Problem Statement

December 18, 2020

Dependently typed languages have great potential to enforce invariants, making programming more productive (as with type generic programming), and communicating programmer intent. However, the benefits of dependent type programming can often come at a high burden to the programmer. I propose here a dependently typed programming language where as many obligations as possible are transparently removed from the programmer, even at the expense of normal considerations such as correctness guarantees, decidable type-checking, and efficiency. Not meeting the normal obligations would result in poor compile-time error messages or stuck computation, instead we should give precise runtime and test time error messages. By making type checking less strict, incremental improvements can be made to code. While there may be instances where these features are usable in "production programs" they are not primarily intended as an end to themselves, instead as a more realistic way to build up to the correct, terminating and efficient code that all dependent type research strives for.

Dependent type systems obligate programmers inhabit proof terms of properties that may be tangential to the programmer's current area of work. This problem is pervasive and virtually every dependently typed language provides a mechanism to circumvent the type system (often breaking type soundness and causing computation to get stuck). While the proof terms themselves are rarely of interest to the programmer, the properties they witness are often important: a concrete counter-example can save the programmer time, and even if there is no direct proof the programmer can get confidence if a large number of examples have been tried. I propose giving the assertions runtime behavior, specifically that of proof search. I believe this is the most natural and flexible approach, since it requires a minimum commitment from the programmer, but will preform reasonably on user defined functions and data.

Mainstream typed languages tend to satisfy a replacement property: any well typed sub-expression may be replaced with a different term of the same type without changing the type of the larger expression. This property does not hold in Dependently typed languages, where even contextually equivalent terms of equivalent type are not replaceable (x+0 cannot be replaced with 0+x in many dependently type languages). Associating contextual equivalence with definitional equality is at odds with the decidability of type checking. However, virtually all systems provide a conservative approximation of contextual equiva-

lence that admit as many equalities as possible. This causes a subtle burden to the programmer who is always obligated to contort to the specific definitional syntax of the system. Definitional inequality is one source of the confusing error messages that dependently typed languages are known for. Instead of under-estimating equality, I propose that we over-approximate equality and monitor any observations that could differentiate terms, if an observation is observed at runtime, then a precise error message can be returned.

Since most of the features explicitly trade compile-time restrictions for run-time errors, it is important for the programmer to get feedback on possible defects of there program. To achieve this I propose a partly symbolic testing strategy that is in some sense complete. In the presence of non-terminating recursion or non-terminating proof search, warnings will be given for programs that take a long time to terminate on given inputs.

There is some evidence that pure full-spectrum dependently typed languages like Agda and Idris are easier to learn My language is generally in that style (omitting nice features like nested pattern matching and implicit arguments). As in the Zombie language my language is call by value which allows for partial correctness. Additionally no restrictions are placed on recursion, though this complicates the theory.