Dynamically Typed Languages

Laurence Tratt laurie@tratt.net
Bournemouth University, Poole, Dorset, BH12 5BB, United Kingdom.

Mar 13, 2009

Dynamically typed languages such as Python and Ruby have experienced a rapid grown in popularity in recent times. However, there is much confusion as to what makes these languages interesting relative to statically typed languages, and little knowledge of their rich history. In this chapter I explore the general topic of dynamically typed languages, how they differ from statically typed languages, their history, and their defining features.

1 Introduction

As computing is often split into software and hardware, so programming languages are often split into dynamically and statically typed languages. The traditional, simplified, definition of dynamically typed languages are that they do not enforce or check type-safety at compile-time, deferring such checks until run-time. While factually true, this definition leaves out what makes dynamically typed languages interesting—for example that they lower development costs [Ous98] and provide the flexibility required by specific domains such as data processing [MD04].

For many people, dynamically typed languages are the youthful face of a new style of programming introduced in the past few years. In fact, they trace their roots back to the earliest days of high-level programming languages in the 1950's via Lisp [McC60]. Many important techniques have been pioneered in dynamically typed languages from lexical scoping [SJ75] to Just-In-Time (JIT) compilation [Ayc03], and they remain a breeding ground for new ideas.

Systems programming – seen as 'serious' and thus demanding statically typed languages – is often contrasted with scripting programming – seen as 'amateurish' and thus needing little more than dynamically typed languages [Ous98]. The derogative use of the term 'scripting' led to the creation of other terms such as 'latently typed' and 'lightweight languages' to avoid the associated stigma. In reality the absence or presence of static typing has a number of effects on the use and applicability of a language that simple comparisons ignore [Pau07]. So while some tasks such as low-level systems (e.g. operating systems), resource critical systems (e.g. databases), or safety critical systems (e.g. systems for nuclear reactors) benefit from the extra rigour of statically typed

languages, for many other systems the associated costs outweigh the benefits [Lou08]. Gradually, dynamically typed languages have come to be seen as a valid part of the software development toolkit, and not merely second-class citizens [SG97].

From the mainstream's perspective, dynamically typed languages have finally come of age. More than ever before, they are used to build widely used real-world systems, often for the web, but increasingly for domains that were previously the sole preserve of statically typed languages (e.g. [KG07]), often because of lower development costs and increased flexibility [MMMP90, Nor92].

It should be noted that since there is no central authority defining dynamically typed languages, there is great variation within those languages which are typically classified as dynamically typed languages; nevertheless all such languages share a great deal in common. In this chapter, I explore the general topic of dynamically typed languages, how they differ from statically typed languages, their history, and their defining features. The purpose of this chapter is not to be a cheer-leader for dynamically typed languages—it is my contention that both statically typed and dynamically typed languages are required for the increasingly broad range of tasks that software is put to. Rather this chapter aims to explain what dynamically typed languages are and, by extension, to show where they may and may not be useful.

2 Defining types

The lack of a widely understood definition of dynamically typed languages has resulted in many misunderstandings about what dynamic typing is. Perhaps because of this, alternative terms such as 'soft typing' are sometimes used instead. Earlier I gave the simplified, and oft-heard, definition that a dynamically typed language is one that does not check or enforce type-safety at compile-time. Inevitably this simplified definition does not capture everything it should—the subtleties and variations in the use of dynamic typing preclude a short, precise definition.

In this section, I define various terms relating to dynamically typed languages, building up an increasingly accurate picture of what is meant by this term. Further reading on these topics can be found in [Car97, Pie02].

2.1 Types

At an abstract level, a type is a constraint which defines the set of valid values which *conform* to it. At the simplest level all apples conform to an 'Apples' type and all oranges to an 'Oranges' type. Types often define additional constraints: red apples are conformant to the 'Red Apples' type, whereas green apples are not. Types are typically organised into hierarchies, meaning that all apples which conform to the 'Red Apples' type also conform to the 'Apples' type but not necessarily vice versa.

In programming languages, types are typically used to both classify values, and to determine the valid operations for a given type. For example the `int` type in most programming language represents integers, upon which the operations `+`, `-`, and so on are valid. Most programming languages define a small number of built-in types, and allow

user programs to add new types to the system. While, abstractly, most types define an infinite set, many built-in programming language types represent finite sets; for example, in most languages the `int` type is tied to an underlying machine representation of n bits meaning that only a finite subset of integers conform to it.

In many Object Orientated (OO) programming languages, the notions of type and class are conflated. That is, a class 'Apple' which defines the attribute 'pip' and the operation 'peel' also implicitly defines a type of the same name to which instances of the class automatically conform to. Because classes do not always define types to which the classes' instances conform [CHC90, AC96], in this chapter I treat the two notions separately. This means that, abstractly, one must define a separate type 'Apples Type' to which instances of the 'Apple Class' conform to. This definition of types may seem unnecessarily abstract but, as shall be seen later, the notion of type is used in many different contexts.

2.2 Compile-time vs. run-time

In this chapter, I differentiate between errors which happen at *compile-time* and *run-time*. Compile-time errors are those which are determined by analyzing program code without executing it; run-time errors are those that occur during program execution.

Statically typed languages typically have clearly distinct compile-time and run-time phases, with program code converted by a compiler into a binary executable which is then run separately. In most dynamically typed languages (e.g. Converge, Perl, and Python) 'running' a file both compiles and executes it. The blurring, from an external perspective, of these two stages often leads to dynamically typed languages being incorrectly classified as 'interpreted' languages. Internally, most dynamically typed languages have distinct compilation and execution phases and therefore I use the terms compile-time and run-time identically for both statically and dynamically typed languages.

2.3 Static typing

Before defining what dynamic typing is, it is easiest to define its 'opposite'. Statically typed languages are those which define and enforce types at compile-time. Consider the following Java [GJSB00] code:

```
int i = 3;
String s = "4";
int x = i + s;
```

It uses two built-in Java types: `int` (representing integers) and `String` (Unicode character arrays). While a layman might expect that when this program is run, `x` will be set to `7`, the Java compiler refuses to compile this code; the compile-time error that results says that the `+` operation is not defined between values of type `int` and `String` (though see Section 2.6 to see why the opposite does in fact work). This is the essence of static typing: code which violates a type's definition is invalid and is not compiled. Such type related errors can thus never occur in run-time code.

2.3.1 Implicit type declarations

Many statically typed languages, such as Java, require the explicit static *declaration* of types. That is, whenever a type is used it must be declared before hand, hence `int i = 3` and so on.

It is often incorrectly assumed that all statically typed languages require explicit type declarations. Some statically typed languages can automatically infer the correct type of many expressions, requiring explicit declarations only when automatic inference by the compiler fails. For example, the following Haskell [Jon03] code gives an equivalent compile-time error message to its Java cousin, despite the fact that the types of `i` and `s` are not explicitly declared:

```
let
  i = 3
  s = "4"
in
  i + s
```

In this chapter, I define the term 'statically typed languages' to include both implicitly and explicitly statically typed languages.

2.3.2 Nominal and structural typing

As stated earlier, types are typically organised into hierarchies. There are two chief mechanisms for organising such hierarchies. Nominal typing, as found in languages such as Java, is when an explicit named relationship between two types is recorded; for example, a user explicitly stating that Oranges are a sub-type of Fruit. Structural typing, as found in languages such as Haskell, is when the components of two types allow a type system to automatically infer that they are related in some way. For example, the Orange type contains all the components of the Fruit type, plus an extra 'peel thickness' component—a structurally typed system will automatically infer that all Oranges are Fruits, but that opposite is not necessarily true. Sturctural typing as described here is only found in statically typed languages although a similar feature – duck typing – is found in dynamically typed languages (see Section 5.7).

2.4 Dynamic typing

Dynamic typing, at its simplest level, is when type checks are left until run-time. It is important to note that this is different than being *typeless*: both statically and dynamically typed languages are typed, the chief technical difference between them being *when* types are enforced. For example the following Converge [Tra07] code compiles correctly but when run, the `Int.+` function raises a run-time type exception `Expected arg 2 to be conformant to Number but got instance of String`:

```
i := 3
s := "4"
x := i + s
```

In this example one can trivially statically analyse the code and determine the eventual run-time error. However, in general, dynamically typed languages allow code which is more expressive than any current type system can statically check [CF91]. For example, in non-OO languages static type systems typically prevent an individual function from having multiple return points if each returns results of differing, incompatible, types. In OO languages, on the other hand, the compiler statically determines the set of methods (considering subtypes) that an object method call refers to; in dynamically typed languages the method lookup happens at run-time. This run-time lookup is known as *late binding* and allows objects to dynamically alter their behaviour, allowing greater flexibility in the manipulation of objects, the price being that lookups can fail as in the above example.

