CoqからのCプログラム生成

田中 哲
産業技術総合研究所 情報技術研究部門
2017-07-23
Proof Summit 2017

元ネタ

- 既発表の話です
- そのうち論文が出ます
- Safe Low-level Code Generation in Coq using Monomorphization and Monadification
  Akira Tanaka, Reynald Affeldt, Jacques Garrigue
  IPSJ SIGPRO 114, 2017-06-09,
  will be appear at IPSJ JIP.
- ここで出てくる plugin は github にあります
  - https://github.com/akr/monomorphization
  - https://github.com/akr/monadification

目標:CoqからCに素直に変換する

- Coq

```
Fixpoint pow a k :=
 match k with
 | 0 => 1
 | k'.+1 =>
     a * pow a k'
 end.
```

- 証明が簡単

- C

```
int pow(int a, int k) {
  switch (k) {
    case 0: return 1;
    default: {
      int k_ = k-1;
      return
        a * pow(a, k_);
}}}
```

- Cプログラムから利用可能

高速な実装も書ける

- Definition uphalf' n := n − n./2.
- (* fastpow_iter a k x = (a ^ k) * x *)
  Fixpoint fastpow_iter a k x :=
  if k is k'.+1 then
  if odd k then
  fastpow_iter a k' (a * x)
  else
  fastpow_iter (a * a) (uphalf' k') x
  else
  x.
- Definition fastpow a k := fastpow_iter a k 1.

fastpow からCコード生成

- nat n3_fastpow_iter(nat v2_a, nat v1_k, nat v0_x) {
  n3_fastpow_iter:;
  switch (sw_nat(v1_k)) {
  case_O_nat: { return v0_x; }
  case_S_nat: {
  nat v4_k_ = field0_S_nat(v1_k);
  bool v5_b = n1_odd(v1_k);
  switch (sw_bool(v5_b)) {
  case_true_bool: {
  nat v6_n = n2_muln(v2_a, v0_x);
  v1_k = v4_k_; v0_x = v6_n; goto n3_fastpow_iter; }
  case_false_bool: {
  nat v7_n = n2_muln(v2_a, v2_a);
  nat v8_n = n1_uphalf_(v4_k_);
  v2_a = v7_n; v1_k = v8_n; goto n3_fastpow_iter; }}}}}
- nat n2_fastpow(nat v10_a, nat v9_k) {
  nat v11_n = n0_O();
  nat v12_n = n1_S(v11_n);
  return n3_fastpow_iter(v10_a, v9_k, v12_n); }

証明された累乗関数の開発

- 仕様として pow を書く
- 実装として fastpow を書く
- 正しさを証明: ∀ a k, pow a k = fastpow a k
- fastpowをCに変換して失敗しないことを証明
  nat を uint64_t にしても途中でオーバーフローしない
- fastpow を C に変換する
  高速で証明されたプログラムを利用できる

なんでわざわざCoqからCを生成するのか

- 意外な組み合わせが面白い
- Cで直接書くのと比較して
  - 未定義動作を避けている証明をできるから
    整数オーバーフローやバッファオーバーランなど
  - 期待した結果になる証明をできるから
- 普通の「安全な言語」と比較して
  - 実行時オーバーヘッドをなくせるから
  - 期待した結果になる証明をできるから
- 証明を扱える言語と比較して
  - 単に証明が可能なだけでなく、手厚く支援してくれないとつらいんじゃないかなぁ（私見）
  - もちろん、いろいろ挑戦するのは素晴らしいことです

背景

- Cは低レベルのインフラによく使われる
  プログラミング言語, OS, ネットワークサーバ, 組み込みデバイス, IoT, 簡潔データ構造
  - Cは素晴らしい
    効率的な動作, 低レベルな機能へのアクセス, 現実的なポータビリティ, さまざまなシステムとの相互運用性など
  - Cは危険
    バッファオーバーラン, 整数オーバーフローなど
- インフラには頑健性が望まれる
  - 失敗しない: Cの未定義動作を避ける
  - 正しさ: 期待した結果を実現する

