# An Intrinsically Typed Compiler for Algebraic Effect Handlers

Syouki Tsuyama
Tokyo Institute of Technology
Tokyo, Japan
s-tsuyama@prg.is.titech.ac.jp

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

Hidehiko Masuhara
Tokyo Institute of Technology
Tokyo, Japan
masuhara@acm.org

## Abstract

A type-preserving compiler converts a well-typed input program into a well-typed output program. Previous studies have developed type-preserving compilers for various source languages, including the simply-typed lambda calculus and calculi with control constructs. Our goal is to realize type-preserving compilation of languages that have facilities for manipulating first-class continuations. In this paper, we focus on algebraic effects and handlers, a generalization of exceptions and their handlers with resumable continuations. Specifically, we choose an effect handler calculus and a typed stack-machine-based assembly language as the source and the target languages, respectively, and formalize the target language and a type preserving compiler. The main challenge posed by first-class continuation is how to ensure safety of continuation capture and resumption, which involves concatenation of unknown stacks. We solve this challenge by incorporating stack polymorphism, a technique that has been used for compilation from a language without first-class continuations to a stack-based assembly language. To prove that our compiler is type preserving, we implemented the compiler in Agda as a function between intrinsically typed ASTs. We believe that our contributions could lead to correct and efficient compilation of continuation-manipulating facilities in general.

***CCS Concepts:*** • **Theory of computation** → **Control primitives**; **Logic and verification**; *Type theory*; • **Software and its engineering** → **Compilers**.

***Keywords:*** type-preserving compilers, algebraic effect handlers, dependent types


*PEPM '24, January 16, 2024, London, UK*

ACM ISBN 979-8-4007-0487-1/24/01
https://doi.org/10.1145/3635800.3636968

**ACM Reference Format:**
Syouki Tsuyama, Youyou Cong, and Hidehiko Masuhara. 2024. An Intrinsically Typed Compiler for Algebraic Effect Handlers. In *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation (PEPM '24), January 16, 2024, London, UK*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3635800.3636968

## 1 Introduction

Type-preserving compilers convert a well-typed source program into well-typed target code. The type preservation property is useful for ensuring correctness of the compiler, in that any invariants encoded by source types cannot get lost during compilation [3]. Type preservation is also important for improving efficiency of generated code, because the type information helps us find opportunities for optimization [14].

Previous studies have developed type-preserving compilers for various source languages, including the simply-typed lambda calculus [6, 16], System F [8, 14], an imperative language with conditionals and loops [19], and an exception calculus [16]. One feature that is missing in these languages is facilities for manipulating first-class continuations, which have recently introduced into several practical languages [10,22].

Our long-term goal is to realize type-preserving compilation of continuation-manipulating facilities. As a first step towards this goal, we consider type-preserving compilation of algebraic effects and handlers [18]. Effect handlers are a generalization of exception handlers that allow the programmer to express a wide range of computational effects using continuations. In this paper, we formalize a typed target language based on a stack machine, as well as a type-preserving compiler from an effect handler calculus to the target language.

The main challenge posed by first-class continuations is how to ensure safety of continuation capture and resumption. Specifically, in our target language, the type of a captured continuation must refer to the type of the stack used for resumption, which is unknown at capture time.

To solve the above problem, we use a technique called *stack polymorphism*, proposed by Morrisett et al. [13]. The technique was originally uesd for compiling recursive functions into a stack-based typed assembly language. The idea is to abstract over stack types to allow flexibility in the shape

$$
\begin{array}{rrcl}
\text{Value types} & A, B & ::= & Unit \mid A \to C \\
\text{Computation types} & C, D & ::= & A!E \\
\text{Handler types} & F & ::= & C \Rightarrow D \\
\text{Effect signatures} & S & ::= & l : A \to B \\
\text{Effects} & E & ::= & \emptyset \mid \{S\} \uplus E \\
\text{Values} & V, W & ::= & unit \mid x \mid \lambda x.M \\
\text{Computations} & M, N & ::= & V\ W \mid return\ V \mid \\
 & & & do\ l\ V \mid \\
 & & & let\ x = M\ in\ N \mid \\
 & & & handle\ M\ with\ H \\
\text{Handlers} & H & ::= & \{return\ x \to M\} \mid \\
 & & & \{l\ p\ k \to M\} \uplus H
\end{array}
$$

**Figure 1.** Syntax of $L_S$

of the stack at the point of continuation resumption. This is the key novelty of our work: although stack polymorphism itself is not a new technique, it has never been used in the context of compiling continuations. We believe that our finding would lead to safer implementations of first-class continuations.

To prove that our compiler preserves types, we follow the approach of *intrinsically typed compilers* [6, 8, 16, 19]. Specifically, we formalize the source and target languages as intrinsically typed ASTs in Agda [15], and then define the compiler as a function between these ASTs. By implementing the compiler in this way, we automatically obtain the type preservation proof.

In this paper, we provide the following.

- A typed target language that is powerful enough to express manipulation of first-class continuations.
- A proof of type preservation of the compiler in the form of an Agda function between intrinsically typed ASTs.

The rest of the paper is organized as follows. In Sections 2, we define the intrinsically typed ASTs and semantics of the source language. In Sections 3, we motivate the use of stack polymorphism and define the target language. In Section 4, we present a compiler from the source to the target. In Section 5, we compare our work to existing work, and in Section 6, we conclude with a discussion of future directions.

The source code of our formalization is available at: https://github.com/prg-titech/Effect-Handler-Compiler.

## 2 Source Language $L_S$

We formalize the source language $L_S$ of our compiler. The language is an extension of the lambda calculus with deep effect handlers, modeled after the effect handler calculus of Hillerström et al. [9]. The difference is that we do not include polymorphic types, nor do we forward unhandled operations (i.e., we require handlers to include one clause for every operation). This helps us simplify the language specification and highlight the challenges with first-class continuations, which is the main focus of this paper. Below, we first present the specification of $L_S$ using mathematical notations, and then in Agda.

### 2.1 Mathematical Specification

The source language is a fine-grain call-by-value calculus [11], where types and terms are classified into three categories: values, computations, and handlers.

***Syntax.*** We define the syntax of $L_S$ in Figure 1. At the level of types, we have value types, computation types, and handler types, which are mutually defined with effects. Value types consist of the unit type ($Unit$) and function types ($A \to C$). Computation types $A!E$ are pairs of a value type $A$ and an effect $E$, which respectively represent the result of the computation and the effect that may be performed during the computation. Effects are a set of effect signatures $S$, and signatures are operation types $l : A \to B$, where $l$ is a label, $A$ is input type, and $B$ is an output type. A handler type $C \Rightarrow D$ represents handlers that handle a computation of type $C$ and return a computation of type $D$.

