動的言語型付けバトル

Dynamic Language Typings Battle

pixiv Inc.
USAMI Kenta

動的言語とは何か

動的プログラミング言語(dynamic programming languages)はプログラミング言語の一類型で、実行時(run-time)にプログラムの性質を制御したり、実行時まで決定が遅延される性質が強いプログラミング言語の総称です。

プログラムの実行時に何ができるか(どんな操作を許すか)は、プログラミング言語設計の重要な一要素であり、言語によってかなり大きく異なるといえます。

一般に動的言語といわれる言語にはLisp族, シェルスクリプト, AWK, Perl, Smalltalk, Python, Ruby, JavaScriptなどがあります。動的言語に特徴的な機能としては**eval**があります。これは文字列またはリストなどでソースコードを構築し、実行時に評価(evaluate)できると機能です。これは柔軟な処理ができる反面、不用意な実装によって脆弱性を孕むリスクがあるため敬遠される傾向にある一方、ライブラリ(スクリプト)の動的ロードなどは意識せず使われる場面も多いでしょう。

静的型付き vs 動的型チェック

静的型付き言語(statically typed languages)は静的、つまりプログラムを実行する前に関数・変数・項などのとりうる型が決定される言語のことです。

誤解しないでおきたいのは、静的型のプログラミング言語というのはソースコードに型をいちいち書かなければいけない言語、**というわけではありません。**

動的言語と呼ばれる言語では一般に静的には型を書かず(あるいは型付けの機能そのものを持たず)実行時まで型が定まりません。動的言語では実行時に値の型情報(型タグ)を検査することで安全に実行できます。このような性質を**型なし**、あるいは便宜的に**動的型付け**(dynamic typing)と呼ぶことがあります。

静的型付き言語は収集した情報をもとに効率的かつ実行効率のよい実行ファイルにコンパイルできる余地があります。**これは語弊のある表現ですが**、コンパイルされたプログラムは鉄道のようなもので、バスと違って道路信号で頻繁に一時停止させられることなく移動できるもの、のように考えられるかもしれませんね。

静的型にもグラデーションがある

静的型付き言語として分類される言語は多くありますが、ひとえに静的型付きと言っても言語が提供する型システムには大きな違いがあります。

型システムは、**健全性**（soundness）＝正しく型がついていればエラーなく実行できるか、**完全性**（completeness）＝プログラムの性質を適切に型で表現できるか、といった指標で評価できます。

一般に静的型付き言語と分類される言語でも、C++のように実行時型情報（RTTI）を提供して動的な処理を可能にする言語があります。C#もdynamic型を提供しています。これらの機能は、プログラムの柔軟性を上げることと引き換えに健全性が低下し、コンパイル時に検出できないエラーのリスクが生じます。

また、**代数的データ型**（ADT）という型同士を式のように扱える型も考えられます。これをサポートする言語の型システムでは、静的な型として簡潔かつ柔軟なコード記述が可能になります。

型推論の完全性

静的型付き言語と呼ばれる言語の中にも、型を明示しなくてもコンパイラがコードをきちんと調べ上げることで、静的な型を与えてくれるものがあります。

```
# let add x y = x + y;;        val add : int -> int -> int = <fun>
# let id a = a ;;              val id : 'a -> 'a = <fun>
```

これはOCamlというプログラミング言語の例ですが、きちんと関数に型がついている（add関数は二つのint引数をとってintを返す、id関数は引数と同じ型をそのまま返す）ことが確認できます。

すべての言語で同じように型がつけばハッピーなのですが、現実は厳しい。
特に動的言語ではevalなど動的な言語機能により推論困難な場面が多いです。
静的型付きの言語でも型システムの兼ね合いで完全なものは難しいのですが、
ローカル変数に型を明示する必要がない言語が近年浸透しつつあります。

動的言語、世にはばかる

動的言語の処理系は静的言語のコンパイラのように実行前にプログラムの型の整合性をチェックすることは(あまり)やりません。つまり型の健全性なんてあったものではなく、基本的にノーガード戦法をとっています。

