# Combining Control Operators and Dependent Types

Youyou Cong
Ochanomizu University
so.yuyu@is.ocha.ac.jp

## ABSTRACT

Dependent types, which are types that depend on terms, are used to provide safety guarantees of programs. Meanwhile, control operators, which are a tool for manipulating control flow, are used to increase a programming language's expressiveness. These language constructs have separately yielded a wide range of interesting applications. However, previous work has shown that it is difficult to have them both, as their combination gives rise to a sort of type dependency that prevents us from viewing types as safety certificates. The key to mixing the two constructs is therefore to find a way to avoid the problematic dependency.

In this study, we build a dependently typed language that has a powerful variant of control operators. Our contributions are two-fold. First, we identify a set of restrictions that are necessary for making the language meaningful. Second, we design a sound type system that imposes the restrictions. The resulting language enjoys pleasant properties, leaving the possibility of extensions for making the language more realistic.

## 1 PROBLEM AND MOTIVATION

Types are a powerful tool to ensure safety of programs. As Milner [1978] states in his seminal paper, *well-typed programs cannot go wrong*, that is, once a program has passed type checking, we know that it never raises type errors at runtime.

The scope of errors detectable as type mismatch has been significantly extended by *dependent types*, which are types depending on terms such as numbers and strings. Dependency on terms allows us to write programs that are guaranteed to satisfy non-trivial properties. For instance, we can represent the property "a list containing $n$ natural numbers" as a type $\mathrm{L}(n)$, and using this type, we are able to define an error-free `head` function, which returns the first element of a *non-empty* list. The idea is to define `head` as only accepting lists of type $\mathrm{L}(n+1)$, for some natural number $n$. With this definition, application of `head` to an empty list results in a type error, since an empty list has type $\mathrm{L}(0)$ and 0 cannot be represented as $n+1$. Thus, dependent types enable implementation of high-assurance programs, where all functions are used in the intended way, producing values satisfying expected properties.

The expressive power of dependent types has also broadened the famous *Curry-Howard correspondence*, which relates types to propositions, and terms to proofs of the proposition their type represents. A dependent type system is as expressive as predicate logic, and this means we can prove propositions expressible in predicate logic, such as associativity of addition, by writing dependently typed programs. Languages with dependent types, like Coq [The Coq Development Team 2016] and Agda [Norell 2007], are therefore called *proof assistants*, and have been used to formally certify correctness of real-world software, including compilers [Leroy 2006], OS kernels [Gu et al. 2016], and cryptography frameworks [Barthe et al. 2009].

Meanwhile, *control operators* have been extensively studied in order to increase a programming language's expressiveness and convenience. These operators turn program contexts, often referred to as *continuations*, into user-accessible objects, enabling sophisticated manipulation of control flow. For example, we can "forget" the rest of the computation by capturing the current continuation and then dropping it. This allows us to simulate error handling in languages without support for exceptions. By capturing continuations, we can also suspend subsequent computations to an arbitrary execution point. This is the key idea underlying the implementation of the PLT Scheme Web Server [Krishnamurthi et al. 2007], which uses control operators to describe user-browser interactions in a correct and concise manner. Furthermore, if we resume a continuation multiple times with different arguments, we can make a program behave non-deterministically. This ability helps us naturally implement backtracking search, which is a well-known application of control operators.

The advantage of having control operators is that we can manipulate flow of control without making it explicit everywhere in a program. There is a trick, called the *continuation-passing style (CPS) translation*, to handle continuations in a language that does not have control operators. The idea is to let every function receive an additional parameter representing the surrounding context, but this would clutter program structure and in many cases leads to loss of efficiency. Therefore, integration of control operator is vital to achieve convenience, performance, and readability at the same time.

