Discussion: Views of Genericity

Ulrich W. Eisenecker: I would like generic programming to simplify the programming process — so that we have to write fewer lines of code, for instance, and the lines we do write are more readable, and easier to maintain. What are the techniques we need to reach these goals? To engage in some self-criticism: when I consider C++ and specifically template meta-programming code, I have some doubts whether this is easy to read and to understand.

Arturo J. Sánchez-Ruíz: There should be no doubt: it is completely unreadable!

S. Doaitse Swierstra: For me, being a Haskell programmer, listening to those talks and seeing that template code is like going to a horror show. [*laughter*]

Eerke A. Boiten: I don't think generic programs need to be readable. Requiring generic programs to be readable is the poor man's approach to generic programming. Their interfaces and specifications should be readable, but they're made generic precisely because they are going to be used many more times than a non-generic program would be.

SDS: It should be just as readable as a mathematics book.

C. Barry Jay: I would put it slightly differently, and say that readability is context sensitive. If you are trying to improve or understand a generic function, you have to be able to read the code and understand what it is doing. If you're using it, you're thinking about your domain-specific issues, and the context switch required to understand this generic business might be enormous. So I think readability really depends on the context in which you are reading it. If generic programs are unreadable to the people who are trying to maintain them, that's a problem; if people with a domain-specific application have to understand things about the generic program that go beyond their application, that's a problem. But beyond those situations, readability is not such an issue.

* * *

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35672-3_13

J. Gibbons et al. (eds.), *Generic Programming*

Tim Sheard: To make programs simpler, I think it is really important to have languages which are layered, with explicit boundaries. For instance, consider the DrScheme system, which is a sort of teaching environment for teaching programming in Scheme. This implementation has about five or six subsets; each one is completely inside the other. This is a very good thing, because you can program in a subset, and none of the features of the enclosing subsets are visible from within the subset. So you don't get horrible error messages.

It seems to me that if we really want to push generics, we're going to need a mechanism like that. For example, we might have a Hindley-Milner subset; when you program in this subset, you're only allowed to do certain things, and you will only see error messages which relate to that kind of programming. You surround that subset with some richer language, and it's a kind of rite of passage to get to program in a bigger subset. As you go up through the subsets, the programming gets harder and harder, and maybe there are more and more requirements on the programmer to provide information that is used to prove safety: you might have to give bigger and stronger hints, and so on. I think that's really going to be the only possible model. Without something like that, a truly generic language will be unusable by mostly everyone.

Conor T. McBride: I think it's perfectly reasonable that at lower levels in this learning process, you nonetheless get access to libraries which have an interface at the lower level, but are implemented in terms of higher level operations. That's where the work I was talking about fits in. I don't think every programmer should get up in the morning and construct a universe full of data types; some library writer should manufacture that package of generic functionality, and provides an interface to it through suitable programming tools. So what is currently the domain of compiler writers becomes the job of experts nonetheless, but programmers *in* the language rather than implementors *of* the language. The interfaces to those libraries are perhaps comprehensible to less advanced users, even if the actual implementations are not.

SDS: But that raises a whole new issue of how to determine those nice layers.

Robert S. Cartwright: That's right. And it's very important to support those layers, because you need a separate parser, at least at the level of scanning the abstract syntax tree, to make sure that things are well-formed. You probably can't even use the same universal parser, because it will recognize things from one level that are illegal at another, and give the wrong diagnostics.

SDS: We have a compiler for a subset of Haskell in which basically there are no monads, no classes whatsoever — but you still want to provide I/O. So you don't want the students to really use the *IO* monad, because then they will get very strange error messages that they don't understand. If you hide that level, and you want to show something, make sure that any error messages or other feedback that you want to give to the user are not expressed in the language of those higher levels.

TS: In some sense, building such tools could be an exercise in generic programming, self-applied. It's an interesting incestuous research agenda, but it's important I think: most programmers will not write generic programs, though most of them would love to use them. To make a language or a programming environment where that's possible is an important goal for the community.

* * *

SDS: I think that dependent types, Generic Haskell, and the C++ library approaches all have in common that you program your compiler. You have a mechanism for having your compiler generating or producing extra code, which might involve inspecting type information which has been deduced by the compiler already. In the extreme case you get dependent types, but it also covers partial evaluation, for instance, in which you use static analysis for specializing code with static data.

UWE: Currently I'm interested in developing programs for creating software systems families or product line architectures. I'm trying to develop a domain-specific language for specifying such families. I really see advantages there for understanding the specifications, and for maintaining them because you can design domain-specific languages in such a way that you can extend them easily.

SDS: But that is the case for many languages. As soon as you start using an abstraction mechanism, you have extended your language. That holds for any language, even Fortran.

UWE: I want to have the possibility to write libraries that participate in the compilation process, because then I can experiment much more quickly with new concepts and new ideas, and I can even configure my development environment. I want to have an programming environment which I can extend using a domain-specific language.

* * *

UWE: My impression of yesterday and today is that we have a certain common idea of what genericity is, but there is no *definition* of genericity that comprises all the variants we have seen yesterday and today. Do we have some way to *measure* genericity? Is there some sort of primitive metric that tells us whether a program in some language is more generic than another program in the same language?