証明支援系 Coq

- 証明を支援する成熟したシステム
- 内部にMLに似た言語(Gallina)を持つ
- さまざまな証明ライブラリが提供されている
- OCaml で書いた plugin で拡張可能
- Extraction 機能により Gallina プログラムを OCaml, Haskell, Scheme, JSONに変換可能

アイデア: Coqで証明してCで実行する

- Gallinaでプログラムを書く
- そのプログラムをCoq内で証明する
  - 失敗しない
  - 正しさ
- GallinaからCに変換する
- 効率的で証明されたCプログラムを堪能する

GallinaからCへ素直に変換する

- Gallinaの型→Cの型
- Gallinaの関数→Cの関数
- Gallinaの変数→Cの変数
- Gallinaのlet式→Cの変数宣言・初期化
- Gallinaの関数適用→Cの関数呼び出し、もしくは goto (tail recursion の場合)
- Gallinaのmatch式→Cのswitch文

Gallinaの構文でCを書く感じ

完全なGallinaコンパイラを作るつもりはない

素直な変換の問題と解決

- 型: Gallinaの型はCよりも表現力が高い(依存型、多相型)
  →MLの型に制限した上で単相化する
  単相化したGallinaの型をCで(ユーザに)実装させる
- 評価戦略: Gallinaではどんな評価戦略でも使えるので、
  Cの評価戦略を使っても問題ない
  (実際には A正規形っぽく変換し、引数は左から右に評価する)
- match: switchは文なので値を持たない
  →事前に用意した変数に代入するようなコードを生成する
  (末尾位置ならreturnする)
- 関数: Gallinaには部分適用、クロージャなどがあるが、Cにはない
  →Cで難しいものはサポートしない(いまのところ)
- ループ: Gallinaにはループがないので、
  末尾再帰をgotoに変換してループ相当のコードに変換する
- GC: Boehm GC をリンクできる
- 部分関数: Gallinaの関数はつねに全域関数だが、Cではそうでない
  →モナド化して問題が起きないことを証明する

厄介で興味深い問題の解決

- 型: Gallinaの型はCよりも表現力が高い
  (依存型、多相型)
  →MLの型に制限した上で単相化する
  単相化したGallinaの型をCで(ユーザに)実装させる
- 部分関数: Gallinaの関数はつねに全域関数だが、Cではそうでない
  →モナド化して問題が起きないことを証明する

提案:単相化(Monomorphization)

効率の良いCプログラムを生成するのが狙い

- 依存型を使いたいわけではない
  (どうせ効率的なCコードにはならない)
- 多相型くらいは使いたい(prod 型とか)
- MLの単相化は可能 (前例: MLton)
  単純には多相関数を利用されるそれぞれの型ごとに複製する

→ MLっぽい型に制限して単相化する
単相化した型をユーザにCで実装させる
→ どのようなデータ構造でも自由に選べる
nat を uint64_t にすることもできる

全域関数と部分関数 (前提知識)

- 全域関数
  引数に対する結果が常に定義されている
  例: 自然数の加算 x + y は常に結果が定義されている
- 部分関数:
  引数によっては結果が定義されていない
  例: 自然数の除算で x / 0 は定義されていない

効率的なCの整数型をそのまま使いたい

ここでCoqは部分関数を扱えないことが問題になる

- Coq: すべての関数は全域関数（失敗できない）
  0 / 0 = 0
  n + 1 − 1 = n
- C: 関数（演算子を含む）は失敗しうる
  0 / 0 は未定義 (SIGFPE)
  n + 1 − 1 はオーバーフローするかもしれない

→ Coqの全域関数の実装として部分関数を使うには、未定義な部分を使っていないことを確認しなければならない (OCaml の int を使う場合も同じ)

