A Type-theoretic Study on Partial Continuations

Yukiyoshi Kameyama

Graduate School of Informatics, Kyoto University 606-8501, JAPAN,
kameyama@kuis.kyoto-u.ac.jp

Abstract. *Partial continuations* are control operators in functional programming such that a function-like object is abstracted from a part of the rest of computation, rather than the whole rest of computation. Several different formulations of partial continuations have been proposed by Felleisen, Danvy&Filinski, Hieb et al, and others, but as far as we know, no one ever studied logic for partial continuations, nor proposed a typed calculus of partial continuations which corresponds to a logical system through the Curry-Howard isomorphism. This paper gives a simple type-theoretic formulation of a form of *partial continuations* (which we call delimited continuations), and study its properties. Our calculus does reflect the intended operational semantics, and enjoys nice properties such as subject reduction and confluence. By restricting the type of delimiters to be atomic, we obtain the normal form property. We also show a few examples.

1 Introduction

The mechanism of first-class continuations (the `call/cc`-mechanism in Scheme [1]) is a quite powerful control facility, and is equipped with many modern programming languages such as Standard ML. Felleisen et al [5] established a theory for first-class continuations by which we can reason about properties of programs with first-class continuations.

Partial continuation is a refinement of first-class continuation in that a continuation object is abstracted from a part of the rest of computation, rather than the whole rest of computation. Felleisen [6] introduced a pair of operators # and $\mathcal{F}$ to represent partial continuations. The former delimits the range of continuations which will be later invoked by the latter operator. The other distinguished feature of partial continuations is that, its invocation does not abort the current continuation, contrary to the first-class continuations. Hence the abstracted objects are normal functions which can be composed with other functions. As Felleisen showed, the concept of partial continuation is useful in practice; interesting examples can be implemented more concisely and efficiently using partial continuations. After then, several different operators for partial continuations have been proposed by Queinnec and Serpette [20], Gunter [11], Danvy and Filinski [2] and others.

If we want to give a logical view to partial continuations through the Curry-Howard isomorphism, a fundamental problem arises in these approaches. Namely, the scope of Felleisen's # operator (and its counterpart in other researcher's calculi) is dynamic; for each $\mathcal{F}$ operator, the matching # is determined at the time of evaluation. Consequently, we cannot represent the scope of # by a simple variable-binding mechanism, thus cannot construct a simple logic corresponding to their operators. One exception is the subcontinuation by Hieb et al [13] whose operator has static scope. However, their operator may generate run-time errors and we cannot develop a type safe calculus for subcontinuations.

In this paper, we give a simple typed calculus for partial continuations which have static scope. Since the scope of a partial continuation is lexically determined by the corresponding delimiter, our variant is called a delimited continuation.[1] Our calculus is designed to satisfy the following conditions: (1) it is a statically typed system which corresponds to, through the Curry-Howard isomorphism, a consistent logical system, (2) it is type safe in the sense that it enjoys the subject reduction property and reductions never get stuck, and (3) its reduction rules are confluent, and compatible with any contexts. By (1), our calculus can be viewed as a logical system. Indeed, it corresponds to classical logic. By (2) and (3), we have an equational theory for programs with partial continuations.

In order to make our type-theoretic analysis easier, we represent (the counterpart of) $\mathcal{F}$ by two operators, an invoker of partial continuations and a throw operation, and give reduction rules. Our reduction rules reflect the intended operational semantics, and enjoy the above properties (1)-(3). We also show that the subcalculus with the delimiter and the throw operation (without the invoker) corresponds to the classical catch/throw calculus in [18, 14], while the subcalculus with the delimiter and the invoker (without the throw operation) corresponds to intuitionistic calculus.

The rest of this paper is organized as follows. Section 2 gives the background and motivation of our formulation. Sections 3, 4 and 5 give the type system, the operational semantics, and the reduction rules of our calculus, respectively. Section 6 proves theoretical properties such as subject reduction and confluence. Section 7 shows that our calculus corresponds to classical logic through the Curry-Howard isomorphism. Section 8 gives the conclusion.

2 Formulating Partial Continuations in Type Theory

We start our analysis with Felleisen's formulation. Felleisen [6] proposed a form of partial partial continuations, which has the following reduction rule:

$$\#E[\mathcal{F}V] \to V(\lambda x.E[x])$$

where E is an evaluation context, V is a value, # is a delimiter which restricts the range of partial continuations, and $\mathcal{F}$ is an operator which creates a partial-continuation object up to the delimiter. In the above term, the created partial continuation object $\lambda x.E[x]$ is applied to the term V. Note that, this object is a simple function, and not abortive in the sense that, when some value is

[1] Olivier Danvy coined this term.

applied, it does not throw away the current continuation. These two features are characteristic points for partial continuations compared with first-class (full) continuations.

Felleisen's partial continuations are refinement of full continuations; theoretically partial continuations behaves well, and are useful in practice. However, if we want to construct a typed calculus for partial continuations which corresponds to a sound logical system, it is problematic.

The problem is that, the scope of the #-operator is dynamic, for instance, $(\lambda x.\#(xN))(\lambda y.\mathcal{F}M)$ reduces to $\#((\lambda y.\mathcal{F}M)N)$, thus $\mathcal{F}$ gets captured by $\#$ through this reduction. In this system, we cannot intuitively understand the meaning of programs. Consider the term $\lambda x.\#(x(\mathcal{F}M))$. We are tempted to consider this $\#$ and $\mathcal{F}$ are connected. But if we apply $\lambda y.C[\#y]$ to it, the above $\mathcal{F}$ is delimited by the latter $\#$. It seems impossible to have a Curry-Howard isomorphism of this kind of calculi and ordinary logical systems, since the correspondence between the #-operator and the $\mathcal{F}$-operator cannot be represented by the variable-binding mechanism (which is lexical).

Danvy and Filinski proposed another formulation of partial continuations [2] [3]. Their operators `reset` and `shift` differ from Felleisen's ones in that the created partial continuation object is again delimited. This change has a better effect for formalizing partial continuations, and in fact, they successfully gave a CPS-translation in an ordinary, functional style. However, still the scope of their `reset` operator (denoted by $\#$ in the above reduction) is also dynamic, hence the same problem applies if one want to formalize their operator in a type theory which admits the Curry-Howard isomorphism.

