Coq用Cコード生成器の線形性検査拡張

田中 哲　Reynald Affeldt　Jacques Garrigue

C 言語は重要な低レベル言語であり、とくに IoT デバイスのような小さなシステムにおいて重要である。我々は Coq 用の C コード生成器を開発している。この生成器により、プログラムを Coq で形式検証し、効率的なコードを生成できる。この生成器は Gallina (Coq で用いられる言語) の ML 的なサブセットから C コードに変換する。しかし、Gallina は純粋関数型言語であるため効率のよい破壊的なデータ構造を使うことができず、これが効率的なコード生成の障害となっていた。この問題に対処するため、我々は C コード生成器に線形性検査を導入した。線形性検査は、ユーザが線形と指定した型の変数が実際に線形に使われていること、つまり、変数が必ず一回だけ使われていることを確認する。これにより、その型を C レベルで実装する場合に、破壊的なデータ構造を利用できる。C コード生成器の線形性検査拡張により、Gallina から生成されたコードで破壊的なデータ構造を安全かつ効率的に利用できる。

The C programming language plays an essential role for low-level programming, in particular for small systems such as IoT devices. We are developing a C code generator for Coq so that one can formally verify programs and still extract efficient code. Our code generator translates a program written in the ML subset of Gallina (the internal language of Coq) to C. However, the fact that Gallina is a pure functional language, and cannot use mutable data structures, is an obstacle to the generation of efficient code. We address this problem by extending our C code generator with a linearity checker. Given a specific type, the linearity checker guarantees that variables of this type are used exactly once, so that one can implement the corresponding data structure at the C level in imperative style. This extension therefore makes it possible to use mutable data structures in code generated from Gallina, safely and efficiently.

1　はじめに

我々の目的は信頼性が高く効率の良い低レベルコードを実現することである。そのために、定理証明支援系 Coq を利用してプログラムを記述し、そのプログラムを C 言語に変換するコード生成器を実現する Coq プラグイン codegen [3] を開発している [1]。このコード生成器は、Coq に組み込まれている Gallina という ML に似た言語で記述したプログラムを C 言語に変換する。Coq では Gallina で記述されたプログラムのさまざまな性質を証明できるので、Gallina でプログラムを記述して必要な性質を定理として証明することで信頼性の高いプログラムを記述できる。そして、Gallina から C 言語に変換することにより、プログラムを効率良く実行できる。

効率的な実行のためには、C 言語のデータ表現を直接扱うことが重要である。たとえば、Coq では自然数をペアノの自然数として定義する。これは証明には便利であるが、ペアノの自然数を素朴に C 言語で実装すると、効率的な実行は望めない。また、配列で十分な場合にリストを使うのも非効率である。そこで、我々のコード生成器では帰納型の実装を C レベルで自由に行えることとした。これにより、`int`などのプリミティブ型や配列などに対し帰納型としての API を実装すれば、Gallina から効率の良いデータ表現を利用できる。

ただし、Gallina は純粋関数型言語なので参照透

Extension of a C code generator for Coq with a linearity checker.

Akira Tanaka, Reynald Affeldt, 国立研究開発法人産業技術総合研究所情報技術研究部門, National Institute of Advanced Industrial Science and Technology (AIST).

Jacques Garrigue, 名古屋大学, Nagoya University.

明性が前提で、データ構造を破壊的に変更する関数は利用できない。そのため、破壊的なデータ構造を Gallina から利用するには、そのデータ構造に対して参照透明性をもつ API を提供した上で、その API を利用する必要があった。

本論文ではコード生成器に線形性検査を拡張して参照透明性を保証できる範囲を拡大し、破壊的なデータ構造を直接利用して高速化を行うことについて述べる。

本論文は以下の構成になっている。2 節で我々のコード生成器が効率の良い C コードを生成する方法について述べ、3 節ではバッファというデータ構造について述べ、4 節で線形性検査を利用して破壊的なバッファを直接利用する方法を述べる。5 節で線形性検査の機構について述べる。6 節で、バッファの高速化についての実験結果を述べる。7 節で関連研究について述べ、8 節でまとめる。

2 効率的な C コード生成

我々は効率のよい C プログラムを Gallina から生成するため、C とほぼ直接に対応する Gallina のサブセットを定義し、そのサブセットから C 言語へのコード生成器を実装している。

Gallina はどのような評価順序によっても停止することが保証されているため、正格評価を行ってもよい。その場合 Gallina と C には表 1 に示すように、基本的なプログラムを記述できるだけの共通部分がある。そのため、このようなサブセットについては Gallina と C をほぼ直接に対応させることが可能であり、効率的な C プログラムを Gallina から生成することができる。

