

言語組：講義資料

いまどきの！？ プログラミング言語理論

いなば かずひろ

稲葉 一浩

<http://www.kmonos.net/>

<http://twitter.com/kinaba>

お話しする内容

- ・時は2009年
- ・「プログラミング言語」の
「理論」の
「研究者」のあいだで
- ・今なにがアツい？

理論系のトップレベルの学会

- POPL 2009
 - Symposium on Principles of Programming Languages
 - » “プログラミング言語の本質”
- ESOP 2009
 - European Symposium on Programming

今年の時間割: POPL 2009

- 並行性 (Concurrency)
- 型 1 (Types)
- その他 1 (Medley)
- 静的解析 1 (Static analysis)
- 関数型プログラミング (Functional programming)
- その他 2 (Medley)
- 静的解析 2 (Static analysis)
- 静的解析 3 (Static analysis)
- プログラム論理学 (Program Logics)
- 型 2 (Types)
- マルチコア (Multicore)
- 検証 (Verification)

今年の時間割: ESOP 2009

- 型付き関数型言語
(Typed Functional Programming)
- 計算の(副)作用
(Computational Effects)
- オブジェクト指向
言語の型
(Types for Object-Oriented Languages)
- 検証 (Verification)
- セキュリティ
(Security)
- 並行性 (Concurrency)
- サービス指向
(Service-Oriented Computing)
- 並列・並行プログラミング
(Parallel and Concurrent Programming)

わかること

- ・ 静的解析・検証・型
 - 古くから理論屋さんの主な話題

- ・ 並行・並列・マルチコア
 - 当たり前ですが流行ってます

静的解析・検証・型

・ 例

- 「このポインタがこの変数を指す可能性はあるか無いか？」の解析 → 無いと**証明**できればできる最適化がある (参考: C99のrestricted)
- 「このint型の変数はどのくらいの範囲の値になる可能性があるか？」の解析 → 配列のオーバーランが絶対起きないことの**保証**
- Rubyのような動的型付け言語で静的に型**検査**する研究
- ...

並行・並列・マルチコア

・例

- 並列/分散プログラムの数学的に扱いやすいモデルを考える
- デッドロックしないことの自動/手動証明手法
- ロックフリーデータ構造の正しさの証明手法
- STM(トランザクショナルメモリ)が正しく動いていることの保証…

わかること：結論

- ・ プログラム理論の学者さんは
「保証」「証明」がお好き
- (※脇道：
なんで？なんのため？)

ここでニュース！！

- ・速報！！

- ・いま
「証明」界に異変！？

最近の論文からの引用

1. ...*Certicrypt*, a framework that enables the *machine-checked construction and verification* of code-based *proofs*... [G. Barthe et al. , POPL 2009]
2. ...a compiler from *Cminor* to *PowerPC* assembly code, using the *Coq proof assistant* both for *programming* the compiler and for *proving*... [X. Leroy, POPL 2006]
3. ... accurate semantics for *x86* ... *mechanised* in *HOL*. ...For programs ...we *prove* in *HOL* that their behaviour is sequentially consistent. [S. Sarkar et al., POPL 2009]
4. ... formal verification of an *SLR* parser generator... This conversion also illustrated the gap between simple textbook definitions and a *verifiable executable implementation* in a theorem prover.[M. Schäfer et al., ESOP 2009]

異変：急増しているキーワード

- ・ Mechanized 機械化
- ・ Machine-checked 機械的
チェック済
- ・ Proof 証明
- ・ Assistant 支援
- ・ Executable 実行可能
- ・ Programming プログラミング

証明のニューウェーブ

- ・紙とペンでの証明
- ・コンピュータでの全自動証明
- ・証明が書けるプログラミング言語を使う (←New!!)

(※注: このアイデア自体はNewというほどでもなく30年以上前のものです。が、ここ数年で爆発的に使われ始めてます)

発想：計算したい問題がある

- ・紙とペンで計算

↓ (大きな計算はできない)

- ・問題を書いたら全自動で解いてくれる夢の計算機

↓ (一部の問題に限れば可能。でも一般には無理)

- ・人間がアルゴリズムをプログラミング！ (大成功！)

発想：証明したい問題がある

- ・ 紙とペンで証明
↓ (大きな証明はできない/間違うかも)
- ・ 問題を書いたら全自動で証明してくれる夢の計算機
↓ (一部の問題なら可能。でも一般には無理)
- ・ 人間が証明をプログラミング！ (大成功！?)

