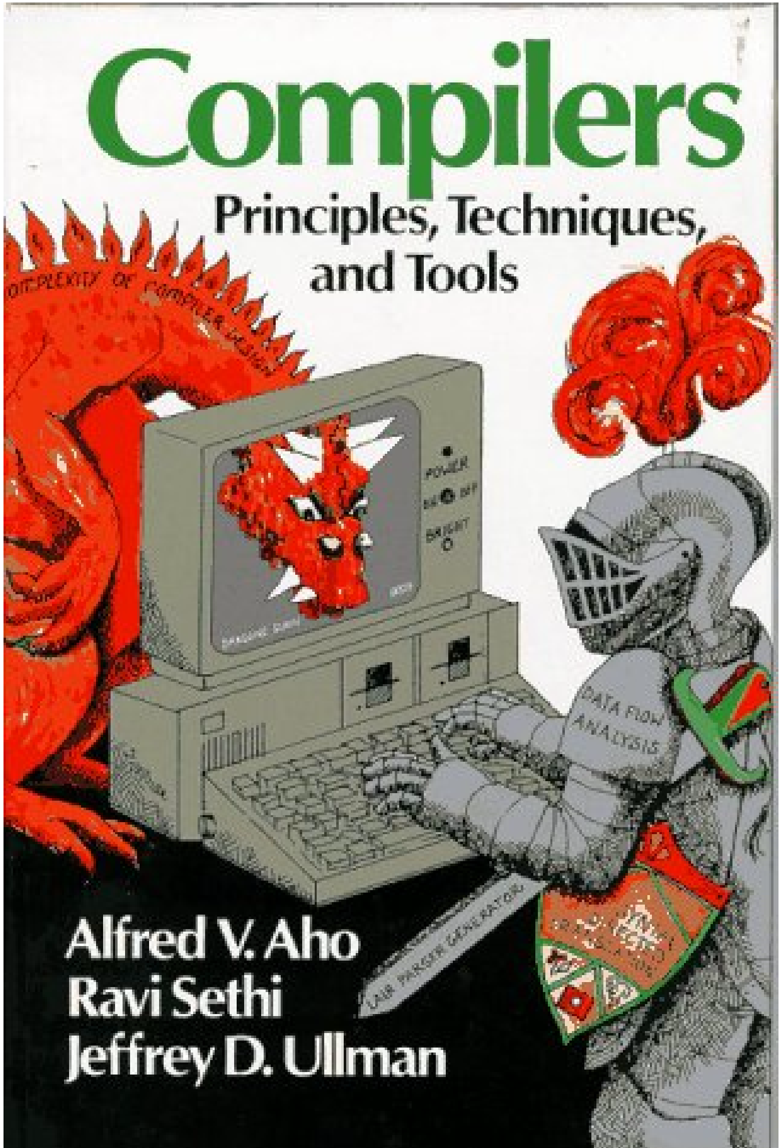
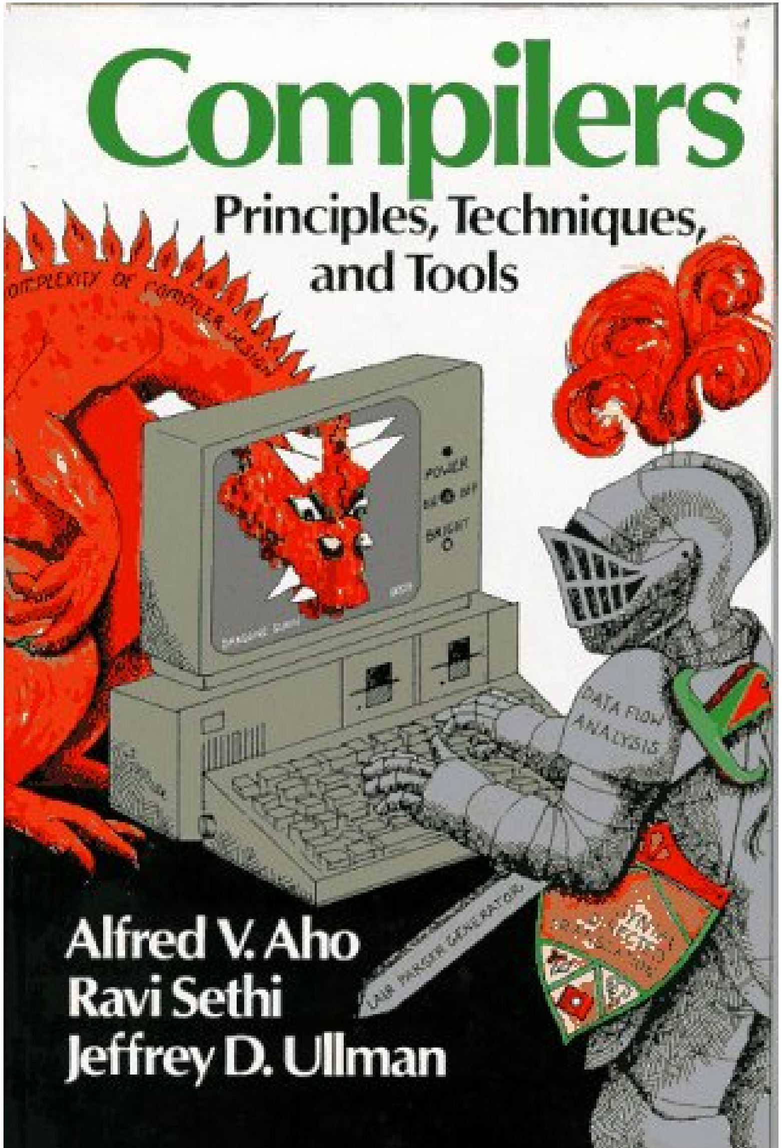
例外処理機構を備えた命令型言語のCPS変換とその定式化

