# Towards Dependently-Typed Control Effects

## Extended Abstract

Youyou Cong
Tokyo Institute of Technology
Tokyo, Japan
cong@c.titech.ac.jp

Kenichi Asai
Ochanomizu University
Tokyo, Japan
asai@is.ocha.ac.jp

## 1 Introduction

Dependent types and computational effects are indispensable for safe implementation of realistic programs. The past decade has seen several languages designed for effectful programming with dependent types, as well as their applications in diverse domains. For instance, Brady [4] implements an effect library in the Idris language, using state-indexed types to statically enforce resource access protocols. As another example, Maillard et al. [11] formalize an effect framework in the F$^\star$ language, using monad-indexed types to enable verification of user-defined effects.

In this abstract, we consider a dependently-typed language that has delimited control operators `shift` and `reset` [7]. These operators are useful for programming: as shown by Filinski [8], `shift` and `reset` can express any monadic effects, including exceptions, non-determinism, and mutable state. They are also useful for proving: as shown by Herbelin [9] and Ilik [10], `shift` and `reset` can prove theorems that are not provable in intuitionistic logic, such as Markov's principle and Double Negation Shift.

The combination of dependent types and `shift`/`reset` has previously been studied by Cong and Asai [5, 6]. In their first paper [5], they define a type system where types may depend only on *pure* terms, *i.e.*, terms that do not execute the `shift` operator. They also define a CPS translation of the language, serving as an elaboration into the $\lambda$-calculus. Thanks to the restriction on type dependency, they were able to prove type preservation of the CPS translation along the same lines of Bowman et al. [3], who establish a type-preserving CPS translation for a pure dependently-typed language. However, they assume that the target language permits parametricity reasoning, which is undesirable because there are type theories in which parametricity does not hold [2]. In their follow-up paper [6], Cong and Asai propose to use a *selective* CPS translation [12], which converts effectful terms into CPS and keeps pure terms in direct style. By being selective, they were able to prove type preservation without relying on parametricity, but they do not discuss whether the approach scales to a larger language.

To answer the question left by previous work, we extend Cong and Asai's [6] language with type- and effect-level conditionals. Our key observation is that having effect-level conditionals makes it challenging to type and CPS-translate programs.

*TyDe '22, September 11, Ljubljana, Slovenia*
2022.

$$
\begin{array}{rcll}
K & ::= & * \mid \square & \text{Kinds} \\
A, B, \alpha & ::= & C \mid \Pi x :^{\epsilon} A.\, B \mid \texttt{if}\ e\ \texttt{then}\ A\ \texttt{else}\ B & \text{Types} \\
\epsilon & ::= & \iota \mid \alpha & \text{Effects} \\
v & ::= & c \mid x \mid \lambda x.\, e & \text{Values} \\
e & ::= & v \mid e\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \mathcal{S}k.\, e \mid \langle e \rangle & \text{Terms}
\end{array}
$$

$$
F ::= [\,] \mid F\ e \mid v\ F \mid \texttt{if}\ F\ \texttt{then}\ e\ \texttt{else}\ e \qquad \text{Pure Contexts}
$$

$$
\begin{array}{rcll}
(\lambda x.\, e)\ v & \triangleright & e[v/x] & (\beta) \\
\texttt{if true then}\ e_1\ \texttt{else}\ e_2 & \triangleright & e_1 & (\mathbb{B}_1) \\
\texttt{if false then}\ e_1\ \texttt{else}\ e_2 & \triangleright & e_2 & (\mathbb{B}_2) \\
\langle F[\mathcal{S}k.\, e] \rangle & \triangleright & \langle e[\lambda x.\, \langle F[x] \rangle / k] \rangle & (\mathcal{S}) \\
\langle v \rangle & \triangleright & v & (\langle\rangle)
\end{array}
$$

**Figure 1.** $\lambda_{\mathcal{S}}$ Syntax and Reduction

## 2 A Language with Shift/Reset and Type-Level Conditionals

In this section, we consider $\lambda_{\mathcal{S}}$, a dependently-typed language with `shift`/`reset` and type-level conditionals. The language allows us to define functions that return different types of values depending on their arguments. As an example, consider the `div` function below, which returns an error message when the divisor is zero:

```
let div x y =
  if y = 0 then "div by zero" else x / y
```

We can assign `div` the following type (where the annotation $\iota$ means the function body is pure):

$$\Pi x :^{\iota} \texttt{int}.\, \Pi y :^{\iota} \texttt{int}.\, \texttt{if}\ y = 0\ \texttt{then string else int}$$

Notice that the type of the function body is a conditional that depends on the value $y$. This dependency allows us to return a string in the error case.

