# Portable Implementation of Continuation Operators in Imperative Languages by Exception Handling

Tatsurou Sekiguchi[1][2], Takahiro Sakamoto[1], and Akinori Yonezawa[1]

[1] Department of Information Science, Faculty of Science, University of Tokyo
[2] PRESTO, Japan Science and Technology Corporation
{cocoa, takas, yonezawa}@is.s.u-tokyo.ac.jp

**Abstract.** This paper describes a scheme of manipulating (partial) continuations in imperative languages such as Java and C++ in a portable manner, where the *portability* means that this scheme does not depend on structure of the native stack frame nor implementation of virtual machines and runtime systems. Exception handling plays a significant role in this scheme to reduce overheads. The scheme is based on program transformation, but in contrast to CPS transformation, our scheme preserves the call graph of the original program. This scheme has two important applications: *transparent migration* in mobile computation and *checkpointing* in a highly reliable system. The former technology enables running computations to move to a remote computer, while the latter one enables running computations to be saved into storages.

## 1 Introduction

A situation often occurs that execution states of a running program have to be encoded into a data structure. Checkpointing [6] is a technique that improves reliability of a system by saving execution states of a running program periodically. In the context of mobile computation [2], there is a form of computation migration called *transparent migration* [13] or *strong mobility* [4], which means that the entire execution state of a running program including the call stack and (part of) the heap image are preserved on migration. During a transparent migration process, the execution states of a program are saved into a data structure, the data structure is transmitted to a destination host over the network, and finally execution states are reconstructed at that host from the data structure.

It is not difficult to capture and recover the call stack when a program language has `call/cc` (call with current continuation) primitive. Even in imperative languages, the capability of storing execution states including the call stack has often been implemented by compiler support and/or by runtime support. EmacsLisp, SmallTalk and some implementation of Standard ML (SML/NJ) have a primitive that dumps the execution image into a local disk.

This paper reports a completely different approach based on program transformation. This scheme is portable in the sense that it does not depend on structure of native stack frames nor implementation of virtual machines and runtime systems. An existing system therefore does not need to be extended to capture and restore execution states. This scheme has been developed mainly through the study on implementation of transparent migration on Java because Java allows Java programs to manipulate stack frames only in a restricted form. Exception handling plays a significant role in this scheme to reduce overheads.

The rest of this paper is organized as follows. Sect. 2 introduces operators manipulating continuations to clarify what is implemented by the scheme described in this paper. Sect. 3 describes how continuation operators are implemented by using an exception handling mechanism. Sect. 4 shows several applications. Sect. 5 discusses difficulties and limitations in the technique. In Sect. 6, we compare our technique with related work. Sect. 7 summarizes this work.

## 2 Partial Continuations

This section introduces a simple calculus of control [14] that was devised through development of SML/NJ. This calculus provides two *tagged* operators for manipulating partial continuations, which are almost analogous to Danvy's `shift` and `reset` [5], and Felleisen's $\mathcal{F}$ and `prompt` [9]. Our program transformation scheme essentially implements those operators for partial continuations in imperative languages.

This calculus is an extension of a call-by-value lambda calculus, and its semantics is defined in the style of structured operational semantics. A polymorphic type system is provided for the calculus. In this paper, however, we focus only on its operators for partial continuations. The syntax of these operators are defined as follows:

| | |
|---|---|
| `cupto` $p$ `as` $x$ `in` $e$ | capturing the functional continuation |
| `set` $p$ `in` $e$ | delimiting the effect of `cupto` |

where $p$, $e$ and $x$ are metavariables that denote a *prompt*, an *expression* and a *variable*, respectively. [1] A prompt is a special constant, which is actually a tag that determines which `set` and `cupto` expressions are matched. The `cupto` operator captures the functional continuation up to the innermost `set` expression with the same prompt. The captured continuation is bound to variable $x$, and expression $e$ is evaluated with this extended environment. The `set` expression delimits the effect of capturing a continuation. The outer context of the `set` operator is thus not captured. In contrast, `call/cc` (call with current continuation) operator in Scheme always captures the full continuation.

