LOGICAL RELATIONS FOR COHERENCE OF EFFECT SUBTYPING

DARIUSZ BIERNACKI AND PIOTR POLESIUK

Institute of Computer Science, University of Wrocław, Joliot-Curie 15, 50-383 Wrocław, Poland
e-mail address: dabi@cs.uni.wroc.pl

Institute of Computer Science, University of Wrocław, Joliot-Curie 15, 50-383 Wrocław, Poland
e-mail address: ppolesiuk@cs.uni.wroc.pl

ABSTRACT. A coercion semantics of a programming language with subtyping is typically defined on typing derivations rather than on typing judgments. To avoid semantic ambiguity, such a semantics is expected to be coherent, i.e., independent of the typing derivation for a given typing judgment. In this article we present heterogeneous, biorthogonal, step-indexed logical relations for establishing the coherence of coercion semantics of programming languages with subtyping. To illustrate the effectiveness of the proof method, we develop a proof of coherence of a type-directed, selective CPS translation from a typed call-by-value lambda calculus with delimited continuations and control-effect subtyping. The article is accompanied by a Coq formalization that relies on a novel shallow embedding of a logic for reasoning about step-indexing.

1. INTRODUCTION

Programming languages that allow for subtyping, i.e., a mechanism facilitating coercions of expressions of one type to another, are usually given either a subset semantics, where one type is considered a subset of another type, or a coercion semantics, where expressions are explicitly converted from one type to another. In the presence of subtyping, typing derivations depend on the occurrences of the subtyping judgments and, therefore, typing judgments do not have unique typing derivations. Consequently, a coercion semantics that interprets subtyping judgments by introducing explicit type coercions is defined on typing derivations rather than on typing judgments. But then a natural question arises as to whether such a semantics is coherent, i.e., whether it does not depend on the typing derivation.

The problem of coherence has been considered in a variety of typed lambda calculi. Reynolds proved the coherence of the denotational semantics for intersection types in the category-theoretic setting [32]. Breazu-Tannen et al. proved the coherence of a coercion

translation from the lambda calculus with polymorphic, recursive and sum types to system F, by showing that any two derivations of the same judgment are translated to provably equal terms in the target calculus [12]. Curien and Ghelli introduced a translation from system $F_\le$ to a calculus with explicit coercions and showed that any two derivations of the same judgment are translated to terms that are normalizable to a unique normal form [13]. Finally, Schwinghammer followed Curien and Ghelli's approach to prove the coherence of coercion translation from Moggi's computational lambda calculus with subtyping, except that he normalizes derivations in a semantics-preserving way, rather than terms in a dedicated calculus of coercions [34]. Schwinghammer's presentation is akin to Mitchell's for the simply-typed lambda calculus [27, Chapter 10].

The results listed above fall into two categories: those that hinge on the existence of a common subtype of two different types for the same term [32, 12], and those that rely on finding a normal form for a representation of the derivation and hinge on showing that such normal forms are unique for a given typing judgment [13, 34, 27]. When the source calculus under consideration is presented in the spirit of the lambda calculus à la Church, i.e., the lambda abstractions are type annotated, as is the case in all the aforementioned articles that follow the normalization-based approach, the term and the typing context indeed determine the shape of the normal derivation (modulo a top level coercion that depends on the type of the term) [27, Chapter 10]. However, in calculi à la Curry this is no longer the case and the method cannot be directly applied. Still, if the calculus is at least weakly normalizing, one can hope to recover the uniqueness property for normal typing derivations for source terms in normal form, assuming that term normalization preserves the coercion semantics. For instance, in the simply typed λ-calculus the typing context uniquely determines the type of the term in the function position in applications building a β-normal form, and, hence, derivations in normal form for such terms are unique. This line of reasoning cannot be used when the calculus includes recursion. Similarly, the lambda calculus à la Curry (and other systems extending it) does not in general satisfy the property of common subtype.

In this article, we consider the coherence problem in calculi where none of the existing techniques can be directly applied. The coercion semantics we study translate typing derivations in the source calculus to a corresponding target calculus with explicit type coercions (that in some cases can be further replaced with equivalent lambda-term representations) and our criterion for coherence of the translation is contextual equivalence [28] in the target calculus.

The main result of this work is a construction of logical relations for establishing such a notion of coherence of coercion semantics, applicable in a variety of calculi. In particular, we address the problem of coherence of a type-directed CPS (continuation-passing style) translation from the call-by-value λ-calculus with delimited-control operators and control-effect subtyping introduced by Materzok and the first author [25], extended with recursion. While the translation for the calculus with explicit type annotations has been shown to be coherent in terms of an equational theory in a target calculus [24], no CPS coercion translation for the original version, let alone extended with recursion, has been proven coherent.

The reasons why coherence in this calculus is important are twofold. First of all, it is very expressive and therefore interesting from the theoretical point of view. In particular, the calculus has been shown to generalize the canonical type-and-effect system for Danvy and Filinski's shift and reset control operators [14, 15], and, furthermore, that it is strictly

more expressive than the CPS hierarchy of Danvy and Filinski [26]. These results heavily rely on the effect subtyping relation that, e.g., allows to coerce pure expressions (i.e., control-effect free) to effectful ones. From a more practical point of view, the selective CPS translation, that leaves pure expressions in direct style and introduces explicit coercions to interpret effect subtyping in the source calculus, is a good candidate for embedding the control operators in an existing programming language, such as Scala [33].

In order to deal with the complexity of the source calculus and of the translation itself, we introduce binary logical relations on terms of the target calculus that are: heterogeneous, biorthogonal [23, 30, 19], and step-indexed [3, 2, 1]. Heterogeneity allows us to relate terms of different types, and in particular those in continuation-passing style with those in direct style. This is a crucial property, since the same term can have a pure type, resulting in a direct-style term through the translation and another, impure type, resulting in a term in continuation-passing style. Relating such terms requires quantification over types and to assure well-foundedness of the construction, we need to use step-indexing, which also supports reasoning about recursion, even if not in a critical way. We follow Dreyer et al. [18] in using logical step-indexed logical relations in our presentation of step-indexing. Biorthogonality, by imposing a particular order of evaluation on expressions, simplifies the construction of the logical relations. It also facilitates reasoning about continuations represented as evaluation contexts.

Apart from the calculus with effect subtyping, we have used the ideas presented in this article to show coherence of subtyping in several other calculi, including the simply typed lambda calculus with subtyping [27, Chapter 10] extended with recursion, the calculus of intersection types [32], and the lambda calculus with subtyping and the control operator call/cc.

The article is accompanied by a Coq development containing a library IxFree that provides a new shallow embedding of the logic for reasoning about step-indexed logical relations, and a complete formalization of the proofs presented in the rest of the article. The code is available at https://bitbucket.org/pl-uwr/coherence-logrel.

The rest of this article is structured as follows. In Section 2, we briefly present Dreyer et al.'s logic for reasoning about step indexing [18] on which we base our presentation. In Section 3, we introduce the construction of the logical relations in a simple yet sufficiently interesting scenario—the simply typed lambda calculus à la Curry with natural numbers, type Top, general recursion and standard subtyping. The goal of this section is to introduce the basic ingredients of the proof method before embarking on a considerably more challenging journey in the subsequent section. In Section 4, we present the main result of the article—the logical relations for establishing the coherence of the CPS translation from the calculus of delimited control with effect subtyping. In Section 5, we describe the main ideas behind our Coq formalization. In Section 6, we summarize the article.

2. REASONING ABOUT STEP-INDEXED LOGICAL RELATIONS

Step-indexed logical relations [3, 2, 1] are a powerful tool for reasoning about programming languages. Instead of describing a general behavior of program execution, they focus on the first n computation steps, where the step index n is an additional parameter of the relation. This additional parameter makes it possible to define logical relations inductively not only on the structure of types, but also on the number of computation steps that are allowed for a program to make and, therefore, they provide an elegant way to reason about features

that introduce non-termination to the programming language, including recursive types [2] and references [1].

However, reasoning directly about step-indexed logical relations is tedious because proofs become obscured by step-index arithmetic. Dreyer et al. [18] proposed logical step-indexed logical relations (LSLR) to avoid this problem. The LSLR logic is an intuitionistic logic for reasoning about one particular Kripke model: where possible worlds are natural numbers (step-indices) and where future worlds have smaller indices than the present one. All formulas are interpreted as monotone (non-increasing) sequences of truth values, whereas the connectives are interpreted as usual. In particular, in the case of implication we quantify over all future worlds to ensure monotonicity, so the formula $\varphi \Rightarrow \psi$ is valid at index n (written $n \models \varphi \Rightarrow \psi$) iff $k \models \varphi$ implies $k \models \psi$ for every $k \leq n$. In contrast to Dreyer et al. we do not assume that all formulas are valid in world 0, because it is not necessary.

The LSLR logic is also equipped with a modal operator $\rhd$ (later), to provide access to strictly future worlds. The formula $\rhd\varphi$ means φ *holds in any future world*, or formally $\rhd\varphi$ is always valid at world 0, and $n + 1 \models \rhd\varphi$ iff φ is valid at n (and other future worlds by monotonicity). The later operator comes with two inference rules:

$$\frac{\Gamma, \Sigma \vdash \varphi}{\Gamma, \rhd\Sigma \vdash \rhd\varphi} \rhd\text{-intro} \qquad \frac{\Gamma, \rhd\varphi \vdash \varphi}{\Gamma \vdash \varphi} \text{Löb}$$

The first rule allows one to shift reasoning to a future world, making the assumptions about the future world available. The Löb rule expresses an induction principle for indices. Note that the premise of the rule also captures the base case, because the assumption $\rhd\varphi$ is trivial in the world 0. The later operator comes with no general elimination rule.

Predicates in LSLR logic as well as step-indexed logical relations can be defined inductively on indices. More generally, we can define a recursive predicate $\mu r.\varphi(r)$, provided all occurrences of r in φ are guarded by the later operator, to guarantee well-foundedness of the definition. For the sake of readability, in this paper we define recursive predicates and relations by giving a set of clauses instead of using the μ operator.

Since the logic is developed for reasoning about one particular model, we can freely add new inference rules for the logic if we prove they are valid in the model. We can also add new relations or predicates to the logic if we provide their monotone interpretation. In particular, constant functions are monotone, so we can safely use predicates defined outside of the logic, such as typing or reduction relations.

3. Introducing the logical relations

In this section we prove the coherence of subtyping in the simply-typed call-by-value lambda calculus extended with recursion, where the coercion semantics is given by a standard translation to the simply-typed lambda calculus with explicit coercions [13]. Our goal here is to introduce the proof method in a simple scenario, so that in Section 4 we can focus on issues specific to control effects. The logical relations we present in this section are biorthogonal and step-indexed, which is not strictly necessary but it makes the development more elegant. Furthermore, biorthogonality and step-indexing become crucial in handling more complicated calculi such as the one of Section 4 and, therefore, are essential for the method to scale.

$$\tau \quad ::= \quad \mathsf{Nat} \mid \mathsf{Top} \mid \tau \to \tau \qquad\qquad\qquad\qquad \text{(types)}$$
$$e \quad ::= \quad x \mid \lambda x.e \mid e\,e \mid \mathsf{fix}\,x(x).e \mid n \qquad\qquad\qquad \text{(expressions)}$$

$$\frac{}{\tau \le \tau}\ \text{S-Refl} \qquad \frac{\tau_2 \le \tau_3 \quad \tau_1 \le \tau_2}{\tau_1 \le \tau_3}\ \text{S-Trans} \qquad \frac{}{\tau \le \mathsf{Top}}\ \text{S-Top}$$

$$\frac{\tau_2' \le \tau_1' \quad \tau_1 \le \tau_2}{(\tau_1' \to \tau_1) \le (\tau_2' \to \tau_2)}\ \text{S-Arr} \qquad \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x\ :\ \tau}\ \text{T-Var} \qquad \frac{\Gamma, x : \tau_1 \vdash e\ :\ \tau_2}{\Gamma \vdash \lambda x.e\ :\ \tau_1 \to \tau_2}\ \text{T-Abs}$$

$$\frac{\Gamma \vdash e_1\ :\ \tau_2 \to \tau_1 \quad \Gamma \vdash e_2\ :\ \tau_2}{\Gamma \vdash e_1\,e_2\ :\ \tau_1}\ \text{T-App} \qquad \frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e\ :\ \tau_2}{\Gamma \vdash \mathsf{fix}\,f(x).e\ :\ \tau_1 \to \tau_2}\ \text{T-Fix}$$

$$\frac{}{\Gamma \vdash n\ :\ \mathsf{Nat}}\ \text{T-Const} \qquad \frac{\Gamma \vdash e\ :\ \tau \quad \tau \le \tau'}{\Gamma \vdash e\ :\ \tau'}\ \text{T-Sub}$$

Figure 1: The source language—the λ-calculus with subtyping

3.1. **The simply-typed lambda calculus with subtyping.** The syntax and typing rules for the source language are given in Figure 1. The language is the simply-typed lambda calculus extended with recursive functions ($\mathsf{fix}\,f(x).e$) and natural numbers (n). For clarity of the presentation we do not consider any primitive operations on natural numbers, but they could be seamlessly added to the language. Extending the language with additional basic types is a little bit more subtle, as discussed in Section 3.5.2. We include the type Top, to make the subtyping relation interesting. The typing and subtyping rules are standard [27, Chapter 10], where the typing environment Γ associates variables with their types and is represented as a finite set of such pairs, noted $(x : \tau)$. In the rest of the article we assume the standard notions and conventions concerning variable binding and α-conversion of terms [7].

