Programming Languages

Jens Palsberg
Purdue University

November 27, 2004

1 Introduction

The goal of a programming language is to make it easier to build software. A programming language can help make software flexible, correct, and efficient. How good are today's languages and what hope is there for better languages in the future?

Many programming languages are in use today, and new languages are designed every year; however, the quest for a language that fits all software projects has so far come up short. Each language has its strengths and weaknesses, and offers ways of dealing with some of the issues that confront software development today. The purpose of this chapter is to highlight and discuss some of those issues, to give examples from languages in which the issues are addressed well, and to point to ongoing research that may improve the state of the art. It is *not* our intention to give a history of programming languages [42, 10].

We will focus on flexibility, correctness, and efficiency. While these issues are not completely orthogonal or comprehensive, they will serve as a way of structuring the discussion. We will view a programming language as more than just the programs one can write; it will also include the implementation, the programming environment, various programming tools, and any software libraries. It is the combination of all these things that makes a programming language a powerful means for building software.

Two particular ideas are used for many purposes in programming languages: compilers and type systems. Traditionally, the purpose of a compiler is to translate a program to executable machine code, and the purpose of a type system is to check that all values in a program are used correctly. Today, compilers and type systems serve more needs than ever, as we will discuss along the way.

2 Flexibility

We say that software is flexible when it can run on a variety of platforms and is easy to understand, modify, and extend. Software flexibility is to a large extent achieved by good program structure. A programming language can help with the structuring of programs by offering language constructs for making the program structure explicit. Many such language constructs have been designed, and some of them will be highlighted in the following discussion of several aspects of software flexibility.

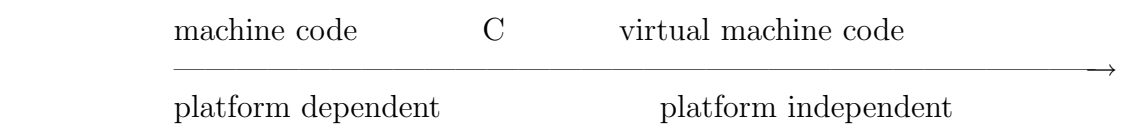
2.1 Platform Independence

Goal: Write once, run everywhere.

The C programming language [27] was invented in the 1970s. Over the years, it became the "gold standard" for platform independence: most general-purpose processors, supercomputers, etc. have their own C compiler, which, in principle, enables any C program to run on any such computer. This is much more flexible than rewriting every program from scratch, perhaps in machine code, for each new computer architecture.

In practice, most C programs will have some aspects that tie them closely to a particular architecture. Programmers are well aware of this and tend to design their programs such that porting software from one computer to the next entails minimal changes to the C source code. The main challenges to porting software lie partly in finding out the places in the C program where changes have to be made, and partly in understanding the differences between the two architectures in question.

In the 1990s, the Java programming language [23] popularized the use of virtual machines. Virtual machines increase platform independence by being an intermediate layer between high-level languages such as Java, and low-level languages such as machine code. A Java compiler will translate a Java program to virtual machine code, which the virtual machine executes. Each kind of computer will have its own implementation of the virtual machine, much like it will have its own C compiler. The virtual machine offers capabilities such as event handling in a platform-independent way, which in C would be platform dependent. This is a step towards the ultimate goal of "write once, run everywhere." Intuitively, we have a spectrum



Virtual machines such as those for Java and the .NET Common Language Runtime [30] are slowly replacing C as "portable assembly code," that is, the preferred target for compiling languages such as Java and C# [29]. Some virtual machines work in part by translating virtual machine code to C.

Platform independence is particularly important for mobile code that can move from one computer to another while it is executing. If the computers all run the same kind of virtual machine, then such movement is greatly simplified.

Platform independence via virtual machines comes with a price: it is more difficult to execute virtual machine code efficiently than it is to execute C code efficiently. Efficient implementation of virtual machines remains an active research area [2].

2.2 Abstraction

Goal: Never duplicate code.

When a programmer faces a programming task, it is tempting to program the needed functionality from scratch. It gives complete control over the code and minimizes the amount

of attention that needs to be paid to other code. But it also runs the risk of duplicating work already done by another programmer.

A better scenario for the programmer would be to come across some portion of an already written program that does almost exactly the same as what needs to be programmed. The programmer can now copy and paste the code and perhaps customize it to fit the particular situation. This increases the amount of reading of code by the programmer, but greatly decreases the amount of writing of code. Thus, the programmer is likely to finish the job in less time. However, the copied code may have a problem or defect, and if the original code gets fixed some day, the copy will stay the same. In other words, improvements cannot be done once and for all. The problem is that the code "lives" in several duplicates.