Hieb, Dybvig and Anderson [13] proposed subcontinuations which essentially have the following reduction rule:

$$\#_l(E[\mathcal{F}_l V]) \to V(\lambda x.\#_l(E[x]))$$

where l is a label attached to the operators. The notable point in their approach is that (i) they can treat multiple labels so that the operator $\mathcal{F}$ can specify the matching delimiter, and (ii) the binding mechanism of the label l is the ordinary variable binding so that it is static. These two points are big benefits for logical viewpoint. However, the $\mathcal{F}$-operator may become unbound through the reduction, causing a run-time type-error (when V contains an occurrence of the label l, then the reduced term may contain l free). Also they did not study a typed calculus for their operator.

Our aim is to develop a theory for partial continuations which is logically well-founded. Since existing partial continuation operators are not satisfactory in the sense of logic, we shall change the operational behavior of existing partial continuations to obtain a logically well-behaved system. We should be careful for this change of semantics so that interesting programming examples with existing partial continuations can be expressible in our calculus. In particular, we should try to make this change as little as possible.

As a conclusion we decided to formalize an improved version of Hieb et al's operator so that no run-time type error may occur. In order to make our analysis easier, we shall not directly formalize the $\mathcal{F}$-operator, but instead we treat two operators `calldc` (stands for "call with partial continuation") and `throw`, which essentially have the following reductions:

$$\#_\alpha E[\mathtt{calldc}_\alpha V] \to \#_\alpha E[V(\lambda x.\#_\alpha E[x])]$$

$$\#_\alpha E[\mathtt{throw}_\alpha V] \to \#_\alpha V$$

where V is a value and E is an evaluation context defined later. The point is that, in the first rule we attach one more delimiter to enclose the resulting term. As is shown later, our (counterpart of the) $\mathcal{F}$-operator can be represented by `calldc` and `throw`.

3 The Type System

We now define the type system of our calculus. Actually we are defining two calculi $\lambda_{\mathtt{DC}}$ and $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$. $\lambda_{\mathtt{DC}}$ is the full calculus, and by putting restriction on terms we obtain $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$.

Types and terms are defined by the following grammar where K is an atomic type, c^K is a constant of atomic type K, and x and α are variables[2]. We assume that `Unit` is an atomic type, and $\bullet$ is a constant (its single element) of type `Unit`.

$$\begin{aligned} A, B &::= K \mid A \to B \mid \neg A \\ M, N &::= x \mid c \mid \lambda x.M \mid MN \\ &\qquad \mid \#_\alpha M \mid \mathtt{calldc}_\alpha M \mid \mathtt{throw}_\alpha M \end{aligned}$$

The type $\neg A$ is the type of tags of control operators for type A. Note that $\neg$ is a primitive type constructor, and not a defined symbol. In $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$, we restrict that the types of tags be atomic, namely in formulating $\neg A$, the type A must be atomic. $\lambda_{\mathtt{DC}}$ has no restriction.

The first line of terms are usual λ-terms. The second line consists of control operators. The term $\#_\alpha M$ delimits the scope of partial continuation which may be created inside M (with the tag α). The term $\mathtt{calldc}_\alpha M$ creates a partial-continuation (delimited continuation) object like $\mathcal{F}$-operator. Our calculi also have an abortive operator $\mathtt{throw}_\alpha M$ which finishes the current continuation up to the corresponding delimiter. As a notational convention, $\#_\alpha M_1 \ldots M_n$ should be parsed as $\#_\alpha (M_1 \ldots M_n)$.

A type environment is a finite set of the form $x : A$ where no variable appears more than once. For instance $\{x : A,\ \alpha : \neg(B \to C)\}$ is a type environment. We use Γ for a type environment. A judgement is in the form $\Gamma \vdash M : A$. The typing rules to derive a judgement are displayed in Table 1. As usual, if two

[2] There is no syntactic distinction of x and α, but we use x for ordinary variable and α for tags.

type environments Γ_1 and Γ_2 are not compatible, then $\Gamma_1 \cup \Gamma_2$ is not defined. If $\Gamma \vdash M : A$ is derived, we say M is a (typable) term of type A under the type environment Γ. We sometimes omit type environments if they are apparent.

Table 1. Typing Rules

$$\frac{}{\Gamma \cup \{x : A\} \vdash x : A} \qquad \frac{}{\Gamma \vdash c^K : K}$$

$$\frac{\Gamma \cup \{x : A\} \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad \frac{\Gamma_1 \vdash M : A \to B \quad \Gamma_2 \vdash N : A}{\Gamma_1 \cup \Gamma_2 \vdash MN : B}$$

$$\frac{\Gamma \cup \{\alpha : \neg B\} \vdash M : B}{\Gamma \vdash \#_\alpha M : B}$$

$$\frac{\Gamma \vdash M : ((\mathtt{Unit} \to A) \to B) \to A}{\Gamma \cup \{\alpha : \neg B\} \vdash \mathtt{calldc}_\alpha M : A} \qquad \frac{\Gamma \vdash M : B}{\Gamma \cup \{\alpha : \neg B\} \vdash \mathtt{throw}_\alpha M : C}$$

In Table 1, The first three rules are as usual. The fourth rule is for the control delimiter, and as we explained, it discharges the assumption $\alpha : \neg B$, namely, it binds the variable α.

The fifth and sixth rules are for `calldc` and `throw`. The role of `calldc` is almost the same as $\mathcal{F}$, but its type is slightly different, since (i) we split the role of $\mathcal{F}$ into two operators `calldc` and `throw`, and (ii) by controlling the evaluation order suitably, we used a *thunk* of type $\mathtt{Unit} \to A$ in the rule for `calldc`. We can delay the evaluation of a term M of type A by encapsulating it as $\lambda x^{\mathtt{Unit}}.M$. This technique is useful when we argue the correspondence between the ML-like operational semantics and this formulation in Section 6. Since $\mathtt{throw}_\alpha M$ aborts the current continuation and jumps to the corresponding control-delimiter, its type can be any type. The variable α in these two operators is a free occurrence.