At the level of terms, we again have three categories. Values consist of unit ($unit$), variables ($x$), and functions ($\lambda x.M$). Computations are function applications ($V\ W$), return expressions ($return\ V$), operation calls ($do\ l\ V$), let bindings ($let\ x = M\ in\ N$), and effect handling constructs ($handle\ M\ with\ H$). Handlers include a return clause and zero or more operation clauses, specifying what to do when the handled computation returns a value and performs an operation. The two arguments $p$ and $k$ represent the parameter and continuation of the operation.

***Typing Rules.*** We define the typing rules of $L_S$ in Figure 2. The typing judgments for values, computations, and handlers take the form $\Gamma \vdash V : A$, $\Gamma \vdash M : C$, and $\Gamma \vdash H : F$, respectively. These judgments are understood in the standard way. For instance, $\Gamma \vdash V : A$ means "value $V$ has type $A$ under context $\Gamma$." Note that a typing context $\Gamma$ is a sequence of pairs of a variable and a value type.

To go through the rules for effect constructs, rule T-Do introduces an effect $l : A \to B$ in the conclusion. Rule T-Handle turns the type $C$ of the handled computation into a different type $D$ that comes from the handler clauses. Rule T-Handler requires that the return and operation clauses all have the same type.

***Operational Semantics.*** We define a small-step operational semantics of $L_S$ in Figure 3. The reduction rules are all standard. Rules E-APP and E-LET enforce the call-by-value evaluation strategy. Rule E-HANDLE-RET processes the value returned from a handled computation using the return clause $H^{ret}$ of the surrounding handler. Rule E-HANDLE-OP processes the operation performed by a handled computation using the matching operation clause $H^{l}$. Handlers in $L_S$

$\boxed{\Gamma \vdash V : A}$ $\quad \Gamma \vdash \mathit{unit} : \mathit{Unit}$ (T-Unit) $\quad \dfrac{x : A \in \Gamma}{\Gamma \vdash x : A}$ (T-Var) $\quad \dfrac{\Gamma, x : A \vdash M : C}{\Gamma \vdash \lambda x.M : A \to C}$ (T-Abs)

$\boxed{\Gamma \vdash M : C}$

$$\dfrac{\Gamma \vdash V : A \to C \qquad \Gamma \vdash W : A}{\Gamma \vdash V\ W : C}\ \text{(T-App)} \qquad \dfrac{(l : A \to B) \in E \qquad \Gamma \vdash V : A}{\Gamma \vdash\ \mathit{do}\ l\ V : B!E}\ \text{(T-Do)}$$

$$\dfrac{\Gamma \vdash V : A}{\Gamma \vdash\ \mathit{return}\ V : A!E}\ \text{(T-Return)} \qquad \dfrac{\Gamma \vdash M : A!E \qquad \Gamma, x : A \vdash N : B!E}{\Gamma \vdash \mathit{let}\ x = M\ \mathit{in}\ N : B!E}\ \text{(T-Let)}$$

$$\dfrac{\Gamma \vdash M : C \qquad \Gamma \vdash H : C \Rightarrow D}{\Gamma \vdash \mathit{handle}\ M\ \mathit{with}\ H : D}\ \text{(T-Handle)}$$

$\boxed{\Gamma \vdash H : F}$

$$\dfrac{C = A!\{\ l : A_l \to B_l\}_{l \in E} \qquad \Gamma, x : A \vdash M : D \qquad \forall l \in E.\, \Gamma, (p : A_l), (k : B_l \to D) \vdash N_l : D}{\Gamma \vdash \{\mathit{return}\ x \to M\} \uplus \{l\ p\ k \to N_l\}_{l \in E} : C \Rightarrow D}\ \text{(T-Handler)}$$

**Figure 2.** Typing Rules of $L_S$

| | | | |
|---|---|---|---|
| E-APP | $(\lambda x.M)V$ | $\longrightarrow$ | $M[V/x]$ |
| E-LET | $\mathit{let}\ x = \mathit{return}\ V\ \mathit{in}\ N$ | $\longrightarrow$ | $N[V/x]$ |
| E-HANDLE-RET | $\mathit{handle}\ (\mathit{return}\ V)\ \mathit{with}\ H$ | $\longrightarrow$ | $M[V/x]$ where $H^{ret} = \{\mathit{return}\ x \to M\}$ |
| E-HANDLE-OP | $\mathit{handle}\ E[\mathit{do}\ l\ V]\ \mathit{with}\ H$ | $\longrightarrow$ | $M[V/p, (\lambda y.\mathit{handle}\ E[\mathit{return}\ y]\ \mathit{with}\ H)/k]$ where $H^l = \{l\ p\ k \to M\}$ |
| E-Lift | $E[M]$ | $\longrightarrow$ | $E[N]\ \mathit{if}\ M \longrightarrow N$ |
| Evaluation Contexts | $E$ | $::=$ | $[]\ \vert\ \mathit{let}\ x = E\ \mathit{in}\ N$ |

**Figure 3.** Operational Semantics of $L_S$

```
data Sig where
  op : VTy → VTy → Sig

Eff = List Sig

data VTy where
  Unit  : VTy
  _⇒_ : VTy → CTy →
        VTy

CTy = VTy × Eff

data HTy where
  _⟹_ : CTy → CTy →
           HTy

Ctx = List VTy

variable
  A B A' B' : VTy
  E E' E₁ E₂ : Eff
  C D : CTy
  H : HTy
  Γ Γ' Γ₁ Γ₂ : Ctx
```

**Figure 4.** Agda Definition of $L_S$ Types

are deep: captured continuations are automatically handled by the same handler.

### 2.2 Agda Representation

We now present a formalization of $L_S$ as intrinsically typed ASTs in Agda. The ASTs encode both the syntax and typing rules from Section 2.1, meaning that they can only express well-typed terms in $L_S$.

First, we define data types representing types and effects (Figure 4). We name the three kinds of types VTy, CTy, and HTy, and define one constructor for each syntactic form given in Figure 1. For computation types CTy, we use Agda's product type _×_, and for typing contexts, we use Agda's list type constructor List. To keep the formalization concise, we declare variables of each data type using Agda's variable keyword.