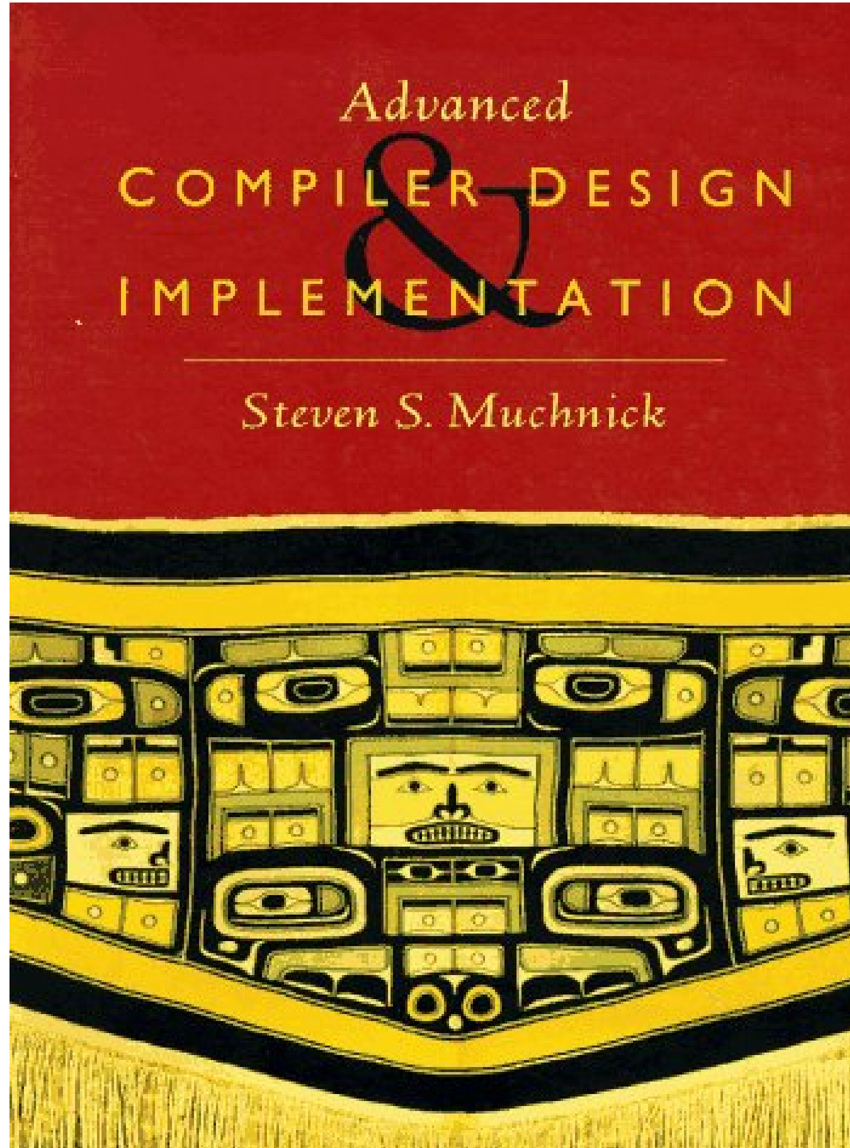
住井 英二郎
（ペンシルバニア大学）
大根田 裕一　米澤 明憲
（東京大学）

背景1: 命令型言語のコンパイラ

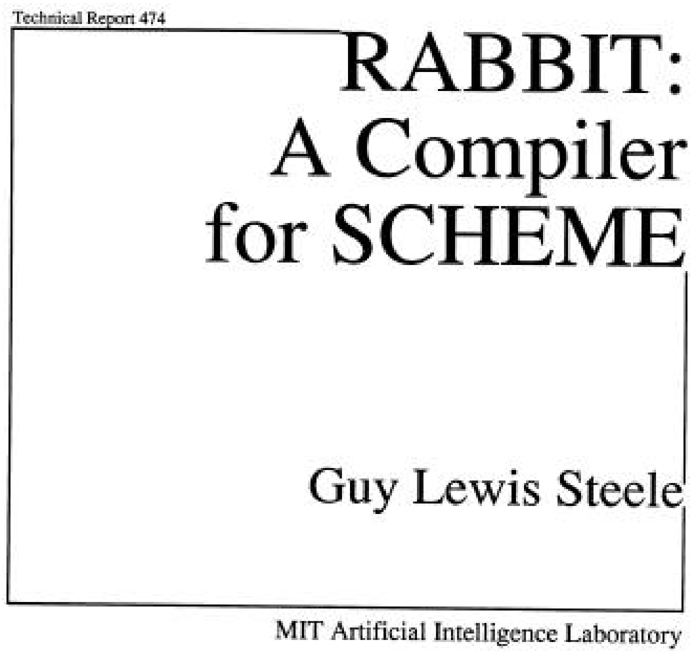


背景1: 命令型言語のコンパイラ





背景2: 関数型言語のコンパイラ

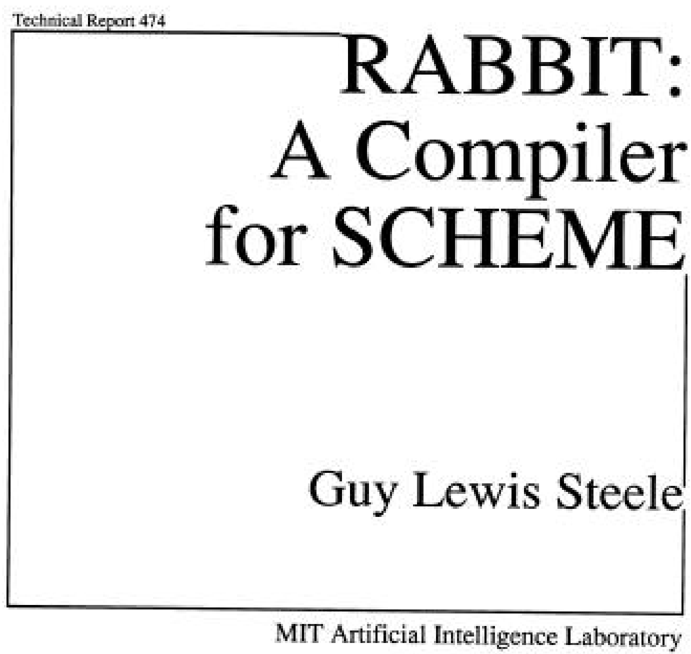

Technical Report 474

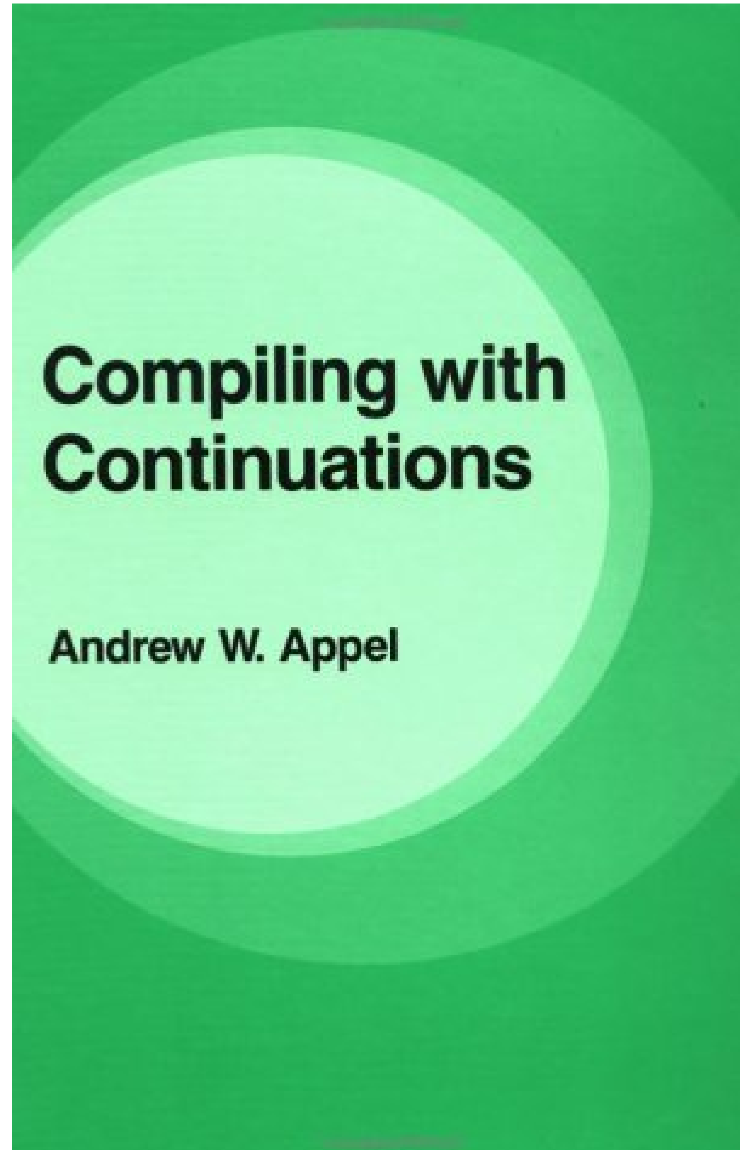
RABBIT:
A Compiler
for SCHEME

Guy Lewis Steele

MIT Artificial Intelligence Laboratory

背景2: 関数型言語のコンパイラ





背景2: 関数型言語のコンパイラ

The Essence of Compiling with Continuations

Cormac Flanagan* Amr Sabry* Bruce F. Duba Matthias Felleisen*

Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

In order to simplify the compilation process, many compilers for higher-order languages use the continuation-passing style (CPS) transformation in a first phase to generate an intermediate representation of the source program. The salient aspect of this intermediate form is that all procedures take an argument that represents the rest of the computation (the "continuation"). Since the naïve CPS transformation considerably increases the size of programs, CPS compilers perform reductions to produce a more compact intermediate representation. Although often implemented as a part of the CPS transformation, this step is conceptually a second phase. Finally, code generators for typical CPS compilers treat continuations specially in order to optimize the interpretation of continuation parameters.

A thorough analysis of the abstract machine for CPS terms shows that the actions of the code generator *invert* the naïve CPS translation step. Put differently, the combined effect of the three phases is equivalent to a source-to-source transformation that simulates the compaction phase. Thus, fully developed CPS compilers do not need to employ the CPS transformation but can achieve the same results with a simple source-level transformation.

1 Compiling with Continuations