このアイデアにもとづくコード生成器について我々は [1] で述べた。このコード生成器は単相化により多相型を Gallina プログラムから最初に取り除いてから C コードを生成する。これにより、型について最適化されたコードを生成し、効率的な実行が可能となる。また、単相化された帰納型それぞれについての実装を C 言語で自由に行える。そのため、帰納型を素朴に実装するよりも効率の良い実装を行える。たとえば、Coq では標準で真偽値型`bool`と自然数型`nat`が提供されている。我々のコード生成器ではこれらに対応する C 言語の実装を付録 A のように与えられる。

ここで帰納型に対して C 言語による実装を与える際には、Gallina という純粋関数型言語から利用しても矛盾がおきないよう、データ構造の破壊的変更を観測できない参照透明性がある実装を行う。本論文では、コード生成器に線形性検査を拡張して、参照透明性を保証できる範囲を拡大し、データ構造の破壊的変更を利用したさらに高速な実装を安全に利用可能とする。ここで線形性検査は図 1 のように、コード生成器の中で単相化の後に行うため、多相型を考慮する必要はない。

データ構造の破壊的変更はすでに存在するデータの一部を破壊的に変更することである。C 言語ではデータ構造の一部を破壊的代入によって変更することで行われる。

Gallina は純粋関数型言語であり、破壊的代入が存在しないため、データ構造を破壊的に変更することはできない。そのため、我々は以前 [1] で破壊的な操作を内部に隠蔽したデータ構造を Gallina から用いる方法を述べた。この隠蔽により、破壊的な操作を参照透明性のある形で Gallina に提供することができる。

本論文では、それとは異なり、線形性検査により参照透明性を保ったまま、破壊的な操作を直接 Gallina から用いる方法について述べる。線形性は変数が必ず一回だけ使われる性質である。この性質が成り立つ場合、破壊的なデータ構造を安全に利用できる。ある変数 x にデータ構造が束縛されていたとして、データ構造を破壊する関数 f を $f(x)$ として呼び出した場合、その呼び出しの後には x を使えないため、f がデータ構造を破壊したことは呼出側からは観測できない。つまり、参照透明性は保たれ、f は安全にデータ構造を破壊できる。

3 Motivating Example: バッファ

C 言語でよく利用するデータ構造にバッファがある。バッファは連続したメモリ領域であり、データを最後尾に追加することができる。バッファは単に連続したメモリ領域という単純な構造で高速なため C 言語でよく利用するデータ構造である。概念的には要

表 1　C と Gallina の機能の対応

機能	C	Gallina
順次実行	文の並び	let 式
条件分岐	switch 文	match 式
ループ	goto による末尾再帰	末尾再帰
関数呼び出し	関数呼び出し	関数呼び出し
ローカル変数への代入	ローカル変数への代入	末尾再帰における引数の束縛
データ構造の破壊的変更	データ構造の部分的な書き換え	線形性検査で安全性を保証した上での データ構造の部分的な書き換え (本論文による実現)

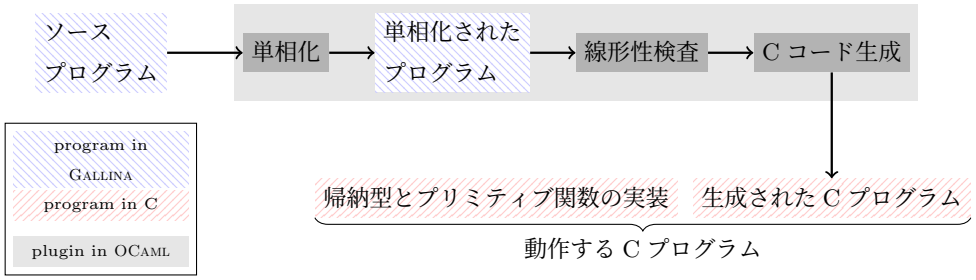

図 1　コード生成器内の線形性検査

素が並んでいる構造であるためリストに似ているが、高速性を保ったまま Gallina から利用することは容易でない。

本論文では必要に応じてバッファが伸長する可変長のバッファを扱う。追加したいデータに対して確保したメモリ領域が足りない場合、より大きなメモリ領域を確保して、それまでに記録されたデータをコピーし、データの追加を行う。このとき、確保する領域のサイズをもとのサイズの定数倍とすれば、コピーのコストを含めても計算量のオーダは悪化しない。

バッファは要素の並びであるため、ある特定のタイミングにおけるバッファの状態はリストとして表現できる。バッファの要素はさまざまなものが考えられるが、ここでは簡単のため要素は`bool`と考える。このようなリストを`bits`型と名づけると、Coq[†1] では以下のように定義できる。ここで、リスト自体をバッファと考えるのでなく、別の帰納型を定義しているのは、コード生成時にバッファとして扱うリストと、バッファとしては扱わないリストを別の型として区別するためである。

```
Inductive bits : Type := buf of seq bool.
```