Next, we define data types representing terms (Figure 5). We name the three classes of terms Val, Cmp, and Hdl, and define them as parameterized and indexed data types[1]. The parameters and indices allow us to hard-code the typing rules of the language into the definition of constructors. For example, the signature of the App constructor encodes the typing rule T-App in Figure 2. In particular, the two arguments of the App constructor are indexed by types $A \Rightarrow B$ and $A$, respectively, preventing us from constructing an ill-typed application. Note that we use de Bruijn indices to represent variables.

[1]In Agda, parameters of data types appear to the left of the colon, while indices appear to the right of the colon. Their difference is that the former must be the same for all constructors, while the latter may be different across constructors.

```
data Val (Γ : Ctx) : VTy → Set
data Cmp (Γ : Ctx) : CTy → Set
data Hdl (Γ : Ctx) : HTy → Set
OperationClauses : Ctx → Eff → CTy → Set

data Val Γ where
  Unit : Val Γ Unit
  Var : A ∈ Γ → Val Γ A
  Lam : Cmp (A :: Γ) C → Val Γ (A ⇒ C)

data Cmp Γ where
  Return : Val Γ A → Cmp Γ (A , E)
  Do : (op A B) ∈ E → Val Γ A → Cmp Γ (B , E)
  Handle_With_ : Cmp Γ C → Hdl Γ (C ⟹ D) →
                 Cmp Γ D
  App : Val Γ (A ⇒ C) → Val Γ A → Cmp Γ C
  Let_In_ : Cmp Γ (A , E) → (Cmp (A :: Γ) (B , E))
                          → Cmp Γ (B , E)

data Hdl Γ where
  λx_|λx,r_ :
    Cmp (A :: Γ) C → -- return clause
    OperationClauses Γ E C → -- operation clauses
    Hdl Γ ((A , E) ⟹ C)

OperationClauses Γ E₁ D =
  All (λ { (op A' B') → Cmp ((B' ⇒ D) :: A' :: Γ) D }) E₁
```

**Figure 5.** Intrinsically Typed ASTs of $L_S$ Terms

As an auxiliary data type, we define OperationClauses, which represent operation clauses of a handler. Here, the All keyword is a type-level map function provided by Agda.

## 3 Target Language $L_T$

We formalize the target language $L_T$. The target language $L_T$ is a stack-based abstract machine, built by combining and extending the target languages for existing intrinsically typed compilers [2, 16]. In $L_T$, any effect-related computation is realized by manipulation of the stack and environment. Concretely, installing a handler involves pushing a handler value onto the stack, performing an operation involves storing the parameter and continuation in the runtime environment, and resuming a continuation involves concatenation of stacks. Below, we first give an overview of the language and then detail individual language components.

### 3.1 Overview

**Execution of Effectful Programs.** To help the reader understand the design of the target language, we demonstrate how we execute effectful programs in the target language. Consider the following source program, which performs an operation *op* within a handler whose operation clause resumes the continuation *k* with *unit*.

```
handle ( let x = do op unit in return x )
with {
  return x → return x;
  op p k → k unit ;
}
```

The execution of the above program takes four steps. Each step involves manipulation of the stack in the target language, as shown in Figure 6.

1. The compiled handler and its meta-continuation *mk* (which simply returns the given value) are pushed onto the stack.
2. When the operation *op* is performed, the operation clause of the handler on the stack is executed. This involves capturing the continuation of *op* and binding it to *k*.
3. The continuation *k* is resumed with a stack that is extended with two things. One is the handler *hand* included in the continuation *k*. The other is the *RET* instruction, which represents the end of operation clause and executes the meta-continuation *mk*.
4. The return clause of the handler is executed with the paraeter *unit*, which is on the top of the stack.

Next, we will informally introduce the components of the target language that are necessary for realizing such computation.

**Components of Language.** The main component of the target language is instructions, which we represent as the Code data type.

```
data Code (Γ : Ctx) : StackTy → StackTy → Set
```

An instruction of type Code $\Gamma$ $S$ $S'$ requires that it requires a runtime environment whose shape is $\Gamma$ and a stack whose shape is $S$, and it produces a stack whose shape is $S'$.

The semantics of $L_T$ instructions is given as the exec function.

```
exec : Code Γ S S' → Stack S → RuntimeEnv Γ → Stack S'
```

The function executes executes an instruction with a stack and a runtime environment of the required shape, and produces a stack of the expected shape.

Executing an instruction involves storing values in the runtime environment and stack.

```
RuntimeEnv Γ = All (λ A → EnvVal A) Γ

Stack S = All (λ T → StackVal T) S
```

A runtime environment is a sequence of values to be substituted for source variables. A stack is a sequence of values to be used later in the computation.

Lastly, the target language has a special data type PureCodeCont for captured continuations.

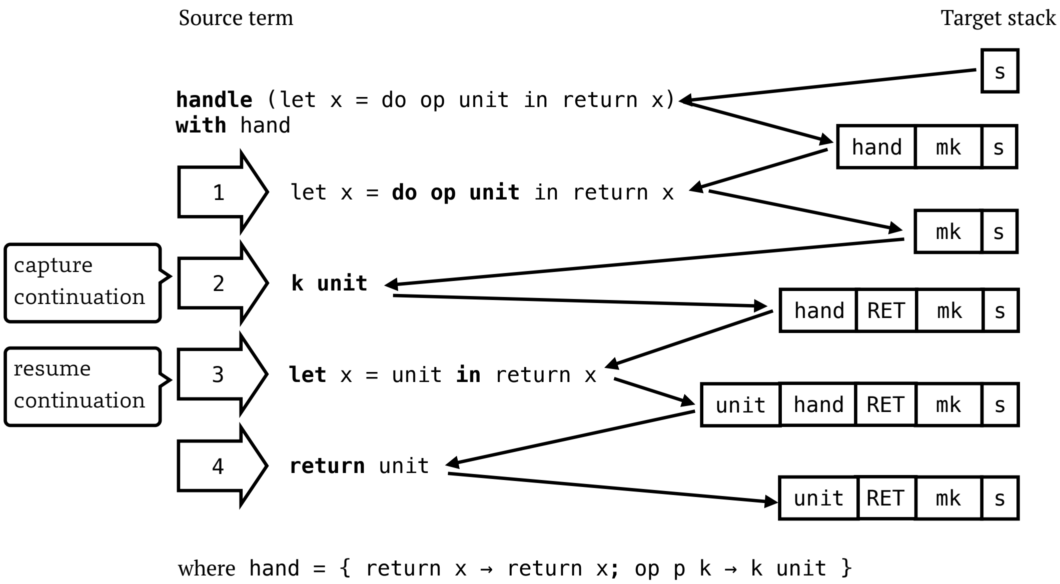


**Figure 6.** Execution of an Effectful Program in the Source and Target Languages

PureCodeCont : Ctx ⟶ StackTy ⟶ CTy ⟶ Set

A captured continuation of type PureCodeCont $\Gamma$ $S$ $C$ is an instruction whose input stack consists of an $S$-typed stack and $C$-accepting handler. The typing of this handler is the main challenge of this work, and it is detailed in Sections 3.4 and 3.6.