*Conference'17, July 2017, Washington, DC, USA*
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

This study is motivated by a somehow disappointing fact: no existing programming language provides both dependent types and control operators. What prevents us from combining these constructs is their contrasting nature. Since dependent types are used for verification purposes, they must be *statically* determined, that is, type checking of a program should depend only on information that is locally available at compile time. By contrast, since control operators interact with contexts, their behavior is *dynamically* determined, *i.e.*, interpretation of effectful terms requires non-local information. In a simply typed language, this causes no serious problem since types and control operators reside in different worlds. However, in a dependent setting, dynamic behavior of control operators can flow into types, breaking the "types-as-static-objects" principle. Indeed, a negative result has already been reported by Herbelin [2005]: unconstrained use of control operators in the presence of dependent types makes the calculus inconsistent when viewed as a logic. Nevertheless, the combination is not entirely impossible; we can retain consistency by appropriately restricting terms appearing in types, as shown by Herbelin [2012].

We present a dependently typed language with `shift` and `reset`, a pair of powerful control operators proposed by Danvy and Filinski [1990]. Our long-term goal is to build a realistic language where we enjoy both safety and expressiveness. In this work, we take one step forward by considering minimal but non-trivial language features.

## 2 BACKGROUND AND RELATED WORK

*Dependent Types and Control Operators.* Herbelin [2005] designs a calculus with a dependent variant of the pair types, which is interpreted as existential quantification, and `call/cc`, which is a control operator supported in the Scheme language. It turns out that the resulting calculus is inconsistent, as one can prove absurd propositions like $0 = 1$. The inconsistency comes from the dynamic nature of `call/cc`: when building a proof using this operator, it may be witnessed by different terms depending on the surrounding context. In a dependently typed calculus like Herbelin's, the witnesses can appear in types, allowing us to derive an inconsistent conclusion. What this means is that when extending a dependently typed language with `call/cc` without any restriction, the language no longer serves as a proof assistant, since any proposition is provable.

In his later work, Herbelin [2012] establishes a syntactic restriction called the *Negative-Elimination-Free (NEF)* condition. NEF terms are defined as being free from control effects, hence they behave the same way in whatever context. He shows that when requiring types to depend only on NEF terms, integration of `call/cc` does not lead to inconsistency. Recent work by Miquey [2017] shows that a similar restriction further allows us to design a dependent version of the $\lambda\mu\tilde{\mu}$-calculus [Curien and Herbelin 2000], a sequent-style calculus with support for manipulation of control via value- and context-abstractions.

*Dependent Types and Effects.* To obtain a language that is both safe and expressive, researchers have also been looking at the combination of dependent types and *effects*, such as read and write operations. Idris [Brady 2013] and F$^\star$ [Swamy et al. 2016] are proof assistants with a rich dependent type system as well as a user-extensible effect facility. The latter language, while still young, has already yielded practically interesting applications, including safe distributed programming and efficient heaps [Swamy et al. 2011, 2016]. Ahman [2017] and Vákár [2017] study dependent and effectful calculi from a theoretical point of view, and give a semantic account in terms of category theory. Interestingly, a majority of languages developed in this line of work share the principle "types may depend only on effect-free terms", which is exactly the approach taken by Herbelin [2012] and Miquey [2017]. This is not a mere coincidence, because effects communicate with the outside world, just like control operators interact with their surroundings.

## 3 APPROACH AND UNIQUENESS

We present $\lambda_{\Pi}^{s/r}$, a dependently typed language with the `shift` and `reset` operators. Compared with previous work on dependent types and control operators, our work is unique in two ways. First, while previous studies are centered around logical interests, our study is more programming-oriented. Specifically, our language includes conventional programming constructs such as lists and pattern-matching. Incorporating these constructs is not an easy matter, though, since they are known to misbehave in a dependently typed setting [Barthe and Uustalu 2002]. However, it turns out that the negative result does not apply if we restrict type dependency to pure, effect-free terms.

The second uniqueness is that while previous work considers *undelimited* control, which always deals with the entire continuation surrounding a term, we discuss *delimited* control, where we talk about continuations with a fixed extent specified by a delimiter. Delimited control is more powerful than undelimited control, since reified contexts return and compose like normal functions [Downen and Ariola 2014]. The additional feature, however, requires a slightly complicated type system to account for [Danvy and Filinski 1989]. We observe that when integrating delimited control, we have to take special care of dependencies associated with captured continuations.

