arise in any lan-
guage with features like state, exceptions, or other effects
that give program contexts greater discriminating power. In
these cases we expect the frame-stack approach to be appro-
piate: and note that the triangulation proofs are reassuringly
similar in shape when moving from Milner’s original context
lemma to a CIU version. We hope to maintain this in scaling
the development to support effectful calculi, in particular
those based on algebraic effects and handlers [6, 18].
5 Related Work

Program equivalence has a long history, so an exhaustive account of the literature on the subject is impossible in the space available. We identify brief highlights that have had a direct influence on this work.

Howe developed a general method for proving coincidence of applicative bisimilarity and observational equivalence for a broad class of (untyped) lazy languages [12] and subsequently extended the method to support call-by-value calculi [13]. A key element of the approach is to introduce a relation, called the ‘precongruence candidate’, which bridges the gap between the applicative and observational notions. The technique has been shown to extend to other languages, including a typed metalanguage with recursive types [11] and extensions of PCF [15, 28]. Howe’s method does not reason explicitly about reduction sequences in contrast to others [19, 32]. Instead, the precongruence candidate is sufficient to show that the relation is closed under all contexts, treating contexts abstractly. Our approach is a middle ground in that we reason explicitly about the structure of contexts but do not explicitly analyse reduction sequences.

Since our original submission, we have formalised Howe’s construction as (yet another) operator \([\_ ]^H\) on relations, proving its basic properties (reflexivity, transitivity, substitutivity, and crucially, the analogue of Lemma 3.22). We have further proved that \([\_ ]^H \subseteq \subseteq\), making essential use of (an analogue of) Lemma 3.20. Hence, by virtue of our existing (big-step) triangulation, we obtain that \([\_ ]^H \subseteq \subseteq\), since \(R \subseteq [R]^H\) for \(R\) reflexive.

Stark showed a triangle of relations equivalent for the nu-calculus and proved a context lemma for an instantiated hole applied to functions returning a boolean value [32]. However, this special form of context lemma is proved by analysing the process of reduction and the particular forms for closed expressions in the calculus. In contrast, our fine-grained calculus is simple enough for consideration of applicative contexts à la Milner [20] to suffice, and hence the context lemma follows from the triangulation result.

Pitts and Stark prove a context lemma, which includes the CIU theorem, for a functional language with local state [30]. Their use of the triangulation technique inspired this work and our Section 4 essentially isolates their approach, free from considerations of state relations. However, our formal approach gives a more satisfactory account of contexts, their scope and the variable capture involved when instantiating a hole with a term. In particular, there is no requirement for a side-condition in our analogue (Theorem 4.14) of their Theorem 4.9 since by construction every occurrence of a hole (in a VSC) is paired with a well-typed and wellscoped substitution.

Pitts [29] developed a triangle for an ML-like language, but in contrast to ReFS [30], it is based on an extension of Howe’s relational approach [11, 15]. The relational approach for representing contexts is favoured in order to mitigate the difficulties involved with concrete contexts and their potential capture of free variables of a term. The folklore belief is that such issues are especially difficult to handle in the setting of machine checked proofs. On the contrary, a careful choice of representation and appropriate use of state-of-the-art technology makes their representation routine. Moreover, the relational approach relies on an inductive characterisation in terms of the language syntax. Some of these rules involve the use of variables from the context. It is not clear how one would formalise such definitions without having to deal with issues of naming and \(\alpha\)-convertibility; issues handled automatically by the ACMM framework.

We note that we have in this work used straightforward recursion and structural induction in our definitions and proofs. This is to be expected for a simply-typed lambda-calculus, where all evaluation is terminating. However, previous work on ReFS shows that frame stacks are also sufficient to handle recursive definitions, termination, and unwinding [30, §3]. The key feature is how the structure of frame stacks captures evaluation in fine detail, guiding the corresponding proof structure. We conjecture that this auto-calibration may help postpone or even avoid the need for step-indexing [1, 4] when moving to more elaborate language features.