In what follows, we provide more details of each component. For conciseness, we will only show the Agda representation of the definitions.

### 3.2 Instructions

We define instructions as the Code data type in Figure 7. Among the three arguments of Code, the first one (of type Ctx) represents the shape of a runtime environment, and the other two (of type StackTy) represent the shapes of the stack before and after the execution of an instruction.

The instructions all have a counterpart in the source language. PUSH and ABS push a unit value and a lambda abstraction respectively. LOOKUP pushes a value referred to by an $L_S$ variable. APP, CALLOP, and BIND correspond to a function application, an operation call, and a let binding. RET unifies the returning behavior after a function application and effect handling. MARK installs a handler, while UNMARK removes a handler upon execution of the return clause. Lastly, INITHAND pushes a trivial handler for a top-level computation that has no effects[2]. Note that some of them receive a *code continuation* as the last argument, representing what to do after execution of an instruction[3]. The type of code continuations is either Code or PureCodeCont; we will explain their difference in Section 3.6.

### 3.3 Environments and Stacks

We define runtime environments in Figure 8. A runtime environment has type of the form RuntimeEnv $\Gamma$, where $\Gamma$ contains the types of variables to be replaced by values.

Environment values represent the results of evaluating an $L_S$ computation. Their type EnvVal $A$ is indexed by a value type $A$ in $L_S$. The four constructors of EnvVal build a plain value (pval), a closure (clos), a first-class continuation (fc-resump) respectively. A closure in $L_T$, which we call *code closure*, holds the code of the function body and the runtime environment in which the body is executed. The function body requires that the continuation of the caller is on the top of the stack so that control can be returned after execution of the body. A first-class continuation consists of the code of the continuation body (of type PureCodeCont, to be defined in Section 3.4), its runtime environment and handler, plus the stack used by the continuation.

In Figure 9, we provide an example of a runtime environment. It has two elements: a plain value and a code closure. Correspondingly, the type of the runtime environment has a plain value type and a function type.

We next define stacks in Figure 10. A stack has type takes the form Stack $S$, where $S$ is a list of types (of type SValTy)

[2]We need the top-level handler because we require every compiled program to have a handler on the stack. This eliminates the need for distinguishing between computations that require a handler and those that do not.

[3]The arguments in curly braces are implicit, meaning that they are meant to be automatically inferred by Agda.

```
data Code Γ where
  PUSH : PVal A ⟶ Code Γ (ValTy A :: S) S' ⟶ Code Γ S S'
  ABS :
    -- function body
    (∀{S₁ S₂ S₃ Γ₁ Γ'₁ A'} ⟶ Code (A :: Γ) (ContTy Γ₁ (ValTy B :: S₁) (A' , E₁) :: (S₁ ++ HandTy Γ'₁ S₂ S₃ (A' , E₁) :: S₂)) S₃) ⟶
    Code Γ (ValTy (A ⇒ (B , E₁)) :: S) S' ⟶ Code Γ S S'
  LOOKUP : A ∈ Γ ⟶ Code Γ (ValTy A :: S) S' ⟶ Code Γ S S'
  APP : PureCodeCont Γ (ValTy B :: S₁) (A' , E) ⟶
          Code Γ (ValTy A :: ValTy (A ⇒ (B , E)) :: S₁ ++ HandTy Γ₁ S₂ S₃ (A' , E) :: S₂) S₃
  CALLOP : (op A B) ∈ E
              ⟶ PureCodeCont Γ (ValTy B :: S₁) (A' , E)
              ⟶ Code Γ (ValTy A :: S₁ ++ HandTy Γ₁ S S' (A' , E) :: S) S'
  BIND : Code (A :: Γ) (ContTy Γ (ValTy B :: S) C :: (S ++ HandTy Γ₂ S₂ S₃ C :: S₂)) S₃ -- let body
          ⟶ PureCodeCont Γ (ValTy B :: S) C ⟶ Code Γ (ValTy A :: (S ++ HandTy Γ₂ S₂ S₃ C :: S₂)) S₃
  RET : Code Γ (ValTy A :: ContTy Γ₁ (ValTy A :: S) C :: (S ++ HandTy Γ₂ S₂ S₃ C :: S₂)) S₃
  MARK :
    HandlerCode Γ (A , E₁) (B , E₂) ⟶ -- handler
    PureCodeCont Γ (ValTy B :: S₁) (B' , E₂) ⟶ -- meta-continuation
    -- handled computation
    Code Γ (
      HandTy Γ (ContTy Γ (ValTy B :: S₁) (B' , E₂) :: S₁ ++ HandTy Γ₁ S₂ S₃ (B' , E₂) :: S₂) S₃ (A , E₁) ::
                ContTy Γ (ValTy B :: S₁) (B' , E₂) :: S₁ ++ HandTy Γ₁ S₂ S₃ (B' , E₂) :: S₂
    ) S₃ ⟶
    Code Γ (S₁ ++ HandTy Γ₁ S₂ S₃ (B' , E₂) :: S₂) S₃
  UNMARK : Code Γ (ValTy A :: HandTy Γ₁ S S' (A , E₁) :: S) S'
  INITHAND : Code Γ (HandTy Γ S (ValTy A :: S) (A , []) :: S) (ValTy A :: S) ⟶ Code Γ S (ValTy A :: S)
```

**Figure 7.** Intrinsically-Typed ASTs of Instructions

of stack values to be used later. A stack value (Figure 11) is either the result of a computation (of type ValTy), a continuation (of type ContTy), a handler (of type HandTy), or a top-level handler. These values are constructed using val, cont, hand, init-hand of type SValTy. Continuations built with cont are not first-class continuations captured during execution; they are instructions to be executed after a RET instruction. Handlers built with hand include its clauses and environment. As expressed in its type, handler clauses require that the stack has the continuation of the handler (i.e., the meta-continuation) specifying what to do after effect handling. The init-hand constructor takes no argument because the top-level handler has the fixed behavior: it returns the result of the computation as is.