部分関数をCoqで扱う現在の方法

- "失敗しない" 証明
  プログラムを書き換えなければならない:
  - 失敗しうるところすべてを option 型に変える
    (少しきれいな方法: option monad を使う)
    → Noneを伝播するのは面倒くさい

  or
  - 部分関数の引数に事前条件を満たす証明を与える
    →Certified programming には依存型が必要
- "正しさ" の証明が困難になる
  上記の書き換えられたプログラムについて証明するのは難しい
- Extractionもうまくいかなくなる
  Noneの伝播の条件分岐は実行時に残る
  依存型はすべて取り除けるとは限らない

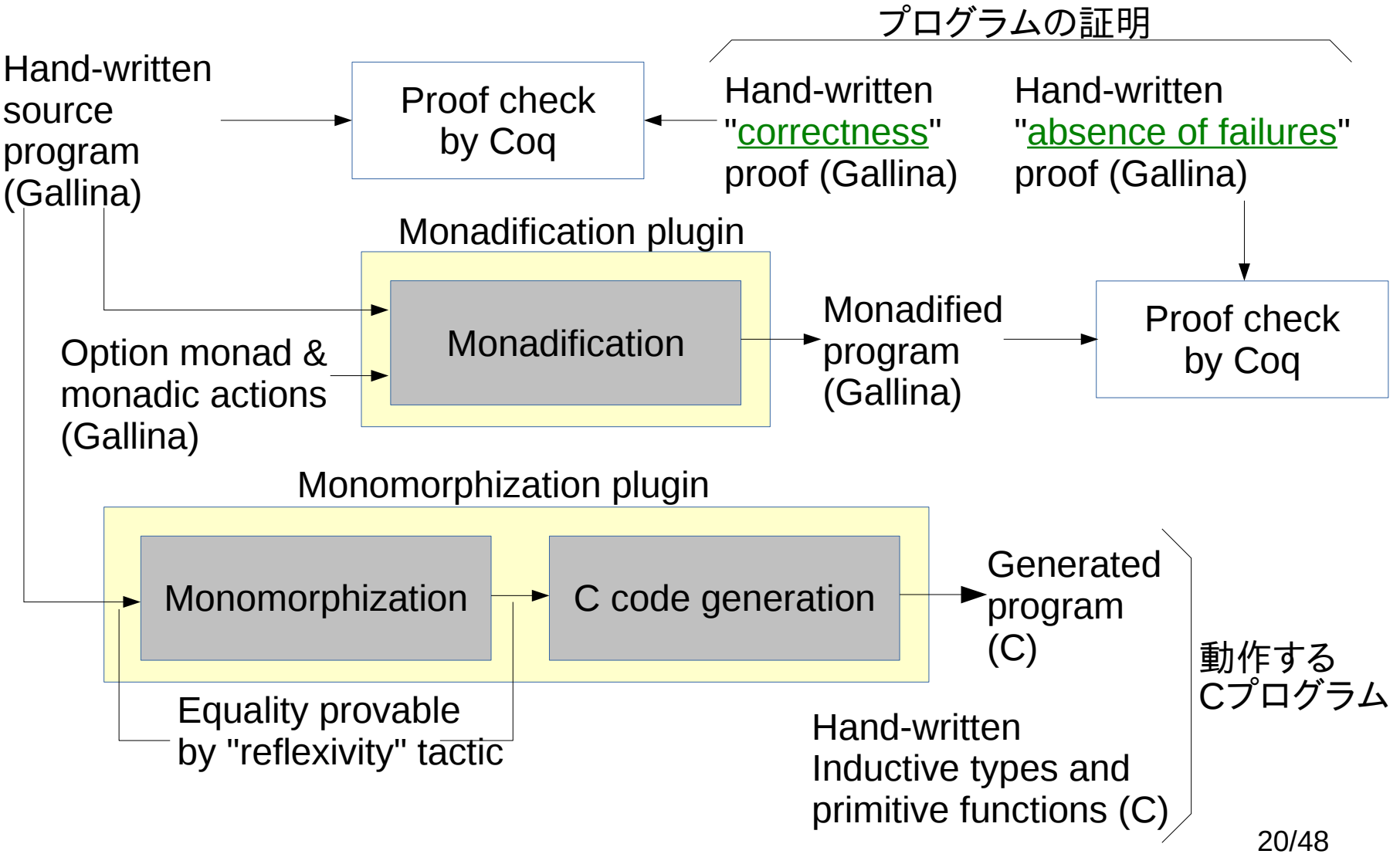
提案:機械的なモナド化(Monadification)