A vastly superior technique is to use abstraction to isolate and harness a piece of code that is useful in many scenarios. A popular abstraction mechanism is that of procedures which allow a code sequence to be called from many places. For example, suppose we have written code for sorting a list of data. In a language like C, we can abstract it into a procedure

```
void sort(int a[ ], int n) { ... }
```

which sorts the argument array a from positions 1 to n, whenever it is called. The devices of passing arguments and possibly returning results increase the usefulness of procedures as an abstraction mechanism.

An additional advantage of a procedure is that it creates a natural interface between the user and the implementer of the procedure. The implementer is free to change the implementation of the procedure as long as the new implementation has the same behavior. Thus, if the current implementation has a problem, then it can be fixed once and for all, to the benefit of all call sites.

Another abstraction mechanism is that of data abstraction, for example in the form of classes and objects. The idea is to let the programmer define a new data type with a well-defined set of operations and a hidden representation. Such a data type can be used in many places without duplication of code. Similar to the case of procedures, one can change the representation of a data type without affecting the clients.

Constructs for procedural and data abstraction are an integral part of object-oriented languages such as Java and C++. The highly successful and widely-used libraries that come with these languages were to a large extent enabled by these abstraction mechanisms.

The use of abstractions, particularly in libraries, switches some of the work of a programmer from writing to reading code. If a good abstraction can be located, perhaps the programmer needs to do little more than use it. The better the documentation, the more likely the programmer is to find something useful in a library.

Programmers continue to develop useful abstractions that cannot be easily expressed with the abstraction mechanisms of today's programming languages. Instead they are expressed and documented more informally, for example, in the form of design patterns [21]. As a result, the code for a design pattern is often duplicated to each place it is used. Developing new abstraction mechanisms that can capture some of the growing list of design patterns remains an active research area [8, 13].

2.3 Code Management

Goal: Separation of concerns.

A large program tends to have other big features than many lines of code. It may have a large state space, represented by many variables, and it may have several rather independent subtasks that are handled by quite separate parts of the code. A programmer will manage the code complexity by trying to keep separate things separate. For example, a classical system design is to have three layers: a user interface, the business logic, and a database. If the code for each of the layers is kept separate and has minimal and well-defined interaction with the other layers, then the overall system will be easier to build, understand, and maintain.

A programming language can help enforce separation of concerns by offering ways of creating separate name spaces. For that purpose, Java has packages, ML [32] has modules, and C++ has so-called namespaces. In each case, the idea is, for example, to enable the name space of the user interface to be separate from that of the database. The advantage is that the programmer of the user interface will not accidentally or purposefully manipulate the internals of the database. Furthermore, the programmer of the database can choose names of variables, procedures, etc without worrying about clashes with names used in the user interface code.

Good constructs for code management greatly enhance software engineering by teams of programmers where each programmer is responsible for a module. As long as the interfaces between the modules are well defined, each programmer can work independently.

Java packages are a simple mechanism for separation of concerns. Each package has a name and a separate name space, and it can access names from other packages only by explicitly importing them. For example, in Java, if we write

```
import java.util.*;
```

then we get access to all classes and interfaces defined in the package java.util.

The import statements make explicit the relationships with other modules and therefore avoid name clashes. The collection of Java classes is flat and unstructured; a Java package cannot be nested in another package. Moreover, a Java package is not a value; it cannot be stored in a variable, passed as an argument, or returned as a result.

In ML, a module is a first-class entity which can be stored and passed around in much the same way as a basic value such as an integer. There is a layering, though: functions that take modules as arguments are functors, not functions. So, functors are functions from modules to modules, while normal functions are functions from normal values to values.

ML modules and functors blend mechanisms for abstraction and for separation of concerns. Further blending of modules with mechanisms of object-oriented programming remains an active research area [20].

2.4 Concurrency Control

Goal: Control access to shared resources.

At the hardware level, synchronization of concurrent processes can be supported in a variety of ways. For example, there can be an atomic test-and-set operation that allows a shared register to be simultaneously read and written. Another example is an atomic operation for swapping the contents of two registers. Such operations enable a register to be a *lock* for a shared resource. With either of the two mentioned operations, one can ask for the lock, and possibly get it, in one computation step. If it had happened in two computation steps, then it is possible that two different concurrent processes simultaneously would ask for the lock, both find that the lock is not currently held, and then both processes take the lock. This would be a programming error.

