Handling Delimited Continuations with Dependent Types

YOUYOU CONG, Ochanomizu University, Japan
KENICHI ASAI, Ochanomizu University, Japan

Dependent types are a powerful tool for maintaining program invariants. To take advantage of this aspect in real-world programming, efforts have been put into enriching dependently typed languages with missing constructs, most notably, effects. This paper presents a language that has two practically interesting ingredients: dependent inductive types, and the delimited control constructs shift and reset. When integrating delimited control into a dependently typed language, however, two challenges arise. First, the dynamic nature of control operators, which is the source of their expressiveness, can break fundamental language properties such as logical consistency and subject reduction. Second, CPS translations, which we often use to define the semantics of control operators, do not scale straightforwardly to dependently typed languages. We solve the former issue by restricting dependency of types, and the latter using answer-type polymorphism of pure terms. The main contribution of this paper is to give a sound type system of our language, as well as a type-preserving CPS translation. We also discuss various extensions, which would make our language more like a full-spectrum proof assistant but pose non-trivial issues.

CCS Concepts: · Software and its engineering → Functional languages; Polymorphism; · Theory of computation → Type theory; Control primitives;

Additional Key Words and Phrases: dependent types, delimited continuations, CPS translation

ACM Reference Format:
Youyou Cong and Kenichi Asai. 2018. Handling Delimited Continuations with Dependent Types. Proc. ACM Program. Lang. 2, ICFP, Article 69 (September 2018), 31 pages. https://doi.org/10.1145/3236764

1 INTRODUCTION

Dependently typed languages such as Coq [The Coq Development Team 2018] and Agda [Norell 2007] are used for building secure software and reliable proofs. By letting terms to appear in types, these languages greatly broaden the scope of expressible program specifications and mathematical statements. The last decade has seen diverse applications of dependently typed languages, ranging from formalization of the famous four-color theorem [Gonthier 2008], to verification of realistic compilers [Anand et al. 2017; Leroy 2006].

Although widely used, dependently typed languages are sometimes not as convenient as ordinary programming languages. One limitation is that most languages only allow pure, effect-free programs: e.g., one cannot raise exceptions or use mutable references. From the perspective of theorem proving, disallowing effects seems like a reasonable restriction, since they have no logical meaning. However, the absence of effects can be an obstacle to using dependent types in real-world programming. To close this gap, researchers have been seeking dependent and effectful languages that still make

*We use a blue, sans-serif font for our language and a red, serif font for the target of the CPS translation. Although they are distinguishable in black-and-white, the paper is easier to read if viewed/printed in color.

Authors’ addresses: Youyou Cong, Department of Information Sciences, Ochanomizu University, Japan, so.yuyu@is.ocha.ac.jp; Kenichi Asai, Department of Information Sciences, Ochanomizu University, Japan, asai@is.ocha.ac.jp.

This work is licensed under a Creative Commons Attribution 4.0 International License.
sense as a logic. The F* language of Swamy et al. [2016] is a successful attempt in this line of work: it provides support for user-extensible effects, together with a rich type system allowing us to maintain strong invariants. Others have looked at dependent variants of Levy’s call-by-push-value calculus [Levy 2012], which is a framework for formalizing effectful languages, and studied their semantics from a category-theoretic point of view [Ahman 2017b; Vákár 2017].

Our work aims at enriching dependently typed languages with a different language construct: control operators. Control operators enable explicit handling of control flow in direct-style programs, instead of those written in continuation-passing style (CPS). This ability enables us to simulate monadic effects [Filinski 1994], implement program optimization algorithms [Danvy 1996; Kameyama et al. 2009], and build Web applications [Flatt et al. 2007; Krishnamurthi et al. 2007] in a natural and concise manner. The long-term goal of this study is to lift these applications to a dependently typed setting, where we can enjoy more safety guarantees.

Handling control effects while allowing type dependency, however, is known to be a delicate issue. As Herbelin [2005] shows, unrestricted use of the call/cc operator with strong Σ-types (dependent pairs with projections) leads to inconsistency. More recently, Miquey [2017] reported that a naïve extension of the λµ˜µ-calculus [Curien and Herbelin 2000] with dependent types results in failure of subject reduction. To avoid these undesirable consequences, one has to design the language in such a way that types depend only on pure, effect-free terms [Herbelin 2012; Miquey 2017]. Interestingly, but unsurprisingly, this restriction is also adopted by the above-mentioned studies that deal with non-control effects.

Another problem arises when scaling the semantics of control operators to a dependently typed language. Since control operators handle continuations, their semantics is often given in terms of a CPS translation, and in a typed setting, the translation must be type-preserving. However, as reported by Barthe and Uustalu [2002], dependent types make type preservation hard to prove, even if the source language is free from control effects. Specifically, their result shows that, the standard CPS translation, which uses a fixed answer type, fails to preserve typing when the language has strong Σ types or dependent elimination of sum types. Fifteen years later, a breakthrough was made by Bowman et al. [2017]: they solved the issue with Σ types using a polymorphic answer type, and conjectured that the technique scales to dependent elimination. Unfortunately, the translation of Bowman et al. does not allow implementing any control effects, as it requires every term to use its continuation in a trivial way.

We present λsΠr, a dependently typed language with Danvy and Filinski’s shift and reset operators. Our work extends the above mentioned studies in three ways. First, while Herbelin [2012] considered control operators for undelimited continuations, we discuss operators for delimited continuations. Unlike undelimited ones, delimited continuations behave like ordinary functions, and thus can be used in a more flexible manner. This gives rise to type dependency associated with continuations, which Herbelin did not account for. Second, we scale the result of Bowman et al. [2017] to effectful languages by making the answer type non-uniformly polymorphic. It turns out that the technique of Bowman et al. works perfectly for languages with control effects, as long as no type depends on impure terms. Thirdly, our language includes inductive datatypes, which are supported in neither study. While inductive types can be simulated in the plain Calculus of Constructions [Pfenning and Paulin-Mohring 1990], it is more convenient to directly incorporating them, and this extension is non-trivial as implied by the result of Barthe and Uustalu [2002].

This paper makes the following contributions.

• We identify a set of restrictions that are necessary for integrating delimited control into a dependent calculus. In particular, we show that the notion of answer types gives rise to two
issues that have not been discussed in the literature, namely dependency of continuations, and dependency on continuations.

- We give a sound type system of $\lambda_{\Pi}^{s/r}$, by extending the type-and-effect system of Asai and Kameyama [2007] with dependent types. The idea is to use different typing judgments for pure and impure terms, which help us rule out invalid type dependency.
- We define a CPS translation of $\lambda_{\Pi}^{s/r}$ into a shift/reset-free language, and prove that the translation preserves typing. A key observation is that, in the presence of inductive types, type preservation requires a non-standard constraint for inductive definitions in the source language, as well as some additional/modified typing rules in the target.
- We discuss several language extensions that are necessary for scaling our result to the full Calculus of Inductive Constructions (CIC) [Werner 1994], the underlying calculus of Coq. We describe the challenges with those extensions, and conjecture that some of them can be solved by adopting an optimizing CPS translation.

In the rest of the paper, we give an introduction to shift and reset (Section 2), and observe the issues with using these operators in combination with dependent types (Section 3). We then present our language (Section 4), and give a CPS translation (Section 5). To demonstrate the practical impact of our study, we show two example programs that use control operators and dependnet types in a meaningful way (Section 6). Lastly, we describe the problems we will encounter when extending the language with advanced features from CIC (Section 7). Due to space limitations, we refer the reader to our technical appendix [Cong and Asai 2018] for elided proofs and detailed description of the languages we build in this paper.

2 SHIFT, RESET, AND ANSWER-TYPE MODIFICATION

The shift and reset operators [Danvy and Filinski 1990] are language constructs for handling delimited continuations. Intuitively, shift captures a continuation, making it available to the program, whereas reset delimits a continuation, telling shift the extent of relevant computation.

Let us look at example (1) below, where $S$ and $\langle \rangle$ denote shift and reset, respectively:

$$(1) \quad 1 :: \langle 2 + Sk : \text{Int} \to \text{Int}.(k(k 3)) :: \text{nil} \rangle$$

$$= 1 :: \langle((k(k 3)) :: \text{nil}) [\lambda x : \text{Int}.\langle 2 + x \rangle/k] \rangle = 1 :: \langle 7 :: \text{nil} \rangle = 1 :: 7 :: \text{nil}$$

The shift operator captures the context $2 + []$ (where $[]$ represents the hole), and turns it into a function $\lambda x : \text{Int}.\langle 2 + x \rangle$. Then it "shifts" the control to its body $(k(k 3)) :: \text{nil}$, with $k$ bound to the captured continuation. After the body has reduced to 7, we remove the surrounding reset, and obtain a two-element list $1 :: 7 :: \text{nil}$. Note that elimination of a shift operator is only possible when it is surrounded by a reset. For example, the shift construct in (1) does not take step on its own, and the whole subterm inside the reset clause is also a stuck term.

An interesting thing that happens when programming with shift and reset is that evaluation of programs can change the answer type, i.e., the return type of the nearest enclosing reset. Let us observe example (1) again. Before evaluation, the reset clause appears to return an integer, because its body takes the form $2 + x$. However, if the reset clause had type $\text{Int}$, it would not compose with the consing context $1 :: []$. Why does example (1) make sense as a whole? By looking at the reduction sequence, we find that elimination of the shift operator replaces the addition by a list $(k(k 3)) :: \text{nil}$. After this step, the reset clause no longer looks like an integer; it has turned into a list. This phenomenon is called answer-type modification, ATM for short. In our particular example, we say that the shift operator causes ATM from $\text{Int}$ to $\text{List Int}$.

---

1Readers curious to see examples may jump from Section 3 to Section 6 and return to the technical sections later.
The ability of modifying answer types is specific to *impure* terms, *i.e.*, those terms whose evaluation involves accessing of contexts. A *shift* construct is trivially an impure term, and any term that contains an “active” *shift*, which is not surrounded by a *reset* or a λ, is also impure. In contrast, *pure* terms do not touch their surroundings, and hence cannot change the answer type. Syntactically pure terms include values, such as variables and functions, since no evaluation can happen. A *reset* clause is also a pure term, because any control effect in its body has no access to the context outside the *reset* boundary. Complex terms are pure if their subterms are all pure: *e.g.*, 1 + 2 and (λ x : Int. x) (3 * (Sk : Int → Int. 4)) are both pure terms.

In this paper, we use different typing judgments for pure and impure terms, and distinguish between them by looking at their derivation. For pure terms, we use a judgment of the form Γ ⊢ p e : A [Asai and Kameyama 2007], which consists of usual components but has a subscript telling us that e is a pure term. For impure terms, we use a five-place judgment Γ; α ⊢ e : A; β [Danvy and Filinski 1989], which carries two additional types α and β. Among these, α is the initial *answer type*, representing what is inside the surrounding *reset* before evaluation. The other type β is the final *answer type*, which tells us what the *reset* returns after evaluation. The additional information allows us to type check programs like example (1): since we know the body of *reset* changes the answer type from Int to List Int, we can safely cons 1 to the result of the *reset* clause.

We close this section by presenting the typing rules of *shift* and *reset* in a simply typed calculus [Asai and Kameyama 2007], which would give the reader a better idea of how these operators impose and remove requirements on answer types:

\[ \frac{\Gamma; k : A \rightarrow \alpha; B \vdash e : B; \beta}{\Gamma; \alpha \vdash Sk : A \rightarrow \alpha. e : A; \beta} \quad (\text{SHIFT}) \]
\[ \frac{\Gamma ; B \vdash e : B; A}{\Gamma \vdash \langle e \rangle : A} \quad (\text{RESET}) \]

Let us first look at *(SHIFT)*. We see that the initial answer type \(\alpha\) in the conclusion comes from the return type of the captured continuation \(k\). This intuitively makes sense, since \(k\) represents the context inside the surrounding *reset*. We next find that the final answer type \(\beta\) coincides with that of the body \(e\). This is also a natural consequence, because when we eliminate \(Sk : A \rightarrow \alpha. e\), we replace the body of the surrounding *reset* with \(e\). Now we observe *(RESET)*. The premise says that when we evaluate the body \(e\) in an empty context (whose hole and return types are both \(B\)), we obtain a value of type \(A\). Since we regard the value of \(e\) as the value of \(\langle e \rangle\), we conclude \(\langle e \rangle\) is a pure term of type \(A\).

### 3 MIXING CONTROL EFFECTS AND DEPENDENCY

What happens if we use *shift* and *reset* in a dependently typed language? This section studies a series of examples showing the challenges with combining these components.