A number of prominent compilers for applicative higher-order programming languages use the language of continuation-passing style (CPS) terms as their intermediate representation for programs [2, 14, 18, 19]. This strategy apparently offers two major advantages. First, Plotkin [16] showed that the λ-value calculus based on the β-value rule is an operational semantics for the source language, that the conventional *full* λ-calculus is a semantics for the intermediate language, and, most importantly, that the λ-calculus proves more equations between CPS terms than the λ_v-calculus does between corresponding terms of the source language. Translated into practice, a compiler can perform more transformations on the intermediate language than on the source language [2:4–5]. Second, the language of CPS terms is basically a stylized assembly language, for which it is easy to generate actual assembly programs for different machines [2, 13, 20]. In short, the CPS transformation provides an organizational principle that simplifies the construction of compilers.

To gain a better understanding of the role that the CPS transformation plays in the compilation process, we recently studied the precise connection between the λ_v-calculus for source terms and the λ-calculus for CPS terms. The result of this research [17] was an extended λ_v-calculus that precisely corresponds to the λ-calculus of the intermediate CPS language and that is still semantically sound for the source language. The extended calculus includes a set of reductions, called the A-reductions, that simplify source terms in the same manner as realistic CPS transformations simplify the output of the naïve transformation. The effect of these reductions is to name all intermediate results and to merge code blocks across declarations and conditionals. Direct compilers typically perform these reductions on an *ad hoc* and incomplete basis.[1]