2.5 Safe and unsafe typing

Programs written with static types are often said to be *safe*[1] in the sense that type-related errors caught at compile-time can not occur at run-time. However most statically typed languages allow user programs to *cast* (i.e. force) values of one type to be considered as conformant to another type. For example, in C one can cast an underlying `int` value to be considered as an Orange, even if this is semantically nonsensical; instances of the two types are unlikely to share the same memory representation, and indeed may use different quantities of memory. Programs which abuse this feature can crash arbitrarily. Languages whose type systems can be completely overruled by the user are said to have an *unsafe* typing system.

In contrast to unsafe typing, languages with a safe type system do not allow the user to subvert it. This can be achieved either by disallowing casting (e.g. Haskell) or inserting run-time checks to ensure that casts do not subvert the type system (e.g. Java). For example, an object which conforms to the Red Apple type can always be cast to the Apple type. However objects which conform to the Apple type can only be cast to the Red Apple type if the object genuinely conforms to the Red Apple type (or one of its sub-types); attempting to cast a Green Apple object to the Red Apple type will cause a run-time check to fail and an exception to be raised.

The concept of safe and unsafe type systems is orthogonal to that of static and dynamic typing. Static type systems can be safe (Java) or unsafe (C); all dynamically typed languages of which I am aware are safe[2].

2.6 Implicit type conversions

In many languages – both statically and dynamically typed – a number of implicit type conversions (also known as 'coercions') are defined. This means that, in a given context, values of an 'incorrect' type are automatically converted into the 'correct' type.

[1] This terminology is not universal, with 'strong' and 'weak' sometimes used in place of 'safe' and 'unsafe' respectively.

[2] Note that assembly languages are often classified as dynamically and weakly typed; such languages fall considerably outside the scope of this chapter and are not considered herein.

	C	Converge	Haskell	Java	Perl	Python	Ruby
Compile-time type checking	●	○	●	●	○	○	○
Run-time type checking	○	●	○	●	●	●	●
Safe typing	○	●	●	●	●	●	●
Implicit typing	○	n/a	●	○	n/a	n/a	n/a
Structural typing	○	n/a	●	○	n/a	n/a	n/a
Run-time type errors	○	●	○	●	●	●	●
Implicit type conversions	●	○	○	●	●	○	●

Table 1: Language comparison with respect to typing ('n/a' meaning 'not applicable').

For example, in Perl [WCO00], the addition of a number and a string evaluates to a number as the string is implicitly converted into a number; in contrast in Python [vR03] a run-time type error is raised. The C language defines a large number of implicit type conversions between number types. At the extreme end of the spectrum, the TCL language implicitly converts every type into a string [Ous94]. Implicit type conversions need not be symmetrical; for example in Java adding a string to a number gives a compile-time warning (see Section 2.3 for an example) while adding a number to a string returns a string.

2.7 Terminology summary

Table 1 shows a comparison of a number of languages with respect to the terms defined in this section. As is clearly shown, languages utilise types in almost every conceivable combination, making the traditional 'hard' distinction between statically and dynamically typed languages seem very simplistic. Both classes of languages are typed, the chief technical difference between them being *when* types are enforced. The terms 'statically typed' and 'dynamically typed' are the source of much confusion but are sufficiently embedded within the community that it is unlikely that they will be superseded—hence why I use those terms in this chapter. However readers may find it more helpful to think of 'static typing' as that performed at compile-time and dynamic typing that performed at run-time. This can help understand the real-world, where most 'statically typed' languages also utilise run-time type checking, and where some 'dynamically typed' languages allow optional compile-time type checking.

3 Disadvantages of static typing

The advantages of static typing are widely known [Bra04] and include:

- Each errors detected at compile-time prevents a run-time error.
- Types are a form of documentation / comment.

- Types enable many forms of optimisation.

Taken at face value, the first of these is a particularly compelling argument: why would anyone choose to use less reliable languages? In reality the absence or presence of static typing has a number of effects on the use and applicability of a language that are not explained by the above. In particular, because the overwhelming body of research on programming languages has been on statically typed languages, the disadvantages of statically typed languages are rarely enumerated. In this section I enumerate some of the weaknesses of static typing and why it is therefore not equally applicable to every programming task.

3.1 Static types are inexpressive

As defined in Section 2.1, types are constraints. In practice, programming language types most closely conform to the intuitive notion of 'shape' or 'form'. Perhaps surprisingly, in some situations types can be too permissive and in others too restrictive (for an extreme example of this duality, see overloading in Java [AZD01]). Furthermore as static types need to be checked at compile-time, by definition they lack run-time information about values, further limiting their expressivity (interestingly, the types used in dynamically typed languages are virtually identical in expressivity to those used in statically typed languages, probably due to cultural expectations rather than technical issues).

3.1.1 Overly permissive types

Consider the following Java code which fails at run-time with a `division by zero` exception:

```
int x = 2;
int y = 0;
int z = x / y;
```

Looking at this, programmers of even moderate experience can statically spot the cause of the error: the divisor should not be zero. Java's compiler can not statically detect this error because the `int` type represents real numbers including zero; thus the above code is statically type correct according to Java's types. Not only is there not a type in Java which represents the real numbers excluding zero, there is no mechanism for defining such a type in a way that would result in equivalent code leading to a compile-time error. This limitation is shared by virtually all statically typed languages.

As suggested above, the static types available in todays mainstream languages are particularly inexpressive. Though research languages such as Haskell contain more advanced type systems, they still have many practical limitations. Consider the `head` function, which takes a list and returns its first element; given an empty list, `head` raises a run-time exception. Taking the head of an empty list is a common programming error, and is particularly frustrating in programming languages such as Haskell whose run-time error reporting makes tracking down run-time errors difficult [SSJ98]. It is possible to make a new list type, and a corresponding `head` function, which can statically guarantee

that the head of an empty list will never be taken [XP98]; however this only works for lists whose size is always statically known. Lists that are created on the basis of user input – a far more likely scenario – are highly unlikely to be statically checkable. Trying to use a type system in this way adds significant complexity to user programs with only minimal benefits.

Because of the general inexpressiveness of static types, an entirely separate strand of research tries to statically analyse programs to detect errors that escape static type checkers (see e.g. [MR05] for work directly related to the `head` function).

3.1.2 Overly restrictive types

Since any practical type system needs to be both decidable and sound, they are not complete; in other words, certain valid programs will be rejected by the type checker [AWL94, Mat90]. For example, type systems provide a fixed, typically small (or even empty), number of ways of relating types, with object orientated languages allowing types to be defined as sub-types of others allowing a certain kind of polymorphism. However programmers often need to express relationships between types that static types prevent, even in research languages with advanced type systems such as ML [CF91].

3.1.3 Type system complexity

From a pragmatic point of view, relatively small increases in the expressivity of static type systems cause a disproportionately large increase in complexity [Mac93, MD04]. This can be seen clearly in Abadi and Cardelli's theoretical work which defines static type systems of increasing expressiveness for object orientated languages [AC96]; their latter systems, though expressive, are sufficiently complex that, to the best of my knowledge, they have never been implemented in any language.

3.2 Types are represented by a separate language

Since most of us are used to the presence of explicit static types, it is easy to overlook the fact that they are represented by an entirely different language from the base programming language. In other words, when learning the syntax and semantics of programming X, one must also learn the syntactically and semantically distinct static type language XT. That X and XT are, at heart, separate languages can be seen by the very different types of errors that result from violating each's semantics. While programming languages have developed various mechanisms when presenting error information to aid programmers, the error messages from static type systems are often baroque and hard to understand [Mei07].

3.3 Type systems' correctness

Static type systems are often the most complex parts of a programming language's specification. Because of this it is easy for them to contain errors which then result in 'impossible' run-time behaviour [Car97].