option monad を使うプログラムに自動変換する

- 正しさの証明と失敗しない証明を別々に書ける
  (観点の分離)
  - 正しさの証明は元のプログラムに対して行う
    例: 高速な pow = 素朴な pow
  - 失敗しない証明は機械的にモナド化したプログラムに対して行う
    例: nat は uint64_t の範囲内しか使わないので uint64_t を使っても整数オーバーフローがおきない
- 効率的なCコード生成
  元のプログラムからコード生成する
  - 実行時オーバーヘッドがない
    None伝播の分岐がない
  - 失敗しないことを前提として型を自由に実装できる
    例: nat を uint64_t に対応させる

変換の全体構成

Monomorphization pluginとMonadification plugin



CoqのExtractionは使わない

- ExtractionはCをサポートしていない
  低レベルな機能を使うのが難しい
  - 64-bit 整数
  - SSE, AVX など
  - proper tail-recursionのためのgoto
- Extractionでは型ごとに最適な実装を行うことが難しい
  - 依存型をサポートするためuniformなデータ表現にならざるをえない
  - MiniML(Extractionの中間言語)には型注釈がないところがある
    そのため型推論が必要になる(もともと型がついているのに!)
- モジュール性
  ExtractionはCの生成に対して無用に大きく、修正版の配布も困難
  - 不要な機能: 依存型のサポート、証明除去
  - Extractionの修正版を使うにはCoq自体のビルドが必要

興味深い変換ステップ

- 単相化 (Monomorphization)
  - 多相性の除去
  - この変換の入力と出力は convertible
    (Coqで機械的に等価性を確認できる)
- Cコード生成
  - 基本的には単純なシンタックスの変換
  - 型の実装は生成せず、ユーザが自由に実装できる
- モナド化 (Monadification)
  - 失敗しないことの証明のために使う
  - 計算に関する他の証明にも使える
    計算量の証明など

単相化の例

- 多相関数

```
Definition swap {A B}
  (p : A * B) :=
let (a, b) := p in (b, a).

Definition swap_bb p:=
 @swap bool bool p.
```

- 単相関数

```
Definition _pair_bool_bool:=
 @pair bool bool.

Definition _swap_bool_bool
 (p : bool * bool) :=
 let (a, b) := p in
 _pair_bool_bool b a.

Definition _swap_bb p :=
 _swap_bool_bool p.
```

```
Goal swap_bb = _swap_bb. Proof. reflexivity. Qed.
```

単相化

- 関数を型引数について特殊化する
- ついでにletを挿入する
  (コード生成のためにA正規形っぽくする)
- 変換元と変換結果はconvertibleになる
  Coqのconversion ruleの以下が使われる
  - β-reduction: 関数適用
  - ζ-reduction: let除去
- convertible なので reflexivity 一発で等価性を証明できる

単相化できる対象言語

- ML的な多相性をもつGallinaのサブセット
- Gallina全体は明らかに単相化できない
  - 多相再帰
  - 依存型
- MLを単相化することが可能なのは既知
  cf. MLton
- MLは強力な言語なので、これだけできればCで書きたいプログラムには充分なのではないか(仮説)

Cコード生成の特徴

- データ型の実装はユーザが自由に行える
  - nat を uint64_t にするなど
- gotoによるproper tail recursionの実現
  - 末尾再帰でスタックを消費しない
    昔からよく知られているやりかた
  - でも、CoqのExtractionのOCaml生成でnatをintにして
    ocamloptを使うとスタックを消費することがあるのでそう
    いうことが起きない保証があるのは重要