#### 3.1 Types Dependent on Impure Terms

Let us begin by discussing what kind of terms can appear in types and what kind cannot. Compare the following programs, where \(\mathbb{N}\) represents the type of natural numbers and \(L\ n\) denotes the type of lists containing \(n\) natural numbers (we use the conventional list notation for brevity):

\[
(2) \ (\lambda x : L. (1 + Sk : \mathbb{N} \rightarrow \mathbb{N}. 2). x) \ [0; 1] \\
(3) \ (\lambda x : L. (1 + Sk : \mathbb{N} \rightarrow \mathbb{N}. 2). x) \ [0; 1]
\]

When deciding equality between two dependent types, we first normalize every term that the types depend on, and then compare the results. In example (2), the list type is indexed by a term that uses control constructs, but we can still apply the standard method to type check the application: the index \(1 + Sk : \mathbb{N} \rightarrow \mathbb{N}. 2\) reduces to 2, and since the list \([0; 1]\) has type \(L\ 2\), it is a valid argument to the identity function.
Example (3) differs from example (2) in only one way: it lacks the `reset` clause surrounding
the list index. To type check the application, we have to normalize \(1 + Sk : N \rightarrow N\) 2 and see
whether the result is 2 or not. However, due to the absence of `reset`, we cannot normalize this
index as we did in example (2)—remember that elimination of `shift` is only possible when there is
a surrounding `reset`. What this means is that the normalize-and-compare method does not apply
to types dependent on impure terms.

Aside from type checking, allowing dependency on impure terms does not seem to be beneficial
to the programmer either. Consider example (3) again. In the body of the identity function, we
know that \(x\) has type \(L (1 + Sk : N \rightarrow N, 2)\), but we still do not know how many elements \(x\)
contains, since the index does not reduce to a number. In general, impure type indices do not help
us reason about programs, hence there is no advantage in having them.

The above discussion tells us that types should not depend on impure terms; they must depend
only on terms that we can locally normalize to a value.

### 3.2 Continuations Dependent on Their Argument

We next turn our viewpoint to dependencies associated with continuations. In a dependently typed
language, function types take the form \(\Pi x : A. B\), where \(B\) can depend on the argument \(x\). Since
continuations are functions, we will naturally wonder whether they can have a dependent type as
well. Let us look at examples (4) to (7), assuming \(mk\):

\[
\begin{align*}
(4) & \quad \text{\langle mk-lst : } \Pi x : N. L x \text{ append : } \Pi m : N. \Pi n : N. L m \rightarrow L n \rightarrow L (m + n) : \\
(5) & \quad \text{\langle mk-lst : } \Pi x : N. L x \text{ append : } \Pi m : N. L x : N \text{.} \\
(6) & \quad \text{\langle mk-lst : } \Pi x : N. L x : N \text{.} \\
(7) & \quad \text{\langle mk-lst : } \Pi x : N. L x : N \text{.} 
\end{align*}
\]

In all the four programs, the captured continuation \(k\) is the application of `mk-lst`, and thus has a
dependent function type \(\Pi x : N. L x\). To type the `shift` constructs in these programs, we have to
slightly modify Asai and Kameyama’s `(SHIFT)` rule, by replacing the assumption \(k : A \rightarrow \alpha\) with
\(k : \Pi x : A. \alpha\). Then, the `shift` constructs will have a derivation of the following form:

\[
\begin{array}{c}
\bullet, k : \Pi x : N. L x; B + e : B; \beta \\
\bullet ; L x + Sk : \Pi x : N. L x. e : N; \beta \\
\end{array}
\]

(\text{(SHIFT)})

Recall that the initial answer type of `shift` coincides with the return type of the captured continu-
ation. In the above examples, the continuation returns a value of type \(L x\). Clearly, this type does
not serve as a valid initial answer type, because it has a free occurrence of variable \(x\).

The reader might think that we could close off this variable by substituting the actual argument
we pass to the continuation. This works for example (4): when we type the `shift` clause, we see
that \(k\) is called exactly once with argument 1, hence we could "instantiate" \(\alpha\) to a closed type \(L 1\).
However, type instantiation fails in the rest of the examples, where \(k\) is used multiple times with
different arguments (example (5)), or is not used at all (example (6)), or is applied to an argument
outside the `shift` construct (example (7)).

Summing up these observations, we conclude that continuations must be non-dependent func-
tions. Put differently, we must use the `shift` operator only in a non-dependent context.

### 3.3 Answers Dependent on Continuations

Having discussed dependency of continuations, let us look at types dependent on continuations.
Consider the following program:

\[
(8) (1 + Sk : N \rightarrow N. mk-lst (k 2))
\]

The typing rule of Asai and Kameyama gives us the following derivation for the `shift` construct:
Recall that the final answer type of a shift construct is determined by what its body evaluates to. In example (8), the body evaluates to a value of type L (k 2). Similarly to the situation with dependent continuations, this type does not serve as a valid final answer type of the shift construct, because it contains a free continuation variable k.

Just like we tried to obtain a closed initial answer type by substituting the actual argument, we might attempt to obtain a closed final answer type by substituting the actual continuation. In the case of example (8), it is not hard to see that k will be the function λ x : N. ⟨1 + x⟩, hence we could instantiate L (k 2) to a closed type L ((λ x : N. ⟨1 + x⟩) 2). However, in the general case, figuring out what continuation is to be captured requires evaluating the whole program, which is not allowed during type checking in a statically typed language. Moreover, there is no guarantee that we have a candidate continuation to substitute for every continuation variable, because well-typedness of terms does not require shift operators to have a matching reset.

Wrapping up the discussion, we conclude that the final answer type of a shift construct should not refer to the continuation variable.

4 \( \lambda^s/r \): A DEPENDENTLY TYPED LANGUAGE WITH SHIFT AND RESET

This section presents \( \lambda^s/r \), a call-by-value\(^2\) language with dependent types and the shift/reset operators. We build \( \lambda^s/r \) under the three principles we discussed in the previous section: (i) types do not depend on impure terms; (ii) continuations are non-dependent functions; and (iii) final answer types do not refer to continuations. To highlight these principles, we make the following design decisions:

- We consider a restricted form of inductive datatypes. Specifically, we require that every datatype constant is indexed by one term, and every term constructor receives one argument, instead of supporting multi-arity datatypes.
- We only account for the dependency axis of the lambda cube. That is, we do not include type abstractions or type operators.

The former decision, which we borrow from Jia et al. [2010], helps us avoid the complication caused by effectful arguments, while keeping all the challenges with mixing control effects and dependency. We describe how to relax this restriction in Section 4.5. The latter design choice also does not affect our language design principle, since shift and reset are compatible with polymorphism [Asai and Kameyama 2007], and we assume that they only deal with term-level contexts, i.e., type-level computation is free from control effects.

4.1 Syntax

We define the syntax of \( \lambda^s/r \) in Figure 1. The first category, \( \Psi \), denotes signatures, which are sets of inductive definitions. Each inductive definition is a pair of the form \( \text{Ind}(D : \kappa, \{c_i : C_i\}) \). Here, D is a datatype constant, and \( \kappa \) is the arity of D, which, in our single-index setting, is a functional kind of the form \( \Pi a : A. * \). The c_i's denote the constructors of D, each of which has type C_i. We provide some examples of (multi-arity) inductive data in Section 4.6, which will help the reader figure out what each component of an inductive definition stands for.

Next, we have typing environments \( \Gamma \). Among the two extension cases, \( \Gamma, x = e : A \) represents context extension by a let expression.

\(^2\)We choose this evaluation strategy because shift and reset are mainly studied in a call-by-value setting. Section 7 provides a brief discussion on adopting a call-by-name semantics.
\[
\begin{align*}
\Psi & ::= \bullet | \Psi, \text{Ind}(D : \kappa, \{c_i : C_i\}) \\
\Gamma & ::= \bullet | \Gamma, x : A | \Gamma, x = e : A \\
\kappa & ::= * | \Pi x : A. * \\
A, \alpha & ::= \text{Unit} | D e | \Pi x : A. B | \Pi x : A. \alpha \parallel B \parallel \beta \\
v & ::= () | x | \lambda x : A. e | \text{rec } f (x : A). e | c_i \nu \\
e & ::= \nu | e\ e | \text{let } x = e : A \text{ in } e | c_i e \\
& | \text{match } e \text{ as } x \text{ in } D \text{ a return } P \text{ with } \{c_i \gamma_i \rightarrow e_i\} \\
& | \text{Sk: A. e } \langle e \rangle
\end{align*}
\]

Fig. 1. \(\lambda^{s/r}_\Pi\) Syntax

Kinds \(\kappa\) are either the base kind \(*\), which is the type of types, or the functional kind \(\Pi x : A. *\), which represents the arity of single-index datatype constants. Note that since \(*\) can never depend on \(x\), the type is equivalent to the non-dependent function type \(A \rightarrow \ast\). The dependency arises when we relax the single-index restriction, as we will see later.

Following the convention from the continuations literature, we usually use \(\alpha, \beta, \gamma\) to evoke answer types and \(A, B, C\) to evoke other types. The first component of types is the unit type \(\text{Unit}\), which we use for unused type indices as well as constructor arguments. For instance, the type constant for natural numbers does not require any index, but we can define it as \(\mathbb{N}()\) in our single-index setting. The second element \(D e\) represents an inductive datatype indexed by term \(e\). Then we have two kinds of function types. The standard one, \(\Pi x : A. B,\) is the type of functions whose body is a pure term of type \(B\). The other one, \(\Pi x : A. \alpha \parallel B \parallel \beta,\) is for functions with an impure body, which changes the answer type from \(\alpha\) to \(\beta\). Note that types \(B, \alpha,\) and \(\beta\) may depend on variable \(x\). Note also that we do not have variants of function types whose domain carries answer types, since functions can only receive pure values in a call-by-value language.

Lastly, terms \(e\) consist of values and computations. The former include the unit value, variables, functions, (guarded) recursive functions, and constructors applied to a value. Non-values are application, the let expression, inductive data, pattern matching, and the \(\text{shift}\) and \(\text{reset}\) constructs. In the pattern matching construct, \(P\) is the return type, which may depend on variables \(a\) (representing the index) and \(x\) (representing the scrutinee).

### 4.2 Evaluation

In Figure 2, we present rules for evaluation and equivalence. We have two kinds of evaluation contexts \(E\) and \(F,\) where the latter denotes those contexts whose hole is not surrounded by a \(\text{reset}\) (notice that we do not have \(\langle F \rangle\)). Both \(E\) and \(F\) feature a call-by-value, left-to-right evaluation strategy. For example, when we have an application \(e_0\ e_1,\) we first evaluate the function \(e_0,\) which corresponds to the context \(E\ e,\) then evaluate the argument \(e_1,\) which corresponds to \(v\ E\).

Single-step reduction is defined as the compatible closure of \(\Gamma \vdash e \triangleright e',\) where \(\triangleright\) is one of the seven instances found in the figure. By \(e [v/x],\) we mean capture-avoiding substitution of \(v\) for \(x\) in \(e.\) The first rule, \(\delta\)-reduction, allows us to replace a variable with the value it is bound to. Note that we should not allow replacing a variable with a computation, since our language is call-by-value. A \(\text{shift}\) construct can be eliminated when it is surrounded by a context of the form \(\langle F[\_] \rangle\). Since the context is \(F\) instead of \(E,\) we know that the \(\text{reset}\) is the innermost one. On the right-hand side, we find that \(k\) is replaced by the function \(\lambda x : A. (F[x]).\) Observe that the function’s body is a pure term (since it is a \(\text{reset}\) clause); this is why the annotation of \(k\) uses a pure function type. The
last rule tells us that a reset clause surrounding a value is redundant, because values do not have control effects. We will write $\Gamma \vdash e \triangleright \star t$ to mean the reflexive and transitive closure of $\Gamma \vdash e \triangleright t$.

At the bottom of Figure 2, we define equivalence of $\lambda^*/r_\Pi$. The rule ($\equiv$) says: two expressions (terms, types, or kinds) are equivalent if they reduce to the same expression in the $\triangleright \star$-relation. The other two rules define $\eta$-equivalence. Notice that our equivalence is untyped, i.e., we do not require the left- and right-hand sides to have the same type, or even be well-typed. This design choice helps us avoid a difficulty with the CPS translation; we will elaborate this in Section 5.3.

4.3 Typing Rules

In Figures 3 - 5, we present typing rules. These rules play a key role in our language, as they impose the three restrictions on type dependency. Following Asai and Kameyama [2007], we use $\vdash_p$ to type pure terms and $\vdash$ to type impure terms. We also keep signatures implicit in typing judgments.

The first two rules in Figure 3 define well-formed signatures. In (ExtendSig), the arity takes the form $\Pi a : A. \ast$, which means $D$ receives one index. Similarly, the shape of the constructor type $\Pi y_i : B_i. D u_i$ tells us that $c_i$ receives one argument. We type check the former in an empty environment, and the latter with the knowledge $D : \Pi a : A. \ast$, which is implicit in the typing rule. The last premise safe($D, B_i$) tells us that the datatype constant $D$ appears safely in type $B_i$.  

Proc. ACM Program. Lang., Vol. 2, No. ICFP, Article 69. Publication date: September 2018.
Handling Delimited Continuations with Dependent Types

The safety condition is a stronger version of the strict positivity condition [Coquand and Paulin 1990], which disallows \( D \) to appear in the domain, co-domain, and the initial answer type of a function type. We use this condition to ensure that the image of the CPS translation fits within the strictly positive fragment. If we adopt the standard positivity condition, we will end up with either a non-type-preserving translation (if we keep the positivity condition in the target), or an inconsistent target language (if we turn off positivity check).

The next three rules in Figure 3 define well-formed typing environments. (Extend1) is the standard rule for context extension, whereas (Extend2) has a premise that \( e \) is a pure term. This is because \( \lambda s/r \Pi \) is a call-by-value language, where we regard variables as values. Since values are pure, it is not valid to bind \( x \) to an impure \( e \).