A famous example comes from Eiffel [Mey92], one of the first 'mainstream' object orientated languages. Eiffel allows overridden methods to use subtypes of the parameters in the superclass. Consider classes `A1`, `A2`, `B1`, `B2`, and `B3`, where `A2` subclasses `A1`, and `B3` subclasses `B2` which subclasses `B1`. In object orientated languages in general, instances of subclasses (e.g. `A2`) can be considered as instances of superclasses (e.g. `A1`); intuitively this is because subclasses have type-identical versions of everything in the superclass plus, optionally, extra things. Eiffel subtly changes this, so that subclasses can contain type-compatible versions of everything in the superclass plus, optionally, extra things. Therefore in Eiffel one can define a method `m(p1:B2)` (meaning that `m` has a parameter `p1` of type `B2`) in class `A1` that is overridden in class `A2` by `m(p1:B3)`. If an instance of `A2` is considered to be an instance of its superclass `A1`, then an instance of `B2` can validly be passed to `A2::m` which may then attempt to access an attribute present only in instances of the subclass `B3`. Such *covariant* typing is unsafe and programs which utilise it can crash arbitrarily at run-time despite it satisfying Eiffel's type safety rules [Coo89].

As the Eiffel example suggests, and despite their formal veneer, the vast majority of static type systems are not proved correct; some are sufficiently complex that a full proof of correctness is impractical or impossible [Bra04]. Eiffel again gives us a good example of the subtleties that type systems involve: counter-intuitively type theory shows that `A2::m` could safely use super-types of the parameter types in `A1::m` (i.e. *contravariant* typing), so `A2::m(p1:B1)` is type-safe [Cas95].

Flaws discovered in type systems are particularly invidious, because changes to type systems will typically break most extant programs; for this reason, even modern versions of Eiffel contain the above flaw (whilst alleviating it to some extent).

3.4 System ossification

Virtually all software systems are changed, often continuously, and rarely in a planned or anticipated manner, after their original development [LB85]. It is therefore an implicit requirement that software be amenable to such change, which further implies that programming languages facilitate such change.

When changing a program, it is often desirable to change small sections at a time and see the effect of that change on that particular part of the program, so that any new errors can be easily related to the change; when performing such changes it is often expected that the program as a whole may not work correctly. Static type systems often prevent this type of development, because they require that the system as a whole is always type correct: it is not possible to temporarily turn off static type-checking. As static types make changing a system difficult, they inevitably cause systems to prematurely ossify, making them harder to adapt to successive changes [NBD+05].

3.5 Run-time dynamicity

Software is increasingly required to inspect and alter its behaviour at run-time, often in the context of critical systems that are expected to run without downtime, which must be patched whilst still running [HN05]. Traditionally statically typed languages' compilers

have discarded most information about a programs structure, its types, and so on during the compilation process, as they are not considered central to the programs execution. This means that most such languages are incapable of meaningful *reflection* [DM95]. Of those that do (e.g. Java), the ability to change the run-time behaviour of a program is relatively limited because of the possibility of subverting the type system. This means that statically typed languages have typically proved difficult to use in systems that require run-time dynamicity [NBD+05].

4 History

Dynamically typed languages have a long and varied history. While few dynamically typed languages have had a direct impact on the programming mainstream, they have had a disproportionate effect on programming languages in general. Perhaps because of their inherently flexible nature, or the nature of the people attracted to them, dynamically typed languages have pioneered a bewildering array of features. Thus the history of dynamically typed languages is intertwined with that of statically typed programming languages which, often after a significant delay, have incorporated the features pioneered in dynamically typed languages.

To the best of my knowledge, a history of dynamically typed languages has not yet been published, although the History of Programming Languages (HOPL) conferences[3] include histories of several of the most important languages (see e.g. [SG96, Kay96, GG96a]). A full history is far beyond the scope of this chapter. However there have been several important innovations and trends which explain the direction that dynamically typed languages have taken and why current dynamically typed languages take the shape they do. The initial history of dynamically typed languages is largely of individual languages – Lisp and Smalltalk in particular – while the more recent history sees groups of languages – such as so-called 'scripting' languages including Perl, Python, and Ruby – forging a common direction. Therefore this section enumerates, in approximately chronological order, the major points in the evolution of dynamically typed languages.

4.1 Lisp and its derivatives

Arguably the first dynamically typed language, certainly the oldest still in use, and without doubt the most influential dynamically typed language is Lisp [McC60]. Created in the 1950's, Lisp was originally intended as a practical notation for the λ-calculus [McC78]. Lisp is notable for its minimal syntax, the smallest of any extant programming language used in the real-world, allowing it a similarly small and uniform semantics. This simplicity – it was quickly discovered that it is possible to specify a minimal Lisp interpreter in a single page of Lisp code – made its implementation practical on machines of the day. That the innovations pioneered by, and within, Lisp are too many too mention can be inferred from its introduction of the `if-then-else` construct now taken for granted in virtually all programming languages.

[3] `http://research.ihost.com/hopl/`

Simply labelled, Lisp is an impure functional language. To modern eyes, Lisp is unusual because its concrete syntax uses prefix notation as can be seen from this simple example of a Fibonacci function:

```
(defun fib (n)
  (if (= n 0)
    0
    (if (= n 1)
      1
      (+ (fib (- n 1)) (fib (- n 2))))))
```

Lisp's minimal syntax allows it to be naturally represented by Lisp lists. Since lists can be inspected, altered, and created this led to what is arguably Lisp's most distinctive feature: macros. A macro is effectively a special function which, at compile-time, generates code. Macros allow users to extend a programming language in ways unforeseen by its creators [BS00]. Macros have therefore been a key facilitator in Lisp's continued existence, as they allow the spartan base language to be seamlessly extended: a typical Lisp implementation will implement most of its seemingly 'primitive' control structures through macros (see Section 5.3.2). Despite many attempts, it was not until the late 1990's that a syntactically rich, statically typed language gained a practical macro-like facility broadly equivalent to Lisp's (see [She98, SJ02]).

Lisp invented the concept of garbage collection [JL99] where memory allocation and deallocation is handled automatically by the Lisp interpreter or VM. Lisp was also the first language whose implementations made significant efforts to address performance concerns [Gab86]; many of the resulting implementation techniques have become standard parts of subsequent language implementations.

4.1.1 Scheme

Lisp has spawned many dialects, the most significant of which is Scheme [SJ75]. For the purposes of this chapter, Scheme can be thought of as a version of Lisp with a minimalist aesthetic, particularly with regard to its libraries. While Lisp has seen reasonable industrial usage (particularly in the 1980's, when it was the language of choice for artificial intelligence work), Scheme has largely been a research language, albeit a very influential one.

Scheme was the first language to introduce closures, allowing full lexical scoping, simplifying many types of programming such as Graphical User Interface (GUI) programming. It also popularised the concept of continuations, allowing arbitrary control structures to be constructed by the user [HFW84]. Scheme also showed that functions and continuations could be treated as first-class objects. Much of the foundational work on safe, powerful, macros was done in Scheme (see e.g. [KFFD86, CR91]).

4.2 Smalltalk

Smalltalk is Lisp's nearest rival in influence. Put simply, Smalltalk is a small, uniform object orientated language, heavily influenced by Lisp and Simula [DN66]. Compared

to later languages, Smalltalk's syntax is small and uncomplicated (though not as minimalistic in nature as Lisp's); however, in most other ways, Smalltalk-80 [GR89] (the root of all extant Smalltalk's) is recognisably a modern, object orientated, imperative programming language.

Smalltalk pioneered the idea of 'everything is an object' where even primitive values (integers etc.) appear as normal objects whose classes are part of the standard class hierarchy. Smalltalk has extensive meta-programming abilities. Reflection allows programs to query and alter themselves [Mae87]. A Meta-Object Protocol (MOP) [KdRB91] allows objects to change the way they behave; from the perspective of this chapter, the most significant of these abilities is meta-classes [FD98] (see Section 5.3.1).

In Smalltalk every object can be queried at run-time to find out its type. In common with most object orientated languages, a Smalltalk class also implicitly defines a type (see Section 2.1), so the 'type' of an object is the `Class` object which created it. A meta-class is simply the type of a class. In Smalltalk the default meta-class for a class is called `Metaclass`; a cycle is created in the type hierarchy so that `Metaclass` is its own type. Meta-classes allow Smalltalk to present a uniform, closed world where every object in a running system is typed by an object in the same running system. Only a small amount of bootstrapping is needed to create this powerful illusion (later proposals have shown how the meta-class concept can be further simplified [Coi87]).

4.3 Text processing languages

Text processing is a perennial programming task, and several languages have been wholly or mostly designed with this in mind. This domain has been dominated by dynamically typed languages, because the processing of unstructured data benefits greatly from the flexibility afforded by such languages [MD04].