The evaluation rule is defined formally as follows:

$$\texttt{set}\ p\ \texttt{in}\ E[\texttt{cupto}\ p\ \texttt{as}\ x\ \texttt{in}\ e] \longrightarrow (\lambda x.e)(\lambda y.E[y])$$

where $E$ is an evaluation context in which the hole is not in the scope of a `set` expression with prompt $p$, and $y$ is a fresh variable. An evaluation context [10] is

[1] Strictly speaking, there is a slight difference from the original operators. The set of valid expressions are restricted for our convenience.

an expression with a single special constant called *hole*. The position of the hole in an evaluation context is syntactically defined so that it designates the expression to be evaluated next, i.e. the redex of the evaluation context. We denote by $E[e]$ the result of replacing the hole in $E$ with $e$, which implies that $e$ is the redex of expression $E[e]$. In this evaluation rule, $E$ represents the continuation of `cupto` expression up to the `set` expression. The continuation becomes $\lambda y.E[y]$ by the eta conversion. In the righthand side, $e$ will be evaluated with an environment where the continuation is bound to $x$.

Various control operators can be implemented by composing these primitive control operators. Ref. [14] provides examples of implementation of `call/cc`, exception handling, and coroutines.

The notion of partial continuations was partly implemented in early programming languages such as PL/I and Mesa [17], where execution can restart at the instruction following the one that raised an exception when the exception is captured by an exception handler. In addition, a model of exception handling is proposed [8] in which resumption contexts are first class objects and are modifiable. Our scheme can be used to implement such an exception handling mechanism on imperative languages in a portable manner.

## 3 Emulating Continuations

This section describes how the control operators in Sect. 2 are implemented in imperative languages such as C++ and Java. Our scheme of implementation does not need to manipulate stack frames, but the overheads are quite low due to exception handling mechanism. Our scheme is based on program transformation. In contrast to the CPS transformation, however, a code transformed by our scheme preserves the original call graph (although additional method invocations are inserted to maintain continuation operation). A CPS transformed program easily overflows the call stack if the base language is a typical imperative language. In this section, we suppose the target language to be Java bytecode because it is suitable for explaining the idea of our program transformation. Java virtual machine forbids a stack manipulation by Java bytecode itself because of concern for security. A program in Java bytecode cannot inspect nor modify stack frames. These facts obviously show that the scheme is portable and widely applicable to various imperative languages.

The transformer takes a program in an imperative language with the continuation operators as input, and produces an equivalent program in the base language. Moreover, the transformation is on per-method basis, i.e. from a method in the source program, a method and a class are created. This created class represents the execution states of the method. A program is transformed so that it explicitly manages its execution states. A captured continuation is a standard data structure in the base language, which implies that one can save and modify it, moreover, it is transmittable to a remote host. Though the transformation is implemented by using an exception handling mechanism, it does not prevent use of exception handling in a source program.

The transformation consists of two different sub-transformations: one for saving execution states and a resumption point, and one for restoring execution states. The transformation for saving execution states is described in Sect. 3.3, while the one for restoring is described in Sect. 3.4. Actually, the effects of two transformations are mingled in a transformed code.

Since a program is transformed and additional fragments are inserted to the original program, it incurs slowdown of *ordinary* execution (note that the part *not* relating to continuation operation also slows down). In addition, our scheme changes method signatures. An extra parameter is inserted to each method to pass a state object (this will be explained in Sect. 3.4).

```
public class Fib {
    public static void fib( int v1 ) {
        if ( v1 <= 1 )
            return 1;
        else {
            int v2 = fib (v1 - 2);
            return v2 + fib (v1 - 1);
        }
    }
}
```

**Fig. 1.** A pseudo code of Fibonacci function.

We use Fibonacci function in Fig. 1 to illustrate the transformation throughout this section. For readability, we use a Java-like pseudo code to denote a program, but in reality it consists of Java bytecodes.

## 3.1 Bytecode Analysis

To transform a bytecode program, we need information on a set of all *valid* frame variables (a kind of registers) and entries in the operand stack for each program point. A variable or an entry is valid if a value on it is available for every possible control flow. Types of frame variables and entries in the operand stack are also necessary. In addition, a transformed code must pass a Java bytecode verifier if the original code passes it. To obtain such necessary information on bytecode, bytecode analysis must be performed before transformation.