Representing and operating on locks with atomic operations is fairly cumbersome and highly machine dependent. Programming languages tend to provide more convenient constructs for concurrency control. Those constructs can often be easily implemented using test-and-set, etc. A classical example is that of a semaphore [19], which is a data structure with just two operations: wait and signal. We can think of a semaphore as an abstraction of a lock. When a process wants access to a shared resource, it issues a wait to the semaphore that guards the resource. If no other process is currently accessing the resource, then the process is granted access right away and is now holding the lock. When the process is done with the access, it issues a signal to the semaphore, thereby releasing the lock. If the wait is issued while another process holds the lock, then the process will wait until the lock is released. In general, the semaphore may have a queue of processes waiting to access the shared resource.

A different construct for concurrency control is that of monitors [24]. A monitor is a data structure where we can think of all the operators as being guarded by a single semaphore. Whenever an operator is called, a wait to the semaphore is automatically issued, and when an operation is about to return, a signal is issued. The effect is that the data structure is accessed by at most one concurrent process at a time.

In Java, any variable can be used as a lock, and any block of code can be guarded by a lock. For example, we can write

```
synchronized(lock){ balance = balance + x; }.
```

Here we can think of lock as a semaphore on which a wait is automatically issued on entry to the guarded block of code, and on which a signal is automatically issued on exit. If every method of a class is guarded by *synchronized* on the same lock, then each object of that class is effectively a monitor.

Much research has gone into finding efficient implementations of the various constructs for concurrency control [5]. A radical idea is to let a compiler prove that a certain lock can be held by at most one process at a time. In such a case, the lock is unnecessary and the calls to be operations on it can be eliminated [1].

3 Correctness

We say that software is correct when it does what it is supposed to do. Most software has complex behavior and one can rarely get a waterproof guarantee that software is fully correct. However, a programming language can help establish partial correctness by offering ways of

specifying and checking key properties. A variety of approaches to software verification and validation have been devised, and some of them will be highlighted in the following discussion of several aspects of software correctness.

3.1 Memory Safety

Goal: Avoid jumps to data addresses and data manipulation of code addresses.

One of the most dreaded software errors is reported at run time as something like "illegal address—core dump." Such errors preclude graceful recovery and continuation of the computation, and they can be difficult to troubleshoot. They typically arise from careless manipulation of memory addresses which can look innocent in a machine code program. For example, a program may by mistake jump to a data address, or it may add a number to a code address and thereby get an illegal jump target. The idea of memory safety is to avoid all such errors.

For a given machine code program, it can be difficult to determine whether it is memory safe. Extensive software testing can increase confidence in memory safety, but, as always, testing can only show the presence of bugs, not prove the absence of bugs.

In high-level languages, such as Java, ML, and Scheme [15], memory safety is guaranteed via a type system. In these languages, data and code addresses are not manipulated directly; rather, the programmer works with convenient abstractions such as objects and functions. The type system enforces that those abstractions are used correctly. From the programmer's point of view, data and code manipulation errors no longer manifest themselves as core dumps at the hardware level; instead they show up as type errors that are reported in a more intelligible way by the run-time system of the language. For example, in Scheme, each data value is tagged with information about its type. The tag can be "integer," "procedure," "heap address," etc. Whenever a value is used, the run-time system checks whether the value is of the appropriate type. The run-time system will allow a number to be added to another number, but will disallow a number to be added to a procedure. A type error in Scheme will terminate the execution, but in a graceful way. An alternative is that the run-time system throws a type exception which can then, perhaps, be caught by an exception handler and thereby allow the computation to continue.

The Scheme style of type checking is known as dynamic type checking because it takes place at run time. It comes with an overhead in time and space because each value must be tagged and because the tags must be checked each time values are used. In contrast, ML has entirely static type checking. The idea is that before the program is run, a type checker will determine whether all data manipulation and procedure calls will operate on values of the correct types. If not, then the program is not executed at all. While this may seem restrictive, it has the benefit of saving time and space at run time, and of enforcing the program abstractions in a static way, that is, reporting problems early—even before running the program. Static type checking takes time itself, but is typically done along with compilation of the program and tends to take less time that the rest of the compilation process. Static type checking is a win-win technology: (1) problems are caught early, making it possible to complete software testing faster, and (2) it comes with no overhead for the running program.

In an object-oriented language such as Java, a data type can be a subtype of another type. For example, Plane and Helicopter can be two subtypes of Aircraft. The idea is that every Plane is an Aircraft, but not vice versa. If we have a variable of type Aircraft, the actual object residing in the variable may be a Plane or a Helicopter. Since the type system does not know which one, it would be a type error to do a Plane-specific operation on the variable. This is where type casts come in: we can cast the variable to be a Plane before operating on it. If the variable actually contains a Plane object, then all is well; otherwise a type cast exception is thrown. If the exception is not caught, then the effect is the same as a dynamic type error in Scheme.