More generally, we consider that the significant characteristic of all our structural relations, applicative or logical, is not recursion or induction but their closure under inference rules and biorthogonality / \(\top\top\)-lifting.

Regarding formalisations of context-lemma-like results, the nearest related work we can find is by Ambler and Crole, in Isabelle/HOL [3], subsequently with Momigliano in the Hybrid system [22], then by Momigliano in the Abella theorem prover [21], and by Thibodeau et al. in the Beluga theorem prover [35]. With the exception of Ambler and Crole’s work in HOL, also based on a concrete de Bruijn representation, the other authors address higher-order abstract syntax (HOAS) presentations, together with greater or lesser sophistication in the implementation of the meta-level to account for term contexts and open approximation relations. In each case, the authors use Howe’s coinductive method for defining applicable simulation and proving it a pre-congruence, but do not show its coincidence with observational approximation as we do in Lemmas 3.23 and 4.15. Indeed, Thibodeau et al.’s HOAS approach does not support any kind of variable-capturing contexts, while the authors note Beluga requires a non-trivial extension to encode observational approximation using the relational approach [16, 27].

Prior to Thibodeau et al., it seems to have been extremely difficult to judge formalisation work as robust or reusable. Ambler and Crole themselves note how much work went into the machinery of substitution and de Bruijn manipulation; we might conjecture that our use of a dependently-typed metalanguage confers many advantages over HOL formalisations in terms of building-in well-typed invariants to our
definitions, while our exploitation of ACMM demonstrates the versatility and abstraction in a first-order deep embedding combined with shallow HOAS reasoning techniques. Momigliano’s suggestive title hints at his reluctance to repeat the experiment. Working in Beluga. Thibodeau et al. seem to have achieved robustness, readability, and (potential) reusability, but at the cost of a much more sophisticated meta-language than our concrete approach in Agda.

Johann et al. study similar results in a general operational setting for a polymorphic calculus extended with algebraic effects [14]. Like us, they parameterise their approximations with respect to a notion of ‘basic’ preorder for making observations on ground type computations. Their expressive setting provides justification for the parameterisation, permitting different choices of basic preorder to give a semantics with respect to different collections of effects. Their work offers hope that our results will extend to calculi with algebraic effects and handlers.

6 Conclusion
We have drawn out the distinctive proof technique of triangulation utilised in proofs of previous context lemmas. We isolated the method by considering two versions in the setting of a fine-grained call-by-value calculus: the applicative or classical result inspired by Milner; and the CIU-like result required of effectful calculi. Moreover, we formalised our results using state-of-the-art technology for representing well-typed and well-scoped syntax — in particular, the explicit first-order representation of observational contexts.

We have contributed to the exposition of triangulation as a general technique for proving context lemmas; its formalisation provides a robust basis for future work on more exotic calculi. In particular, we see frame stacks as a good model for reasoning about handler stacks and the operational semantics of algebraic effects in languages like Eff and Frank [6, 18].

Acknowledgments
We are grateful to the anonymous referees for their comments and suggestions, and to the Agda developers for their help in ironing out inconsistencies in checking our proofs under versions 2.5.1, 2.5.2 and 2.5.3 of the Agda system. All three authors acknowledge the support of the Laboratory for Foundations of Computer Science (LFCS); the first author is also supported by the EPSRC under doctoral training grant EP/K503034/1.

References
Table 1. Selected notation used in the paper and corresponding Agda definitions