The goal of the present paper is to show that the true purpose of using CPS terms as an intermediate representation is also achieved by using A-normal forms. We base our argument on a formal development of the abstract machine for the intermediate code of a CPS-based compiler. The development shows that this machine is identical to a machine for A-normal forms. Thus, the back end of an A-normal form compiler can employ the same code generation techniques that a CPS compiler uses. In short, A-normalization provides an organiza-

*Supported in part by NSF grants CCR 89-17022 and CCR 91-22518 and Texas ATP grant 91-003604014.

To appear in:
1993 Conference on Programming Language Design and Implementation.
June 21–25, 1993
Albuquerque, New Mexico

[1] Personal communication: H. Boehm (also [4]), K. Dybvig, R. Hieb (April 92).

背景2: 関数型言語のコンパイラ

The Essence of Compiling with Continuations

Cormac Flanagan* Amr Sabry* Bruce F. Duba Matthias Felleisen*

Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

In order to simplify the compilation process, many compilers for higher-order languages use the continuation-passing style (CPS) transformation in a first phase to generate an intermediate representation of the source program. The salient aspect of this intermediate form is that all procedures take an argument that represents the rest of the computation (the "continuation"). Since the naïve CPS transformation considerably increases the size of programs, CPS compilers perform reductions to produce a more compact intermediate representation. Although often implemented as a part of the CPS transformation, this step is conceptually a second phase. Finally, code generators for typical CPS compilers treat continuations specially in order to optimize the interpretation of continuation parameters.