The first languages aimed at these tasks, most noticeably SNOBOL4 [GPP71], were effectively Domain Specific Languages (DSLs) for text processing, and were not suitable for more general tasks [Gri78]. One of SNOBOL4's direct successor languages was Icon [GG96b], which introduced a unique expression evaluation system which dispenses with boolean logic and allows limited backtracking within an imperative language. This allows one to express complex string matching which can naturally evaluate multiple possibilities.

Sed and AWK [AKW98] represent an entirely different strand of text processing languages from SNOBOL and Icon. They can be thought of as enhanced UNIX shell languages, with AWK extending Sed with a number of more general programming language constructs. Perl [WCO00] represents the final evolution of this family of languages. Reflecting its rôle as a tool for ad-hoc development, it integrates a bewildering number of influences to an AWK base, and is notable for having arguably the most sophisticated – or, depending on ones point of view, complex – syntax of any programming language.

Most of the above languages are not, in the widely understood sense, general purpose languages. Icon is the most obviously general purpose language, although because of the many idioms it encompasses, Perl has been used in many domains. Because of the ubiquity of Sed and AWK and, in the early years of the web, Perl's dominance of server

side processing, these languages have been more widely used than any other category of dynamically typed languages.

4.4 Declarative languages

Although dynamically typed languages are often implicitly assumed to be imperative languages, dynamic typing is equally applicable to declarative languages which, for the purposes of this chapter, I define to mean logic and 'pure' functional languages (i.e. those without side effects). Prolog [SS94] was amongst the first, and remains the most widely used, logic language. Logic languages are very unlike 'normal' languages, with the user declaring relations amongst data, and then stating a goal over this which the language engine then attempts to solve—the order in which statements in the language are executed is non-linear.

Pure functional languages[4] have largely been confined to the research lab and have tended to be coupled with exotic static type systems. Although Erlang [VWWA96] started existence as a distributed variant of Prolog, it has since evolved to become one of the few dynamically typed pure functional languages. This perhaps reflect its industrial origins where it was designed to implement robust, scalable, distributed systems, particularly telephony systems [Arm07]. Erlang is arguably the most successful pure functional language yet with several million LoC systems. By eschewing static types, it is able to focus on the hard issues surrounding distributed systems, including a number of unique concepts relating to message passing and fault tolerance.

4.5 Prototyping languages

Object orientated languages derived from SIMULA such as Smalltalk are class-based languages: objects are created by instantiating classes. While everything in Smalltalk is an object, practically speaking classes are a very distinguished type of object from the users perspective. Self [US87] aimed to distill the object orientated paradigm down to its bare essentials: objects, methods, and message sends. In particular Self removed classes as a fundamental construct; new objects are created by *cloning* another object. The notion of type in Self, and other prototyping languages, is thus subtly different than in other languages.

Because of their minimalistic nature, raw prototyping languages tend to be particularly inefficient. Self pioneered a number of important implementation techniques [CU89] that ultimately allowed Self to become one of the highest performing dynamically typed languages. Much of this work has found its way into other languages, including statically typed languages such as Java [Ayc03].

[4] The 'pure' name is a misnomer, since a truly side effect free program would be incapable of input / output. Informally 'pure' is generally used to mean 'no explicit side effects such as assignment'.

4.6 Modern 'scripting' languages

The resurgence of interest in dynamically typed languages is largely due to what were originally dismissively called 'scripting' languages [Ous98], which had their roots in text processing languages such as Sed and AWK (see Section 4.3). Unlike many of the languages described earlier in this section, these languages were not designed with innovation as a primary goal, and instead emphasised consolidation and popularisation. They have therefore focused on practical issues such as portability, and shipping with extensive libraries. TCL [Ous94] was the first such language, which gained reasonable popularity in large part because of its bundled GUI toolkit. Python and Ruby [TH00] – fundamentally very similar languages once surface syntax issues are ignored – can be seen as modernised, if less internally consistent, versions of Smalltalk. Because of their inherent flexibility, such languages were initially often used to 'glue' other systems together, but have increasingly seen to be useful for a wide range of programming tasks, such as web programming tasks. Lua [Ier06] is a smaller language (both conceptually, and in its implementation) than either Python and Ruby, and has been more explicitly designed as an embeddable programming language; it has been used widely in the computer games industry to allow the high-level definition and extension of games [IdFC07].

While this sub-category of dynamically typed languages has not greatly advanced the state of the art, it has been the driving factor in validating dynamically typed languages and making them a respected part of a programmers toolbox. Most new systems written using dynamically typed languages use this category of languages.

5 Defining features

In previous sections I have defined the fundamental terms surrounding types and programming languages, and presented a brief history of dynamically typed languages. In this section I enumerate the defining features and characteristics of dynamically typed languages, and explain why they make such languages interesting and useful. Some of these features and characteristics have recently found their way into new statically typed languages, either as a core feature or as library add-ons. However no statically typed language contains all of them, nor is that likely to occur both for technical and cultural reasons.

5.1 Simplicity

A defining characteristic of virtually all dynamically typed languages is conceptual simplicity. Fundamentally dynamically typed languages are willing to trade run-time efficiency for programmer productivity. Such simplicity makes both learning and using dynamically typed languages simpler, in general, than statically typed languages since there are less 'corner cases' to be aware of. At its most extreme, Lisp's minimal syntax means that a full interpreter written in Lisp can fit on one page. Although most dynamically typed languages include as standard a greater degree of syntax and control structures than Lisp, this general principle remains.

At the risk of stating the obvious, dynamically typed languages do not contain constructs relating to static types. This is a significant form of simplification, as although static typing is sometimes considered to be the simple 'tagging' of variables with a given type name, static typing has a much more pervasive effect on a language. For example: static typing requires an (often significant) extension to a language's grammar to allow type 'tags' to be expressed and requires concept(s) allowing static types to be related to one another (e.g. the Java concept of interface).

The learning curve of dynamically typed languages is considerably shallower than for most statically typed languages. For example, in many dynamically typed languages the classic 'hello world' program is simply `print "Hello world!"` or a minor syntactic variant. In Java, at the other extreme, it requires a 7 line program – in a file whose name must exactly match the class contained within it – using a bewildering array of unfamiliar concepts. While programming beginners obviously struggle with the complexity that a language like Java forces on every user, it is widely known that programming professionals find it easier to learn new dynamically typed languages [Ous98].

5.2 High level features

Dynamically typed languages pioneered what are often informally known as 'high level features'—those which abstract away from low-level machine concerns.

5.2.1 Built-in Data types

Whereas many statically typed languages provide only very simple built-in data types – integers and user-defined structures – dynamically typed languages typically provide a much richer set. The two universal data types are lists (automatically resizing arrays) and strings (arbitrary character arrays); most dynamically typed languages also provide support for dictionaries (also known as associative arrays or hash tables; fast key / value lookup) and sets. These data types are typically tightly integrated into the main language, often with their own syntax, and used consistently and frequently throughout libraries. In contrast, most statically typed languages defer most such data types to libraries; consequently they are rarely as consistently or frequently used.

Complex data structures are often naturally expressed using just built-in data types. For example, the following Converge code shows how dictionaries of sets representing room numbers and employees are naturally represented:

```
x := Dict{10 : Set{"Fred","Sue"}, 17 : Set{"Barry","George","Steve"}, 18 : Set{"Mark"}}
x[10].add("Andy")
x[17].del("Steve")
```

After the above has been evaluated the dictionary referenced by `x` looks as follows:

```
Dict{10 : Set{"Fred", "Andy", "Sue"}, 17 : Set{"Barry", "George"}, 18 : Set{"Mark"}}
```

Using built-in data types not only improves programmer productivity, but also execution speed as built-in data types are highly optimised.

5.2.2 Automatic memory management

Manual memory management – when the programmer must manually allocate and free memory – wastes programmer resources (consuming perhaps around 30% – 40% of a programmer's time [Rov85]) and is a significant source of bugs [JL99]. Lisp was the first programming language to introduce the concept of garbage collection, meaning that memory is automatically allocated and freed by the language run-time, largely removing this burden from the programmer. Virtually all dynamically typed languages (and, more recently, most statically typed languages) have followed this lead.

5.3 Meta-programming

Meta-programming is the querying, manipulation, or creation of one program by another; often a program will perform such actions upon itself. Meta-programming can occur at either, or both of, compile-time or run-time. Dynamically typed languages have extensive meta-programming abilities.

5.3.1 Reflection

Formally, reflection can be split into three main aspects [BU04, MvCT+08]:

Introspection: the ability of a program to examine itself.

