Discussion: The Rôle of Types in Generic Programming

C. Barry Jay: I've noticed that types are being used in different ways in the different approaches. At one extreme we have the dependent-type view, that we should not be shy about computing with types in the same way that we compute with terms. At the other extreme, in Generic Haskell there's a sense in which types are a part of the computation, but the intention is that they will all be compiled away.

A third way, my way, is not to use types in the computation at all. The types play a role only in safety, making sure that everything is well-typed. The actual evaluation does not depend on the types at all. I think that it is 'A Good Thing' if you don't need to compute with types, restricting them to safety properties; but I don't think everybody agrees with me.

Johan Jeuring: The genericity in Generic Haskell is driven by the types. Without types it would be impossible to think of a generic program. The types and the different levels structure my ideas about a program. If types and terms are put into a single framework, I no longer have the language to express this structure. I prefer a situation where I have type-indexed values and normal values.

CBJ: When we write a list program, we write a pattern for `nil`, and a pattern for `cons`; we're certainly thinking about lists. But we don't feel that we want to compute with lists, or that we need to insert the list description into the pattern matching. Is there something different about a generic program?

JJ: I think genericity *is* possible in the absence of types, but it helps my understanding of the programs if types are there. But indeed, you can write a generic program in a programming language without types, like Scheme.

Tim Sheard: But that is much more difficult. You then compute over these structures that are supposed to represent types, but there is no connection between those things and the actual values. There isn't a type language in which you can express the strong connection between a value and its type. You have a program with some parameters, but no

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*

way of knowing that the first parameter is some very neatly described subset of all types in the program.

Conor T. McBride: The point Barry Jay is making is that it is fine for programs at run-time just to look at constructors, when constructors capture the distinctions that are being made. But there are plenty of programs that just aren't like that — for instance, the generic program which, given an input string, attempts to parse it as an element of a specified type. The input just contains the constructors of *String*, but to figure out how to build the desired output, you have to look at the type you're trying to construct. Another example is Huet's idea of The Zipper, where the intermediate data structure of one-hole contexts is computed from the structure of the type they are one-hole contexts in. Such functions just are not accessible if all you can see are the data constructors.

* * *

Ulrich W. Eisenecker: I would like to take a different perspective. From a more practical point of view, types introduce some safety on the compilation level. If a program has passed the compiler, it is at least partially correct. Another point where types help during compilation is when type information can be used to optimize and transform expressions, as with expression templates in C++.

Now, which benefits and constraints do we obtain when we add typing or genericity at run-time? Todd Veldhuizen has proved some nice space and time trade-offs concerning such compilation models of C++. Another question we heard this afternoon is, what benefits will we gain by recasting types and genericity in terms of generic algorithms and functions? What are the results from a practical point of view? Do we get more safety, or more compact programs, much smaller in size and hence easier to maintain, and so on?

CBJ: Some of the work here has been looking at how to represent arbitrary datatypes in a uniform language, so that we can write patterns for computing on arbitrary datatypes. We've been doing this in a single-language context. But in terms of interoperability, this very same framework might provide a way of marshalling data from one programming style into another one. You could go from C++ to Java, or from ML to Haskell, or whatever it happens to be.

CTM: One of the reasons why generic programming is extremely important is that it is exactly what you need when you move beyond the point of view that a type is just something that legitimizes a data representation to a compiler, just saying what the arrangement of bits is going

to be in the machine. You have to see types as a language for describing much more of the design than has been done in the past. The key to unlocking more and more genericity is to concentrate on how much you can say in the language of types.

CBJ: This is precisely the issue. When we talk about mapping a function over each bit of data in a structure, we say nothing about the internal structure of that type at all, and yet in every system here, mine included, in order to tease out what's involved we have to do some kind of case analysis. So the description of what mapping is is parametric — 'just apply this function to all the data that's in there' — but all of our algorithms fail to capture that in a nice way.

CTM: But we might want to have two types with the same structure that are used for completely different purposes and handled in different ways; we want the sanity check of knowing that they are only used compatibly with their *purpose*, as well as compatibly with their *data representation*. We want to place finer distinctions on the type structure than what the data representation, but we don't want to get penalized for doing so. So more stratification in the type structure of programs demands more abstraction in the code of programs.

CBJ: I think we can all capture those fine distinctions, if I understood you correctly.

* * *

Thorsten Altenkirch: For me, genericity means that you parameterize something by something else. That is precisely what functional programming is about: the idea of first-class citizens, that anything can be parameterized by anything else. Hindley-Milner typing allows, for example, a type to be parameterized by another type, and a program parameterized by a type. But one thing is missing: a type parameterized by a program. So I suppose what we are looking for is a functional programming language with dependent types.

TS: But the amount of boilerplate code that you have to write when you get into a full dependently-typed language is significant.

TA: I completely agree that there is still something to be done in sugaring, in making dependent types digestible.

TS: Even if you sugar them, there's a lot under the surface. So when you do something wrong, and the sugaring fails, it's very hard to figure out what went wrong.

