第 1 章

はじめに

1.1 計算機科学における型

現代のソフトウェア工学において、多種多様な**形式手法**は、システムが期待された動作の（明示的または暗黙の）仕様に照らして正しく振る舞うことを保証する手助けになるものと評価されている。広範に及ぶ形式手法の一端は、Hoare 論理、代数的仕様記述言語、様相論理、表示的意味論といった強力な枠組みである。これらは、ソフトウェアの多種多様な正しさを記述できるが、利用するのはしばしば煩雑で、プログラマ側にかなり高度な知識を要求する。一方、それよりずっと控えめであるが、控えめであるゆえに、自動検査器をコンパイラやリンカ、あるいはプログラム解析器に埋め込むことができ、背景となっている理論に馴染みのないプログラマでも使えるような技術がある。このような**軽量形式手法**のよく知られた例に、チップ設計や通信プロトコルといった有限状態システムのエラーを探索するツールである**モデル検査器**と呼ばれるツールがある。別の現在普及しつつある例としては**実行時モニタリング**と呼ばれる一連の技術があり、これらはシステムの一部が仕様通りに動いていないことを動的に検知する。しかし、今のところそれらより遥かに普及していて、最も確立された軽量形式手法は、本書で中心的に扱う**型システム**である。

「型システム」とは何か、プログラミング言語の設計者や開発者たちが日常的に口にする際の意味合いをすべて反映し、なおかつ十分に意味のある具体的な定義を与えるのが難しいのは、複数の大きなコミュニティが共有する多くの用語と同様である。しかし、次のような定義はできよう。

> **型システム**とは、プログラムの各部分を、それが計算する値の種類に沿って分類することにより、プログラムがある種の振る舞いを起こさないことを保証する、計算量的に扱いやすい構文的手法である。

幾つか補足すべき点がある。まず、この定義は型システムを、**プログラム**に関する推論のためのツールだとみなしている。このような用語の使い方は、本書が、プログラミング言語で用いられる型システムに関するものであるという方向性を反映している。より一般的には、**型システム**（あるいは**型理論**）という用語は、論理学や数学、哲学の、もっと広範な研究分野に言及するものである。この意味での型システムは、1900 年代初頭、Russell のパラドックス [424] など、数学の基盤を揺るがした論理的パラドックスを回避する手段として、最初に形式化された。20 世紀を通じて、型は論理学、特に証明論（Gandy [167] や Hindley [209] を見よ）において標準的に用いられる道具となり、また哲学や科学の言語として浸透していった。この分野における主要な成果として、原点である Russell の**分岐階型理論** [493]、Ramsey による**単純階型理論** [394]（これは Church の単純型付きラムダ計算 [100] の基礎となった）、Martin-Löf の**構成的型理論** [299; 301]、Berardi、Terlouw、Barendregt による**純粋型システム** [43; 458; 40] などがある。

計算機科学に限っても型システムの研究には二つの大きな分野がある。より実践的なほうはプログラミング言語への応用を扱う分野で、本書で中心的に扱うのもこちらの分野である。もう一方のより抽象的な分野では、**Curry-Howard 対応**（9.4 節）を通じて、さまざまな「純粋型付きラムダ計算」とさまざまな論理との間のつながりに着目する。どちらの分野においても同様の概念や表記法、技法が用いられるが、方向性で大きく異なる点が幾つかある。例えば、型付きラムダ計算の研

究では通常、正しく型付けされた計算はすべて停止することが保証されるような体系を考える。一方、ほとんどのプログラミング言語では、再帰的関数定義のような機能のため、停止性を犠牲にしている。

もう一つ上述の型システムの定義で重要なのは、実行時に計算される値の性質によって項（構文的表現）を**分類**することに重きが置かれている点である。型システムは、プログラム中の項が実行時にどう振る舞うかの**静的な**近似を求めるものとみなせる。（さらに言えば、項に関連付けられた型は、一般に式の型がその部分式の型にだけ依存するよう**合成的に**求められる。）

