Subject: **Re: Static/Strong/Implicit Typing**
From: Erik Naggum <erik@naggum.no>
Date: 28 Jan 2004 14:32:58 +0000
Newsgroups: comp.lang.lisp
Message-ID: <3284289178877169KL2065E@naggum.no>

* Gareth McCaughan
> Erik Naggum observed that you can *already* consider CL to be
> strongly, statically, implicitly typed, with everything having type T.

* sajiimori
| Still sounds like a joke.

  To see its truth, one needs a level of mathematical sophistication
  that I take for granted. Hence my disappointment when so many people
  evidently lack it. I keep wondering how they manage to live in our
  modern world, but they just do, and never wonder about it. Puzzling,
  this is.

| Implementing such a type system requires exactly 0 lines of code.

  Repeatedly jumping to conclusions is really not a good way to stay in
  mental shape.

  There is a difference between having 0 types and having 1 type that
  you have managed to overlook somehow. I do not understand how this is
  possible, but it appears that you have somehow managed to believe that
  a system with a single static type is equivalent to not knowing the
  type at all. Since this is rather incredulous to me, let me at least
  try to explain this so that those who understand both you and me can
  explain it to you.

  Suppose we have a machine with /n/ native types each with their set of
  operations (which is how this whole type thing began). Values of each
  disjoint type may freely share binary representation; operations only
  work on untyped bits and it is the programmer's responsibility to give
  them the right values to operate on. The fundamental observation was
  that while memory is untyped, the operations are typed and yield bad
  results if given values that were expected to be of a different type
  than they actually are, but the computer was utterly unable to report
  any of this as a problem because by the time the operations got their
  values, the binary representation was assumed to be that of the type
  the operations had advertised that they required. The reason for the
  compile-time type analysis is precisely that the execution of the
  program has no idea whatsoever which of the many types that map onto
  to the binary representation the actual value in memory is of, and the
  kind of mistakes that were made in the past when programmers had to
  keep track of this thing by hand was very expensive. So instead of
  the programmer having to keep the proper type in mind, the compiler
  would ensure that a binary representation of an object only had one
  meaning, and they did this by making the memory have type, too, not
  just the operations, however fleetingly in the compiler's view.
  Suppose further that all the values you could possibly think of are
  representable using those /n/ types. In this perfect world, there
  would be no need for any run-time type dispatch or checking at all,
  and every programmer would instantly revere the static type analysis.

  Now suppose you want to add an additional type to your programs, one
  which is not supported by the hardware operations. Because of the
  nature of the computer, it will necessarily require a both a binary
  representation that could be mistaken for some of the native types and
  a set of operations that work on this binary representation that are
  now defined by the programmer. If you insist on the hardware view of
  your computer, you will now create a virtual or abstract type, and you

will inform your compiler of your construction. It will use the exact same methods it used to keep binary representations of native types from being confused or conflated, and you will never give any of your new, virtual values to an operation that requires a native type, nor mistake a native value for your new, virtual type. Meanwhile, in the memory of the computer running your program, there will be a binary representation of an object that does not correspond to any of the machine's own types and whose meaning it is now even more impossible for anyone who looks only at the memory to determine than if they were native types.

Now suppose you add two varieties of an abstract type upon which a number of operations exhibit no variation, but at least one does. This situation is qualitatively different from the native types, for the hardware uses disjoint types. The solution to this problem that is not qualitatively different is to duplicate the functionality for the two varieties and continue to keep them disjoint. However, as you realize that they are interchangeable in some respects, you have to add code to support the interchangeability. This way of implementing abstract types is extremely artificial, however, and breaks completely with the old way of doing things -- any sane programmer would conflate the two varieties and consciously break the type disjointness.   And so the run-time type dispatch is born as the qualitatively different solution.

Since the compiler is now instructed to treat two types as the same, which they actually are not, the language or the programmer must add the type-distinguishing information to the binary representation one way or another and add operations that are usedly only to distinguish types from eachother. Building a harness for this meta-information is not the trivial job you think it is with your «0 lines of code». In fact, meta-information fundamentally contradicts the hardware view of native values, for in the hardware view, a value exists only as the input parameter of some computational operation. In order for the meta-information paradigm to work, we now have to attach meaning to the fact that two innocuous native values subtract to zero. We must never regard these values we use for this purpose as numeric -- they are only to be used for their equality with known values, or identity.

> It is vitally important at this point to realize that the type-distinguishing meta-information is not used to /check/ the types of arguments, but is used to /dispatch/ on them, selecting the operation according to the type of the value.

Suppose we have built the type-dispatching harness for our abstract types such that we can freely add more abstract types, organize them into hierarchies of supertypes and subtypes, and share (or inherit) operations and properties alike in a web of types. All functions that advertise that they accept values of a particular type know that they may also receive values of any of its subtypes and that even if they do not distinguish them from eachother, one of the functions it calls may. So functions that need to make special provisions for the types of its arguments are /generic/ and perform the exact same abstract operation on the abstract types, but different /concrete/ operations according to the types of the values. For instance, all programming languages make the basic arithmetic operations generic over the whole spectrum of numeric types supported by the hardware, usually all the additional numeric types defined by the language and sometimes also the user-defined numeric types. If we retain the type-checking of the compiler that we built for the native types to keep from confusing them, such as catching the obvious misuse of generic functions that only work on any of the various types of numbers, but not on any other supertypes, we now face both compile-time type checking and run-time type dispatch. The compile-time restrictions on correct programs may very well reduce the amount of run-time dispatching, but all functions that /could/ accept any of the abstract types must be prepared to do type dispatching.

