The shape of things:
A reason for type preference

Tyson Whitehead

August 1, 2010

Abstract

Assuming only that type shapes are self-similar and of finite length, I present a statistical argument for preferring the shape of one type above another based on the principle of indifference. The result is a probability measure on the space of type shapes and, by extension, unconstrained and partially constrained polymorphic type shapes. Such a measure could be useful, for example, to rigorously handle ambiguous instances and types, and order type space for searching.

Key words: Type, shape, probability, inference, polymorphic, measure.

1 Framework

Types can always be resolved by the programmer supplying them. The goal of a type-inferencing system is to avoid this as much as possible. In the face of ambiguity, the best any algorithm can do is to choose the type that is most likely to arise in the program under the circumstances. Any other choice will result in the programmer having to supply more types. The underlying problem then is how to define a probability measure on the space of types that reflects the frequency with which they arise in ambiguous situations.

1.1 Empirical Distribution

From a strictly technical point of view, the best choice for the set of all existing programs is the empirical distribution of ambiguous types generated by said pro-

grams. This and, by extension, other empirical-based distributions, is unsatisfying for at least three reasons.

- The historical distribution is not stationary. If it is not allowed to evolve, it becomes less and less applicable to the evolving body of programs. If it is allowed to evolve, it changes how ambiguity is resolved, breaking programs in which everything has not been manually resolved.
- The set of types is not closed. The programmer can create any number of new elementary terms and compositions for which there is no historical distribution to draw on.
- Using a historical distribution is intellectually unsatisfying. It is effectively stating there is either nothing deeper at play than what is seen on the surface, or that what is at play is too deep to penetrate.

1.2 Theoretical Distribution

This leaves theoretical distributions. The path to defining such a distribution robustly is to capture something truly fundamental about the underlying system. To this end, consider two arbitrary types

$$(\text{Float} \to \text{Int}) \to [\text{Float}] \to [\text{Int}],$$
$$\text{IO}\ (\text{IO Int}) \to \text{IO Int}.$$

Without going outside the scope of type inferencing by assuming a minimal understanding of the ambiguity in which these types arose (i.e., by assuming a minimal understand of part of the program), they can only be taken at face value.

Taking types at face value leaves seemingly little upon which to base the distribution. There are names differentiating the identically kinded elementary types used to compose them (e.g., Float versus Int) and the structure of their composition as constrained by the required kinds (e.g., $(\text{Float} \to \text{Int}) \to [\text{Int}]$ versus $\text{Float} \to [\text{Int}] \to \text{Int}$).[1] Consider these in greater detail.

1.2.1 Elementary Types

The names of the elementary types apparently present a way to distinguish them. Does it make sense to do this? Should, for example, the elementary type Float be

[1] The kind of a type is its type, which dictates how it can be composed.

preferred over Int given a requirement for a type of kind $*$? I argue no for at least three reasons.

- To promote a specific applicable elementary type over another is to promote one type of program over another (e.g., while Float might be preferred in numerical programs, it is certainly not in other types of programs).
- The set of elementary types is not closed. The programmer can create any number of new elementary types that are applicable. The algorithm can have no prior information about these and thus no reason for preference.
- Elementary types that are more pervasive due to their requirement by a language construct (e.g., every function application results in $\rightarrow$ appearing in a type) cannot skew the distribution of ambiguous types because their additional appearances due to language requirements are not ambiguous.

Without privileged information regarding the intrinsics of the program, equally applicable elementary types are only distinguishable by their assigned names. The assigned names under these circumstances are indistinguishable from arbitrary ones. To not treat them equally likely is a contradiction. It implies reason for doing so, access to the privileged information that is inaccessible. The practice of assigning equal probabilities to mutually exclusive indistinguishable items for this reason is known as the principal of indifference.

1.2.2 Type Structure

What about the structure? Given no reason to distinguish between equally applicable elementary types, is there reason to distinguish between the kind-constrained structures formed in composing them? I believe there is for two reasons.

- Because functional languages are higher order, all types can be pervasive at all levels. Without reason to expect them not to be equally distributed at all levels, self-similarity follows from the principal of indifference.
- Composed types must be of finite length as non-recursive infinite types cannot arise in finite-length programs and recursive infinite types are strictly not allowed.

Together these two points give a non-uniform theoretical distribution for type shapes. The principal of indifference dictates that the most uniform distribution

possible under the finite-length constraint is assumed for the number of arguments required to reduce an applicable elementary type. This provides a spine for composed types. The principal of self-similarity fills in the spine as each argument is again an elementary type requiring some number of arguments to reduce it to a required kind.

