ソースレベルチェックポインティングの実現に向けて

Towards Implementing the Source Level Checkpointing

長坂卓哉 † 平石拓 † 八杉昌宏 † 馬谷誠二 † 湯淺太一 †
Takuya NAGASAKA, Tasuku HIRAISHI, Masahiro YASUGI, Seiji UMATANI, Taiichi YUASA

† 京都大学情報学研究科通信情報システム専攻
Department of Communications and Computer Engineering,
Guraduate School of Informatics, Kyoto University
{nagasaka,hiraisi,yasugi,umatani,yuasa}@kuis.kyoto-u.ac.jp

我々は，計算途中の段階を明示した操作的意味論を持つソース言語を設計し，そのプログラムの実行時にチェックポインティングを行い，その結果として機械独立かつ可読性に富むソースレベルのコードを得るという手法を提案する．また，非チェックポインティング時の実行効率を改善するため，ソース言語で書かれたプログラムから，チェックポインティング時まで動作しないソースコード生成機能を埋め込んだ C プログラムを得る変換器を試験実装した．

1 はじめに

プログラムの実行中に実行再開のための情報を保存することをチェックポインティングという．しばしば，チェックポインティング結果には，レジスタ，スタック，大域データ，テキストなどのプロセスの状態のコピーが用いられるが，このような形のデータでは，異なるアーキテクチャで実行を再開するのは困難である．同じ計算機であったとしても，例えばライブラリや OS の問題修正やバージョンアップの後にそのまま再開できるとは限らない．また，プログラムそのものにバグがあった場合に修正して再開するのも困難である．

本研究ではチェックポインティングの結果としてソースコードを得る手法，ソースレベル・チェックポインティングを提案する．提案方式では，異なるアーキテクチャの計算機においても実行の再開が可能となり，またチェックポインティング結果は可読で変更も容易となる．そのためにはソース言語に計算途中の段階を明示した操作的意味論を与えればよい．本研究では，そのようなソース言語の例として，定数または手続きを更新可能な値としてスタックまたはヒープに保持する C 言語のような意味論を持つ手続き型言語を設計した．

ソースレベル・チェックポインティングは元々のプログラムに対応するソースレベルの計算状況を操作的意味論の遷移関係に基づき書き換えて実行すれば，いつでも計算状況をチェックポイント結果として使うことができる．しかし，これでは実行効率に問題がある．そこで，非チェックポインティング時の実行効率を改善するため，ソース言語で書かれたプログラムから，チェックポインティング時まで動作しないソースコード生成機能を埋め込んだ C プログラムを得る変換器を試験実装した．もちろん，チェックポインティング時には，操作的意味論の遷移関係に基づき計算状況を書き換えて実行した場合と同様なチェックポイント結果が出力されるようにする．

変換された C コードは，チェックポインティングが発生するまでは，ソースレベルの実行状況を表現することを先送りする．チェックポインティングを行う際には，変換後の C コード内の実行中の位置などから生成するチェックポイント結果の一部が決まり，呼び出し元の位置などと組み合わせることで全体のチェックポイント結果が得られる．

この実装には S 式ベースの C 言語のための SC 言語処理系 [1] を用いた．この処理系を利用すると，S 式で記述されたプログラムを C のプログラムにする変換器の実装が，元の言語から SC-0 言語への変形規則を記述するだけでよくなり，実装が比較的容易にできるという利点がある．

2 例とするソース言語の仕様

本節では，今回設計した，計算途中の段階を明示した操作的意味論を持つプログラミング言語 SSC の仕様を述べる．SSC は，性能を重視した場合に使われることの多い C 言語に似た意味論をもつ．