The rest of the rules are for kinds and types. Observe that rule (Data) requires a pure index \( e \). This enforces our first restriction “no impure terms in types.” The rules for the functions types, (PrT1) and (PrT2), check if the co-domain \( B \) as well as the answer types \( \alpha \) and \( \beta \) are well-formed under the extended context \( \Gamma, x : A \).

The typing rules of terms are defined in Figures 4 - 5. Notice that complex terms have multiple rules differing in the purity of subterms. This is in contrast to the type system of Asai and Kameyama [2007], where we have one rule per language construct, with all subterms being impure (note that pure terms can be casted into impure ones via (Exp) in Figure 5). If we take Asai and Kameyama’s approach, all terms other than values and \( \text{reset} \) constructs are regarded impure, even if they have no real control effect (i.e., all effects are from purity casting by (Exp)). Having less pure terms does not affect typability in a simply typed setting, but it does in \( \lambda s/r \Pi \), because impure terms are not allowed to appear in types. Therefore, we design the type system in such a way that it generates as many \( \vdash_p \)-judgments as possible.
### Terms $\Gamma \vdash e : A$, $\alpha \vdash e : A; \beta$

| Rule | Description |
|------|-------------|
| $\vdash \Gamma$ | $() : \text{Unit}$ (UNIT) |
| $\vdash \Gamma$ | $\Gamma \vdash x : A \in \Gamma$ or $x = e : A \in \Gamma$ (VAR) |
| $\Gamma, x : A \vdash e : B$ | $\Gamma \vdash \lambda x : A. e : \Pi x : A. B$ (ABS1) |
| $\Gamma, x : A; \alpha \vdash e : B; \beta$ | $\Gamma \vdash \lambda x : A. e : \Pi x : A. \alpha \parallel B \parallel \beta$ (ABS2) |
| $\Gamma, f : \Pi x : A. B, x : A \vdash e : B$ guard($f, x, e, \{\}$) | $\Gamma \vdash \text{rec } f (x : A). e : \Pi x : A. B$ (REC1) |
| $\Gamma, f : \Pi x : A. \alpha \parallel B \parallel \beta, x : A; \alpha \vdash e : B; \beta$ guard($f, x, e, \{\}$) | $\Gamma \vdash \text{rec } f (x : A). e : \Pi x : A. \alpha \parallel B \parallel \beta$ (REC2) |
| $\Gamma \vdash e_0 : \Pi x : A. B$ $\Gamma \vdash e_1 : A$ | $\Gamma \vdash e_0 \ e_1 : B[e_1/x]$ (APP1) |
| $\Gamma ; \alpha \vdash e_0 : \Pi x : A. B$ $\Gamma ; \alpha \vdash e_1 : A$ | $\Gamma ; \alpha \vdash e_0 \ e_1 : B[e_1/x] ; \beta$ (APP2) |
| $\Gamma \vdash e_0 : \Pi x : A. \alpha \parallel B \parallel \beta$ $\Gamma \vdash e_1 : A$ | $\Gamma \vdash [e_1/x] \ e_0 \ e_1 : B[e_1/x] ; \beta$ (APP5) |
| $\Gamma ; \alpha \vdash e_0 : \Pi x : A. \alpha \parallel B \parallel \beta$ $\Gamma ; \alpha \vdash e_1 : A$ | $\Gamma ; \alpha \vdash e_0 \ e_1 : B[e_1/x] ; \gamma$ (APP6) |
| $\Gamma ; \beta \vdash e_0 : \Pi x : A. \alpha \parallel B \parallel \beta$ $\Gamma ; \alpha \vdash e_1 : A ; \gamma$ | $\Gamma ; \beta \vdash e_0 \ e_1 : B[e_1/x] ; \gamma$ (APP7) |
| $\Gamma \vdash e_0 : \Pi x : A. \alpha \parallel B \parallel \beta$ $\Gamma \vdash e_1 : A$ | $\Gamma ; \gamma \vdash e_0 : A \rightarrow \alpha \parallel B \parallel \beta$ $\Gamma ; \beta \vdash e_1 : A$ (APP8) |
| $\Gamma \vdash e_1 : A$ $\Gamma, x = e_1 : A \vdash e_2 : B$ | $\Gamma \vdash \text{let } x = e_1 : A \text{ in } e_2 : B[e_1/x]$ (LET1) |
| $\Gamma \vdash e_1 : A$ $\Gamma, x = e_1 : A; \alpha \vdash e_2 : B; \beta$ | $\Gamma \vdash [e_1/x] \ e_0 \ e_1 : B[e_1/x] ; \beta [e_1/x]$ (LET2) |
| $\Gamma ; \alpha \vdash e_1 : A; \beta$ $\Gamma, x : A \vdash e_2 : B$ | $\Gamma ; \alpha \vdash e_1 : A \text{ in } e_2 : B[e_1/x]$ (LET3) |
| $\Gamma ; \beta \vdash e_1 : A; \gamma$ $\Gamma, x : A; \alpha \vdash e_2 : B; \beta$ | $\Gamma ; \beta \vdash e_1 : A; \gamma$ (LET4) |

Fig. 4. Typing Rules
We begin with the rules for the familiar \(\lambda\)-terms and the let expression. \((\text{Var})\) is standard, except that we use the \(\vdash_p\)-judgment to explicitly state that it is a pure term. Functions are also pure, and are given different types depending on whether their body is pure or not. The side condition \(\text{guard}(f, x, e, \{\})\) in rules \((\text{Rec1})\) and \((\text{Rec2})\) is called the guard condition [Giménez 1995], which ensures termination of recursive functions. Intuitively, we say \(f\) is guarded in \(e\) when every occurrence of \(f\) in \(e\) receives a structurally smaller argument. This requires that \(e\) pattern matches on the argument \(x\) of \(f\) and calls \(f\) only with the constructor argument \(y_i\), which we collect in \(V\). The formal definition of \(\text{guard}(f, x, e, V)\) looks as follows (elided cases are defined by straightforward induction on the structure):

- When \(f\) does not appear in \(e\), \(\text{guard}(f, x, e, V)\) holds without any other condition.
- When \(e = f\ e_1\), \(e_1 \in V\).
- When \(e = \text{match } x \text{ as } x' \text{ in } D \text{ a return } P \text{ with } \{c_i \ y_i \rightarrow e_i\}\), \(\text{guard}(f, x, e_1, V \cup \{y_i\})\).

A function application is pure if the function, the argument, and the body of the function are all pure, as in \((\text{App1})\). Otherwise, it is an impure term with a \(\vdash\)-judgment. By comparing the eight rules, we find that dependency of the result type and impureness of the argument cannot coexist. That is, when the function is dependent, we require a pure argument, and when the argument is impure, we require a non-dependent function. We also find that certain answer types have duplicated occurrences. For instance, in \((\text{App4})\), the final answer type of \(e_1\) coincides with the initial answer type of \(e_0\). The coincidence is determined by the evaluation strategy and the semantics of the shift operator; we encourage the interested reader to visit Section 2 of our technical appendix [Cong and Asai 2018]. The rules for the let expression are defined based on the same idea. That is, the type of body \(e_2\) may depend on \(x\) only when \(e_1\) is pure, which is enforced by the well-formedness premises \(\Gamma \vdash B : *\) and \(\Gamma \vdash x : *\) in rules \((\text{Let3})\) and \((\text{Let4})\). Notice that in the non-dependent cases, we extend the context with \(x : A\) instead of \(x = e_1 : A\), since variables should not be bound to an impure term.

We next look at the rules for inductive data in Figure 5. \((\text{Const1})\) and \((\text{Const2})\) deal with an application-like construct, where the function part and its body are both pure. Therefore, the premises are similar to what we have in \((\text{App1})\) and \((\text{App2})\): the former rule requires a pure argument since the constructor has a dependent type, while the latter requires a non-dependent constructor since the argument is impure. The pattern is present in the rules for elimination constructs as well: dependency of the return type \(P\) is allowed only when the scrutinee \(e\) is a pure term.

Control constructs have one single rule each, since their purity does not depend on the subterm. Observe the premises of rule (\(\text{Shift}\)). When type checking the body \(e\), we assume \(k\) has a non-dependent arrow type \(A \rightarrow \alpha\), reflecting our design principle “continuations are non-dependent functions.” We also require that \(\beta\) is well-formed under a context where \(k\) is not available, enforcing the restriction “final answer types do not refer to continuations.” The rule \((\text{Reset})\) is exactly the same as its simply typed counterpart.

Lastly, we have rules for changing the type and purity of terms. The conversion rules let us regard a term of type \(A\) as being of type \(B\) when \(A \equiv B\). Note that \((\text{Conv2})\) further allows us to convert answer types. Rule \((\text{Exp})\) casts a pure term into an impure one by instantiating answer types. We use this rule when \(e\) typing a pattern matching whose branches have different purities. In \((\text{Match2})\) and \((\text{Match3})\), we require that the branches are uniformly impure, but the matching construct still makes sense when some of the branches are pure, as long as all the impure branches do not involve ATM. Using \((\text{Exp})\), we can type such a construct by casting pure branches into impure ones, with the initial and final answer types being equal. The rule also allows us to type the shift/reset constructs that have a pure body.
4.4 Type Soundness

Following Wright and Felleisen [1994], we prove soundness of our type system by showing the preservation and progress properties. Preservation states that for any well-typed, reducible term $e$, it has the same purity and type after reduction.

**Theorem 4.1 (Preservation).**

1. If $\Gamma \vdash e : A$ and $\Gamma \vdash e \rightarrow^* e'$, then $\Gamma \vdash e' : A$.
2. If $\Gamma; \alpha \vdash e : A; \beta$ and $\Gamma \vdash e \rightarrow^* e'$, then $\Gamma; \alpha \vdash e' : A; \beta$.

The proof is by induction on the derivation of $e$. As usual, we need the substitution lemma, which states that substitution preserves types, and several inversion lemmas, which let us invert typing relations.

The progress property says that a well-typed pure term is either a value or a reducible term:

**Theorem 4.2 (Progress).** If $\vdash e : A$, then either $e$ is a value, or $\Gamma \vdash e \rightarrow^* e'$ for some $e'$.

The proof is again by induction on the derivation of $e$. We only account for pure terms because impure terms are not reducible in general. Proving the value cases is trivial; note that the variable
case is impossible since no variable is well-typed under an empty context. In the non-value cases, we use the following canonical forms lemma:

**Lemma 4.3 (Canonical Forms).**

1. If $v$ has type $\Pi x : A. B$ or $\Pi x : A. \alpha \parallel B \parallel \beta$ in an empty environment, then $v$ is of the form $\lambda x : A'. e$ or $\text{rec } f (x : A'). e$.
2. If $v$ has type $D u$ in an empty environment, then $v$ is of the form $c_i v'$.

This lemma makes the $\beta$, $\mu$, and $\iota$ reduction cases go through. The reset case needs a progress lemma for impure terms, which is similar to Theorem 4.2 but has an extra possibility that $e$ is a stuck term of the form $F[Sk : A \rightarrow \alpha. e']$. The lemma is proved mutually with Theorem 4.2.

### 4.5 Scaling to General Inductive Datatypes

To write programs that use dependent types in an interesting way, we need a general form of inductive datatypes, where type constants and constructors can take an arbitrary number of arguments. In this subsection, we briefly describe how to extend $\lambda^{s/r}$ with this feature.

First, we modify the syntax. Functional kinds now take the form $\Pi x_1 : A_1. x_2 : A_2. \ldots x_n : A_n. \ast$, where each $A_i$ may depend on preceding variables $x_j$ with $j < i$. When type constant $D$ has such a type, it takes in a sequence of indices and forms a datatype $D \bar{\iota}$. We extend functional types and constructor application in a similar way. When pattern matching an inductive datum that has $n$ indices, the return type $P$ depends on $n + 1$ terms, and the type following the in keyword takes the form $D \bar{\alpha}$, where $\alpha_i$ is a sequence of variables.

Next, we revise evaluation contexts and reduction rules. When evaluating constructor application $c_i \bar{\iota}$, we evaluate the arguments $\bar{\iota}$ from left to right, and once all of them have reduced to values $\bar{v_i}$, we can perform $\iota$ reduction (selecting a branch of pattern matching), which will sequentially substitute the values for pattern variables.

We then adapt the typing rules. Extending the formation rule (DATA) of datatypes is easy: $D \bar{\iota}$ is well-formed when all indices $\iota_i$ are pure terms. The rules for constructor application, however, require some non-trivial work. Specifically, we have to make sure that:

1. the domain and result types only depend on pure arguments; and
2. the sequence of initial and final answer types of impure arguments accords with evaluation.

To meet the former condition, we require that if the constructor’s $i$’th domain is $x_i : B_i$ and the $i$’th argument is an impure term, $x_i$ does not appear in the succeeding domains or the co-domain. The latter requirement generalizes the chaining of the answer types we observed in Section 4.3.

Precisely, we require that the initial answer type of an impure argument coincides with the final answer type of the closest succeeding impure argument (if any). Further details of the multi-arity extension can be found in our technical appendix [Cong and Asai 2018].

