Translation Style Semantics and Type System of Control Capturing

Shohei Matsumoto and Shin-ya Nishizaki

Department of Computer Science,Tokyo Institute of Technology,Tokyo, Japan

nisizaki@cs.titech.ac.jp

Keywords: component; programming language theory; functional programming; type system; control capturing; non-local jump

Abstract. Many programming languages provides non-local exit. In C language, it is implemented by setjump and longjump functions in its standard library. In Java, a try-catch-finally statement is equipped as non-local exit with control capturing. The try-catch-finally mechanism can be categorized into two parts: global jump and control capturing. The non-local jump has been studied well for a long time by many researchers. On the other hand, control capturing has not yet been researched well enough. In this paper, we propose a lambda calculus with non-local jump and control capturing and its operational semantics based on small-step transition. We provide continuation-passing style translation of the calculus into the usual lambda calculus. The continuation-passing style translation is known as a translation style semantics of a control structure such as a non-local jump or first-class continuation. We extend the continuation passing style translation in order to formalize the control capturing in the framework of a functional programming language paradigm. We develop a type system for the calculus with control capturing and show conformity of the typed version of the CPS translation with the type system of the calculus.

Introduction

We begin this paper with several related backgrounds.

A. Continuation and Translation Style Semantics

Continuation represents the rest of computation at the point of time of executing a program. In actual implementations of programming languages, the current continuation corresponds to a memory image of a control stack. We have a programming style called the continuation-passing style, in which each function takes a continuation explicitly as one of the arguments. In programming language Scheme[1], a factorial function is described as follows in the non-continuation-passing style.

```
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

In the continuation-passing style, we can write the factorial function as follows.

```
(define (fact-cps n k)
  (if (= n 0) (k 1)
    (fact-cps (- n 1)
       (lambda (i) (* n (k i))))))
 (define (factorial n)
(fact-cps n (lambda (j) j)))
```

As seen in this example, the continuation-passing style enables us to rewrite a recursively-defined function as a tail-recursive one. In lambda calculus, continuation-passing style translation [2] is

known as a method for enforcing an evaluation strategy on evaluation. The following is a continuation-passing style translation for a call-by-value evaluation strategy [3].

$$\begin{aligned}[\![x]\!] &= \lambda k.kx \\ [\![\lambda x.M]\!] &= \lambda k.k(\lambda x.[\![M]\!]) \\ [\![(MN)]\!] &= \lambda k.[\![M]\!](\lambda m.[\![N]\!](\lambda n.(k(mn))))\end{aligned}$$

This translation preserves the meaning of an expression with respect to the call-by-value evaluation, that is,

$$M \xrightarrow{*}_{cbv} V \text{ if and only if } [\![M]\!](\lambda v.v) \overset{*}{\leftrightarrow} V$$

The continuation-passing style translation respects the simple type system of the lambda calculus. If you define a translation Φ[| α |] of a type A of values as

$$\begin{aligned}\Phi[\![\alpha]\!] &= \alpha \\ \Phi[\![(A \to B)]\!] &= \Phi[\![A]\!] \to \neg(\neg\Phi[\![B]\!])\end{aligned}$$

where ¬A is defined as (A → ϕ) → ϕ for a fixed type ϕ called a top-level type [4][5], a translation [| A |] of a type A of terms is defined as

$$[\![A]\!] = \neg(\neg\Phi[\![A]\!])$$

We have a theorem for the CPS and its type translation:

If a term *M* is of type *A*, then its CPS-translation result [| *M* |] is of type [| *A* |].

The Curry-Howard isomorphism is known as correspondence between the intuitionistic logic and the programming language. From the Curry-Howard isomorphism, the typed lambda calculus with first-class continuations corresponds to the classical logic [4] [5] [6] and the typed lambda calculus with non-local jump corresponds to the intuitionistic logic [7].

B. *Global Exit and Control Capturing*

A non-local jump leap-frogs the flow of execution over the current context and resumes at another control point pre-declared in advance. In Common Lisp [8], catch and throw forms are provided as dynamic non-local exits.

