ゼミ: Types and Programming Language

久野　靖*

2009.9.11

Benjamin C. Pierce, Types and Programming Language, MIT Press, 2002.

0　Preface

□ 型システムの研究「と」型理論的側面からのプログラミング言語の研究

- 重要な分野 in ソフトウェア工学、言語設計、高性能コンパイラ、セキュリティ
- 本書→この分野に関する基本的な定義、結果、手法の「わかりやすい」解説

0.1　聴衆

□ この分野の大学院生研究者 + CS の他分野の院生+できる学部生

- このため、丁寧にひととおり解説、十分な入門材料、練習問題
- 大学院の教科書として、ないしゼミの材料として適切

0.2　目標

□ 網羅性→ひととおりのコアトピックをカバー

□ 実用性→「プログラミング言語の」型システムに集中 (型理論は深くない) (各言語機能について、「動機」「安全性」「実装」をとりあげる)

□ 多様性→なるべく多くの範囲をカバー (でも広がりつつあるので…)

□ 使いやすさ→練習問題はできるだけ解答、依存関係は明示、等

□ 正直→すべての型システムは実装されている

□ (これらのために犠牲になったかもしれないもの) →完全性、効率、…

0.3　構造

□ Part I → 型のないシステム

- 基本的な概念や手法の紹介、型のないλ計算

□ Part II → 単純な型のλ計算 + 基本的な機能

- 積と和、レコード、可変型、参照、例外
- Tait's method による正規化 (option)

□ Part III → サブタイプの基本機構

- メタ理論、2 つのケーススタディ

□ Part IV → 再帰型

- iso-recursive、equi-recursive

□ Part V → 多態

- ML スタイルの型、System F、存在型、ADT、サブタイプと制約型

□ Part IV → 型オペレータ

- Fw、bounded qualification

□ 章の間の依存関係 → 図 P-1 (灰色は部分的な依存)

□ 各側面の取り上げ方:　動機→形式的定義→証明 →(別の章) より深い扱い

- 深い扱い:　メタ理論、型検査アルゴリズム、証明 (堅固さ、正しさ、終了性)、OCaml による実装

□ 例題の主要な出どころは OO 言語→ 4 つのケーススタディ

- 18 章→伝統的な命令型のクラスとオブジェクト
- 19 章→ Java に基づくコア計算
- 27 章→ bounded qualification による深化
- 32 章→純粋関数型言語のクラスとオブジェクト

□ 本がぶ厚すぎないように簡単なところでとどめた題材多数

- (以下は直接見てください)

*筑波大学ビジネス科学研究科

1 Introduction

1.1 Types in Computer Science

□ 今日のソフトウェア工学 → 正しさを保証するための「形式的手法 (formal methods)」の採用

- プログラムのふるまいに関する何らかの仕様 (暗黙/明示的) に照らして「正しいかどうか」

□ 一端には…強力なフレームワーク

- Hoare Logic、代数的仕様記述言語、時相論理、表示意味論
- 非常に一般的な「正しさの特性」を示すのに使えるが、繁雑/プログラマの負担

□ 他端には…

- 強力さは劣る代わりにコンパイラ、リンカ等に組み込める方式
- 理論に暗いプログラマでも使いこなせる方式
- 例: モデル検査器 (軽量な形式的手法) →チップやプロトコルの設計で有限状態システムの誤りを探索
- 例: 実行時モニタリング→実行中に仕様から逸脱したふるまいを検出
- 最も普及し確立: 型システム (type system)

□ 「型システム」の定義を

- 言語設計者や実装者のインフォーマルな使い方までカバーする定義は難しいが…

□ 「型システムとは、フレーズをそれが計算する値の種別に応じて分類することで、プログラムがある種のふるまいをしないことを示すような、取り扱いの容易な構文的手法」

□ 型システムは「プログラムに対する論証を行うツール」(本書では)

- 「型理論」だと、論理学、数学、哲学にまたがるより広い概念←例: ラッセルのパラドクスを回避するために 1900 年ころから形式化 (文献多数あり)
- CS の枠内でも→「プログラミング言語への適用 (本書)」と「理論的な側面 (純粋な型つきλ計算との関連など)」
- 用語とかも共通しているが重要な違い: 型つきλ計算では「適切に型付けできる計算は停止性を保証」、実用言語ではこれはまずない