### 4.6 Examples

By allowing multi-argument datatypes, we can define natural numbers and indexed lists as follows:

\[
\text{Ind}(N : \ast, \{ z : \mathbb{N}; \text{succ} : \mathbb{N} \rightarrow \mathbb{N} \})
\]

\[
\text{Ind}(L : \Pi a : \mathbb{N}. \ast, \{ \text{nil} : L z; \text{succ} : \Pi n : \mathbb{N}. \mathbb{N} \rightarrow L n \rightarrow L (\text{succ } n) \})
\]

The datatype constant $L$ has arity $\Pi a : \mathbb{N}. \ast$, meaning that it is indexed by a natural number. The index is instantiated to $z$ in the empty case, and to $\text{succ } n$ in the cons case $:: n h t$.

We next define the append function using these datatypes:

\[
\text{rec } f (n_1 : \mathbb{N}, n_2 : \mathbb{N}, l_1 : L n_1, l_2 : L n_2).
\]

\[
\text{match } l_1 \text{ as } _\_ \text{ in } L \text{ a return } L (a + n_2) \text{ with } \text{nil} \rightarrow l_2 | :: n h t \rightarrow :: (n + n_2) h (f n n_2 t l_2)
\]
The pattern matching construct tells us that the return type depends on the index of \( I_1 \) but not the list itself. In the first branch, we check if \( I_2 \) has type \( L(z + n_2) \) or not. It is easy to see that this is the case, since \( I_2 : L(n_2) \) and \( n_2 \equiv z + n_2 \). The second branch requires a term of type \( L(suc n + n_2) \), whereas we have \( :: (n + n_2) h (f n n_2 t I_2) \) of type \( L(suc(n + n_2)) \). By the definition of addition, the former type reduces to the latter, hence type checking again succeeds.

5 CPS TRANSLATING \( \lambda^{s/r}_\Pi \)

This section presents a call-by-value CPS translation from \( \lambda^{s/r}_\Pi \) into a pure language. In a simply typed setting, elimination of \( \text{shift} \) and \( \text{reset} \) are known to preserve typability, but it remains open whether this is the case in a dependently typed language\(^3\). Indeed, as mentioned in Section 1, CPS translations for dependent types are hard even when the source language is effect-free. Nevertheless, here we show that our “no dependency on impure terms” policy, together with answer-type polymorphism of pure terms, gives us a type-preserving CPS translation of \( \lambda^{s/r}_\Pi \). In the next subsection, we review the challenges with CPS and dependent types, and sketch our solution.

5.1 CPS Translating Dependent Types: Challenge and Solution

The difficulties with CPS translating dependently typed languages were first noticed by Barthe and Uustalu [2002]. They showed that the call-by-name, double-negation-based translation does not preserve typing in the presence of \( \Sigma \) types and sum types with dependent elimination. Recently, Bowman et al. [2017] reported that a similar problem arises when translating function application by forcing call-by-value evaluation. These failure cases share the same pattern: the type of the whole term \( e \) depends on its subterm \( e' \), and evaluation of \( e \) requires evaluation of \( e' \). To see why this pattern makes things difficult, let us look at the translation of application \( e_0 e_1 \) in a pure, call-by-value language. Suppose the application has the following derivation:

\[
\Gamma \vdash e_0 : \Pi x : A. B \quad \Gamma \vdash e_1 : A
\]

(\text{App})

In the source language, we evaluate \( e_0 e_1 \) by first reducing \( e_0 \) to a value \( v_0 \) and then reducing \( e_1 \) to \( v_1 \). Then, we either perform \( \beta \)-reduction, or finish evaluation with \( v_0 \) \( v_1 \) if \( v_0 \) is a non-\( \lambda \). Notice that in either case, the post-evaluation term has type \( B[v_1/x] \), which depends on the value of \( e_1 \), while the original application has type \( B[e_1/x] \), which depends on the original argument. These types are equivalent since \( e_1 \triangleright v_1 \). In general, when we have dependent types, there may be a syntactic mismatch between the type we see in a derivation and the canonical type of the term we obtain after evaluation. However, typing rules do not need to explicitly mention the value each subterm evaluates to, because we can reason about type equality by simply evaluating subterms that appear in types.

Now we CPS translate \( e_0 e_1 \). The standard translation turns this application into the following term, where \( e^* \) and \( A^+ \) denote the CPS counterpart of term \( e \) and type \( A \), respectively\(^4\):

\[
\text{shift} k : (B[e_1/x])^+ \rightarrow \bot_e^+ (\lambda v_0 : (\Pi x : A. B)^+ . e_1^+ (\lambda v_1 : A^+. v_0 v_1 k))
\]

The application of \( e_0^+ \) to its continuation represents evaluation of \( e_0 \) in the source language, and similarly for \( e_1^+ \). Since both \( e_0 \) and \( e_1 \) are pure, we know that their CPS counterparts use their continuation in only one way: they pass their result to the continuation. This implies that variables

\(^3\)An exception would be Ilik [2012], who defines a CPS-like translation for a dependently typed language with an ATM-free variant of \( \text{shift} \) and \( \text{reset} \). Ilik cleverly avoids the challenges by distinguishing between dependent and non-dependent application, which, in essence, serves as our no-impure-terms restriction. However, Ilik does not mention why he makes this design choice, nor does he discuss difficulties with the translation that would arise when allowing arbitrary dependency.

\(^4\)Strictly speaking, the translations \( ^+ \) and \( ^* \) are defined on the derivation, as we will see in Section 5.3.
$v_0$ and $v_1$ stand for the values $e_0^+ \mathbf{e}$ and $e_1^+$ evaluate to. Thus, we can see that the translation turns the value of a term into a variable, more specifically, a $\lambda$-bound variable.

With these observations in mind, let us look at the application $v_0 \ v_1 \ k$. We find that while $v_0 \ v_1$ requires a $B^+ \ [v_1/x]$-accepting continuation, $k$ is a $(B \ [e_1/x])^+$-accepting continuation. The mismatch is caused by the fact that while $k$ is established before evaluation of $e_1$, it is used after $e_1$ has evaluated to a value, which is to be substituted for $v_1$. As Bowman et al. [2017] point out, solving this mismatch is non-trivial for two reasons. First, the fact that $v_1$ is bound by a $\lambda$ disrupts our knowledge that it represents a unique value, since a $\lambda$-abstraction can receive an arbitrary value. Second, the fixed answer type $\perp$ does not allow extracting the unique value from $e_1^+$, because all we can do with $e_1^+$ is to apply it to a continuation that never returns.

Bowman et al. [2017] solve the above issue by adopting a polymorphic answer type translation, which produces computations $e^+$ of type $\Pi \alpha : * \rightarrow (A^+ \rightarrow \alpha) \rightarrow \alpha$. This allows us to extract the value of $e^+$ in a type-safe manner: we first instantiate the answer type by $e^+ \ A^+$, and then pass the identity continuation of type $A^+ \rightarrow A^+$. Note that in the pre-CPS language, this corresponds to evaluating $e$ in an empty context (i.e., obtaining the value of $e$). The polymorphism also gives us a free theorem [Wadler 1989], which, in essence, represents the knowledge that the continuation passed to a CPS translated pure term receives one specific value. Bowman et al. internalize this free theorem by equipping the target language of the translation with additional equivalence and typing rules, and successfully prove the type preservation property for the difficult cases.

The polymorphic answer type translation, however, only applies to pure terms. The reason is that impure terms require a particular type of context and use it in an arbitrary manner. This means the value-extraction technique does not work for impure terms, and the free theorem does not hold of them either. Nevertheless, it is possible to define a type-preserving translation of $\lambda^k_{\Pi}$, because no type depends on impure terms. Specifically, this restriction ensures that whenever we have a source term whose type depends on subterm $e'$, we can give $e^+\alpha$ a polymorphic type, and reason about type equality just like Bowman et al. do. In Section 5.3, we show such a non-uniformly polymorphic CPS translation of $\lambda^k_{\Pi}$, which translates pure terms and impure terms in different ways.

### 5.2 Target Language

In Figure 6, we present $\lambda^k_{\Pi}$, the target language of the CPS translation. The language is different from $\lambda^{s/r}_{\Pi}$ in several ways. First, $\lambda^k_{\Pi}$ is a pure, impredicatively polymorphic language, which has type abstraction and instantiation but not control operators or function types carrying answer types. Second, the language has a call-by-name semantics, since type preservation of the CPS translation requires substitution of computations. Third, the language has a non-standard construct $e \ @ A \ e$, representing application of a CPS translated pure term to its continuation. As we will see later, we use this application to invoke the above-mentioned free theorem, which helps us prove type preservation. The last difference is that typing environments may contain equivalence information, which we need when typing the CPS image of pattern matching.

In the same figure, we present some of the rules defining evaluation of $\lambda^k_{\Pi}$ terms. Since $\lambda^k_{\Pi}$ is call-by-name, $\beta$ and $\zeta$ reduction can replace a variable by a computation. We also have some new rules for application of a polymorphic function.

The key changes are in the equivalence and typing rules, which we present in Figure 7. The new equivalence rule, $[\equiv\text{-}\text{Cont}]$, is the free theorem we obtain via answer-type polymorphism [Bowman et al. 2017], and tells us that applying a CPS translated pure term $e$ to a continuation is equivalent to first running $e$ with the identity continuation and then passing the result to the actual continuation. The theorem is known as continuation shuffling [Ahmed and Blume 2011; Bowman et al. 2017] and naturality [Thielecke 2003] in the literature.
Typing rules include the \( \lambda_{11}^{s/r} \) rules that deal with pure terms, \( i.e. \), those rules whose name ends in “1”, as well as rules that account for polymorphism. Rule [MATCH] extends the \( \lambda_{11}^{s/r} \) counterpart in that it type checks each branch with equivalence information: the actual index \( u \) is equivalent to the index \( u_i \) of the current branch, and the scrutinee \( e \) is equivalent to the current pattern \( c_i \ y_i \). We need this information because the CPS translation “distributes” the continuation of the whole matching construct, which depends on the scrutinee, to each branch, which depends on the pattern. Note that the equivalence can be inconsistent: for instance, when we pattern match against \( z : A \), we would extend the context with \( z \equiv \text{suc} \ n \) during type checking of the suc-branch. However, in this case we know that the suc-branch can never be chosen, hence we skip type checking of this branch by directly deriving what we want using [INCON].

Besides these changes, we have a new typing rule [T-CONT] [Bowman et al. 2017], which we use to type application of a CPS translated pure term \( e_1 \) to its continuation \( \lambda \ \nu : B. \ e_2 \). Observe that the premise lets us identify \( \nu \) with \( e_1 \ \text{B id} \) when typing \( e_2 \). This additional binding helps us “remember” the value of \( e_1 \) after the translation has introduced variable \( \nu \).

Note that while \( \equiv \text{-CONT} \) makes sense only if \( e_1 \) has a polymorphic type \( \Pi \alpha : \ast. (B \rightarrow \alpha) \rightarrow \alpha \), the rule does not have a typing premise as our equivalence is untyped. However, in \( \lambda_{11}^{k} \) we will never use equivalence between ill-typed terms, because the only way to use equivalence \( A \equiv B \) is via the conversion rule, which guarantees that \( A \) and \( B \) are well-formed. Notice that [T-CONT], which is the sole introduction rule of \( \@ \), requires \( e_1 \) to have a polymorphic type. This ensures that

---

5The problem is closely related to the challenges with commutative cuts [Boutillier 2012], where we move the context surrounding a pattern matching construct to each branch. Sozeau [2008] gives an equality-based solution similar to ours, and points out that the approach requires the K axiom in the general case.
Handling Delimited Continuations with Dependent Types

69:17

Equivalence

\[ \Gamma \vdash e_1 @ A (\lambda v : B. e_2) \equiv (\lambda v : B. e_2) (e_1 B \text{id}) \]  

[≡-CONT]

Typing Environments \vdash \Gamma

\[ \vdash \Gamma \quad \vdash \Gamma, \alpha : * \quad \vdash \Gamma, e : * \quad \vdash \Gamma, e_1 : A \quad \vdash e_2 : A \]

[EXTEND3] [EXTEND4]

Types \Gamma \vdash A : *

\[ \vdash \Gamma \quad \vdash \Gamma, \alpha : * \vdash e : \alpha \quad \vdash \Gamma, \alpha : * \vdash B : * \]

[VAR] [PrT2]

Terms \Gamma \vdash e : A

\[ \Gamma, \alpha : * \vdash e : B \quad \Gamma, \alpha : * \vdash \Pi \alpha : * B : * \quad \Gamma \vdash \Pi \alpha : * B : * \quad \Gamma \vdash e_1 : \Pi \alpha : (* B \to A) \quad \Gamma \vdash e_2 : A \]

[ABS2] [APP2] [T-CONT]

\[ \quad \text{Ind}(D : \Pi a : A, \alpha, \{c_i : \Pi y_i : B_i, D u_i\}) \in \Psi \quad \Gamma, a : A, x : D a \vdash P : * \quad (\Gamma, y_1 : B_i, u \equiv u_i, e \equiv c_i y_1 + e_1 : P [u/a, c_i y_1/x])_{i=1...k} \]

\[ \Gamma \vdash \text{match} e \text{ as } x \text{ in } D \text{ a return } P \text{ with } \{c_i y_1 \to e_1 : P [u/a, e/x] \}
\]

\[ e_1 \equiv e_2 \equiv e_1 u \equiv e_2 c_j \equiv c_i \vdash c_i \neq c_j \]

\[ \Gamma \vdash e : A \]

[Match] [INCON]

Fig. 7. \(\lambda^*_\Pi\) Equivalence and Typing (excerpt)

when we use [≡-CONT], \(e_1\) has the correct type. The well-typedness of \(e_1\) further implies that the equivalence cannot be inconsistent.