「静的」という語を明示的に付加することがある。例えば、「静的型付きプログラミング言語」という言い方をする。これは、本書で考察しているような種類のコンパイル時解析と、Scheme（Sussman and Steele [450]; Kelsey, Clinger, and Rees [252]; Dybvig [142]）のような言語で使われる**動的型付け**（あるいは**潜在的型付け**）とを区別するためである。後者ではヒープ中の異なる種類の構造を区別するのに実行時の**型タグ**が利用される。「動的型付けされる」といった言い回しは誤っているといって差し支えなく、おそらく「動的検査される」と言い換えるべきであるが、標準的に使われる用語法である。

静的であるがゆえに、型システムは必然的に**保守的**でもある。つまり、プログラムがある種の振る舞いを**起こさない**ことは断定的に証明できるが、起こすことは証明できない。それゆえ、時として実際には実行時に正しく振る舞うプログラムを拒絶してしまう。例えば、

```
if <複雑な条件> then 5 else <型エラー>
```

のようなプログラムは、正しく型付けできないとして拒絶されるだろう。たとえ <複雑な条件> が常に `true` に評価されるものであっても、静的解析ではそう判断できないからである。型システムの設計では、根本的に保守的性質と表現力との間の綱引きが避けられない。より多くのプログラムを（各部分により正確な型を与えて）型付けしたいという欲求が、この分野における研究の主要な牽引力である。

関連して、ほとんどの型システムで実現されている比較的単純な解析では、任意の望ましくないプログラムの挙動を排斥することはできない。正しく型付けされたプログラムには**ある**種の違反がない、ということしか保証できない。例えば、ほとんどの型システムでは、プリミティブな算術演算の引数が常に数であることや、メソッド呼び出し時にレシーバオブジェクトが要求されたメソッドを常に提供していることなどは静的に検査できるが、除算の二つ目の引数がゼロでない、あるいは配列へのアクセスが常に境界内に収まっている、などの検査はできない。

与えられた言語において型システムによって除去できる不正な振る舞いのことをしばしば**実行時型エラー**と呼ぶ。ここで留意すべきことは、何を不正な振る舞いとするかは、言語によって変わるという点である。実行時型エラーとされる振る舞いには、異なる言語同士で重なる部分も大きいが、どの型システムにも防ぎたい振る舞いの定義が付随する。各型システムの**安全性**（あるいは**健全性**）は、それぞれの実行時エラーの定義に照らし合わせて判断されねばならない。

型の解析で見つかる不正な振る舞いは、存在しないメソッドの呼び出しといった低水準な不具合に限らない。型システムはより高水準な**モジュール性**を強制することや、ユーザの定義した**抽象**の整合性を保護するのにも使われる。情報が正しく隠蔽されないこと、例えば内部表現が抽象的に扱われているデータのフィールドへの直接アクセスは実行時型エラーである。これは整数をポインタとして扱った結果、計算機をクラッシュさせるのと、全く同様である。

型検査器はコンパイラやリンカの中に構築されるのが一般的である。これはつまり、型検査器は**自動的に**、プログラマの介入や対話なしに動作しなければならないということである。すなわち、**計算量的に扱いやすい**解析でなければならない。しかしそれでも、プログラムに**型注釈**を明示することで、プログラマが手引きする余地もかなり残っている。こうした型注釈はプログラムの書きや

すさと読みやすさを考慮して比較的簡潔に保たれることが多い。しかし原理的には、プログラムが何らかの仕様を満たすことの完全な証明を型注釈で書き下すことも可能であろう。この場合、型検査器は事実上の**証明**検査器になる。拡張静的検査（Detlefs, Leino, Nelson, and Saxe [137]）といった技術は、型システムと、本格的なプログラム検証手法との間にそのような領域を設けようとしており、「適度に軽量な」プログラム注釈にのみ依存する、広い範囲の正当性を完全自動で検査する環境を実現する。

同じ理由から、原理上自動化可能であるだけでなく、型を検査する**効率的な**アルゴリズムを実際に備えている手法に最も興味がある。しかし、何をもって効率的とするかは議論の余地がある。MLの型システム（Damas and Milner [129]）のような広く使われているものですら、病的なケース（Henglein and Mairson [204]）では非常に長い時間を型検査に要する。型検査や型再構築が、決定不能であるが「実用上関心のあるほとんどの場合においては」素早く停止するアルゴリズムを備えた言語も存在する（Pierce and Turner [376]; Nadathur and Miller [333]; Pfenning [359]）。