Our bytecode transformer requires exactly the same information as that for bytecode verification [15]. We had adopted type systems for Java bytecode verification to keep information on bytecode. Our transformer transforms bytecode programs based on this information. We use the type system of Stata and Abadi [22, 23], and that of Freund and Mitchell [11]. Stata's type system provides information on types of frame variables and the operand stack. In addition, bytecode subroutines can be described. On the other hand, Freund's type system focuses on uninitialized values, that are fresh objects whose constructors are not invoked yet. An uninitialized value exists only for a brief period in ordinary execution since a constructor of an object is always invoked as soon as it is created, but it can be a source of type system violation [11].

If a bytecode is well-typed in their type systems, it tells that the bytecode program is verifiable. The type reconstruction problem is to find an appropriate type judgment for a given program (method). It is actually a verification algorithm itself. We have implemented a type reconstruction algorithm for the type systems although we had to extend them to the full set of Java bytecode except bytecode subroutine facility. As will be mentioned in Sect. 5, it is difficult to transform a bytecode subroutine into an efficient code. Bytecode subroutines are not supported in our current implementation.

```
public class ST_Fib_fib extends StateObject {
    public int      EntryPoint;
    public int[]    ArrayI;
    public long[]   ArrayL;
    public float[]  ArrayF;
    public double[] ArrayD;
    public Object[] ArrayA;

    public void     Resume() {
        Fib.fib( this, 0 );
    }
}
```

**Fig. 2.** A state class.

## 3.2 State Class

Our transformation algorithm defines a state class for each method. An execution state of a method is stored into an instance of the state class. Fig. 2 shows an example of a state class, where `EntryPoint` designates a resumption point, and variables of array types keep frame variable values and operand stack values. In addition, special values that manage state capture and restoration are also stored into those arrays. These special values include the state object for the current method, the state object for the caller of the current method, and a special exception that notifies migration. The size of each array is determined statically when a method is analyzed. Every state class is a subclass of a common super class (`StateObject`). Every state class has method `Resume`, which resumes the execution stored in a state object. This will be explained in Sect. 3.4.

## 3.3 Capturing a Continuation

Capturing a continuation consists of the following operations:

1. Saving all frame variable values and operand stack values in a method,
2. Saving a resumption point information in a method, and
3. Repeating the above for each method.

These operations essentially yield a logical copy of the stack. In case of the Java bytecode language, frame variables include all parameters of a method, and a resumption point is actually a program counter. In case of C++ and Java source-to-source transformation, operand stack values are not saved. Instead, temporary variables are introduced. When we have to save an execution state of a *partially* evaluated expression, that expression is split so that intermediate values can be saved. Consider the following piece of code:

```
x = foo() + bar();
```

To save the result of `foo`, the above expression is split in advance as follows:

```
tmp = foo();
x = tmp + bar();
```

The transformation algorithm inserts the following code fragments to a method:

– An exception handler for each method invocation. An occurrence of state capturing is notified by a special exception. The exception handler is responsible for saving an execution state. The program counter to be saved is known since an exception handler is unique for each resumption point. The set of valid frame variables and their types are found by the bytecode analysis described in Sect. 3.1.
– Instructions for saving valid entries on the operand stack into frame variables. Entries on the operand stack are defined to be discarded when an exception is thrown, which means that their values cannot be fetched from an exception handler. The basic idea for saving values on the operand stack is to make their copies in frame variables before the contents of entries on the operand stack are set. The valid entries on the operand stack are also found by the bytecode analysis. This care is needed only in case of Java bytecode.

When a continuation is captured by invoking a `cupto` operator, a special exception is thrown. If a method captures the exception, the method stores its execution state in a newly created state object defined for each method, and then it propagates the exception to the caller of the method. This process is repeated until the exception reaches a `set` operator with the same prompt.