5.3 CPS Translation

Having designed a target language that is "rich enough" to reason about CPS terms, we define a CPS translation of \(\lambda^*_\Pi\). The key ideas are as follows:

(1) For every pure term \(e : A\), we translate it to \(\lambda \alpha : * \lambda k : A^+ \to \alpha. e\).

(2) For every subterm \(e'\) of \(e\) that appears in the type of \(e\) and would be evaluated during evaluation of \(e\), we represent application of \(e'^+\) to its continuation \(\lambda v : B. e_2\) as \(e'^+ \circ @ A (\lambda v : B. e_2)\).

The former gives us a type-safe interface \(e^+ A^+ \text{id}\) to the value hidden in \(e^+\), and makes the free theorem available when proving type preservation. The latter allows us to type check the body of the continuation with the knowledge that \(v\) represents the value of \(e^+\), solving the type mismatch we encountered in the application case.

We define the translation of typing environments, kinds, types, and terms by mutual induction on the derivation. That is, for each rule in Figures 3 - 5, we give a different CPS image. We cannot define the translation on the syntax for two reasons. First, to give an appropriate type annotation to every \(\lambda\)-bound variable introduced by the translation, we need to know the type of the source term, which is not present in the syntax. Second, to apply the polymorphic answer type translation
whenever applicable, we have to know the purity of terms we are translating, which requires us to look at their derivation. For the sake of readability, we will use the following abbreviations:

\[
\begin{align*}
\Psi^+ &\overset{\text{def}}{=} \Psi \\
\Gamma^+ &\overset{\text{def}}{=} \Gamma \\
\kappa^+ &\overset{\text{def}}{=} \kappa
\end{align*}
\]

We start with the translation of signatures (Figure 8). When we have an inductive datatype \( D \) in \( \lambda^{s/r}_\Pi \), the translation gives us a \( \lambda^k_\Pi \) counterpart \( D \), which has the same number of constructors as \( D \). The type of the index and constructor arguments are translated using the type translation. Readers familiar with CPS translations might find that, unlike the standard translation of the function type, the conclusion of a constructor type is not doubly negated into the form \((\text{\textbackslash D u}_i)^{+} \to \alpha \to \beta\). This is because application of a constructor to a value involves no further computation; the intuition becomes clear when comparing the translation of function and constructor application.

The translation of typing environments is basically a mapping of the \( ^+ \)-translation. The only non-trivial rule is (EXTEND2): we convert term \( e \) into \( e^+ A^+ \text{id} \), which denotes the value of \( e^+ \). We bind \( x \) to this value rather than the CPS computation \( e^+ \) because \( \lambda^{s/r}_\Pi \) is call-by-value, where variables must be replaced by values. The application of \( e^+ \) to the identity continuation is guaranteed to type check, because (EXTEND2) requires a pure \( e \), which implies that \( e^+ \) has a polymorphic type.

The translation of kinds is simple. Similarly to constructor types, we apply the \( ^+ \)-translation to the domain of functional kinds, and do not insert double negation into the co-domain. This time, the reason is that our translation only internalize term-level contexts.

The type translation does two things. First, it converts inductive types into their CPS counterparts, turning the index \( e \) into \( e^+ A^+ \text{id} \). This application is type safe since (DATA) requires that \( e_i \) is a pure term. The second task is to turn the function types with bars into those without. Notice that the translation of function types has two occurrences of the non-dependent arrow type. The first arrow represents “continuations are non-dependent functions,” while the second one corresponds to “final answer types do not depend on continuations.”

Now we look at the translation of terms, which we define in Figures 9 and 10. As mentioned above, we convert pure terms into a polymorphic function of the form \( \lambda \alpha : *, \lambda k : A^+ \to \alpha^+ . e \), and impure terms into the standard CPS form \( \lambda k : A^+ \to \alpha^+ . e \). Values are translated into a common form: given a continuation \( k \), we immediately apply \( k \) to their \( \lambda^k_\Pi \) counterpart. Application is basically translated in the same way as the standard call-by-value translation, except that we use a polymorphic answer type translation when applicable, and that we manually instantiate the answer type of pure terms. Observe the translation of dependent application derived by the odd-numbered rules. As we saw in Section 5.1, these are the challenging cases where type preservation requires

Proc. ACM Program. Lang., Vol. 2, No. ICFP, Article 69. Publication date: September 2018.
(UNIT) \[ \hat{\beta} \lambda \alpha : \ast. \lambda k : \text{Unit} \to \alpha . k () \]

(VAR) \[ \hat{\beta} \lambda \alpha : \ast. \lambda k : A^+ \to \alpha . k x \]

(ABS1) \[ \hat{\beta} \lambda \alpha : \ast. \lambda k : (\Pi x : A. B)^+ \to \alpha . k (\lambda x : A^+. e^+) \]

(ABS2) \[ \hat{\beta} \lambda \alpha : \ast. \lambda k : (\Pi x : A. \alpha \parallel B \parallel \beta)^+ \to \alpha . k (\lambda x : A^+. e^+) \]

(REC1) \[ \hat{\beta} \lambda \alpha : \ast. \lambda k : (\Pi x : A. B)^+ \to \alpha . k (\text{rec } f (x : A^+). e^+) \]

(REC2) \[ \hat{\beta} \lambda \alpha : \ast. \lambda k : (\Pi x : A. \alpha \parallel B \parallel \beta)^+ \to \alpha . k (\text{rec } f (x : A^+). e^+) \]

(APP1) \[ \hat{\beta} \lambda \alpha : \ast. \lambda k : (B [e_1/x])^+ \to \alpha . e_0^+ \alpha (\lambda v_0 : (\Pi x : A. B)^+. e_1^+ @ \alpha (\lambda v_1 : A^+. v_0 v_1 \alpha k)) \]

(APP2) \[ \hat{\beta} \lambda k : B^+ \to \alpha^+. e_0^+ \beta^+ (\lambda v_0 : (A \to B)^+. e_1^+ (\lambda v_1 : A^+. v_0 v_1 \alpha^+ k)) \]

(APP3) \[ \hat{\beta} \lambda k : (B [e_1/x])^+ \to \alpha^+. e_0^+ (\lambda v_0 : (\Pi x : A. B)^+. e_1^+ @ \alpha^+ (\lambda v_1 : A^+. v_0 v_1 \alpha^+ k)) \]

(APP4) \[ \hat{\beta} \lambda k : B^+ \to \alpha^+. e_0^+ (\lambda v_0 : (A \to B)^+. e_1^+ (\lambda v_1 : A^+. v_0 v_1 \alpha^+ k)) \]

(APP5) \[ \hat{\beta} \lambda k : (B [e_1/x])^+ \to (\alpha [e_1/x])^+. e_0^+ (\lambda v_0 : (\Pi x : A. \alpha \parallel B \parallel \beta)^+. e_1^+ @ (\lambda v_1 : A^+. v_0 v_1 \kappa)) \]

(APP6) \[ \hat{\beta} \lambda k : B^+ \to \alpha^+. e_0^+ \gamma^+ (\lambda v_0 : (A \to \alpha \parallel B \parallel \beta)^+. e_1^+ (\lambda v_1 : A^+. v_0 v_1 \kappa)) \]

(APP7) \[ \hat{\beta} \lambda k : (B [e_1/x])^+ \to (\alpha [e_1/x])^+. e_0^+ (\lambda v_0 : (\Pi x : A. \alpha \parallel B \parallel \beta)^+. e_1^+ @ (\lambda v_1 : A^+. v_0 v_1 \kappa)) \]

(APP8) \[ \hat{\beta} \lambda k : B^+ \to \alpha^+. e_0^+ (\lambda v_0 : (A \to \alpha \parallel B \parallel \beta)^+. e_1^+ (\lambda v_1 : A^+. v_0 v_1 \kappa)) \]

(LET1) \[ \hat{\beta} \lambda \alpha : \ast. \lambda k : (B [e_1/x])^+ \to \alpha . e_1^+ @ \alpha (\lambda x : A^+. e_2^+ \alpha k) \]

(LET2) \[ \hat{\beta} \lambda k : (B [e_1/x])^+ \to (\alpha [e_1/x])^+. e_1^+ @ (\beta [e_1/x])^+ (\lambda x : A^+. e_2^+ k) \]

(LET3) \[ \hat{\beta} \lambda k : B^+ \to \alpha^+. e_1^+ (\lambda x : A^+. e_2^+ \alpha^+ k) \]

(LET4) \[ \hat{\beta} \lambda k : B^+ \to \alpha^+. e_1^+ (\lambda x : A^+. e_2^+ k) \]

Fig. 9. CPS Translation of Terms

non-trivial reasoning. By the typing rules, we know that the argument \( e_1 \) is a pure term. This implies that \( e_1^+ \) must have a polymorphic type. Since \( e_1 \) appears in the result type, and evaluation of \( e_0 e_1 \) involves evaluation of \( e_1 \), we invoke the free theorem by representing the application of \( e_1^+ \) to its continuation using \( @ \). This forces us to type the application using \( [T-\text{CONT}] \), which makes the
equality \( v_1 = e_1 \vdash A^+ \) id available for type checking. Note that we do not use @ for application of \( e_0 \). The reason is that the type of an application cannot depend on the function, which means we do not need to know what \( v_0 \) represents when typing the CPS image. This idea is also used in the rules for the let expression: we use @ for \( e_1 \) when \( e_1 \) is pure, but not for the body \( e_2 \) even if \( e_2 \) is pure.

The translation of constructor application uses @ in the same way as function application, but it uses the continuation \( k \) differently. When translating function application, we pass \( k \) to application \( v_0 \ v_1 \), whereas when translating constructor application, we pass the inductive datum \( c_1 \ v \) to \( k \). This reflects the fact that application of a function to a value results in a computation, while application of a constructor to a value results in a value. At the level of types, the contrasting nature of \( \alpha \) and \( \beta \) differs. When translating function application, we pass the inductive datum \( \alpha \) to \( v \), whereas when translating constructor application, we pass \( \beta \) to \( \alpha \). This reflects the fact that application of a function to a value results in a computation, while application of a constructor to a value results in a value. At the level of types, the contrasting nature of \( \alpha \) and \( \beta \) differs.

Pattern matching is another case that uses the @ application form. Specifically, we use @ in rules (MATCH1) and (MATCH2), where the scrutinee \( e \) is a pure term and thus the result type depends on it. By taking a closer look at the three rules, we find that the return type of the matching construct has changed from \( P \) to the final answer type. This is because each branch has been passed the continuation \( k \), which produces the final answer.

The translation of shift and reset tells us how they “shift” and “reset” control. The latter is simply represented as running a computation with the identity continuation. The former can be seen from the way we use the continuation \( k \): while all other cases use \( k \) to produce an answer, the shift case uses it in an arbitrary manner, meaning that we are not returning a value as we usually do. In the CPS image, we can see the two reset’s in the reducible form \( (e \ [λ x : A, \langle f[x]\rangle /c) \) of a shift redex. The outer one is the identity continuation applied to \( e \), and the inner one is represented by the constructor \( λ \alpha', \λ k', k' \ e \), which is structurally similar to the translation of reset.

Lastly, we look at the type conversion and purity casting cases. Application of rules (CONV1) and (CONV2) does not change the shape of CPS images, since it has no computational effect. Rule (EXP) simply serves as answer type instantiation.
5.4 Proving Type Preservation

The carefully designed target language and CPS translation allow us to prove the type preservation property: well-typed $\lambda^s/r$ terms are translated to well-typed $\lambda^k$ terms. But before that, we have to prove coherence, which states that the translation preserves equivalence. We need coherence because the translation is defined on the derivation, which requires us to account for the conversion cases. Consider the translation of (Conv1). We must show $e^+ : \Pi \alpha : \ast. (B^+ \rightarrow \alpha) \rightarrow \alpha$, whereas the induction hypothesis gives us $e^+ : \Pi \alpha : \ast. (A^+ \rightarrow \alpha) \rightarrow \alpha$. The two types are convertible if $A \equiv B$ implies $A^+ \equiv B^+$, i.e., if coherence holds.

As Bowman et al. [2017] point out, it is not possible to prove coherence separately from type preservation when we use a typed equivalence. The reason is that a typed equivalence requires $A^+$ and $B^+$ to have the same type, which relies on type preservation. This is why we use an untyped equivalence in the source and target languages.

In the rest of this subsection, we show three lemmas: compositionality, correctness, and coherence. Then, we show the main theorem: the translation preserves typing.