The quest of type systems research is to strike a balance between expressiveness, that is, the number of programs that can be type checked, and efficiency, that is, the time it takes to do the type checking. One of the current active research areas is to combine static type systems for object-oriented languages with a notion of generic types [11].

3.2 Proof

Goal: A guarantee that the program meets the specification.

Mathematical proofs of program properties are based on a formal description of program behavior. A description of program behavior is also known as the semantics of a programming language. There are several approaches to formal semantics. Operational semantics [40] models a computation as a step-by-step process in which commands are executed one after the other. The execution process is modeled on a mathematical, high-level representation of the bits and bytes in an actual computer. Denotational semantics [34] formalizes a program as simply its input-output behavior, that is, in the simplest case, how it maps an initial state to a final state. Axiomatic semantics [25] specifies relationships between program states, such as if a predicate is true of a state, then after executing a command, a somewhat different predicate is true of the next state. Once a formal semantics is in place, we have a foundation for mathematical reasoning about programs. The properties we want to prove must also be stated formally.

In general, proving that a program has a property can be difficult. The ultimate goal of proving that a program meets its entire specification remains elusive and is getting increasingly problematic as software grows in size and complexity. One kind of program has received particular attention when it comes to proving correctness: compilers. The motivation is that compilers are an important part of today's computational infrastructure; if the compiler is not working correctly, then all bets are off. Moreover, many aspects of compilers are well understood which increases the chance that good proof techniques can be found [17]. One aspect of compiler correctness can be stated as follows:

> If a program p is compiled to code c, and evaluating p gives result r, then evaluating c gives a result which is a machine representation of r.

Notice that the semantics of both the source language and the target language play a crucial role in the statement of compiler correctness.

One of the goals of modern programming language design is to ensure that *all* programs in the language have a certain correctness property. One example is type soundness [31], which states:

Well-typed programs cannot go wrong.

Here, "well-typed" simply means that the program type checks, and "go wrong" is, intuitively, an abstraction of the error "illegal address—core dump" that one can encounter at the hardware level. The standard way of proving type soundness is to first prove two lemmas [37, 43]:

Preservation: If a well-typed program state takes a step, then the new program state is also well typed.

Progress: A well-typed program state is either done with its computation, or it can take a step.

The Preservation lemma says that if we have type checked a program, then during the computation, all reachable program states will also type check. The Progress lemma says that if we have reached a program state that type checks, then we cannot go wrong in the next step of computation. Together, the two lemmas are sufficient to prove type soundness: when we have type checked a program, Preservation ensures that we will only reach typable program states, and Progress ensures that in those states, the computation cannot go wrong.

The field of program synthesis [7] is concerned with generating programs from specifications. This obviates the need for proofs altogether; if we can prove the correctness of the program generator, of course!

An active research area is that of proving the correctness of optimizing compilers [28]. Some optimizations radically change the code and remain a challenge to prove correct.