証明をプログラミングする！
～Proof meets Programming～

証明が書ける言語の雰囲気

```
int[] qsort( int[] xs ) {  
    if( xs.size <= 1 )  
        return xs  
    else {  
        int[] ls = xs[1..-1].select{|x|x<xs[0]}  
        int[] gs = xs[1..-1].select{|x|x>=xs[0]}  
        return qsort(ls) + [xs[0]] + qsort(gs)  
    }  
}
```

※コードはイメージです : (Ruby+C) ÷ 2 くらいの仮想言語

“qsort の返値は必ずソート済み(小さい順に並んでる)”を「証明」してみる

証明が書ける言語の雰囲気

```
(int[], sorted(.)) qsort( int[] xs ) {  
  if( xs.size <= 1 )  
    return xs  
  else {  
    int[] ls = xs[1..-1].select{|x|x<xs[0]}  
    int[] gs = xs[1..-1].select{|x|x>=xs[0]}  
    return qsort(ls) + [xs[0]] + qsort(gs)  
  }  
}
```

関数の型を、「配列を返す」から
「配列と、その配列がソート済みなことの証明」
のペアを返すように変更

証明が書ける言語の雰囲気

```
(int[], sorted(.)) qsort( int[] xs ) {  
  if( xs.size <= 1 )  
    return (xs, “個数が1個以下なら当然sorted”)  
  else {  
    int[] ls = xs[1..-1].select{|x|x<xs[0]}  
    int[] gs = xs[1..-1].select{|x|x>=xs[0]}  
    return qsort(ls) + [xs[0]] + qsort(gs)  
  }  
}
```

サイズが 1 以下の場合
配列と、sortedなことの証明を返すようにする。

証明が書ける言語の雰囲気

```
(int[], sorted(.)) qsort( int[] xs ) {  
  if( xs.size <= 1 )  
    return (xs, “個数が1個以下なら当然sorted”)  
  else {  
    int[] ls = xs[1..-1].select{|x|x<xs[0]}  
    int[] gs = xs[1..-1].select{|x|x>=xs[0]}  
    (int[] lss, sorted(.)) lsp) = qsort(ls)  
    (int[] gss, sorted(.)) gsp) = qsort(gs)  
    return (lss + [xs[0]] + gss, ???)  
  }  
}
```

qsort の再帰呼びは配列と証明のペアを返します
あとはこれを使って全体がsortedな証明を作る！

証明が書ける言語の雰囲気

あとはこれを使って全体がsortedな証明を作る！

```
(int[] lss, sorted(⋅) lsp) = qsort(ls)
(int[] gss, sorted(⋅) lsp) = qsort(gs)

return (lss + [xs[0]] + gss, ???)
```

どうやって？

今作ってある証明は

- 「lssはソート済み」と「gssはソート済み」
だけ

証明が書ける言語の雰囲気

あとはこれを使って全体がsortedな証明を作る！

```
(int[] lss, sorted(.) lsp) = qsort(ls);  
(int[] gss, sorted(.) lsp) = qsort(gs);  
return (lss + [xs[0]] + gss, “ソート済みな  
配列とソート済みな配列をつなげたらソート済みに  
決まってるじゃん。根拠→”(lsp, gsp) )
```

↑ ...

証明が書ける言語の雰囲気

あとはこれを使って全体がsortedな証明を作る！

```
(int[] lss, sorted(.) lsp) = qsort(ls)
(int[] gss, sorted(.) lsp) = qsort(gs)
return (lss + [xs[0]] + gss, “ソート済みな
配列とソート済みな配列をつなげたらソート済みに
決まってるじゃん。根拠→”(lsp, gsp) )
```

↑ 残念、間違いです！！！！

例: [4,5,6] + [1,2,3] == [4,5,6,1,2,3]

証明が書ける言語の雰囲気

lss は xs[0] より小さくて、gss は xs[0] 以上
なんだから、そんなことにはならないよ！

```
(int[] ls, small(..) lsmall) =  
  xs[1..-1].select{|x|x<xs[0]}
```

```
(int[] gs, big(..) gbig) =  
  xs[1..-1].select{|x|x>=xs[0]}
```

```
(int[] lss, sorted(.) lsp) = qsort(ls)
```

```
(int[] gss, sorted(.) gsp) = qsort(gs)
```

```
return (lss + [xs[0]] + gss, “xs[0]より  
小さいソート済みの配列とxs[0]以上の  
ソート済み配列を繋げたらソート済みに決まってる  
じゃん。根拠→”(lsmall, gbig, lsp, gsp))
```

矢印は
一部
省略...


証明が書ける言語の雰囲気

```
(int[] ls, small(..) lsSmall) =  
  xs[1..-1].select{|x|x<xs[0]}  
(int[] gs, big(..) gsBig) =  
  xs[1..-1].select{|x|x>=xs[0]}  
(int[] lss, sorted(.) lsp) = qsort(ls)  
(int[] gss, sorted(.) gsp) = qsort(gs)  
return (lss + [xs[0]] + gss, "xs[0]より  
小さいソート済みの配列とxs[0]以上の  
ソート済み配列を繋げたらソート済みに決まってる  
じゃん。根拠→"(lsSmall, gsBig, lsp, gsp))
```

↑ 残念、まだ不完全です!!! lsがxs[0]より小さいからって、lssがxs[0]より小さいとは限らないよね?