□ 型システムは「項 (構文上のフレーズ) をそれが計算する値の種類に応じて分類」

- つまり、プログラム中の項の実行時のふるまいを静的に近似
- 項の型は合成的に計算←式の型はその部分式の型によって決まる

□ 「静的 (static)」を明示的につけることも

- コンパイル時の解析が前提 (cf. 動的な/潜在的な型)
- Scheme 等では、実行時の型タグ (type tag) で値の種別を区分
- 「動的な型」というのは命名がおかしい…「動的に検査」の方がよいが…
- 静的だということは→保守的
- まずいふるまいが無いことを証明 (あることを証明ではない) →実行させてみると OK なプログラムを拒絶することがある

```
if <complex test> then 5 else <type error>
```

- 保守性と表現力→型システムの設計において重要。より多くのプログラムをより正確に型付けしたい

□ 任意の悪い性質をチェックできるわけではない

- ある特定の (予め決まった) 悪いふるまいをもたないことだけを示す
- 例: 数値演算のオペランドは必ず数値
- 例: レシーバは必ず記述されたメソッドを持つ
- ×: 0 割り、配列の添字範囲外など

□ チェックされ無いことが保証されるふるまい: 「実行時の型エラー」

- その内容は (共通部分も多いが) 言語ごとに異なっている

□ チェックできるものは低レベルのものだけとは限らない

- 高レベルのモジュラー性 (ユーザが定義した抽象化の整合性) なども
- 情報隠蔽の侵害も実行時の型エラーの一種

□ 型チェッカはコンパイラやリンカに組み込み

- →自動的に動作する必要 (人手の介在はできない)
- アノテーションとかは可
- アノテーションは通常は「少し」だが、より高度なものもあってよい (そうなると型チェッカというより証明チェッカ?)
- Extended Static Checking (Deltefs 他 1998) → 「それなりの」アノテーションでかなり広いクラスの正しさに関する性質を証明可能

□ 自動的なだけでなく、効率も必要

- しかしどれくらいの効率でよいかは?
- ML の型チェッカでも病的なケースでは大きな計算時間
- 型割り当て問題が「決定不能」な言語 (ただし通常必要な場合にはすぐ終わるアルゴリズムがある)

1.2 What Type Systems Are Good For

1.2.1 Detecting Errors

□ 最大の利点は、ある種のエラーが早期に検出可能

- 早期に検出→すぐ修正
- 早期でないと→あちこち探したりする必要、出荷後だったりするかも?
- 実行時の検出よりも「ここだ」と正確に指示

□ 静的型検査で検出可能なエラーは非常に多い

- 強い型の言語を使うプログラマ→「型検査を通ればまず動く」と感じる (実際にそうする権利がある以上に?)
- 簡単な見落としを検出 (文字列を数値に変換し忘れなど)
- より深い概念エラーも検出 (複雑なケース分析で境界条件忘れ、科学計算で単位の混乱など) ←これらも型エラーとして現れがち
- この点は型システムの表現力や対象プログラムにもよる (複雑なデータ構造を扱うもの vs 単純な数値ばかりのもの)(後者も次元解析のような型システムがあると有益)

□ 型システムから最大の利得を得たいなら…

- →プログラマの配慮と言語機構を活用とする意思が必要
- 例: 複雑なデータ構造を全部リスト型にする vs それらを別の型や抽象データ型にする

□ 型チェッカはある種のプログラムに対して保守ツールともなる

- 例: 複雑なデータの定義を変更→当該箇所を手で探さなくても、コンパイルすれば変更箇所を列挙してくれる

1.2.2 Abstraction

□ 型システムは統率のとれたプログラミングを強制することでプログラミングをサポート

- 大規模ソフトウェア開発では、部品どうしをパッケージし統合するモジュール言語の骨格となる
- 型はモジュールのインタフェースとなる → モジュールの機能のサマリ、実装者と使用者の部分的な契約

