1L-9

実行時情報を活用した第一級継続のオーバーヘッド削減手法

片桐　国建†　　小宮　常康†
†電気通信大学大学院 情報システム学研究科

1　はじめに

第一級継続という継続を第一級のオブジェクトとしてとりだす言語機能が知られており、この言語機能をJavaScript などの言語に、例外処理を用いて追加するプログラム変換が知られている [1][2]。JavaScript においては、長時間に渡る処理やブロックする処理によって他のイベント処理が停滞するのを避けるために第一級継続が利用できる場面がある。

近年、JavaScript の処理系の開発は活発に進んでおり多くの高度な最適化が行われている。しかし、第一級継続の実現のために例外処理が多く挿入されたプログラムは、この最適化が抑制され、例外処理の挿入に起因するオーバーヘッドが大きく増加する。本研究では、実行時情報を活用し動的に try-catch 挿入を決定することでこの問題を改善する手法を提案する。

2　例外処理を用いた第一級継続の実現手法

Sekiguchi ら [1] は Java のような命令型言語で部分継続を操作するための手法を示している。オーバーヘッドの少ない継続操作の実現方法として例外処理を利用したプログラム変換を提案している。Loitsch[2] は、JavaScript における継続のキャプチャを実現するために、例外処理を用いたプログラム変換を提案している。

継続の操作を行う機能の実現には、ランタイムスタックが保持する情報にアクセスするしくみを必要とする。前述の既存手法では、このしくみを実現するために例外処理を用いており、関数呼び出し部に try-catch を挿入するプログラム変換を行う (図 1 参照)。図 1 において、縦の矢印は関数の処理、黒丸は例外処理の挿入箇所を表している。suspend は継続をキャプチャするための関数である。継続をキャプチャを行う際に、図 1 のように例外を throw し、呼び出し元の全ての関数ローカルな情報にアクセスする。

既存手法では原則、全ての関数に try-catch を挿入するが try-catch の挿入は最適化の妨げになる。また、全ての関数呼び出しのうち実際に try-catch 挿入が必要なのは一部の関数呼出しに限られることも多い。例えば、図 2 のような場合では 4 つの関数呼び出しがあるが、継続のキャプチャに try-catch の挿入の必要な呼び出しは上部の 2 つだけである。

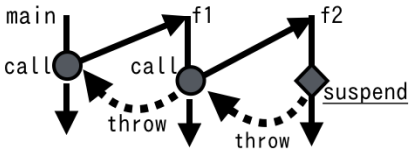

図 1: 例題処理による継続の操作の実現

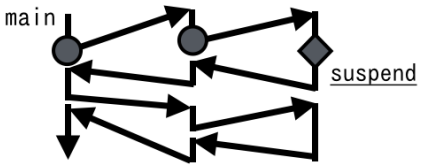

図 2: 継続キャプチャに必要な try-catch 挿入

この問題に対して、静的解析により try-catch の挿入を抑制し、オーバーヘッドを削減することが考えられる。しかし、確実にプログラムの実行を完遂するためには保守的な try-ctach の挿入を行わなければならず、多くの無駄な部分に try-catch を挿入することとなる。

3　再実行による動的な try-catch の追加

本研究では、実行時情報を用いて動的に例外処理 (try-catch) の挿入箇所を決定する。このとき、try-catch の挿入不足により継続のキャプチャに失敗したとしても、大域脱出を行い、挿入不足箇所に try-catch を挿入した上で関数呼び出しの再実行を行う手法を提案する。この手法により、できる限り try-catch の挿入を抑える投機的な戦略をとることができるようになる。図 3 は try-catch の挿入不足時にどのような手順で try-catch の挿入、再実行を行うかの流れを表した図である。

本手法では、従来の手法における try-catch 挿入に加えて、実行時情報を収集するコード、動的に try-catch 有無を切り替えを可能にする条件分岐のコード、try-catch の不足時に try-catch を挿入し関数呼び出しを再実行するためのコードが追加される。これらの処理はオーバーヘッド増加の原因になるため、提案手法導入によるオー

A Technique for Reducing the Overhead of First-Class Continuations by Using Run-Time Information
†Kunitate KATAGIRI　†Tsuneyasu KOMIYA
†Graduate School of Information Systems, The University of Electro-Communications