The variables x and α are bound by $\lambda x.M$ and $\#_\alpha M$, respectively, and $FV(M)$ denotes the set of free variables in M. As usual, we identify two terms which differ in bound variables only. The term $M[N/x]$ denotes the term M substituted N for x. We also say that a term M is closed if $FV(M) = \{\}$.

Note that α is an ordinary variable, so we may abstract α like $\lambda\alpha.M$. This has practical benefit, since we often want to define functions with free tags, and bind them in another function. Let us show an example in Scheme-like language. We often want to write the following program:

```
(define (foo x)
   ...   (callpc alpha ...) ...)
```

```
(define (goo y)
   (catch alpha (foo y)))
```

In our calculus, such a program is not allowed (since our control operators have static scope). However, we can represent the above program by abstracting the variable `alpha` in `foo`. The resulting program is:

```
(define (foo x beta)
   ...   (callpc beta ...) ...)

(define (goo y)
   (catch alpha (foo y alpha)))
```

By this technique, we can (partly) recover the expressiveness, which was lost by changing the scope of delimiters from dynamic one to static one. However, not all expressible programs with dynamic scope operators can be expressed in our calculus. We think that it is a trade-off of theory and practice.

4 ML-like Operational Semantics

In this section we give an operational semantics of our calculi in the style of Felleisen et al [8]. Note that this operational semantics may cause run-time type errors, therefore its direct formalization (in a type safe way) is not possible. We nevertheless state the operational semantics here to clarify our intented semantics. Later we show that the equality in $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$ corresponds to this operational semantics.

We first define values (denoted by V), redexes (denoted by R), and evaluation contexts (denoted by E) as follows:

$$
\begin{aligned}
V &::= x \mid c \mid \lambda x.M \\
R &::= (\lambda x.M)V \mid \#_\alpha V \mid \mathtt{calldc}_\alpha V \mid \mathtt{throw}_\alpha V \\
E &::= [\,] \mid EM \mid VE \mid \mathtt{calldc}_\alpha E \mid \mathtt{throw}_\alpha E \mid \#_\alpha E
\end{aligned}
$$

Then we have that, any closed term M is either a value, or there uniquely exists a pair of a redex R and an evaluation context E such that $M \equiv E[R]$.

We have the following 1-step reduction rules where E_0 is an evaluation context which does not bind α.

$$
\begin{aligned}
E[(\lambda x.M)V] &\rightsquigarrow_1 E[M[V/x]] \\
E[\#_\alpha V] &\rightsquigarrow_1 E[V] \\
E[\#_\alpha E_0[\mathtt{calldc}_\alpha V]] &\rightsquigarrow_1 E[\#_\alpha E_0[V(\lambda u.\#_\alpha E_0[u\bullet])]] \\
E[\#_\alpha E_0[\mathtt{throw}_\alpha V]] &\rightsquigarrow_1 E[V] \\
E_0[\mathtt{calldc}_\alpha V] &\rightsquigarrow_1 \mathrm{error} \\
E_0[\mathtt{throw}_\alpha V] &\rightsquigarrow_1 \mathrm{error}
\end{aligned}
$$

Since the decomposition of a term is unique, the above set of reduction rules induces a deterministic evaluation strategy.

Run-time errors may happen for $\lambda_{\texttt{DC}}$ even if a reduction begins with a closed term; in the second and the fourth rules, the value V may be $\lambda x.M$ and M may contain free occurrences of α. For instance:

$$(\texttt{\#}_\alpha \lambda x.\texttt{calldc}_\alpha x)V \rightsquigarrow_1 (\lambda x.\texttt{calldc}_\alpha x)V \rightsquigarrow_1 \texttt{calldc}_\alpha V \rightsquigarrow_1 \text{error}$$

On the contrary, no run-time errors occur in $\lambda_{\texttt{DC}}^{\text{atomic}}$, since the value V in $\texttt{\#}_\alpha V$ and $\texttt{throw}_\alpha V$ must be either a variable or a constant.

5 Small-Step Reductions

We want to set up an equational theory to reason about the programs in $\lambda_{\texttt{DC}}^{\text{atomic}}$ and $\lambda_{\texttt{DC}}$. In this respect the reduction step given in the previous section is too big. This section gives finer reduction rules which are easier to study.

First we define a singular context E_s as follows:

$$E_s ::= [\,]M \mid V[\,] \mid \texttt{calldc}_\alpha[\,] \mid \texttt{throw}_\alpha[\,]$$

Next we define the notion of one-step reduction denoted by $\to_1$, which is defined as the compatible closure of the following reduction rules.

We split the reduction rules into two groups. The first group of reduction rules are as follows:

$$\begin{aligned} (\lambda x.M)V &\to_1 M[V/x] && && (1)\\ \texttt{\#}_\alpha M &\to_1 M && (\text{if } \alpha \notin FV(M)) && (2)\\ \texttt{\#}_\alpha \texttt{\#}_\beta M &\to_1 \texttt{\#}_\alpha M[\alpha/\beta] && && (3) \end{aligned}$$

The first one is the usual call-by-value β-reduction. The second one means that if there are no control operators with tag α, then the delimiter is useless. The last one means that, if two delimiters are set in the same place, they can be unified.

The second group of reduction rules are as follows:

$$\begin{aligned} \texttt{\#}_\alpha(\texttt{calldc}_\alpha V) &\to_1 \texttt{\#}_\alpha(V(\lambda u.\texttt{\#}_\alpha u\bullet)) && && (4)\\ \texttt{\#}_\alpha(\texttt{throw}_\alpha V) &\to_1 \texttt{\#}_\alpha V && && (5)\\ E_s[\texttt{calldc}_\alpha V] &\to_1 \texttt{calldc}_\alpha(\lambda x.E_s[V(\lambda y.\texttt{\#}_\alpha x(\lambda z.E_s[y\bullet]))]) && && (6)\\ \texttt{\#}_\beta(\texttt{calldc}_\alpha \lambda x.M) &\to_1 \texttt{calldc}_\alpha \lambda x.\texttt{\#}_\beta M && (if\ \alpha \not\equiv \beta) && (7)\\ E_s[\texttt{throw}_\alpha V] &\to_1 \texttt{throw}_\alpha V && && (8) \end{aligned}$$

In these reductions we assume x, y, z are fresh variables.