<table>
<thead>
<tr>
<th>Concept</th>
<th>Notation</th>
<th>Reference</th>
<th>Agda Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simultaneous substitution</td>
<td>$\theta(\cdot)$</td>
<td>Definition 2.1</td>
<td>$_\text{Val}$</td>
</tr>
<tr>
<td>Lookup in substitution</td>
<td>$\theta_k$</td>
<td>Definition 2.2</td>
<td>$\text{var}$</td>
</tr>
<tr>
<td>Successor substitution</td>
<td>$\text{succ} \cdot$</td>
<td>Definition 2.2</td>
<td>$\text{succ}$</td>
</tr>
<tr>
<td>Lift value relation to terms by evaluation</td>
<td>$\lceil \cdot \rceil^F$</td>
<td>Definition 2.4</td>
<td>$\text{\llbracket \cdot \rrbracket}^F$</td>
</tr>
<tr>
<td>Ground equivalence</td>
<td>$\cong$</td>
<td>Definition 2.6</td>
<td>$\text{gnd-eqv}$</td>
</tr>
<tr>
<td>Variable-capturing contexts</td>
<td>$(\cdot \vdash \cdot)^{\text{VCC}}$</td>
<td>Figure 6</td>
<td>$\text{VCC}(\cdot \vdash \cdot)$</td>
</tr>
<tr>
<td>Observational approximation</td>
<td>$\subseteq^{\text{VSC}}$</td>
<td>Definition 2.8</td>
<td>$\text{vsc-apx}$</td>
</tr>
<tr>
<td>Applicative substituting contexts</td>
<td>$(\cdot \vdash \cdot)^{\text{ASC}}$</td>
<td>Definition 3.1</td>
<td>$\text{ASC}(\cdot \vdash \cdot)$</td>
</tr>
<tr>
<td>ASC approximation</td>
<td>$\subseteq^{\text{ASC}}$</td>
<td>Definition 3.2</td>
<td>$\text{asc-apx}$</td>
</tr>
<tr>
<td>Beta-redexes from substitution</td>
<td>$\mathcal{D}^\theta$</td>
<td>Definition 3.3</td>
<td>$\text{VCC}(\cdot \vdash \cdot)$</td>
</tr>
<tr>
<td>Transform ASC to VCC</td>
<td>$\star$</td>
<td>Definition 3.5</td>
<td>$\text{asc-to-vcc}$</td>
</tr>
<tr>
<td>Applicative approximation</td>
<td>$\subseteq$</td>
<td>Definition 3.8</td>
<td>$\text{app-apx}_\theta$</td>
</tr>
<tr>
<td>Open applicative approximation</td>
<td>$\Gamma \vdash M \subseteq_A N$</td>
<td>Definition 3.9</td>
<td>$\text{app-apx}$</td>
</tr>
<tr>
<td>Logical approximation</td>
<td>$\subseteq$</td>
<td>Definition 3.12</td>
<td>$\text{log-apx}_\theta$</td>
</tr>
<tr>
<td>Open logical approximation</td>
<td>$\Gamma \vdash M \subseteq_A N$</td>
<td>Definition 3.14</td>
<td>$\text{log-apx}_\theta$</td>
</tr>
<tr>
<td>CIU contexts</td>
<td>$(\cdot \vdash \cdot)^{\text{CIU}}$</td>
<td>Definition 4.1</td>
<td>$\text{CIU}(\cdot \vdash \cdot)$</td>
</tr>
<tr>
<td>CIU Approximation</td>
<td>$\subseteq^{\text{CIU}}$</td>
<td>Definition 4.2</td>
<td>$\text{ciu-apx}$</td>
</tr>
<tr>
<td>Transform CIU Contexts to VCC</td>
<td>$\flat$</td>
<td>Definition 4.3</td>
<td>$\text{ciu-to-vcc}$</td>
</tr>
<tr>
<td>Frame stacks relation transformer</td>
<td>$\lceil \cdot \rceil^F$</td>
<td>Definition 4.7</td>
<td>$\text{\llbracket \cdot \rrbracket}^F$</td>
</tr>
<tr>
<td>Applicative Frame Approximation</td>
<td>$\langle \subseteq \rangle$</td>
<td>Definition 4.8</td>
<td>$\text{app-frm-apx}_\theta$</td>
</tr>
<tr>
<td>Logical Frame Approximation</td>
<td>$\langle \subseteq \rangle$</td>
<td>Definition 4.11</td>
<td>$\text{log-frm-apx}_\theta$</td>
</tr>
</tbody>
</table>