### 2.1 Syntax and Reduction

We define the syntax and reduction of $\lambda_{\mathcal{S}}$ in Figure 1. Expressions are stratified into five categories: kinds, types, effects, values, and terms. The star kind $*$ is the kind of types, whereas the box kind $\square$ is the kind of $*$. Base types $C$ are inhabited by constants $c$. Function types $\Pi x :^{\epsilon} A. B$ carry an effect $\epsilon$ of the function body, which is either $\iota$ (pure) or an *answer type* (return type of the continuation to be captured). Note that function types may involve dependency on the argument $x$ in the codomain $B$; we use arrow types $A \xrightarrow{\epsilon} B$ when there is no such dependency. Term-level conditionals may introduce corresponding type-level conditionals. The `shift` construct $\mathcal{S}k.\, e$ and `reset` construct $\langle e \rangle$ serve as a control trigger and delimiter, respectively.

Terms in $\lambda_{\mathcal{S}}$ are evaluated under the call-by-value, left-to-right evaluation strategy. As defined by rules ($\mathcal{S}$) and ($\langle\rangle$), `shift` captures the continuation delimited by the closest `reset` operator, and `reset` returns the value of its body as the eventual answer within a delimited context.

### 2.2 Typing

Having defined the syntax and reduction, we design a type system of $\lambda_{\mathcal{S}}$. Following Cong and Asai [6], we maintain a fine-grained distinction between pure and effectful terms. This allows us to have more terms in types, and to define a faster CPS translation.

In Figure 2, we present the typing rules of terms (we ignore the right-hand side of $\rightsquigarrow$ for now). A typing judgment takes the form $\Gamma \vdash e : A \,!\, \epsilon$, which reads: term $e$ has type $A$ and effect $\epsilon$ under environment $\Gamma$. When $\epsilon = \iota$, we call $e$ a pure term. When $\epsilon = \alpha$ for some type $\alpha$, we call $e$ effectful.

Let us go through individual rules with a focus on effect assignment. Values and `reset` constructs are all judged pure, whereas `shift` constructs are judged effectful. Applications and conditionals are pure if their subterms are all pure. Note that conditionals must consist of branches having the same effect. This restriction limits expressiveness of types but not typability of terms: when one branch is pure and the other is effectful, we can make the whole conditional well-typed by casting the pure branch into an effectful one via (Exp).

Let us now shift our attention to type dependency. When typing a dependent application (DApp), whose result type depends on the argument $e_2$, we require $e_2$ to be a pure term. Similarly, when typing a dependent conditional (DIf), whose result type depends on test term $e_1$, we require $e_1$ to be a pure term. By imposing these requirements, we obtain the guarantee that terms appearing in types are never translated into CPS. This eliminates the need for parametricity in the type preservation proof.

### 2.3 CPS Translation

Guided by the typing rules, we define a selective CPS translation of $\lambda_{\mathcal{S}}$. The target of the translation is a pure $\lambda$-calculus in which one can reason about dependent conditionals using equality information. We show the key rules in Figure 3; note that the idea of using equalities is borrowed from existing dependent type systems [1, 6, 13].

The translation is defined on the typing derivation, and its output is written on the right-hand side of $\rightsquigarrow$ in the typing rules (Figure 2). Pure terms are uniformly translated to a direct-style term, whereas effectful terms are all translated to a continuation-taking function. For applications, conditionals, and control constructs, there is one CPS image for each combination of the effects of their subterms.

The CPS translation is type-preserving, that is, it converts a well-typed $\lambda_{\mathcal{S}}$ term into a well-typed pure $\lambda$-term.