The first two rules express reductions with an empty partial continuation. The third and fourth rules are one-step reductions for `calldc` and `throw`.

Although the righthand side of the third rule (Rule (6)) may look quite complex, it is similar to Felleisen's one-step reduction rule for first-class continuations. A crucial point of this reduction rule is that, in the righthand side, the partial continuation object is delimited by a newly introduced delimiter $\#_\alpha$, so the occurrences of $\mathtt{calldc}_\alpha$ in E_s are bound by this new $\#_\alpha$. If we do not have a new delimiter in the reduct, we cannot simulate many interesting reductions which were written with partial continuations of dynamic scope.

Let $\rightarrow$ be a reflexive, transitive closure of $\rightarrow_1$, and $=$ be the least equivalence relation which contains $\rightarrow$.

Representing the $\mathcal{F}$-like operator

The `calldc` operator does not discard the current continuation. But we can define an operator for creating partial continuation objects, which discards the current continuation. Let us define `control` as follows:

$$\mathtt{control}_\alpha M \stackrel{\triangle}{=} \mathtt{calldc}_\alpha(\lambda x.\mathtt{throw}_\alpha M x)$$

Then this operator has the following reduction in the ML-like operational semantics as desired.

$$E[\mathtt{\#}_\alpha E_0[\mathtt{control}_\alpha V]] \leadsto E[\mathtt{\#}_\alpha V(\lambda u.\mathtt{\#}_\alpha E_0[u\bullet])]$$

The `control` operator is closer to Felleisen's $\mathcal{F}$-operator and Danvy and Filinski's `shift` operator.

A Small Example

The following example was given by Danvy and Filinski [2]. In order to express this example, we assume that $+$ and its reductions were added to our calculus.

$$\begin{aligned}
& 1 + \mathtt{\#}_\alpha(10 + (\lambda x.\mathtt{control}_\alpha(\lambda k.k(k(x))))100) \\
\rightarrow\ & 1 + \mathtt{\#}_\alpha(10 + \mathtt{control}_\alpha(\lambda k.k(k(100)))) \\
\equiv\ & 1 + \mathtt{\#}_\alpha(10 + \mathtt{calldc}_\alpha(\lambda y.\mathtt{throw}_\alpha(\lambda k.k(k(100)))y)) \\
\rightarrow\ & 1 + \mathtt{\#}_\alpha(10 + (\lambda y.\mathtt{throw}_\alpha(y(y(100))))(\lambda u.\mathtt{\#}_\alpha 10 + u)) \\
\rightarrow\ & 1 + \mathtt{\#}_\alpha\mathtt{throw}_\alpha((\lambda u.10 + u)((\lambda u.10 + u)(100))) \\
\rightarrow\ & 1 + \mathtt{\#}_\alpha 120 \\
\rightarrow\ & 121
\end{aligned}$$

A Note on Control Operators with Dynamic Scope

As we explained, our control operators have static scopes. If we would change them to have dynamic scopes in $\lambda_{\mathtt{DC}}$, we would have a non-terminating reduction as follows.

Let P be $\lambda x.(\#_\alpha(\lambda y.z)(xw))x$, and Q be $\lambda u.\mathtt{throw}_\alpha P$. Then, PQ has type $(C \to A) \to B$ under the type environment $\{z : (C \to A) \to B,\ w : C\}$. If the delimiter has dynamic scope, the reduction sequence from PQ does not terminate as follows:

$$\begin{aligned}
PQ &\to (\#_\alpha((\lambda y.z)(Qw)))Q \\
&\to (\#_\alpha((\lambda y.z)(\mathtt{throw}_\alpha P)))Q \\
&\to (\#_\alpha \mathtt{throw}_\alpha P)Q \\
&\to (\#_\alpha P)Q \\
&\to PQ
\end{aligned}$$

We do not have this reduction sequence with static scope operators, since we must rename α in P before substituting Q for x in P.

$$\begin{aligned}
PQ &\to (\#_\beta((\lambda y.z)(Qw)))Q \\
&\to (\#_\beta((\lambda y.z)(\mathtt{throw}_\alpha P)))Q \\
&\to (\#_\beta \mathtt{throw}_\alpha P)Q \\
&\to (\mathtt{throw}_\alpha P)Q \\
&\to \mathtt{throw}_\alpha P
\end{aligned}$$

The term $\mathtt{throw}_\alpha P$ cannot be reduced any further.

6 Properties of our Calculus

This section gives properties of $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$ and $\lambda_{\mathtt{DC}}$.

First of all, the reductions are closed under substitution.

Theorem 1. *If $M \to M'$ and $N \to N'$, then $M[N/x] \to M'[N'/x]$.*

This is obvious since our reduction rules are compatible with arbitrary contexts. Note that the calculus for the full continuations contains so called a computation rule which is not compatible with contexts. For instance $\mathcal{C}M \to M(\lambda x.\mathcal{A}x)$ is only applicable for the top-level context where $\mathcal{C}$ is Felleisen's control operator, and $\mathcal{A}$ is the abort operator.

The subject reduction property expresses the type safety.

Theorem 2 (Subject Reduction Property). *If $\Gamma \vdash M : A$ and $M \to N$, then we have that $\Gamma \vdash N : A$ and $FV(M) \subseteq FV(N)$.*

Proof. We only verify the most complex reduction rule (6). Suppose that E_s has type C, and its hole has type A. The lefthand-side term is typed as follows (omitting the type environment for readability):

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ M : ((\texttt{Unit} \to A) \to B) \to A\end{array}}{\texttt{calldc}_\alpha M : A}}{E_s[\texttt{calldc}_\alpha M] : C}$$