The difference between $\lambda_{\Pi}^{s/r}$ and the existing dependent, effectful languages is that our language does not require external devices to define its meaning. The reason is that there is a semantics-preserving translation from a `shift`/`reset` calculus to the plain $\lambda$-calculus [Barendregt 1984], which is the basis of functional programming languages as well as proof assistants. Although `shift` and `reset` do not cover the full expressiveness of effects, their simple semantics makes them well-suited for use in theorem proving, since interpretation of proofs (*i.e.*, programs) can be done within the underlying logic of existing proof assistants.

Now we describe the design of $\lambda_{\Pi}^{s/r}$. We equip the language with the following syntactic constructs:

- basic terms from the $\lambda$-calculus
  (variables, functions, function application)
- built-in inductive datatypes
  (units, natural numbers, length-indexed lists)
- conventional programming constructs
  (recursive functions, pattern-matching, the let-expression)
- the `shift` and `reset` operators
  (denoted $\mathcal{S}$ and $\langle\,\rangle$, respectively)

Note that our language is intended to serve both as a proof assistant and as a general-purpose programming language. Therefore, we include constructs that are useful from each perspective. For instance, inductive datatypes and recursive functions do not have a logical meaning, but they are essential when writing programs.

The crux of this work is to identify a set of restrictions that make our language meaningful, and to impose the restrictions in an appropriate manner. This requires a rough understanding of how `shift` and `reset` behave. Let us look at the example below:

(1) $\texttt{length}\,\langle 1 + (\mathcal{S}k.\,[k\ 2])\rangle$

Here, `length` is a function that computes the length of a given list. As a notational convention, we represent application of function $f$ to arguments $a$, $b$, $c$ as $f\ a\ b\ c$, and a list of elements $a$, $b$, $c$ as $[a, b, c]$. Now we observe how program (1) evaluates:

$$\begin{aligned}\texttt{length}\,\langle 1 + (\mathcal{S}k.\,[k\ 2])\rangle &= \texttt{length}\,\langle[(\texttt{fun}\ (x : \mathbb{N}).\ 1 + x)\ 2]\rangle\\ &= \texttt{length}\,\langle[3]\rangle\\ &= \texttt{length}\,[3]\\ &= 1\end{aligned}$$

The `shift` operator captures the surrounding context up to the `reset` clause, namely $1 + \ldots$, where the dots denote the hole of the context. The context is turned into an anonymous function $\texttt{fun}\ (x : \mathbb{N}).\ 1 + x$, which takes in a natural number $x$ and returns $1 + x$. Then, we replace what is inside `reset` by $[k\ 2]$, with $k$ bound to the reified context. Since $k\ 2$ is equal to $1 + 2 = 3$, the `reset` clause evaluates to a one-element list [3], and after the application of the `length` function, we obtain 1 as the value of the whole term.

We say a term $e$ is impure when $e$ or its subterm manipulates the context surrounding $e$. For example, the `shift` clause and the addition $1 + (\mathcal{S}k.\,[k\ 2])$ in (1) are both impure terms. When $e$ involves no interaction with its surroundings, we say $e$ is a pure term. For instance, the numbers and the `reset` clause in (1) are pure terms, and the whole program is also pure. Purity of a term is determined by its syntactic structure as well as purity of the subterms. Constants, functions, and `reset` clauses are syntactically pure terms, whereas `shift` clauses are syntactically impure. A function application is pure if the function, its body, and the argument are all pure; otherwise it is an impure term.

Besides the ability of accessing contexts, there is another difference between pure and impure terms: the former evaluate to a value (if they terminate), while the latter do not. Observe the pure terms in (1): the numbers are obviously values, and as we can see from the reduction sequence, the `reset` clause and the whole term evaluate to values [3] and 1, respectively. In contrast, the `shift` clause, which is an impure term, does not have a value on its own. Recall that `shift` captures a continuation *delimited by a* `reset`. This means that when we have a bare `shift` clause that does not have a matching `reset`, we cannot apply any reduction rule to it. Similarly, the impure subterm $1 + (\mathcal{S}k.\,[k\ 2])$ is also a stuck term; the computation $k\ 2$ happens only when the addition is surrounded by a `reset` as in (1).