空のバッファを生成する関数は以下のように定義できる。

```
Definition allocbits 'tt := buf [::].
```

バッファの状態を調べる関数として、バッファの長さを調べる関数と、自然数のインデックスで指定した要素を取り出す関数は以下のように定義できる。

```
Definition numbits '(buf s) := size s.
Definition nthbit '(buf s) i := nth false s i.
```

Gallina では破壊的変更は扱えないため、破壊的変更による変化は、変化前の状態から変化後の状態への関数として表現する。バッファに 1 要素追加する関数は以下のように定義できる。

```
Definition addbit '(buf s) v := buf (rcons s v).
```

これらの関数があればバッファに対する基本的な操作として、バッファを生成し、要素を追加し、バッファの内容を調べることが実現できる。C 言語レベルでこれらの関数を用意すれば、C 言語で実装したバッファを Gallina から利用できることになる。

†1　`seq` は SSReflect によるリストである。本論文では Coq/SSReflect を前提とする。

しかし、C 言語レベルでバッファをリストとして実現してしまうと、期待する性能を得られない。単方向リストにおける要素の追加を素朴に行うのは時間計算量のオーダが悪化するため許容できない。また、ポインタを使うことにより空間効率も悪くなる。そこで、C 言語における通常のバッファを実装として利用し、その上で上記の関数を提供することが必要となる。

これを行う 2 種類の方法を以下の節で述べる。3.1 節の方法は C 言語レベルのデータ構造を工夫してオーバーヘッドがあるもののコード生成器はそのまま使う方法であり、4 節の方法はコード生成器に線形性検査を拡張してオーバーヘッドなしにデータ構造を扱える方法である。

3.1 非破壊的バッファAPI

この節では、C 言語で実装したバッファを Gallina から使えるように、内部的な破壊的変更が観測できない API を用意するやりかたについて述べる。この方法は我々が以前簡潔データ構造のために利用したものである [1]。

Gallina で定義したbits型およびbits型の値を扱うための 4 つの関数 (allocbits, numbits, nthbit, addbit) を以下のように C で定義する。(関数の内容は付録に示す)

```
typedef struct {
  uint64_t *buf;
  nat len; /* current length [bit] */
  nat max; /* maximum length [bit] */
} bits_heap;
typedef struct {
  bits_heap *heap;
  nat orglen; /* original length [bit] */
} bits;
#define numbits(s) ((s).orglen)
bits allocbits(void);
bool nthbit(bits s, nat i);
bits addbit(bits s, bool b);
```

bits 型は bits_heap 型をポインタで参照する形で定義する。後者はヒープ領域に確保されるが、前者はプログラム中の変数の型として直接使われる。つまり、bits 型の値はスタックないしレジスタに保持される。

bits_heap型の値には、uint64_tの配列を指すポインタが含まれており、この配列に個々のbool型の値が保持される。実際に保持されている値の数は len フィールドに保持されており、未使用のものも含めた数は maxフィールドに保持されている。

ここで addbit 関数は、効率的に要素を追加するため、bits_heap内のデータを破壊的に変更する。つまり、(len < max である場合) 要素を追加したときにはbufが指す先の配列に要素を格納し、lenをインクリメントする。

しかし、addbitは Gallina から用いられるので、この破壊的変更が Gallina 側から観測できてはならない。これを実現するため、bits型には、bits_heap型へのポインタheapに加えて、orglenフィールドをもつ。bits型に対する破壊的変更は最後尾への追加のみであり、すでに追加された要素が変更されることはない。そのため、破壊的変更が行われる前の要素数さえわかれば、破壊前の状態に基づいた動作を実現できる。このためにorglenフィールドは破壊的変更が行われる前の要素数を保持する。

なお、bits型のひとつの値に対して、複数回 addbit 関数を呼び出す以下のような場合には、初回以外の呼び出しでは、バッファの内容を複製する必要がある。

```
s = allocbits(void);
s1 = addbit(s, true);
s2 = addbit(s, false);
```

これは、Gallina でバッファはリストとして定義されておりリストは分岐可能なので、C 言語で定義したバッファでも動作を一致させるためには分岐を実現しなければならないためである。ただし、このような呼び出しは複製が起きるため遅い。しかし、そもそもバッファを使うのは分岐を行わない場合なので、複製のコストは問題にならない。

ここで、通常の C 言語のバッファでは最新でない状態に基づいた動作は不可能であり、アプリケーションがバッファを使うのは最新の状態に対する動作だけで十分な場合である。そのため、最新でない状態に基づいた動作を実現するための orglenフィールドは本質的には不要であり、オーバーヘッドである。