Then, the righthand-side term is typed as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ M : ((\texttt{Unit} \to A) \to B) \to A\end{array} \quad \dfrac{\dfrac{\dfrac{x : (\texttt{Unit} \to C) \to B \quad \dfrac{\dfrac{\dfrac{y : \texttt{Unit} \to A \quad \bullet : \texttt{Unit}}{y\bullet : A}}{E_s[y\bullet] : C}}{\lambda z.E_s[y\bullet] : \texttt{Unit} \to C}}{x(\lambda z.E_s[y\bullet]) : B}}{\texttt{\#}_\alpha x(\lambda z.E_s[y\bullet]) : B}}{\lambda y.\texttt{\#}_\alpha x(\lambda z.E_s[y\bullet]) : (\texttt{Unit} \to A) \to B}}{M(\lambda y.\texttt{\#}_\alpha x(\lambda z.E_s[y\bullet])) : A}}{E_s[M(\lambda y.\texttt{\#}_\alpha x(\lambda z.E_s[y\bullet]))] : C}}{\lambda x.E_s[M(\lambda y.\texttt{\#}_\alpha x(\lambda z.E_s[y\bullet]))] : ((\texttt{Unit} \to C) \to B) \to C}}{\texttt{calldc}_\alpha(\lambda x.E_s[M(\lambda y.\texttt{\#}_\alpha x(\lambda z.E_s[y\bullet]))]) : C}$$

We also have that the set of free variables are the same.

Other cases are proved easily. □

We then show that $\lambda_{\texttt{DC}}$ and $\lambda_{\texttt{DC}}^{\text{atomic}}$ are confluent by using Takahashi's parallel reduction method [19] in conjunction with Hardin's interpretation method.

Theorem 3. *The calculi $\lambda_{\texttt{DC}}$ and $\lambda_{\texttt{DC}}^{\text{atomic}}$ are confluent.*

Proof. We first define a d-normal form $d(M)$ of a term M as the term M where the reduction (3) (unification of delimiters) is applied as many times as possible, namely, contracting successive application of delimiters. Apparently, for each term, its d-normal form is unique up to renaming of bound variables.

We then define the parallel reduction $\Rightarrow$ on terms as follows:

- $x \Rightarrow x$, and $c \Rightarrow c$.
- If $M_i \Rightarrow M_i'$ for $i = 1, 2$, then $\lambda x.M_1 \Rightarrow \lambda x.M_1'$, $M_1 M_2 \Rightarrow M_1' M_2'$, $\texttt{calldc}_\alpha M_1 \Rightarrow \texttt{calldc}_\alpha M_1'$, $\texttt{throw}_\alpha M_1 \Rightarrow \texttt{throw}_\alpha M_1'$, and $\texttt{\#}_\alpha M_1 \Rightarrow \texttt{\#}_\alpha M_1'$.
- If $M \Rightarrow M'$, and $V \Rightarrow V'$, then $(\lambda x.M)V \Rightarrow M'[V'/x]$.
- If $M \Rightarrow M'$ and $\alpha \notin FV(M)$, then $\texttt{\#}_\alpha M \Rightarrow M'$.
- If $V \Rightarrow V'$, then $\texttt{\#}_\alpha \texttt{calldc}_\alpha V \Rightarrow \texttt{\#}_\alpha V'(\lambda u.\texttt{\#}_\alpha u\bullet)$.
- If $V \Rightarrow V'$, then $\texttt{\#}_\alpha \texttt{throw}_\alpha V \Rightarrow \texttt{\#}_\alpha V'$.
- If $V \Rightarrow V'$, $E_s \Rightarrow E_s'$, then $E_s[\texttt{calldc}_\alpha V] \Rightarrow \texttt{calldc}_\alpha(\lambda x.E_s'[V'(\lambda y.\texttt{\#}_\alpha x(\lambda z.E_s'[y\bullet]))])$.
- If $V \Rightarrow V'$, then $E_s[\texttt{throw}_\alpha V] \Rightarrow \texttt{throw}_\alpha V'$.

- If $M \Rightarrow M'$, then $\texttt{\#}_\beta\texttt{calldc}_\alpha\lambda x.M \Rightarrow \texttt{calldc}_\alpha\lambda x.\texttt{\#}_\beta M'$.
- If $M \Rightarrow M'$, then $M \Rightarrow d(M')$.

Next, we define the term M^* for each term M as follows. In the following definition, if more than one clause match the term M, then we always take the first matching clause as the definition of M^*.

- $x^* \equiv x$, and $c^* \equiv c$.
- $(\lambda x.M)^* \equiv \lambda x.M^*$,
- $((\lambda x.M)V)^* \equiv d(M^*[V^*/x])$.
- $(E_s[\texttt{calldc}_\alpha V])^* \equiv d(\texttt{calldc}_\alpha \lambda x.E_s^*[V^*(\lambda y.\texttt{\#}_\alpha x(\lambda z.E_s^*[y\bullet]))])$.
- $(E_s[\texttt{throw}_\alpha V])^* \equiv \texttt{throw}_\alpha V^*$.
- $(MN)^* \equiv M^*N^*$.
- $(\texttt{calldc}_\alpha M)^* \equiv \texttt{calldc}_\alpha M^*$, and $(\texttt{throw}_\alpha M)^* \equiv \texttt{throw}_\alpha M^*$.
- $(\texttt{\#}_\alpha M)^* \equiv M^*$ if $\alpha \notin FV(M)$.
- $(\texttt{\#}_\alpha\texttt{\#}_\beta M)^* \equiv (\texttt{\#}_\beta M)^*[\alpha/\beta]$.
- $(\texttt{\#}_\alpha\texttt{calldc}_\alpha V)^* \equiv \texttt{\#}_\alpha(V^*(\lambda u.u\bullet))$.
- $(\texttt{\#}_\alpha\texttt{throw}_\alpha V)^* \equiv d(\texttt{\#}_\alpha V^*)$.
- $(\texttt{\#}_\beta\texttt{calldc}_\alpha\lambda x.M)^* \equiv d(\texttt{calldc}_\alpha\lambda x.\texttt{\#}_\beta M^*)$.
- $(\texttt{\#}_\alpha M)^* \equiv d(\texttt{\#}_\alpha M^*)$.

Then by case-analysis, we have that if $M \Rightarrow N$ then $N \Rightarrow M^*$, which implies the diamond property of $\Rightarrow$. In this proof, the only problematic case is $\texttt{\#}_\gamma\texttt{\#}_\beta\texttt{calldc}_\alpha\lambda x.M \Rightarrow \texttt{\#}_\gamma\texttt{calldc}_\alpha\lambda x.\texttt{\#}_\beta M$, but it can be probed by some calculation.