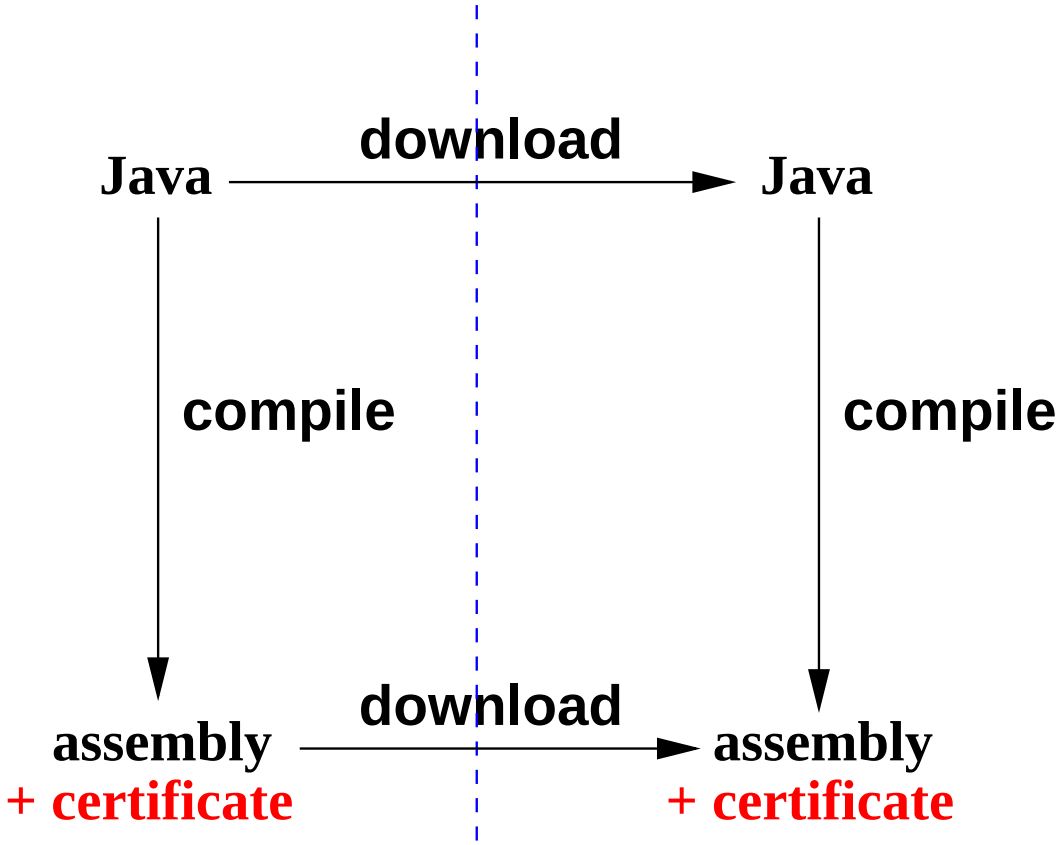
3.3 Certification

Goal: Verify a certificate before executing the program.

Clicking on a link may lead to the downloading and execution of Java bytecode. In addition, the bytecodes will be verified before execution, giving the same kind of guarantee as we get from type checking source code. This can be done easily because Java bytecode contains type information that guides the verifier. We can view this extra type information as a certificate which is checked by the bytecode verifier. If the raw bytecode does not match the certificate, then the bytecode will not be executed. The combination of code and certificate achieves a level of tamperproofing: one can try to tamper with either the code, the certificate, or both, but if the end result does not verify, then it will not be executed. In Java, bytecode verification first and foremost guarantees memory safety. Even if an attacker manages to change the code and the certificate in such a way that the result verifies, the code will be memory safe. It may no longer compute factorial, but will not crash your computer!

A weakness of the Java bytecode verification model is that after the verification is done, it may be necessary to compile the bytecode in order to achieve efficient execution. This entails that the compiler is part of the trusted computing base: if the compiler is faulty, then the guarantees obtained at the bytecode level are worthless. This raises the challenge of whether we can avoid trusting the compiler and have a certificate for machine code. The idea is that, instead of downloading Java bytecodes, we would like to download machine code

plus a certificate which can be verified before execution. The certificate will be issued by the code producer, and there will be nothing to compile on the receiving end, thus obviating the need for a compiler in the trusted computing base.



The obvious place to put the burden of producing the certificate is with the compiler. This leads to the concept of a certifying compiler, which not only produces memory-safe code, but also a certificate that the code is memory safe.

One approach to low-level certified code is typed assembly language [33]. The idea is the same as for Java bytecode: extra type annotations in the code guide the type checker, and type checking guarantees memory safety. One of the main challenges in designing a typed assembly language is that high-level constructs and values, such as procedures and objects, at the assembly level are represented by a multitude of individual entities and instructions. For example, there may be one type rule for a Java method call, but at the machine level, the method call is represented by several instructions, each of which must type check. Another challenge is to design a type system that will be a convenient target for the compilation of a wide variety of source languages. The ultimate typed assembly language would enable easy compilation of all source languages. This goal seems elusive, and it may be that in the future there will be several competing type systems for the same assembly language.

While certified assembly code does eliminate the compiler from the trusted computing base, it does introduce the need for having a verifier in the trusted computing base. The good news is that a verifier tends to be much simpler than a compiler.

If we allow certificates to be written as proofs in a full-blown logic, then the certified code is known as proof-carrying code [36]. This is vastly more powerful and flexible than typed assembly language, but it is also more difficult to produce the proofs. It remains an active research area to develop industrial-strength certifying compilers that produce proof-carrying code. Another active research area concerns producing better certified bytecode representations of high-level programs [3].

3.4 Bug Finding

Goal: Find bugs faster than via software testing.

In most software engineering efforts, more time is spent on testing software than on writing software. The goal of software testing is to find as many bugs as possible. It is widely believed that no large piece of software can be bug free, even after extensive testing. Part of the testing process can be automated, but testing remains labor intensive and therefore costly. It is particularly time consuming to invent test cases and to figure out what the correct output should be.

Modern programming languages help with bug finding, even without running the program. Structured programming, variables instead of registers, static type checking, automatic memory management, and certifying compilers all contribute to lowering the number of bugs. Still, a program can contain errors in the program logic that won't be caught by, say, the Java type system.