SSC の構文を図 1 に示す．ここで $\vec{e}$ は e の 0 個

以上の列を表す．$\vec{d}$，$\vec{x}$ も同様である．また{$\vec{d}$ e}は (begin $\vec{d}$ e) の省略である．

$$
\begin{array}{lcll}
e & ::= & v & (\text{値}) \\
 & | & x & (\text{変数}) \\
 & | & (e\ \vec{e}) & (\text{call}) \\
 & | & \{\vec{d}\ e\} & (\text{begin 式}) \\
 & | & (\mathtt{if}\ e\ e\ e) & (\text{if 式}) \\
v & ::= & c & (\text{定数}) \\
 & | & (\mathtt{lam}\ (\vec{x})\ e) & (\text{手続き}) \\
d & ::= & (\mathtt{def}\ x\ e) & (\text{stack allocation}) \\
 & | & (\mathtt{=}\ x\ e) & (\text{stack update})
\end{array}
$$

図 1: SSC の構文

(S, e) を計算状況として計算途中の段階を明示した操作的意味論を図 2 に示す．ここで，メタ変数 b，S の構文は次の通りであり，

$$
\begin{array}{lcll}
b & ::= & (\mathtt{def}\ x\ v) & (\text{stack slot}) \\
S & ::= & \bot & (\text{empty stack}) \\
 & | & b :: S & (\text{non-empty stack})
\end{array}
$$

例えば，(def x v) :: S は，先頭に変数 x の値 v を保持するスタックを表す．

$lookup(S, x) = v$ はスタック S における変数 x の値は v となることを表す．$update(S, x, v) = S'$ はスタック S における変数 x の値を値 v に更新するとスタック S' となることを表す．$\mathit{fresh\text{-}defs}(e) = e'$ は，変数の衝突を防ぐために，e の (束縛変数を含む) 変数を fresh な変数となるよう名前を付け替える (いわゆるスタックフレームを割り当てることに相当する) と e' となることを表す．

プログラム全体 e について遷移関係 $e \to e'$ は

$$\frac{(\bot, e) \to (\bot, e')}{e \to e'}$$

により与えることができ，例えば，遷移関係 {(def x 1) x} → {(def x 1) 1} は図 3 のようにして得られる．

関数呼び出し式が遷移関係に基づきどのように書き換えられていくかを，図 4 に示すフィボナッチを計算する SSC のプログラムを例にみる．すると，遷移関係に従ったプログラムの書き換えは図 5 のようになる．ただし，わかりやすさのため関数の部分は呼び出し直前まで関数名で示した．

$$\frac{(b :: S, \{\vec{d}\ e\}) \to (b' :: S', \{\vec{d'}\ e'\})}{(S, \{b\ \vec{d}\ e\}) \to (S', \{b'\ \vec{d'}\ e'\})}$$

$$(S, \{\vec{b}\ v\}) \to (S, v)$$

$$\frac{update(S, x, v) = S'}{(S, \{(\mathtt{=}\ x\ v)\ \vec{d}\ e\}) \to (S', \{\vec{d}\ e\})}$$

$$\frac{lookup(S, x) = v}{(S, x) \to (S, v)}$$

$$\frac{\mathit{fresh\text{-}defs}(\{(\mathtt{def}\ \vec{x}\ \vec{v})\ e\}) = \{(\mathtt{def}\ \vec{x'}\ \vec{v})\ e'\}}{(S, ((\mathtt{lam}\ (\vec{x})\ e)\ \vec{v})) \to (S, \{(\mathtt{def}\ \vec{x'}\ \vec{v})\ e'\})}$$

$$\frac{(S, e) \to (S', e')}{(S, \{(\mathtt{def}\ x\ e)\ \vec{d}\ e_1\}) \to (S', \{(\mathtt{def}\ x\ e')\ \vec{d}\ e_1\})}$$

$$\frac{(S, e) \to (S', e')}{(S, \{e\}) \to (S', \{e'\})}$$

$$\frac{(S, e) \to (S', e')}{(S, \{(\mathtt{=}\ x\ e)\ \vec{d}\ e_1\}) \to (S', \{(\mathtt{=}\ x\ e')\ \vec{d}\ e_1\})}$$

$$\frac{(S, e) \to (S', e')}{(S, (e\ \vec{e})) \to (S', (e'\ \vec{e}))}$$

$$\frac{(S, e) \to (S', e')}{(S, (v\ \vec{v}\ e\ \vec{e})) \to (S', (v\ \vec{v}\ e'\ \vec{e}))}$$