We also have, $M \to N$ implies $M \Rightarrow N$ and then $\to$ is confluent.

Note that these arguments apply to both $\lambda_{\texttt{DC}}^{\text{atomic}}$ and $\lambda_{\texttt{DC}}$. $\square$

Subject reduction and confluence are most basic properties of the typed calculi, but by restricting the tag types to be atomic, we have a few more desirable properties.

Theorem 4 (Normal Form Property). *Let M be a closed normal term in $\lambda_{\texttt{DC}}^{\text{atomic}}$. Then M is either a constant or in the form of $\lambda x.M'$.*

Proof. We say a term M is half-closed if $FV(M) = \{x_1, \cdots, x_n\}$ and the type of x_i is $\neg A_i$ for $1 \le i \le n$. We can show by induction that, a half-closed normal term M is in the following forms:

$$x_i, \quad c, \quad \lambda x.M, \quad \texttt{calldc}_\alpha V, \quad \text{or} \quad \texttt{throw}_\alpha V$$

The point here is that the type $\neg A$ is not defined as $A \supset \bot$, in which case $x_i c$ may be a half-closed normal term.

It follows that, if M is a closed normal term, it is a constant or a λ-abstract. $\square$

The normal form property together with the subject reduction property ensures the type soundness for $\lambda_{\texttt{DC}}^{\text{atomic}}$.

Unfortunately, the normal form property does not hold for $\lambda_{\texttt{DC}}$; there is a closed normal term of the form $\texttt{\#}_\alpha\lambda x.M$ in $\lambda_{\texttt{DC}}$. To obtain the property in $\lambda_{\texttt{DC}}$, we should add reduction rules such as $(\#_\alpha V_1)V_2 \to_1 \#_\alpha V_1'$ where V_1' is obtained by appropriate substitution. However this single reduction rule does not suffice, and we should add more and more. We believe that, even under the restriction of $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$, we can express many programming examples, since we usually do not want to place delimiters for function types.

We finally show that our small step reductions do characterize the operational semantics given earlier.

Theorem 5. *Let M and N be closed terms. If $M \leadsto N$, then $M = N$ in $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$.*

Proof. The theorem is proved by induction on the length of evaluation. The base case is trivial. We list here only mojor cases for the induction step.

Case-1. $E[(\lambda x.M)V] \leadsto E[M[V/x]]$.

The same reduction is included in $\to_1$.

Case-2. $E[\#_\alpha V] \leadsto E[V]$ where $\alpha \notin FV(V)$.

Since we restricted the type of α be of the form $\neg K$ where K is atomic, the term V must be a constant of that type. We have $\#_\alpha c \to_1 c$ in $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$.

Case-3. $E[\#_\alpha E_0[\mathtt{calldc}_\alpha V]] \leadsto E[\#_\alpha E_0[V(\lambda u.\#_\alpha E_0[u\bullet])]]$.

We use one more induction on the structure of E_0 to prove this case.

Case-3-1. If $E_0 \equiv [\ ]$ then, $E[\#_\alpha \mathtt{calldc}_\alpha V] \to_1 E[\#_\alpha V(\lambda u.\#_\alpha u\bullet)]$, which is $E[\#_\alpha E'[V(\lambda u.\#_\alpha u\bullet)]]$.

Case-3-2. If $E_0 \equiv F[E_s]$ where F is an evaluation context and E_s is a singular context, then we have

$$\begin{aligned}
\#_\alpha F[E_s[\mathtt{calldc}_\alpha V]] &\to_1 \#_\alpha F[\mathtt{calldc}_\alpha \lambda x.E_s[V(\lambda y.\#_\alpha x(\lambda z.E_s[y\bullet]))]] \\
&= \#_\alpha F[(\lambda x.E_s[V(\lambda y.\#_\alpha x(\lambda z.E_s[y\bullet]))])(\lambda u.\#_\alpha F[u\bullet])] \\
&\qquad \text{(by induction hypothesis)} \\
&\to \#_\alpha F[E_s[V(\lambda y.\#_\alpha(\lambda u.\#_\beta F'[u\bullet])(\lambda z.E_s[y\bullet]))]] \\
&\to \#_\alpha F[E_s[V(\lambda y.\#_\alpha \#_\beta F'[E_s[y\bullet]])]] \\
&\to \#_\alpha F[E_s[V(\lambda y.\#_\alpha F[E_s[y\bullet]])]] \\
&\equiv \#_\alpha E_0[V(\lambda u.\#_\alpha E_0[u\bullet])]
\end{aligned}$$

where F' is the result of substituting β for free occurrences of α in F to avoid conflict of bound variables.

Case-3-3. If $E_0 \equiv F[\#_\beta[\ ]]$ where F is an evaluation context, then we have (assuming V is $\lambda x.M$)

$$\begin{aligned}
E[\#_\alpha F[\#_\beta \mathtt{calldc}_\alpha \lambda x.M]] &\to_1 E[\#_\alpha F[\mathtt{calldc}_\alpha \lambda x.\#_\beta M]] \\
&= E[\#_\alpha F[(\lambda x.\#_\beta M)(\lambda u.\#_\alpha F[u\bullet])]] \\
&\qquad \text{(by induction hypothesis)} \\
&\to E[\#_\alpha F[\#_\beta M[\lambda u.\#_\alpha F[u\bullet]/x]]]
\end{aligned}$$

On the other hand, we have

$$E[\#_\alpha F[\#_\beta(\lambda x.M)(\lambda u.\#_\alpha F[\#_\beta u\bullet])]] \to_1 E[\#_\alpha F[\#_\beta(\lambda x.M)(\lambda u.\#_\alpha F[u\bullet])]]$$
$$\to_1 E[\#_\alpha F[\#_\beta M[\lambda u.\#_\alpha F[u\bullet]/x]]]$$

So we have the equality. When V is not in the form $\lambda x.M$, namely, it is a variable or a constant, then the proof is easier since there are no free occurrences of the tag β.