Cコード生成の例: pow

- 生成されたCコード

```
nat n2_pow(
    nat v88_a, nat v87_k) {
  switch (sw_nat(v87_k)) {
    case_O_nat: {
      nat v90_n = n0_O();
      return n1_S(v90_n); }
    case_S_nat: {
      nat v91_k_ =
        field0_S_nat(v87_k);
      nat v92_n =
        n2_pow(v88_a, v91_k_);
      return
        n2_muln(v88_a, v92_n);
}}}
```

- ユーザによるデータ型の実装

```
#define nat uint64_t
#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)
```

- プリミティブ関数の実装

```
#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))
```

プログラムのモナド化

- direct style

```
Fixpoint pow a k :=
 match k with
 | 0 => 1
 | k'.+1 =>
     a * pow a k'
 end.
```

- monadic style

```
Fixpoint powM a k :=
 match k with
 | 0 => SM 0
 | k'.+1 =>
     powM a k' >>=
     mulM a
 end.
```

モナド化を行う理由

計算の内容に関する証明を行うため

例:

- プログラムがnatの値として2^{64} 未満しか扱わない(natをuint64_tで実装してもオーバーフローしない)
- 配列アクセスのインデックスが常に配列サイズ未満(バッファオーバーランしない)
- プログラムが "cons" を n回呼び出す(計算量)

機械的にモナド化を行う理由

- モナド化したプログラムを書くのは面倒くさい
- 元のプログラムと両方書くのは面倒くさい
- 行いたい証明によって、異なるモナドやアクションを使わなければならない
  証明ごとにモナド化を繰り返すのは面倒くさすぎる
  - cons 呼び出しを数えるには cons をアクションにする
  - 整数オーバーフローを証明するには S (後者関数) をアクションにする
- consやSはライブラリの中でも使っているので、ライブラリもモナド化する必要がある

Monadification Plugin の使い方

1. monadic triple の設定
   Monadify Type M.
   Monadify Return f.
   Monadify Bind f.
2. アクションの登録
   Monadify Action f => fM.
3. 関数（およびそれが依存する関数）のモナド化
   Monadification f.

失敗を表現する Option Monad

```
Definition ret {A} (x : A) := Some x.
Definition bind {A} {B}
    (x' : option A) (f : A → option B) :=
  match x' with None => None
                | Some x => f x
  end.
Monadify Type option.
Monadify Return @ret.
Monadify Bind @bind.
(* Notations for ">>=" and "return" *)
```

整数オーバーフローの検出

整数オーバーフローを検出するためのアクションの登録:

```
Definition check x :=
  if Nat.log2 x < 64 then Some x else None.
Definition SM a := check a.+1.
Definition mulM a b := check (a * b).
Monadify Action S => SM.
Monadify Action muln => mulM.
```

プログラムのモナド化

- direct style (source)

```
Fixpoint pow a k :=
 match k with
 | 0 => 1
 | k'.+1 =>
     a * pow a k'
 end.
```

- monadic style (generated)

```
Fixpoint powM a k :=
 match k with
 | 0 => SM 0
 | k'.+1 =>
     powM a k' >>=
     mulM a
 end.
```

オーバーフローが起きない証明

- 一般にプログラムがある条件の元で失敗しないことは以下のような命題として記述できる
  forall x, *condition* → fM x = Some (f x)
- powが（最終的な結果がオーバーフローしないという条件の元で）オーバーフローが起きないことは以下の定理を証明することで検証できる

```
Theorem powM_ok :
  forall a b, Nat.log2 (pow a b) < 64 →
  (powM a b) = Some (pow a b).
```

計算量のためのモナド
(モナド化の別の応用)

- カウンタモナド:
```
Definition counter_with A : Type := nat * A.
Definition ret {A} (x : A) := (0, x).
Definition bind {A} {B}
  (x ' : counter_with A)
  (f : A →counter_with B) :=
  let (m, x) := x ' in let (n, y) := f x in
  (m+n, y).
```
- cons 呼び出しを数えるアクション:
```
Definition consM {T} (hd : T) tl := (1, cons hd tl).
Monadify Action cons => @consM.
```
- たとえば素朴なリスト反転がn(n+1)/2回呼ぶのに対し、末尾再帰版はn回しか呼ばないことを証明できる