2 Model

Let q_n be the probability that the number of arguments required to reduce an applicable elementary type to a required kind is $n-1$ (if only elementary types were applied, the total number of types in the resulting expression would be n). The most uniform distribution for q_n given an expected number of arguments N (currently unknown) can be found by maximizing the associated entropy function

$$-\sum_n \ln(q_n)\, q_n$$

subject to the constraints $\sum_n q_n = 1$, $\sum_n n\, q_n = N$, and a zero probability of infinite types.

2.1 Constrained Shapes

Solving this problem under the first two constraints via Lagrange multipliers gives a family of exponentially decaying probabilities

$$q_n = p^{n-1}\bar{p}$$

where $p = (N-1)/N$ and $\bar{p} = 1-p = 1/N$. Substituting q_n into the entropy expression and simplifying gives $-\ln(p^{p/\bar{p}}\bar{p}) = N\ln(N) - (N-1)\ln(N-1)$. From the equivalent perspectives of maximizing the entropy and making the probabilities as uniform as possible, it is clear that N, and hence p, should be as large as possible so long as the probability of an infinite type is zero.

Viewing the reduction of an applicable elementary type to the required kind as a series of reductions (i.e., apply another argument: yes/no; if yes, apply another argument: yes/no; etc.), it is evident from the expression for q_n that p and $\bar{p}$ are the probabilities of another argument and no more arguments, respectively, at each step. From the principal of self-similarity, a depth-first application of arguments gives

$$r_n = p^{n-1}\bar{p}^n,$$

where r_n is the probability of a given type shape and n is the number of elementary types in the final expression ($n-1$ is the number of applications).

Let c_n be the number of different type shapes composed from n elementary types. It follows that

$$s_n = c_n\, r_n = c_n\, p^{n-1}\bar{p}^n,$$

where s_n is the probability of an arbitrary type shape being composed from n elementary types. The sequence c_n gives the number of ways n elementary types can be parenthesized into groups of two (i.e., a partially applied type applied to its next argument) and forms a series known as the shifted Catalan numbers.[2]

A recursive expression for c_n follows by noting that the number of elementary types to the left and right of the root must sum to n. Summing over all the ways this can be done gives

$$c_n = c_1 c_{n-1} + c_2 c_{n-2} + \ldots + c_{n-2} c_2 + c_{n-1} c_1.$$

This summation is precisely that given by squaring the shifted generating function $c(x)$ and collecting equal powers of x. Combined with $c_1 = 1$, this gives

$$c(x) = \sum_n c_n\, x^n = x + \Big(\sum_n c_n\, x^n\Big)^2 = x + c(x)^2 = \frac{1 \pm \sqrt{1-4x}}{2},$$

where the last expression is just an application of the quadratic equation to the first and second-last expressions. The expression for the shifted Catalan series

$$c_n = \frac{(2(n-1))!}{n!(n-1)!}$$

follows by expanding the negative root in a power series. The negative root is used as only it is applicable near $x = 0$ because $c(0) = 0$.

The shifted generating function also gives a way to write the probability of a finite-length type

$$\sum_n s_n = \sum_n c_n\, p^{n-1}\bar{p}^n = \frac{c(p\bar{p})}{p} = \frac{1 \pm \sqrt{1-4p\bar{p}}}{2p} = \frac{1 \pm (2p-1)}{2p}.$$

The two branches give probabilities 1 and $\bar{p}/p$. These two probabilities must apply, respectively, to small and large values of p as the probability of a finite type

[2]The series is referred to as shifted because the Catalan numbers are usually indexed from 0.

must be 1 and 0, respectively, when $p = 0$ and $p = 1$. The bifurcation point occurs when the two branches are equal at $p = 1/2$, giving

$$\mathbb{P}[\text{finite type}] = \begin{cases} 1 & p \in [0, 1/2], \\ \bar{p}/p & p \in [1/2, 1]. \end{cases}$$

It follows that the maximum values are $p = 1/2$ and $N = 2$.

The probability that a type shape is composed of n elementary types is then

$$r_n = \frac{1}{2^{2n-1}}.$$

The factor of $1/2$ is both the probability of an elementary type requiring at least one more argument to reduce it to a required kind and an elementary type requiring no more arguments to reduce it to a required kind.

2.2 Unconstrained Shapes

Type inferencing consists of starting with unconstrained types, that is, ones having as of yet unspecified components, and progressively constraining these components to more specific types until all the requirements are met. Components that are never restricted to specific types give rise to polymorphic types, such as the type of the list map function

$$\forall_a \forall_b \; (\mathrm{a} \to \mathrm{b}) \to [\mathrm{a}] \to [\mathrm{b}].$$

The universal quantification is understood to be over all types and is usually not stated explicitly.