A thorough analysis of the abstract machine for CPS terms shows that the actions of the code generator *invert* the naïve CPS translation step. Put differently, the combined effect of the three phases is equivalent to a source-to-source transformation that simulates the compaction phase. Thus, fully developed CPS compilers do not need to employ the CPS transformation but can achieve the same results with a simple source-level transformation.

1 Compiling with Continuations

A number of prominent compilers for applicative higher-order programming languages use the language of continuation-passing style (CPS) terms as their intermediate representation for programs [2, 14, 18, 19]. This strategy apparently offers two major advantages. First, Plotkin [16] showed that the λ-value calculus based on the β-value rule is an operational semantics for the source language, that the conventional *full* λ-calculus is a semantics for the intermediate language, and, most importantly, that the λ-calculus proves more equations between CPS terms than the λ_v-calculus does between corresponding terms of the source language. Translated into practice, a compiler can perform more transformations on the intermediate language than on the source language [2:4–5]. Second, the language of CPS terms is basically a stylized assembly language, for which it is easy to generate actual assembly programs for different machines [2, 13, 20]. In short, the CPS transformation provides an organizational principle that simplifies the construction of compilers.

To gain a better understanding of the role that the CPS transformation plays in the compilation process, we recently studied the precise connection between the λ_v-calculus for source terms and the λ-calculus for CPS terms. The result of this research [17] was an extended λ_v-calculus that precisely corresponds to the λ-calculus of the intermediate CPS language and that is still semantically sound for the source language. The extended calculus includes a set of reductions, called the *A*-reductions, that simplify source terms in the same manner as realistic CPS transformations simplify the output of the naïve transformation. The effect of these reductions is to name all intermediate results and to merge code blocks across declarations and conditionals. Direct compilers typically perform these reductions on an *ad hoc* and incomplete basis.[1]

The goal of the present paper is to show that the true purpose of using CPS terms as an intermediate representation is also achieved by using *A*-normal forms. We base our argument on a formal development of the abstract machine for the intermediate code of a CPS-based compiler. The development shows that this machine is identical to a machine for *A*-normal forms. Thus, the back end of an *A*-normal form compiler can employ the same code generation techniques that a CPS compiler uses. In short, *A*-normalization provides an organiza-

*Supported in part by NSF grants CCR 89-17022 and CCR 91-22518 and Texas ATP grant 91-003604014.

To appear in:
1993 Conference on Programming Language Design and Implementation.
June 21–25, 1993
Albuquerque, New Mexico

[1] Personal communication: H. Boehm (also [4]), K. Dybvig, R. Hieb (April 92).

Page 1

問題意識: もっと両者の知見を相互に応用できないか?

- 今回の研究: 例外処理つき命令型言語の最適化コンパイルにCPSを利用
  - 複雑な制御を単純に表現 (cf. [Strachey/Wadsworth 74])

  ⇒ 自明でない変換を自明に実現

最適化の例

```
int main(int x)
{ goto L0;
  try { L0: return sub(x); }
  catch { raise; }}
int sub(int x)
{ if x = 0 then raise;
  return x; }
```

```
int main(int x)
{ if x = 0 then raise;
  return x; }
```

発表の構成

- 序論
- 例題

  関数呼び出しの例

  単純な例

  関数間例外処理の例

  関数内例外処理の例

- 定式化と証明
- 関連研究
- 今後の課題

変換の例1: 破壊的代入の変換

```
  int a[], r, i;
L0:
  r = 1;
  i = 0;
L1:
  if i = 10
  then return r;
L2:
  r = r * a[i];
  i = i + 1;
  goto L1;
```

⇒

```
f0(a, r, i) =
  let r = 1 in
  let i = 0 in
  f1(a, r, i)
f1(a, r, i) =
  if i = 10
  then r
  else f2(a, r, i)
f2(a, r, i) =
  let r = r * a[i] in
  let i = i + 1 in
  f1(a, r, i)
```

変換の例2: 関数呼び出しの変換

```
int main(int a[])
{ int r, i;
  r = 1;
  i = 0;
L1:
  if i = 10
  then return r;
L2:
  r = mul(r, a[i]);
  i = i + 1;
  goto L1; }

int mul(int x, int y)
{ return x * y; }
```

⇒

```
main(k, a) =
  let r = 1 in
  let i = 0 in
  f1(k, a, r, i)
f1(k, a, r, i) =
  if i = 10
  then k(r)
  else f2(k, a, r, i)
f2(k, a, r, i) =
  let k'(r) =
    let i = i + 1 in
    f1(k, a, r, i)
  in mul(k', r, a[i])
mul(k, x, y) = k(x * y)
```

