Problem Statement

October 16, 2020

The Curry-Howard correspondence identifies functions with theorems, providing a promising link between well explored areas of math and software engineering. This connection is most pronounced in dependent typed systems that provide a common language to write (total functional) programs about proofs and (intuitionistic) proofs about programs. Specifically, dependent type systems allow types to depend on terms, supporting an extremely powerful system of specification, computation and proof evidence.

Many languages have been written or modified to take advantage of dependent types, exploring potential trade offs in the design space. The most conservative use of dependent types can be seen in languages like Liquid Haskell, and Scala 3: function specifications may depend on values of some specific types such as int or bool, and users are never expected to write there own proofs. More advanced languages like Ats and F* allow function specifications to depend on a wider range of types, have built in automated support to solve some proofs automatically, and allow users to supply their own proof when automation fails. Full-spectrum dependent type languages like Agda, Idris, Zombie and the underlining language of Coq place almost no restriction on what terms may appear in types, and use the same syntactic construct to represent functions, type polymorphism, logical implication, and universal quantification.

However, dependent type systems, especially full-spectrum dependent type systems, can be cumbersome when used as programming languages. If functions are exactly theorems then termination must be proven for every function to guarantee logical soundness. If the user takes full advantage of the dependent types, they will be obligated to prove many tedious facts about their user defined types and functions. The logic in which to prove these facts is conservative: the law of the excluding the law of the excluded middle is unavailable since it cannot be given a pure computational interpretation. A user may be prevented from running their program until the lemmas are proven or unsafely postulated. For this reason, is commonly recommended that programmers limit their use of dependent types and postulate all "obvious" properties to be proven last. There will be no warning that a postulate is incorrect unless complicated 3rd party tools are used.

Most issues of dependent types systems can be translated into the inability to summon a convenient term at type checking time. On one hand this is reasonable: such a term may not exists, require unsupported reasoning like the

law of the excluded middle, or take too long to construct. On the other hand, all of these arguments may be equally made against postulates, even though they are a critical tool to develop dependently typed programs. Interestingly, if proof functions were to be evaluated, the postulate could always be replaced with a value of it's type if the postulate is correct. If a postulate is incorrect, a clear error can often be presented. In the case of postulated equality, a term will always be easy to synthesize or refute at runtime.

I propose interpreting postulates as impure proof search that may be carried out at runtime. Since proof terms are rarely evaluated directly, I also propose an apparently novel symbolic execution of dependently typed programs that reveal issues with postulated facts without delaying type checking. Additionally, proof search can be precompute offline allowing postulates to be used in a pragmatically relevant way.