Monadificationアルゴリズムの基本

- 証明を簡単にするため、モナドはなるべく入れない (fM は f と似ている形の方が良い)
  - Best: $t_1 \to t_2 \to t_3$　(そのままの型で済むならそれが一番)
  - Good: $t_1 \to t_2 \to M\ t_3$
  - Bad: $M\ (t_1 \to M\ (t_2 \to M\ t_3))$
- Cの関数では通常Mは最後にいれるだけで十分
  - 関数は引数を途中まで与えた段階では作用を持たない
- 提案するアルゴリズムは作用が最初に発生する引数の数を推論する (impure arity)

Impure Arityを使ってモナドを挿入

- f をモナド化した結果を fM とする

  f: $t_1 \to \ldots \to t_k \to t_{k+1} \to \ldots \to t_n$

  fM : $t_1 \to \ldots \to t_k \to$

  $M\ (t_{k+1} \to \ldots \to M\ (t_{n-1} \to M\ t_n)\ldots)$

- k を impure arity と呼ぶ
- 提案するアルゴリズムは証明を容易にするためなるべく大きな k を選ぶ
- 実は常に k=0 とすると Coq が受け付けない定義を生成してしまう
  - decreasing argument の検出に失敗するようになる
  - 型引数がモナド化に巻き込まれて型検査を通らなくなる
- アルゴリズムの詳細についてはいずれ出る論文で

実験

1. 既存のライブラリのモナド化

- seq.v: SSReflect のリストライブラリ
- 49個の関数のモナド化に挑戦
- 7個はpure, 36個成功，6個失敗
  (失敗は、依存型を使っているものと、関数をとるコンストラクタを使うもので、そのようなケースで失敗することはわかっていた。)

2. 簡潔データ構造のrank関数

- モナド化により以下を証明した:
  - 失敗しないこと
  - 計算量
- Cコードを生成した
  以下のデータ型を実装して利用した:
  - ビット列
  - 小さな整数の配列

具体例

概要

単相化

Cコード生成

モナド化

実験

Trusted Base

まとめ

Trusted Base が小さい

- 提案するCコード生成
  - g_monomorph.ml 430
  - monoutil.ml 136
  - genc.ml 696
- 1000行未満で済む
- 単相化は結果を簡単に証明できるのでここには含まれない

- Coq 8.6 extraction
  - g_extraction.ml4 152
  - common.ml 648
  - extract_env.ml 682
  - extraction.ml 1098
  - mlutil.ml 1524
  - modutil.ml 411
  - table.ml 921
  - ocaml.ml 773
- 6000行超

具体例

概要

単相化

Cコード生成

モナド化

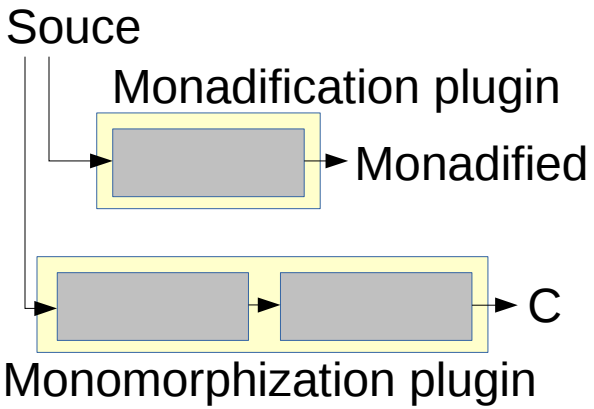
実験

Trusted Base

まとめ

まとめ

- 証明した低レベルCコードを小さなTrusted Baseで生成できた
- 単相化
  - 多相性を除去する
  - 変換の正しさは reflexivity で証明可能
- Cコード生成
  - データ型の実装を自由に選べる
  - goto による proper tail recursion
- モナド化
  - Coq で利用できるモナド化アルゴリズムを提案した
  - 計算に関する証明を行える: 失敗しない、計算量など