**Theorem 2.1** (Type Preservation of CPS Translation). *Let $\Gamma'$, $A'$, and $\alpha'$ be the CPS translation of $\Gamma$, $A$, and $\alpha$.*

1. *If $\Gamma \vdash e : A \,!\, \iota \rightsquigarrow e'$, then $\Gamma' \vdash e' : A'$.*
2. *If $\Gamma \vdash e : A \,!\, \alpha \rightsquigarrow e'$, then $\Gamma' \vdash e' : (A' \rightarrow \alpha') \rightarrow \alpha'$.*

*Proof.* By induction on the derivation of $e$. As an example, consider one case of the (DIf) rule, where $\epsilon = \alpha$. Our goal is to show

$$\begin{gathered}\Gamma' \vdash \lambda k.\, \texttt{if}\ e_1{}'\ \texttt{then}\ e_2{}'\ k\ \texttt{else}\ e_3{}'\ k : \\ ((\texttt{if}\ e_1{}'\ \texttt{then}\ A'\ \texttt{else}\ B') \rightarrow \alpha') \rightarrow \alpha'\end{gathered}$$

By the induction hypothesis, we have

$$\begin{gathered}\Gamma' \vdash e_1{}' : \texttt{bool} \\ \Gamma' \vdash e_2{}' : (A' \rightarrow \alpha') \rightarrow \alpha' \\ \Gamma' \vdash e_3{}' : (B' \rightarrow \alpha') \rightarrow \alpha'\end{gathered}$$

To type the application $e_2{}'\ k$, we use the equivalence $e_1{}' \equiv \texttt{true}$ provided by [DIf] to convert the type of $k$ to $A' \rightarrow \alpha'$. We do the same for the other application $e_3{}'\ k$, this time using $e_1{}' \equiv \texttt{false}$. Now we can conclude that the conditional $\texttt{if}\ e_1{}'\ \texttt{then}\ e_2{}'\ k\ \texttt{else}\ e_3{}'\ k$ has type $\texttt{if}\ e_1{}'\ \texttt{then}\ \alpha'\ \texttt{else}\ \alpha'$, which is equivalent to $\alpha'$. This implies the goal. $\square$

## 3 Extension with Effect-Level Conditionals

We now extend $\lambda_{\mathcal{S}}$ with effect-level conditionals. This extension allows us to assign precise effects to functions. As an example, consider the `div2` function below, which aborts the computation when the divisor is zero:

```
let rec div2 x y =
  if y = 0 then shift k "div by zero" else x / y
```

We can assign `div2` the following type (using equivalence rules in Figure 4 of the appendix):

$$\Pi x :^{\iota} \texttt{int}.\, \Pi y :^{\texttt{if}\ y=0\ \texttt{then}\ \texttt{int}\ \texttt{else}\ \iota} \texttt{int}.\, \texttt{int}$$

Notice that the effect of the function body is a conditional that depends on the value $y$. This dependency allows us to treat `div2` as a pure function if we restrict the second argument of `div2` to a non-zero integer.

With this motivating example in mind, we extend $\lambda_{\mathcal{S}}$ with conditional effects (Figure 6 in the appendix). We first enrich the syntax of effects with $\texttt{if}\ e\ \texttt{then}\ \epsilon\ \texttt{else}\ \epsilon$. We next change the typing rule of dependent conditionals so that the overall effect is $\texttt{if}\ e\ \texttt{then}\ \epsilon_2\ \texttt{else}\ \epsilon_3$, where $\epsilon_2$ and $\epsilon_3$ are the effects of the two branches.

While the extension appears to be simple, it poses a challenge to static reasoning of programs. In particular, when