1.2 型システムで何ができるか

エラーの検出

静的型検査の最もわかりやすい恩恵は、ある種のプログラミングのエラーを早期に検出できることである。早期に検出されたエラーはすぐに修正できるが、コード内に潜んでいて相当後に発見されるエラーはそうはいかない。その頃にはプログラマは別のことをしているし、プログラムの運用が開始された後の場合さえある。さらに、実行時よりも型検査時のほうがエラーの箇所を正確に特定しやすいことが多い。実行時では、何かおかしくなってからしばらく経った後でないと影響が見えてこないことがあるからだ。

実際、静的型検査は驚くほど広範囲のエラーを明らかにしてくれる。強力な型のある言語で開発を行うプログラマは、自分たちのプログラムがいったん型検査器を通れば「ちゃんと動く」傾向にあり、それが期待している以上によくあるとしばしば言う。考えられる説明としては、ただの不注意（平方根を取る前に文字列を数に変換し忘れる、など）だけでなく、もっと深い概念上の間違い（複雑な場合分けで境界条件を見落とす、あるいは科学計算で単位を勘違いする、など）が、しばしば型レベルの不整合となって現れる、ということが挙げられる。この効果の大きさは、型システムの表現力と取り組んでいるプログラミングの内容とに依存する。多様なデータ構造を扱うプログラム（コンパイラのような記号処理アプリケーションなど）は、科学系アプリケーションにおける数値計算のような数種類の単純な型しか使わないプログラムよりも型検査器から得られる効能が大きい（ただし、**次元解析**（Kennedy [253]）をサポートする詳細型システムは数値計算のようなプログラムにおいても極めて有用である）。

一般に、型システムの恩恵を最大限に享受するには、言語に用意された機構を活用する意思と併せてプログラマ側で意識すべきことがある。例えば、複雑なプログラムに現れるデータ構造をすべてリストで表現してしまうと、それらをそれぞれ異なるデータ型や抽象型で定義するのに比べて、コンパイラの援助は得られにくくなる。表現力の高い型システムは、データ構造に関する情報を型を使って表現するための数々の「仕掛け」を提供する。

ある種のプログラムにとって、型検査器はかけがえのない**保守**のためのツールともなる。例えば、複雑なデータ構造の定義を変更する必要があるとき、その構造に関連するソースコードで修正を要する箇所を探すのに大きなプログラム全体を手作業で調べる必要はない。データ型の定義を変更すれば、修正を要する箇所では必ず型の不整合が起こるので、コンパイラにかけて型検査が失敗する

箇所を調べるだけで列挙できる。

抽象化

型システムがプログラミングの工程に寄与する重要な点として、プログラミングの規律が強化されるということも挙げられる。特に大規模ソフトウェアを組み立てる際には、型システムは大規模システムのコンポーネントをパッケージ化し、つなぎ合わせる**モジュール言語**の基幹をなす。型はモジュール（あるいはクラスのような関連する構造）のインターフェイスに登場する。実際のところ、インターフェイス自身「モジュールの型」だとみなすことができ、モジュールが提供する機構の要約を提供する。これはモジュールの実装者と利用者との間の一種の契約ともいえる。

曖昧さのないインターフェイスを持つモジュールを用いて大規模システムを構成すれば、設計のスタイルがより抽象的になる。そのようなスタイルでは、インターフェイスを最終的な実装と切り離して設計、議論できる。インターフェイスをより抽象的に考えることは、一般に、より良い設計につながるものである。

ドキュメント化

型はプログラムを**読む**際にも有用である。手続きの先頭やモジュールインターフェイスの型宣言はある種のドキュメントを構成し、振る舞いを理解する有用なヒントになる。さらに、コンパイラが実行されるたびに検査されるので、コメント中に埋め込まれた記述とは違ってこのドキュメントは古くなることがない。型の役割は、モジュールのシグネチャにおいて特に重要である。