4 線形性検査を利用した安全なバッファ

本節では前節で述べたorglenフィールドのオーバー

ヘッドを線形性検査で取り除く方法を述べる。

前節で述べたように、アプリケーションがバッファを用いるのは、バッファの最新でない状態にアクセスする必要がない場合である。そのため、アプリケーションがバッファの最新でない状態にアクセスしないことを保証できるのであれば、オーバーヘッドを取り除くことができる。

このような保証は変数が必ず一回だけ使われるという性質を検査すれば保証できる。このような性質を変数の線形性と呼ぶ。検査する対象の変数は型によって区別する。変数の利用に線形性を要求する型を linear な型と呼び、変数の利用回数に制限がない型を unrestricted な型と呼ぶ。

線形性を検査する方法としては型システムを用いるものもあるが、ここでは単に C コード生成の時点で変数の出現回数を数えることによって検査する。そのため、証明など、C コード生成の対象にならない部分については、線形性は検査されず、linear な型の変数でも利用回数の制限はない。

線形性を必要とする型はバッファのように線形性を必要とする C レベル実装を持つ型である。そのため、C レベル実装に付随して型を linear と宣言すればよい。また、そのように宣言した型をフィールドに持つ帰納型も linear とする。それ以外のすべての型は unrestricted とする。とくに、関数型は常に unrestricted である。そのため、クロージャが linear な変数をキャプチャすることは禁止する。なお、単相化により事前に多相型は取り除かれるため、多相型はこの linear かどうかの区別の対象にはならない。

型が linear であることを宣言するには以下のように行う。

```
CodeGen Linear bits.
```

この宣言により、コード生成時に `bits`型の変数は必ず一回だけ使われることが検査される。たとえば、以下の `appb2` と `invalid` をコード生成すると、前者は `b` が 1 回だけ使われているため線形性検査を通って成功するが、後者は `b` が 2 回使われているため線形性検査が失敗してコード生成に失敗する。

```
Definition appb2 (b : bits) (v : bool) :=
  addbit (addbit b v) v.
Definition invalid (b : bits) (v : bool) :=
  (addbit b v, addbit b v).
```

このようにして変数が必ず一回だけ使われる保証があると、`bits` の実装は前節で述べたオーバーヘッドを取り除いた以下の実装を利用できる。

```
typedef struct {
  uint64_t *buf;
  nat len; /* current length [bit] */
  nat max; /* maximum length [bit] */
} *bits;
#define numbits(s) ((s)->len)
bits allocbits(void);
bool getbit(bits s, nat i);
bits addbit(bits s, bool b);
```

ここで、`bits`型は構造体へのポインタとなっており、前節の `bits`型にあった `orglen`フィールドがなくなっている。これは、最新の状態のみを扱えばよいことから、値が生成されたときの状態を保存する必要がないためである。

`orglen`フィールドを除去したことにより、関数呼び出しにおいて渡さなければならない情報が減少し、関数呼び出しが高速になることが期待できる。また、`bits`型が構造体から単なるポインタに変わったことにより、C コンパイラが`bits`型の変数で構造体を扱う必要がなくなり、直接レジスタで扱うコードを生成することも容易になっている。`addbit`関数の中でも、古い状態に対する追加を検知してバッファをコピーする処理は不要になっており、これも高速化に寄与する。

5 線形性検査

線形性検査は単相化後の Gallina の項を検査し、その中に現れる linear な型の変数の使用回数を確認し、linear な変数が正しく使われていることを保証する。

ここで扱う Gallina の項は Gallina 全体ではなく、単相化が行われた結果であり、多相型はすべて具体化されている。そのため、項の構文は以下のようになる。

$$
\begin{array}{lcl}
I & : & \text{inductive type name} \\
t & ::= & I\ t \ldots t \mid t \to t \\
x & : & \text{variable} \\
c & : & \text{constant} \\
C & : & \text{constructor} \\
ar(C) & : & \text{arity of } C \\
k & : & \text{natural number} \\
f & ::= & \mathsf{fun}\ (x_1 : t_1) \ldots (x_n : t_n) \Rightarrow e \\
 & \mid & \mathsf{fix}\ k\ \{x_i : t_i := f_i\}_{i=1}^{n} \\
e & ::= & x \mid c \mid C \mid f \mid e\, e \ldots \mid \mathsf{let}\ x : t = e\ \mathsf{in}\ e \\
 & \mid & \mathsf{match}\ e\ \mathsf{with}\ \{C_i\ x_1 \ldots x_{ar(C_i)} \Rightarrow e_i\}_{i=1}^{n}
\end{array}
$$