□ 大規模なシステムを明確なインタフェースを持つモジュールに基づいて構造化→より抽象化された設計スタイル→よりよい設計

- インタフェースは最終的な実装とは独立に検討

1.2.3 Documentation

□ 型があるとプログラムを読む手助けに

- 手続きのヘッダやモジュールインタフェースの型宣言はドキュメントであり、コードのふるまいのヒントを提供
- コメントと異なり、型情報は「古い」ということがない (毎回コンパイラが検査)

1.2.4 Language Safety

□ 「安全な言語」…「型システム」よりもさらにバラバラな意見

- 誰でも「安全な言語」と言われると「ああ、あれね」と思うが実は自分がどの言語コミュニティに属しているかに影響される
- インフォーマルには「プログラミングをしていて自分の足を撃たない」言語

□ もう少し改良:「安全な言語とは、自身の抽象化を保護する言語」

- 高水準言語であればマシンの機能を抽象化
- →その抽象化の整合性を保証できる言語、プログラマが定義機能を使って定義した高レベルの抽象化の整合性を保証できる言語
- 例: 配列〜メモリの抽象化。プログラマは明示的に配列に書き込むことによってのみその内容が変化するものと期待 (他のデータ構造の範囲外に書き込んだことで変化するのは×)
- レキシカルスコープの変数はそのスコープ内からだけアクセス可能と期待
- 呼び出しスタックは本当にスタックとしての動作を行うと期待
- 安全な言語→これらの抽象化は抽象的なまま使用可能

- 安全でない言語→ふるまいを完全に理解するためには、内部を熟知

□ 「言語の安全性」は静的な型安全と同じではない

- 安全性は静的なチェックに加え、動的なチェック (不正な動作を起こそうとしたところで止める) でも達成可能
- 例: Scheme は静的な型システムを持たないが安全
- 逆に、安全でない言語は「ベストエフォート型」の静的型検査を提供 (プログラマが明らかな間違いを侵したときにその除去を助ける)
- しかし「適切に型付けされたプログラムはまずい動作を起こさない」という保証は得られない
- コンパイラは実行時の型エラーが起こる可能性を指摘してくれるだけ (これでも無いよりはずっとましだが)

□ 安全性と静的/動的検査の表

```
      |  Staticall chk.  | Dynamically chk.
------+------------------+--------------------
Safe  | ML/Haskell/Java..| Lisp/Scheme/Perl/PS...
Unsafe| C/C++...         |
```

- なぜ「右下」が空? ← 実行時に「ほとんどの」操作をチェックするようにしたら、「全部」チェックするのもすぐできる
- 「右下」の言語もなくはない ← PC 用 Basic など

□ 一方、静的検査「だけ」で安全を達成することは通常できない

- 例: 配列の添字検査 (動的)
- 例: 堅固でない型検査を採用する場合 (Java のダウンキャストなど)

□ 安全は絶対的でない←安全な言語でも「抜け道」を用意する場合

- OCaml の Obj.magic、SML-NJ の Unsafe.cast
- Modula-3、C ♯ →「安全でないサブ言語」(GC など低レベル機能を実装するために使用 --- とくに unsafe と指定したモジュールのみ)

□ Cardelli --- 実行時エラーについて trapped / untrapped の区分

- trapped --- エラーの時点で停止ないし例外を投げるなど対処
- untrapped --- そのまま続行 (C の配列アクセスなど)
- この区分では、安全な言語とは実行時の untrapped エラーを回避できる言語

□ ポータビリティの視点「安全な言語とは、programmer's manual で完全に定義されている言語」

- 「言語の定義」: プログラマがその言語によるあらゆるふるまいを予期するのに必要なことがらのすべて
- C 言語などのマニュアルは合致しない (配列範囲外アクセスなどのふるまいを予期するにはメモリ配置やハードの動作まで知る必要)
- Java、Scheme、ML などではすべて書かれている --- 型検査を通ったプログラムはどこの (正しい) 実装でも同じ結果

1.2.5 Efficiency

□ CS における最初期の型システム (1950's、FORTRAN など) → 数値計算の効率を高めるため (整数と実数を区別し適切な演算命令を生成)