言語の安全性

「安全な言語」という言葉は、残念ながら「型システム」よりも議論を呼ぶ言葉である。字面でわかった気になる人も多いが、何をもって言語の安全性と考えるかは各人が属する言語のコミュニティから強く影響を受ける。とはいえ、平たく言えば、安全な言語とはプログラミングの最中にうっかり自分の足を撃つことが不可能な言語であると定義できる。

この直観的な定義を少し洗練すれば、**安全な言語とは自らがもたらす抽象を保護するものである**といえるだろう。いかなる高水準言語も、計算機が提供するサービスを抽象化する。安全性と言ったときに指し示しているのは、そのような抽象や、それより高位の抽象（プログラマが言語の機能を用いて定義するもの）の整合性を保証する、言語の能力である。例えばある言語で、計算機が備えるメモリの抽象として、配列の機能が参照と更新の操作付きで提供されているとしよう。この言語を使うプログラマは、配列が変更されうるのは明示的に更新の操作を行った場合のみであり、他のデータ構造の境界を越えてきたデータが書き込まれるようなことはないと期待するだろう。同様に、レキシカルスコープの変数はそのスコープ内でのみアクセス可能であると期待するし、呼び出しスタックは実際のスタックのような振る舞いをすると期待するだろう。計算機が提供するこうしたサービスの抽象を、安全な言語では**抽象的に**用いることができる。安全でない言語ではそれができず、プログラムがどう（誤）動作するかを完全に理解するのに、メモリ内のデータ構造のレイアウトやコンパイラが配置する順序といった低水準な詳細をプログラマがすべて意識していなければならない。極端な話、安全でない言語で書かれたプログラムは、そのプログラムのデータ構造を破壊しうるだけでなく、実行時システムのデータ構造も破壊しうる。その場合、何が起きても不思議ではない。

言語の安全性は静的型安全性と同じものではない。言語の安全性は静的型検査を使って**達成**しうるが、実行時検査により、おかしな操作を実行の瞬間に捕捉して、プログラムを止めるか例外を発

生させることにより達成することもできる。例えば、Scheme は静的型システムを持たないが安全な言語である。

一方、安全でない言語にも明らかな不注意の類いを除去するのに役立つ程度の「ベストエフォート」な静的型検査器が提供されることはよくあるが、我々の定義に従えばそのような言語は型安全であるとはいえない。正しく型付けされたプログラムが正しく振る舞うという**保証**を与えることが、一般的にはできないからである。それらの言語用の型検査器は、実行時型エラーが存在することを示唆してはくれるが（もちろん、何もしないよりはましである）、存在しないことを証明してはくれない。

	静的検査	動的検査
安全	ML、Haskell、Java など	Lisp、Scheme、Perl、PostScript など
安全でない	C、C++ など	

上の表で右下の欄が空白なのは、**ほとんどの**操作の安全性を実行時に強制する機構が既にあるなら、**すべての**操作を検査するために追加で必要なコストはほとんどないからである。（実際には、誤って用いれば実行時システムの整合性を破壊できるような、任意のメモリロケーションを読み書きできる低水準プリミティブを備えた動的検査言語も稀に存在する。例えば、最低限のオペレーティングシステムの上で動くマイクロコンピュータ向けの Basic の方言などである。）

実行時の安全性は通常、静的型付けだけでは達成できない。例えば、上記の表で安全だとした**すべての**言語は、**配列の境界検査**を動的に行う[†1]。同様に、静的検査言語であっても、あえて実際には健全ではない型検査規則を持つ操作（例えば Java のダウンキャスト操作。151 ページを見よ）を提供し、そうした構文が利用されている箇所を動的に検査することで言語の安全性を確保していることがある。