$$\frac{c \text{ is a constant of type } C}{\Gamma \vdash c : C \,!\, \iota \rightsquigarrow c}\ (\textsc{Const}) \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A \,!\, \iota \rightsquigarrow x}\ (\textsc{Var}) \qquad \frac{\Gamma, x : A \vdash e : B \,!\, \epsilon \rightsquigarrow e'}{\Gamma \vdash \lambda x.\, e : (\Pi x :^{\epsilon} A.\, B) \,!\, \iota \rightsquigarrow \lambda x.\, e'}\ (\textsc{Abs})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (\Pi x :^{\epsilon_3} A.\, B) \,!\, \epsilon_1 \rightsquigarrow e_1' \quad \Gamma \vdash e_2 : A \,!\, \iota \rightsquigarrow e_2' \\ \text{if } \epsilon_1 = \epsilon_3 = \iota, \text{ then } \epsilon = \iota; \text{ otherwise } \epsilon_1, \epsilon_3 = \{\iota, \alpha\}, \epsilon = \alpha\end{array}}{\Gamma \vdash e_1\ e_2 : B[e_2/x] \,!\, \epsilon \rightsquigarrow \begin{cases} e_1'\ e_2' & \text{if } \epsilon_1 = \epsilon_3 = \iota \\ \lambda k.\, e_1'\ e_2'\ k & \text{if } \epsilon_1 = \iota, \epsilon_3 = \alpha \\ \lambda k.\, e_1'\ (\lambda v_1.\, k\ (v_1\ e_2')) & \text{if } \epsilon_1 = \alpha, \epsilon_3 = \iota \\ \lambda k.\, e_1'\ (\lambda v_1.\, v_1\ e_2'\ k) & \text{if } \epsilon_1 = \epsilon_3 = \alpha \end{cases}}\ (\textsc{DApp})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (A \xrightarrow{\epsilon_3} B) \,!\, \epsilon_1 \rightsquigarrow e_1' \quad \Gamma \vdash e_2 : A \,!\, \alpha \rightsquigarrow e_2' \\ \epsilon_1, \epsilon_3 = \{\iota, \alpha\}\end{array}}{\Gamma \vdash e_1\ e_2 : B[e_2/x] \,!\, \alpha \rightsquigarrow \begin{cases} \lambda k.\, e_2'\ (\lambda v_2.\, k\ (e_1'\ v_2)) & \text{if } \epsilon_1 = \epsilon_3 = \iota \\ \lambda k.\, e_2'\ (\lambda v_2.\, e_1'\ v_2\ k) & \text{if } \epsilon_1 = \iota, \epsilon_3 = \alpha \\ \lambda k.\, e_1'\ (\lambda v_1.\, e_2'\ (\lambda v_2.\, k\ (v_1\ v_2))) & \text{if } \epsilon_1 = \alpha, \epsilon_3 = \iota \\ \lambda k.\, e_1'\ (\lambda v_1.\, e_2'\ (\lambda v_2.\, v_1\ v_2\ k)) & \text{if } \epsilon_1 = \epsilon_3 = \alpha \end{cases}}\ (\textsc{SApp})$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \,!\, \iota \rightsquigarrow e_1' \quad \Gamma \vdash e_2 : A \,!\, \epsilon \rightsquigarrow e_2' \quad \Gamma \vdash e_3 : B \,!\, \epsilon \rightsquigarrow e_3'}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \texttt{if } e_1 \texttt{ then } A \texttt{ else } B \,!\, \epsilon \rightsquigarrow \begin{cases} \texttt{if } e_1' \texttt{ then } e_2' \texttt{ else } e_3' & \text{if } \epsilon = \iota \\ \lambda k.\, \texttt{if } e_1' \texttt{ then } e_2'\ k \texttt{ else } e_3'\ k & \text{if } \epsilon = \alpha \end{cases}}\ (\textsc{DIf})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \texttt{bool} \,!\, \alpha \rightsquigarrow e_1' \quad \Gamma \vdash e_2 : A \,!\, \epsilon \rightsquigarrow e_2' \quad \Gamma \vdash e_3 : A \,!\, \epsilon \rightsquigarrow e_3' \\ \epsilon = \{\iota, \alpha\}\end{array}}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : A \,!\, \alpha \rightsquigarrow \begin{cases} \lambda k.\, e_1'\ (\lambda v_1.\, k\ (\texttt{if } v_1 \texttt{ then } e_2' \texttt{ else } e_3')) & \text{if } \epsilon = \iota \\ \lambda k.\, e_1'\ (\lambda v_1.\, \texttt{if } v_1 \texttt{ then } e_2'\ k \texttt{ else } e_3'\ k) & \text{if } \epsilon = \alpha \end{cases}}\ (\textsc{SIf})$$