図 2: SSC の遷移規則 (if とプリミティブ関数を除く)

$$\cfrac{\cfrac{\cfrac{lookup(\mathtt{(def\ x\ 1)::\bot,\ x)} = 1}{\mathtt{((def\ x\ 1)::\bot,\ x)} \to \mathtt{((def\ x\ 1)::\bot,\ 1)}}}{\mathtt{((def\ x\ 1)::\bot,\ \{x\})} \to \mathtt{((def\ x\ 1)::\bot,\ \{1\})}}}{\mathtt{(\bot,\ \{(def\ x\ 1)\ x\})} \to \mathtt{(\bot,\ \{(def\ x\ 1)\ 1\})}}$$

図 3: {(def x 1) x} → {(def x 1) 1}

```
(begin
  (def fib (lam (n)
    (if (< n 2)
        1
        (+ (fib (- n 2)) (fib (- n 1))))))
  (fib 2))
```

図 4: (fib 2) を計算する SSC のプログラム

```
1. (fib 2)
2. (begin (def n 2)
     (if (< n 2)
         1
         (+ (fib (- n 2)) (fib (- n 1)))))
3. (begin (def n 2)
     (if (< 2 2)
         1
         (+ (fib (- n 2)) (fib (- n 1)))))
4. (begin (def n 2)
     (+ (fib (- n 2)) (fib (- n 1))))
5. (begin (def n 2)
     (+ (fib 0) (fib (- n 1))))
6. (begin (def n 2)
     (+ (begin (def n_1 0)
          (if (< n_1 2)
              1
              (+ (fib (- n_1 2)) (fib (- n_1 1)))))
        (fib (- n 1))))
7. (begin (def n 2)
     (+ (begin (def n_1 0) 1)
        (fib (- n 1))))
8. (begin (def n 2)
     (+ 1 (fib (- n 1))))
9. (begin (def n 2)
     (+ 1 (fib 1)))
10. (begin (def n 2) 2)
11. 2
```

図 5: 図 4 の計算を行う際の書き換え列 (抜粋)

$$\frac{\textit{fresh-name}(H) = c}{(H, (\texttt{cons}\ v_1\ v_2)) \to ((c\ v_1\ v_2) :: H, c)}$$

$$\frac{\textit{lookup}(H, c) = (c\ v_1\ v_2)}{(H, (\texttt{car}\ c)) \to (H, v_1)}$$

$$\frac{\textit{lookup}(H, c) = (c\ v_1\ v_2)}{(H, (\texttt{cdr}\ c)) \to (H, v_2)}$$

図 6: cons, car, cdr の遷移規則

また，ヒープを扱うには，例えばプリミティブ関数として定数 cons, car, cdr を用いればよい．

$$\begin{array}{llll} H & ::= & \bot & \text{(empty heap)} \\ & | & (c\ v\ v) :: H & \text{(non-empty heap)} \end{array}$$

を用いて，(H, e) を計算状況として計算途中の段階を明示した操作的意味論を図 6 に示す．ただし今回の試験実装では，ヒープを用いたものは対象外としている．

3 SSC の C への変換

3.1 提案方式の概要

2 節の書き換えによって SSC プログラムの実行が可能で，書き換え途中のプログラムを出力することでソースレベルのチェックポインティングも実現できる．しかし既に述べたとおり，実行時にプログラムの書き換えを逐一行うという方法では実行速度が大幅に落ちてしまう．そこで本研究では，SSC のプログラムを C 言語のプログラムに変換して実行することで性能低下を最小限に抑えた．