The idea of model checking [14] is to try to find bugs in a model, that is, an abstraction of the program. If the model faithfully preserves some aspects of the program and eliminates others, then it will preserve some bugs and eliminate others. Thus, if we can find bugs in the model, then those bugs will likely correspond to bugs in the program. Creating good models is difficult. On one hand, we want small models that will make powerful bug finding algorithms feasible, and on the other hand, we want models that are large enough to contain at least some of the bugs.

For example, suppose we have a program and want to check whether all read operations on a file only happen after the file has been opened. We can create a model which eliminates all operations on data other than files, and which for every conditional statement embodies the abstraction that both branches are executable. This abstraction may well eliminate some bugs in the part of the code that is not concerned with files. However, if we find a read on a file without a preceding open operation, then chances are that we have identified a bug in the original program.

A model checking problem has two components: the model and the property we want to check. To enable flexible bug finding, we want to have a property language in which we can express a variety of properties to be checked. Otherwise, we would need a separate model checking algorithm for every property. One of the popular approaches to designing a property language is to base it on regular expressions. For example, we might choose the alphabet {open, read, write, close}, and check the property:

$$\text{open} \cdot \text{read}^*$$

We can understand the property as stating that a file must be opened exactly once, and only after that can it be read any number of times. A somewhat different property could be:

open · (read | write)* · close

The same general model checking algorithm would handle both properties and possibly report bugs.

While it is easiest to build the model independently of the desired property, it can lead to faster model checking if the model is property driven. For example, the first property above has no need for a model with write and close operations.

Model checking has been immensely successful at bug finding for hardware. Software model checking has turned out to be more difficult and remains an active research area [16, 6].

4 Efficiency

We say that software is efficient when it can do its job using the available resources. A variety of resources can be available to software, including time, space, power, databases, and networks. When resources are constrained, a programming language can help with using them judiciously by offering resource-aware language constructs and compilers. A variety of approaches to resource awareness have been devised, and some of them will be highlighted in the following discussion of several aspects of efficiency.

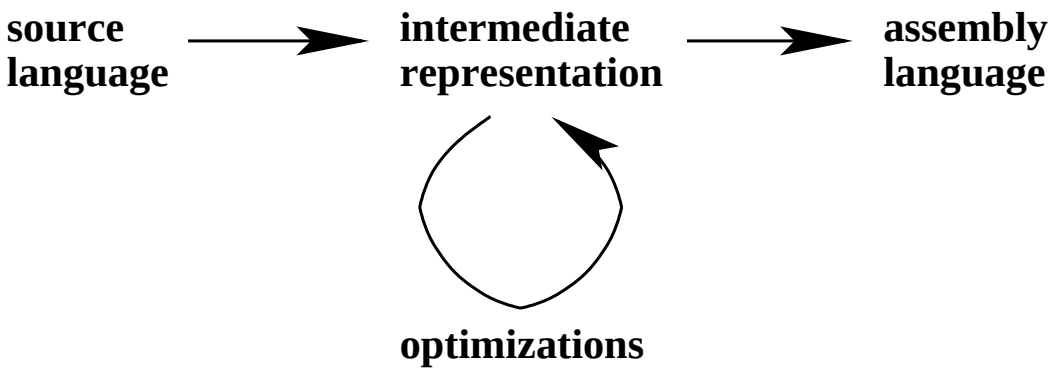
4.1 Execution Time

Goal: Complete the task faster.

For desktop computing, users want spreadsheets that update the fields quickly after changes, they want internet searches to complete quickly, they want games with fancy graphics and computer players that move quickly, and so on. High speed is in practice achieved by a combination of good algorithms, careful coding, good compilers, etc.

A good programming language can contribute to fast execution time in at least two ways. First, the language can make it easier to express efficient algorithms. For example, a recursive algorithm such as quicksort is easier to express in a language such as C than in assembly language because recursion is supported in C. Second, the compiler can have a major impact by doing good instruction selection, register allocation, etc. The difference between a simple compiler and a highly optimizing compiler can be several factors when measuring execution speed.

The traditional role of a compiler is to do its job "way ahead of time" (WAOT), that is, before the compiled code is run. A WAOT compiler can take as long time as it likes, as long as it generates efficient code. Usually, a highly-optimizing compiler has the following structure:



Each optimization step may use a static analysis that determines useful facts about the program. For example, a static analysis in a Java compiler may determine that certain call sites can be inlined, and the optimizer may then go ahead and inline the calls. In more detail, consider a Java call site x.m() where x is variable and m is a method name. From the syntax alone, we cannot determine which method will be called. However, a static analysis might determine that only objects of class C will be assigned to x, so only the m method in C can be called, and therefore we can inline it [22].