Self-modification: the ability of a program to alter its structure.

Intercession: the ability of a program to alter its behaviour.

For the purposes of this chapter, reflection is considered to be a run-time ability. For example in Smalltalk, programs can perform deep introspection on objects at run-time to determine their types (see Section 4.2). In the following Smalltalk examples '→' means 'evaluates to':

`2 + 2` → *4*
`(2 + 2) class` → *SmallInteger*
`(2 + 2) class class` → *SmallInteger class*
`(2 + 2) class class class` → *Metaclass*

Self-modification allows behaviour to be added, removed, or changed at run-time. For example in Smalltalk if a variable `ie` references an appropriate method (the definition of which is left to the reader), then it can be added to the `Number` class, so that all numbers can easily test whether they are odd or even:

`3 isEven` → *Message not understood*
`Number addSelector: #isEven withMethod ie` → *Adds method* ***isEven*** *to* ***Number***
`3 isEven` → *false*

Unfettered run-time modification of a system is dangerous, since it can have subtle, unintended consequences. However careful use of reflection allows programmers to bend

a language to their particular circumstances rather than the other way round. Most dynamically typed languages are capable of introspection; many are capable of self-modification; relatively few are capable of intercession (Smalltalk being one of the few). While a few statically typed languages such as Java support the introspective aspects of reflection, few are as consistently reflective as Smalltalk and its descendants, and none allow the level of manipulation as shown above.

Some OO languages have a meta-object protocol (MOP) [KdRB91] which allows intercession, as objects can alter the way they respond to message sends. For example in Python objects can override the `__getattribute__` function which receives a message `name` and returns an object of its choosing. The following example code (although too simple for production use) shows how Python objects can be made to appear to automatically have automatic 'getter' methods if they don't exist:

```
class C(object):
    x = 2
    def __getattribute__(self, name):
        if name.startswith("get_"):
            v = object.__getattribute__(self, name[4 : ])
            return lambda : v
        else:
            return object.__getattribute__(self, name)

i = C()
print i.x
print i.get_x()
```

In this example, both `i.x` and `i.get_x()` evaluate to the same result. Similar tricks can be played with the setting and querying of object slots. While delving into the MOP can easily introduce complications such as infinite loops, it can be useful, as in this example, to allow one object to emulate the behaviour of another, allowing otherwise incompatible frameworks and libraries to interact. Reflection also allows much deeper changes to a system such as allowing run-time modification of whole program aspects [OC04].

5.3.2 Compile-time meta-programming

Compile-time meta-programming allows the user to interact with the compiler to allow the construction of arbitrary program fragments. Lisp's macros are the traditional form of compile-time meta-programming and are used extensively to extend the minimal base language. For example the `when` control structure is a specialised form of `if`, taking a condition and a list of expressions; if the condition holds, `when` evaluates all expressions, returning the result of the final expression. In Common Lisp [Ste90] (alongside Emacs Lisp, one of the major extant Lisp implementations) `when` can be implemented as follows:

```
(defmacro when (cond &rest body)
  `(if ~cond (progn ~@body)))
```

Whenever a 'function call' to `when` is encountered during compilation, the above macro is executed and the resultant generated code statically replaces the 'function call'. The two

major features in the above are the quote ‘ which in essence returns the quoted expression as an Abstract Syntax Tree (AST) (i.e. without evaluating it) and the insertion ~ which inserts one Lisp AST in another.

Because macros in Lisp are often considered to rely on some of Lisp's defining features – in particular its minimal syntax which means that Lisp ASTs are simply lists of lists – subsequent dynamically typed languages did not have an equivalent system. In a rare occurrence, the statically typed languages MetaML [She98] and then Template Haskell [SJ02] showed how a practical compile-time meta-programming system could be naturally integrated into a modern syntactically rich language. Compile-time meta-programming is slightly more generic in concept than traditional macros, as it allows users to interact with the compiler, where such interactions may not always lead to the generation of code. Converge (created by this chapters author) integrates a Template Haskell-like system into a dynamically typed language, and uses it to implement a syntax extension feature which allows syntactically distinct DSLs to be embedded into normal programs.

5.3.3 Eval

Colloquially referred to by its short name, 'eval' refers to the ability, almost wholly confined to dynamically typed languages, to evaluate arbitrary code expressions as strings at run-time. In other words, code fragments can be received from, for example, end users, evaluated and the resulting value used for arbitrary purposes. Note that eval is very different from compile-time meta-programming, since expressions are evaluated at run-time, not compile-time, and any value can be returned (not just ASTs). While eval has many obvious downsides – allowing arbitrary code to be executed at run-time has severe security implications – when used carefully (e.g. in configuration files) it can reduce the need for arbitrary mini-programming languages to be implemented within a system.

5.3.4 Continuations

Popularised in Scheme, continuations remain a relatively exotic construct, with support only found in a handful of other languages, noticeably including Smalltalk. At a high-level, they can be thought of as a generalised form of co-routine [HFW84] which allows a safe way of defining 'goto' points, capturing a certain part of the current program state and allowing that part to be suspended and later resumed. Continuations are sufficiently powerful that all other control structures can be defined in terms of them.

The low-level power of continuations, and the fact that they subvert normal expectations of control flow, has meant that they have been talked about rather more than they have been used. However they have recently shown to be a natural match for web programming, where the back button in web browsers causes huge problems because it is effectively an 'undo'; most web systems give unpredictable and confusing results if the back button is used frequently. Continuations can naturally model the chain of resumption points that represent each point in the users browsing history, as can be

seen in the Smalltalk Seaside framework [DLR07]. This means that web systems respect users intuition when the back button is used, but are not unduly difficult to develop.

5.4 Refactoring

Refactoring is the act of applying small, behaviour-preserving transformations, to a system [FBB+99]. The general aim of refactoring is to maintain, or restore, the internal quality of a system after a series of changes so that further changes to the system are practical. A key part of the refactoring definition 'behaviour-preserving': it is vital that refactorings do not introduce new errors into a system. In practice, two distinct types of refactorings can be identified:

1. Small, tightly defined, and automatable refactorings. Exemplified by the 'move method' refactoring where a method is moved from class `C` to `D`.

2. Larger, typically project specific, non-automatable refactorings. A typical example is splitting a module or class into two to separate out functionality.

Statically typed languages have an inherent advantage over dynamically typed languages in the first type of refactoring because of the extra information encoded in static types. However static types are a burden in the second type of refactoring because they always require the entire system to be type correct. This means that it is not possible to make, and test, small local changes to a sub-system when such changes temporarily violate the type system; instead the entire refactoring must be implemented in one fell swoop which means that any resulting errors are difficult to relate to an individual action. Counter-intuitively, perhaps, static types inhibit large-scale refactorings, tending to ossify a program's structure (see Section 3.4). The flexibility of dynamically typed languages on the other hand encourages continual changes to a system [NBD+05], though it is often wise to pair it with a suitable test suite to prevent regressions (see Section 6.2).

5.5 'Batteries included' libraries

Traditionally, many statically typed languages – from Algol to Ada – have been designed as paper standards, detailing their syntax and semantics, but typically agnostic as to libraries. Such languages are then implemented by multiple vendors, each of which is likely to provide different libraries. In contrast, most dynamically typed languages – with the notable exception of the Lisp family – have been defined by their initial implementation and its accompanying libraries. The majority of modern dynamically typed languages (see Section 4.6) come with a rich set of standard libraries – the so-called 'batteries included' approach[5] [Oli07] – which encompass enough functionality to be suitable for a majority of common programming tasks. Implicit in this is the assumption that if the initial implementation is replaced, the standard library will be provided in a backwards-compatible fashion; in comparison to paper-based standards, it

[5] While this phrase originated in the Python community, it reflects a common belief amongst most dynamically typed languages.

is often difficult to distinguish between the language and its libraries. Furthermore, due to the emphasis on a rich set of standard libraries, it is relatively easy to define new, external libraries without requiring the installation of many dependent libraries.

As described in Section 6.1, the performance of dynamically typed languages varies from slightly to significantly slower than statically typed languages; however, suitable use of libraries (which are typically highly optimised) can often significantly diminish performance issues.

5.6 Portability

Portable software is that which runs on multiple target platforms. For the purposes of this chapter, a platform can be considered to be a combination of hardware and operating system[6]. For most non-specialised purposes, users wish their software to run on as many platforms as practical.