変換の例3: 関数内の例外処理

```
int main(int a[])
{ int r, i;
  try
  { r = 1;
    i = 0;
  L1:
    if i = 10
    then return r;
  L2:
    if a[i] = 0
    then raise;
  L3:
    r = r * a[i];
    i = i + 1;
    goto L1; }
  catch
  { L4: return 0; }}
```

```
main(k, a) =
  let r = 1 in
  let i = 0 in
  f1(k, a, r, i)
f1(k, a, r, i) =
  if i = 10
  then k(r)
  else f2(k, a, r, i)
f2(k, a, r, i) =
  if a[i] = 0
  then f4(k, a, r, i)
  else f3(k, a, r, i)
f3(k, a, r, i) =
  let r = r * a[i] in
  let i = i + 1 in
  f1(k, a, r, i)
f4(k, a, r, i) = k(0)
```

変換の例4: 関数間の例外処理

```
int main(int a[])
{ int r, i;
  try
  { r = 1;
    i = 0;
  L1:
    if i = 10
    then return r;
  L2:
    r = mul(r, a[i]);
    i = i + 1;
    goto L1; }
  catch
  { L3: return 0; }}

int mul(int x, int y)
{ if y = 0 then raise;
  L4: return x * y; }
```

```
main(k, h, a) =
  let r = 1 in
  let i = 0 in
  f1(k, h, a, r, i)
f1(k, h, a, r, i) =
  if i = 10 then k(r)
  else f2(k, h, a, r, i)
f2(k, h, a, r, i) =
  let k'(r) =
    let i = i + 1 in
    f1(k, h, a, r, i)
  and h'() =
    f3(k, h, a, r, i)
  in mul(k', h', r, a[i]
f3(k, h, a, r, i) = k(0)
mul(k, h, x, y) =
  if y = 0 then h()
  else f4(k, h, x, y)
f4(k, h, x, y) = k(x * y
```

最初の例

```
int main(int x) {
  goto L0;
  try {
    L0: return sub(x);
  } catch {
    raise;
  }
}

int sub(int x) {
  if x = 0
    then raise;
  return x;
}
```

```
main(k, h, x) =
  f1(k, h, x)
f1(k, h, x) =
  let k'(r) = k(r)
  and h'() = f2(k, h, x) i
  sub(k', h', x)
f2(k, h, x) = h()

sub(k, h, x) =
  if x = 0
    then h()
    else k(x)
```

最初の例

```
int main(int x) {
  goto L0;
  try {
    L0: return sub(x);
  } catch {
    raise;
  }
}

int sub(int x) {
  if x = 0
    then raise;
  return x;
}
```

```
main(k, h, x) =
  f1(k, h, x)
f1(k, h, x) =
  let k'(r) = k(r)
  and h'() = f2(k, h, x) i
  sub(k', h', x)
f2(k, h, x) = h()

sub(k, h, x) =
  if x = 0
    then h()
    else k(x)
```

最初の例

```
int main(int x) {
  goto L0;
  try {
    L0: return sub(x);
  } catch {
    raise;
  }
}

int sub(int x) {
  if x = 0
    then raise;
  return x;
}
```

```
main(k, h, x) =
  f1(k, h, x)
f1(k, h, x) =
  let k'(r) = k(r)
  and h'() = f2(k, h, x) i
  if x = 0
    then h'()
    else k'(x)
f2(k, h, x) = h()
```

最初の例

```
int main(int x) {
  goto L0;
  try {
    L0: return sub(x);
  } catch {
    raise;
  }
}

int sub(int x) {
  if x = 0
    then raise;
  return x;
}
```

```
main(k, h, x) =
  f1(k, h, x)
f1(k, h, x) =
  let k'(r) = k(r)
  and h'() = f2(k, h, x) i
  if x = 0
    then h'()
    else k'(x)
f2(k, h, x) = h()
```

最初の例

```
int main(int x) {
  goto L0;
  try {
    L0: return sub(x);
  } catch {
    raise;
  }
}

int sub(int x) {
  if x = 0
    then raise;
  return x;
}
```

```
main(k, h, x) =
  f1(k, h, x)
f1(k, h, x) =
  if x = 0
    then f2(k, h, x)
    else k(x)
f2(k, h, x) = h()
```

最初の例

```
int main(int x) {
  goto L0;
  try {
    L0: return sub(x);
  } catch {
    raise;
  }
}

int sub(int x) {
  if x = 0
    then raise;
  return x;
}
```

```
main(k, h, x) =
  f1(k, h, x)
f1(k, h, x) =
  if x = 0
    then f2(k, h, x)
    else k(x)
f2(k, h, x) = h()
```

最初の例

```
int main(int x) {
  goto L0;
  try {
    L0: return sub(x);
  } catch {
    raise;
  }
}

int sub(int x) {
  if x = 0
    then raise;
  return x;
}
```

```
main(k, h, x) =
  f1(k, h, x)
f1(k, h, x) =
  if x = 0
    then h()
    else k(x)
```