$$\frac{\Gamma, k : A \to B \vdash e : B \,!\, \epsilon \rightsquigarrow e' \quad \epsilon = \{\iota, B\}}{\Gamma \vdash \mathcal{S}k.\, e : A \,!\, B \rightsquigarrow \begin{cases} \lambda k.\, e'\ (\lambda x.\, x) & \text{if } \epsilon = \iota \\ \lambda k.\, e' & \text{if } \epsilon = B \end{cases}}\ (\textsc{Shift}) \qquad \frac{\Gamma \vdash e : A \,!\, \epsilon \rightsquigarrow e' \quad \epsilon = \{\iota, A\}}{\Gamma \vdash \langle e \rangle : A \,!\, \iota \rightsquigarrow \begin{cases} e' & \text{if } \epsilon = \iota \\ e'\ (\lambda x.\, x) & \text{if } \epsilon = A \end{cases}}\ (\textsc{Reset})$$

$$\frac{\Gamma \vdash e : A \,!\, \epsilon \rightsquigarrow e' \quad A \equiv B \quad \epsilon \equiv \epsilon'}{\Gamma \vdash e : B \,!\, \epsilon \rightsquigarrow e'}\ (\textsc{Conv}) \qquad \frac{\Gamma \vdash e : A \,!\, \iota \rightsquigarrow e' \quad \Gamma \vdash \alpha : *}{\Gamma \vdash e : A \,!\, \alpha \rightsquigarrow \lambda k.\, k\ e'}\ (\textsc{Exp})$$

**Figure 2.** $\lambda_{\mathcal{S}}$ Typing and CPS Translation

$$\frac{p : e \equiv e' \in \Gamma}{\Gamma \vdash e \equiv e'} \qquad \frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma, p : e_1 \equiv \texttt{true} \vdash e_2 : A \quad \Gamma, p : e_1 \equiv \texttt{false} \vdash e_3 : B}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \texttt{if } e_1 \texttt{ then } A \texttt{ else } B}\ [\textsc{DIf}]$$

**Figure 3.** Target Equivalence and Typing (excerpt)

a conditional effect has an open test term (such as `y = 0` in the `div2` example), it is not equivalent to the pure effect, nor is it an answer type. This prevents us from deciding whether a term may appear in a type, and whether it should be translated into CPS. On the other hand, at runtime, every computation that is actually executed must have a closed effect, which reduces to either $\iota$ or a type. We are currently seeking a sound semantics for conditional effects, and we hope to receive suggestions from the audience at the workshop.

## References


[1] Bruno Barras, Pierre Corbineau, Benjamin Grégoire, Hugo Herbelin, and Jorge Luis Sacchini. 2008. A new elimination rule for the calculus of inductive constructions. In *International Workshop on Types for Proofs and Programs (TYPES '08)*. Springer, 32–48.

[2] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The next 700 Syntactical Models of Type Theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs* (Paris, France) *(CPP 2017)*. Association for Computing Machinery, New York, NY, USA, 182–194. https://doi.org/10.1145/3018610.3018620

[3] 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

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

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

[6] Youyou Cong and Kenichi Asai. 2018. Shifting and Resetting in the Calculus of Constructions. The 19th International Symposium on Trends in Functional Programming (TFP 2018).

[7] Olivier Danvy and Andrzej Filinski. 1990. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*. ACM, 151–160.

[8] Andrzej Filinski. 1994. Representing Monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. ACM, New York, NY, USA, 446–457. https://doi.org/10.1145/174675.178047

[9] Hugo Herbelin. 2010. An intuitionistic logic that proves Markov's principle. In *25th Annual IEEE Symposium on Logic in Computer Science (LICS '10)*. IEEE, 50–56.

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

[11] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hriţcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra Monads for All. *Proc. ACM Program. Lang.* 3, ICFP, Article 104 (jul 2019), 29 pages. https://doi.org/10.1145/3341708

[12] Lasse R Nielsen. 2001. A selective CPS transformation. *Electronic Notes in Theoretical Computer Science* 45 (2001), 311–331.