Now we present three restrictions we impose on our language. The first one is essentially the NEF condition of Herbelin [2012]: *types should not depend on impure terms*. To see why we need this restriction, let us briefly explain how we perform type checking in a dependently typed language. Since dependent types contain terms, deciding equality between types requires evaluation of terms inside types. Suppose we have a function application $f\ a$, where $f$ is a function requiring a three-element list of type $\mathrm{L}(3)$, whereas the actual argument $a$ is of type $\mathrm{L}(1 + 2)$. In this case, the application is type-safe because $1 + 2$ evaluates to 3. Now suppose we have a term $b$ of type $\mathrm{L}(\mathcal{S}k.\,3)$. To type check $f\ b$, we have to locally evaluate $\mathcal{S}k.\,3$ to a value, and then compare it with 3. However, as we have stated above, we cannot evaluate this `shift` clause because it is a stuck term. What this suggests is that if we have types depending on impure terms, the standard method of type checking no longer works. Hence, we rule out dependency on impure terms by imposing purity conditions on formation rules of types and typing rules of terms. For instance, in the rule for generating a well-formed indexed list type $\mathrm{L}(e)$, we require that the index $e$ is a pure term, since it appears inside a type.

The second restriction is specific to languages with delimited control: *continuations should not have a dependent*

*function type*. In simply typed languages, a function from $A$ to $B$ has a type of the form $A \to B$. In dependently typed languages, function types have the form $(x : A) \to B$, where type $B$ may contain free occurrences of variable $x$ representing the argument. Now recall that a continuation captured by a `shift` operator is a function. This means that in a dependently typed setting, the return type of a continuation may depend on its argument. However, if we allow this dependency, type checking of a program would require the information how we use a captured continuation. To keep track of all arguments that may be passed to a continuation, we need to look at the whole program, but this breaks the principle that type checking only uses local information. Therefore, we disallow dependency on continuation arguments by equipping the typing rule of $\mathcal{S}k.\, e$ with an additional premise, requiring that $k$ has a non-dependent function type. Note that when dealing with undelimited control, we do not need this restriction because undelimited continuations do not return normally, *i.e.*, they do not have a return type.

The third restriction is also a novel one: *types should not depend on continuations*. As the reader may have noticed, the $\mathcal{S}$ operator in $\mathcal{S}k.\, e$ serves as a continuation binder, making the variable $k$ available in the body $e$. In a dependently typed setting, this implies that the type of $e$ may depend on the captured continuation. However, if we allow this dependency, type checking would require the information what context is surrounding a `shift` clause. To figure it out, we again need to look at the whole program, which is not allowed in static type checking. For this reason, we put another premise in the typing rule of $\mathcal{S}k.\, e$, ensuring that $k$ does not appear in the type of $e$. Note that in languages with undelimited control operators, this restriction is redundant because they are not flexible enough to yield the problematic dependency.

## 4 RESULTS AND CONTRIBUTIONS

*Type Soundness.* The main result is that when imposing the three restrictions on type dependency, we can prove that the type system of $\lambda_{\Pi}^{s/r}$ is *sound* [Wright and Felleisen 1994], *i.e.*, well-typed pure terms evaluate to a value of the same type, if they terminate. We prove this by showing the following properties:

Theorem 4.1. *Evaluation of well-typed terms do not change their type and purity. That is, if a pure (resp. impure) term $e$ has type $A$ and $e$ evaluates to $e'$, then $e'$ is a pure (resp. impure) term of type $A$.*

Theorem 4.2. *Well-typed, closed pure terms do not get stuck. That is, if we can type $e$ in an empty context, then either $e$ is an irreducible value, or $e$ takes at least one step.*

Both theorems can be proven by induction on the typing derivation of $e$. The second one, known as the *progress* property, only holds for pure terms because evaluation of impure terms is not possible in general.