最初の例

```
int main(int x) {
  goto L0;
  try {
    L0: return sub(x);
  } catch {
    raise;
  }
}

int sub(int x) {
  if x = 0
    then raise;
  return x;
}
```

```
main(k, h, x) =
  f1(k, h, x)
f1(k, h, x) =
  if x = 0
    then h()
    else k(x)
```

最初の例

```
int main(int x) {
  goto L0;
  try {
    L0: return sub(x);
  } catch {
    raise;
  }
}

int sub(int x) {
  if x = 0
    then raise;
  return x;
}
```

```
main(k, h, x) =
  if x = 0
    then h()
    else k(x)
```

最初の例

```
int main(int x) {
  if x = 0
    then raise;
  return x;
}
```

```
main(k, h, x) =
  if x = 0
    then h()
    else k(x)
```

定式化と証明

- フローグラフ風命令型言語IMPとその操作的意味論を定義
- IMPからCPSへの変換を定義
- MPでの実行とCPSでの実行が対応することを証明

変換元言語IMP

$$
\begin{array}{rcl}
P & ::= & \{D_1, \ldots, D_n\} \\
D & ::= & f(\overline{x})\{\texttt{var}\ \overline{y}; B_0; \\
 & & \quad L_1(H_1) : B_1; \ldots; L_n(H_n) : B_n \\
H & ::= & L \\
 & | & \bot \\
B & ::= & x := i; B \\
 & | & x := y; B \\
 & | & x := y - z; B \\
 & | & x := f(\overline{y}); B \\
 & | & \texttt{goto}\ L \\
 & | & \texttt{return}\ x \\
 & | & \texttt{if}\ x \leq y\ \texttt{then}\ L_1\ \texttt{else}\ L_2 \\
 & | & \texttt{raise}
\end{array}
$$

変換T: プログラムと関数宣言

$$\mathcal{T}(\{D_1,\ldots,D_n\}) = \mathcal{T}(D_1)\cup\ldots\cup\mathcal{T}(D_n)$$

$$\mathcal{T}(f(\overline{x})\{\texttt{var}\ \overline{y}; B_0; L_1(H_1):B_1;\ldots L_n(H_n):B_n\})$$

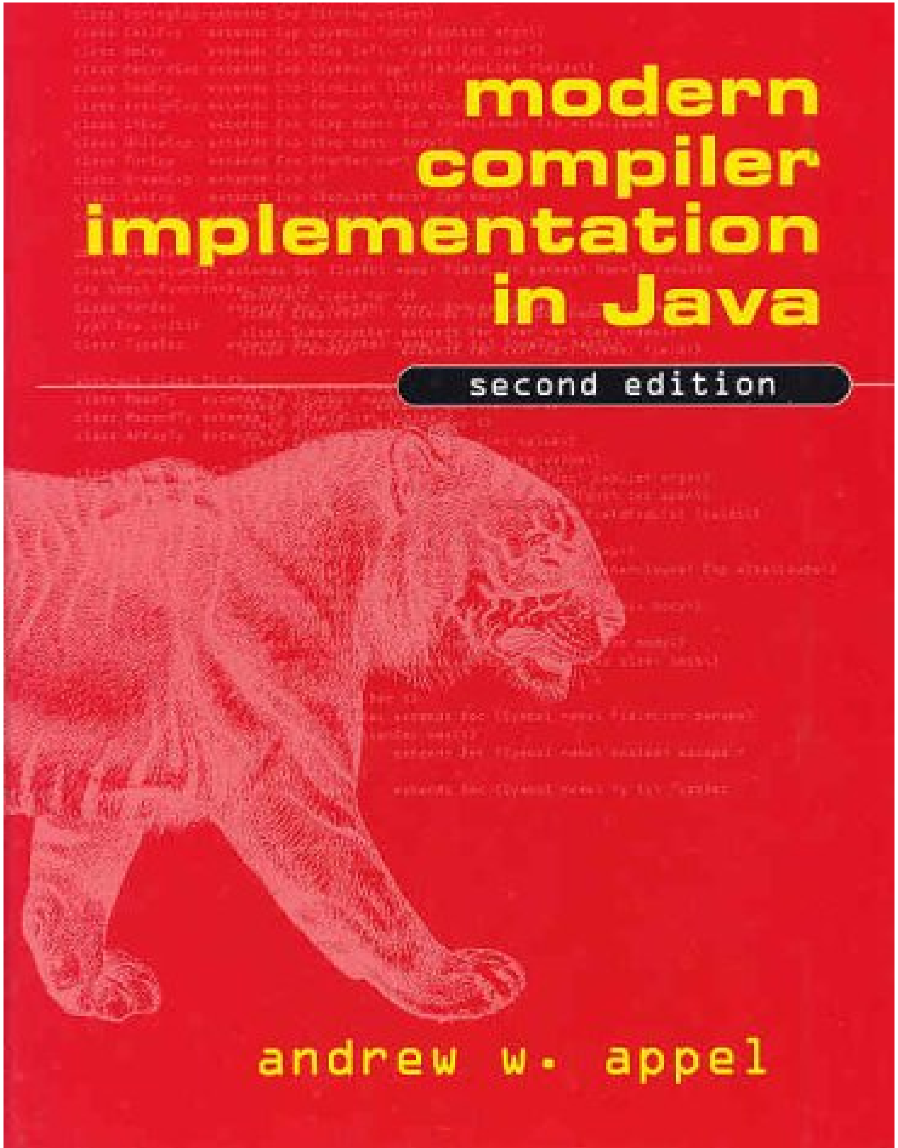
$$\begin{aligned}= \{f(k,h,\overline{x}) \ = \ & \texttt{let}\ y_1 = \texttt{0 in} \\ & \ldots \\ & \texttt{let}\ y_m = \texttt{0 in} \\ & \mathcal{T}(f,k,h,(\overline{x},\overline{y}),\bot,B_0), \\ f.L_1(k,h,\overline{x},\overline{y}) &= \mathcal{T}(f,k,h,(\overline{x},\overline{y}),H_1,B_1), \\ \ldots & \\ f.L_n(k,h,\overline{x},\overline{y}) &= \mathcal{T}(f,k,h,(\overline{x},\overline{y}),H_n,B_n)\}\end{aligned}$$