[13] Matthieu Sozeau. 2008. *Un environnement pour la programmation avec types dépendants*. Ph. D. Dissertation. University of Paris-Sud. https://tel.archives-ouvertes.fr/tel-00640052

# A Elided Rules

$$\frac{t \text{ is a type, effect, or term}}{t \equiv t} \qquad \frac{t \equiv t'}{t' \equiv t} \qquad \frac{t \equiv t' \quad t' \equiv t''}{t \equiv t''}$$

$$\frac{A \equiv A' \quad B \equiv B' \quad \epsilon \equiv \epsilon'}{\Pi x :^{\epsilon} A.\, B \equiv \Pi x :^{\epsilon'} A'.\, B'}$$

$$\frac{}{\texttt{if true then } A \texttt{ else } B \equiv A} \qquad \frac{}{\texttt{if false then } A \texttt{ else } B \equiv B}$$

$$\frac{}{\texttt{if } e \texttt{ then } A \texttt{ else } A \equiv A} \qquad \frac{e \equiv e' \quad A \equiv A' \quad B \equiv B'}{\texttt{if } e \texttt{ then } A \texttt{ else } B \equiv \texttt{if } e' \texttt{ then } A' \texttt{ else } B'}$$

$$\frac{e \triangleright^{*} e'}{e \equiv e'} \qquad \frac{e \equiv e'}{\lambda x.\, e \equiv \lambda x.\, e'}$$

$$\frac{e_1 \equiv {e_1}' \quad e_2 \equiv {e_2}'}{e_1\ e_2 \equiv {e_1}'\ {e_2}'} \qquad \frac{e_1 \equiv {e_1}' \quad e_2 \equiv {e_2}' \quad e_3 \equiv {e_3}'}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \equiv \texttt{if } {e_1}' \texttt{ then } {e_2}' \texttt{ else } {e_3}'}$$

$$\frac{e \equiv e'}{\mathcal{S}k.\, e \equiv \mathcal{S}k.\, e'} \qquad \frac{e \equiv e'}{\langle e \rangle \equiv \langle e' \rangle}$$

**Figure 4.** $\lambda_{\mathcal{S}}$ Equivalence

$$\frac{\vdash \Gamma}{\Gamma \vdash * : \Box \rightsquigarrow *}\ (\textsc{Star}) \qquad \frac{C \text{ is a base type}}{\Gamma \vdash C : * \rightsquigarrow C}\ (\textsc{Base})$$

$$\frac{\Gamma \vdash A : * \rightsquigarrow A' \quad \Gamma,\, x : A \vdash B : * \rightsquigarrow B'}{\Gamma \vdash \Pi x :^{\iota} A.\, B : * \rightsquigarrow \Pi x :^{\iota} A'.\, B'}\ (\textsc{PiPure}) \qquad \frac{\Gamma \vdash A : * \rightsquigarrow A' \quad \Gamma,\, x : A \vdash B : * \rightsquigarrow B' \quad \Gamma \vdash \alpha : * \rightsquigarrow \alpha'}{\Gamma \vdash \Pi x :^{\alpha} A.\, B : * \rightsquigarrow \Pi x :^{\alpha'} A'.\, B'}\ (\textsc{PiEff})$$

$$\frac{\Gamma \vdash e : \texttt{bool} \,!\, \iota \rightsquigarrow e' \quad \Gamma \vdash A : * \rightsquigarrow A' \quad \Gamma \vdash B : * \rightsquigarrow B'}{\Gamma \vdash \texttt{if } e \texttt{ then } A \texttt{ else } B : * \rightsquigarrow \texttt{if } e' \texttt{ then } A' \texttt{ else } B'}\ (\textsc{If})$$

**Figure 5.** $\lambda_{\mathcal{S}}$ Kinding and CPS Translation of Types

$\epsilon ::= \ldots \mid \texttt{if } e \texttt{ then } \epsilon_1 \texttt{ else } \epsilon_2$ Effects

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \,!\, \iota \quad \Gamma \vdash e_2 : A_2 \,!\, \epsilon_2 \quad \Gamma \vdash e_3 : A_3 \,!\, \epsilon_3}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \texttt{if } e_1 \texttt{ then } A_2 \texttt{ else } A_3 \,!\, \texttt{if } e_1 \texttt{ then } \epsilon_2 \texttt{ else } \epsilon_3}\ (\textsc{DIf})$$

**Figure 6.** Syntax and Typing of Effect-Level If