ここで、Gallina では通常カリー化した関数を扱うが、C 言語でカリー化した関数を扱うことは困難であるため、カスケードした関数抽象 (fun) および関数呼び出しはひとまとまりとして扱う。つまり、$\mathsf{fun}\ x_1 : t_1 \Rightarrow \mathsf{fun}\ x_2 : t_2 \Rightarrow \ldots\ \mathsf{fun}\ x_n : t_n \Rightarrow e$ という形の関数では、$x_1 \ldots x_n$ をひとまとまりとして受け取り、また、対応する関数呼び出しでは同じ数だけの引数を与えるものとする。そのため、構文の中では関数抽象と関数呼び出しは複数の引数をとる形で記述している。

型が linear であるかどうかは以下のように判定する。ここで、単相化により型変数は存在しないため、プログラムに現れる型はすべて linear かどうか判定することができる。

- 型 t が帰納型 $I\ t_1 \ldots t_n$ の形であり、ユーザが `CodeGen Linear` $t.$ として linear であることを宣言している場合、t は linear である。
- 型 t が帰納型 $I\ t_1 \ldots t_n$ の形であり、I のパラメータを $t_1 \ldots t_n$ に具体化した場合に、linear な引数をもつコンストラクタが存在するなら、t は linear である。
- 型 t が関数型 $t_1 \to t_2$ の形である場合、t は unrestricted である (linear ではない)。

関数型が常に unrestricted であるため、関数抽象の内部では外部で束縛された linear な変数を参照できない。なお、カスケードした関数抽象はひとまとまりとして扱うため、$\mathsf{fun}\ x_1 : t_1 \Rightarrow \mathsf{fun}\ x_2 : t_2 \Rightarrow x_1$ という関数抽象で t_1 が linear だった場合でも、外部の linear な変数を参照するとはみなさない。

式 e において、linear な変数が正確に 1 回だけ使われていることは、linear な変数に対して初期値 0 のカウンタを利用し、式を以下のように再帰的に深さ優先のトラバースを行うことによって確認する。

- e が変数 x であり、x の型が unrestricted である場合はなにもせずにトラバースを続ける。
- e が変数 x であり、x の型が linear である場合、x が最内の (カスケードした) 関数抽象よりも外側で束縛された変数かどうかで場合分けする。外側で束縛された変数の場合は linear な変数を関数抽象の内部から参照することは禁止しているため、エラーとする。内側で束縛された変数の場合は、x に対応するカウンタをインクリメントする。その結果カウンタが 2 となったのであれば、x を複数回使用しているのでエラーとする。そうでなければ (カウンタが 1 であれば) トラバースを続ける。
- e が定数 c, コンストラクタ C の場合はなにもせずにトラバースを続ける。
- e が関数呼び出し $e_0\, e_1 \ldots e_n$ の場合、$e_0, e_1, \ldots, e_n$ を順にトラバースする。
- e が $\mathsf{let}\ x : t = e_1\ \mathsf{in}\ e_2$ であり、t が unrestricted である場合、e_1, e_2 を順にトラバースする。
- e が $\mathsf{let}\ x : t = e_1\ \mathsf{in}\ e_2$ であり、t が linear である場合、まず e_1 をトラバースし、次に x に対応するカウンタを用意して e_2 をトラバースする。トラバース後に x のカウンタが 1 でない場合は変数の使用回数が 1 回ではないのでエラーとする。
- e が $\mathsf{match}\ e_0\ \mathsf{with}\ \{C_i\ x_1 \ldots x_{ar(C_i)} \Rightarrow e_i\}_{i=1}^{n}$ の場合、まず e_0 をトラバースし、そのトラバース後におけるすべてのカウンタの内容を保存する。そして、各分岐 e_i について、保存したカウンタの内容を復元し、各フィールド $x_1, \ldots, x_{ar(C_i)}$ の中で linear な型のものに対応するカウンタを用意する。その後、e_i をトラバースする。e_i のトラバース後、$x_1, \ldots, x_{ar(C_i)}$ に対応するカウンタが 1 でなければ、変数の使用回数が 1 回ではないのでエラーとする。すべての分岐のトラバースがエラーにならなかった場合、各分岐のトラ

バース後のカウンタの内容を比較する。カウンタの内容が (各コンストラクタのフィールドに対応するものを除いて) 一致しない場合、linear な変数の使用回数が一定でないのでエラーとする。そうでなければ、一致したカウンタを使ってトラバースを続ける。

- e が関数抽象 fun $(x_1 : t_1) \ldots (x_n : t_n) \Rightarrow e$ あるいは fix k $\{x_i : t_i := f_i\}_{i=1}^{n}$ である場合、引数のうち型が linear なものについてカウンタを用意し、関数のボディをトラバースする。トラバース後、引数のカウンタが 1 でないものがあった場合は使用回数が 1 回でないのでエラーとする。なお、fix により束縛される変数は関数型で unrestricted でありカウンタは不要である。