言語の安全性が絶対的であることは滅多にない。安全な言語がプログラマに「脱出用ハッチ」を準備していることは多い。安全でない可能性がある他の言語で書かれたコードを呼び出す外部関数呼び出しなどがそれにあたる。実際のところ、こうした脱出用ハッチが言語そのものの内部で制御された形で提供されることもある。OCaml（Leroy [280]）の `Obj.magic` や SML/NJ の `Unsafe.cast` などである。Modula-3（Cardelli et al. [83]; Nelson [338]）や C#（Wille [495]）はさらに進んで、ガベージコレクタといった低水準の実行時機構を実装することを意図した「安全でない部分言語」を提供している。この部分言語の特別な機能は、`unsafe` と明示的に印が付いたモジュールでしか使えないようになっている。

Cardelli [82] は少し違った観点から言語安全性についてまとめており、Cardelli が言うところの**捕捉される**実行時エラーと**捕捉されない**実行時エラーを区別している。捕捉されるエラーは計算を即座に止める（あるいは、そのプログラム内で楽に処理可能な例外を発生させる）が、捕捉されないエラーは計算を（少なくともしばらくの間）継続させる可能性がある。捕捉されないエラーの例としては、C のような言語で配列の境界を越えたデータアクセスがある。この観点においては、安全な言語とは、実行時に捕捉されないエラーを防止する言語のことである。

さらに別の観点では、可搬性に着目する。これは「安全な言語はプログラマ用マニュアルによって完全に定義される」というスローガンとして表現できる。ここで言語の**定義**とは、その言語で記述できるすべてのプログラムの振る舞いを予測するためにプログラマが理解しなければならない物事の集合だとしよう。すると、C のような言語のマニュアルは定義にならない。なぜなら、ある種

[†1] 配列の境界検査を静的になくしてしまうことは、型システム設計者の長年の目標である。原理的に必要な仕組み（**依存型**（30.5 節を見よ）に基づくもの）はよく理解されているが、表現力、型検査の予測可能性と計算量的な扱いやすさ、およびプログラム注釈の複雑さをうまく均衡させた形でその機能を言語に組み込むことは重要な課題として残されている。この分野に関する最近の進んだ研究成果は Xi and Pfenning [502; 503] に記載されている。

のプログラム（境界検査を行わない配列アクセスやポインタ算術が関与するもの）は、特定のCコンパイラによるデータ構造のメモリ配置といった詳細を知らなければ振る舞いを予測できず、別のコンパイラで実行すると、同じプログラムで振る舞いが全く異なる可能性があるからである。対照的に、Java、Scheme、MLのマニュアルは、その言語のすべてのプログラムの振る舞いの仕様を（厳密さの度合いは異なるが）正確に定める。正しく型付けされたプログラムは、これらの言語が正しく実装されている限り、常に同じ結果を返す。

効率性

計算機科学における最初の型システムは、1950年代初頭、Fortran（Backus [34]）などの言語で整数の算術式と実数の式とを区別し、数値演算の効率化を図るために導入された。型の導入により、コンパイラはプリミティブな演算に異なる表現を用い、適切な機械語命令を生成できるようになった。安全な言語では、安全性を保証するために必要な動的検査の多くを（常に成功することを静的に証明することで）除去し、さらなる効率改善を図ることができる。今日、ほとんどの高性能コンパイラは、最適化とコード生成のフェーズにおいて型検査器が収集する情報に大きく依存している。型システム**そのもの**は持たない言語のコンパイラにしても、多大な苦労をしてこの種の型情報を近似的に復元している。

型情報による効率改善は意外な部分からも得られることがある。例えば近年、型解析から生成される情報を使うことで、科学計算の並列プログラムではコード生成時の判断だけでなくポインタ表現も改善できることが示された。Titanium言語（Yelick et al. [506]）は、型推論の技術を用いてポインタのスコープを解析し、それをもとに、プログラマが明示的にプログラムを手動でチューニングするより有意に良い判断を可能としている。ML Kitコンパイラは強力な**リージョン推論**アルゴリズム（Gifford, Jouvelot, Lucassen, and Sheldon [176]; Jouvelot and Gifford [244]; Talpin and Jouvelot [454]; Tofte and Talpin [465; 466]; Tofte and Birkedal [463]）を使い、スタックベースのメモリ管理をすることで、ガベージコレクションをほとんど（プログラムによっては完全に）不要にしている。