When a type has components that can range over all types, those components in the type shape range over all type shapes. The probability of such an unconstrained type shape is given by summing over the probabilities given by letting the unconstrained components range over every possible type shape. That is, an incomplete type shape having n elementary types and m unconstrained types, with each unconstrained type occurring m_k times, has probability

$$r_n^{\mathbf{m}} = \sum_{n_1} \cdots \sum_{n_m} \frac{1}{2^{2n-1}} \; c_{n_1} \left(\frac{1}{2^{2n_1}} \right)^{m_1} \cdots \; c_{n_m} \left(\frac{1}{2^{2n_m}} \right)^{m_m},$$

where $\mathbf{m} = (m_1, \ldots, m_m)$.

Moving the sums inward to their respective terms gives m terms of the form

$$\sum_{n_k} c_{n_k} \left(\frac{1}{2^{2n_k}} \right)^{m_k} = \sum_{n} c_{n_k} \left(\frac{1}{2^{2m_k}} \right)^{n_k} = c(2^{-2m_k}) = \frac{1 \pm \sqrt{1 - 4 \cdot 2^{-2m_k}}}{2}.$$

As the bifurcation point occurs at $m_k = 1$ and the probability is 0 when $m_k = \infty$, taking the negative root, simplifying a bit, and substituting back into the original expression gives

$$r_n^{\mathbf{m}} = \frac{1}{2^{2n-1}} \prod_{k=1}^{m} \frac{1 - \sqrt{1 - 2^{-2(m_k - 1)}}}{2}$$

for the probability of a type shape having n elementary types and m unconstrained types, with each unconstrained type occurring m_k times. When $m = 0$ this reduces to the previously defined r_n.

2.3 Partially Constrained Shapes

In a language such as Haskell, it is also possible for types to be partially constrained. For example, a generalization of the list map functions is the functor fmap function, whose type is

$$\forall_{f \in \mathcal{F}} \forall_a \forall_b \ (\mathrm{a} \to \mathrm{b}) \to \mathrm{f\ a} \to \mathrm{f\ b},$$

where $\mathcal{F}$ is the class of functor types. The type shapes fitting a profile such as this can be no more than those fitting the form without class restrictions, and thus the latter must form an upper bound on the probability of the former.

The question of probability reduction is ultimately one of type membership. Given a class and a random type of a specific shape, is there reason to believe membership should be more or less probable than non-membership or that the answer to this should depend on the class in question? I would argue no for at least three reasons.

- Each class of types is not closed. The programmer can add any number of new members. Promoting membership or non-membership in specific or all classes is to make complex assertions about the programmer and/or types of programs.

- The class of classes of types is not closed. The programmer can create any number of new classes about which the algorithm can have no prior information and thus no reason for preference.

- Any special treatment of built-in classes (e.g., the Haskell Show class) cannot evolve with the language as it would change how ambiguity is resolved, breaking programs in which everything has not been manually resolved.

Without reason for preference of membership or non-membership for a random type, the principal of indifference dictates making these equally probable. This introduces an additional factor of 1/2 for each class membership requirement. The derivation remains basically the same as before because the additional factors of 1/2 factor out of the sums. This gives

$$r_n^{\mathbf{m},\mathbf{w}} = \frac{1}{2^{2n-1}} \prod_{k=1}^{m} \frac{1 - \sqrt{1 - 2^{-2(m_k-1)}}}{2^{w_k+1}},$$

as the probability of a given type shape having n elementary types and m unconstrained/partially constrained types, with each unconstrained/partially constrained type having w_k class membership requirements and occurring m_k times. This reduces to the prior expression when $\mathbf{w} = \mathbf{0}$.

3 Conclusion

No program can correctly resolve ambiguity more frequently than that which chooses the most probable alternative under the circumstances. This paper presents a very general and systematic argument for such a measure based on the principal of indifference and the assumption that types are self-similar and of finite length. Arguments for further assumptions are rejected for a variety of reasons, the most significant likely being that against encoding in special cases. Doing so makes only fully specified programs safe as any future additions or changes to these cases will change resolution order.

Ambiguity resolution is a key feature of the function overloading and templating systems of a variety of languages. Mostly this has revolved around some ad hoc method to pick a particular function or template body based on some definition as to how well parameters match. In some sense, such systems define a measure on types, but I am aware of no work arguing a strong basis for the choices made in such systems. There have been a variety of rigorous measures defined for the purpose of proving progression in type-inferencing algorithms. As far as I know, the intent of these measures makes their form quite different from that of this work.

Based on the arguments for the principal of indifference and the assumptions of self-similarity and finiteness of length, I derive a closed-form solution for my type measure. I then expand this closed-form solution to polymorphic/universally quantified types by allowing the unconstrained components to vary over all type shapes. Finally, based on similar arguments to those previously used, I allow for type-class restrictions on the polymorphic/universal quantifications (as allowed by Haskell). I believe the availability of such a well-founded measure opens the door to a variety of interesting areas, including rigorously handling ambiguous instances and types, and ordering type space for searching.