このようにトラバースすることにより、式の中で linear な変数が正しく使われていることを確認できる。

6 ベンチマーク

線形性検査を利用してオーバーヘッドを安全に取り除いたことによる速度向上を確認するため、`addbit` を 100000000 回呼び出すプログラムの実行時間を測定した。CPU は Core i7-6600U 2.60GHz であり、Debian GNU/Linux 9.5 (stretch) 上で gcc 6.3.0 を最適化オプション -O2 を利用した。結果は表 2 のとおりで、非破壊的 API 版は平均 0.338 秒、線形性検査版は平均 0.255 秒となり、約 1.3 倍の速度向上が実現できた。

ベンチマークに利用した非破壊的 API 版のソースを付録 B, 線形性検査版のソースを付録 C, それらを動作させたメインプログラムを付録 D に示す。

表 2　ベンチマーク結果

試行回	非破壊的 API 版 実行時間 [s]	線形性検査版 実行時間 [s]
1 回目	0.380	0.291
2 回目	0.337	0.252
3 回目	0.339	0.252
4 回目	0.330	0.250
5 回目	0.333	0.253
6 回目	0.332	0.253
7 回目	0.331	0.249
8 回目	0.331	0.250
9 回目	0.335	0.251
10 回目	0.330	0.252
平均	0.338	0.255

7 関連研究

線形型を用いて破壊的なデータ構造を安全かつ高速に利用することは古くから提案されている [4]。本論文と [4] を比較すると、多相型を扱っていない点は同じであり、関数がすべて unrestricted な点はより単純になっている。

我々は Coq からビット列の効率的な実装を用いることについて、[2] と [1] で報告してきた。

[2] では Coq の extraction 機能を使って生成するプログラムから、OCaml と C で実装したビット列を使うというものであった。基本的なアイデアは本論文でも説明した非破壊的なバッファと同じであるが、OCaml は型に関係なくすべての値が 1 ワードで表現されるため、(2 ワード必要な) `bits` 型に相当するデータはヒープ領域に置く必要があった。また、ビット列の中で 1 のビットを数えるため、Intel SSE4.2 で利用できる POPCNT 命令を用いたが、これを OCaml から呼び出すためには関数とせねばならず、たったひとつの命令を使うために関数呼び出しのオーバーヘッドが生じていた。

[1] では Coq の extraction 機能は使わず、我々が独自に開発した Coq プラグインで C コードを生成した。C 言語では型ごとに異なる表現が可能なので、非破壊的なバッファの `bits` 型をスタックに配置することが可能となった。また、POPCNT 命令をマクロやインライン関数で生成することにより、関数呼び出しのオーバーヘッドを減らすことができた。本論文は線形性検査により、`bits`型内の `orglen`フィールドを除去し、さらにオーバーヘッドを減らしたものである。

Coq から C コードを生成するプロジェクトには CertiCoq [5] とŒuf [6] がある。これらは trusted base を減らすことに注力しており、現実的に利用するために必要なデータ構造の置き換えはいまのところ扱っていない。たとえば、ペアノの自然数も素朴な実装とな

り、実用的とはいえない。

8 おわりに

本論文では、我々が開発している Coq から C コードを生成する Coq プラグインについて線形性検査を拡張して、破壊的なデータ構造を安全に使えるようにしたことについて述べた。これによりオーバーヘッドが減少し、速度を測定したケースでは 1.3 倍の速度向上が実現できた。

将来課題としては、データ構造を非破壊的にアクセスする関数を支援することがあげられる。そのような関数はデータ構造を破壊しないが、線形性の制約を満たすために、引数として受け取ったデータ構造を返値として返さなければならない。これはプログラムが煩雑になり、また、返値が増えるというオーバヘッドによる速度低下も発生する。これを解決する方法は [4] にも提案されており、あるスコープから線形な値がエスケープしないことを保証すればそのスコープの中では非破壊的な関数を自由に利用できる。今後そのような拡張を行うことを計画している。

謝辞

この研究は JSPS-CNRS 二国間交流事業「IoT の安全性のための形式検証ツール」(FoRmal tools for IoT sEcurity) による助成を受けている。

参 考 文 献

[1] Akira Tanaka, Reynald Affeldt, Jacques Garrigue. Safe Low-level Code Generation in Coq using Monomorphization and Monadification. Journal of Information Processing, Volume 26 (2018) Pages 54-72, 2018-01-15. https://doi.org/10.2197/ipsjjip.26.54

[2] Tanaka, A., Affeldt, R., Garrigue, J.: Formal Verification of the rank Algorithm for Succinct Data Structures, 18th International Conference on Formal Engineering Methods (ICFEM 2016), pp. 242–260.