Case-4. $E[\#_\alpha E_0[\mathtt{throw}_\alpha V]] \rightsquigarrow E[V]$

Since $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$ does not allow λ-abstraction for the above V, V must be either a variable or a constant. We then prove this case similarly to the Case-3. If E_0 is composed by a delimiter, namely, $E_0 \equiv \#_\beta[\ ]$, then we use the fact that V does not contain β free. We also have $E[\#_\alpha E'[\mathtt{throw}_\alpha V]] \to E[\#_\alpha V]$, and $\#_\alpha V \to_1 V$. Other cases are easy. □

By this theorem and the confluence of $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$, we have the following corollary, which means our reduction rules really reflect the intended operational semantics.

Corollary 1 (Correspondence of $\rightsquigarrow$ and $\to$ in $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$). *Let M be a closed term and V be a value. If $M \rightsquigarrow V$, then $M \to V$ in $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$.*

7 A Logical View

The Curry-Howard isomorphism relates typed lambda calculi and intuitionistic logical systems. As Griffin and other researchers showed, the isomorphism can be extended to the relationship between typed lambda calculi with sequential control operators and classical logic [10]. In this section, we show that $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$ and $\lambda_{\mathtt{DC}}$ also correspond to classical logic.

$\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$ and $\lambda_{\mathtt{DC}}$ are Classical Logic

We assume that $\bot$ is included as an atomic type in $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$. (If it is not the case, choose any atomic type as $\bot$, since we do not use the $\bot$-elimination rule.) Let ϕ be a map which simply discards the lefthand-side of colon from a judgement $M : A$, namely, $\phi(M : A) \equiv A$. A type in $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$ and $\lambda_{\mathtt{DC}}$ can be regarded as a formula in implicational logic where $\to$ is interpreted by implication, $\neg A$ is interpreted by $A \supset \bot$, and `Unit` is a provable formula (say, $\bot \to \bot$). The map ϕ naturally extends to type environments.

Theorem 6 (Isomorphism between $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}/\lambda_{\mathtt{DC}}$ and classical logic). *Let Γ be a type environment, M be a preterm, and A be a type (a formula). Then $\Gamma \vdash M : A$ holds in $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$ if and only if $\phi(\Gamma) \vdash A$ holds in a classical implicational logic. The same thing holds for $\lambda_{\mathtt{DC}}$.*

Proof (only-if part). We only have to verify the following three typing rules are preserved through ϕ.

(i) For the case of the delimiter, we have to infer $\Gamma \vdash B$ from $\Gamma \cup \{\neg B\} \vdash B$. (ii) For the case of `calldc`, we have to infer $\Gamma \cup \{\neg B\} \vdash A$ from $\Gamma \vdash ((\mathtt{Unit} \to A) \to B) \to A$ (`Unit` is some provable formula). (iii) For the case of `throw`, we have to infer $\Gamma \cup \{\neg B\} \vdash C$ from $\Gamma \vdash B$.

All of them can be proved in classical logic.

Proof (if part).

We only have to show the classical reasoning can be simulated by $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$. To show this, we shall prove that, for any type A and B, there exists a closed term of type $((A \to B) \to A) \to A$. This is shown by induction on the type A.

For the base case (A is an atomic type K), the following figure shows that $((K \to B) \to K) \to K$ is inhabited.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{y : (K \to B) \to K\} \vdash y : (K \to B) \to K} \quad \dfrac{\dfrac{\dfrac{}{\{x : K\} \vdash x : K}}{\{x : K, \alpha : \neg K\} \vdash \mathtt{throw}_\alpha x : B}}{\{\alpha : \neg K\} \vdash \lambda x.\mathtt{throw}_\alpha x : K \to B}}{\{y : (K \to B) \to K, \alpha : \neg(K)\} \vdash y(\lambda x.\mathtt{throw}_\alpha x) : K}}{\{y : (K \to B) \to K\} \vdash \#_\alpha y(\lambda x.\mathtt{throw}_\alpha x) : K}}{\{\} \vdash \lambda y.\#_\alpha y(\lambda x.\mathtt{throw}_\alpha x) : ((K \to B) \to K) \to K}$$

For the induction step, let A be $C \to D$. By induction hypothesis, we have a closed term M of type $((D \to B) \to D) \to D$. Then we can easily show that a closed term $\lambda ux.M(\lambda y.u(\lambda z.y(zx))x)$ has type $((A \to B) \to A) \to A$. Hence we can prove all the instances of Peirce's law in $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$.

The proof is even easier for $\lambda_{\mathtt{DC}}$, since we can use an arbitrary type in place of K in the figure above. $\square$

Note that we used the `throw`-operator only, which means that $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$ (and $\lambda_{\mathtt{DC}}$) without the `calldc`-operator is still a classical calculus. In fact, Sato and the present author developed calculi for the catch/throw mechanism [18, 14] which correspond to classical propositional logic. It is easy to see that the delimiter and the `throw`-operator can be understood as the catch and the throw operators, and such a subcalculus of $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$ (or $\lambda_{\mathtt{DC}}$) constitutes a confluent subcalculus of the classical catch/throw calculi in [18, 14].

Note also that our reduction rules can be thought as proof reductions in classical logic. We already gave the subject reduction for the reduction (6), and if we forget the terms, we obtain a proof reduction. The resulting proof-reduction is (by nature) similar to Griffin's reduction rules for Felleisen's control operator.

The `calldc`-operator is not Classical

On the other hand, if we eliminate the `throw`-operator from $\lambda_{\mathtt{DC}}^{\mathrm{atomic}}$ (or $\lambda_{\mathtt{DC}}$), the resulting calculus becomes strictly weaker in the logical sense.

Suppose $\{\} \vdash M : A$ is inferred in $\lambda_{\mathtt{DC}}$ where M does not contain the `throw`-operator. We shall show that A is provable in minimal logic. To simplify the matter, we assume M uses only one tag variable α. Suppose M contains k

subterms of the form $\texttt{calldc}_\alpha N_i$, and the type of N_i is $((\texttt{Unit} \to A_i) \to B) \to A_i$. We put $P_i \equiv (((\texttt{Unit} \to A_i) \to B) \to A_i) \to A_i$. The introduction rule of `calldc` is (when mapped by ϕ) provable if we assume each P_i. Hence we can regard the delimiter introduction rule (through ϕ) as eliminating $P_1, \cdots, P_k$ from the assumption list. In other words, our goal is to prove $\Gamma \vdash B$ from $\Gamma, P_1, \cdots, P_k \vdash B$. But the formula $(\bigwedge_{i=1}^{k} P_i \to B) \to B$ is provable in minimal logic (which can be shown by induction on k).