3.2. **Coercion semantics.** The semantics of the source language is given by a translation of the typing derivations to a target language that extends the source language with explicit type coercions (and replaces Top with Unit).

3.2.1. *Target calculus.* The coercions express conversion of a term from one type to another, according to the subtyping relation. Figure 2 contains syntax, typing rules and reduction rules of the target language. The type coercions c and their typing rules correspond exactly to the subtyping rules of the source language. The grammar of terms contains explicit coercion application of the form $c\,e$.

The operational semantics of the target language takes the form of the reduction semantics, where terms are decomposed into an evaluation context and a redex. We use the standard notation $E[e]$ for plugging the term e into the context E, and similarly—$E[E']$ for plugging the context E' in E, i.e., for context composition. The grammar of evaluation contexts extends the standard call-by-value λ-calculus contexts with contexts of the form $c\,E$ that enforce evaluating an argument of a coercion before the actual conversion takes place. It can be shown that the reduction relation of Figure 2 is deterministic.

The semantics distinguishes between β-rules that perform actual computations and ι-rules that rearrange coercions. Both of them are used during program evaluation. We

$$\tau \ ::= \ \mathsf{Nat} \mid \mathsf{Unit} \mid \tau \to \tau \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(types)}$$

$$c \ ::= \ \mathsf{id} \mid c \circ c \mid \mathsf{top} \mid c \to c \qquad\qquad\qquad\qquad\qquad\qquad \text{(coercions)}$$

$$e \ ::= \ x \mid \lambda x.e \mid e\ e \mid c\ e \mid \mathsf{fix}\ x(x).e \mid n \mid \langle\rangle \qquad\qquad \text{(expressions)}$$

$$v \ ::= \ x \mid \lambda x.e \mid \mathsf{fix}\ x(x).e \mid (c \to c)\ v \mid n \mid \langle\rangle \qquad\qquad \text{(values)}$$

$$E \ ::= \ \square \mid E\ e \mid v\ E \mid c\ E \qquad\qquad\qquad\qquad \text{(evaluation contexts)}$$

$$\frac{}{\mathsf{id} :: \tau \rhd \tau}\ \text{S-Refl} \qquad\qquad \frac{c_1 :: \tau_2 \rhd \tau_3 \qquad c_2 :: \tau_1 \rhd \tau_2}{c_1 \circ c_2 :: \tau_1 \rhd \tau_3}\ \text{S-Trans}$$

$$\frac{}{\mathsf{top} :: \tau \rhd \mathsf{Unit}}\ \text{S-Top} \qquad\qquad \frac{c_1 :: \tau_2' \rhd \tau_1' \qquad c_2 :: \tau_1 \rhd \tau_2}{c_1 \to c_2 :: (\tau_1' \to \tau_1) \rhd (\tau_2' \to \tau_2)}\ \text{S-Arr}$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x \ : \ \tau}\ \text{T-Var} \qquad\qquad \frac{\Gamma, x : \tau_1 \vdash e \ : \ \tau_2}{\Gamma \vdash \lambda x.e \ : \ \tau_1 \to \tau_2}\ \text{T-Abs}$$

$$\frac{\Gamma \vdash e_1 \ : \ \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 \ : \ \tau_2}{\Gamma \vdash e_1\ e_2 \ : \ \tau_1}\ \text{T-App} \qquad\qquad \frac{c :: \tau \rhd \tau' \qquad \Gamma \vdash e \ : \ \tau}{\Gamma \vdash c\ e \ : \ \tau'}\ \text{T-CApp}$$

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e \ : \ \tau_2}{\Gamma \vdash \mathsf{fix}\ f(x).e \ : \ \tau_1 \to \tau_2}\ \text{T-Fix} \qquad\qquad \frac{}{\Gamma \vdash n \ : \ \mathsf{Nat}}\ \text{T-Const}$$

$$\frac{}{\Gamma \vdash \langle\rangle \ : \ \mathsf{Unit}}\ \text{T-Unit}$$

$$E[(\lambda x.e)\ v] \to_\beta E[e\{v/x\}] \qquad\qquad E[\mathsf{id}\ v] \to_\iota E[v]$$

$$E[(\mathsf{fix}\ f(x).e)\ v] \to_\beta E[e\{\mathsf{fix}\ f(x).e/f, v/x\}] \qquad\qquad E[(c_1 \circ c_2)\ v] \to_\iota E[c_1\ (c_2\ v)]$$

$$E[\mathsf{top}\ v] \to_\iota E[\langle\rangle]$$

$$E[(c_1 \to c_2)\ v_1\ v_2] \to_\iota E[c_2\ (v_1\ (c_1\ v_2))]$$

Figure 2: The target language—the λ-calculus with explicit coercions

say that program e terminates (written $e\!\downarrow$) when it can be reduced to a value using both sorts of reduction rules, according to the evaluation strategy determined by the evaluation contexts.

The principle behind the operational semantics of coercions, given by the ι-rules, is to structurally reduce complex coercions $c_1 \circ c_2$ and $c_1 \to c_2$ to their subcoercions c_1 and c_2, until one of the two basic coercions id or top is reached and can be trivially applied to perform the actual conversion of a value. We can see that the ι-rules for $c_1 \circ c_2$ and $c_1 \to c_2$ have an administrative rather than computational role in that erasing the coercions (defined in the expected way [13]) in the redex and in the contractum of these rules leads to the same expression. It is worth noting that terms of the form $(c \to c)\ v$ are considered

$$\llbracket \tau_1 \to \tau_2 \rrbracket \;=\; \llbracket \tau_1 \rrbracket \to \llbracket \tau_2 \rrbracket$$
$$\llbracket \mathsf{Nat} \rrbracket \;=\; \mathsf{Nat}$$
$$\llbracket \mathsf{Top} \rrbracket \;=\; \mathsf{Unit}$$

$$\mathcal{S}\llbracket \tau \leq \tau \rrbracket_{\text{S-Refl}} \;=\; \mathsf{id}$$
$$\mathcal{S}\llbracket \tau \leq \mathsf{Top} \rrbracket_{\text{S-Top}} \;=\; \mathsf{top}$$
$$\mathcal{S}\llbracket \tau_1 \leq \tau_3 \rrbracket_{\text{S-Trans}(D_1, D_2)} \;=\; \mathcal{S}\llbracket \tau_2 \leq \tau_3 \rrbracket_{D_1} \circ \mathcal{S}\llbracket \tau_1 \leq \tau_2 \rrbracket_{D_2}$$
$$\mathcal{S}\llbracket \tau_1' \to \tau_1 \leq \tau_2' \to \tau_2 \rrbracket_{\text{S-Arr}(D_1, D_2)} \;=\; \mathcal{S}\llbracket \tau_2' \leq \tau_1' \rrbracket_{D_1} \to \mathcal{S}\llbracket \tau_1 \leq \tau_2 \rrbracket_{D_2}$$

$$\mathcal{T}\llbracket x \rrbracket_{\text{T-Var}} \;=\; x$$
$$\mathcal{T}\llbracket \lambda x.e \rrbracket_{\text{T-Abs}(D)} \;=\; \lambda x.\mathcal{T}\llbracket e \rrbracket_D$$
$$\mathcal{T}\llbracket e_1\, e_2 \rrbracket_{\text{T-App}(D_1, D_2)} \;=\; \mathcal{T}\llbracket e_1 \rrbracket_{D_1} \, \mathcal{T}\llbracket e_2 \rrbracket_{D_2}$$
$$\mathcal{T}\llbracket \mathsf{fix}\, f(x).e \rrbracket_{\text{T-Fix}(D)} \;=\; \mathsf{fix}\, f(x).\mathcal{T}\llbracket e \rrbracket_D$$
$$\mathcal{T}\llbracket e \rrbracket_{\text{T-Sub}(D_1, D_2)} \;=\; \mathcal{S}\llbracket \tau \leq \tau' \rrbracket_{D_2} \, \mathcal{T}\llbracket e \rrbracket_{D_1}$$
$$\mathcal{T}\llbracket n \rrbracket_{\text{T-Const}} \;=\; n$$

Figure 3: Coercion semantics for the λ-calculus with subtyping

values, since they represent a coercion expecting another value as argument (witness the last ι-rule).

General contexts are closed terms with one hole (possibly under some binders), and are ranged over by the metavariable C. We write $\vdash C \,:\, (\Gamma; \tau_1) \;\rightsquigarrow\; \tau_2$ if for any e with $\Gamma \vdash e \,:\, \tau_1$ we have $\vdash C[e] \,:\, \tau_2$. Contextual approximation, written $\Gamma \vdash e_1 \;\precsim_{ctx}\; e_2 \,:\, \tau$, means that for any context C and type τ', such that $\vdash C \,:\, (\Gamma; \tau) \;\rightsquigarrow\; \tau'$ if $C[e_1]$ terminates, then so does $C[e_2]$. If $\Gamma \vdash e_1 \;\precsim_{ctx}\; e_2 \,:\, \tau$ and $\Gamma \vdash e_2 \;\precsim_{ctx}\; e_1 \,:\, \tau$, then we say that e_1 and e_2 are contextually equivalent. It is this notion of program equivalence that we take to express coherence of the coercion semantics and characterize with logical relations later on in this section.

3.2.2. *Translation.* The coercion semantics of the source language is given in Figure 3. The function $\mathcal{S}\llbracket.\rrbracket$ translates subtyping proofs into coercions, and function $\mathcal{T}\llbracket.\rrbracket$ translates typing derivations into terms of the target language, whereas types are translated by the function $\llbracket.\rrbracket$, which we extend to a point-wise translation of typing environments. Both $\mathcal{S}\llbracket.\rrbracket$ and $\mathcal{T}\llbracket.\rrbracket$ are defined by structural recursion on derivation trees, where the structure of the tree D is given by the second argument, consisting of the name of the final rule in the derivation D and the immediate subtrees of D. For example, in the equation

$$\mathcal{T}\llbracket e \rrbracket_{\text{T-Sub}(D_1, D_2)} \;=\; \mathcal{S}\llbracket \tau \leq \tau' \rrbracket_{D_2} \, \mathcal{T}\llbracket e \rrbracket_{D_1}$$

T-Sub(D_1, D_2) represents the tree

$$\frac{\begin{array}{cc} D_1 & D_2 \\ \Gamma \vdash e \; : \; \tau & \tau \leq \tau' \end{array}}{\Gamma \vdash e \; : \; \tau'} \text{ T-SUB}$$

The translation functions themselves are rather straightforward; their role is to replace subtyping derivations with coercions applied to expressions being coerced from one type to another. The soundness of the translation functions is ensured by the following lemma.

Lemma 3.1. *Coercion semantics preserves types.*

(1) *If $D :: \tau_1 \leq \tau_2$ then $\mathcal{S}[\![\tau_1 \leq \tau_2]\!]_D :: [\![\tau_1]\!] \triangleright [\![\tau_2]\!]$.*

(2) *If $D :: \Gamma \vdash e \; : \; \tau$ then $[\![\Gamma]\!] \vdash \mathcal{T}[\![e]\!]_D \; : \; [\![\tau]\!]$.*

The following example demonstrates the translation function and the coercions at work.

Example 3.2. Consider the program $(\lambda f.f\,1)\,(\lambda x.x)$ in the source language. Let D be the derivation where variable f has type $\mathsf{Nat} \to \mathsf{Top}$ and the type for expression $\lambda x.x$ is derived in the following way:

$$\frac{\dfrac{\dfrac{}{x : \mathsf{Top} \vdash x \; : \; \mathsf{Top}} \text{ T-VAR}}{\vdash \lambda x.x \; : \; \mathsf{Top} \to \mathsf{Top}} \text{ T-ABS} \quad \dfrac{\dfrac{}{\mathsf{Nat} \leq \mathsf{Top}} \text{ S-TOP} \quad \dfrac{}{\mathsf{Top} \leq \mathsf{Top}} \text{ S-REFL}}{\mathsf{Top} \to \mathsf{Top} \leq \mathsf{Nat} \to \mathsf{Top}} \text{ S-ARR}}{\vdash \lambda x.x \; : \; \mathsf{Nat} \to \mathsf{Top}} \text{ T-SUB}$$

The coercion translation of such derivation puts a coercion application in a place, where the subsumption rule was used in the type derivation:

$$\mathcal{T}[\![(\lambda f.f\,1)\,(\lambda x.x)]\!]_D = (\lambda f.f\,1)\,((\mathsf{top} \to \mathsf{id})\,(\lambda x.x))$$

The result of the translation can be reduced using β and ι-reductions:

$$
\begin{array}{ll}
(\lambda f.f\,1)\,((\mathsf{top} \to \mathsf{id})\,(\lambda x.x)) & \to_\beta \\
(\mathsf{top} \to \mathsf{id})\,(\lambda x.x)\,1 & \to_\iota \\
\mathsf{id}\,((\lambda x.x)\,(\mathsf{top}\,1)) & \to_\iota \\
\mathsf{id}\,((\lambda x.x)\,\langle\rangle) & \to_\beta \\
\mathsf{id}\,\langle\rangle & \to_\iota \\
\langle\rangle &
\end{array}
$$

First, we perform β-reduction, since $(\mathsf{top} \to \mathsf{id})\,(\lambda x.x)$ is a value. Thereafter, the arrow coercion $(\mathsf{top} \to \mathsf{id})$ gets two arguments, so it can be ι-reduced by distributing coercions top and id between the argument and the result of the identity function. Then we continue the reduction using the call-by-value strategy. Note that both β- and ι-reductions are needed during the evaluation. □

The problem of coherence is illustrated in the next example.