Fig. 3 shows a result of transforming the method in Fig. 1 for state capturing. An exception handler that captures exception `Notify` is inserted for each method invocation. In the exception handlers, a resumption point and local variables are saved into a created state object. Since variable `v2` is undefined at the first recursive invocation of method `fib`, The value of `v2` is not saved in the first exception handler. The state object is stored in the exception object by `e.append (s)`. Finally, the exception is re-thrown.

```
public static void fib( int v1 ) throws Notify {
    if ( v1 <= 1 )
        return 1;
    else {
        int v2;
        try {
            v2 = fib (v1 - 2);
        } catch ( Notify e ) {
            ST_Fib_fib s = new ST_Fib_fib();
            s.EntryPoint = 1;
            s.v1 = v1;
            e.append (s);
            throw e;
        }
        try {
            return v2 + fib (v1 - 1);
        } catch ( Notify e ) {
            ST_Fib_fib s = new ST_Fib_fib();
            s.EntryPoint = 2;
            s.v1 = v1;
            s.v2 = v2;
            e.append (s);
            throw e;
        }
    }
}
```

**Fig. 3.** A pseudo code transformed for state capturing.

### 3.4 Invoking a Continuation

Invoking a continuation consists of the following operations:

1. Restoring all frame variable values and operand stack values in a method,
2. Transferring the control to the resumption point in a method,
3. Reconstructing dynamic extents of active exception handlers, and
4. Reconstructing the call stack.

The execution states of a method are reconstructed from a state object. The call stack is reconstructed by calling the methods in the order in which they were invoked. Each method is transformed in advance so that it can restore its execution state from a state object. When a method is called with a state object as an extra parameter, it restores all the values of frame variables and the operand stack, and then it continues execution from the resumption point. When the extra parameter for a method is null, it indicates ordinary execution.

The transformation algorithm inserts the following code fragments to a method:

– Instructions that put a state object as an extra parameter for a method invocation instruction.
– Instructions, at the head of the method, that restore all valid frame variables and all valid entries on the operand stack. When the execution state of a method is restored, a state object is passed to the method as an extra parameter. The inserted code restores all valid frame variables and all entries on the operand stack at the resumption point. After restoring the frame variables and entries on the operand stack, the control is transferred to the resumption point.

Fig. 4 shows a result of transforming the method in Fig. 1 for state restoration. A parameter is added to pass a state object. When the extra parameter is null, the original body of the method is executed. Otherwise, the execution state is reconstructed from the state object. Variable `c` holds the state object of this method. It has a valid value only during state restoration. All the local variable values are restored from the state object, and then the control is transferred to the resumption point. Remember that this is actually a Java bytecode. We can therefore use goto instructions. A transformer for C++ can also use goto instructions. A source code transformer for Java uses another technique for control transfer [19]. Note that the scope of an exception handler is automatically recovered because restoring the execution state of a callee method is done by invoking the callee. This implies that the instruction that resumes the callee is the same instruction that invokes the callee in ordinary execution, which takes place in the scope of an exception handler.

Now we can see how `Resume` method in Fig. 2 resumes the execution. The first parameter (`this`) is a state object itself, and the second parameter is dummy since it is not used.

```
public static void fib( StateObject s, int v1 ) {
    int v2;
    ST_Fib_fib c = null;
    if ( s != null ) {
        c = (ST_Fib_fib)s.callee;
        switch (s.EntryPoint) {
            case 1:  v1 = c.v1;
                     goto l1;
            default: v1 = c.v1;
                     v2 = c.v2;
                     goto l2;
        }
    }
    if ( v1 <= 1 )
        return 1;
    else {
     l1:
        v2 = fib( c, v1 - 2 );
        c = null;
     l2:
        return v2 + fib( c, v1 - 1 );
    }
}
```

**Fig. 4.** A pseudo code transformed for state restoration.

```
public interface Receiver {
    public void Receive( StateObject s ) throws Exception;
}
```

**Fig. 5.** An interface that receives a captured continuation.

### 3.5 The `cupto` and `set` Operators

Now we can explain how the operators manipulating partial continuations in Sect. 2 are implemented by using the techniques just shown. As mentioned in Sect. 3.3, we use a special exception (`Notify`) to notify the occurrence of state capturing. We define a subclass of class `Notify` for each occurrence of a prompt in the operators for continuations. A prompt corresponds to a subclass of class `Notify`.