変換結果として得られる C 言語のプログラムは以下のような動作をする．

- C プログラムの実行結果は，SSC プログラムの書き換えの最終結果と一致する．
- 通常の実行時は与えられた式の評価だけを行い，ソースコードの書き換えは行わない．
- 実行中，外部からチェックポインティングの要求（現在の実装では SIGINT シグナル）を受けると，そのときの状態にしたがって SSC のプログラムを出力し，プログラムの実行を中断する[1]．出力されるプログラムは，書き換えによる実行の場合における，対応する時点での書き換え途中のプログラムと一致する[2]．したがって，このプログラムを再び C に変換することで，その時点から実行を再開することができる．

図 7 に，SSC のプログラムを C に変換して実行を開始し，チェックポインティングを行なって中断したあと，再び実行再開するまでの流れを示す．

[1] ただし本研究の実装では，オーバヘッドを減らすため，チェックポインティング開始の判定は関数からリターンした直後にのみ行っている．

[2] だたし後で述べるように，既に参照されないことがわかっている変数の定義は，出力コードでは省略している．

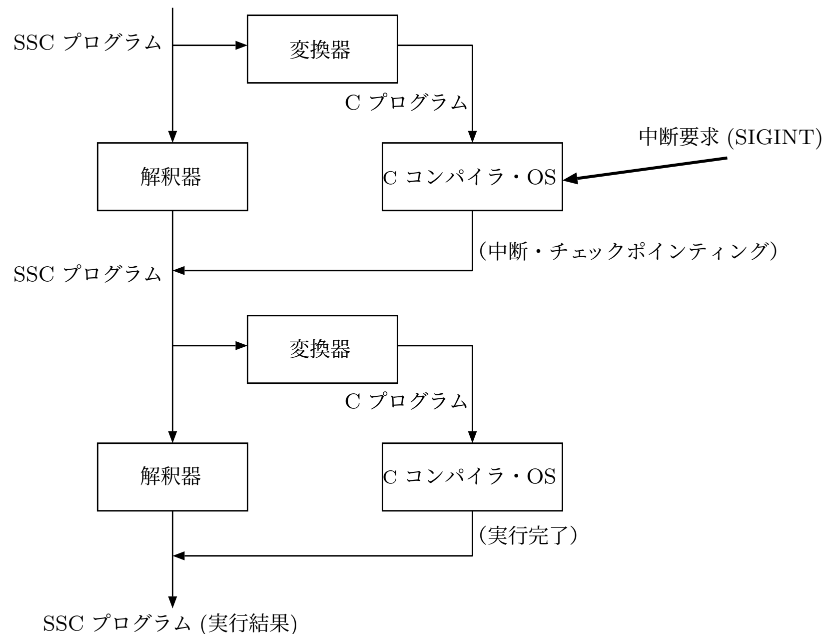

図 7: SSC プログラムの C への変換による実行

3.2 実装

この節では，上で述べた SSC から C への変換の具体的な実装方法について説明する．ただし現時点では，以下の実装の制限がある．

- 定数はプリミティブ関数を除き整数のみしか扱えない．
- 手続き（関数）の値としての使用は，定義と呼び出しのみに限定される．
- プログラムは (`begin` $\vec{d}\ e$) の形として，大域関数のみとする．

SSC で定義された関数は，同等の動作をする C の関数に変換される．このとき，通常のプログラムの動作を行うコード（通常実行部）のほかに，チェックポインティング時のコード生成のためのコード（コード生成部）も同じ関数内に生成される．コード生成部には通常の実行時には通ることはなく，チェックポインティング実行時に goto 文によってそこにジャンプされるようにしておく．

3.2.1 通常実行部の生成

通常実行部を生成するときに問題となる点とその理由及び解決法を説明する．

関数呼び出しが部分式として現れたとき 関数呼び出しからのリターン後にチェックポインティングを開始するかどうかを判定するため，部分式に関数呼び出しが含まれていると，リターン直後に判定を行うことができない．そのため関数呼び出しが部分式に含まれているならば，その関数呼び出しの部分を前に出し，関数呼び出し式があった部分は一時変数で置換する必要がある．
例えば，関数呼び出しを部分式として含む次のコードは，g(x) のリターン直後にチェックポインティングの判定式を挿入することができない．

```
int f (int x) {
  retrun g(x) + 1;
}
```