```
(catch 'exit
(let ((port (open "/tmp/data.csv")))
  (unwind-protect (read-csv port)
                  (close port))
  (close port)))


(defun read-csv (file-handle)
...
   (if *error-occurring*
(throw 'exit))
...)
```

If the current control of the program goes out of normal processing and a file handler is already open, then the file handler should be closed after exiting. The variable port is bound to a file handler

and the procedure read-csv is assumed to read CSV data from the file. If an error such as malformed CSV occurs, the control flow jumps out of the read-scv block to the catch point tagged with the exit tab. As seen in this example, Common Lisp [8] provides a control capturing mechanism with an unwind-protect form. If there is a non-local jump across the unwind-protect, its second argument (close port) is evaluated.

C. *Purpose of this paper*

The purpose of this paper is to study the control capturing mechanism from the viewpoint of the lambda calculus. We firstly propose a lambda calculus with non-local jump abort and control capturing unwind-protect. Secondly, we define the operational semantics of the calculus as a call-by-value reduction using evaluation contexts. Thirdly, we introduce a simple type system for the calculus. Fourthly, we give a continuation-passing style translation of the calculus into the usual lambda calculus, extending the existing continuation passing style translation for the lambda calculus based on call-by-value evaluation. We also mention that the type system respects the CPS translation.

Lambda Calculus with Control Capturing

In this section, we define a lambda calculus with control capturing, λuw. The calculus λuw is an extension of the lambda calculus created by adding a global exit construct $\abort$ and a control capturing construct **unwind-protect**.

In advance of the definition of values and terms, we are assumed to be given a countably-infinite set of variables. Elements of the set are represented by the symbols *x,y,z,…*

Definition 1 (Terms and Values of λuw*):*

We define terms *M* and values *V,W* of λuw inductively by the following grammar.

$$V := x \mid \lambda x.M$$

An expression λ*x.M* is called a lambda abstraction. This is a constructor of a function, which is represented as (lambda (x) M) in Lisp.

$M ::=$	term
x	variable
$\mid V$	value
$\mid (MN)$	function application
$\mid$ (**abort** M)	global exist
$\mid$ (**unwind-protect** M N)	unwind-protect

The expression (*M N*) is called a function application, in which a function *M* is applied to an actual parameter *N*.

We call the expression (**abort** *M*) a global exit. This provides a non-local jump mechanism similar to exit in C language.

The expression (**unwind-protect** *M N)* is called an unwind-protect. This captures control flow jumping globally caused by **abort** in *M* and causes *N* to be executed.

The expression (**uwp** *M V*) is an intermediate form in evaluating an unwind-protect expression. In evaluating (**unwind-protect** *M N), N* is firstly evaluated to a unary function *V* and then *M* is evaluated as a part of an expression (**uwp** *M V*). This process of evaluation will be defined formally as an operational semantics later.

Next, we give an operational semantics based on a call-by-value evaluation strategy to the λuw-calculus. In its preparation, we define an evaluation context, which designates a sub-expression to be evaluated in the call-by-value evaluation strategy.