In Java, classes can be loaded and used while a program is running. If we want newly loaded code to execute fast, then we need to compile "just in time" (JIT), that is, after the code is loaded but before it is executed. A JIT compiler is executed at run time, so the compilation time directly impacts the run time. This creates a trade off between compilation time and run time. For example, an inlining optimization for Java may take 4 seconds to execute on a large program and make the program execute 3 seconds faster. This is fine for a WAOT compiler. In a JIT compiler, we would prefer a simpler inlining optimization that takes just 1 second to execute and makes the program execute 2 seconds faster. Each optimization needs to "pay for itself" by creating a larger speed up than the time it takes to do the optimization.

Some program analyses and optimizations work on a per-procedure basis. That property make them good candidates for JIT compilers. Other program analyses are inherently whole-program analyses. Those analyses do not work so well for languages such as Java because the program can change after the analysis is done, due to dynamic class loading.

Some program analyses take advantage of being applied to programs that have already been type checked. Such analyses are called type-base analyses [38] and are the topic of much current research. Another active research area is to find fast and effective program analyses for use in JIT compilers [18, 41].

4.2 Memory Management

Goal: Use less space.

Many efficient algorithms require the use of pointers and dynamically allocated data structures. In languages such as C, programs manage a memory space known as the "heap" for dynamically allocated data. A program can allocate and deallocate heap space, and it can access and update heap objects. The "do-it-yourself" style of heap management can lead to efficient programs but also to nasty bugs. For example, a program may reference

a memory area after it has been deallocated, a bug known as the "dangling pointer" error. A program might also keep allocating heap space indefinitely, a bug known as a "memory leak." Some of these problems can be avoided by dealing with the heap at a higher level of abstraction and leaving the details to a memory management system. The memory manager is a part of the run-time system which is present during the execution of a program.

For example, in Java all dynamic memory allocation is done with expressions of the form

```
new C()
```

or something similar, where C is a class name. The memory manager will allocate space for a C object, that is, sufficient heap space for representing the fields and methods of a C object. When the program can no longer reach the object, then the memory manager will automatically deallocate it. The automatic deallocation, also known as garbage collection, ensures that dangling-pointer errors cannot occur.

A type system can help the compiler when it computes how much space is needed for the fields of an object. In Java, a field of type int takes four bytes, while a field of type long takes eight bytes. For a language without a static type system, it may not be easy to determine ahead of time what will be stored in a field. The usual solution is to "box" the data in the field, that is, instead of storing the data itself in the field, the program will store a pointer to the data. Given that all pointers have the same size, it is easy to determine the size of each field.

A compiler can help save data space by doing a data-flow analysis to determine, for example, whether the values stored in a field of type long really go beyond what could be stored in a field of type int. If not, then the compiler can treat the field as if it is of type int, and thereby save four bytes. Such an optimization is particularly important in embedded systems where memory may be limited [4].

The use of dynamic memory allocation can make it difficult to determine an upper bound on the need for heap space. As a consequence, some embedded software refrains entirely from dynamic memory allocation and instead allocates all data in a global area or on a stack. Even without dynamic memory allocation, it can be difficult to determine an upper bound on the need for stack space [12]. In particular, recursion makes it difficult to find such upper bounds. So, some embedded software also refrains from using recursion.

A compiler can help save stack space by inlining procedure calls. If a procedure is called more than once, such inlining may in turn increase the code size, creating a trade off between stack size and code size.

Current research includes developing memory managers that can allocate and deallocate data more efficiently and with smaller overhead [9]. Another active area is the development of methods for statically determining the need for heap space in recursive programs with dynamic memory allocation [26]. Yet another active area is resource-aware compilation [35, 39], where the compiler is told up front about resource constraints, such as memory limits, and the compiler then generates code that meets the requirements, or tells the programmer that it cannot be done.

5 Concluding Remarks

As the need for software grows, so does the need for better programming languages. New directions include power-aware compilers that can help save energy in embedded systems, and XML processing languages for programming of web services.

Acknowledgments. Thanks to Mayur Naik, Vidyut Samanta, and Thomas VanDrunen for helpful comments of a draft of the chapter.

References

[1] Jonathan Aldrich, Craig Chambers, Emin Gün Sirer, and Susan J. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *Proceedings of SAS'99, 6th International Static Analysis Symposium*, pages 19–38. Springer-Verlag (*LNCS* 1694), 1999.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno virtual machine. *IBM System Journal*, 39(1), February 2000.

[3] Wolfram Amme, Niall Dalton, Michael Franz, and Jeffery Von Ronne. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of PLDI'01, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 137–147, 2001.