そこで，次のように変換される必要がある．ただし，chkptflag は，チェックポインティングを行う際に非 0 となるフラグである．

```
int f (int x) {
  int temp;
  temp = g(x);
  if (chkptflag) {
    /* チェックポインティング開始 */
    goto ckpt_g;
  }
```

```
  return temp + 1;
  /* ... コード生成部 ... */
}
```

値の受け渡し　SSC の言語仕様で，全ての式は値を持つと定義されている．したがって，SSC の begin 式，if 式をそれぞれ単に C の複文（{...}），if 文に置き換えただけでは不十分である．そこで，追加の一時変数を利用して，これらの式の値も正しく扱えるようにする．例えば，

```
(def f (lam (m n)
  (if (> m n)
      (g m)
      (begin (def p n) (+ p m)))))
```

のコードは，

```
int f (int m, int n) {
  int temp_1;
  if (m > n) {
    temp_1 = g(m);
    if (chkptflag) {
        /* チェックポインティング開始 */
        goto chkpt_g;
    }
  }
  else {
    int temp_2;
    {
      int p = n;
      temp_2 = p + m;
    }
    temp_1 = temp_2;
  }
  return temp_1;
  /* ... コード生成部 ... */
}
```

と変換されるようにすればよい．

if 式については，条件演算子（?，:）の式に置き換える方法も考えられるが，その内部には式しか書くことができないので，内部に呼出しや begin 式が現れた場合には上記のように if 文に置き換える必要がある．

3.2.2　コード生成部の実装

コード生成部が実行されるのは，通常実行部において関数呼び出しからのリターン後，チェックポイントフラグが立っていたときである．

以下に，チェックポインティング時に SSC のコードを生成する手順の概要を示す．

1. 関数からリターンした後，チェックポイントフラグが立っているかを調べ，立っていた場合は以前の関数で既に出力されたコードがあるかを調べる．まだ出力されたコードがなければ，関数の返り値を出力し，出力済みのコードで使われている変数を記憶するためのテーブルを作り，関数からリターンする（リターン先で再び 1. の処理を行う）．

2. 既に出力したコードがあった場合，それが

- (def x e) の e の部分だった場合は，(def x と) で，既に出力したコードを挟む．
- (= x e) の e の部分だった場合は，(= x と) で，既に出力したコードを挟み，x をテーブルに登録する．
- (e e_1 ... e_{n-1} e_n e_{n+1} ... e_m) の第 n 引数 e_n の部分だった場合は，e_1 から e_{n-1} までは既に評価されているので，それらの値を既に出力したコードの前に並べる．そして，e_{n+1} から e_m まではまだ評価されていないので，既に出力したコードの後ろに元のコードをそのまま並べる．その中で未定義のまま使われている変数はテーブルに登録する．最後に，(e と) で，並べたコードを挟み出力する．
- (begin d_1 ... d_{n-1} d_n d_{n+1} ... d_m e) の d_n だった場合，d_{n+1} から d_m および e で未定義のまま使われている変数をテーブルに登録する．そして，d_1 から d_{n-1} の (def x $e1$) の形をしているものに対し，x がテーブルに登録されていた場合は，その時点における x の値 c を調べ，(def x c) を既に出力したコードの前に並べる．x はテーブルから除く．その後，未評価の d_{n+1}, ..., d_m, e を後ろに並べ，(begin と) で，全体を挟み，出力する．
- (begin d_1 ... d_m e) の e だった場合，d_1 から d_m の (def x $e1$) の形をしているものに対

し，x がテーブルに存在すれば，その時点における x の値 c を調べ，(def x c) を既に出力したコードの前に並べる．x はテーブルから除く．そして，(begin と) で，全体を挟み，出力する．