Definition 2(Evaluation Contexts of λuw*):* We define the evaluation contexts *E*[ ] of λuw and those not including unwinding-protect *F*[ ], inductively by the grammar.

$$E[\,] ::= F[\,] \mid F[(\textbf{uwp}\ E[\,]\ V)]$$

$$\begin{aligned} F[\,] ::= & \\ & [\,] \\ & \mid (F[\,]\ M) \\ & \mid (V\ F[\,]) \\ & \mid (\textbf{abort}\ F[\,]) \\ & \mid (\textbf{uwp}\ M\ V) \\ & \mid (\textbf{unwind-protect}\ M\ F[\,]) \end{aligned}$$

$$E[(\lambda x.M)\ V] \to E[M[x := V]] \qquad \text{E-Beta}$$

$$E[(\textbf{unwind-protect}\ M\ V)] \to E[(\textbf{uwp}\ M\ V)] \qquad \text{E-Set-Unwind}$$

$$E[(\textbf{uwp}\ V\ W)] \to E[(W\ V)] \qquad \text{E-Uwp-Normal}$$

$$E[(\textbf{unwind-protect}\ F[(\textbf{abort}\ V)]\ W)] \to E[(\textbf{abort}\ (W\ V))] \qquad \text{E-Uwp-Abort}$$

$$F[(\textbf{abort}\ V)] \to V \qquad \text{E-Uncaught-Abort}$$

Fig. 1. Reduction rules of call-by-value reduction

An evaluation context designates the sub-expression to be evaluated in a given expression. In the above definition, the evaluation contexts are categorized into two groups, evaluation contexts without the unwinding-mark **uwp**, *F*[ ], and those with **uwp**, *E*[ ].

(*F*[ ] *M*) means that the function part of a function application is evaluated firstly and (*V F*[ ]) means that the argument part is evaluated after the evaluation of the function part. Similarly,

(**unwind-protect** *M F*[ ]) means that the second argument is evaluated before evaluation of the first argument. This syntactical structure makes the call-by-value evaluation possible.

Next, we give an operational semantics to the λuw-calculus as a call-by-value reduction in the style of Plotkin [3].

Definition 3 (Reduction of λuw*):* We give a *call-by-value reduction* to the λuw-calculus, as a binary relation between terms, by the rules in Figure 3.

Rule E-Beta is beta-reduction under the call-by-value evaluation strategy. In the redex ((λ*x.M) V*), the actual parameter is restricted to a value by designated as *V*, which means that the formal parameter *x* is bound to the actual parameter after evaluating the parameter and obtaining its result.

If **abort** is called in evaluating the first parameter of **unwind-protect**, rule E-Uwp-Abort is applied: the abort expression (**abort** *V*) involves the second parameter *W* of **uwp** as (**abort** (*W V*)).

If the abort expression (**abort** *V*) reaches an evaluation context *F*[ ] without an unwinding-mark, then rule E-Uncaught-Abort is applied, and **abort** is eliminated and the body *v* is passed to the context *F*[ ].

If (**unwind-protect** *M N*) is found, *N* is evaluated and then *M* is evaluated, which is determined by the evaluation context.

If the second argument is evaluated and implies a term (**unwind-protect** *M V*), then the mark **uwp** is added and the first argument *M* starts to be evaluated as (**uwp** *M V*).

Example 1 (Reduction Sequence with **abort***):* First, we give an example of a reduction sequence in which **abort** appears and **unwind-protect** does not.

$$\begin{aligned} & \Big(\lambda x.\big(\textbf{abort}\ (\lambda y.y)\big)x\Big)c \\ \to & \big(\textbf{abort}\ (\lambda y.y)\big)c \\ \to & \lambda y.y \end{aligned}$$

Example 2 (Reduction Sequence with **unwind-protect***):*

$$
\begin{aligned}
&\Big(\textbf{unwind-protect}\ \big((\lambda x.(\textbf{abort}\ d)x)c\big)(\lambda x.fxx)\Big)\\
\rightarrow&\Big(\textbf{uwp}\ \big((\lambda x.(\textbf{abort}\ d)x)c\big)(\lambda x.fxx)\Big)\\