Example 3.3. Coercion semantics can produce distant results for different typing derivations, even in such simple calculus as presented in this section. Consider the fixed-point operator $\mathsf{fix}\,y(f).\lambda x.f\,(y\,f)\,x$ expressed using recursive functions. Assuming $\tau \leq \tau'$, one possible type of such an expression is $((\tau \to \tau') \to \tau \to \tau) \to \tau \to \tau$. Let D_1 be a derivation where variable y has the same type as whole expression, and we coerce only the subexpression $(y\,f)$ from type $\tau \to \tau$ to $\tau \to \tau'$. On the other hand, let D_2 be a derivation where the

whole expression is coerced from the type $((\tau \to \tau) \to \tau \to \tau) \to \tau \to \tau$, which is derived directly. As a result of the coercion semantics for the derivations D_1 and D_2 we get the following programs in the target calculus:

$$e_1 \; := \; \mathcal{T}[\![\mathsf{fix}\; y(f).\lambda x.f \;(y\; f)\; x]\!]_{D_1} = \mathsf{fix}\; y(f).\lambda x.f \;((\mathsf{id} \to c) \;(y\; f))\; x$$

$$e_2 \; := \; \mathcal{T}[\![\mathsf{fix}\; y(f).\lambda x.f \;(y\; f)\; x]\!]_{D_2} = (((\mathsf{id} \to c) \to \mathsf{id}) \to \mathsf{id}) \;(\mathsf{fix}\; y(f).\lambda x.f \;(y\; f)\; x),$$

where $c :: \tau \triangleright \tau'$ is a result of translating the proof of $\tau \leq \tau'$. These terms are different values and it is hard to find any reasonable equational theory to equate them. However, as a consequence of next sections, they are contextually equivalent. Indeed, they exhibit similar behavior when applied to two values f and v. We can perform three β-reductions starting from the term $e_1\; f\; v$.

$$
\begin{array}{ll}
e_1\; f\; v & \to_\beta^2 \\
f \;((\mathsf{id} \to c) \;(e_1\; f))\; v & \to_\beta \\
f \;((\mathsf{id} \to c) \;(\lambda x.f \;((\mathsf{id} \to c) \;(e_1\; f))\; x))\; v &
\end{array}
$$

Reducing the term $e_2\; f\; v$ requires some extra ι-reductions. Let $e_0 = \mathsf{fix}\; y(f).\lambda x.f \;(y\; f)\; x$ and $f_0 = ((\mathsf{id} \to c) \to \mathsf{id})\; f$. We have the following reduction path.

$$
\begin{array}{ll}
e_2\; f\; v & \to_\iota \\
\mathsf{id} \;(e_0\; f_0)\; v & \to_\beta \\
\mathsf{id} \;(\lambda x.f_0 \;(e_0\; f_0)\; x)\; v & \to_\iota \to_\beta \\
f_0 \;(e_0\; f_0)\; v & \to_\beta \\
f_0 \;(\lambda x.f_0 \;(e_0\; f_0)\; x)\; v & \to_\iota \\
\mathsf{id} \;(f \;((\mathsf{id} \to c) \;(\lambda x.f_0 \;(e_0\; f_0)\; x)))\; v &
\end{array}
$$

In both cases we obtained a term of the form $f \;((\mathsf{id} \to c)\; e)\; v$ (modulo insignificant identity coercions), where e is a result of applying e_i to f. $\qquad\square$

3.3. **Logical relations.** In order to reason about contextual equivalence in the target language, we define logical relations (Figure 4). Relations are expressed in the LSLR logic described in Section 2, so they are implicitly step-indexed.

We call these relations heterogeneous because they are parameterized by two types, one for each of the arguments. This property is important for our coherence proof, since it makes it possible to relate the results of the translation of two typing derivations which assign different types to the same term, e.g., as in Example 3.3. When both types τ_1 and τ_2 are Nat or both are arrow types, the value relation $\mathcal{V}[\![\tau_1; \tau_2]\!]$ is standard. Two values are related for type Nat if they are the same constant, and two functions are related when they map related arguments to related results. Because we have many kinds of values representing functions, we follow Pitts and Stark [30] in using an application for testing functions, instead of a substitution (as in, e.g., [19, 2]). The most interesting are the cases when type parameters of the relation are different. When one of these types is Unit, then any values are in the relation, because we do not expect them to carry any information— Unit is the result of translating the Top type. In such a case we do not even require that related values are of the kind described by their corresponding type. We can do so since

$$
\begin{aligned}
(v_1, v_2) \in \mathcal{V}[\![\mathsf{Nat}; \mathsf{Nat}]\!] &\iff \exists n, v_1 = v_2 = n \\
(v_1, v_2) \in \mathcal{V}[\![\tau_1' \to \tau_1; \tau_2' \to \tau_2]\!] &\iff \forall (a_1, a_2) \in \mathcal{V}[\![\tau_1'; \tau_2']\!].(v_1\, a_1, v_2\, a_2) \in \mathcal{E}[\![\tau_1; \tau_2]\!] \\
(v_1, v_2) \in \mathcal{V}[\![\mathsf{Unit}; \tau_2]\!] &\iff \top \\
(v_1, v_2) \in \mathcal{V}[\![\tau_1; \mathsf{Unit}]\!] &\iff \top \\
(v_1, v_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!] &\iff \bot \qquad \text{otherwise} \\[4pt]
(e_1, e_2) \in \mathcal{E}[\![\tau_1; \tau_2]\!] &\iff \forall (E_1, E_2) \in \mathcal{K}[\![\tau_1; \tau_2]\!].E_1[e_1] \precsim E_2[e_2] \\[4pt]
(E_1, E_2) \in \mathcal{K}[\![\tau_1; \tau_2]\!] &\iff \forall (v_1, v_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!].E_1[v_1] \precsim E_2[v_2] \\[4pt]
k \models e_1 \precsim e_2 &\iff e_1\!\downarrow^k \implies e_2\!\downarrow \\[4pt]
(\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma_1; \Gamma_2]\!] &\iff \forall x, (\gamma_1(x), \gamma_2(x)) \in \mathcal{V}[\![\Gamma_1(x); \Gamma_2(x)]\!] \\[4pt]
\Gamma_1; \Gamma_2 \vdash e_1 \precsim_{log} e_2 \ : \ \tau_1; \tau_2 &\iff \forall (\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma_1; \Gamma_2]\!].(e_1 \gamma_1, e_2 \gamma_2) \in \mathcal{E}[\![\tau_1; \tau_2]\!]
\end{aligned}
$$

Figure 4: Logical relations for the λ-calculus with explicit coercions

in the calculus each coercion applied to a value is or reduces to a value. In calculi without this property we have to be more careful (see Section 3.5.2). The logical relation is empty for different types which are not Unit.

The relation $\mathcal{E}[\![\tau_1; \tau_2]\!]$ for closed terms is defined by biorthogonality. Two terms are related if they behave the same in related contexts, and contexts are related (relation $\mathcal{K}[\![\tau_1; \tau_2]\!]$) if they yield the same observations when plugged with related values. Yielding the same observations (relation $\precsim$) is defined for each step-index separately: $e_1 \precsim e_2$ is valid at k iff termination of e_1 using at most k β-steps and any number of ι-steps (written $e_1\!\downarrow^k$), implies termination of e_2 in any number of β-steps and ι-steps. This interpretation is monotone, so the relation $\precsim$ can be added to the LSLR logic.

In order to extend the relation $\mathcal{E}[\![\tau_1; \tau_2]\!]$ to open terms we first define a relation $\mathcal{G}[\![\Gamma_1; \Gamma_2]\!]$ on substitutions (mapping variables to closed values) parameterized by a pair of typing environments. Then we say that two open terms are related (written $\Gamma_1; \Gamma_2 \vdash e_1 \precsim_{log} e_2 \ : \ \tau_1; \tau_2$) when every pair of related closing substitutions makes them related.

Notice that we do not assume that related terms have valid types. Our relations may include some "garbage", e.g., $(1, \lambda x.x) \in \mathcal{V}[\![\mathsf{Unit}; \mathsf{Nat}]\!]$, but it is non-problematic. One can mechanically prune these relations to well-typed terms, but this change complicates formalization and we did not find it useful.

The relation $\precsim$ is preserved by reductions in the following sense, where the third assertion expresses an elimination rule of the later modality that is crucial in the subsequent proofs.

Lemma 3.4. *The following assertions hold:*
 (1) If $e_1 \to_\iota e_1'$ and $e_1' \precsim e_2$ then $e_1 \precsim e_2$.
 (2) If $e_2 \to_\iota e_2'$ and $e_1 \precsim e_2'$ then $e_1 \precsim e_2$.
 (3) If $e_1 \to_\beta e_1'$ and $\triangleright e_1' \precsim e_2$ then $e_1 \precsim e_2$.
 (4) If $e_2 \to_\beta e_2'$ and $e_1 \precsim e_2'$ then $e_1 \precsim e_2$.

The proof of soundness of the logical relations follows closely the standard technique for biorthogonal logical relations [30, 19]. First, we need to show compatibility lemmas, which state that the relation is preserved by every language construct.

Lemma 3.5 (Compatibility). *The following assertions hold:*

(1) *if* $(x : \tau_1) \in \Gamma_1$ *and* $(x : \tau_2) \in \Gamma_2$ *then* $\Gamma_1; \Gamma_2 \vdash x \precsim_{log} x \ : \ \tau_1; \tau_2$;

(2) *if* $(\Gamma_1, x : \tau_1'); (\Gamma_2, x : \tau_2') \vdash e_1 \precsim_{log} e_2 \ : \ \tau_1; \tau_2$
   then $\Gamma_1; \Gamma_2 \vdash \lambda x.e_1 \precsim_{log} \lambda x.e_2 \ : \ \tau_1' \to \tau_1; \tau_2' \to \tau_2$;

(3) *if* $\Gamma_1; \Gamma_2 \vdash e_1 \precsim_{log} e_2 \ : \ \tau_1' \to \tau_1; \tau_2' \to \tau_2$ *and* $\Gamma_1; \Gamma_2 \vdash e_1' \precsim_{log} e_2' \ : \ \tau_1'; \tau_2'$
   then $\Gamma_1; \Gamma_2 \vdash e_1 e_1' \precsim_{log} e_2 e_2' \ : \ \tau_1; \tau_2$;

(4) *if* $(\Gamma_1, f : \tau_1' \to \tau_1, x : \tau_1'); (\Gamma_2, f : \tau_2' \to \tau_2, x : \tau_2') \vdash e_1 \precsim_{log} e_2 \ : \ \tau_1; \tau_2$
   then $\Gamma_1; \Gamma_2 \vdash \mathsf{fix}\ f(x).e_1 \precsim_{log} \mathsf{fix}\ f(x).e_2 \ : \ \tau_1' \to \tau_1; \tau_2' \to \tau_2$;

(5) *we have* $\Gamma_1; \Gamma_2 \vdash n \precsim_{log} n \ : \ \mathsf{Nat}; \mathsf{Nat}$;

(6) *we have* $\Gamma_1; \Gamma_2 \vdash \langle\rangle \precsim_{log} \langle\rangle \ : \ \mathsf{Unit}; \mathsf{Unit}$.

Proof. The proof is standard and directed by the definition of logical relations. We only show the proof for the case with recursive functions, where step indexing simplifies reasoning. Assume $(\Gamma_1, f : \tau_1' \to \tau_1, x : \tau_1'); (\Gamma_2, f : \tau_2' \to \tau_2, x : \tau_2') \vdash e_1 \precsim_{log} e_2 \ : \ \tau_1; \tau_2$ (*). Since $\mathsf{fix}\ f(x).e_1$ and $\mathsf{fix}\ f(x).e_2$ are values, it suffices to show that for every substitutions $(\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma_1; \Gamma_2]\!]$ we have $(\mathsf{fix}\ f(x).e_1\gamma_1, \mathsf{fix}\ f(x).e_2\gamma_2) \in \mathcal{V}[\![\tau_1' \to \tau_1; \tau_2' \to \tau_2]\!]$. Now, we use the Löb rule to assume the induction hypothesis[1] $(\mathsf{fix}\ f(x).e_1\gamma_1, \mathsf{fix}\ f(x).e_2\gamma_2) \in \triangleright\mathcal{V}[\![\tau_1' \to \tau_1; \tau_2' \to \tau_2]\!]$ (**). Unfolding the definition of the relation $\mathcal{V}[\![\tau_1' \to \tau_1; \tau_2' \to \tau_2]\!]$, we need to show that for every $(v_1, v_2) \in \mathcal{V}[\![\tau_1'; \tau_2']\!]$ and $(E_1, E_2) \in \mathcal{K}[\![\tau_1; \tau_2]\!]$, we have $E_1[(\mathsf{fix}\ f(x).e_1\gamma_1)\ v_1] \precsim E_2[(\mathsf{fix}\ f(x).e_2\gamma_2)\ v_2]$. By Lemma 3.4 (used twice), it suffices to prove that $\triangleright E_1[e_1\gamma_1\{\mathsf{fix}\ f(x).e_1\gamma_1/f, v_1/x\}] \precsim E_2[e_2\gamma_2\{\mathsf{fix}\ f(x).e_2\gamma_2/f, v_2/x\}]$. Using the later introduction rule, we can remove the later operator both in the goal and in the assumption (**). Now, we can show that the substitutions $\gamma_1\{\mathsf{fix}\ f(x).e_1\gamma_1/f, v_1/x\}$ and $\gamma_2\{\mathsf{fix}\ f(x).e_2\gamma_2/f, v_2/x\}$ are related, hence using (*) we conclude the proof. $\square$

The only compatibility lemma specific to our relations is the lemma for coercion application. Since the subsumption rule is not syntax-directed, we expect from the coercions to preserve the logical relation, even when they are applied to only one of the related expressions.