証明が書ける言語の雰囲気

qsort したって値の範囲が変わるわけないじゃん！！
→「証明」しないとわかんない！



```
(int[], sorted(.), sameRange(..)) qsort(int[] xs) {  
  if( xs.size <= 1 )  
    return xs  
  else {  
    int[] ls = xs[1..-1].select{|x|x<xs[0]}  
    int[] gs = xs[1..-1].select{|x|x>=xs[0]}  
    return qsort(ls) + [xs[0]] + qsort(gs)  
  }  
}
```

できました!

```
(int[], sorted(.), sameRange(..)) qsort(int[] xs) {  
  if( xs.size <= 1 )  
    return (xs, "1個以下なら当然ソート済み",  
           "引数そのまま返してるんだから値の範囲は当然同じ")  
  else {  
    (int[] ls, range(., min(.)~.) lsSmall)  
      = xs[1..-1].select{|x|x<xs[0]}  
    (int[] gs, range(., .~max(.)) gsBig)  
      = xs[1..-1].select{|x|x>=xs[0]}  
    (int[] lss, sorted lsp, sameRange sl) = qsort(ls)  
    (int[] gss, sorted gsp, sameRange sg) = qsort(gs)  
    return (lss + [xs[0]] + gss, "ちっちゃいソート済みと  
      大きいソート済みを繋げたらソート済み。証拠  
      →"(lsp, gsp, lsSmall, gsBig, sl, sg),  
           "元の範囲に収まってる配列をくっつけてもやっぱり  
      元の範囲に収まる"(lsSmall, gsBig, sl, sg))  
  }}
```

Good: 「紙とペン」より确实

- ・ 「証明が正しい」ことのチェックは全自動でできる！
 - 間違った証明を書いたらコンパイラが教えてくれる
- ・ C言語等の型チェックが全自動なのと同じ原理

Good: 人間の直感



```
(int[], sorted(.), sameRange(...)) qsort(int[] xs)
```

- 「sortedの証明には値の範囲が変わらないことの証明も同時に要る！」
 - のような発想は、コンピュータに全自動でやらせるのはなかなか難しい
(※これくらいならできるかもですが)
 - こういうところをプログラマの発想で補いつつ「正しい」証明を書ける

Good: 大規模証明開発!

- ・ 証明支援環境
 - 「あとこれとこれとこれを証明すれば全部証明終わるよ」と教えてくれる
 - 簡単な証明は補完機能で勝手に書いてくれる
 - 使えそうな定理のリストアップ機能
- ・ プログラミングといっしょ
 - 「証明」も普通のプログラムなので関数に分けたりモジュール化したり...

Good: 証明もオブジェクト

- `int* bsearch(int[] xs, int key)`

↓ 二分探索する関数：

↓ コメントに”ソート済み配列にしか

使えないよ”と書いておく

↓

- `int*`
`bsearch`
`(int[] xs, sorted(.) prf, int key)`

↑ ソート済みしか受け取らん！とコードで語る

Bad: 書きにくい

- ・ それでもまだ凄くプログラミングがめんどくさい
 - 研究者自身にとっても大変
 - 今までサラッと流してた部分が厳密にプログラム化すると実は面倒だったり
- ・ 画期的な証明記述言語/ライブラリが求められています！

利用例: プログラム powered by 証明

- ・ コンパイル結果が正しいことの証明つきコンパイラ
- ・ 暗号プロトコルがアタックされにくいことの証明付き暗号エンコーダ
- ・ ...

さっき+twitterに貼られてた

- <http://ertos.nicta.com.au/research/14.verified/>

twitter

ホーム プロフィール 友だちを検索 設定 ヘルプ ログアウト

NICTA claims to be the world's first to develop a formal machine-checked proof of a general-purpose OS

<http://tinyurl.com/nrlupv>

約2時間前 TwitterFox で

m0h1can

モヒカンではない

利用例: 証明 powered by プログラム

・ wikipedia 「四色問題」

- いかなる地図も、隣接する領域が異なる色になるように塗るには4色あれば十分だという定理
- 1976年に ケネス・アッペル (Kenneth Appel) とヴォルフガング・ハーケン (Wolfgang Haken) はコンピュータを利用して、本定理を証明した。しかし、あまりに複雑なプログラムのため他人による検証が困難であることや、プログラム自体の誤りの可能性を考慮して、この証明を疑問視する声があった。

利用例: 証明 powered by フログラム

- G. Gonthier, “A Computer-Checked Proof of the Four Color Theorem”
 - 四色定理のコンピュータプログラムを使った証明を、証明が書ける言語「Coq」で全部書いた
 - (Coqの型チェックが正しいと信頼できるなら) 確実に正しい証明

参考資料：証明支援言語の例 (検索用キーワード)

・実装

- Agda
 - Epigram
 - Ω mega (Omega)
 - Coq
 - Isabelle/HOL
-
- 証明プログラムを直接書く系
- 証明スクリプト系

・理論

- Dependent Type
- Calculus of (Inductive) Construction

まとめ

- ・ 「証明できる
プログラミング言語」
を使って、
今までは正しさを「証明」
できなかったような
巨大プログラムの正しさを
証明するのがブームです