\rightarrow&\Big(\textbf{uwp}\ \big((\textbf{abort}\ d)c\big)\ (\lambda x.fxx)\Big)\\
\rightarrow&\Big(\textbf{abort}\ \big((\lambda x.fxx)d\big)\Big)\\
\rightarrow&\big(\textbf{abort}\ (f\ d\ d)\big)\\
\rightarrow&(f\ d\ d)
\end{aligned}
$$

Type System for Control Capturing

In this section, we introduce a simple type system into the calculus λuw of control capturing. First, we define the type of the type system.

Definition 4 (Type of λuw*):* We define the type of λuw inductively by the following grammar.

$$A ::= \alpha \mid (A \rightarrow B)$$

Symbol α represents a primitive type, given in advance to the definition. The type ($A \rightarrow B$) is called a function type; its domain type is A and co-domain is B.

A type assignment, defined in the following, gives information on free variables for the term to be typed.

*Definition 5 (Type Assignment):*A type assignment is a mapping of variables to types whose domain is finite. We write a type assignment as a finite set of pairs as

$$\{x_1 : A_1, \ldots, x_n : A_n\}.$$

We use symbols Γ, Δ, … for type assignments.

$$
\begin{array}{rcl}
\Phi[\![x]\!] & = & x, \\
\Phi[\![\lambda x.M]\!] & = & \lambda x.[\![M]\!], \\
 & & \\
[\![V]\!] & = & \lambda kd.(k\ \Psi[\![V]\!]), \\
[\![(M\ N)]\!] & = & \lambda kd.[\![M]\!](\lambda m.[\![N]\!](\lambda n.mnkd)d)d, \\
[\![(\textbf{abort}\ M)]\!] & = & \lambda kd.[\![M]\!]dd \\
[\![(\textbf{unwind-protect}\ M\ N)]\!] & = & \lambda kd.[\![N]\!]\Big(\lambda n.\big([\![M]\!](\lambda m.nmkd)(\lambda x.nxdd)\big)\Big)d \\
[\![(\textbf{uwp}\ M\ V)]\!] & = & \lambda kd.[\![M]\!](\lambda m.\Phi[\![V]\!]mkd)(\lambda x.\Phi[\![V]\!]xdd)
\end{array}
$$

Fig. 2. CPS Translation

$$
\begin{array}{l}
(M\ N) : K, D = M : \ (\lambda m.[\![N]\!](\lambda n.\ m\ n\ K\ D)D),\ D \\
(V\ N) : K, D = N : \ (\lambda n.\Psi[\![V]\!]\ n\ K\ D),\ D \\
(V\ W) : K, D = \Psi(V)\Psi(W)KD \\
 \\
(\textbf{abort}\ M) : K, D = M\ :\ D\ ,\ D \\
 \\
(\textbf{unwind-protect}\ M\ N) : K, D = N : (\lambda n.[\![M]\!](\lambda m.m(\lambda x.nxKD)(\lambda x.nxDD))) \\
(\textbf{unwind-protect}\ M\ V) : K, D = M : (\lambda m.m(\lambda x.\Phi[\![V]\!]xKD)(\lambda x.\Phi[\![V]\!]xDD)), \\
(\textbf{unwind-protect}\ V\ W) : K, D = \Phi[\![V]\!](\lambda x.\Phi[\![W]\!]xKD)(\lambda x.\Phi[\![W]\!]xDD).
\end{array}
$$

Fig. 3. Colon Translation

$$
\begin{array}{rcl}
\Phi[\![x^A]\!]^{\Phi[\![A]\!]} & = & x^{\Phi[\![A]\!]}, \\
\Phi[\![(\lambda x.M)^{A\to B}]\!]^{\Phi[\![A\to B]\!]} & = & \lambda x^{\Phi[\![A]\!]}.[\![M]\!]^{[\![B]\!]}, \\
 & & \\
[\![V^A]\!]^{[\![A]\!]} & = & \lambda k^{\Phi[\![A]\!]\to\bot} d^{\bot\to\bot}.(k\ \Psi[\![V^A]\!]^{\Phi[\![A]\!]}), \\
[\![(M^{A\to B}\ N^A)]\!]^{[\![B]\!]} & = & \lambda k^{\Phi[\![B]\!]\to\bot} d^{\bot\to\bot}. \\
 & & [\![M]\!]^{((\Phi[\![A]\!]\to[\![B]\!])\to\bot)\to(\bot\to\bot)\to\bot}\Big(\lambda m^{\Phi[\![A]\!]\to[\![B]\!]}. \\
 & & [\![N]\!]^{[\![A]\!]}\big(\lambda n^{\Phi[\![A]\!]}.(mn)^{[\![B]\!]}kd\big)^{\Phi[\![A]\!]\to\bot}d\Big)d, \\
[\![(\textbf{abort}\ M^\phi)^B]\!]^{(\Phi[\![B]\!]\to\bot)\to(\bot\to\bot)\to\bot} & = & \lambda k^{\Phi[\![B]\!]\to\bot} d^{\bot\to\bot}.[\![M]\!]^{(\bot\to\bot)\to(\bot\to\bot)\to\bot}dd \\
[\![(\textbf{unwind-protect}\ M^\phi\ N^{\phi\to\phi})^\phi]\!]^{(\bot\to\bot)\to(\bot\to\bot)\to\bot} & = & \lambda k^{\bot\to\bot} d^{\bot\to\bot}. \\
 & & [\![N]\!]^{((\Phi[\![\phi]\!]\to[\![\phi]\!])\to\bot)\to(\bot\to\bot)\to\bot}\Big(\lambda n^{\Phi[\![\phi]\!]\to[\![\phi]\!]}.(\\