Lemma 3.6 (Coercion compatibility). *The logical relation is preserved by coercion application:*

(1) *If* $c :: \tau_1 \triangleright \tau_2$ *and* $\Gamma_1; \Gamma_2 \vdash e_1 \precsim_{log} e_2 \ : \ \tau_1; \tau_0$ *then* $\Gamma_1; \Gamma_2 \vdash c\ e_1 \precsim_{log} e_2 \ : \ \tau_2; \tau_0$.

(2) *If* $c :: \tau_1 \triangleright \tau_2$ *and* $\Gamma_1; \Gamma_2 \vdash e_1 \precsim_{log} e_2 \ : \ \tau_0; \tau_1$ *then* $\Gamma_1; \Gamma_2 \vdash e_1 \precsim_{log} c\ e_2 \ : \ \tau_0; \tau_2$.

Proof. We prove both cases by induction on the typing derivation of the coercion c. $\square$

Compatibility lemmas allow us to show the fundamental property of the logical relations, stating that the logical relation is reflexive for well-typed terms.

Theorem 3.7 (Fundamental property). *If* $\Gamma \vdash e \ : \ \tau$ *then* $\Gamma; \Gamma \vdash e \precsim_{log} e \ : \ \tau; \tau$.

Proof. By induction on the derivation $\Gamma \vdash e \ : \ \tau$. In each case we apply the corresponding compatibility lemma. $\square$

---

[1]This reasoning step corresponds to the induction on indices.

The soundness of the logical relations is a direct consequence of the following properties: *precongruence* which says that the logical relation is preserved by any well-typed context, and *adequacy* which says that related programs have the same observable behavior.

Lemma 3.8 (Precongruence)**.** *If* $\vdash C \ : \ (\Gamma; \tau) \ \rightsquigarrow \ \tau_0$ *and* $\Gamma; \Gamma \vdash e_1 \precsim_{log} e_2 \ : \ \tau; \tau$ *then* $(C[e_1], C[e_2]) \in \mathcal{E}[\![\tau_0; \tau_0]\!]$.

Proof. By induction on the derivation of context typing, using the appropriate compatibility lemma in each case. For contexts containing subterms we also need the fundamental property. For the empty context we use the empty substitution, since the empty substitutions are in relation $\mathcal{G}[\![\varnothing; \varnothing]\!]$. $\square$

Lemma 3.9 (Adequacy)**.** *If* $(e_1, e_2) \in \mathcal{E}[\![\tau; \tau]\!]$ *then* $e_1 \precsim e_2$.

Proof. Let us show $\square[e_1] \precsim \square[e_2]$. Using the assertion $(e_1, e_2) \in \mathcal{E}[\![\tau; \tau]\!]$, it suffices to show $(\square, \square) \in \mathcal{K}[\![\tau; \tau]\!]$, which is trivial, since values always terminate. $\square$

Theorem 3.10 (Soundness)**.** *If* $k \models \Gamma; \Gamma \vdash e_1 \precsim_{log} e_2 \ : \ \tau; \tau$ *holds for every* k, *then* $\Gamma \vdash e_1 \precsim_{ctx} e_2 \ : \ \tau$.

Proof. Suppose $\vdash C \ : \ (\Gamma; \tau) \ \rightsquigarrow \ \tau_0$ and $C[e_1]\downarrow$, we need to show $C[e_2]\downarrow$. By Lemma 3.8 and Lemma 3.9 we know $k \models C[e_1] \precsim C[e_2]$ for every k. Taking k to be the number of steps in which $C[e_1]$ terminates, we have that $C[e_2]$ also terminates, by the definition of $\precsim$. $\square$

3.4. **Coherence of the coercion semantics.** Having established soundness of the logical relations, we are in a position to prove the main coherence lemma, phrased in terms of the logical relations, and the coherence theorem.

Lemma 3.11. *If* $D_i :: \Gamma_i \vdash e \ : \ \tau_i$ *for* $i = 1, 2$ *are two typing derivations for the same term* e *of the source language, then* $[\![\Gamma_1]\!]; [\![\Gamma_2]\!] \vdash \mathcal{T}[\![e]\!]_{D_1} \precsim_{log} \mathcal{T}[\![e]\!]_{D_2} \ : \ [\![\tau_1]\!]; [\![\tau_2]\!]$.

Proof. The proof follows by induction on the structure of both derivations D_1 and D_2. At least one of these derivations is decreased in every case. When one of the derivations starts with the subsumption rule (T-SUB), we apply Lemma 3.6. The coercion that we get after the translation is well-typed by Lemma 3.1. In other cases we just apply the appropriate compatibility lemma. $\square$

Theorem 3.12 (Coherence)**.** *If* D_1 *and* D_2 *are derivations of the same typing judgment* $\Gamma \vdash e \ : \ \tau$, *then* $[\![\Gamma]\!] \vdash \mathcal{T}[\![e]\!]_{D_1} \precsim_{ctx} \mathcal{T}[\![e]\!]_{D_2} \ : \ [\![\tau]\!]$.

Proof. Immediately from Lemma 3.11 and Theorem 3.10. $\square$

3.5. **Variants.** In this section we briefly discuss some possible extensions of the results presented so far.

3.5.1. *Coercions as λ-terms.* The coercion semantics described here translates the source language into the language with explicit coercions. We chose coercions to be a separate syntactic category, because we found it very convenient, especially for proving Lemma 3.6. However, one can define a coercion semantics which translates subtyping proofs directly to λ-expressions. Our result can be easily extended for such a translation. Let $|e|$ be a term e with all the coercions replaced by the corresponding expressions. To prove that for any contextually equivalent terms e_1 and e_2 in the language with coercions, terms $|e_1|$ and $|e_2|$ are contextually equivalent in the language without coercions, we need three simple facts that can be easily verified:

(1) every well-typed term in the language without coercions is well typed in the language with coercions,
(2) term e terminates iff $|e|$ terminates,
(3) if context C does not contain coercions then $C[|e|] = |C[e]|$.

3.5.2. *Multiple base types.* In this presentation we consider languages with only one base type. Adding more base types and some subtyping between them will not change the general shape of the proof, but defining logical relations for such a case is a little trickier.

Let $\mathcal{B}$ be a set of base types and $\leq_b$ be a subtyping relation on them. Assume for every $b \in \mathcal{B}$ we have set $\mathcal{V}_b$ of constants of type b. These constants are values in both source and target calculi. Additionally, for each $b \leq_b b'$ we have a corresponding coercion $\mathsf{c}_b^{b'}$ and a function $f_b^{b'} : \mathcal{V}_b \to \mathcal{V}_{b'}$. The ι-rule for a coercion $\mathsf{c}_b^{b'}$ is defined as follows: if $v \in \mathcal{V}_b$, then $E[\mathsf{c}_b^{b'} v] \to_\iota E[f_b^{b'}(v)]$.

The coherence of coercion semantics requires coherence on base types. More precisely, we assume the following properties:

(1) relation $\leq_b$ is reflexive and transitive;
(2) for each $b \in \mathcal{B}$ the function f_b^b is an identity;
(3) if $b_1 \leq_b b_2$ and $b_2 \leq_b b_3$ then $f_{b_2}^{b_3} \circ f_{b_1}^{b_2} = f_{b_1}^{b_3}$.

We would stipulate that two values v_1 and v_2 are related for base types b_1 and b_2 iff for every common supertype b of b_1 and b_2, coercing v_1 and v_2 to b yields the same constant:

$$(v_1, v_2) \in \mathcal{V}[\![b_1; b_2]\!] \iff v_1 \in \mathcal{V}_{b_1} \wedge v_2 \in \mathcal{V}_{b_2} \wedge \left(\forall b. b_1 \leq_b b \wedge b_2 \leq_b b \Rightarrow f_{b_1}^b(v_1) = f_{b_2}^b(v_2) \right)$$

Note that since relation $\leq_b$ is reflexive, for $b_1 = b_2$ this definition yields the identity relation on values of a base type b_1, the same as in Section 3.3.

Moreover, we have to be more careful with defining the logical relation for the Unit type. The proof of Lemma 3.6 relies on the fact that for every $c :: \tau_1 \triangleright \tau_2$ and $(v_1, v_2) \in \mathcal{V}[\![\tau; \tau_1]\!]$, the expression $c\, v_2$ either is or reduces to a value. To ensure that property, the relation for Unit and base type b should relate any value with any value of type b:

$$(v_1, v_2) \in \mathcal{V}[\![\mathsf{Unit}; b]\!] \iff v_2 \in \mathcal{V}_b$$
$$(v_1, v_2) \in \mathcal{V}[\![b; \mathsf{Unit}]\!] \iff v_1 \in \mathcal{V}_b$$

4. Coherence of a CPS translation of control-effect subtyping

In this section we show that the results presented in Section 3 can be adapted to a considerably more complex calculus—a calculus of delimited control with control-effect subtyping [25].

4.1. **Delimited continuations, informally.** Control operators for delimited continuations, introduced independently by Felleisen [20] and by Danvy and Filinski [15], allow the programmer to delimit the current context of computation and to abstract such a delimited context as a first-class value. They have found numerous applications (see, e.g., [9] for a list), including Filinski's result showing that all computational effects are expressible in terms of the delimited-control operators shift and reset [21].

The calculus of delimited control studied in this work is the call-by-value λ-calculus extended with natural numbers, recursion, and the control operators shift_0 ($\mathcal{S}_0$) and reset_0 ($\langle \cdot \rangle$)—a variant of shift and reset [15]. These operators have recently enjoyed an upsurge of interest due to their considerable expressive power and connections with the $\lambda\mu$-calculi [25, 26, 24, 17, 16, 29]. Both the calculus and the coercion semantics we consider in the rest of the article are based on the type system and the CPS translation introduced by Materzok and the first author [25].

We will define the semantics of the calculus by a CPS translation to a target calculus endowed with a reduction semantics, but if we were to directly give reduction rules for shift_0 and reset_0, they would be [25]:

$$F[\langle E[\mathcal{S}_0 x.e]\rangle] \quad \rightarrow \quad F[e\{\lambda y.\langle E[y]\rangle/x\}]$$
$$F[\langle v \rangle] \quad \rightarrow \quad F[v]$$

where E is a pure call-by-value evaluation context representing the current delimited continuation (delimited by $\langle \cdot \rangle$ and captured by $\mathcal{S}_0$), and F is a metacontext, i.e., a general evaluation context that consists of a number of pure evaluation contexts separated by control delimiters.

Let us consider a simple example

$$1 + \langle 10 + \mathcal{S}_0 k.100 + k\,(k\,0)\rangle$$

that represents an arithmetic expression over natural numbers. Here is how this expression is evaluated according to the reduction rules (we assume the standard reduction rules for $+$ and the call-by-value β-reduction):

$$
\begin{array}{lll}
1 + \langle 10 + \mathcal{S}_0 k.100 + k\,(k\,0)\rangle & \rightarrow & (1) \\
1 + 100 + (\lambda y.\langle 10 + y\rangle)\,((\lambda y.\langle 10 + y\rangle)\,0) & \rightarrow^3 & (2) \\
1 + 100 + (\lambda y.\langle 10 + y\rangle)\,10 & \rightarrow^3 & (3) \\
1 + 100 + 20 & \rightarrow^2 & (4) \\
121
\end{array}
$$

In step (1) the delimited continuation $\lambda y.\langle 10 + y\rangle$ is captured and substituted for k. In step (2) the captured continuation is applied to 0, and the result of this application, the value 10, is returned—the captured continuation is functional in that it is composed with

the remaining computation, rather than abortive as it would be the case for call/cc. In step
(3) the captured continuation is applied to the passed value, and again it returns a value
to the remaining computation that consists in simple arithmetic, carried out in step (4).