One way of achieving portability is to allow programs to deal, on an as-needs basis, with known variations in the underlying platform; the other is to provide abstractions which abstract away from the hardware and the operating system [SC92]. Since dynamically typed languages aim to present a higher-level view of the world to programs (see e.g. Section 5.2), they follow this latter philosophy. There are many examples of such abstractions, but two in particular show the importance of abstracting away from the hardware and the operating system. First, 'primitive types' such as integers will typically automatically change their representation from an efficient but limited machine type to a variably sized container as necessary, thus preventing unintended overflow errors. Second, file libraries provide simple `open` and `read` calls (note that garbage collection typically closes files automatically in dynamically typed languages, so explicit calls to `close` are less important) which abstract away from the wide variety of file processing calls found in different operating systems. By providing such abstractions, dynamically typed programs are typically more portable than most statically typed languages because there is less direct reliance on features of the underlying platform.

5.7 Unanticipated reuse

A powerful type of reuse is when functionality is composed from smaller units in ways that are reasonable and valid, but not anticipated by the authors of each sub-unit. Ousterhout shows how, by using untyped text as its medium and lazy evaluation as its process, the UNIX shell can chain together arbitrary commands with pipes [Ous98]. For example the following command counts how many lines the word 'dynamic' occurs in `.c` files:

```
find . -name "*.c" | grep -i dynamic | wc -l
```

The enabling factor in such reuse is the loose contracts placed on input and output data: if the UNIX shell, for example, forced data passed through pipes to be statically typed it

[6] A precise definition of platform would have to cope with many ontological difficulties, such as the Java Virtual Machine which defines a 'platform independent' platform of its own.

is unlikely that such powerful chains of commands could be created as commands would not be as easily reusable.

Dynamically typed languages allow similar reuse to the UNIX shell, but with a subtle twist. While most Unix shell commands demand nothing of input text (which may be empty, all on one line etc.), and statically typed languages demand the complete typing of all inputs, dynamically typed languages allow shades of grey in-between. Essentially the idea is that functions should demand (and, possibly, check) the minimum of any inputs to ensure correct functionality, thus allowing functions to operate correctly on a wide range of seemingly unrelated input. This philosophy, while long-standing, has recently acquired the name *duck typing* to reflect the intuitive notion that if an input 'talks like a duck and quacks like a duck, it is a duck'—even if other aspects of the input may not look like a duck [KM05]. Duck typing can be seen as the run-time, dynamically typed equivalent of structural typing (see Section 2.3.2). A good example of the virtues of duck typing can be found in Python where functions that deal with files often expect only a simple `read` method in any input objects; this allows programs to make many non-file objects (e.g. network streams) appear as files, thus reducing the number of cases where specialised functions must be created for different types.

5.8 Interactivity

Virtually all dynamically typed languages are interactive, in the sense that users can execute commands on a running instance of the system and, if desired, perform further interactive computations on the result. Arguably the most powerful interactive systems are for Smalltalk, where systems are generally developed within an interactive GUI system containing both system tools (the compiler etc.) and the users code [GR89]. Most languages however provide such interactivity via a command-line interface which allows normal expressions to be entered and immediately evaluated. This allows the run-time system presented by the language to be explored and understood. For example the following session shows how the Python shell can be used to explore the type system and find out help on a method:

```
>>> True.__class__
<type 'bool'>
>>> True.__class__.__class__
<type 'type'>
>>> dir(True.__class__.__class__)
['__base__', '__bases__', '__basicsize__', '__call__', '__class__', '__cmp__',
'__delattr__', '__dict__', '__dictoffset__', '__doc__', '__flags__',
'__getattribute__', '__hash__', '__init__', '__itemsize__', '__module__', '__mro__',
'__name__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__str__', '__subclasses__', '__weakrefoffset__', 'mro']
>>> help(True.__class__.__class__.mro)
mro(...)
    mro() -> list
    return a type's method resolution order
>>>
```

By providing an interactive interface, dynamically typed languages encourage exploration of the run-time system, and also allow small examples to be worked on without any 'compile link' overhead.

5.9 Compile-link-run cycle

In the majority of programming languages – with the notable exception of Smalltalk and languages directly influenced by it such as Self (see Section 5.8) – programs are stored in one or more files. In order to run a program in a statically typed language, one must typically compile each individual file of the program, and link them together to produce a binary, which can then be run. This process is know as the 'compile-link-run' cycle. Because statically typed languages are relatively complex to compile and link, this is often a lengthy process—even on modern machines, large applications can take several hours to compile and link from scratch. This is often a limiting factor in rapid application development [TW07].

In contrast, most dynamically typed languages conflate the compile-link-run cycle, allowing source files to be directly 'run'. As compilation of individual modules is often done on an 'as needs' basis, and since the compilation and linking of dynamically typed languages is much simpler since no static types need to be checked, this means the user experiences a much shorter compile-link-run cycle.

5.10 Run-time updates

With the increasing trend of software providing long-running services (e.g. switches, financial applications), it is necessary to upgrade software without stopping it [HN05]. This means replacing or augmenting values in the run-time system, typically with data and functionality in the 'old' system existing side-by-side with the 'new'.

While it is possible to perform limited run-time updates with statically typed languages, the general requirement to retain the type safety of the running system (without which random low-level crashes are likely), and the difficulty of migrating data, makes this extremely challenging in such languages (see Section 3.5). Dynamically typed languages have two significant advantages in such situations. First reflection allows arbitrary manipulation and emulation of data. Second there is no absolute requirement to maintain type safety in the updated system as, at worse, any type errors resulting from updating data or functionality will result in a standard run-time type error (in contrast, subverting the type system of a statically typed language is likely to lead to a low-level crash). Erlang makes heavy use of these features to allow extensive run-time updating in a way that allows resultant systems to keep running for very long periods of time [VWWA96].

6 Disadvantages of dynamic typing

6.1 Performance

Much has been said and written about the relative performance of various programming languages over the years; regrettably, much has been based on superstition, supposition, or unrepresentatively small examples. There is little doubt that, in practice, equivalent programs in dynamically typed languages are slower than in statically typed languages. While on certain macro benchmarks some language implementations (typically Lisp or Smalltalk implementations) can achieve approximate parity with statically typed languages, a general rule of thumb is that the most finely tuned dynamically typed language implementations are approximately two times slower than the equivalent statically typed implementation[7].

The performance gap between dynamically typed and statically typed languages has lowered over recent years, in large part due to innovations surrounding JIT compilation [Ayc03]—the difference in speed between dynamically typed language implementations with and without JIT compilation is typically a factor of three to five. Currently the performance between different dynamically typed language implementations varies wildly, with languages such as Ruby an order of magnitude slower than leading Lisp's. As there are few technical reasons for such differences, and given recent trends such as common virtual machines and the awareness of the benefits of JIT compilation, it is likely that the performance gap between implementations will narrow considerably.

Arguably more important than absolute performance measured in minutes and seconds is the performance relative to requirements: in other words, does the program 'run fast enough?' Thanks in part to the advancements of commodity computers, for most real-world purposes, this question is often redundant. For certain tasks, particularly very low-level tasks, or those on low-performance computers such as some embedded systems, statically typed languages retain an important advantage. However it is interesting to note that in certain data-intensive and performance sensitive domains such as scientific computing dynamically typed languages have proved to be very successful (see e.g. [CLM05, Oli07]). There are two explanations for this. First, the high-level nature of dynamically typed languages allows programmers to focus on improving algorithms rather than low-level coding tricks. Second, dynamically typed languages typically come with extensive, highly optimised libraries to which the most performance critical work is often deferred (the so-called 'batteries included' approach [Oli07]).

6.2 Debugging

A fundamental difference between statically and dynamically typed languages is that the former can detect and prevent certain errors at compile-time (see Section 2.3). Logically this implies that dynamically typed programs are inherently more error-prone than

[7] As shown by 'The Computer Language Benchmarks Game' `http://shootout.alioth.debian.org/` which, despite its stated limitations, is one of the best attempts to compare performance, and is notable for the variety of language implementations it includes.

statically typed languages. This is potentially a real problem, hence why it is included in the 'disadvantages' section. However in practice, run-time type errors in deployed programs are exceedingly rare [TW07].

