Editors: Konrad Hinsen, konrad.hinsen@cnrs-orleans.fr
Konstantin Läufer, laufer@cs.luc.edu

A Glimpse of the Future of Scientific Programming

By Konrad Hinsen

Technology being developed right now in computer science labs might change the way scientists and engineers write programs in the future.

Every scientist who has done some programming is familiar with using a compiler: First you write a program in the form of text files containing the source code, and then you run a compiler that reads the source code and produces an executable program for the kind of machine you're using. If you want to change your program, you modify the source code and recompile. This is known as the edit-compile-run cycle, and it hasn't really changed since the early days of scientific computing in the 1960s, when the transition from assembly language to compiled Fortran began to make computing accessible to a much wider group of users.

A second program development style that many of us are familiar with is the use of interpreters: Instead of compiling the source code to produce an executable program, you directly run an interpreter on it. By eliminating the compilation step, you can develop and test faster. Moreover, interpreters also permit interactive development. The price to pay is performance, because interpreters are usually much slower than optimized executables produced by a compiler. Interpreters have been around for almost as long as compilers, starting with the first Lisp interpreter in the early 1960s.

Most programming languages were designed with either compilation or interpretation in mind. Fortran, C, and C++ are classic compiler languages, whereas Python or Matlab are usually interpreted. In principle, every language can be both compiled and interpreted, but often one or the other technique simply doesn't make much sense. For example, there's nothing much to be gained by compiling Python, because the dynamic nature of the language prevents the optimizations that compilers typically apply.

Many programming tools that seem (to their users) like compilers or interpreters are actually hybrids. The Python interpreter, for example, compiles Python source code to an intermediate representation called *bytecode*, and then it interprets this bytecode. This is done for improving performance, as it permits the Python interpreter to do a few processing steps, such as checking syntax or removing comments, only once. The Java compiler also produces bytecode, which in early Java implementations was also interpreted. The reason why we classify Python as an interpreted language but Java as a compiled language is partly superficial: Python hides its bytecode from programmers, whereas Java stores it in a file for further processing by other tools. However, there's also a deeper reason for the distinction, because the division of work between the compilation and the interpretation phase is quite different. Python bytecode is so similar to the source code that it can easily be "uncompiled" into a readable form, whereas the Java bytecode is more heavily processed.

The current trend in programming tools is to further dissolve the borderline between compilers and interpreters, and to add more layers to the transformation process from source code to executable binary code. Although these techniques aren't very visible yet in scientific programming, they're starting to be used in other application domains. In this article, I provide a short overview over these developments, many of which are likely to find their way into computational science in the next few years.

Simpler Programming, Fewer Mistakes, Better Performance

Before discussing the techniques that are likely to change our programming habits, it's worth thinking a bit about the problems they're trying to solve. One goal should be obvious: We always want our results faster, so better performance is one of the major driving forces. The second goal is to make program development simpler and safer. Here, simpler means that programmers should be able to work at a level closer to their problem specifications, which in scientific computing often means mathematical equations. Safer means that the program development tools should be able to catch more mistakes, increasing our confidence that our programs actually do what we think they do.

One approach that's already being applied by some pioneers in scientific

computing is to define domain-specific programming languages. A domain-specific language (DSL) differs from a general-purpose language by having constructs tailor-made for the concepts used in a specific application domain, and intentionally lacking constructs that aren't important for that domain. For example, a DSL for signal processing would have a data type for a *time series* instead of a generic type such as *array*, and built-in operators for Fourier and Wavelet transforms. At first sight this looks quite similar to what a library for signal processing would contain. The difference between a library and a DSL is that the DSL has its own compiler that knows about the domain-specific constructs. This lets DSL do better error checking and generate better code. We've published previous examples of scientific DSLs in the "Scientific Programming" department.[1–3]

There are two different techniques for implementing DSLs, and both are worth knowing about because they can also be of use in other contexts. The first approach, *code generation*, is the one chosen by Andy Terrel for his integration example.[1] It consists of writing a compiler that translates the DSL to some standard programming language (C in Andy's case). Writing a compiler sounds like an enormous amount of work, but it isn't as bad as it seems at first. A DSL is much simpler than a general-purpose language such as C, and C code generation is simpler than compiling to machine language because there are no platform-specific issues to deal with. Moreover, some of the ingredients of a compiler are readily available in libraries nowadays. For example, Andy's DSL compiler takes advantage of the Python language and the SymPy library (http://sympy.org).