In contrast to shift, shift_0 is a control operator that can explore and reorganize an
arbitrary portion of the metacontext. Here is an example:

$$
\begin{aligned}
\langle 1 + \langle 10 \times \mathcal{S}_0 k_1 . \mathcal{S}_0 k_2 . k_1 \ (k_2 \ 0)\rangle\rangle &\quad \rightarrow &\quad (1)\\
\langle 1 + \mathcal{S}_0 k_2 . (\lambda y. \langle 10 \times y\rangle) \ (k_2 \ 0)\rangle &\quad \rightarrow &\quad (2)\\
(\lambda y. \langle 10 \times y\rangle) \ ((\lambda y. \langle 1 + y\rangle) \ 0) &\quad \rightarrow^3 &\quad (3)\\
(\lambda y. \langle 10 \times y\rangle) \ 1 &\quad \rightarrow^3 &\quad (4)\\
10 & &
\end{aligned}
$$

In step (1) k_1 is bound to the captured continuation representing multiplication by 10.
In step (2) k_2 is bound to the captured continuation representing incrementation by 1.
In step (3) and (4) 0 is first incremented and the result is then multiplied by 10—the
order of these operations is reversed compared to their occurrence in the initial expression,
which is achieved by repeatedly shifting delimited continuations in steps (1) and (2) and by
composing them in the desired order.

Expressive type systems for delimited continuations are built around the idea that
the type of an expression depends on the type of a context in which the expression is
immersed [14, 8]. For example, the expression

$$
42 + \mathcal{S}_0 k.k
$$

is well typed in such systems. Given a context E that can be plugged with a value of type
Nat and that returns a value of some type τ, assuming that E does not trigger control effects
when plugged with a value, the evaluation of this expression would return a value of type
$\mathsf{Nat} \to \tau$. Then, given a metacontext F that expects a value of that type, the expression

$$
F[\langle E[42 + \mathcal{S}_0 k.k]\rangle]
$$

would be well typed. We observe that the answer type τ of the context E differs from
$\mathsf{Nat} \to \tau$, the type expected by the metacontext F. Such answer-type modification is
characteristic of type systems à la Danvy and Filinski [14] and is necessary to exploit the
expressive power of typed delimited-control operators [6, 8, 25].

Since the control operator shift_0 is allowed to explore the metacontext arbitrarily deep,
the type of the expression should actually depend not only on the type of its nearest enclosing
context, but also on the types of the remaining contexts that form the metacontext. For
example, the type of the term

$$
\mathcal{S}_0 k_1 . \mathcal{S}_0 k_2 . k_1 \ (k_2 \ 42)
$$

would express that given a context E_1 expecting a value of type τ and with answer type
τ', and a context E_2 expecting a value of type Nat and with answer type τ, the type of the
expression

$$
\langle E_2[\langle E_1[\mathcal{S}_0 k_1 . \mathcal{S}_0 k_2 . k_1 \ (k_2 \ 42)]\rangle]\rangle
$$

$$\tau \quad ::= \quad \mathsf{Nat} \mid \tau \to T \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(pure types)}$$

$$T \quad ::= \quad \tau \mid \tau[T]T \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(types)}$$

$$e \quad ::= \quad x \mid \lambda x.e \mid e\,e \mid \mathsf{fix}\,x(x).e \mid \mathcal{S}_0 x.e \mid \langle e \rangle \mid n \qquad\qquad \text{(expressions)}$$

$$\frac{}{T \leq T}\ \text{S-Refl} \qquad \frac{T_2 \leq T_3 \quad T_1 \leq T_2}{T_1 \leq T_3}\ \text{S-Trans} \qquad \frac{\tau_2 \leq \tau_1 \quad T_1 \leq T_2}{(\tau_1 \to T_1) \leq (\tau_2 \to T_2)}\ \text{S-Arr}$$

$$\frac{\tau_1 \leq \tau_2 \quad T_2 \leq T_1 \quad U_1 \leq U_2}{\tau_1[T_1]U_1 \leq \tau_2[T_2]U_2}\ \text{S-Cons} \qquad \frac{T_1 \leq T_2}{\tau \leq \tau[T_1]T_2}\ \text{S-Lift}$$

$$\frac{\Gamma \vdash e\ :\ T \quad T \leq U}{\Gamma \vdash e\ :\ U}\ \text{T-Sub} \qquad \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x\ :\ \tau}\ \text{T-Var} \qquad \frac{\Gamma, x : \tau \vdash e\ :\ T}{\Gamma \vdash \lambda x.e\ :\ \tau \to T}\ \text{T-Abs}$$

$$\frac{\Gamma \vdash e_1\ :\ \tau \to T \quad \Gamma \vdash e_2\ :\ \tau}{\Gamma \vdash e_1\,e_2\ :\ T}\ \text{T-PApp}$$

$$\frac{\Gamma \vdash e_1\ :\ (\tau_2 \to \tau_1[U_4]U_3)[U_2]U_1 \quad \Gamma \vdash e_2\ :\ \tau_2[U_3]U_2}{\Gamma \vdash e_1\,e_2\ :\ \tau_1[U_4]U_1}\ \text{T-App}$$

$$\frac{\Gamma, f : \tau \to T, x : \tau \vdash e\ :\ T}{\Gamma \vdash \mathsf{fix}\,f(x).e\ :\ \tau \to T}\ \text{T-Fix} \qquad \frac{}{\Gamma \vdash n\ :\ \mathsf{Nat}}\ \text{T-Const}$$

$$\frac{\Gamma, x : \tau \to T \vdash e\ :\ U}{\Gamma \vdash \mathcal{S}_0 x.e\ :\ \tau[T]U}\ \text{T-Sft} \qquad \frac{\Gamma \vdash e\ :\ \tau[\tau]T}{\Gamma \vdash \langle e \rangle\ :\ T}\ \text{T-Rst}$$

Figure 5: The source language—the λ-calculus with delimited control and effect subtyping

is τ'. In fact the types in this example could be more complex and express, e.g., that both E_1 and E_2 are effectful.

The calculus considered in the rest of this article was built around the idea of types describing the relevant portion of the metacontext, where, under some conditions, an expression that imposes certain requirements on the metacontext can be used with a metacontext of which more is known or assumed [25]. For example, a pure expression such as the constant 42 can be plugged in a pure evaluation context expecting values of type Nat, but also in arbitrarily complex metacontexts that have the inner-most context accepting values of type Nat. Coercions between types describing metacontexts are possible thanks to the subtyping relation that lies at the heart of the calculus presented in Section 4.2.

4.2. **The lambda calculus with delimited control and effect subtyping.** The syntax and typing rules of the calculus of delimited control are shown in Figure 5. Our presentation differs slightly from the original one [25], but only in some inessential details, and the two type systems are equally expressive. Types are either pure (τ) or effect annotated ($\tau[T_1]T_2$). A type $\tau[T_1]T_2$ describes a computation of type τ that when run in a delimited context with an answer type T_1, yields a computation described by T_2. For instance,

the expression $\mathcal{S}_0 k_1.\mathcal{S}_0 k_2.k_1 \, (k_2 \, 42)$, considered in the previous section, can be given type Nat[Nat](Nat[Nat]Nat), whereas $\mathcal{S}_0 k.42$ can be given type Nat[Nat[Nat]Nat]Nat.

The calculus comprises the simply typed lambda calculus (rules T-Sub, T-Var, T-Abs, T-PApp) with the standard subtyping rules (S-Refl, S-Trans, S-Arr), general recursion (T-Fix), natural numbers (T-Const), and the remaining rules that describe control effects at the level of types. First, the rule T-Sft corresponds to the operational behavior of $\mathcal{S}_0 x.e$: assuming that e, possibly using a captured context $\langle E \rangle$ of type $\tau \to T$, can be plugged into a metacontext F of type U, it is sound to use the whole expression with the metacontext $F[\langle E \rangle]$ of type $\tau[T]U$. Accordingly, the rule T-Rst expresses that $\langle e \rangle$ can be used in a metacontext F of type T provided e can be plugged in the metacontext $F[\langle \Box \rangle]$ of type $\tau[\tau]T$. Then, the rule T-App describes an effectful application $e_1 \, e_2$, where each of the computation e_1, e_2, and the application itself can manipulate the metacontext. This is a standard rule found already in Danvy and Filinski's type-and-effect system for shift and reset [14], where it was derived from the CPS semantics of these operators.

Finally, we have two rules governing the subtyping of effectful computations, namely S-Cons and S-Lift. The rule S-Cons follows from the CPS interpretation of delimited continuations—a type $\tau[T_1]T_2$ is interpreted in CPS as $(\tau \to T_1) \Rightarrow T_2$, where $\Rightarrow$ means an effectful function space (see Section 4.3). So, $\tau_1[T_1]U_1$ is a subtype of $\tau_2[T_2]U_2$ when $\tau_2 \to T_2$ is a subtype of $\tau_1 \to T_1$ (the argument type is, as always, treated contravariantly), and U_1 is a subtype of U_2 (the result type is, as always, treated covariantly). The rule S-Lift is more interesting and it says that a pure computation can be considered impure, provided the answer type of the inner-most context can be coerced into the type of the rest of the metacontext. We have, e.g., $\mathsf{Nat} \leq \mathsf{Nat}[\tau]\tau$ by using S-Lift, which combined with S-Cons also implies, e.g., $\mathsf{Nat}[\mathsf{Nat}[\tau]\tau]\tau' \leq \mathsf{Nat}[\mathsf{Nat}]\tau'$.

The following example illustrates some of the typing rules of the type system.

Example 4.1. Taking $T = \mathsf{Nat}[\mathsf{Nat}]\mathsf{Nat}$ and $\tau = \mathsf{Nat} \to T$ as well as $\Gamma = x : \mathsf{Nat} \to \mathsf{Nat}, y : \tau, z : \mathsf{Nat} \to \mathsf{Nat}$ and $\Delta = \Gamma, k : \mathsf{Nat} \to \mathsf{Nat}$, we have the following derivation D:

$$
\cfrac{
\cfrac{\Gamma \vdash x \ : \ \mathsf{Nat} \to \mathsf{Nat}}{} {\scriptstyle \text{T-Var}}
\qquad
\cfrac{
\cfrac{
\cfrac{\overset{D_1}{\Gamma \vdash y \ : \ \tau[\mathsf{Nat}]\mathsf{Nat}} \qquad \overset{D_2}{\Gamma \vdash \mathcal{S}_0 k.z \, (k \, 42) \ : \ T}}{\Gamma \vdash y \, \mathcal{S}_0 k.z \, (k \, 42) \ : \ T} {\scriptstyle \text{T-App}}
}{\Gamma \vdash \langle y \, \mathcal{S}_0 k.z \, (k \, 42) \rangle \ : \ \mathsf{Nat}} {\scriptstyle \text{T-Rst}}
}{}
}{\Gamma \vdash x \, \langle y \, \mathcal{S}_0 k.z \, (k \, 42) \rangle \ : \ \mathsf{Nat}} {\scriptstyle \text{T-PApp}}
$$

where D_1 is

$$
\cfrac{
\cfrac{}{\Gamma \vdash y \ : \ \tau} {\scriptstyle \text{T-Var}}
\qquad
\cfrac{
\cfrac{\cfrac{}{\mathsf{Nat} \leq \mathsf{Nat}} {\scriptstyle \text{S-Refl}}}{\tau \leq \tau[\mathsf{Nat}]\mathsf{Nat}} {\scriptstyle \text{S-Lift}}
}{}
}{\Gamma \vdash y \ : \ \tau[\mathsf{Nat}]\mathsf{Nat}} {\scriptstyle \text{T-Sub}}
$$

and D_2 is

$$
\cfrac{
\cfrac{}{\Delta \vdash z \ : \ \mathsf{Nat} \to \mathsf{Nat}} {\scriptstyle \text{T-Var}}
\qquad
\cfrac{
\cfrac{\cfrac{}{\Delta \vdash k \ : \ \mathsf{Nat} \to \mathsf{Nat}} {\scriptstyle \text{T-Var}} \qquad \cfrac{}{\Delta \vdash 42 \ : \ \mathsf{Nat}} {\scriptstyle \text{T-Const}}}{\Delta \vdash k \, 42 \ : \ \mathsf{Nat}} {\scriptstyle \text{T-PApp}}
}{}
}{
\cfrac{\Delta \vdash z \, (k \, 42) \ : \ \mathsf{Nat}}{\Gamma \vdash \mathcal{S}_0 k.z \, (k \, 42) \ : \ T} {\scriptstyle \text{T-Sft}}
} {\scriptstyle \text{T-PApp}}
$$

