The Programming Languages and Formal Methods Research Group develops techniques to help ensure program correctness. Our research in programming languages focuses on syntactic, semantic, and pragmatic issues related to the development of correct programs. The research in formal methods emphasizes the algorithmic, computational, and practical aspects in the analysis of real-world programs. We apply mathematical and formal techniques to the investigation of research problems. We also aim to develop tools and methodologies to help developers write correct codes.

Multi-core processors are now widely used in desktop computers, notebooks, and even smart phones. Making use of their ability to handle concurrency to improve the efficiency of applications, however, is difficult. Even experienced developers may make mistakes in writing concurrent programs. Subsequently, despite of the prevalence of concurrent computer systems there are only a few concurrent programs. In order to help developers exploit the computational power in modern processors, the Programming Languages and Formal Methods Research Group is working on language and verification techniques for concurrent programs.

Type systems are widely deployed in modern programming languages to ensure type safety. For example, an ill-typed expression may add the number 0 to a character '0' and thus produce an incorrect result. Type systems allow developers to identify such type errors at the time of compilation. Concurrent programming is known to be error prone; and the challenge of programmers is made still more difficult by modern programs, which often consist of hundreds or even thousands of threads communicating with each other. Modern programming languages adopt expressive type systems to guarantee certain properties of programs.

A type system for message-passing concurrent programs that ensures communication between threads follow a certain protocol is a “session type.” More advanced variations can guarantee no deadlocks. Efforts to implement session-type systems by embedding them in existing languages, however, have not been very productive yet, due to the inexpressiveness of the type system in the host language. We plan to embed session type into Agda, a language with an expressive dependent type system, in the hope of building an integrated environment in which programmers may program in Agda/Haskell and use the type system to guarantee correctness and termination.

We also plan to investigate the “proofs-as-programs” approach to program correctness, as embodied in proof assistants such as Coq and their program extraction mechanisms, and to familiarize ourselves with the details of these tools so as to understand their capacities and limitations. We will concentrate on three application domains: (1) derivatives of regular expressions, (2) transformations of structural contents, and (3) interactions of software components. The problems dealt with in those three domains are often related to those encountered in general application development. For example, Web application scripts should process all user inputs correctly (while avoiding common security pitfalls, such as those caused by SQL injection) and should always produce correct outputs (such as generating only valid XML documents). These issues fall into domains 1 and 2 above. How these scripts interact with one another is also a major research issue (domain 3). We want to certify, by formal proofs, that the scripts for these applications will have the required behaviors before they are deployed. These application domains will provide many interesting and important problems for our investigation into program constructions with formal
 proofs.

Model checking is a technique to identify errors in program designs. Given a proposed property of the design, a model checker verifies the given property conclusively by exploring all design behaviors. For concurrent systems, the number of design behaviors grows exponentially with the number of design components. This is the well-known state explosion problem. Local reasoning is a compositional technique for alleviating the state explosion problem in the verification of concurrent programs. Instead of exploring global states, the compositional technique synthesizes local properties and verifies these on design components. Local reasoning has been used to analyze parametric systems as well.

Local property synthesis is key to the success of local reasoning. Traditionally, local properties are obtained by fixed-point computation. Iterative computation, however, can be unnecessarily expensive. We plan to apply algorithmic learning to the synthesis problem in local reasoning. When many local properties are feasible, algorithmic learning allows us to infer one of them efficiently. Moreover, learning-based synthesis algorithms give us additional flexibility in the forms of inferred properties. Such flexibility will help us understand local properties of programs.

Applying formal techniques, however, can be challenging for average developers, who may lack such training. Yet more and more concurrent programs are being written by average developers. Verification tools for concurrent programs will undoubtedly help such developers improve their productivity. A large portion of existing concurrent programs are written in modern programming languages such as C++ and JAVA. For those existing codes, we also plan to develop automatic verification approaches that can ensure correctness. More specifically, we plan to develop concurrent software model checking approaches. The core technique of model checking is efficient exploration of the entire state spaces of the program to be verified.

For concurrent programs, the state space to be explored is normally extremely large due to the interleaving of behaviors of each individual thread. In order to explore the state space efficiently, abstraction techniques must be employed. We plan to combine existing abstraction techniques for concurrent programs. For example, we might use thread modular abstraction for the control part and predicate abstraction for data part. The goal is scale up concurrent software model checking to programs that cannot be handled by existing approaches.

Going beyond our research activities, we also dedicate significant resources to education. In order to introduce our research to students, since 2007 we have organized the annual Formosan Summer School on Logic, Language, and Computation (FLOLAC). Through FLOLAC, more and more students in Taiwan have been encouraged to study and conduct research in programming languages and formal methods.