**Lemma 5.1 (Compositionality).** Suppose $e'$ is a pure term.

1. $(\kappa [e'/x])^+ \equiv \kappa^+ [e'^+ \_id/x]$
2. $(A[e'/x])^+ \equiv A^+ [e'^+ \_id/x]$
3. $(e[e'/x])^+ \equiv e^+ [e'^+ \_id/x]

The proof is by induction on the derivation of $\kappa, A$, and $e$. The only interesting case is when $e = x$, which uses the new equivalence rule $[\equiv\text{-Cont}]$ (see Section 6.1 of Bowman et al. [2017]). Note that we are substituting a non-value $e'^+ \_id$. This is why the target language must be call-by-name.

**Lemma 5.2 (Correctness).**

1. If $\Gamma \vdash \kappa$ and $\kappa \Rightarrow^* \kappa'$, then $\kappa^+ \equiv \kappa'^+$.
2. If $\Gamma \vdash A : \ast$ and $A \Rightarrow^* A'$, then $A^+ \equiv A'^+$.
3. If $\Gamma \vdash e : A$ or $\Gamma; \alpha \vdash e : A$; $\beta$ and $e \Rightarrow^* e'$, then $e^+ \equiv e'^+$.

We first prove correctness with regard to single-step reduction, by cases on the $\Rightarrow$-relation, and then extend the result to multi-step reduction, by induction on the length of the reduction sequence. Here we show one of the $\beta$-reduction cases. Suppose we have $(\lambda x : A. e_0) v_1$ derived by (App5):

$$(\lambda x. e_0) v_1^+ = \lambda k. (\lambda x. e_0)^+ (\beta [v_1/x])^+ (\lambda v_0. v_1^+ @ (\beta [v_1/x])^+ (\lambda v_1. v_0 v_1 k))$$

$= \lambda k. (\lambda x. \lambda k. k (\lambda x. e_0)) (\beta [v_1/x])^+ (\lambda v_0. v_1^+ @ (\beta [v_1/x])^+ (\lambda v_1. v_0 v_1 k))$ $\Rightarrow^* \lambda k. v_1^+ @ (\beta [v_1/x])^+ (\lambda v_1. (\lambda x. e_0)^+ v_1 k)$ $\Rightarrow^* \lambda k. v_1^+ @ (\beta [v_1/x])^+ (\lambda v_1. (e_0^+ [v_1/x]) k)$ $\equiv \lambda k. (\lambda v_1. (e_0^+ [v_1/x]) k) (v_1^+ A^+ id)$ $\equiv (e_0^+ [v_1/x])^+$ by $[\equiv\text{-Cont}]$

**Lemma 5.3 (Coherence).**

1. $\kappa \equiv \kappa' \Rightarrow \kappa^+ \equiv \kappa'^+$
2. $A \equiv A' \Rightarrow A^+ \equiv A'^+$
3. $e \equiv e' \Rightarrow e^+ \equiv e'^+$

In both $\lambda^s/r$ and $\lambda^k$, equivalence is defined in terms of reduction. Since the translation is correct (Lemma 5.2), equivalent expressions in $\lambda^s/r$ are mapped to equivalent expressions in $\lambda^k$.
The proof is by mutual induction on the derivation. Here we give the case (MATCH1), which shows how the restriction on type dependency and the typing rule of $\lambda_k^\Pi$ pattern matching help us in the proof. Our goal is to show the term

$$
\lambda \alpha : * . \lambda k : (P [u/a, e/x])^+ \to \alpha.
$$

$$
e^\pi \land (\lambda v : (D u)^+. \text{match } v \text{ as } x \text{ in } D \text{ a return } \alpha \text{ with } \{c_i \overrightarrow{y_i} \to e_i^\pi \alpha k})
$$

has type

$$
P \alpha : *(P [u/a, c_i y_i/x])^+ \to \alpha
$$

under context $\Gamma^+$. By [T-Cont] and [Match], it suffices to show that each branch $e_i^\pi \alpha k$ has type $\alpha$ under $\Gamma^+$ extended with the following information:

$$
\alpha : *, k : (P [u/a, e/x])^+ \to \alpha, v = e^\pi (D u)^+. \text{id} : (D u)^+, y_i : B_i^+, u^+ A^+ \text{id} \equiv u_i^+ A^+ \text{id}, v \equiv c_i y_i
$$

By the induction hypothesis, $e_i^\pi$ has the following type:

$$
P \alpha : *(P [u_i/a, c_i y_i/x])^+ \to \alpha
$$

Now, let us prove that $(P [u_i/a, c_i y_i/x])^+$ is equivalent to the domain of the continuation $k$:

$$(P [u_i/a, c_i y_i/x])^+ \equiv P^+ [u_i^+ \text{id} / a, (c_i y_i)^+ \text{id} / x]$$

by Lemma 5.1

$$
\sim^k P^+ [u_i^+ \text{id} / a, c_i y_i / x]
$$

$$
\equiv P^+ [u_i^+ \text{id} / a, v / x]
$$

by $v \equiv c_i y_i$

$$
\equiv P^+ [u^+ \text{id} / a, v / x]
$$

by $u^+ \text{id} \equiv u_i^+ \text{id}$

$$
\sim^k P^+ [u^+ \text{id} / a, e^\pi (D u)^+ \text{id} / x]
$$

$$
\equiv (P [u/a, e/x])^+
$$

by Lemma 5.1

The rule [Match] gives us the information $u^+ \text{id} \equiv u_i^+ \text{id}$ and $v \equiv c_i y_i$. Since (MATCH1) requires a pure scrutinee $e$, we further know that $v$ is the result of running $e^\pi$ with the identity continuation. This knowledge is made explicit by [T-Cont], allowing us to $\delta$-reduce $v$ to $e^\pi (D u)^+ \text{id}$. Thus we conclude that the translation of the matching construct has the correct type.

Typability of CPS terms also relies on preservation of well-formedness of inductive definitions and guardedness of recursive functions. The former follows by the safety condition of $\lambda_k^{\Pi/r}$. The latter only holds of terms after administrative reduction (i.e., reduction of non-source redexes). The reason is that the translation turns fully applied function variables into those receiving no arguments. Specifically, it turns a guarded recursive call $f y_i$ into:

$$
\lambda k. (\lambda \alpha. \lambda k.k f) _\_ (\lambda v_0. (\lambda \alpha. \lambda k.k y_i) _\_ (\lambda v_1.v_0 v_1 k))
$$

With the non-applied $f$, we cannot know from the surface structure whether $f$ receives a smaller argument or not. However, after reducing administrative redexes, the application will look like $\lambda k.f y_i k$, which satisfies the guard condition. Therefore, we equip $\lambda_k^{\Pi}$ with a slightly modified typing rule for recursive functions $\text{rec } f (x : A). e$, which requires that $f$ is guarded in $e'$, where $e'$ is the post-administrative-reduction counterpart of $e$. 

Proc. ACM Program. Lang., Vol. 2, No. ICFP, Article 69. Publication date: September 2018.
5.5 Scaling to General Inductive Types

Our type-preserving CPS translation scales to multi-arity inductive types as well. The idea is the same: we translate every pure constructor argument into a polymorphic form, and for those arguments \( e_i \) that appear in types, we use \( @ \) to turn on the CPS reasoning. As an example, the term

\[
\cdot z \downarrow (suc z) nol : L (suc z)
\]

representing a singleton list containing 1, is translated as follows:

\[
\lambda \alpha : \ast. \lambda k : (L (suc z))^{+} \rightarrow \alpha. \\
\cdot z^{+} @ \alpha (\lambda v_0 : \mathbb{N}^{+}. (suc z)^{+} \alpha (\lambda v_1 : \mathbb{N}^{+}. nil^{+} \alpha (\lambda v_2 : (L z)^{+}. k (\cdot : v_0 v_1 v_2))))
\]

After a sequence of administrative reductions, we obtain:

\[
\lambda \alpha : \ast. \lambda k : L (suc z) \rightarrow \alpha. k (\cdot : z (suc z) nil)
\]

Recall from Section 4.5 that the multi-arity extension requires a chaining rule for answer types of impure constructor arguments. Dually, the CPS translation of the extended language needs a certain rule for instantiating the answer type of CPS translated pure arguments. Finding an appropriate answer type is not so difficult; see our technical appendix [Cong and Asai 2018] for details.

6 Programming with Shift, Reset, and Dependent Types

This section presents two programs that use control operators and dependent types in an interesting way. The first one is the \( \text{mk-lst} \) function, which receives a natural number \( n \) and builds a list \([1; 2; \ldots; n]\) of type \( L \ n \). Here, we generate each element of the list using a mutable state, which we simulate with \( \text{shift} \) and \( \text{reset} \) [Asai and Kiselyov 2011]. Since we need to know the top-level argument (instead of the argument at each recursive call) to type the \( \text{shift} \) operator, we define an auxiliary function \( \text{mk-lst-aux} \) as follows:

\[
\text{rec } f (n : \mathbb{N}) (n_0 : \mathbb{N}). \text{match } n \text{ as } x \text{ in } \mathbb{N} \text{ return } L \ n \text{ with } \\
z \rightarrow \text{nil } | \text{suc } m \rightarrow \cdot : m \text{ (tick } () \text{; get } ()) \text{ (if } m \text{n_0) }
\]

and define \( \text{mk-lst} \ n \) as \( \text{mk-lst-aux} \ n \ n \). The definition of \( \text{tick} \) and \( \text{get} \) looks like:

\[
\text{tick def } \lambda \_ : \text{Unit. sk : } \mathbb{N} \rightarrow L \ n_0, \lambda s : \mathbb{N} \rightarrow L \ k () \text{ (suc } s) \text{, get def } \lambda \_ : \text{Unit. sk : } \mathbb{N} \rightarrow L \ n_0, \lambda s : \mathbb{N} \rightarrow L \ k \ s \ s
\]

Notice that we use \( n_0 \) in the annotation for the captured continuation. Now we run the program \( \text{mk-lst } 3 \) by wrapping the function call in an appropriate context:

\[
\langle \text{let } l = \text{mk-lst } 3 : \mathbb{L} 3 \text{ in } \lambda s : \mathbb{N}. l \rangle 0 = [1; 2; 3]
\]

The main program is an application of a \( \text{reset} \) clause to 0, which is the initial value of the state. We update and access the state using \( \text{tick} \) and \( \text{get} \): intuitively, the former returns nothing but increments the value of the state, while the latter returns the current value of the state without changing it. These are done by setting the surrounding context to the form “given a state, return a value.” Observe that \( \text{tick} \) and \( \text{get} \) call the captured continuation \( k \) with two arguments. The first one serves as the return value of these functions, whereas the second one is the value of the state for future computation.

As \( \text{mk-lst} \) increments the state before accessing it, \( \text{get} \) returns a different value on each recursive call of \( \text{mk-lst} \). This dynamic behavior does not break typability, because we use the returned values as elements of the list, which do not appear in types.

From the return type of the pattern matching construct, we can see that \( \text{mk-lst} \) has type \( \Pi n : \mathbb{N}. L \ n \| \mathbb{L} \ n \| L \ n \), which tells us that the function gives us back a list whose length is exactly the number we pass to it. This invariant cannot be encoded in a simply typed language.

We next show a more practical example. Note that the example is not implementable in the current language because it requires \textit{large elimination}, \textit{i.e.,} pattern matching constructs that inspect
a term and return a type. We do not include large elimination because it is not always compatible with impredicativity, which our CPS translation relies on, but for now, let us assume this feature and see how it would be useful in combination with control operators.

In this example, we consider a tiny language $L_{BN}$ consisting of booleans and natural numbers. Following Altenkirch and Reus [1999], we define $L_{BN}$ as an intrinsically typed language—which does not allow constructing ill-typed terms—using dependent inductive types:

$$\text{Ind}(\text{Ty} : \ast, \{\text{bool} : \text{Ty}; \text{nat} : \text{Ty}\})$$

Notice that the type constant $\text{Tm}$ for $L_{BN}$ terms is indexed by a $L_{BN}$ type, which is a meta-level term. The index prevents meaningless $L_{BN}$ terms from being accepted by the meta-language type checker. For example, we cannot write $\text{div} (\text{num} \, z) \, \text{true}$ because $\text{true}$ is not of type $\text{Tm} \, \text{nat}$.

Our goal is to implement an evaluator of $L_{BN}$, which works in the following way. Given a $L_{BN}$ term, the evaluator returns a meta-level number (of type $\mathbb{N}$) or a boolean (of type $\mathbb{B}$). For example, if the input is $\text{num} \, z : \text{Tm} \, \text{nat}$, we obtain a meta-language value $z : \mathbb{N}$. When the term is a division by zero, the evaluator raises an error. To implement such an evaluator, we need a type interpretation $\text{interpTy}$, which converts $L_{BN}$ types into meta-level types. The definition of $\text{interpTy}$ looks as follows; observe that it is a type-level function that uses large elimination:

$$\lambda \, t : \text{Ty}. \, \text{match} \, t \, \text{as} \, x \, \text{in} \, \text{Ty} \, \text{return} \, \ast \, \text{with} \, \text{bool} \to \mathbb{B} \, | \, \text{nat} \to \mathbb{N}$$

We also need a datatype $\text{Ans}$ representing answers, which are either a value or an error:

$$\text{Ind}(\text{Ans} : \text{Ty} \to \ast, \{\text{val} : \Pi \, t : \text{Ty}. \, \text{interpTy} \, t \to \text{Ans} \, t; \text{err} : \Pi \, t : \text{Ty}. \, \text{Ans} \, t\})$$

We use this datatype when we call the evaluator. Specifically, the main program takes the form $\langle \text{val} \, (\text{eval} \, t \, e) \rangle$, meaning that the evaluator will only ever return a value.