$\Box$

$$\tau \quad ::= \quad \mathsf{Nat} \mid \tau \to T \qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{(pure types)}$$

$$T \quad ::= \quad \tau \mid (\tau \to T) \Rightarrow T \qquad\qquad\qquad\qquad\qquad\qquad\text{(types)}$$

$$c \quad ::= \quad \mathsf{id} \mid c \circ c \mid c \to c \mid {\uparrow}c \mid c[c]c \qquad\qquad\qquad\qquad\text{(coercions)}$$

$$e \quad ::= \quad x \mid \lambda x.e \mid e\,e \mid c\,e \mid \mathsf{fix}\,x(x).e \mid n \qquad\qquad\qquad\text{(expressions)}$$

$$v \quad ::= \quad x \mid \lambda x.e \mid \mathsf{fix}\,x(x).e \mid (c \to c)\,v \mid {\uparrow}c\,v \mid (c[c]c)\,v \mid n \qquad\text{(values)}$$

$$E \quad ::= \quad \square \mid E\,e \mid v\,E \mid c\,E \qquad\qquad\qquad\qquad\text{(evaluation contexts)}$$

$$\frac{}{\mathsf{id} :: T \rhd T}\ \text{S-Refl} \qquad \frac{c_1 :: T_2 \rhd T_3 \qquad c_2 :: T_1 \rhd T_2}{c_1 \circ c_2 :: T_1 \rhd T_3}\ \text{S-Trans}$$

$$\frac{c_1 :: \tau_2 \rhd \tau_1 \qquad c_2 :: T_1 \rhd T_2}{c_1 \to c_2 :: (\tau_1 \to T_1) \rhd (\tau_2 \to T_2)}\ \text{S-Arr} \qquad \frac{c :: T_1 \rhd T_2}{{\uparrow}c :: \tau \rhd ((\tau \to T_1) \Rightarrow T_2)}\ \text{S-Lift}$$

$$\frac{c :: \tau_1 \rhd \tau_2 \qquad c_1 :: T_2 \rhd T_1 \qquad c_2 :: U_1 \rhd U_2}{c[c_1]c_2 :: ((\tau_1 \to T_1) \Rightarrow U_1) \rhd ((\tau_2 \to T_2) \Rightarrow U_2)}\ \text{S-Cons}$$

$$\frac{}{\Gamma \vdash n\ :\ \mathsf{Nat}}\ \text{T-Const} \qquad \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x\ :\ \tau}\ \text{T-Var} \qquad \frac{\Gamma, x : \tau \vdash e\ :\ T}{\Gamma \vdash \lambda x.e\ :\ \tau \to T}\ \text{T-Abs}$$

$$\frac{\Gamma \vdash e_1\ :\ \tau \to T \qquad \Gamma \vdash e_2\ :\ \tau}{\Gamma \vdash e_1\,e_2\ :\ T}\ \text{T-App}$$

$$\frac{\Gamma, x : \tau \to T \vdash e\ :\ U}{\Gamma \vdash \lambda x.e\ :\ (\tau \to T) \Rightarrow U}\ \text{T-KAbs}$$

$$\frac{\Gamma \vdash e\ :\ (\tau \to T) \Rightarrow U \qquad \Gamma \vdash v\ :\ \tau \to T}{\Gamma \vdash e\,v\ :\ U}\ \text{T-KApp}$$

$$\frac{c :: T \rhd U \qquad \Gamma \vdash e\ :\ T}{\Gamma \vdash c\,e\ :\ U}\ \text{T-CApp} \qquad \frac{\Gamma, f : \tau \to T, x : \tau \vdash e\ :\ T}{\Gamma \vdash \mathsf{fix}\,f(x).e\ :\ \tau \to T}\ \text{T-Fix}$$

$$E[(\lambda x.e)\,v] \to_\beta E[e\{v/x\}] \qquad\qquad\qquad E[\mathsf{id}\,v] \to_\iota E[v]$$

$$E[(\mathsf{fix}\,f(x).e)\,v] \to_\beta E[e\{\mathsf{fix}\,f(x).e/f, v/x\}] \qquad E[(c_1 \circ c_2)\,v] \to_\iota E[c_1\,(c_2\,v)]$$

$$E[{\uparrow}c\,v_1\,v_2] \to_\beta E[c\,(v_2\,v_1)] \qquad\qquad E[(c_1 \to c_2)\,v_1\,v_2] \to_\iota E[c_2\,(v_1\,(c_1\,v_2))]$$

$$E[(c[c_1]c_2)\,v_1\,v_2] \to_\iota E[c_2\,(v_1\,((c \to c_1)\,v_2))]$$

Figure 6: The target language—the λ-calculus with explicit coercions of control effects

4.3. **Coercion semantics: a type-directed selective CPS translation.** The type structure of the source calculus has been used by Materzok and the first author to define the semantics of well-typed expressions by a selective CPS translation of typing derivations

into the call-by-value λ-calculus [25]. Their translation can be seen as a coercion semantics of the source calculus that introduces explicit coercions in the image of the translation and leaves pure expressions in direct style. Such a semantics thus can serve as a basis for implementing delimited continuations as a fragment of a conventional functional language. However, one should first make sure that it is coherent.

The target calculus that we present next differs from the one considered in [25] in that it contains a separate syntactic category of coercions as well as a dedicated function type for expressions in CPS.

4.3.1. *Target calculus.* The syntax and typing rules of the target language are presented in Figure 6. There are two kinds of arrow type: the usual one $\tau \to T$ for regular functions and the effectful one $(\tau \to T) \Rightarrow U$ for expressions in CPS. We make this distinction to express the fact that the CPS translation (see Figure 7) yields expressions with strong restrictions on the occurrence of terms in CPS: they are never passed as arguments (typing environment consists of only pure types) and they can be applied only to values representing continuations (witness the rule T-KApp). Furthermore, observe that in general terms in CPS, i.e., of type $(\tau \to T) \Rightarrow U$ expect a bunch of (delimited) continuations to produce the final answer. For example the type $(\mathsf{Nat} \to \mathsf{Nat}) \Rightarrow (\mathsf{Nat} \to \mathsf{Nat}) \Rightarrow \mathsf{Nat}$ is inhabited by the term $\lambda k_1.\lambda k_2.k_1 \, (k_2 \, 42)$.

The syntactic category c of coercions and the typing rules defining judgment $c :: T_1 \rhd T_2$ are in one-to-one correspondence with the subtyping rules in the source calculus, discussed in Section 4.2. In particular the rule S-Lift allows us to treat a pure computation of type τ as an effectful one (i.e., in CPS) of type $(\tau \to T_1) \Rightarrow T_2$, provided the answer type T_1 of the immediate continuation is a subtype of T_2, the type describing the remaining continuations.

Again, the operational semantics distinguishes between β-rules and ι-rules. We classified the last β-rule as "actual computation" because it does not only rearrange coercions. It translates back a lifted value v_1 and applies to it a given continuation v_2. This rule and the last ι-rule reduce a coerced value applied to a continuation, so terms of the form $(\uparrow c \, v)$ and $(c[c] c \, v)$ are considered values. Notice that these values have effectful types. We extend the notion of ι-reduction to evaluation contexts: $E_1 \to_\iota E_2$ holds iff $E_1[v] \to_\iota E_2[v]$ for every value v.

As in Section 3, the metavariable C ranges over general closed contexts. We also define typing of general contexts $\vdash C \; : \; (\Gamma; T) \; \rightsquigarrow \; T_0$ as before. The definition of contextual approximation is necessarily slightly weaker, because we allow only contexts with pure answer type: we have $\Gamma \vdash e_1 \precsim_{ctx} e_2 \; : \; T$ if for every $\vdash C \; : \; (\Gamma; T) \; \rightsquigarrow \; \tau$ a termination of $C[e_1]$ implies a termination of $C[e_2]$. Indeed, an expression that requires a continuation to trigger computation can hardly be considered a complete program.

4.3.2. *Translation.* The coercion semantics of the source language is given by the type-directed selective CPS translation presented in Figure 7. The translation is selective because it leaves terms of pure type in direct style—witness, e.g, the equations for variable or pure application. Effectful applications are translated according to Plotkin's call-by-value CPS translation [31], whereas the translation of shift_0 and reset_0 is surprisingly straightforward—shift_0 is turned into a lambda-abstraction expecting a delimited continuation, and reset_0 is interpreted by providing its subexpression with the reset delimited continuation, represented by the identity function. The following example illustrates the main points of the CPS translation.

$$\llbracket \mathsf{Nat} \rrbracket_p = \mathsf{Nat} \qquad\qquad \llbracket \tau \rrbracket = \llbracket \tau \rrbracket_p$$

$$\llbracket \tau \to T \rrbracket_p = \llbracket \tau \rrbracket_p \to \llbracket T \rrbracket \qquad\qquad \llbracket \tau[T]U \rrbracket = (\llbracket \tau \rrbracket_p \to \llbracket T \rrbracket) \Rightarrow \llbracket U \rrbracket$$

$$\mathcal{S}\llbracket T \leq T \rrbracket_{\text{S-Refl}} = \mathsf{id}$$

$$\mathcal{S}\llbracket T_1 \leq T_3 \rrbracket_{\text{S-Trans}(D_1,D_2)} = \mathcal{S}\llbracket T_2 \leq T_3 \rrbracket_{D_1} \circ \mathcal{S}\llbracket T_1 \leq T_2 \rrbracket_{D_2}$$

$$\mathcal{S}\llbracket \tau_1 \to T_1 \leq \tau_2 \to T_2 \rrbracket_{\text{S-Arr}(D_1,D_2)} = \mathcal{S}\llbracket \tau_2 \leq \tau_1 \rrbracket_{D_1} \to \mathcal{S}\llbracket T_1 \leq T_2 \rrbracket_{D_2}$$

$$\mathcal{S}\llbracket \tau \leq \tau[T]U \rrbracket_{\text{S-Lift}(D)} = {\uparrow}\mathcal{S}\llbracket T \leq U \rrbracket_D$$

$$\mathcal{S}\llbracket \tau_1[T_1]U_1 \leq \tau_2[T_2]U_2 \rrbracket_{\text{S-Cons}(D,D_1,D_2)} = \mathcal{S}\llbracket \tau_1 \leq \tau_2 \rrbracket_D[\mathcal{S}\llbracket T_2 \leq T_1 \rrbracket_{D_1}]\mathcal{S}\llbracket U_1 \leq U_2 \rrbracket_{D_2}$$

$$\mathcal{T}\llbracket e \rrbracket_{\text{T-Sub}(D_1,D_2)} = \mathcal{S}\llbracket T \leq U \rrbracket_{D_2} \mathcal{T}\llbracket e \rrbracket_{D_1}$$

$$\mathcal{T}\llbracket x \rrbracket_{\text{T-Var}} = x$$

$$\mathcal{T}\llbracket \lambda x.e \rrbracket_{\text{T-Abs}(D)} = \lambda x.\mathcal{T}\llbracket e \rrbracket_D$$

$$\mathcal{T}\llbracket e_1\, e_2 \rrbracket_{\text{T-PApp}(D_1,D_2)} = \mathcal{T}\llbracket e_1 \rrbracket_{D_1}\, \mathcal{T}\llbracket e_2 \rrbracket_{D_2}$$

$$\mathcal{T}\llbracket e_1\, e_2 \rrbracket_{\text{T-App}(D_1,D_2)} = \lambda k.\mathcal{T}\llbracket e_1 \rrbracket_{D_1}\, (\lambda f.\mathcal{T}\llbracket e_2 \rrbracket_{D_2}\, (\lambda x.f\, x\, k))$$

$$\mathcal{T}\llbracket \mathsf{fix}\, f(x).e \rrbracket_{\text{T-Fix}(D)} = \mathsf{fix}\, f(x).\mathcal{T}\llbracket e \rrbracket_D$$

$$\mathcal{T}\llbracket n \rrbracket_{\text{T-Const}} = n$$

$$\mathcal{T}\llbracket \mathcal{S}_0 x.e \rrbracket_{\text{T-Sft}(D)} = \lambda x.\mathcal{T}\llbracket e \rrbracket_D$$

$$\mathcal{T}\llbracket \langle e \rangle \rrbracket_{\text{T-Rst}(D)} = \mathcal{T}\llbracket e \rrbracket_D\, (\lambda x.x)$$

Figure 7: Type-directed selective CPS translation

Example 4.2. Let us consider the derivation D given in Example 4.1. We have

$$\mathcal{T}\llbracket x\, \langle y\, \mathcal{S}_0 k.z\, (k\, 42) \rangle \rrbracket_D = x\, ((\lambda l.\, {\uparrow}\mathsf{id}\, y\, (\lambda f.(\lambda k.z\, (k\, 42))\, (\lambda u.f\, u\, l)))\, (\lambda v.v))$$

where

- the applications $x\, \langle y\, \mathcal{S}_0 k.z\, (k\, 42) \rangle$, $z\, (k\, 42)$, and $k\, 42$ are pure and, hence, stay in direct style through the translation;
- the application $y\, \mathcal{S}_0 k.z\, (k\, 42)$ is effectful, and therefore translated to CPS, with y coerced to a continuation-expecting expression.

$\square$

The following lemma establishes the type soundness of the CPS translation.

Lemma 4.3. *Coercion semantics preserves types:*

   (1) If $D :: T_1 \leq T_2$ then $\mathcal{S}\llbracket T_1 \leq T_2 \rrbracket_D :: \llbracket T_1 \rrbracket \rhd \llbracket T_2 \rrbracket$.
   (2) If $D :: \Gamma \vdash e\ :\ T$ then $\llbracket \Gamma \rrbracket \vdash \mathcal{T}\llbracket e \rrbracket_D\ :\ \llbracket T \rrbracket$.

The problem of coherence of the CPS translation is demonstrated in the following example.