3. 2. で出力したコードが

- (def f (lam ($\vec{x}$) e)) の e の部分であった場合は，関数の各仮引数 x_i のうちテーブルに存在するものに対し，その時点における x_i の値 c_i を調べ，(def x_i c_i) を既に出力したコードの前に並べる（テーブルに存在しない変数は実行再開後は参照されないので，その定義を出力する必要はない）．そして全体を (begin と) で挟み出力した後，現在の関数からリターンする（1. に戻る）．
- 元のコードのトップレベルの $\vec{d}$ e の e の部分だった場合は，$\vec{d}$ のうち，テーブルに登録されているものを前につけ，出力を終了する．
- どちらでもなければ 2. に戻る．

4 性能評価

2 節で定義した変形規則をそのまま実行するインタプリタを Kyoto Common Lisp [3] で実装し，SSC のプログラムを直接書き換えながら実行した場合と，3 節で実装した変換器により生成される C コード，及びチェックポインティング機能を持たない C コードの通常実行時の実行速度を比較する．比較には，Fibonacci 関数を計算するプログラム（図 4 に SSC コード）を用いた．測定環境は Ultra-SPARC-III(750MHz) で，C コンパイラには gcc（最適化オプションは-O2）を用いた．

表 1 に測定結果を示す．インタプリタ実行では，Lisp 処理系自体の動作が遅い上に，項のパターンの判定に非常に大きなオーバヘッドがかかるため，圧倒的に性能が悪い．提案手法を用いて C に変換してから実行する場合には，パターンの判定も，項の書き換えにかかる時間も存在しないため，非常に高速に動作する．

表 1: 実行時間の比較 (秒)

fib	C コード	変換 C コード	インタプリタ
10	$<$ 0.01	$<$ 0.01	0.07
15	$<$ 0.01	$<$ 0.01	0.78
20	$<$ 0.01	$<$ 0.01	8.76
25	$<$ 0.01	$<$ 0.01	101.20
30	0.05	0.08	1138.39
35	0.54	0.87	13298.04

5 議論

Porch[2] などの一般に知られているチェックポインティングの実装とは異なり，SSC から変換された C のプログラムが出力するチェックポイントコードは，SSC のプログラムコードそのものであるため，人間が直接読んで解釈しやすい．このため，出力されたコードを読んでデバッグを行ったり，プログラムの一部を変更して実行を再開するということが容易に行える（S 式なので，Lisp 処理系などを用いた機械的な処理も行いやすい）．

また，出力されたソースコード（チェックポイントコード）はアーキテクチャに依存しない形式なので，ある環境で実行停止して出力されたコードを C に変換し，別の環境でコンパイル・実行再開するということも可能である．

6 おわりに

本研究では，計算途中の段階を明示した操作的意味論を持つ言語を定義し，ソースレベルでの表現を書き換えながら動作するインタプリタよりはるかに高速な実行が可能な C コードを生成する変換器を実装した．

これにより，チェックポインティング機能を持たせながら，高速に動作させることが可能になった．生成されたチェックポインティング出力は，可読性に優れており，アーキテクチャに依存しないソースコードの形式であるため，他の環境において再開することも可能である．

今回は試験実装のため，扱う型が整数型のみである，オブジェクトを動的に生成できないなど，一般的なプログラミング言語と比べて制限が多い．今後これらの機能を実装し，より実用的なプログラミング言語となるようにしていきたい．

謝辞

本研究の一部は，文部科学省科学研究費萌芽研究 17650008，特定 (C)(2)13324050 の補助を得て行った．

参考文献

[1] 平石拓, 李暁ろ, 八杉昌宏, 馬谷誠二, 湯淺太一: S 式ベース C 言語における変形規則による言語拡張機構, **情報処理学会論文誌：プログラミング**, Vol. 46,No. SIG1(PRO 24)(2005), pp. 40–56.

[2] Strumpen, V.: Compiler Technology for Portable Checkpoints, 1998.

[3] Yuasa, T.: Design and Implementation of Kyoto Common Lisp, *Journal of Information Processing*, Vol. 13,No. 3(1990), pp. 284–293.