Since Java does not have a way to extend an environment, we modify the semantics of the `cupto` operator so that it fits in Java. The abstract syntax of the `cupto` operator is as follows:

$$\texttt{cupto}\ p\ \texttt{as}\ x\ \texttt{in}\ e$$

When the above expression is evaluated, the functional continuation up to the innermost `set` expression with the same prompt is captured and bound to variable $x$. Then expression $e$ is evaluated in the extended environment. Instead, we restrict the form of $e$ to an object that implements interface `Receiver` in Fig. 5. The interface has method `Receive`, which takes an instance of a state class. When a continuation is captured, it is passed to $e$ by invoking method `Receive` with the continuation. Variable $x$ is thus not used. In sum, the new operator looks like:

$$\texttt{cupto}\ p\ \texttt{in}\ o$$

where $o$ denotes an object that implements interface `Receiver`.

This operator is implemented just as follows:

```
throw new p(o);
```

where `p` is a class name that corresponds to prompt $p$, and `o` is a variable that refers to object $o$. Object $o$ is stored in the exception object in a constructor of `p`. This statement initiates the state capture process by throwing exception `p`.

On the other hand, the `set` operator, `set` $p$ `in` $e$, is translated as follows:

```
try {
    e
} catch ( p x ) {
    Receiver r = x.getReceiver();
    StateObject s = x.getStateObject();
    r.Receive (s);
}
```

The receiver object and all the state objects are stored in the exception object. Method `getStateObject` returns the bottom of all state objects.

Finally, invoking a continuation is achieved by calling method `Resume` (shown in Fig. 2) in a state object.

### 3.6 Experimental Results

This section reports performance results on our implementation of program transformers. We measured execution efficiency, code size growth, and elapsed time of program transformation.

| | elapsed time (ms) | | | | | |
|---|---|---|---|---|---|---|
| | with JIT | | | without JIT | | |
| program | original | JavaGo | JavaGoX | original | JavaGo | JavaGoX |
| fib(30) | 111 | 263 (+137%) | 173 (+56%) | 870 | 2553 (+193%) | 1516 (+74%) |
| qsort(400000) | 214 | 279 (+30%) | 248 (+16%) | 2072 | 2856 (+38%) | 2597 (+25%) |
| nqueen(12) | 1523 | 2348 (+54%) | 1731 (+14%) | 30473 | 36470 (+20%) | 30843 (+1.2%) |
| _201_compress | 33685 | 61629 (+83%) | 40610 (+21%) | 365661 | 713936 (+95%) | 433439 (+19%) |

(JDK 1.2.2, Intel Celeron(TM) Processor 500MHz)

| | elapsed time (sec) | |
|---|---|---|
| program | original | transformed |
| fib(40) | 40.0 | 36.1 (−10%) |
| qsort(4000) | 36.4 | 37.0 (+2%) |
| multimat | 14.0 | 15.2 (+9%) |
| bintree | 3.4 | 3.9 (+15%) |

(egcs-2.91.66, UltraSPARC Processor 168MHz)

**Table 1.** Comparison in execution efficiency.

**Execution Efficiency of Transformed Programs** Three kinds of code are evaluated: the original program, that transformed at source code level, and that transformed at bytecode level. The elapsed times of transformed programs were measured and compared. We use three transformers: JavaGo [19] as a Java source code transformer, JavaGoX [18] as a Java bytecode transformer, and a source code transformer for C++. The purpose of this experiment is to identify the overheads induced by inserted code fragments to the original programs. Captured continuations are thus not invoked during the execution of benchmark programs. The results are shown in Table 1 where _201_compress is a benchmark program included in SpecJVM98, multimat is an integer matrix multiply whose size is $200 \times 200$, and bintree is an application that inserts a random integer into a binary tree 100000 times.

Most part of the overheads in Java applications is due to the code fragments for saving the operand stack at resumption points. The overheads of the Fibonacci method is rather high because the method does almost nothing but invokes the method itself recursively. When the body of a method are so small, the relative overheads of inserted code fragments tend to be high. In this experiment, the overheads induced by our bytecode transformation are always less than those induced by JavaGo. For quick sort and N-queen programs, the overheads were approximately 15% of the original programs when the applications were executed with just-in-time compilation.