□ 安全な言語 → 静的に「起こり得ない」と分かる条件のチェックを省略することで高速化

□ 今日の高性能コンパイラ → 型チェッカが集めた情報を最適化のために多く利用 (言語に型システムの無い場合でさえ)

□ 型情報に基づく効率化は意外なところからも来る

- 並列科学技術計算コードのポインタ表現を型情報に基づいて改良
- Titanium 言語→型推論によりポインタのスコープを判断し、手で並列化したものよりよい性能を達成
- ML Kit コンパイラ→領域推論によりヒープ利用の多く (プログラムによっては全て) を GC 不要なスタック利用に置き換え

1.2.6 Further Applications

□ 伝統的な用途 (プログラミング言語) 以外の型の用途も拡大

□ コンピュータやネットワークのセキュリティが重要に

- 静的な型は Java や Jini のセキュリティモデルのコア
- セキュリティコミュニティの基本的なアイデアの多くは型の視点で見直し可能
- プログラミング言語理論の成果をセキュリティ分野に適用する動きも

□ コンパイラ以外の言語ツールへの利用

- COBOL プログラムの Y2K 対応変換ツール→ ML ふうの型推論を利用

- 別名解析や例外解析にも型推論手法を利用

□ 自動定理証明の分野→依存型を持つ強力な型システムが言明や証明の表現に利用される

- Nuprl, Lego, Coq, Alf など代表的な証明ツールが型理論を土台としている

□ データベース分野でも型に対する関心

- DTD、メタデータ、XML-Schema なども型を扱う

□ 計算言語学方面での応用←型付きλ計算に基盤

1.3 Type System and Language Design

□ 型を考えていない言語に型システムを導入→難しい。最初から型を前提に言語を設計するべき

- 理由: 型システムなしの言語だと (安全な言語であっても) 型検査が困難/不能な機能を持ってしまっていることが多い
- 静的型検査の言語では、型が言語設計の基盤となって設計が進む
- 構文も、型を指定する分だけ複雑になるので、最初からそのための構文で設計した方がよい
- 型が組み込まれているからといって、プログラマが型を記述しなければならないとは限らない (←型推論)
- でも、どれくらい型の記述をした方がいい/しなくてもいいという点では見解の相違 (ML 派 --- 少ない方がいい、C/Java 派 --- 多くてもいい)

1.4 Capsule History

□ 図 1-1: 型システムの理論において重要なできごと

- 最初期 --- 整数と実数の区分のための型
- 1950s〜1960s --- 構造型 (配列/レコード)、高階関数
- 1970s --- 型パラメタ、ADT、モジュール、サブタイプ→型理論の発達、数学/論理学との関連の発見

1.5 Related Readings

□ (省略)

2 Mathematical Preliminaries

□ 記法と基本的な数学的事実について。多くの人は飛ばしてよい

2.1 Sets, Relations, and Functions

2.1.1 集合の記法

2.1.2 自然数

2.1.3 N 項関係

2.1.4 1 項関係 (述語)

2.1.5 2 項関係

2.1.6 中置記法

2.1.7 ドメイン

2.1.8 部分関数、全域関数

2.1.9 関数の定義域、fail

2.1.10 2 項関係が述語を保存

2.2 Ordered Sets

2.2.1 2 項関係 R が反射的、対称的、推移的、反対称的

2.2.2 半順序、半順序集合、部分順序、全順序

2.2.3 join (最小上界)、meet (最大下界)

2.2.4 等値関係

2.2.5 反射閉包、推移閉包

2.2.6 練習問題**

2.2.7 練習問題**

2.2.8 練習問題**

2.2.9 下方連鎖

2.2.10 well founded (整礎)

2.3 Sequences

2.3.1 列、cons、append、入れ替え

2.4 Induction

2.4.1 AXIOM(自然数に基づく機能法)

2.4.2 AXIOM(自然数に基づく機能法)

2.4.3 字句順序 (辞書順序)

2.4.4 AXIOM(字句順序に基づく機能法

2.5 Background Reading