From this fact, one may think that the `calldc`-operator may be expressible by standard combinators such as S and K, but we believe this is not the case. The proof term of the above theorem has the same type as our control operators, but it behaves quite differently (the latter term is not interesting in computational aspects).

8 Conclusion

Partial continuations were proposed by Felleisen and others and there are many researches on partial continuations since then. Compared to existing calculi for partial continuations, the characteristic feature of our approach is that our calculus is based on a type-theoretic framework. We showed that our calculus (i) enjoys the subject reduction property (ii) is confluent, (iii) does admit the standard Curry-Howard isomorphism (by which it corresponds to classical logic). Hieb et al's subcontinuation also has static scope, but their approach also lacks the type-safety property (which means that it sometimes generates uncaught partial-continuation object). Our approach can be thought as a refinement of (a typed version of) Hieb et al's subcontinuations, and we believe that our calculus can be a basis of syntactic, type-theoretic analysis for partial continuations and other variations of control operators.

Since we can abstract tags, we believe that many examples by Felleisen's one and Danvy and Filinski's one can be written in our calculus. In fact we already worked in tree-traversal example by Felleisen [6].

Future Work. So far, several research topics are left for future work. The first target is the strong normalization (SN) of $\lambda_{\texttt{DC}}^{\text{atomic}}$. A common tool to show it is a type-preserving CPS-translation. We gave a CPS-translation for $\lambda_{\texttt{DC}}^{\text{atomic}}$ in our draft, but it does not preserve typing, so the SN of $\lambda_{\texttt{DC}}^{\text{atomic}}$ is an open problem. Other directions are expressiveness and application. In this paper, we confined ourselves to sequential programs, but as many authors pointed out, partial continuations are a useful tool for giving control over parallel/concurrent programs. Also, there should be applications for formalizing mobile computing.

Acknowledgement

The author would like to thank Olivier Danvy, Kenichi Asai and anonymous referees for pointing out errors in the earlier drafts and let him know references. He would also like to thank Masahiko Sato and Izumi Takeuti for helpful discussions. This work was supported in part by Grant-in-Aid for Scientific Research from the Ministry of Education, Science and Culture of Japan, No. 11780213.

References

1. Kelsey, R., W. Clinger, and J. Rees (eds.): Revised^5 Report on the Algorithmic Language Scheme, 1998.
2. Danvy, O. and A. Filinski: Abstracting Control, Proc. 1990 ACM Conference on Lisp and Functional Programming, pp. 151-160, 1990.
3. Danvy, O. and A. Filinski: Representing Control: a Study of the CPS Transformation, Mathematical Structures in Computer Science 2(4), pp. 361-391, 1992.
4. de Groote, P.: A Simple Calculus of Exception Handling, Typed Lambda Calculi and its Applications (Dezani-Ciancaglini, M. and G. Plotkin eds.), Lecture Notes in Computer Science **902**, pp. 201–215. Springer, 1995.
5. Felleisen, M., D. Friedman, E. Kohlbecker, and B. Duba: A Syntactic Theory of Sequential Control, Theoretical Computer Science 52, pp. 205-237, 1987.
6. Felleisen, M.: The Theory and Practice of First-Class Prompts, Proc. 15th ACM Symp. on Principles of Programming Languages, pp. 180-190, 1988.
7. Felleisen, M., M. Wand, D. Friedman, and B. Duba: Abstract Continuations: A Mathematical Semantics for Handling Full Functional Jumps, Proc. 1988 ACM Conf. on Lisp and Functional Programming, pp. 52-62, 1988.
8. Felleisen, M., and R. Hieb: The Revised Report on the Syntactic Theories of Sequential Control and State, Theoretical Computer Science 103, pp. 235-271, 1992.
9. Filinski, A.: Representing Monads, Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 446-457, 1994.
10. Griffin, T.: A Formulae-as-Types Notion of Control, Conference Record of 17th ACM Symp. on Principles of Programming Languages, pp. 47-58, 1990.
11. Gunter, C. A., D. Remy, and J. G. Riecke: A Generalization of Exceptions and Control in ML-Like Languages, Functional Programming and Computer Architecture, pp. 12-23, 1995.
12. Harper, R., B. Duba, and D. Macqueen: Typing First-Class Continuations in ML, J. Functional Programming 3(4), pp. 465-484, 1993.
13. Hieb, R., R. Dybvig, and C. W. Anderson: Subcontinuations, Lisp and Symbolic Computation 6, pp. 453–484, 1993.
14. Kameyama, Y. and M. Sato: Strong Normalizability of the Non-deterministic Catch/Throw Calculii, Theoretical Computer Science, to appear.
15. Kameyama, Y.: A Type System for Delimited Continuations (Preliminary Version), Workshop on Programming and Programming Languages (PPL2000), Japan Society for Software Science and Technology, Kanzanji, Japan, Mar. 2000.
16. Ong, C.-H. L. and C. A. Stewart: A Curry-Howard Foundation for Functional Computation, Proc. 24th ACM Symposium on Principles of Programming Languages, pp. 215–227, 1997.
17. Parigot, M.: $\lambda\mu$-calculus: An Algorithmic Interpretation of Classical Natural Deduction, Lecture Notes in Computer Science **624**, Springer, pp. 190-201, 1992.
18. Sato, M.: Intuitionistic and Classical Natural Deduction Systems with the Catch and the Throw Rules, Theoretical Computer Science Vol. 175, No. 1, pp. 75-92, 1997.
19. Takahashi, M: Parallel Reductions in λ-Calculus, J. Symbolic Computation **7**, 1989, pp. 113–123.
20. Queinnec, C. and B. Serpette: A Dynamic Extent Control Operator for Partial Continuation, Proc. 18th ACM Symp. on Principles of Programming Languages, pp. 174-184, 1991.