Our scheme works with C++ better than Java. The overheads due to source code transformation are less than those of Java bytecode transformation.

|  | bytecode size (in bytes) |  |  |
|---|---|---|---|
| program | original | JavaGo | JavaGoX |
| fib | 276 | 884 (3.2 times) | 891 (3.2 times) |
| qsort | 383 | 1177 (3.1 times) | 1253 (3.3 times) |
| nqueen | 393 | 1146 (2.9 times) | 976 (2.5 times) |
| _201_compress | 13895 | 22029 (1.6 times) | 18171 (1.3 times) |

**Table 2.** Comparison in bytecode size.

**Growth in Bytecode Size of Transformed Programs** The growth in bytecode size due to program transformations is shown in Table 2. The growth rates for these programs are approximately three times. We think that these results would be the worst case because the relative overheads of inserted code fragments tend to be high when an original method is small. Actually, growth rate falls down in a large application (_201_compress).

The size of bytecode produced by the bytecode transformer is very similar to the size of bytecode produced in the source code transformation. But their characteristics are quite different each other. In case of JavaGo, the size of transformed bytecode is proportional to square of the deepest depth of loops. In contrast, the size of bytecode transformed at bytecode level is proportional to the number of resumption points and valid values.

|  | elapsed time per method (ms) |  |
|---|---|---|
| program | analysis | transformation |
| fib | 235 | 79 |
| qsort | 285 | 81 |
| nqueen | 267 | 80 |
| _201_compress | 150 | 59 |

**Table 3.** Elapsed time for analysis and transformation.

**Elapsed Time of Program Transformation** The elapsed time for analysis and transformation of the bytecode transformer is shown in Table 3. In every case, analysis takes more time than transformation. However, the total elapsed time is short. We believe that these figures show our bytecode transformer is practical enough. The elapsed time for _201_compress is obviously shorter than those of the other applications. The reason is that _201_compress has many methods. The other applications are quite small one. They have only one or a few methods. In case of _201_compress, the memory cache can work effectively.

## 4 Application

We point out that our scheme for continuation manipulation based on program transformation finds at least two applications.

### 4.1 Mobile Computation

Mobile computation is a promising programming paradigm for network-oriented applications where running computations roam over the network. Various kinds of applications are proposed such as electric commerce, auction, automatic information retrieval, workflow management and automatic installation.

To move a program execution to a remote host, the execution states of a thread must be saved and be restored. It is, however, difficult for a Java program to manipulate the stack because the Java security policy forbids it. Two different approaches have been proposed for realizing transparent thread migration in Java: virtual machine extension [21] and program transformation schemes [1, 12, 18, 19, 26]. Migration is called *transparent* [13] or *strong* [4] if a program execution is resumed at a destination site with exactly the same execution state as that of the migration time. The relationship between partial continuation and transparent thread migration was first pointed out by Watanabe [28]. In the program transformation schemes, a thread migration is accomplished by three steps:

– The execution states of a target thread are saved into a machine-independent data structure at the departure site. The thread terminates itself when the migration succeeds.
– The data structure representing the execution states of a target thread is transmitted through the network to the destination site.
– A new thread is created at the destination. Equivalent execution states of the target thread are reconstructed for the new thread.

The above entire process can be implemented by using only standard mechanisms of Java.

### 4.2 Checkpointing

Checkpointing [6] is a technique that makes a system more reliable by saving its execution states into a local disk periodically. When a system fails for some reason, the system states can be recovered from the last saved system image. A source code transformer for portable checkpointing is developed [24, 25]. The idea is analogous to the case of mobile computation. Instead of sending encoded execution states to a remote computer, they are saved into a local disk.

## 5 Limitations

This section discusses the limitations of our scheme due to program transformation and Java proper problems.

### 5.1 Limitations due to Program Transformation

To save a continuation, all methods associated to the continuation must be transformed in advance. This implies that if the call stack includes stack frames of non-transformed methods, that part of the execution states cannot be saved. This situation often occurs in a program using graphical user interface since it often needs callback methods. Callback methods are invoked by a runtime system.

In our program transformation scheme, a continuation can be captured by the thread that will execute the continuation. A thread cannot make another thread capture a continuation in an efficient manner. This strongly restricts a way programs migrate in mobile computation. When a program execution that involves multiple threads is migrated to a remote host, we want a thread to move the other threads. But an efficient way of moving a set of threads by a particular thread has not been clear. In other words, subjective move [3] can be implemented in the way described in this paper, but objective move cannot be.