There are three main reasons why run-time type errors are rarely an issue. First, type errors represent a small, generally immediately obvious, trivially fixed class of errors and are thus typically detected and fixed quickly during development. Second – as shown in Section 3 – static types do not capture many of the more important and subtle errors that one might hoped would have been detected; such errors thus occur with equal frequency in statically and dynamically typed programs. Third, automated testing will tend to detect most type errors. This last point is particularly interesting. Unit testing is when a test suite is created that can, without user intervention, be used to check that a system conforms to the tests. Unit tests are often called 'regression suites' to emphasise that they are intended to prevent errors creeping back into a system. The first unit test suite was for Smalltalk [Bec94], but virtually all languages now have an equivalent library or facility e.g. Java [LF03]. As this suggests, unit testing allows developers to make guarantees of their programs that are considerably in excess of anything that static typing can provide.

6.3 Code completion

Many modern developers make use of sophisticated Integrated Development Environments (IDEs) to edit programs. One feature associated with such tools is code completion. In particular when a variable of type `T` is used in a slot lookup, the functions and attributes of the type are automatically displayed. This feature makes use of static types to ensure that (modulo any use of reflection) its answers are fully accurate. A fully equivalent feature is not possible for dynamically typed languages since it is not possible to accurately determine the static type of an arbitrary expression.

6.4 Types as documentation

Since most statically typed languages force users to explicitly state the types that functions consume and return, statically typed programs have an implicit form of documentation within them, which happens to be machine checkable [Bra04]. There is little doubt that this form of documentation is often useful and that dynamically typed languages do not include it. However since it is possible to informally notate the expected types of a function in comments, or associated documentation strings processed by external tools, this is not a major disadvantage; furthermore some dynamically typed languages include optional type systems (see Section 7.2) that allow code to be annotated with type declarations when desired.

7 Variations

In the majority of this chapter I have described a homogenised picture of dynamically typed languages, emphasising the culturally common aspects of most languages. In-

evitably this smooths over some important differences and variations between languages; this section details some of these.

7.1 Non-OO and OO languages

Dynamically typed languages come in both OO (e.g. Converge, Python) and non-OO (e.g. Lisp) flavours. Unsurprisingly, older dynamically typed languages tend to be non-OO, with languages of the past decade or more almost exclusively OO. Interestingly, the transition between these two schools can be seen in languages such as Python (and, to a lesser extent, Lua) which started as non-OO languages but which were subsequently retro-fitted with sufficient OO features that their early history is only rarely evident. The general principles are largely the same in both cases, and in most of this chapter I have avoided taking an exclusively OO or non-OO approach.

OO does however introduce some new differentiating factors between statically and dynamically typed languages. In particular, static typing allows OO languages to introduce new ways of method dispatch (such as method overloading) due to polymorphism. While meta-programming allows dynamically typed languages to introduce analogous features, they are not tightly integrated into the language, or frequently used. In part because of this, it is generally easier to move between non-OO and OO programming styles in dynamically typed languages such as Python than to attempt the same in a statically typed OO language such as Java.

It is notable that dynamically typed languages have played a major part in the continued development of OO. For example, languages such as Self introduced the notable concept of prototyping [US87]; Smalltalk has been used as the workbench for innovations such as traits [SDNB03] which defines an alternative to inheritance for composing functionality.

7.2 Optional types

In most of this chapter, dynamic and static typing have been talked about as if they are mutually exclusive—and in most current languages this is true. While not integrated into any mainstream language, there is a long history of work which aims to utilise the benefits of both approaches [MD04] and blur this distinction. There are three main ways of achieving this. First, one can add a 'dynamic type' to a statically typed language, meaning that most data is statically typed, with some 'dynamically typed' (see e.g. [ACPP91, Hen94]). Second, and of greater interest to this chapter, one can add an *optional type* system to a dynamically typed language.

Intuitively, optional typing is easily defined: static types can be added at selected points in a program, or discovered through type inference, and those types are statically checked by a compiler. Optional typing thus means that portions a program can be guaranteed not to have type errors. Exactly how much of a program needs to be statically typed varies between approaches e.g. some proposal require whole modules to be fully statically typed [THF08] where others allow a free mixture of dynamic and static typing [SV08]. Optional types have two further advantages: they offer the possibility that

extra optimisations can be used on statically typed portions [CF91]; they also provide a machine-checkable form of documentation within source code (see Section 6.4).

Optional typing raises two particularly important questions:

1. Are type violations fatal errors (as they are in fully statically typed languages), or merely informative warnings?

2. Should static typing effect the run-time semantics of the system?

There is currently no agreement on either of these points. For example, as described in Section 7.1 static typing in OO languages can affect method dispatch, meaning that OO programs could perform method dispatch differently in statically and dynamically typed portions. Because of this, one possibility is to make optional types truly optional, in that their presence or absence does not effect the run-time semantics of a program [Bra04]. Taking this route also raises the possibility of using different type systems within one program.

For the purposes of this chapter, optional typing is considered to subsume a number of related concepts – including gradual typing, soft typing, and pluggable typing. As this may suggest, optional typing in its various form is still relatively immature and remains an active area of research.

7.3 Analysis

One approach to validating the correctness of a program is analysis. Static analysis involves analysing the source code of a system for errors, and is capable of finding various classes of errors, not just type errors. Static analysis is a well-established technique in certain limited areas, such as safety critical systems, where developers are prepared to constrain the systems they write in order to be assured of correctness. Such a philosophy is at odds with that of dynamically typed languages, which emphasise flexibility. Furthermore the inherent flexibility of dynamically typed languages would lead to a huge increase in the search space. Therefore static analysis is unlikely to be a practical approach for analysing dynamically typed programs. Another approach to analysis is to perform it at run-time – dynamic analysis – when virtual machines, libraries and so on are augmented with extra checks which aim to detect many errors at the earliest possible point, rather than waiting until a program crashes. Although such tools are in their infancy some, such as the Dialyzer system which performs such analysis for Erlang systems [LS04], are in real-world use.

8 The future

Definitively predicting the future of dynamically typed languages is impossible since there is no central authority, or single technology, which defines such languages. Nevertheless certain trends are currently evident. The increasing popularity of dynamically typed languages mean a revived interest in performance issues; while languages such as Self have shown that dynamically typed languages can have efficient implementations,

few current languages have adopted such techniques. As dynamically typed languages continue to be used in the real-world, increasingly for larger systems, users are likely to demand better performance. Experimentation in optional typing is likely to continue, with optional type systems eventually seeing real use in mainstream languages. The cross-fertilisation of ideas between statically and dynamically typed languages will continue, with language features such as compile-time meta-programming crossing both ways across the divide. It is also likely that we will see an increase in the number of dynamically typed domain specific languages, since such languages tend by nature to be small and 'lightweight' in feel.

9 Conclusions

In this chapter I detailed the general philosophy, history, and defining features of dynamically typed languages. I showed that, while a broad banner, such languages share much in common. Furthermore I have highlighted their contribution to the development of programming languages in general and, I hope, a sense of why they are currently enjoying such a resurgence.

I am grateful to Éric Tanter who provided insightful comments on a draft of this chapter. All remaining errors and infelicities are my own.

References

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.

[AKW98] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison Wesley, 1998.

[Arm07] Joe Armstrong. A history of Erlang. In *Proc. History of programming languages*. ACM, 2007.

[AWL94] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proc. Symposium on Principles of programming languages*, pages 163–173. ACM, 1994.

[Ayc03] John Aycock. A brief history of Just-In-Time. *ACM Computing Surveys*, 35(2):97–113, 2003.

[AZD01] D. Ancona, E. Zucca, and S. Drossopoulou. Overloading and inheritance. In *Workshop on Foundations of Object-Oriented Languages (FOOL8)*, 2001.

[Bec94] Kent Beck. Simple Smalltalk testing: With patterns, 1994. `http://www.xprogramming.com/testfram.htm` Accessed Jul 14 2008.

[Bra04] Gilad Bracha. Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, October 2004.

[BS00] Claus Brabrand and Michael Schwartzbach. Growing languages with metamorphic syntax macros. In *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN. ACM, 2000.

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proc. OOPSLA*, pages 331–344. ACM, 2004.

[Car97] Luca Cardelli. Type systems. In *The Computer Science and Engineering Handbook*, pages 2208–2236. 1997.

[Cas95] Giuseppe Castagna. Covariance versus contravariance: Conflict without a cause. In *ACM Transactions on Programming Languages and Systems*, pages 431–447. May 1995.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, 1991.

[CHC90] William Cook, Walter Hill, and Peter Canning. Inheritance is not subtyping. In *Seventeenth Symposium on Principles of Programming Languages*, pages 125–135, 1990.

[CLM05] Xing Cai, Hans Petter Langtangen, and Halvard Moe. On the performance of the Python programming language for serial and parallel scientific computations. *Scientific Programming*, 13(1):31–56, 2005.