Douglas Gregor: We could use concept lattices as a metric of genericity. I don't know if they have been used in the functional side or not, but in C++, we arrange all of our concepts — that is, the requirements on a type or a function — into lattices. Further up the lattice is more generic, so there are more abstractions you can use; further down the lattice is more specific, and this gives you more efficient algorithms.

CBJ: In my view, genericity is a property of functions or methods. In the most general sense, a function that is applicable to more arguments, or more data structures, or more objects, is more generic than one that applies to fewer. In the functional world, if you can have a function that is applicable to a list, or a tree, or a matrix, or some other thing, then it is more generic than one that can just be applied to lists, or just be applied to trees. In the object-oriented world, if you can build container classes, then you get a whole family of types, and you have methods that can apply to many different kinds of data structure...

Johan Jeuring: So the identity function is the *most* generic function?

CBJ: Yes, but it's not a very exciting one; any good definition allows for trivial cases. It seems to me that the inherent feature is not that you can do it in the compiler. It's a consequence of genericity that the compiler has to work harder in order to get meaningful code, and so you see our compilers are becoming more sophisticated. But it's not the goal to transfer work to the compiler; you could build an interpreter and achieve that goal.

The thing that I'm particularly interested in is where there is some uniformity or parametricity in the algorithm. In my world view, the datatypes have common properties: operations like mapping, folding, zipping, traversals, and so on. In some deep semantic sense, these operations are the same on all datatypes. It seems to me that part of the story in generic programming is to understand what it is that makes the datatypes special, or, in the object-oriented world, what the container classes are.

Functional programming has done extremely well with the view that everything is a function. This unifying principle has had tremendous benefits. But if you think back to an earlier world view, where the program is a different thing from the data it acts on, you see that we've

lost something by unifying those two concepts: you can't do things like equality or traversals on function types. When we prove representation theorems that say that any datatype can be interpreted out of, say, products and coproducts and constants, what we're doing is getting a deep understanding of what it is that makes a type a datatype. That's the basis on which generic programming is built. In that sense, if it's driven by the types, then somehow the job doesn't feel complete to me, because these data structures are built in uniform ways, and we should be able to exploit that.

TS: Isn't that really just a phase distinction? The type is an earlier-phase description of the structure of the run-time value. All run-time values are constructed in similar, uniform ways. The type tells me, 'I don't know what what *value* it will be, but I do know it will have this *shape*'. That information is very useful in a language with a phase distinction, because I can exploit that earlier, concrete description of what shape it will have to make better and more efficient implementations.

CBJ: I'm not trying to abolish phase distinctions — type checking is a wonderful thing. But I still think that there are deeper semantic concepts that we're getting hold of here, and they're not dependent on knowing the name of the type, or at least they don't have to be done that way.

Thorsten Altenkirch: I understand your point. At one extreme we have parametric programs, which never look at the type parameter, and at the other extreme we have programs that actually depend on the type, which now we call generic programs. In between there's a subset of generic programs which never look at the type, but which are still not parametric: they're uniform, they only look at the constructors. It's an interesting class of generic programs, and certainly it deserves to be looked at. One thing I have wondered about in the past is whether there is a sort of abstract description of this class. We know that the parametric programs satisfy relational parametricity, but I couldn't see any semantic description of algorithms which never look at the type.

CBJ: I also wondered about that, and it seems to me a bit too hard. When you give a parametrically polymorphic definition of mapping on lists, you never actually ask anything about the type of the entries; the program is oblivious. In any of the systems here you still have to examine the structure somehow. So by asking what is the type constructor at the head of the term, you can get type information. Somehow that specialization information is important; it's not parametric in the usual sense.

Zhaohui Luo: You are implying that we should look at the data more, possibly forgetting types. But another view would be to say that data

came with types. Without types, you don't have data. That's the type-theoretic point of view. With natural numbers, you don't have the numbers first; you have the type 'natural numbers', and the numbers come from that.

* * *

RSC: It seems to me that the notion of genericity is relative to the linguistic context. The first time I ever saw the word 'generics' used was in connection with Ada. The designers of Ada precluded passing procedures as parameters, considering this to be just too risky; they wanted to be cautious and conservative. So they divided programs into packages, and allowed passing procedures as parameters at the level of packages. This was a second-class mechanism for procedure passing: Ada had no procedures as parameters, and in order to link these units of code, these packages, the designers introduced generics, an abstraction mechanism that wasn't available in the programming language. They were stepping outside the linguistic boundaries that they'd established. It seems to me that that's the common theme of all the talks we've heard: whatever linguistic context you're in, generic programming pushing the boundary of the forms of abstraction.

TS: In the call for papers there is a little paragraph that I wrote for the *Workshop on Generic Programming* about four years ago. It says that genericity is about abstracting over concepts that we don't normally think about abstracting over: the structure of types, the structure of the inheritance hierarchy, and so on. We can certainly have genericity even if we don't compute over the structure of types, because there are many other concepts that we can abstract over.