### 3.4 Pure Code Continuations

A pure code continuation is a continuation up to the nearest handler. It differs from an ordinary code continuation in that it assumes the presence of a handler on the stack. This assumption is reflected in the type of pure continuations, defined as PureCodeCont below.

```
PureCodeCont : Ctx ⟶ StackTy ⟶ CTy ⟶ Set
PureCodeCont Γ S₁ C =
  ∀{Γ₁ S₂ S₃} ⟶ Code Γ (S₁ ++ HandTy Γ₁ S₂ S₃ C :: S₂) S₃
```

We see that the type is a Code type whose first stack index contains a handler. What is important here is that the types $S_2$ and $S_3$, which represent the stacks before and after executing the handler of the continuation itself, are universally quantified.

### 3.5 Semantics

Having defined the syntax and typing of $L_T$, we define the semantics in the form of the exec function (Figure 12)[4]. As the signature says, exec executes an instruction with a stack

[4]We use {-# TERMINATING #-} pragma for interpreter functions because execution of instructions and handler bodies involves non-structural recursion. We believe that these functions are in fact terminating, as our source language does not have recursive functions or loops. To convince Agda, however, we would need to add a termination measure, which would complicate the overall development. We also plan to develop a version of our compiler that does not use the pragma as future work.

```
data EnvVal : VTy → Set
RuntimeEnv : Ctx → Set

data EnvVal where
  pval : PVal A → EnvVal A
  clos : (∀{Γ₁ Γ'₁ S₁ S₂ S₃ A'} → Code (A :: Γ) (ContTy Γ₁ (ValTy B :: S₁) (A' , E) :: (S₁ ++ HandTy Γ'₁ S₂ S₃ (A' , E) :: S₂)) S₃)
           → RuntimeEnv Γ → EnvVal (A ⇒ (B , E))
  fc-resump :
    PureCodeCont Γ (ValTy A :: S) (A' , E) × Stack S × RuntimeEnv Γ → -- continuation body and its stores
    HandlerCode Γ₁ (A' , E) (B , E') × RuntimeEnv Γ₁ → -- handler and its environment
    EnvVal (A ⇒ (B , E'))

RuntimeEnv Γ = All (λ A → EnvVal A) Γ

HandlerCode Γ (A , E₁) (B , E₂) =
  (∀{Γ₁ Γ'₁ S₁ S₂ S₃ A'} →
    -- return clause
    Code (A :: Γ) (ContTy Γ'₁ (ValTy B :: S₁) (A' , E₂) :: (S₁ ++ HandTy Γ₁ S₂ S₃ (A' , E₂) :: S₂)) S₃ ×
    -- operation clauses
    OperationCodes B E₁ E₂ Γ (ContTy Γ'₁ (ValTy B :: S₁) (A' , E₂) :: (S₁ ++ HandTy Γ₁ S₂ S₃ (A' , E₂) :: S₂)) S₃
  )

OperationCodes B E₁ E₂ Γ SS S₃ = All (λ {(op A' B') → Code ((B' ⇒ (B , E₂)) :: A' :: Γ) SS S₃ }) E₁
```

**Figure 8.** Definition of Runtime Environments

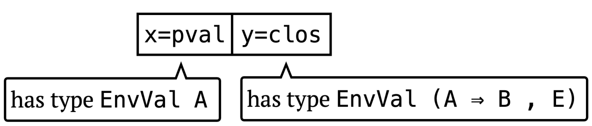


**Figure 9.** Example of Runtime Environment

```
data SValTy : Set
StackTy : Set

data SValTy where
  ValTy : VTy → SValTy
  ContTy : Ctx → StackTy → CTy → SValTy
  HandTy : Ctx → StackTy → StackTy → CTy →
           SValTy

StackTy = List SValTy

variable
  T : SValTy
  S S' S₁ S₂ S₃ : StackTy


data StackVal : SValTy → Set

Stack : StackTy → Set
Stack S = All (λ T → StackVal T) S
```

**Figure 10.** Definition of Stacks

and a runtime environment of the required shape, and produces a stack of the expected shape. This signature can be read as the type soundness property of $L_T$. By the Curry-Howard isomorphism, the definition of exec serves as the type soundness proof.

Let us begin with the cases for instructions that push values. PUSH pushes a plain value onto the stack, whereas ABS pushes a code closure. LOOKUP reads the value referred to by the argument variable $x$ and pushes it.

We next look at instructions for function applications, operation calls, and let bindings. APP behaves differently depending on whether the second element of the stack is a code closure (constructed by clos) or a continuation (constructed by fc-resump). In the former case, the instruction extends the environment with the argument value $v$ and executes the code closure. In the latter case, the instruction adds $v$ to the environment, builds a new stack, and executes the continuation body $c'$. The stack built here is basically a concatenation (denoted by _++s_) of the stack $s'$ of the captured continuation and the stack $s$ of the caller, but it additionally contains a handler, enforcing the deep handler semantics. CALLOP finds and executes the code of an operation clause

```
data StackVal where
  val : EnvVal A ⟶ StackVal (ValTy A)
  cont : PureCodeCont Γ S₁ (A , E) ⟶ RuntimeEnv Γ ⟶ StackVal (ContTy Γ S₁ (A , E))
  hand : HandlerCode Γ (A , E₁) (B , E₂) ⟶ RuntimeEnv Γ ⟶
         StackVal (HandTy Γ (ContTy Γ₂ (ValTy B :: S₁) (A' , E₂) :: S₁ ++ HandTy Γ₁ S₂ S₃ (A' , E₂) :: S₂) S₃ (A , E₁))
  init-hand : StackVal (HandTy Γ S (ValTy A :: S) (A , []))
```