Now we define $\text{eval} \, t \, e$ as $\text{eval-aux} \, t \, t'$, where $\text{eval-aux}$ is the following recursive function:

$$\text{rec} \, f \, (t : \text{Ty}, \, t' : \text{Ty}, \, e : \text{Tm} \, t).$$

$$\text{match} \, e \, \text{as} \, _{\,} \, \text{in} \, \text{Tm} \, x \, \text{return} \, \text{interpTy} \, x \, \text{with}$$

$$\ldots \, | \, \text{div} \, e_{1} \, e_{2} \to \text{match} \, \text{eval-aux} \, \text{nat} \, t' \, e_{2} \, \text{as} \, _{\,} \, \text{in} \, \mathbb{N} \, \text{return} \, \mathbb{N} \, \text{with}$$

$$z \to \langle \text{Sk} : \mathbb{N} \to \text{Ans} \, t', \, \text{err} \, t' \rangle \ldots$$

The auxiliary function receives two $L_{BN}$ types $t$ and $t'$, as well as a $L_{BN}$ term $e$. Among the two types, the former is the type of $e$, which is what we are going to evaluate at the current call, while the latter represents the type of the top-level term, which determines the answer types of the function’s body. When the input of the evaluator is a division by zero, we capture the context using $\text{shift}$, discard it, and directly return $\text{err} \, t'$ to the surrounding reset. Observe that we implement efficient exception raising in direct style. If we do not have control operators but wish to avoid propagation of exceptions (as in the exception monad), we have to write the entire evaluator in CPS, which is less convenient.

The type of the evaluator tells us that, in the normal case, a term of type $\text{Tm} \, t$ is evaluated to a value of type $\text{interpTy} \, t$, i.e., the evaluator preserves types. In the exception case, we drop the surrounding context, whose return type is $\text{Ans} \, t'$, and raise an exception of type $\text{Ans} \, t'$. Therefore, the initial and final answer types of the function’s body are both $\text{Ans} \, t'$.

7 Future Work

Scaling to CIC. In this work, we designed $\lambda_{\Pi}^{\Sigma/r}$ as a minimal language that has both dependent types and the $\text{shift}$/$\text{reset}$ operators. To integrate these operators into the full CIC, we have to support various features that are currently missing.
The first step is to add polymorphism. This extension should not be hard, since the (non-dependent) type system of shift and reset scales naturally to Girard’s System F [Girard 1972], as shown by Asai and Kameyama [2007]. The combination of polymorphism and dependent types does not complicate the CPS translation either [Bowman et al. 2017].

Next, we intend to introduce the Set/Prop distinction. The distinction allows us to soundly mix imprecadativity and large elimination, which, when unrestrictedly integrated into a classical calculus, would lead to inconsistency Coquand [1989]. However, it is not clear how we should CPS translate pure terms in Set, because our polymorphic answer type translation relies on imprecadativity. For instance, a source function \( f : \mathbb{N} \to B \) would be turned into \( f^\pi : \Pi \alpha : \text{Set.} \quad ((\mathbb{N} \to B)^\pi \to \alpha) \to \alpha \), for which the value-extraction operation \( f^\pi \) \((\mathbb{N} \to B)^\pi \id \) is not type-safe since the instantiated answer type has a quantification over Set and thus is not an inhabitant of Set.

Another interesting extension would be adding an infinite hierarchy of universes. This motivates us to consider control effects at higher universe levels. Let us look at the three examples below, where \( \text{Vec} \mathbb{N} n \) denotes the type of vectors containing \( n \) natural numbers:

\[
(9) \langle \text{Vec} (\text{Sk} : \text{Set} \to \mathbb{B}) 1 \rangle \quad (10) \langle \text{Vec} \mathbb{N} (\text{Sk} : \mathbb{N} \to \text{Set.} \mathbb{B}) \rangle \quad (11) \langle 1 + \text{Sk} : \mathbb{N} \to \mathbb{N.} \mathbb{B} \rangle
\]

Example (9) has a type-level shift operator, which captures a type-level context and returns a type. If we allow a more flexible form of contexts, we would be able to handle contexts whose hole and return types are inhabitants of different universes. However, it does not seem to be a good idea to allow term-to-type contexts as in example (10), since it violates our no-impure-term restriction. We might also want to allow answer-universe modification as in example (11): observe that the shift operator changes the answer type from \( \mathbb{N} \) to *.

While we have not yet found interesting applications of higher-level control effects, we noticed that when not fixing the level of answer types, a source term \( e \) and its CPS counterpart may reside in different universes. For instance, the shift construct in example (11) is a Set-term, but the CPS translation turns it into a Type1-term. As future work, we plan to investigate whether the CPS translation can be type-preserving in the presence of universe shifting.

**Scaling to the CPS Hierarchy.** Analogously to the universe hierarchy, we have the notion of hierarchical contexts. Danvy and Filinski [1990] extend shift and reset to a family of control operators shift\( n \) and reset\( n \), allowing us to manipulate different levels of contexts: e.g., \( \langle (1 + (2 + \text{Sk}(3))_1) \rangle_2 \) evaluates to 4 when \( i = 1 \) and 3 when \( i = 2 \). In general, one can simulate shift\( n \) and reset\( n \) by \( n \) successive CPS translations, which yield terms requiring \( n \) continuations.

In this paper, we only discussed level-1 shift and reset. To support higher-level operators, we need equivalence and typing rules for terms requiring multiple continuations in the target of the CPS translation. For example, when we CPS translate a pure term twice, we will obtain a term of type \( \Pi \alpha \alpha' : \ast \to B \to (\alpha \to \alpha') \to \alpha' \to (\alpha \to \alpha') \to \alpha' \). The \([\ast\text{-CONT}]\) rule for such a term will look like \( e \) \( A A' \) \( k \gamma \equiv k (e B B \id' \id) \gamma \), where \( \id = \lambda x : B. x \) and \( \id' = \lambda x : B. \lambda y : B. \gamma x \). It would be interesting to see if this approach extends naturally to the full CPS hierarchy.

**Optimizing CPS Translations.** It is well-known that naïve CPS translations generate a number of administrative redexes, making resulting programs verbose and inefficient. They also disrupt guardedness of recursive functions, as we saw in Section 5. To avoid generation of administrative redexes, researchers have developed a variety of optimizing translations. Danvy and Filinski [1992] give an administrative-redex-free translation of the simply typed \( \lambda \)-calculus with shift and reset, which uses meta-level (instead of object-level) abstraction to represent continuations. This allows us to reduce the extra redexes at compile time. Recent work by Asai and Uehara [2018] extends the translation to a selective one, where only impure terms are translated into CPS and pure terms

Proc. ACM Program. Lang., Vol. 2, No. ICFP, Article 69. Publication date: September 2018.
are kept in direct style. This makes output programs more compact, and significantly improves run-time performance.

As \( \lambda^{s/r}_\Pi \) has facilities for distinguishing between pure and impure terms, it would be straightforward to define a selective CPS translation, and according to our latest study, selectiveness seems to help us solve a number of issues with dependent and inductive types. First, a selective translation allows us to admit more inductive definitions in the source language, because it does not insert negation into the co-domain of pure function types. Second, selectiveness lets us adopt the standard guard condition in the target, as it turns a guarded recursive call \( f \ y \) into either \( f \ y_1 \) (if \( f \)'s body is pure) or \( \lambda k. f \ y_1 \ k \) (otherwise). Thirdly and most importantly, typing a selectively translated term does not require parametricity reasoning, since pure terms are converted into terms we can directly evaluate. This means we no longer need \([\equiv \text{-CONT}]\) or \([\text{T-CONT}]\) in the target language. It also implies that the translation can target a predicative language like Agda. As future work, we intend to give a selective (and higher-order) translation of \( \lambda^{s/r}_\Pi \), and investigate what other advantages it has.

**Call-By-Name Shift and Reset.** Although control operators are mainly studied in a call-by-value setting, there are call-by-name variants of \texttt{shift} and \texttt{reset} as well [Herbelin and Ghilezan 2008; Tanaka and Kameyama 2012]. We believe that it is possible to build a call-by-name version of \( \lambda^{s/r}_\Pi \), using the same restriction on type dependency. A type-preserving CPS translation should also exist, but if the translation is unselective, it applies only to inductive types without any recursive arguments, because a call-by-name translation inserts double negation to every domain of a constructor type. On the other hand, the translation, as well as the proof of type preservation, will be simpler than the call-by-value counterparts. The reason is that evaluation of subterms, which makes type preservation hard to prove, happens less frequently in a call-by-name language. Indeed, Barthe et al. [1999] give a type-preserving translation from CC to CC itself.

### 8 RELATED WORK

**Control Operators and Dependent Types.** Herbelin [2005] integrates \texttt{call/cc} into a language with strong \( \Sigma \) types and equalities, and shows that the resulting language allows one to prove absurd propositions like \( 0 = 1 \). Intuitively, the inconsistency is due to the ability of proofs to backtrack, that is, one can build a proof that appears to be witnessed by a certain term but is actually witnessed by a different term. To safely mix the three ingredients, Herbelin [2012] defines a syntactic category called the \textit{negative-elimination-free (NEF)} fragment, consisting of proofs whose evaluation causes no effect. By restricting dependency to NEF proofs, Herbelin gives a consistent classical calculus where one can constructively prove two weak instances of the axiom of choice.

NEF proofs essentially correspond to what we call pure terms. Instead of classifying terms using syntactic information, we determine purity using typing rules, but the two approaches should work in the same way. The difference is that while Herbelin deals with undelimited control operators, we choose delimited control operators, which are more expressive and practically useful. As we saw in Section 3, the existence of answer types requires two additional restrictions on type dependency, namely \"continuations are non-dependent functions\" and \"final answer types do not depend on continuations.\" These restrictions are not necessary when incorporating undelimited control constructs, because their type does not allow the problematic dependency. Specifically, they usually have a type of the form \((A \to \bot) \to A\) or \((A \to \bot) \to \bot\) \to A\), where the second and third types correspond to initial and final answer types, respectively.

The NEF restriction is also adopted by Miquey [2017], who extends the \( \lambda\mu\tilde{\mu} \)-calculus with dependent types. The main challenge of this extension is that the issue with CPS and dependent types is already present in the pre-CPS world, since the calculus lets us write evaluation contexts.
(i.e., continuations). Miquey calls this “desynchronization of the type system with respect to the reduction,” and solves the problem using two tricks. First, he annotates typing judgments with a list of variable-proof pairs, which tells us what value is to be plugged into the hole of a context. Second, he introduces a control delimiter [Ariola et al. 2009], which allows us to define the set of NEF proofs and restrict dependency to those proofs. Thus, Miquey obtains desirable properties such as subject reduction and type preservation of the CPS translation.

An interesting fact is that Miquey gives a CPS translated NEF proof \( p^+ \) a polymorphic type of the form \( \forall R. (\Pi a : A^+. R(a)) \rightarrow R(p^+) \). Observe that \( p^+ \) receives a dependent continuation, and returns a value whose type depends on \( p^+ \), which is essentially the value we obtain by computing \( p^+ id \). These dependencies make the type look like an elimination rule, but the type should be equivalent to \( \forall R. (A^+ \rightarrow R) \rightarrow R \), since \( p^+ \) uses its continuation exactly once, and the unique value passed to the continuation must be \( p^+ \).

Similarly to Miquey [2017], Lepigre [2016] extends the \( \lambda \mu \)-calculus [Parigot 1992] with ML-like constructors and dependent types, but taking a different approach. Instead of syntactically constraining dependency, Lepigre imposes a semantic value restriction, which allows types to depend on terms that are observationally equal to values. As Miquey [2017] points out, the NEF fragment is compatible with Lepigre’s value restriction, and the same should hold for our pure-terms-only principle. On the other hand, the semantic approach makes it harder to decide whether a term is allowed to appear in a type.

**Effects and Dependent Types.** In the last few years, considerable progress has been made in combining dependent types and effects. The \( F^* \) language [Swamy et al. 2016] is a dependently typed, effectful language that aims to serve both as a proof assistant and as a general-purpose programming language. In \( F^* \), the semantics of effects is defined in terms of a lattice of Dijkstra monads [Dijkstra 1975]; the idea is to characterize pure and effectful computations using their pre- and post-conditions. As in our language, types in \( F^* \) are not allowed to depend on effectful terms, but it turns out that we can still use the language for practical purposes under this restriction. For instance, \( F^* \) allows encoding of hyper-heaps [Swamy et al. 2016], a generalized form of heaps that help us speed up verification of programs. An earlier version of \( F^* \), which imposes a stronger restriction that types can only depend on values, has also proven useful for secure distributed programming [Swamy et al. 2011].