Example 4.4. Let us consider the term $(\text{fix } f(x).f\ x)\ 1$ in the source language. We derive the type $\text{Nat}[T]T$ for it in two ways: let D_1 be the derivation

$$\vdots$$

$$\cfrac{\cfrac{f : \text{Nat} \to \text{Nat}[T]T, x : \text{Nat} \vdash f\ x\ :\ \text{Nat}[T]T}{\vdash \text{fix } f(x).f\ x\ :\ \text{Nat} \to \text{Nat}[T]T}\text{ T-Fix} \qquad \cfrac{}{\vdash 1\ :\ \text{Nat}}\text{ T-Const}}{\vdash (\text{fix } f(x).f\ x)\ 1\ :\ \text{Nat}[T]T}\text{ T-PApp}$$

and D_2 be the derivation

$$\vdots$$

$$\cfrac{\cfrac{\vdash \text{fix } f(x).f\ x\ :\ \text{Nat} \to \text{Nat}[T]T \qquad \vdots}{\vdash \text{fix } f(x).f\ x\ :\ (\text{Nat} \to \text{Nat}[T]T)[T]T}\text{ T-Sub} \quad \cfrac{\cfrac{}{\vdash 1\ :\ \text{Nat}}\text{ T-Const} \quad \vdots}{\vdash 1\ :\ \text{Nat}[T]T}\text{ T-Sub}}{\vdash (\text{fix } f(x).f\ x)\ 1\ :\ \text{Nat}[T]T}\text{ T-App}$$

Then we have

$$\mathcal{T}[\![(\text{fix } f(x).f\ x)\ 1]\!]_{D_1}\ =\ (\text{fix } f(x).f\ x)\ 1$$

$$\mathcal{T}[\![(\text{fix } f(x).f\ x)\ 1]\!]_{D_2}\ =\ \lambda k.\ {\uparrow}\text{id}\ (\text{fix } f(x).f\ x)\ (\lambda g.\ {\uparrow}\text{id}\ 1\ (\lambda y.g\ y\ k))$$

We observe that the two terms are quite distinct: one is a diverging expression, and the other is a lambda abstraction. However, the results of the next two sections show that these two terms are contextually equivalent, and so both typing derivations have equivalent coercion semantics. $\qquad\square$

4.4. **Logical relations.** The logical relations are defined in Figure 8. We use the metavariable κ to range over values that are meant to represent continuations. The relation $\mathcal{V}[\![\tau_1; \tau_2]\!]$ for pure values and the relation $\mathcal{E}[\![T_1; T_2]\!]$ for expressions are similar to the relations defined in Section 3.3. All information about control effects is captured in the relation $\mathcal{K}[\![T_1; T_2]\!]$ for contexts. If T_1 and T_2 are pure, then we proceed as usual with biorthogonal logical relations: two contexts are related if they behave the same way for related values, since pure computations can interact with their context only by returning a value.

For impure types (of the form $(\tau \to T) \Rightarrow U$) contexts should be plugged with effectful expressions which expect a continuation (represented as a function) to trigger computation. A context is able to provide such a continuation κ if it can be decomposed as an application of the hole to κ and the rest of the context. In general it does not mean that the context has necessarily the form $E'[\Box\ \kappa]$, but that it can be ι-reduced to such a form. For instance, context $E[((c[c_1]c_2)\ \Box)\ \kappa]$ does not have an application to κ as the inner-most element, but still applies plugged value to a continuation $(c \to c_1)\ \kappa$ after one ι-step. The logical relation for contexts of impure types (case $\mathcal{K}[\![(\tau_1 \to T_1) \Rightarrow U_1; (\tau_2 \to T_2) \Rightarrow U_2]\!]$) relates two contexts iff they can be decomposed (using ι-reduction) as applications to related continuations in related contexts.

The most interesting are the cases that relate pure and impure contexts. As previously, the impure context should be decomposed to a continuation κ and the rest of the context. Then the pure context should be decomposed in such a way that the continuation κ is related with some portion E of the pure context. The answer type of E cannot be retrieved

$$
\begin{aligned}
(v_1, v_2) \in \mathcal{V}[\![\mathsf{Nat}; \mathsf{Nat}]\!] &\iff \exists n, v_1 = v_2 = n \\
(v_1, v_2) \in \mathcal{V}[\![\tau_1 \to T_1; \tau_2 \to T_2]\!] &\iff \forall (a_1, a_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!].(v_1\, a_1, v_2\, a_2) \in \mathcal{E}[\![T_1; T_2]\!] \\
(v_1, v_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!] &\iff \bot \quad\text{otherwise} \\[6pt]
(e_1, e_2) \in \mathcal{E}[\![T_1; T_2]\!] &\iff \forall (E_1, E_2) \in \mathcal{K}[\![T_1; T_2]\!].E_1[e_1] \precsim E_2[e_2] \\[6pt]
(E_1, E_2) \in \mathcal{K}[\![\tau_1; \tau_2]\!] &\iff \forall (v_1, v_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!].E_1[v_1] \precsim E_2[v_2] \\
(E_1, E_2) \in \mathcal{K}[\![\tau_1; (\tau_2 \to T_2) \Rightarrow U_2]\!] &\iff \exists T, (E, \kappa) \in \mathcal{KV}[\![\tau_1 \leadsto T; \tau_2 \to T_2]\!], \\
&\qquad (E_1', E_2') \in \mathcal{K}[\![T; U_2]\!]. \\
&\qquad\qquad E_1 \to_\iota^* E_1'[E] \wedge E_2 \to_\iota^* E_2'[\Box\,\kappa] \\
(E_1, E_2) \in \mathcal{K}[\![(\tau_1 \to T_1) \Rightarrow U_1; \tau_2]\!] &\iff \exists T, (\kappa, E) \in \triangleright\mathcal{VK}[\![\tau_1 \to T_1; \tau_2 \leadsto T]\!], \\
&\qquad (E_1', E_2') \in \triangleright\mathcal{K}[\![U_1; T]\!]. \\
&\qquad\qquad E_1 \to_\iota^* E_1'[\Box\,\kappa] \wedge E_2 \to_\iota^* E_2'[E] \\
(E_1, E_2) \in \mathcal{K}[\![(\tau_1 \to T_1) \Rightarrow U_1; & \\
(\tau_2 \to T_2) \Rightarrow U_2]\!] &\iff \exists (\kappa_1, \kappa_2) \in \mathcal{V}[\![\tau_1 \to T_1; \tau_2 \to T_2]\!], \\
&\qquad (E_1', E_2') \in \mathcal{K}[\![U_1; U_2]\!]. \\
&\qquad\qquad E_1 \to_\iota^* E_1'[\Box\,\kappa_1] \wedge E_2 \to_\iota^* E_2'[\Box\,\kappa_2] \\[6pt]
(E, \kappa) \in \mathcal{KV}[\![\tau_1 \leadsto T_1; \tau_2 \to T_2]\!] &\iff \forall (a_1, a_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!].(E[a_1], \kappa\, a_2) \in \mathcal{E}[\![T_1; T_2]\!] \\
(\kappa, E) \in \mathcal{VK}[\![\tau_1 \to T_1; \tau_2 \leadsto T_2]\!] &\iff \forall (a_1, a_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!].(\kappa\, a_1, E[a_2]) \in \mathcal{E}[\![T_1; T_2]\!] \\[6pt]
(\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma_1; \Gamma_2]\!] &\iff \forall x.(\gamma_1(x), \gamma_2(x)) \in \mathcal{V}[\![\Gamma_1(x); \Gamma_2(x)]\!] \\[6pt]
\Gamma_1; \Gamma_2 \vdash e_1 \precsim_{log} e_2 : T_1; T_2 &\iff \forall (\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma_1; \Gamma_2]\!].(e_1\gamma_1, e_2\gamma_2) \in \mathcal{E}[\![T_1; T_2]\!]
\end{aligned}
$$

Figure 8: Logical relations for the λ-calculus with explicit coercions of control effects

from the type of the initial pure context, so we quantify over all possible types. Unlike the logical relations for parametricity [2, 1] we quantify over syntactic types. In order to make the construction well-founded, the relations are defined by nested induction on step indices and on the structure of the second type. Notice that step indices play a role only in one case—when we quantify over the second type and the later operator guards the non-structural use of the relations $\mathcal{VK}[\![\tau_1 \to T_1; \tau_2 \leadsto T]\!]$ and $\mathcal{K}[\![U_1; T]\!]$. The auxiliary relations $\mathcal{KV}[\![\tau_1 \leadsto T_1; \tau_2 \to T_2]\!]$ and $\mathcal{VK}[\![\tau_1 \to T_1; \tau_2 \leadsto T_2]\!]$ relate a portion of an evaluation context with a value of an arrow type and they are defined analogously to the value relation for functions.

The relations of this section possess properties analogous to the ones of Section 3.3, in particular the relation $\precsim$ is preserved by reduction (Lemma 3.4) and the compatibility lemmas (including Lemma 3.6) hold. However, the proof of the compatibility lemmas requires the following results that establish the preservation of relations with respect to ι-reductions of evaluation contexts.

Lemma 4.5. *The following assertions hold:*

(1) If $E \to_\iota^* E'$ and $E'[e_1] \precsim e_2$ then $E[e_1] \precsim e_2$.

(2) If $E \to_\iota^* E'$ and $e_1 \precsim E'[e_2]$ then $e_1 \precsim E[e_2]$.

(3) If $E_1 \to_\iota^* E_1'$ and $(E_1', E_2) \in \mathcal{K}[\![T_1; T_2]\!]$ then $(E_1, E_2) \in \mathcal{K}[\![T_1; T_2]\!]$.

(4) If $E_2 \to_\iota^* E_2'$ and $(E_1, E_2') \in \mathcal{K}[\![T_1; T_2]\!]$ then $(E_1, E_2) \in \mathcal{K}[\![T_1; T_2]\!]$.

The rest of the soundness proof follows the same lines as in Section 3.3. Interestingly, the adequacy lemma can be proved only for pure types, which is in harmony with the notion of contextual equivalence in the target calculus.

Theorem 4.6 (Fundamental property). *If $\Gamma \vdash e \; : \; T$ then $\Gamma; \Gamma \vdash e \precsim_{log} e \; : \; T; T$.*

Lemma 4.7 (Precongruence). *If $\vdash C \; : \; (\Gamma; T) \; \rightsquigarrow \; \tau$ and $\Gamma; \Gamma \vdash e_1 \precsim_{log} e_2 \; : \; T; T$, then $(C[e_1], C[e_2]) \in \mathcal{E}[\![\tau; \tau]\!]$.*

Lemma 4.8 (Adequacy). *If $(e_1, e_2) \in \mathcal{E}[\![\tau; \tau]\!]$ then $e_1 \precsim e_2$.*

Theorem 4.9 (Soundness). *If $k \models \Gamma; \Gamma \vdash e_1 \precsim_{log} e_2 \; : \; T; T$ holds for every k, then $\Gamma \vdash e_1 \precsim_{ctx} e_2 \; : \; T$.*

4.5. **Coherence of the CPS translation.** Although standard compatibility lemmas and coercion compatibility suffice to prove soundness of logical relations, we need another kind of compatibility to prove coherence, since there is another source of ambiguity. Two typing derivations in the source language can be different not only because of the subsumption rule, but also because of two rules for application.

Lemma 4.10 (Mixed application compatibility). *The following assertions hold:*

(1) If $\Gamma_1; \Gamma_2 \vdash f_1 \precsim_{log} f_2 \; : \; ((\tau_1' \to (\tau_1 \to U_4) \Rightarrow U_3) \to U_2) \Rightarrow U_1; \tau_2' \to T_2$
and $\Gamma_1; \Gamma_2 \vdash e_1 \precsim_{log} e_2 \; : \; (\tau_1' \to U_3) \Rightarrow U_2; \tau_2'$
then $\Gamma_1; \Gamma_2 \vdash \lambda k.f_1 (\lambda f.e_1 (\lambda x.f \; x \; k)) \precsim_{log} f_2 \; e_2 \; : \; (\tau_1' \to U_4) \Rightarrow U_1; T$.

(2) If $\Gamma_1; \Gamma_2 \vdash f_1 \precsim_{log} f_2 \; : \; \tau_1' \to T_1; ((\tau_2' \to (\tau_2 \to U_4) \Rightarrow U_3) \to U_2) \Rightarrow U_1$
and $\Gamma_1; \Gamma_2 \vdash e_1 \precsim_{log} e_2 \; : \; \tau_1'; (\tau_2' \to U_3) \Rightarrow U_2$
then $\Gamma_1; \Gamma_2 \vdash f_1 \; e_1 \precsim_{log} \lambda k.f_2 (\lambda f.e_2 (\lambda x.f \; x \; k)) \; : \; T; (\tau_2' \to U_4) \Rightarrow U_1$.