**Figure 11.** Definition of Stack Values

```
{-# TERMINATING #-}
exec : Code Γ S S' ⟶ Stack S ⟶ RuntimeEnv Γ ⟶ Stack S'
exec (PUSH v c) s = exec c $ (val (pval v)) :: s
exec (ABS c' c) s env = exec c (val (clos c' env) :: s) env
exec (LOOKUP x c) s env = exec c ((val $ lookup env x) :: s) env
exec (APP c) (val v :: val (clos c' env') :: s) env = exec c' (cont c env :: s) (v :: env')
exec (APP c) (v :: val (fc-resump (c' , s' , env₂) (h , envh)) :: s) env =
  exec c' (v :: (s' ++s (hand h envh :: cont c env :: s))) env₂
exec (CALLOP l c) (val v :: s) env with split s
... | (s1 , (hand h env') , s2) with h
... | (_ , ops) = exec (lookup ops l) s2 (fc-resump (c , s1 , env) (h , env') :: v :: env')
exec (BIND c k) (val v :: s) env = exec c (cont k env :: s) (v :: env)
exec RET (val v :: cont c env :: s) _ = exec c (val v :: s) env
exec (MARK h mk c) s env = exec c (hand h env :: cont mk env :: s) env
exec (UNMARK) (val x :: (hand h env') :: s) env with h
... | (ret , ops) = exec ret s (x :: env')
exec (UNMARK) (val x :: init-hand :: s) env = val x :: s
exec (INITHAND c) s env = exec c (init-hand :: s) env

split : Stack (S₁ ++ HandTy Γ₁ S S' (A , E) :: S) ⟶ Stack S₁ × StackVal (HandTy Γ₁ S S' (A , E)) × Stack S
split {S₁ = []} (H :: S) = ([] , H , S)
split {S₁ = _ :: _} (V :: S) with split S
... | (S1 , H , S2) = (V :: S1 , H , S2)
```

**Figure 12.** Definition of Execution Function

using the label $l$, while adding the argument value to the stack and the captured code continuation to the runtime environment. Observe that the stack is split into two ($s_1$ and $s_2$) by the split function. The two portions contain elements that appear before and after the handler, and are used for execution of the continuation and meta-continuation. BIND adds to the runtime environment the value on the top of the stack and shifts control to the body of let.

We are now left with instructions for effect handling. MARK pushes a handler $h$ and its continuation $mk$, and starts execution of a computation $c$. UNMARK behaves differently depending on whether the handler on the stack is a regular handler or a top-level handler init-hand. In the former case, the instruction pushes the meta-continuation of the handler and executes the return clause. The meta-continuation is resumed by RET when the execution of the return clause finishes. In the latter case, the instruction simply pushes the value $v$. Lastly, INITHAND pushes init-hand onto the stack, allowing us to start execution of a top-level computation.

### 3.6 Stack Polymorphism

As we saw earlier, in the type PureCodeCont of pure code continuations, we universally quantify the type of the stacks before and after executing the continuation of the handler. The need for this quantification comes from the fact that, when capturing a continuation, we do not know the stack at the point where the continuation is resumed.

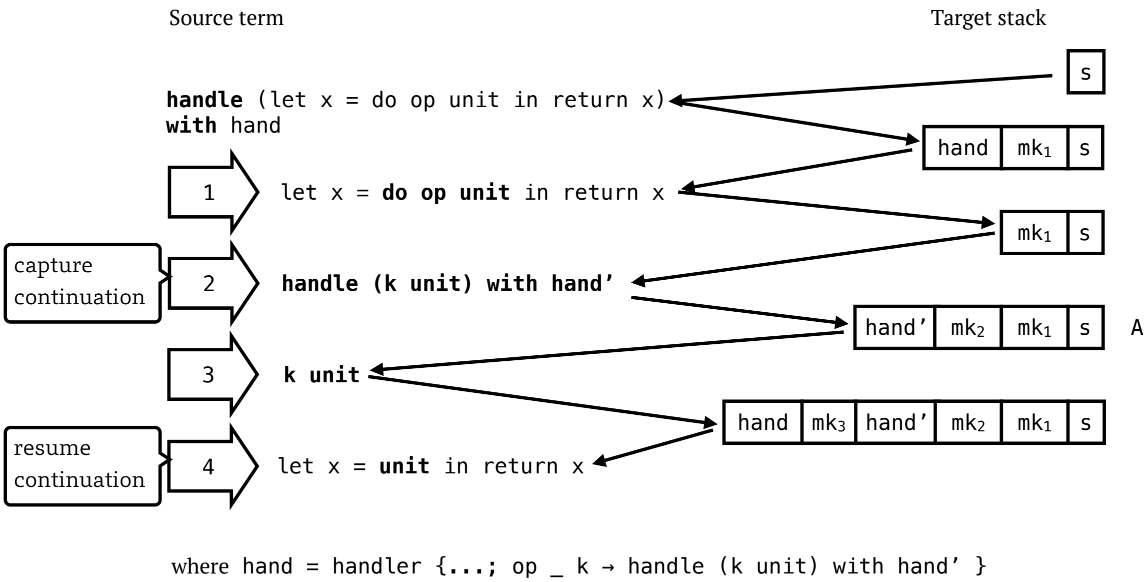


**Figure 13.** Example of Continuation Capture and Resumption

To understand the challenge, let us consider the example in Figure 13. The source computation performs an operation *op* within a handler *hand* whose operation clause resumes the captured continuation $k$ within a handler *hand′*. The execution of this computation goes as follows.

1. The handler *hand* is installed. This is done by pushing handler and its meta continuation $mk_1$ onto the stack.
2. The handled computation is executed. This involves a call to the operation *op*, which captures the continuation and binds it to $k$.
3. The operation clause of the handler hand is executed. This involves pushing the handler *hand′* and its meta continuation $mk_2$ onto the stack.
4. The continuation $k$ is resumed with a stack that is extended with the handler *hand* stored in the continuation.

With this in mind, we consider the type of the handler *hand*, which appears in the type of the captured continuation. When the captured continuation is resumed, the stack has additional elements *hand′* and $mk_2$ as in stack A. These elements appear in the type of the captured continuation. However, when the continuation is captured, we do not know what will be in the stack at the point of resumption.

The above problem can be solved by universally quantifying over the stack types in PureCodeCont. This way, we can be abstract about the stack types when we capture a continuation, and instantiate them appropriately when we resume the continuation.

Note that the HandlerCode type in Figure 11 also requires stack polymorphism (observe the universal quantification on $S_2$ and $S_3$). The reason is similar to that for PureCodeCont: when we compile a handler, we do not know what the stack looks like when it is used, hence we need to be abstract about the stack types.

## 4 Intrinsically Typed Compiler

We formalize an intrinsically typed compiler from $L_S$ to $L_T$. We do this by defining compiling functions for top-level computations, values, and general computations.

We first define compile (Figure 14) that takes care of top-level computations. The function produces an instruction that pushes the top-level handler init-hand, performs the source computation, and returns the resulting value via UNMARK. Notice that the signature of compile tells us that the function is type-preserving: if the source computation has type $A$, then the output instruction pushes onto the stack a value of ValTy $A$ where ValTy is a translation from source types to target types.