どうしてそんな安全性も保障できない不安全なプログラミング言語が世の中に蔓延しているのでしょうか。コンパイラによる検査も無料ではなく、型システムが高度になり対象コードが増えるほどコンパイル時間が増加します。

動的言語は90年代から2000年代にかけて、コンパイル時間がなく型の知識も不要で軽快に実行できるストレスフリーな言語として普及した側面があります。特にWebアプリケーションではボトルネックがデータベースアクセスなどであり、言語による実行速度が生じない(あるいは、処理系の性能改善によって許容範囲内に抑えられている)ことが現代でも動的言語が許容される理由のひとつです。

なぜ動的言語に… 型がつかないのか

型推論という素晴しいものがあるなら、PHPのような動的言語のソースコードも丹念に解析してやればOCamlのような静的型付き言語と同等になるのではないか。……そう思っていた時代が私にもありました。

とても残念なことに動的言語にはそれを阻むダイナミックな性質がたくさんあります。evalやjson_decode()、unserialize()などの関数、include式による動的ロードなどは、原理上あらゆる型の値を返します。

PHPには存在しない(ことになっている)関数の再定義や、Rubyにはクラスではなくインスタンス固有の特異メソッドというものも存在します。PHP関数マニュアルを手がかりに型を付けても、実用にならない型に発散していくばかりです。

動的言語での演算子のオーバーロードは型推論と非常に相性が悪いです。PHPは基本はオーバーロードはない(ことになっている)のですが、問題があります。

動的言語でも静的解析したい

二項演算子 + が整数の加算のみをサポートしている言語を考えてみましょう。関数 add(x, y) => x + y (引数xとyをとって加算結果を返す)を定義するとき、add(1, false) のような呼び出しが間違っていることは明らかです。

静的な型検査(つまり実行する前にソースコードの整合性を調べる)ことで、そのような明らかに間違ったコードを見過さないようにできます。静的型付き言語の多くで実行ファイルのコンパイルの一部として型検査を実施することで、実行すると間違った動作をするプログラム実行ファイルを生成しないようにします。

これが先述の**健全性**という概念ですが、多くの動的言語ではほとんど何の期待もできませんでした。足し算をするadd(1, false)が間違ったコードだということは人間様の目には自明ですが、常に人間が整合性を監視し続けることは困難です。コード規模が大きくなるほど不整合を事前に確かめたくなるでしょう。いかに人間様が賢くても、24/365で人間の手を煩わせるわけにはいきません。

どうして動的言語に型をつけたがるの

ここまで説明した通り、動的言語の静的解析、特に型をつける簡単なことではありません。それでも型検査をしたくなるのはなぜでしょうか。

ひとつは、実行ファイルのコンパイルが必要なくとも実装中(コーディング中)のlinterや継続的インテグレーションの一環として静的解析することで、確実ではなくとも辻褄の合わない不安全な可能性を早めに検知したいという需要です。

もうひとつの動機としてはIDEなどのツールの支援を受けやすくして生産性を高めることも挙げられるでしょうか。動的言語ではスクリプトとしてコードそのものが再配布されることが多いため、パラメータや戻り値の型がわからなければマニュアルを読むかコードそのものを読み解け、という思想でもあります。

型が明示されていれば、そのような手間と迷いを減らせるほか、戻り値の型が適切に定義されていれば、静的解析に基いた入力補完で、コーディング作業をよりなめらかにできるかもしれません。

PHP vs 型

　PHPは紛れもなく**強い動的型**の言語であり変数に型はありませんが、関数・メソッドやプロパティには型宣言ができます。これは実行時に処理系が型をチェックしてくれるので、型宣言された文脈では確実に型が保障されるという有利な特徴があります。もちろん関数呼び出しの度に型を調べるので実行時コストはかかるのですが、一般的な用途で型宣言がボトルネックになることは多くないと考えられます。パフォーマンスをプロファイラで検査した上でボトルネックになることも考えられますが、そのような場合は単に型宣言を削除すると検査コストを削減できます。

　ただし私の持論では