[4] C. Scott Ananian and Martin Rinard. Data size optimizations for Java programs. In *LCTES'03, Languages, Compilers, and Tools for Embedded Systems*, 2003.

[5] David F. Bacon, Ravi B. Konuru, Chet Murthy, and Mauricio J. Serrano. Thin locks: Featherweight synchronization for Java. In *Proceedings of PLDI'98, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, 1998.

[6] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of PLDI'01, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[7] David A. Basin. The next 700 synthesis calculi. In *Proceedings of FME'02, International Symposium of Formal Methods Europe*, page 430. Springer-Verlag (*LNCS* 2391), 2002.

[8] Gerald Baumgartner, Konstantin Läufer, and Vincent F. Russo. On the interaction of object-oriented design patterns and programming languages. Technical Report CSD-TR-96-020, 1998. citeseer.nj.nec.com/baumgartner96interaction.html.

[9] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *Proceedings of OOPSLA'02, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 1–12, 2002.

[10] Thomas J. Bergin and Richard G. Gibson. *History of Programming Languages (Proceedings)*. Addison-Wesley, 1993.

[11] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of OOPSLA'98, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 183–200, 1998.

[12] Dennis Brylow, Niels Damgaard, and Jens Palsberg. Static checking of interrupt-driven software. In *Proceedings of ICSE'01, 23rd International Conference on Software Engineering*, pages 47–56, Toronto, May 2001.

[13] Craig Chambers, Bill Harrison, and John M. Vlissides. A debate on language and tool support for design patterns. In *Proceedings of POPL'00, 27nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 277–289, 2000.

[14] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.

[15] William Clinger and Jonanthan Rees (editors). *Revised*^{4} *Report on the Algortihmic Language Scheme*, November 1991.

[16] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of ICSE'00, 22nd International Conference on Software Engineering*, pages 439–448, 2000.

[17] Joëlle Despeyroux. Proof of translation in natural semantics. In *LICS'86, First Symposium on Logic in Computer Science*, pages 193–205, June 1986.

[18] David Detlefs and Ole Agesen. Inlining of virtual methods. In *Proceedings of ECOOP'99, European Conference on Object-Oriented Programming*, pages 258–278. Springer-Verlag (*LNCS* 1628), 1999.

[19] E. W. Dijkstra. The structure of the t.h.e. multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[20] Kathleen Fisher and John Reppy. Extending Moby with inheritance-based subtyping. In *Proceedings of ECOOP'00, European Conference on Object-Oriented Programming*, pages 83–107. Springer-Verlag (*LNCS* 1850), 2000.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[22] Neal Glew and Jens Palsberg. Type-safe method inlining. In *Proceedings of ECOOP'02, European Conference on Object-Oriented Programming*, pages 525–544. Springer-Verlag (*LNCS* 2374), Malaga, Spain, June 2002.

[23] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[24] C. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557., October 1974.

[25] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[26] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of POPL'03, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 185–197, 2003.

[27] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[28] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proceedings of POPL'02, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 283–294, 2002.

[29] Microsoft. Microsoft Visual C#. http://msdn.microsoft.com/vcsharp.

[30] Microsoft. The .NET common language runtime. http://msdn.microsoft.com/net.

[31] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[32] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[33] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Progamming Languages and Systems*, 21(3):528–569, May 1999.

[34] Peter D. Mosses. Denotational semantics. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science*, volume B, chapter 11, pages 575–631. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[35] Mayur Naik and Jens Palsberg. Compiling with code-size constraints. In *LCTES'02, Languages, Compilers, and Tools for Embedded Systems joint with SCOPES'02, Software and Compilers for Embedded Systems*, pages 120–129, Berlin, Germany, June 2002.

[36] George Necula. Proof-carrying code. In *Proceedings of POPL'97, 24th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[37] Flemming Nielson. The typed lambda-calculus with first-class processes. In *Proceedings of PARLE*, pages 357–373, April 1989.

[38] Jens Palsberg. Type-based analysis and applications. In *Proceedings of PASTE'01, ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools*, pages 20–27, Snowbird, Utah, June 2001. Invited paper.

[39] Jens Palsberg and Di Ma. A typed interrupt calculus. In *FTRTFT'02, 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, pages 291–310. Springer-Verlag (*LNCS* 2469), Oldenburg, Germany, September 2002.

[40] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN–19, Computer Science Department, Aarhus University, September 1981.

[41] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Progamming Languages and Systems*, 21(5):895–913, 1999.

[42] Richard L. Wexelblat. *History of Programming Languages (Proceedings).* Academic Press, 1981.

[43] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.