その他の応用例

型システムは、プログラミングや言語設計における従来の利用方法の枠を越え、計算機科学やその関連領域においてさまざまな具体的方法で応用され始めている。ここでその一部を概説する。

型システムの応用分野として重要さが増しているのが、計算機やネットワークのセキュリティである。例えば、静的型付けはJavaとJINIのネットワーク機器向け「プラグアンドプレイ」アーキテクチャ（Arnold et al. [24]）のセキュリティモデルの核になっており、また証明付きコード（Proof-Carrying Code）（Necula and Lee [336; 337]; Necula [335]）を可能にする重要な技術である[†2]。同時に、セキュリティの分野で開拓された多くの基盤的なアイディアが、プログラミング言語の文脈で、しばしば型解析の形で再び研究の対象となろうとしている（例えばAbadi, Banerjee, Heintze, and Riecke [2]; Abadi [1]; Leroy and Rouaix [283]など）。逆に、プログラミング言語理論をセキュリティの領域の問題（例えばAbadi [1]やSumii and Pierce [449]）に直接応用することが注目を浴びつつある。

型検査・推論のアルゴリズムはコンパイラ以外のプログラム解析ツールでも多用されている。例えばAnnoDominiはCOBOLプログラムの2000年問題対応変換ユーティリティだが、ML風の型推

[†2] 著者からの補足：より最近の例としては、C#およびCLR、Andrew Myersによるセキュリティ型付きJava（JFlow、JIF、SIF、Fabric）、およびMicrosoft社のF*などがある。

論エンジン（Eidorff et al. [143]）がもとになっている。型推論の技術は別名解析（O'Callahan and Jackson [340]）や例外解析（Leroy and Pessaux [282]）のためのツールにも使われている。

自動定理証明においては、論理的性質や証明を表現するために型システム（通常は依存型に基づく非常に強力なもの）が使われる。NuPRL（Constable et al. [110]）、Lego（Luo and Pollack [291]; Pollack [390]）、Coq（Barras et al. [41]）、Alf（Magnusson and Nordström [297]）といった、ポピュラーな対話的証明系は型理論を直接用いている。Constable [109] と Pfenning [361] で、これらの体系の歴史を議論している。

データベースの分野においても、Document Type Definition（文書型定義、DTD）（XML 1998 [504]）や他のスキーマ（XML Schema 標準（XS 2000 [505]）など）といった、構造化されたデータを XML で記述する「ウェブ・メタデータ」の爆発的な広まりと共に、型システムへの興味が増加傾向にある。XML へのクエリ・操作を行う新しい言語は、これらのスキーマ言語に直接基づいた、強力な静的型システムを提供している（Hosoya and Pierce [221]; Hosoya, Vouillon, and Pierce [222]; Hosoya and Pierce [219]; Relax [398]; Shields [433]）。

型システムの、かなり毛色の異なる応用例として、計算言語学の分野に登場するものがある。型付きラムダ計算は**範疇文法**（van Benthem [473]; van Benthem and Meulen [474]; Ranta [395] など）の形式化の基礎をなす。

1.3 型システムと言語設計

型検査を考慮して設計されていない言語に型システムを組み込むのは困難な問題である。理想的には、言語設計は型システムの設計と手を取り合って進めるべきである。

その理由の一つとしては、型システムのない言語では、たとえ安全な動的検査言語であっても、型検査を困難あるいは不可能にする機能が提供されていたり、そうしたプログラミングイディオムが推奨されたりする傾向にある点が挙げられる。実際、型付き言語では型システムそのものが設計の基盤であり、他のすべての側面がそれを踏まえて考慮される、中心的原理であるとされることもしばしばある。

もう一つの理由として、型付き言語の具象構文は型注釈の考慮が必要なため、型無し言語より複雑になる傾向がある、という点がある。きれいで包括的な構文設計は、すべての課題に同時に対処するほうがうまくいきやすい。