 & & [\![M]\!]^{(\Phi[\![\phi]\!]\to\bot)\to(\bot\to\bot)\to\bot}(\lambda m^{\Phi[\![\phi]\!]}.nmkd) \\
 & & (\lambda x^\bot.nxdd)\Big)d \\
[\![(\textbf{uwp}\ M^\phi\ V^{\phi\to\phi})^\phi]\!]^{(\bot\to\bot)\to(\bot\to\bot)\to\bot} & = & \lambda k^{\bot\to\bot} d^{\bot\to\bot}. \\
 & & [\![M^{\Phi[\![\phi]\!]\to\bot}]\!](\lambda m^{\Phi[\![\phi]\!]}. \\
 & & \Phi[\![V]\!]^{\Phi[\![\phi]\!]\to[\![\phi]\!]}mkd)(\lambda x^\bot.\Phi[\![V]\!]xdd)
\end{array}
$$

where $\bot$ means $\Phi[\![\phi]\!]$ and $[\![\phi]\!]$ equals to $\Phi[\![\phi]\!]\to(\bot\to\bot)\to\bot$.

Fig. 4. Typed Annotated CPS Translation

Definition 6 (Typing Rules): Type judgment $\Gamma \vdash M : A$ shows a ternary relation among a type assignment Γ, a term *M*, and a type *A* inductively defined by the following rules.

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}$$

$$\frac{\{x : A\}\Gamma \vdash M : B}{\Gamma \vdash \lambda x.M : (A \to B)}$$

$$\frac{\Gamma \vdash M : (A \to B) \quad \Gamma \vdash N : A}{\Gamma \vdash (M\ N) : B}$$

$$\frac{\Gamma \vdash M : \phi \quad \Gamma \vdash N : (\phi \to \phi)}{\Gamma \vdash (\textbf{unwind-protect}\ M\ N) : \phi}$$

$$\frac{\Gamma \vdash M : \phi}{\Gamma \vdash (\textbf{abort}\ M) : A}$$

In the above rules, ϕ is the type of a given term to be defined, similarly to the one in studies [4] [5] corresponding to the absurdity $\perp$ from the viewpoint of the Curry-Howard isomorphism. This type is often called the *top-level type*.

Continuation-Passing Style Translation

Continuation-passing style translation, abbreviated to CPS translation, is a form of mapping between the lambda calculi. The translation keeps meaning to a given source expression based on an evaluation strategy such as call-by-value evaluation. The CPS translation consists of translations of values and terms.

Definition 7 (Continuation-Passing Style Translation): Translations of values $\Phi[|\ V\ |]$ and terms $[|\ M\ |]$ are defined mutual-recursively by the equations in Fig. 2.

Using this definition of CPS translation, a result of translation includes many redundant beta-redices. In order to control such redices, we improve the CPS translation as follows.

*Definition 8 (Colon Translation):*Colon translation $(M : K, D)$ of a term M of λuw and terms K and D of the lambda calculus is defined inductively by the equations in Fig. 3.

The continuation-passing style translation respects the type system of λuw. Translations of value types and term types are defined as follows.

Definition 9 (Type Translation): Translations $\Phi[|\ A\ |]$ and $[|\ A\ |]$ of types are defined inductively by the following equations.

$$\Phi[\![\alpha]\!] = \alpha$$
$$\Phi[\![(A \to B)]\!] = \Phi[\![A]\!] \to [\![B]\!]$$

$$[\![A]\!] = (\Phi[\![A]\!] \to \Phi[\![\phi]\!]) \to (\Phi[\![\phi]\!] \to \Phi[\![\phi]\!]) \to \Phi[\![\phi]\!]$$

where ϕ is the top-level type.

In the definition of $[|\ A\ |]$, $(\Phi[|\ A\ |] \to [|\ \phi\ |])$ is a type of a continuation and $\Phi[|\ \phi\ |] \to \Phi[|\ \phi\ |]$ a type of a function of the closing process after control-capturing.

The CPS translation respects the typing that is known from the type annotation in Figure 4, which gives us the following theorem.

Theorem 1 (Typing of CPS Translation): If

$$\{x_1 : A_1, \ldots, x_n : A_n\} \vdash M : A$$

then we have

$$\{x_1 : \Phi[\![A_1]\!], \ldots, x_n : \Phi[\![A_n]\!]\} \vdash [\![M]\!] : [\![A]\!].$$

Conclusion