The other technique for implementing DSLs is called *metaprogramming*. It consists of writing code that manipulates other code before passing it to the compiler. There's no clear-cut borderline between code generation and metaprogramming, but both can be seen as variants of the same idea. However, the term metaprogramming is most commonly applied when the manipulation of the initial source code and the subsequent compilation to machine code are tightly integrated, meaning that the programming language being used has explicit support for metaprogramming. Although this idea is rather old (it became common in the Lisp language in the 1960s), languages with explicit metaprogramming support are still exceptional, and none of the languages popular in scientific computing is among them. C++ comes close with its template system, which wasn't designed for metaprogramming but is powerful enough to do the job. While basic C++ templates are a rather cumbersome way to do metaprogramming, the Boost.Proto library (www.boost.org/doc/libs/1_51_0/doc/html/proto.html) provides support code that makes C++ a serious candidate for implementing DSLs, even though it still involves more work than a language with proper metaprogramming support. In the "Scientific Programming" department, we've presented two examples of C++ metaprogramming for implementing parallel skeletons[2] and GPU programming.[3] Researchers in computer science also explore new languages with better support for DSL construction. An interesting example is the use of the Scala language in the Delite project (http://stanford-ppl.github.com/Delite/index.html), which uses DSLs for automatic parallelization. For a simple example in Lisp, see the sidebar "Implementing a Domain-Specific Language by Metaprogramming."

Metaprogramming has one advantage over code generation: DSLs implemeted via metaprogramming are better integrated with the underlying language, to the point that they're often referred to as embedded DSLs. There are two aspects to this integration: First, application programs can freely mix the DSL features with the underlying language's features. Second, a single set of tools handles everything. C++ programs based on templates or Lisp programs using macros require nothing but the respective compilers and specific libraries. By contrast, DSLs implemented via code generation typically require a compiler for the generated code in addition to whatever tools handle the primary language, which can make deployment more difficult for users of the resulting software package.

Code Optimization at Runtime

Some interesting innovation also happens at the other end of the code-transformation process, where processor-specific machine code is generated. Traditional compilers use nothing but the program's source code in this process. They don't have any information about how this code will actually be used. Not only must they generate code that works in all circumstances that are allowed by the programming language's specification, but they can't even optimize the code for the use cases that will matter most during execution. Consider, for example, a function of a real number that will in practice only be evaluated for positive arguments. There's no way to provide this information to the compiler in the source code, so the compiler must generate code that also works for

Implementing a Domain-Specific Language by Metaprogramming

As an example for implementing a Domain-Specific Language (DSL) by metaprogramming, I provide an overview of an implementation of a DSL for computing with physical units in Clojure, a modern dialect of Lisp.[1] The full implementation of this DSL is available at http://code.google.com/p/clj-units.

Libraries for working with physical units have been written for many programming languages. Most of them follow the same basic approach: Define a data type that represents a physical quantity, consisting of a number and a unit, and then implement arithmetic on this data type that checks the compatibility of the units of the arguments and computes the units of the result. The Clojure library works in exactly the same way. The DSL comes into play for defining units and unit systems, something that in most other libraries is complicated or impossible.

First, let's look at how the DSL is used. The International System of Units (SI system) is represented by its base units as follows:

```
(defunitsystem SI
  length                "meter"        m
  mass                  "kilogram"     kg
  time                  "second"       s
  electric-current      "ampere"       A
  temperature           "kelvin"       K
  luminous-intensity    "candela"      cd
  amount-of-substance   "mole"         mol)
```

Each line gives the name of the dimension (length, time, and so on), the name of the corresponding SI unit, and the shorthand symbol for the unit. Derived dimensions, with or without associated unit names, can be defined by reference to the base units:

```
(defdimension area
  [length 2])
(defdimension force "newton" N
  [mass 1 length 1 time -2])
```