型がプログラミング言語にとって不可欠であるという主張は、プログラマが型注釈を物理的に書き下すべき場所はどこか、代わりにコンパイラによる推論が可能な場所はどこか、という議論とは別である。きちんと設計された静的型付き言語であれば、膨大な型情報の明示的で退屈な維持管理をプログラマに強いることは決してない。とはいえ、どの程度の量の型情報を過剰と考えるかは議論の余地がある。ML 系言語の設計者は、必要な情報の復元に型推論を使うことで、注釈が最小限で済むように大変な労力を費やしている。Java を含む C 系の言語は、もう少し冗長な流儀を選択している。

1.4 歴史の概要

計算機科学において、最初期の型システムは、（例えば Fortran における）整数と浮動小数点数の間の、非常に単純な区別をするために使われた。1950 年代終わりから 1960 年代初期にかけて、この分類はデータ構造（レコードの配列など）と高階関数にまで拡張された。1970 年代においては、幾つかのより高度な概念（パラメータ多相、抽象データ型、モジュールシステム、部分型付け）が導入され、型システムはそれ自身が一つの研究分野となった。同時に計算機科学者は、プログラミング言語に見られる型システムと数理論理学で研究されてきたそれとの間の関係に気づき始め、現在へと至る豊富な相互作用につながっている。

図 1-1 は、計算機科学における型システムの歴史で重要な点を短く（そして恥ずかしいほど不完全に）まとめた年表である。関連する論理学の発展についてはアスタリスク（*）付きで記述しており、この分野での貢献がいかに重要かを示すようにしている。右側にある参考文献は巻末の一覧で確認できる。

1.5 関連する文献

本書は自己完結することを目指してはいるが、すべてを包括しているとはとてもいえない。この分野は大変広く、多くの観点からのアプローチが可能なため、一冊の本だけで十分には取り扱えない。本節では良い入門となる資料を列挙する。

Cardelli によるハンドブックの記事 [82] や Mitchell によるもの [321] は、この分野への手軽な入門である。Barendregt の記事 [40] は数学的に少し高度である。Mitchell の分厚い教科書 “*Foundations for Programming Languages*” [322] はラムダ計算の基本と、さまざまな型システム、意味論の多くの観点を押さえている。実装上の問題よりは意味論に重点が置かれている。Reynolds の “*Theories of Programming Languages*” [420] は大学院レベルのプログラミング言語理論のサーベイで、多相性、部分型付け、交差型を見事に解説している。Schmidt による “*The Structure of Typed Programming Languages*” [428] は言語設計の文脈で型システムの核となる概念を解説している。幾つかの章では従来的な手続き型言語も対象にしている。Hindley の “*Basic Simple Type Theory*” [209] は型付きラムダ計算と、それに密接にかかわる体系に関する結果をうまくまとめている。内容は広範というより深く掘り下げられている。

Abadi と Cardelli の “*A Theory of Objects*” [4] は、本書と同じ題材を多く扱っているが、実装の側面にはあまり重きをおかず、代わりにオブジェクト指向プログラミングの基礎的な取り扱いにおけるこれらのアイディアの応用に集中している。Kim Bruce の “*Foundations of Object-Oriented Languages: Types and Semantics*” [60] は同様の問題を扱う。オブジェクト指向型システムの入門的な資料は Palsberg and Schwartzbach [350] と Castagna [92] にも含まれている。

型無し言語、型付き言語の双方に通じる意味論的な基礎付けは、Gunter [188] や Winskel [496]、Mitchell [322] といった教科書で詳細に扱っている。操作的意味論は Hennessy [205] でも詳細に扱っている。**圏論**という数学的枠組みの中で型の意味論の基礎を扱う文献は多数あり、Jacobs [235] や、Asperti and Longo [25]、Crole [125] などがある。その簡単な入門として、“*Basic Category Theory for Computer Scientists*” [364] がある。

Girard、Lafont、Taylor による “*Proofs and Types*” [179] は型システムの論理学的な側面（Curry-Howard 対応など）を扱っている。その中では System F を創造者自身が説明している。線形論理を紹介する付録もある。型と論理のつながりは Pfenning の “*Computation and Deduction*” でさらに読み進むことができる。Thompson の “*Type Theory and Functional Programming*” [460] と Turner の “*Constructive Foundations for Functional Languages*” [469] は関数的プログラミング

