Over a decade ago, the POPLmark challenge [2] suggested that the theorem proving community had tools that were close to being usable by programming language researchers to formally prove properties of their designs and implementations. The authors of the POPLmark challenge looked at existing practices and systems and urged the developers of proof assistants to make improvements to existing systems.

Our conclusion from these experiments is that the relevant technology has developed almost to the point where it can be widely used by language researchers. We seek to push it over the threshold, making the use of proof tools common practice in programming language research—mechanized metatheory for the masses. [2]

In fact, a number of research teams have used proof assistants to formally proved significant properties of entire programming languages. Such properties include type preservation, determinacy of evaluation, and the correctness of an OS microkernel and of various compilers: see, for example, [9, 10, 11, 15].

As noted in [2], the poor support for binders in syntax was one problem that held back proof assistants from achieving even more widespread use by programming language researchers and practitioners. In recent years, a number of extensions to programming languages and to proof assistants have been developed for treating bindings. These go by names such as locally nameless [4,18], nominal reasoning [1,5,17,19], and parametric higher-order abstract syntax [6]. Some of these approaches involve extending underlying programming language implementations while the others do not extend the proof assistant or programming language but provide packages, libraries, and/or abstract datatypes that attempt to hide and orchestrate various issues surrounding the syntax of bindings. In the end, nothing canonical seems to have appeared since the POPLmark challenge was made: we are left with a simple grid that rates different approaches on various attributes [16].

Clearly, mature and extensible proof assistants, such as, say, Coq, HOL, and Isabelle/HOL can be extended to deal with syntactic challenges (such as bindings in syntax) that they were not originally designed to handle. At the same time, it seems plausible and desirable to pursue approaches to the problem of bindings in syntax and metatheory more generally.

In this talk, I will argue that bindings are such an intimate aspect of the structure of expressions that they should be accounted for directly in the underlying programming language support for proof assistants. High-level and semantically elegant programming language support can be found in rather old and familiar concepts. In particular, Church’s Simple Theory of Types [7] has long ago provided answers to how bindings interact with logical connectives and quantifiers. Similarly, the proof search interpretation [13] of Gentzen’s proof theory [8] provides a rich model of computation that supports bindings. I outline several principles for dealing computationally with bindings that follow from their treatments in quantificational logic and sequent calculus. One of the most central principles about bindings is that bound variables never become free: instead bindings can move from term-level bindings (λ-abstractions) to formula-level bindings (quantifiers) to proof-level bindings (eigenvariables and nominal constants) [12,14] I will also describe some implementations [3,12] of these principles that have helped to validate their effectiveness as computational principles.
Acknowledgments. This work has been funded by the ERC Advanced Grant ProofCert.

References