*Application.* To demonstrate the practical impact of our work, we show a $\lambda_{\Pi}^{s/r}$ program that uses effects while being dependently typed:

$$\langle \texttt{let}\ l = [\texttt{get}\ (), (\texttt{tick}\ (); \texttt{get}\ ())]\ \texttt{in}\ \texttt{fun}\ (s : \mathbb{N}).\, l \rangle\ 0 = [0, 1]$$

where

$$\texttt{get} \stackrel{\text{def}}{\equiv} \texttt{fun}\ (x : \mathsf{Unit}).\, \mathcal{S}k.\, \texttt{fun}\ (s : \mathbb{N}).\, k\ s\ s$$

$$\texttt{tick} \stackrel{\text{def}}{\equiv} \texttt{fun}\ (x : \mathsf{Unit}).\, \mathcal{S}k.\, \texttt{fun}\ (s : \mathbb{N}).\, k\ ()\ (s + 1)$$

The program builds a two-element list (of type L(2)) using the value of a mutable state, which we simulate using control operators [Asai and Kiselyov 2011]. When we start the computation, the state is initialized to 0 outside the `reset` clause. In the list we are building, we find two functions `get` and `tick`. The former returns the current value of the state without changing it, whereas the latter returns nothing but increments the state. We implement these behaviors by cleverly using the `shift` operator; the idea that when we access or update the state, we capture the continuation, ask for the current value $s$ of the state, and resume the continuation with an appropriate value ($s$ or the unit value ()) and a possibly new state ($s$ or $s + 1$). Observe that since the second access takes place after an update, the `get` function returns a different value at each call, which is something that does not happen in the mainstream proof assistants.

*Further Extension.* In this work, we considered a rather simple language, but as long as we keep the three restrictions on type dependency, we can enrich $\lambda_{\Pi}^{s/r}$ with more language features and make it more practical. One interesting extension would be supporting user-defined inductive types. In proof assistants like Coq, dependent inductive types are widely used to define data having a particular property, such as binary tree and length-indexed arrays. Using `shift` and `reset`, we would be able to write programs that manipulate these data in a sophisticated manner.

Another extension worth looking at is to integrate exceptions, mutable references, and other frequently used effects as primitive constructs. Although effects do not have a logical interpretation, they allow us to lift various existing applications of delimited control to a dependently typed setting. One example is partial evaluation, a program optimization technique for producing faster code. The main task of partial evaluation is called *let-insertion*, which inserts the let-expressions into programs to specify evaluation order and prevent code duplication. Danvy [1996] implements

let-insertion in an elegant way using the `shift` and `reset` operators, together with a mutable reference for generating fresh variables used in the let-expressions.

In summary, we have shown how to combine control operators and dependent types in order to obtain a safe and convenient language. Our results open the possibility of extending proof assistants with a restricted form of delimited control, which we believe will yield practically interesting applications, as well as novel proof techniques.

## REFERENCES


Danel Ahman. 2017. Handling Fibred Algebraic Effects. *Proc. ACM Program. Lang.* 2, POPL, Article 7 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158095

Kenichi Asai and Oleg Kiselyov. 2011. Introduction to programming with shift and reset. In *ACM SIGPLAN Continuation Workshop 2011*.

Hendrik Pieter Barendregt. 1984. *The lambda calculus*. Vol. 3. North-Holland Amsterdam.

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. 2009. Formal Certification of Code-Based Cryptographic Proofs. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*. ACM, 90–101. http://dx.doi.org/10.1145/1480881.1480894

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

Edwin Brady. 2013. Programming and Reasoning with Algebraic Effects and Dependent Types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2500365.2500581

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.

Paul Downen and Zena M Ariola. 2014. Delimited control and computational effects. *Journal of functional programming* 24, 1 (2014), 1–55.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 653–669. http://dl.acm.org/citation.cfm?id=3026877.3026928

Hugo Herbelin. 2005. On the degeneracy of Σ-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.

Shriram Krishnamurthi, Peter Walton Hopkins, Jay McCarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias Felleisen. 2007. Implementation and Use of the PLT Scheme Web Server. *Higher Order Symbolic Computation* 20, 4 (Dec. 2007), 431–460. https://doi.org/10.1007/s10990-007-9008-y

Xavier Leroy. 2006. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*. ACM Press, 42–54. http://gallium.inria.fr/~xleroy/publi/compiler-certif.pdf

Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences* 17, 3 (1978), 348–375.

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

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Chalmers University of Technology.

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

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

The Coq Development Team. 2016. The Coq Proof Assistant Reference Manual. (Nov. 2016). https://web.archive.org/web/https://coq.inria.fr/doc/Reference-Manual006.html

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

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