1870 年代	*形式論理学の起こり	Frege, 1879 [161]
1900 年代	*数学の形式化	Whitehead and Russell, 1910 [493]
1930 年代	*型無しラムダ計算	Church, 1941 [101]
1940 年代	*単純型付きラムダ計算	Church, 1940 [100]; Curry and Feys, 1958 [128]
1950 年代	Fortran	Backus, 1981 [34]
	Algol-60	Naur et al., 1963 [334]
1960 年代	*Automath プロジェクト	de Bruijn, 1980 [135]
	Simula	Birtwistle et al., 1979 [46]
	*Curry-Howard 対応	Howard, 1980 [224]
	Algol-68	van Wijngaarden et al., 1975 [476]
1970 年代	Pascal	Wirth, 1971 [497]
	*Martin-Löf 型理論	Martin-Löf, 1973 [299], 1982 [300]
	*System F、F^{ω}	Girard, 1972 [177]
	多相ラムダ計算	Reynolds, 1974 [408]
	CLU	Liskov et al., 1981 [288]
	多相型推論	Milner, 1978 [307]; Damas and Milner, 1982 [129]
	ML	Gordon, Milner, and Wadsworth, 1979 [183]
	*交差型	Coppo and Dezani-Ciancaglini, 1978 [115]; Coppo, Dezani-Ciancaglini, and Sallé, 1979 [116]; Pottinger, 1980 [392]
1980 年代	NuPRL プロジェクト	Constable et al., 1986 [110]
	部分型付け	Reynolds, 1980 [411]; Cardelli, 1984 [73]; Mitchell, 1984 [317]
	存在型としての ADT	Mitchell and Plotkin, 1988 [325]
	*Calculus of Constructions	Coquand, 1985 [117]; Coquand and Huet, 1988 [118]
	*線形論理	Girard, 1987 [178]; Girard et al., 1989 [179]
	有界量化	Cardelli and Wegner, 1985 [88]; Curien and Ghelli, 1992 [127]; Cardelli et al., 1994 [86]
	*Edinburgh Logical Framework	Harper, Honsell, and Plotkin, 1992 [195]
	Forsythe	Reynolds, 1988 [415]
	*純粋型システム	Terlouw, 1989 [458]; Berardi, 1988 [43]; Barendregt, 1991 [39]
	依存型とモジュール性	Burstall and Lampson, 1984 [68]; MacQueen, 1986 [294]
	Quest	Cardelli, 1991 [79]
	作用システム	Gifford et al., 1987 [176]; Talpin and Jouvelot, 1992 [454]
	列変数、拡張可能レコード	Wand, 1987 [487]; Rémy, 1989 [399]; Cardelli and Mitchell, 1991 [87]
1990 年代	高階部分型付け	Cardelli, 1990 [78]; Cardelli and Longo, 1991 [85]
	型付き中間言語	Tarditi, Morrisett, et al. 1996 [455]
	オブジェクト計算	Abadi and Cardelli, 1996 [4]
	透過型とモジュール性	Harper and Lillibridge, 1994 [196]; Leroy, 1994 [279]
	型付きアセンブリ言語	Morrisett et al., 1998 [328]

▲ 図 1-1. 計算機科学と論理学における型の歴史年表

（Haskell や Miranda のような「純粋関数的プログラミング」の意味）と構成的型理論の間の、論理学的な観点から見たつながりに着目している。証明論における関連トピックは、Goubault-Larrecq と Mackie の “*Proof Theory and Automated Deduction*” [185] で多数解説されている。論理学や哲学の中での型の歴史は、Constable [109]、Wadler [482]、Huet [229]、Pfenning [361] といった記事や、Laan の博士論文 [267]、Grattan-Guinness [186] や Sommaruga [438] といった書籍でより詳細に説明されている。

> 何らかのプログラミング言語が型健全であるという主張が誤っていて恥をかかないようにするには、相当量の注意深い分析が必要なことがわかる。結果として、型システムの分類、記述、研究が形式的な学問領域として浮上したのである。
> —*Luca Cardelli, 1996* [82]