Finally, it's possible to define additional units for previously defined dimensions:

```
(defunit min "minute" (* 60 s))
```

The definitions shown above generate data structures that store the information about the units, dimensions, and their relations. They also generate functions which are used to work with physical quantities in application programs. For example, the kinetic energy of a point mass can be computed as follows:

```
(defn kinetic-energy [m v]
   {:pre [(mass? m) (pos? m) (velocity? v)]
    :post [(energy? %)]}
   (* 1/2 m v v))

(prn (kinetic-energy (kg 1000) (/ (km 50)
  (h 1))))
```

The function `kinetic-energy` checks that its arguments have the dimensions of mass and velocity, respectively, and also verifies that the mass is positive. It then calculates the kinetic energy and checks that the result is indeed an energy. This last step isn't necessary in principle, because the product of a mass and a squared velocity must be an energy—but it never hurts to double check.

Before looking at the implementation techniques for this DSL, it's useful to consider why it can't be implemented

negative arguments. If the function can be simplified for positive arguments, an optimization opportunity is lost.

One way to obtain such information is to observe the program when it's executed and generate a protocol of which functions are called with which arguments and how often. The protocol could then be used in a second compiler pass for additional optimization. Of course, the compiler would still have to generate code that works in all possible situations, as the program is likely to be run with different input. However, it could generate optimized code for cases that have been observed to be frequent.

Going back and forth between compilation and execution under supervision is a tedious and error-prone task for the user, which is why optimization based on runtime profiles never became popular. This has changed with the invention of the *Just-in-Time* (JIT) compilation, a technique that was popularized by HotSpot, a Java virtual machine (JVM) implementation making extensive use of optimization at runtime. It has promoted Java from a language known for its slowness to a serious competitor for C and C++ in terms of performance. The PyPy project (http://pypy.org) has applied the same ideas to the Python language, yielding spectacular speedups. A JIT compiler starts from an intermediate-level code representation such as bytecode. Initially, this bytecode is interpreted or compiled without any optimization to get the program running. As execution progresses, frequently called functions are compiled with more aggressive optimization, which moreover is based on the program's execution profile. The longer a program runs, the faster it becomes.

JIT compilation doesn't require any changes to the source code and doesn't change a program's results. At first sight, programmers needn't even

as a library module containing function and data-type definitions. One reason is that `defunitsystem`, `defdimension`, and `defunit` do not just compute values, they also create new functions and variables. For example, `defunitsystem` creates a test function for each dimension, such as `mass?` and `time?`. Although there are programming languages that allow a function to create another function (for example, Python), I'm not aware of any language that would permit attaching such a dynamically created function to a dynamically created name in the global scope, at least not without some obscure (and usually discouraged) trickery. Moreover, if `defunitsystem` were a function, its first argument wouldn't be the symbol `length`, but the value of the variable `length`, which doesn't exist before `defunitsystem` is called.

Clojure (like other Lisp dialects) provides a special construct for such definitions, which is known as a *macro*. A macro is in fact a function, but a function that's called as part of the compilation process, whereas a normal function is called when the program is executed. As an illustration, let's look at a simplified version of the definition of `defunit`:

```
(defmacro defunit
  [unit-symbol unit-name quantity]
  '(def ~unit-symbol
     (as-unit ~quantity (quote ~(symbol
       unit-name)) (quote ~unit-symbol))))
```

When the compiler processes the statement

```
(defunit min "minute" (* 60 s))
```

it calls `defunit` with the arguments `min`, `"minute"`, and `(* 60 s)`, without preprocessing them in any way. This is greatly facilitated by the peculiar syntax common to Lisp dialects: The only syntax rules that exist describe data structures, in particular the "nested list" data structure that we've seen in the examples, and a program's statements and functions are then encoded as pieces of data. It's therefore simple to pass pieces of code to a macro that treats them as data and returns another data structure that the compiler then interprets as program code.