[3] https://github.com/akr/codegen

[4] Philip Wadler, Linear types can change the world, IFIP TC2 Working Conference on Programming Concepts and Methods, North Holland (1990)

[5] Anand, A., Appel, A. W., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Savary Bélanger, O., Sozeau, M., Weaver, M.: CertiCoq: A verified compiler for Coq, The Third International Workshop on Coq for Programming Languages (2017).

[6] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, Dan Grossman, Œuf: minimizing the Coq extraction TCB, Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, Los Angeles, CA, USA, January 8-9, 2018, pages 172–185, ACM

A カスタマイズ可能な C レベルデータ構造

Gallina と C で扱うデータ型は異なる。Gallina で扱うデータ型は Inductive コマンドにより帰納的に定義される。C で扱うデータ型は int などのプリミティブ型および、それを組み合わせる構造体などの composite type である。これらの差異により、C のデータ型を Gallina から用いるには、データ型の対応を定義してギャップを埋める必要がある。

我々のコード生成器では Gallina の帰納的データ構造を C 言語で実装することが可能である。そのために、C 言語でデータ型を定義し、その値を生成するコンストラクタ関数と値の中身を識別する match 式を switch 文で実現するための定義を用意する。それらをコード生成器に登録することで C 言語で実装されたデータ構造を Gallina から使うことが可能になる。

コード生成器は登録された型を用いたコードを生成するが、その型の中身には関知しない。そのため、ユーザは自由にデータ構造を実現できる。

コンストラクタは呼び出したときに値を生成する関数とみなせるので、C 言語で値を生成する関数を用意する。

match 式は値がどのコンストラクタで生成されたのかを識別し、コンストラクタに与えられた引数を取り出すものである。これを switch 文で実現するため、値から switch 文で分岐するためのタグを取り出す関数、case ラベルないしは default ラベルを生成するためのマクロ、分岐した後に引数を取り出す関数を用意する。

また、このようにして定義したデータ型に関するプリミティブ関数を C 言語で実装することも可能である。このような関数の実装ではデータ構造の内部に直接アクセスできるため、効率のよい実装が可能である。

たとえば、Gallina のbool型は以下のように定義される。

```
Inductive bool : Set :=
  | true : bool
  | false : bool.
```

これを C で実装するには、C の標準ヘッダstdbool.hで提供されるbool型を利用して以下のように定義できる。

```
#include <stdbool.h>
#define n0_true() true
#define n0_false() false
#define sw_bool(b) (b)
#define case_true_bool default
#define case_false_bool case false
#define negb(b) (!(b))
```

n0_true および n0_falseがコンストラクタとなる。(n0_ というプリフィクスは引数が 0 個であることを意味している。) sw_bool で bool型の値から switch 文のタグに変換するが、ここではbool型の値をそのまま使っている。case_true_bool と case_false_bool により、true と false の場合それぞれの分岐先のラベルを生成できる。bool型ではコンストラクタに引数はないので、フィールドを取り出す関数は不要である。また、ここではプリミティブ関数として、bool型の値を反転させるnegb関数を用意している。

また、Gallina で自然数を表現するnat型は以下のように定義されている。

```
Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.
```

これを C で実装するには、以下のように定義できる。

```
#include <stdint.h>
typedef uint64_t nat;
#define n0_O() ((nat)0)
#define n1_S(n) ((n)+1)
#define sw_nat(n) (n)
#define case_O_nat case 0
#define case_S_nat default
#define field0_S_nat(n) ((n)-1)
#define n2_addn(a,b) ((a)+(b))
#define n2_subn(a,b) ((a)-(b))
#define n2_muln(a,b) ((a)*(b))
#define n2_divn(a,b) ((a)/(b))
#define n2_modn(a,b) ((a)%(b))
```

ここでは C の標準ヘッダstdbool.hで提供される 64bit 符号なし整数uint64_tを利用して自然数natを実装している。この実装はオーバーフローがありうるため危険であるが、我々はこの危険性が実際には存在しないことを monadification という方法により証明可能としている [1]。n0_Oとn1_Sがコンストラクタであり、sw_nat, case_O_nat, case_S_nat, field0_S_natでnat型に対する match 式に対応する switch 文が構成される。field0_S_natはコンストラクタS (後者関数) で構築されたnat型の値からSの引数を求める関数であり、1 を減じる関数となっている。また、プリミティブ関数として、加算、減算、乗算、除算、余りを求める関数を用意している。

このように、C の演算子に対応する関数をマクロとして提供することにより、オーバーヘッドのないコードを生成可能である。

なお、Gallina プログラムの中でコンストラクタおよび match 式を利用しないのであれば、C 言語による実装を与えなくてもよい。この場合でもプリミティブ関数は利用可能である。データ構造によっては、コンストラクタと match 式を実現することが困難な場合がある。例えば、配列に対するコンストラクタと match 式がどのような動作をすべきかは自明でないし、仮にリスト的な動作を選んだとしても、実現にはオーバーヘッドが発生してしまう。プリミティブ関数でデータ構造内部にアクセスできれば十分な場合には、そのようなオーバーヘッドを負う必要はない。