今後の予定

- opam package 化
- 簡単なデータ型実装の生成
- いろいろな簡潔データ構造の実装
  select, wavelet tree, etc.
- 型引数以外の引数に対する特殊化
  (部分評価)
- メモリ管理
  - Pluggable GC (Rubyに組み込むときはRubyのGCを使いたい)
  - 線形型
- クロージャ

Extra Slides

Prove a Program Never Fail

- Option monad
  - Automatic monadification and proof
    → Needs plugin
  - Write a program in monadic style with option monad (No program in direct style)
    →Tedious programming
    Difficult to remove option monad at extraction (Runtime overhead)
    - functor and identity monad
      →modules are not expanded in OCaml extraction
    - section and identity monad
      →needs to inline all functions (Too much code duplication)
- Don't fail in C
  #define n2_divn(a,b) ((b) == 0 ? 0 : (a)/(b))
  Use GMP for integer overflow
  → Runtime overhead
- Certified Programming (cf. CPDT)
  → Very difficult proof
  Needs extraction (proof erasure)
  Need to decide uint64_t or GMP at beginning
- Deep embedding using template-coq
  → Difficult proof
- Return an unknown value, u, for failures
  → wrong proof
  (let x := u in 0) = 0. (u – u) = 0 for u:nat. (if u then e else e) = e for u:bool.

Details of C Code Generation

Monomorphization before C Code Generation Example

- source program

```
Fixpoint buildDir2 b s sz2
    c i D2 m2 :=
  if c is cp.+1 then
    let m := bcount b i sz2 s i
n
    buildDir2 b s sz2
      cp (i + sz2)
      (pushD D2 m2) (m2 +
m)
  else
    (D2, m2).
```

- monomorphized program

```
Fixpoint _buildDir2 b s sz2
    c i D2 m2 :=
match c with
| 0 =>_pair_DArr_nat D2 m2
| cp.+1 =>
  let m:=_bcount b i sz2 s in
  let n := _addn i sz2 in
  let d := _pushD D2 m2 in
  let n0 := _addn m2 m in
  _buildDir2 b s sz2 cp n d n0
end.
```

C Code Generation Example

- monomorphized program

```
Fixpoint _buildDir2 b s sz2
    c i D2 m2 :=
match c with
| 0 =>_pair_DArr_nat D2 m
2
| cp.+1 =>
  let m:=_bcount b i sz2 s in
  let n := _addn i sz2 in
  let d := _pushD D2 m2 in
  let n0 := _addn m2 m in
  _buildDir2 b s sz2 cp n d
n0
end.
```

- C program

```
prod_DArr_nat n7_buildDir2(bool v10_b,
    bits v9_s, nat v8_sz2, nat v7_c,
    nat v6_i, DArr v5_D2, nat v4_m2)
{ n7_buildDir2:;
  switch (sw_nat(v7_c)) {
   case_O_nat:
    return n2_pair_DArr_nat(v5_D2,v4_m
2);
   case_S_nat: {
      nat v12_cp = field0_S_nat(v7_c);
      nat v13_m =
        n4_bcount(v10_b,v6_i,v8_sz2,v9_s);
      nat v14_n = n2_addn(v6_i, v8_sz2);
      DArr v15_d=n2_pushD(v5_D2,v4_m
2);
      nat v16_n = n2_addn(v4_m2, v13_
m);
      v7_c = v12_cp;v6_i = v14_n;
      v5_D2 = v15_d;v4_m2 = v16_n;
      goto n7_buildDir2;
}}}
```

C Code Generation is Direct

- monomorphized type name is used as-is
- function name is prefixed with the arity
  _buildDir2 → n7_buildDir2
- variable → variable
- let → variable initialization
- application → function call
  or goto for tail recursion
- match → switch