Zhaohui Luo: Our experience working with dependent types in teaching undergraduate and postgraduate students shows that it is not much

more difficult than teaching them functional programming. They accept the examples very easily. The more difficult part is when you get to the type systems — but then, that is also the difficult part when teaching Haskell. Still, within dependently-typed programming, we have not done enough examples; we don't even have a platform — that's what Conor McBride is working on.

JJ: I think you need a kind of domain-specific language for expressing generic programs over type structures. In Generic Haskell we write generic programs in such a domain specific language. I definitely think you can do generic programming using dependent types, and you can also do it in Scheme; but what you really want is a domain-specific library or sublanguage that helps you expressing these kinds of programs. If you don't have that, it is very hard to get your ideas clearly formulated.

TA: A problem I have with a combinator libraries is that sometimes you would like to have a specific type system for your domain-specific language. Hindley-Milner does not give you that, but the universe construction that Conor McBride showed does — you have static type checking for your embedded languages. I think that's an important feature: easier to learn and less complicated. There are values, types and kinds, but sometimes it is easier to see them as one thing.

JJ: I think it is important to see values, types, and kinds as *different* concepts when you write generic programs.

* * *

UWE: I think there is one more important point. First we use types for making programs safer, to get rid of some errors. Types might also be used to execute programs in a dynamic environment. Here the types are preserved after compilation. There is a third way to use type information, in which you capture design knowledge in the program, which may not be important for the compiler, but which may be very important for programmers who have to maintain this code: to understand what the program actually does, and what its conditions are, and so on.

Ernie Cohen: You're basically describing what are usually called 'specifications', and then the question is, where do you stop?

TA: Why do you want to stop at all?

EC: Well, then you want to talk about specifications, but then you should not expect them to be checkable. But if you see types as decidable specifications, then why use these very difficult sorts of language, when there are much more convenient languages for expressing specifications? You should use logic, not something impoverished like a type system.

TS: Types are properties. They may not be the strongest kind of properties that you want, but if a program has a parametrically polymorphic type, this gives a very strong property of the program. We often put types in my programs to say something about my programs, even if we don't really care if the compiler takes advantage of those types.

CTM: In your world, if you wrote a first-order unification algorithm, and you wanted to know that it terminated, you would have to prove that fact. I prefer to write a unification algorithm in a language in which all programs terminate. It's almost exactly the same program; but because the language has a richer type structure, the essential argument that the number of variables decreases when you do a substitution naturally fits the type structure of the program. So the fact that the program exists at all means that it terminates.

EC: If you want to give any property or description of a program in its type, then you should start at the other end, with its specification.

CTM: The point is that you can build this rich expressivity in the type system; you pay as you go. You can write down properties; certainly in a dependent type theory you've got a very powerful logic there, and of course, what's being checked is the text of your proof. You could use theorem-proving technology to help you build that proof; you just have to type it out in Emacs...

EC: I wasn't arguing for specifications as a replacement for types. I was just arguing not to try to use types as a replacement for specifications.

TA: I don't agree. Dependent types provide a smooth transition, from the types that ordinary programmers use to specifications. As Conor McBride pointed out, the type gives you additional structure, and actually simplifies the verification of your program. Types give you a way to construct the program more easily than having specification language on top would. Termination is an important property, but I would also like to allow partial programs, and then prove them total.

TS: I think there needs to be a spectrum of specification methods. If you require every program to be fully specified and proved, you won't get many users.

EC: I just want to point out that there is an implicit assumption being made: that by using stronger and stronger type systems, eventually we will be able to prove all programs correct, using perhaps not just the type system, but also verification tools. But I think there is essentially no empirical support for this claim as yet.

* * *

Robert S. Cartwright: I do see one huge divide that is represented here: the issue of whether the underlying linguistic mechanism is safe, with a static type system and a type soundness theorem. When I want to build tools, I take it as a presupposition that I have type safety in the components I am using, and that when I assemble components, all I have to do is check that the types of the interfaces match, which is a trivial thing to do, and then know that my program is not going to generate any type faults when it executes — it's a well-defined composition. So I just won't go near C++ for that reason.

Lutz Kettner: For people using C++, that is not such a big issue in practice. You rely on strong typing in C++, and the pitfalls that you have where it is not strong, you can easily avoid them.

Douglas Gregor: C++ programmers generally program in a type-safe manner; there are mechanisms for circumventing the type system, so you can break it whenever you like, but that doesn't mean that you have to. Besides, breaking the type system can actually be very powerful. If you use C++ templates, and you are willing to break the type system in a local way, you can actually regenerate all of object-oriented programming: you can build in multiple inheritance and virtual functions, and you can build in sum types like the variants you see in functional languages. There really isn't any notion of a sum type in C++, but you can build it out of templates quite easily, and very efficiently.

CTM: That's a failure of the type system. If you have to take the mickey out of a type system to do what you want, if it doesn't say what you mean, then your type system is not good enough.