(Of course, one way to cheat in this process is to optimize away the type dispatching of a generic function and let the compiler call the type-specific function directly if it can determine the types to a useful degree at compile-time. However, the mechanics of cheating tend to cloud the more central issues, so I just want to mention that it is often more natural to cheat than to do the right thing, which in no way reduces the responsibility to know what the right thing is.)

Now, since we have built this meta-information and type-dispatching harness and forced every function in the entire programming language into it, we face an engineering decision to retain the hardware-native types or to regard them as a special case of the general concept. We also face a theoretical decision, even a philosophical one, in that we have to ask ourselves if there actually can exist disjoint types in the real world, or if disjointness is an artifice of the hardware that it is clearly easier to implement than one which wastes resources to check the types of its arguments. The division of labor that is so evidently very intelligent in the design of hardware and software, is neither evident nor intelligent in the design of programming languages that are intended more for humans than for machines.

If we make the philosophical leap from hardware values to values that reflect the human view of the world, we immediately observe that there can be no such thing as disjoint types in reality -- disjointness of types is an artifice of our own desires and our own limitations in approaching the mass of information that is the real world observed by our brains. Any categorization serves a purpose, and while it takes a genius most of the time to re-categorize for a different purpose, we must be aware of the purpose to which our categorization was designed if we are to avoid the extremely difficult problem of working at cross purposes with our conceptualization of the world we work in and on.

Being exceedingly intelligent and remarkably introspective, we look at our type systems and realize that there must exist a supertype of all types such that what appears to be disjointness is only disjointness for a particular purpose, and we realize intuitively that if there can be supertypes of all types, there must be subtypes of all types, and we invent the conceptual null type of which there are no values, but which is a subtype of all types, if for nothing else, then for the sake of mathematical completeness. Disjointness is now in the eyes of the beholder, only, and any beholder is free to partition the value space into any categories that fit his purpose. Our categorization is now not free of purpose -- it will always continue to have /some/ -- but it is now possible for the programmer to partition the entire set of machine-representable values into his own types any way he likes.

Common Lisp is not quite as abstract, but very close. In Common Lisp, we observe that there is a supertype of all types, T, on which there are no useful operations, but which means that every function that has not advertised to the compiler that it only accepts a subset of the type space, must be prepared to perform type dispatch to weed out the types for which there exist no useful operation. This process is evidenced in the construction of operations that branch out over the type space, incidentally. COND, CASE, etc, all accept T as the final branch, T being that which is always true, the supertype of all types, etc. I delight in this mathematical elegance.

This way of thinking necessarily differs from the artificial, disjoint thinking. We regard the generic operations we define as having well-defined semantics for a subset of the available types, so if they are invoked on values of any other types, we expect that this condition is handled exceptionally. If we define operations that can only, ever, accept values of an a priori set of types, we communicate this to the compiler and expect it to help us avoid mistakes, but in general, we regard the categorization that we employ to model and understand the world we program in and for, as a posteriori, and we do not make the common mistake of believing that what we already know is all that we can possibly know. As a consequence, it is a design mistake in Common Lisp to write functions that expect only very specific types to the exclusion of other natural types and values.

To illustrate just how type-flexible Common Lisp is, the standard has the concept of a designator for a type. Most Common Lisp functions that take only string arguments are not defined merely on strings, but on designators for strings, which means that they can be called with a character which is a designator for a strong of length 1, or a symbol which is a designator for the string that is its name. Whereas most programming languages have rich libraries of string functions, Common Lisp defines a rich library of functions that work on sequences of any kind: lists, vectors, strings. A language in the disjoint tradition may provide one function to search for a character in a string, one function to search for substrings in strings, and another to search a vector of characters for a character. Common Lisp defines the generic function SEARCH which accepts sequences of any type and returns the position of the match. This genericity is so common in Common Lisp that the traditional meaning is regarded as ubiquitous and therefore not in need of special mention. Instead, «generic function» is a new kind of function which allows programmer-defined methods to be defined on particular programmer-defined classes of arguments, but this should not be interpreted to mean that normal Common Lisp functions are not just as generic in the traditional sense.

A request for more static type checking in Common Lisp is regarded as a throw-back to the times before we realized that disjointness is in the eye of the beholder, or as a missing realization that disjointness does not exist in the real world and therefore should not exist in the virtual world we create with our software. Just because computers are designed a particular way that makes certain types of values much more efficient to compute with than others, does not mean that efficiency is /qualitative/. Efficiency is only quantitative and subordinate to correctness. It is a very serious error in the Common Lisp world to write a function that returns the wrong result quickly, but does not know that it was the wrong result. For this reason, type correctness is considered to be the responsibility of the function that makes the requirements, not of the caller or the compiler. If the programmer who makes those requirements is sufficiently communicative, however, the compiler should come to his assistance. The default behavior, on the other hand, is that functions have to accept values of type T.

I trust that by now, you realize that your saying «Implementing such a type system requires exactly 0 lines of code.» is evidence of a level of ignorance that can only be cured by study and willingness to listen to those who do not suffer from it, but if left uncured will lead to numerous misguided if not completely wrong conclusions. If you really want to understand instead of just telling us what you believe before you understand what we already do, you will appreciate that there was something you had not understood that adversely affected your ability to have and express opinions that those who had understood it would be willing to entertain. Ignorance can be cured, however, and everyone who has once been ignorant knows that it is but a temporary condition, but stupidity is not curable, because stupidity is evidence of the more or less conscious decision to remain ignorant when faced with an opportunity to learn. Your dismissal of all arguments from authority sounded alarmingly stupid at the time, but you may yet decide that you benefit from listening to people who understand more than you do.

--

Erik Naggum | Oslo, Norway   2004-028

Act from reason, and failure makes you rethink and study harder.
Act from faith, and failure makes you blame someone and push harder.