Ahman [2017a] and Vákár [2017] independently formalize a dependent calculus with computational effects by extending Levy’s Call-By-Push-Value (CBPV) calculus [Levy 2012] with dependent types. CBPV distinguishes between values and computations, which roughly correspond to pure and impure terms in our language. Similarly to our work, Ahman and Vákár do not allow types to depend on computations, but in addition to this, they had to tackle a problem specific to CBPV: sequential composition. This construct can be understood as a let expression binding an impure term, that is, let \( x = e_1 : A \) in \( e_2 \) where \( e_1 \) is impure. Computationally, the construct evaluates \( e_1 \) to some value \( v_1 \), then evaluates \( e_2 \) with the binding \( x = v_1 \). In a dependently typed setting, the type of \( e_2 \) may contain free occurrences of \( x \), which is to be replaced by the value of \( e_1 \). The problem is that, when typing the whole construct, the variable \( x \) remains free in the conclusion, because we do not know what \( e_1 \) evaluates to. Note that it is invalid to substitute \( e_1 \) into the result type, since \( e_1 \) is impure. Ahman solves this issue by introducing computational \( \Sigma \) types, whereas Vákár avoids the problem by turning \( e_1 \) into a pure thunk. When dealing with delimited control, however, allowing such dependency does not seem to be possible, since impure terms do not reduce to a value.

The difference between directly integrating effects and simulating them with \texttt{shift} and \texttt{reset} is that the former approach can handle different effects at one time, while the latter cannot. This is because \texttt{shift} lacks the ability of choosing a particular \texttt{reset}: it has no choice but to associate with...
the nearest enclosing one. To handle multiple effects, we need the variants of shift and reset that are more powerful than the ordinary ones [Forster et al. 2017; Kiselyov and Sivaramakrishnan 2016]. One such variant is multi-prompt shift and reset, which carry a tag representing the intended association. Switching to this variant comes at the cost of losing the CPS semantics; indeed, prompt tags can express dynamic binding [Kiselyov et al. 2006], which, to the authors’ knowledge, has not yet been studied in a dependently typed setting. An alternative choice is shift0 and reset0, which are known as a dynamic variant of shift and reset in that one cannot see what continuation a shift0 captures merely from syntactic information. These operators have a CPS semantics in both untyped [Shan 2007] and typed [Materzok and Biernacki 2011] settings, but the semantics is not as simple as that of the standard shift and reset; it either uses recursive continuations or relies on subtyping relations. Our language does not support mixed uses of effects, but it has a reasonably concise CPS semantics (although there is some complication caused by dependent types), which is a desirable property from both theoretical and implementation points of view.

Logical Interpretation of Control Operators. In this paper, we have studied λs/rΠ mainly as a programming language, focusing on the computational aspect of shift and reset. To use λs/rΠ as a proof assistant, we must figure out how these operators affect the underlying logic of the language.

It is well-known that incorporating undelimited control makes a calculus classical: e.g., the call/cc operator corresponds to the Peirces’s law ((A → B) → A) → A [Griffin 1990], which implies classical theorems such as double-negation elimination and the law of excluded middle. In the case of delimited control, logical interpretation seems to be a more subtle problem. Ilik [2012] extends a dependent calculus with a variant of shift and reset that do not admit ATM, and shows that the calculus proves Double-Negation Shift (∀x. ¬¬A(x)) → ¬¬∀x. A(x), which is a non-intuitionistic theorem. Shan [2003], on the other hand, designs a simply typed language with ATM-allowing shift and reset, and proves that the calculus is intuitionistic, using a type-preserving CPS translation and its inverse. The combination of simple types and the full shift/reset has also been studied by Zeilberger [2010], in terms of intuitionistic polarity. We consider that the ability of ATM is the key factor that affects the logical interpretation of a language. In particular, we claim that impure terms should not be considered as proofs, since they do something for us only when they are surrounded by a delimited context of an appropriate type, and it is not clear how to encode this requirement in a standard logic. The presence of non-proofs seems to greatly restrict the use of shift and reset in proofs, leading to the conjecture that λs/rΠ is intuitionistic.

9 CONCLUSION
Dependent types and control operators are essential elements in a programmer’s toolbox. The last few years have seen several successful attempts at mixing the two constructs, but none of them yielded a language that is “ready for use.” We have combined user-defined inductive datatypes, which are indispensable when programming with dependent types, and the shift and reset operators, which allow flexible manipulation of control flow but still have a succinct semantics. The three restrictions on type dependency, together with the non-uniformly polymorphic CPS translation, take us one step towards a practical programming language where both dependent types and control operators are available. We believe that such a language will enable efficient implementation of secure programs, and lead us to discover novel proof techniques.

ACKNOWLEDGMENTS
We gratefully acknowledge the anonymous reviewers for valuable feedback. We would also like to thank Andreas Abel and William Bowman for technical discussions. This work was supported in part by JSPS KAKENHI under Grant No. JP18H03218.
REFERENCES

Danel Ahman. 2017a. Fibred Computational Effects. Ph.D. Dissertation. University of Edinburgh.
Danel Ahman. 2017b. Handling Fibred Algebraic Effects. Proc. ACM Program. Lang. 2, POPL, Article 7 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158095
Amal Ahmed and Matthias Blume. 2011. An Equivalence-Preserving CPS Translation via Multi-Language Semantics. In Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP ’11), 431–444.
Thorsten Altenkirch and Bernhard Reus. 1999. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In Proceedings of the 13th International Workshop and 8th Annual Conference of the EACSL on Computer Science Logic (CSL ’99). Springer-Verlag, London, UK, 453–468. http://dl.acm.org/citation.cfm?id=647849.737066
Abhishek Anand, A Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In The Third International Workshop on Coq for Programming Languages (CoqPL).
Zena M. Ariola, Hugo Herbelin, and Amr Sabry. 2009. A Type-Theoretic Foundation of Delimited Continuations. Higher Order and Symbolic Computation 22, 3 (Sept. 2009), 233–273. https://doi.org/10.1007/s10990-007-9006-0 online from 2007.
Kenichi Asai and Yukiyoshi Kameyama. 2007. Polymorphic Delimited Continuations. In Proceedings of the 5th Asian Conference on Programming Languages and Systems (APLAS ’07). Springer-Verlag, Berlin, Heidelberg, 239–254. http://dl.acm.org/citation.cfm?id=1784774.1784797
Kenichi Asai and Oleg Kiselyov. 2011. Introduction to programming with shift and reset. In ACM SIGPLAN Continuation Workshop 2011.
Kenichi Asai and Chihiro Uehara. 2018. Selective CPS Transformation for Shift and Reset. In Proceedings of the 2018 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM ’18), 40–52.
Gilles Barthe, John Hatcliff, and Morten Heine B. Sørensen. 1999. CPS Translations and Applications: The Cube and Beyond. Higher-Order and Symbolic Computation 12, 2 (Sept. 1999), 125–170. https://doi.org/10.1023/a:101000206149
Gilles Barthe and Tarmo Uustalu. 2002. CPS Translating Inductive and Coinductive Types. In Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM ’02). ACM, New York, NY, USA, 131–142. https://doi.org/10.1145/503032.503043
Pierre Bouillot. 2012. A relaxation of Coq’s guard condition. In JFLA - Journées Francophones des langages applicatifs - 2012. Carnac, France, 1 – 14. https://hal.archives-ouvertes.fr/hal-00651870
William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. 2017. Type-preserving CPS Translation of Σ and Π Types is Not Not Possible. Proc. ACM Program. Lang. 2, POPL, Article 22 (Dec. 2017), 33 pages. https://doi.org/10.1145/3158110
Youyou Cong and Kenichi Asai. 2018. Handling Delimited Continuations with Dependent Types (Technical Appendix). (June 2018). https://github.com/YouyouCong/icfp18/
Thierry Coquand. 1989. Metamathematical investigations of a calculus of constructions. Ph.D. Dissertation. INRIA.
Thierry Coquand and Christine Paulin. 1990. Inductively Defined Types. In Proceedings of the International Conference on Computer Logic (COLOG ’88). Springer-Verlag, London, UK, 50–66. http://dl.acm.org/citation.cfm?id=646125.758641
Pierre-Louis Curien and Hugo Herbelin. 2000. The duality of computation. In ACM sigplan notices, Vol. 35. ACM, 233–243.
Olivier Danvy. 1996. Type-directed Partial Evaluation. In Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’96). ACM, New York, NY, USA, 242–257. https://doi.org/10.1145/237721.237784
Olivier Danvy and Andrzej Filinski. 1989. A functional abstraction of typed contexts. Technical Report. University of Copenhagen.
Olivier Danvy and Andrzej Filinski. 1990. Abstracting control. In Proceedings of the 1990 ACM conference on LISP and functional programming. ACM, 151–160.
Olivier Danvy and Andrzej Filinski. 1992. Representing control: a study of the CPS transformation. Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Commun. ACM 18, 8 (Aug. 1975), 453–457. https://doi.org/10.1145/360933.360975
Andrzej Filinski. 1994. Representing Monads. In Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’94). ACM, New York, NY, USA, 446–457. https://doi.org/10.1145/174675.178047
Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. 2007. Adding Delimited and Composable Control to a Production Programming Environment. In Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP ’07). ACM, New York, NY, USA, 165–176. https://doi.org/10.1145/1291151.1291178
Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2017. On the Expressive Power of User-defined Effects: Effect Handlers, Monadic Reflection, Delimited Control. Proc. ACM Program. Lang. 1, ICFP, Article 13 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110257
Eduardo Giménez. 1995. Codifying Guarded Definitions with Recursive Schemes. In Selected Papers from the International Workshop on Types for Proofs and Programs (TYPES ’94). Springer-Verlag, London, UK, 39–59. http://dl.acm.org/citation.
Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures dans l’arithmétique*. Ph.D. Dissertation. Université Paris VII.

Georges Gonthier. 2008. The four colour theorem: Engineering of a formal proof. In *Computer Mathematics*. Springer, 333–333.

Timothy G. Griffin. 1990. A Formulae-as-Types Notion of Control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL ’90). ACM, New York, NY, USA, 47–58. https://doi.org/10.1145/96709.96714

Hugo Herbelin. 2005. On the degeneracy of $\Sigma$-types in presence of computational classical logic. In *International Conference on Typed Lambda Calculi and Applications* (TLCA ’05). Springer, 209–220.

Hugo Herbelin. 2012. A constructive proof of dependent choice, compatible with classical logic. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science* (LICS ’12). IEEE Computer Society, 365–374.

Hugo Herbelin and Silvia Ghilezan. 2008. An Approach to Call-by-name Delimited Continuations. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL ’08). ACM, New York, NY, USA, 383–394. https://doi.org/10.1145/1328438.1328484

Danko Ilik. 2012. Delimited control operators prove double-negation shift. *Annals of Pure and Applied logic* 163, 11 (2012), 1549–1559.

Limin Jia, Jianzhou Zhao, Vilhelm Sjöberg, and Stephanie Weirich. 2010. Dependent Types and Program Equivalence. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL ’10). ACM, New York, NY, USA, 275–286. https://doi.org/10.1145/1706299.1706333

Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2009. Shifting the Stage: Staging with Delimited Control. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (PEPM ’09). ACM, New York, NY, USA, 111–120. https://doi.org/10.1145/1480945.1480962

Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. 2006. Delimited dynamic binding. In *The 33th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages* (POPL ’06). ACM, 26–37.

Rodolphe Lepigre. 2016. A classical realizability model for a semantical value restriction. In *European Symposium on Programming Languages and Systems* (ESOP ’16). Springer, 476–502.

Etienne Miquey. 2017. A classical sequent calculus with dependent types. In *European Symposium on Programming*. Springer, 777–803.

Matthieu Sozeau. 2008. *Un environnement pour la programmation avec types dépendants*. Ph.D. Dissertation. Université de Paris-Sud. Faculté des Sciences d’Orsay (Essonne).

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure Distributed Programming with Value-dependent Types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (ICFP ’11). ACM, New York, NY, USA, 266–278. https://doi.org/10.1145/2034773.2034811

Étienne Miquey. 2017. A classical sequent calculus with dependent types. In *European Symposium on Programming*. Springer, 777–803.
Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-monadic Effects in F*. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’16). ACM, New York, NY, USA, 256–270. https://doi.org/10.1145/2837614.2837655

Asami Tanaka and Yukiyoshi Kameyama. 2012. A Call-by-Name CPS Hierarchy. In Functional and Logic Programming, Tom Schrijvers and Peter Thiemann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 260–274.

The Coq Development Team. 2018. The Coq Proof Assistant Reference Manual. https://coq.inria.fr/refman/

Hayo Thielecke. 2003. From Control Effects to Typed Continuation Passing. In Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages” (POPL ’03).

Matthijs Vákár. 2017. In Search of Effectful Dependent Types. Ph.D. Dissertation. Oxford University.

Philip Wadler. 1989. Theorems for Free!. In Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA ’89). ACM, ACM, New York, NY, USA, 347–359. https://doi.org/10.1145/99370.99404

Benjamin Werner. 1994. Une Théorie des Constructions Inductives. Ph.D. Dissertation. Université Paris-Diderot - Paris VII. https://tel.archives-ouvertes.fr/tel-00196524

Andrew K Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. Information and computation 115, 1 (1994), 38–94.

Noam Zeilberger. 2010. Polarity and the logic of delimited continuations. In Logic in Computer Science (LICS), 2010 25th Annual IEEE Symposium on. IEEE, 219–227.