We next define compileV and compileC, which respectively compile values and computations. These functions receive a code continuation to be executed after execution of the output instruction. Again, the signatures can be read as type preservation statements.

Compilation of values produces an instruction that pushes the values of themselves onto the stack. For lambda abstractions, compileV produces an ABS instruction that pushes a code closure. In the compilation of the function body, the RET instruction is used as the code continuation because control must return to the caller after the function application.

```
compile : Cmp Γ (A , []) → Code Γ S (ValTy A :: S)
compile c = INITHAND (compileC {S₁ = []} c UNMARK)

compileV : Val Γ A → Code Γ (ValTy A :: S) S' → Code Γ S S'
compileC : Cmp Γ (A , E) → PureCodeCont Γ (ValTy A :: S₁) (A' , E) → Code Γ (S₁ ++ HandTy Γ₁ S S' (A' , E) :: S) S'
-- auxiliary function for compiling handlers
compileH : Hdl Γ (C ⟹ D) → HandlerCode Γ C D
-- auxiliary function for compiling operation clauses
compileOps :
  OperationClauses Γ E₁ (B , E₂) →
  ∀{S₁ S₂ S₃ Γ₁ Γ'₁ A} →
    OperationCodes B E₁ E₂ Γ (ContTy Γ'₁ (ValTy B :: S₁) (A , E₂) :: (S₁ ++ HandTy Γ₁ S₂ S₃ (A , E₂) :: S₂)) S₃


compileV Unit = PUSH unit
compileV (Var x) = LOOKUP x
compileV {A = A ⟹ (B , E₁)} (Lam e) =
  ABS (λ {S₁ S₂ S₃ Γ₁ Γ'₁ A'} → compileC {S₁ = (ContTy Γ₁ (ValTy B :: S₁) (A' , E₁)) :: _} e RET)

compileC (Handle e With h) k = MARK (compileH h) k (compileC {S₁ = []} e UNMARK)
compileC (Let_In_ {A = A}{E = E₁}{B = B} e1 e2) k =
  compileC e1 $ BIND (compileC {S₁ = (ContTy _ (ValTy B :: _) (_ , _) :: _)} e2 RET) k
compileC (Return v) k = compileV v k
compileC (Do l v) k = compileV v $ CALLOP l k
compileC (App v1 v2) k = compileV v1 $ compileV v2 $ APP k

compileH {D = (B , E₂)} (λx ret |λx,r ops) {Γ₁} {Γ'₁} {S₁} {S₂} {S₃} {A'} =
  (compileC {S₁ = ContTy Γ'₁ (ValTy B :: S₁) (A' , E₂) :: _} ret RET , compileOps ops)

compileOps {E₁ = []} [] = []
compileOps {E₁ = (op A' B') :: E'}{B = B}{E₂ = E₂} (e :: es) {S₁} {S₂} {S₃} {Γ₁} {Γ'₁} {A₁} =
  (compileC {S₁ = ContTy Γ'₁ (ValTy B :: S₁) (A₁ , E₂) :: _} e RET) :: (compileOps es)
```

**Figure 14.** Definition of Compiler

Compilation of computations pushes the value of their subterms onto the stack and then performs necessary actions. For effect handling, compileC uses the MARK instruction to push a compiled handler. In the compilation of handler by compileH, the RET instruction is used as the code continuation because the meta-continuation must be invoked after handling. The code continuation passed to MARK is the code continuation passed to compileC, meaning that the latter is a meta-continuation for the handled computation. In the compilation of the handled computation, the UNMARK instruction is used as the code continuation because the return clause of the handler must be executed after the execution of the handled computation.

The compiling functions all pass the type checking of Agda. This means that our compiler is type preserving.

In Figure 15, we provide a test of our compiler. The test compiles the source program discussed in Section 3.1. The source program is defined as *c*, which includes the handled computation *c1* and the handler *h1*. The compiled version of these expressions are named *code*, *code1*, and *hcode*, respectively. The correctness of the compiler is checked by proving the equivalence between compile *c* and *c1*

## 5 Related Work

***Type-Preserving Compilers.*** Type-preserving compilers have been actively studied since the late 90's. The pioneer work by Morrisett et al. [14] defines a series of type-preserving program transformations: CPS translation, closure conversion, hoisting, allocation, and code generation. The source and target languages of these transformations are all strongly typed: the highest level is System F, the lowest level is a typed assembly language, and the middle levels are typed intermediate languages with specific constructs introduced by the transformations. Our compiler differs from Morrisett et

```
eff : Eff
eff = [ op Unit Unit ]

c1 : Cmp [] (Unit , eff)
c1 = Let Do (here refl) Unit In Return (Var $ here refl)

h1 : Hdl [] ((Unit , eff) ⟹ (Unit , []))
h1 = λx Return (Var $ here refl)
      |λx,r (App (Var $ here refl) Unit :: [])

c : Cmp [] (Unit , [])
c = Handle c1 With h1

code1 : Code [] (HandTy Γ1 S S' (Unit , eff) :: S) S'
code1 = PUSH unit $
          CALLOP {S1 = []} (here refl) $
          BIND {S = []}
            (LOOKUP (here refl) $ RET)
            UNMARK

hcode : HandlerCode [] (Unit , eff) (Unit , [])
hcode {S1 = S1} =
  LOOKUP (here refl) RET ,
  (LOOKUP (here refl) $ PUSH unit $
      APP {S1 = ContTy _ _ _ :: S1} $ RET) :: []

code : Code [] S (ValTy Unit :: S)
code = INITHAND $
         MARK {S1 = []} hcode UNMARK (code1)

compileTest : compile {S = S} c ≡ code {S = S}
compileTest = refl
```

**Figure 15.** Test for compile function

al.'s in that it converts a source program directly into low-level code. However, we believe that it is possible to build a multi-pass version of our compiler.

***Intrinsically Typed Compilers.*** Intrinsically typed compilers are a more recent approach to correct and secure compilation. Chlipala [6] presents a compiler for the simply-typed lambda calculus implemented in Coq. They define the source, intermediate, and target languages all as intrinsically typed ASTs, and implement each compiler pass as a function between those ASTs. Guillemette and Monnier [8] develop a similar compiler in Haskell, using generalized algebraic data types [5, 21] to encode typing rules. Rouvoet et al. [19] show an Agda formalization of a compiler for an imperative language with conditionals and while loops. They use linear types to maintain the invariant that every label in the target language is defined exactly once. Pickard and Hutton [16] formalize in Agda two compilers for functional languages, one featuring exceptions and the other featuring functions. They introduce the notion of code continuations to discard a portion of computation upon exception raising. Our compiler can be understood as a combination of Pickard and Hutton's compilers extended with resumable code continuations.