これらの定義により、Gallina から C で定義したデータ構造を利用可能となるが、実現できるデータ構造はコンストラクタ及び match 式で扱えるものに限られる。C 言語では破壊的変更を利用できるが、それは match 式で観測できない形でしか利用できない。

B 非破壊的バッファの実装

```
typedef struct {
  uint64_t *buf;
  nat len; /* current length [bit] */
  nat max; /* maximum length [bit] */
} bits_heap;

typedef struct {
  bits_heap *heap;
  nat orglen; /* original length [bit] */
} bits;
#define numbits(s) ((s).orglen)
```

```
bits allocbits(void)
{
  nat max = 64;
  bits_heap *bh;
  bh = malloc(sizeof(bits_heap));
  if (bh == NULL)
  { perror("malloc"); exit(EXIT_FAILURE); }
  /* aligned up to 64bit */
  max = ((max + 63) / 64 * 64);
  /* avoid malloc(0) which is imp.-defined */
  if (max == 0) max = 64;
  bh->buf = malloc(max / CHAR_BIT);
  if (bh->buf == NULL)
  { perror("malloc"); exit(EXIT_FAILURE); }
  bh->len = 0;
  bh->max = max;
  bits s;
  s.heap = bh;
  s.orglen = 0;
  return s;
}

bool nthbit(bits s, nat i)
{
  assert(i < s.orglen);
  return (s.heap->buf[i / 64] >> (i % 64)) & 1;
}

bits copy_prefix(bits s, nat n)
{
  nat max = (n + 63) / 64 * 64;
  if (max == 0) max = 64;
  bits s2 = allocbits();
  memcpy(s2.heap->buf, s.heap->buf, max / CHAR_BIT);
  s2.orglen = s2.heap->len = n;
  return s2;
}

bits addbit(bits s, bool b)
{
  if (s.orglen != s.heap->len)
    s = copy_prefix(s, s.orglen);

  if (s.heap->max <= s.orglen) {
    uint64_t *buf;
    nat max;
    max = s.heap->max * 2;
    assert(s.heap->max < max);
    buf = realloc(s.heap->buf, max / CHAR_BIT);
    if (buf == NULL)
    { perror("realloc"); exit(EXIT_FAILURE); }
    s.heap->buf = buf;
    s.heap->max = max;
  }

  if (b)
    s.heap->buf[s.orglen / 64] |=
      (uint64_t)1 << (s.orglen % 64);
  else
    s.heap->buf[s.orglen / 64] &=
      ~((uint64_t)1 << (s.orglen % 64));
  s.orglen++;
  s.heap->len++;
  return s;
}
```

C　破壊的バッファの実装

```
typedef struct {
  uint64_t *buf;
  nat len; /* current length [bit] */
  nat max; /* maximum length [bit] */
} *bits;

#define numbits(s) ((s)->len)

bits allocbits(void)
{
  nat max = 64;
  bits s;
  s = malloc(sizeof(*s));
  if (s == NULL)
  { perror("malloc"); exit(EXIT_FAILURE); }
  /* aligned up to 64bit */
  max = ((max + 63) / 64 * 64);
  /* avoid malloc(0) which is imp.-defined */
  if (max == 0) max = 64;
  s->buf = malloc(max / CHAR_BIT);
  if (s->buf == NULL)
  { perror("malloc"); exit(EXIT_FAILURE); }
  s->len = 0;
  s->max = max;
  return s;
}

bool nthbit(bits s, nat i)
{
  assert(i < s->len);
  return (s->buf[i / 64] >> (i % 64)) & 1;
}

bits addbit(bits s, bool b)
{
  if (s->max <= s->len) {
    uint64_t *buf;
    nat max;
    max = s->max * 2;
    assert(s->max < max);
    buf = realloc(s->buf, max / CHAR_BIT);
    if (buf == NULL)
    { perror("realloc"); exit(EXIT_FAILURE); }
    s->buf = buf;
    s->max = max;
```

```
  }

  if (b)
    s->buf[s->len / 64] |=
      (uint64_t)1 << (s->len % 64);
  else
    s->buf[s->len / 64] &=
      ~((uint64_t)1 << (s->len % 64));
  s->len++;
  return s;
}
```

D　ベンチマークプログラム

```
int main(int argc, char *argv[])
{
  nat i;
  bits s = allocbits();
  for (i = 0; i < 100000000; i++) {
    s = addbit(s, true);
  }
  return 0;
}
```