The paper shows how to integrate fully-annotated dependently typed terms with unannotated simply-typed terms via compiler-inserted assertions [1]. This allows so-called gradual verification, where dependent types can be added post-hoc to working simply-typed code.

Strengths

- Type-checking is divided between a typed translation of (mixed) surface terms to (dependent) core terms and type coercion rules. The coercions are runtime checks when all static checking fails.
- The authors clearly state their restrictions for "sound and tractable" type-checking.
- Addresses an interesting problem and raises many questions along the way. Can we scale this to a "real" type system, like OCaml's or Java's or Haskell's? Are these pure dependent types sufficient for specifying program properties? Could we go all the way from dependent types to a dynamic language with Finder/Felleisen contracts?

Weaknesses

- The paper's motivation is not clearly connected to their result. The core problem they hope to solve is that dependent types impose a large annotation burden in two respects: it is difficult to come up with the right specification and the specifications often take many lines of code. They avoid these problems by defaulting to simple types, but this does not help developers add dependent types to their programs. Those types are not any easier to write.

  Why is integrating simple & dependent types a better solution than improving type inference or building an Agda-like interactive typechecker?
- Adding references was nice, but that space would have been better spent on their correctness theorems. As is, the paper does not define what "stuck" means, so the type soundness result is not very interesting. (Do assertion failures count as stuck terms? I hope not, else we've lost the whole point of interoperability with unverified simply-typed terms.) I think it also would have been worthwhile to state that the principle type property still holds for the simply-typed fragment.
- The surface language is very limited, to the point of being slightly misleading. There is no polymorphism. Section 2 gives vectors of natural numbers as a motivating example, but datatypes can only be expressed

as axioms. The subtyping rule for function types, SFUN would incorrectly prove:

$$\Pi\, x : \mathbf{int}.\, x <: \Pi\, x : \mathbf{nat}.\, x$$

by the premise $x : \mathbf{nat} \vdash x <: x$, but for the fact that x is not allowed as a type and $\mathbf{nat} <: \mathbf{int}$ is not allowed as a subtyping relationship.

References

[1] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *TCS*, 2004.