k, h fresh

変換T: 基本ブロック

$$
\begin{array}{lcl}
\mathcal{T}(f,k,h,V,H,(x := i; B)) & = & \texttt{let } x = i \texttt{ in } \mathcal{T}(f,k,h,V,H,B) \\
\mathcal{T}(f,k,h,V,H,(x := y; B)) & = & \texttt{let } x = y \texttt{ in } \mathcal{T}(f,k,h,V,H,B) \\
\mathcal{T}(f,k,h,V,H,(x := y - z; B)) & = & \texttt{let } x = y - z \texttt{ in } \mathcal{T}(f,k,h,V,H,B \\
\mathcal{T}(f,k,h,V,\bot,(x := g(\overline{y}); B)) & = & \texttt{let } k' = \lambda x.\, \mathcal{T}(f,k,h,V,\bot,B) \texttt{ in} \\
 & & g(k',h,\overline{y}) \qquad k' \text{ fres} \\
\mathcal{T}(f,k,h,(\overline{z}),L,(x := g(\overline{y}); B)) & = & \texttt{let } k' = \lambda x.\, \mathcal{T}(f,k,h,V,\bot,B) \\
 & & \texttt{and } h' = \lambda\_.\, f.L(k,h,\overline{z}) \texttt{ in} \\
 & & g(k',h',\overline{y}) \qquad k',\ h' \text{ fres} \\
\mathcal{T}(f,k,h,(\overline{z}),H,\texttt{goto } L) & = & f.L(k,h,\overline{z}) \\
\mathcal{T}(f,k,h,V,H,\texttt{return } x) & = & k(x) \\
\mathcal{T}(f,k,h,(\overline{z}),H,\texttt{if } x \le y \texttt{ then } L_1 \texttt{ else } L_2) & & \\
 & = & \texttt{if } x \le y \texttt{ then } f.L_1(k,h,\overline{z}) \\
 & & \qquad \texttt{else } f.L_2(k,h,\overline{z}) \\
\mathcal{T}(f,k,h,V,\bot,\texttt{raise}) & = & h() \\
\mathcal{T}(f,k,h,(\overline{z}),L,\texttt{raise}) & = & f.L(k,h,\overline{z})
\end{array}
$$

関連研究


modern compiler implementation in Java
second edition
andrew w. appel

関連研究(1/1)

- [Appel 92]: MLの例外処理を、特殊なプリミティブ(gethdlr, sethdlr)のあるCPSに変換
  - 我々の変換元言語は破壊的代入のある命令型言語(特有の問題あり)
  - 我々の変換先言語は通常のCPS
- [Kelsey 95]: 一階のCPSとSSAを相互に変換
  - 我々の変換元言語は例外処理と破壊的代入あり

関連研究(2/2)

- [Appel 98]: SSAからCPSへの変換を定義
  - 我々は例外処理つき命令型言語からSSAを経由せずCPSに変換
    - 不要変数は[小林 00]などで除去可能
- [Ramsey/PeytonJones 00]: C--における例外処理の様々な実装について解説
  - 我々の結果はstack cuttingに相当
    - OCamlと同様

結論

- 例外処理つき命令型言語からCPSへの変換を定義、正当性を証明
- 今後の課題
  - フローグラフや SSAの様々なアルゴリズムをCPSに応用
    - レジスタ割り当て等
  - 実装による検証（JavaやC++のコンパイラ?）
    - ポインタ算術, オブジェクト指向, etc.と今回の研究は直交 ⇒ テクニカルな困難はない（はず）