Data Type Implementation

- Data representation is fully customizable
- bool in Coq:
  Inductive bool : Set := true : bool | false : bool.
- bool implementation in C:

```
#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
```

nat Implementation

- natural number in Coq: nat
  Inductive nat : Set := O : nat | S : nat → nat.
- nat implementation in C:

```
#define nat uint64_t
#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))
```

- Integer overflow on uint64_t doesn't occur if we prove it using monadification

match → switch

- Coq

```
Inductive I :=
...
| Ci : ... → tij → ... → I
...
match v with
...
| Ci ...xij... => e
...
end
```

- C

```
switch (sw_I(v)) {
  ...
  case_Ci_I: {
    ...
    tij xij = field(i-1)_I
(v);
    ...
    /* code for ei */
  }
  ...
}
```

Experiment
Monadification of SSReflect's seq.v

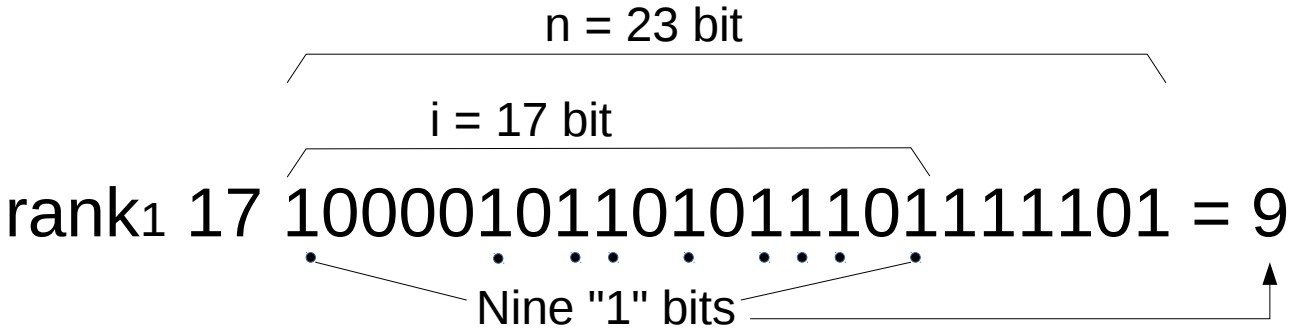
Monadification of seq.v

- Monadify 49 functions: all, allpairs, behead, belast, cat, catrev, constant, count, drop, filter, find, flatten, foldl, foldr, has, head, incr_nth, index, iota, iter, last, map, mask, mkseq, ncons, nilp, nth, ohead, pairmap, perm_eq, pmap, rem, reshape, rev, rot, rotr, scanl, seqn, set_nth, shape, size, subseq, sumn, take, undup, uniq, unzip1, unzip2 and zip.
- Monadic action: S and cons
- 7 is pure: behead, drop, head, last, nth, ohead and subseq
- 36 is successfully monadified
- 6 couldn't: constant, index, perm_eq, undup, uniq and seqn
  - seqn uses dependent type
  - others use higher order constructor (nat_eqType and seq_eqType also have same problem)

Experiment
rank function for succinct data structure

rank Function

- "rank$_b$ i s" counts the number of "b" in th e first "i" bits of "s" (which length is "n")



- Naive implementation needs O(i) time:

```
Definition rank b i s := count_mem b (take i s).
```

rank for Succinct Data Structure

- "rank_init b s" precomputes the auxiliary data: o(n) size in O(n) time
- "rank_lookup aux i" compute rank: O(1) time
- Functional correctness proved

```
Lemma RankCorrect b s i : i <= bsize s →
  rank_lookup (rank_init b s) i = rank b i s.
```

- It never fail if $n < 2^{64}$

```
Lemma RankSuccess b s i :
  let n := bsize s in log2 n < 64 → i <= n →
  (rank_initM b s >>= fun aux => rank_lookupM aux i)
  = Some (rank_lookup (rank_init b s) i).
```

- We also proved the time complexity