When a continuation is invoked, stack frames are reconstructed from state objects. This implies that values on the stack can be on different addresses from the original addresses when they were captured. When an object is allocated on the stack in C++, special care must be taken. For instance, the programmer should not derefer the address of that object since it changes when a continuation is invoked. Ramkumar gives a partial solution for this problem [25].

### 5.2 Java Proper Problems

It is difficult to save the state of bytecode subroutines in an efficient way since a return address of a bytecode subroutine cannot be saved into an object under the restriction of a Java bytecode verifier.

It is difficult to save the execution states in a class initializer because the programmer cannot call a class initializer. It is invoked by a runtime system when a class is loaded.

Locking is lost when a continuation is captured though locking is correctly recovered after state restoration. When a lock is acquired by a synchronized statement or a synchronized method, it will be released by an exception notifying state capturing.

## 6 Related Work

Implementation technique of partial continuations based on program transformation was studied to implement a Prolog system on the C language [16]. The notion of partial continuations is useful to implement the `cut` and `delay` primitives in Prolog: the former causes *backtracking* and the latter *freezes* computation of the proof of a goal until a particular variable is instantiated. Recently, the implementation technique has received much attention again and been developed through the study on implementation of transparent migration on Java. Java allows Java programs to manipulate stack frames only in a restricted form. Program transformation is known as the only way by which transparent migration is accomplished without extending virtual machines. The relationship between partial continuation and transparent thread migration was pointed out by Watanabe [28] and Sekiguchi [20]. Fünfrocken [12] pointed out that an exception handling mechanism could be used for notifying occurrence of state capturing with low costs. He developed a scheme of transparent migration for standard Java, but his scheme had difficulties in resumption of control in a compound statement. These difficulties were eliminated based on the idea of unfolding [1, 19]. All these schemes were based on source code level transformation. Then a scheme based on bytecode transformation was devised [18, 27].

The technique has been also developed for C and C++. Arachne threads system [7] is a distributed system in which a thread can be migrated to a remote host. It is also based on source-to-source transformation, but the overheads on normal execution in the system are more than 100% since every access to a local variable always incurs memory access. Porch [24] is a source code transformer for checkpointing. It shares a large part of our scheme, but it does not use exception handling to roll back the call stack since it is for the C language. Taga [26] developed a thread migration scheme based on source code transformation. It also exploits the exception handling mechanism to roll back the call stack.

The overheads due to the program transformation described in this paper can be reduced by the technique by Abe [1] and Taga [26]. The code fragments inserted for state restoration are needed only when execution states are reconstructed. When a method is transformed, their scheme generates two versions: one is fully transformed and the other is transformed only for state capturing. In ordinary execution, only the latter methods are used.

## 7 Summary

We have shown a scheme by which operators for partial continuations can be implemented on imperative languages such as C++ and Java. It implies that continuation operators such as `call/cc` can be implemented on C++ and Java. This scheme is so portable that it does not need the knowledge of native stack frames nor runtime systems. It is based on program transformation, yet overheads on execution performance are quite low due to an exception handling mechanism. We have actually implemented transformers for Java [18, 19] and C++ [26], and several benchmark measurements are reported in Sect. 3.6.

The study on this technique is not completed yet as we show several limitations in Sect. 5. Further work is needed to eliminate those limitations.

## References

1. Hirotake Abe, Yuuji Ichisugi, and Kazuhiko Kato. An Implementation Scheme of Mobile Threads with a Source Code Translation Technique in Java. In *Proceedings of Summer United Workshops on Parallel, Distributed and Cooperative Processing*, July 1999. (in Japanese).