***Type-Preserving Compilation of Effect Handlers.*** There are several program transformations that compile effect handlers to more primitive (but still high-level) constructs. For example, the evidence translation of Xie et al. [22] eliminates operations and handlers by packaging operation clauses as an evidence vector, which is passed to functions as an additional argument. The CPS translation of Schuster et al. [20] does a similar job by making the continuation inside each handler explicit. Our compiler translates into lower-level language for a stack-based machine. It would be interesting to investigate what we obtain by combining these transformations and the above-mentioned compilers for pure languages.

***Stack Polymorphism.*** Stack polymorphism, which we use to type continuation resumptions, was originally introduced by Morrisett et al. [13]. Their goal is to design a stack-based typed assembly language, and they use stack polymorphism for two purposes: (i) to prevent a function body from manipulating the caller's stack frame, and (ii) to allow functions to be invoked from states with varying stacks. Our purpose of using stack polymorphism is close to the second one: we allow continuations to be resumed from a state with an arbitrary stack.

# 6 Conclusion and Future Work

In this paper, we implemented an intrinsically typed compiler for effect handlers. Following previous work, we formalized the source and target languages as intrinsically typed ASTs, and defined the compiler as a function between those ASTs. To solve the challenge with capture and resumption of continuations, we introduced stack polymorphism that abstracts over the shape of unknown stacks.

With type preservation established, a natural next step is to prove semantics preservation. We conjecture that the property would hold, as our compiler is essentially an extension of Pickard and Hutton's compiler [16] for exceptions, which is correct.

After proving semantics preservation, we plan to investigate type-preserving compilation of other continuation facilities. For instance, the `shift0/reset0` operators [12] have a close relationship with deep effect handlers [7, 17], hence we believe that they can be compiled to a target language similar to $L_T$.

As an orthogonal direction, we intend to extend our compiler with *multiplicities*. This notion comes from quantitative type theories [1], where one can express at the level of types how many times a variable is used in a program. The information about multiplicities has been proven useful for optimizations: for example, in Idris 2 [4], function arguments with multiplicity 0 are erased at runtime since they are not used for computation. In the context of effect handlers, we can use multiplicities to eliminate unnecessary continuation capture: for example, in Koka [10], an operation whose handler calls the continuation only once at a tail position is executed in place. Our plan is to decorate our Agda data types with multiplicities and thus realize a compiler that is efficient by construction.

## Acknowledgments

We thank the anonymous reviewers for their valuable comments, and the members of the Programming Research Group at Tokyo Institute of Technology for helpful discussions. This work was supported by JSPS KAKENHI grant number 23H03368.

## References

[1] Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory *(LICS '18)*. https://doi.org/10.1145/3209108.3209189

[2] Patrick Bahr and Graham Hutton. 2022. Monadic Compiler Calculation (Functional Pearl). *Proc. ACM Program. Lang.* 6, ICFP, Article 93 (Aug 2022), 29 pages. https://doi.org/10.1145/3547624

[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. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming (ECOOP '21)*. https://doi.org/10.4230/DARTS.7.2.10

[5] James Cheney and Ralf Hinze. 2003. *First-class phantom types.* Technical Report. Cornell University.

[6] Adam Chlipala. 2007. A Certified Type-Preserving Compiler from Lambda Calculus to Assembly Language. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 54–65. https://doi.org/10.1145/1250734.1250742

[7] 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

[8] Louis-Julien Guillemette and Stefan Monnier. 2008. A Type-Preserving Compiler in Haskell. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 75–86. https://doi.org/10.1145/1411204.1411218

[9] Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *2nd International Conference on Formal Structures for Computation and Deduction (FSCD '17, Vol. 84)*, Dale Miller (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 18:1–18:19. https://doi.org/10.4230/LIPIcs.FSCD.2017.18

[10] Daan Leijen. 2014. Koka: Programming with row polymorphic effect types. In *5th Workshop on Mathematically Structured Functional Programming (MSFP '14)*.

[11] Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling Environments in Call-by-Value Programming Languages. *Inf. Comput.* 185, 2 (sep 2003), 182–210. https://doi.org/10.1016/S0890-5401(03)00088-9

[12] Marek Materzok and Dariusz Biernacki. 2011. Subtyping Delimited Continuations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 81–93. https://doi.org/10.1145/2034773.2034786

[13] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. 2002. Stack-based typed assembly language. *Journal of Functional Programming* 12, 1 (2002), 43–88. https://doi.org/10.1017/S0956796801004178

[14] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to Typed Assembly Language. *ACM Trans. Program. Lang. Syst.* 21, 3 (may 1999), 527–568. https://doi.org/10.1145/319301.319345

[15] Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory.* Ph. D. Dissertation. Chalmers University of Technology. https://api.semanticscholar.org/CorpusID:118357515

[16] Mitchell Pickard and Graham Hutton. 2021. Calculating Dependently-Typed Compilers (Functional Pearl). *Proc. ACM Program. Lang.* 5, ICFP, Article 82 (Aug 2021), 27 pages. https://doi.org/10.1145/3473587

[17] Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Typed Equivalence of Effect Handlers and Delimited Control. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 131)*, Herman Geuvers (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 30:1–30:16. https://doi.org/10.4230/LIPIcs.FSCD.2019.30

[18] Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

[19] Arjen Rouvoet, Robbert Krebbers, and Eelco Visser. 2021. Intrinsically Typed Compilation with Nameless Labels. *Proceedings of the ACM on Programming Languages* 5, POPL, Article 22 (Jan 2021), 28 pages. https://doi.org/10.1145/3434303

[20] Philipp Schuster, Jonathan Immanuel Brachthäuser, Marius Müller, and Klaus Ostermann. 2022. A Typed Continuation-Passing Translation for Lexical Effect Handlers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 566–579. https://doi.org/10.1145/3519939.3523710

[21] Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*. Association for Computing Machinery, New York, NY, USA, 224–235. https://doi.org/10.1145/604131.604150

[22] Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. 2020. Effect Handlers, Evidently. *Proc. ACM Program. Lang.* 4, ICFP, Article 99 (Aug 2020), 29 pages. https://doi.org/10.1145/3408981

Received 2023-10-20; accepted 2023-11-20