Proof. Both cases are similar, so we show only the first one. We have to show that both terms closed by substitutions have the same observations in related contexts $(E_1, E_2) \in \mathcal{K}[\![(\tau_1' \to U_4) \Rightarrow U_1; T]\!]$. Since context E_1 is in relation for effectful type, by the definition of logical relations and Lemma 4.5, it can be decomposed as a continuation κ and the rest of the context. Now we have the missing continuation κ that can trigger computation in the first term, so the rest of the proof consists in simple context manipulations, applying definitions and performing reductions. $\square$

Lemma 4.11. *If $D_i :: \Gamma_i \vdash e \; : \; T_i$ for $i = 1, 2$ are two typing judgments for the same term e of the source language, then $[\![\Gamma_1]\!]; [\![\Gamma_2]\!] \vdash \mathcal{T}[\![e]\!]_{D_1} \precsim_{log} \mathcal{T}[\![e]\!]_{D_2} \; : \; [\![T_1]\!]; [\![T_2]\!]$.*

Theorem 4.12 (Coherence). *If D_1 and D_2 are derivations of the same typing judgment $\Gamma \vdash e \; : \; T$, then $[\![\Gamma]\!] \vdash \mathcal{T}[\![e]\!]_{D_1} \precsim_{ctx} \mathcal{T}[\![e]\!]_{D_2} \; : \; [\![T]\!]$.*

4.6. **Coercions as λ-terms.** In contrast to the calculus considered in Section 3.4, such a coherence theorem does not imply coherence of the translation directly to the simply typed λ-calculus (where coercions are expressed as λ-terms). As a counterexample, consider the expression $(\mathsf{fix}\ f(x).f\ x)\ 1$ and the two derivations D_1 and D_2 presented in Example 4.4. Recall that

$$\mathcal{T}[\![(\mathsf{fix}\ f(x).f\ x)\ 1]\!]_{D_1}\ =\ (\mathsf{fix}\ f(x).f\ x)\ 1$$
$$\mathcal{T}[\![(\mathsf{fix}\ f(x).f\ x)\ 1]\!]_{D_2}\ =\ \lambda k.\uparrow\!\mathsf{id}\ (\mathsf{fix}\ f(x).f\ x)\ (\lambda g.\uparrow\!\mathsf{id}\ 1\ (\lambda y.g\ y\ k))$$

The former term is a diverging computation, but the latter is a lambda abstraction waiting for an argument (continuation). After translation to simply typed λ-calculus, these terms can be distinguished even by the context $C = (\lambda x.1)\ \square$ with answer type Nat. But by Theorem 4.12 these terms are equivalent. This is because types in the target language carry more information than simple types, and in particular, an expression of a type $(\tau \to T) \Rightarrow U$ is not a usual function, but a computation waiting for a continuation, as explained in Section 4.3. Computations cannot be passed as arguments, so the context C is not well-typed in the target calculus.

But still we can prove some interesting properties of a direct translation to the simply typed λ-calculus in two cases: when control effects do not leak to the context or when we relate only whole programs. Let $|e|$ be a term e with all coercions replaced by corresponding expressions.

Corollary 4.13. *If $D_1, D_2 :: \Gamma \vdash e\ :\ \tau$ and τ does not contain any type of the form $\tau'[T]U$, then $|\mathcal{T}[\![e]\!]_{D_1}|$ and $|\mathcal{T}[\![e]\!]_{D_2}|$ are contextually equivalent.*

Corollary 4.14. *If $D_1, D_2 :: \Gamma \vdash e\ :\ \tau$ then $|\mathcal{T}[\![e]\!]_{D_1}|$ terminates iff $|\mathcal{T}[\![e]\!]_{D_2}|$ terminates. Moreover, if $\tau = \mathsf{Nat}$ and one of the expressions terminates to a constant, then the other term evaluates to the same constant.*

5. Coq formalization

5.1. **The library IxFree.** Our Coq formalization accompanying this article is built on our IxFree library that contains a shallow embedding of the LSLR logic similar to Appel et al.'s formalization of the "very modal model" [4] and Krebbers et al.'s Iris proof mode [22]. Instead of using type `Prop` to represent propositions, we use a special type of "indexed propositions" defined as a type of monotone functions from `nat` to `Prop`.

```
Definition monotone (P : nat → Prop) := ∀ n, P (S n) → P n.
Definition IProp := { P : nat → Prop | monotone P }.

Definition I_valid_at (n : nat) (P : IProp) := proj1_sig P n.
Notation "n ⊨ P" := (I_valid_at n P).
```

One of the main differences between our library and Iris proof mode is a way of keeping track of the assumptions. Instead of interpreting a sequent $\varphi_1, \ldots, \varphi_n \vdash \psi$ directly, we treat it as $k \models \psi$ with the standard Coq assumptions $k \models \varphi_1, \ldots, k \models \varphi_n$. This approach is very convenient since it allows for reusing a number of existing Coq tactics, but it does not scale to e.g. linear logic like Iris.

Logical connectives including the later operator are functions on type `IProp` with defined human readable notation. The library provides lemmas and tactics representing the most important inference rules. Tactics not only apply the corresponding lemmas, but also hide the step index arithmetic from the user. For instance, when proving sequent $Q \vdash P \Rightarrow Q$ represented by the following Coq goal

```
P : IProp
Q : IProp
k : nat
H1 : k ⊨ Q
============================
k ⊨ P ⇒ Q
```

the introduction of implication tactic `iintro H2` behaves exactly like introduction of implication rule, producing the goal

```
P : IProp
Q : IProp
k : nat
H1 : k ⊨ Q
H2 : k ⊨ P
============================
k ⊨ Q
```

even if the lemma corresponding to that rule requires quantification over all smaller indices:

```
Lemma I_arrow_intro {n : nat} {P Q : IProp} :
    (∀ k, k ≤ n, (k ⊨ P) → (k ⊨ Q)) → (n ⊨ P ⇒ Q).
```

5.2. Recursive predicates.

The LSLR logic allows for recursive predicates and relations, provided all recursive occurrences are guarded by the later operator. Such a syntactic requirement is not compatible with structural recursion in Coq, so we rely on the notion of *contractiveness*[4]. Informally, a function is contractive if it maps approximately equal arguments to more equal results. This intuition can be expressed using the later modality:

```
Definition contractive (l : list Type) (f : IRel l → IRel l)
    : Prop := ∀ R₁ R₂, ⊨ ▷(R₁ ≈ᵢ R₂) ⇒ f R₁ ≈ᵢ f R₂.
```

where `IRel l` is a type of indexed relations on types described by `l`, and $\approx_i$ is an indexed version of relation equivalence. The library provides a general method of constructing recursive relations as a fixed point of a contractive function:

```
Definition I_fix (l : list Type) (f : IRel l → IRel l) :
    contractive l f → IRel l.
```

If all occurrences of the function argument are guarded by the later operator, then the function can be proven to be contractive, and the proof can be (mostly) automatized by the `auto_contr` tactic.

6. Conclusion

We have shown that the technique of logical relations can be used for establishing the coherence of subtyping, when it is phrased in terms of contextual equivalence in the target of the coercion translation. In particular, we have demonstrated that a combination of heterogeneity, biorthogonality and step-indexing provides a sufficiently powerful tool for establishing coherence of effect subtyping in a calculus of delimited control with the coercion semantics given by a type-directed selective CPS translation. Moreover, we have successfully applied the presented approach also to other calculi with subtyping, e.g., as demonstrated in this article for the simply-typed λ-calculus with recursion. The Coq development accompanying this paper is based on a new embedding of Dreyer et al.'s logic for reasoning about step-indexing [18] that, we believe, considerably improves the presentation and formalization of the logical relations.

Regarding logical relations for type-and-effect systems, there has been work on proving correctness of a partial evaluator for shift and reset by Asai [5], and on termination of evaluation of the λ-calculi with delimited-control operators by Biernacka et al. [8, 10] and by Materzok and the first author [25]. Unsurprisingly, all these results, like ours, are built on the notion of biorthogonality, even if not mentioned explicitly. The distinctive feature of our construction is a combination of heterogeneity and step-indexing that supports reasoning about the observational equivalence of terms of different types whose structure is very distant from each other, e.g., about direct-style and continuation-passing-style terms.

Logical relations presented in Section 4 require step-indexing in order to ensure well-formedness of the definition in the presence of quantification over types. A similar problem occurs in polymorphic λ-calculi and is usually resolved using quantification over relations that describe semantic types. Adapting this approach to our calculus is not straightforward, because we need quantification over a single type, whereas the semantic types are defined for pairs of types. An interesting question is if the step-indexing in the relations of Section 4 can be avoided by using quantification over relations.

The type systems considered in this work are monomorphic. It remains to be investigated how the ideas presented in this article would carry over to system $F_{\leq}$ and its extensions. In particular, we find it worthwhile to develop a polymorphic type-and-effect system for $shift_0$, perhaps by marrying the type system of Section 4 with Asai and Kameyama's polymorphic type system for delimited continuations [6], along with a type-directed selective CPS translation to system F with explicit coercions. Establishing the coherence of the translation would again be a crucial step in such a development.

References

[1] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In Benjamin C. Pierce, editor, *Proceedings of the 36th Annual ACM Symposium on Principles of Programming Languages, POPL 2009*, pages 340–353, Savannah, GA, USA, January 2009. ACM Press.

[2] Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 69–83, Vienna, Austria, March 2006. Springer.

[3] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, 2001.

[4] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In Matthias Felleisen, editor, *Proceedings of the 34th Annual ACM Symposium on Principles of Programming Languages, POPL 2007*, pages 109–122, Nice, France, January 2007. ACM Press.

[5] Kenichi Asai. Logical relations for call-by-value delimited continuations. In Marko van Eekelen, editor, *Proceedings of the 6th Symposium on Trends in Functional Programming (TFP 2005)*, pages 413–428, Tallinn, Estonia, September 2005. Institute of Cybernetics at Tallinn Technical University.

[6] Kenichi Asai and Yukiyoshi Kameyama. Polymorphic delimited continuations. In Zhong Shao, editor, *Proceedings of the 5th Asian Symposium on Programming Languages and Systems, APLAS'07*, volume 4807 of *Lecture Notes in Computer Science*, pages 239–254, Singapore, December 2007.

[7] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics*. North-Holland, revised edition, 1984.

[8] Małgorzata Biernacka and Dariusz Biernacki. Context-based proofs of termination for typed delimited-control operators. In Francisco J. López-Fraguas, editor, *Proceedings of the 11th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 289–300, Coimbra, Portugal, September 2009. ACM Press.

[9] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, November 2005.

[10] Małgorzata Biernacka, Dariusz Biernacki, and Sergueï Lenglet. Typing control operators in the CPS hierarchy. In Michael Hanus, editor, *Proceedings of the 13th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'11)*, pages 149–160, Odense, Denmark, July 2011. ACM Press.

[11] Dariusz Biernacki and Piotr Polesiuk. Logical relations for coherence of effect subtyping. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015)*, volume 38 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 107–122, Warsaw, Poland, July 2015. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik.

[12] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, 1991.

[13] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2(1):55–91, 1992.

[14] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. DIKU Rapport 89/12, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, July 1989.

[15] Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

[16] Paul Downen and Zena M. Ariola. Compositional semantics for composable continuations: from abortive to delimited control. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP'14), Gothenburg, Sweden, September 1-3, 2014*, pages 109–122. ACM Press, 2014.

[17] Paul Downen and Zena M. Ariola. Delimited control and computational effects. *Journal of Functional Programming*, 24(1):1–55, 2014.

[18] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. *Logical Methods in Computer Science*, 7(2:16):1–37, 2011.

[19] Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming*, 22(4-5):477–528, 2012.

[20] Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and Peter Mager, editors, *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages, POPL 1988*, pages 180–190, San Diego, California, January 1988. ACM Press.

[21] Andrzej Filinski. Representing monads. In Hans-J. Boehm, editor, *Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages, POPL 1994*, pages 446–457, Portland, Oregon, January 1994. ACM Press.

[22] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*, pages 205–217, Paris, France, January 2017. ACM Press.

[23] Jean-Louis Krivine. Classical logic, storage operators and second-order lambda-calculus. *Annals of Pure and Applied Logic*, 68(1):53–78, 1994.

[24] Marek Materzok. Axiomatizing subtyped delimited continuations. In Simona Ronchi Della Rocca, editor, *Computer Science Logic 2013 (CSL 2013)*, volume 23 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 521–539, Torino, Italy, Sep 2013. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik.

[25] Marek Materzok and Dariusz Biernacki. Subtyping delimited continuations. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceedings of the 2011 ACM SIGPLAN International Conference on Functional Programming (ICFP'11)*, pages 81–93, Tokyo, Japan, September 2011. ACM Press.

[26] Marek Materzok and Dariusz Biernacki. A dynamic interpretation of the CPS hierarchy. In Ranjit Jhala and Atsushi Igarashi, editors, *Proceedings of the 10th Asian Symposium on Programming Languages and Systems, APLAS'12*, volume 7705 of *Lecture Notes in Computer Science*, pages 296–311, Kyoto, Japan, December 2012. Springer.

[27] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[28] James H. Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, Massachusets Institute of Technology, 1968.

[29] Guillaume Munch-Maccagnoni. Formulae-as-types for an involutive negation. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic (CSL) and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14*, pages 70:1–70:10, Vienna, Austria, July 2014. ACM Press.

[30] Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In Andrew Gordon and Andrew Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 227–273. Publications of the Newton Institute, Cambridge University Press, 1998.

[31] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[32] John C. Reynolds. The coherence of languages with intersection types. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software, International Conference TACS '91, Sendai, Japan, September 24-27, 1991, Proceedings*, volume 526 of *Lecture Notes in Computer Science*, pages 675–700. Springer, 1991.

[33] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In Andrew Tolmach, editor, *Proceedings of the 2009 ACM SIGPLAN International Conference on Functional Programming (ICFP'09)*, pages 317–328, Edinburgh, UK, August 2009. ACM Press.

[34] Jan Schwinghammer. Coherence of subsumption for monadic types. *Journal of Functional Programming*, 19(2):157–172, 2009.