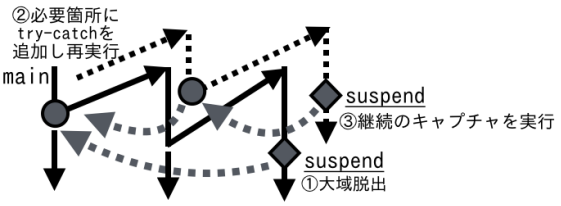

図 3: 再実行による継続のキャプチャ

```
// 変換前
var f = function ( i ) {
  if( i < 1 ) return;
  f( i - 1 );
};
// 変換後
var f = (function(){
  // try-catch を含まない関数
  var funcA = function ( i ) {
    if( i < 1 ) return;
    f( i - 1 );
  };
  // try-catch を含む関数
  var funcB = function ( i ) {
    if ( i < 1 ) return;
    t1 = i - 1.0;
    try { f( t1 ); } catch( ex ) {
      /* 継続キャプチャのためのコード */
    }
  };
  var currentFunc = funcB;
  var self = function( i ){ currentFunc( i ); };
  /* 外部から関数の内部状態を変更するためのコード */
  return self;
})();
```

図 4: プログラム変換の例

バーヘッド増加と try-catch の挿入削減によるオーバーヘッド削減はトレードオフの関係となる。

4 プログラム変換

本研究では、既存の研究で紹介された try-catch 挿入によるプログラム変換を用いた継続のキャプチャの手法を改良する。動的な try-catch の挿入・削除を可能にし、実行時情報を活用して適切な箇所に try-catch を挿入することでオーバーヘッド削減を行う。

図 4 に、本手法によって動的に try-catch の挿入を切り替えるための変換を行った例を示す。図 4 では変換前と変換後のそれぞれの関数を並べて比較している。変換後の関数は、funcA と funcB という 2 種類の関数を内部の環境に持っており、この 2 種類の関数を呼び分けることで try-catch の有無を決定している。

プログラム変換時には、try-catch の挿入不足をチェックするためのコードも追加する。try-catch の挿入不足時に継続をキャプチャする場合には、大域脱出を行い try-catch を挿入し関数呼び出しを再実行するための throw を行うように設計されている。

全ての関数は、実行時情報を収集するためのプログラム変換が施されており、呼出し回数の情報と、try-catch が継続のキャプチャに利用されたかの真偽値の情報を保持している。これらの情報を利用し、適切に関数呼び出しを切り替える機構を持つ。

5 評価・考察

提案手法における関数呼び出しのオーバーヘッドを Tak 関数を用いて実験した。この実験では Tak の実行中に suspend することが決してなく try-catch が不要であると想定した場合のベンチマークで行った。既存手法と提案手法の比較を行っており、既存手法においては、try-catch 挿入による変換後の Tak の実行のオーバーヘッドの測定を行った。また、提案手法においては、動的に try-catch を取り除くしくみを導入する変換をした Tak を用意し、try-catch が動的に取り除かれる前、取り除かれた後のそれぞれのオーバーヘッドの測定を行った。この 3 種類のオーバーヘッドを比較した結果を表 1 に示す。全ての関数呼び出しのうち約 3 割以上が try-catch を含まない関数呼び出しとなったとき、提案手法は既存手法より高速に動作することが予想される。

表 1: 関数呼び出しのコスト

Tak 関数の呼び出し方式	実行時間 (s)	
	SpiderMonkey	v8
既存手法	2.99	0.75
提案手法 (try-catch の削減前)	3.65	0.95
提案手法 (try-catch の削減後)	0.24	0.18

6 まとめ

実行時情報を活用して動的に try-catch を挿入／削減することで、オーバーヘッドを削減するためのプログラム変換手法を提案し、提案手法の有効性を調べる実験を行った。本手法により try-catch の挿入の大幅な抑制が可能になり、try-catch の削減が有効に働いた場合に実行性能を向上させることができる。

参考文献

[1] Sekiguchi, T., Sakamoto, T., and Yonezawa, A. Portable implementation of continuation operators in imperative languages by exception handling. Lecture Notes in Computer Science 2022 (2001), pp. 217 – 233.

[2] Florian Loitsch, Exceptional Continuations in JavaScript, Proceedings of the 2007 Workshop on Scheme and Functional Programming, pp. 37 – 46.