We have proposed an extended lambda calculus λuw for control capturing of non-local jumps. The operational semantics of λuw is defined as a call-by-value reduction using evaluation contexts.

We give a simple type system to λuw in Curry's style.

The continuation-passing style translation is given to λuw,

which maps types and terms of λuw to those of the usual simply-typed lambda calculus. We show that the type system respects the CPS translation.

In this paper, we do not mention non-local jumps with control capturing under imperative computation, found in practical programming languages such as Java and Scheme. In future, we should study control capturing under imperative computation. A promising research direction is extension of Gifford's effect system [10] [11] [12].

The operating semantics defined in this paper is a small-step semantics based on Plotkin's [3]. In our previous papers [13] [14], we developed an operational semantics using abstract machines. We can provide another view by studying such an operational semantics.

Acknowledgment

This work was supported by Grants-in-Aid for Scientific Research (C) (24500009).

References

[1] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten, Eds., *Revised [6] Report on the Algorithmic Language Scheme*. Cambridge University Press, 2010.

[2] M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. F. Duba, "A syntactic theory of sequential control," *Theoretical Computer Science*, vol. 52, no. 3, 1987.

[3] G. Plotkin, "Call-by-name, call-by-value, and the λ-calculus," *Theoretical Computer Science*, vol. 1, pp. 125–159, 1975.

[4] C. Murthy, "An evaluation semantics for classical proofs," in Proc. *5th IEEE Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1991.

[5] T. G. Griffin, "A formulae-as-types notion of control," in *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, 1990.

[6] S. Nishizaki, "Programs with continuations and linear logic," *Science of Computer Programming*, vol. 21, no. 2, pp. 95–116, 1994.

[7] H. Nakano, "A constructive formalization of the catch and throw mechanism," in *Proceedings of the Symposium on Logic in*

Computer Science. IEEE Computer Society Press, 1992.

[8] G. L. Steele, *Common Lisp the Language*, 2nd Edition. Digital Press, 1990

[9] S. Matsumoto and S. Nishizaki, "Continuation-passing style translation of non-local jumps and control capturing," in *Proceedings of the 2015 Tbilisi International Conference on Computer Sciences and Applied Mathematics (TICCSAM 2015)*, 2015, to appear.

[10] J. M. Lucassen and D. K. Gifford, "Polymorphic effect systems," in *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, 1988, pp. 47–57.

[11] P. Jouvelot and D. K. Gifford, "Reasoning about continuations with control effects," in *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, 1989, pp. 218–226.

[12] P. Wadler and P. Thiemann, "The marriage of effects and monads," *ACM Transactions on Computation Logic*, vol. 4, no. 1, pp. 1–32, 2003.

[13] K. Narita, S. Nishizaki, and T. Mizuno, "A simple abstract machine for functional first-class continuations," in *Proceedings of the 2010 Tenth International Symposium on Communications and Information Technologies (ISCIT)*. IEEE, 2010, pp. 111–114.

[14] K. Narita and S. Nishizaki, "A Parallel Abstract Machine for the RPC Calculus," in *Proceedings of the International Conference on Informatics Engineering and Information Science – ICIEIS 2011*, ser. Communications in Computer and Information Science, vol. 253. Springer, 2011, pp. 320–332.