2. Luca Cardelli. Mobile Computation. In *Mobile Object System: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 3–6. Springer-Verlag, April 1997.
3. Luca Cardelli and Andrew D. Gordon. Mobile Ambients. In Maurice Nivat, editor, *First International Conference on Foundations of Software Science and Computational Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1998.
4. Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna. Analyzing Mobile Code Languages. In *Mobile Object System: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 93–109, April 1996.
5. Olivier Danvy and Andrzej Filinski. Abstracting Control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, 1990.
6. Geert Deconinck, Johan Vounckx, Rudi Cuyvers, and Rudy Lauwereins. Survey of Checkpointing and Rollback Techniques. Technical report, ESAT-ACCA Laboratory, Katholieke Universiteit Leuven, Belgium, June 1993. O3.1.8 and O3.1.12.
7. Bozhidar Dimitrov and Vernon Rego. Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms. In *Proceedings of IEEE Parallel and Distributed Systems*, volume 9(5), pages 459–469, 1998.
8. Christophe Dony. Improving Exception Handling with Object-Oriented Programming. In *Proceedings of the 14th IEEE computer software and application conference COMPSAC'90*, pages 36–42, November 1990.
9. Matthias Felleisen. The Theory and Practice of First-Class Prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, 1988.
10. Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. A Syntactic Theory of Sequential Control. In *Theoretical Computer Science*, volume 52, pages 205–237, 1987.
11. S.N. Freund and J.C. Mitchell. A Type System for Object Initialization in the Java Bytecode Language. *ACM Transaction on Programming Languages and Systems*, 21(6):1196–1250, November 1999.
12. Stefan Fünfrocken. Transparent Migration of Java-Based Mobile Agents. In *MA'98 Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 26–37. Springer-Verlag, 1998.
13. Robert S. Gray. Agent Tcl: A Transportable Agent System. In *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management*, 1995.
14. Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A Generalization of Exceptions and Control in ML-like Languages. In *Conference Record of FPCA'95 SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 12–23, June 1995.
15. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification Second Edition.* Addison-Wesley, 1999.
16. Vincenzo Loia and Michel Quaggetto. High-level Management of Computation History for the Design and Implementation of a Prolog System. *Software – Practice and Experience*, 23(2):119–150, February 1993.
17. J. G. Mitchell and W. Maybury. *Mesa language manual.* Xerox PARC, April 1979. CSL-79-3.
18. Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa. Bytecode Transformation for Portable Thread Migration in Java. In *Proceedings of the Joint Sym-*

*posium on Agent Systems and Applications / Mobile Agents (ASA/MA)*, pages 16–28, September 2000.

19. Tatsurou Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation. In *Coordination Languages and Models*, volume 1594 of *Lecture Notes in Computer Science*, pages 211–226. Springer-Verlag, April 1999.
20. Tatsurou Sekiguchi and Akinori Yonezawa. A Calculus with Code Mobility. In H. Bowman and J. Derrick, editors, *Proceedings of Second IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*, pages 21–36. Chapman & Hall, 1997.
21. Kazuyuki Shudo. Thread Migration on Java Environment. Master's thesis, University of Waseda, 1997.
22. Raymie Stata and Martín Abadi. A Type System for Java Bytecode Subroutines. SRC Research Report 158, Digital Systems Research Center, June 1998.
23. Raymie Stata and Martín Abadi. A Type System for Java Bytecode Subroutines. In *Conference Record of POPL'98: 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 149–160, 1998.
24. Volker Strumpen and Balkrishna Ramkumar. Portable Checkpointing and Recovery in Heterogeneous Environments. Technical report, University of Iowa, 1996. TR-96.6.1.
25. Volker Strumpen and Balkrishna Ramkumar. Portable Checkpointing for Heterogeneous Architectures. In *Fault-Tolerant Parallel and Distributed Systems*, chapter 4, pages 73–92. Kluwer Academic Press, 1998.
26. Nayuta Taga, Tatsurou Sekiguchi, and Akinori Yonezawa. An Extension of C++ that Supports Thread Migration with Little Loss of Normal Execution Efficiency. In *Proceedings of Summer United Workshops on Parallel, Distributed and Cooperative Processing*, July 1999. (in Japanese).
27. Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen, and Pieere Verbaeten. Portable Support for Transparent Thread Migration in Java. In *Proceedings of the Joint Symposium on Agent Systems and Applications / Mobile Agents (ASA/MA)*, pages 29–43, September 2000.
28. Takuo Watanabe. Mobile Code Description using Partial Continuations: Definition and Operational Semantics. In *Proceedings of WOOC'97*, 1997.