The macro `defunit` inserts its arguments into a template, indicated by the back-tick symbol. Every expression in that template starting with a tilde is replaced by its value. For our example, `defunit` thus returns

```
(def min
 (as-unit (* 60 s)
      (quote minute) (quote min)))
```

and that's what the compiler compiles instead of the original statement. The substituted code defines a variable `min` with an initial value calculated by calling the function `as-unit`, which is a standard function that adds an entry to the unit-conversion table saying that one minute should be replaced by 60 seconds, and returns a data structure that defines the unit "minute."

Although many Lisp macros do little more than insert their arguments into a template, there are also much more complex macros. In fact, a macro can use all of the data manipulation techniques provided by the language to inspect and transform its arguments. As an extreme case, a macro could implement a complete compiler. For further study, Paul Graham's classic book *On Lisp*[2] provides an in-depth discussion of Lisp macros and their applications.

References

1. K. Hinsen, "The Promises of Functional Programming," *Computing in Science & Eng.*, vol. 11, no. 4, 2009, pp. 86–90.
2. P. Graham, *On Lisp*, Prentice Hall, 1993; www.paulgraham.com/onlisp.html.

know about it. They could just wait for the ever-better JIT compilers to speed up their code ever more. However, for applications where performance really matters, this isn't quite true: programmers must estimate the performance impact of choices in algorithms, data structures, and their implementations while writing their code. In this respect, JIT compilation is a mixed blessing: It can improve performance, but at the same time makes it unpredictable and even hard to understand a posteriori.

JIT compilation is still a relatively new technique, so we can expect it to evolve and mature over the next few years. One exciting possibility is to integrate parallelization into the JIT compilation process. One of the obstacles that's prevented automatic parallelization by compilers is the difficulty of estimating the runtime of a piece of code by inspecting its source code. A JIT compiler could rely on the measured runtime of the program to choose the granularity of parallelism in a data-parallel program, for example.

Will these new technologies have an impact on computational science in the near future? I'm rather confident that the answer is yes for DSLs. In addition to the advantages I've outlined, they have one more attractive feature for computational science. Scientific software development is increasingly handled by specialists, but nevertheless end users (that is, scientists applying numerical methods) still need to define the top layer of their algorithms themselves. A scientific DSL is situated at exactly that interface between software specialists and advanced software users. The fact that computational science DSLs are already appearing also suggests that they provide a real benefit.

However, the use of DSLs has some implications for how we'll be

doing computational science. First of all, designing and implementing DSLs requires competences that most computational scientists don't have. This task will thus require a software development team, which exists only in well-established and important application domains. Second, DSLs are either embedded into a language that supports metaprogramming, which in today's computational science landscape means C++, or implemented via a code generator that typically leverages some interpreted language's infrastructure. Scientists working with languages such as Fortran or Matlab will thus find it more difficult to make the transition to a DSL-based development system. Of course, nothing prevents the design of a DSL with syntax resembling Matlab or Fortran, it's just a lot more work because an additional infrastructure must be developed.

The future of bytecode-based virtual machines with JIT compilers performing adaptive optimization is less clear in computational science. The currently established ones (HotSpot for Java, .NET and Mono for Microsoft's Common Language Infrastructure [CLI]) weren't designed for the requirements of computational science and have some serious shortcomings for numerical applications. Developing a comparable system optimized for scientific computing is certainly feasible, but unlikely to happen as long as most computational scientists have never experienced the benefits of such an approach. CiSE

References

1. A.R. Terrel, "From Equations to Code: Automated Scientific Computing," *Computing in Science & Eng.*, vol. 13, no. 2, 2011, pp. 78–82.
2. J. Falcou, "Parallel Programming with Skeletons," *Computing in Science & Eng.*, vol. 11, no. 3, 2009, pp. 58–63.
3. P. Esterie et al., "Exploiting Multimedia Extensions in C++: A Portable Approach," *Computing in Science & Eng.*, vol. 14, no. 5, 2012, pp. 72–77.

Konrad Hinsen is a researcher at the Centre de Biophysique Moléculaire in Orléans, France, and at the Synchrotron Soleil in Saint Aubin, France. His research interests include protein structure and dynamics as well as scientific computing. Hinsen has a PhD in theoretical physics from RWTH Aachen University (Germany). Contact him at konrad.hinsen@cnrs-orleans.fr.