[Coi87] Pierre Cointe. Metaclasses are first class: the ObjVLisp model. In *Object Oriented Programming Systems Languages and Applications*, pages 156–162, October 1987.

[Coo89] William R. Cook. A proposal for making Eiffel type-safe. *The Computer Journal*, 32(4):305–311, 1989.

[CR91] William Clinger and Jonathan Rees. Macros that work. In *19th ACM Symposium on Principles of Programming Languages*, pages 155–162. ACM, January 1991.

[CU89] C. Chambers and D. Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. *SIGPLAN Notices*, 24(7):146–160, 1989.

[DLR07] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside: A flexible environment for building dynamic web applications. *IEEE Software*, 24(5):56–63, 2007.

[DM95] François-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *Proc. IJCAI'95 Workshop on Reflection and Metalevel Architectures and Their Applications in AI*, pages 29–38, August 1995.

[DN66] Ole-Johan Dahl and Kristen Nygaard. An Algol-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[FD98] Ira R. Forman and Scott H. Danforth. *Putting Metaclasses to Work: A New Dimension in Object-Oriented Programming*. Addison-Wesley, 1998.

[Gab86] Richard P. Gabriel. *Performance and Evaluation of LISP Systems*. MIT Press, 1986.

[GG96a] Ralph E. Griswold and Madge T. Griswold. History of the Icon programming language. pages 599–624, 1996.

[GG96b] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Peer-to-Peer Communications, third edition, 1996.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[GPP71] R. E. Griswold, J. F. Poage, and I. P. Polonsky. *The SNOBOL4 Programming Language*. Prentice-Hall, second edition, 1971.

[GR89] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley, January 1989.

[Gri78] Ralph E. Griswold. A history of the SNOBOL programming languages. *SIGPLAN Notices*, 13(8):275–308, 1978.

[Hen94] Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, 1994.

[HFW84] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *Proc. Symposium on LISP and functional programming*, pages 293–298. ACM, 1984.

[HN05] Michael Hicks and Scott M. Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 27(6):1049 – 1096, 2005.

[IdFC07] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of Lua. In *Proc. History of programming languages*. ACM, 2007.

[Ier06] Roberto Ierusalimschy. *Programming in Lua*. Lua.org, second edition, 2006.

[JL99] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1999.

[Jon03] Simon Peyton Jones. *Haskell 98 Languages and Libraries: The Revised Report*. Cambridge University Press, April 2003.

[Kay96] Alan C. Kay. The early history of Smalltalk. pages 511–598, 1996.

[KdRB91] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[KFFD86] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Symposium on Lisp and Functional Programming*, pages 151–161. ACM, 1986.

[KG07] Suleyman Karabuk and F. Hank Grant. A common medium for programming operations-research models. *IEEE Software*, 24(5):39–47, 2007.

[KM05] Andrew Koenig and Barbara E. Moo. Templates and duck typing. *Dr. Dobb's*, 2005.

[LB85] M. M. Lehman and L. A. Belady. *Program Evolution: Processes of Software Change*. Academic Press, 1985.

[LF03] Johannes Link and Peter Fröhlich. *Unit Testing in Java: How Tests Drive the Code*. Morgan Kaufmann, 2003.

[Lou08] Ronald P. Loui. In praise of scripting: Real programming pragmatism. *Computer*, 41(7):22–26, 2008.

[LS04] Tobias Lindahl and Konstantinos Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In Chin Wei-Ngan, editor, *Programming Languages and Systems: Proceedings of the Second Asian Symposium (APLAS'04)*, volume 3302 of *LNCS*, pages 91–106. Springer, November 2004.

[Mac93] David B. MacQueen. Reflections on standard ML. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *LNCS*, pages 32–46. Springer-Verlag, 1993.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proc. OOPSLA*, pages 147–155, New York, NY, USA, 1987. ACM.

[Mat90] David C. J. Matthews. Static and dynamic type checking. *Advances in database programming languages*, pages 67–73, 1990.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine (part I). *Communications of the ACM*, 3(4):184–195, 1960.

[McC78] John McCarthy. History of LISP. In *History of programming languages*, pages 173–185. ACM, 1978.

[MD04] Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, October 2004.

[Mei07] Erik Meijer. Confessions of a used programming language salesman. *SIGPLAN Notices*, 42(10):677–694, 2007.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall International, 1992.

[MMMP90] Ole Lehrmann Madsen, Boris Magnusson, and Birger Mølier-Pedersen. Strong typing of object-oriented languages revisited. In *Proc. OOPSLA*, pages 140–150. ACM, 1990.

[MR05] Neil Mitchell and Colin Runciman. Unfailing Haskell: A static checker for pattern matching. In *Proc. Symposium on Trends in Functional Programming*, pages 313–328, 2005.

[MvCT+08] Stijn Mostinckx, Tom van Cutsem, Stijn Timbermont, Elisa Gonzalez Boix, Éric Tanter, and Wolfgang de Meuter. Mirror-based reflection in AmbientTalk. *Software—Practice and Experience*, 2008. To appear.

[NBD+05] Oscar Nierstrasz, Alexandre Bergel, Marcus Denker, Stéphane Ducasse, Markus Gälli, and Roel Wuyts. On the revival of dynamic languages. In *Proc. Software Composition 2005*, volume 3628 of *LNCS*, pages 1–13, 2005.

[Nor92] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, 1992.

[OC04] Francisco Ortin and Juan Manuel Cueva. Dynamic adaptation of application aspects. *Journal of Systems and Software*, 71:229–243, May 2004.

[Oli07] T.E. Oliphant. Python for scientific computing. *Computing in Science and Engineering*, 9(3):10–20, May 2007.

[Ous94] John Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[Ous98] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[Pau07] Linda Dailey Paulson. Developers shift to dynamic programming languages. *Computer*, 40(2):12–15, 2007.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Rov85] Paul Rovner. On adding garbage collection and runtime types to a strongly-typed, statically-checked, concurrent language. Technical Report CSL-84-7, Xerox Parc, 1985.

[SC92] Henry Spencer and Geoff Collyer. #ifdef considered harmful, or portability experience with CNews. In *Proc. of the Summer 1992 USENIX Conference*, pages 185–198, San Antionio, Texas, 1992.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour? In *Proc. ECOOP*, volume 2743 of *LNCS*, pages 248–274, July 2003.

[SG96] Guy L. Steele and Richard P. Gabriel. The evolution of Lisp. pages 233–330, 1996.

[SG97] Diomidis Spinellis and V. Guruprasad. Lightweight languages as software engineering tools. In *USENIX Conference on Domain-Specific Languages*, pages 67–76, Berkeley, CA, October 1997. USENIX Association.

[She98] Tim Sheard. Using MetaML: A staged programming language. *Advanced Functional Programming*, pages 207–239, September 1998.

[SJ75] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: An interpreter for extended lambda calculus. Technical Report AI Lab Memo AIM-349, MIT AI Lab, December 1975.

[SJ02] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the Haskell workshop 2002*. ACM, 2002.

[SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, second edition, March 1994.

[SSJ98] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic typing as staged type inference. In *Proc. Symposium on Principles of Programming Languages*, pages 289–302, January 1998.

[Ste90] Guy L. Steele, Jr. *Common Lisp the Language*. Digital Press, 2nd edition, 1990.

[SV08] Jeremy G. Siek and Manish Vacharajani. Gradual typing with unification-based inference. In *Dynamic Languages Symposium*, 2008.

[TH00] David Thomas and Andrew Hunt. *Programming Ruby: A Pragmatic Programmer's Guide*. Addison-Wesley, 2000.

[THF08] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed Scheme. *SIGPLAN Notices*, 43(1):395–406, 2008.

[Tra07] Laurence Tratt. *Converge Reference Manual*, July 2007. `http://www.convergepl.org/documentation/` Accessed June 3 2008.

[TW07] Laurence Tratt and Roel Wuyts. Dynamically typed languages. *IEEE Software*, 24(5):28–30, 2007.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proc. OOPSLA*, pages 227–241, October 1987.

[vR03] Guido van Rossum. Python 2.3 reference manual, 2003. `http://www.python.org/doc/2.3/ref/ref.html` Accessed June 3 2008.

[VWWA96] Robert Virding, Claes Wikstrom, Mike Williams, and Joe Armstrong. *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[WCO00] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, third edition, 2000.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proc. Conference on Programming Language Design and Implementation*, pages 249–257, 1998.